Celery Documentation

Release 3.0.25

Ask Solem
Contributors

July 10, 2014

Contents

Celery is a simple, flexible and reliable distributed system to process vast amounts of messages, while providing operations with the tools required to maintain such a system.

It's a task queue with focus on real-time processing, while also supporting task scheduling.

Celery has a large and diverse community of users and contributors, you should come join us *on IRC* or *our mailing-list*.

Celery is Open Source and licensed under the BSD License.

CHAPTER 1

Getting Started

- If you are new to Celery you can get started by following the *First Steps with Celery* tutorial.
- You can also check out the *FAQ*.

CHAPTER 2

Contents

2.1 Copyright

Celery User Manual

by Ask Solem

Note: While the *Celery* documentation is offered under the Creative Commons *attribution-nonconmmercial-share alike 3.0 united states* license, the Celery *software* is offered under the less restrictive BSD License (3 Clause)

2.2 Getting Started

Release 3.0

Date July 10, 2014

2.2.1 Introduction to Celery

What is a Task Queue?

Task queues are used as a mechanism to distribute work across threads or machines.

A task queue's input is a unit of work, called a task, dedicated worker processes then constantly monitor the queue for new work to perform.

Celery communicates via messages using a broker to mediate between clients and workers. To initiate a task a client puts a message on the queue, the broker then delivers the message to a worker.

A Celery system can consist of multiple workers and brokers, giving way to high availability and horizontal scaling.

Celery is written in Python, but the protocol can be implemented in any language. So far there's RCelery for the Ruby programming language, and a *PHP client*, but language interoperability can also be achieved by *using webhooks*.

What do I need?

Version Requirements

Celery version 3.0 runs on

- Python 2.5, 2.6, 2.7, 3.2, 3.3
- PyPy 1.8, 1.9
- Jython 2.5, 2.7.

This is the last version to support Python 2.5, and from the next version Python 2.6 or newer is required. The last version to support Python 2.4 was Celery series 2.2.

Celery requires a message broker to send and receive messages. The RabbitMQ, Redis and MongoDB broker transports are feature complete, but there's also support for a myriad of other solutions, including using SQLite for local development.

Celery can run on a single machine, on multiple machines, or even across data centers.

Get Started

If this is the first time you're trying to use Celery, or you are new to Celery 3.0 coming from previous versions then you should read our getting started tutorials:

- *First Steps with Celery*
- *Next Steps*

Celery is...

- **Simple**
  Celery is easy to use and maintain, and it *doesn't need configuration files*.
  It has an active, friendly community you can talk to for support, including a mailing-list and an *IRC channel*.
  Here's one of the simplest applications you can make:

```
from celery import Celery

celery = Celery('hello', broker='amqp://guest@localhost//')

@celery.task
def hello():
    return 'hello world'
```

- **Highly Available**
  Workers and clients will automatically retry in the event of connection loss or failure, and some brokers support HA in way of *Master/Master* or *Master/Slave* replication.
- **Fast**
  A single Celery process can process millions of tasks a minute, with sub-millisecond round-trip latency (using RabbitMQ, py-librabbitmq, and optimized settings).
- **Flexible**
  Almost every part of *Celery* can be extended or used on its own, Custom pool implementations, serializers, compression schemes, logging, schedulers, consumers, producers, autoscalers, broker transports and much more.

It supports

- **Brokers**
- *RabbitMQ*, *Redis*,
- *MongoDB*, *Beanstalk*
- *CouchDB*, *SQLAlchemy*
- *Django ORM*, *Amazon SQS*,
- and more...
- **Concurrency**
- multiprocessing,
- Eventlet, gevent
- threads/single threaded
- **Result Stores**
- AMQP, Redis
- memcached, MongoDB
- SQLAlchemy, Django ORM
- Apache Cassandra
- **Serialization**
- *pickle*, *json*, *yaml*, *msgpack*.
- *zlib*, *bzip2* compression.
- Cryptographic message signing.

Features

- **Monitoring**
  A stream of monitoring events is emitted by workers and is used by built-in and external tools to tell you what your cluster is doing – in real-time.
  Read more....
- **Workflows**
  Simple and complex workflows can be composed using a set of powerful primitives we call the "canvas", including grouping, chaining, chunking and more.
  Read more....
- **Time & Rate Limits**
  You can control how many tasks can be executed per second/minute/hour, or how long a task can be allowed to run, and this can be set as a default, for a specific worker or individually for each task type.
  Read more....
- **Scheduling**
  You can specify the time to run a task in seconds or a `datetime`, or or you can use periodic tasks for recurring events based on a simple interval, or crontab expressions supporting minute, hour, day of week, day of month, and month of year.
  Read more....
- **Autoreloading**
  In development workers can be configured to automatically reload source code as it changes, including *`inotify(7)`* support on Linux.
  Read more....
- **Autoscaling**
  Dynamically resizing the worker pool depending on load, or custom metrics specified by the user, used to limit memory usage in shared hosting/cloud environments or to enforce a given quality of service.
  Read more....
- **Resource Leak Protection**
  The *`--maxtasksperchild`* option is used for user tasks leaking resources, like memory or file descriptors, that are simply out of your control.
  Read more....
- **User Components**
  Each worker component can be customized, and additional components can be defined by the user. The worker is built up using "boot steps" — a dependency graph enabling fine grained control of the worker's internals.

Framework Integration

Celery is easy to integrate with web frameworks, some of which even have integration packages:

Django	django-celery
Pyramid	pyramid_celery
Pylons	celery-pylons
Flask	not needed
web2py	web2py-celery
Tornado	tornado-celery

The integration packages are not strictly necessary, but they can make development easier, and sometimes they add important hooks like closing database connections at *`fork(2)`*.

Quickjump

I want to

- *get the return value of a task*
- *use logging from my task*
- *learn about best practices*
- *create a custom task base class*
- *add a callback to a group of tasks*
- *split a task into several chunks*
- *optimize the worker*
- *see a list of built-in task states*
- *create custom task states*
- *set a custom task name*
- *track when a task starts*
- *retry a task when it fails*
- *get the id of the current task*
- *know what queue a task was delivered to*
- *see a list of running workers*
- *purge all messages*
- *inspect what the workers are doing*
- *see what tasks a worker has registerd*
- *migrate tasks to a new broker*
- *see a list of event message types*
- *contribute to Celery*
- *learn about available configuration settings*
- *receive email when a task fails*
- *get a list of people and companies using Celery*
- *write my own remote control command*
- *change worker queues at runtime*

Jump to

- *Brokers*
- *Applications*
- *Tasks*
- *Calling*
- *Workers*
- *Daemonizing*
- *Monitoring*
- *Optimizing*
- *Security*
- *Routing*
- *Configuration*
- *Django*
- *Contributing*
- *Signals*
- *FAQ*
- *API Reference*

Installation

You can install Celery either via the Python Package Index (PyPI) or from source.

To install using *pip*,:

```
$ pip install -U Celery
```

To install using *easy_install*,:

```
$ easy_install -U Celery
```

Bundles

Celery also defines a group of bundles that can be used to install Celery and the dependencies for a given feature.

The following bundles are available:

celery-with-redis for using Redis as a broker.

celery-with-mongodb for using MongoDB as a broker.

django-celery-with-redis for Django, and using Redis as a broker.

django-celery-with-mongodb for Django, and using MongoDB as a broker.

Downloading and installing from source

Download the latest version of Celery from http://pypi.python.org/pypi/celery/

You can install it by doing the following,:

```
$ tar xvfz celery-0.0.0.tar.gz
$ cd celery-0.0.0
$ python setup.py build
# python setup.py install
```

The last command must be executed as a privileged user if you are not currently using a virtualenv.

Using the development version

You can clone the repository by doing the following:

```
$ git clone https://github.com/celery/celery
$ cd celery
$ python setup.py develop
```

The development version will usually also depend on the development version of kombu, the messaging framework Celery uses to send and receive messages, so you should also install that from git:

```
$ git clone https://github.com/celery/kombu
$ cd kombu
$ python setup.py develop
```

2.2.2 Brokers

Release 3.0

Date July 10, 2014

Celery supports several message transport alternatives.

Broker Instructions

Using RabbitMQ

Installation & Configuration RabbitMQ is the default broker so it does not require any additional dependencies or initial configuration, other than the URL location of the broker instance you want to use:

```
>>> BROKER_URL = 'amqp://guest:guest@localhost:5672//'
```

For a description of broker URLs and a full list of the various broker configuration options available to Celery, see *Broker Settings*.

Installing the RabbitMQ Server See Installing RabbitMQ over at RabbitMQ's website. For Mac OS X see Installing RabbitMQ on OS X.

Note: If you're getting *nodedown* errors after installing and using **rabbitmqctl** then this blog post can help you identify the source of the problem:

http://somic.org/2009/02/19/on-rabbitmqctl-and-badrpcnodedown/

Setting up RabbitMQ To use celery we need to create a RabbitMQ user, a virtual host and allow that user access to that virtual host:

```
$ rabbitmqctl add_user myuser mypassword

$ rabbitmqctl add_vhost myvhost

$ rabbitmqctl set_permissions -p myvhost myuser ".*" ".*" ".*"
```

See the RabbitMQ Admin Guide for more information about access control.

Installing RabbitMQ on OS X The easiest way to install RabbitMQ on Snow Leopard is using Homebrew; the new and shiny package management system for OS X.

In this example we'll install Homebrew into `/lol`, but you can choose whichever destination, even in your home directory if you want, as one of the strengths of Homebrew is that it's relocatable.

Homebrew is actually a git repository, so to install Homebrew, you first need to install git. Download and install from the disk image at http://code.google.com/p/git-osx-installer/downloads/list?can=3

When git is installed you can finally clone the repository, storing it at the `/lol` location:

```
$ git clone git://github.com/mxcl/homebrew /lol
```

Brew comes with a simple utility called **brew**, used to install, remove and query packages. To use it you first have to add it to `PATH`, by adding the following line to the end of your `~/.profile`:

```
export PATH="/lol/bin:/lol/sbin:$PATH"
```

Save your profile and reload it:

```
$ source ~/.profile
```

Finally, we can install rabbitmq using **brew**:

```
$ brew install rabbitmq
```

Configuring the system host name If you're using a DHCP server that is giving you a random host name, you need to permanently configure the host name. This is because RabbitMQ uses the host name to communicate with nodes.

Use the **scutil** command to permanently set your host name:

```
$ sudo scutil --set HostName myhost.local
```

Then add that host name to `/etc/hosts` so it's possible to resolve it back into an IP address:

```
127.0.0.1       localhost myhost myhost.local
```

If you start the rabbitmq server, your rabbit node should now be *rabbit@myhost*, as verified by **rabbitmqctl**:

```
$ sudo rabbitmqctl status
Status of node rabbit@myhost ...
[{running_applications,[{rabbit,"RabbitMQ","1.7.1"},
                        {mnesia,"MNESIA  CXC 138 12","4.4.12"},
                        {os_mon,"CPO  CXC 138 46","2.2.4"},
                        {sasl,"SASL  CXC 138 11","2.1.8"},
                        {stdlib,"ERTS  CXC 138 10","1.16.4"},
                        {kernel,"ERTS  CXC 138 10","2.13.4"}]},
{nodes,[rabbit@myhost]},
{running_nodes,[rabbit@myhost]}]
...done.
```

This is especially important if your DHCP server gives you a host name starting with an IP address, (e.g. *23.10.112.31.comcast.net*), because then RabbitMQ will try to use *rabbit@23*, which is an illegal host name.

Starting/Stopping the RabbitMQ server To start the server:

```
$ sudo rabbitmq-server
```

you can also run it in the background by adding the `-detached` option (note: only one dash):

```
$ sudo rabbitmq-server -detached
```

Never use **kill** to stop the RabbitMQ server, but rather use the **rabbitmqctl** command:

```
$ sudo rabbitmqctl stop
```

When the server is running, you can continue reading Setting up RabbitMQ.

Using Redis

Installation For the Redis support you have to install additional dependencies. You can install both Celery and these dependencies in one go using either the celery-with-redis, or the *django-celery-with-redis* bundles:

```
$ pip install -U celery-with-redis
```

Configuration Configuration is easy, just configure the location of your Redis database:

```
BROKER_URL = 'redis://localhost:6379/0'
```

Where the URL is in the format of:

```
redis://:password@hostname:port/db_number
```

all fields after the scheme are optional, and will default to localhost on port 6379, using database 0.

Visibility Timeout The visibility timeout defines the number of seconds to wait for the worker to acknowledge the task before the message is redelivered to another worker. Be sure to see *Caveats* below.

This option is set via the BROKER_TRANSPORT_OPTIONS setting:

```
BROKER_TRANSPORT_OPTIONS = {'visibility_timeout': 3600}  # 1 hour.
```

The default visibility timeout for Redis is 1 hour.

Results If you also want to store the state and return values of tasks in Redis, you should configure these settings:

```
CELERY_RESULT_BACKEND = 'redis://localhost:6379/0'
```

For a complete list of options supported by the Redis result backend, see *Redis backend settings*

Caveats

- If a task is not acknowledged within the *Visibility Timeout* the task will be redelivered to another worker and executed.

  This causes problems with ETA/countdown/retry tasks where the time to execute exceeds the visibility timeout; in fact if that happens it will be executed again, and again in a loop.

  So you have to increase the visibility timeout to match the time of the longest ETA you are planning to use.

  Note that Celery will redeliver messages at worker shutdown, so having a long visibility timeout will only delay the redelivery of 'lost' tasks in the event of a power failure or forcefully terminated workers.

  Periodic tasks will not be affected by the visibility timeout, as this is a concept separate from ETA/countdown.

  You can increase this timeout by configuring a transport option with the same name:

  BROKER_TRANSPORT_OPTIONS = {'visibility_timeout': 43200}

  The value must be an int describing the number of seconds.

- Monitoring events (as used by flower and other tools) are global and is not affected by the virtual host setting.

This is caused by a limitation in Redis. The Redis PUB/SUB channels are global and not affected by the database number.

Using SQLAlchemy

Installation

Configuration Celery needs to know the location of your database, which should be the usual SQLAlchemy connection string, but with 'sqla+' prepended to it:

```
BROKER_URL = 'sqla+sqlite:///celerydb.sqlite'
```

This transport uses only the `BROKER_URL` setting, which have to be an SQLAlchemy database URI.

Please see SQLAlchemy: Supported Databases for a table of supported databases.

Here's a list of examples using a selection of other SQLAlchemy Connection String's:

```
# sqlite (filename)
BROKER_URL = 'sqla+sqlite:///celerydb.sqlite'

# mysql
BROKER_URL = 'sqla+mysql://scott:tiger@localhost/foo'

# postgresql
BROKER_URL = 'sqla+postgresql://scott:tiger@localhost/mydatabase'

# oracle
BROKER_URL = 'sqla+oracle://scott:tiger@127.0.0.1:1521/sidname'
```

Results To store results in the database as well, you should configure the result backend. See *Database backend settings*.

Limitations The SQLAlchemy database transport does not currently support:

- Remote control commands (celeryev, broadcast)
- Events, including the Django Admin monitor.
- Using more than a few workers (can lead to messages being executed multiple times).

Using the Django Database

Installation

Configuration The database transport uses the Django *DATABASE_** settings for database configuration values.

1. Set your broker transport:

   ```
   BROKER_URL = 'django://'
   ```

2. Add `kombu.transport.django` to *INSTALLED_APPS*:

```
INSTALLED_APPS = ('kombu.transport.django', )
```

3. Sync your database schema:

```
$ python manage.py syncdb
```

Limitations The Django database transport does not currently support:

- Remote control commands (celeryev, broadcast)
- Events, including the Django Admin monitor.
- Using more than a few workers (can lead to messages being executed multiple times).

Using MongoDB

Installation For the MongoDB support you have to install additional dependencies. You can install both Celery and these dependencies in one go using either the celery-with-mongodb, or the *django-celery-with-mongodb* bundles:

```
$ pip install -U celery-with-mongodb
```

Configuration Configuration is easy, set the transport, and configure the location of your MongoDB database:

```
BROKER_URL = 'mongodb://localhost:27017/database_name'
```

Where the URL is in the format of:

```
mongodb://userid:password@hostname:port/database_name
```

The host name will default to `localhost` and the port to 27017, and so they are optional. userid and password are also optional, but needed if your MongoDB server requires authentication.

Results If you also want to store the state and return values of tasks in MongoDB, you should see *MongoDB backend settings*.

Using Amazon SQS

Installation For the Amazon SQS support you have to install the boto library:

```
$ pip install -U boto
```

Configuration You have to specify SQS in the broker URL:

```
BROKER_URL = 'sqs://ABCDEFGHIJKLMNOPQRST:ZYXK7NiynGlTogH8Nj+P9nlE73sq3@'
```

where the URL format is:

```
sqs://aws_access_key_id:aws_secret_access_key@
```

you must *remember to include the "@" at the end.*

The login credentials can also be set using the environment variables `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`, in that case the broker url may only be `sqs://`.

Note: If you specify AWS credentials in the broker URL, then please keep in mind that the secret access key may contain unsafe characters that needs to be URL encoded.

Options

Region The default region is `us-east-1` but you can select another region by configuring the `BROKER_TRANSPORT_OPTIONS` setting:

```
BROKER_TRANSPORT_OPTIONS = {'region': 'eu-west-1'}
```

See also:

An overview of Amazon Web Services regions can be found here:

http://aws.amazon.com/about-aws/globalinfrastructure/

Visibility Timeout The visibility timeout defines the number of seconds to wait for the worker to acknowledge the task before the message is redelivered to another worker. Also see caveats below.

This option is set via the `BROKER_TRANSPORT_OPTIONS` setting:

```
BROKER_TRANSPORT_OPTIONS = {'visibility_timeout': 3600}  # 1 hour.
```

The default visibility timeout is 30 seconds.

Polling Interval The polling interval decides the number of seconds to sleep between unsuccessful polls. This value can be either an int or a float. By default the value is 1 second, which means that the worker will sleep for one second whenever there are no more messages to read.

You should note that **more frequent polling is also more expensive, so increasing the polling interval can save you money**.

The polling interval can be set via the `BROKER_TRANSPORT_OPTIONS` setting:

```
BROKER_TRANSPORT_OPTIONS = {'polling_interval': 0.3}
```

Very frequent polling intervals can cause *busy loops*, which results in the worker using a lot of CPU time. If you need sub-millisecond precision you should consider using another transport, like *RabbitMQ <broker-amqp>*, or *Redis <broker-redis>*.

Queue Prefix By default Celery will not assign any prefix to the queue names, If you have other services using SQS you can configure it do so using the `BROKER_TRANSPORT_OPTIONS` setting:

```
BROKER_TRANSPORT_OPTIONS = {'queue_name_prefix': 'celery-'}
```

Caveats

- If a task is not acknowledged within the `visibility_timeout`, the task will be redelivered to another worker and executed.

This causes problems with ETA/countdown/retry tasks where the time to execute exceeds the visibility timeout; in fact if that happens it will be executed again, and again in a loop.

So you have to increase the visibility timeout to match the time of the longest ETA you are planning to use.

Note that Celery will redeliver messages at worker shutdown, so having a long visibility timeout will only delay the redelivery of 'lost' tasks in the event of a power failure or forcefully terminated workers.

Periodic tasks will not be affected by the visibility timeout, as it is a concept separate from ETA/countdown.

The maximum visibility timeout supported by AWS as of this writing is 12 hours (43200 seconds):

```
BROKER_TRANSPORT_OPTIONS = {'visibility_timeout': 43200}
```

- SQS does not yet support worker remote control commands.
- SQS does not yet support events, and so cannot be used with **celery events**, **celerymon** or the Django Admin monitor.

Results Multiple products in the Amazon Web Services family could be a good candidate to store or publish results with, but there is no such result backend included at this point.

Warning: Do not use the `amqp` backend with SQS.
It will create one queue for every task, and the queues will not be collected. This could cost you money that would be better spent contributing an AWS result store backend back to Celery :)

Using CouchDB

Installation For the CouchDB support you have to install additional dependencies. You can install both Celery and these dependencies in one go using either the celery-with-couchdb, or the *django-celery-with-couchdb* bundles:

```
$ pip install -U celery-with-couchdb
```

Configuration Configuration is easy, set the transport, and configure the location of your CouchDB database:

```
BROKER_URL = 'couchdb://localhost:5984/database_name'
```

Where the URL is in the format of:

```
couchdb://userid:password@hostname:port/database_name
```

The host name will default to `localhost` and the port to 5984, and so they are optional. userid and password are also optional, but needed if your CouchDB server requires authentication.

Results Storing task state and results in CouchDB is currently **not supported**.

Limitations The CouchDB message transport does not currently support:

- Remote control commands (celeryctl, broadcast)

Using Beanstalk

Installation For the Beanstalk support you have to install additional dependencies. You can install both Celery and these dependencies in one go using either the celery-with-beanstalk, or the *django-celery-with-beanstalk* bundles:

```
$ pip install -U celery-with-beanstalk
```

Configuration Configuration is easy, set the transport, and configure the location of your Beanstalk database:

```
BROKER_URL = 'beanstalk://localhost:11300'
```

Where the URL is in the format of:

```
beanstalk://hostname:port
```

The host name will default to `localhost` and the port to 11300, and so they are optional.

Results Using Beanstalk to store task state and results is currently **not supported**.

Limitations The Beanstalk message transport does not currently support:

- Remote control commands (celeryctl, broadcast)
- Authentication

Using IronMQ

Installation For IronMQ support, you'll need the [iron_celery](http://github.com/iron-io/iron_celery) library:

```
$ pip install iron_celery
```

As well as an [Iron.io account](http://www.iron.io). Sign up for free at [iron.io](http://www.iron.io).

Configuration First, you'll need to import the iron_celery library right after you import Celery, for example:

```
from celery import Celery
import iron_celery

celery = Celery('mytasks', broker='ironmq://', backend='ironcache://')
```

You have to specify IronMQ in the broker URL:

```
BROKER_URL = 'ironmq://ABCDEFGHIJKLMNOPQRST:ZYXK7NiynGlTogH8Nj+P9nlE73sq3@'
```

where the URL format is:

```
ironmq://project_id:token@
```

you must *remember to include the "@" at the end.*

The login credentials can also be set using the environment variables `IRON_TOKEN` and `IRON_PROJECT_ID`, which are set automatically if you use the IronMQ Heroku add-on. And in this case the broker url may only be:

```
ironmq://
```

Clouds The default cloud/region is `AWS us-east-1`. You can choose the IronMQ Rackspace (ORD) cloud by changing the URL to:

```
ironmq://project_id:token@mq-rackspace-ord.iron.io
```

Results You can store results in IronCache with the same Iron.io credentials, just set the results URL with the same syntax as the broker URL, but changing the start to `ironcache`:

```
ironcache::://project_id:token@
```

This will default to a cache named "Celery", if you want to change that:

```
ironcache::://project_id:token@/awesomecache
```

More Information You can find more information in the [iron_celery README](http://github.com/iron-io/iron_celery).

Broker Overview

This is comparison table of the different transports supports, more information can be found in the documentation for each individual transport (see *Broker Instructions*).

Name	Status	Monitoring	Remote Control
RabbitMQ	Stable	Yes	Yes
Redis	Stable	Yes	Yes
Mongo DB	Experimental	Yes	Yes
Beanstalk	Experimental	No	No
Amazon SQS	Experimental	No	No
Couch DB	Experimental	No	No
Zookeeper	Experimental	No	No
Django DB	Experimental	No	No
SQLAlchemy	Experimental	No	No
Iron MQ	3rd party	No	No

Experimental brokers may be functional but they do not have dedicated maintainers.

Missing monitor support means that the transport does not implement events, and as such Flower, *celery events*, *celerymon* and other event-based monitoring tools will not work.

Remote control means the ability to inspect and manage workers at runtime using the *celery inspect* and *celery control* commands (and other tools using the remote control API).

2.2.3 First Steps with Celery

Celery is a task queue with batteries included. It is easy to use so that you can get started without learning the full complexities of the problem it solves. It is designed around best practices so that your product can scale and integrate with other languages, and it comes with the tools and support you need to run such a system in production.

In this tutorial you will learn the absolute basics of using Celery. You will learn about;

- Choosing and installing a message broker.
- Installing Celery and creating your first task
- Starting the worker and calling tasks.

- Keeping track of tasks as they transition through different states, and inspecting return values.

Celery may seem daunting at first - but don't worry - this tutorial will get you started in no time. It is deliberately kept simple, so to not confuse you with advanced features. After you have finished this tutorial it's a good idea to browse the rest of the documentation, for example the *Next Steps* tutorial, which will showcase Celery's capabilities.

Choosing a Broker

Celery requires a solution to send and receive messages, usually this comes in the form of a separate service called a *message broker*.

There are several choices available, including:

RabbitMQ

RabbitMQ is feature-complete, stable, durable and easy to install. It's an excellent choice for a production environment. Detailed information about using RabbitMQ with Celery:

Using RabbitMQ

If you are using Ubuntu or Debian install RabbitMQ by executing this command:

```
$ sudo apt-get install rabbitmq-server
```

When the command completes the broker is already running in the background, ready to move messages for you: `Starting rabbitmq-server: SUCCESS`.

And don't worry if you're not running Ubuntu or Debian, you can go to this website to find similarly simple installation instructions for other platforms, including Microsoft Windows:

http://www.rabbitmq.com/download.html

Redis

Redis is also feature-complete, but is more susceptible to data loss in the event of abrupt termination or power failures. Detailed information about using Redis:

Using Redis

Using a database

Using a database as a message queue is not recommended, but can be sufficient for very small installations. Your options include:

- *Using SQLAlchemy*
- *Using the Django Database*

If you're already using a Django database for example, using it as your message broker can be convenient while developing even if you use a more robust system in production.

Other brokers

In addition to the above, there are other experimental transport implementations to choose from, including *Amazon SQS*, *Using MongoDB* and *IronMQ*.

See *Broker Overview*.

Installing Celery

Celery is on the Python Package Index (PyPI), so it can be installed with standard Python tools like `pip` or `easy_install`:

```
$ pip install celery
```

Application

The first thing you need is a Celery instance, this is called the celery application or just app in short. Since this instance is used as the entry-point for everything you want to do in Celery, like creating tasks and managing workers, it must be possible for other modules to import it.

In this tutorial you will keep everything contained in a single module, but for larger projects you want to create a *dedicated module*.

Let's create the file `tasks.py`:

```
from celery import Celery

celery = Celery('tasks', broker='amqp://guest@localhost//')

@celery.task
def add(x, y):
    return x + y
```

The first argument to `Celery` is the name of the current module, this is needed so that names can be automatically generated, the second argument is the broker keyword argument which specifies the URL of the message broker you want to use, using RabbitMQ here, which is already the default option. See *Choosing a Broker* above for more choices, e.g. for Redis you can use `redis://localhost`, or MongoDB: `mongodb://localhost`.

You defined a single task, called `add`, which returns the sum of two numbers.

Running the celery worker server

You now run the worker by executing our program with the `worker` argument:

```
$ celery -A tasks worker --loglevel=info
```

In production you will want to run the worker in the background as a daemon. To do this you need to use the tools provided by your platform, or something like supervisord (see *Running the worker as a daemon* for more information).

For a complete listing of the command line options available, do:

```
$  celery worker --help
```

There also several other commands available, and help is also available:

```
$ celery help
```

Calling the task

To call our task you can use the `delay()` method.

This is a handy shortcut to the `apply_async()` method which gives greater control of the task execution (see *Calling Tasks*):

```
>>> from tasks import add
>>> add.delay(4, 4)
```

The task has now been processed by the worker you started earlier, and you can verify that by looking at the workers console output.

Calling a task returns an `AsyncResult` instance, which can be used to check the state of the task, wait for the task to finish or get its return value (or if the task failed, the exception and traceback). But this isn't enabled by default, and you have to configure Celery to use a result backend, which is detailed in the next section.

Keeping Results

If you want to keep track of the tasks' states, Celery needs to store or send the states somewhere. There are several built-in result backends to choose from: SQLAlchemy/Django ORM, Memcached, Redis, AMQP (RabbitMQ), and MongoDB – or you can define your own.

For this example you will use the *amqp* result backend, which sends states as messages. The backend is specified via the `backend` argument to `Celery`, (or via the `CELERY_RESULT_BACKEND` setting if you choose to use a configuration module):

```
celery = Celery('tasks', backend='amqp', broker='amqp://')
```

or if you want to use Redis as the result backend, but still use RabbitMQ as the message broker (a popular combination):

```
celery = Celery('tasks', backend='redis://localhost', broker='amqp://')
```

To read more about result backends please see *Result Backends*.

Now with the result backend configured, let's call the task again. This time you'll hold on to the `AsyncResult` instance returned when you call a task:

```
>>> result = add.delay(4, 4)
```

The `ready()` method returns whether the task has finished processing or not:

```
>>> result.ready()
False
```

You can wait for the result to complete, but this is rarely used since it turns the asynchronous call into a synchronous one:

```
>>> result.get(timeout=1)
8
```

In case the task raised an exception, `get()` will re-raise the exception, but you can override this by specifying the `propagate` argument:

```
>>> result.get(propagate=True)
```

If the task raised an exception you can also gain access to the original traceback:

```
>>> result.traceback
...
```

See `celery.result` for the complete result object reference.

Configuration

Celery, like a consumer appliance doesn't need much to be operated. It has an input and an output, where you must connect the input to a broker and maybe the output to a result backend if so wanted. But if you look closely at the back there's a lid revealing loads of sliders, dials and buttons: this is the configuration.

The default configuration should be good enough for most uses, but there's many things to tweak so Celery works just the way you want it to. Reading about the options available is a good idea to get familiar with what can be configured. You can read about the options in the the *Configuration and defaults* reference.

The configuration can be set on the app directly or by using a dedicated configuration module. As an example you can configure the default serializer used for serializing task payloads by changing the `CELERY_TASK_SERIALIZER` setting:

```
celery.conf.CELERY_TASK_SERIALIZER = 'json'
```

If you are configuring many settings at once you can use `update`:

```
celery.conf.update(
    CELERY_TASK_SERIALIZER='json',
    CELERY_RESULT_SERIALIZER='json',
    CELERY_TIMEZONE='Europe/Oslo',
    CELERY_ENABLE_UTC=True,
)
```

For larger projects using a dedicated configuration module is useful, in fact you are discouraged from hard coding periodic task intervals and task routing options, as it is much better to keep this in a centralized location, and especially for libraries it makes it possible for users to control how they want your tasks to behave, you can also imagine your SysAdmin making simple changes to the configuration in the event of system trouble.

You can tell your Celery instance to use a configuration module, by calling the `config_from_object()` method:

```
celery.config_from_object('celeryconfig')
```

This module is often called "`celeryconfig`", but you can use any module name.

A module named `celeryconfig.py` must then be available to load from the current directory or on the Python path, it could look like this:

`celeryconfig.py`:

```
BROKER_URL = 'amqp://'
CELERY_RESULT_BACKEND = 'amqp://'

CELERY_TASK_SERIALIZER = 'json'
CELERY_RESULT_SERIALIZER = 'json'
CELERY_TIMEZONE = 'Europe/Oslo'
CELERY_ENABLE_UTC = True
```

To verify that your configuration file works properly, and doesn't contain any syntax errors, you can try to import it:

```
$ python -m celeryconfig
```

For a complete reference of configuration options, see *Configuration and defaults*.

To demonstrate the power of configuration files, this how you would route a misbehaving task to a dedicated queue:

`celeryconfig.py`:

```
CELERY_ROUTES = {
    'tasks.add': 'low-priority',
}
```

Or instead of routing it you could rate limit the task instead, so that only 10 tasks of this type can be processed in a minute (10/m):

`celeryconfig.py`:

```
CELERY_ANNOTATIONS = {
    'tasks.add': {'rate_limit': '10/m'}
}
```

If you are using RabbitMQ, Redis or MongoDB as the broker then you can also direct the workers to set a new rate limit for the task at runtime:

```
$ celery control rate_limit tasks.add 10/m
worker.example.com: OK
    new rate limit set successfully
```

See *Routing Tasks* to read more about task routing, and the `CELERY_ANNOTATIONS` setting for more about annotations, or *Monitoring and Management Guide* for more about remote control commands, and how to monitor what your workers are doing.

Where to go from here

If you want to learn more you should continue to the *Next Steps* tutorial, and after that you can study the *User Guide*.

2.2.4 Next Steps

The *First Steps with Celery* guide is intentionally minimal. In this guide I will demonstrate what Celery offers in more detail, including how to add Celery support for your application and library.

This document does not document all of Celery's features and best practices, so it's recommended that you also read the *User Guide*

Using Celery in your Application

Our Project

Project layout:

```
proj/__init__.py
    /celery.py
    /tasks.py
```

proj/celery.py

```
from __future__ import absolute_import

from celery import Celery

celery = Celery('proj.celery',
                broker='amqp://',
                backend='amqp://',
                include=['proj.tasks'])

# Optional configuration, see the application user guide.
celery.conf.update(
    CELERY_TASK_RESULT_EXPIRES=3600,
)

if __name__ == '__main__':
    celery.start()
```

In this module you created our `Celery` instance (sometimes referred to as the *app*). To use Celery within your project you simply import this instance.

- The `broker` argument specifies the URL of the broker to use.

  See *Choosing a Broker* for more information.

- The `backend` argument specifies the result backend to use,

  It's used to keep track of task state and results. While results are disabled by default I use the amqp backend here because I demonstrate how retrieving results work later, you may want to use a different backend for your application. They all have different strengths and weaknesses. If you don't need results it's better to disable them. Results can also be disabled for individual tasks by setting the `@task(ignore_result=True)` option.

  See *Keeping Results* for more information.

- The `include` argument is a list of modules to import when the worker starts. You need to add our tasks module here so that the worker is able to find our tasks.

proj/tasks.py

```
from __future__ import absolute_import

from proj.celery import celery


@celery.task
def add(x, y):
    return x + y


@celery.task
def mul(x, y):
    return x * y


@celery.task
def xsum(numbers):
    return sum(numbers)
```

Starting the worker

The **celery** program can be used to start the worker:

```
$ celery worker --app=proj -l info
```

When the worker starts you should see a banner and some messages:

```
-------------- celery@halcyon.local v3.0 (Chiastic Slide)
---- **** -----
--- * ***  * -- [Configuration]
-- * - **** --- . broker:      amqp://guest@localhost:5672//
- ** ---------- . app:         __main__:0x1012d8590
- ** ---------- . concurrency: 8 (processes)
- ** ---------- . events:      OFF (enable -E to monitor this worker)
- ** ----------
- *** --- * --- [Queues]
-- ******* ---- . celery:      exchange:celery(direct) binding:celery
--- ***** -----

[2012-06-08 16:23:51,078: WARNING/MainProcess] celery@halcyon.local has started.
```

– The *broker* is the URL you specifed in the broker argument in our `celery` module, you can also specify a different broker on the command line by using the *-b* option.

– *Concurrency* is the number of multiprocessing worker process used to process your tasks concurrently, when all of these are busy doing work new tasks will have to wait for one of the tasks to finish before it can be processed.

The default concurrency number is the number of CPU's on that machine (including cores), you can specify a custom number using *-c* option. There is no recommended value, as the optimal number depends on a number of factors, but if your tasks are mostly I/O-bound then you can try to increase it, experimentation has shown that adding more than twice the number of CPU's is rarely effective, and likely to degrade performance instead.

Including the default multiprocessing pool, Celery also supports using Eventlet, Gevent, and threads (see *Concurrency*).

– *Events* is an option that when enabled causes Celery to send monitoring messages (events) for actions occurring in the worker. These can be used by monitor programs like `celery events`, and Flower - the real-time Celery monitor, which you can read about in the *Monitoring and Management guide*.

– *Queues* is the list of queues that the worker will consume tasks from. The worker can be told to consume from several queues at once, and this is used to route messages to specific workers as a means for Quality of Service, separation of concerns, and emulating priorities, all described in the *Routing Guide*.

You can get a complete list of command line arguments by passing in the *–help* flag:

```
$ celery worker --help
```

These options are described in more detailed in the *Workers Guide*.

Stopping the worker To stop the worker simply hit Ctrl+C. A list of signals supported by the worker is detailed in the *Workers Guide*.

In the background In production you will want to run the worker in the background, this is described in detail in the *daemonization tutorial*.

The daemonization scripts uses the **celery multi** command to start one or more workers in the background:

```
$ celery multi start w1 -A proj -l info
celeryd-multi v3.0.0 (Chiastic Slide)
> Starting nodes...
    > w1.halcyon.local: OK
```

You can restart it too:

```
$ celery multi restart w1 -A proj -l info
celeryd-multi v3.0.0 (Chiastic Slide)
> Stopping nodes...
    > w1.halcyon.local: TERM -> 64024
> Waiting for 1 node.....
    > w1.halcyon.local: OK
> Restarting node w1.halcyon.local: OK
celeryd-multi v3.0.0 (Chiastic Slide)
> Stopping nodes...
    > w1.halcyon.local: TERM -> 64052
```

or stop it:

```
$ celery multi stop -w1 -A proj -l info
```

Note: **celery multi** doesn't store information about workers so you need to use the same command line parameters when restarting. Also the same pidfile and logfile arguments must be used when stopping/killing.

By default it will create pid and log files in the current directory, to protect against multiple workers launching on top of each other you are encouraged to put these in a dedicated directory:

```
$ mkdir -p /var/run/celery
$ mkdir -p /var/log/celery
$ celery multi start w1 -A proj -l info --pidfile=/var/run/celery/%n.pid \
                                        --logfile=/var/log/celery/%n.pid
```

With the multi command you can start multiple workers, and there is a powerful command line syntax to specify arguments for different workers too, e.g:

```
$ celeryd multi start 10 -A proj -l info -Q:1-3 images,video -Q:4,5 data \
    -Q default -L:4,5 debug
```

For more examples see the `celeryd_multi` module in the API reference.

About the *--app* argument The *--app* argument specifies the Celery app instance to use, it must be in the form of `module.path:celery`, where the part before the colon is the name of the module, and the attribute name comes last. If a package name is specified instead it will automatically try to find a `celery` module in that package, and if the name is a module it will try to find a `celery` attribute in that module. This means that these are all equal:

```
$ celery --app=proj
$ celery --app=proj.celery:
$ celery --app=proj.celery:celery
```

Calling Tasks

You can call a task using the `delay()` method:

```
>>> add.delay(2, 2)
```

This method is actually a star-argument shortcut to another method called `apply_async()`:

```
>>> add.apply_async((2, 2))
```

The latter enables you to specify execution options like the time to run (countdown), the queue it should be sent to and so on:

```
>>> add.apply_async((2, 2), queue='lopri', countdown=10)
```

In the above example the task will be sent to a queue named `lopri` and the task will execute, at the earliest, 10 seconds after the message was sent.

Applying the task directly will execute the task in the current process, so that no message is sent:

```
>>> add(2, 2)
4
```

These three methods - `delay()`, `apply_async()`, and applying (`__call__`), represents the Celery calling API, which are also used for subtasks.

A more detailed overview of the Calling API can be found in the *Calling User Guide*.

Every task invocation will be given a unique identifier (an UUID), this is the task id.

The `delay` and `apply_async` methods return an `AsyncResult` instance, which can be used to keep track of the tasks execution state. But for this you need to enable a *result backend* so that the state can be stored somewhere.

Results are disabled by default because of the fact that there is no result backend that suits every application, so to choose one you need to consider the drawbacks of each individual backend. For many tasks keeping the return value isn't even very useful, so it's a sensible default to have. Also note that result backends are not used for monitoring tasks and workers, for that Celery uses dedicated event messages (see *Monitoring and Management Guide*).

If you have a result backend configured you can retrieve the return value of a task:

```
>>> res = add.delay(2, 2)
>>> res.get(timeout=1)
4
```

You can find the task's id by looking at the `id` attribute:

```
>>> res.id
d6b3aea2-fb9b-4ebc-8da4-848818db9114
```

You can also inspect the exception and traceback if the task raised an exception, in fact `result.get()` will propagate any errors by default:

```
>>> res = add.delay(2)
>>> res.get(timeout=1)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/opt/devel/celery/celery/result.py", line 113, in get
    interval=interval)
File "/opt/devel/celery/celery/backends/amqp.py", line 138, in wait_for
    raise self.exception_to_python(meta['result'])
TypeError: add() takes exactly 2 arguments (1 given)
```

If you don't wish for the errors to propagate then you can disable that by passing the `propagate` argument:

```
>>> res.get(propagate=False)
TypeError('add() takes exactly 2 arguments (1 given)',)
```

In this case it will return the exception instance raised instead, and so to check whether the task succeeded or failed you will have to use the corresponding methods on the result instance:

```
>>> res.failed()
True

>>> res.successful()
False
```

So how does it know if the task has failed or not? It can find out by looking at the tasks *state*:

```
>>> res.state
'FAILURE'
```

A task can only be in a single state, but it can progress through several states. The stages of a typical task can be:

```
PENDING -> STARTED -> SUCCESS
```

The started state is a special state that is only recorded if the `CELERY_TRACK_STARTED` setting is enabled, or if the `@task(track_started=True)` option is set for the task.

The pending state is actually not a recorded state, but rather the default state for any task id that is unknown, which you can see from this example:

```
>>> from proj.celery import celery

>>> res = celery.AsyncResult('this-id-does-not-exist')
>>> res.state
'PENDING'
```

If the task is retried the stages can become even more complex, e.g, for a task that is retried two times the stages would be:

```
PENDING -> STARTED -> RETRY -> STARTED -> RETRY -> STARTED -> SUCCESS
```

To read more about task states you should see the *States* section in the tasks user guide.

Calling tasks is described in detail in the *Calling Guide*.

Canvas: Designing Workflows

You just learned how to call a task using the tasks `delay` method, and this is often all you need, but sometimes you may want to pass the signature of a task invocation to another process or as an argument to another function, for this Celery uses something called *subtasks*.

A subtask wraps the arguments and execution options of a single task invocation in a way such that it can be passed to functions or even serialized and sent across the wire.

You can create a subtask for the `add` task using the arguments `(2, 2)`, and a countdown of 10 seconds like this:

```
>>> add.subtask((2, 2), countdown=10)
tasks.add(2, 2)
```

There is also a shortcut using star arguments:

```
>>> add.s(2, 2)
tasks.add(2, 2)
```

And there's that calling API again...

Subtask instances also supports the calling API, which means that they have the `delay` and `apply_async` methods.

But there is a difference in that the subtask may already have an argument signature specified. The `add` task takes two arguments, so a subtask specifying two arguments would make a complete signature:

```
>>> s1 = add.s(2, 2)
>>> res = s1.delay()
>>> res.get()
4
```

But, you can also make incomplete signatures to create what we call *partials*:

```
# incomplete partial: add(?, 2)
>>> s2 = add.s(2)
```

`s2` is now a partial subtask that needs another argument to be complete, and this can be resolved when calling the subtask:

```
# resolves the partial: add(8, 2)
>>> res = s2.delay(8)
>>> res.get()
10
```

Here you added the argument 8, which was prepended to the existing argument 2 forming a complete signature of `add(8, 2)`.

Keyword arguments can also be added later, these are then merged with any existing keyword arguments, but with new arguments taking precedence:

```
>>> s3 = add.s(2, 2, debug=True)
>>> s3.delay(debug=False)   # debug is now False.
```

As stated subtasks supports the calling API, which means that:

- `subtask.apply_async(args=(), kwargs={}, **options)`

  Calls the subtask with optional partial arguments and partial keyword arguments. Also supports partial execution options.

- `subtask.delay(*args, **kwargs)`

  Star argument version of `apply_async`. Any arguments will be prepended to the arguments in the signature, and keyword arguments is merged with any existing keys.

So this all seems very useful, but what can you actually do with these? To get to that I must introduce the canvas primitives...

The Primitives

- *group*
- *chain*
- *chord*
- *map*
- *starmap*
- *chunks*

The primitives are subtasks themselves, so that they can be combined in any number of ways to compose complex workflows.

Note: These examples retrieve results, so to try them out you need to configure a result backend. The example project above already does that (see the backend argument to `Celery`).

Let's look at some examples:

Groups A `group` calls a list of tasks in parallel, and it returns a special result instance that lets you inspect the results as a group, and retrieve the return values in order.

```
>>> from celery import group
>>> from proj.tasks import add

>>> group(add.s(i, i) for i in xrange(10))().get()
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

- Partial group

```
>>> g = group(add.s(i) for i in xrange(10))
>>> g(10).get()
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Chains Tasks can be linked together so that after one task returns the other is called:

```
>>> from celery import chain
>>> from proj.tasks import add, mul

# (4 + 4) * 8
```

```
>>> chain(add.s(4, 4) | mul.s(8))().get()
64
```

or a partial chain:

```
# (? + 4) * 8
>>> g = chain(add.s(4) | mul.s(8))
>>> g(4).get()
64
```

Chains can also be written like this:

```
>>> (add.s(4, 4) | mul.s(8))().get()
64
```

Chords A chord is a group with a callback:

```
>>> from celery import chord
>>> from proj.tasks import add, xsum

>>> chord((add.s(i, i) for i in xrange(10)), xsum.s())().get()
90
```

A group chained to another task will be automatically converted to a chord:

```
>>> (group(add.s(i, i) for i in xrange(10)) | xsum.s())().get()
90
```

Since these primitives are all of the subtask type they can be combined almost however you want, e.g:

```
>>> upload_document.s(file) | group(apply_filter.s() for filter in filters)
```

Be sure to read more about workflows in the *Canvas* user guide.

Routing

Celery supports all of the routing facilities provided by AMQP, but it also supports simple routing where messages are sent to named queues.

The `CELERY_ROUTES` setting enables you to route tasks by name and keep everything centralized in one location:

```
celery.conf.update(
    CELERY_ROUTES = {
        'proj.tasks.add': {'queue': 'hipri'},
    },
)
```

You can also specify the queue at runtime with the `queue` argument to `apply_async`:

```
>>> from proj.tasks import add
>>> add.apply_async((2, 2), queue='hipri')
```

You can then make a worker consume from this queue by specifying the `-Q` option:

```
$ celery -A proj worker -Q hipri
```

You may specify multiple queues by using a comma separated list, for example you can make the worker consume from both the default queue, and the `hipri` queue, where the default queue is named `celery` for historical reasons:

```
$ celery -A proj worker -Q hipri,celery
```

The order of the queues doesn't matter as the worker will give equal weight to the queues.

To learn more about routing, including taking use of the full power of AMQP routing, see the *Routing Guide*.

Remote Control

If you're using RabbitMQ (AMQP), Redis or MongoDB as the broker then you can control and inspect the worker at runtime.

For example you can see what tasks the worker is currently working on:

```
$ celery -A proj inspect active
```

This is implemented by using broadcast messaging, so all remote control commands are received by every worker in the cluster.

You can also specify one or more workers to act on the request using the `--destination` option, which is a comma separated list of worker host names:

```
$ celery -A proj inspect active --destination=worker1.example.com
```

If a destination is not provided then every worker will act and reply to the request.

The **celery inspect** command contains commands that does not change anything in the worker, it only replies information and statistics about what is going on inside the worker. For a list of inspect commands you can execute:

```
$ celery -A proj inspect --help
```

Then there is the **celery control** command, which contains commands that actually changes things in the worker at runtime:

```
$ celery -A proj control --help
```

For example you can force workers to enable event messages (used for monitoring tasks and workers):

```
$ celery -A proj control enable_events
```

When events are enabled you can then start the event dumper to see what the workers are doing:

```
$ celery -A proj events --dump
```

or you can start the curses interface:

```
$ celery -A proj events
```

when you're finished monitoring you can disable events again:

```
$ celery -A proj control disable_events
```

The **celery status** command also uses remote control commands and shows a list of online workers in the cluster:

```
$ celery -A proj status
```

You can read more about the **celery** command and monitoring in the *Monitoring Guide*.

Timezone

All times and dates, internally and in messages uses the UTC timezone.

When the worker receives a message, for example with a countdown set it converts that UTC time to local time. If you wish to use a different timezone than the system timezone then you must configure that using the `CELERY_TIMEZONE` setting.

To use custom timezones you also have to install the `pytz` library:

```
$ pip install pytz
```

Setting a custom timezone:

```
celery.conf.CELERY_TIMEZONE = 'Europe/London'
```

Optimization

The default configuration is not optimized for throughput by default, it tries to walk the middle way between many short tasks and fewer long tasks, a compromise between throughput and fair scheduling.

If you have strict fair scheduling requirements, or want to optimize for throughput then you should read the *Optimizing Guide*.

If you're using RabbitMQ then you should install the `librabbitmq` module, which is an AMQP client implemented in C:

```
$ pip install librabbitmq
```

What to do now?

Now that you have read this document you should continue to the *User Guide*.

There's also an *API reference* if you are so inclined.

2.2.5 Resources

Getting Help

Mailing list

For discussions about the usage, development, and future of celery, please join the celery-users mailing list.

IRC

Come chat with us on IRC. The **#celery** channel is located at the Freenode network.

Bug tracker

If you have any suggestions, bug reports or annoyances please report them to our issue tracker at http://github.com/celery/celery/issues/

Wiki

http://wiki.github.com/celery/celery/

Contributing

Development of *celery* happens at Github: http://github.com/celery/celery

You are highly encouraged to participate in the development of *celery*. If you don't like Github (for some reason) you're welcome to send regular patches.

Be sure to also read the Contributing to Celery section in the documentation.

License

This software is licensed under the *New BSD License*. See the `LICENSE` file in the top distribution directory for the full license text.

2.3 User Guide

Release 3.0

Date July 10, 2014

2.3.1 Application

The Celery library must be instantiated before use, this instance is called an application (or *app* for short).

The application is thread-safe so that multiple Celery applications with different configuration, components and tasks can co-exist in the same process space.

Let's create one now:

```
>>> from celery import Celery
>>> celery = Celery()
>>> celery
<Celery __main__:0x100469fd0>
```

The last line shows the textual representation of the application, which includes the name of the celery class (`Celery`), the name of the current main module (`__main__`), and the memory address of the object (`0x100469fd0`).

Main Name

Only one of these is important, and that is the main module name, let's look at why that is.

When you send a task message in Celery, that message will not contain any source code, but only the name of the task you want to execute. This works similarly to how host names works on the internet: every worker maintains a mapping of task names to their actual functions, called the *task registry*.

Whenever you define a task, that task will also be added to the local registry:

```
>>> @celery.task
... def add(x, y):
...     return x + y

>>> add
<@task: __main__.add>

>>> add.name
__main__.add

>>> celery.tasks['__main__.add']
<@task: __main__.add>
```

and there you see that `__main__` again; whenever Celery is not able to detect what module the function belongs to, it uses the main module name to generate the beginning of the task name.

This is only a problem in a limited set of use cases:

1. If the module that the task is defined in is run as a program.
2. If the application is created in the Python shell (REPL).

For example here, where the tasks module is also used to start a worker:

`tasks.py`:

```
from celery import Celery
celery = Celery()

@celery.task
def add(x, y): return x + y

if __name__ == '__main__':
    celery.worker_main()
```

When this module is executed the tasks will be named starting with "`__main__`", but when it the module is imported by another process, say to call a task, the tasks will be named starting with "`tasks`" (the real name of the module):

```
>>> from tasks import add
>>> add.name
tasks.add
```

You can specify another name for the main module:

```
>>> celery = Celery('tasks')
>>> celery.main
'tasks'

>>> @celery.task
... def add(x, y):
...     return x + y

>>> add.name
tasks.add
```

See also:

Names

Configuration

There are lots of different options you can set that will change how Celery work. These options can be set on the app instance directly, or you can use a dedicated configuration module.

The configuration is available as `Celery.conf`:

```
>>> celery.conf.CELERY_TIMEZONE
'Europe/London'
```

where you can set configuration values directly:

```
>>> celery.conf.CELERY_ENABLE_UTC = True
```

or you can update several keys at once by using the `update` method:

```
>>> celery.conf.update(
...     CELERY_ENABLE_UTC=True,
...     CELERY_TIMEZONE='Europe/London',
...)
```

The configuration object consists of multiple dictionaries that are consulted in order:

1. Changes made at runtime.
2. The configuration module (if any)
3. The default configuration (`celery.app.defaults`).

See also:

Go to the *Configuration reference* for a complete listing of all the available settings, and their default values.

config_from_object

Timezones & pytz

Setting a time zone other than UTC requires the `pytz` library to be installed, see the `CELERY_TIMEZONE` setting for more information.

The `Celery.config_from_object()` method loads configuration from a configuration object.

This can be a configuration module, or any object with configuration attributes.

Note that any configuration that was previous set will be reset when `config_from_object()` is called. If you want to set additional configuration you should do so after.

Example 1: Using the name of a module

```
from celery import Celery

celery = Celery()
celery.config_from_object('celeryconfig')
```

The `celeryconfig` module may then look like this:

`celeryconfig.py`:

```
CELERY_ENABLE_UTC = True
CELERY_TIMEZONE = 'Europe/London'
```

Example 2: Using a configuration module

```
from celery import Celery

celery = Celery()
import celeryconfig
celery.config_from_object(celeryconfig)
```

Example 3: Using a configuration class/object

```
from celery import Celery

celery = Celery()

class Config:
    CELERY_ENABLE_UTC = True
    CELERY_TIMEZONE = 'Europe/London'

celery.config_from_object(Config)
```

config_from_envvar

The `Celery.config_from_envvar()` takes the configuration module name from an environment variable

For example – to load configuration from a module specified in the environment variable named `CELERY_CONFIG_MODULE`:

```
import os
from celery import Celery

#: Set default configuration module name
os.environ.setdefault('CELERY_CONFIG_MODULE', 'celeryconfig')

celery = Celery()
celery.config_from_envvar('CELERY_CONFIG_MODULE')
```

You can then specify the configuration module to use via the environment:

```
$ CELERY_CONFIG_MODULE="celeryconfig.prod" celery worker -l info
```

Laziness

The application instance is lazy, meaning that it will not be evaluated until something is actually needed.

Creating a `Celery` instance will only do the following:

1. Create a logical clock instance, used for events.
2. Create the task registry.
3. Set itself as the current app (but not if the `set_as_current` argument was disabled)
4. Call the `Celery.on_init()` callback (does nothing by default).

The `task()` decorator does not actually create the tasks at the point when it's called, instead it will defer the creation of the task to happen either when the task is used, or after the application has been *finalized*,

This example shows how the task is not created until you use the task, or access an attribute (in this case `repr()`):

```
>>> @celery.task
>>> def add(x, y):
...     return x + y

>>> type(add)
<class 'celery.local.PromiseProxy'>

>>> add.__evaluated__()
False

>>> add         # <-- causes repr(add) to happen
<@task: __main__.add>

>>> add.__evaluated__()
True
```

Finalization of the app happens either explicitly by calling `Celery.finalize()` – or implicitly by accessing the `tasks` attribute.

Finalizing the object will:

1. Copy tasks that must be shared between apps

   Tasks are shared by default, but if the `shared` argument to the task decorator is disabled, then the task will be private to the app it's bound to.

2. Evaluate all pending task decorators.
3. Make sure all tasks are bound to the current app.

   Tasks are bound to apps so that it can read default values from the configuration.

The "default app".

Celery did not always work this way, it used to be that there was only a module-based API, and for backwards compatibility the old API is still there.
Celery always creates a special app that is the "default app", and this is used if no custom application has been instantiated.
The `celery.task` module is there to accommodate the old API, and should not be used if you use a custom app. You should always use the methods on the app instance, not the module based API.
For example, the old Task base class enables many compatibility features where some may be incompatible with newer features, such as task methods:

```
from celery.task import Task    # << OLD Task base class.

from celery import Task         # << NEW base class.
```

The new base class is recommended even if you use the old module-based API.

Breaking the chain

While it's possible to depend on the current app being set, the best practice is to always pass the app instance around to anything that needs it.

I call this the "app chain", since it creates a chain of instances depending on the app being passed.

The following example is considered bad practice:

```
from celery import current_app

class Scheduler(object):

    def run(self):
        app = current_app
```

Instead it should take the `app` as an argument:

```
class Scheduler(object):

    def __init__(self, app):
        self.app = app
```

Internally Celery uses the `celery.app.app_or_default()` function so that everything also works in the module-based compatibility API

```
from celery.app import app_or_default

class Scheduler(object):
    def __init__(self, app=None):
        self.app = app_or_default(app)
```

In development you can set the `CELERY_TRACE_APP` environment variable to raise an exception if the app chain breaks:

```
$ CELERY_TRACE_APP=1 celery worker -l info
```

Evolving the API

Celery has changed a lot in the 3 years since it was initially created.
For example, in the beginning it was possible to use any callable as a task:

```
def hello(to):
    return 'hello %s' % to

>>> from celery.execute import apply_async

>>> apply_async(hello, ('world!', ))
```

or you could also create a `Task` class to set certain options, or override other behavior

```
from celery.task import Task
from celery.registry import tasks

class Hello(Task):
    send_error_emails = True

    def run(self, to):
        return 'hello %s' % to
tasks.register(Hello)

>>> Hello.delay('world!')
```

Later, it was decided that passing arbitrary call-ables was an anti-pattern, since it makes it very hard to use serializers other than pickle, and the feature was removed in 2.0, replaced by task decorators:

```
from celery.task import task

@task(send_error_emails=True)
def hello(x):
    return 'hello %s' % to
```

Abstract Tasks

All tasks created using the `task()` decorator will inherit from the applications base `Task` class.

You can specify a different base class with the `base` argument:

```
@celery.task(base=OtherTask):
def add(x, y):
    return x + y
```

To create a custom task class you should inherit from the neutral base class: `celery.Task`.

```
from celery import Task

class DebugTask(Task):
    abstract = True

    def __call__(self, *args, **kwargs):
        print('TASK STARTING: %s[%s]' % (self.name, self.request.id))
        return self.run(*args, **kwargs)
```

The neutral base class is special because it's not bound to any specific app yet. Concrete subclasses of this class will be bound, so you should always mark generic base classes as `abstract`

Once a task is bound to an app it will read configuration to set default values and so on.

It's also possible to change the default base class for an application by changing its `Celery.Task()` attribute:

```
>>> from celery import Celery, Task

>>> celery = Celery()

>>> class MyBaseTask(Task):
...     abstract = True
...     send_error_emails = True

>>> celery.Task = MyBaseTask
>>> celery.Task
<unbound MyBaseTask>

>>> @x.task
... def add(x, y):
...     return x + y

>>> add
<@task: __main__.add>

>>> add.__class__.mro()
[<class add of <Celery __main__:0x1012b4410>>,
 <unbound MyBaseTask>,
 <unbound Task>,
 <type 'object'>]
```

2.3.2 Tasks

Tasks are the building blocks of Celery applications.

A task is a class that can be created out of any callable. It performs dual roles in that it defines both what happens when a task is called (sends a message), and what happens when a worker receives that message.

Every task class has a unique name, and this name is referenced in messages so that the worker can find the right function to execute.

A task message does not disappear until the message has been *acknowledged* by a worker. A worker can reserve many messages in advance and even if the worker is killed – caused by power failure or otherwise – the message will be redelivered to another worker.

Ideally task functions should be *idempotent*, which means that the function will not cause unintented effects even if called multiple times with the same arguments. Since the worker cannot detect if your tasks are idempotent, the default behavior is to acknowledge the message in advance, before it's executed, so that a task that has already been started is never executed again..

If your task is idempotent you can set the `acks_late` option to have the worker acknowledge the message *after* the task returns instead. See also the FAQ entry *Should I use retry or acks_late?*.

–

In this chapter you will learn all about defining tasks, and this is the **table of contents**:

Basics

You can easily create a task from any callable by using the `task()` decorator:

```
from .models import User

@celery.task
def create_user(username, password):
    User.objects.create(username=username, password=password)
```

There are also many *options* that can be set for the task, these can be specified as arguments to the decorator:

```
@celery.task(serializer='json')
def create_user(username, password):
    User.objects.create(username=username, password=password)
```

How do I import the task decorator?

The task decorator is available on your `Celery` instance, if you don't know what that is then please read *First Steps with Celery*.
If you're using Django or are still using the "old" module based celery API, then you can import the task decorator like this:

```
from celery import task

@task
def add(x, y):
    return x + y
```

Multiple decorators

When using multiple decorators in combination with the task decorator you must make sure that the *task* decorator is applied last (which in Python oddly means that it must be the first in the list):

```
@celery.task
@decorator2
@decorator1
def add(x, y):
    return x + y
```

Names

Every task must have a unique name, and a new name will be generated out of the function name if a custom name is not provided.

For example:

```
>>> @celery.task(name='sum-of-two-numbers')
>>> def add(x, y):
...     return x + y

>>> add.name
'sum-of-two-numbers'
```

A best practice is to use the module name as a namespace, this way names won't collide if there's already a task with that name defined in another module.

```
>>> @celery.task(name='tasks.add')
>>> def add(x, y):
...     return x + y
```

You can tell the name of the task by investigating its name attribute:

```
>>> add.name
'tasks.add'
```

Which is exactly the name that would have been generated anyway, if the module name is "tasks.py":

`tasks.py`:

```
@celery.task
def add(x, y):
    return x + y

>>> from tasks import add
>>> add.name
'tasks.add'
```

Automatic naming and relative imports

Relative imports and automatic name generation does not go well together, so if you're using relative imports you should set the name explicitly.

For example if the client imports the module "myapp.tasks" as ".tasks", and the worker imports the module as "myapp.tasks", the generated names won't match and an `NotRegistered` error will be raised by the worker.

This is also the case if using Django and using *project.myapp*:

```
INSTALLED_APPS = ('project.myapp', )
```

The worker will have the tasks registered as "project.myapp.tasks.*", while this is what happens in the client if the module is imported as "myapp.tasks":

```
>>> from myapp.tasks import add
>>> add.name
'myapp.tasks.add'
```

For this reason you should never use "project.app", but rather add the project directory to the Python path:

```
import os
import sys
sys.path.append(os.path.dirname(os.path.realpath(__file__)))

INSTALLED_APPS = ('myapp', )
```

This makes more sense from the reusable app perspective anyway.

Context

`request` contains information and state related to the executing task.

The request defines the following attributes:

id The unique id of the executing task.

taskset The unique id of the taskset this task is a member of (if any).

chord The unique id of the chord this task belongs to (if the task is part of the header).

args Positional arguments.

kwargs Keyword arguments.

retries How many times the current task has been retried. An integer starting at *0*.

is_eager Set to `True` if the task is executed locally in the client, and not by a worker.

eta The original ETA of the task (if any). This is in UTC time (depending on the `CELERY_ENABLE_UTC` setting).

expires The original expiry time of the task (if any). This is in UTC time (depending on the `CELERY_ENABLE_UTC` setting).

logfile The file the worker logs to. See Logging.

loglevel The current log level used.

hostname Hostname of the worker instance executing the task.

delivery_info Additional message delivery information. This is a mapping containing the exchange and routing key used to deliver this task. Used by e.g. `retry()` to resend the task to the same destination queue. Availability of keys in this dict depends on the message broker used.

called_directly This flag is set to true if the task was not executed by the worker.

callbacks A list of subtasks to be called if this task returns successfully.

errback A list of subtasks to be called if this task fails.

utc Set to true the caller has utc enabled (`CELERY_ENABLE_UTC`).

An example task accessing information in the context is:

```
@celery.task
def add(x, y):
    print('Executing task id %r, args: %r kwargs: %r' % (
        add.request.id, add.request.args, add.request.kwargs))
    return x + y
```

`current_task` can also be used:

```
from celery import current_task

@celery.task
def add(x, y):
    request = current_task.request
    print('Executing task id %r, args: %r kwargs: %r' % (
        request.id, request.args, request.kwargs))
    return x + y
```

Logging

The worker will automatically set up logging for you, or you can configure logging manually.

A special logger is available named "celery.task", you can inherit from this logger to automatically get the task name and unique id as part of the logs.

The best practice is to create a common logger for all of your tasks at the top of your module:

```
from celery.utils.log import get_task_logger

logger = get_task_logger(__name__)

@celery.task
def add(x, y):
    logger.info('Adding %s + %s' % (x, y))
    return x + y
```

Celery uses the standard Python logger library, for which documentation can be found in the `logging` module.

You can also simply use `print()`, as anything written to standard out/-err will be redirected to the workers logs by default (see `CELERY_REDIRECT_STDOUTS`).

Retrying

`retry()` can be used to re-execute the task, for example in the event of recoverable errors.

When you call `retry` it will send a new message, using the same task-id, and it will take care to make sure the message is delivered to the same queue as the originating task.

When a task is retried this is also recorded as a task state, so that you can track the progress of the task using the result instance (see *States*).

Here's an example using `retry`:

```
@celery.task
def send_twitter_status(oauth, tweet):
    try:
        twitter = Twitter(oauth)
        twitter.update_status(tweet)
    except (Twitter.FailWhaleError, Twitter.LoginError), exc:
        raise send_twitter_status.retry(exc=exc)
```

Note: The `retry()` call will raise an exception so any code after the retry will not be reached. This is the `RetryTaskError` exception, it is not handled as an error but rather as a semi-predicate to signify to the worker that the task is to be retried, so that it can store the correct state when a result backend is enabled.

This is normal operation and always happens unless the `throw` argument to retry is set to `False`.

The `exc` method is used to pass exception information that is used in logs, and when storing task results. Both the exception and the traceback will be available in the task state (if a result backend is enabled).

If the task has a `max_retries` value the current exception will be re-raised if the max number of retries has been exceeded, but this will not happen if:

- An `exc` argument was not given.

  In this case the `celery.exceptions.MaxRetriesExceeded` exception will be raised.

- There is no current exception

  If there's no original exception to re-raise the `exc` argument will be used instead, so:

  ```
  send_twitter_status.retry(exc=Twitter.LoginError())
  ```

  will raise the `exc` argument given.

Using a custom retry delay

When a task is to be retried, it can wait for a given amount of time before doing so, and the default delay is defined by the `default_retry_delay` attribute. By default this is set to 3 minutes. Note that the unit for setting the delay is in seconds (int or float).

You can also provide the *countdown* argument to `retry()` to override this default.

```
@celery.task(default_retry_delay=30 * 60)  # retry in 30 minutes.
def add(x, y):
    try:
        ...
    except Exception, exc:
        raise add.retry(exc=exc, countdown=60)  # override the default and
                                                # retry in 1 minute
```

List of Options

The task decorator can take a number of options that change the way the task behaves, for example you can set the rate limit for a task using the `rate_limit` option.

Any keyword argument passed to the task decorator will actually be set as an attribute of the resulting task class, and this is a list of the built-in attributes.

General

`Task.`**`name`**
: The name the task is registered as.

  You can set this name manually, or a name will be automatically generated using the module and class name. See *Names*.

`Task.`**`request`**
: If the task is being executed this will contain information about the current request. Thread local storage is used.

  See *Context*.

`Task.`**`abstract`**
: Abstract classes are not registered, but are used as the base class for new task types.

Task.**max_retries**
The maximum number of attempted retries before giving up. If the number of retries exceeds this value a `MaxRetriesExceeded` exception will be raised. *NOTE:* You have to call `retry()` manually, as it will not automatically retry on exception..

Task.**default_retry_delay**
Default time in seconds before a retry of the task should be executed. Can be either `int` or `float`. Default is a 3 minute delay.

Task.**rate_limit**
Set the rate limit for this task type which limits the number of tasks that can be run in a given time frame. Tasks will still complete when a rate limit is in effect, but it may take some time before it's allowed to start.

If this is `None` no rate limit is in effect. If it is an integer or float, it is interpreted as "tasks per second".

The rate limits can be specified in seconds, minutes or hours by appending *"/s"*, *"/m"* or *"/h"* to the value. Example: *"100/m"* (hundred tasks a minute). Default is the `CELERY_DEFAULT_RATE_LIMIT` setting, which if not specified means rate limiting for tasks is disabled by default.

Task.**time_limit**
The hard time limit for this task. If not set then the workers default will be used.

Task.**soft_time_limit**
The soft time limit for this task. If not set then the workers default will be used.

Task.**ignore_result**
Don't store task state. Note that this means you can't use `AsyncResult` to check if the task is ready, or get its return value.

Task.**store_errors_even_if_ignored**
If `True`, errors will be stored even if the task is configured to ignore results.

Task.**send_error_emails**
Send an email whenever a task of this type fails. Defaults to the `CELERY_SEND_TASK_ERROR_EMAILS` setting. See *Error E-Mails* for more information.

Task.**ErrorMail**
If the sending of error emails is enabled for this task, then this is the class defining the logic to send error mails.

Task.**serializer**
A string identifying the default serialization method to use. Defaults to the `CELERY_TASK_SERIALIZER` setting. Can be *pickle* *json*, *yaml*, or any custom serialization methods that have been registered with `kombu.serialization.registry`.

Please see *Serializers* for more information.

Task.**compression**
A string identifying the default compression scheme to use.

Defaults to the `CELERY_MESSAGE_COMPRESSION` setting. Can be *gzip*, or *bzip2*, or any custom compression schemes that have been registered with the `kombu.compression` registry.

Please see *Compression* for more information.

Task.**backend**
The result store backend to use for this task. Defaults to the `CELERY_RESULT_BACKEND` setting.

Task.**acks_late**
If set to `True` messages for this task will be acknowledged **after** the task has been executed, not *just before*, which is the default behavior.

Note that this means the task may be executed twice if the worker crashes in the middle of execution, which may be acceptable for some applications.

The global default can be overridden by the CELERY_ACKS_LATE setting.

Task.**track_started**
If True the task will report its status as "started" when the task is executed by a worker. The default value is False as the normal behaviour is to not report that level of granularity. Tasks are either pending, finished, or waiting to be retried. Having a "started" status can be useful for when there are long running tasks and there is a need to report which task is currently running.

The host name and process id of the worker executing the task will be available in the state metadata (e.g. *result.info['pid']*)

The global default can be overridden by the CELERY_TRACK_STARTED setting.

See also:

The API reference for Task.

States

Celery can keep track of the tasks current state. The state also contains the result of a successful task, or the exception and traceback information of a failed task.

There are several *result backends* to choose from, and they all have different strengths and weaknesses (see *Result Backends*).

During its lifetime a task will transition through several possible states, and each state may have arbitrary metadata attached to it. When a task moves into a new state the previous state is forgotten about, but some transitions can be deducted, (e.g. a task now in the FAILED state, is implied to have been in the STARTED state at some point).

There are also sets of states, like the set of FAILURE_STATES, and the set of READY_STATES.

The client uses the membership of these sets to decide whether the exception should be re-raised (PROPAGATE_STATES), or whether the state can be cached (it can if the task is ready).

You can also define *Custom states*.

Result Backends

Celery needs to store or send the states somewhere. There are several built-in backends to choose from: SQLAlchemy/Django ORM, Memcached, RabbitMQ (amqp), MongoDB, and Redis – or you can define your own.

No backend works well for every use case. You should read about the strengths and weaknesses of each backend, and choose the most appropriate for your needs.

See also:

Task result backend settings

RabbitMQ Result Backend The RabbitMQ result backend (amqp) is special as it does not actually *store* the states, but rather sends them as messages. This is an important difference as it means that a result *can only be retrieved once*; If you have two processes waiting for the same result, one of the processes will never receive the result!

Even with that limitation, it is an excellent choice if you need to receive state changes in real-time. Using messaging means the client does not have to poll for new states.

There are several other pitfalls you should be aware of when using the RabbitMQ result backend:

- Every new task creates a new queue on the server, with thousands of tasks the broker may be overloaded with queues and this will affect performance in negative ways. If you're using RabbitMQ then each queue will be a separate Erlang process, so if you're planning to keep many results simultaneously you may have to increase the Erlang process limit, and the maximum number of file descriptors your OS allows.
- Old results will be cleaned automatically, based on the `CELERY_TASK_RESULT_EXPIRES` setting. By default this is set to expire after 1 day: if you have a very busy cluster you should lower this value.

For a list of options supported by the RabbitMQ result backend, please see *AMQP backend settings*.

Database Result Backend Keeping state in the database can be convenient for many, especially for web applications with a database already in place, but it also comes with limitations.

- Polling the database for new states is expensive, and so you should increase the polling intervals of operations such as *result.get()*.
- Some databases use a default transaction isolation level that is not suitable for polling tables for changes.

  In MySQL the default transaction isolation level is *REPEATABLE-READ*, which means the transaction will not see changes by other transactions until the transaction is committed. It is recommended that you change to the *READ-COMMITTED* isolation level.

Built-in States

PENDING Task is waiting for execution or unknown. Any task id that is not known is implied to be in the pending state.

STARTED Task has been started. Not reported by default, to enable please see `celery.Task.track_started`.

metadata *pid* and *hostname* of the worker process executing the task.

SUCCESS Task has been successfully executed.

metadata *result* contains the return value of the task.

propagates Yes

ready Yes

FAILURE Task execution resulted in failure.

metadata *result* contains the exception occurred, and *traceback* contains the backtrace of the stack at the point when the exception was raised.

propagates Yes

RETRY Task is being retried.

metadata *result* contains the exception that caused the retry, and *traceback* contains the backtrace of the stack at the point when the exceptions was raised.

propagates No

REVOKED Task has been revoked.

propagates Yes

Custom states

You can easily define your own states, all you need is a unique name. The name of the state is usually an uppercase string. As an example you could have a look at `abortable tasks` which defines its own custom `ABORTED` state.

Use `update_state()` to update a task's state:

```
from celery import current_task

@celery.task
def upload_files(filenames):
    for i, file in enumerate(filenames):
        current_task.update_state(state='PROGRESS',
            meta={'current': i, 'total': len(filenames)})
```

Here I created the state *"PROGRESS"*, which tells any application aware of this state that the task is currently in progress, and also where it is in the process by having *current* and *total* counts as part of the state metadata. This can then be used to create e.g. progress bars.

Creating pickleable exceptions

A rarely known Python fact is that exceptions must conform to some simple rules to support being serialized by the pickle module.

Tasks that raise exceptions that are not pickleable will not work properly when Pickle is used as the serializer.

To make sure that your exceptions are pickleable the exception *MUST* provide the original arguments it was instantiated with in its `.args` attribute. The simplest way to ensure this is to have the exception call `Exception.__init__`.

Let's look at some examples that work, and one that doesn't:

```
# OK:
class HttpError(Exception):
    pass

# BAD:
class HttpError(Exception):

    def __init__(self, status_code):
        self.status_code = status_code

# OK:
class HttpError(Exception):

    def __init__(self, status_code):
        self.status_code = status_code
        Exception.__init__(self, status_code)  # <-- REQUIRED
```

So the rule is: For any exception that supports custom arguments `*args`, `Exception.__init__(self, *args)` must be used.

There is no special support for *keyword arguments*, so if you want to preserve keyword arguments when the exception is unpickled you have to pass them as regular args:

```
class HttpError(Exception):

    def __init__(self, status_code, headers=None, body=None):
        self.status_code = status_code
        self.headers = headers
        self.body = body

        super(HttpError, self).__init__(status_code, headers, body)
```

Custom task classes

All tasks inherit from the `celery.Task` class. The `run()` method becomes the task body.

As an example, the following code,

```
@celery.task
def add(x, y):
    return x + y
```

will do roughly this behind the scenes:

```
@celery.task
class AddTask(Task):

    def run(self, x, y):
        return x + y
add = registry.tasks[AddTask.name]
```

Instantiation

A task is **not** instantiated for every request, but is registered in the task registry as a global instance.

This means that the `__init__` constructor will only be called once per process, and that the task class is semantically closer to an Actor.

If you have a task,

```
from celery import Task

class NaiveAuthenticateServer(Task):

    def __init__(self):
        self.users = {'george': 'password'}

    def run(self, username, password):
        try:
            return self.users[username] == password
        except KeyError:
            return False
```

And you route every request to the same process, then it will keep state between requests.

This can also be useful to cache resources, e.g. a base Task class that caches a database connection:

```
from celery import Task

class DatabaseTask(Task):
    abstract = True
```

```
    _db = None

    @property
    def db(self):
        if self._db is None:
            self._db = Database.connect()
        return self._db
```

that can be added to tasks like this:

```
@celery.task(base=DatabaseTask)
def process_rows():
    for row in process_rows.db.table.all():
        ...
```

The `db` attribute of the `process_rows` task will then always stay the same in each process.

Abstract classes

Abstract classes are not registered, but are used as the base class for new task types.

```
from celery import Task

class DebugTask(Task):
    abstract = True

    def after_return(self, *args, **kwargs):
        print('Task returned: %r' % (self.request, ))


@celery.task(base=DebugTask)
def add(x, y):
    return x + y
```

Handlers

after_return(*self*, *status*, *retval*, *task_id*, *args*, *kwargs*, *einfo*)
Handler called after the task returns.

Parameters

- **status** – Current task state.
- **retval** – Task return value/exception.
- **task_id** – Unique id of the task.
- **args** – Original arguments for the task that failed.
- **kwargs** – Original keyword arguments for the task that failed.
- **einfo** – `ExceptionInfo` instance, containing the traceback (if any).

The return value of this handler is ignored.

on_failure(*self*, *exc*, *task_id*, *args*, *kwargs*, *einfo*)
This is run by the worker when the task fails.

Parameters

- **exc** – The exception raised by the task.
- **task_id** – Unique id of the failed task.
- **args** – Original arguments for the task that failed.
- **kwargs** – Original keyword arguments for the task that failed.
- **einfo** – `ExceptionInfo` instance, containing the traceback.

The return value of this handler is ignored.

on_retry(*self*, *exc*, *task_id*, *args*, *kwargs*, *einfo*)
This is run by the worker when the task is to be retried.

Parameters

- **exc** – The exception sent to `retry()`.
- **task_id** – Unique id of the retried task.
- **args** – Original arguments for the retried task.
- **kwargs** – Original keyword arguments for the retried task.
- **einfo** – `ExceptionInfo` instance, containing the traceback.

The return value of this handler is ignored.

on_success(*self*, *retval*, *task_id*, *args*, *kwargs*)
Run by the worker if the task executes successfully.

Parameters

- **retval** – The return value of the task.
- **task_id** – Unique id of the executed task.
- **args** – Original arguments for the executed task.
- **kwargs** – Original keyword arguments for the executed task.

The return value of this handler is ignored.

on_retry

How it works

Here comes the technical details, this part isn't something you need to know, but you may be interested.

All defined tasks are listed in a registry. The registry contains a list of task names and their task classes. You can investigate this registry yourself:

```
>>> from celery import current_app
>>> current_app.tasks
{'celery.chord_unlock':
    <@task: celery.chord_unlock>,
 'celery.backend_cleanup':
    <@task: celery.backend_cleanup>,
 'celery.chord':
    <@task: celery.chord>}
```

This is the list of tasks built-in to celery. Note that tasks will only be registered when the module they are defined in is imported.

The default loader imports any modules listed in the `CELERY_IMPORTS` setting.

The entity responsible for registering your task in the registry is the metaclass: `TaskType`.

If you want to register your task manually you can mark the task as `abstract`:

```
class MyTask(Task):
    abstract = True
```

This way the task won't be registered, but any task inheriting from it will be.

When tasks are sent, no actual function code is sent with it, just the name of the task to execute. When the worker then receives the message it can look up the name in its task registry to find the execution code.

This means that your workers should always be updated with the same software as the client. This is a drawback, but the alternative is a technical challenge that has yet to be solved.

Tips and Best Practices

Ignore results you don't want

If you don't care about the results of a task, be sure to set the `ignore_result` option, as storing results wastes time and resources.

```
@celery.task(ignore_result=True)
def mytask(...)
    something()
```

Results can even be disabled globally using the `CELERY_IGNORE_RESULT` setting.

Disable rate limits if they're not used

Disabling rate limits altogether is recommended if you don't have any tasks using them. This is because the rate limit subsystem introduces quite a lot of complexity.

Set the `CELERY_DISABLE_RATE_LIMITS` setting to globally disable rate limits:

```
CELERY_DISABLE_RATE_LIMITS = True
```

You find additional optimization tips in the *Optimizing Guide*.

Avoid launching synchronous subtasks

Having a task wait for the result of another task is really inefficient, and may even cause a deadlock if the worker pool is exhausted.

Make your design asynchronous instead, for example by using *callbacks*.

Bad:

```
@celery.task
def update_page_info(url):
    page = fetch_page.delay(url).get()
    info = parse_page.delay(url, page).get()
    store_page_info.delay(url, info)
```

```
@celery.task
def fetch_page(url):
    return myhttplib.get(url)

@celery.task
def parse_page(url, page):
    return myparser.parse_document(page)

@celery.task
def store_page_info(url, info):
    return PageInfo.objects.create(url, info)
```

Good:

```
def update_page_info(url):
    # fetch_page -> parse_page -> store_page
    chain = fetch_page.s() | parse_page.s() | store_page_info.s(url)
    chain()

@celery.task()
def fetch_page(url):
    return myhttplib.get(url)

@celery.task()
def parse_page(page):
    return myparser.parse_document(page)

@celery.task(ignore_result=True)
def store_page_info(info, url):
    PageInfo.objects.create(url=url, info=info)
```

Here I instead created a chain of tasks by linking together different `subtask()`'s. You can read about chains and other powerful constructs at *Canvas: Designing Workflows*.

Performance and Strategies

Granularity

The task granularity is the amount of computation needed by each subtask. In general it is better to split the problem up into many small tasks, than have a few long running tasks.

With smaller tasks you can process more tasks in parallel and the tasks won't run long enough to block the worker from processing other waiting tasks.

However, executing a task does have overhead. A message needs to be sent, data may not be local, etc. So if the tasks are too fine-grained the additional overhead may not be worth it in the end.

See also:

The book Art of Concurrency has a section dedicated to the topic of task granularity [AOC1].

Data locality

The worker processing the task should be as close to the data as possible. The best would be to have a copy in memory, the worst would be a full transfer from another continent.

If the data is far away, you could try to run another worker at location, or if that's not possible - cache often used data, or preload data you know is going to be used.

The easiest way to share data between workers is to use a distributed cache system, like memcached.

See also:

The paper Distributed Computing Economics by Jim Gray is an excellent introduction to the topic of data locality.

State

Since celery is a distributed system, you can't know in which process, or on what machine the task will be executed. You can't even know if the task will run in a timely manner.

The ancient async sayings tells us that "asserting the world is the responsibility of the task". What this means is that the world view may have changed since the task was requested, so the task is responsible for making sure the world is how it should be; If you have a task that re-indexes a search engine, and the search engine should only be re-indexed at maximum every 5 minutes, then it must be the tasks responsibility to assert that, not the callers.

Another gotcha is Django model objects. They shouldn't be passed on as arguments to tasks. It's almost always better to re-fetch the object from the database when the task is running instead, as using old data may lead to race conditions.

Imagine the following scenario where you have an article and a task that automatically expands some abbreviations in it:

```
class Article(models.Model):
    title = models.CharField()
    body = models.TextField()

@celery.task
def expand_abbreviations(article):
    article.body.replace('MyCorp', 'My Corporation')
    article.save()
```

First, an author creates an article and saves it, then the author clicks on a button that initiates the abbreviation task:

```
>>> article = Article.objects.get(id=102)
>>> expand_abbreviations.delay(model_object)
```

Now, the queue is very busy, so the task won't be run for another 2 minutes. In the meantime another author makes changes to the article, so when the task is finally run, the body of the article is reverted to the old version because the task had the old body in its argument.

Fixing the race condition is easy, just use the article id instead, and re-fetch the article in the task body:

```
@celery.task
def expand_abbreviations(article_id):
    article = Article.objects.get(id=article_id)
    article.body.replace('MyCorp', 'My Corporation')
    article.save()

>>> expand_abbreviations(article_id)
```

There might even be performance benefits to this approach, as sending large messages may be expensive.

Database transactions

Let's have a look at another example:

```
from django.db import transaction

@transaction.commit_on_success
def create_article(request):
    article = Article.objects.create(....)
    expand_abbreviations.delay(article.pk)
```

This is a Django view creating an article object in the database, then passing the primary key to a task. It uses the *commit_on_success* decorator, which will commit the transaction when the view returns, or roll back if the view raises an exception.

There is a race condition if the task starts executing before the transaction has been committed; The database object does not exist yet!

The solution is to *always commit transactions before sending tasks depending on state from the current transaction*:

```
@transaction.commit_manually
def create_article(request):
    try:
        article = Article.objects.create(...)
    except:
        transaction.rollback()
        raise
    else:
        transaction.commit()
        expand_abbreviations.delay(article.pk)
```

Example

Let's take a real wold example; A blog where comments posted needs to be filtered for spam. When the comment is created, the spam filter runs in the background, so the user doesn't have to wait for it to finish.

I have a Django blog application allowing comments on blog posts. I'll describe parts of the models/views and tasks for this application.

blog/models.py

The comment model looks like this:

```
from django.db import models
from django.utils.translation import ugettext_lazy as _


class Comment(models.Model):
    name = models.CharField(_('name'), max_length=64)
    email_address = models.EmailField(_('email address'))
    homepage = models.URLField(_('home page'),
                               blank=True, verify_exists=False)
    comment = models.TextField(_('comment'))
    pub_date = models.DateTimeField(_('Published date'),
                                    editable=False, auto_add_now=True)
    is_spam = models.BooleanField(_('spam?'),
                                  default=False, editable=False)

    class Meta:
        verbose_name = _('comment')
        verbose_name_plural = _('comments')
```

In the view where the comment is posted, I first write the comment to the database, then I launch the spam filter task in the background.

blog/views.py

```
from django import forms
from django.http import HttpResponseRedirect
from django.template.context import RequestContext
from django.shortcuts import get_object_or_404, render_to_response

from blog import tasks
from blog.models import Comment


class CommentForm(forms.ModelForm):

    class Meta:
        model = Comment


def add_comment(request, slug, template_name='comments/create.html'):
    post = get_object_or_404(Entry, slug=slug)
    remote_addr = request.META.get('REMOTE_ADDR')

    if request.method == 'post':
        form = CommentForm(request.POST, request.FILES)
        if form.is_valid():
            comment = form.save()
            # Check spam asynchronously.
            tasks.spam_filter.delay(comment_id=comment.id,
                                    remote_addr=remote_addr)
            return HttpResponseRedirect(post.get_absolute_url())
    else:
        form = CommentForm()

    context = RequestContext(request, {'form': form})
    return render_to_response(template_name, context_instance=context)
```

To filter spam in comments I use Akismet, the service used to filter spam in comments posted to the free weblog platform *Wordpress*. Akismet is free for personal use, but for commercial use you need to pay. You have to sign up to their service to get an API key.

To make API calls to Akismet I use the akismet.py library written by Michael Foord.

blog/tasks.py

```
import celery

from akismet import Akismet

from django.core.exceptions import ImproperlyConfigured
from django.contrib.sites.models import Site

from blog.models import Comment
```

```
@celery.task
def spam_filter(comment_id, remote_addr=None):
    logger = spam_filter.get_logger()
    logger.info('Running spam filter for comment %s' % comment_id)

    comment = Comment.objects.get(pk=comment_id)
    current_domain = Site.objects.get_current().domain
    akismet = Akismet(settings.AKISMET_KEY, 'http://%s' % domain)
    if not akismet.verify_key():
        raise ImproperlyConfigured('Invalid AKISMET_KEY')


    is_spam = akismet.comment_check(user_ip=remote_addr,
                        comment_content=comment.comment,
                        comment_author=comment.name,
                        comment_author_email=comment.email_address)
    if is_spam:
        comment.is_spam = True
        comment.save()

    return is_spam
```

2.3.3 Calling Tasks

Basics

This document describes Celery's uniform "Calling API" used by task instances and the *canvas*.

The API defines a standard set of execution options, as well as three methods:

- `apply_async(args[, kwargs[, ...]])`

  Sends a task message.

- `delay(*args, **kwargs)`

  Shortcut to send a task message, but does not support execution options.

- *calling* (`__call__`)

  Applying an object supporting the calling API (e.g. `add(2, 2)`) means that the task will be executed in the current process, and not by a worker (a message will not be sent).

Quick Cheat Sheet

- **`T.delay(arg, kwarg=value)`** always a shortcut to `.apply_async`.
- `T.apply_async((arg, ), {'kwarg': value})`
- **`T.apply_async(countdown=10)`** executes 10 seconds from now.
- **`T.apply_async(eta=now + timedelta(seconds=10))`** executes 10 seconds from now, specifed using `eta`
- **`T.apply_async(countdown=60, expires=120)`** executes in one minute from now, but expires after 2 minutes.
- **`T.apply_async(expires=now + timedelta(days=2))`** expires in 2 days, set using `datetime`.

Example

The `delay()` method is convenient as it looks like calling a regular function:

```
task.delay(arg1, arg2, kwarg1='x', kwarg2='y')
```

Using `apply_async()` instead you have to write:

```
task.apply_async(args=[arg1, arg2], kwargs={'kwarg1': 'x', 'kwarg2': 'y'})
```

Tip

If the task is not registered in the current process you can use `send_task()` to call the task by name instead.

So *delay* is clearly convenient, but if you want to set additional execution options you have to use `apply_async`.

The rest of this document will go into the task execution options in detail. All examples use a task called *add*, returning the sum of two arguments:

```
@celery.task
def add(x, y):
    return x + y
```

There's another way...

You will learn more about this later while reading about the *Canvas*, but `subtask`'s are objects used to pass around the signature of a task invocation, (for example to send it over the network), and they also support the Calling API:

```
task.s(arg1, arg2, kwarg1='x', kwargs2='y').apply_async()
```

Linking (callbacks/errbacks)

Celery supports linking tasks together so that one task follows another. The callback task will be applied with the result of the parent task as a partial argument:

```
add.apply_async((2, 2), link=add.s(16))
```

What is s?

The `add.s` call used here is called a subtask, I talk more about subtasks in the *canvas guide*, where you can also learn about `chain`, which is a simpler way to chain tasks together.
In practice the `link` execution option is considered an internal primitive, and you will probably not use it directly, but rather use chains instead.

Here the result of the first task (4) will be sent to a new task that adds 16 to the previous result, forming the expression $(2 + 2) + 16 = 20$

You can also cause a callback to be applied if task raises an exception (*errback*), but this behaves differently from a regular callback in that it will be passed the id of the parent task, not the result. This is because it may not always be possible to serialize the exception raised, and so this way the error callback requires a result backend to be enabled, and the task must retrieve the result of the task instead.

This is an example error callback:

```
@celery.task
def error_handler(uuid):
    result = AsyncResult(uuid)
    exc = result.get(propagate=False)
    print('Task %r raised exception: %r\n%r' % (
          exc, result.traceback))
```

it can be added to the task using the `link_error` execution option:

```
add.apply_async((2, 2), link_error=error_handler.s())
```

In addition, both the `link` and `link_error` options can be expressed as a list:

```
add.apply_async((2, 2), link=[add.s(16), other_task.s()])
```

The callbacks/errbacks will then be called in order, and all callbacks will be called with the return value of the parent task as a partial argument.

ETA and countdown

The ETA (estimated time of arrival) lets you set a specific date and time that is the earliest time at which your task will be executed. *countdown* is a shortcut to set eta by seconds into the future.

```
>>> result = add.apply_async((2, 2), countdown=3)
>>> result.get()    # this takes at least 3 seconds to return
20
```

The task is guaranteed to be executed at some time *after* the specified date and time, but not necessarily at that exact time. Possible reasons for broken deadlines may include many items waiting in the queue, or heavy network latency. To make sure your tasks are executed in a timely manner you should monitor the queue for congestion. Use Munin, or similar tools, to receive alerts, so appropriate action can be taken to ease the workload. See *Munin*.

While *countdown* is an integer, *eta* must be a `datetime` object, specifying an exact date and time (including millisecond precision, and timezone information):

```
>>> from datetime import datetime, timedelta

>>> tomorrow = datetime.utcnow() + timedelta(days=1)
>>> add.apply_async((2, 2), eta=tomorrow)
```

Expiration

The *expires* argument defines an optional expiry time, either as seconds after task publish, or a specific date and time using `datetime`:

```
>>> # Task expires after one minute from now.
>>> add.apply_async((10, 10), expires=60)

>>> # Also supports datetime
>>> from datetime import datetime, timedelta
>>> add.apply_async((10, 10), kwargs,
...                 expires=datetime.now() + timedelta(days=1)
```

When a worker receives an expired task it will mark the task as `REVOKED` (`TaskRevokedError`).

Message Sending Retry

Celery will automatically retry sending messages in the event of connection failure, and retry behavior can be configured – like how often to retry, or a maximum number of retries – or disabled all together.

To disable retry you can set the `retry` execution option to `False`:

```
add.apply_async((2, 2), retry=False)
```

Related Settings

- `CELERY_TASK_PUBLISH_RETRY`
- `CELERY_TASK_PUBLISH_RETRY_POLICY`

Retry Policy

A retry policy is a mapping that controls how retries behave, and can contain the following keys:

- *max_retries*

  Maximum number of retries before giving up, in this case the exception that caused the retry to fail will be raised.

  A value of 0 or `None` means it will retry forever.

  The default is to retry 3 times.

- *interval_start*

  Defines the number of seconds (float or integer) to wait between retries. Default is 0, which means the first retry will be instantaneous.

- *interval_step*

  On each consecutive retry this number will be added to the retry delay (float or integer). Default is 0.2.

- *interval_max*

  Maximum number of seconds (float or integer) to wait between retries. Default is 0.2.

For example, the default policy correlates to:

```
add.apply_async((2, 2), retry=True, retry_policy={
    'max_retries': 3,
    'interval_start': 0,
    'interval_step': 0.2,
    'interval_max': 0.2,
})
```

the maximum time spent retrying will be 0.4 seconds. It is set relatively short by default because a connection failure could lead to a retry pile effect if the broker connection is down: e.g. many web server processes waiting to retry blocking other incoming requests.

Serializers

Security

The pickle module allows for execution of arbitrary functions, please see the *security guide*.
Celery also comes with a special serializer that uses cryptography to sign your messages.

Data transferred between clients and workers needs to be serialized, so every message in Celery has a `content_type` header that describes the serialization method used to encode it.

The default serializer is `pickle`, but you can change this using the `CELERY_TASK_SERIALIZER` setting, or for each individual task, or even per message.

There's built-in support for `pickle`, *JSON*, *YAML* and *msgpack*, and you can also add your own custom serializers by registering them into the Kombu serializer registry (see ref:*kombu:guide-serialization*).

Each option has its advantages and disadvantages.

json – JSON is supported in many programming languages, is now a standard part of Python (since 2.6), and is fairly fast to decode using the modern Python libraries such as `cjson` or `simplejson`.

The primary disadvantage to JSON is that it limits you to the following data types: strings, Unicode, floats, boolean, dictionaries, and lists. Decimals and dates are notably missing.

Also, binary data will be transferred using Base64 encoding, which will cause the transferred data to be around 34% larger than an encoding which supports native binary types.

However, if your data fits inside the above constraints and you need cross-language support, the default setting of JSON is probably your best choice.

See http://json.org for more information.

pickle – If you have no desire to support any language other than Python, then using the pickle encoding will gain you the support of all built-in Python data types (except class instances), smaller messages when sending binary files, and a slight speedup over JSON processing.

See http://docs.python.org/library/pickle.html for more information.

yaml – YAML has many of the same characteristics as json, except that it natively supports more data types (including dates, recursive references, etc.)

However, the Python libraries for YAML are a good bit slower than the libraries for JSON.

If you need a more expressive set of data types and need to maintain cross-language compatibility, then YAML may be a better fit than the above.

See http://yaml.org/ for more information.

msgpack – msgpack is a binary serialization format that is closer to JSON in features. It is very young however, and support should be considered experimental at this point.

See http://msgpack.org/ for more information.

The encoding used is available as a message header, so the worker knows how to deserialize any task. If you use a custom serializer, this serializer must be available for the worker.

The following order is used to decide which serializer to use when sending a task:

1. The *serializer* execution option.
2. The `Task.serializer` attribute
3. The `CELERY_TASK_SERIALIZER` setting.

Example setting a custom serializer for a single task invocation:

```
>>> add.apply_async((10, 10), serializer='json')
```

Compression

Celery can compress the messages using either *gzip*, or *bzip2*. You can also create your own compression schemes and register them in the `kombu compression registry`.

The following order is used to decide which compression scheme to use when sending a task:

1. The *compression* execution option.
2. The `Task.compression` attribute.
3. The `CELERY_MESSAGE_COMPRESSION` attribute.

Example specifying the compression used when calling a task:

```
>>> add.apply_async((2, 2), compression='zlib')
```

Connections

Automatic Pool Support

Since version 2.3 there is support for automatic connection pools, so you don't have to manually handle connections and publishers to reuse connections.
The connection pool is enabled by default since version 2.5.
See the `BROKER_POOL_LIMIT` setting for more information.

You can handle the connection manually by creating a publisher:

```
results = []
with add.app.pool.acquire(block=True) as connection:
    with add.get_publisher(connection) as publisher:
        try:
            for args in numbers:
                res = add.apply_async((2, 2), publisher=publisher)
                results.append(res)
print([res.get() for res in results])
```

Though this particular example is much better expressed as a group:

```
>>> from celery import group

>>> numbers = [(2, 2), (4, 4), (8, 8), (16, 16)]
>>> res = group(add.subtask(n) for i in numbers).apply_async()

>>> res.get()
[4, 8, 16, 32]
```

Routing options

Celery can route tasks to different queues.

Simple routing (name <-> name) is accomplished using the `queue` option:

```
add.apply_async(queue='priority.high')
```

You can then assign workers to the `priority.high` queue by using the workers `-Q` argument:

```
$ celery worker -l info -Q celery,priority.high
```

See also:

Hard-coding queue names in code is not recommended, the best practice is to use configuration routers (`CELERY_ROUTES`).

To find out more about routing, please see *Routing Tasks*.

Advanced Options

These options are for advanced users who want to take use of AMQP's full routing capabilities. Interested parties may read the *routing guide*.

- exchange

  Name of exchange (or a `kombu.entity.Exchange`) to send the message to.

- routing_key

  Routing key used to determine.

- priority

  A number between *0* and *9*, where *0* is the highest priority.

  Supported by: redis, beanstalk

2.3.4 Canvas: Designing Workflows

Subtasks

New in version 2.0.

You just learned how to call a task using the tasks `delay` method in the *calling* guide, and this is often all you need, but sometimes you may want to pass the signature of a task invocation to another process or as an argument to another function, for this Celery uses something called *subtasks*.

A `subtask()` wraps the arguments, keyword arguments, and execution options of a single task invocation in a way such that it can be passed to functions or even serialized and sent across the wire.

- You can create a subtask for the `add` task using its name like this:

```
>>> from celery import subtask
>>> subtask('tasks.add', args=(2, 2), countdown=10)
tasks.add(2, 2)
```

  This subtask has a signature of arity 2 (two arguments): `(2, 2)`, and sets the countdown execution option to 10.

- or you can create one using the task's `subtask` method:

```
>>> add.subtask((2, 2), countdown=10)
tasks.add(2, 2)
```

- There is also a shortcut using star arguments:

```
>>> add.s(2, 2)
tasks.add(2, 2)
```

- Keyword arguments are also supported:

```
>>> add.s(2, 2, debug=True)
tasks.add(2, 2, debug=True)
```

- From any subtask instance you can inspect the different fields:

```
>>> s = add.subtask((2, 2), {'debug': True}, countdown=10)
>>> s.args
(2, 2)
>>> s.kwargs
{'debug': True}
>>> s.options
{'countdown': 10}
```

- It supports the "Calling API" which means it takes the same arguments as the `apply_async()` method:

```
>>> add.apply_async(args, kwargs, **options)
>>> add.subtask(args, kwargs, **options).apply_async()

>>> add.apply_async((2, 2), countdown=1)
>>> add.subtask((2, 2), countdown=1).apply_async()
```

- You can't define options with `s()`, but a chaining `set` call takes care of that:

```
>>> add.s(2, 2).set(countdown=1)
proj.tasks.add(2, 2)
```

Partials

A subtask can be applied too:

```
>>> add.s(2, 2).delay()
>>> add.s(2, 2).apply_async(countdown=1)
```

Specifying additional args, kwargs or options to `apply_async`/`delay` creates partials:

- Any arguments added will be prepended to the args in the signature:

```
>>> partial = add.s(2)           # incomplete signature
>>> partial.delay(4)             # 2 + 4
>>> partial.apply_async((4, ))   # same
```

- Any keyword arguments added will be merged with the kwargs in the signature, with the new keyword arguments taking precedence:

```
>>> s = add.s(2, 2)
>>> s.delay(debug=True)                    # -> add(2, 2, debug=True)
>>> s.apply_async(kwargs={'debug': True})  # same
```

- Any options added will be merged with the options in the signature, with the new options taking precedence:

```
>>> s = add.subtask((2, 2), countdown=10)
>>> s.apply_async(countdown=1)  # countdown is now 1
```

You can also clone subtasks to augment these:

```
>>> s = add.s(2)
proj.tasks.add(2)

>>> s.clone(args=(4, ), kwargs={'debug': True})
proj.tasks.add(2, 4, debug=True)
```

Immutability

New in version 3.0.

Partials are meant to be used with callbacks, any tasks linked or chord callbacks will be applied with the result of the parent task. Sometimes you want to specify a callback that does not take additional arguments, and in that case you can set the subtask to be immutable:

```
>>> add.apply_async((2, 2), link=reset_buffers.subtask(immutable=True))
```

The `.si()` shortcut can also be used to create immutable subtasks:

```
>>> add.apply_async((2, 2), link=reset_buffers.si())
```

Only the execution options can be set when a subtask is immutable, so it's not possible to call the subtask with partial args/kwargs.

Note: In this tutorial I sometimes use the prefix operator ~ to subtasks. You probably shouldn't use it in your production code, but it's a handy shortcut when experimenting in the Python shell:

```
>>> ~subtask

>>> # is the same as
>>> subtask.delay().get()
```

Callbacks

New in version 3.0.

Callbacks can be added to any task using the `link` argument to `apply_async`:

```
add.apply_async((2, 2), link=other_task.subtask())
```

The callback will only be applied if the task exited successfully, and it will be applied with the return value of the parent task as argument.

As I mentioned earlier, any arguments you add to *subtask*, will be prepended to the arguments specified by the subtask itself!

If you have the subtask:

```
>>> add.subtask(args=(10, ))
```

subtask.delay(result) becomes:

```
>>> add.apply_async(args=(result, 10))
```

...

Now let's call our `add` task with a callback using partial arguments:

```
>>> add.apply_async((2, 2), link=add.subtask((8, )))
```

As expected this will first launch one task calculating $2 + 2$, then another task calculating $4 + 8$.

The Primitives

New in version 3.0.

Overview

- `group`
  The group primitive is a subtask that takes a list of tasks that should be applied in parallel.
- `chain`
  The chain primitive lets us link together subtasks so that one is called after the other, essentially forming a *chain* of callbacks.
- `chord`
  A chord is just like a group but with a callback. A chord consists of a header group and a body, where the body is a task that should execute after all of the tasks in the header are complete.
- `map`
  The map primitive works like the built-in `map` function, but creates a temporary task where a list of arguments is applied to the task. E.g. `task.map([1, 2])` results in a single task being called, applying the arguments in order to the task function so that the result is:

  ```
  res = [task(1), task(2)]
  ```

- `starmap`
  Works exactly like map except the arguments are applied as `*args`. For example `add.starmap([(2, 2), (4, 4)])` results in a single task calling:

  ```
  res = [add(2, 2), add(4, 4)]
  ```

- `chunks`
  Chunking splits a long list of arguments into parts, e.g the operation:

  ```
  >>> add.chunks(zip(xrange(1000), xrange(1000), 10))
  ```

  will create 10 tasks that apply 100 items each.

The primitives are also subtasks themselves, so that they can be combined in any number of ways to compose complex workflows.

Here's some examples:

- Simple chain

  Here's a simple chain, the first task executes passing its return value to the next task in the chain, and so on.

  ```
  >>> from celery import chain

  # 2 + 2 + 4 + 8
  >>> res = chain(add.s(2, 2), add.s(4), add.s(8))()
  >>> res.get()
  16
  ```

  This can also be written using pipes:

  ```
  >>> (add.s(2, 2) | add.s(4) | add.s(8))().get()
  16
  ```

- Immutable subtasks

  Signatures can be partial so arguments can be added to the existing arguments, but you may not always want that, for example if you don't want the result of the previous task in a chain.

  In that case you can mark the subtask as immutable, so that the arguments cannot be changed:

  ```
  >>> add.subtask((2, 2), immutable=True)
  ```

  There's also an `.si` shortcut for this:

```
>>> add.si(2, 2)
```

Now you can create a chain of independent tasks instead:

```
>>> res = (add.si(2, 2), add.si(4, 4), add.s(8, 8))()
>>> res.get()
16

>>> res.parent.get()
8

>>> res.parent.parent.get()
4
```

- Simple group

  You can easily create a group of tasks to execute in parallel:

  ```
  >>> from celery import group
  >>> res = group(add.s(i, i) for i in xrange(10))()
  >>> res.get(timeout=1)
  [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
  ```

  - For primitives *.apply_async* is special...

    as it will create a temporary task to apply the tasks in, for example by *applying the group*:

    ```
    >>> g = group(add.s(i, i) for i in xrange(10))
    >>> g()  # << applying
    ```

    the act of sending the messages for the tasks in the group will happen in the current process, but with `.apply_async` this happens in a temporary task instead:

    ```
    >>> g = group(add.s(i, i) for i in xrange(10))
    >>> g.apply_async()
    ```

    This is useful because you can e.g. specify a time for the messages in the group to be called:

    ```
    >>> g.apply_async(countdown=10)
    ```

- Simple chord

  The chord primitive enables us to add callback to be called when all of the tasks in a group have finished executing, which is often required for algorithms that aren't embarrassingly parallel:

  ```
  >>> from celery import chord
  >>> res = chord((add.s(i, i) for i in xrange(10)), xsum.s())()
  >>> res.get()
  90
  ```

  The above example creates 10 task that all start in parallel, and when all of them are complete the return values are combined into a list and sent to the `xsum` task.

  The body of a chord can also be immutable, so that the return value of the group is not passed on to the callback:

  ```
  >>> chord((import_contact.s(c) for c in contacts),
  ...       notify_complete.si(import_id)).apply_async()
  ```

Note the use of `.si` above which creates an immutable subtask.

- Blow your mind by combining

  Chains can be partial too:

```
>>> c1 = (add.s(4) | mul.s(8))

# (16 + 4) * 8
>>> res = c1(16)
>>> res.get()
160
```

  Which means that you can combine chains:

```
# ((4 + 16) * 2 + 4) * 8
>>> c2 = (add.s(4, 16) | mul.s(2) | (add.s(4) | mul.s(8)))

>>> res = c2()
>>> res.get()
352
```

  Chaining a group together with another task will automatically upgrade it to be a chord:

```
>>> c3 = (group(add.s(i, i) for i in xrange(10)) | xsum.s())
>>> res = c3()
>>> res.get()
90
```

  Groups and chords accepts partial arguments too, so in a chain the return value of the previous task is forwarded to all tasks in the group:

```
>>> new_user_workflow = (create_user.s() | group(
...                      import_contacts.s(),
...                      send_welcome_email.s()))
... new_user_workflow.delay(username='artv',
...                         first='Art',
...                         last='Vandelay',
...                         email='art@vandelay.com')
```

  If you don't want to forward arguments to the group then you can make the subtasks in the group immutable:

```
>>> res = (add.s(4, 4) | group(add.si(i, i) for i in xrange(10)))()
>>> res.get()
<GroupResult: de44df8c-821d-4c84-9a6a-44769c738f98 [
    bc01831b-9486-4e51-b046-480d7c9b78de,
    2650a1b8-32bf-4771-a645-b0a35dcc791b,
    dcbee2a5-e92d-4b03-b6eb-7aec60fd30cf,
    59f92e0a-23ea-41ce-9fad-8645a0e7759c,
    26e1e707-eccf-4bf4-bbd8-1e1729c3cce3,
    2d10a5f4-37f0-41b2-96ac-a973b1df024d,
    e13d3bdb-7ae3-4101-81a4-6f17ee21df2d,
    104b2be0-7b75-44eb-ac8e-f9220bdfa140,
    c5c551a5-0386-4973-aa37-b65cbeb2624b,
    83f72d71-4b71-428e-b604-6f16599a9f37]>

>>> res.parent.get()
8
```

Chains

New in version 3.0.

Tasks can be linked together, which in practice means adding a callback task:

```
>>> res = add.apply_async((2, 2), link=mul.s(16))
>>> res.get()
4
```

The linked task will be applied with the result of its parent task as the first argument, which in the above case will result in `mul(4, 16)` since the result is 4.

The results will keep track of what subtasks a task applies, and this can be accessed from the result instance:

```
>>> res.children
[<AsyncResult: 8c350acf-519d-4553-8a53-4ad3a5c5aeb4>]

>>> res.children[0].get()
64
```

The result instance also has a `collect()` method that treats the result as a graph, enabling you to iterate over the results:

```
>>> list(res.collect())
[(<AsyncResult: 7b720856-dc5f-4415-9134-5c89def5664e>, 4),
 (<AsyncResult: 8c350acf-519d-4553-8a53-4ad3a5c5aeb4>, 64)]
```

By default `collect()` will raise an `IncompleteStream` exception if the graph is not fully formed (one of the tasks has not completed yet), but you can get an intermediate representation of the graph too:

```
>>> for result, value in res.collect(intermediate=True)):
....
```

You can link together as many tasks as you like, and subtasks can be linked too:

```
>>> s = add.s(2, 2)
>>> s.link(mul.s(4))
>>> s.link(log_result.s())
```

You can also add *error callbacks* using the `link_error` argument:

```
>>> add.apply_async((2, 2), link_error=log_error.s())

>>> add.subtask((2, 2), link_error=log_error.s())
```

Since exceptions can only be serialized when pickle is used the error callbacks take the id of the parent task as argument instead:

```
from proj.celery import celery

@celery.task
def log_error(task_id):
    result = celery.AsyncResult(task_id)
    result.get(propagate=False)  # make sure result written.
    with open(os.path.join('/var/errors', task_id), 'a') as fh:
        fh.write('--\n\n%s %s %s' % (
            task_id, result.result, result.traceback))
```

To make it even easier to link tasks together there is a special subtask called `chain` that lets you chain tasks together:

```
>>> from celery import chain
>>> from proj.tasks import add, mul

# (4 + 4) * 8 * 10
>>> res = chain(add.s(4, 4), mul.s(8), mul.s(10))
proj.tasks.add(4, 4) | proj.tasks.mul(8) | proj.tasks.mul(10)
```

Calling the chain will call the tasks in the current process and return the result of the last task in the chain:

```
>>> res = chain(add.s(4, 4), mul.s(8), mul.s(10))()
>>> res.get()
640
```

And calling `apply_async` will create a dedicated task so that the act of calling the chain happens in a worker:

```
>>> res = chain(add.s(4, 4), mul.s(8), mul.s(10)).apply_async()
>>> res.get()
640
```

It also sets `parent` attributes so that you can work your way up the chain to get intermediate results:

```
>>> res.parent.get()
64

>>> res.parent.parent.get()
8

>>> res.parent.parent
<AsyncResult: eeaad925-6778-4ad1-88c8-b2a63d017933>
```

Chains can also be made using the `|` (pipe) operator:

```
>>> (add.s(2, 2) | mul.s(8) | mul.s(10)).apply_async()
```

Graphs In addition you can work with the result graph as a `DependencyGraph`:

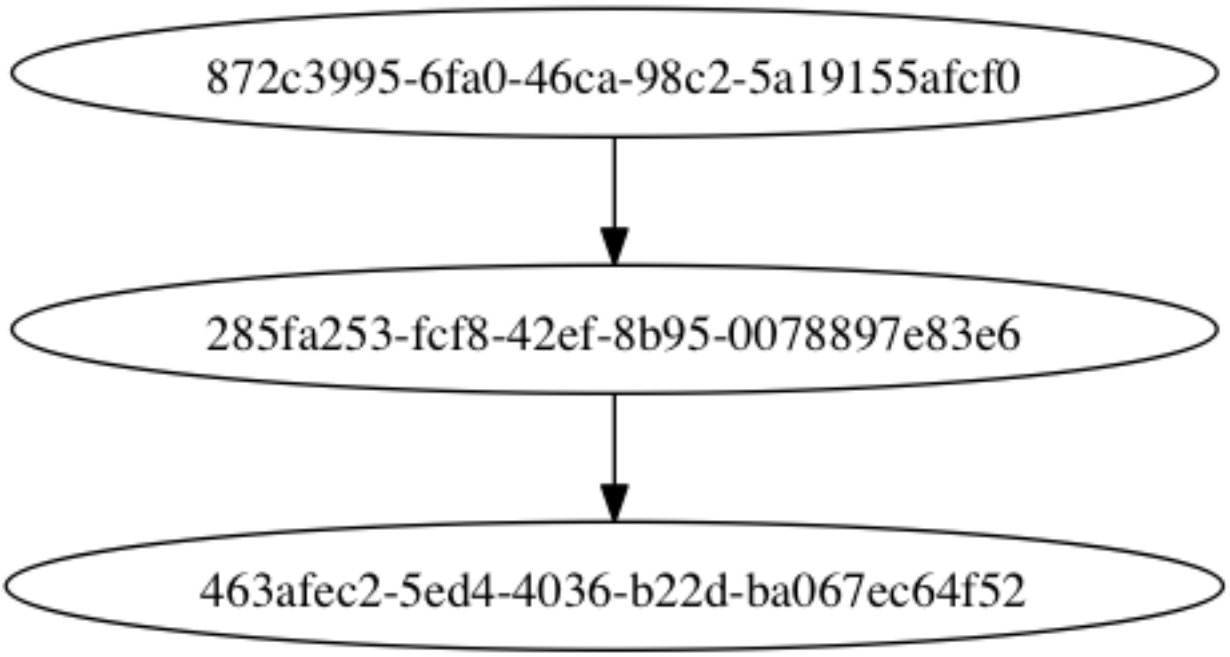
```
>>> res = chain(add.s(4, 4), mul.s(8), mul.s(10))()

>>> res.parent.parent.graph
285fa253-fcf8-42ef-8b95-0078897e83e6(1)
    463afec2-5ed4-4036-b22d-ba067ec64f52(0)
872c3995-6fa0-46ca-98c2-5a19155afcf0(2)
    285fa253-fcf8-42ef-8b95-0078897e83e6(1)
        463afec2-5ed4-4036-b22d-ba067ec64f52(0)
```

You can even convert these graphs to *dot* format:

```
>>> with open('graph.dot', 'w') as fh:
...     res.parent.parent.graph.to_dot(fh)
```

and create images:

```
$ dot -Tpng graph.dot -o graph.png
```



Groups

New in version 3.0.

A group can be used to execute several tasks in parallel.

The `group` function takes a list of subtasks:

```
>>> from celery import group
>>> from proj.tasks import add

>>> group(add.s(2, 2), add.s(4, 4))
(proj.tasks.add(2, 2), proj.tasks.add(4, 4))
```

If you **call** the group, the tasks will be applied one after one in the current process, and a `TaskSetResult` instance is returned which can be used to keep track of the results, or tell how many tasks are ready and so on:

```
>>> g = group(add.s(2, 2), add.s(4, 4))
>>> res = g()
>>> res.get()
[4, 8]
```

However, if you call `apply_async` on the group it will send a special grouping task, so that the action of calling the tasks happens in a worker instead of the current process:

```
>>> res = g.apply_async()
>>> res.get()
[4, 8]
```

Group also supports iterators:

```
>>> group(add.s(i, i) for i in xrange(100))()
```

A group is a subtask instance, so it can be used in combination with other subtasks.

Group Results The group task returns a special result too, this result works just like normal task results, except that it works on the group as a whole:

```
>>> from celery import group
>>> from tasks import add

>>> job = group([
...              add.subtask((2, 2)),
...              add.subtask((4, 4)),
...              add.subtask((8, 8)),
...              add.subtask((16, 16)),
...              add.subtask((32, 32)),
... ])

>>> result = job.apply_async()

>>> result.ready()  # have all subtasks completed?
True
>>> result.successful() # were all subtasks successful?
True
>>> result.join()
[4, 8, 16, 32, 64]
```

The `GroupResult` takes a list of `AsyncResult` instances and operates on them as if it was a single task.

It supports the following operations:

- `successful()`

  Returns `True` if all of the subtasks finished successfully (e.g. did not raise an exception).

- `failed()`

  Returns `True` if any of the subtasks failed.

- `waiting()`

  Returns `True` if any of the subtasks is not ready yet.

- `ready()`

  Return `True` if all of the subtasks are ready.

- `completed_count()`

  Returns the number of completed subtasks.

- `revoke()`

  Revokes all of the subtasks.

- `iterate()`

  Iterates over the return values of the subtasks as they finish, one by one.

- `join()`

  Gather the results for all of the subtasks and return a list with them ordered by the order of which they were called.

Chords

New in version 2.3.

A chord is a task that only executes after all of the tasks in a taskset have finished executing.

Let's calculate the sum of the expression $1 + 1 + 2 + 2 + 3 + 3...n + n$ up to a hundred digits.

First you need two tasks, `add()` and `tsum()` (`sum()` is already a standard function):

```
@celery.task
def add(x, y):
    return x + y

@celery.task
def tsum(numbers):
    return sum(numbers)
```

Now you can use a chord to calculate each addition step in parallel, and then get the sum of the resulting numbers:

```
>>> from celery import chord
>>> from tasks import add, tsum

>>> chord(add.s(i, i)
...       for i in xrange(100))(tsum.s()).get()
9900
```

This is obviously a very contrived example, the overhead of messaging and synchronization makes this a lot slower than its Python counterpart:

```
sum(i + i for i in xrange(100))
```

The synchronization step is costly, so you should avoid using chords as much as possible. Still, the chord is a powerful primitive to have in your toolbox as synchronization is a required step for many parallel algorithms.

Let's break the chord expression down:

```
>>> callback = tsum.subtask()
>>> header = [add.subtask((i, i)) for i in xrange(100)]
>>> result = chord(header)(callback)
>>> result.get()
9900
```

Remember, the callback can only be executed after all of the tasks in the header have returned. Each step in the header is executed as a task, in parallel, possibly on different nodes. The callback is then applied with the return value of each task in the header. The task id returned by `chord()` is the id of the callback, so you can wait for it to complete and get the final return value (but remember to *never have a task wait for other tasks*)

Error handling So what happens if one of the tasks raises an exception?

This was not documented for some time and before version 3.1 the exception value will be forwarded to the chord callback.

From 3.1 errors will propagate to the callback, so the callback will not be executed instead the callback changes to failure state, and the error is set to the `ChordError` exception:

```
>>> c = chord([add.s(4, 4), raising_task.s(), add.s(8, 8)])
>>> result = c()
>>> result.get()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "*/celery/result.py", line 120, in get
    interval=interval)
  File "*/celery/backends/amqp.py", line 150, in wait_for
    raise self.exception_to_python(meta['result'])
```

```
celery.exceptions.ChordError: Dependency 97de6f3f-ea67-4517-a21c-d867c61fcb47
    raised ValueError('something something',)
```

If you're running 3.0.14 or later you can enable the new behavior via the CELERY_CHORD_PROPAGATES setting:

```
CELERY_CHORD_PROPAGATES = True
```

While the traceback may be different depending on which result backend is being used, you can see the error description includes the id of the task that failed and a string representation of the original exception. You can also find the original traceback in `result.traceback`.

Note that the rest of the tasks will still execute, so the third task (`add.s(8, 8)`) is still executed even though the middle task failed. Also the ChordError only shows the task that failed first (in time): it does not respect the ordering of the header group.

Important Notes Tasks used within a chord must *not* ignore their results. In practice this means that you must enable a CELERY_RESULT_BACKEND in order to use chords. Additionally, if CELERY_IGNORE_RESULT is set to `True` in your configuration, be sure that the individual tasks to be used within the chord are defined with `ignore_result=False`. This applies to both Task subclasses and decorated tasks.

Example Task subclass:

```
class MyTask(Task):
    abstract = True
    ignore_result = False
```

Example decorated task:

```
@celery.task(ignore_result=False)
def another_task(project):
    do_something()
```

By default the synchronization step is implemented by having a recurring task poll the completion of the taskset every second, calling the subtask when ready.

Example implementation:

```
def unlock_chord(taskset, callback, interval=1, max_retries=None):
    if taskset.ready():
        return subtask(callback).delay(taskset.join())
    raise unlock_chord.retry(countdown=interval, max_retries=max_retries)
```

This is used by all result backends except Redis and Memcached, which increment a counter after each task in the header, then applying the callback when the counter exceeds the number of tasks in the set. *Note:* chords do not properly work with Redis before version 2.2; you will need to upgrade to at least 2.2 to use them.

The Redis and Memcached approach is a much better solution, but not easily implemented in other backends (suggestions welcome!).

Note: If you are using chords with the Redis result backend and also overriding the `Task.after_return()` method, you need to make sure to call the super method or else the chord callback will not be applied.

```
def after_return(self, *args, **kwargs):
    do_something()
    super(MyTask, self).after_return(*args, **kwargs)
```

Map & Starmap

`map` and `starmap` are built-in tasks that calls the task for every element in a sequence.

They differ from group in that

- only one task message is sent
- the operation is sequential.

For example using `map`:

```
>>> from proj.tasks import add

>>> ~xsum.map([range(10), range(100)])
[45, 4950]
```

is the same as having a task doing:

```
@celery.task
def temp():
    return [xsum(range(10)), xsum(range(100))]
```

and using `starmap`:

```
>>> ~add.starmap(zip(range(10), range(10)))
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

is the same as having a task doing:

```
@celery.task
def temp():
    return [add(i, i) for i in range(10)]
```

Both `map` and `starmap` are subtasks, so they can be used as other subtasks and combined in groups etc., for example to call the starmap after 10 seconds:

```
>>> add.starmap(zip(range(10), range(10))).apply_async(countdown=10)
```

Chunks

Chunking lets you divide an iterable of work into pieces, so that if you have one million objects, you can create 10 tasks with hundred thousand objects each.

Some may worry that chunking your tasks results in a degradation of parallelism, but this is rarely true for a busy cluster and in practice since you are avoiding the overhead of messaging it may considerably increase performance.

To create a chunks subtask you can use `celery.Task.chunks()`:

```
>>> add.chunks(zip(range(100), range(100)), 10)
```

As with `group` the act of **calling** the chunks will call the tasks in the current process:

```
>>> from proj.tasks import add

>>> res = add.chunks(zip(range(100), range(100)), 10)()
>>> res.get()
[[0, 2, 4, 6, 8, 10, 12, 14, 16, 18],
 [20, 22, 24, 26, 28, 30, 32, 34, 36, 38],
 [40, 42, 44, 46, 48, 50, 52, 54, 56, 58],
```

```
 [60, 62, 64, 66, 68, 70, 72, 74, 76, 78],
 [80, 82, 84, 86, 88, 90, 92, 94, 96, 98],
 [100, 102, 104, 106, 108, 110, 112, 114, 116, 118],
 [120, 122, 124, 126, 128, 130, 132, 134, 136, 138],
 [140, 142, 144, 146, 148, 150, 152, 154, 156, 158],
 [160, 162, 164, 166, 168, 170, 172, 174, 176, 178],
 [180, 182, 184, 186, 188, 190, 192, 194, 196, 198]]
```

while calling `.apply_async` will create a dedicated task so that the individual tasks are applied in a worker instead:

```
>>> add.chunks(zip(range(100), range(100), 10)).apply_async()
```

You can also convert chunks to a group:

```
>>> group = add.chunks(zip(range(100), range(100), 10)).group()
```

and with the group skew the countdown of each task by increments of one:

```
>>> group.skew(start=1, stop=10)()
```

which means that the first task will have a countdown of 1, the second a countdown of 2 and so on.

2.3.5 Workers Guide

Starting the worker

Daemonizing

You probably want to use a daemonization tool to start in the background. See *Running the worker as a daemon* for help detaching the worker using popular daemonization tools.

You can start the worker in the foreground by executing the command:

```
$ celery worker --app=app -l info
```

For a full list of available command line options see celeryd, or simply do:

```
$ celery worker --help
```

You can also start multiple workers on the same machine. If you do so be sure to give a unique name to each individual worker by specifying a host name with the *--hostname|-n* argument:

```
$ celery worker --loglevel=INFO --concurrency=10 -n worker1.example.com
$ celery worker --loglevel=INFO --concurrency=10 -n worker2.example.com
$ celery worker --loglevel=INFO --concurrency=10 -n worker3.example.com
```

Stopping the worker

Shutdown should be accomplished using the TERM signal.

When shutdown is initiated the worker will finish all currently executing tasks before it actually terminates, so if these tasks are important you should wait for it to finish before doing anything drastic (like sending the KILL signal).

If the worker won't shutdown after considerate time, for example because of tasks stuck in an infinite-loop, you can use the KILL signal to force terminate the worker, but be aware that currently executing tasks will be lost (unless the tasks have the acks_late option set).

Also as processes can't override the KILL signal, the worker will not be able to reap its children, so make sure to do so manually. This command usually does the trick:

```
$ ps auxww | grep 'celery worker' | awk '{print $2}' | xargs kill -9
```

Restarting the worker

Other than stopping then starting the worker to restart, you can also restart the worker using the HUP signal:

```
$ kill -HUP $pid
```

The worker will then replace itself with a new instance using the same arguments as it was started with.

Note: Restarting by HUP only works if the worker is running in the background as a daemon (it does not have a controlling terminal).

HUP is disabled on OS X because of a limitation on that platform.

Process Signals

The worker's main process overrides the following signals:

TERM	Warm shutdown, wait for tasks to complete.
QUIT	Cold shutdown, terminate ASAP
USR1	Dump traceback for all active threads.
USR2	Remote debug, see celery.contrib.rdb.

Concurrency

By default multiprocessing is used to perform concurrent execution of tasks, but you can also use *Eventlet*. The number of worker processes/threads can be changed using the *--concurrency* argument and defaults to the number of CPUs available on the machine.

Number of processes (multiprocessing)

More pool processes are usually better, but there's a cut-off point where adding more pool processes affects performance in negative ways. There is even some evidence to support that having multiple worker instances running, may perform better than having a single worker. For example 3 workers with 10 pool processes each. You need to experiment to find the numbers that works best for you, as this varies based on application, work load, task run times and other factors.

Remote control

New in version 2.0.

The `celery` command

The **celery** program is used to execute remote control commands from the command line. It supports all of the commands listed below. See *celery: Command Line Management Utility* for more information.

pool support: *processes, eventlet, gevent*, blocking:*threads/solo* (see note) broker support: *amqp, redis, mongodb*

Workers have the ability to be remote controlled using a high-priority broadcast message queue. The commands can be directed to all, or a specific list of workers.

Commands can also have replies. The client can then wait for and collect those replies. Since there's no central authority to know how many workers are available in the cluster, there is also no way to estimate how many workers may send a reply, so the client has a configurable timeout — the deadline in seconds for replies to arrive in. This timeout defaults to one second. If the worker doesn't reply within the deadline it doesn't necessarily mean the worker didn't reply, or worse is dead, but may simply be caused by network latency or the worker being slow at processing commands, so adjust the timeout accordingly.

In addition to timeouts, the client can specify the maximum number of replies to wait for. If a destination is specified, this limit is set to the number of destination hosts.

Note: The solo and threads pool supports remote control commands, but any task executing will block any waiting control command, so it is of limited use if the worker is very busy. In that case you must increase the timeout waiting for replies in the client.

The `broadcast()` function.

This is the client function used to send commands to the workers. Some remote control commands also have higher-level interfaces using `broadcast()` in the background, like `rate_limit()` and `ping()`.

Sending the `rate_limit` command and keyword arguments:

```
>>> celery.control.broadcast('rate_limit',
...                          arguments={'task_name': 'myapp.mytask',
...                                     'rate_limit': '200/m'})
```

This will send the command asynchronously, without waiting for a reply. To request a reply you have to use the *reply* argument:

```
>>> celery.control.broadcast('rate_limit', {
...     'task_name': 'myapp.mytask', 'rate_limit': '200/m'}, reply=True)
[{'worker1.example.com': 'New rate limit set successfully'},
```

```
 {'worker2.example.com': 'New rate limit set successfully'},
 {'worker3.example.com': 'New rate limit set successfully'}]
```

Using the *destination* argument you can specify a list of workers to receive the command:

```
>>> celery.control.broadcast('rate_limit', {
...     'task_name': 'myapp.mytask',
...     'rate_limit': '200/m'}, reply=True,
...                             destination=['worker1.example.com'])
[{'worker1.example.com': 'New rate limit set successfully'}]
```

Of course, using the higher-level interface to set rate limits is much more convenient, but there are commands that can only be requested using `broadcast()`.

Revoking tasks

pool support: all broker support: *amqp, redis, mongodb*

All worker nodes keeps a memory of revoked task ids, either in-memory or persistent on disk (see *Persistent revokes*).

When a worker receives a revoke request it will skip executing the task, but it won't terminate an already executing task unless the *terminate* option is set.

If *terminate* is set the worker child process processing the task will be terminated. The default signal sent is *TERM*, but you can specify this using the *signal* argument. Signal can be the uppercase name of any signal defined in the `signal` module in the Python Standard Library.

Terminating a task also revokes it.

Example

```
>>> celery.control.revoke('d9078da5-9915-40a0-bfa1-392c7bde42ed')

>>> celery.control.revoke('d9078da5-9915-40a0-bfa1-392c7bde42ed',
...                       terminate=True)

>>> celery.control.revoke('d9078da5-9915-40a0-bfa1-392c7bde42ed',
...                       terminate=True, signal='SIGKILL')
```

Persistent revokes

Revoking tasks works by sending a broadcast message to all the workers, the workers then keep a list of revoked tasks in memory.

If you want tasks to remain revoked after worker restart you need to specify a file for these to be stored in, either by using the *–statedb* argument to **celery worker** or the `CELERYD_STATE_DB` setting.

Note that remote control commands must be working for revokes to work. Remote control commands are only supported by the RabbitMQ (amqp), Redis and MongDB transports at this point.

Time Limits

New in version 2.0.

pool support: *processes*

Soft, or hard?

The time limit is set in two values, *soft* and *hard*. The soft time limit allows the task to catch an exception to clean up before it is killed: the hard timeout is not catchable and force terminates the task.

A single task can potentially run forever, if you have lots of tasks waiting for some event that will never happen you will block the worker from processing new tasks indefinitely. The best way to defend against this scenario happening is enabling time limits.

The time limit (*–time-limit*) is the maximum number of seconds a task may run before the process executing it is terminated and replaced by a new process. You can also enable a soft time limit (*–soft-time-limit*), this raises an exception the task can catch to clean up before the hard time limit kills it:

```
from myapp import celery
from celery.exceptions import SoftTimeLimitExceeded

@celery.task
def mytask():
    try:
        do_work()
    except SoftTimeLimitExceeded:
        clean_up_in_a_hurry()
```

Time limits can also be set using the `CELERYD_TASK_TIME_LIMIT` / `CELERYD_TASK_SOFT_TIME_LIMIT` settings.

Note: Time limits do not currently work on Windows and other platforms that do not support the `SIGUSR1` signal.

Changing time limits at runtime

New in version 2.3.

broker support: *amqp, redis, mongodb*

There is a remote control command that enables you to change both soft and hard time limits for a task — named `time_limit`.

Example changing the time limit for the `tasks.crawl_the_web` task to have a soft time limit of one minute, and a hard time limit of two minutes:

```
>>> celery.control.time_limit('tasks.crawl_the_web',
                              soft=60, hard=120, reply=True)
[{'worker1.example.com': {'ok': 'time limits set successfully'}}]
```

Only tasks that starts executing after the time limit change will be affected.

Rate Limits

Changing rate-limits at runtime

Example changing the rate limit for the *myapp.mytask* task to accept 200 tasks a minute on all servers:

```
>>> celery.control.rate_limit('myapp.mytask', '200/m')
```

Example changing the rate limit on a single host by specifying the destination host name:

```
>>> celery.control.rate_limit('myapp.mytask', '200/m',
...                           destination=['worker1.example.com'])
```

Warning: This won't affect workers with the CELERY_DISABLE_RATE_LIMITS setting enabled.

Max tasks per child setting

New in version 2.0.

pool support: *processes*

With this option you can configure the maximum number of tasks a worker can execute before it's replaced by a new process.

This is useful if you have memory leaks you have no control over for example from closed source C extensions.

The option can be set using the workers *–maxtasksperchild* argument or using the CELERYD_MAX_TASKS_PER_CHILD setting.

Autoscaling

New in version 2.2.

pool support: *processes*, *gevent*

The *autoscaler* component is used to dynamically resize the pool based on load:

- **The autoscaler adds more pool processes when there is work to do,**
  - and starts removing processes when the workload is low.

It's enabled by the `--autoscale` option, which needs two numbers: the maximum and minimum number of pool processes:

```
--autoscale=AUTOSCALE
     Enable autoscaling by providing
     max_concurrency,min_concurrency.  Example:
       --autoscale=10,3 (always keep 3 processes, but grow to
      10 if necessary).
```

You can also define your own rules for the autoscaler by subclassing `Autoscaler`. Some ideas for metrics include load average or the amount of memory available. You can specify a custom autoscaler with the CELERYD_AUTOSCALER setting.

Queues

A worker instance can consume from any number of queues. By default it will consume from all queues defined in the CELERY_QUEUES setting (which if not specified defaults to the queue named `celery`).

You can specify what queues to consume from at startup, by giving a comma separated list of queues to the `-Q` option:

```
$ celery worker -l info -Q foo,bar,baz
```

If the queue name is defined in CELERY_QUEUES it will use that configuration, but if it's not defined in the list of queues Celery will automatically generate a new queue for you (depending on the CELERY_CREATE_MISSING_QUEUES option).

You can also tell the worker to start and stop consuming from a queue at runtime using the remote control commands add_consumer and cancel_consumer.

Queues: Adding consumers

The add_consumer control command will tell one or more workers to start consuming from a queue. This operation is idempotent.

To tell all workers in the cluster to start consuming from a queue named "foo" you can use the **celery control** program:

```
$ celery control add_consumer foo
-> worker1.local: OK
    started consuming from u'foo'
```

If you want to specify a specific worker you can use the *--destination* ' argument:

```
$ celery control add_consumer foo -d worker1.local
```

The same can be accomplished dynamically using the celery.control.add_consumer() method:

```
>>> myapp.control.add_consumer('foo', reply=True)
[{u'worker1.local': {u'ok': u"already consuming from u'foo'"}}]

>>> myapp.control.add_consumer('foo', reply=True,
...                            destination=['worker1.local'])
[{u'worker1.local': {u'ok': u"already consuming from u'foo'"}}]
```

By now I have only shown examples using automatic queues, If you need more control you can also specify the exchange, routing_key and even other options:

```
>>> myapp.control.add_consumer(
...     queue='baz',
...     exchange='ex',
...     exchange_type='topic',
...     routing_key='media.*',
...     options={
...         'queue_durable': False,
...         'exchange_durable': False,
...     },
...     reply=True,
...     destination=['worker1.local', 'worker2.local'])
```

Queues: Cancelling consumers

You can cancel a consumer by queue name using the cancel_consumer control command.

To force all workers in the cluster to cancel consuming from a queue you can use the **celery control** program:

```
$ celery control cancel_consumer foo
```

The *--destination* argument can be used to specify a worker, or a list of workers, to act on the command:

```
$ celery control cancel_consumer foo -d worker1.local
```

You can also cancel consumers programmatically using the `celery.control.cancel_consumer()` method:

```
>>> myapp.control.cancel_consumer('foo', reply=True)
[{u'worker1.local': {u'ok': u"no longer consuming from u'foo'"}}]
```

Queues: List of active queues

You can get a list of queues that a worker consumes from by using the `active_queues` control command:

```
$ celery inspect active_queues
[...]
```

Like all other remote control commands this also supports the *--destination* argument used to specify which workers should reply to the request:

```
$ celery inspect active_queues -d worker1.local
[...]
```

This can also be done programmatically by using the `celery.control.inspect.active_queues()` method:

```
>>> myapp.inspect().active_queues()
[...]

>>> myapp.inspect(['worker1.local']).active_queues()
[...]
```

Autoreloading

New in version 2.5.

pool support: *processes, eventlet, gevent, threads, solo*

Starting **celery worker** with the *--autoreload* option will enable the worker to watch for file system changes to all imported task modules imported (and also any non-task modules added to the `CELERY_IMPORTS` setting or the *-I|--include* option).

This is an experimental feature intended for use in development only, using auto-reload in production is discouraged as the behavior of reloading a module in Python is undefined, and may cause hard to diagnose bugs and crashes. Celery uses the same approach as the auto-reloader found in e.g. the Django `runserver` command.

When auto-reload is enabled the worker starts an additional thread that watches for changes in the file system. New modules are imported, and already imported modules are reloaded whenever a change is detected, and if the processes pool is used the child processes will finish the work they are doing and exit, so that they can be replaced by fresh processes effectively reloading the code.

File system notification backends are pluggable, and it comes with three implementations:

- inotify (Linux)

  Used if the `pyinotify` library is installed. If you are running on Linux this is the recommended implementation, to install the `pyinotify` library you have to run the following command:

  ```
  $ pip install pyinotify
  ```

- kqueue (OS X/BSD)

- stat

  The fallback implementation simply polls the files using `stat` and is very expensive.

You can force an implementation by setting the `CELERYD_FSNOTIFY` environment variable:

```
$ env CELERYD_FSNOTIFY=stat celery worker -l info --autoreload
```

Pool Restart Command

New in version 2.5.

Requires the `CELERYD_POOL_RESTARTS` setting to be enabled.

The remote control command `pool_restart` sends restart requests to the workers child processes. It is particularly useful for forcing the worker to import new modules, or for reloading already imported modules. This command does not interrupt executing tasks.

Example Running the following command will result in the *foo* and *bar* modules being imported by the worker processes:

```
>>> celery.control.broadcast('pool_restart',
...                          arguments={'modules': ['foo', 'bar']})
```

Use the `reload` argument to reload modules it has already imported:

```
>>> celery.control.broadcast('pool_restart',
...                          arguments={'modules': ['foo'],
...                                     'reload': True})
```

If you don't specify any modules then all known tasks modules will be imported/reloaded:

```
>>> celery.control.broadcast('pool_restart', arguments={'reload': True})
```

The `modules` argument is a list of modules to modify. `reload` specifies whether to reload modules if they have previously been imported. By default `reload` is disabled. The *pool_restart* command uses the Python `reload()` function to reload modules, or you can provide your own custom reloader by passing the `reloader` argument.

Note: Module reloading comes with caveats that are documented in `reload()`. Please read this documentation and make sure your modules are suitable for reloading.

See also:

- http://pyunit.sourceforge.net/notes/reloading.html
- http://www.indelible.org/ink/python-reloading/
- http://docs.python.org/library/functions.html#reload

Inspecting workers

`celery.control.inspect` lets you inspect running workers. It uses remote control commands under the hood.

You can also use the `celery` command to inspect workers, and it supports the same commands as the `Celery.control` interface.

```
# Inspect all nodes.
>>> i = celery.control.inspect()

# Specify multiple nodes to inspect.
>>> i = celery.control.inspect(['worker1.example.com',
                               'worker2.example.com'])

# Specify a single node to inspect.
>>> i = celery.control.inspect('worker1.example.com')
```

Dump of registered tasks

You can get a list of tasks registered in the worker using the `registered()`:

```
>>> i.registered()
[{'worker1.example.com': ['tasks.add',
                          'tasks.sleeptask']}]
```

Dump of currently executing tasks

You can get a list of active tasks using `active()`:

```
>>> i.active()
[{'worker1.example.com':
    [{'name': 'tasks.sleeptask',
      'id': '32666e9b-809c-41fa-8e93-5ae0c80afbbf',
      'args': '(8,)',
      'kwargs': '{}'}]}]
```

Dump of scheduled (ETA) tasks

You can get a list of tasks waiting to be scheduled by using `scheduled()`:

```
>>> i.scheduled()
[{'worker1.example.com':
    [{'eta': '2010-06-07 09:07:52', 'priority': 0,
      'request': {
        'name': 'tasks.sleeptask',
        'id': '1a7980ea-8b19-413e-91d2-0b74f3844c4d',
        'args': '[1]',
        'kwargs': '{}'}},
     {'eta': '2010-06-07 09:07:53', 'priority': 0,
      'request': {
        'name': 'tasks.sleeptask',
        'id': '49661b9a-aa22-4120-94b7-9ee8031d219d',
        'args': '[2]',
        'kwargs': '{}'}}]}]
```

Note: These are tasks with an eta/countdown argument, not periodic tasks.

Dump of reserved tasks

Reserved tasks are tasks that has been received, but is still waiting to be executed.

You can get a list of these using `reserved()`:

```
>>> i.reserved()
[{'worker1.example.com':
    [{'name': 'tasks.sleeptask',
      'id': '32666e9b-809c-41fa-8e93-5ae0c80afbbf',
      'args': '(8,)',
      'kwargs': '{}'}]}]
```

Additional Commands

Remote shutdown

This command will gracefully shut down the worker remotely:

```
>>> celery.control.broadcast('shutdown') # shutdown all workers
>>> celery.control.broadcast('shutdown, destination='worker1.example.com')
```

Ping

This command requests a ping from alive workers. The workers reply with the string 'pong', and that's just about it. It will use the default one second timeout for replies unless you specify a custom timeout:

```
>>> celery.control.ping(timeout=0.5)
[{'worker1.example.com': 'pong'},
 {'worker2.example.com': 'pong'},
 {'worker3.example.com': 'pong'}]
```

`ping()` also supports the *destination* argument, so you can specify which workers to ping:

```
>>> ping(['worker2.example.com', 'worker3.example.com'])
[{'worker2.example.com': 'pong'},
 {'worker3.example.com': 'pong'}]
```

Enable/disable events

You can enable/disable events by using the *enable_events*, *disable_events* commands. This is useful to temporarily monitor a worker using **celery events/celerymon**.

```
>>> celery.control.enable_events()
>>> celery.control.disable_events()
```

Writing your own remote control commands

Remote control commands are registered in the control panel and they take a single argument: the current `ControlDispatch` instance. From there you have access to the active `Consumer` if needed.

Here's an example control command that restarts the broker connection:

```
from celery.worker.control import Panel

@Panel.register
def reset_connection(panel):
    panel.consumer.reset_connection()
    return {'ok': 'connection reset'}
```

2.3.6 Periodic Tasks

Introduction

celery beat is a scheduler. It kicks off tasks at regular intervals, which are then executed by the worker nodes available in the cluster.

By default the entries are taken from the `CELERYBEAT_SCHEDULE` setting, but custom stores can also be used, like storing the entries in an SQL database.

You have to ensure only a single scheduler is running for a schedule at a time, otherwise you would end up with duplicate tasks. Using a centralized approach means the schedule does not have to be synchronized, and the service can operate without using locks.

Time Zones

The periodic task schedules uses the UTC time zone by default, but you can change the time zone used using the `CELERY_TIMEZONE` setting.

The pytz library is recommended when setting a default timezone. If `pytz` is not installed it will fallback to the mod:*dateutil* library, which depends on a system timezone file being available for the timezone selected.

Timezone definitions change frequently, so for the best results an up to date `pytz` installation should be used.

```
$ pip install -U pytz
```

An example time zone could be *Europe/London*:

```
CELERY_TIMEZONE = 'Europe/London'
```

The default scheduler (storing the schedule in the `celerybeat-schedule` file) will automatically detect that the time zone has changed, and so will reset the schedule itself, but other schedulers may not be so smart (e.g. the Django database scheduler, see below) and in that case you will have to reset the schedule manually.

Django Users

Celery recommends and is compatible with the new `USE_TZ` setting introduced in Django 1.4.

For Django users the time zone specified in the `TIME_ZONE` setting will be used, or you can specify a custom time zone for Celery alone by using the `CELERY_TIMEZONE` setting.

The database scheduler will not reset when timezone related settings change, so you must do this manually:

```
$ python manage.py shell
>>> from djcelery.models import PeriodicTask
>>> PeriodicTask.objects.update(last_run_at=None)
```

Entries

To schedule a task periodically you have to add an entry to the `CELERYBEAT_SCHEDULE` setting.

Example: Run the *tasks.add* task every 30 seconds.

```
from datetime import timedelta

CELERYBEAT_SCHEDULE = {
    'add-every-30-seconds': {
        'task': 'tasks.add',
        'schedule': timedelta(seconds=30),
        'args': (16, 16)
    },
}

CELERY_TIMEZONE = 'UTC'
```

Using a `timedelta` for the schedule means the task will be sent in 30 second intervals (the first task will be sent 30 seconds after *celery beat* starts, and then every 30 seconds after the last run).

A crontab like schedule also exists, see the section on Crontab schedules.

Like with `cron`, the tasks may overlap if the first task does not complete before the next. If that is a concern you should use a locking strategy to ensure only one instance can run at a time (see for example *Ensuring a task is only executed one at a time*).

Available Fields

- *task*

  The name of the task to execute.

- *schedule*

  The frequency of execution.

  This can be the number of seconds as an integer, a `timedelta`, or a `crontab`. You can also define your own custom schedule types, by extending the interface of `schedule`.

- *args*

  Positional arguments (`list` or `tuple`).

- *kwargs*

  Keyword arguments (`dict`).

- *options*

Execution options (`dict`).

This can be any argument supported by `apply_async()`, e.g. *exchange*, *routing_key*, *expires*, and so on.

- *relative*

  By default `timedelta` schedules are scheduled "by the clock". This means the frequency is rounded to the nearest second, minute, hour or day depending on the period of the timedelta.

  If *relative* is true the frequency is not rounded and will be relative to the time when **celery beat** was started.

Crontab schedules

If you want more control over when the task is executed, for example, a particular time of day or day of the week, you can use the `crontab` schedule type:

```
from celery.schedules import crontab

CELERYBEAT_SCHEDULE = {
    # Executes every Monday morning at 7:30 A.M
    'add-every-monday-morning': {
        'task': 'tasks.add',
        'schedule': crontab(hour=7, minute=30, day_of_week=1),
        'args': (16, 16),
    },
}
```

The syntax of these crontab expressions are very flexible. Some examples:

Example	Meaning
`crontab()`	Execute every minute.
`crontab(minute=0, hour=0)`	Execute daily at midnight.
`crontab(minute=0, hour='*/3')`	Execute every three hours: 3am, 6am, 9am, noon, 3pm, 6pm, 9pm.
`crontab(minute=0,` `hour='0,3,6,9,12,15,18,21')`	Same as previous.
`crontab(minute='*/15')`	Execute every 15 minutes.
`crontab(day_of_week='sunday')`	Execute every minute (!) at Sundays.
`crontab(minute='*',` `hour='*',` `day_of_week='sun')`	Same as previous.

`crontab(minute='*/10',`
`hour='3,17,22',`
`day_of_week='thu,fri')`
	Execute every ten minutes, but only between 3-4 am, 5-6 pm and 10-11 pm on Thursdays or Fridays.

`crontab(minute=0, hour='*/2,*/3')`	Execute every even hour, and every hour divisible by three. This means: at every hour *except*: 1am, 5am, 7am, 11am, 1pm, 5pm, 7pm, 11pm
`crontab(minute=0, hour='*/5')`	Execute hour divisible by 5. This means that it is triggered at 3pm, not 5pm (since 3pm equals the 24-hour clock value of "15", which is divisible by 5).
`crontab(minute=0, hour='*/3,8-17')`	Execute every hour divisible by 3, and every hour during office hours (8am-5pm).
`crontab(day_of_month='2')`	Execute on the second day of every month.
`crontab(day_of_month='2-30/3')`	Execute on every even numbered day.
`crontab(day_of_month='1-7,15-21')`	Execute on the first and third weeks of the month.
`crontab(day_of_month='11',` `month_of_year='5')`	Execute on 11th of May every year.
`crontab(month_of_year='*/3')`	Execute on the first month of every quarter.

See `celery.schedules.crontab` for more documentation.

Starting the Scheduler

To start the **celery beat** service:

```
$ celery beat
```

You can also start embed *beat* inside the worker by enabling workers *-B* option, this is convenient if you only intend to use one worker node:

```
$ celery worker -B
```

Beat needs to store the last run times of the tasks in a local database file (named *celerybeat-schedule* by default), so it needs access to write in the current directory, or alternatively you can specify a custom location for this file:

```
$ celery beat -s /home/celery/var/run/celerybeat-schedule
```

Note: To daemonize beat see *Running the worker as a daemon*.

Using custom scheduler classes

Custom scheduler classes can be specified on the command line (the *-S* argument). The default scheduler is `celery.beat.PersistentScheduler`, which is simply keeping track of the last run times in a local database file (a `shelve`).

django-celery also ships with a scheduler that stores the schedule in the Django database:

```
$ celery beat -S djcelery.schedulers.DatabaseScheduler
```

Using *django-celery*'s scheduler you can add, modify and remove periodic tasks from the Django Admin.

2.3.7 HTTP Callback Tasks (Webhooks)

Basics

If you need to call into another language, framework or similar, you can do so by using HTTP callback tasks.

The HTTP callback tasks uses GET/POST data to pass arguments and returns result as a JSON response. The scheme to call a task is:

```
GET http://example.com/mytask/?arg1=a&arg2=b&arg3=c
```

or using POST:

```
POST http://example.com/mytask
```

Note: POST data needs to be form encoded.

Whether to use GET or POST is up to you and your requirements.

The web page should then return a response in the following format if the execution was successful:

```
{'status': 'success', 'retval': ....}
```

or if there was an error:

```
{'status': 'failure': 'reason': 'Invalid moon alignment.'}
```

Enabling the HTTP task

To enable the HTTP dispatch task you have to add `celery.task.http` to `CELERY_IMPORTS`, or start the worker with `-I celery.task.http`.

Django webhook example

With this information you could define a simple task in Django:

```
from django.http import HttpResponse
from anyjson import serialize


def multiply(request):
    x = int(request.GET['x'])
    y = int(request.GET['y'])
    result = x * y
    response = {'status': 'success', 'retval': result}
    return HttpResponse(serialize(response), mimetype='application/json')
```

Ruby on Rails webhook example

or in Ruby on Rails:

```
def multiply
    @x = params[:x].to_i
    @y = params[:y].to_i

    @status = {:status => 'success', :retval => @x * @y}

    render :json => @status
end
```

You can easily port this scheme to any language/framework; new examples and libraries are very welcome.

Calling webhook tasks

To call a task you can use the `URL` class:

```
>>> from celery.task.http import URL
>>> res = URL('http://example.com/multiply').get_async(x=10, y=10)
```

`URL` is a shortcut to the `HttpDispatchTask`. You can subclass this to extend the functionality.

```
>>> from celery.task.http import HttpDispatchTask
>>> res = HttpDispatchTask.delay(
...     url='http://example.com/multiply',
...     method='GET', x=10, y=10)
>>> res.get()
100
```

The output of **celery worker** (or the log file if enabled) should show the task being executed:

```
[INFO/MainProcess] Task celery.task.http.HttpDispatchTask
        [f2cc8efc-2a14-40cd-85ad-f1c77c94beeb] processed: 100
```

Since calling tasks can be done via HTTP using the `djcelery.views.apply()` view, calling tasks from other languages is easy. For an example service exposing tasks via HTTP you should have a look at *examples/celery_http_gateway* in the Celery distribution: http://github.com/celery/celery/tree/master/examples/celery_http_gateway/

2.3.8 Routing Tasks

Note: Alternate routing concepts like topic and fanout may not be available for all transports, please consult the *transport comparison table*.

Basics

Automatic routing

The simplest way to do routing is to use the `CELERY_CREATE_MISSING_QUEUES` setting (on by default).

With this setting on, a named queue that is not already defined in `CELERY_QUEUES` will be created automatically. This makes it easy to perform simple routing tasks.

Say you have two servers, *x*, and *y* that handles regular tasks, and one server *z*, that only handles feed related tasks. You can use this configuration:

```
CELERY_ROUTES = {'feed.tasks.import_feed': {'queue': 'feeds'}}
```

With this route enabled import feed tasks will be routed to the *"feeds"* queue, while all other tasks will be routed to the default queue (named *"celery"* for historical reasons).

Now you can start server *z* to only process the feeds queue like this:

```
user@z:/$ celery worker -Q feeds
```

You can specify as many queues as you want, so you can make this server process the default queue as well:

```
user@z:/$ celery worker -Q feeds,celery
```

Changing the name of the default queue You can change the name of the default queue by using the following configuration:

```
from kombu import Exchange, Queue

CELERY_DEFAULT_QUEUE = 'default'
CELERY_QUEUES = (
    Queue('default', Exchange('default'), routing_key='default'),
)
```

How the queues are defined The point with this feature is to hide the complex AMQP protocol for users with only basic needs. However – you may still be interested in how these queues are declared.

A queue named *"video"* will be created with the following settings:

```
{'exchange': 'video',
 'exchange_type': 'direct',
 'routing_key': 'video'}
```

The non-AMQP backends like *ghettoq* does not support exchanges, so they require the exchange to have the same name as the queue. Using this design ensures it will work for them as well.

Manual routing

Say you have two servers, *x*, and *y* that handles regular tasks, and one server *z*, that only handles feed related tasks, you can use this configuration:

```
from kombu import Queue

CELERY_DEFAULT_QUEUE = 'default'
CELERY_QUEUES = (
    Queue('default',    routing_key='task.#'),
    Queue('feed_tasks', routing_key='feed.#'),
)
CELERY_DEFAULT_EXCHANGE = 'tasks'
CELERY_DEFAULT_EXCHANGE_TYPE = 'topic'
CELERY_DEFAULT_ROUTING_KEY = 'task.default'
```

`CELERY_QUEUES` is a list of `Queue` instances. If you don't set the exchange or exchange type values for a key, these will be taken from the `CELERY_DEFAULT_EXCHANGE` and `CELERY_DEFAULT_EXCHANGE_TYPE` settings.

To route a task to the *feed_tasks* queue, you can add an entry in the `CELERY_ROUTES` setting:

```
CELERY_ROUTES = {
        'feeds.tasks.import_feed': {
            'queue': 'feed_tasks',
            'routing_key': 'feed.import',
        },
}
```

You can also override this using the *routing_key* argument to `Task.apply_async()`, or `send_task()`:

```
>>> from feeds.tasks import import_feed
>>> import_feed.apply_async(args=['http://cnn.com/rss'],
...                         queue='feed_tasks',
...                         routing_key='feed.import')
```

To make server *z* consume from the feed queue exclusively you can start it with the `-Q` option:

```
user@z:/$ celery worker -Q feed_tasks --hostname=z.example.com
```

Servers *x* and *y* must be configured to consume from the default queue:

```
user@x:/$ celery worker -Q default --hostname=x.example.com
user@y:/$ celery worker -Q default --hostname=y.example.com
```

If you want, you can even have your feed processing worker handle regular tasks as well, maybe in times when there's a lot of work to do:

```
user@z:/$ celery worker -Q feed_tasks,default --hostname=z.example.com
```

If you have another queue but on another exchange you want to add, just specify a custom exchange and exchange type:

```
from kombu import Exchange, Queue

CELERY_QUEUES = (
    Queue('feed_tasks',     routing_key='feed.#'),
    Queue('regular_tasks', routing_key='task.#'),
    Queue('image_tasks',    exchange=Exchange('mediatasks', type='direct'),
                            routing_key='image.compress'),
)
```

If you're confused about these terms, you should read up on AMQP.

See also:

In addition to the *AMQP Primer* below, there's Rabbits and Warrens, an excellent blog post describing queues and exchanges. There's also AMQP in 10 minutes*: Flexible Routing Model, and Standard Exchange Types. For users of RabbitMQ the RabbitMQ FAQ could be useful as a source of information.

AMQP Primer

Messages

A message consists of headers and a body. Celery uses headers to store the content type of the message and its content encoding. The content type is usually the serialization format used to serialize the message. The body contains the name of the task to execute, the task id (UUID), the arguments to apply it with and some additional metadata – like the number of retries or an ETA.

This is an example task message represented as a Python dictionary:

```
{'task': 'myapp.tasks.add',
 'id': '54086c5e-6193-4575-8308-dbab76798756',
 'args': [4, 4],
 'kwargs': {}}
```

Producers, consumers and brokers

The client sending messages is typically called a *publisher*, or a *producer*, while the entity receiving messages is called a *consumer*.

The *broker* is the message server, routing messages from producers to consumers.

You are likely to see these terms used a lot in AMQP related material.

Exchanges, queues and routing keys.

1. Messages are sent to exchanges.
2. An exchange routes messages to one or more queues. Several exchange types exists, providing different ways to do routing, or implementing different messaging scenarios.
3. The message waits in the queue until someone consumes it.
4. The message is deleted from the queue when it has been acknowledged.

The steps required to send and receive messages are:

1. Create an exchange
2. Create a queue
3. Bind the queue to the exchange.

Celery automatically creates the entities necessary for the queues in `CELERY_QUEUES` to work (except if the queue's *auto_declare* setting is set to `False`).

Here's an example queue configuration with three queues; One for video, one for images and one default queue for everything else:

```
from kombu import Exchange, Queue

CELERY_QUEUES = (
    Queue('default', Exchange('default'), routing_key='default'),
    Queue('videos',  Exchange('media'),   routing_key='media.video'),
    Queue('images',  Exchange('media'),   routing_key='media.image'),
)
CELERY_DEFAULT_QUEUE = 'default'
CELERY_DEFAULT_EXCHANGE_TYPE = 'direct'
CELERY_DEFAULT_ROUTING_KEY = 'default'
```

Exchange types

The exchange type defines how the messages are routed through the exchange. The exchange types defined in the standard are *direct*, *topic*, *fanout* and *headers*. Also non-standard exchange types are available as plug-ins to RabbitMQ, like the last-value-cache plug-in by Michael Bridgen.

Direct exchanges Direct exchanges match by exact routing keys, so a queue bound by the routing key *video* only receives messages with that routing key.

Topic exchanges Topic exchanges matches routing keys using dot-separated words, and the wildcard characters: `*` (matches a single word), and # (matches zero or more words).

With routing keys like `usa.news`, `usa.weather`, `norway.news` and `norway.weather`, bindings could be `*.news` (all news), `usa.#` (all items in the USA) or `usa.weather` (all USA weather items).

Related API commands

exchange.declare(exchange_name, type, passive, durable, auto_delete, internal)
Declares an exchange by name.

Parameters

- **passive** – Passive means the exchange won't be created, but you can use this to check if the exchange already exists.
- **durable** – Durable exchanges are persistent. That is - they survive a broker restart.
- **auto_delete** – This means the queue will be deleted by the broker when there are no more queues using it.

`queue.declare`(*queue_name*, *passive*, *durable*, *exclusive*, *auto_delete*)
Declares a queue by name.

Exclusive queues can only be consumed from by the current connection. Exclusive also implies *auto_delete*.

`queue.bind`(*queue_name*, *exchange_name*, *routing_key*)
Binds a queue to an exchange with a routing key. Unbound queues will not receive messages, so this is necessary.

`queue.delete`(*name*, *if_unused=False*, *if_empty=False*)
Deletes a queue and its binding.

`exchange.delete`(*name*, *if_unused=False*)
Deletes an exchange.

Note: Declaring does not necessarily mean "create". When you declare you *assert* that the entity exists and that it's operable. There is no rule as to whom should initially create the exchange/queue/binding, whether consumer or producer. Usually the first one to need it will be the one to create it.

Hands-on with the API

Celery comes with a tool called **celery amqp** that is used for command line access to the AMQP API, enabling access to administration tasks like creating/deleting queues and exchanges, purging queues or sending messages. It can also be used for non-AMQP brokers, but different implementation may not implement all commands.

You can write commands directly in the arguments to **celery amqp**, or just start with no arguments to start it in shell-mode:

```
$ celery amqp
-> connecting to amqp://guest@localhost:5672/.
-> connected.
1>
```

Here `1>` is the prompt. The number 1, is the number of commands you have executed so far. Type `help` for a list of commands available. It also supports auto-completion, so you can start typing a command and then hit the *tab* key to show a list of possible matches.

Let's create a queue you can send messages to:

```
$ celery amqp
1> exchange.declare testexchange direct
ok.
2> queue.declare testqueue
ok. queue:testqueue messages:0 consumers:0.
3> queue.bind testqueue testexchange testkey
ok.
```

This created the direct exchange `testexchange`, and a queue named `testqueue`. The queue is bound to the exchange using the routing key `testkey`.

From now on all messages sent to the exchange `testexchange` with routing key `testkey` will be moved to this queue. You can send a message by using the `basic.publish` command:

```
4> basic.publish 'This is a message!' testexchange testkey
ok.
```

Now that the message is sent you can retrieve it again. You can use the `basic.get ` command here, which polls for new messages on the queue (which is alright for maintainence tasks, for services you'd want to use `basic.consume` instead)

Pop a message off the queue:

```
5> basic.get testqueue
{'body': 'This is a message!',
 'delivery_info': {'delivery_tag': 1,
                   'exchange': u'testexchange',
                   'message_count': 0,
                   'redelivered': False,
                   'routing_key': u'testkey'},
 'properties': {}}
```

AMQP uses acknowledgment to signify that a message has been received and processed successfully. If the message has not been acknowledged and consumer channel is closed, the message will be delivered to another consumer.

Note the delivery tag listed in the structure above; Within a connection channel, every received message has a unique delivery tag, This tag is used to acknowledge the message. Also note that delivery tags are not unique across connections, so in another client the delivery tag *1* might point to a different message than in this channel.

You can acknowledge the message you received using `basic.ack`:

```
6> basic.ack 1
ok.
```

To clean up after our test session you should delete the entities you created:

```
7> queue.delete testqueue
ok. 0 messages deleted.
8> exchange.delete testexchange
ok.
```

Routing Tasks

Defining queues

In Celery available queues are defined by the `CELERY_QUEUES` setting.

Here's an example queue configuration with three queues; One for video, one for images and one default queue for everything else:

```
default_exchange = Exchange('default', type='direct')
media_exchange = Exchange('media', type='direct')

CELERY_QUEUES = (
    Queue('default', default_exchange, routing_key='default'),
    Queue('videos', media_exchange, routing_key='media.video')
    Queue('images', media_exchange, routing_key='media.image')
)
CELERY_DEFAULT_QUEUE = 'default'
```

```
CELERY_DEFAULT_EXCHANGE = 'default'
CELERY_DEFAULT_ROUTING_KEY = 'default'
```

Here, the CELERY_DEFAULT_QUEUE will be used to route tasks that doesn't have an explicit route.

The default exchange, exchange type and routing key will be used as the default routing values for tasks, and as the default values for entries in CELERY_QUEUES.

Specifying task destination

The destination for a task is decided by the following (in order):

1. The *Routers* defined in CELERY_ROUTES.
2. The routing arguments to `Task.apply_async()`.
3. Routing related attributes defined on the `Task` itself.

It is considered best practice to not hard-code these settings, but rather leave that as configuration options by using *Routers*; This is the most flexible approach, but sensible defaults can still be set as task attributes.

Routers

A router is a class that decides the routing options for a task.

All you need to define a new router is to create a class with a `route_for_task` method:

```
class MyRouter(object):

    def route_for_task(self, task, args=None, kwargs=None):
        if task == 'myapp.tasks.compress_video':
            return {'exchange': 'video',
                    'exchange_type': 'topic',
                    'routing_key': 'video.compress'}
        return None
```

If you return the `queue` key, it will expand with the defined settings of that queue in CELERY_QUEUES:

```
{'queue': 'video', 'routing_key': 'video.compress'}
```

becomes –>

```
{'queue': 'video',
 'exchange': 'video',
 'exchange_type': 'topic',
 'routing_key': 'video.compress'}
```

You install router classes by adding them to the CELERY_ROUTES setting:

```
CELERY_ROUTES = (MyRouter(), )
```

Router classes can also be added by name:

```
CELERY_ROUTES = ('myapp.routers.MyRouter', )
```

For simple task name -> route mappings like the router example above, you can simply drop a dict into CELERY_ROUTES to get the same behavior:

```
CELERY_ROUTES = ({'myapp.tasks.compress_video': {
                        'queue': 'video',
                        'routing_key': 'video.compress'
                  }}, )
```

The routers will then be traversed in order, it will stop at the first router returning a true value, and use that as the final route for the task.

Broadcast

Celery can also support broadcast routing. Here is an example exchange `broadcast_tasks` that delivers copies of tasks to all workers connected to it:

```
from kombu.common import Broadcast

CELERY_QUEUES = (Broadcast('broadcast_tasks'), )

CELERY_ROUTES = {'tasks.reload_cache': {'queue': 'broadcast_tasks'}}
```

Now the `tasks.reload_tasks` task will be sent to every worker consuming from this queue.

Broadcast & Results

Note that Celery result does not define what happens if two tasks have the same task_id. If the same task is distributed to more than one worker, then the state history may not be preserved.

It is a good idea to set the `task.ignore_result` attribute in this case.

2.3.9 Monitoring and Management Guide

Introduction

There are several tools available to monitor and inspect Celery clusters.

This document describes some of these, as as well as features related to monitoring, like events and broadcast commands.

Workers

`celery`: Command Line Management Utility

New in version 2.1.

celery can also be used to inspect and manage worker nodes (and to some degree tasks).

To list all the commands available do:

```
$ celery help
```

or to get help for a specific command do:

```
$ celery <command> --help
```

Commands

- **shell**: Drop into a Python shell.

  The locals will include the `celery` variable, which is the current app. Also all known tasks will be automatically added to locals (unless the `--without-tasks` flag is set).

  Uses Ipython, bpython, or regular python in that order if installed. You can force an implementation using `--force-ipython|-I`, `--force-bpython|-B`, or `--force-python|-P`.

- **status**: List active nodes in this cluster

  ```
  $ celery status
  ```

- **result**: Show the result of a task

  ```
  $ celery result -t tasks.add 4e196aa4-0141-4601-8138-7aa33db0f577
  ```

  Note that you can omit the name of the task as long as the task doesn't use a custom result backend.

- **purge**: Purge messages from all configured task queues.

  ```
  $ celery purge
  ```

  > **Warning:** There is no undo for this operation, and messages will be permanently deleted!

- **inspect active**: List active tasks

  ```
  $ celery inspect active
  ```

  These are all the tasks that are currently being executed.

- **inspect scheduled**: List scheduled ETA tasks

  ```
  $ celery inspect scheduled
  ```

  These are tasks reserved by the worker because they have the *eta* or *countdown* argument set.

- **inspect reserved**: List reserved tasks

  ```
  $ celery inspect reserved
  ```

  This will list all tasks that have been prefetched by the worker, and is currently waiting to be executed (does not include tasks with an eta).

- **inspect revoked**: List history of revoked tasks

  ```
  $ celery inspect revoked
  ```

- **inspect registered**: List registered tasks

  ```
  $ celery inspect registered
  ```

- **inspect stats**: Show worker statistics

  ```
  $ celery inspect stats
  ```

- **control enable_events**: Enable events

```
$ celery control enable_events
```

- **control disable_events**: Disable events

```
$ celery control disable_events
```

- **migrate**: Migrate tasks from one broker to another (**EXPERIMENTAL**).

```
$ celery migrate redis://localhost amqp://localhost
```

This command will migrate all the tasks on one broker to another. As this command is new and experimental you should be sure to have a backup of the data before proceeding.

Note: All `inspect` commands supports a `--timeout` argument, This is the number of seconds to wait for responses. You may have to increase this timeout if you're not getting a response due to latency.

Specifying destination nodes By default the inspect commands operates on all workers. You can specify a single, or a list of workers by using the *–destination* argument:

```
$ celery inspect -d w1,w2 reserved
```

Flower: Real-time Celery web-monitor

Flower is a real-time web based monitor and administration tool for Celery. It is under active development, but is already an essential tool. Being the recommended monitor for Celery, it obsoletes the Django-Admin monitor, celerymon and the ncurses based monitor.

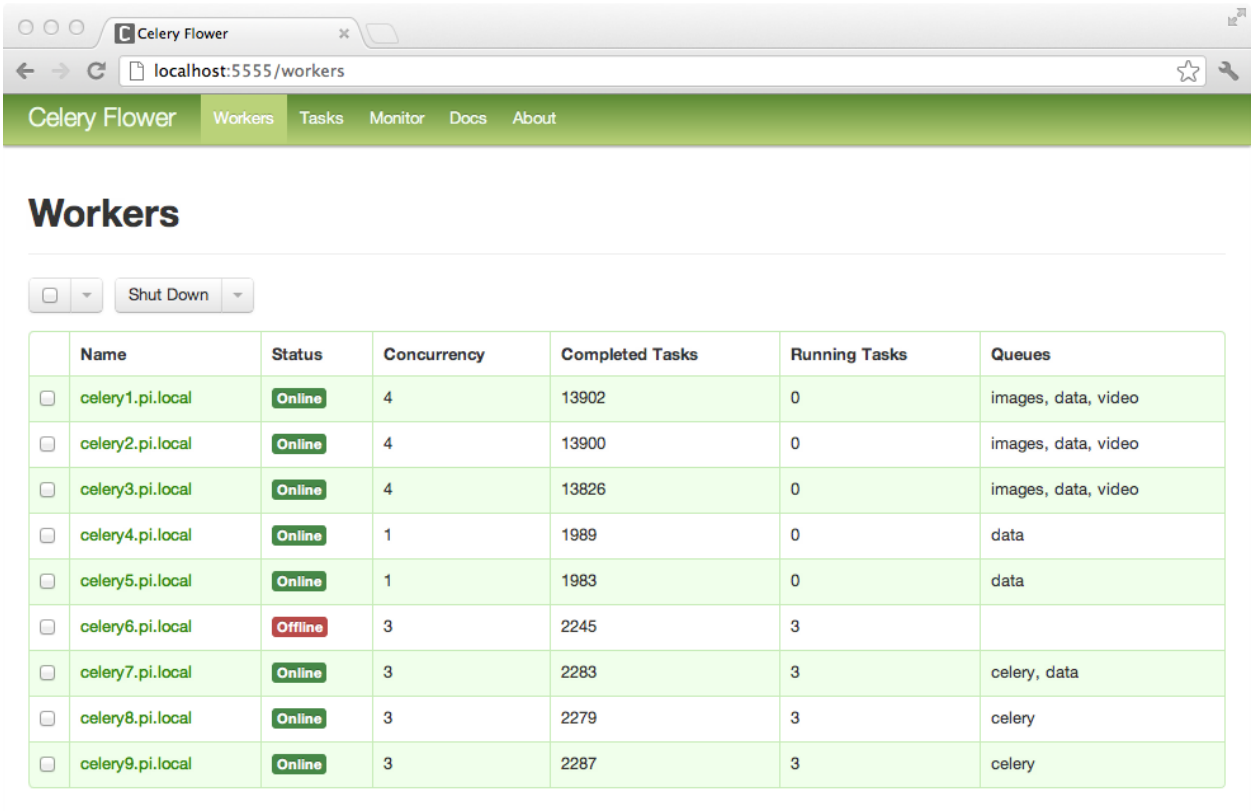
Flower is pronounced like "flow", but you can also use the botanical version if you prefer.

Features

- Real-time monitoring using Celery Events
  - Task progress and history.
  - Ability to show task details (arguments, start time, runtime, and more)
  - Graphs and statistics
- Remote Control
  - View worker status and statistics.
  - Shutdown and restart worker instances.
  - Control worker pool size and autoscale settings.
  - View and modify the queues a worker instance consumes from.
  - View currently running tasks
  - View scheduled tasks (ETA/countdown)
  - View reserved and revoked tasks
  - Apply time and rate limits
  - Configuration viewer
  - Revoke or terminate tasks

- HTTP API

Screenshots



More screenshots:

Usage You can use pip to install Flower:

```
$ pip install flower
```

Running the flower command will start a web-server that you can visit:

```
$ celery flower
```

The default port is http://localhost:5555, but you can change this using the *–port* argument:

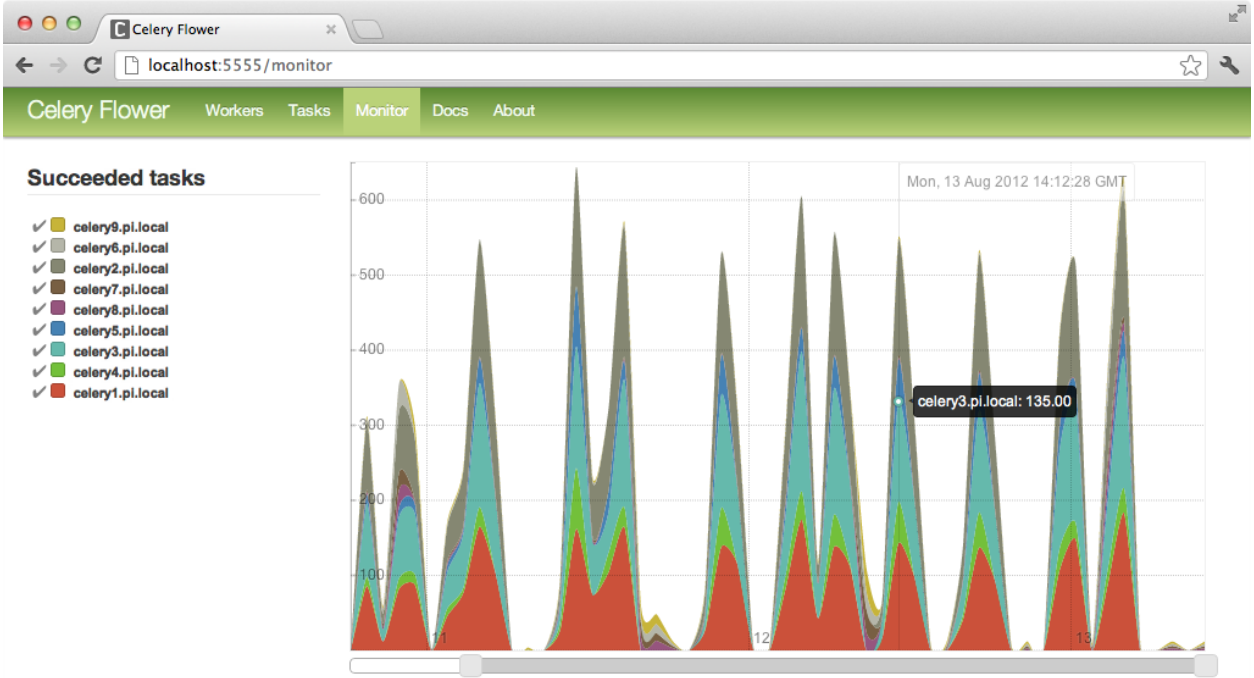
```
$ celery flower --port=5555
```

Broker URL can also be passed through the *–broker* argument :

```
$ celery flower --broker=amqp://guest:guest@localhost:5672//
or
$ celery flower --broker=redis://guest:guest@localhost:6379/0
```

Then, you can visit flower in your web browser :

```
$ open http://localhost:5555
```



celery events: Curses Monitor

New in version 2.0.

celery events is a simple curses monitor displaying task and worker history. You can inspect the result and traceback of tasks, and it also supports some management commands like rate limiting and shutting down workers. This monitor was started as a proof of concept, and you probably want to use Flower instead.

Starting:

```
$ celery events
```

You should see a screen like:

celery events is also used to start snapshot cameras (see *Snapshots*:

```
$ celery events --camera=<camera-class> --frequency=1.0
```

and it includes a tool to dump events to `stdout`:

```
$ celery events --dump
```

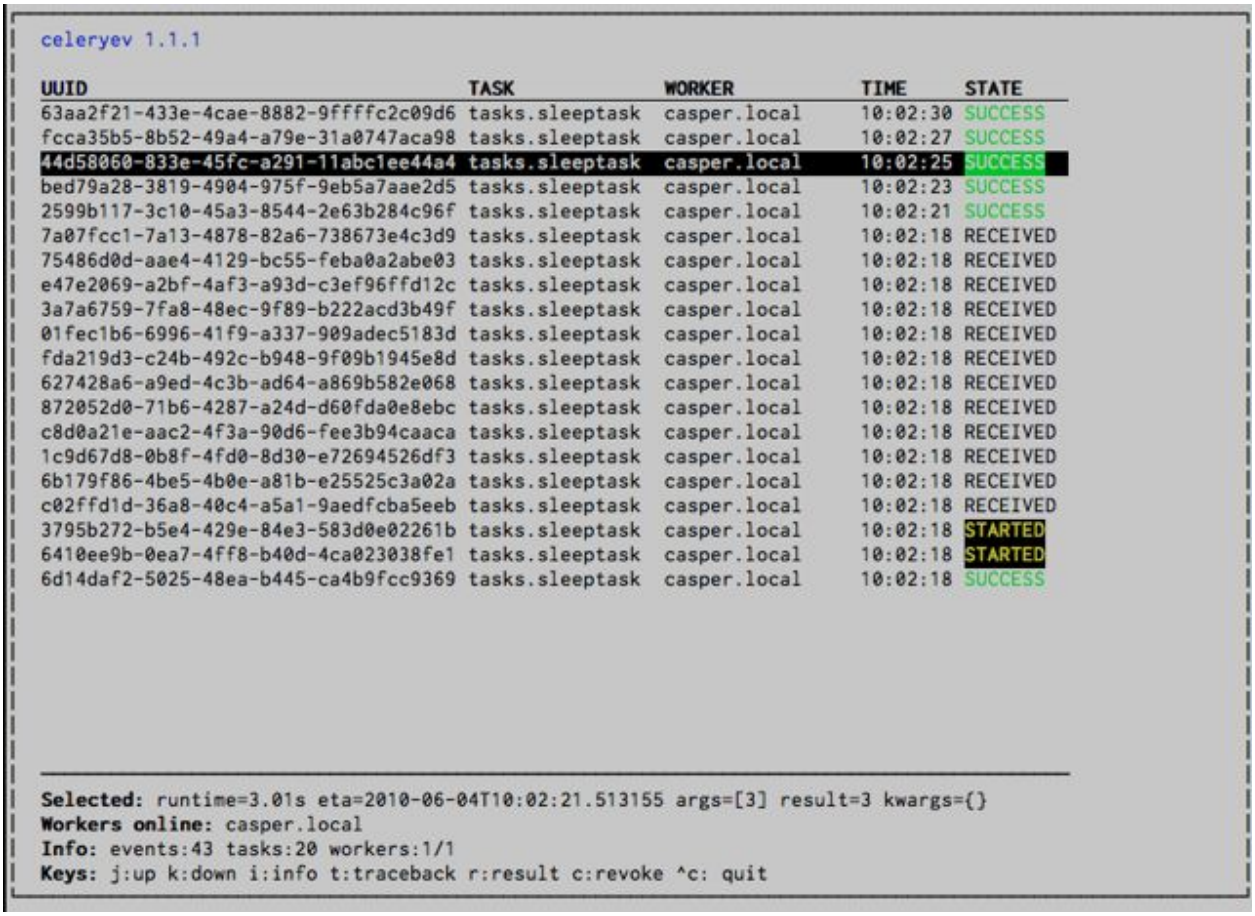
For a complete list of options use `--help`:

```
$ celery events --help
```

RabbitMQ

To manage a Celery cluster it is important to know how RabbitMQ can be monitored.

RabbitMQ ships with the rabbitmqctl(1) command, with this you can list queues, exchanges, bindings, queue lengths, the memory usage of each queue, as well as manage users, virtual hosts and their permissions.


celeryev 1.1.1
UUID TASK WORKER TIME STATE
63aa2f21-433e-4cae-8882-9ffffc2c09d6 tasks.sleeptask casper.local 10:02:30 SUCCESS
fcca35b5-8b52-49a4-a79e-31a0747aca98 tasks.sleeptask casper.local 10:02:27 SUCCESS
44d58060-833e-45fc-a291-11abc1ee44a4 tasks.sleeptask casper.local 10:02:25 SUCCESS
bed79a28-3819-4904-975f-9eb5a7aae2d5 tasks.sleeptask casper.local 10:02:23 SUCCESS
2599b117-3c10-45a3-8544-2e63b284c96f tasks.sleeptask casper.local 10:02:21 SUCCESS
7a07fcc1-7a13-4878-82a6-738673e4c3d9 tasks.sleeptask casper.local 10:02:18 RECEIVED
75486d0d-aae4-4129-bc55-feba0a2abe03 tasks.sleeptask casper.local 10:02:18 RECEIVED
e47e2069-a2bf-4af3-a93d-c3ef96ffd12c tasks.sleeptask casper.local 10:02:18 RECEIVED
3a7a6759-7fa8-48ec-9f89-b222acd3b49f tasks.sleeptask casper.local 10:02:18 RECEIVED
01fec1b6-6996-41f9-a337-909adec5183d tasks.sleeptask casper.local 10:02:18 RECEIVED
fda219d3-c24b-492c-b948-9f09b1945e8d tasks.sleeptask casper.local 10:02:18 RECEIVED
627428a6-a9ed-4c3b-ad64-a869b582e068 tasks.sleeptask casper.local 10:02:18 RECEIVED
872052d0-71b6-4287-a24d-d60fda0e8ebc tasks.sleeptask casper.local 10:02:18 RECEIVED
c8d0a21e-aac2-4f3a-90d6-fee3b94caaca tasks.sleeptask casper.local 10:02:18 RECEIVED
1c9d67d8-0b8f-4fd0-8d30-e72694526df3 tasks.sleeptask casper.local 10:02:18 RECEIVED
6b179f86-4be5-4b0e-a81b-e25525c3a02a tasks.sleeptask casper.local 10:02:18 RECEIVED
c02ffd1d-36a8-40c4-a5a1-9aedfcba5eeb tasks.sleeptask casper.local 10:02:18 RECEIVED
3795b272-b5e4-429e-84e3-583d0e02261b tasks.sleeptask casper.local 10:02:18 STARTED
6410ee9b-0ea7-4ff8-b40d-4ca023038fe1 tasks.sleeptask casper.local 10:02:18 STARTED
6d14daf2-5025-48ea-b445-ca4b9fcc9369 tasks.sleeptask casper.local 10:02:18 SUCCESS
Selected: runtime=3.01s eta=2010-06-04T10:02:21.513155 args=[3] result=3 kwargs={}
Workers online: casper.local
Info: events:43 tasks:20 workers:1/1
Keys: j:up k:down i:info t:traceback r:result c:revoke ^c: quit

Note: The default virtual host (`"/"`) is used in these examples, if you use a custom virtual host you have to add the `-p` argument to the command, e.g: `rabbitmqctl list_queues -p my_vhost ....`

Inspecting queues

Finding the number of tasks in a queue:

```
$ rabbitmqctl list_queues name messages messages_ready \
                          messages_unacknowledged
```

Here *messages_ready* is the number of messages ready for delivery (sent but not received), *messages_unacknowledged* is the number of messages that has been received by a worker but not acknowledged yet (meaning it is in progress, or has been reserved). *messages* is the sum of ready and unacknowledged messages.

Finding the number of workers currently consuming from a queue:

```
$ rabbitmqctl list_queues name consumers
```

Finding the amount of memory allocated to a queue:

```
$ rabbitmqctl list_queues name memory
```

Tip Adding the `-q` option to rabbitmqctl(1) makes the output easier to parse.

Redis

If you're using Redis as the broker, you can monitor the Celery cluster using the *redis-cli(1)* command to list lengths of queues.

Inspecting queues

Finding the number of tasks in a queue:

```
$ redis-cli -h HOST -p PORT -n DATABASE_NUMBER llen QUEUE_NAME
```

The default queue is named *celery*. To get all available queues, invoke:

```
$ redis-cli -h HOST -p PORT -n DATABASE_NUMBER keys \*
```

Note: Queue keys only exists when there are tasks in them, so if a key does not exist it simply means there are no messages in that queue. This is because in Redis a list with no elements in it is automatically removed, and hence it won't show up in the *keys* command output, and *llen* for that list returns 0.

Also, if you're using Redis for other purposes, the output of the *keys* command will include unrelated values stored in the database. The recommended way around this is to use a dedicated *DATABASE_NUMBER* for Celery, you can also use database numbers to separate Celery applications from each other (virtual hosts), but this will not affect the monitoring events used by e.g. Flower as Redis pub/sub commands are global rather than database based.

Munin

This is a list of known Munin plug-ins that can be useful when maintaining a Celery cluster.

- rabbitmq-munin: Munin plug-ins for RabbitMQ.

  http://github.com/ask/rabbitmq-munin

- celery_tasks: Monitors the number of times each task type has been executed (requires *celerymon*).

  http://exchange.munin-monitoring.org/plugins/celery_tasks-2/details

- celery_task_states: Monitors the number of tasks in each state (requires *celerymon*).

  http://exchange.munin-monitoring.org/plugins/celery_tasks/details

Events

The worker has the ability to send a message whenever some event happens. These events are then captured by tools like Flower, and **celery events** to monitor the cluster.

Snapshots

New in version 2.1.

Even a single worker can produce a huge amount of events, so storing the history of all events on disk may be very expensive.

A sequence of events describes the cluster state in that time period, by taking periodic snapshots of this state you can keep all history, but still only periodically write it to disk.

To take snapshots you need a Camera class, with this you can define what should happen every time the state is captured; You can write it to a database, send it by email or something else entirely.

celery events is then used to take snapshots with the camera, for example if you want to capture state every 2 seconds using the camera `myapp.Camera` you run **celery events** with the following arguments:

```
$ celery events -c myapp.Camera --frequency=2.0
```

Custom Camera Cameras can be useful if you need to capture events and do something with those events at an interval. For real-time event processing you should use `celery.events.Receiver` directly, like in *Real-time processing*.

Here is an example camera, dumping the snapshot to screen:

```
from pprint import pformat

from celery.events.snapshot import Polaroid

class DumpCam(Polaroid):

    def on_shutter(self, state):
        if not state.event_count:
            # No new events since last snapshot.
            return
        print('Workers: %s' % (pformat(state.workers, indent=4), ))
        print('Tasks: %s' % (pformat(state.tasks, indent=4), ))
        print('Total: %s events, %s tasks' % (
            state.event_count, state.task_count))
```

See the API reference for `celery.events.state` to read more about state objects.

Now you can use this cam with **celery events** by specifying it with the `-c` option:

```
$ celery events -c myapp.DumpCam --frequency=2.0
```

Or you can use it programmatically like this:

```
from celery import Celery
from myapp import DumpCam

def main(app, freq=1.0):
    state = app.events.State()
    with app.connection() as connection:
        recv = app.events.Receiver(connection, handlers={'*': state.event})
        with DumpCam(state, freq=freq):
            recv.capture(limit=None, timeout=None)

if __name__ == '__main__':
    celery = Celery(broker='amqp://guest@localhost//')
    main(celery)
```

Real-time processing

To process events in real-time you need the following

- An event consumer (this is the `Receiver`)
- A set of handlers called when events come in.

  You can have different handlers for each event type, or a catch-all handler can be used ('*')
- State (optional)

  `celery.events.State` is a convenient in-memory representation of tasks and workers in the cluster that is updated as events come in.

  It encapsulates solutions for many common things, like checking if a worker is still alive (by verifying heartbeats), merging event fields together as events come in, making sure timestamps are in sync, and so on.

Combining these you can easily process events in real-time:

```
from celery import Celery


def monitor_events(app):
    state = app.events.State()

    def on_event(event):
        state.event(event)   # <-- updates in-memory cluster state

        print('Workers online: %r' % ', '.join(
            worker for worker in state.workers if worker.alive
        )

    with app.connection() as connection:
        recv = app.events.Receiver(connection, handlers={'*': on_event})
        recv.capture(limit=None, timeout=None, wakeup=True)
```

Note: The wakeup argument to `capture` sends a signal to all workers to force them to send a heartbeat. This way you can immediately see workers when the monitor starts.

You can listen to specific events by specifying the handlers:

```
from celery import Celery

def my_monitor(app):
    state = app.events.State()

    def announce_failed_tasks(event):
        state.event(event)
        task_id = event['uuid']

        print('TASK FAILED: %s[%s] %s' % (
            event['name'], task_id, state[task_id].info(), ))

    def announce_dead_workers(event):
        state.event(event)
        hostname = event['hostname']

        if not state.workers[hostname].alive:
            print('Worker %s missed heartbeats' % (hostname, ))


    with app.connection() as connection:
        recv = app.events.Receiver(connection, handlers={
                'task-failed': announce_failed_tasks,
                'worker-heartbeat': announce_dead_workers,
        })
        recv.capture(limit=None, timeout=None, wakeup=True)

if __name__ == '__main__':
    celery = Celery(broker='amqp://guest@localhost//')
    my_monitor(celery)
```

Event Reference

This list contains the events sent by the worker, and their arguments.

Task Events

task-sent

signature `task-sent(uuid, name, args, kwargs, retries, eta, expires, queue, exchange, routing_key)`

Sent when a task message is published and the `CELERY_SEND_TASK_SENT_EVENT` setting is enabled.

task-received

signature `task-received(uuid, name, args, kwargs, retries, eta, hostname, timestamp)`

Sent when the worker receives a task.

task-started

signature `task-started(uuid, hostname, timestamp, pid)`

Sent just before the worker executes the task.

task-succeeded

signature `task-succeeded(uuid, result, runtime, hostname, timestamp)`

Sent if the task executed successfully.

Runtime is the time it took to execute the task using the pool. (Starting from the task is sent to the worker pool, and ending when the pool result handler callback is called).

task-failed

signature `task-failed(uuid, exception, traceback, hostname, timestamp)`

Sent if the execution of the task failed.

task-revoked

signature `task-revoked(uuid, terminated, signum, expired)`

Sent if the task has been revoked (Note that this is likely to be sent by more than one worker).

- **`terminated` is set to true if the task process was terminated,** and the `signum` field set to the signal used.
- `expired` is set to true if the task expired.

task-retried

signature `task-retried(uuid, exception, traceback, hostname, timestamp)`

Sent if the task failed, but will be retried in the future.

Worker Events

worker-online

signature `worker-online(hostname, timestamp, freq, sw_ident, sw_ver, sw_sys)`

The worker has connected to the broker and is online.

- *hostname*: Hostname of the worker.
- *timestamp*: Event timestamp.
- *freq*: Heartbeat frequency in seconds (float).
- *sw_ident*: Name of worker software (e.g. `py-celery`).
- *sw_ver*: Software version (e.g. 2.2.0).
- *sw_sys*: Operating System (e.g. Linux, Windows, Darwin).

worker-heartbeat

signature `worker-heartbeat(hostname, timestamp, freq, sw_ident, sw_ver, sw_sys, active, processed)`

Sent every minute, if the worker has not sent a heartbeat in 2 minutes, it is considered to be offline.

- *hostname*: Hostname of the worker.
- *timestamp*: Event timestamp.

- *freq*: Heartbeat frequency in seconds (float).
- *sw_ident*: Name of worker software (e.g. `py-celery`).
- *sw_ver*: Software version (e.g. 2.2.0).
- *sw_sys*: Operating System (e.g. Linux, Windows, Darwin).
- *active*: Number of currently executing tasks.
- *processed*: Total number of tasks processed by this worker.

worker-offline

signature `worker-offline(hostname, timestamp, freq, sw_ident, sw_ver, sw_sys)`

The worker has disconnected from the broker.

2.3.10 Security

Introduction

While Celery is written with security in mind, it should be treated as an unsafe component.

Depending on your Security Policy, there are various steps you can take to make your Celery installation more secure.

Areas of Concern

Broker

It is imperative that the broker is guarded from unwanted access, especially if it is publically accesible. By default, workers trust that the data they get from the broker has not been tampered with. See Message Signing for information on how to make the broker connection more trusthworthy.

The first line of defence should be to put a firewall in front of the broker, allowing only white-listed machines to access it.

Keep in mind that both firewall misconfiguration, and temproraily disabling the firewall, is common in the real world. Solid security policy includes monitoring of firewall equipment to detect if they have been disabled, be it accidentally or on purpose.

In other words, one should not blindly trust the firewall either.

If your broker supports fine-grained access control, like RabbitMQ, this is something you should look at enabling. See for example http://www.rabbitmq.com/access-control.html.

Client

In Celery, "client" refers to anything that sends messages to the broker, e.g. web-servers that apply tasks.

Having the broker properly secured doesn't matter if arbitrary messages can be sent through a client.

[Need more text here]

Worker

The default permissions of tasks running inside a worker are the same ones as the privileges of the worker itself. This applies to resources such as memory, file-systems and devices.

An exception to this rule is when using the multiprocessing based task pool, which is currently the default. In this case, the task will have access to any memory copied as a result of the `fork()` call (does not apply under MS Windows), and access to memory contents written by parent tasks in the same worker child process.

Limiting access to memory contents can be done by launching every task in a subprocess (`fork()` + `execve()`).

Limiting file-system and device access can be accomplished by using chroot, jail, sandboxing, virtual machines or other mechanisms as enabled by the platform or additional software.

Note also that any task executed in the worker will have the same network access as the machine on which it's running. If the worker is located on an internal network it's recommended to add firewall rules for outbound traffic.

Serializers

The default *pickle* serializer is convenient because it supports arbitrary Python objects, whereas other serializers only work with a restricted set of types.

But for the same reasons the *pickle* serializer is inherently insecure [1], and should be avoided whenever clients are untrusted or unauthenticated.

You can disable untrusted content by specifying a whitelist of accepted content-types in the `CELERY_ACCEPT_CONTENT` setting:

New in version 3.0.18.

Note: This setting was first supported in version 3.0.18. If you're running an earlier version it will simply be ignored, so make sure you're running a version that supports it.

```
CELERY_ACCEPT_CONTENT = ['json']
```

This accepts a list of serializer names and content-types, so you could also specify the content type for json:

```
CELERY_ACCEPT_CONTENT = ['application/json']
```

Celery also comes with a special *auth* serializer that validates communication between Celery clients and workers, making sure that messages originates from trusted sources. Using *Public-key cryptography* the *auth* serializer can verify the authenticity of senders, to enable this read *Message Signing* for more information.

[1] http://nadiana.com/python-pickle-insecure

Message Signing

Celery can use the pyOpenSSL library to sign message using *Public-key cryptography*, where messages sent by clients are signed using a private key and then later verified by the worker using a public certificate.

Optimally certificates should be signed by an official Certificate Authority, but they can also be self-signed.

To enable this you should configure the `CELERY_TASK_SERIALIZER` setting to use the *auth* serializer. Also required is configuring the paths used to locate private keys and certificates on the file-system: the `CELERY_SECURITY_KEY`, `CELERY_SECURITY_CERTIFICATE` and `CELERY_SECURITY_CERT_STORE` settings respectively. With these configured it is also necessary to call the `celery.security.setup_security()` function. Note that this will also disable all insucure serializers so that the worker won't accept messages with untrusted content types.

This is an example configuration using the *auth* serializer, with the private key and certificate files located in */etc/ssl*.

```
CELERY_SECURITY_KEY = '/etc/ssl/private/worker.key'
CELERY_SECURITY_CERTIFICATE = '/etc/ssl/certs/worker.pem'
CELERY_SECURITY_CERT_STORE = '/etc/ssl/certs/*.pem'
from celery.security import setup_security
setup_security()
```

Note: While relative paths are not disallowed, using absolute paths is recommended for these files.

Also note that the *auth* serializer won't encrypt the contents of a message, so if needed this will have to be enabled separately.

Intrusion Detection

The most important part when defending your systems against intruders is being able to detect if the system has been compromised.

Logs

Logs are usually the first place to look for evidence of security breaches, but they are useless if they can be tampered with.

A good solution is to set up centralized logging with a dedicated logging server. Acess to it should be restricted. In addition to having all of the logs in a single place, if configured correctly, it can make it harder for intruders to tamper with your logs.

This should be fairly easy to setup using syslog (see also syslog-ng and rsyslog.). Celery uses the `logging` library, and already has support for using syslog.

A tip for the paranoid is to send logs using UDP and cut the transmit part of the logging server's network cable :-)

Tripwire

Tripwire is a (now commercial) data integrity tool, with several open source implementations, used to keep cryptographic hashes of files in the file-system, so that administrators can be alerted when they change. This way when the damage is done and your system has been compromised you can tell exactly what files intruders have changed (password files, logs, backdoors, rootkits and so on). Often this is the only way you will be able to detect an intrusion.

Some open source implementations include:

- OSSEC

- Samhain
- Open Source Tripwire
- AIDE

Also, the ZFS file-system comes with built-in integrity checks that can be used.

2.3.11 Optimizing

Introduction

The default configuration makes a lot of compromises. It's not optimal for any single case, but works well enough for most situations.

There are optimizations that can be applied based on specific use cases.

Optimizations can apply to different properties of the running environment, be it the time tasks take to execute, the amount of memory used, or responsiveness at times of high load.

Ensuring Operations

In the book Programming Pearls, Jon Bentley presents the concept of back-of-the-envelope calculations by asking the question;

> How much water flows out of the Mississippi River in a day?

The point of this exercise [2] is to show that there is a limit to how much data a system can process in a timely manner. Back of the envelope calculations can be used as a means to plan for this ahead of time.

In Celery; If a task takes 10 minutes to complete, and there are 10 new tasks coming in every minute, the queue will never be empty. This is why it's very important that you monitor queue lengths!

A way to do this is by *using Munin*. You should set up alerts, that will notify you as soon as any queue has reached an unacceptable size. This way you can take appropriate action like adding new worker nodes, or revoking unnecessary tasks.

General Settings

librabbitmq

If you're using RabbitMQ (AMQP) as the broker then you can install the `librabbitmq` module to use an optimized client written in C:

```
$ pip install librabbitmq
```

The 'amqp' transport will automatically use the librabbitmq module if it's installed, or you can also specify the transport you want directly by using the `pyamqp://` or `librabbitmq://` prefixes.

Broker Connection Pools

The broker connection pool is enabled by default since version 2.5.

You can tweak the `BROKER_POOL_LIMIT` setting to minimize contention, and the value should be based on the number of active threads/greenthreads using broker connections.

[2] The chapter is available to read for free here: The back of the envelope. The book is a classic text. Highly recommended.

Worker Settings

Prefetch Limits

Prefetch is a term inherited from AMQP that is often misunderstood by users.

The prefetch limit is a **limit** for the number of tasks (messages) a worker can reserve for itself. If it is zero, the worker will keep consuming messages, not respecting that there may be other available worker nodes that may be able to process them sooner [3], or that the messages may not even fit in memory.

The workers' default prefetch count is the CELERYD_PREFETCH_MULTIPLIER setting multiplied by the number of child worker processes [4].

If you have many tasks with a long duration you want the multiplier value to be 1, which means it will only reserve one task per worker process at a time.

However – If you have many short-running tasks, and throughput/round trip latency is important to you, this number should be large. The worker is able to process more tasks per second if the messages have already been prefetched, and is available in memory. You may have to experiment to find the best value that works for you. Values like 50 or 150 might make sense in these circumstances. Say 64, or 128.

If you have a combination of long- and short-running tasks, the best option is to use two worker nodes that are configured separately, and route the tasks according to the run-time. (see *Routing Tasks*).

Reserve one task at a time

When using early acknowledgement (default), a prefetch multiplier of 1 means the worker will reserve at most one extra task for every active worker process.

When users ask if it's possible to disable "prefetching of tasks", often what they really want is to have a worker only reserve as many tasks as there are child processes.

But this is not possible without enabling late acknowledgements acknowledgements; A task that has been started, will be retried if the worker crashes mid execution so the task must be idempotent (see also notes at *Should I use retry or acks_late?*).

You can enable this behavior by using the following configuration options:

```
CELERY_ACKS_LATE = True
CELERYD_PREFETCH_MULTIPLIER = 1
```

Rate Limits

The system responsible for enforcing rate limits introduces some overhead, so if you're not using rate limits it may be a good idea to disable them completely. This will disable one thread, and it won't spend as many CPU cycles when the queue is inactive.

Set the CELERY_DISABLE_RATE_LIMITS setting to disable the rate limit subsystem:

```
CELERY_DISABLE_RATE_LIMITS = True
```

[3] RabbitMQ and other brokers deliver messages round-robin, so this doesn't apply to an active system. If there is no prefetch limit and you restart the cluster, there will be timing delays between nodes starting. If there are 3 offline nodes and one active node, all messages will be delivered to the active node.

[4] This is the concurrency setting; CELERYD_CONCURRENCY or the `-c` option to **celeryd**.

2.3.12 Concurrency

Release 3.0

Date July 10, 2014

Concurrency with Eventlet

Introduction

The Eventlet homepage describes it as; A concurrent networking library for Python that allows you to change how you run your code, not how you write it.

- It uses epoll(4) or libevent for highly scalable non-blocking I/O.
- Coroutines ensure that the developer uses a blocking style of programming that is similar to threading, but provide the benefits of non-blocking I/O.
- The event dispatch is implicit, which means you can easily use Eventlet from the Python interpreter, or as a small part of a larger application.

Celery supports Eventlet as an alternative execution pool implementation. It is in some cases superior to multiprocessing, but you need to ensure your tasks do not perform blocking calls, as this will halt all other operations in the worker until the blocking call returns.

The multiprocessing pool can take use of multiple processes, but how many is often limited to a few processes per CPU. With Eventlet you can efficiently spawn hundreds, or thousands of green threads. In an informal test with a feed hub system the Eventlet pool could fetch and process hundreds of feeds every second, while the multiprocessing pool spent 14 seconds processing 100 feeds. Note that is one of the applications evented I/O is especially good at (asynchronous HTTP requests). You may want a mix of both Eventlet and multiprocessing workers, and route tasks according to compatibility or what works best.

Enabling Eventlet

You can enable the Eventlet pool by using the `-P` option to **celery worker**:

```
$ celery worker -P eventlet -c 1000
```

Examples

See the Eventlet examples directory in the Celery distribution for some examples taking use of Eventlet support.

2.3.13 Signals

Signals allows decoupled applications to receive notifications when certain actions occur elsewhere in the application.

Celery ships with many signals that you application can hook into to augment behavior of certain actions.

Basics

Several kinds of events trigger signals, you can connect to these signals to perform actions as they trigger.

Example connecting to the `task_sent` signal:

```
from celery.signals import task_sent

@task_sent.connect
def task_sent_handler(sender=None, task_id=None, task=None, args=None,
                      kwargs=None, **kwds):
    print('Got signal task_sent for task id %s' % (task_id, ))
```

Some signals also have a sender which you can filter by. For example the `task_sent` signal uses the task name as a sender, so you can connect your handler to be called only when tasks with name *"tasks.add"* has been sent by providing the *sender* argument to `connect`:

```
@task_sent.connect(sender='tasks.add')
def task_sent_handler(sender=None, task_id=None, task=None, args=None,
                      kwargs=None, **kwds):
    print('Got signal task_sent for task id %s' % (task_id, ))
```

Signals

Task Signals

task_sent Dispatched when a task has been sent to the broker. Note that this is executed in the client process, the one sending the task, not in the worker.

Sender is the name of the task being sent.

Provides arguments:

- **task_id** Id of the task to be executed.
- **task** The task being executed.
- **args** the tasks positional arguments.
- **kwargs** The tasks keyword arguments.
- **eta** The time to execute the task.
- **taskset** Id of the taskset this task is part of (if any).

task_prerun Dispatched before a task is executed.

Sender is the task class being executed.

Provides arguments:

- **task_id** Id of the task to be executed.
- **task** The task being executed.
- **args** the tasks positional arguments.
- **kwargs** The tasks keyword arguments.

task_postrun Dispatched after a task has been executed.

Sender is the task class executed.

Provides arguments:

- **task_id** Id of the task to be executed.
- **task** The task being executed.
- **args** The tasks positional arguments.
- **kwargs** The tasks keyword arguments.
- **retval** The return value of the task.
- state

  Name of the resulting state.

task_success Dispatched when a task succeeds.

Sender is the task class executed.

Provides arguments

- **result** Return value of the task.

task_failure Dispatched when a task fails.

Sender is the task class executed.

Provides arguments:

- **task_id** Id of the task.
- **exception** Exception instance raised.
- **args** Positional arguments the task was called with.
- **kwargs** Keyword arguments the task was called with.
- **traceback** Stack trace object.
- **einfo** The `celery.datastructures.ExceptionInfo` instance.

task_revoked Dispatched when a task is revoked/terminated by the worker.

Sender is the task class revoked/terminated.

Provides arguments:

- **terminated** Set to `True` if the task was terminated.
- **signum** Signal number used to terminate the task. If this is `None` and terminated is `True` then `TERM` should be assumed.
- expired Set to `True` if the task expired.

Worker Signals

celeryd_after_setup This signal is sent after the worker instance is set up, but before it calls run. This means that any queues from the `-Q` option is enabled, logging has been set up and so on.

It can be used to e.g. add custom queues that should always be consumed from, disregarding the `-Q` option. Here's an example that sets up a direct queue for each worker, these queues can then be used to route a task to any specific worker:

```
from celery.signals import celeryd_after_setup

@celeryd_after_setup.connect
def setup_direct_queue(sender, instance, **kwargs):
    queue_name = '%s.dq' % sender   # sender is the hostname of the worker
    instance.app.amqp.queues.select_add(queue_name)
```

Provides arguments:

- sender Hostname of the worker.
- **instance** This is the `celery.apps.worker.Worker` instance to be initialized. Note that only the `app` and `hostname` attributes have been set so far, and the rest of `__init__` has not been executed.
- **conf** The configuration of the current app.

celeryd_init This is the first signal sent when **celeryd** starts up. The `sender` is the host name of the worker, so this signal can be used to setup worker specific configuration:

```
from celery.signals import celeryd_init

@celeryd_init.connect(sender='worker12.example.com')
def configure_worker12(conf=None, **kwargs):
    conf.CELERY_DEFAULT_RATE_LIMIT = '10/m'
```

or to set up configuration for multiple workers you can omit specifying a sender when you connect:

```
from celery.signals import celeryd_init

@celeryd_init.connect
def configure_workers(sender=None, conf=None, **kwargs):
    if sender in ('worker1.example.com', 'worker2.example.com'):
        conf.CELERY_DEFAULT_RATE_LIMIT = '10/m'
    if sender == 'worker3.example.com':
        conf.CELERYD_PREFETCH_MULTIPLIER = 0
```

Provides arguments:

- sender Hostname of the worker.
- **instance** This is the `celery.apps.worker.Worker` instance to be initialized. Note that only the `app` and `hostname` attributes have been set so far, and the rest of `__init__` has not been executed.
- **conf** The configuration of the current app.

worker_init Dispatched before the worker is started.

worker_ready Dispatched when the worker is ready to accept work.

worker_process_init Dispatched by each new pool worker process when it starts.

worker_shutdown Dispatched when the worker is about to shut down.

Celerybeat Signals

beat_init Dispatched when celerybeat starts (either standalone or embedded). Sender is the `celery.beat.Service` instance.

beat_embedded_init Dispatched in addition to the `beat_init` signal when celerybeat is started as an embedded process. Sender is the `celery.beat.Service` instance.

Eventlet Signals

eventlet_pool_started Sent when the eventlet pool has been started.

Sender is the `celery.concurrency.eventlet.TaskPool` instance.

eventlet_pool_preshutdown Sent when the worker shutdown, just before the eventlet pool is requested to wait for remaining workers.

Sender is the `celery.concurrency.eventlet.TaskPool` instance.

eventlet_pool_postshutdown Sent when the pool has been joined and the worker is ready to shutdown.

Sender is the `celery.concurrency.eventlet.TaskPool` instance.

eventlet_pool_apply Sent whenever a task is applied to the pool.

Sender is the `celery.concurrency.eventlet.TaskPool` instance.

Provides arguments:

- target

  The target function.

- args

  Positional arguments.

- kwargs

  Keyword arguments.

Logging Signals

setup_logging Celery won't configure the loggers if this signal is connected, so you can use this to completely override the logging configuration with your own.

If you would like to augment the logging configuration setup by Celery then you can use the `after_setup_logger` and `after_setup_task_logger` signals.

Provides arguments:

- **loglevel** The level of the logging object.
- **logfile** The name of the logfile.
- **format** The log format string.
- **colorize** Specify if log messages are colored or not.

after_setup_logger Sent after the setup of every global logger (not task loggers). Used to augment logging configuration.

Provides arguments:

- **logger** The logger object.
- **loglevel** The level of the logging object.
- **logfile** The name of the logfile.
- **format** The log format string.
- **colorize** Specify if log messages are colored or not.

after_setup_task_logger Sent after the setup of every single task logger. Used to augment logging configuration.

Provides arguments:

- **logger** The logger object.
- **loglevel** The level of the logging object.
- **logfile** The name of the logfile.

- **format** The log format string.
- **colorize** Specify if log messages are colored or not.

2.4 Configuration and defaults

This document describes the configuration options available.

If you're using the default loader, you must create the `celeryconfig.py` module and make sure it is available on the Python path.

2.4.1 Example configuration file

This is an example configuration file to get you started. It should contain all you need to run a basic Celery set-up.

```
## Broker settings.
BROKER_URL = "amqp://guest:guest@localhost:5672//"

# List of modules to import when celery starts.
CELERY_IMPORTS = ("myapp.tasks", )

## Using the database to store task state and results.
CELERY_RESULT_BACKEND = "database"
CELERY_RESULT_DBURI = "sqlite:///mydatabase.db"

CELERY_ANNOTATIONS = {"tasks.add": {"rate_limit": "10/s"}}
```

2.4.2 Configuration Directives

Time and date settings

CELERY_ENABLE_UTC

New in version 2.5.

If enabled dates and times in messages will be converted to use the UTC timezone.

Note that workers running Celery versions below 2.5 will assume a local timezone for all messages, so only enable if all workers have been upgraded.

Enabled by default since version 3.0.

CELERY_TIMEZONE

Configure Celery to use a custom time zone. The timezone value can be any time zone supported by the `pytz` library. `pytz` must be installed for the selected zone to be used.

If not set then the systems default local time zone is used.

> **Warning:** Celery requires the `pytz` library to be installed, when using custom time zones (other than UTC). You can install it using **pip** or **easy_install**:
>
> ```
> $ pip install pytz
> ```
>
> Pytz is a library that defines the timzones of the world, it changes quite frequently so it is not included in the Python Standard Library.

Task settings

CELERY_ANNOTATIONS

This setting can be used to rewrite any task attribute from the configuration. The setting can be a dict, or a list of annotation objects that filter for tasks and return a map of attributes to change.

This will change the `rate_limit` attribute for the `tasks.add` task:

```
CELERY_ANNOTATIONS = {"tasks.add": {"rate_limit": "10/s"}}
```

or change the same for all tasks:

```
CELERY_ANNOTATIONS = {"*": {"rate_limit": "10/s"}}
```

You can change methods too, for example the `on_failure` handler:

```
def my_on_failure(self, exc, task_id, args, kwargs, einfo):
    print("Oh no! Task failed: %r" % (exc, ))

CELERY_ANNOTATIONS = {"*": {"on_failure": my_on_failure}}
```

If you need more flexibility then you can use objects instead of a dict to choose which tasks to annotate:

```
class MyAnnotate(object):

    def annotate(self, task):
        if task.name.startswith("tasks."):
            return {"rate_limit": "10/s"}

CELERY_ANNOTATIONS = (MyAnnotate(), {...})
```

Concurrency settings

CELERYD_CONCURRENCY

The number of concurrent worker processes/threads/green threads executing tasks.

If you're doing mostly I/O you can have more processes, but if mostly CPU-bound, try to keep it close to the number of CPUs on your machine. If not set, the number of CPUs/cores on the host will be used.

Defaults to the number of available CPUs.

CELERYD_PREFETCH_MULTIPLIER

How many messages to prefetch at a time multiplied by the number of concurrent processes. The default is 4 (four messages for each process). The default setting is usually a good choice, however – if you have very long running tasks waiting in the queue and you have to start the workers, note that the first worker to start will receive four times the number of messages initially. Thus the tasks may not be fairly distributed to the workers.

Note: Tasks with ETA/countdown are not affected by prefetch limits.

Task result backend settings

CELERY_RESULT_BACKEND

Deprecated aliases `CELERY_BACKEND`

The backend used to store task results (tombstones). Disabled by default. Can be one of the following:

- **database** Use a relational database supported by SQLAlchemy. See *Database backend settings*.
- **cache** Use memcached to store the results. See *Cache backend settings*.
- **mongodb** Use MongoDB to store the results. See *MongoDB backend settings*.
- **redis** Use Redis to store the results. See *Redis backend settings*.
- **amqp** Send results back as AMQP messages See *AMQP backend settings*.
- **cassandra** Use Cassandra to store the results. See *Cassandra backend settings*.
- **ironcache** Use IronCache to store the results. See *IronCache backend settings*.

CELERY_RESULT_SERIALIZER

Result serialization format. Default is *"pickle"*. See *Serializers* for information about supported serialization formats.

Database backend settings

CELERY_RESULT_DBURI

Please see Supported Databases for a table of supported databases. To use this backend you need to configure it with an Connection String, some examples include:

```
# sqlite (filename)
CELERY_RESULT_DBURI = "sqlite:///celerydb.sqlite"

# mysql
CELERY_RESULT_DBURI = "mysql://scott:tiger@localhost/foo"

# postgresql
CELERY_RESULT_DBURI = "postgresql://scott:tiger@localhost/mydatabase"

# oracle
CELERY_RESULT_DBURI = "oracle://scott:tiger@127.0.0.1:1521/sidname"
```

See Connection String for more information about connection strings.

CELERY_RESULT_ENGINE_OPTIONS

To specify additional SQLAlchemy database engine options you can use the CELERY_RESULT_ENGINE_OPTIONS setting:

```
# echo enables verbose logging from SQLAlchemy.
CELERY_RESULT_ENGINE_OPTIONS = {"echo": True}
```

Short lived sessions are disabled by default. If enabled they can drastically reduce performance, especially on systems processing lots of tasks. This option is useful on low-traffic workers that experience errors as a result of cached database connections going stale through inactivity. For example, intermittent errors like *(OperationalError) (2006, 'MySQL server has gone away')* can be fixed by enabling short lived sessions. This option only affects the database backend.

Example configuration

```
CELERY_RESULT_BACKEND = "database"
CELERY_RESULT_DBURI = "mysql://user:password@host/dbname"
```

AMQP backend settings

Note: The AMQP backend requires RabbitMQ 1.1.0 or higher to automatically expire results. If you are running an older version of RabbitmQ you should disable result expiration like this:

CELERY_TASK_RESULT_EXPIRES = None

CELERY_RESULT_EXCHANGE

Name of the exchange to publish results in. Default is *"celeryresults"*.

CELERY_RESULT_EXCHANGE_TYPE

The exchange type of the result exchange. Default is to use a *direct* exchange.

CELERY_RESULT_PERSISTENT

If set to `True`, result messages will be persistent. This means the messages will not be lost after a broker restart. The default is for the results to be transient.

Example configuration

```
CELERY_RESULT_BACKEND = "amqp"
CELERY_TASK_RESULT_EXPIRES = 18000  # 5 hours.
```

Cache backend settings

Note: The cache backend supports the pylibmc and *python-memcached* libraries. The latter is used only if pylibmc is not installed.

CELERY_CACHE_BACKEND

Using a single memcached server:

```
CELERY_CACHE_BACKEND = 'memcached://127.0.0.1:11211/'
```

Using multiple memcached servers:

```
CELERY_RESULT_BACKEND = "cache"
CELERY_CACHE_BACKEND = 'memcached://172.19.26.240:11211;172.19.26.242:11211/'
```

The "memory" backend stores the cache in memory only:

    CELERY_CACHE_BACKEND = "memory"

CELERY_CACHE_BACKEND_OPTIONS

You can set pylibmc options using the `CELERY_CACHE_BACKEND_OPTIONS` setting:

```
CELERY_CACHE_BACKEND_OPTIONS = {"binary": True,
                                "behaviors": {"tcp_nodelay": True}}
```

Redis backend settings

Configuring the backend URL

Note: The Redis backend requires the `redis` library: http://pypi.python.org/pypi/redis/

To install the redis package use *pip* or *easy_install*:

```
$ pip install redis
```

This backend requires the CELERY_RESULT_BACKEND setting to be set to a Redis URL:

```
CELERY_RESULT_BACKEND = "redis://:password@host:port/db"
```

For example:

```
CELERY_RESULT_BACKEND = "redis://localhost/0"
```

which is the same as:

```
CELERY_RESULT_BACKEND = "redis://"
```

The fields of the URL is defined as folows:

- *host*

Host name or IP address of the Redis server. e.g. *"localhost"*.

- *port*

Port to the Redis server. Default is 6379.

- *db*

Database number to use. Default is 0. The db can include an optional leading slash.

- *password*

Password used to connect to the database.

CELERY_REDIS_MAX_CONNECTIONS

Maximum number of connections available in the Redis connection pool used for sending and retrieving results.

MongoDB backend settings

Note: The MongoDB backend requires the `pymongo` library: http://github.com/mongodb/mongo-python-driver/tree/master

CELERY_MONGODB_BACKEND_SETTINGS

This is a dict supporting the following keys:

- **host** Host name of the MongoDB server. Defaults to "localhost".
- **port** The port the MongoDB server is listening to. Defaults to 27017.
- **user** User name to authenticate to the MongoDB server as (optional).
- **password** Password to authenticate to the MongoDB server (optional).
- **database** The database name to connect to. Defaults to "celery".
- **taskmeta_collection** The collection name to store task meta data. Defaults to "celery_taskmeta".

- **max_pool_size** Passed as max_pool_size to PyMongo's Connection or MongoClient constructor. It is the maximum number of TCP connections to keep open to MongoDB at a given time. If there are more open connections than max_pool_size, sockets will be closed when they are released. Defaults to 10.
- options

  Additional keyword arguments to pass to the mongodb connection constructor. See the `pymongo` docs to see a list of arguments supported.

Example configuration

```
CELERY_RESULT_BACKEND = "mongodb"
CELERY_MONGODB_BACKEND_SETTINGS = {
    "host": "192.168.1.100",
    "port": 30000,
    "database": "mydb",
    "taskmeta_collection": "my_taskmeta_collection",
}
```

Cassandra backend settings

Note: The Cassandra backend requires the `pycassa` library: http://pypi.python.org/pypi/pycassa/

To install the pycassa package use *pip* or *easy_install*:

```
$ pip install pycassa
```

This backend requires the following configuration directives to be set.

CASSANDRA_SERVERS

List of `host:port` Cassandra servers. e.g. `["localhost:9160]"`.

CASSANDRA_KEYSPACE

The keyspace in which to store the results. e.g. `"tasks_keyspace"`.

CASSANDRA_COLUMN_FAMILY

The column family in which to store the results. eg `"tasks"`

CASSANDRA_READ_CONSISTENCY

The read consistency used. Values can be `"ONE"`, `"QUORUM"` or `"ALL"`.

CASSANDRA_WRITE_CONSISTENCY

The write consistency used. Values can be `"ONE"`, `"QUORUM"` or `"ALL"`.

CASSANDRA_DETAILED_MODE

Enable or disable detailed mode. Default is `False`. This mode allows to use the power of Cassandra wide columns to store all states for a task as a wide column, instead of only the last one.

To use this mode, you need to configure your ColumnFamily to use the `TimeUUID` type as a comparator:

```
create column family task_results with comparator = TimeUUIDType;
```

CASSANDRA_OPTIONS

Options to be passed to the pycassa connection pool (optional).

Example configuration

```
CASSANDRA_SERVERS = ["localhost:9160"]
CASSANDRA_KEYSPACE = "celery"
CASSANDRA_COLUMN_FAMILY = "task_results"
CASSANDRA_READ_CONSISTENCY = "ONE"
CASSANDRA_WRITE_CONSISTENCY = "ONE"
CASSANDRA_DETAILED_MODE = True
CASSANDRA_OPTIONS = {
    'timeout': 300,
    'max_retries': 10
}
```

IronCache backend settings

Note: The Cassandra backend requires the `iron_celery` library: http://pypi.python.org/pypi/iron_celery

To install the iron_celery package use *pip* or *easy_install*:

```
$ pip install iron_celery
```

IronCache is configured via the URL provided in `CELERY_RESULT_BACKEND`, for example:

```
CELERY_RESULT_BACKEND = 'ironcache://project_id:token@'
```

Or to change the cache name:

```
ironcache://project_id:token@/awesomecache
```

For more information, see: https://github.com/iron-io/iron_celery

Message Routing

CELERY_QUEUES

The mapping of queues the worker consumes from. This is a dictionary of queue name/options. See *Routing Tasks* for more information.

The default is a queue/exchange/binding key of *"celery"*, with exchange type *direct*.

You don't have to care about this unless you want custom routing facilities.

CELERY_ROUTES

A list of routers, or a single router used to route tasks to queues. When deciding the final destination of a task the routers are consulted in order. See *Routers* for more information.

CELERY_QUEUE_HA_POLICY

brokers RabbitMQ

This will set the default HA policy for a queue, and the value can either be a string (usually `all`):

```
CELERY_QUEUE_HA_POLICY = 'all'
```

Using 'all' will replicate the queue to all current nodes, Or you can give it a list of nodes to replicate to:

```
CELERY_QUEUE_HA_POLICY = ['rabbit@host1', 'rabbit@host2']
```

Using a list will implicitly set `x-ha-policy` to 'nodes' and `x-ha-policy-params` to the given list of nodes.

See http://www.rabbitmq.com/ha.html for more information.

CELERY_WORKER_DIRECT

This option enables so that every worker has a dedicated queue, so that tasks can be routed to specific workers.

The queue name for each worker is automatically generated based on the worker hostname and a `.dq` suffix, using the `C.dq` exchange.

For example the queue name for the worker with hostname `w1.example.com` becomes:

```
w1.example.com.dq
```

Then you can route the task to the task by specifying the hostname as the routung key and the `C.dq` exchange:

```
CELERY_ROUTES = {
    'tasks.add': {'exchange': 'C.dq', 'routing_key': 'w1.example.com'}
}
```

This setting is mandatory if you want to use the `move_to_worker` features of `celery.contrib.migrate`.

CELERY_CREATE_MISSING_QUEUES

If enabled (default), any queues specified that are not defined in `CELERY_QUEUES` will be automatically created. See *Automatic routing*.

CELERY_DEFAULT_QUEUE

The name of the default queue used by *.apply_async* if the message has no route or no custom queue has been specified.

This queue must be listed in `CELERY_QUEUES`. If `CELERY_QUEUES` is not specified then it is automatically created containing one queue entry, where this name is used as the name of that queue.

The default is: *celery*.

See also:

Changing the name of the default queue

CELERY_DEFAULT_EXCHANGE

Name of the default exchange to use when no custom exchange is specified for a key in the CELERY_QUEUES setting.

The default is: *celery*.

CELERY_DEFAULT_EXCHANGE_TYPE

Default exchange type used when no custom exchange type is specified for a key in the CELERY_QUEUES setting. The default is: *direct*.

CELERY_DEFAULT_ROUTING_KEY

The default routing key used when no custom routing key is specified for a key in the CELERY_QUEUES setting.

The default is: *celery*.

CELERY_DEFAULT_DELIVERY_MODE

Can be *transient* or *persistent*. The default is to send persistent messages.

Broker Settings

CELERY_ACCEPT_CONTENT

A whitelist of content-types/serializers to allow.

If a message is received that is not in this list then the message will be discarded with an error.

By default any content type is enabled (including pickle and yaml) so make sure untrusted parties do not have access to your broker. See *Security* for more.

Example:

```
# using serializer name
CELERY_ACCEPT_CONTENT = ['json']

# or the actual content-type (MIME)
CELERY_ACCEPT_CONTENT = ['application/json']
```

BROKER_TRANSPORT

Aliases BROKER_BACKEND

Deprecated aliases CARROT_BACKEND

BROKER_URL

Default broker URL. This must be an URL in the form of:

```
transport://userid:password@hostname:port/virtual_host
```

Only the scheme part (`transport://`) is required, the rest is optional, and defaults to the specific transports default values.

The transport part is the broker implementation to use, and the default is `amqp`, which uses `librabbitmq` by default or falls back to `pyamqp` if that is not installed. Also there are many other choices including `redis`, `beanstalk`, `sqlalchemy`, `django`, `mongodb`, `couchdb`. It can also be a fully qualified path to your own transport implementation.

See *URLs* in the Kombu documentation for more information.

BROKER_HEARTBEAT

transports supported `pyamqp`

It's not always possible to detect connection loss in a timely manner using TCP/IP alone, so AMQP defines something called heartbeats that's is used both by the client and the broker to detect if a connection was closed.

Hartbeats are disabled by default.

If the heartbeat value is 10 seconds, then the heartbeat will be monitored at the interval specified by the `BROKER_HEARTBEAT_CHECKRATE` setting, which by default is double the rate of the heartbeat value (so for the default 10 seconds, the heartbeat is checked every 5 seconds).

BROKER_HEARTBEAT_CHECKRATE

transports supported `pyamqp`

At intervals the worker will monitor that the broker has not missed too many heartbeats. The rate at which this is checked is calculated by dividing the `BROKER_HEARTBEAT` value with this value, so if the heartbeat is 10.0 and the rate is the default 2.0, the check will be performed every 5 seconds (twice the heartbeat sending rate).

BROKER_USE_SSL

Use SSL to connect to the broker. Off by default. This may not be supported by all transports.

BROKER_POOL_LIMIT

New in version 2.3.

The maximum number of connections that can be open in the connection pool.

The pool is enabled by default since version 2.5, with a default limit of ten connections. This number can be tweaked depending on the number of threads/greenthreads (eventlet/gevent) using a connection. For example running eventlet with 1000 greenlets that use a connection to the broker, contention can arise and you should consider increasing the limit.

If set to `None` or 0 the connection pool will be disabled and connections will be established and closed for every use.

Default (since 2.5) is to use a pool of 10 connections.

BROKER_CONNECTION_TIMEOUT

The default timeout in seconds before we give up establishing a connection to the AMQP server. Default is 4 seconds.

BROKER_CONNECTION_RETRY

Automatically try to re-establish the connection to the AMQP broker if lost.

The time between retries is increased for each retry, and is not exhausted before `BROKER_CONNECTION_MAX_RETRIES` is exceeded.

This behavior is on by default.

BROKER_CONNECTION_MAX_RETRIES

Maximum number of retries before we give up re-establishing a connection to the AMQP broker.

If this is set to `0` or `None`, we will retry forever.

Default is 100 retries.

BROKER_TRANSPORT_OPTIONS

New in version 2.2.

A dict of additional options passed to the underlying transport.

See your transport user manual for supported options (if any).

Example setting the visibility timeout (supported by Redis and SQS transports):

```
BROKER_TRANSPORT_OPTIONS = {'visibility_timeout': 18000}  # 5 hours
```

Task execution settings

CELERY_ALWAYS_EAGER

If this is `True`, all tasks will be executed locally by blocking until the task returns. `apply_async()` and `Task.delay()` will return an `EagerResult` instance, which emulates the API and behavior of `AsyncResult`, except the result is already evaluated.

That is, tasks will be executed locally instead of being sent to the queue.

CELERY_EAGER_PROPAGATES_EXCEPTIONS

If this is `True`, eagerly executed tasks (applied by *task.apply()*, or when the `CELERY_ALWAYS_EAGER` setting is enabled), will propagate exceptions.

It's the same as always running `apply()` with `throw=True`.

CELERY_IGNORE_RESULT

Whether to store the task return values or not (tombstones). If you still want to store errors, just not successful return values, you can set `CELERY_STORE_ERRORS_EVEN_IF_IGNORED`.

CELERY_MESSAGE_COMPRESSION

Default compression used for task messages. Can be `"gzip"`, `"bzip2"` (if available), or any custom compression schemes registered in the Kombu compression registry.

The default is to send uncompressed messages.

CELERY_TASK_RESULT_EXPIRES

Time (in seconds, or a `timedelta` object) for when after stored task tombstones will be deleted.

A built-in periodic task will delete the results after this time (`celery.task.backend_cleanup`).

A value of `None` or 0 means results will never expire (depending on backend specifications).

Default is to expire after 1 day.

Note: For the moment this only works with the amqp, database, cache, redis and MongoDB backends.

When using the database or MongoDB backends, *celerybeat* must be running for the results to be expired.

CELERY_MAX_CACHED_RESULTS

Result backends caches ready results used by the client.

This is the total number of results to cache before older results are evicted. The default is 5000.

CELERY_CHORD_PROPAGATES

New in version 3.0.14.

This setting defines what happens when a task part of a chord raises an exception:

- If propagate is True the chord callback will change state to FAILURE with the exception value set to a `ChordError` instance containing information about the error and the task that failed.

  This is the default behavior in Celery 3.1+

- If propagate is False the exception value will instead be forwarded to the chord callback.

  This was the default behavior before version 3.1.

CELERY_TRACK_STARTED

If `True` the task will report its status as "started" when the task is executed by a worker. The default value is `False` as the normal behaviour is to not report that level of granularity. Tasks are either pending, finished, or waiting to be retried. Having a "started" state can be useful for when there are long running tasks and there is a need to report which task is currently running.

CELERY_TASK_SERIALIZER

A string identifying the default serialization method to use. Can be *pickle* (default), *json*, *yaml*, *msgpack* or any custom serialization methods that have been registered with `kombu.serialization.registry`.

See also:

Serializers.

CELERY_TASK_PUBLISH_RETRY

New in version 2.2.

Decides if publishing task messages will be retried in the case of connection loss or other connection errors. See also `CELERY_TASK_PUBLISH_RETRY_POLICY`.

Enabled by default.

CELERY_TASK_PUBLISH_RETRY_POLICY

New in version 2.2.

Defines the default policy when retrying publishing a task message in the case of connection loss or other connection errors.

See *Message Sending Retry* for more information.

CELERY_DEFAULT_RATE_LIMIT

The global default rate limit for tasks.

This value is used for tasks that does not have a custom rate limit The default is no rate limit.

CELERY_DISABLE_RATE_LIMITS

Disable all rate limits, even if tasks has explicit rate limits set.

CELERY_ACKS_LATE

Late ack means the task messages will be acknowledged **after** the task has been executed, not *just before*, which is the default behavior.

See also:

FAQ: *Should I use retry or acks_late?*.

Worker: celeryd

CELERY_IMPORTS

A sequence of modules to import when the worker starts.

This is used to specify the task modules to import, but also to import signal handlers and additional remote control commands, etc.

The modules will be imported in the original order.

CELERY_INCLUDE

Exact same semantics as `CELERY_IMPORTS`, but can be used as a means to have different import categories.

The modules in this setting are imported after the modules in `CELERY_IMPORTS`.

CELERYD_FORCE_EXECV

On Unix the processes pool will fork, so that child processes start with the same memory as the parent process.

This can cause problems as there is a known deadlock condition with pthread locking primitives when *fork()* is combined with threads.

You should enable this setting if you are experiencing hangs (deadlocks), especially in combination with time limits or having a max tasks per child limit.

This option will be enabled by default in a later version.

This is not a problem on Windows, as it does not have *fork()*.

CELERYD_WORKER_LOST_WAIT

In some cases a worker may be killed without proper cleanup, and the worker may have published a result before terminating. This value specifies how long we wait for any missing results before raising a `WorkerLostError` exception.

Default is 10.0

CELERYD_MAX_TASKS_PER_CHILD

Maximum number of tasks a pool worker process can execute before it's replaced with a new one. Default is no limit.

CELERYD_TASK_TIME_LIMIT

Task hard time limit in seconds. The worker processing the task will be killed and replaced with a new one when this is exceeded.

CELERYD_TASK_SOFT_TIME_LIMIT

Task soft time limit in seconds.

The `SoftTimeLimitExceeded` exception will be raised when this is exceeded. The task can catch this to e.g. clean up before the hard time limit comes.

Example:

```
from celery.exceptions import SoftTimeLimitExceeded

@celery.task
def mytask():
    try:
        return do_work()
    except SoftTimeLimitExceeded:
        cleanup_in_a_hurry()
```

CELERY_STORE_ERRORS_EVEN_IF_IGNORED

If set, the worker stores all task errors in the result store even if `Task.ignore_result` is on.

CELERYD_STATE_DB

Name of the file used to stores persistent worker state (like revoked tasks). Can be a relative or absolute path, but be aware that the suffix *.db* may be appended to the file name (depending on Python version).

Can also be set via the *--statedb* argument to `celeryd`.

Not enabled by default.

CELERYD_TIMER_PRECISION

Set the maximum time in seconds that the ETA scheduler can sleep between rechecking the schedule. Default is 1 second.

Setting this value to 1 second means the schedulers precision will be 1 second. If you need near millisecond precision you can set this to 0.1.

Error E-Mails

CELERY_SEND_TASK_ERROR_EMAILS

The default value for the *Task.send_error_emails* attribute, which if set to `True` means errors occurring during task execution will be sent to `ADMINS` by email.

Disabled by default.

ADMINS

List of *(name, email_address)* tuples for the administrators that should receive error emails.

SERVER_EMAIL

The email address this worker sends emails from. Default is celery@localhost.

EMAIL_HOST

The mail server to use. Default is *"localhost"*.

EMAIL_HOST_USER

User name (if required) to log on to the mail server with.

EMAIL_HOST_PASSWORD

Password (if required) to log on to the mail server with.

EMAIL_PORT

The port the mail server is listening on. Default is *25*.

EMAIL_USE_SSL

Use SSL when connecting to the SMTP server. Disabled by default.

EMAIL_USE_TLS

Use TLS when connecting to the SMTP server. Disabled by default.

EMAIL_TIMEOUT

Timeout in seconds for when we give up trying to connect to the SMTP server when sending emails.

The default is 2 seconds.

Example E-Mail configuration

This configuration enables the sending of error emails to george@vandelay.com and kramer@vandelay.com:

```
# Enables error emails.
CELERY_SEND_TASK_ERROR_EMAILS = True

# Name and email addresses of recipients
ADMINS = (
    ("George Costanza", "george@vandelay.com"),
    ("Cosmo Kramer", "kosmo@vandelay.com"),
)

# Email address used as sender (From field).
SERVER_EMAIL = "no-reply@vandelay.com"

# Mailserver configuration
EMAIL_HOST = "mail.vandelay.com"
EMAIL_PORT = 25
# EMAIL_HOST_USER = "servers"
# EMAIL_HOST_PASSWORD = "s3cr3t"
```

Events

CELERY_SEND_EVENTS

Send events so the worker can be monitored by tools like *celerymon*.

CELERY_SEND_TASK_SENT_EVENT

New in version 2.2.

If enabled, a `task-sent` event will be sent for every task so tasks can be tracked before they are consumed by a worker.

Disabled by default.

CELERY_EVENT_SERIALIZER

Message serialization format used when sending event messages. Default is *"json"*. See *Serializers*.

Broadcast Commands

CELERY_BROADCAST_QUEUE

Name prefix for the queue used when listening for broadcast messages. The workers host name will be appended to the prefix to create the final queue name.

Default is *"celeryctl"*.

CELERY_BROADCAST_EXCHANGE

Name of the exchange used for broadcast messages.

Default is *"celeryctl"*.

CELERY_BROADCAST_EXCHANGE_TYPE

Exchange type used for broadcast messages. Default is *"fanout"*.

Logging

CELERYD_HIJACK_ROOT_LOGGER

New in version 2.2.

By default any previously configured logging options will be reset, because the Celery programs "hijacks" the root logger.

If you want to customize your own logging then you can disable this behavior.

Note: Logging can also be customized by connecting to the `celery.signals.setup_logging` signal.

CELERYD_LOG_COLOR

Enables/disables colors in logging output by the Celery apps.

By default colors are enabled if

1. the app is logging to a real terminal, and not a file.
2. the app is not running on Windows.

CELERYD_LOG_FORMAT

The format to use for log messages.

Default is *[%(asctime)s: %(levelname)s/%(processName)s] %(message)s*

See the Python `logging` module for more information about log formats.

CELERYD_TASK_LOG_FORMAT

The format to use for log messages logged in tasks. Can be overridden using the `--loglevel` option to `celeryd`.

Default is:

```
[%(asctime)s: %(levelname)s/%(processName)s]
    [%(task_name)s(%(task_id)s)] %(message)s
```

See the Python `logging` module for more information about log formats.

CELERY_REDIRECT_STDOUTS

If enabled *stdout* and *stderr* will be redirected to the current logger.

Enabled by default. Used by **celeryd** and **celerybeat**.

CELERY_REDIRECT_STDOUTS_LEVEL

The log level output to *stdout* and *stderr* is logged as. Can be one of `DEBUG`, `INFO`, `WARNING`, `ERROR` or `CRITICAL`.

Default is `WARNING`.

Security

CELERY_SECURITY_KEY

New in version 2.5.

The relative or absolute path to a file containing the private key used to sign messages when *Message Signing* is used.

CELERY_SECURITY_CERTIFICATE

New in version 2.5.

The relative or absolute path to an X.509 certificate file used to sign messages when *Message Signing* is used.

CELERY_SECURITY_CERT_STORE

New in version 2.5.

The directory containing X.509 certificates used for *Message Signing*. Can be a glob with wildcards, (for example `/etc/certs/*.pem`).

Custom Component Classes (advanced)

CELERYD_BOOT_STEPS

This setting enables you to add additional components to the worker process. It should be a list of module names with `celery.abstract.Component` classes, that augments functionality in the worker.

CELERYD_POOL

Name of the pool class used by the worker.

You can use a custom pool class name, or select one of the built-in aliases: `processes`, `eventlet`, `gevent`.

Default is `processes`.

CELERYD_POOL_RESTARTS

If enabled the worker pool can be restarted using the `pool_restart` remote control command.

Disabled by default.

CELERYD_AUTOSCALER

New in version 2.2.

Name of the autoscaler class to use.

Default is `"celery.worker.autoscale.Autoscaler"`.

CELERYD_AUTORELOADER

Name of the autoreloader class used by the worker to reload Python modules and files that have changed.

Default is: `"celery.worker.autoreload.Autoreloader"`.

CELERYD_CONSUMER

Name of the consumer class used by the worker. Default is `celery.worker.consumer.Consumer`

CELERYD_MEDIATOR

Name of the mediator class used by the worker. Default is `celery.worker.controllers.Mediator`.

CELERYD_TIMER

Name of the ETA scheduler class used by the worker. Default is `celery.utils.timer2.Timer`, or one overrided by the pool implementation.

Periodic Task Server: celerybeat

CELERYBEAT_SCHEDULE

The periodic task schedule used by `celerybeat`. See *Entries*.

CELERYBEAT_SCHEDULER

The default scheduler class. Default is *"celery.beat.PersistentScheduler"*.

Can also be set via the `-S` argument to `celerybeat`.

CELERYBEAT_SCHEDULE_FILENAME

Name of the file used by *PersistentScheduler* to store the last run times of periodic tasks. Can be a relative or absolute path, but be aware that the suffix *.db* may be appended to the file name (depending on Python version).

Can also be set via the `--schedule` argument to `celerybeat`.

CELERYBEAT_MAX_LOOP_INTERVAL

The maximum number of seconds `celerybeat` can sleep between checking the schedule.

The default for this value is scheduler specific. For the default celerybeat scheduler the value is 300 (5 minutes), but for e.g. the django-celery database scheduler it is 5 seconds because the schedule may be changed externally, and so it must take changes to the schedule into account.

Also when running celerybeat embedded (`-B`) on Jython as a thread the max interval is overridden and set to 1 so that it's possible to shut down in a timely manner.

Monitor Server: celerymon

CELERYMON_LOG_FORMAT

The format to use for log messages.

Default is *[%(asctime)s: %(levelname)s/%(processName)s] %(message)s*

See the Python `logging` module for more information about log formats.

2.5 Django

Release 3.0

Date July 10, 2014

2.5.1 First steps with Django

Configuring your Django project to use Celery

You need four simple steps to use celery with your Django project.

1. Install the `django-celery` library:

   ```
   $ pip install django-celery
   ```

2. Add the following lines to `settings.py`:

   ```
   import djcelery
   djcelery.setup_loader()
   ```

3. Add `djcelery` to `INSTALLED_APPS`.

4. Create the celery database tables.

   If you are using south for schema migrations, you'll want to:

   ```
   $ python manage.py migrate djcelery
   ```

   For those who are not using south, a normal `syncdb` will work:

   ```
   $ python manage.py syncdb
   ```

By default Celery uses RabbitMQ as the broker, but there are several alternatives to choose from, see *Choosing a Broker*.

All settings mentioned in the Celery documentation should be added to your Django project's `settings.py` module. For example you can configure the `BROKER_URL` setting to specify what broker to use:

```
BROKER_URL = 'amqp://guest:guest@localhost:5672/'
```

That's it.

Special note for mod_wsgi users

If you're using `mod_wsgi` to deploy your Django application you need to include the following in your `.wsgi` module:

```
import djcelery
djcelery.setup_loader()
```

Defining and calling tasks

Tasks are defined by wrapping functions in the `@task` decorator. It is a common practice to put these in their own module named `tasks.py`, and the worker will automatically go through the apps in `INSTALLED_APPS` to import these modules.

For a simple demonstration create a new Django app called `celerytest`. To create this app you need to be in the directory of your Django project where `manage.py` is located and execute:

```
$ python manage.py startapp celerytest
```

Next you have to add the new app to `INSTALLED_APPS` so that your Django project recognizes it. This setting is a tuple/list so just append `celerytest` as a new element at the end

```
INSTALLED_APPS = (
    ...,
    'djcelery',
    'celerytest',
)
```

After the new app has been created and added to `INSTALLED_APPS`, you can define your tasks by creating a new file called `celerytest/tasks.py`:

```
from celery import task

@task()
def add(x, y):
    return x + y
```

Our example task is pretty pointless, it just returns the sum of two arguments, but it will do for demonstration, and it is referred to in many parts of the Celery documentation.

Relative Imports

You have to be consistent in how you import the task module, e.g. if you have `project.app` in `INSTALLED_APPS` then you also need to import the tasks `from project.app` or else the names of the tasks will be different.

See *Automatic naming and relative imports*

Starting the worker process

In a production environment you will want to run the worker in the background as a daemon - see *Running the worker as a daemon* - but for testing and development it is useful to be able to start a worker instance by using the `celery worker` manage command, much as you would use Django's runserver:

```
$ python manage.py celery worker --loglevel=info
```

For a complete listing of the command line options available, use the help command:

```
$ python manage.py celery help
```

Calling our task

Now that the worker is running, open up a new python interactive shell with `python manage.py shell` to actually call the task you defined:

```
>>> from celerytest.tasks import add

>>> add.delay(2, 2)
```

Note that if you open a regular python shell by simply running `python` you will need to import your Django application's settings by running:

```
# Replace 'myproject' with your project's name
>>> from myproject import settings
```

The `delay` method used above is a handy shortcut to the `apply_async` method which enables you to have greater control of the task execution. To read more about calling tasks, including specifying the time at which the task should be processed see *Calling Tasks*.

Note: Tasks need to be stored in a real module, they can't be defined in the python shell or IPython/bpython. This is because the worker server must be able to import the task function.

The task should now be processed by the worker you started earlier, and you can verify that by looking at the worker's console output.

Calling a task returns an `AsyncResult` instance, which can be used to check the state of the task, wait for the task to finish or get its return value (or if the task failed, the exception and traceback).

By default django-celery stores this state in the Django database. You may consider choosing an alternate result backend or disabling states alltogether (see *Result Backends*).

To demonstrate how the results work call the task again, but this time keep the result instance returned:

```
>>> result = add.delay(4, 4)
>>> result.ready() # returns True if the task has finished processing.
False
>>> result.result # task is not ready, so no return value yet.
None
>>> result.get()   # Waits until the task is done and returns the retval.
8
>>> result.result # direct access to result, doesn't re-raise errors.
8
>>> result.successful() # returns True if the task didn't end in failure.
True
```

If the task raises an exception, the return value of `result.successful()` will be `False`, and `result.result` will contain the exception instance raised by the task.

Where to go from here

To learn more you should read the Celery User Guide, and the Celery Documentation in general.

2.5.2 Unit Testing

Testing with Django

The first problem you'll run in to when trying to write a test that runs a task is that Django's test runner doesn't use the same database as your celery daemon is using. If you're using the database backend, this means that your tombstones won't show up in your test database and you won't be able to get the return value or check the status of your tasks.

There are two ways to get around this. You can either take advantage of `CELERY_ALWAYS_EAGER = True` to skip the daemon, or you can avoid testing anything that needs to check the status or result of a task.

Using a custom test runner to test with celery

If you're going the `CELERY_ALWAYS_EAGER` route, which is probably better than just never testing some parts of your app, a custom Django test runner does the trick. Celery provides a simple test runner, but it's easy enough to roll your own if you have other things that need to be done. http://docs.djangoproject.com/en/dev/topics/testing/#defining-a-test-runner

For this example, we'll use the `djcelery.contrib.test_runner` to test the `add` task from the *Tasks* examples in the Celery documentation.

To enable the test runner, set the following settings:

```
TEST_RUNNER = 'djcelery.contrib.test_runner.CeleryTestSuiteRunner'
```

Then we can put the tests in a `tests.py` somewhere:

```
from django.test import TestCase
from myapp.tasks import add

class AddTestCase(TestCase):

    def testNoError(self):
        """Test that the ``add`` task runs with no errors,
        and returns the correct result."""
        result = add.delay(8, 8)

        self.assertEquals(result.get(), 16)
        self.assertTrue(result.successful())
```

This test assumes that you put your example `add` task in `maypp.tasks` so adjust the import for wherever you put the class.

2.6 Contributing

2.6.1 Community Code of Conduct

The goal is to maintain a diverse community that is pleasant for everyone. That is why we would greatly appreciate it if everyone contributing to and interacting with the community also followed this Code of Conduct.

The Code of Conduct covers our behavior as members of the community, in any forum, mailing list, wiki, website, Internet relay chat (IRC), public meeting or private correspondence.

The Code of Conduct is heavily based on the Ubuntu Code of Conduct, and the Pylons Code of Conduct.

Be considerate.

Your work will be used by other people, and you in turn will depend on the work of others. Any decision you take will affect users and colleagues, and we expect you to take those consequences into account when making decisions. Even if it's not obvious at the time, our contributions to Ubuntu will impact the work of others. For example, changes to code, infrastructure, policy, documentation and translations during a release may negatively impact others work.

Be respectful.

The Celery community and its members treat one another with respect. Everyone can make a valuable contribution to Celery. We may not always agree, but disagreement is no excuse for poor behavior and poor manners. We might all experience some frustration now and then, but we cannot allow that frustration to turn into a personal attack. It's important to remember that a community where people feel uncomfortable or threatened is not a productive one. We expect members of the Celery community to be respectful when dealing with other contributors as well as with people outside the Celery project and with users of Celery.

Be collaborative.

Collaboration is central to Celery and to the larger free software community. We should always be open to collaboration. Your work should be done transparently and patches from Celery should be given back to the community when they are made, not just when the distribution releases. If you wish to work on new code for existing upstream projects, at least keep those projects informed of your ideas and progress. It many not be possible to get consensus from upstream, or even from your colleagues about the correct implementation for an idea, so don't feel obliged to have that agreement before you begin, but at least keep the outside world informed of your work, and publish your work in a way that allows outsiders to test, discuss and contribute to your efforts.

When you disagree, consult others.

Disagreements, both political and technical, happen all the time and the Celery community is no exception. It is important that we resolve disagreements and differing views constructively and with the help of the community and community process. If you really want to go a different way, then we encourage you to make a derivative distribution or alternate set of packages that still build on the work we've done to utilize as common of a core as possible.

When you are unsure, ask for help.

Nobody knows everything, and nobody is expected to be perfect. Asking questions avoids many problems down the road, and so questions are encouraged. Those who are asked questions should be responsive and helpful. However, when asking a question, care must be taken to do so in an appropriate forum.

Step down considerately.

Developers on every project come and go and Celery is no different. When you leave or disengage from the project, in whole or in part, we ask that you do so in a way that minimizes disruption to the project. This means you should tell people you are leaving and take the proper steps to ensure that others can pick up where you leave off.

2.6.2 Reporting a Bug

Bugs can always be described to the *Mailing list*, but the best way to report an issue and to ensure a timely response is to use the issue tracker.

1. Create a GitHub account.

You need to create a GitHub account to be able to create new issues and participate in the discussion.

2. Determine if your bug is really a bug.

You should not file a bug if you are requesting support. For that you can use the *Mailing list*, or *IRC*.

3. Make sure your bug hasn't already been reported.

Search through the appropriate Issue tracker. If a bug like yours was found, check if you have new information that could be reported to help the developers fix the bug.

4. Collect information about the bug.

To have the best chance of having a bug fixed, we need to be able to easily reproduce the conditions that caused it. Most of the time this information will be from a Python traceback message, though some bugs might be in design, spelling or other errors on the website/docs/code.

If the error is from a Python traceback, include it in the bug report.

We also need to know what platform you're running (Windows, OSX, Linux, etc), the version of your Python interpreter, and the version of Celery, and related packages that you were running when the bug occurred.

5. Submit the bug.

By default GitHub will email you to let you know when new comments have been made on your bug. In the event you've turned this feature off, you should check back on occasion to ensure you don't miss any questions a developer trying to fix the bug might ask.

Issue Trackers

Bugs for a package in the Celery ecosystem should be reported to the relevant issue tracker.

- Celery: http://github.com/celery/celery/issues/
- Django-Celery: http://github.com/celery/django-celery/issues
- Celery-Pylons: http://bitbucket.org/ianschenck/celery-pylons/issues
- Kombu: http://github.com/celery/kombu/issues

If you are unsure of the origin of the bug you can ask the *Mailing list*, or just use the Celery issue tracker.

2.6.3 Contributors guide to the codebase

There's a seperate section for internal details, including details about the codebase and a style guide.

Read *Contributors Guide to the Code* for more!

2.6.4 Versions

Version numbers consists of a major version, minor version and a release number. Since version 2.1.0 we use the versioning semantics described by semver: http://semver.org.

Stable releases are published at PyPI while development releases are only available in the GitHub git repository as tags. All version tags starts with "v", so version 0.8.0 is the tag v0.8.0.

2.6.5 Branches

Current active version branches:

- master (http://github.com/celery/celery/tree/master)
- 3.0-devel (http://github.com/celery/celery/tree/3.0-devel)

You can see the state of any branch by looking at the Changelog:

https://github.com/celery/celery/blob/master/Changelog

If the branch is in active development the topmost version info should contain metadata like:

```
2.4.0
======
:release-date: TBA
:status: DEVELOPMENT
:branch: master
```

The `status` field can be one of:

- `PLANNING`

  The branch is currently experimental and in the planning stage.

- `DEVELOPMENT`

  The branch is in active development, but the test suite should be passing and the product should be working and possible for users to test.

- `FROZEN`

  The branch is frozen, and no more features will be accepted. When a branch is frozen the focus is on testing the version as much as possible before it is released.

`master` branch

The master branch is where development of the next version happens.

Maintenance branches

Maintenance branches are named after the version, e.g. the maintenance branch for the 2.2.x series is named `2.2`. Previously these were named `releaseXX-maint`.

The versions we currently maintain is:

- 2.3

  This is the current series.

- 2.2

  This is the previous series, and the last version to support Python 2.4.

- 2.1

  This is the last version to use the `carrot` AMQP framework. Recent versions use `kombu`.

Archived branches

Archived branches are kept for preserving history only, and theoretically someone could provide patches for these if they depend on a series that is no longer officially supported.

An archived version is named `X.Y-archived`.

Our currently archived branches are:

- 2.1-archived
- 2.0-archived
- 1.0-archived

Feature branches

Major new features are worked on in dedicated branches. There is no strict naming requirement for these branches.

Feature branches are removed once they have been merged into a release branch.

2.6.6 Tags

Tags are used exclusively for tagging releases. A release tag is named with the format `vX.Y.Z`, e.g. `v2.3.1`. Experimental releases contain an additional identifier `vX.Y.Z-id`, e.g. `v3.0.0-rc1`. Experimental tags may be removed after the official release.

2.6.7 Working on Features & Patches

Note: Contributing to Celery should be as simple as possible, so none of these steps should be considered mandatory.

You can even send in patches by email if that is your preferred work method. We won't like you any less, any contribution you make is always appreciated!

However following these steps may make maintainers life easier, and may mean that your changes will be accepted sooner.

Forking and setting up the repository

First you need to fork the Celery repository, a good introduction to this is in the Github Guide: Fork a Repo.

After you have cloned the repository you should checkout your copy to a directory on your machine:

```
$ git clone git@github.com:username/celery.git
```

When the repository is cloned enter the directory to set up easy access to upstream changes:

```
$ cd celery

$ git remote add upstream git://github.com/celery/celery.git

$ git fetch upstream
```

If you need to pull in new changes from upstream you should always use the *--rebase* option to `git pull`:

```
git pull --rebase upstream master
```

With this option you don't clutter the history with merging commit notes. See Rebasing merge commits in git. If you want to learn more about rebasing see the Rebase section in the Github guides.

If you need to work on a different branch than `master` you can fetch and checkout a remote branch like this:

```
git checkout --track -b 3.0-devel origin/3.0-devel
```

For a list of branches see *Branches*.

Running the unit test suite

To run the Celery test suite you need to install a few dependencies. A complete list of the dependencies needed are located in `requirements/test.txt`.

Installing the test requirements:

```
$ pip -E $VIRTUAL_ENV install -U -r requirements/test.txt
```

When installation of dependencies is complete you can execute the test suite by calling `nosetests`:

```
$ nosetests
```

Some useful options to **nosetests** are:

- *-x*

  Stop running the tests at the first test that fails.

- *-s*

  Don't capture output

- *--nologcapture*

  Don't capture log output.

- *-v*

  Run with verbose output.

If you want to run the tests for a single test file only you can do so like this:

```
$ nosetests celery.tests.test_worker.test_worker_job
```

Creating pull requests

When your feature/bugfix is complete you may want to submit a pull requests so that it can be reviewed by the maintainers.

Creating pull requests is easy, and also let you track the progress of your contribution. Read the Pull Requests section in the Github Guide to learn how this is done.

You can also attach pull requests to existing issues by following the steps outlined here: http://bit.ly/koJoso

Calculating test coverage

Code coverage in HTML:

```
$ nosetests --with-coverage3 --cover3-html
```

The coverage output will then be located at `celery/tests/cover/index.html`.

Code coverage in XML (Cobertura-style):

```
$ nosetests --with-coverage3 --cover3-xml --cover3-xml-file=coverage.xml
```

The coverage XML output will then be located at `coverage.xml`

Running the tests on all supported Python versions

There is a `tox` configuration file in the top directory of the distribution.

To run the tests for all supported Python versions simply execute:

```
$ tox
```

If you only want to test specific Python versions use the `-e` option:

```
$ tox -e py25,py26
```

Building the documentation

To build the documentation you need to install the dependencies listed in `requirements/docs.txt`:

```
$ pip -E $VIRTUAL_ENV install -U -r requirements/docs.txt
```

After these dependencies are installed you should be able to build the docs by running:

```
$ cd docs
$ rm -rf .build
$ make html
```

Make sure there are no errors or warnings in the build output. After building succeeds the documentation is available at `.build/html`.

Verifying your contribution

To use these tools you need to install a few dependencies. These dependencies can be found in `requirements/pkgutils.txt`.

Installing the dependencies:

```
$ pip -E $VIRTUAL_ENV install -U -r requirements/pkgutils.txt
```

pyflakes & PEP8

To ensure that your changes conform to PEP8 and to run pyflakes execute:

```
$ paver flake8
```

To not return a negative exit code when this command fails use the *-E* option, this can be convenient while developing:

```
$ paver flake8 -E
```

API reference

To make sure that all modules have a corresponding section in the API reference please execute:

```
$ paver autodoc
$ paver verifyindex
```

If files are missing you can add them by copying an existing reference file.

If the module is internal it should be part of the internal reference located in `docs/internals/reference/`. If the module is public it should be located in `docs/reference/`.

For example if reference is missing for the module `celery.worker.awesome` and this module is considered part of the public API, use the following steps:

```
$ cd docs/reference/
$ cp celery.schedules.rst celery.worker.awesome.rst

$ vim celery.worker.awesome.rst

    # change every occurance of ``celery.schedules`` to
    # ``celery.worker.awesome``

$ vim index.rst

    # Add ``celery.worker.awesome`` to the index.

# Add the file to git
$ git add celery.worker.awesome.rst
$ git add index.rst
$ git commit celery.worker.awesome.rst index.rst \
    -m "Adds reference for celery.worker.awesome"
```

2.6.8 Coding Style

You should probably be able to pick up the coding style from surrounding code, but it is a good idea to be aware of the following conventions.

- All Python code must follow the PEP-8 guidelines.

pep8.py is an utility you can use to verify that your code is following the conventions.

- Docstrings must follow the PEP-257 conventions, and use the following style.

  Do this:

```
def method(self, arg):
    """Short description.

    More details.

    """
```

or:

```
def method(self, arg):
    """Short description."""
```

but not this:

```
def method(self, arg):
    """
    Short description.
    """
```

- Lines should not exceed 78 columns.

  You can enforce this in **vim** by setting the `textwidth` option:

  ```
  set textwidth=78
  ```

  If adhering to this limit makes the code less readable, you have one more character to go on, which means 78 is a soft limit, and 79 is the hard limit :)

- Import order

  - Python standard library (*import xxx*)
  - Python standard library ('from xxx import')
  - Third party packages.
  - Other modules from the current package.

  or in case of code using Django:

  - Python standard library (*import xxx*)
  - Python standard library ('from xxx import')
  - Third party packages.
  - Django packages.
  - Other modules from the current package.

  Within these sections the imports should be sorted by module name.

  Example:

  ```
  import threading
  import time

  from collections import deque
  from Queue import Queue, Empty

  from .datastructures import TokenBucket
  from .utils import timeutils
  from .utils.compat import all, izip_longest, chain_from_iterable
  ```

- Wildcard imports must not be used (*from xxx import **).
- For distributions where Python 2.5 is the oldest support version additional rules apply:
  - Absolute imports must be enabled at the top of every module:

    ```
    from __future__ import absolute_import
    ```

  - If the module uses the with statement it must also enable that:

    ```
    from __future__ import with_statement
    ```

  - Every future import must be on its own line, as older Python 2.5 releases did not support importing multiple features on the same future import line:

    ```
    # Good
    from __future__ import absolute_import
    from __future__ import with_statement

    # Bad
    from __future__ import absolute_import, with_statement
    ```

    (Note that this rule does not apply if the package does not include support for Python 2.5)
- Note that we use "new-style' relative imports when the distribution does not support Python versions below 2.5

```
from . import submodule
```

2.6.9 Contacts

This is a list of people that can be contacted for questions regarding the official git repositories, PyPI packages Read the Docs pages.

If the issue is not an emergency then it is better to *report an issue*.

Committers

Ask Solem

github https://github.com/ask

twitter http://twitter.com/#!/asksol

Mher Movsisyan

github https://github.com/mher

Steeve Morin

github https://github.com/steeve

twitter http://twitter.com/#!/steeve

Website

The Celery Project website is run and maintained by

Mauro Rocco

github https://github.com/fireantology

twitter https://twitter.com/#!/fireantology

with design by:

Jan Henrik Helmers

web http://www.helmersworks.com

twitter http://twitter.com/#!/helmers

2.6.10 Packages

celery

git https://github.com/celery/celery

CI http://travis-ci.org/#!/celery/celery

PyPI http://pypi.python.org/pypi/celery

docs http://docs.celeryproject.org

kombu

Messaging framework.

git https://github.com/celery/kombu

CI http://travis-ci.org/#!/celery/kombu

PyPI http://pypi.python.org/pypi/kombu

docs http://kombu.readthedocs.org

billiard

Fork of multiprocessing containing improvements that will eventually be merged into the Python stdlib.

git https://github.com/celery/billiard

PyPI http://pypi.python.org/pypi/billiard

librabbitmq

Very fast Python AMQP client written in C.

git https://github.com/celery/librabbitmq

PyPI http://pypi.python.org/pypi/librabbitmq

celerymon

Celery monitor web-service.

git https://github.com/celery/celerymon

PyPI http://pypi.python.org/pypi/celerymon

django-celery

Django <-> Celery Integration.

git https://github.com/celery/django-celery

PyPI http://pypi.python.org/pypi/django-celery

docs http://docs.celeryproject.org/en/latest/django

cl

Actor framework.

git https://github.com/celery/cl

PyPI http://pypi.python.org/pypi/cl

cyme

Distributed Celery Instance manager.

git https://github.com/celery/cyme

PyPI http://pypi.python.org/pypi/cyme

docs http://cyme.readthedocs.org/

Deprecated

- Flask-Celery

  git https://github.com/ask/Flask-Celery

  PyPI http://pypi.python.org/pypi/Flask-Celery

- carrot

  git https://github.com/ask/carrot

  PyPI http://pypi.python.org/pypi/carrot

- ghettoq

git https://github.com/ask/ghettoq

PyPI http://pypi.python.org/pypi/ghettoq

- kombu-sqlalchemy

  git https://github.com/ask/kombu-sqlalchemy

  PyPI http://pypi.python.org/pypi/kombu-sqlalchemy

- django-kombu

  git https://github.com/ask/django-kombu

  PyPI http://pypi.python.org/pypi/django-kombu

- pylibrabbitmq

Old name for `librabbitmq`.

git `None`

PyPI http://pypi.python.org/pypi/pylibrabbitmq

2.6.11 Release Procedure

Updating the version number

The version number must be updated two places:

- `celery/__init__.py`
- `docs/include/introduction.txt`

After you have changed these files you must render the `README` files. There is a script to convert sphinx syntax to generic reStructured Text syntax, and the paver task *readme* does this for you:

```
$ paver readme
```

Now commit the changes:

```
$ git commit -a -m "Bumps version to X.Y.Z"
```

and make a new version tag:

```
$ git tag vX.Y.Z
$ git push --tags
```

Releasing

Commands to make a new public stable release:

```
$ paver releaseok  # checks pep8, autodoc index, runs tests and more
$ paver removepyc  # Remove .pyc files
$ git clean -xdn   # Check that there's no left-over files in the repo
$ python setup.py sdist upload  # Upload package to PyPI
```

If this is a new release series then you also need to do the following:

- **Go to the Read The Docs management interface at:** http://readthedocs.org/projects/celery/?fromdocs=celery
- Enter "Edit project"

Change default branch to the branch of this series, e.g. `2.4` for series 2.4.

- Also add the previous version under the "versions" tab.

Updating bundles

First you need to make sure the bundle entrypoints have been installed, but either running *develop*, or *install*:

```
$ python setup.py develop
```

Then make sure that you have your PyPI credentials stored in `~/.pypirc`, and execute the command:

```
$ python setup.py upload_bundles
```

If you broke something and need to update new versions of the bundles, then you can use `upload_bundles_fix`.

2.7 Community Resources

This is a list of external blog posts, tutorials and slides related to Celery. If you have a link that's missing from this list, please contact the mailing-list or submit a patch.

2.7.1 Resources

Who's using Celery

http://wiki.github.com/celery/celery/using

Wiki

http://wiki.github.com/celery/celery/

Celery questions on Stack Overflow

http://stackoverflow.com/search?q=celery&tab=newest

Mailing-list Archive: celery-users

http://blog.gmane.org/gmane.comp.python.amqp.celery.user

2.7.2 News

This section has moved to the Celery homepage: http://celeryproject.org/community/

2.8 Tutorials

Release 3.0

Date July 10, 2014

2.8.1 Running the worker as a daemon

Celery does not daemonize itself, please use one of the following daemonization tools.

Generic init scripts

See the extra/generic-init.d/ directory Celery distribution.

This directory contains generic bash init scripts for **celeryd**, that should run on Linux, FreeBSD, OpenBSD, and other Unix platforms.

Init script: celeryd

Usage */etc/init.d/celeryd {start|stop|restart|status}*

Configuration file /etc/default/celeryd

To configure celeryd you probably need to at least tell it where to change directory to when it starts (to find your *celeryconfig*).

Example configuration This is an example configuration for a Python project.

`/etc/default/celeryd`:

```
# Name of nodes to start
# here we have a single node
CELERYD_NODES="w1"
# or we could have three nodes:
#CELERYD_NODES="w1 w2 w3"

# Where to chdir at start.
CELERYD_CHDIR="/opt/Myproject/"

# Extra arguments to celeryd
CELERYD_OPTS="--time-limit=300 --concurrency=8"

# %n will be replaced with the nodename.
CELERYD_LOG_FILE="/var/log/celery/%n.log"
CELERYD_PID_FILE="/var/run/celery/%n.pid"

# Workers should run as an unprivileged user.
CELERYD_USER="celery"
CELERYD_GROUP="celery"
```

Example Django configuration This is an example configuration for those using *django-celery*:

```
# Name of nodes to start, here we have a single node
CELERYD_NODES="w1"
# or we could have three nodes:
#CELERYD_NODES="w1 w2 w3"

# Where to chdir at start.
CELERYD_CHDIR="/opt/Myproject/"

# How to call "manage.py celeryd_multi"
CELERYD_MULTI="$CELERYD_CHDIR/manage.py celeryd_multi"

# How to call "manage.py celeryctl"
CELERYCTL="$CELERYD_CHDIR/manage.py celeryctl"

# Extra arguments to celeryd
CELERYD_OPTS="--time-limit=300 --concurrency=8"

# %n will be replaced with the nodename.
CELERYD_LOG_FILE="/var/log/celery/%n.log"
CELERYD_PID_FILE="/var/run/celery/%n.pid"

# Workers should run as an unprivileged user.
CELERYD_USER="celery"
CELERYD_GROUP="celery"

# Name of the projects settings module.
export DJANGO_SETTINGS_MODULE="MyProject.settings"
```

Example Django configuration Using Virtualenv In case you are using virtualenv, you should add the path to your environment's python interpreter:

```
# Name of nodes to start, here we have a single node
CELERYD_NODES="w1"
# or we could have three nodes:
```

```
#CELERYD_NODES="w1 w2 w3"

# Where to chdir at start.
CELERYD_CHDIR="/opt/Myproject/"

# Python interpreter from environment.
ENV_PYTHON="$CELERYD_CHDIR/env/bin/python"

# How to call "manage.py celeryd_multi"
CELERYD_MULTI="$ENV_PYTHON $CELERYD_CHDIR/manage.py celeryd_multi"

# How to call "manage.py celeryctl"
CELERYCTL="$ENV_PYTHON $CELERYD_CHDIR/manage.py celeryctl"

# Extra arguments to celeryd
CELERYD_OPTS="--time-limit=300 --concurrency=8"

# %n will be replaced with the nodename.
CELERYD_LOG_FILE="/var/log/celery/%n.log"
CELERYD_PID_FILE="/var/run/celery/%n.pid"

# Workers should run as an unprivileged user.
CELERYD_USER="celery"
CELERYD_GROUP="celery"

# Name of the projects settings module.
export DJANGO_SETTINGS_MODULE="MyProject.settings"
```

Available options

- **CELERYD_NODES** Node names to start.
- **CELERYD_OPTS** Additional arguments to celeryd, see *celeryd –help* for a list.
- **CELERYD_CHDIR** Path to change directory to at start. Default is to stay in the current directory.
- **CELERYD_PID_FILE** Full path to the PID file. Default is /var/run/celeryd%n.pid
- **CELERYD_LOG_FILE** Full path to the celeryd log file. Default is /var/log/celeryd@%n.log
- **CELERYD_LOG_LEVEL** Log level to use for celeryd. Default is INFO.
- **CELERYD_MULTI** Path to the celeryd-multi program. Default is *celeryd-multi*. You can point this to a virtualenv, or even use manage.py for django.
- **CELERYCTL** Path to the celeryctl program. Default is *celeryctl*. You can point this to a virtualenv, or even use manage.py for django.
- **CELERYD_USER** User to run celeryd as. Default is current user.
- **CELERYD_GROUP** Group to run celeryd as. Default is current user.

Init script: celerybeat

Usage */etc/init.d/celerybeat {start|stop|restart}*

Configuration file /etc/default/celerybeat or /etc/default/celeryd

Example configuration This is an example configuration for a Python project:

/etc/default/celerybeat:

```
# Where to chdir at start.
CELERYBEAT_CHDIR="/opt/Myproject/"

# Extra arguments to celerybeat
CELERYBEAT_OPTS="--schedule=/var/run/celerybeat-schedule"
```

Example Django configuration This is an example configuration for those using *django-celery*

/etc/default/celerybeat:

```
# Where the Django project is.
CELERYBEAT_CHDIR="/opt/Project/"

# Name of the projects settings module.
export DJANGO_SETTINGS_MODULE="settings"

# Path to celerybeat
CELERYBEAT="/opt/Project/manage.py celerybeat"

# Extra arguments to celerybeat
CELERYBEAT_OPTS="--schedule=/var/run/celerybeat-schedule"
```

Available options

- **CELERYBEAT_OPTS** Additional arguments to celerybeat, see *celerybeat –help* for a list.
- **CELERYBEAT_PID_FILE** Full path to the PID file. Default is /var/run/celeryd.pid.
- **CELERYBEAT_LOG_FILE** Full path to the celeryd log file. Default is /var/log/celeryd.log
- **CELERYBEAT_LOG_LEVEL** Log level to use for celeryd. Default is INFO.
- **CELERYBEAT** Path to the celeryd program. Default is *celeryd*. You can point this to an virtualenv, or even use manage.py for django.
- **CELERYBEAT_USER** User to run celeryd as. Default is current user.
- **CELERYBEAT_GROUP** Group to run celeryd as. Default is current user.

Troubleshooting

If you can't get the init scripts to work, you should try running them in *verbose mode*:

```
$ sh -x /etc/init.d/celeryd start
```

This can reveal hints as to why the service won't start.

Also you will see the commands generated, so you can try to run the celeryd command manually to read the resulting error output.

For example my *sh -x* output does this:

```
++ start-stop-daemon --start --chdir /opt/App/release/app --quiet \
    --oknodo --background --make-pidfile --pidfile /var/run/celeryd.pid \
    --exec /opt/App/release/app/manage.py celeryd -- --time-limit=300 \
    -f /var/log/celeryd.log -l INFO
```

Run the celeryd command after *–exec* (without the –) to show the actual resulting output:

```
$ /opt/App/release/app/manage.py celeryd --time-limit=300 \
    -f /var/log/celeryd.log -l INFO
```

supervisord

- extra/supervisord/

launchd (OS X)

- extra/mac/

Windows

See this excellent external tutorial:

http://www.calazan.com/windows-tip-run-applications-in-the-background-using-task-scheduler/

CentOS

In CentOS we can take advantage of built-in service helpers, such as the pid-based status checker function in `/etc/init.d/functions`. See the sample script in http://github.com/celery/celery/tree/3.0/extra/centos/.

2.8.2 Debugging Tasks Remotely (using pdb)

Basics

`celery.contrib.rdb` is an extended version of `pdb` that enables remote debugging of processes that does not have terminal access.

Example usage:

```
from celery import task
from celery.contrib import rdb

@task()
def add(x, y):
    result = x + y
    rdb.set_trace()  # <- set breakpoint
    return result
```

`set_trace()` sets a breakpoint at the current location and creates a socket you can telnet into to remotely debug your task.

The debugger may be started by multiple processes at the same time, so rather than using a fixed port the debugger will search for an available port, starting from the base port (6900 by default). The base port can be changed using the environment variable `CELERY_RDB_PORT`.

By default the debugger will only be available from the local host, to enable access from the outside you have to set the environment variable `CELERY_RDB_HOST`.

When the worker encounters your breakpoint it will log the following information:

```
[INFO/MainProcess] Got task from broker:
    tasks.add[d7261c71-4962-47e5-b342-2448bedd20e8]
[WARNING/PoolWorker-1] Remote Debugger:6900:
    Please telnet 127.0.0.1 6900.  Type `exit` in session to continue.
[2011-01-18 14:25:44,119: WARNING/PoolWorker-1] Remote Debugger:6900:
    Waiting for client...
```

If you telnet the port specified you will be presented with a *pdb* shell:

```
$ telnet localhost 6900
Connected to localhost.
Escape character is '^]'.
> /opt/devel/demoapp/tasks.py(128)add()
-> return result
(Pdb)
```

Enter `help` to get a list of available commands, It may be a good idea to read the Python Debugger Manual if you have never used *pdb* before.

To demonstrate, we will read the value of the `result` variable, change it and continue execution of the task:

```
(Pdb) result
4
(Pdb) result = 'hello from rdb'
(Pdb) continue
Connection closed by foreign host.
```

The result of our vandalism can be seen in the worker logs:

```
[2011-01-18 14:35:36,599: INFO/MainProcess] Task
    tasks.add[d7261c71-4962-47e5-b342-2448bedd20e8] succeeded
    in 61.481s: 'hello from rdb'
```

Tips

Enabling the breakpoint signal

If the environment variable `CELERY_RDBSIG` is set, the worker will open up an rdb instance whenever the *SIGUSR2* signal is sent. This is the case for both main and worker processes.

For example starting the worker with:

```
CELERY_RDBSIG=1 celery worker -l info
```

You can start an rdb session for any of the worker processes by executing:

```
kill -USR2 <pid>
```

2.8.3 Task Cookbook

- Ensuring a task is only executed one at a time

Ensuring a task is only executed one at a time

You can accomplish this by using a lock.

In this example we'll be using the cache framework to set a lock that is accessible for all workers.

It's part of an imaginary RSS feed importer called *djangofeeds*. The task takes a feed URL as a single argument, and imports that feed into a Django model called *Feed*. We ensure that it's not possible for two or more workers to import the same feed at the same time by setting a cache key consisting of the MD5 checksum of the feed URL.

The cache key expires after some time in case something unexpected happens (you never know, right?)

```
from celery import task
from celery.utils.log import get_task_logger
from django.core.cache import cache
from django.utils.hashcompat import md5_constructor as md5
from djangofeeds.models import Feed

logger = get_task_logger(__name__)

LOCK_EXPIRE = 60 * 5 # Lock expires in 5 minutes

@task(bind=True)
def import_feed(self, feed_url):
    # The cache key consists of the task name and the MD5 digest
    # of the feed URL.
    feed_url_digest = md5(feed_url).hexdigest()
    lock_id = '%s-lock-%s' % (self.name, feed_url_hexdigest)

    # cache.add fails if if the key already exists
    acquire_lock = lambda: cache.add(lock_id, 'true', LOCK_EXPIRE)
    # memcache delete is very slow, but we have to use it to take
    # advantage of using add() for atomic locking
    release_lock = lambda: cache.delete(lock_id)

    logger.debug('Importing feed: %s' % feed_url)
    if acquire_lock():
        try:
            feed = Feed.objects.import_feed(feed_url)
        finally:
            release_lock()
        return feed.url

    logger.debug(
        'Feed %s is already being imported by another worker' % feed_url)
```

2.9 Frequently Asked Questions

2.9.1 General

What kinds of things should I use Celery for?

Answer: Queue everything and delight everyone is a good article describing why you would use a queue in a web context.

These are some common use cases:

- Running something in the background. For example, to finish the web request as soon as possible, then update the users page incrementally. This gives the user the impression of good performance and "snappiness", even though the real work might actually take some time.
- Running something after the web request has finished.
- Making sure something is done, by executing it asynchronously and using retries.
- Scheduling periodic work.

And to some degree:

- Distributed computing.
- Parallel execution.

2.9.2 Misconceptions

Does Celery really consist of 50.000 lines of code?

Answer: No, this and similarly large numbers have been reported at various locations.

The numbers as of this writing are:

- core: 7,141 lines of code.
- tests: 14,209 lines.
- backends, contrib, compat utilities: 9,032 lines.

Lines of code is not a useful metric, so even if Celery did consist of 50k lines of code you would not be able to draw any conclusions from such a number.

Does Celery have many dependencies?

A common criticism is that Celery uses too many dependencies. The rationale behind such a fear is hard to imagine, especially considering code reuse as the established way to combat complexity in modern software development, and that the cost of adding dependencies is very low now that package managers like pip and PyPI makes the hassle of installing and maintaining dependencies a thing of the past.

Celery has replaced several dependencies along the way, and the current list of dependencies are:

celery

- kombu

Kombu is part of the Celery ecosystem and is the library used to send and receive messages. It is also the library that enables us to support many different message brokers. It is also used by the OpenStack project, and many others, validating the choice to separate it from the Celery codebase.

- billiard

Billiard is a fork of the Python multiprocessing module containing many performance and stability improvements. It is an eventual goal that these improvements will be merged back into Python one day.

It is also used for compatibility with older Python versions.

- python-dateutil

The dateutil module is used by Celery to parse ISO-8601 formatted time strings, as well as its `relativedelta` class which is used in the implementation of crontab style periodic tasks.

django-celery

If you use django-celery then you don't have to install celery separately, as it will make sure that the required version is installed.

django-celery does not have any other dependencies.

kombu

Kombu depends on the following packages:

- amqp

The underlying pure-Python amqp client implementation. AMQP being the default broker this is a natural dependency.

- anyjson

anyjson is an utility library to select the best possible JSON implementation.

Note: For compatibility reasons additional packages may be installed if you are running on older Python versions, for example Python 2.6 depends on the `importlib`, and `ordereddict` libraries.

Also, to handle the dependencies for popular configuration choices Celery defines a number of "bundle" packages, see *Bundles*.

Is Celery heavy-weight?

Celery poses very little overhead both in memory footprint and performance.

But please note that the default configuration is not optimized for time nor space, see the *Optimizing* guide for more information.

Is Celery dependent on pickle?

Answer: No.

Celery can support any serialization scheme and has built-in support for JSON, YAML, Pickle and msgpack. Also, as every task is associated with a content type, you can even send one task using pickle, and another using JSON.

The default serialization format is pickle simply because it is convenient (it supports sending complex Python objects as task arguments).

If you need to communicate with other languages you should change to a serialization format that is suitable for that.

You can set a global default serializer, the default serializer for a particular Task, or even what serializer to use when sending a single task instance.

Is Celery for Django only?

Answer: No.

Celery does not depend on Django anymore. To use Celery with Django you have to use the django-celery package.

Do I have to use AMQP/RabbitMQ?

Answer: No.

You can also use Redis, Beanstalk, CouchDB, MongoDB or an SQL database, see *Brokers*.

These "virtual transports" may have limited broadcast and event functionality. For example remote control commands only works with AMQP and Redis.

Redis or a database won't perform as well as an AMQP broker. If you have strict reliability requirements you are encouraged to use RabbitMQ or another AMQP broker. Redis/database also use polling, so they are likely to consume more resources. However, if you for some reason are not able to use AMQP, feel free to use these alternatives. They will probably work fine for most use cases, and note that the above points are not specific to Celery; If using Redis/database as a queue worked fine for you before, it probably will now. You can always upgrade later if you need to.

Is Celery multilingual?

Answer: Yes.

`celeryd` is an implementation of Celery in Python. If the language has an AMQP client, there shouldn't be much work to create a worker in your language. A Celery worker is just a program connecting to the broker to process messages.

Also, there's another way to be language independent, and that is to use REST tasks, instead of your tasks being functions, they're URLs. With this information you can even create simple web servers that enable preloading of code. See: *User Guide: Remote Tasks*.

2.9.3 Troubleshooting

MySQL is throwing deadlock errors, what can I do?

Answer: MySQL has default isolation level set to *REPEATABLE-READ*, if you don't really need that, set it to *READ-COMMITTED*. You can do that by adding the following to your `my.cnf`:

```
[mysqld]
transaction-isolation = READ-COMMITTED
```

For more information about InnoDB's transaction model see MySQL - The InnoDB Transaction Model and Locking in the MySQL user manual.

(Thanks to Honza Kral and Anton Tsigularov for this solution)

celeryd is not doing anything, just hanging

Answer: See MySQL is throwing deadlock errors, what can I do?. or *Why is Task.delay/apply* just hanging?*.

Task results aren't reliably returning

Answer: If you're using the database backend for results, and in particular using MySQL, see MySQL is throwing deadlock errors, what can I do?.

Why is Task.delay/apply*/celeryd just hanging?

Answer: There is a bug in some AMQP clients that will make it hang if it's not able to authenticate the current user, the password doesn't match or the user does not have access to the virtual host specified. Be sure to check your broker logs (for RabbitMQ that is `/var/log/rabbitmq/rabbit.log` on most systems), it usually contains a message describing the reason.

Does it work on FreeBSD?

Answer: The multiprocessing pool requires a working POSIX semaphore implementation which isn't enabled in FreeBSD by default. You have to enable POSIX semaphores in the kernel and manually recompile multiprocessing.

Luckily, Viktor Petersson has written a tutorial to get you started with Celery on FreeBSD here: http://www.playingwithwire.com/2009/10/how-to-get-celeryd-to-work-on-freebsd/

I'm having *IntegrityError: Duplicate Key* errors. Why?

Answer: See MySQL is throwing deadlock errors, what can I do?. Thanks to howsthedotcom.

Why aren't my tasks processed?

Answer: With RabbitMQ you can see how many consumers are currently receiving tasks by running the following command:

```
$ rabbitmqctl list_queues -p <myvhost> name messages consumers
Listing queues ...
celery     2891    2
```

This shows that there's 2891 messages waiting to be processed in the task queue, and there are two consumers processing them.

One reason that the queue is never emptied could be that you have a stale worker process taking the messages hostage. This could happen if celeryd wasn't properly shut down.

When a message is received by a worker the broker waits for it to be acknowledged before marking the message as processed. The broker will not re-send that message to another consumer until the consumer is shut down properly.

If you hit this problem you have to kill all workers manually and restart them:

```
ps auxww | grep celeryd | awk '{print $2}' | xargs kill
```

You might have to wait a while until all workers have finished the work they're doing. If it's still hanging after a long time you can kill them by force with:

```
ps auxww | grep celeryd | awk '{print $2}' | xargs kill -9
```

Why won't my Task run?

Answer: There might be syntax errors preventing the tasks module being imported.

You can find out if Celery is able to run the task by executing the task manually:

```
>>> from myapp.tasks import MyPeriodicTask
>>> MyPeriodicTask.delay()
```

Watch celeryd's log file to see if it's able to find the task, or if some other error is happening.

Why won't my periodic task run?

Answer: See Why won't my Task run?.

How do I purge all waiting tasks?

Answer: You can use the `celery purge` command to purge all configured task queues:

```
$ celery purge
```

or programatically:

```
>>> from celery import current_app as celery
>>> celery.control.purge()
1753
```

If you only want to purge messages from a specific queue you have to use the AMQP API or the **celery amqp** utility:

```
$ celery amqp queue.purge <queue name>
```

The number 1753 is the number of messages deleted.

You can also start `celeryd` with the *`--purge`* argument, to purge messages when the worker starts.

I've purged messages, but there are still messages left in the queue?

Answer: Tasks are acknowledged (removed from the queue) as soon as they are actually executed. After the worker has received a task, it will take some time until it is actually executed, especially if there are a lot of tasks already waiting for execution. Messages that are not acknowledged are held on to by the worker until it closes the connection to the broker (AMQP server). When that connection is closed (e.g. because the worker was stopped) the tasks will be re-sent by the broker to the next available worker (or the same worker when it has been restarted), so to properly purge the queue of waiting tasks you have to stop all the workers, and then purge the tasks using `celery.control.purge()`.

2.9.4 Results

How do I get the result of a task if I have the ID that points there?

Answer: Use *task.AsyncResult*:

```
>>> result = my_task.AsyncResult(task_id)
>>> result.get()
```

This will give you a `AsyncResult` instance using the tasks current result backend.

If you need to specify a custom result backend, or you want to use the current application's default backend you can use `Celery.AsyncResult`:

```
>>> result = app.AsyncResult(task_id)
>>> result.get()
```

2.9.5 Security

Isn't using *pickle* a security concern?

Answer: Yes, indeed it is.

You are right to have a security concern, as this can indeed be a real issue. It is essential that you protect against unauthorized access to your broker, databases and other services transmitting pickled data.

For the task messages you can set the `CELERY_TASK_SERIALIZER` setting to "json" or "yaml" instead of pickle. There is currently no alternative solution for task results (but writing a custom result backend using JSON is a simple task)

Note that this is not just something you should be aware of with Celery, for example also Django uses pickle for its cache client.

Can messages be encrypted?

Answer: Some AMQP brokers supports using SSL (including RabbitMQ). You can enable this using the `BROKER_USE_SSL` setting.

It is also possible to add additional encryption and security to messages, if you have a need for this then you should contact the *Mailing list*.

Is it safe to run celeryd as root?

Answer: No!

We're not currently aware of any security issues, but it would be incredibly naive to assume that they don't exist, so running the Celery services (**celeryd**, **celerybeat**, **celeryev**, etc) as an unprivileged user is recommended.

2.9.6 Brokers

Why is RabbitMQ crashing?

Answer: RabbitMQ will crash if it runs out of memory. This will be fixed in a future release of RabbitMQ. please refer to the RabbitMQ FAQ: http://www.rabbitmq.com/faq.html#node-runs-out-of-memory

---

Note: This is no longer the case, RabbitMQ versions 2.0 and above includes a new persister, that is tolerant to out of memory errors. RabbitMQ 2.1 or higher is recommended for Celery.

If you're still running an older version of RabbitMQ and experience crashes, then please upgrade!

Misconfiguration of Celery can eventually lead to a crash on older version of RabbitMQ. Even if it doesn't crash, this can still consume a lot of resources, so it is very important that you are aware of the common pitfalls.

- Events.

Running `celeryd` with the `-E`/`--events` option will send messages for events happening inside of the worker.

Events should only be enabled if you have an active monitor consuming them, or if you purge the event queue periodically.

- AMQP backend results.

When running with the AMQP result backend, every task result will be sent as a message. If you don't collect these results, they will build up and RabbitMQ will eventually run out of memory.

Results expire after 1 day by default. It may be a good idea to lower this value by configuring the `CELERY_TASK_RESULT_EXPIRES` setting.

If you don't use the results for a task, make sure you set the *ignore_result* option:

Can I use Celery with ActiveMQ/STOMP?

Answer: No. It used to be supported by Carrot, but is not currently supported in Kombu.

What features are not supported when not using an AMQP broker?

This is an incomplete list of features not available when using the virtual transports:

- Remote control commands (supported only by Redis).
- Monitoring with events may not work in all virtual transports.
- **The *header* and *fanout* exchange types** (*fanout* is supported by Redis).

2.9.7 Tasks

How can I reuse the same connection when calling tasks?

Answer: See the `BROKER_POOL_LIMIT` setting. The connection pool is enabled by default since version 2.5.

Sudo in a `subprocess` returns `None`

There is a sudo configuration option that makes it illegal for process without a tty to run sudo:

```
Defaults requiretty
```

If you have this configuration in your `/etc/sudoers` file then tasks will not be able to call sudo when celeryd is running as a daemon. If you want to enable that, then you need to remove the line from sudoers.

See: http://timelordz.com/wiki/Apache_Sudo_Commands

Why do workers delete tasks from the queue if they are unable to process them?

Answer:

The worker rejects unknown tasks, messages with encoding errors and messages that doesn't contain the proper fields (as per the task message protocol).

If it did not reject them they could be redelivered again and again, causing a loop.

Recent versions of RabbitMQ has the ability to configure a dead-letter queue for exchange, so that rejected messages is moved there.

Can I call a task by name?

Answer: Yes. Use `celery.execute.send_task()`. You can also call a task by name from any language that has an AMQP client.

```
>>> from celery.execute import send_task
>>> send_task("tasks.add", args=[2, 2], kwargs={})
<AsyncResult: 373550e8-b9a0-4666-bc61-ace01fa4f91d>
```

How can I get the task id of the current task?

Answer: The current id and more is available in the task request:

```
@celery.task
def mytask():
    cache.set(mytask.request.id, "Running")
```

For more information see *Context*.

Can I specify a custom task_id?

Answer: Yes. Use the *task_id* argument to `Task.apply_async()`:

```
>>> task.apply_async(args, kwargs, task_id="...")
```

Can I use decorators with tasks?

Answer: Yes. But please see note in the sidebar at *Basics*.

Can I use natural task ids?

Answer: Yes, but make sure it is unique, as the behavior for two tasks existing with the same id is undefined.

The world will probably not explode, but at the worst they can overwrite each others results.

How can I run a task once another task has finished?

Answer: You can safely launch a task inside a task. Also, a common pattern is to add callbacks to tasks:

```
from celery.utils.log import get_task_logger

logger = get_task_logger(__name__)

@celery.task
def add(x, y):
    return x + y

@celery.task(ignore_result=True)
def log_result(result):
    logger.info("log_result got: %r" % (result, ))
```

Invocation:

```
>>> (add.s(2, 2) | log_result.s()).delay()
```

See *Canvas: Designing Workflows* for more information.

Can I cancel the execution of a task?

Answer: Yes. Use *result.revoke*:

```
>>> result = add.apply_async(args=[2, 2], countdown=120)
>>> result.revoke()
```

or if you only have the task id:

```
>>> from celery import current_app as celery
>>> celery.control.revoke(task_id)
```

Why aren't my remote control commands received by all workers?

Answer: To receive broadcast remote control commands, every worker node uses its host name to create a unique queue name to listen to, so if you have more than one worker with the same host name, the control commands will be received in round-robin between them.

To work around this you can explicitly set the host name for every worker using the `--hostname` argument to `celeryd`:

```
$ celeryd --hostname=$(hostname).1
$ celeryd --hostname=$(hostname).2
```

etc., etc...

Can I send some tasks to only some servers?

Answer: Yes. You can route tasks to an arbitrary server using AMQP, and a worker can bind to as many queues as it wants.

See *Routing Tasks* for more information.

Can I change the interval of a periodic task at runtime?

Answer: Yes. You can use the Django database scheduler, or you can override *PeriodicTask.is_due* or turn *PeriodicTask.run_every* into a property:

```
class MyPeriodic(PeriodicTask):

    def run(self):
        # ...

    @property
    def run_every(self):
        return get_interval_from_database(...)
```

Does celery support task priorities?

Answer: No. In theory, yes, as AMQP supports priorities. However RabbitMQ doesn't implement them yet.

The usual way to prioritize work in Celery, is to route high priority tasks to different servers. In the real world this may actually work better than per message priorities. You can use this in combination with rate limiting to achieve a highly responsive system.

Should I use retry or acks_late?

Answer: Depends. It's not necessarily one or the other, you may want to use both.

Task.retry is used to retry tasks, notably for expected errors that is catchable with the *try:* block. The AMQP transaction is not used for these errors: **if the task raises an exception it is still acknowledged!**.

The *acks_late* setting would be used when you need the task to be executed again if the worker (for some reason) crashes mid-execution. It's important to note that the worker is not known to crash, and if it does it is usually an unrecoverable error that requires human intervention (bug in the worker, or task code).

In an ideal world you could safely retry any task that has failed, but this is rarely the case. Imagine the following task:

```
@celery.task
def process_upload(filename, tmpfile):
    # Increment a file count stored in a database
    increment_file_counter()
    add_file_metadata_to_db(filename, tmpfile)
    copy_file_to_destination(filename, tmpfile)
```

If this crashed in the middle of copying the file to its destination the world would contain incomplete state. This is not a critical scenario of course, but you can probably imagine something far more sinister. So for ease of programming we have less reliability; It's a good default, users who require it and know what they are doing can still enable acks_late (and in the future hopefully use manual acknowledgement)

In addition *Task.retry* has features not available in AMQP transactions: delay between retries, max retries, etc.

So use retry for Python errors, and if your task is idempotent combine that with *acks_late* if that level of reliability is required.

Can I schedule tasks to execute at a specific time?

Answer: Yes. You can use the *eta* argument of `Task.apply_async()`.

Or to schedule a periodic task at a specific time, use the `celery.schedules.crontab` schedule behavior:

```
from celery.schedules import crontab
from celery.task import periodic_task

@periodic_task(run_every=crontab(hour=7, minute=30, day_of_week="mon"))
def every_monday_morning():
    print("This is run every Monday morning at 7:30")
```

How do I shut down *celeryd* safely?

Answer: Use the `TERM` signal, and the worker will finish all currently executing jobs and shut down as soon as possible. No tasks should be lost.

You should never stop `celeryd` with the `KILL` signal (*-9*), unless you've tried `TERM` a few times and waited a few minutes to let it get a chance to shut down. As if you do tasks may be terminated mid-execution, and they will not be re-run unless you have the *acks_late* option set (*Task.acks_late* / `CELERY_ACKS_LATE`).

See also:

Stopping the worker

How do I run celeryd in the background on [platform]?

Answer: Please see *Running the worker as a daemon*.

2.9.8 Django

What purpose does the database tables created by django-celery have?

Several database tables are created by default, these relate to

- Monitoring

  When you use the django-admin monitor, the cluster state is written to the `TaskState` and `WorkerState` models.

- Periodic tasks

  When the database-backed schedule is used the periodic task schedule is taken from the `PeriodicTask` model, there are also several other helper tables (`IntervalSchedule`, `CrontabSchedule`, `PeriodicTasks`).

- Task results

  The database result backend is enabled by default when using django-celery (this is for historical reasons, and thus for backward compatibility).

  The results are stored in the `TaskMeta` and `TaskSetMeta` models. *these tables are not created if another result backend is configured*.

2.9.9 Windows

The *-B* / *–beat* option to celeryd doesn't work?

Answer: That's right. Run *celerybeat* and *celeryd* as separate services instead.

2.10 What's new in Celery 3.0 (Chiastic Slide)

Celery is a simple, flexible and reliable distributed system to process vast amounts of messages, while providing operations with the tools required to maintain such a system.

It's a task queue with focus on real-time processing, while also supporting task scheduling.

Celery has a large and diverse community of users and contributors, you should come join us *on IRC* or *our mailing-list*.

To read more about Celery you should go read the *introduction*.

While this version is backward compatible with previous versions it's important that you read the following section.

If you use Celery in combination with Django you must also read the django-celery changelog and upgrade to django-celery 3.0.

This version is officially supported on CPython 2.5, 2.6, 2.7, 3.2 and 3.3, as well as PyPy and Jython.

2.10.1 Highlights

Overview

- A new and improved API, that is both simpler and more powerful.
  Everyone must read the new *First Steps with Celery* tutorial, and the new *Next Steps* tutorial.
  Oh, and why not reread the user guide while you're at it :)
  There are no current plans to deprecate the old API, so you don't have to be in a hurry to port your applications.
- The worker is now thread-less, giving great performance improvements.
- The new "Canvas" makes it easy to define complex workflows.
  Ever wanted to chain tasks together? This is possible, but not just that, now you can even chain together groups and chords, or even combine multiple chains.
  Read more in the *Canvas* user guide.
- All of Celery's command line programs are now available from a single **celery** umbrella command.
- This is the last version to support Python 2.5.
  Starting with Celery 3.1, Python 2.6 or later is required.
- Support for the new librabbitmq C client.
  Celery will automatically use the `librabbitmq` module if installed, which is a very fast and memory-optimized replacement for the py-amqp module.
- Redis support is more reliable with improved ack emulation.
- Celery now always uses UTC
- Over 600 commits, 30k additions/36k deletions.
  In comparison 1.0 2.0 had 18k additions/8k deletions.

2.10.2 Important Notes

Broadcast exchanges renamed

The workers remote control command exchanges has been renamed (a new pidbox name), this is because the `auto_delete` flag on the exchanges has been removed, and that makes it incompatible with earlier versions.

You can manually delete the old exchanges if you want, using the **celery amqp** command (previously called `camqadm`):

```
$ celery amqp exchange.delete celeryd.pidbox
$ celery amqp exchange.delete reply.celeryd.pidbox
```

Eventloop

The worker is now running *without threads* when used with RabbitMQ (AMQP), or Redis as a broker, resulting in:

- Much better overall performance.
- Fixes several edge case race conditions.
- Sub-millisecond timer precision.

- Faster shutdown times.

The transports supported are: `py-amqp` `librabbitmq`, `redis`, and `amqplib`. Hopefully this can be extended to include additional broker transports in the future.

For increased reliability the `CELERY_FORCE_EXECV` setting is enabled by default if the eventloop is not used.

New `celery` umbrella command

All Celery's command line programs are now available from a single **celery** umbrella command.

You can see a list of subcommands and options by running:

```
$ celery help
```

Commands include:

- `celery worker` (previously `celeryd`).
- `celery beat` (previously `celerybeat`).
- `celery amqp` (previously `camqadm`).

The old programs are still available (`celeryd`, `celerybeat`, etc), but you are discouraged from using them.

Now depends on `billiard`.

Billiard is a fork of the multiprocessing containing the no-execv patch by sbt (http://bugs.python.org/issue8713), and also contains the pool improvements previously located in Celery.

This fork was necessary as changes to the C extension code was required for the no-execv patch to work.

- Issue #625
- Issue #627
- Issue #640
- *django-celery #122 <http://github.com/celery/django-celery/issues/122*
- *django-celery #124 <http://github.com/celery/django-celery/issues/122*

`celery.app.task` no longer a package

The `celery.app.task` module is now a module instead of a package.

The setup.py install script will try to remove the old package, but if that doesn't work for some reason you have to remove it manually. This command helps:

```
$ rm -r $(dirname $(python -c '
    import celery;print(celery.__file__)'))/app/task/
```

If you experience an error like `ImportError: cannot import name _unpickle_task`, you just have to remove the old package and everything is fine.

Last version to support Python 2.5

The 3.0 series will be last version to support Python 2.5, and starting from 3.1 Python 2.6 and later will be required.

With several other distributions taking the step to discontinue Python 2.5 support, we feel that it is time too.

Python 2.6 should be widely available at this point, and we urge you to upgrade, but if that is not possible you still have the option to continue using the Celery 3.0, and important bug fixes introduced in Celery 3.1 will be back-ported to Celery 3.0 upon request.

UTC timezone is now used

This means that ETA/countdown in messages are not compatible with Celery versions prior to 2.5.

You can disable UTC and revert back to old local time by setting the `CELERY_ENABLE_UTC` setting.

Redis: Ack emulation improvements

Reducing the possibility of data loss.

Acks are now implemented by storing a copy of the message when the message is consumed. The copy is not removed until the consumer acknowledges or rejects it.

This means that unacknowledged messages will be redelivered either when the connection is closed, or when the visibility timeout is exceeded.

- Visibility timeout

  This is a timeout for acks, so that if the consumer does not ack the message within this time limit, the message is redelivered to another consumer.

  The timeout is set to one hour by default, but can be changed by configuring a transport option:

```
BROKER_TRANSPORT_OPTIONS = {'visibility_timeout': 18000}  # 5 hours
```

Note: Messages that have not been acked will be redelivered if the visibility timeout is exceeded, for Celery users this means that ETA/countdown tasks that are scheduled to execute with a time that exceeds the visibility timeout will be executed twice (or more). If you plan on using long ETA/countdowns you should tweak the visibility timeout accordingly.

Setting a long timeout means that it will take a long time for messages to be redelivered in the event of a power failure, but if so happens you could temporarily set the visibility timeout lower to flush out messages when you start up the systems again.

2.10.3 News

Chaining Tasks

Tasks can now have callbacks and errbacks, and dependencies are recorded

- The task message format have been updated with two new extension keys

  Both keys can be empty/undefined or a list of subtasks.

  - `callbacks`

Applied if the task exits successfully, with the result of the task as an argument.

- `errbacks`

  Applied if an error occurred while executing the task, with the uuid of the task as an argument. Since it may not be possible to serialize the exception instance, it passes the uuid of the task instead. The uuid can then be used to retrieve the exception and traceback of the task from the result backend.

- `link` and `link_error` keyword arguments has been added to `apply_async`.

  These add callbacks and errbacks to the task, and you can read more about them at *Linking (callbacks/errbacks)*.

- We now track what subtasks a task sends, and some result backends supports retrieving this information.

  * task.request.children

    Contains the result instances of the subtasks the currently executing task has applied.

  * AsyncResult.children

    Returns the tasks dependencies, as a list of `AsyncResult`/`ResultSet` instances.

  * AsyncResult.iterdeps

    Recursively iterates over the tasks dependencies, yielding *(parent, node)* tuples.

    Raises IncompleteStream if any of the dependencies has not returned yet.

  * AsyncResult.graph

    A `DependencyGraph` of the tasks dependencies. This can also be used to convert to dot format:

    ```
    with open('graph.dot') as fh:
        result.graph.to_dot(fh)
    ```

    which can than be used to produce an image:

    ```
    $ dot -Tpng graph.dot -o graph.png
    ```

- A new special subtask called `chain` is also included:

```
>>> from celery import chain

# (2 + 2) * 8 / 2
>>> res = chain(add.subtask((2, 2)),
                mul.subtask((8, )),
                div.subtask((2,))).apply_async()
>>> res.get() == 16

>>> res.parent.get() == 32

>>> res.parent.parent.get() == 4
```

- Adds `AsyncResult.get_leaf()`

  Waits and returns the result of the leaf subtask. That is the last node found when traversing the graph, but this means that the graph can be 1-dimensional only (in effect a list).

- Adds `subtask.link(subtask) + subtask.link_error(subtask)`

  Shortcut to `s.options.setdefault('link', []).append(subtask)`

- Adds `subtask.flatten_links()`

  Returns a flattened list of all dependencies (recursively)

Redis: Priority support.

The message's `priority` field is now respected by the Redis transport by having multiple lists for each named queue. The queues are then consumed by in order of priority.

The priority field is a number in the range of 0 - 9, where 0 is the default and highest priority.

The priority range is collapsed into four steps by default, since it is unlikely that nine steps will yield more benefit than using four steps. The number of steps can be configured by setting the `priority_steps` transport option, which must be a list of numbers in **sorted order**:

```
>>> BROKER_TRANSPORT_OPTIONS = {
...     'priority_steps': [0, 2, 4, 6, 8, 9],
... }
```

Priorities implemented in this way is not as reliable as priorities on the server side, which is why the feature is nicknamed "quasi-priorities"; **Using routing is still the suggested way of ensuring quality of service**, as client implemented priorities fall short in a number of ways, e.g. if the worker is busy with long running tasks, has prefetched many messages, or the queues are congested.

Still, it is possible that using priorities in combination with routing can be more beneficial than using routing or priorities alone. Experimentation and monitoring should be used to prove this.

Contributed by Germán M. Bravo.

Redis: Now cycles queues so that consuming is fair.

This ensures that a very busy queue won't block messages from other queues, and ensures that all queues have an equal chance of being consumed from.

This used to be the case before, but the behavior was accidentally changed while switching to using blocking pop.

group/*chord*/*chain* are now subtasks

- group is no longer an alias to TaskSet, but new alltogether, since it was very difficult to migrate the TaskSet class to become a subtask.
- A new shortcut has been added to tasks:

  ```
  >>> task.s(arg1, arg2, kw=1)
  ```

  as a shortcut to:

  ```
  >>> task.subtask((arg1, arg2), {'kw': 1})
  ```

- Tasks can be chained by using the `|` operator:

  ```
  >>> (add.s(2, 2), pow.s(2)).apply_async()
  ```

- Subtasks can be "evaluated" using the `~` operator:

```
>>> ~add.s(2, 2)
4

>>> ~(add.s(2, 2) | pow.s(2))
```

is the same as:

```
>>> chain(add.s(2, 2), pow.s(2)).apply_async().get()
```

- A new subtask_type key has been added to the subtask dicts

  This can be the string "chord", "group", "chain", "chunks", "xmap", or "xstarmap".

- maybe_subtask now uses subtask_type to reconstruct the object, to be used when using non-pickle serializers.
- The logic for these operations have been moved to dedicated tasks celery.chord, celery.chain and celery.group.
- subtask no longer inherits from AttributeDict.

  It's now a pure dict subclass with properties for attribute access to the relevant keys.

- The repr's now outputs how the sequence would like imperatively:

```
>>> from celery import chord

>>> (chord([add.s(i, i) for i in xrange(10)], xsum.s())
      | pow.s(2))
tasks.xsum([tasks.add(0, 0),
            tasks.add(1, 1),
            tasks.add(2, 2),
            tasks.add(3, 3),
            tasks.add(4, 4),
            tasks.add(5, 5),
            tasks.add(6, 6),
            tasks.add(7, 7),
            tasks.add(8, 8),
            tasks.add(9, 9)]) | tasks.pow(2)
```

New remote control commands

These commands were previously experimental, but they have proven stable and is now documented as part of the offical API.

- `add_consumer`/`cancel_consumer`

  Tells workers to consume from a new queue, or cancel consuming from a queue. This command has also been changed so that the worker remembers the queues added, so that the change will persist even if the connection is re-connected.

  These commands are available programmatically as `celery.control.add_consumer()` / `celery.control.cancel_consumer()`:

```
>>> celery.control.add_consumer(queue_name,
...     destination=['w1.example.com'])
>>> celery.control.cancel_consumer(queue_name,
...     destination=['w1.example.com'])
```

  or using the **celery control** command:

```
$ celery control -d w1.example.com add_consumer queue
$ celery control -d w1.example.com cancel_consumer queue
```

Note: Remember that a control command without *destination* will be sent to **all workers**.

- `autoscale`

  Tells workers with *–autoscale* enabled to change autoscale max/min concurrency settings.

  This command is available programmatically as `celery.control.autoscale()`:

  ```
  >>> celery.control.autoscale(max=10, min=5,
  ...     destination=['w1.example.com'])
  ```

  or using the **celery control** command:

  ```
  $ celery control -d w1.example.com autoscale 10 5
  ```

- `pool_grow`/`pool_shrink`

  Tells workers to add or remove pool processes.

  These commands are available programmatically as `celery.control.pool_grow()` / `celery.control.pool_shrink()`:

  ```
  >>> celery.control.pool_grow(2, destination=['w1.example.com'])
  >>> celery.contorl.pool_shrink(2, destination=['w1.example.com'])
  ```

  or using the **celery control** command:

  ```
  $ celery control -d w1.example.com pool_grow 2
  $ celery control -d w1.example.com pool_shrink 2
  ```

- **celery control** now supports `rate_limit` & `time_limit` commands.

  See `celery control --help` for details.

Crontab now supports Day of Month, and Month of Year arguments

See the updated list of examples at *Crontab schedules*.

Immutable subtasks

`subtask`'s can now be immutable, which means that the arguments will not be modified when calling callbacks:

```
>>> chain(add.s(2, 2), clear_static_electricity.si())
```

means it will not receive the argument of the parent task, and `.si()` is a shortcut to:

```
>>> clear_static_electricity.subtask(immutable=True)
```

Logging Improvements

Logging support now conforms better with best practices.

- Classes used by the worker no longer uses app.get_default_logger, but uses *celery.utils.log.get_logger* which simply gets the logger not setting the level, and adds a NullHandler.
- Loggers are no longer passed around, instead every module using logging defines a module global logger that is used throughout.
- All loggers inherit from a common "celery".
- Before task.get_logger would setup a new logger for every task, and even set the loglevel. This is no longer the case.
  - Instead all task loggers now inherit from a common "celery.task" logger that is set up when programs call *setup_logging_subsystem*.
  - Instead of using LoggerAdapter to augment the formatter with the task_id and task_name field, the task base logger now use a special formatter adding these values at runtime from the currently executing task.
- In fact, `task.get_logger` is no longer recommended, it is better to add a module-level logger to your tasks module.

  For example, like this:

  ```
  from celery.utils.log import get_task_logger

  logger = get_task_logger(__name__)

  @celery.task
  def add(x, y):
      logger.debug('Adding %r + %r' % (x, y))
      return x + y
  ```

  The resulting logger will then inherit from the `"celery.task"` logger so that the current task name and id is included in logging output.
- Redirected output from stdout/stderr is now logged to a "celery.redirected" logger.
- In addition a few warnings.warn have been replaced with logger.warn.
- Now avoids the 'no handlers for logger multiprocessing' warning

Task registry no longer global

Every Celery instance now has its own task registry.

You can make apps share registries by specifying it:

```
>>> app1 = Celery()
>>> app2 = Celery(tasks=app1.tasks)
```

Note that tasks are shared between registries by default, so that tasks will be added to every subsequently created task registry. As an alternative tasks can be private to specific task registries by setting the `shared` argument to the `@task` decorator:

```
@celery.task(shared=False)
def add(x, y):
    return x + y
```

Abstract tasks are now lazily bound.

The `Task` class is no longer bound to an app by default, it will first be bound (and configured) when a concrete subclass is created.

This means that you can safely import and make task base classes, without also initializing the app environment:

```
from celery.task import Task

class DebugTask(Task):
    abstract = True

    def __call__(self, *args, **kwargs):
        print('CALLING %r' % (self, ))
        return self.run(*args, **kwargs)

>>> DebugTask
<unbound DebugTask>

>>> @celery1.task(base=DebugTask)
... def add(x, y):
...     return x + y
>>> add.__class__
<class add of <Celery default:0x101510d10>>
```

Lazy task decorators

The `@task` decorator is now lazy when used with custom apps.

That is, if `accept_magic_kwargs` is enabled (herby called "compat mode"), the task decorator executes inline like before, however for custom apps the @task decorator now returns a special PromiseProxy object that is only evaluated on access.

All promises will be evaluated when *app.finalize* is called, or implicitly when the task registry is first used.

Smart *–app* option

The `--app` option now 'auto-detects'

- If the provided path is a module it tries to get an attribute named 'celery'.
- If the provided path is a package it tries to import a submodule named 'celery', and get the celery attribute from that module.

E.g. if you have a project named 'proj' where the celery app is located in 'from proj.celery import celery', then the following will be equivalent:

```
$ celery worker --app=proj
$ celery worker --app=proj.celery:
$ celery worker --app=proj.celery:celery
```

In Other News

- New `CELERYD_WORKER_LOST_WAIT` to control the timeout in seconds before `billiard.WorkerLostError` is raised when a worker can not be signalled (Issue #595).

Contributed by Brendon Crawford.

- Redis event monitor queues are now automatically deleted (Issue #436).
- App instance factory methods have been converted to be cached descriptors that creates a new subclass on access.

  This means that e.g. `celery.Worker` is an actual class and will work as expected when:

  ```
  class Worker(celery.Worker):
      ...
  ```

- New signal: `task_success`.
- Multiprocessing logs are now only emitted if the `MP_LOG` environment variable is set.
- The Celery instance can now be created with a broker URL

  ```
  celery = Celery(broker='redis://')
  ```

- Result backends can now be set using an URL

  Currently only supported by redis. Example use:

  ```
  CELERY_RESULT_BACKEND = 'redis://localhost/1'
  ```

- Heartbeat frequency now every 5s, and frequency sent with event

  The heartbeat frequency is now available in the worker event messages, so that clients can decide when to consider workers offline based on this value.

- Module celery.actors has been removed, and will be part of cl instead.
- Introduces new `celery` command, which is an entrypoint for all other commands.

  The main for this command can be run by calling `celery.start()`.

- Annotations now supports decorators if the key startswith '@'.

  E.g.:

  ```
  def debug_args(fun):

      @wraps(fun)
      def _inner(*args, **kwargs):
          print('ARGS: %r' % (args, ))
      return _inner

  CELERY_ANNOTATIONS = {
      'tasks.add': {'@__call__': debug_args},
  }
  ```

  Also tasks are now always bound by class so that annotated methods end up being bound.

- Bugreport now available as a command and broadcast command
  - Get it from a Python repl:

    ```
    >>> import celery
    >>> print(celery.bugreport())
    ```

  - Using the `celery` command line program:

```
$ celery report
```

– Get it from remote workers:

```
$ celery inspect report
```

- Module `celery.log` moved to `celery.app.log`.
- Module `celery.task.control` moved to `celery.app.control`.
- New signal: `task_revoked`

  Sent in the main process when the task is revoked or terminated.

- `AsyncResult.task_id` renamed to `AsyncResult.id`
- `TasksetResult.taskset_id` renamed to `.id`
- `xmap(task, sequence)` and `xstarmap(task, sequence)`

  Returns a list of the results applying the task function to every item in the sequence.

  Example:

```
>>> from celery import xstarmap

>>> xstarmap(add, zip(range(10), range(10)).apply_async()
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

- `chunks(task, sequence, chunksize)`
- `group.skew(start=, stop=, step=)`

  Skew will skew the countdown for the individual tasks in a group, e.g. with a group:

```
>>> g = group(add.s(i, i) for i in xrange(10))
```

  Skewing the tasks from 0 seconds to 10 seconds:

```
>>> g.skew(stop=10)
```

  Will have the first task execute in 0 seconds, the second in 1 second, the third in 2 seconds and so on.

- 99% test Coverage
- `CELERY_QUEUES` can now be a list/tuple of `Queue` instances.

  Internally `celery.amqp.queues` is now a mapping of name/Queue instances, instead of converting on the fly.

- Can now specify connection for `celery.control.inspect`.

```
from kombu import Connection

i = celery.control.inspect(connection=Connection('redis://'))
i.active_queues()
```

- `CELERY_FORCE_EXECV` is now enabled by default.

  If the old behavior is wanted the setting can be set to False, or the new *--no-execv* to **celery worker**.

- Deprecated module `celery.conf` has been removed.
- The `CELERY_TIMEZONE` now always require the `pytz` library to be installed (exept if the timezone is set to *UTC*).

- The Tokyo Tyrant backend has been removed and is no longer supported.
- Now uses `maybe_declare()` to cache queue declarations.
- There is no longer a global default for the `CELERYBEAT_MAX_LOOP_INTERVAL` setting, it is instead set by individual schedulers.
- Worker: now truncates very long message bodies in error reports.
- No longer deepcopies exceptions when trying to serialize errors.
- `CELERY_BENCH` environment variable, will now also list memory usage statistics at worker shutdown.
- Worker: now only ever use a single timer for all timing needs, and instead set different priorities.
- An exceptions arguments are now safely pickled

  Contributed by Matt Long.
- Worker/Celerybeat no longer logs the startup banner.

  Previously it would be logged with severity warning, now it's only written to stdout.
- The `contrib/` directory in the distribution has been renamed to `extra/`.
- New signal: `task_revoked`
- celery.contrib.migrate: Many improvements including filtering, queue migration, and support for acking messages on the broker migrating from.

  Contributed by John Watson.
- Worker: Prefetch count increments are now optimized and grouped together.
- Worker: No longer calls `consume` on the remote control command queue twice.

  Probably didn't cause any problems, but was unecessary.

Internals

- `app.broker_connection` is now `app.connection`

  Both names still work.
- Compat modules are now generated dynamically upon use.

  These modules are `celery.messaging`, `celery.log`, `celery.decorators` and `celery.registry`.
- `celery.utils` refactored into multiple modules:

  `celery.utils.text` `celery.utils.imports` `celery.utils.functional`
- Now using `kombu.utils.encoding` instead of `celery.utils.encoding`.
- Renamed module `celery.routes` -> `celery.app.routes`.
- Renamed package `celery.db` -> `celery.backends.database`.
- Renamed module `celery.abstract` -> `celery.worker.bootsteps`.
- Command line docs are now parsed from the module docstrings.
- Test suite directory has been reorganized.
- **setup.py** now reads docs from the `requirements/` directory.
- Celery commands no longer wraps output (Issue #700).

Contributed by Thomas Johansson.

2.10.4 Experimental

`celery.contrib.methods`: Task decorator for methods

This is an experimental module containing a task decorator, and a task decorator filter, that can be used to create tasks out of methods:

```
from celery.contrib.methods import task_method

class Counter(object):

    def __init__(self):
        self.value = 1

    @celery.task(name='Counter.increment', filter=task_method)
    def increment(self, n=1):
        self.value += 1
        return self.value
```

See `celery.contrib.methods` for more information.

2.10.5 Unscheduled Removals

Usually we don't make backward incompatible removals, but these removals should have no major effect.

- The following settings have been renamed:
  - `CELERYD_ETA_SCHEDULER` -> `CELERYD_TIMER`
  - `CELERYD_ETA_SCHEDULER_PRECISION` -> `CELERYD_TIMER_PRECISION`

2.10.6 Deprecations

See the *Celery Deprecation Timeline*.

- The `celery.backends.pyredis` compat module has been removed.

  Use `celery.backends.redis` instead!

- The following undocumented API's has been moved:
  - `control.inspect.add_consumer` -> `celery.control.add_consumer()`.
  - `control.inspect.cancel_consumer` -> `celery.control.cancel_consumer()`.
  - `control.inspect.enable_events` -> `celery.control.enable_events()`.
  - `control.inspect.disable_events` -> `celery.control.disable_events()`.

  This way `inspect()` is only used for commands that do not modify anything, while idempotent control commands that make changes are on the control objects.

2.10.7 Fixes

- Retry sqlalchemy backend operations on DatabaseError/OperationalError (Issue #634)
- Tasks that called `retry` was not acknowledged if acks late was enabled

  Fix contributed by David Markey.
- The message priority argument was not properly propagated to Kombu (Issue #708).

  Fix contributed by Eran Rundstein

2.11 What's new in Celery 2.5

Celery aims to be a flexible and reliable, best-of-breed solution to process vast amounts of messages in a distributed fashion, while providing operations with the tools to maintain such a system.

Celery has a large and diverse community of users and contributors, you should come join us *on IRC* or *our mailing-list*.

To read more about Celery you should visit our website.

While this version is backward compatible with previous versions it is important that you read the following section.

If you use Celery in combination with Django you must also read the *django-celery changelog <djcelery:version-2.5.0>* and upgrade to django-celery 2.5.

This version is officially supported on CPython 2.5, 2.6, 2.7, 3.2 and 3.3, as well as PyPy and Jython.

2.11.1 Important Notes

Broker connection pool now enabled by default

The default limit is 10 connections, if you have many threads/green-threads using connections at the same time you may want to tweak this limit to avoid contention.

See the `BROKER_POOL_LIMIT` setting for more information.

Also note that publishing tasks will be retried by default, to change this default or the default retry policy see CELERY_TASK_PUBLISH_RETRY and CELERY_TASK_PUBLISH_RETRY_POLICY.

Rabbit Result Backend: Exchange is no longer *auto delete*

The exchange used for results in the Rabbit (AMQP) result backend used to have the *auto_delete* flag set, which could result in a race condition leading to an annoying warning.

For RabbitMQ users

Old exchanges created with the *auto_delete* flag enabled has to be removed.

The **camqadm** command can be used to delete the previous exchange:

```
$ camqadm exchange.delete celeryresults
```

As an alternative to deleting the old exchange you can configure a new name for the exchange:

```
CELERY_RESULT_EXCHANGE = 'celeryresults2'
```

But you have to make sure that all clients and workers use this new setting, so they are updated to use the same exchange name.

Solution for hanging workers (but must be manually enabled)

The CELERYD_FORCE_EXECV setting has been added to solve a problem with deadlocks that originate when threads and fork is mixed together:

```
CELERYD_FORCE_EXECV = True
```

This setting is recommended for all users using the processes pool, but especially users also using time limits or a max tasks per child setting.

- See Python Issue 6721 to read more about this issue, and why resorting to `execv()` is the only safe solution.

Enabling this option will result in a slight performance penalty when new child worker processes are started, and it will also increase memory usage (but many platforms are optimized, so the impact may be minimal). Considering that it ensures reliability when replacing lost worker processes, it should be worth it.

- It's already the default behavior on Windows.
- It will be the default behavior for all platforms in a future version.

2.11.2 Optimizations

- The code path used when the worker executes a task has been heavily optimized, meaning the worker is able to process a great deal more tasks/second compared to previous versions. As an example the solo pool can now process up to 15000 tasks/second on a 4 core MacBook Pro when using the pylibrabbitmq transport, where it previously could only do 5000 tasks/second.
- The task error tracebacks are now much shorter.
- Fixed a noticeable delay in task processing when rate limits are enabled.

2.11.3 Deprecations

Removals

- The old `TaskSet` signature of `(task_name, list_of_tasks)` can no longer be used (originally scheduled for removal in 2.4). The deprecated `.task_name` and `.task` attributes has also been removed.
- The functions `celery.execute.delay_task`, `celery.execute.apply`, and `celery.execute.apply_async` has been removed (originally) scheduled for removal in 2.3).
- The built-in `ping` task has been removed (originally scheduled for removal in 2.3). Please use the ping broadcast command instead.
- It is no longer possible to import `subtask` and `TaskSet` from `celery.task.base`, please import them from `celery.task` instead (originally scheduled for removal in 2.4).

Deprecations

- The `celery.decorators` module has changed status from pending deprecation to deprecated, and is scheduled for removal in version 4.0. The `celery.task` module must be used instead.

2.11.4 News

Timezone support

Celery can now be configured to treat all incoming and outgoing dates as UTC, and the local timezone can be configured.

This is not yet enabled by default, since enabling time zone support means workers running versions pre 2.5 will be out of sync with upgraded workers.

To enable UTC you have to set `CELERY_ENABLE_UTC`:

```
CELERY_ENABLE_UTC = True
```

When UTC is enabled, dates and times in task messages will be converted to UTC, and then converted back to the local timezone when received by a worker.

You can change the local timezone using the `CELERY_TIMEZONE` setting. Installing the `pytz` library is recommended when using a custom timezone, to keep timezone definition up-to-date, but it will fallback to a system definition of the timezone if available.

UTC will enabled by default in version 3.0.

Note: django-celery will use the local timezone as specified by the `TIME_ZONE` setting, it will also honor the new USE_TZ setting introuced in Django 1.4.

New security serializer using cryptographic signing

A new serializer has been added that signs and verifies the signature of messages.

The name of the new serializer is `auth`, and needs additional configuration to work (see *Security*).

See also:

Security

Contributed by Mher Movsisyan.

Experimental support for automatic module reloading

Starting **celeryd** with the *--autoreload* option will enable the worker to watch for file system changes to all imported task modules imported (and also any non-task modules added to the CELERY_IMPORTS setting or the *-I|--include* option).

This is an experimental feature intended for use in development only, using auto-reload in production is discouraged as the behavior of reloading a module in Python is undefined, and may cause hard to diagnose bugs and crashes. Celery uses the same approach as the auto-reloader found in e.g. the Django `runserver` command.

When auto-reload is enabled the worker starts an additional thread that watches for changes in the file system. New modules are imported, and already imported modules are reloaded whenever a change is detected, and if the processes pool is used the child processes will finish the work they are doing and exit, so that they can be replaced by fresh processes effectively reloading the code.

File system notification backends are pluggable, and Celery comes with three implementations:

- inotify (Linux)

  Used if the `pyinotify` library is installed. If you are running on Linux this is the recommended implementation, to install the `pyinotify` library you have to run the following command:

  ```
  $ pip install pyinotify
  ```

- kqueue (OS X/BSD)
- stat

  The fallback implementation simply polls the files using `stat` and is very expensive.

You can force an implementation by setting the `CELERYD_FSNOTIFY` environment variable:

```
$ env CELERYD_FSNOTIFY=stat celeryd -l info --autoreload
```

Contributed by Mher Movsisyan.

New CELERY_ANNOTATIONS setting

This new setting enables the configuration to modify task classes and their attributes.

The setting can be a dict, or a list of annotation objects that filter for tasks and return a map of attributes to change.

As an example, this is an annotation to change the `rate_limit` attribute for the `tasks.add` task:

```
CELERY_ANNOTATIONS = {'tasks.add': {'rate_limit': '10/s'}}
```

or change the same for all tasks:

```
CELERY_ANNOTATIONS = {'*': {'rate_limit': '10/s'}}
```

You can change methods too, for example the `on_failure` handler:

```
def my_on_failure(self, exc, task_id, args, kwargs, einfo):
    print('Oh no! Task failed: %r' % (exc, ))

CELERY_ANNOTATIONS = {'*': {'on_failure': my_on_failure}}
```

If you need more flexibility then you can also create objects that filter for tasks to annotate:

```
class MyAnnotate(object):

    def annotate(self, task):
        if task.name.startswith('tasks.'):
            return {'rate_limit': '10/s'}

CELERY_ANNOTATIONS = (MyAnnotate(), {...})
```

`current` provides the currently executing task

The new `celery.task.current` proxy will always give the currently executing task.

Example:

```
from celery.task import current, task

@task
def update_twitter_status(auth, message):
    twitter = Twitter(auth)
    try:
        twitter.update_status(message)
    except twitter.FailWhale, exc:
        # retry in 10 seconds.
        current.retry(countdown=10, exc=exc)
```

Previously you would have to type `update_twitter_status.retry(...)` here, which can be annoying for long task names.

Note: This will not work if the task function is called directly, i.e: `update_twitter_status(a, b)`. For that to work `apply` must be used: `update_twitter_status.apply((a, b))`.

In Other News

- Now depends on Kombu 2.1.0.
- Efficient Chord support for the memcached backend (Issue #533)

  This means memcached joins Redis in the ability to do non-polling chords.

  Contributed by Dan McGee.
- Adds Chord support for the Rabbit result backend (amqp)

  The Rabbit result backend can now use the fallback chord solution.
- Sending `QUIT` to celeryd will now cause it cold terminate.

That is, it will not finish executing the tasks it is currently working on.

Contributed by Alec Clowes.

- New "detailed" mode for the Cassandra backend.

  Allows to have a "detailed" mode for the Cassandra backend. Basically the idea is to keep all states using Cassandra wide columns. New states are then appended to the row as new columns, the last state being the last column.

  See the `CASSANDRA_DETAILED_MODE` setting.

  Contributed by Steeve Morin.

- The crontab parser now matches Vixie Cron behavior when parsing ranges with steps (e.g. 1-59/2).

  Contributed by Daniel Hepper.

- celerybeat can now be configured on the command line like celeryd.

  Additional configuration must be added at the end of the argument list followed by `--`, for example:

```
$ celerybeat -l info -- celerybeat.max_loop_interval=10.0
```

- Now limits the number of frames in a traceback so that celeryd does not crash on maximum recursion limit exceeded exceptions (Issue #615).

  The limit is set to the current recursion limit divided by 8 (which is 125 by default).

  To get or set the current recursion limit use `sys.getrecursionlimit()` and `sys.setrecursionlimit()`.

- More information is now preserved in the pickleable traceback.

  This has been added so that Sentry can show more details.

  Contributed by Sean O'Connor.

- CentOS init script has been updated and should be more flexible.

  Contributed by Andrew McFague.

- MongoDB result backend now supports `forget()`.

  Contributed by Andrew McFague

- `task.retry()` now re-raises the original exception keeping the original stack trace.

  Suggested by ojii.

- The *–uid* argument to daemons now uses `initgroups()` to set groups to all the groups the user is a member of.

  Contributed by Łukasz Oleś.

- celeryctl: Added `shell` command.

  The shell will have the current_app (`celery`) and all tasks automatically added to locals.

- celeryctl: Added `migrate` command.

  The migrate command moves all tasks from one broker to another. Note that this is experimental and you should have a backup of the data before proceeding.

  Examples:

```
$ celeryctl migrate redis://localhost amqp://localhost
$ celeryctl migrate amqp://localhost//v1 amqp://localhost//v2
$ python manage.py celeryctl migrate django:// redis://
```

- Routers can now override the `exchange` and `routing_key` used to create missing queues (Issue #577).

  By default this will always use the name of the queue, but you can now have a router return exchange and routing_key keys to set them.

  This is useful when using routing classes which decides a destination at runtime.

  Contributed by Akira Matsuzaki.

- Redis result backend: Adds support for a `max_connections` parameter.

  It is now possible to configure the maximum number of simultaneous connections in the Redis connection pool used for results.

  The default max connections setting can be configured using the `CELERY_REDIS_MAX_CONNECTIONS` setting, or it can be changed individually by `RedisBackend(max_connections=int)`.

  Contributed by Steeve Morin.

- Redis result backend: Adds the ability to wait for results without polling.

  Contributed by Steeve Morin.

- MongoDB result backend: Now supports save and restore taskset.

  Contributed by Julien Poissonnier.

- There's a new *Security* guide in the documentation.

- The init scripts has been updated, and many bugs fixed.

  Contributed by Chris Streeter.

- User (tilde) is now expanded in command line arguments.

- Can now configure CELERYCTL envvar in `/etc/default/celeryd`.

  While not necessary for operation, **celeryctl** is used for the `celeryd status` command, and the path to **celeryctl** must be configured for that to work.

  The daemonization cookbook contains examples.

  Contributed by Jude Nagurney.

- The MongoDB result backend can now use Replica Sets.

  Contributed by Ivan Metzlar.

- gevent: Now supports autoscaling (Issue #599).

  Contributed by Mark Lavin.

- multiprocessing: Mediator thread is now always enabled, even though rate limits are disabled, as the pool semaphore is known to block the main thread, causing broadcast commands and shutdown to depend on the semaphore being released.

2.11.5 Fixes

- Exceptions that are re-raised with a new exception object now keeps the original stack trace.

- Windows: Fixed the `no handlers found for multiprocessing` warning.

- Windows: The `celeryd` program can now be used.

  Previously Windows users had to launch celeryd using `python -m celery.bin.celeryd`.

- Redis result backend: Now uses `SETEX` command to set result key, and expiry atomically.

  Suggested by yaniv-aknin.

- celeryd: Fixed a problem where shutdown hanged when Ctrl+C was used to terminate.

- celeryd: No longer crashes when channel errors occur.

  Fix contributed by Roger Hu.

- Fixed memory leak in the eventlet pool, caused by the use of `greenlet.getcurrent`.

  Fix contributed by Ignas Mikalajnas.

- Cassandra backend: No longer uses `pycassa.connect()` which is deprecated since `pycassa` 1.4.

  Fix contributed by Jeff Terrace.

- Fixed unicode decode errors that could occur while sending error emails.

  Fix contributed by Seong Wun Mun.

- `celery.bin` programs now always defines `__package__` as recommended by PEP-366.

- `send_task` now emits a warning when used in combination with `CELERY_ALWAYS_EAGER` (Issue #581).

  Contributed by Mher Movsisyan.

- `apply_async` now forwards the original keyword arguments to `apply` when `CELERY_ALWAYS_EAGER` is enabled.

- celeryev now tries to re-establish the connection if the connection to the broker is lost (Issue #574).

- celeryev: Fixed a crash occurring if a task has no associated worker information.

  Fix contributed by Matt Williamson.

- The current date and time is now consistently taken from the current loaders `now` method.

- Now shows helpful error message when given a config module ending in `.py` that can't be imported.

- celeryctl: The `--expires` and `-eta` arguments to the apply command can now be an ISO-8601 formatted string.

- celeryctl now exits with exit status `EX_UNAVAILABLE` (69) if no replies have been received.

2.12 Change history

If you're looking for versions prior to 3.0.x you should go to *History*.

2.12.1 3.0.25

release-date 2014-07-10 05:00 P.M UTC

Security Fixes

- [Security: CELERYSA-0002] Insecure default umask.

  The built-in utility used to daemonize the Celery worker service sets an insecure umask by default (umask 0).

  This means that any files or directories created by the worker will end up having world-writable permissions.

  Special thanks to Red Hat for originally discovering and reporting the issue!

  This version will no longer set a default umask by default, so if unset the umask of the parent process will be used.

2.12.2 3.0.24

release-date 2013-10-11 04:40 P.M BST

- Now depends on *Kombu 2.5.15*.
- Now depends on `billiard` version 2.7.3.34.
- AMQP Result backend: No longer caches queue declarations.

  The queues created by the AMQP result backend are always unique, so caching the declarations caused a slow memory leak.
- Worker: Fixed crash when hostname contained Unicode characters.

  Contributed by Daodao.
- The worker would no longer start if the *-P solo* pool was selected (Issue #1548).
- Redis/Cache result backends would not complete chords if any of the tasks were retried (Issue #1401).
- Task decorator is no longer lazy if app is finalized.
- AsyncResult: Fixed bug with `copy(AsyncResult)` when no `current_app` available.
- ResultSet: Now properly propagates app when passed string id's.
- Loader now ignores `CELERY_CONFIG_MODULE` if value is empty string.
- Fixed race condition in Proxy object where it tried to delete an attribute twice, resulting in `AttributeError`.
- Task methods now works with the `CELERY_ALWAYS_EAGER` setting (Issue #1478).
- `Broadcast` queues were accidentally declared when publishing tasks (Issue #1540).
- New `C_FAKEFORK` environment variable can be used to debug the init scripts.

  Setting this will skip the daemonization step so that errors printed to stderr after standard outs are closed can be seen:

  ```
  $ C_FAKEFORK /etc/init.d/celeryd start
  ```

  This works with the *celery multi* command in general.
- `get_pickleable_etype` did not always return a value (Issue #1556).
- Fixed bug where `app.GroupResult.restore` would fall back to the default app.
- Fixed rare bug where built-in tasks would use the current_app.
- `maybe_fileno()` now handles `ValueError`.

2.12.3 3.0.23

release-date 2013-09-02 01:00 P.M BST

- Now depends on *Kombu 2.5.14*.
- `send_task` did not honor `link` and `link_error` arguments.

  This had the side effect of chains not calling unregistered tasks, silently discarding them.

  Fix contributed by Taylor Nelson.
- `celery.state`: Optimized precedence lookup.

  Contributed by Matt Robenolt.
- Posix: Daemonization did not redirect `sys.stdin` to `/dev/null`.

  Fix contributed by Alexander Smirnov.

- Canvas: group bug caused fallback to default app when `.apply_async` used (Issue #1516)
- Canvas: generator arguments was not always pickleable.

2.12.4 3.0.22

release-date 2013-08-16 16:30 P.M BST

- Now depends on *Kombu 2.5.13*.
- Now depends on `billiard` 2.7.3.32
- Fixed bug with monthly and yearly crontabs (Issue #1465).

  Fix contributed by Guillaume Gauvrit.
- Fixed memory leak caused by time limits (Issue #1129, Issue #1427)
- Worker will now sleep if being restarted more than 5 times in one second to avoid spamming with `worker-online` events.
- Includes documentation fixes

  Contributed by: Ken Fromm, Andreas Savvides, Alex Kiriukha, Michael Fladischer.

2.12.5 3.0.21

release-date 2013-07-05 16:30 P.M BST

- Now depends on `billiard` 2.7.3.31.

  This version fixed a bug when running without the billiard C extension.
- 3.0.20 broke eventlet/gevent support (worker not starting).
- Fixed memory leak problem when MongoDB result backend was used with the gevent pool.

  Fix contributed by Ross Lawley.

2.12.6 3.0.20

release-date 2013-06-28 16:00 P.M BST

- Contains workaround for deadlock problems.

  A better solution will be part of Celery 3.1.
- Now depends on *Kombu 2.5.12*.
- Now depends on `billiard` 2.7.3.30.
- `--loader` argument no longer supported importing loaders from the current directory.
- [Worker] Fixed memory leak when restarting after connection lost (Issue #1325).
- [Worker] Fixed UnicodeDecodeError at startup (Issue #1373).

  Fix contributed by Jessica Tallon.
- [Worker] Now properly rewrites unpickleable exceptions again.
- Fixed possible race condition when evicting items from the revoked task set.
- [generic-init.d] Fixed compatibility with Ubuntu's minimal Dash shell (Issue #1387).

Fix contributed by monkut.

- `Task.apply`/`ALWAYS_EAGER` now also executes callbacks and errbacks (Issue #1336).
- [Worker] The `worker-shutdown` signal was no longer being dispatched (Issue #1339)j
- [Python 3] Fixed problem with threading.Event.

  Fix contributed by Xavier Ordoquy.
- [Python 3] Now handles `io.UnsupportedOperation` that may be raised by `file.fileno()` in Python 3.
- [Python 3] Fixed problem with qualname.
- [events.State] Now ignores unknown event-groups.
- [MongoDB backend] No longer uses deprecated `safe` parameter.

  Fix contributed by rfkrocktk
- The eventlet pool now imports on Windows.
- [Canvas] Fixed regression where immutable chord members may receive arguments (Issue #1340).

  Fix contributed by Peter Brook.
- [Canvas] chain now accepts generator argument again (Issue #1319).
- `celery.migrate` command now consumes from all queues if no queues specified.

  Fix contributed by John Watson.

2.12.7 3.0.19

release-date 2013-04-17 04:30:00 P.M BST

- Now depends on `billiard` 2.7.3.28
- A Python 3 related fix managed to disable the deadlock fix announced in 3.0.18.

  Tests have been added to make sure this does not happen again.
- Task retry policy: Default max_retries is now 3.

  This ensures clients will not be hanging while the broker is down.

  Note: You can set a longer retry for the worker by using the `celeryd_after_setup` signal:

```
from celery.signals import celeryd_after_setup

@celeryd_after_setup.connect
def configure_worker(instance, conf, **kwargs):
    conf.CELERY_TASK_PUBLISH_RETRY_POLICY = {
        'max_retries': 100,
        'interval_start': 0,
        'interval_max': 1,
        'interval_step': 0.2,
    }
```

- Worker: Will now properly display message body in error messages even if the body is a buffer instance.
- 3.0.18 broke the MongoDB result backend (Issue #1303).

2.12.8 3.0.18

release-date 2013-04-12 05:00:00 P.M BST

- Now depends on `kombu` 2.5.10.

  See the *kombu changelog*.

- Now depends on `billiard` 2.7.3.27.

- Can now specify a whitelist of accepted serializers using the new `CELERY_ACCEPT_CONTENT` setting.

  This means that you can force the worker to discard messages serialized with pickle and other untrusted serializers. For example to only allow JSON serialized messages use:

  ```
  CELERY_ACCEPT_CONTENT = ['json']
  ```

  you can also specify MIME types in the whitelist:

  ```
  CELERY_ACCEPT_CONTENT = ['application/json']
  ```

- Fixed deadlock in multiprocessing's pool caused by the semaphore not being released when terminated by signal.

- Processes Pool: It's now possible to debug pool processes using GDB.

- `celery report` now censors possibly secret settings, like passwords and secret tokens.

  You should still check the output before pasting anything on the internet.

- Connection URLs now ignore multiple '+' tokens.

- Worker/statedb: Now uses pickle protocol 2 (Py2.5+)

- Fixed Python 3 compatibility issues.

- Worker: A warning is now given if a worker is started with the same node name as an existing worker.

- Worker: Fixed a deadlock that could occur while revoking tasks (Issue #1297).

- Worker: The `HUP` handler now closes all open file descriptors before restarting to ensure file descriptors does not leak (Issue #1270).

- Worker: Optimized storing/loading the revoked tasks list (Issue #1289).

  After this change the `--statedb` file will take up more disk space, but loading from and storing the revoked tasks will be considerably faster (what before took 5 minutes will now take less than a second).

- Celery will now suggest alternatives if there's a typo in the broker transport name (e.g. `ampq` -> `amqp`).

- Worker: The auto-reloader would cause a crash if a monitored file was unlinked.

  Fix contributed by Agris Ameriks.

- Fixed AsyncResult pickling error.

  Fix contributed by Thomas Minor.

- Fixed handling of Unicode in logging output when using log colors (Issue #427).

- `ConfigurationView` is now a `MutableMapping`.

  Contributed by Aaron Harnly.

- Fixed memory leak in LRU cache implementation.

Fix contributed by Romuald Brunet.

- `celery.contrib.rdb`: Now works when sockets are in non-blocking mode.

  Fix contributed by Theo Spears.

- The *inspect reserved* remote control command included active (started) tasks with the reserved tasks (Issue #1030).
- The `task_failure` signal received a modified traceback object meant for pickling purposes, this has been fixed so that it now receives the real traceback instead.
- The `@task` decorator silently ignored positional arguments, it now raises the expected `TypeError` instead (Issue #1125).
- The worker will now properly handle messages with invalid eta/expires fields (Issue #1232).
- The `pool_restart` remote control command now reports an error if the `CELERYD_POOL_RESTARTS` setting is not set.
- `celery.conf.add_defaults` can now be used with non-dict objects.
- Fixed compatibility problems in the Proxy class (Issue #1087).

  The class attributes `__module__`, `__name__` and `__doc__` are now meaningful string objects.

  Thanks to Marius Gedminas.

- MongoDB Backend: The `MONGODB_BACKEND_SETTINGS` setting now accepts a `option` key that lets you forward arbitrary kwargs to the underlying ‘‘pymongo.Connection‘ object (Issue #1015).
- Beat: The daily backend cleanup task is no longer enabled for result backends that support automatic result expiration (Issue #1031).
- Canvas list operations now takes application instance from the first task in the list, instead of depending on the `current_app` (Issue #1249).
- Worker: Message decoding error log message now includes traceback information.
- Worker: The startup banner now includes system platform.
- `celery inspect|status|control` now gives an error if used with an SQL based broker transport.

2.12.9 3.0.17

release-date 2013-03-22 04:00:00 P.M UTC

- Now depends on kombu 2.5.8
- Now depends on billiard 2.7.3.23
- RabbitMQ/Redis: thread-less and lock-free rate-limit implementation.

  This means that rate limits pose minimal overhead when used with RabbitMQ/Redis or future transports using the eventloop, and that the rate-limit implementation is now thread-less and lock-free.

  The thread-based transports will still use the old implementation for now, but the plan is to use the timer also for other broker transports in Celery 3.1.

- Rate limits now works with eventlet/gevent if using RabbitMQ/Redis as the broker.
- A regression caused `task.retry` to ignore additional keyword arguments.

  Extra keyword arguments are now used as execution options again. Fix contributed by Simon Engledew.

- Windows: Fixed problem with the worker trying to pickle the Django settings module at worker startup.
- generic-init.d: No longer double quotes `$CELERYD_CHDIR` (Issue #1235).
- generic-init.d: Removes bash-specific syntax.

  Fix contributed by Pär Wieslander.
- Cassandra Result Backend: Now handles the `AllServersUnavailable` error (Issue #1010).

  Fix contributed by Jared Biel.
- Result: Now properly forwards apps to GroupResults when deserializing (Issue #1249).

  Fix contributed by Charles-Axel Dein.
- `GroupResult.revoke` now supports the `terminate` and `signal` keyword arguments.
- Worker: Multiprocessing pool workers now import task modules/configuration before setting up the logging system so that logging signals can be connected before they're dispatched.
- chord: The `AsyncResult` instance returned now has its `parent` attribute set to the header `GroupResult`.

  This is consistent with how `chain` works.

2.12.10 3.0.16

release-date 2013-03-07 04:00:00 P.M UTC

- Happy International Women's Day!

  We have a long way to go, so this is a chance for you to get involved in one of the organizations working for making our communities more diverse.

  - PyLadies — http://pyladies.com
  - Girls Who Code — http://www.girlswhocode.com
  - Women Who Code — http://www.meetup.com/Women-Who-Code-SF/
- Now depends on `kombu` version 2.5.7
- Now depends on `billiard` version 2.7.3.22
- AMQP heartbeats are now disabled by default.

  Some users experiences issues with heartbeats enabled, and it's not strictly necessary to use them.

  If you're experiencing problems detecting connection failures, you can re-enable heartbeats by configuring the `BROKER_HEARTBEAT` setting.
- Worker: Now propagates connection errors occurring in multiprocessing callbacks, so that the connection can be reset (Issue #1226).
- Worker: Now propagates connection errors occurring in timer callbacks, so that the connection can be reset.
- The modules in `CELERY_IMPORTS` and `CELERY_INCLUDE` are now imported in the original order (Issue #1161).

  The modules in `CELERY_IMPORTS` will be imported first, then continued by `CELERY_INCLUDE`.

  Thanks to Joey Wilhelm.
- New bash completion for `celery` available in the git repository:

https://github.com/celery/celery/tree/3.0/extra/bash-completion

You can source this file or put it in `bash_completion.d` to get auto-completion for the `celery` command-line utility.

- The node name of a worker can now include unicode characters (Issue #1186).
- The repr of a `crontab` object now displays correctly (Issue #972).
- `events.State` no longer modifies the original event dictionary.
- No longer uses `Logger.warn` deprecated in Python 3.
- Cache Backend: Now works with chords again (Issue #1094).
- Chord unlock now handles errors occurring while calling the callback.
- Generic worker init.d script: Status check is now performed by querying the pid of the instance instead of sending messages.

  Contributed by Milen Pavlov.
- Improved init scripts for CentOS.
  - Updated to support celery 3.x conventions.
  - Now uses CentOS built-in `status` and `killproc`
  - Support for multi-node / multi-pid worker services.
  - Standard color-coded CentOS service-init output.
  - A test suite.

  Contributed by Milen Pavlov.
- `ResultSet.join` now always works with empty result set (Issue #1219).
- A `group` consisting of a single task is now supported (Issue #1219).
- Now supports the `pycallgraph` program (Issue #1051).
- Fixed Jython compatibility problems.
- Django tutorial: Now mentions that the example app must be added to `INSTALLED_APPS` (Issue #1192).

2.12.11 3.0.15

release-date 2013-02-11 04:30:00 P.M UTC

- Now depends on billiard 2.7.3.21 which fixed a syntax error crash.
- Fixed bug with `CELERY_SEND_TASK_SENT_EVENT`.

2.12.12 3.0.14

release-date 2013-02-08 05:00:00 P.M UTC

- Now depends on Kombu 2.5.6
- Now depends on billiard 2.7.3.20
- `execv` is now disabled by default.

It was causing too many problems for users, you can still enable it using the CELERYD_FORCE_EXECV setting.

execv was only enabled when transports other than amqp/redis was used, and it's there to prevent deadlocks caused by mutexes not being released before the process forks. Sadly it also changes the environment introducing many corner case bugs that is hard to fix without adding horrible hacks. Deadlock issues are reported far less often than the bugs that execv are causing, so we now disable it by default.

Work is in motion to create non-blocking versions of these transports so that execv is not necessary (which is the situation with the amqp and redis broker transports)

- Chord exception behavior defined (Issue #1172).

  From Celery 3.1 the chord callback will change state to FAILURE when a task part of a chord raises an exception.

  It was never documented what happens in this case, and the actual behavior was very unsatisfactory, indeed it will just forward the exception value to the chord callback.

  For backward compatibility reasons we do not change to the new behavior in a bugfix release, even if the current behavior was never documented. Instead you can enable the CELERY_CHORD_PROPAGATES setting to get the new behavior that will be default from Celery 3.1.

  See more at *Error handling*.

- worker: Fixes bug with ignored and retried tasks.

  The on_chord_part_return and Task.after_return callbacks, nor the task_postrun signal should be called when the task was retried/ignored.

  Fix contributed by Vlad.

- GroupResult.join_native now respects the propagate argument.

- subtask.id added as an alias to subtask['options'].id

```
>>> s = add.s(2, 2)
>>> s.id = 'my-id'
>>> s['options']
{'task_id': 'my-id'}

>>> s.id
'my-id'
```

- worker: Fixed error *Could not start worker processes* occurring when restarting after connection failure (Issue #1118).

- Adds new signal task-retried (Issue #1169).

- *celery events –dumper* now handles connection loss.

- Will now retry sending the task-sent event in case of connection failure.

- amqp backend: Now uses Message.requeue instead of republishing the message after poll.

- New BROKER_HEARTBEAT_CHECKRATE setting introduced to modify the rate at which broker connection heartbeats are monitored.

  The default value was also changed from 3.0 to 2.0.

- celery.events.state.State is now pickleable.

  Fix contributed by Mher Movsisyan.

- `celery.datastructures.LRUCache` is now pickleable.

  Fix contributed by Mher Movsisyan.

- The stats broadcast command now includes the workers pid.

  Contributed by Mher Movsisyan.

- New `conf` remote control command to get a workers current configuration.

  Contributed by Mher Movsisyan.

- Adds the ability to modify the chord unlock task's countdown argument (Issue #1146).

  Contributed by Jun Sakai

- beat: The scheduler now uses the *now()*‘ method of the schedule, so that schedules can provide a custom way to get the current date and time.

  Contributed by Raphaël Slinckx

- Fixed pickling of configuration modules on Windows or when execv is used (Issue #1126).

- Multiprocessing logger is now configured with loglevel `ERROR` by default.

  Since 3.0 the multiprocessing loggers were disabled by default (only configured when the `MP_LOG` environment variable was set).

2.12.13 3.0.13

release-date 2013-01-07 04:00:00 P.M UTC

- Now depends on Kombu 2.5
  - py-amqp has replaced amqplib as the default transport, gaining support for AMQP 0.9, and the RabbitMQ extensions including Consumer Cancel Notifications and heartbeats.
  - support for multiple connection URLs for failover.
  - Read more in the *Kombu 2.5 changelog*.
- Now depends on billiard 2.7.3.19
- Fixed a deadlock issue that could occur when the producer pool inherited the connection pool instance of the parent process.
- The `--loader` option now works again (Issue #1066).
- **celery** umbrella command: All subcommands now supports the `--workdir` option (Issue #1063).
- Groups included in chains now give GroupResults (Issue #1057)

  Previously it would incorrectly add a regular result instead of a group result, but now this works:

```
# [4 + 4, 4 + 8, 16 + 8]
>>> res = (add.s(2, 2) | group(add.s(4), add.s(8), add.s(16)))()
>>> res
<GroupResult: a0acf905-c704-499e-b03a-8d445e6398f7 [
    4346501c-cb99-4ad8-8577-12256c7a22b1,
    b12ead10-a622-4d44-86e9-3193a778f345,
    26c7a420-11f3-4b33-8fac-66cd3b62abfd]>
```

- Chains can now chain other chains and use partial arguments (Issue #1057).

  Example:

```
>>> c1 = (add.s(2) | add.s(4))
>>> c2 = (add.s(8) | add.s(16))

>>> c3 = (c1 | c2)

# 8 + 2 + 4 + 8 + 16
>>> assert c3(8).get() == 38
```

- Subtasks can now be used with unregistered tasks.

  You can specify subtasks even if you just have the name:

  ```
  >>> s = subtask(task_name, args=(), kwargs=())
  >>> s.delay()
  ```

- The **celery shell** command now always adds the current directory to the module path.
- The worker will now properly handle the `pytz.AmbiguousTimeError` exception raised when an ETA/countdown is prepared while being in DST transition (Issue #1061).
- force_execv: Now makes sure that task symbols in the original task modules will always use the correct app instance (Issue #1072).
- AMQP Backend: Now republishes result messages that have been polled (using `result.ready()` and friends, `result.get()` will not do this in this version).
- Crontab schedule values can now "wrap around"

  This means that values like `11-1` translates to `[11, 12, 1]`.

  Contributed by Loren Abrams.

- multi stopwait command now shows the pid of processes.

  Contributed by Loren Abrams.

- **Handling of ETA/countdown fixed when the `CELERY_ENABLE_UTC` setting is disabled (Issue #1065).**
- A number of uneeded properties were included in messages, caused by accidentally passing `Queue.as_dict` as message properties.
- Rate limit values can now be float

  This also extends the string format so that values like `"0.5/s"` works.

  Contributed by Christoph Krybus

- Fixed a typo in the broadcast routing documentation (Issue #1026).
- Rewrote confusing section about idempotence in the task user guide.
- Fixed typo in the daemonization tutorial (Issue #1055).
- Fixed several typos in the documentation.

  Contributed by Marius Gedminas.

- Batches: Now works when using the eventlet pool.

  Fix contributed by Thomas Grainger.

- Batches: Added example sending results to `celery.contrib.batches`.

  Contributed by Thomas Grainger.

- Mongodb backend: Connection `max_pool_size` can now be set in `CELERY_MONGODB_BACKEND_SETTINGS`.

Contributed by Craig Younkins.

- Fixed problem when using earlier versions of `pytz`.

  Fix contributed by Vlad.

- Docs updated to include the default value for the `CELERY_TASK_RESULT_EXPIRES` setting.
- Improvements to the django-celery tutorial.

  Contributed by Locker537.

- The `add_consumer` control command did not properly persist the addition of new queues so that they survived connection failure (Issue #1079).

2.12.14 3.0.12

release-date 2012-11-06 02:00 P.M UTC

- Now depends on kombu 2.4.8
  - [Redis] New and improved fair queue cycle algorithm (Kevin McCarthy).
  - [Redis] Now uses a Redis-based mutex when restoring messages.
  - **[Redis] Number of messages that can be restored in one interval is no** longer limited (but can be set using the `unacked_restore_limit` `transport option`.)
  - Heartbeat value can be specified in broker URLs (Mher Movsisyan).
  - Fixed problem with msgpack on Python 3 (Jasper Bryant-Greene).
- Now depends on billiard 2.7.3.18
- Celery can now be used with static analysis tools like PyDev/PyCharm/pylint etc.
- Development documentation has moved to Read The Docs.

  The new URL is: http://docs.celeryproject.org/en/master

- New `CELERY_QUEUE_HA_POLICY` setting used to set the default HA policy for queues when using RabbitMQ.
- New method `Task.subtask_from_request` returns a subtask using the current request.
- Results get_many method did not respect timeout argument.

  Fix contributed by Remigiusz Modrzejewski

- generic_init.d scripts now support setting `CELERY_CREATE_DIRS` to always create log and pid directories (Issue #1045).

  This can be set in your `/etc/default/celeryd`.

- Fixed strange kombu import problem on Python 3.2 (Issue #1034).
- Worker: ETA scheduler now uses millisecond precision (Issue #1040).
- The `--config` argument to programs is now supported by all loaders.
- The `CASSANDRA_OPTIONS` setting has now been documented.

  Contributed by Jared Biel.

- Task methods (`celery.contrib.methods`) cannot be used with the old task base class, the task decorator in that module now inherits from the new.
- An optimization was too eager and caused some logging messages to never emit.

- `celery.contrib.batches` now works again.
- Fixed missing whitespace in `bdist_rpm` requirements (Issue #1046).
- Event state's `tasks_by_name` applied limit before filtering by name.

  Fix contributed by Alexander A. Sosnovskiy.

2.12.15 3.0.11

release-date 2012-09-26 04:00 P.M UTC

- [security:low] generic-init.d scripts changed permissions of /var/log & /var/run

  In the daemonization tutorial the recommended directories were as follows:

  ```
  CELERYD_LOG_FILE="/var/log/celery/%n.log"
  CELERYD_PID_FILE="/var/run/celery/%n.pid"
  ```

  But in the scripts themselves the default files were `/var/log/celery%n.log` and `/var/run/celery%n.pid`, so if the user did not change the location by configuration, the directories `/var/log` and `/var/run` would be created - and worse have their permissions and owners changed.

  This change means that:

  - Default pid file is `/var/run/celery/%n.pid`
  - Default log file is `/var/log/celery/%n.log`
  - The directories are only created and have their permissions changed if *no custom locations are set*.

  Users can force paths to be created by calling the `create-paths` subcommand:

  ```
  $ sudo /etc/init.d/celeryd create-paths
  ```

  Upgrading Celery will not update init scripts

  To update the init scripts you have to re-download the files from source control and update them manually. You can find the init scripts for version 3.0.x at:

  http://github.com/celery/celery/tree/3.0/extra/generic-init.d

- Now depends on billiard 2.7.3.17
- Fixes request stack protection when app is initialized more than once (Issue #1003).
- ETA tasks now properly works when system timezone is not the same as the configured timezone (Issue #1004).
- Terminating a task now works if the task has been sent to the pool but not yet acknowledged by a pool process (Issue #1007).

  Fix contributed by Alexey Zatelepin

- Terminating a task now properly updates the state of the task to revoked, and sends a `task-revoked` event.
- Generic worker init script now waits for workers to shutdown by default.
- Multi: No longer parses –app option (Issue #1008).
- Multi: stop_verify command renamed to stopwait.

- Daemonization: Now delays trying to create pidfile/logfile until after the working directory has been changed into.
- **celery worker** and **celery beat** commands now respects the `--no-color` option (Issue #999).
- Fixed typos in eventlet examples (Issue #1000)

  Fix contributed by Bryan Bishop. Congratulations on opening bug #1000!
- Tasks that raise `Ignore` are now acknowledged.
- Beat: Now shows the name of the entry in `sending due task` logs.

2.12.16 3.0.10

release-date 2012-09-20 05:30 P.M BST

- Now depends on kombu 2.4.7
- Now depends on billiard 2.7.3.14
  - Fixes crash at startup when using Django and pre-1.4 projects (setup_environ).
  - Hard time limits now sends the KILL signal shortly after TERM, to terminate processes that have signal handlers blocked by C extensions.
  - Billiard now installs even if the C extension cannot be built.

    It's still recommended to build the C extension if you are using a transport other than rabbitmq/redis (or use forced execv for some other reason).
  - Pool now sets a `current_process().index` attribute that can be used to create as many log files as there are processes in the pool.
- Canvas: chord/group/chain no longer modifies the state when called

  Previously calling a chord/group/chain would modify the ids of subtasks so that:

  ```
  >>> c = chord([add.s(2, 2), add.s(4, 4)], xsum.s())
  >>> c()
  >>> c() <-- call again
  ```

  at the second time the ids for the tasks would be the same as in the previous invocation. This is now fixed, so that calling a subtask won't mutate any options.
- Canvas: Chaining a chord to another task now works (Issue #965).
- Worker: Fixed a bug where the request stack could be corrupted if relative imports are used.

  Problem usually manifested itself as an exception while trying to send a failed task result (`NoneType does not have id attribute`).

  Fix contributed by Sam Cooke.
- Tasks can now raise `Ignore` to skip updating states or events after return.

  Example:

  ```
  from celery.exceptions import Ignore

  @task
  def custom_revokes():
      if redis.sismember('tasks.revoked', custom_revokes.request.id):
          raise Ignore()
  ```

- The worker now makes sure the request/task stacks are not modified by the initial `Task.__call__`.

  This would previously be a problem if a custom task class defined `__call__` and also called `super()`.

- Because of problems the fast local optimization has been disabled, and can only be enabled by setting the `USE_FAST_LOCALS` attribute.
- Worker: Now sets a default socket timeout of 5 seconds at shutdown so that broken socket reads do not hinder proper shutdown (Issue #975).
- More fixes related to late eventlet/gevent patching.
- Documentation for settings out of sync with reality:
  - `CELERY_TASK_PUBLISH_RETRY`

    Documented as disabled by default, but it was enabled by default since 2.5 as stated by the 2.5 changelog.

  - `CELERY_TASK_PUBLISH_RETRY_POLICY`

    The default max_retries had been set to 100, but documented as being 3, and the interval_max was set to 1 but documented as 0.2. The default setting are now set to 3 and 0.2 as it was originally documented.

  Fix contributed by Matt Long.
- Worker: Log messages when connection established and lost have been improved.
- The repr of a crontab schedule value of '0' should be '*' (Issue #972).
- Revoked tasks are now removed from reserved/active state in the worker (Issue #969)

  Fix contributed by Alexey Zatelepin.
- gevent: Now supports hard time limits using `gevent.Timeout`.
- Documentation: Links to init scripts now point to the 3.0 branch instead of the development branch (master).
- Documentation: Fixed typo in signals user guide (Issue #986).

  `instance.app.queues` -> `instance.app.amqp.queues`.
- Eventlet/gevent: The worker did not properly set the custom app for new greenlets.
- Eventlet/gevent: Fixed a bug where the worker could not recover from connection loss (Issue #959).

  Also, because of a suspected bug in gevent the `BROKER_CONNECTION_TIMEOUT` setting has been disabled when using gevent

2.12.17 3.0.9

release-date 2012-08-31 06:00 P.M BST

- Important note for users of Django and the database scheduler!

  Recently a timezone issue has been fixed for periodic tasks, but erroneous timezones could have already been stored in the database, so for the fix to work you need to reset the `last_run_at` fields.

  You can do this by executing the following command:

```
$ python manage.py shell
>>> from djcelery.models import PeriodicTask
>>> PeriodicTask.objects.update(last_run_at=None)
```

You also have to do this if you change the timezone or `CELERY_ENABLE_UTC` setting.

- Note about the `CELERY_ENABLE_UTC` setting.

  If you previously disabled this just to force periodic tasks to work with your timezone, then you are now *encouraged to re-enable it*.

- Now depends on Kombu 2.4.5 which fixes PyPy + Jython installation.
- Fixed bug with timezones when `CELERY_ENABLE_UTC` is disabled (Issue #952).
- Fixed a typo in the celerybeat upgrade mechanism (Issue #951).
- Make sure the *exc_info* argument to logging is resolved (Issue #899).
- Fixed problem with Python 3.2 and thread join timeout overflow (Issue #796).
- A test case was occasionally broken for Python 2.5.
- Unit test suite now passes for PyPy 1.9.
- App instances now supports the with statement.

  This calls the new `close()` method at exit, which cleans up after the app like closing pool connections.

  Note that this is only necessary when dynamically creating apps, e.g. for "temporary" apps.

- Support for piping a subtask to a chain.

  For example:

  ```
  pipe = sometask.s() | othertask.s()
  new_pipe = mytask.s() | pipe
  ```

  Contributed by Steve Morin.

- Fixed problem with group results on non-pickle serializers.

  Fix contributed by Steeve Morin.

2.12.18 3.0.8

release-date 2012-08-29 05:00 P.M BST

- Now depends on Kombu 2.4.4
- Fixed problem with amqplib and receiving larger message payloads (Issue #922).

  The problem would manifest itself as either the worker hanging, or occasionally a `Framing error` exception appearing.

  Users of the new `pyamqp://` transport must upgrade to `amqp` 0.9.3.

- Beat: Fixed another timezone bug with interval and crontab schedules (Issue #943).
- Beat: The schedule file is now automatically cleared if the timezone is changed.

  The schedule is also cleared when you upgrade to 3.0.8 from an earlier version, this to register the initial timezone info.

- Events: The `worker-heartbeat` event now include processed and active count fields.

Contributed by Mher Movsisyan.

- Fixed error with error email and new task classes (Issue #931).
- `BaseTask.__call__` is no longer optimized away if it has been monkey patched.
- Fixed shutdown issue when using gevent (Issue #911 & Issue #936).

  Fix contributed by Thomas Meson.

2.12.19 3.0.7

release-date 2012-08-24 05:00 P.M BST

- Fixes several problems with periodic tasks and timezones (Issue #937).
- Now depends on kombu 2.4.2
  - Redis: Fixes a race condition crash
  - Fixes an infinite loop that could happen when retrying establishing the broker connection.
- Daemons now redirect standard file descriptors to `/dev/null`

  Though by default the standard outs are also redirected to the logger instead, but you can disable this by changing the `CELERY_REDIRECT_STDOUTS` setting.
- Fixes possible problems when eventlet/gevent is patched too late.
- `LoggingProxy` no longer defines `fileno()` (Issue #928).
- Results are now ignored for the chord unlock task.

  Fix contributed by Steeve Morin.
- Cassandra backend now works if result expiry is disabled.

  Fix contributed by Steeve Morin.
- The traceback object is now passed to signal handlers instead of the string representation.

  Fix contributed by Adam DePue.
- Celery command: Extensions are now sorted by name.
- A regression caused the `task-failed` event to be sent with the exception object instead of its string representation.
- The worker daemon would try to create the pid file before daemonizing to catch errors, but this file was not immediately released (Issue #923).
- Fixes Jython compatibility.
- `billiard.forking_enable` was called by all pools not just the processes pool, which would result in a useless warning if the billiard C extensions were not installed.

2.12.20 3.0.6

release-date 2012-08-17 11:00 P.M BST

- Now depends on kombu 2.4.0
- Now depends on billiard 2.7.3.12
- Redis: Celery now tries to restore messages whenever there are no messages in the queue.

- Crontab schedules now properly respects `CELERY_TIMEZONE` setting.

  It's important to note that crontab schedules uses UTC time by default unless this setting is set.

  Issue #904 and django-celery #150.

- `billiard.enable_forking` is now only set by the processes pool.
- The transport is now properly shown by **celery report** (Issue #913).
- The *–app* argument now works if the last part is a module name (Issue #921).
- Fixed problem with unpickleable exceptions (billiard #12).
- Adds `task_name` attribute to `EagerResult` which is always `None` (Issue #907).
- Old Task class in `celery.task` no longer accepts magic kwargs by default (Issue #918).

  A regression long ago disabled magic kwargs for these, and since no one has complained about it we don't have any incentive to fix it now.

- The `inspect reserved` control command did not work properly.
- Should now play better with static analyzation tools by explicitly specifying dynamically created attributes in the `celery` and `celery.task` modules.
- Terminating a task now results in `RevokedTaskError` instead of a `WorkerLostError`.
- `AsyncResult.revoke` now accepts `terminate` and `signal` arguments.
- The `task-revoked` event now includes new fields: `terminated`, `signum`, and `expired`.
- The argument to `TaskRevokedError` is now one of the reasons `revoked`, `expired` or `terminated`.
- Old Task class does no longer use classmethods for push_request and pop_request (Issue #912).
- `GroupResult` now supports the `children` attribute (Issue #916).
- `AsyncResult.collect` now respects the `intermediate` argument (Issue #917).
- Fixes example task in documentation (Issue #902).
- Eventlet fixed so that the environment is patched as soon as possible.
- eventlet: Now warns if celery related modules that depends on threads are imported before eventlet is patched.
- Improved event and camera examples in the monitoring guide.
- Disables celery command setuptools entrypoints if the command can't be loaded.
- Fixed broken `dump_request` example in the tasks guide.

2.12.21 3.0.5

release-date 2012-08-01 04:00 P.M BST

- Now depends on kombu 2.3.1 + billiard 2.7.3.11
- Fixed a bug with the -B option (`cannot pickle thread.lock objects`) (Issue #894 + Issue #892, + django-celery #154).
- The `restart_pool` control command now requires the `CELERYD_POOL_RESTARTS` setting to be enabled

  This change was necessary as the multiprocessing event that the restart command depends on is responsible for creating many semaphores/file descriptors, resulting in problems in some environments.

- `chain.apply` now passes args to the first task (Issue #889).

- Documented previously secret options to the Django-Celery monitor in the monitoring userguide (Issue #396).
- Old changelog are now organized in separate documents for each series, see *History*.

2.12.22 3.0.4

release-date 2012-07-26 07:00 P.M BST

- Now depends on Kombu 2.3
- New experimental standalone Celery monitor: Flower

  See *Flower: Real-time Celery web-monitor* to read more about it!

  Contributed by Mher Movsisyan.
- Now supports AMQP heartbeats if using the new `pyamqp://` transport.
  - The py-amqp transport requires the `amqp` library to be installed:

    ```
    $ pip install amqp
    ```

  - Then you need to set the transport URL prefix to `pyamqp://`.
  - The default heartbeat value is 10 seconds, but this can be changed using the `BROKER_HEARTBEAT` setting:

    ```
    BROKER_HEARTBEAT = 5.0
    ```

  - If the broker heartbeat is set to 10 seconds, the heartbeats will be monitored every 5 seconds (double the hertbeat rate).

  See the *Kombu 2.3 changelog* for more information.
- Now supports RabbitMQ Consumer Cancel Notifications, using the `pyamqp://` transport.

  This is essential when running RabbitMQ in a cluster.

  See the *Kombu 2.3 changelog* for more information.
- Delivery info is no longer passed directly through.

  It was discovered that the SQS transport adds objects that can't be pickled to the delivery info mapping, so we had to go back to using the whitelist again.

  Fixing this bug also means that the SQS transport is now working again.
- The semaphore was not properly released when a task was revoked (Issue #877).

  This could lead to tasks being swallowed and not released until a worker restart.

  Thanks to Hynek Schlawack for debugging the issue.
- Retrying a task now also forwards any linked tasks.

  This means that if a task is part of a chain (or linked in some other way) and that even if the task is retried, then the next task in the chain will be executed when the retry succeeds.
- Chords: Now supports setting the interval and other keyword arguments to the chord unlock task.
  - The interval can now be set as part of the chord subtasks kwargs:

    ```
    chord(header)(body, interval=10.0)
    ```

  - In addition the chord unlock task now honors the Task.default_retry_delay option, used when none is specified, which also means that the default interval can also be changed using annotations:

```
CELERY_ANNOTATIONS = {
    'celery.chord_unlock': {
        'default_retry_delay': 10.0,
    }
}
```

- New `Celery.add_defaults()` method can add new default configuration dicts to the applications configuration.

  For example:

  ```
  config = {'FOO': 10}

  celery.add_defaults(config)
  ```

  is the same as `celery.conf.update(config)` except that data will not be copied, and that it will not be pickled when the worker spawns child processes.

  In addition the method accepts a callable:

  ```
  def initialize_config():
      # insert heavy stuff that can't be done at import time here.

  celery.add_defaults(initialize_config)
  ```

  which means the same as the above except that it will not happen until the celery configuration is actually used.

  As an example, Celery can lazily use the configuration of a Flask app:

  ```
  flask_app = Flask()
  celery = Celery()
  celery.add_defaults(lambda: flask_app.config)
  ```

- Revoked tasks were not marked as revoked in the result backend (Issue #871).

  Fix contributed by Hynek Schlawack.

- Eventloop now properly handles the case when the epoll poller object has been closed (Issue #882).
- Fixed syntax error in `funtests/test_leak.py`

  Fix contributed by Catalin Iacob.

- group/chunks: Now accepts empty task list (Issue #873).
- New method names:
  - `Celery.default_connection()` `connection_or_acquire()`.
  - `Celery.default_producer()` `producer_or_acquire()`.

  The old names still work for backward compatibility.

2.12.23 3.0.3

release-date 2012-07-20 09:17 P.M BST

by Ask Solem

- amqplib passes the channel object as part of the delivery_info and it's not pickleable, so we now remove it.

2.12.24 3.0.2

release-date 2012-07-20 04:00 P.M BST

by Ask Solem

- **A bug caused the following task options to not take defaults from the** configuration (Issue #867 + Issue #858)

  The following settings were affected:

  - `CELERY_IGNORE_RESULT`
  - `CELERYD_SEND_TASK_ERROR_EMAILS`
  - `CELERY_TRACK_STARTED`
  - `CElERY_STORE_ERRORS_EVEN_IF_IGNORED`

  Fix contributed by John Watson.

- Task Request: `delivery_info` is now passed through as-is (Issue #807).
- The eta argument now supports datetime's with a timezone set (Issue #855).
- The worker's banner displayed the autoscale settings in the wrong order (Issue #859).
- Extension commands are now loaded after concurrency is set up so that they don't interfere with e.g. eventlet patching.
- Fixed bug in the threaded pool (Issue #863)
- The task failure handler mixed up the fields in `sys.exc_info()`.

  Fix contributed by Rinat Shigapov.

- Fixed typos and wording in the docs.

  Fix contributed by Paul McMillan

- New setting: `CELERY_WORKER_DIRECT`

  If enabled each worker will consume from their own dedicated queue which can be used to route tasks to specific workers.

- Fixed several edge case bugs in the add consumer remote control command.
- `migrate`: Can now filter and move tasks to specific workers if `CELERY_WORKER_DIRECT` is enabled.

  Among other improvements, the following functions have been added:

  - `move_direct(filterfun, **opts)`
  - `move_direct_by_id(task_id, worker_hostname, **opts)`
  - `move_direct_by_idmap({task_id: worker_hostname, ...}, **opts)`
  - `move_direct_by_taskmap({task_name: worker_hostname, ...}, **opts)`

- `default_connection()` now accepts a pool argument that if set to false causes a new connection to be created instead of acquiring one from the pool.
- New signal: `celeryd_after_setup`.
- Default loader now keeps lowercase attributes from the configuration module.

2.12.25 3.0.1

release-date 2012-07-10 06:00 P.M BST

by Ask Solem

- Now depends on kombu 2.2.5
- inspect now supports limit argument:

```
myapp.control.inspect(limit=1).ping()
```

- Beat: now works with timezone aware datetime's.
- Task classes inheriting `from celery import Task` mistakingly enabled `accept_magic_kwargs`.
- Fixed bug in `inspect scheduled` (Issue #829).
- Beat: Now resets the schedule to upgrade to UTC.
- The **celery worker** command now works with eventlet/gevent.

  Previously it would not patch the environment early enough.
- The **celery** command now supports extension commands using setuptools entry-points.

  Libraries can add additional commands to the **celery** command by adding an entry-point like:

```
setup(
    entry_points=[
        'celery.commands': [
            'foo = my.module:Command',
    ],
],
...)
```

  The command must then support the interface of `celery.bin.base.Command`.
- contrib.migrate: New utilities to move tasks from one queue to another.
  - `move_tasks()`
  - `move_task_by_id()`
- The `task-sent` event now contains `exchange` and `routing_key` fields.
- Fixes bug with installing on Python 3.

  Fix contributed by Jed Smith.

2.12.26 3.0.0 (Chiastic Slide)

release-date 2012-07-07 01:30 P.M BST

by Ask Solem

See *What's new in Celery 3.0 (Chiastic Slide)*.

2.13 API Reference

Release 3.0

Date July 10, 2014

2.13.1 celery

Application

class `celery.`**`Celery`**(*main='__main__'*, *broker='amqp://localhost//'*, *...*)

Parameters

- **main** – Name of the main module if running as *__main__*.
- **broker** – URL of the default broker used.
- **loader** – The loader class, or the name of the loader class to use. Default is `celery.loaders.app.AppLoader`.
- **backend** – The result store backend class, or the name of the backend class to use. Default is the value of the `CELERY_RESULT_BACKEND` setting.
- **amqp** – AMQP object or class name.
- **events** – Events object or class name.
- **log** – Log object or class name.
- **control** – Control object or class name.
- **set_as_current** – Make this the global current app.
- **tasks** – A task registry or the name of a registry class.

`main`
Name of the *__main__* module. Required for standalone scripts.

If set this will be used instead of *__main__* when automatically generating task names.

`conf`
Current configuration.

`current_task`
The instance of the task that is being executed, or `None`.

`amqp`
AMQP related functionality: `amqp`.

`backend`
Current backend instance.

`loader`
Current loader instance.

`control`
Remote control: `control`.

`events`
Consuming and sending events: `events`.

`log`
Logging: `log`.

tasks
Task registry.

Accessing this attribute will also finalize the app.

pool
Broker connection pool: `pool`. This attribute is not related to the workers concurrency pool.

Task
Base task class for this app.

close()
Cleans-up after application, like closing any pool connections. Only necessary for dynamically created apps for which you can use the with statement:

```
with Celery(set_as_current=False) as app:
    with app.connection() as conn:
        pass
```

bugreport()
Returns a string with information useful for the Celery core developers when reporting a bug.

config_from_object(*obj*, *silent=False*)
Reads configuration from object, where object is either an object or the name of a module to import.

Parameters **silent** – If true then import errors will be ignored.

```
>>> celery.config_from_object("myapp.celeryconfig")

>>> from myapp import celeryconfig
>>> celery.config_from_object(celeryconfig)
```

config_from_envvar(*variable_name*, *silent=False*)
Read configuration from environment variable.

The value of the environment variable must be the name of a module to import.

```
>>> os.environ["CELERY_CONFIG_MODULE"] = "myapp.celeryconfig"
>>> celery.config_from_envvar("CELERY_CONFIG_MODULE")
```

add_defaults(*d*)
Add default configuration from dict `d`.

If the argument is a callable function then it will be regarded as a promise, and it won't be loaded until the configuration is actually needed.

This method can be compared to:

```
>>> celery.conf.update(d)
```

with a difference that 1) no copy will be made and 2) the dict will not be transferred when the worker spawns child processes, so it's important that the same configuration happens at import time when pickle restores the object on the other side.

start(*argv=None*)
Run **celery** using *argv*.

Uses `sys.argv` if *argv* is not specified.

task(*fun*, ...)
Decorator to create a task class out of any callable.

Examples:

```
@celery.task
def refresh_feed(url):
    return ...
```

with setting extra options:

```
@celery.task(exchange="feeds")
def refresh_feed(url):
    return ...
```

App Binding

For custom apps the task decorator returns proxy objects, so that the act of creating the task is not performed until the task is used or the task registry is accessed.

If you are depending on binding to be deferred, then you must not access any attributes on the returned object until the application is fully set up (finalized).

---

send_task(*name*[, *args*[, *kwargs*[, ...]]])
Send task by name.

Parameters

- **name** – Name of task to call (e.g. *"tasks.add"*).
- **result_cls** – Specify custom result class. Default is using `AsyncResult()`.

Otherwise supports the same arguments as `Task.apply_async()`.

AsyncResult
Create new result instance. See `AsyncResult`.

GroupResult
Create new taskset result instance. See `GroupResult`.

worker_main(*argv=None*)
Run **celeryd** using *argv*.

Uses `sys.argv` if *argv* is not specified."""

Worker
Worker application. See `Worker`.

WorkController
Embeddable worker. See `WorkController`.

Beat
Celerybeat scheduler application. See `Beat`.

connection(*url=default*[, *ssl*[, *transport_options={}*]])
Establish a connection to the message broker.

Parameters

- **url** – Either the URL or the hostname of the broker to use.
- **hostname** – URL, Hostname/IP-address of the broker. If an URL is used, then the other argument below will be taken from the URL instead.
- **userid** – Username to authenticate as.
- **password** – Password to authenticate with
- **virtual_host** – Virtual host to use (domain).

- **port** – Port to connect to.
- **ssl** – Defaults to the BROKER_USE_SSL setting.
- **transport** – defaults to the BROKER_TRANSPORT setting.

:returns kombu.connection.Connection:

connection_or_acquire(*connection=None*)
For use within a with-statement to get a connection from the pool if one is not already provided.

Parameters **connection** – If not provided, then a connection will be acquired from the connection pool.

producer_or_acquire(*producer=None*)
For use within a with-statement to get a producer from the pool if one is not already provided

Parameters **producer** – If not provided, then a producer will be acquired from the producer pool.

mail_admins(*subject*, *body*, *fail_silently=False*)
Sends an email to the admins in the ADMINS setting.

select_queues(*queues=[]*)
Select a subset of queues, where queues must be a list of queue names to keep.

now()
Returns the current time and date as a datetime object.

set_current()
Makes this the current app for this thread.

finalize()
Finalizes the app by loading built-in tasks, and evaluating pending task decorators

Pickler
Helper class used to pickle this application.

Grouping Tasks

class celery.**group**(*task1*[, *task2*[, *task3*[, ... *taskN*]]])
Creates a group of tasks to be executed in parallel.

Example:

```
>>> res = group([add.s(2, 2), add.s(4, 4)]).apply_async()
>>> res.get()
[4, 8]
```

The apply_async method returns GroupResult.

class celery.**chain**(*task1*[, *task2*[, *task3*[, ... *taskN*]]])
Chains tasks together, so that each tasks follows each other by being applied as a callback of the previous task.

If called with only one argument, then that argument must be an iterable of tasks to chain.

Example:

```
>>> res = chain(add.s(2, 2), add.s(4)).apply_async()
```

is effectively $(2 + 2) + 4$):

```
>>> res.get()
8
```

Calling a chain will return the result of the last task in the chain. You can get to the other tasks by following the `result.parent`'s:

```
>>> res.parent.get()
4
```

class `celery.`**`chord`**(*header*[, *body*])

A chord consists of a header and a body. The header is a group of tasks that must complete before the callback is called. A chord is essentially a callback for a group of tasks.

Example:

```
>>> res = chord([add.s(2, 2), add.s(4, 4)])(sum_task.s())
```

is effectively $\Sigma((2+2)+(4+4))$:

```
>>> res.get()
12
```

The body is applied with the return values of all the header tasks as a list.

class `celery.`**`subtask`**(*task=None*, *args=()*, *kwargs={}*, *options={}*)

Describes the arguments and execution options for a single task invocation.

Used as the parts in a `group` or to safely pass tasks around as callbacks.

Subtasks can also be created from tasks:

```
>>> add.subtask(args=(), kwargs={}, options={})
```

or the `.s()` shortcut:

```
>>> add.s(*args, **kwargs)
```

Parameters

- **task** – Either a task class/instance, or the name of a task.
- **args** – Positional arguments to apply.
- **kwargs** – Keyword arguments to apply.
- **options** – Additional options to `Task.apply_async()`.

Note that if the first argument is a `dict`, the other arguments will be ignored and the values in the dict will be used instead.

```
>>> s = subtask("tasks.add", args=(2, 2))
>>> subtask(s)
{"task": "tasks.add", args=(2, 2), kwargs={}, options={}}
```

`delay`(**args*, ***kwargs*)

Shortcut to `apply_async()`.

`apply_async`(*args=()*, *kwargs={}*, ...)

Apply this task asynchronously.

Parameters

- **args** – Partial args to be prepended to the existing args.

- **kwargs** – Partial kwargs to be merged with the existing kwargs.
- **options** – Partial options to be merged with the existing options.

See `apply_async()`.

apply(*args=()*, *kwargs={}*, ...)
Same as `apply_async()` but executed the task inline instead of sending a task message.

clone(*args=()*, *kwargs={}*, ...)
Returns a copy of this subtask.

Parameters

- **args** – Partial args to be prepended to the existing args.
- **kwargs** – Partial kwargs to be merged with the existing kwargs.
- **options** – Partial options to be merged with the existing options.

replace(*args=None*, *kwargs=None*, *options=None*)
Replace the args, kwargs or options set for this subtask. These are only replaced if the selected is not `None`.

link(*other_subtask*)
Add a callback task to be applied if this task executes successfully.

Returns `other_subtask` (to work with `reduce()`).

link_error(*other_subtask*)
Add a callback task to be applied if an error occurs while executing this task.

Returns `other_subtask` (to work with `reduce()`)

set(...)
Set arbitrary options (same as `.options.update(...)`).

This is a chaining method call (i.e. it returns itself).

flatten_links()
Gives a recursive list of dependencies (unchain if you will, but with links intact).

Proxies

celery.**current_app**
The currently set app for this thread.

celery.**current_task**
The task currently being executed (only set in the worker, or when eager/apply is used).

2.13.2 celery.app

Celery Application.

Proxies

celery.app.**default_app = <Celery default:0x7fb766b79910>**

Functions

celery.app.**app_or_default**(*app=None*)
Function returning the app provided or the default app if none.

The environment variable CELERY_TRACE_APP is used to trace app leaks. When enabled an exception is raised if there is no active app.

celery.app.**enable_trace**()

celery.app.**disable_trace**()

Data

celery.app.**default_loader = 'default'**
The 'default' loader is the default loader used by old applications. This is deprecated and should no longer be used as it's set too early to be affected by –loader argument.

2.13.3 celery.app.task

celery.app.task

Task Implementation: Task request context, and the base task class.

class celery.app.task.**Task**
Task base class.

When called tasks apply the run() method. This method must be defined by all tasks (that is unless the __call__() method is overridden).

AsyncResult(*task_id*, ***kwargs*)
Get AsyncResult instance for this kind of task.

Parameters **task_id** – Task id to get result for.

class ErrorMail(*task*, ***kwargs*)
Defines how and when task error e-mails should be sent.

Parameters **task** – The task instance that raised the error.

subject and body are format strings which are passed a context containing the following keys:

•name

Name of the task.

•id

UUID of the task.

- exc

  String representation of the exception.

- args

  Positional arguments.

- kwargs

  Keyword arguments.

- traceback

  String representation of the traceback.

- hostname

  Worker hostname.

should_send(*context*, *exc*)
Returns true or false depending on if a task error mail should be sent for this type of error.

exception Task.**MaxRetriesExceededError**
The tasks max restart limit has been exceeded.

Task.**Strategy = 'celery.worker.strategy:default'**
Execution strategy used, or the qualified name of one.

Task.**abstract = None**
If True the task is an abstract base class.

Task.**accept_magic_kwargs = False**
If disabled the worker will not forward magic keyword arguments. Deprecated and scheduled for removal in v4.0.

Task.**acks_late = False**
When enabled messages for this task will be acknowledged **after** the task has been executed, and not *just before* which is the default behavior.

Please note that this means the task may be executed twice if the worker crashes mid execution (which may be acceptable for some applications).

The application default can be overridden with the CELERY_ACKS_LATE setting.

Task.**after_return**(*status*, *retval*, *task_id*, *args*, *kwargs*, *einfo*)
Handler called after the task returns.

Parameters

- **status** – Current task state.
- **retval** – Task return value/exception.
- **task_id** – Unique id of the task.
- **args** – Original arguments for the task that failed.
- **kwargs** – Original keyword arguments for the task that failed.
- **einfo** – ExceptionInfo instance, containing the traceback (if any).

The return value of this handler is ignored.

Task.**apply**(*args=None*, *kwargs=None*, *link=None*, *link_error=None*, ***options*)
Execute this task locally, by blocking until the task returns.

Parameters

- **args** – positional arguments passed on to the task.
- **kwargs** – keyword arguments passed on to the task.
- **throw** – Re-raise task exceptions. Defaults to the CELERY_EAGER_PROPAGATES_EXCEPTIONS setting.

:rtype celery.result.EagerResult:

Task.**apply_async**(*args=None*, *kwargs=None*, *task_id=None*, *producer=None*, *connection=None*, *router=None*, *link=None*, *link_error=None*, *publisher=None*, *add_to_parent=True*, ***options*)

Apply tasks asynchronously by sending a message.

Parameters

- **args** – The positional arguments to pass on to the task (a list or tuple).
- **kwargs** – The keyword arguments to pass on to the task (a dict)
- **countdown** – Number of seconds into the future that the task should execute. Defaults to immediate execution (do not confuse with the *immediate* flag, as they are unrelated).
- **eta** – A datetime object describing the absolute time and date of when the task should be executed. May not be specified if *countdown* is also supplied. (Do not confuse this with the *immediate* flag, as they are unrelated).
- **expires** – Either a int, describing the number of seconds, or a datetime object that describes the absolute time and date of when the task should expire. The task will not be executed after the expiration time.
- **connection** – Re-use existing broker connection instead of establishing a new one.
- **retry** – If enabled sending of the task message will be retried in the event of connection loss or failure. Default is taken from the CELERY_TASK_PUBLISH_RETRY setting. Note you need to handle the producer/connection manually for this to work.
- **retry_policy** – Override the retry policy used. See the CELERY_TASK_PUBLISH_RETRY setting.
- **routing_key** – Custom routing key used to route the task to a worker server. If in combination with a queue argument only used to specify custom routing keys to topic exchanges.
- **queue** – The queue to route the task to. This must be a key present in CELERY_QUEUES, or CELERY_CREATE_MISSING_QUEUES must be enabled. See *Routing Tasks* for more information.
- **exchange** – Named custom exchange to send the task to. Usually not used in combination with the queue argument.
- **priority** – The task priority, a number between 0 and 9. Defaults to the priority attribute.
- **serializer** – A string identifying the default serialization method to use. Can be *pickle*, *json*, *yaml*, *msgpack* or any custom serialization method that has been registered with kombu.serialization.registry. Defaults to the serializer attribute.
- **compression** – A string identifying the compression method to use. Can be one of zlib, bzip2, or any custom compression methods registered with kombu.compression.register(). Defaults to the CELERY_MESSAGE_COMPRESSION setting.

- **link** – A single, or a list of subtasks to apply if the task exits successfully.
- **link_error** – A single, or a list of subtasks to apply if an error occurs while executing the task.
- **producer** – :class:~@amqp.TaskProducer‘ instance to use.
- **add_to_parent** – If set to True (default) and the task is applied while executing another task, then the result will be appended to the parent tasks `request.children` attribute.
- **publisher** – Deprecated alias to `producer`.

Also supports all keyword arguments supported by `kombu.messaging.Producer.publish()`.

Note: If the `CELERY_ALWAYS_EAGER` setting is set, it will be replaced by a local `apply()` call instead.

`Task.`**`autoregister = True`**
If disabled this task won't be registered automatically.

`Task.`**`backend = <celery.backends.cache.CacheBackend object at 0x7fb767118c10>`**
The result store backend used for this task.

`Task.`**`chunks`**(*it*, *n*)
Creates a `chunks` task for this task.

`Task.`**`default_retry_delay = 180`**
Default time in seconds before a retry of the task should be executed. 3 minutes by default.

`Task.`**`delay`**(**args*, ***kwargs*)
Star argument version of `apply_async()`.

Does not support the extra options enabled by `apply_async()`.

Parameters

- ***args** – positional arguments passed on to the task.
- ****kwargs** – keyword arguments passed on to the task.

:returns `celery.result.AsyncResult`:

`Task.`**`expires = None`**
Default task expiry time.

`Task.`**`ignore_result = False`**
If enabled the worker will not store task state and return values for this task. Defaults to the `CELERY_IGNORE_RESULT` setting.

`Task.`**`map`**(*it*)
Creates a `xmap` task from `it`.

`Task.`**`max_retries = 3`**
Maximum number of retries before giving up. If set to `None`, it will **never** stop retrying.

`Task.`**`name = None`**
Name of the task.

classmethod `Task.`**`on_bound`**(*app*)
This method can be defined to do additional actions when the task class is bound to an app.

`Task.`**`on_failure`**(*exc*, *task_id*, *args*, *kwargs*, *einfo*)
Error handler.

This is run by the worker when the task fails.

Parameters

- **exc** – The exception raised by the task.
- **task_id** – Unique id of the failed task.
- **args** – Original arguments for the task that failed.
- **kwargs** – Original keyword arguments for the task that failed.
- **einfo** – `ExceptionInfo` instance, containing the traceback.

The return value of this handler is ignored.

`Task.`**`on_retry`**(*exc*, *task_id*, *args*, *kwargs*, *einfo*)
Retry handler.

This is run by the worker when the task is to be retried.

Parameters

- **exc** – The exception sent to `retry()`.
- **task_id** – Unique id of the retried task.
- **args** – Original arguments for the retried task.
- **kwargs** – Original keyword arguments for the retried task.
- **einfo** – `ExceptionInfo` instance, containing the traceback.

The return value of this handler is ignored.

`Task.`**`on_success`**(*retval*, *task_id*, *args*, *kwargs*)
Success handler.

Run by the worker if the task executes successfully.

Parameters

- **retval** – The return value of the task.
- **task_id** – Unique id of the executed task.
- **args** – Original arguments for the executed task.
- **kwargs** – Original keyword arguments for the executed task.

The return value of this handler is ignored.

`Task.`**`rate_limit = None`**
Rate limit for this task type. Examples: `None` (no rate limit), *'100/s'* (hundred tasks a second), *'100/m'* (hundred tasks a minute),"100/h'' (hundred tasks an hour)

`Task.`**`request`**
Get current request object.

`Task.`**`retry`**(*args=None*, *kwargs=None*, *exc=None*, *throw=True*, *eta=None*, *countdown=None*, *max_retries=None*, ***options*)
Retry the task.

Parameters

- **args** – Positional arguments to retry with.
- **kwargs** – Keyword arguments to retry with.

- **exc** – Custom exception to report when the max restart limit has been exceeded (default: `MaxRetriesExceededError`).

  If this argument is set and retry is called while an exception was raised (`sys.exc_info()` is set) it will attempt to reraise the current exception.

  If no exception was raised it will raise the `exc` argument provided.

- **countdown** – Time in seconds to delay the retry for.
- **eta** – Explicit time and date to run the retry at (must be a `datetime` instance).
- **max_retries** – If set, overrides the default retry limit.
- ****options** – Any extra options to pass on to meth:*apply_async*.
- **throw** – If this is `False`, do not raise the `RetryTaskError` exception, that tells the worker to mark the task as being retried. Note that this means the task will be marked as failed if the task raises an exception, or successful if it returns.

Raises **celery.exceptions.RetryTaskError** To tell the worker that the task has been re-sent for retry. This always happens, unless the *throw* keyword argument has been explicitly set to `False`, and is considered normal operation.

Example

```
>>> @task()
>>> def tweet(auth, message):
...     twitter = Twitter(oauth=auth)
...     try:
...         twitter.post_status_update(message)
...     except twitter.FailWhale, exc:
...         # Retry in 5 minutes.
...         raise tweet.retry(countdown=60 * 5, exc=exc)
```

Although the task will never return above as *retry* raises an exception to notify the worker, we use *return* in front of the retry to convey that the rest of the block will not be executed.

`Task.`**`run`**(**args*, ***kwargs*)
The body of the task executed by workers.

`Task.`**`s`**(**args*, ***kwargs*)
`.s(*a, **k) -> .subtask(a, k)`

`Task.`**`send_error_emails = False`**
If enabled an email will be sent to `ADMINS` whenever a task of this type fails.

`Task.`**`serializer = 'pickle'`**
The name of a serializer that are registered with `kombu.serialization.registry`. Default is *'pickle'*.

`Task.`**`si`**(**args*, ***kwargs*)
`.si(*a, **k) -> .subtask(a, k, immutable=True)`

`Task.`**`soft_time_limit = None`**
Soft time limit. Defaults to the `CELERYD_TASK_SOFT_TIME_LIMIT` setting.

`Task.`**`starmap`**(*it*)
Creates a `xstarmap` task from `it`.

`Task.`**`store_errors_even_if_ignored = False`**
When enabled errors will be stored even if the task is otherwise configured to ignore results.

Task.**subtask**(*args=None*, **starargs*, ***starkwargs*)
Returns subtask object for this task, wrapping arguments and execution options for a single task invocation.

Task.**time_limit** = **None**
Hard time limit. Defaults to the CELERYD_TASK_TIME_LIMIT setting.

Task.**track_started** = **False**
If enabled the task will report its status as 'started' when the task is executed by a worker. Disabled by default as the normal behaviour is to not report that level of granularity. Tasks are either pending, finished, or waiting to be retried.

Having a 'started' status can be useful for when there are long running tasks and there is a need to report which task is currently running.

The application default can be overridden using the CELERY_TRACK_STARTED setting.

Task.**update_state**(*task_id=None*, *state=None*, *meta=None*)
Update task state.

Parameters

- **task_id** – Id of the task to update, defaults to the id of the current task
- **state** – New state (str).
- **meta** – State metadata (dict).

class celery.app.task.**TaskType**
Meta class for tasks.

Automatically registers the task in the task registry, except if the *abstract* attribute is set.

If no *name* attribute is provided, then no name is automatically set to the name of the module it was defined in, and the class name.

2.13.4 celery.app.amqp

Sending and receiving messages using Kombu.

AMQP

class celery.app.amqp.**AMQP**(*app*)

Connection
Broker connection class used. Default is kombu.connection.Connection.

Consumer
Base Consumer class used. Default is kombu.compat.Consumer.

queues
All currently defined task queues. (A Queues instance).

Queues(*queues*, *create_missing=None*, *ha_policy=None*)
Create new `Queues` instance, using queue defaults from the current configuration.

Router(*queues=None*, *create_missing=None*)
Returns the current task router.

TaskConsumer
Return consumer configured to consume from the queues we are configured for (`app.amqp.queues.consume_from`).

TaskProducer
Returns publisher used to send tasks.

You should use *app.send_task* instead.

flush_routes()

default_queue

default_exchange

publisher_pool

router

routes

Queues

class `celery.app.amqp.`**Queues**(*queues=None*, *default_exchange=None*, *create_missing=True*, *ha_policy=None*)
Queue name declaration mapping.

Parameters

- **queues** – Initial list/tuple or dict of queues.
- **create_missing** – By default any unknown queues will be added automatically, but if disabled the occurrence of unknown queues in *wanted* will raise `KeyError`.
- **ha_policy** – Default HA policy for queues with none set.

add(*queue*, ***kwargs*)
Add new queue.

Parameters

- **queue** – Name of the queue.
- **exchange** – Name of the exchange.
- **routing_key** – Binding key.
- **exchange_type** – Type of exchange.
- ****options** – Additional declaration options.

add_compat(*name*, ***options*)

consume_from

format(*indent=0*, *indent_first=True*)
Format routing table into string for log dumps.

new_missing(*name*)

select_add(*queue*, ***kwargs*)
Add new task queue that will be consumed from even when a subset has been selected using the `-Q` option.

select_remove(*queue*)

select_subset(*wanted*)
Sets `consume_from` by selecting a subset of the currently defined queues.

Parameters **wanted** – List of wanted queue names.

TaskPublisher

class `celery.app.amqp.`**TaskPublisher**(*channel=None*, *exchange=None*, **args*, ***kwargs*)
Deprecated version of `TaskProducer`.

2.13.5 celery.app.defaults

- celery.app.defaults

celery.app.defaults

Configuration introspection and defaults.

class `celery.app.defaults.`**Option**(*default=None*, **args*, ***kwargs*)

alt = None

deprecate_by = None

remove_by = None

to_python(*value*)

typemap = {'bool': <function strtobool at 0x7fb768d6f488>, 'string': <type 'str'>, 'tuple': <type 'tuple'>, 'int': <type 'in

`celery.app.defaults.`**find**(**args*, ***kwargs*)

`celery.app.defaults.`**find_deprecated_settings**(*source*)

`celery.app.defaults.`**flatten**(*d*, *ns=''*)

2.13.6 celery.app.control

- celery.app.control

celery.app.control

Client for worker remote control commands. Server implementation is in `celery.worker.control`.

class `celery.app.control.`**Control**(*app=None*)

class **Mailbox**(*namespace*, *type='direct'*, *connection=None*, *clock=None*, *accept=None*)

Node(*hostname=None*, *state=None*, *channel=None*, *handlers=None*)

abcast(*command*, *kwargs={}*)

call(*destination*, *command*, *kwargs={}*, *timeout=None*, *callback=None*, *channel=None*)

cast(*destination*, *command*, *kwargs={}*)

connection = None

exchange = None

exchange_fmt = '%s.pidbox'

get_queue(*hostname*)

get_reply_queue()

multi_call(*command*, *kwargs={}*, *timeout=1*, *limit=None*, *callback=None*, *channel=None*)

namespace = None

node_cls
alias of Node

oid

reply_exchange = None

reply_exchange_fmt = 'reply.%s.pidbox'

reply_queue

type = 'direct'

Control.**add_consumer**(*queue*, *exchange=None*, *exchange_type='direct'*, *routing_key=None*, *options=None*, ***kwargs*)
Tell all (or specific) workers to start consuming from a new queue.

Only the queue name is required as if only the queue is specified then the exchange/routing key will be set to the same name (like automatic queues do).

Note: This command does not respect the default queue/exchange options in the configuration.

Parameters

- **queue** – Name of queue to start consuming from.
- **exchange** – Optional name of exchange.
- **exchange_type** – Type of exchange (defaults to 'direct') command to, when empty broadcast to all workers.
- **routing_key** – Optional routing key.
- **options** – Additional options as supported by kombu.entitiy.Queue.from_dict().

See broadcast() for supported keyword arguments.

Control.**broadcast**(*command*, *arguments=None*, *destination=None*, *connection=None*, *reply=False*, *timeout=1*, *limit=None*, *callback=None*, *channel=None*, ***extra_kwargs*)
Broadcast a control command to the celery workers.

Parameters

- **command** – Name of command to send.
- **arguments** – Keyword arguments for the command.
- **destination** – If set, a list of the hosts to send the command to, when empty broadcast to all workers.
- **connection** – Custom broker connection to use, if not set, a connection will be established automatically.
- **reply** – Wait for and return the reply.
- **timeout** – Timeout in seconds to wait for the reply.
- **limit** – Limit number of replies.
- **callback** – Callback called immediately for each reply received.

Control.**cancel_consumer**(*queue*, ***kwargs*)
Tell all (or specific) workers to stop consuming from `queue`.

Supports the same keyword arguments as `broadcast()`.

Control.**disable_events**(*destination=None*, ***kwargs*)
Tell all (or specific) workers to enable events.

Control.**discard_all**(*connection=None*)
Discard all waiting tasks.

This will ignore all tasks waiting for execution, and they will be deleted from the messaging server.

Returns the number of tasks discarded.

Control.**enable_events**(*destination=None*, ***kwargs*)
Tell all (or specific) workers to enable events.

Control.**inspect**

Control.**ping**(*destination=None*, *timeout=1*, ***kwargs*)
Ping all (or specific) workers.

Returns answer from alive workers.

See `broadcast()` for supported keyword arguments.

Control.**pool_grow**(*n=1*, *destination=None*, ***kwargs*)
Tell all (or specific) workers to grow the pool by `n`.

Supports the same arguments as `broadcast()`.

Control.**pool_shrink**(*n=1*, *destination=None*, ***kwargs*)
Tell all (or specific) workers to shrink the pool by `n`.

Supports the same arguments as `broadcast()`.

Control.**purge**(*connection=None*)
Discard all waiting tasks.

This will ignore all tasks waiting for execution, and they will be deleted from the messaging server.

Returns the number of tasks discarded.

Control.**rate_limit**(*task_name*, *rate_limit*, *destination=None*, ***kwargs*)
Tell all (or specific) workers to set a new rate limit for task by type.

Parameters

- **task_name** – Name of task to change rate limit for.
- **rate_limit** – The rate limit as tasks per second, or a rate limit string (*'100/m'*, etc. see `celery.task.base.Task.rate_limit` for more information).

See `broadcast()` for supported keyword arguments.

Control.**revoke**(*task_id*, *destination=None*, *terminate=False*, *signal='SIGTERM'*, ***kwargs*)
Tell all (or specific) workers to revoke a task by id.

If a task is revoked, the workers will ignore the task and not execute it after all.

Parameters

- **task_id** – Id of the task to revoke.
- **terminate** – Also terminate the process currently working on the task (if any).
- **signal** – Name of signal to send to process if terminate. Default is TERM.

See `broadcast()` for supported keyword arguments.

Control.**time_limit**(*task_name*, *soft=None*, *hard=None*, ***kwargs*)
Tell all (or specific) workers to set time limits for a task by type.

Parameters

- **task_name** – Name of task to change time limits for.
- **soft** – New soft time limit (in seconds).
- **hard** – New hard time limit (in seconds).

Any additional keyword arguments are passed on to `broadcast()`.

class celery.app.control.**Inspect**(*destination=None*, *timeout=1*, *callback=None*, *connection=None*, *app=None*, *limit=None*)

active(*safe=False*)

active_queues()

app = **None**

conf()

ping()

registered(**taskinfoitems*)

registered_tasks(**taskinfoitems*)

report()

reserved(*safe=False*)

revoked()

scheduled(*safe=False*)

stats()

celery.app.control.**flatten_reply**(*reply*)

2.13.7 celery.app.registry

celery.app.registry

Registry of available tasks.

class `celery.app.registry.`**`TaskRegistry`**

exception `NotRegistered`
The task is not registered.

`TaskRegistry.`**`filter_types`**(*type*)

`TaskRegistry.`**`periodic`**()

`TaskRegistry.`**`register`**(*task*)
Register a task in the task registry.

The task will be automatically instantiated if not already an instance.

`TaskRegistry.`**`regular`**()

`TaskRegistry.`**`unregister`**(*name*)
Unregister task by name.

Parameters **name** – name of the task to unregister, or a `celery.task.base.Task` with a valid *name* attribute.

Raises **celery.exceptions.NotRegistered** if the task has not been registered.

2.13.8 celery.app.builtins

celery.app.builtins

Built-in tasks that are always available in all app instances. E.g. chord, group and xmap.

`celery.app.builtins.`**`add_backend_cleanup_task`**(*app*)
The backend cleanup task can be used to clean up the default result backend.

This task is also added do the periodic task schedule so that it is run every day at midnight, but **celerybeat** must be running for this to be effective.

Note that not all backends do anything for this, what needs to be done at cleanup is up to each backend, and some backends may even clean up in realtime so that a periodic cleanup is not necessary.

`celery.app.builtins.`**`add_chain_task`**(*app*)

`celery.app.builtins.`**`add_chord_task`**(*app*)
Every chord is executed in a dedicated task, so that the chord can be used as a subtask, and this generates the task responsible for that.

celery.app.builtins.**add_chunk_task**(*app*)

celery.app.builtins.**add_group_task**(*app*)

celery.app.builtins.**add_map_task**(*app*)

celery.app.builtins.**add_starmap_task**(*app*)

celery.app.builtins.**add_unlock_chord_task**(*app*)
The unlock chord task is used by result backends that doesn't have native chord support.

It creates a task chain polling the header for completion.

celery.app.builtins.**load_shared_tasks**(*app*)
Loads the built-in tasks for an app instance.

celery.app.builtins.**shared_task**(*constructor*)
Decorator that specifies that the decorated function is a function that generates a built-in task.

The function will then be called for every new app instance created (lazily, so more exactly when the task registry for that app is needed).

2.13.9 celery.app.log

celery.app.log

The Celery instances logging section: `Celery.log`.

Sets up logging for the worker and other programs, redirects stdouts, colors log output, patches logging related compatibility fixes, and so on.

class celery.app.log.**Logging**(*app*)

colored(*logfile=None*, *enabled=None*)

get_default_logger(*name='celery'*, ***kwargs*)

redirect_stdouts_to_logger(*logger*, *loglevel=None*, *stdout=True*, *stderr=True*)
Redirect `sys.stdout` and `sys.stderr` to a logging instance.

Parameters

- **logger** – The `logging.Logger` instance to redirect to.
- **loglevel** – The loglevel redirected messages will be logged as.

setup(*loglevel=None*, *logfile=None*, *redirect_stdouts=False*, *redirect_level='WARNING'*, *colorize=None*)

setup_handlers(*logger*, *logfile*, *format*, *colorize*, *formatter=<class 'celery.utils.log.ColorFormatter'>*, ***kwargs*)

setup_logger(*name='celery'*, **args*, ***kwargs*)
Deprecated: No longer used.

setup_logging_subsystem(*loglevel=None*, *logfile=None*, *format=None*, *colorize=None*, ***kwargs*)

setup_task_loggers(*loglevel=None*, *logfile=None*, *format=None*, *colorize=None*, *propagate=False*, ***kwargs*)

Setup the task logger.

If *logfile* is not specified, then *sys.stderr* is used.

Returns logger object.

supports_color(*colorize=None*, *logfile=None*)

class celery.app.log.**TaskFormatter**(*fmt=None*, *use_color=True*)

format(*record*)

2.13.10 celery.app.utils

- celery.app.utils

celery.app.utils

App utilities: Compat settings, bugreport tool, pickling apps.

class celery.app.utils.**AppPickler**

Default application pickler/unpickler.

build_kwargs(**args*)

build_standard_kwargs(*main*, *changes*, *loader*, *backend*, *amqp*, *events*, *log*, *control*, *accept_magic_kwargs*, *config_source=None*)

construct(*cls*, ***kwargs*)

prepare(*app*, ***kwargs*)

celery.app.utils.**BUGREPORT_INFO = '\nsoftware -> celery:%(celery_v)s kombu:%(kombu_v)s py:%(py_v)s\n billiard:**

Format used to generate bugreport information.

class celery.app.utils.**Settings**(*changes*, *defaults*)

Celery settings object.

BROKER_BACKEND

Deprecated compat alias to BROKER_TRANSPORT.

BROKER_HOST

BROKER_TRANSPORT

CELERY_RESULT_BACKEND

CELERY_TIMEZONE

find_option(*name*, *namespace='celery'*)

Search for option by name.

Will return `(namespace, option_name, Option)` tuple, e.g.:

```
>>> celery.conf.find_option('disable_rate_limits')
('CELERY', 'DISABLE_RATE_LIMITS',
 <Option: type->bool default->False>))
```

Parameters

- **name** – Name of option, cannot be partial.
- **namespace** – Preferred namespace (`CELERY` by default).

find_value_for_key(*name*, *namespace='celery'*)
Shortcut to `get_by_parts(*find_option(name)[:-1])`

get_by_parts(**parts*)
Returns the current value for setting specified as a path.

Example:

```
>>> celery.conf.get_by_parts('CELERY', 'DISABLE_RATE_LIMITS')
False
```

humanize()
Returns a human readable string showing changes to the configuration.

without_defaults()
Returns the current configuration, but without defaults.

`celery.app.utils.`**bugreport**(*app*)
Returns a string containing information useful in bug reports.

`celery.app.utils.`**filter_hidden_settings**(*conf*)

2.13.11 celery.task

celery.task

This is the old task module, it should not be used anymore, import from the main 'celery' module instead. If you're looking for the decorator implementation then that's in `celery.app.base.Celery.task`.

`celery.task.`**task**(**args*, ***kwargs*)
Decorator to create a task class out of any callable.

Examples

```
@task()
def refresh_feed(url):
    return Feed.objects.get(url=url).refresh()
```

With setting extra options and using retry.

```
@task(max_retries=10)
def refresh_feed(url):
    try:
        return Feed.objects.get(url=url).refresh()
    except socket.error, exc:
        refresh_feed.retry(exc=exc)
```

Calling the resulting task:

```
>>> refresh_feed('http://example.com/rss') # Regular
<Feed: http://example.com/rss>
>>> refresh_feed.delay('http://example.com/rss') # Async
<AsyncResult: 8998d0f4-da0b-4669-ba03-d5ab5ac6ad5d>
```

celery.task.**periodic_task**(**args*, ***options*)

Decorator to create a task class out of any callable.

Examples

```
@task()
def refresh_feed(url):
    return Feed.objects.get(url=url).refresh()
```

With setting extra options and using retry.

```
from celery.task import current

@task(exchange='feeds')
def refresh_feed(url):
    try:
        return Feed.objects.get(url=url).refresh()
    except socket.error, exc:
        current.retry(exc=exc)
```

Calling the resulting task:

```
>>> refresh_feed('http://example.com/rss') # Regular
<Feed: http://example.com/rss>
>>> refresh_feed.delay('http://example.com/rss') # Async
<AsyncResult: 8998d0f4-da0b-4669-ba03-d5ab5ac6ad5d>
```

---

class celery.task.**Task**

Deprecated Task base class.

Modern applications should use `celery.Task` instead.

See also:

`celery.task.base.BaseTask`.

2.13.12 celery.task.base (Deprecated)

celery.task.base

The task implementation has been moved to `celery.app.task`.

This contains the backward compatible Task class used in the old API, and shouldn't be used in new applications.

celery.task.base.**BaseTask**

alias of `Task`

class celery.task.base.**PeriodicTask**
A periodic task is a task that adds itself to the CELERYBEAT_SCHEDULE setting.

class celery.task.base.**TaskType**
Meta class for tasks.

Automatically registers the task in the task registry, except if the *abstract* attribute is set.

If no *name* attribute is provided, then no name is automatically set to the name of the module it was defined in, and the class name.

2.13.13 celery.result

celery.result

Task results/state and groups of results.

class celery.result.**AsyncResult**(*id*, *backend=None*, *task_name=None*, *app=None*, *parent=None*)
Query task state.

Parameters

- **id** – see id.
- **backend** – see backend.

exception TimeoutError
Error raised for timeouts.

AsyncResult.**app = None**

AsyncResult.**backend = None**
The task result backend to use.

AsyncResult.**build_graph**(*intermediate=False*)

AsyncResult.**children**

AsyncResult.**collect**(*intermediate=False*, ***kwargs*)
Iterator, like get() will wait for the task to complete, but will also follow AsyncResult and ResultSet returned by the task, yielding for each result in the tree.

An example would be having the following tasks:

```
@task()
def A(how_many):
    return group(B.s(i) for i in xrange(how_many))

@task()
def B(i):
    return pow2.delay(i)

@task()
def pow2(i):
    return i ** 2
```

Calling `collect()` would return:

```
>>> result = A.delay(10)
>>> list(result.collect())
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

AsyncResult.**failed**()
Returns `True` if the task failed.

AsyncResult.**forget**()
Forget about (and possibly remove the result of) this task.

AsyncResult.**get**(*timeout=None*, *propagate=True*, *interval=0.5*)
Wait until task is ready, and return its result.

> **Warning:** Waiting for tasks within a task may lead to deadlocks. Please read *Avoid launching synchronous subtasks*.

Parameters

- **timeout** – How long to wait, in seconds, before the operation times out.
- **propagate** – Re-raise exception if the task failed.
- **interval** – Time to wait (in seconds) before retrying to retrieve the result. Note that this does not have any effect when using the amqp result store backend, as it does not use polling.

Raises **celery.exceptions.TimeoutError** if *timeout* is not `None` and the result does not arrive within *timeout* seconds.

If the remote call raised an exception then that exception will be re-raised.

AsyncResult.**get_leaf**()

AsyncResult.**graph**

AsyncResult.**id** = **None**
The task's UUID.

AsyncResult.**info**
When the task has been executed, this contains the return value. If the task raised an exception, this will be the exception instance.

AsyncResult.**iterdeps**(*intermediate=False*)

AsyncResult.**parent** = **None**
Parent result (if part of a chain)

AsyncResult.**ready**()
Returns `True` if the task has been executed.

If the task is still running, pending, or is waiting for retry then `False` is returned.

AsyncResult.**result**
When the task has been executed, this contains the return value. If the task raised an exception, this will be the exception instance.

AsyncResult.**revoke**(*connection=None*, *terminate=False*, *signal=None*)
Send revoke signal to all workers.

Any worker receiving the task, or having reserved the task, *must* ignore it.

Parameters

- **terminate** – Also terminate the process currently working on the task (if any).
- **signal** – Name of signal to send to process if terminate. Default is TERM.

AsyncResult.**serializable**()

AsyncResult.**state**
The tasks current state.

Possible values includes:

PENDING

The task is waiting for execution.

STARTED

The task has been started.

RETRY

The task is to be retried, possibly because of failure.

FAILURE

The task raised an exception, or has exceeded the retry limit. The `result` attribute then contains the exception raised by the task.

SUCCESS

The task executed successfully. The `result` attribute then contains the tasks return value.

AsyncResult.**status**
The tasks current state.

Possible values includes:

PENDING

The task is waiting for execution.

STARTED

The task has been started.

RETRY

The task is to be retried, possibly because of failure.

FAILURE

The task raised an exception, or has exceeded the retry limit. The `result` attribute then contains the exception raised by the task.

SUCCESS

The task executed successfully. The `result` attribute then contains the tasks return value.

AsyncResult.**successful**()
Returns `True` if the task executed successfully.

AsyncResult.**supports_native_join**

AsyncResult.**task_id**

AsyncResult.**traceback**
Get the traceback of a failed task.

AsyncResult.**wait**(*timeout=None*, *propagate=True*, *interval=0.5*)
Wait until task is ready, and return its result.

> **Warning:** Waiting for tasks within a task may lead to deadlocks. Please read *Avoid launching synchronous subtasks*.

Parameters

- **timeout** – How long to wait, in seconds, before the operation times out.
- **propagate** – Re-raise exception if the task failed.
- **interval** – Time to wait (in seconds) before retrying to retrieve the result. Note that this does not have any effect when using the amqp result store backend, as it does not use polling.

Raises **celery.exceptions.TimeoutError** if *timeout* is not None and the result does not arrive within *timeout* seconds.

If the remote call raised an exception then that exception will be re-raised.

celery.result.**BaseAsyncResult**
alias of AsyncResult

class celery.result.**EagerResult**(*id*, *ret_value*, *state*, *traceback=None*)
Result that we know has already been executed.

forget()

get(*timeout=None*, *propagate=True*, ***kwargs*)

ready()

result
The tasks return value

revoke(**args*, ***kwargs*)

state
The tasks state.

status
The tasks state.

supports_native_join

task_name = None

traceback
The traceback if the task failed.

wait(*timeout=None*, *propagate=True*, ***kwargs*)

class celery.result.**GroupResult**(*id=None*, *results=None*, ***kwargs*)
Like ResultSet, but with an associated id.

This type is returned by group, and the deprecated TaskSet, meth:*~celery.task.TaskSet.apply_async* method.

It enables inspection of the tasks state and return values as a single entity.

Parameters

- **id** – The id of the group.
- **results** – List of result instances.

children

delete(*backend=None*)
Remove this result if it was previously saved.

id = None
The UUID of the group.

classmethod restore(*id*, *backend=None*)
Restore previously saved group result.

results = None
List/iterator of results in the group

save(*backend=None*)
Save group-result for later retrieval using `restore()`.

Example:

```
>>> result.save()
>>> result = GroupResult.restore(group_id)
```

serializable()

class celery.result.**ResultBase**
Base class for all results

class celery.result.**ResultSet**(*results*, *app=None*, ***kwargs*)
Working with more than one result.

Parameters **results** – List of result instances.

add(*result*)
Add `AsyncResult` as a new member of the set.

Does nothing if the result is already a member.

app = None

clear()
Remove all results from this set.

completed_count()
Task completion count.

Returns the number of tasks completed.

discard(*result*)
Remove result from the set if it is a member.

If it is not a member, do nothing.

failed()
Did any of the tasks fail?

Returns `True` if one of the tasks failed. (i.e., raised an exception)

forget()
Forget about (and possible remove the result of) all the tasks.

get(*timeout=None*, *propagate=True*, *interval=0.5*)
See `join()`

This is here for API compatibility with `AsyncResult`, in addition it uses `join_native()` if available for the current result backend.

iter_native(*timeout=None*, *interval=None*)
Backend optimized version of `iterate()`.

New in version 2.2.

Note that this does not support collecting the results for different task types using different backends.

This is currently only supported by the amqp, Redis and cache result backends.

iterate(*timeout=None*, *propagate=True*, *interval=0.5*)
Iterate over the return values of the tasks as they finish one by one.

Raises The exception if any of the tasks raised an exception.

join(*timeout=None*, *propagate=True*, *interval=0.5*)
Gathers the results of all tasks as a list in order.

Note: This can be an expensive operation for result store backends that must resort to polling (e.g. database).

You should consider using `join_native()` if your backend supports it.

Warning: Waiting for tasks within a task may lead to deadlocks. Please see *Avoid launching synchronous subtasks*.

Parameters

- **timeout** – The number of seconds to wait for results before the operation times out.
- **propagate** – If any of the tasks raises an exception, the exception will be re-raised.
- **interval** – Time to wait (in seconds) before retrying to retrieve a result from the set. Note that this does not have any effect when using the amqp result store backend, as it does not use polling.

Raises **celery.exceptions.TimeoutError** if *timeout* is not `None` and the operation takes longer than *timeout* seconds.

join_native(*timeout=None*, *propagate=True*, *interval=0.5*)
Backend optimized version of `join()`.

New in version 2.2.

Note that this does not support collecting the results for different task types using different backends.

This is currently only supported by the amqp, Redis and cache result backends.

ready()
Did all of the tasks complete? (either by success of failure).

Returns `True` if all of the tasks has been executed.

remove(*result*)
Removes result from the set; it must be a member.

Raises **KeyError** if the result is not a member.

results = None
List of results in in the set.

revoke(*connection=None*, *terminate=False*, *signal=None*)
Send revoke signal to all workers for all tasks in the set.

Parameters

- **terminate** – Also terminate the process currently working on the task (if any).
- **signal** – Name of signal to send to process if terminate. Default is TERM.

subtasks
Deprecated alias to `results`.

successful()
Was all of the tasks successful?

Returns `True` if all of the tasks finished successfully (i.e. did not raise an exception).

supports_native_join

update(*results*)
Update set with the union of itself and an iterable with results.

waiting()
Are any of the tasks incomplete?

Returns `True` if one of the tasks are still waiting for execution.

class celery.result.**TaskSetResult**(*taskset_id*, *results=None*, ***kwargs*)
Deprecated version of `GroupResult`

itersubtasks()
Deprecated. Use `iter(self.results)` instead.

taskset_id

total
Deprecated: Use `len(r)`.

celery.result.**from_serializable**(*r*, *app=None*)

2.13.14 celery.task.http

celery.task.http

Webhook task implementation.

class celery.task.http.**HttpDispatch**(*url*, *method*, *task_kwargs*, ***kwargs*)
Make task HTTP request and collect the task result.

Parameters

- **url** – The URL to request.
- **method** – HTTP method used. Currently supported methods are *GET* and *POST*.
- **task_kwargs** – Task keyword arguments.
- **logger** – Logger used for user/system feedback.

dispatch()
Dispatch callback and return result.

http_headers

make_request(*url*, *method*, *params*)
Makes an HTTP request and returns the response.

timeout = 5

user_agent = 'celery/3.0.25'

class celery.task.http.**HttpDispatchTask**
Task dispatching to an URL.

Parameters

- **url** – The URL location of the HTTP callback task.
- **method** – Method to use when dispatching the callback. Usually *GET* or *POST*.
- ****kwargs** – Keyword arguments to pass on to the HTTP callback.

url
If this is set, this is used as the default URL for requests. Default is to require the user of the task to supply the url as an argument, as this attribute is intended for subclasses.

method
If this is set, this is the default method used for requests. Default is to require the user of the task to supply the method as an argument, as this attribute is intended for subclasses.

accept_magic_kwargs = False

backend = <celery.backends.cache.CacheBackend object at 0x7fb767118c10>

delivery_mode = 2

exchange_type = 'direct'

method = None

name = 'celery.task.http.HttpDispatchTask'

rate_limit = None

request_stack = <celery.utils.threads._LocalStack object at 0x7fb76403be50>

run(*url=None*, *method='GET'*, ***kwargs*)

url = None

exception celery.task.http.**InvalidResponseError**
The remote server gave an invalid response.

class celery.task.http.**MutableURL**(*url*)
Object wrapping a Uniform Resource Locator.

Supports editing the query parameter list. You can convert the object back to a string, the query will be properly urlencoded.

Examples

```
>>> url = URL('http://www.google.com:6580/foo/bar?x=3&y=4#foo')
>>> url.query
{'x': '3', 'y': '4'}
>>> str(url)
'http://www.google.com:6580/foo/bar?y=4&x=3#foo'
>>> url.query['x'] = 10
>>> url.query.update({'George': 'Costanza'})
>>> str(url)
'http://www.google.com:6580/foo/bar?y=4&x=10&George=Costanza#foo'
```

exception celery.task.http.**RemoteExecuteError**
The remote task gave a custom error.

class celery.task.http.**URL**(*url*, *dispatcher=None*)
HTTP Callback URL

Supports requesting an URL asynchronously.

Parameters

- **url** – URL to request.
- **dispatcher** – Class used to dispatch the request. By default this is HttpDispatchTask.

dispatcher
alias of HttpDispatchTask

get_async(***kwargs*)

post_async(***kwargs*)

exception celery.task.http.**UnknownStatusError**
The remote server gave an unknown status.

celery.task.http.**extract_response**(*raw_response*, *loads=<function loads at 0x7fb768fc0848>*)
Extract the response text from a raw JSON response.

celery.task.http.**maybe_utf8**(*value*)
Encode to utf-8, only if the value is Unicode.

celery.task.http.**utf8dict**(*tup*)
With a dict's items() tuple return a new dict with any utf-8 keys/values encoded.

2.13.15 celery.schedules

celery.schedules

Schedules define the intervals at which periodic tasks should run.

exception celery.schedules.**ParseException**
Raised by crontab_parser when the input can't be parsed.

celery.schedules.**cronfield**(*s*)

class celery.schedules.**crontab**(*minute='*'*, *hour='*'*, *day_of_week='*'*, *day_of_month='*'*, *month_of_year='*'*, *nowfun=None*)
A crontab can be used as the *run_every* value of a PeriodicTask to add cron-like scheduling.

Like a *cron* job, you can specify units of time of when you would like the task to execute. It is a reasonably complete implementation of cron's features, so it should provide a fair degree of scheduling needs.

You can specify a minute, an hour, a day of the week, a day of the month, and/or a month in the year in any of the following formats:

minute

- A (list of) integers from 0-59 that represent the minutes of an hour of when execution should occur; or
- A string representing a crontab pattern. This may get pretty advanced, like *minute='*/15'* (for every quarter) or *minute='1,13,30-45,50-59/2'*.

hour

- A (list of) integers from 0-23 that represent the hours of a day of when execution should occur; or
- A string representing a crontab pattern. This may get pretty advanced, like *hour='*/3'* (for every three hours) or *hour='0,8-17/2'* (at midnight, and every two hours during office hours).

day_of_week

- A (list of) integers from 0-6, where Sunday = 0 and Saturday = 6, that represent the days of a week that execution should occur.
- A string representing a crontab pattern. This may get pretty advanced, like *day_of_week='mon-fri'* (for weekdays only). (Beware that *day_of_week='*/2'* does not literally mean 'every two days', but 'every day that is divisible by two'!)

day_of_month

- A (list of) integers from 1-31 that represents the days of the month that execution should occur.
- A string representing a crontab pattern. This may get pretty advanced, such as *day_of_month='2-30/3'* (for every even numbered day) or *day_of_month='1-7,15-21'* (for the first and third weeks of the month).

month_of_year

- A (list of) integers from 1-12 that represents the months of the year during which execution can occur.
- A string representing a crontab pattern. This may get pretty advanced, such as *month_of_year='*/3'* (for the first month of every quarter) or *month_of_year='2-12/2'* (for every even numbered month).

It is important to realize that any day on which execution should occur must be represented by entries in all three of the day and month attributes. For example, if *day_of_week* is 0 and *day_of_month* is every seventh day, only months that begin on Sunday and are also in the *month_of_year* attribute will have execution events. Or, *day_of_week* is 1 and *day_of_month* is '1-7,15-21' means every first and third monday of every month present in *month_of_year*.

is_due(*last_run_at*)
Returns tuple of two items *(is_due, next_time_to_run)*, where next time to run is in seconds.

See `celery.schedules.schedule.is_due()` for more information.

now()

remaining_delta(*last_run_at*, *tz=None*)
Returns when the periodic task should run next as a timedelta.

remaining_estimate(*last_run_at*)
Returns when the periodic task should run next as a timedelta.

class celery.schedules.**crontab_parser**(*max_=60*, *min_=0*)
Parser for crontab expressions. Any expression of the form 'groups' (see BNF grammar below) is accepted and expanded to a set of numbers. These numbers represent the units of time that the crontab needs to run on:

```
digit   :: '0'..'9'
dow     :: 'a'..'z'
number  :: digit+ | dow+
```

```
steps   :: number
range   :: number ( '-' number ) ?
numspec :: '*' | range
expr    :: numspec ( '/' steps ) ?
groups  :: expr ( ',' expr ) *
```

The parser is a general purpose one, useful for parsing hours, minutes and day_of_week expressions. Example usage:

```
>>> minutes = crontab_parser(60).parse('*/15')
[0, 15, 30, 45]
>>> hours = crontab_parser(24).parse('*/4')
[0, 4, 8, 12, 16, 20]
>>> day_of_week = crontab_parser(7).parse('*')
[0, 1, 2, 3, 4, 5, 6]
```

It can also parse day_of_month and month_of_year expressions if initialized with an minimum of 1. Example usage:

```
>>> days_of_month = crontab_parser(31, 1).parse('*/3')
[1, 4, 7, 10, 13, 16, 19, 22, 25, 28, 31]
>>> months_of_year = crontab_parser(12, 1).parse('*/2')
[1, 3, 5, 7, 9, 11]
>>> months_of_year = crontab_parser(12, 1).parse('2-12/2')
[2, 4, 6, 8, 10, 12]
```

The maximum possible expanded value returned is found by the formula:

```
max_ + min_ - 1
```

exception ParseException
Raised by crontab_parser when the input can't be parsed.

crontab_parser.**parse**(*spec*)

celery.schedules.**maybe_schedule**(*s*, *relative=False*)

class celery.schedules.**schedule**(*run_every=None*, *relative=False*, *nowfun=None*)

app

human_seconds

is_due(*last_run_at*)
Returns tuple of two items *(is_due, next_time_to_run)*, where next time to run is in seconds.

e.g.

- ***(True, 20)*, means the task should be run now, and the next** time to run is in 20 seconds.
- *(False, 12)*, means the task should be run in 12 seconds.

You can override this to decide the interval at runtime, but keep in mind the value of CELERYBEAT_MAX_LOOP_INTERVAL, which decides the maximum number of seconds celerybeat can sleep between re-checking the periodic task intervals. So if you dynamically change the next run at value, and the max interval is set to 5 minutes, it will take 5 minutes for the change to take effect, so you may consider lowering the value of CELERYBEAT_MAX_LOOP_INTERVAL if responsiveness is of importance to you.

Scheduler max interval variance

The default max loop interval may vary for different schedulers. For the default scheduler the value is 5 minutes, but for e.g. the django-celery database scheduler the value is 5 seconds.

maybe_make_aware(*dt*)

now()

relative = False

remaining_estimate(*last_run_at*)

seconds

to_local(*dt*)

tz

utc_enabled

2.13.16 celery.signals

- celery.signals

celery.signals

This module defines the signals (Observer pattern) sent by both workers and clients.

Functions can be connected to these signals, and connected functions are called whenever a signal is called.

See *Signals* for more information.

2.13.17 celery.security

- celery.security

celery.security

Module implementing the signing message serializer.

celery.security.**disable_untrusted_serializers**(*whitelist=None*)

celery.security.**setup_security**(*allowed_serializers=None*, *key=None*, *cert=None*, *store=None*, *digest='sha1'*, *serializer='json'*)

Setup the message-signing serializer.

Disables untrusted serializers and if configured to use the `auth` serializer will register the auth serializer with the provided settings into the Kombu serializer registry.

Parameters

- **allowed_serializers** – List of serializer names, or content_types that should be exempt from being disabled.
- **key** – Name of private key file to use. Defaults to the `CELERY_SECURITY_KEY` setting.

- **cert** – Name of certificate file to use. Defaults to the CELERY_SECURITY_CERTIFICATE setting.
- **store** – Directory containing certificates. Defaults to the CELERY_SECURITY_CERT_STORE setting.
- **digest** – Digest algorithm used when signing messages. Default is `sha1`.
- **serializer** – Serializer used to encode messages after they have been signed. See CELERY_TASK_SERIALIZER for the serializers supported. Default is `json`.

2.13.18 celery.utils.debug

Sampling Memory Usage

This module can be used to diagnose and sample the memory usage used by parts of your application.

E.g to sample the memory usage of calling tasks you can do this:

```
from celery.utils.debug import sample_mem, memdump

from tasks import add


try:
    for i in range(100):
        for j in range(100):
            add.delay(i, j)
        sample_mem()
finally:
    memdump()
```

API Reference

celery.utils.debug

Utilities for debugging memory usage.

`celery.utils.debug.`**`sample_mem`**`()`
Sample RSS memory usage.

Statistics can then be output by calling `memdump()`.

`celery.utils.debug.`**`memdump`**`(`*`samples=10`*`)`
Dump memory statistics.

Will print a sample of all RSS memory samples added by calling `sample_mem()`, and in addition print used RSS memory after `gc.collect()`.

celery.utils.debug.**sample**(*x*, *n*, *k=0*)
Given a list *x* a sample of length `n` of that list is returned.

E.g. if *n* is 10, and *x* has 100 items, a list of every 10th item is returned.

`k` can be used as offset.

celery.utils.debug.**mem_rss**()
Returns RSS memory usage as a humanized string.

celery.utils.debug.**ps**()
Returns the global `psutil.Process` instance, or `None` if `psutil` is not installed.

2.13.19 celery.utils.mail

celery.utils.mail

How task error emails are formatted and sent.

class celery.utils.mail.**ErrorMail**(*task*, ***kwargs*)
Defines how and when task error e-mails should be sent.

Parameters **task** – The task instance that raised the error.

`subject` and `body` are format strings which are passed a context containing the following keys:

•name

Name of the task.

•id

UUID of the task.

•exc

String representation of the exception.

•args

Positional arguments.

•kwargs

Keyword arguments.

•traceback

String representation of the traceback.

•hostname

Worker hostname.

EMAIL_SIGNATURE_SEP = '-- '

body = '\nTask %(name)s with id %(id)s raised exception:\n%(exc)r\n\n\nTask was called with args: %(args)s kwargs: %
Format string used to generate error email content.

error_whitelist = None

format_body(*context*)

format_subject(*context*)

send(*context*, *exc*, *fail_silently=True*)

should_send(*context*, *exc*)
Returns true or false depending on if a task error mail should be sent for this type of error.

subject = ' [celery@%(hostname)s] Error: Task %(name)s (%(id)s): %(exc)s\n '
Format string used to generate error email subjects.

class celery.utils.mail.**Mailer**(*host='localhost'*, *port=0*, *user=None*, *password=None*, *timeout=2*, *use_ssl=False*, *use_tls=False*)

send(*message*, *fail_silently=False*)

supports_timeout = True

class celery.utils.mail.**Message**(*to=None*, *sender=None*, *subject=None*, *body=None*, *charset='us-ascii'*)

exception celery.utils.mail.**SendmailWarning**
Problem happened while sending the email message.

celery.utils.mail.**get_local_hostname**()

2.13.20 celery.exceptions

celery.exceptions

This module contains all exceptions used by the Celery API.

exception celery.exceptions.**AlreadyRegistered**
The task is already registered.

exception celery.exceptions.**AlwaysEagerIgnored**
send_task ignores CELERY_ALWAYS_EAGER option

exception celery.exceptions.**CDeprecationWarning**

exception celery.exceptions.**CPendingDeprecationWarning**

exception celery.exceptions.**ChordError**
A task part of the chord raised an exception.

exception celery.exceptions.**Ignore**
A task can raise this to ignore doing state updates.

exception celery.exceptions.**ImproperlyConfigured**
Celery is somehow improperly configured.

exception celery.exceptions.**IncompleteStream**
Found the end of a stream of data, but the data is not yet complete.

exception celery.exceptions.**InvalidTaskError**
The task has invalid data or is not properly constructed.

exception celery.exceptions.**MaxRetriesExceededError**
The tasks max restart limit has been exceeded.

exception celery.exceptions.**NotConfigured**
Celery has not been configured, as no config module has been found.

exception celery.exceptions.**NotRegistered**
The task is not registered.

exception celery.exceptions.**QueueNotFound**
Task routed to a queue not in CELERY_QUEUES.

exception celery.exceptions.**RetryTaskError**(*message=None*, *exc=None*, *when=None*, ***kwargs*)
The task is to be retried later.

exc = None
Exception (if any) that caused the retry to happen.

humanize()

message = None
Optional message describing context of retry.

when = None
Time of retry (ETA), either int or datetime.

exception celery.exceptions.**SecurityError**
Security related exceptions.

Handle with care.

exception celery.exceptions.**SystemTerminate**
Signals that the worker should terminate.

exception celery.exceptions.**TaskRevokedError**
The task has been revoked, so no result available.

exception celery.exceptions.**TimeoutError**
The operation timed out.

2.13.21 celery.loaders

celery.loaders

Loaders define how configuration is read, what happens when workers start, when tasks are executed and so on.

celery.loaders.**current_loader**(**args*, ***kwargs*)

celery.loaders.**get_loader_cls**(*loader*)
Get loader class by name/alias

celery.loaders.**load_settings**(**args*, ***kwargs*)

2.13.22 celery.loaders.app

celery.loaders.app

The default loader used with custom app instances.

class `celery.loaders.app.`**`AppLoader`**(*app=None*, ***kwargs*)

2.13.23 celery.loaders.default

celery.loaders.default

The default loader used when no custom app has been initialized.

`celery.loaders.default.`**`C_WNOCONF = False`**
Warns if configuration file is missing if `C_WNOCONF` is set.

class `celery.loaders.default.`**`Loader`**(*app=None*, ***kwargs*)
The loader used by the default app.

`read_configuration`()
Read configuration from `celeryconfig.py` and configure celery and Django so it can be used by regular Python.

`setup_settings`(*settingsdict*)

2.13.24 celery.loaders.base

celery.loaders.base

Loader base class.

class `celery.loaders.base.`**`BaseLoader`**(*app=None*, ***kwargs*)
The base class for loaders.

Loaders handles,

•Reading celery client/worker configurations.

•**What happens when a task starts?** See `on_task_init()`.

•**What happens when the worker starts?** See `on_worker_init()`.

- **What happens when the worker shuts down?** See `on_worker_shutdown()`.
- What modules are imported to find tasks?

builtin_modules = frozenset([])

cmdline_config_parser(*args*, *namespace='celery'*, *re_type=<_sre.SRE_Pattern object at 0x7fb768d5cbe8>*, *extra_types={'json': <function loads at 0x7fb768fc0848>}*, *override_types={'dict': 'json', 'list': 'json', 'tuple': 'json'}*)

conf
Loader configuration.

config_from_envvar(*variable_name*, *silent=False*)

config_from_object(*obj*, *silent=False*)

configured = False

error_envvar_not_set = 'The environment variable %r is not set,\nand as such the configuration could not be loade

find_module(*module*)

import_default_modules()

import_from_cwd(*module*, *imp=None*, *package=None*)

import_module(*module*, *package=None*)

import_task_module(*module*)

init_worker()

init_worker_process()

mail

mail_admins(*subject*, *body*, *fail_silently=False*, *sender=None*, *to=None*, *host=None*, *port=None*, *user=None*, *password=None*, *timeout=None*, *use_ssl=False*, *use_tls=False*)

now(*utc=True*)

on_process_cleanup()
This method is called after a task is executed.

on_task_init(*task_id*, *task*)
This method is called before a task is executed.

on_worker_init()
This method is called when the worker (**celery worker**) starts.

on_worker_process_init()
This method is called when a child process starts.

on_worker_shutdown()
This method is called when the worker (**celery worker**) shuts down.

override_backends = {}

read_configuration()

shutdown_worker()

worker_initialized = False

2.13.25 celery.states

Built-in task states.

States

See *States*.

Sets

READY_STATES

Set of states meaning the task result is ready (has been executed).

UNREADY_STATES

Set of states meaning the task result is not ready (has not been executed).

EXCEPTION_STATES

Set of states meaning the task returned an exception.

PROPAGATE_STATES

Set of exception states that should propagate exceptions to the user.

ALL_STATES

Set of all possible states.

Misc.

celery.states.**PRECEDENCE = ['SUCCESS', 'FAILURE', None, 'REVOKED', 'STARTED', 'RECEIVED', 'RETRY', 'PE**

State precedence. None represents the precedence of an unknown state. Lower index means higher precedence.

celery.states.**PRECEDENCE_LOOKUP = {'RECEIVED': 5, 'RETRY': 6, 'REVOKED': 3, 'SUCCESS': 0, 'STARTED': 4,**
Hash lookup of PRECEDENCE to index

celery.states.**precedence**(*state*)
Get the precedence index for state.

Lower index means higher precedence.

class celery.states.**state**
State is a subclass of str, implementing comparison methods adhering to state precedence rules:

```
>>> from celery.states import state, PENDING, SUCCESS

>>> state(PENDING) < state(SUCCESS)
True
```

Any custom state is considered to be lower than FAILURE and SUCCESS, but higher than any of the other built-in states:

```
>>> state('PROGRESS') > state(STARTED)
True

>>> state('PROGRESS') > state('SUCCESS')
False
```

2.13.26 celery.contrib.abortable

Abortable tasks overview

For long-running Task's, it can be desirable to support aborting during execution. Of course, these tasks should be built to support abortion specifically.

The AbortableTask serves as a base class for all Task objects that should support abortion by producers.

- Producers may invoke the abort() method on AbortableAsyncResult instances, to request abortion.
- Consumers (workers) should periodically check (and honor!) the is_aborted() method at controlled points in their task's run() method. The more often, the better.

The necessary intermediate communication is dealt with by the AbortableTask implementation.

Usage example

In the consumer:

```
from celery.contrib.abortable import AbortableTask
from celery.utils.log import get_task_logger

logger = get_logger(__name__)

class MyLongRunningTask(AbortableTask):
```

```
    def run(self, **kwargs):
        results = []
        for x in xrange(100):
            # Check after every 5 loops..
            if x % 5 == 0:  # alternatively, check when some timer is due
                if self.is_aborted(**kwargs):
                    # Respect the aborted status and terminate
                    # gracefully
                    logger.warning('Task aborted.')
                    return
            y = do_something_expensive(x)
            results.append(y)
        logger.info('Task finished.')
        return results
```

In the producer:

```
from myproject.tasks import MyLongRunningTask

def myview(request):

    async_result = MyLongRunningTask.delay()
    # async_result is of type AbortableAsyncResult

    # After 10 seconds, abort the task
    time.sleep(10)
    async_result.abort()

    ...
```

After the *async_result.abort()* call, the task execution is not aborted immediately. In fact, it is not guaranteed to abort at all. Keep checking the *async_result* status, or call *async_result.wait()* to have it block until the task is finished.

Note: In order to abort tasks, there needs to be communication between the producer and the consumer. This is currently implemented through the database backend. Therefore, this class will only work with the database backends.

---

class celery.contrib.abortable.**AbortableAsyncResult**(*id*, *backend=None*, *task_name=None*, *app=None*, *parent=None*)

Represents a abortable result.

Specifically, this gives the *AsyncResult* a `abort()` method, which sets the state of the underlying Task to *'ABORTED'*.

abort()

Set the state of the task to `ABORTED`.

Abortable tasks monitor their state at regular intervals and terminate execution if so.

Be aware that invoking this method does not guarantee when the task will be aborted (or even if the task will be aborted at all).

is_aborted()

Returns `True` if the task is (being) aborted.

class celery.contrib.abortable.**AbortableTask**

A celery task that serves as a base class for all `Task`'s that support aborting during execution.

All subclasses of AbortableTask must call the is_aborted() method periodically and act accordingly when the call evaluates to True.

classmethod AsyncResult(*task_id*)
Returns the accompanying AbortableAsyncResult instance.

backend = <celery.backends.cache.CacheBackend object at 0x7fb767118c10>

delivery_mode = 2

exchange_type = 'direct'

is_aborted(***kwargs*)
Checks against the backend whether this AbortableAsyncResult is ABORTED.

Always returns False in case the *task_id* parameter refers to a regular (non-abortable) Task.

Be aware that invoking this method will cause a hit in the backend (for example a database query), so find a good balance between calling it regularly (for responsiveness), but not too often (for performance).

name = 'celery.contrib.abortable.AbortableTask'

rate_limit = None

request_stack = <celery.utils.threads._LocalStack object at 0x7fb76428db50>

2.13.27 celery.contrib.batches

Experimental task class that buffers messages and processes them as a list.

Warning: For this to work you have to set CELERYD_PREFETCH_MULTIPLIER to zero, or some value where the final multiplied value is higher than flush_every.
In the future we hope to add the ability to direct batching tasks to a channel with different QoS requirements than the task channel.

Simple Example

A click counter that flushes the buffer every 100 messages, and every seconds. Does not do anything with the data, but can easily be modified to store it in a database.

```
# Flush after 100 messages, or 10 seconds.
@app.task(base=Batches, flush_every=100, flush_interval=10)
def count_click(requests):
    from collections import Counter
    count = Counter(request.kwargs['url'] for request in requests)
    for url, count in count.items():
        print('>>> Clicks: %s -> %s' % (url, count))
```

Then you can ask for a click to be counted by doing:

```
>>> count_click.delay('http://example.com')
```

Example returning results

An interface to the Web of Trust API that flushes the buffer every 100 messages, and every 10 seconds.

```
import requests
from urlparse import urlparse

from celery.contrib.batches import Batches

wot_api_target = "https://api.mywot.com/0.4/public_link_json"

@app.task(base=Batches, flush_every=100, flush_interval=10)
def wot_api(requests):
    sig = lambda url: url
    reponses = wot_api_real(
        (sig(*request.args, **request.kwargs) for request in requests)
    )
    # use mark_as_done to manually return response data
    for response, request in zip(reponses, requests):
        app.backend.mark_as_done(request.id, response)


def wot_api_real(urls):
    domains = [urlparse(url).netloc for url in urls]
    response = requests.get(
        wot_api_target,
        params={"hosts": ('/').join(set(domains)) + '/'}
    )
    return [response.json[domain] for domain in domains]
```

Using the API is done as follows:

```
>>> wot_api.delay('http://example.com')
```

Note: If you don't have an `app` instance then use the current app proxy instead:

```
from celery import current_app
app.backend.mark_as_done(request.id, response)
```

API

class `celery.contrib.batches.`**`Batches`**

`Strategy`(*task*, *app*, *consumer*)

`apply_buffer`(*requests*, *args=()*, *kwargs={}*)

`flush`(*requests*)

`flush_every` = 10
Maximum number of message in buffer.

`flush_interval` = 30
Timeout in seconds before buffer is flushed anyway.

`run`(*requests*)

class `celery.contrib.batches.`**`SimpleRequest`**(*id*, *name*, *args*, *kwargs*, *delivery_info*, *hostname*)
Pickleable request.

args = ()
positional arguments

delivery_info = None
message delivery information.

classmethod from_request(*request*)

hostname = None
worker node name

id = None
task id

kwargs = {}
keyword arguments

name = None
task name

2.13.28 celery.contrib.migrate

celery.contrib.migrate

Migration tools.

class celery.contrib.migrate.**State**

count = 0

filtered = 0

strtotal

total_apx = 0

exception celery.contrib.migrate.**StopFiltering**

celery.contrib.migrate.**expand_dest**(*ret*, *exchange*, *routing_key*)

celery.contrib.migrate.**filter_callback**(*callback*, *tasks*)

celery.contrib.migrate.**filter_status**(*state*, *body*, *message*)

celery.contrib.migrate.**migrate_task**(*producer*, *body_*, *message*, *queues=None*)

celery.contrib.migrate.**migrate_tasks**(*source*, *dest*, *migrate=<function migrate_task at 0x7fb76261caa0>*, *app=None*, *queues=None*, ***kwargs*)

celery.contrib.migrate.**move**(*predicate*, *connection=None*, *exchange=None*, *routing_key=None*, *source=None*, *app=None*, *callback=None*, *limit=None*, *transform=None*, ***kwargs*)
Find tasks by filtering them and move the tasks to a new queue.

Parameters

- **predicate** – Filter function used to decide which messages to move. Must accept the standard signature of `(body, message)` used by Kombu consumer callbacks. If the predicate wants the message to be moved it must return either:
  1. a tuple of `(exchange, routing_key)`, or
  2. a `Queue` instance, or
  3. any other true value which means the specified `exchange` and `routing_key` arguments will be used.
- **connection** – Custom connection to use.
- **source** – Optional list of source queues to use instead of the default (which is the queues in `CELERY_QUEUES`). This list can also contain new `Queue` instances.
- **exchange** – Default destination exchange.
- **routing_key** – Default destination routing key.
- **limit** – Limit number of messages to filter.
- **callback** – Callback called after message moved, with signature `(state, body, message)`.
- **transform** – Optional function to transform the return value (destination) of the filter function.

Also supports the same keyword arguments as `start_filter()`.

To demonstrate, the `move_task_by_id()` operation can be implemented like this:

```
def is_wanted_task(body, message):
    if body['id'] == wanted_id:
        return Queue('foo', exchange=Exchange('foo'),
                     routing_key='foo')

move(is_wanted_task)
```

or with a transform:

```
def transform(value):
    if isinstance(value, basestring):
        return Queue(value, Exchange(value), value)
    return value

move(is_wanted_task, transform=transform)
```

The predicate may also return a tuple of `(exchange, routing_key)` to specify the destination to where the task should be moved, or a `Queue` instance. Any other true value means that the task will be moved to the default exchange/routing_key.

`celery.contrib.migrate.`**`move_by_idmap`**(*map*, ***kwargs*)

Moves tasks by matching from a `task_id: queue` mapping, where `queue` is a queue to move the task to.

Example:

```
>>> reroute_idmap({
...     '5bee6e82-f4ac-468e-bd3d-13e8600250bc': Queue(...),
...     'ada8652d-aef3-466b-abd2-becdaf1b82b3': Queue(...),
...     '3a2b140d-7db1-41ba-ac90-c36a0ef4ab1f': Queue(...)},
...   queues=['hipri'])
```

celery.contrib.migrate.**move_by_taskmap**(*map*, ***kwargs*)

Moves tasks by matching from a `task_name: queue` mapping, where `queue` is the queue to move the task to.

Example:

```
>>> reroute_idmap({
...     'tasks.add': Queue(...),
...     'tasks.mul': Queue(...),
... })
```

celery.contrib.migrate.**move_task_by_id**(*task_id*, *dest*, ***kwargs*)

Find a task by id and move it to another queue.

Parameters

- **task_id** – Id of task to move.
- **dest** – Destination queue.

Also supports the same keyword arguments as `move()`.

celery.contrib.migrate.**prepare_queues**(*queues*)

celery.contrib.migrate.**republish**(*producer, message, exchange=None, routing_key=None, remove_props=['application_headers', 'content_type', 'content_encoding', 'headers']*)

celery.contrib.migrate.**start_filter**(*app, conn, filter, limit=None, timeout=1.0, ack_messages=False, tasks=None, queues=None, callback=None, forever=False, on_declare_queue=None, consume_from=None, state=None, **kwargs*)

celery.contrib.migrate.**task_id_eq**(*task_id*, *body*, *message*)

celery.contrib.migrate.**task_id_in**(*ids*, *body*, *message*)

2.13.29 celery.contrib.rdb

Remote debugger for Celery tasks running in multiprocessing pool workers. Inspired by http://snippets.dzone.com/posts/show/7248

Usage

```
from celery.contrib import rdb
from celery import task

@task()
def add(x, y):
    result = x + y
    rdb.set_trace()
    return result
```

Environment Variables

CELERY_RDB_HOST

Hostname to bind to. Default is '127.0.01', which means the socket will only be accessible from the local host.

CELERY_RDB_PORT

Base port to bind to. Default is 6899. The debugger will try to find an available port starting from the base port. The selected port will be logged by the worker.

celery.contrib.rdb.**set_trace**(*frame=None*)
Set breakpoint at current location, or a specified frame

celery.contrib.rdb.**debugger**()
Returns the current debugger instance (if any), or creates a new one.

class celery.contrib.rdb.**Rdb**(*host='127.0.0.1', port=6899, port_search_limit=100, port_skew=0, out=<open file '<stdout>', mode 'w' at 0x7fb76c556150>*)

2.13.30 celery.contrib.methods

Task decorator that supports creating tasks out of methods.

Examples

```
from celery.contrib.methods import task

class X(object):

    @task()
    def add(self, x, y):
            return x + y
```

or with any task decorator:

```
from celery.contrib.methods import task_method

class X(object):

    @celery.task(filter=task_method)
    def add(self, x, y):
        return x + y
```

Note: The task must use the new Task base class (`celery.Task`), and the old base class using classmethods (`celery.task.Task`, `celery.task.base.Task`).

This means that you have to use the task decorator from a Celery app instance, and not the old-API:

```
from celery import task        # BAD
from celery.task import task  # ALSO BAD

# GOOD:
celery = Celery(...)

@celery.task(filter=task_method)
def foo(self): pass

# ALSO GOOD:
from celery import current_app

@current_app.task(filter=task_method)
def foo(self): pass
```

Caveats

- Automatic naming won't be able to know what the class name is.

  The name will still be module_name + task_name, so two methods with the same name in the same module will collide so that only one task can run:

```
class A(object):

    @task()
    def add(self, x, y):
        return x + y

class B(object):

    @task()
    def add(self, x, y):
        return x + y
```

  would have to be written as:

```
class A(object):
    @task(name='A.add')
    def add(self, x, y):
        return x + y

class B(object):
    @task(name='B.add')
    def add(self, x, y):
        return x + y
```

celery.contrib.methods.**task**(**args*, ***kwargs*)

class celery.contrib.methods.**task_method**(*task*, **args*, ***kwargs*)

2.13.31 celery.events

celery.events

Events is a stream of messages sent for certain actions occurring in the worker (and clients if CELERY_SEND_TASK_SENT_EVENT is enabled), used for monitoring purposes.

celery.events.**Event**(*type*, *_fields=None*, ***fields*)
Create an event.

An event is a dictionary, the only required field is type.

class celery.events.**EventDispatcher**(*connection=None*, *hostname=None*, *enabled=True*, *channel=None*, *buffer_while_offline=True*, *app=None*, *serializer=None*)
Send events as messages.

Parameters

- **connection** – Connection to the broker.

- **hostname** – Hostname to identify ourselves as, by default uses the hostname returned by `socket.gethostname()`.
- **enabled** – Set to `False` to not actually publish any events, making `send()` a noop operation.
- **channel** – Can be used instead of *connection* to specify an exact channel to use when sending events.
- **buffer_while_offline** – If enabled events will be buffered while the connection is down. `flush()` must be called as soon as the connection is re-established.

You need to `close()` this after use.

DISABLED_TRANSPORTS = set(['sql'])

app = None

close()
Close the event dispatcher.

copy_buffer(*other*)

disable()

enable()

flush()

publish(*type*, *fields*, *producer*, *retry=False*, *retry_policy=None*)

publisher

send(*type*, ***fields*)
Send event.

Parameters

- **type** – Kind of event.
- ****fields** – Event arguments.

class `celery.events.`**EventReceiver**(*connection*, *handlers=None*, *routing_key='#'*, *node_id=None*, *app=None*, *queue_prefix='celeryev'*)
Capture events.

Parameters

- **connection** – Connection to the broker.
- **handlers** – Event handlers.

`handlers` is a dict of event types and their handlers, the special handler *"*"* captures all events that doesn't have a handler.

app = None

capture(*limit=None*, *timeout=None*, *wakeup=True*)
Open up a consumer capturing events.

This has to run in the main process, and it will never stop unless forced via `KeyboardInterrupt` or `SystemExit`.

consumer(**args*, ***kwds*)
Create event consumer.

drain_events(***kwargs*)

handlers = {}

itercapture(*limit=None*, *timeout=None*, *wakeup=True*)

process(*type*, *event*)
Process the received event by dispatching it to the appropriate handler.

wakeup_workers(*channel=None*)

class celery.events.**Events**(*app=None*)

Dispatcher

Receiver

State

default_dispatcher(**args*, ***kwds*)

celery.events.**get_exchange**(*conn*)

2.13.32 celery.events.state

celery.events.state

This module implements a datastructure used to keep track of the state of a cluster of workers and the tasks it is working on (by consuming events).

For every event consumed the state is updated, so the state represents the state of the cluster at the time of the last event.

Snapshots (celery.events.snapshot) can be used to take "pictures" of this state at regular intervals to e.g. store that in a database.

class celery.events.state.**Element**
Base class for worker state elements.

class celery.events.state.**State**(*callback=None*, *workers=None*, *tasks=None*, *taskheap=None*, *max_workers_in_memory=5000*, *max_tasks_in_memory=10000*)
Records clusters state.

alive_workers()
Returns a list of (seemingly) alive workers.

clear(*ready=True*)

clear_tasks(*ready=True*)

event(*event*)

event_count = 0

freeze_while(*fun*, **args*, ***kwargs*)

get_or_create_task(*uuid*)
Get or create task by uuid.

get_or_create_worker(*hostname*, ***kwargs*)
Get or create worker by hostname.

itertasks(*limit=None*)

task_count = 0

task_event(*type*, *fields*)
Process task event.

task_types()
Returns a list of all seen task types.

tasks_by_timestamp(*limit=None*)
Get tasks by timestamp.

Returns a list of *(uuid, task)* tuples.

tasks_by_type(*name*, *limit=None*)
Get all tasks by type.

Returns a list of *(uuid, task)* tuples.

tasks_by_worker(*hostname*, *limit=None*)
Get all tasks by worker.

Returns a list of *(uuid, task)* tuples.

worker_event(*type*, *fields*)
Process worker event.

class celery.events.state.**Task**(***fields*)
Task State.

info(*fields=None*, *extra=[]*)
Information about this task suitable for on-screen display.

merge(*state*, *timestamp*, *fields*)
Merge with out of order event.

merge_rules = {'RECEIVED': ('name', 'args', 'kwargs', 'retries', 'eta', 'expires')}
How to merge out of order events. Disorder is detected by logical ordering (e.g. `task-received` must have happened before a `task-failed` event).

A merge rule consists of a state and a list of fields to keep from that state. `(RECEIVED, ('name', 'args')`, means the name and args fields are always taken from the RECEIVED state, and any values for these fields received before or after is simply ignored.

on_failed(*timestamp=None*, ***fields*)
Callback for the `task-failed` event.

on_received(*timestamp=None*, ***fields*)
Callback for the `task-received` event.

on_retried(*timestamp=None*, ***fields*)
Callback for the `task-retried` event.

on_revoked(*timestamp=None*, ***fields*)
Callback for the `task-revoked` event.

on_sent(*timestamp=None*, ***fields*)
Callback for the `task-sent` event.

on_started(*timestamp=None*, ***fields*)
Callback for the `task-started` event.

on_succeeded(*timestamp=None*, ***fields*)
Callback for the task-succeeded event.

on_unknown_event(*type*, *timestamp=None*, ***fields*)

ready

update(*state*, *timestamp*, *fields*)
Update state from new event.

Parameters

- **state** – State from event.
- **timestamp** – Timestamp from event.
- **fields** – Event data.

class celery.events.state.**Worker**(***fields*)
Worker State.

alive

expire_window = 200

heartbeat_expires

heartbeat_max = 4

on_heartbeat(*timestamp=None*, ***kwargs*)
Callback for the worker-heartbeat event.

on_offline(***kwargs*)
Callback for the worker-offline event.

on_online(*timestamp=None*, ***kwargs*)
Callback for the worker-online event.

celery.events.state.**heartbeat_expires**(*timestamp*, *freq=60*, *expire_window=200*)

2.13.33 celery.apps.worker

celery.apps.worker

This module is the 'program-version' of celery.worker.

It does everything necessary to run that module as an actual application, like installing signal handlers, platform tweaks, and so on.

class celery.apps.worker.**Worker**(*hostname=None*, *purge=False*, *beat=False*, *queues=None*, *include=None*, *app=None*, *pidfile=None*, *autoscale=None*, *autoreload=False*, *no_execv=False*, *no_color=None*, ***kwargs*)

class WorkController(*loglevel=None*, *hostname=None*, *ready_callback=<function noop at 0x7fb768d6c848>*, *queues=None*, *app=None*, *pidfile=None*, *use_eventloop=None*, ***kwargs*)
Unmanaged worker instance.

CLOSE = 2

RUN = 1

TERMINATE = 3

app = None

autoreloader_cls = None

autoscaler_cls = None

concurrency = None

consumer_cls = None

disable_rate_limits = None

force_execv = None

logfile = None

loglevel = 40

max_tasks_per_child = None

mediator_cls = None

pool_cls = None

pool_putlocks = None

pool_restarts = None

prefetch_multiplier = None

process_task(*req*)
Process task by sending it to the pool of workers.

process_task_sem(*req*)

reload(*modules=None*, *reload=False*, *reloader=None*)

schedule_filename = None

scheduler_cls = None

send_events = None

should_use_eventloop()

signal_consumer_close()

start()
Starts the workers main loop.

state

state_db = None

stop(*in_sighandler=False*)
Graceful shutdown of the worker server.

task_soft_time_limit = None

task_time_limit = None

terminate(*in_sighandler=False*)
Not so graceful shutdown of the worker server.

timer_cls = None

timer_precision = None

worker_lost_wait = None

Worker.**app = None**

Worker.**extra_info**()

Worker.**inherit_confopts = (<class 'celery.worker.WorkController'>,)**

Worker.**init_queues**()

Worker.**install_platform_tweaks**(*worker*)
Install platform specific tweaks and workarounds.

Worker.**loglevel = None**

Worker.**on_consumer_ready**(*consumer*)

Worker.**osx_proxy_detection_workaround**()
See http://github.com/celery/celery/issues#issue/161

Worker.**purge_messages**()

Worker.**redirect_stdouts = None**

Worker.**redirect_stdouts_level = None**

Worker.**run**()

Worker.**run_worker**()

Worker.**set_process_status**(*info*)

Worker.**setup_logging**(*colorize=None*)

Worker.**startup_info**()

Worker.**tasklist**(*include_builtins=True*, *sep='n'*, *int_='celery.'*)

celery.apps.worker.**active_thread_count**()

celery.apps.worker.**install_HUP_not_supported_handler**(*worker*, *sig='SIGHUP'*)

celery.apps.worker.**install_cry_handler**()

celery.apps.worker.**install_rdb_handler**(*envvar='CELERY_RDBSIG'*, *sig='SIGUSR2'*)

celery.apps.worker.**install_worker_restart_handler**(*worker*, *sig='SIGHUP'*)

celery.apps.worker.**on_SIGINT**(*worker*)

celery.apps.worker.**safe_say**(*msg*)

2.13.34 celery.apps.beat

- celery.apps.beat

celery.apps.beat

This module is the 'program-version' of `celery.beat`.

It does everything necessary to run that module as an actual application, like installing signal handlers and so on.

class `celery.apps.beat.`**Beat**(*max_interval=None*, *app=None*, *socket_timeout=30*, *pidfile=None*, *no_color=None*, ***kwargs*)

class **Service**(*max_interval=None*, *schedule_filename=None*, *scheduler_cls=None*, *app=None*)

get_scheduler(*lazy=False*)

scheduler

scheduler_cls
alias of `PersistentScheduler`

start(*embedded_process=False*)

stop(*wait=False*)

sync()

`Beat.`**app** = **None**

`Beat.`**init_loader**()

`Beat.`**install_sync_handler**(*beat*)
Install a *SIGTERM* + *SIGINT* handler that saves the celerybeat schedule.

`Beat.`**logfile** = **None**

`Beat.`**loglevel** = **None**

`Beat.`**redirect_stdouts** = **None**

`Beat.`**redirect_stdouts_level** = **None**

`Beat.`**run**()

`Beat.`**schedule** = **None**

`Beat.`**scheduler_cls** = **None**

`Beat.`**set_process_title**()

`Beat.`**setup_logging**(*colorize=None*)

`Beat.`**start_scheduler**()

`Beat.`**startup_info**(*beat*)

2.13.35 celery.bin.base

Preload Options

These options are supported by all commands, and usually parsed before command-specific arguments.

-A, --app
app instance to use (e.g. module.attr_name)

-b, --broker
url to broker. default is 'amqp://guest@localhost//'

--loader
name of custom loader class to use.

--config
Name of the configuration module

Daemon Options

These options are supported by commands that can detach into the background (daemon). They will be present in any command that also has a *–detach* option.

-f, --logfile
Path to log file. If no logfile is specified, *stderr* is used.

--pidfile
Optional file used to store the process pid.

The program will not start if this file already exists and the pid is still alive.

--uid
User id, or user name of the user to run as after detaching.

--gid
Group id, or group name of the main group to change to after detaching.

--umask
Effective umask (in octal) of the process after detaching. Inherits the umask of the parent process by default.

--workdir
Optional directory to change to after detaching.

class `celery.bin.base.`**Command**(*app=None*, *get_app=None*)
Base class for command line applications.

Parameters

- **app** – The current app.
- **get_app** – Callable returning the current app if no app provided.

Parser
alias of `OptionParser`

args = ''
Arg list used in help.

check_args(*args*)

create_parser(*prog_name*, *command=None*)

description = ''
Text to print in –help before option list.

die(*msg*, *status=1*)

doc = None

early_version(*argv*)

enable_config_from_cmdline = False
Enable if the application should support config from the cmdline.

epilog = None
Text to print at end of –help

execute_from_commandline(*argv=None*)
Execute application from command line.

Parameters **argv** – The list of command line arguments. Defaults to `sys.argv`.

expanduser(*value*)

find_app(*app*)

get_cls_by_name(*name*)

get_options()
Get supported command line options.

handle_argv(*prog_name*, *argv*)
Parses command line arguments from `argv` and dispatches to `run()`.

Parameters

- **prog_name** – The program name (`argv[0]`).
- **argv** – Command arguments.

Exits with an error message if `supports_args` is disabled and `argv` contains positional arguments.

leaf = True
Set to true if this command doesn't have subcommands

maybe_patch_concurrency(*argv=None*)

namespace = 'celery'
Default configuration namespace.

on_concurrency_setup()

option_list = ()
List of options (without preload options).

parse_doc(*doc*)

parse_options(*prog_name*, *arguments*)
Parse the available options.

parse_preload_options(*args*)

preload_options = (<Option at 0x7fb762fd3248: -A/–app>, <Option at 0x7fb762bfebd8: -b/–broker>, <Option at 0x7
List of options to parse before parsing other options.

prepare_args(*options*, *args*)

prepare_parser(*parser*)

process_cmdline_config(*argv*)

respects_app_option = True

run(**args*, ***options*)
This is the body of the command called by `handle_argv()`.

run_from_argv(*prog_name*, *argv=None*)

setup_app_from_commandline(*argv*)

supports_args = True
If false the parser will raise an exception if positional args are provided.

symbol_by_name(*name*)

usage(*command*)
Returns the command line usage string for this app.

version = '3.0.25 (Chiastic Slide)'
Application version.

with_pool_option(*argv*)
Returns tuple of `(short_opts, long_opts)` if the command supports a pool argument, and used to monkey patch eventlet/gevent environments as early as possible.

E.g:: has_pool_option = (['-P'], ['–pool'])

class celery.bin.base.**HelpFormatter**(*indent_increment=2*, *max_help_position=24*, *width=None*, *short_first=1*)

format_description(*description*)

format_epilog(*epilog*)

celery.bin.base.**daemon_options**(*default_pidfile=None*, *default_logfile=None*)

2.13.36 celery.bin.celeryd

The **celery worker** command (previously known as `celeryd`)

See also:

See *Preload Options*.

-c, --concurrency
Number of child processes processing the queue. The default is the number of CPUs available on your system.

-P, --pool
Pool implementation:

processes (default), eventlet, gevent, solo or threads.

-f, --logfile
Path to log file. If no logfile is specified, *stderr* is used.

-l, --loglevel
Logging level, choose between *DEBUG*, *INFO*, *WARNING*, *ERROR*, *CRITICAL*, or *FATAL*.

-n, --hostname
Set custom hostname, e.g. 'foo.example.com'.

-B, --beat
Also run the *celerybeat* periodic task scheduler. Please note that there must only be one instance of this service.

-Q, --queues
List of queues to enable for this worker, separated by comma. By default all configured queues are enabled. Example: *-Q video,image*

-I, --include
Comma separated list of additional modules to import. Example: -I foo.tasks,bar.tasks

-s, --schedule
Path to the schedule database if running with the *-B* option. Defaults to *celerybeat-schedule*. The extension ".db" may be appended to the filename.

--scheduler
Scheduler class to use. Default is celery.beat.PersistentScheduler

-S, --statedb
Path to the state database. The extension '.db' may be appended to the filename. Default: %(default)s

-E, --events
Send events that can be captured by monitors like **celeryev**, *celerymon*, and others.

--purge
Purges all waiting tasks before the daemon is started. **WARNING**: This is unrecoverable, and the tasks will be deleted from the messaging server.

--time-limit
Enables a hard time limit (in seconds int/float) for tasks.

--soft-time-limit
Enables a soft time limit (in seconds int/float) for tasks.

--maxtasksperchild
Maximum number of tasks a pool worker can execute before it's terminated and replaced by a new worker.

--pidfile
Optional file used to store the workers pid.

The worker will not start if this file already exists and the pid is still alive.

--autoscale
Enable autoscaling by providing max_concurrency, min_concurrency. Example:

```
--autoscale=10,3
```

(always keep 3 processes, but grow to 10 if necessary)

--autoreload
Enable autoreloading.

--no-execv
Don't do execv after multiprocessing child fork.

class celery.bin.celeryd.**WorkerCommand**(*app=None*, *get_app=None*)

doc = '\n\nThe :program:'celery worker' command (previously known as ''celeryd'')\n\n.. program:: celery worker\n\n..

enable_config_from_cmdline = True

execute_from_commandline(*argv=None*)

get_options()

namespace = 'celeryd'

run(**args*, ***kwargs*)

supports_args = False

with_pool_option(*argv*)

celery.bin.celeryd.**main**()

2.13.37 celery.bin.celerybeat

The **celery beat** command.

See also:

See *Preload Options* and *Daemon Options*.

--detach
Detach and run in the background as a daemon.

-s, --schedule
Path to the schedule database. Defaults to *celerybeat-schedule*. The extension '.db' may be appended to the filename. Default is %(default)s.

-S, --scheduler
Scheduler class to use. Default is `celery.beat.PersistentScheduler`.

max-interval
Max seconds to sleep between schedule iterations.

-f, --logfile
Path to log file. If no logfile is specified, *stderr* is used.

-l, --loglevel
Logging level, choose between *DEBUG*, *INFO*, *WARNING*, *ERROR*, *CRITICAL*, or *FATAL*.

class `celery.bin.celerybeat.`**BeatCommand**(*app=None*, *get_app=None*)

doc = '\n\nThe :program:'celery beat' command.\n\n.. program:: celery beat\n\n.. seealso::\n\n See :ref:'preload-options

enable_config_from_cmdline = True

get_options()

run(*detach=False*, *logfile=None*, *pidfile=None*, *uid=None*, *gid=None*, *umask=None*, *working_directory=None*, ***kwargs*)

supports_args = False

`celery.bin.celerybeat.`**main**()

2.13.38 celery.bin.celeryev

The **celery events** command.

See also:

See *Preload Options* and *Daemon Options*.

-d, --dump
Dump events to stdout.

-c, --camera
Take snapshots of events using this camera.

--detach
Camera: Detach and run in the background as a daemon.

-F, --freq, --frequency
Camera: Shutter frequency. Default is every 1.0 seconds.

-r, --maxrate
Camera: Optional shutter rate limit (e.g. 10/m).

-l, --loglevel
Logging level, choose between *DEBUG*, *INFO*, *WARNING*, *ERROR*, *CRITICAL*, or *FATAL*. Default is INFO.

class celery.bin.celeryev.**EvCommand**(*app=None*, *get_app=None*)

doc = '\n\nThe :program:`celery events` command.\n\n.. program:: celery events\n\n.. seealso::\n\n See :ref:`preload-opti

get_options()

run(*dump=False*, *camera=None*, *frequency=1.0*, *maxrate=None*, *loglevel='INFO'*, *logfile=None*, *prog_name='celeryev'*, *pidfile=None*, *uid=None*, *gid=None*, *umask=None*, *working_directory=None*, *detach=False*, ***kwargs*)

run_evcam(*camera*, *logfile=None*, *pidfile=None*, *uid=None*, *gid=None*, *umask=None*, *working_directory=None*, *detach=False*, ***kwargs*)

run_evdump()

run_evtop()

set_process_status(*prog*, *info=''*)

supports_args = False

celery.bin.celeryev.**main**()

2.13.39 celery.bin.celery

The **celery** umbrella command.

class celery.bin.celery.**CeleryCommand**(*app=None*, *get_app=None*)

commands = {'control': <class 'celery.bin.celery.control'>, 'status': <class 'celery.bin.celery.status'>, 'multi': <class 'cele

enable_config_from_cmdline = True

execute(*command*, *argv=None*)

execute_from_commandline(*argv=None*)

classmethod get_command_info(*command*, *indent=0*, *color=None*)

handle_argv(*prog_name*, *argv*)

classmethod list_commands(*indent=0*)

on_concurrency_setup()

prog_name = 'celery'

remove_options_at_beginning(*argv*, *index=0*)

with_pool_option(*argv*)

class celery.bin.celery.**Command**(*app=None*, *no_color=False*, *stdout=<open file '<stdout>', mode 'w' at 0x7fb76c556150>*, *stderr=<open file '<stderr>', mode 'w' at 0x7fb76c5561e0>*, *show_reply=True*)

args = ''

description

error(*s*)

help = ‘’

option_list = (<Option at 0x7fb762af2a70: -q/–quiet>, <Option at 0x7fb762af2098: -C/–no-color>)

out(*s*, *fh=None*)

prettify(*n*)

prettify_dict_ok_error(*n*)

prettify_list(*n*)

prog_name = ‘celery’

run_from_argv(*prog_name*, *argv*)

say_chat(*direction*, *title*, *body=’‘*)

say_remote_command_reply(*replies*)

show_body = True

show_help(*command*)

show_reply = True

usage(*command*)

class celery.bin.celery.**Delegate**(**args*, ***kwargs*)

create_parser(*prog_name*, *command*)

get_options()

run(**args*, ***kwargs*)

exception celery.bin.celery.**Error**(*reason*, *status=1*)

class celery.bin.celery.**amqp**(**args*, ***kwargs*)
AMQP Administration Shell.

Also works for non-amqp transports.

Examples:

```
celery amqp
    start shell mode
celery amqp help
    show list of commands

celery amqp exchange.delete name
celery amqp queue.delete queue
celery amqp queue.delete queue yes yes
```

Command = ‘celery.bin.camqadm:AMQPAdminCommand’

sortpri = 30

class celery.bin.celery.**beat**(**args*, ***kwargs*)
Start the celerybeat periodic task scheduler.

Examples:

```
celery beat -l info
celery beat -s /var/run/celerybeat/schedule --detach
celery beat -S djcelery.schedulers.DatabaseScheduler
```

Command = 'celery.bin.celerybeat:BeatCommand'

sortpri = 20

class celery.bin.celery.**call**(*app=None*, *no_color=False*, *stdout=<open file '<stdout>'*, *mode 'w' at 0x7fb76c556150>*, *stderr=<open file '<stderr>'*, *mode 'w' at 0x7fb76c5561e0>*, *show_reply=True*)

Call a task by name.

Examples:

```
celery call tasks.add --args='[2, 2]'
celery call tasks.add --args='[2, 2]' --countdown=10
```

args = '<task_name>'

option_list = (<Option at 0x7fb762af2a70: -q/–quiet>, <Option at 0x7fb762af2098: -C/–no-color>, <Option at 0x7fb7

run(*name*, *_, **kw*)

sortpri = 0

celery.bin.celery.**command**(*fun*, *name=None*, *sortpri=0*)

class celery.bin.celery.**control**(*app=None*, *no_color=False*, *stdout=<open file '<stdout>'*, *mode 'w' at 0x7fb76c556150>*, *stderr=<open file '<stderr>'*, *mode 'w' at 0x7fb76c5561e0>*, *show_reply=True*)

Workers remote control.

Availability: RabbitMQ (amqp), Redis, and MongoDB transports.

Examples:

```
celery control enable_events --timeout=5
celery control -d worker1.example.com enable_events
celery control -d w1.e.com,w2.e.com enable_events

celery control -d w1.e.com add_consumer queue_name
celery control -d w1.e.com cancel_consumer queue_name

celery control -d w1.e.com add_consumer queue exchange direct rkey
```

add_consumer(*method*, *queue*, *exchange=None*, *exchange_type='direct'*, *routing_key=None*, ***kwargs*)

<queue> [exchange [type [routing_key]]]

autoscale(*method*, *max=None*, *min=None*, ***kwargs*)

[max] [min]

call(*method*, **args*, ***options*)

cancel_consumer(*method*, *queue*, ***kwargs*)

<queue>

choices = {'time_limit': (1.0, 'tell worker(s) to modify the time limit for a task type.'), 'cancel_consumer': (1.0, 'tell wor

name = 'control'

pool_grow(*method*, *n=1*, ***kwargs*)

[N=1]

pool_shrink(*method*, *n=1*, ***kwargs*)
[N=1]

rate_limit(*method*, *task_name*, *rate_limit*, ***kwargs*)
<task_name> <rate_limit> (e.g. 5/s | 5/m | 5/h)>

sortpri = 0

time_limit(*method*, *task_name*, *soft*, *hard=None*, ***kwargs*)
<task_name> <soft_secs> [hard_secs]

celery.bin.celery.**determine_exit_status**(*ret*)

celery.bin.celery.**ensure_broadcast_supported**(*app*)

class celery.bin.celery.**events**(**args*, ***kwargs*)
Event-stream utilities.

Commands:

```
celery events --app=proj
    start graphical monitor (requires curses)
celery events -d --app=proj
    dump events to screen.
celery events -b amqp://
celery events -C <camera> [options]
    run snapshot camera.
```

Examples:

```
celery events
celery events -d
celery events -C mod.attr -F 1.0 --detach --maxrate=100/m -l info
```

Command = 'celery.bin.celeryev:EvCommand'

sortpri = 10

class celery.bin.celery.**help**(*app=None*, *no_color=False*, *stdout=<open file '<stdout>'*, *mode 'w' at 0x7fb76c556150>*, *stderr=<open file '<stderr>'*, *mode 'w' at 0x7fb76c5561e0>*, *show_reply=True*)
Show help screen and exit.

run(**args*, ***kwargs*)

sortpri = 0

usage(*command*)

class celery.bin.celery.**inspect**(*app=None*, *no_color=False*, *stdout=<open file '<stdout>'*, *mode 'w' at 0x7fb76c556150>*, *stderr=<open file '<stderr>'*, *mode 'w' at 0x7fb76c5561e0>*, *show_reply=True*)
Inspect the worker at runtime.

Availability: RabbitMQ (amqp), Redis, and MongoDB transports.

Examples:

```
celery inspect active --timeout=5
celery inspect scheduled -d worker1.example.com
celery inspect revoked -d w1.e.com,w2.e.com
```

call(*method*, **args*, ***options*)

choices = {'scheduled': (1.0, 'dump scheduled tasks (eta/countdown/retry)'), 'active_queues': (1.0, 'dump queues being

name = 'inspect'

sortpri = 0

class celery.bin.celery.**list_**(*app=None*, *no_color=False*, *stdout=<open file '<stdout>', mode 'w' at 0x7fb76c556150>*, *stderr=<open file '<stderr>', mode 'w' at 0x7fb76c5561e0>*, *show_reply=True*)

Get info from broker.

Examples:

```
celery list bindings
```

NOTE: For RabbitMQ the management plugin is required.

args = '[bindings]'

list_bindings(*management*)

run(*what=None*, **_*, ***kw*)

sortpri = 0

celery.bin.celery.**load_extension_commands**(*namespace='celery.commands'*)

celery.bin.celery.**main**(*argv=None*)

class celery.bin.celery.**migrate**(*app=None*, *no_color=False*, *stdout=<open file '<stdout>', mode 'w' at 0x7fb76c556150>*, *stderr=<open file '<stderr>', mode 'w' at 0x7fb76c5561e0>*, *show_reply=True*)

Migrate tasks from one broker to another.

Examples:

```
celery migrate redis://localhost amqp://guest@localhost//
celery migrate django:// redis://localhost
```

NOTE: This command is experimental, make sure you have a backup of the tasks before you continue.

args = '<source_url> <dest_url>'

on_migrate_task(*state*, *body*, *message*)

option_list = (<Option at 0x7fb762af2a70: -q/–quiet>, <Option at 0x7fb762af2098: -C/–no-color>, <Option at 0x7fb7

run(**args*, ***kwargs*)

sortpri = 0

class celery.bin.celery.**multi**(*app=None*, *no_color=False*, *stdout=<open file '<stdout>', mode 'w' at 0x7fb76c556150>*, *stderr=<open file '<stderr>', mode 'w' at 0x7fb76c5561e0>*, *show_reply=True*)

Start multiple worker instances.

get_options()

respects_app_option = False

run_from_argv(*prog_name*, *argv*)

sortpri = 0

class celery.bin.celery.**purge**(*app=None*, *no_color=False*, *stdout=<open file '<stdout>', mode 'w' at 0x7fb76c556150>*, *stderr=<open file '<stderr>', mode 'w' at 0x7fb76c5561e0>*, *show_reply=True*)

Erase all messages from all known task queues.

WARNING: There is no undo operation for this command.

run(**args*, ***kwargs*)

sortpri = 0

class celery.bin.celery.**report**(*app=None*, *no_color=False*, *stdout=<open file '<stdout>', mode 'w' at 0x7fb76c556150>*, *stderr=<open file '<stderr>', mode 'w' at 0x7fb76c5561e0>*, *show_reply=True*)

Shows information useful to include in bugreports.

run(**args*, ***kwargs*)

sortpri = 0

class celery.bin.celery.**result**(*app=None*, *no_color=False*, *stdout=<open file '<stdout>', mode 'w' at 0x7fb76c556150>*, *stderr=<open file '<stderr>', mode 'w' at 0x7fb76c5561e0>*, *show_reply=True*)

Gives the return value for a given task id.

Examples:

```
celery result 8f511516-e2f5-4da4-9d2f-0fb83a86e500
celery result 8f511516-e2f5-4da4-9d2f-0fb83a86e500 -t tasks.add
celery result 8f511516-e2f5-4da4-9d2f-0fb83a86e500 --traceback
```

args = '<task_id>'

option_list = (<Option at 0x7fb762af2a70: -q/–quiet>, <Option at 0x7fb762af2098: -C/–no-color>, <Option at 0x7fb7

run(*task_id*, **args*, ***kwargs*)

sortpri = 0

class celery.bin.celery.**shell**(*app=None*, *no_color=False*, *stdout=<open file '<stdout>', mode 'w' at 0x7fb76c556150>*, *stderr=<open file '<stderr>', mode 'w' at 0x7fb76c5561e0>*, *show_reply=True*)

Start shell session with convenient access to celery symbols.

The following symbols will be added to the main globals:

- celery: the current application.
- chord, group, chain, chunks, xmap, xstarmap subtask, Task
- all registered tasks.

Example Session:

```
$ celery shell

>>> celery
<Celery default:0x1012d9fd0>
>>> add
<@task: tasks.add>
>>> add.delay(2, 2)
<AsyncResult: 537b48c7-d6d3-427a-a24a-d1b4414035be>
```

invoke_bpython_shell()

invoke_default_shell()

invoke_fallback_shell()

invoke_ipython_shell()

option_list = (<Option at 0x7fb762af2a70: -q/–quiet>, <Option at 0x7fb762af2098: -C/–no-color>, <Option at 0x7fb7

run(*force_ipython=False*, *force_bpython=False*, *force_python=False*, *without_tasks=False*, *eventlet=False*, *gevent=False*, ***kwargs*)

sortpri = 0

class celery.bin.celery.**status**(*app=None*, *no_color=False*, *stdout=<open file '<stdout>', mode 'w' at 0x7fb76c556150>*, *stderr=<open file '<stderr>', mode 'w' at 0x7fb76c5561e0>*, *show_reply=True*)
Show list of workers that are online.

option_list = (<Option at 0x7fb762af2a70: -q/–quiet>, <Option at 0x7fb762af2098: -C/–no-color>, <Option at 0x7fb7

run(**args*, ***kwargs*)

sortpri = 0

class celery.bin.celery.**worker**(**args*, ***kwargs*)
Start worker instance.

Examples:

```
celery worker --app=proj -l info
celery worker -A proj -l info -Q hipri,lopri

celery worker -A proj --concurrency=4
celery worker -A proj --concurrency=1000 -P eventlet

celery worker --autoscale=10,0
```

Command = 'celery.bin.celeryd:WorkerCommand'

sortpri = 1

2.13.40 celery.bin.camqadm

The **celery amqp** command.

class celery.bin.camqadm.**AMQPAdmin**(**args*, ***kwargs*)
The celery **camqadm** utility.

Shell
alias of AMQShell

connect(*conn=None*)

note(*m*)

run()

class celery.bin.camqadm.**AMQPAdminCommand**(*app=None*, *get_app=None*)

run(**args*, ***options*)

class celery.bin.camqadm.**AMQShell**(**args*, ***kwargs*)
AMQP API Shell.

Parameters

- **connect** – Function used to connect to the server, must return connection object.
- **silent** – If True, the commands won't have annoying output not relevant when running in non-shell mode.

amqp
Mapping of AMQP API commands and their Spec.

amqp = {'queue.declare': <celery.bin.camqadm.Spec object at 0x7fb7635bc310>, 'queue.purge': <celery.bin.camqadm.Sp

builtins = {'exit': 'do_exit', 'EOF': 'do_exit', 'help': 'do_help'}

chan = None

completenames(*text*, **ignored*)
Return all commands starting with *text*, for tab-completion.

conn = None

counter = 1

default(*line*)

dispatch(*cmd*, *argline*)
Dispatch and execute the command.

Lookup order is: builtins -> amqp.

display_command_help(*cmd*, *short=False*)

do_exit(**args*)
The *'exit'* command.

do_help(**args*)

get_amqp_api_command(*cmd*, *arglist*)
With a command name and a list of arguments, convert the arguments to Python values and find the corresponding method on the AMQP channel object.

Returns tuple of *(method, processed_args)*.

Example:

```
>>> get_amqp_api_command('queue.delete', ['pobox', 'yes', 'no'])
(<bound method Channel.queue_delete of
 <amqp.channel.Channel object at 0x...>>,
 ('testfoo', True, False))
```

get_names()

identchars = '.'

inc_counter = <method-wrapper 'next' of itertools.count object at 0x7fb7631ddd40>

needs_reconnect = False

note(*m*)
Say something to the user. Disabled if silent.

onecmd(*line*)
Parse line and execute command.

parseline(*line*)
Parse input line.

Returns tuple of three items: *(command_name, arglist, original_line)*

E.g:

```
>>> parseline('queue.delete A 'B' C')
('queue.delete', 'A 'B' C', 'queue.delete A 'B' C')
```

prompt

prompt_fmt = '%d> '

respond(*retval*)
What to do with the return value of a command.

say(*m*)

class celery.bin.camqadm.**Spec**(**args*, ***kwargs*)
AMQP Command specification.

Used to convert arguments to Python values and display various help and tooltips.

Parameters

- **args** – see `args`.
- **returns** – see `returns`.

coerce(*index*, *value*)
Coerce value for argument at index.

E.g. if `args` is *[('is_active', bool)]*:

```
>>> coerce(0, 'False')
False
```

format_arg(*name*, *type*, *default_value=None*)

format_response(*response*)
Format the return value of this command in a human-friendly way.

format_signature()

str_args_to_python(*arglist*)
Process list of string arguments to values according to spec.

e.g:

```
>>> spec = Spec([('queue', str), ('if_unused', bool)])
>>> spec.str_args_to_python('pobox', 'true')
('pobox', True)
```

celery.bin.camqadm.**camqadm**(**args*, ***options*)

celery.bin.camqadm.**dump_message**(*message*)

celery.bin.camqadm.**format_declare_queue**(*ret*)

celery.bin.camqadm.**main**()

celery.bin.camqadm.**say**(*m*, *fh=<open file '<stderr>', mode 'w' at 0x7fb76c5561e0>*)

2.13.41 celery.bin.celeryd_multi

Examples

```
# Single worker with explicit name and events enabled.
$ celeryd-multi start Leslie -E

# Pidfiles and logfiles are stored in the current directory
# by default.  Use --pidfile and --logfile argument to change
# this.  The abbreviation %n will be expanded to the current
# node name.
$ celeryd-multi start Leslie -E --pidfile=/var/run/celery/%n.pid
                                --logfile=/var/log/celery/%n.log


# You need to add the same arguments when you restart,
# as these are not persisted anywhere.
$ celeryd-multi restart Leslie -E --pidfile=/var/run/celery/%n.pid
                                  --logfile=/var/run/celery/%n.log

# To stop the node, you need to specify the same pidfile.
$ celeryd-multi stop Leslie --pidfile=/var/run/celery/%n.pid

# 3 workers, with 3 processes each
$ celeryd-multi start 3 -c 3
celeryd -n celeryd1.myhost -c 3
celeryd -n celeryd2.myhost -c 3
celeryd- n celeryd3.myhost -c 3

# start 3 named workers
$ celeryd-multi start image video data -c 3
celeryd -n image.myhost -c 3
celeryd -n video.myhost -c 3
celeryd -n data.myhost -c 3

# specify custom hostname
$ celeryd-multi start 2 -n worker.example.com -c 3
celeryd -n celeryd1.worker.example.com -c 3
celeryd -n celeryd2.worker.example.com -c 3

# Advanced example starting 10 workers in the background:
#   * Three of the workers processes the images and video queue
#   * Two of the workers processes the data queue with loglevel DEBUG
#   * the rest processes the default' queue.
$ celeryd-multi start 10 -l INFO -Q:1-3 images,video -Q:4,5 data
    -Q default -L:4,5 DEBUG

# You can show the commands necessary to start the workers with
# the 'show' command:
$ celeryd-multi show 10 -l INFO -Q:1-3 images,video -Q:4,5 data
    -Q default -L:4,5 DEBUG

# Additional options are added to each celeryd',
# but you can also modify the options for ranges of, or specific workers

# 3 workers: Two with 3 processes, and one with 10 processes.
$ celeryd-multi start 3 -c 3 -c:1 10
celeryd -n celeryd1.myhost -c 10
celeryd -n celeryd2.myhost -c 3
celeryd -n celeryd3.myhost -c 3
```

```
# can also specify options for named workers
$ celeryd-multi start image video data -c 3 -c:image 10
celeryd -n image.myhost -c 10
celeryd -n video.myhost -c 3
celeryd -n data.myhost -c 3

# ranges and lists of workers in options is also allowed:
# (-c:1-3 can also be written as -c:1,2,3)
$ celeryd-multi start 5 -c 3  -c:1-3 10
celeryd -n celeryd1.myhost -c 10
celeryd -n celeryd2.myhost -c 10
celeryd -n celeryd3.myhost -c 10
celeryd -n celeryd4.myhost -c 3
celeryd -n celeryd5.myhost -c 3

# lists also works with named workers
$ celeryd-multi start foo bar baz xuzzy -c 3 -c:foo,bar,baz 10
celeryd -n foo.myhost -c 10
celeryd -n bar.myhost -c 10
celeryd -n baz.myhost -c 10
celeryd -n xuzzy.myhost -c 3
```

class celery.bin.celeryd_multi.**MultiTool**(*env=None*, *fh=None*, *quiet=False*, *verbose=False*, *no_color=False*, *nosplash=False*)

DOWN

FAILED

OK

colored

error(*msg=None*)

execute_from_commandline(*argv*, *cmd='celeryd'*)

expand(*argv*, *cmd=None*)

get(*argv*, *cmd*)

getpids(*p*, *cmd*, *callback=None*)

help(*argv*, *cmd=None*)

info(*msg*, *newline=True*)

kill(*argv*, *cmd*)

names(*argv*, *cmd*)

node_alive(*pid*)

note(*msg*, *newline=True*)

restart(*argv*, *cmd*)

retcode = **0**

say(*m*, *newline=True*)

show(*argv*, *cmd*)

shutdown_nodes(*nodes*, *sig=15*, *retry=None*, *callback=None*)

signal_node(*nodename*, *pid*, *sig*)

splash()

start(*argv*, *cmd*)

stop(*argv*, *cmd*, *retry=None*, *callback=None*)

stop_verify(*argv*, *cmd*)

stopwait(*argv*, *cmd*)

usage()

waitexec(*argv*, *path='/home/docs/checkouts/readthedocs.org/user_builds/celery/envs/3.0/bin/python'*)

with_detacher_default_options(*p*)

class celery.bin.celeryd_multi.**NamespacedOptionParser**(*args*)

add_option(*name*, *value*, *short=False*, *ns=None*)

optmerge(*ns*, *defaults=None*)

parse()

process_long_opt(*arg*, *value=None*)

process_short_opt(*arg*, *value=None*)

celery.bin.celeryd_multi.**abbreviations**(*mapping*)

celery.bin.celeryd_multi.**findsig**(*args*, *default=15*)

celery.bin.celeryd_multi.**format_opt**(*opt*, *value*)

celery.bin.celeryd_multi.**main**()

celery.bin.celeryd_multi.**multi_args**(*p*, *cmd='celeryd'*, *append=''*, *prefix=''*, *suffix=''*)

celery.bin.celeryd_multi.**parse_ns_range**(*ns*, *ranges=False*)

celery.bin.celeryd_multi.**quote**(*v*)

2.14 Internals

Release 3.0

Date July 10, 2014

2.14.1 Contributors Guide to the Code

Philosophy

The API>RCP Precedence Rule

- The API is more important than Readability
- Readability is more important than Convention
- **Convention is more important than Performance**
  - ...unless the code is a proven hotspot.

More important than anything else is the end-user API. Conventions must step aside, and any suffering is always alleviated if the end result is a better API.

Conventions and Idioms Used

Classes

Naming

- Follows **PEP 8**.
- Class names must be *CamelCase*.
- but not if they are verbs, verbs shall be *lower_case*:

```
# - test case for a class
class TestMyClass(Case):                # BAD
    pass

class test_MyClass(Case):               # GOOD
    pass

# - test case for a function
class TestMyFunction(Case):             # BAD
    pass

class test_my_function(Case):           # GOOD
    pass

# - "action" class (verb)
class UpdateTwitterStatus(object):      # BAD
    pass
```

```
class update_twitter_status(object):    # GOOD
    pass
```

Note: Sometimes it makes sense to have a class mask as a function, and there is precedence for this in the stdlib (e.g. `contextmanager`). Celery examples include `subtask`, `chord`, `inspect`, `promise` and more..

- Factory functions and methods must be *CamelCase* (excluding verbs):

```
class Celery(object):

    def consumer_factory(self):       # BAD
        ...

    def Consumer(self):               # GOOD
        ...
```

Default values Class attributes serve as default values for the instance, as this means that they can be set by either instantiation or inheritance.

Example:

```
class Producer(object):
    active = True
    serializer = "json"

    def __init__(self, serializer=None):
        self.serializer = serializer or self.serializer

        # must check for None when value can be false-y
        self.active = active if active is not None else self.active
```

A subclass can change the default value:

```
TaskProducer(Producer):
    serializer = "pickle"
```

and the value can be set at instantiation:

```
>>> producer = TaskProducer(serializer="msgpack")
```

Exceptions Custom exceptions raised by an objects methods and properties should be available as an attribute and documented in the method/property that throw.

This way a user doesn't have to find out where to import the exception from, but rather use `help(obj)` and access the exception class from the instance directly.

Example:

```
class Empty(Exception):
    pass

class Queue(object):
```

```
    Empty = Empty

    def get(self):
        """Get the next item from the queue.

        :raises Queue.Empty: if there are no more items left.

        """
        try:
            return self.queue.popleft()
        except IndexError:
            raise self.Empty()
```

Composites Similarly to exceptions, composite classes should be override-able by inheritance and/or instantiation. Common sense can be used when selecting what classes to include, but often it's better to add one too many: predicting what users need to override is hard (this has saved us from many a monkey patch).

Example:

```
class Worker(object):
    Consumer = Consumer

    def __init__(self, connection, consumer_cls=None):
        self.Consumer = consumer_cls or self.Consumer

    def do_work(self):
        with self.Consumer(self.connection) as consumer:
            self.connection.drain_events()
```

Applications vs. "single mode"

In the beginning Celery was developed for Django, simply because this enabled us get the project started quickly, while also having a large potential user base.

In Django there is a global settings object, so multiple Django projects can't co-exist in the same process space, this later posed a problem for using Celery with frameworks that doesn't have this limitation.

Therefore the app concept was introduced. When using apps you use 'celery' objects instead of importing things from celery submodules, this sadly also means that Celery essentially has two API's.

Here's an example using Celery in single-mode:

```
from celery import task
from celery.task.control import inspect

from .models import CeleryStats

@task
def write_stats_to_db():
    stats = inspect().stats(timeout=1)
    for node_name, reply in stats:
        CeleryStats.objects.update_stat(node_name, stats)
```

and here's the same using Celery app objects:

```
from .celery import celery
from .models import CeleryStats

@celery.task
def write_stats_to_db():
    stats = celery.control.inspect().stats(timeout=1)
    for node_name, reply in stats:
        CeleryStats.objects.update_stat(node_name, stats)
```

In the example above the actual application instance is imported from a module in the project, this module could look something like this:

```
from celery import Celery

celery = Celery()
celery.config_from_object(BROKER_URL="amqp://")
```

Module Overview

- celery.app

  This is the core of Celery: the entry-point for all functionality.

- celery.loaders

  Every app must have a loader. The loader decides how configuration is read, what happens when the worker starts, when a task starts and ends, and so on.

  The loaders included are:

  - app

    Custom celery app instances uses this loader by default.

  - default

    "single-mode" uses this loader by default.

  Extension loaders also exist, like `django-celery`, `celery-pylons` and so on.

- celery.worker

  This is the worker implementation.

- celery.backends

  Task result backends live here.

- celery.apps

  Major user applications: `celeryd`, and `celerybeat`

- celery.bin

  Command line applications. setup.py creates setuptools entrypoints for these.

- celery.concurrency

  Execution pool implementations (processes, eventlet, gevent, threads).

- celery.db

Database models for the SQLAlchemy database result backend. (should be moved into `celery.backends.database`)

- celery.events

  Sending and consuming monitoring events, also includes curses monitor, event dumper and utilities to work with in-memory cluster state.

- celery.execute.trace

  How tasks are executed and traced by the worker, and in eager mode.

- celery.security

  Security related functionality, currently a serializer using cryptographic digests.

- celery.task

  single-mode interface to creating tasks, and controlling workers.

- celery.tests

  The unittest suite.

- celery.utils

  Utility functions used by the celery code base. Much of it is there to be compatible across Python versions.

- celery.contrib

  Additional public code that doesn't fit into any other namespace.

2.14.2 Celery Deprecation Timeline

Removals for version 4.0

Old Task API

Compat Task Modules

- Module `celery.decorators` will be removed:

  Which means you need to change:

  ```
  from celery.decorators import task
  ```

Into:

```
from celery import task
```

- Module `celery.task` *may* be removed (not decided)

  This means you should change:

  ```
  from celery.task import task
  ```

  into:

  ```
  from celery import task
  ```

  —and::

  ```
  from celery.task import Task
  ```

  into:

  ```
  from celery import Task
  ```

Note that the new `Task` class no longer uses classmethods for these methods:

- delay
- apply_async
- retry
- apply
- AsyncResult
- subtask

This also means that you can't call these methods directly on the class, but have to instantiate the task first:

```
>>> MyTask.delay()           # NO LONGER WORKS

>>> MyTask().delay()         # WORKS!
```

TaskSet TaskSet has been renamed to group and TaskSet will be removed in version 4.0.

Old:

```
>>> from celery.task import TaskSet

>>> TaskSet(add.subtask((i, i)) for i in xrange(10)).apply_async()
```

New:

```
>>> from celery import group
>>> group(add.s(i, i) for i in xrange(10))()
```

Magic keyword arguments The magic keyword arguments accepted by tasks will be removed in 4.0, so you should start rewriting any tasks using the `celery.decorators` module and depending on keyword arguments being passed to the task, for example:

```
from celery.decorators import task

@task()
def add(x, y, task_id=None):
    print("My task id is %r" % (task_id, ))
```

must be rewritten into:

```
from celery import task

@task()
def add(x, y):
    print("My task id is %r" % (add.request.id, ))
```

Task attributes

The task attributes:

- `queue`
- `exchange`
- `exchange_type`
- `routing_key`
- `delivery_mode`
- `priority`

is deprecated and must be set by `CELERY_ROUTES` instead.

`celery.result`

- `BaseAsyncResult` -> `AsyncResult`.
- `TaskSetResult` -> `GroupResult`.
- `TaskSetResult.total` -> `len(GroupResult)`
- `TaskSetResult.taskset_id` -> `GroupResult.id`

Apply to: `AsyncResult`, `EagerResult`:

```
- ``Result.wait()`` -> ``Result.get()``
```

- `Result.task_id()` -> `Result.id`
- `Result.status` -> `Result.state`.

celery.loader

- current_loader() -> current_app.loader
- load_settings() -> current_app.conf

Modules to Remove

- celery.execute

  This module only contains send_task, which must be replaced with celery.send_task instead.

- celery.decorators

  See *Compat Task Modules*

- celery.log

  Use celery.log instead.

- celery.messaging

  Use celery.amqp instead.

- celery.registry

  Use celery.app.registry instead.

- celery.task.control

  Use celery.control instead.

- celery.task.schedules

  Use celery.schedules instead.

- celery.task.chords

  Use celery.chord() instead.

Settings

BROKER Settings

Setting name	Replace with
BROKER_HOST	BROKER_URL
BROKER_PORT	BROKER_URL
BROKER_USER	BROKER_URL
BROKER_PASSWORD	BROKER_URL
BROKER_VHOST	BROKER_URL
BROKER_INSIST	*no alternative*

REDIS Result Backend Settings

Setting name	Replace with
CELERY_REDIS_HOST	CELERY_RESULT_BACKEND
CELERY_REDIS_PORT	CELERY_RESULT_BACKEND
CELERY_REDIS_DB	CELERY_RESULT_BACKEND
CELERY_REDIS_PASSWORD	CELERY_RESULT_BACKEND
REDIS_HOST	CELERY_RESULT_BACKEND
REDIS_PORT	CELERY_RESULT_BACKEND
REDIS_DB	CELERY_RESULT_BACKEND
REDIS_PASSWORD	CELERY_RESULT_BACKEND

Logging Settings

Setting name	Replace with
CELERYD_LOG_LEVEL	*--loglevel*
CELERYD_LOG_FILE	*--logfile* '
CELERYBEAT_LOG_LEVEL	*--loglevel*
CELERYBEAT_LOG_FILE	*--loglevel* '
CELERYMON_LOG_LEVEL	*--loglevel*
CELERYMON_LOG_FILE	*--loglevel* '

Other Settings

Setting name	Replace with
CELERY_TASK_ERROR_WITELIST	Annotate `Task.ErrorMail`
CELERY_AMQP_TASK_RESULT_EXPIRES	CELERY_TASK_RESULT_EXPIRES

Removals for version 2.0

- The following settings will be removed:

Setting name	Replace with
CELERY_AMQP_CONSUMER_QUEUES	*CELERY_QUEUES*
CELERY_AMQP_CONSUMER_QUEUES	*CELERY_QUEUES*
CELERY_AMQP_EXCHANGE	*CELERY_DEFAULT_EXCHANGE*
CELERY_AMQP_EXCHANGE_TYPE	*CELERY_DEFAULT_AMQP_EXCHANGE_TYPE*
CELERY_AMQP_CONSUMER_ROUTING_KEY	*CELERY_QUEUES*
CELERY_AMQP_PUBLISHER_ROUTING_KEY	*CELERY_DEFAULT_ROUTING_KEY*

- `CELERY_LOADER` definitions without class name.

  E.g. *celery.loaders.default*, needs to include the class name: *celery.loaders.default.Loader*.

- **`TaskSet.run()`. Use `celery.task.base.TaskSet.apply_async()`** instead.
- The module `celery.task.rest`; use `celery.task.http` instead.

2.14.3 Internals: The worker

Introduction

The worker consists of 4 main components: the consumer, the scheduler, the mediator and the task pool. All these components runs in parallel working with two data structures: the ready queue and the ETA schedule.

Data structures

ready_queue

The ready queue is either an instance of `Queue.Queue`, or `celery.buckets.TaskBucket`. The latter if rate limiting is enabled.

eta_schedule

The ETA schedule is a heap queue sorted by time.

Components

Consumer

Receives messages from the broker using Kombu.

When a message is received it's converted into a `celery.worker.job.TaskRequest` object.

Tasks with an ETA are entered into the *eta_schedule*, messages that can be immediately processed are moved directly to the *ready_queue*.

ScheduleController

The schedule controller is running the *eta_schedule*. If the scheduled tasks eta has passed it is moved to the *ready_queue*, otherwise the thread sleeps until the eta is met (remember that the schedule is sorted by time).

Mediator

The mediator simply moves tasks in the *ready_queue* over to the task pool for execution using `celery.worker.job.TaskRequest.execute_using_pool()`.

TaskPool

This is a slightly modified `multiprocessing.Pool`. It mostly works the same way, except it makes sure all of the workers are running at all times. If a worker is missing, it replaces it with a new one.

2.14.4 Task Messages

Message format

- **task**

  string

  Name of the task. **required**

- **id**

  string

  Unique id of the task (UUID). **required**

- **args**

  list

  List of arguments. Will be an empty list if not provided.

- **kwargs**

  dictionary

  Dictionary of keyword arguments. Will be an empty dictionary if not provided.

- **retries**

  int

  Current number of times this task has been retried. Defaults to *0* if not specified.

- **eta**

  string **(ISO 8601)**

  Estimated time of arrival. This is the date and time in ISO 8601 format. If not provided the message is not scheduled, but will be executed asap.

- **expires**

  string **(ISO 8601)**

  New in version 2.0.2.

  Expiration date. This is the date and time in ISO 8601 format. If not provided the message will never expire. The message will be expired when the message is received and the expiration date has been exceeded.

Extensions

Extensions are additional keys in the message body that the worker may or may not support. If the worker finds an extension key it doesn't support it should optimally reject the message so another worker gets a chance to process it.

- **taskset**

  string

  The taskset this task is part of (if any).

- **chord**

  subtask

New in version 2.3.

Signifies that this task is one of the header parts of a chord. The value of this key is the body of the cord that should be executed when all of the tasks in the header has returned.

- **utc**

  bool

  New in version 2.5.

  If true time uses the UTC timezone, if not the current local timezone should be used.

- **callbacks**

  <list>subtask

  New in version 3.0.

  A list of subtasks to apply if the task exited successfully.

- **errbacks**

  <list>subtask

  New in version 3.0.

  A list of subtasks to apply if an error occurs while executing the task.

Example message

This is an example invocation of the *celery.task.PingTask* task in JSON format:

```
{"id": "4cc7438e-afd4-4f8f-a2f3-f46567e7ca77",
 "task": "celery.task.PingTask",
 "args": [],
 "kwargs": {},
 "retries": 0,
 "eta": "2009-11-17T12:30:56.527191"}
```

Serialization

Several types of serialization formats are supported using the *content_type* message header.

The MIME-types supported by default are shown in the following table.

Scheme	MIME Type
json	application/json
yaml	application/x-yaml
pickle	application/x-python-serialize
msgpack	application/x-msgpack

2.14.5 "The Big Instance" Refactor

The *app* branch is a work-in-progress to remove the use of a global configuration in Celery.

Celery can now be instantiated, which means several instances of Celery may exist in the same process space. Also, large parts can be customized without resorting to monkey patching.

Examples

Creating a Celery instance:

```
>>> from celery import Celery
>>> celery = Celery()
>>> celery.config_from_object("celeryconfig")
>>> celery.config_from_envvar("CELERY_CONFIG_MODULE")
```

Creating tasks:

```
@celery.task
def add(x, y):
    return x + y
```

Creating custom Task subclasses:

```
Task = celery.create_task_cls()

class DebugTask(Task):
    abstract = True

    def on_failure(self, *args, **kwargs):
        import pdb
        pdb.set_trace()

@celery.task(base=DebugTask)
def add(x, y):
    return x + y
```

Starting a worker:

```
worker = celery.Worker(loglevel="INFO")
```

Getting access to the configuration:

```
celery.conf.CELERY_ALWAYS_EAGER = True
celery.conf["CELERY_ALWAYS_EAGER"] = True
```

Controlling workers:

```
>>> celery.control.inspect().active()
>>> celery.control.rate_limit(add.name, "100/m")
>>> celery.control.broadcast("shutdown")
>>> celery.control.discard_all()
```

Other interesting attributes:

```
# Establish broker connection.
>>> celery.broker_connection()

# AMQP Specific features.
>>> celery.amqp
>>> celery.amqp.Router
```

```
>>> celery.amqp.get_queues()
>>> celery.amqp.get_task_consumer()

# Loader
>>> celery.loader

# Default backend
>>> celery.backend
```

As you can probably see, this really opens up another dimension of customization abilities.

Deprecations

- celery.task.ping celery.task.PingTask

  Inferior to the ping remote control command. Will be removed in Celery 2.3.

Removed deprecations

- ***celery.utils.timedelta_seconds*** Use: `celery.utils.timeutils.timedelta_seconds()`
- ***celery.utils.defaultdict*** Use: `celery.utils.compat.defaultdict()`
- ***celery.utils.all*** Use: `celery.utils.compat.all()`
- ***celery.task.apply_async*** Use app.send_task
- ***celery.task.tasks*** Use `celery.registry.tasks`

Aliases (Pending deprecation)

- **celery.task.base**
  - .Task -> {app.create_task_cls}
- **celery.task.sets**
  - .TaskSet -> {app.TaskSet}
- **celery.decorators / celery.task**
  - .task -> {app.task}
- **celery.execute**
  - .apply_async -> {task.apply_async}
  - .apply -> {task.apply}
  - .send_task -> {app.send_task}
  - .delay_task -> no alternative
- **celery.log**
  - .get_default_logger -> {app.log.get_default_logger}
  - .setup_logger -> {app.log.setup_logger}
  - .get_task_logger -> {app.log.get_task_logger}
  - .setup_task_logger -> {app.log.setup_task_logger}

  - .setup_logging_subsystem -> {app.log.setup_logging_subsystem}
  - .redirect_stdouts_to_logger -> {app.log.redirect_stdouts_to_logger}

- **celery.messaging**
  - .establish_connection -> {app.broker_connection}
  - .with_connection -> {app.with_connection}
  - .get_consumer_set -> {app.amqp.get_task_consumer}
  - .TaskPublisher -> {app.amqp.TaskPublisher}
  - .TaskConsumer -> {app.amqp.TaskConsumer}
  - .ConsumerSet -> {app.amqp.ConsumerSet}

- celery.conf.* -> {app.conf}

  NOTE: All configuration keys are now named the same as in the configuration. So the key "CELERY_ALWAYS_EAGER" is accessed as:

  ```
  >>> app.conf.CELERY_ALWAYS_EAGER
  ```

  instead of:

  ```
  >>> from celery import conf
  >>> conf.ALWAYS_EAGER
  ```

  - .get_queues -> {app.amqp.get_queues}

- **celery.task.control**
  - .broadcast -> {app.control.broadcast}
  - .rate_limit -> {app.control.rate_limit}
  - .ping -> {app.control.ping}
  - .revoke -> {app.control.revoke}
  - .discard_all -> {app.control.discard_all}
  - .inspect -> {app.control.inspect}

- **celery.utils.info**
  - .humanize_seconds -> celery.utils.timeutils.humanize_seconds
  - .textindent -> celery.utils.textindent
  - .get_broker_info -> {app.amqp.get_broker_info}
  - .format_broker_info -> {app.amqp.format_broker_info}
  - .format_queues -> {app.amqp.format_queues}

Default App Usage

To be backward compatible, it must be possible to use all the classes/functions without passing an explicit app instance.

This is achieved by having all app-dependent objects use `default_app` if the app instance is missing.

```
from celery.app import app_or_default

class SomeClass(object):

    def __init__(self, app=None):
        self.app = app_or_default(app)
```

The problem with this approach is that there is a chance that the app instance is lost along the way, and everything seems to be working normally. Testing app instance leaks is hard. The environment variable `CELERY_TRACE_APP` can be used, when this is enabled `celery.app.app_or_default()` will raise an exception whenever it has to go back to the default app instance.

App Dependency Tree

- **{app}**
  - celery.loaders.base.BaseLoader
  - celery.backends.base.BaseBackend
  - **{app.TaskSet}**
    - celery.task.sets.TaskSet (app.TaskSet)
  - **[app.TaskSetResult]**
    - celery.result.TaskSetResult (app.TaskSetResult)
- **{app.AsyncResult}**
  - celery.result.BaseAsyncResult / celery.result.AsyncResult
- **celery.bin.celeryd.WorkerCommand**
  - **celery.apps.worker.Worker**
    - **celery.worker.WorkerController**
      - **celery.worker.consumer.Consumer**

        celery.worker.job.TaskRequest

        celery.events.EventDispatcher

        celery.worker.control.ControlDispatch

        celery.woker.control.registry.Panel

        celery.pidbox.BroadcastPublisher

        celery.pidbox.BroadcastConsumer
      - celery.worker.controllers.Mediator
      - celery.beat.EmbeddedService
- **celery.bin.celeryev.run_celeryev**
  - **celery.events.snapshot.evcam**
    - celery.events.snapshot.Polaroid
    - celery.events.EventReceiver
  - **celery.events.cursesmon.evtop**

        - celery.events.EventReceiver
        - celery.events.cursesmon.CursesMonitor
    - **celery.events.dumper**
        - celery.events.EventReceiver
- **celery.bin.celeryctl.celeryctl**
    - celery.bin.celeryctl.Command
- celery.bin.caqmadm.AMQPAdmin
- **celery.bin.celerybeat.BeatCommand**
    - **celery.apps.beat.Beat**
        - **celery.beat.Service**
            - celery.beat.Scheduler

2.14.6 Internal Module Reference

Release 3.0

Date July 10, 2014

celery.worker

celery.worker

`WorkController` can be used to instantiate in-process workers.

The worker consists of several components, all managed by boot-steps (mod:*celery.worker.bootsteps*).

class `celery.worker.`**`Beat`**(*w*, *beat=False*, ***kwargs*)

Component used to embed a celerybeat process.

This will only be enabled if the `beat` argument is set.

`create`(*w*)

`name` = **'beat'**

`namespace` = **'worker'**

class `celery.worker.`**`EvLoop`**(*w*, ***kwargs*)

`create`(*w*)

`include_if`(*w*)

`name` = **'ev'**

`namespace` = **'worker'**

class celery.worker.**Namespace**(*name=None*, *app=None*)
This is the boot-step namespace of the WorkController.

It loads modules from CELERYD_BOOT_STEPS, and its own set of built-in boot-step modules.

builtin_boot_steps = ('celery.worker.autoscale', 'celery.worker.autoreload', 'celery.worker.consumer', 'celery.worke

modules()

name = 'worker'

class celery.worker.**Pool**(*w*, *autoscale=None*, *autoreload=False*, *no_execv=False*, ***kwargs*)
The pool component.

Describes how to initialize the worker pool, and starts and stops the pool during worker startup/shutdown.

Adds attributes:
- autoscale
- pool
- max_concurrency
- min_concurrency

create(*w*, *semaphore=None*, *max_restarts=None*)

name = 'pool'

namespace = 'worker'

on_poll_init(*pool*, *w*, *hub*)

requires = ('queues', 'beat')

class celery.worker.**Queues**(*parent*, ***kwargs*)
This component initializes the internal queues used by the worker.

create(*w*)

name = 'queues'

namespace = 'worker'

requires = ('ev',)

celery.worker.**RUN = 1**
Worker states

celery.worker.**SHUTDOWN_SOCKET_TIMEOUT = 5.0**
Default socket timeout at shutdown.

class celery.worker.**StateDB**(*w*, ***kwargs*)
This component sets up the workers state db if enabled.

create(*w*)

name = 'state-db'

namespace = 'worker'

class celery.worker.**Timers**(*parent*, ***kwargs*)
This component initializes the internal timers used by the worker.

create(*w*)

include_if(*w*)

name = 'timers'

namespace = 'worker'

on_timer_error(*exc*)

on_timer_tick(*delay*)

requires = ('pool',)

class celery.worker.**WorkController**(*loglevel=None*, *hostname=None*, *ready_callback=<function noop at 0x7fb768d6c848>*, *queues=None*, *app=None*, *pidfile=None*, *use_eventloop=None*, ***kwargs*)

Unmanaged worker instance.

CLOSE = 2

RUN = 1

TERMINATE = 3

app = None

autoreloader_cls = None

autoscaler_cls = None

concurrency = None

consumer_cls = None

disable_rate_limits = None

force_execv = None

logfile = None

loglevel = 40

max_tasks_per_child = None

mediator_cls = None

pool_cls = None

pool_putlocks = None

pool_restarts = None

prefetch_multiplier = None

process_task(*req*)

Process task by sending it to the pool of workers.

process_task_sem(*req*)

reload(*modules=None*, *reload=False*, *reloader=None*)

schedule_filename = None

scheduler_cls = None

send_events = None

should_use_eventloop()

signal_consumer_close()

start()

Starts the workers main loop.

state

state_db = None

stop(*in_sighandler=False*)
Graceful shutdown of the worker server.

task_soft_time_limit = None

task_time_limit = None

terminate(*in_sighandler=False*)
Not so graceful shutdown of the worker server.

timer_cls = None

timer_precision = None

worker_lost_wait = None

celery.worker.consumer

celery.worker.consumer

This module contains the component responsible for consuming messages from the broker, processing the messages and keeping the broker connections up and running.

- `start()` is an infinite loop, which only iterates again if the connection is lost. For each iteration (at start, or if the connection is lost) it calls `reset_connection()`, and starts the consumer by calling `consume_messages()`.
- `reset_connection()`, clears the internal queues, establishes a new connection to the broker, sets up the task consumer (+ QoS), and the broadcast remote control command consumer.

  Also if events are enabled it configures the event dispatcher and starts up the heartbeat thread.
- Finally it can consume messages. `consume_messages()` is simply an infinite loop waiting for events on the AMQP channels.

  Both the task consumer and the broadcast consumer uses the same callback: `receive_message()`.
- So for each message received the `receive_message()` method is called, this checks the payload of the message for either a *task* key or a *control* key.

  If the message is a task, it verifies the validity of the message converts it to a `celery.worker.job.Request`, and sends it to `on_task()`.

  If the message is a control command the message is passed to `on_control()`, which in turn dispatches the control command using the control dispatcher.

  It also tries to handle malformed or invalid messages properly, so the worker doesn't choke on them and die. Any invalid messages are acknowledged immediately and logged, so the message is not resent again, and again.
- If the task has an ETA/countdown, the task is moved to the *timer* so the `timer2.Timer` can schedule it at its deadline. Tasks without an eta are moved immediately to the *ready_queue*, so they can be picked up by the `Mediator` to be sent to the pool.
- When a task with an ETA is received the QoS prefetch count is also incremented, so another message can be reserved. When the ETA is met the prefetch count is decremented again, though this cannot happen immediately because most broker clients don't support doing broker requests across threads. Instead the current prefetch

count is kept as a shared counter, so as soon as `consume_messages()` detects that the value has changed it will send out the actual QoS event to the broker.

- Notice that when the connection is lost all internal queues are cleared because we can no longer ack the messages reserved in memory. However, this is not dangerous as the broker will resend them to another worker when the channel is closed.
- **WARNING**: `stop()` does not close the connection! This is because some pre-acked messages may be in processing, and they need to be finished before the channel is closed. For celeryd this means the pool must finish the tasks it has acked early, *then* close the connection.

class `celery.worker.consumer.`**BlockingConsumer**(*ready_queue*, *init_callback=<function noop at 0x7fb768d6c848>*, *send_events=False*, *hostname=None*, *initial_prefetch_count=2*, *pool=None*, *app=None*, *timer=None*, *controller=None*, *hub=None*, *amqheartbeat=None*, ***kwargs*)

consume_messages()

class `celery.worker.consumer.`**Component**(*parent*, ***kwargs*)

Consumer(*w*)

create(*w*)

last = True

name = 'consumer'

namespace = 'worker'

class `celery.worker.consumer.`**Consumer**(*ready_queue*, *init_callback=<function noop at 0x7fb768d6c848>*, *send_events=False*, *hostname=None*, *initial_prefetch_count=2*, *pool=None*, *app=None*, *timer=None*, *controller=None*, *hub=None*, *amqheartbeat=None*, ***kwargs*)

Listen for messages received from the broker and move them to the ready queue for task processing.

Parameters

- **ready_queue** – See `ready_queue`.
- **timer** – See `timer`.

add_task_queue(*queue*, *exchange=None*, *exchange_type=None*, *routing_key=None*, ***options*)

apply_eta_task(*task*)

Method called by the timer to apply a task with an ETA/countdown.

broadcast_consumer = None

The consumer used to consume broadcast commands.

cancel_task_queue(*queue*)

close()

close_connection()

Closes the current broker connection and all open channels.

connection = None

The broker connection.

consume_messages(*sleep=<built-in function sleep>*, *min=<built-in function min>*, *Empty=<class 'Queue.Empty'>*)

Consume messages forever (or until an exception is raised).

event_dispatcher = None
A celery.events.EventDispatcher for sending events.

handle_invalid_task(*body*, *message*, *exc*)

handle_unknown_message(*body*, *message*)

handle_unknown_task(*body*, *message*, *exc*)

heart = None
The thread that sends event heartbeats at regular intervals. The heartbeats are used by monitors to detect that a worker went offline/disappeared.

hostname = None
The current hostname. Defaults to the system hostname.

info
Returns information about this consumer instance as a dict.

This is also the consumer related info returned by `celeryctl stats`.

init_callback = None
Optional callback to be called when the connection is established. Will only be called once, even if the connection is lost and re-established.

initial_prefetch_count = 0
Initial QoS prefetch count for the task channel.

maybe_conn_error(*fun*)
Applies function but ignores any connection or channel errors raised.

maybe_shutdown()

on_control(*body*, *message*)
Process remote control command message.

on_decode_error(*message*, *exc*)
Callback called if an error occurs while decoding a message received.

Simply logs the error and acknowledges the message so it doesn't enter a loop.

Parameters
- **message** – The message with errors.
- **exc** – The original exception instance.

on_poll_init(*hub*)

on_task(*task*, *task_reserved=<built-in method add of set object at 0x7fb763d30de8>*, *to_system_tz=<bound method _Zone.to_system of <celery.utils.timeutils._Zone object at 0x7fb766cac950>>*)
Handle received task.

If the task has an *eta* we enter it into the ETA schedule, otherwise we move it the ready queue for immediate processing.

pidbox_node = None
The process mailbox (kombu pidbox node).

pool = None
The current worker pool instance.

ready_queue = None
The queue that holds tasks ready for immediate processing.

receive_message(*body*, *message*)
Handles incoming messages.

Parameters
- **body** – The message body.
- **message** – The kombu message object.

reset_connection()
Re-establish the broker connection and set up consumers, heartbeat and the event dispatcher.

reset_pidbox_node()
Sets up the process mailbox.

restart_count = -1

restart_heartbeat()
Restart the heartbeat thread.

This thread sends heartbeat events at intervals so monitors can tell if the worker is off-line/missing.

send_events = False
Enable/disable events.

start()
Start the consumer.

Automatically survives intermittent connection failure, and will retry establishing the connection and restart consuming messages.

stop()
Stop consuming.

Does not close the broker connection, so be sure to call `close_connection()` when you are finished with it.

stop_consumers(*close_connection=True*, *join=True*)
Stop consuming tasks and broadcast commands, also stops the heartbeat thread and event dispatcher.
Parameters **close_connection** – Set to False to skip closing the broker connection.

stop_pidbox_node()

task_consumer = None
The consumer used to consume task messages.

timer = None
A timer used for high-priority internal tasks, such as sending heartbeats.

update_strategies()

celery.worker.consumer.**INVALID_TASK_ERROR = 'Received invalid task message: %s\nThe message has been ignored**
Error message for when an invalid task message is received.

celery.worker.consumer.**PREFETCH_COUNT_MAX = 65535**
Prefetch count can't exceed short.

class celery.worker.consumer.**QoS**(*consumer*, *initial_value*)
Thread safe increment/decrement of a channels prefetch_count.
Parameters
- **consumer** – A `kombu.messaging.Consumer` instance.
- **initial_value** – Initial prefetch count value.

decrement_eventually(*n=1*)
Decrement the value, but do not update the channels QoS.

The MainThread will be responsible for calling `update()` when necessary.

increment_eventually(*n=1*)
Increment the value, but do not update the channels QoS.

The MainThread will be responsible for calling `update()` when necessary.

prev = None

set(*pcount*)
Set channel prefetch_count setting.

update()
Update prefetch count with current value.

celery.worker.consumer.**UNKNOWN_TASK_ERROR = 'Received unregistered task of type %s.\nThe message has been ign**
Error message for when an unregistered task is received.

celery.worker.consumer.**debug**(*msg*, **args*, ***kwargs*)

celery.worker.consumer.**dump_body**(*m*, *body*)

celery.worker.consumer.**task_reserved**()
Add an element to a set.

This has no effect if the element is already present.

celery.worker.job

celery.worker.job

This module defines the `Request` class, which specifies how tasks are executed.

class celery.worker.job.**Request**(*body*, *on_ack=<function noop at 0x7fb768d6c848>*, *hostname=None*, *eventer=None*, *app=None*, *connection_errors=None*, *request_dict=None*, *delivery_info=None*, *task=None*, ***opts*)
A request for task execution.

acknowledge()
Acknowledge task.

acknowledged

app

args

connection_errors

delivery_info

error_msg = ' Task %(name)s[%(id)s] raised exception: %(exc)s\n '
Format string used to log task failure.

eta

eventer

execute(*loglevel=None*, *logfile=None*)
Execute the task in a `trace_task()`.

Parameters

- **loglevel** – The loglevel used by the task.

- **logfile** – The logfile used by the task.

execute_using_pool(*pool*, ***kwargs*)
Like `execute()`, but using a worker pool.
Parameters **pool** – A `celery.concurrency.base.TaskPool` instance.
Raises **celery.exceptions.TaskRevokedError** if the task was revoked and ignored.

expires

extend_with_default_kwargs()
Extend the tasks keyword arguments with standard task arguments.

Currently these are *logfile*, *loglevel*, *task_id*, *task_name*, *task_retries*, and *delivery_info*.

See `celery.task.base.Task.run()` for more information.

Magic keyword arguments are deprecated and will be removed in version 4.0.

classmethod from_message(*message*, *body*, ***kwargs*)

hostname

id

ignored_msg = ' Task %(name)s[%(id)s] ignored\n '

info(*safe=False*)

internal_error_msg = ' Task %(name)s[%(id)s] INTERNAL ERROR: %(exc)s\n '
Format string used to log internal error.

kwargs

maybe_expire()
If expired, mark the task as revoked.

name

on_accepted(*pid*, *time_accepted*)
Handler called when task is accepted by worker pool.

on_ack

on_failure(*exc_info*)
Handler called if the task raised an exception.

on_retry(*exc_info*)
Handler called if the task should be retried.

on_success(*ret_value*, *now=None*)
Handler called if the task was successfully processed.

on_timeout(*soft*, *timeout*)
Handler called if the task times out.

repr_result(*result*, *maxlen=46*)

request_dict

retry_msg = 'Task %(name)s[%(id)s] retry: %(exc)s'
Format string used to log task retry.

revoked()
If revoked, skip task and mark state.

send_event(*type*, ***fields*)

shortinfo()

store_errors

success_msg = ' Task %(name)s[%(id)s] succeeded in %(runtime)ss: %(return_value)s\n '
Format string used to log task success.

task

task_id

task_name

terminate(*pool*, *signal=None*)

time_start

tzlocal

utc

worker_pid

class celery.worker.job.**TaskRequest**(*name*, *id*, *args=()*, *kwargs={}*, *eta=None*, *expires=None*, ***options*)

celery.worker.mediator

celery.worker.mediator

The mediator is an internal thread that moves tasks from an internal Queue to the worker pool.

This is only used if rate limits are enabled, as it moves messages from the rate limited queue (which holds tasks that are allowed to be processed) to the pool. Disabling rate limits will also disable this machinery, and can improve performance.

class celery.worker.mediator.**Mediator**(*ready_queue*, *callback*, *app=None*, ***kw*)
Mediator thread.

body()

callback = None
Callback called when a task is obtained.

ready_queue = None
The task queue, a Queue instance.

class celery.worker.mediator.**WorkerComponent**(*w*, ***kwargs*)

create(*w*)

include_if(*w*)

name = 'mediator'

namespace = 'worker'

requires = ('pool', 'queues')

celery.worker.buckets

celery.worker.buckets

This module implements the rate limiting of tasks, by having a token bucket queue for each task type. When a task is allowed to be processed it's moved over the the `ready_queue`

The `celery.worker.mediator` is then responsible for moving tasks from the `ready_queue` to the worker pool.

class `celery.worker.buckets.`**AsyncTaskBucket**(*task_registry*, *callback=None*, *worker=None*)

add_bucket_for_type(*name*)

clear()

cont(*request*, *bucket*, *tokens*)

get(**args*, ***kwargs*)

put(*request*)

refresh()

class `celery.worker.buckets.`**FastQueue**(*maxsize=0*)
`Queue.Queue` supporting the interface of `TokenBucketQueue`.

clear()

expected_time(*tokens=1*)

items

wait(*block=True*)

exception `celery.worker.buckets.`**RateLimitExceeded**
The token buckets rate limit has been exceeded.

class `celery.worker.buckets.`**TaskBucket**(*task_registry*, *callback=None*, *worker=None*)
This is a collection of token buckets, each task type having its own token bucket. If the task type doesn't have a rate limit, it will have a plain `Queue` object instead of a `TokenBucketQueue`.

The `put()` operation forwards the task to its appropriate bucket, while the `get()` operation iterates over the buckets and retrieves the first available item.

Say we have three types of tasks in the registry: *twitter.update*, *feed.refresh* and *video.compress*, the TaskBucket will consist of the following items:

```
{'twitter.update': TokenBucketQueue(fill_rate=300),
 'feed.refresh': Queue(),
 'video.compress': TokenBucketQueue(fill_rate=2)}
```

The get operation will iterate over these until one of the buckets is able to return an item. The underlying datastructure is a *dict*, so the order is ignored here.

Parameters **task_registry** – The task registry used to get the task type class for a given task name.

add_bucket_for_type(*task_name*)
Add a bucket for a task type.

Will read the tasks rate limit and create a `TokenBucketQueue` if it has one. If the task doesn't have a rate limit `FastQueue` will be used instead.

clear()
Delete the data in all of the buckets.

empty()
Returns `True` if all of the buckets are empty.

get(*block=True*, *timeout=None*)
Retrive the task from the first available bucket.

Available as in, there is an item in the queue and you can consume tokens from it.

get_bucket_for_type(*task_name*)
Get the bucket for a particular task type.

get_nowait()

init_with_registry()
Initialize with buckets for all the task types in the registry.

items
Flattens the data in all of the buckets into a single list.

put(*request*)
Put a `Request` into the appropiate bucket.

put_nowait(*request*)
Put a `Request` into the appropiate bucket.

qsize()
Get the total size of all the queues.

refresh()
Refresh rate limits for all task types in the registry.

update_bucket_for_type(*task_name*)

class celery.worker.buckets.**TokenBucketQueue**(*fill_rate*, *queue=None*, *capacity=1*)
Queue with rate limited get operations.

This uses the token bucket algorithm to rate limit the queue on get operations.

Parameters

- **fill_rate** – The rate in tokens/second that the bucket will be refilled.
- **capacity** – Maximum number of tokens in the bucket. Default is 1.

exception RateLimitExceeded
The token buckets rate limit has been exceeded.

TokenBucketQueue.**clear**()
Delete all data in the queue.

TokenBucketQueue.**empty**()
Returns `True` if the queue is empty.

TokenBucketQueue.**expected_time**(*tokens=1*)
Returns the expected time in seconds of when a new token should be available.

TokenBucketQueue.**get**(*block=True*)
Remove and return an item from the queue.

Raises

- **RateLimitExceeded** – If a token could not be consumed from the token bucket (consuming from the queue too fast).
- **Queue.Empty** – If an item is not immediately available.

`TokenBucketQueue.get_nowait()`
Remove and return an item from the queue without blocking.

Raises

- **RateLimitExceeded** – If a token could not be consumed from the token bucket (consuming from the queue too fast).
- **Queue.Empty** – If an item is not immediately available.

`TokenBucketQueue.items`
Underlying data. Do not modify.

`TokenBucketQueue.put(item, block=True)`
Put an item onto the queue.

`TokenBucketQueue.put_nowait(item)`
Put an item into the queue without blocking.

Raises Queue.Full If a free slot is not immediately available.

`TokenBucketQueue.qsize()`
Returns the size of the queue.

`TokenBucketQueue.wait(block=False)`
Wait until a token can be retrieved from the bucket and return the next item.

`celery.worker.buckets.chain_from_iterable()`
chain.from_iterable(iterable) –> chain object

Alternate chain() contructor taking a single iterable argument that evaluates lazily.

celery.worker.heartbeat

celery.worker.heartbeat

This is the internal thread that sends heartbeat events at regular intervals.

class `celery.worker.heartbeat.Heart(timer, eventer, interval=None)`
Timer sending heartbeats at regular intervals.

Parameters

- **timer** – Timer instance.
- **eventer** – Event dispatcher used to send the event.
- **interval** – Time in seconds between heartbeats. Default is 30 seconds.

`start()`

`stop()`

celery.worker.hub

celery.worker.hub

Event-loop implementation.

class `celery.worker.hub.`**BoundedSemaphore**(*value*)
Asynchronous Bounded Semaphore.

Bounded means that the value will stay within the specified range even if it is released more times than it was acquired.

This type is *not thread safe*.

Example:

```
>>> x = BoundedSemaphore(2)

>>> def callback(i):
...     print('HELLO %r' % i)

>>> x.acquire(callback, 1)
HELLO 1

>>> x.acquire(callback, 2)
HELLO 2

>>> x.acquire(callback, 3)
>>> x._waiters   # private, do not access directly
[(callback, 3)]

>>> x.release()
HELLO 3
```

acquire(*callback*, **partial_args*)
Acquire semaphore, applying `callback` when the semaphore is ready.

Parameters

- **callback** – The callback to apply.
- ***partial_args** – partial arguments to callback.

clear()
Reset the sempahore, including wiping out any waiting callbacks.

grow(*n=1*)
Change the size of the semaphore to hold more values.

release()
Release semaphore.

This will apply any waiting callbacks from previous calls to `acquire()` done when the semaphore was busy.

shrink(*n=1*)
Change the size of the semaphore to hold less values.

class `celery.worker.hub.`**DummyLock**
Pretending to be a lock.

class `celery.worker.hub.`**Hub**(*timer=None*)
Event loop object.

Parameters **timer** – Specify custom `Schedule`.

ERR = 24
Flag set on error, and the fd should be read from asap.

READ = 1
Flag set if reading from an fd will not block.

WRITE = 4
Flag set if writing to an fd will not block.

add(*fd*, *callback*, *flags*)

add_reader(*fd*, *callback*)

add_writer(*fd*, *callback*)

close(**args*)

fire_timers(*min_delay=1*, *max_delay=10*, *max_timers=10*, *propagate=()*)

init()

on_close = None
List of callbacks to be called when the loop is exiting, applied with the hub instance as sole argument.

on_init = None
List of callbacks to be called when the loop is initialized, applied with the hub instance as sole argument.

on_task = None
List of callbacks to be called when a task is received. Takes no arguments.

remove(*fd*)

scheduler

start()
Called by StartStopComponent at worker startup.

stop()
Called by StartStopComponent at worker shutdown.

update_readers(*readers*)

update_writers(*writers*)

celery.worker.control

celery.worker.control

Remote control commands.

class `celery.worker.control.`**Panel**(*dict=None*, ***kwargs*)

data = {'time_limit': <function time_limit at 0x7fb764db6848>, 'revoke': <function revoke at 0x7fb7650ba488>, 'dump_

classmethod register(*method*, *name=None*)

celery.worker.control.**active_queues**(*panel*)
Returns the queues associated with each worker.

celery.worker.control.**add_consumer**(*panel*, *queue*, *exchange=None*, *exchange_type=None*, *routing_key=None*, ***options*)

celery.worker.control.**autoscale**(*panel*, *max=None*, *min=None*)

celery.worker.control.**cancel_consumer**(*panel*, *queue=None*, ***_*)

celery.worker.control.**disable_events**(*panel*)

celery.worker.control.**dump_active**(*panel*, *safe=False*, ***kwargs*)

celery.worker.control.**dump_conf**(*panel*, ***kwargs*)

celery.worker.control.**dump_reserved**(*panel*, *safe=False*, ***kwargs*)

celery.worker.control.**dump_revoked**(*panel*, ***kwargs*)

celery.worker.control.**dump_schedule**(*panel*, *safe=False*, ***kwargs*)

celery.worker.control.**dump_tasks**(*panel*, *taskinfoitems=None*, ***kwargs*)

celery.worker.control.**enable_events**(*panel*)

celery.worker.control.**heartbeat**(*panel*)

celery.worker.control.**ping**(*panel*, ***kwargs*)

celery.worker.control.**pool_grow**(*panel*, *n=1*, ***kwargs*)

celery.worker.control.**pool_restart**(*panel*, *modules=None*, *reload=False*, *reloader=None*, ***kwargs*)

celery.worker.control.**pool_shrink**(*panel*, *n=1*, ***kwargs*)

celery.worker.control.**rate_limit**(*panel*, *task_name*, *rate_limit*, ***kwargs*)
Set new rate limit for a task type.

See `celery.task.base.Task.rate_limit`.

Parameters

- **task_name** – Type of task.
- **rate_limit** – New rate limit.

celery.worker.control.**report**(*panel*)

celery.worker.control.**revoke**(*panel*, *task_id*, *terminate=False*, *signal=None*, ***kwargs*)
Revoke task by task id.

celery.worker.control.**shutdown**(*panel*, *msg='Got shutdown from remote'*, ***kwargs*)

celery.worker.control.**stats**(*panel*, ***kwargs*)

celery.worker.control.**time_limit**(*panel*, *task_name=None*, *hard=None*, *soft=None*, ***kwargs*)

celery.worker.state

celery.worker.state

Internal worker state (global)

This includes the currently active and reserved tasks, statistics, and revoked tasks.

class celery.worker.state.**Persistent**(*filename*)
This is the persistent data stored by the worker when *--statedb* is enabled.

It currently only stores revoked task id's.

close()

db

merge(*d*)

open()

protocol = 2

save()

storage = <module 'shelve' from '/usr/lib/python2.7/shelve.pyc'>

sync(*d*)

celery.worker.state.**REVOKES_MAX = 10000**
maximum number of revokes to keep in memory.

celery.worker.state.**REVOKE_EXPIRES = 3600**
how many seconds a revoke will be active before being expired when the max limit has been exceeded.

celery.worker.state.**SOFTWARE_INFO = {'sw_sys': 'Linux', 'sw_ident': 'py-celery', 'sw_ver': '3.0.25'}**
Worker software/platform information.

celery.worker.state.**active_requests = set([])**
set of currently active Request's.

celery.worker.state.**reserved_requests = set([])**
set of all reserved Request's.

celery.worker.state.**revoked = LimitedSet([])**
the list of currently revoked tasks. Persistent if statedb set.

celery.worker.state.**task_accepted**(*request*)
Updates global state when a task has been accepted.

celery.worker.state.**task_ready**(*request*)
Updates global state when a task is ready.

celery.worker.state.**task_reserved**()
Updates global state when a task has been reserved.

celery.worker.state.**total_count = defaultdict(<type 'int'>, {})**
count of tasks executed by the worker, sorted by type.

celery.worker.strategy

celery.worker.strategy

Task execution strategy (optimization).

celery.worker.strategy.**default**(*task*, *app*, *consumer*)

celery.worker.autoreload

celery.worker.autoreload

This module implements automatic module reloading

class celery.worker.autoreload.**Autoreloader**(*controller*, *modules=None*, *monitor_cls=None*, ***options*)

Tracks changes in modules and fires reload commands

Monitor

alias of StatMonitor

body()

on_change(*files*)

on_init()

on_poll_close(*hub*)

on_poll_init(*hub*)

stop()

class celery.worker.autoreload.**BaseMonitor**(*files*, *on_change=None*, *shutdown_event=None*, *interval=0.5*)

on_change(*modified*)

start()

stop()

class celery.worker.autoreload.**InotifyMonitor**(*modules*, *on_change=None*, ***kwargs*)

File change monitor based on Linux kernel *inotify* subsystem

on_change(*modified*)

process_(*event*)

process_IN_ATTRIB(*event*)

process_IN_MODIFY(*event*)

start()

stop()

class celery.worker.autoreload.**KQueueMonitor**(**args*, ***kwargs*)

File change monitor based on BSD kernel event notifications

add_events(*poller*)

close(*poller*)

handle_event(*events*)

on_poll_close(*hub*)

on_poll_init(*hub*)

start()

stop()

celery.worker.autoreload.**Monitor**
alias of StatMonitor

class celery.worker.autoreload.**StatMonitor**(*files*, *on_change=None*, *shutdown_event=None*, *interval=0.5*)
File change monitor based on the stat system call.

start()

class celery.worker.autoreload.**WorkerComponent**(*w*, *autoreload=None*, ***kwargs*)

create(*w*)

create_ev(*w*)

create_threaded(*w*)

name = 'autoreloader'

namespace = 'worker'

requires = ('pool',)

celery.worker.autoreload.**default_implementation**()

celery.worker.autoreload.**file_hash**(*filename*, *algorithm='md5'*)

celery.worker.autoscale

celery.worker.autoscale

This module implements the internal thread responsible for growing and shrinking the pool according to the current autoscale settings.

The autoscale thread is only enabled if autoscale has been enabled on the command line.

class celery.worker.autoscale.**Autoscaler**(*pool*, *max_concurrency*, *min_concurrency=0*, *keepalive=30*, *mutex=None*)

body()

force_scale_down(*n*)

force_scale_up(*n*)

info()

maybe_scale()

processes

qty

scale_down(*n*)

scale_up(*n*)

update(*max=None*, *min=None*)

class celery.worker.autoscale.**WorkerComponent**(*w*, ***kwargs*)

create(*w*)

create_ev(*w*)

create_threaded(*w*)

name = 'autoscaler'

namespace = 'worker'

on_poll_init(*scaler*, *hub*)

requires = ('pool',)

celery.worker.bootsteps

celery.worker.bootsteps

The boot-step components.

class celery.worker.bootsteps.**Component**(*parent*, ***kwargs*)

A component.

The `__init__()` method is called when the component is bound to a parent object, and can as such be used to initialize attributes in the parent object at parent instantiation-time.

abstract = None

if set the component will not be registered, but can be used as a component base class.

create(*parent*)

Create the component.

enabled = True

This provides the default for `include_if()`.

include(*parent*)

include_if(*parent*)

An optional predicate that decided whether this component should be created.

instantiate(*qualname*, **args*, ***kwargs*)

last = False
This flag is reserved for the workers Consumer, since it is required to always be started last. There can only be one object marked with lsat in every namespace.

name = None
The name of the component, or the namespace and the name of the component separated by dot.

namespace = None
can be used to specify the namespace, if the name does not include it.

obj = None
Optional obj created by the `create()` method. This is used by StartStopComponents to keep the original service object.

requires = ()
List of component names this component depends on. Note that the dependencies must be in the same namespace.

class `celery.worker.bootsteps.`**ComponentType**
Metaclass for components.

class `celery.worker.bootsteps.`**Namespace**(*name=None*, *app=None*)
A namespace containing components.

Every component must belong to a namespace.

When component classes are created they are added to the mapping of unclaimed components. The components will be claimed when the namespace they belong to is created.

Parameters
- **name** – Set the name of this namespace.
- **app** – Set the Celery app for this namespace.

apply(*parent*, ***kwargs*)
Apply the components in this namespace to an object.

This will apply the `__init__` and `include` methods of each components with the object as argument.

For `StartStopComponents` the services created will also be added the the objects `components` attribute.

bind_component(*name*, *parent*, ***kwargs*)
Bind component to parent object and this namespace.

import_module(*module*)

load_modules()
Will load the component modules this namespace depends on.

modules()
Subclasses can override this to return a list of modules to import before components are claimed.

name = None

class `celery.worker.bootsteps.`**StartStopComponent**(*parent*, ***kwargs*)

include(*parent*)

start()

stop()

terminable = False

terminate()

celery.concurrency

celery.concurrency

Pool implementation abstract factory, and alias definitions.

celery.concurrency.**get_implementation**(*cls*)

celery.concurrency.solo

celery.concurrency.solo

Single-threaded pool implementation.

class celery.concurrency.solo.**TaskPool**(**args*, ***kwargs*)

Solo task pool (blocking, inline, fast).

celery.concurrency.processes

celery.concurrency.processes

Pool implementation using multiprocessing.

We use the billiard fork of multiprocessing which contains numerous improvements.

class celery.concurrency.processes.**TaskPool**(*limit=None*, *putlocks=True*, *forking_enable=True*, *callbacks_propagate=()*, ***options*)

Multiprocessing Pool implementation.

class **Pool**(*processes=None*, *initializer=None*, *initargs=()*, *maxtasksperchild=None*, *timeout=None*, *soft_timeout=None*, *lost_worker_timeout=10.0*, *max_restarts=None*, *max_restart_freq=1*, *on_process_up=None*, *on_process_down=None*, *on_timeout_set=None*, *on_timeout_cancel=None*, *threads=True*, *semaphore=None*, *putlocks=False*, *allow_restart=False*)

Class which supports an async version of applying functions to arguments.

class **Process**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*, *daemon=None*, ***_kw*)

Process objects represent activity that is run in a separate process

The class is analagous to *threading.Thread*

authkey

daemon

exitcode
Return exit code of process or *None* if it has yet to stop

ident
Return identifier (PID) of process or *None* if it has yet to start

is_alive()
Return whether process is alive

join(*timeout=None*)
Wait until child process terminates

name

pid
Return identifier (PID) of process or *None* if it has yet to start

run()
Method to be run in sub-process; can be overridden in sub-class

sentinel
Return a file descriptor (Unix) or handle (Windows) suitable for waiting for process termination.

start()
Start child process

terminate()
Terminate process; sends SIGTERM signal or uses TerminateProcess()

class TaskPool.Pool.**ResultHandler**(*outqueue*, *get*, *cache*, *poll*, *join_exited_workers*, *putlock*, *restart_state*, *check_timeouts*)

body()

finish_at_shutdown(*handle_timeouts=False*)

handle_event(**args*)

on_stop_not_started()

exception TaskPool.Pool.**SoftTimeLimitExceeded**
The soft time limit has been exceeded. This exception is raised to give the task a chance to clean up.

class TaskPool.Pool.**Supervisor**(*pool*)

body()

class TaskPool.Pool.**TaskHandler**(*taskqueue*, *put*, *outqueue*, *pool*)

body()

on_stop_not_started()

tell_others()

class TaskPool.Pool.**TimeoutHandler**(*processes*, *cache*, *t_soft*, *t_hard*)

body()

handle_event(**args*)

handle_timeouts()

on_hard_timeout(*job*)

on_soft_timeout(*job*)

TaskPool.Pool.**apply**(*func*, *args=()*, *kwds={}*)
Equivalent of *func(*args, **kwargs)*.

TaskPool.Pool.**apply_async**(*func*, *args=()*, *kwds={}*, *callback=None*, *error_callback=None*, *accept_callback=None*, *timeout_callback=None*, *waitforslot=None*, *soft_timeout=None*, *timeout=None*, *lost_worker_timeout=None*, *callbacks_propagate=()*)
Asynchronous equivalent of *apply()* method.

Callback is called when the functions return value is ready. The accept callback is called when the job is accepted to be executed.

Simplified the flow is like this:

```
>>> if accept_callback:
...     accept_callback()
>>> retval = func(*args, **kwds)
>>> if callback:
...     callback(retval)
```

TaskPool.Pool.**close**()

TaskPool.Pool.**did_start_ok**()

TaskPool.Pool.**grow**(*n=1*)

TaskPool.Pool.**imap**(*func*, *iterable*, *chunksize=1*, *lost_worker_timeout=None*)
Equivalent of *map()* – can be MUCH slower than *Pool.map()*.

TaskPool.Pool.**imap_unordered**(*func*, *iterable*, *chunksize=1*, *lost_worker_timeout=None*)
Like *imap()* method but ordering of results is arbitrary.

TaskPool.Pool.**join**()

TaskPool.Pool.**maintain_pool**(**args*, ***kwargs*)

TaskPool.Pool.**map**(*func*, *iterable*, *chunksize=None*)
Apply *func* to each element in *iterable*, collecting the results in a list that is returned.

TaskPool.Pool.**map_async**(*func*, *iterable*, *chunksize=None*, *callback=None*, *error_callback=None*)
Asynchronous equivalent of *map()* method.

TaskPool.Pool.**restart**()

TaskPool.Pool.**shrink**(*n=1*)

TaskPool.Pool.**starmap**(*func*, *iterable*, *chunksize=None*)
Like *map()* method but the elements of the *iterable* are expected to be iterables as well and will be unpacked as arguments. Hence *func* and (a, b) becomes func(a, b).

TaskPool.Pool.**starmap_async**(*func*, *iterable*, *chunksize=None*, *callback=None*, *error_callback=None*)
Asynchronous version of *starmap()* method.

TaskPool.Pool.**terminate**()

TaskPool.Pool.**terminate_job**(*pid*, *sig=None*)

TaskPool.**did_start_ok**()

TaskPool.**grow**(*n=1*)

TaskPool.**handle_timeouts**()

TaskPool.**init_callbacks**(***kwargs*)

TaskPool.**num_processes**

TaskPool.**on_close**()

TaskPool.**on_start**()
Run the task pool.

Will pre-fork all workers so they're ready to accept tasks.

TaskPool.**on_stop**()
Gracefully stop the pool.

TaskPool.**on_terminate**()
Force terminate the pool.

TaskPool.**readers**

TaskPool.**requires_mediator** = True

TaskPool.**restart**()

TaskPool.**shrink**(*n=1*)

TaskPool.**terminate_job**(*pid*, *signal=None*)

TaskPool.**timers**

TaskPool.**uses_semaphore** = True

TaskPool.**writers**

celery.concurrency.processes.**WORKER_SIGIGNORE** = frozenset(['SIGINT'])
List of signals to ignore when a child process starts.

celery.concurrency.processes.**WORKER_SIGRESET** = frozenset(['SIGHUP', 'SIGTERM', 'SIGTTOU', 'SIGTTIN', '
List of signals to reset when a child process starts.

celery.concurrency.processes.**process_initializer**(*app*, *hostname*)
Initializes the process so it can be used to process tasks.

celery.concurrency.eventlet† (*experimental*)

- celery.concurrency.eventlet

celery.concurrency.eventlet

Eventlet pool implementation.

class celery.concurrency.eventlet.**Schedule**(**args*, ***kwargs*)

clear()

queue

class celery.concurrency.eventlet.**TaskPool**(**args*, ***kwargs*)

class Timer(*schedule=None*, *on_error=None*, *on_tick=None*, *max_interval=None*, ***kwargs*)

class Schedule(**args*, ***kwargs*)

clear()

queue

TaskPool.Timer.**cancel**(*tref*)

TaskPool.Timer.**ensure_started**()

TaskPool.Timer.**start**()

TaskPool.Timer.**stop**()

TaskPool.**is_green** = True

TaskPool.**on_apply**(*target*, *args=None*, *kwargs=None*, *callback=None*, *accept_callback=None*, ***_*)

TaskPool.**on_start**()

TaskPool.**on_stop**()

TaskPool.**rlimit_safe** = False

TaskPool.**signal_safe** = False

class celery.concurrency.eventlet.**Timer**(*schedule=None*, *on_error=None*, *on_tick=None*, *max_interval=None*, ***kwargs*)

class Schedule(**args*, ***kwargs*)

clear()

queue

Timer.**cancel**(*tref*)

Timer.**ensure_started**()

Timer.**start**()

Timer.**stop**()

celery.concurrency.eventlet.**apply_target**(*target*, *args=()*, *kwargs={}*, *callback=None*, *accept_callback=None*, *getpid=None*)

celery.concurrency.gevent† (*experimental*)

celery.concurrency.gevent

gevent pool implementation.

class celery.concurrency.gevent.**Schedule**(**args*, ***kwargs*)

clear()

queue

class celery.concurrency.gevent.**TaskPool**(**args*, ***kwargs*)

class Timer(*schedule=None*, *on_error=None*, *on_tick=None*, *max_interval=None*, ***kwargs*)

class Schedule(**args*, ***kwargs*)

clear()

queue

TaskPool.Timer.**ensure_started**()

TaskPool.Timer.**start**()

TaskPool.Timer.**stop**()

TaskPool.**grow**(*n=1*)

TaskPool.**is_green** = True

TaskPool.**num_processes**

TaskPool.**on_apply**(*target*, *args=None*, *kwargs=None*, *callback=None*, *accept_callback=None*, *timeout=None*, *timeout_callback=None*, ***_*)

TaskPool.**on_start**()

TaskPool.**on_stop**()

TaskPool.**rlimit_safe** = False

TaskPool.**shrink**(*n=1*)

TaskPool.**signal_safe** = False

class celery.concurrency.gevent.**Timer**(*schedule=None*, *on_error=None*, *on_tick=None*, *max_interval=None*, ***kwargs*)

class Schedule(**args*, ***kwargs*)

clear()

queue

Timer.**ensure_started**()

Timer.**start**()

Timer.**stop**()

celery.concurrency.gevent.**apply_timeout**(*target*, *args=()*, *kwargs={}*, *callback=None*, *accept_callback=None*, *pid=None*, *timeout=None*, *timeout_callback=None*, ***rest*)

celery.concurrency.base

celery.concurrency.base

TaskPool interface.

class celery.concurrency.base.**BasePool**(*limit=None*, *putlocks=True*, *forking_enable=True*, *callbacks_propagate=()*, ***options*)

CLOSE = 2

RUN = 1

TERMINATE = 3

class Timer(*schedule=None*, *on_error=None*, *on_tick=None*, *max_interval=None*, ***kwargs*)

class Entry(*fun*, *args=None*, *kwargs=None*)

args

cancel()

cancelled

fun

kwargs

tref

class BasePool.Timer.**Schedule**(*max_interval=None*, *on_error=None*, ***kwargs*)

ETA scheduler.

class Entry(*fun*, *args=None*, *kwargs=None*)

args

cancel()

cancelled

fun

kwargs

tref

BasePool.Timer.Schedule.**apply_after**(*msecs*, *fun*, *args=()*, *kwargs={}*, *priority=0*)

BasePool.Timer.Schedule.**apply_at**(*eta*, *fun*, *args=()*, *kwargs={}*, *priority=0*)

BasePool.Timer.Schedule.**apply_entry**(*entry*)

BasePool.Timer.Schedule.**apply_interval**(*msecs*, *fun*, *args=()*, *kwargs={}*, *priority=0*)

BasePool.Timer.Schedule.**cancel**(*tref*)

BasePool.Timer.Schedule.**clear**()

BasePool.Timer.Schedule.**empty**()
Is the schedule empty?

BasePool.Timer.Schedule.**enter**(*entry*, *eta=None*, *priority=0*)
Enter function into the scheduler.

Parameters

- **entry** – Item to enter.
- **eta** – Scheduled time as a `datetime.datetime` object.
- **priority** – Unused.

BasePool.Timer.Schedule.**enter_after**(*msecs*, *entry*, *priority=0*, *time=<built-in function time>*)

BasePool.Timer.Schedule.**handle_error**(*exc_info*)

BasePool.Timer.Schedule.**info**()

BasePool.Timer.Schedule.**on_error = None**

BasePool.Timer.Schedule.**queue**
Snapshot of underlying datastructure.

BasePool.Timer.Schedule.**schedule**

BasePool.Timer.Schedule.**stop**()

BasePool.Timer.**apply_after**(**args*, ***kwargs*)

BasePool.Timer.**apply_at**(**args*, ***kwargs*)

BasePool.Timer.**apply_interval**(**args*, ***kwargs*)

BasePool.Timer.**cancel**(*tref*)

BasePool.Timer.**clear**()

BasePool.Timer.**empty**()

BasePool.Timer.**ensure_started**()

BasePool.Timer.**enter**(*entry*, *eta*, *priority=None*)

BasePool.Timer.**enter_after**(**args*, ***kwargs*)

BasePool.Timer.**exit_after**(*msecs*, *priority=10*)

BasePool.Timer.**next**()

BasePool.Timer.**on_tick = None**

BasePool.Timer.**queue**

BasePool.Timer.**run**()

BasePool.Timer.**running** = False

BasePool.Timer.**stop**()

BasePool.**active**

BasePool.**apply_async**(*target*, *args*=[], *kwargs={}*, ***options*)
Equivalent of the `apply()` built-in function.

Callbacks should optimally return as soon as possible since otherwise the thread which handles the result will get blocked.

BasePool.**close**()

BasePool.**did_start_ok**()

BasePool.**info**

BasePool.**init_callbacks**(***kwargs*)

BasePool.**is_green** = False
set to true if pool uses greenlets.

BasePool.**maintain_pool**(**args*, ***kwargs*)

BasePool.**maybe_handle_result**(**args*)

BasePool.**num_processes**

BasePool.**on_apply**(**args*, ***kwargs*)

BasePool.**on_close**()

BasePool.**on_hard_timeout**(*job*)

BasePool.**on_soft_timeout**(*job*)

BasePool.**on_start**()

BasePool.**on_stop**()

BasePool.**on_terminate**()

BasePool.**readers**

BasePool.**requires_mediator** = False
set to true if pool requires the use of a mediator thread (e.g. if applying new items can block the current thread).

BasePool.**restart**()

BasePool.**rlimit_safe** = True
set to true if pool supports rate limits. (this is here for gevent, which currently does not implement the necessary timers).

BasePool.**signal_safe** = True
set to true if the pool can be shutdown from within a signal handler.

BasePool.**start**()

BasePool.**stop**()

BasePool.**terminate**()

BasePool.**terminate_job**(*pid*)

BasePool.**timers**

BasePool.**uses_semaphore** = False
only used by multiprocessing pool

BasePool.**writers**

celery.concurrency.base.**apply_target**(*target*, *args=()*, *kwargs={}*, *callback=None*, *accept_callback=None*, *pid=None*, ***_*)

celery.concurrency.threads‡ (minefield)

celery.concurrency.threads

Pool implementation using threads.

class celery.concurrency.threads.**NullDict**(*dict=None*, ***kwargs*)

class celery.concurrency.threads.**TaskPool**(**args*, ***kwargs*)

on_apply(*target*, *args=None*, *kwargs=None*, *callback=None*, *accept_callback=None*, ***_*)

on_start()

on_stop()

celery.beat

celery.beat

The periodic task scheduler.

celery.beat.**EmbeddedService**(**args*, ***kwargs*)
Return embedded clock service.
Parameters **thread** – Run threaded instead of as a separate process. Default is False.

class celery.beat.**PersistentScheduler**(**args*, ***kwargs*)

close()

get_schedule()

info

known_suffixes = ('', '.db', '.dat', '.bak', '.dir')

persistence = <module 'shelve' from '/usr/lib/python2.7/shelve.pyc'>

schedule

set_schedule(*schedule*)

setup_schedule()

sync()

class celery.beat.**ScheduleEntry**(*name=None*, *task=None*, *last_run_at=None*, *total_run_count=None*, *schedule=None*, *args=()*, *kwargs={}*, *options={}*, *relative=False*)

An entry in the scheduler.

Parameters

- **name** – see name.
- **schedule** – see schedule.
- **args** – see args.
- **kwargs** – see kwargs.
- **options** – see options.
- **last_run_at** – see last_run_at.
- **total_run_count** – see total_run_count.
- **relative** – Is the time relative to when the server starts?

args = None

Positional arguments to apply.

is_due()

See is_due().

kwargs = None

Keyword arguments to apply.

last_run_at = None

The time and date of when this task was last scheduled.

name = None

The task name

next(*last_run_at=None*)

Returns a new instance of the same class, but with its date and count fields updated.

options = None

Task execution options.

schedule = None

The schedule (run_every/crontab)

total_run_count = 0

Total number of times this task has been scheduled.

update(*other*)

Update values from another entry.

Does only update "editable" fields (task, schedule, args, kwargs, options).

class celery.beat.**Scheduler**(*schedule=None*, *max_interval=None*, *app=None*, *Publisher=None*, *lazy=False*, ***kwargs*)

Scheduler for periodic tasks.

Parameters

- **schedule** – see schedule.
- **max_interval** – see max_interval.

Entry

alias of ScheduleEntry

add(***kwargs*)

apply_async(*entry*, *publisher=None*, ***kwargs*)

close()

connection

get_schedule()

info

install_default_entries(*data*)

logger = <celery.utils.log.SigSafeLogger object at 0x7fb7655e1150>

max_interval = 300
Maximum time to sleep between re-checking the schedule.

maybe_due(*entry*, *publisher=None*)

merge_inplace(*b*)

publisher

reserve(*entry*)

schedule
The schedule dict/shelve.

send_task(**args*, ***kwargs*)

set_schedule(*schedule*)

setup_schedule()

should_sync()

sync()

sync_every = 180
How often to sync the schedule (3 minutes by default)

tick()
Run a tick, that is one iteration of the scheduler.

Executes all due tasks.

update_from_dict(*dict_*)

exception celery.beat.**SchedulingError**
An error occured while scheduling a task.

class celery.beat.**Service**(*max_interval=None*, *schedule_filename=None*, *scheduler_cls=None*, *app=None*)

get_scheduler(*lazy=False*)

scheduler

scheduler_cls
alias of PersistentScheduler

start(*embedded_process=False*)

stop(*wait=False*)

sync()

celery.backends

celery.backends

Backend abstract factory (...did I just say that?) and alias definitions.

`celery.backends.`**`default_backend = <celery.backends.cache.CacheBackend object at 0x7fb767118c10>`**
deprecated alias to `current_app.backend`.

`celery.backends.`**`get_backend_by_url`**(*backend=None*, *loader=None*)

`celery.backends.`**`get_backend_cls`**(**args*, ***kwargs*)
Get backend class by name/alias

celery.backends.base

celery.backends.base

Result backend base classes.

- `BaseBackend` defines the interface.
- `BaseDictBackend` assumes the fields are stored in a dict.
- `KeyValueStoreBackend` is a common base class using K/V semantics like _get and _put.

class `celery.backends.base.`**`BaseBackend`**(**args*, ***kwargs*)
Base backend class.

`EXCEPTION_STATES` **= frozenset([’FAILURE’, ‘RETRY’, ‘REVOKED’])**

`READY_STATES` **= frozenset([’FAILURE’, ‘REVOKED’, ‘SUCCESS’])**

exception `TimeoutError`
The operation timed out.

`BaseBackend.`**`UNREADY_STATES`** **= frozenset([’STARTED’, ‘RECEIVED’, ‘RETRY’, ‘PENDING’])**

`BaseBackend.`**`cleanup`**()
Backend cleanup. Is run by `celery.task.DeleteExpiredTaskMetaTask`.

`BaseBackend.`**`current_task_children`**()

`BaseBackend.`**`decode`**(*payload*)

`BaseBackend.`**`delete_group`**(*group_id*)

`BaseBackend.`**`encode`**(*data*)

`BaseBackend.`**`encode_result`**(*result*, *status*)

BaseBackend.**exception_to_python**(*exc*)
Convert serialized exception to Python exception.

BaseBackend.**fail_from_current_stack**(*task_id*, *exc=None*)

BaseBackend.**fallback_chord_unlock**(*group_id*, *body*, *result=None*, *countdown=1*, ***kwargs*)

BaseBackend.**forget**(*task_id*)

BaseBackend.**get_children**(*task_id*)

BaseBackend.**get_result**(*task_id*)
Get the result of a task.

BaseBackend.**get_status**(*task_id*)
Get the status of a task.

BaseBackend.**get_traceback**(*task_id*)
Get the traceback for a failed task.

BaseBackend.**is_cached**(*task_id*)

BaseBackend.**mark_as_done**(*task_id*, *result*)
Mark task as successfully executed.

BaseBackend.**mark_as_failure**(*task_id*, *exc*, *traceback=None*)
Mark task as executed with failure. Stores the execption.

BaseBackend.**mark_as_retry**(*task_id*, *exc*, *traceback=None*)
Mark task as being retries. Stores the current exception (if any).

BaseBackend.**mark_as_revoked**(*task_id*, *reason=''*)

BaseBackend.**mark_as_started**(*task_id*, ***meta*)
Mark a task as started

BaseBackend.**on_chord_apply**(*group_id*, *body*, *result=None*, *countdown=1*, ***kwargs*)

BaseBackend.**on_chord_part_return**(*task*, *propagate=True*)

BaseBackend.**prepare_exception**(*exc*)
Prepare exception for serialization.

BaseBackend.**prepare_expires**(*value*, *type=None*)

BaseBackend.**prepare_value**(*result*)
Prepare value for storage.

BaseBackend.**process_cleanup**()
Cleanup actions to do at the end of a task worker process.

BaseBackend.**reload_group_result**(*task_id*)
Reload group result, even if it has been previously fetched.

BaseBackend.**reload_task_result**(*task_id*)
Reload task result, even if it has been previously fetched.

BaseBackend.**restore_group**(*group_id*, *cache=True*)
Get the result of a group.

BaseBackend.**save_group**(*group_id*, *result*)
Store the result and status of a task.

BaseBackend.**store_result**(*task_id*, *result*, *status*, *traceback=None*)
Store the result and status of a task.

BaseBackend.**subpolling_interval** = None
Time to sleep between polling each individual item in *ResultSet.iterate*. as opposed to the *interval* argument which is for each pass.

BaseBackend.**supports_autoexpire** = False
If true the backend must automatically expire results. The daily backend_cleanup periodic task will not be triggered in this case.

BaseBackend.**supports_native_join** = False
If true the backend must implement `get_many()`.

BaseBackend.**wait_for**(*task_id*, *timeout=None*, *propagate=True*, *interval=0.5*)
Wait for task and return its result.

If the task raises an exception, this exception will be re-raised by `wait_for()`.

If *timeout* is not `None`, this raises the `celery.exceptions.TimeoutError` exception if the operation takes longer than *timeout* seconds.

class celery.backends.base.**BaseDictBackend**(**args*, ***kwargs*)

delete_group(*group_id*)

forget(*task_id*)

get_children(*task_id*)
Get the list of subtasks sent by a task.

get_group_meta(*group_id*, *cache=True*)

get_result(*task_id*)
Get the result of a task.

get_status(*task_id*)
Get the status of a task.

get_task_meta(*task_id*, *cache=True*)

get_traceback(*task_id*)
Get the traceback for a failed task.

is_cached(*task_id*)

reload_group_result(*group_id*)

reload_task_result(*task_id*)

restore_group(*group_id*, *cache=True*)
Get the result for a group.

save_group(*group_id*, *result*)
Store the result of an executed group.

store_result(*task_id*, *result*, *status*, *traceback=None*, ***kwargs*)
Store task result and status.

class celery.backends.base.**DisabledBackend**(**args*, ***kwargs*)

get_result(**args*, ***kwargs*)

get_status(**args*, ***kwargs*)

get_traceback(**args*, ***kwargs*)

store_result(**args*, ***kwargs*)

wait_for(**args*, ***kwargs*)

class celery.backends.base.**KeyValueStoreBackend**(**args*, ***kwargs*)

chord_keyprefix = 'chord-unlock-'

delete(*key*)

expire(*key*, *value*)

get(*key*)

get_key_for_chord(*group_id*)
Get the cache key for the chord waiting on group with given id.

get_key_for_group(*group_id*)
Get the cache key for a group by id.

get_key_for_task(*task_id*)
Get the cache key for a task by id.

get_many(*task_ids*, *timeout=None*, *interval=0.5*)

group_keyprefix = 'celery-taskset-meta-'

implements_incr = False

incr(*key*)

mget(*keys*)

on_chord_apply(*group_id*, *body*, *result=None*, ***kwargs*)

on_chord_part_return(*task*, *propagate=None*)

set(*key*, *value*)

task_keyprefix = 'celery-task-meta-'

celery.backends.base.**unpickle_backend**(*cls*, *args*, *kwargs*)
Returns an unpickled backend.

celery.backends.database

celery.backends.database

SQLAlchemy result store backend.

class celery.backends.database.**DatabaseBackend**(*dburi=None*, *expires=None*, *engine_options=None*, ***kwargs*)
The database result backend.

ResultSession()

cleanup()
Delete expired metadata.

subpolling_interval = 0.5

celery.backends.database.**retry**(*fun*)

celery.backends.cache

celery.backends.cache

Memcache and in-memory cache result backend.

class celery.backends.cache.**CacheBackend**(*expires=None*, *backend=None*, *options={}*, ***kwargs*)

client

delete(*key*)

get(*key*)

implements_incr = True

incr(*key*)

mget(*keys*)

on_chord_apply(*group_id*, *body*, *result=None*, ***kwargs*)

servers = None

set(*key*, *value*)

supports_autoexpire = True

supports_native_join = True

class celery.backends.cache.**DummyClient**(**args*, ***kwargs*)

delete(*key*, **args*, ***kwargs*)

get(*key*, **args*, ***kwargs*)

get_multi(*keys*)

incr(*key*, *delta=1*)

set(*key*, *value*, **args*, ***kwargs*)

celery.backends.cache.**get_best_memcache**(**args*, ***kwargs*)

celery.backends.cache.**import_best_memcache**()

celery.backends.amqp

celery.backends.amqp

The AMQP result backend.

This backend publishes results as messages.

class celery.backends.amqp.**AMQPBackend**(*connection=None*, *exchange=None*, *exchange_type=None*, *persistent=None*, *serializer=None*, *auto_delete=True*, ***kwargs*)

Publishes results by sending messages.

exception BacklogLimitExceeded

Too much state history to fast-forward.

class AMQPBackend.**Consumer**(*channel*, *queues=None*, *no_ack=None*, *auto_declare=None*, *callbacks=None*, *on_decode_error=None*, *on_message=None*, *accept=None*)

Message consumer.

Parameters

- **channel** – see channel.
- **queues** – see queues.
- **no_ack** – see no_ack.
- **auto_declare** – see auto_declare
- **callbacks** – see callbacks.
- **on_message** – See on_message
- **on_decode_error** – see on_decode_error.

accept = None

add_queue(*queue*)

Add a queue to the list of queues to consume from.

This will not start consuming from the queue, for that you will have to call consume() after.

add_queue_from_dict(*queue*, ***options*)

This method is deprecated.

Instead please use:

```
consumer.add_queue(Queue.from_dict(d))
```

auto_declare = True

callbacks = None

cancel()

End all active queue consumers.

This does not affect already delivered messages, but it does mean the server will not send any more messages for this consumer.

cancel_by_queue(*queue*)

Cancel consumer by queue name.

channel = None

close()

End all active queue consumers.

This does not affect already delivered messages, but it does mean the server will not send any more messages for this consumer.

connection

consume(*no_ack=None*)
Start consuming messages.

Can be called multiple times, but note that while it will consume from new queues added since the last call, it will not cancel consuming from removed queues (use `cancel_by_queue()`).

Parameters **no_ack** – See `no_ack`.

consuming_from(*queue*)
Returns `True` if the consumer is currently consuming from queue'.

declare()
Declare queues, exchanges and bindings.

This is done automatically at instantiation if `auto_declare` is set.

flow(*active*)
Enable/disable flow from peer.

This is a simple flow-control mechanism that a peer can use to avoid overflowing its queues or otherwise finding itself receiving more messages than it can process.

The peer that receives a request to stop sending content will finish sending the current content (if any), and then wait until flow is reactivated.

no_ack = None

on_decode_error = None

on_message = None

purge()
Purge messages from all queues.

> **Warning:** This will *delete all ready messages*, there is no undo operation.

qos(*prefetch_size=0*, *prefetch_count=0*, *apply_global=False*)
Specify quality of service.

The client can request that messages should be sent in advance so that when the client finishes processing a message, the following message is already held locally, rather than needing to be sent down the channel. Prefetching gives a performance improvement.

The prefetch window is Ignored if the `no_ack` option is set.

Parameters

- **prefetch_size** – Specify the prefetch window in octets. The server will send a message in advance if it is equal to or smaller in size than the available prefetch size (and also falls within other prefetch limits). May be set to zero, meaning "no specific limit", although other prefetch limits may still apply.
- **prefetch_count** – Specify the prefetch window in terms of whole messages.
- **apply_global** – Apply new settings globally on all channels. Currently not supported by RabbitMQ.

queues = None

receive(*body*, *message*)
Method called when a message is received.

This dispatches to the registered `callbacks`.

Parameters

- **body** – The decoded message body.
- **message** – The *Message* instance.

Raises NotImplementedError If no consumer callbacks have been registered.

recover(*requeue=False*)

Redeliver unacknowledged messages.

Asks the broker to redeliver all unacknowledged messages on the specified channel.

Parameters **requeue** – By default the messages will be redelivered to the original recipient. With *requeue* set to true, the server will attempt to requeue the message, potentially then delivering it to an alternative subscriber.

register_callback(*callback*)

Register a new callback to be called when a message is received.

The signature of the callback needs to accept two arguments: *(body, message)*, which is the decoded message body and the *Message* instance (a subclass of `Message`.

revive(*channel*)

Revive consumer after connection loss.

class AMQPBackend.**Exchange**(*name=''*, *type=''*, *channel=None*, ***kwargs*)

An Exchange declaration.

Parameters

- **name** – See `name`.
- **type** – See `type`.
- **channel** – See `channel`.
- **durable** – See `durable`.
- **auto_delete** – See `auto_delete`.
- **delivery_mode** – See `delivery_mode`.
- **arguments** – See `arguments`.

name

Name of the exchange. Default is no name (the default exchange).

type

AMQP defines four default exchange types (routing algorithms) that covers most of the common messaging use cases. An AMQP broker can also define additional exchange types, so see your broker manual for more information about available exchange types.

•*direct* (*default*)

Direct match between the routing key in the message, and the routing criteria used when a queue is bound to this exchange.

•*topic*

Wildcard match between the routing key and the routing pattern specified in the exchange/queue binding. The routing key is treated as zero or more words delimited by *"."* and supports special wildcard characters. *"*"* matches a single word and *"#"* matches zero or more words.

•*fanout*

Queues are bound to this exchange with no arguments. Hence any message sent to this exchange will be forwarded to all queues bound to this exchange.

•*headers*

Queues are bound to this exchange with a table of arguments containing headers and values (optional). A special argument named "x-match" determines the matching algorithm, where *"all"* implies an *AND* (all pairs must match) and *"any"* implies *OR* (at least one pair must match).

`arguments` is used to specify the arguments.

This description of AMQP exchange types was shamelessly stolen from the blog post AMQP in 10 minutes: Part 4 by Rajith Attapattu. This article is recommended reading.

channel
The channel the exchange is bound to (if bound).

durable
Durable exchanges remain active when a server restarts. Non-durable exchanges (transient exchanges) are purged when a server restarts. Default is `True`.

auto_delete
If set, the exchange is deleted when all queues have finished using it. Default is `False`.

delivery_mode
The default delivery mode used for messages. The value is an integer, or alias string.

- 1 or *"transient"*
  The message is transient. Which means it is stored in memory only, and is lost if the server dies or restarts.
- **2 or "persistent" (*default*)** The message is persistent. Which means the message is stored both in-memory, and on disk, and therefore preserved if the server dies or restarts.

The default value is 2 (persistent).

arguments
Additional arguments to specify when the exchange is declared.

Message(*body*, *delivery_mode=None*, *priority=None*, *content_type=None*, *content_encoding=None*, *properties=None*, *headers=None*)
Create message instance to be sent with `publish()`.

Parameters

- **body** – Message body.
- **delivery_mode** – Set custom delivery mode. Defaults to `delivery_mode`.
- **priority** – Message priority, 0 to 9. (currently not supported by RabbitMQ).
- **content_type** – The messages content_type. If content_type is set, no serialization occurs as it is assumed this is either a binary object, or you've done your own serialization. Leave blank if using built-in serialization as our library properly sets content_type.
- **content_encoding** – The character set in which this object is encoded. Use "binary" if sending in raw binary objects. Leave blank if using built-in serialization as our library properly sets content_encoding.
- **properties** – Message properties.
- **headers** – Message headers.

PERSISTENT_DELIVERY_MODE = 2

TRANSIENT_DELIVERY_MODE = 1

attrs = ((‘name’, None), (‘type’, None), (‘arguments’, None), (‘durable’, <type ‘bool’>), (‘passive’, <type ‘bool’>),

auto_delete = False

bind_to(*exchange=’‘*, *routing_key=’‘*, *arguments=None*, *nowait=False*, ***kwargs*)
Binds the exchange to another exchange.
Parameters **nowait** – If set the server will not respond, and the call will not block waiting for a response. Default is `False`.

can_cache_declaration

declare(*nowait=False*, *passive=None*)
Declare the exchange.

Creates the exchange on the broker.
Parameters **nowait** – If set the server will not respond, and a response will not be waited for. Default is `False`.

delete(*if_unused=False*, *nowait=False*)
Delete the exchange declaration on server.
Parameters

- **if_unused** – Delete only if the exchange has no bindings. Default is `False`.
- **nowait** – If set the server will not respond, and a response will not be waited for. Default is `False`.

delivery_mode = 2

durable = True

name = ‘’

passive = False

publish(*message*, *routing_key=None*, *mandatory=False*, *immediate=False*, *exchange=None*)
Publish message.
Parameters

- **message** – `Message()` instance to publish.
- **routing_key** – Routing key.
- **mandatory** – Currently not supported.
- **immediate** – Currently not supported.

type = ‘direct’

unbind_from(*source=’‘*, *routing_key=’‘*, *nowait=False*, *arguments=None*)
Delete previously created exchange binding from the server.

class AMQPBackend.**Producer**(*channel*, *exchange=None*, *routing_key=None*, *serializer=None*, *auto_declare=None*, *compression=None*, *on_return=None*)
Message Producer.
Parameters

- **channel** – Connection or channel.
- **exchange** – Optional default exchange.
- **routing_key** – Optional default routing key.
- **serializer** – Default serializer. Default is *“json”*.
- **compression** – Default compression method. Default is no compression.
- **auto_declare** – Automatically declare the default exchange at instantiation. Default is `True`.

- **on_return** – Callback to call for undeliverable messages, when the *mandatory* or *immediate* arguments to `publish()` is used. This callback needs the following signature: *(exception, exchange, routing_key, message)*. Note that the producer needs to drain events to use this feature.

auto_declare = True

channel

close()

compression = None

connection

declare()
Declare the exchange.

This happens automatically at instantiation if `auto_declare` is enabled.

exchange = None

maybe_declare(*entity*, *retry=False*, ***retry_policy*)
Declare the exchange if it hasn't already been declared during this session.

on_return = None

publish(*body*, *routing_key=None*, *delivery_mode=None*, *mandatory=False*, *immediate=False*, *priority=0*, *content_type=None*, *content_encoding=None*, *serializer=None*, *headers=None*, *compression=None*, *exchange=None*, *retry=False*, *retry_policy=None*, *declare=[]*, ***properties*)
Publish message to the specified exchange.

Parameters

- **body** – Message body.
- **routing_key** – Message routing key.
- **delivery_mode** – See `delivery_mode`.
- **mandatory** – Currently not supported.
- **immediate** – Currently not supported.
- **priority** – Message priority. A number between 0 and 9.
- **content_type** – Content type. Default is auto-detect.
- **content_encoding** – Content encoding. Default is auto-detect.
- **serializer** – Serializer to use. Default is auto-detect.
- **compression** – Compression method to use. Default is none.
- **headers** – Mapping of arbitrary headers to pass along with the message body.
- **exchange** – Override the exchange. Note that this exchange must have been declared.
- **declare** – Optional list of required entities that must have been declared before publishing the message. The entities will be declared using `maybe_declare()`.
- **retry** – Retry publishing, or declaring entities if the connection is lost.
- **retry_policy** – Retry configuration, this is the keywords supported by `ensure()`.

- ****properties** – Additional message properties, see AMQP spec.

release()

revive(*channel*)
Revive the producer after connection loss.

routing_key = ''

serializer = None

AMQPBackend.**Queue**
alias of NoCacheQueue

AMQPBackend.**consume**(*task_id*, *timeout=None*)

AMQPBackend.**delete_group**(*group_id*)

AMQPBackend.**drain_events**(*connection*, *consumer*, *timeout=None*, *now=<built-in function time>*)

AMQPBackend.**get_many**(*task_ids*, *timeout=None*, ***kwargs*)

AMQPBackend.**get_task_meta**(*task_id*, *backlog_limit=1000*)

AMQPBackend.**poll**(*task_id*, *backlog_limit=1000*)

AMQPBackend.**reload_group_result**(*task_id*)
Reload group result, even if it has been previously fetched.

AMQPBackend.**reload_task_result**(*task_id*)

AMQPBackend.**restore_group**(*group_id*, *cache=True*)

AMQPBackend.**retry_policy** = {'interval_start': 0, 'interval_max': 1, 'max_retries': 20, 'interval_step': 1}

AMQPBackend.**revive**(*channel*)

AMQPBackend.**save_group**(*group_id*, *result*)

AMQPBackend.**supports_autoexpire** = True

AMQPBackend.**supports_native_join** = True

AMQPBackend.**wait_for**(*task_id*, *timeout=None*, *cache=True*, *propagate=True*, ***kwargs*)

exception celery.backends.amqp.**BacklogLimitExceeded**
Too much state history to fast-forward.

class celery.backends.amqp.**NoCacheQueue**(*name=''*, *exchange=None*, *routing_key=''*, *channel=None*, *bindings=None*, *on_declared=None*, ***kwargs*)

can_cache_declaration = False

celery.backends.amqp.**repair_uuid**(*s*)

celery.backends.mongodb

- celery.backends.mongodb

celery.backends.mongodb

MongoDB result store backend.

class celery.backends.mongodb.**Bunch**(***kw*)

class celery.backends.mongodb.**MongoBackend**(**args*, ***kwargs*)

cleanup()
Delete expired metadata.

collection
Get the metadata task collection.

database
Get database from MongoDB connection and perform authentication if necessary.

mongodb_database = 'celery'

mongodb_host = 'localhost'

mongodb_max_pool_size = 10

mongodb_options = None

mongodb_password = None

mongodb_port = 27017

mongodb_taskmeta_collection = 'celery_taskmeta'

mongodb_user = None

process_cleanup()

supports_autoexpire = False

celery.backends.redis

celery.backends.redis

Redis result store backend.

class celery.backends.redis.**RedisBackend**(*host=None*, *port=None*, *db=None*, *password=None*, *expires=None*, *max_connections=None*, *url=None*, ***kwargs*)

Redis task result store.

client

db = 0
default Redis db number (0)

delete(*key*)

expire(*key*, *value*)

get(*key*)

host = 'localhost'
default Redis server hostname (*localhost*).

implements_incr = True

incr(*key*)

max_connections = None
Maximium number of connections in the pool.

mget(*keys*)

password = None
default Redis password (`None`)

port = 6379
default Redis server port (6379)

redis = None
redis-py client module.

set(*key*, *value*)

supports_autoexpire = True

supports_native_join = True

celery.backends.cassandra

celery.backends.cassandra

Apache Cassandra result store backend.

class `celery.backends.cassandra.`**CassandraBackend**(*servers=None*, *keyspace=None*, *column_family=None*, *cassandra_options=None*, *detailed_mode=False*, ***kwargs*)

Highly fault tolerant Cassandra backend.

servers
List of Cassandra servers with format: `hostname:port`.

Raises **celery.exceptions.ImproperlyConfigured** if module `pycassa` is not available.

column_family = None

detailed_mode = False

keyspace = None

process_cleanup()

servers = []

supports_autoexpire = True

celery.task.trace

celery.task.trace

This module defines how the task execution is traced: errors are recorded, handlers are applied and so on.

class celery.task.trace.**TraceInfo**(*state*, *retval=None*)

handle_error_state(*task*, *eager=False*)

handle_failure(*task*, *store_errors=True*)
Handle exception.

handle_retry(*task*, *store_errors=True*)
Handle retry exception.

retval

state

celery.task.trace.**build_tracer**(*name*, *task*, *loader=None*, *hostname=None*, *store_errors=True*, *Info=<class 'celery.task.trace.TraceInfo'>*, *eager=False*, *propagate=False*, *IGNORE_STATES=frozenset(['IGNORED', 'RETRY'])*)
Builts a function that tracing the tasks execution; catches all exceptions, and saves the state and result of the task execution to the result backend.

If the call was successful, it saves the result to the task result backend, and sets the task status to *"SUCCESS"*.

If the call raises RetryTaskError, it extracts the original exception, uses that as the result and sets the task status to *"RETRY"*.

If the call results in an exception, it saves the exception as the task result, and sets the task status to *"FAILURE"*.

Returns a function that takes the following arguments:

param uuid The unique id of the task.
param args List of positional args to pass on to the function.
param kwargs Keyword arguments mapping to pass on to the function.
keyword request Request dict.

celery.task.trace.**eager_trace_task**(*task*, *uuid*, *args*, *kwargs*, *request=None*, ***opts*)

celery.task.trace.**mro_lookup**(*cls*, *attr*, *stop=()*, *monkey_patched=[]*)
Returns the first node by MRO order that defines an attribute.

Parameters
- **stop** – A list of types that if reached will stop the search.
- **monkey_patched** – Use one of the stop classes if the attr's module origin is not in this list, this to detect monkey patched attributes.

Returns None if the attribute was not found.

celery.task.trace.**report_internal_error**(*task*, *exc*)

celery.task.trace.**reset_worker_optimizations**()

celery.task.trace.**setup_worker_optimizations**(*app*)

celery.task.trace.**task_has_custom**(*task*, *attr*)
Returns true if the task or one of its bases defines `attr` (excluding the one in BaseTask).

celery.task.trace.**trace_task**(*task*, *uuid*, *args*, *kwargs*, *request={}*, ***opts*)

celery.task.trace.**trace_task_ret**(*name*, *uuid*, *args*, *kwargs*, *request={}*, ***opts*)

celery.app.abstract

celery.app.abstract

Abstract class that takes default attribute values from the configuration.

class celery.app.abstract.**configurated**

confopts_as_dict()

setup_defaults(*kwargs*, *namespace='celery'*)

class celery.app.abstract.**from_config**(*key=None*)

get_key(*attr*)

celery.app.annotations

celery.app.annotations

Annotations is a nice term for moneky patching task classes in the configuration.

This prepares and performs the annotations in the `CELERY_ANNOTATIONS` setting.

class celery.app.annotations.**MapAnnotation**

annotate(*task*)

annotate_any()

celery.app.annotations.**prepare**(*annotations*)
Expands the `CELERY_ANNOTATIONS` setting.

celery.app.annotations.**resolve_all**(*anno*, *task*)

celery.app.routes

celery.routes

Contains utilities for working with task routers, (CELERY_ROUTES).

class celery.app.routes.**MapRoute**(*map*)
Creates a router out of a dict.

route_for_task(*task*, **args*, ***kwargs*)

class celery.app.routes.**Router**(*routes=None*, *queues=None*, *create_missing=False*, *app=None*)

expand_destination(*route*)

lookup_route(*task*, *args=None*, *kwargs=None*)

route(*options*, *task*, *args=()*, *kwargs={}*)

celery.app.routes.**prepare**(*routes*)
Expands the CELERY_ROUTES setting.

celery.security.certificate

celery.security.certificate

X.509 certificates.

class celery.security.certificate.**CertStore**
Base class for certificate stores

add_cert(*cert*)

itercerts()
an iterator over the certificates

class celery.security.certificate.**Certificate**(*cert*)
X.509 certificate.

get_id()
Serial number/issuer pair uniquely identifies a certificate

get_issuer()
Returns issuer (CA) as a string

get_serial_number()
Returns the certificates serial number.

has_expired()
Check if the certificate has expired.

verify(*data*, *signature*, *digest*)
Verifies the signature for string containing data.

class celery.security.certificate.**FSCertStore**(*path*)
File system certificate store

celery.security.key

celery.security.key

Private key for the security serializer.

class celery.security.key.**PrivateKey**(*key*)

sign(*data*, *digest*)
sign string containing data.

celery.security.serialization

celery.security.serialization

Secure serializer.

class celery.security.serialization.**SecureSerializer**(*key=None*, *cert=None*, *cert_store=None*, *digest='sha1'*, *serializer='json'*)

deserialize(*data*)
deserialize data structure from string

serialize(*data*)
serialize data structure into string

celery.security.serialization.**b64decode**(*s*)

celery.security.serialization.**b64encode**(*s*)

celery.security.serialization.**register_auth**(*key=None*, *cert=None*, *store=None*, *digest='sha1'*, *serializer='json'*)
register security serializer

celery.security.utils

celery.security.utils

Utilities used by the message signing serializer.

celery.security.utils.**reraise_errors**(**args*, ***kwds*)

celery.datastructures

Custom types and data structures.

AttributeDict

class celery.datastructures.**AttributeDict**
Dict subclass with attribute access.

class celery.datastructures.**AttributeDictMixin**
Adds attribute access to mappings.

d.key -> d[key]

DictAttribute

class celery.datastructures.**DictAttribute**(*obj*)
Dict interface to attributes.

obj[k] -> obj.k

get(*key*, *default=None*)

items()

iteritems()

iterkeys()

keys()

obj = None

setdefault(*key*, *default*)

ConfigurationView

class celery.datastructures.**ConfigurationView**(*changes*, *defaults*)
A view over an applications configuration dicts.

If the key does not exist in changes, the defaults dicts are consulted.

Parameters

- **changes** – Dict containing changes to the configuration.
- **defaults** – List of dicts containing the default configuration.

add_defaults(*d*)

changes = None

defaults = None

first(**keys*)

get(*key*, *default=None*)

items()

iteritems()

iterkeys()

itervalues()

keys()

setdefault(*key*, *default*)

update(**args*, ***kwargs*)

values()

ExceptionInfo

class celery.datastructures.**ExceptionInfo**(*exc_info=None*, *internal=False*)

Exception wrapping an exception and its traceback.

Parameters **exc_info** – The exception info tuple as returned by `sys.exc_info()`.

exception = None

Exception instance.

internal = False

Set to true if this is an internal error.

tb = None

Pickleable traceback instance for use with `traceback`

traceback = None

String representation of the traceback.

type = None

Exception type.

LimitedSet

class celery.datastructures.**LimitedSet**(*maxlen=None*, *expires=None*, *data=None*, *heap=None*)

Kind-of Set with limitations.

Good for when you need to test for membership (*a in set*), but the list might become to big, so you want to limit it so it doesn't consume too much resources.

Parameters

- **maxlen** – Maximum number of members before we start evicting expired members.
- **expires** – Time in seconds, before a membership expires.

add(*value*)

Add a new member.

as_dict()

chronologically

clear()
Remove all members

expires

first
Get the oldest member.

maxlen

pop_value(*value*)
Remove membership by finding value.

purge(*limit=None*)

update(*other*, *heappush=<built-in function heappush>*)

LRUCache

class celery.datastructures.**LRUCache**(*limit=None*)
LRU Cache implementation using a doubly linked list to track access.
Parameters limit – The maximum number of keys to keep in the cache. When a new key is inserted and the limit has been exceeded, the *Least Recently Used* key will be discarded from the cache.

incr(*key*, *delta=1*)

items()

iteritems()

itervalues()

keys()

update(**args*, ***kwargs*)

values()

celery.events.snapshot

celery.events.snapshot

Consuming the events as a stream is not always suitable so this module implements a system to take snapshots of the state of a cluster at regular intervals. There is a full implementation of this writing the snapshots to a database in djcelery.snapshots in the *django-celery* distribution.

class celery.events.snapshot.**Polaroid**(*state*, *freq=1.0*, *maxrate=None*, *cleanup_freq=3600.0*, *timer=None*, *app=None*)

cancel()

capture()

cleanup()

cleanup_signal = <Signal: Signal>

clear_after = False

install()

on_cleanup()

on_shutter(*state*)

shutter()

shutter_signal = <Signal: Signal>

timer = <module 'celery.utils.timer2' from '../celery/utils/timer2.pyc'>

celery.events.snapshot.**evcam**(*camera*, *freq=1.0*, *maxrate=None*, *loglevel=0*, *logfile=None*, *pidfile=None*, *timer=None*, *app=None*)

celery.events.cursesmon

celery.events.cursesmon

Graphical monitor of Celery events using curses.

class celery.events.cursesmon.**CursesMonitor**(*state*, *keymap=None*, *app=None*)

alert(*callback*, *title=None*)

alert_remote_control_reply(*reply*)

background = 7

display_height

display_task_row(*lineno*, *task*)

display_width

draw()

find_position()

foreground = 0

format_row(*uuid*, *task*, *worker*, *timestamp*, *state*)

greet = 'celeryev 3.0.25 (Chiastic Slide)'

handle_keypress()

help = 'j:up k:down i:info t:traceback r:result c:revoke ^c: quit'

help_title = 'Keys: '

info_str = 'Info: '

init_screen()

keyalias = {258: 'J', 259: 'K', 343: 'I'}

keymap = {}

limit

move_selection(*direction=1*)

move_selection_down()

move_selection_up()

nap()

online_str = 'Workers online: '

readline(*x*, *y*)

resetscreen()

revoke_selection()

safe_add_str(*y*, *x*, *string*, **args*, ***kwargs*)

screen_delay = 10

screen_height

screen_width

selected_position = 0

selected_str = 'Selected: '

selected_task = None

selection_info()

selection_rate_limit()

selection_result()

selection_traceback()

tasks

win = None

workers

class celery.events.cursesmon.**DisplayThread**(*display*)

run()

celery.events.cursesmon.**capture_events**(*app*, *state*, *display*)

celery.events.cursesmon.**evtop**(*app=None*)

celery.events.dumper

celery.events.dumper

This is a simple program that dumps events to the console as they happen. Think of it like a *tcpdump* for Celery events.

class celery.events.dumper.**Dumper**(*out=<open file '<stdout>', mode 'w' at 0x7fb76c556150>*)

format_task_event(*hostname*, *timestamp*, *type*, *task*, *ev*)

on_event(*ev*)

say(*msg*)

celery.events.dumper.**evdump**(*app=None*, *out=<open file '<stdout>', mode 'w' at 0x7fb76c556150>*)

celery.events.dumper.**humanize_type**(*type*)

celery.events.dumper.**say**(*msg*, *out=<open file '<stdout>', mode 'w' at 0x7fb76c556150>*)

celery.backends.database.models

celery.backends.database.models

Database tables for the SQLAlchemy result store backend.

class celery.backends.database.models.**Task**(*task_id*)
Task result/status.

date_done

id

result

status

task_id

to_dict()

traceback

class celery.backends.database.models.**TaskSet**(*taskset_id*, *result*)
TaskSet result

date_done

id

result

taskset_id

to_dict()

celery.backends.database.session

celery.backends.database.session

SQLAlchemy sessions.

celery.backends.database.session.**ResultSession**(*dburi*, ***kwargs*)

celery.backends.database.session.**create_session**(*dburi*, *short_lived_sessions=False*, ***kwargs*)

celery.backends.database.session.**get_engine**(*dburi*, ***kwargs*)

celery.backends.database.session.**setup_results**(*engine*)

celery.utils

celery.utils

Utility functions.

celery.utils.**MP_MAIN_FILE = None**
Billiard sets this when execv is enabled. We use it to find out the name of the original `__main__` module, so that we can properly rewrite the name of the task to be that of `App.main`.

celery.utils.**WORKER_DIRECT_EXCHANGE = <unbound Exchange C.dq(direct)>**
Exchange for worker direct queues.

celery.utils.**WORKER_DIRECT_QUEUE_FORMAT = '%s.dq'**
Format for worker direct queue names.

celery.utils.**cry**()
Return stacktrace of all active threads.

From https://gist.github.com/737056

celery.utils.**deprecated**(*description=None*, *deprecation=None*, *removal=None*, *alternative=None*)

celery.utils.**fun_takes_kwargs**(*fun*, *kwlist=[]*)
With a function, and a list of keyword arguments, returns arguments in the list which the function takes.

If the object has an *argspec* attribute that is used instead of using the `inspect.getargspec()` introspection.

Parameters
- **fun** – The function to inspect arguments of.
- **kwlist** – The list of keyword arguments.

Examples

```
>>> def foo(self, x, y, logfile=None, loglevel=None):
...     return x * y
>>> fun_takes_kwargs(foo, ['logfile', 'loglevel', 'task_id'])
['logfile', 'loglevel']

>>> def foo(self, x, y, **kwargs):
>>> fun_takes_kwargs(foo, ['logfile', 'loglevel', 'task_id'])
['logfile', 'loglevel', 'task_id']
```

celery.utils.**gen_task_name**(*app*, *name*, *module_name*)

celery.utils.**is_iterable**(*obj*)

celery.utils.**isatty**(*fh*)

celery.utils.**jsonify**(*obj*)
Transforms object making it suitable for json serialization

celery.utils.**lpmerge**(*L*, *R*)
In place left precedent dictionary merge.

Keeps values from *L*, if the value in *R* is `None`.

celery.utils.**maybe_reraise**()
Reraise if an exception is currently being handled, or return otherwise.

celery.utils.**strtobool**(*term*, *table={'1': True, '0': False, 'false': False, 'no': False, 'off': False, 'yes': True, 'on': True, 'true': True}*)

celery.utils.**warn_deprecated**(*description=None*, *deprecation=None*, *removal=None*, *alternative=None*)

celery.utils.**worker_direct**(*hostname*)

celery.utils.functional

celery.utils.functional

Utilities for functions.

class celery.utils.functional.**LRUCache**(*limit=None*)
LRU Cache implementation using a doubly linked list to track access.

Parameters limit – The maximum number of keys to keep in the cache. When a new key is inserted and the limit has been exceeded, the *Least Recently Used* key will be discarded from the cache.

incr(*key*, *delta=1*)

items()

iteritems()

itervalues()

keys()

update(**args*, ***kwargs*)

values()

celery.utils.functional.**chunks**(*it*, *n*)
Split an iterator into chunks with *n* elements each.

Examples
n == 2 >>> x = chunks(iter([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]), 2) >>> list(x) [[0, 1], [2, 3], [4, 5], [6, 7], [8, 9], [10]]

n == 3 >>> x = chunks(iter([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]), 3) >>> list(x) [[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10]]

celery.utils.functional.**first**(*predicate*, *it*)
Returns the first element in *iterable* that *predicate* returns a `True` value for.

If *predicate* is None it will return the first item that is not None.

celery.utils.functional.**firstmethod**(*method*)
Returns a function that with a list of instances, finds the first instance that returns a value for the given method.

The list can also contain promises (`promise`.)

celery.utils.functional.**is_list**(*l*)
Returns true if object is list-like, but not a dict or string.

celery.utils.functional.**mattrgetter**(**attrs*)
Like `operator.itemgetter()` but returns `None` on missing attributes instead of raising `AttributeError`.

celery.utils.functional.**maybe_list**(*l*)
Returns list of one element if `l` is a scalar.

celery.utils.functional.**memoize**(*maxsize=None*, *Cache=<class celery.utils.functional.LRUCache at 0x7fb768d58870>*)

class celery.utils.functional.**mpromise**(*fun*, **args*, ***kwargs*)
Memoized promise.

The function is only evaluated once, every subsequent access will return the same value.

evaluated
Set to to `True` after the promise has been evaluated.

evaluate()

evaluated = False

celery.utils.functional.**noop**(**args*, ***kwargs*)
No operation.

Takes any arguments/keyword arguments and does nothing.

celery.utils.functional.**padlist**(*container*, *size*, *default=None*)
Pad list with default elements.

Examples:

```
>>> first, last, city = padlist(['George', 'Costanza', 'NYC'], 3)
('George', 'Costanza', 'NYC')
>>> first, last, city = padlist(['George', 'Costanza'], 3)
('George', 'Costanza', None)
>>> first, last, city, planet = padlist(['George', 'Costanza',
                                         'NYC'], 4, default='Earth')
('George', 'Costanza', 'NYC', 'Earth')
```

celery.utils.functional.**regen**(*it*)
Regen takes any iterable, and if the object is an generator it will cache the evaluated list on first access, so that the generator can be "consumed" multiple times.

celery.utils.functional.**uniq**(*it*)
Returns all unique elements in `it`, preserving order.

celery.utils.term

celery.utils.term

Terminals and colors.

class celery.utils.term.**colored**(**s*, ***kwargs*)
Terminal colored text.
Example::

```
>>> c = colored(enabled=True)
>>> print(str(c.red('the quick '), c.blue('brown ', c.bold('fox ')),
...       c.magenta(c.underline('jumps over')),
...       c.yellow(' the lazy '),
...       c.green('dog ')))
```

black(**s*)

blink(**s*)

blue(**s*)

bold(**s*)

bright(**s*)

cyan(**s*)

embed()

green(**s*)

iblue(**s*)

icyan(**s*)

igreen(**s*)

imagenta(**s*)

ired(**s*)

iwhite(**s*)

iyellow(**s*)

magenta(**s*)

no_color()

node(*s*, *op*)

red(**s*)

reset(**s*)

reverse(**s*)

underline(**s*)

white(**s*)

yellow(**s*)

celery.utils.term.**fg**(*s*)

celery.utils.timeutils

celery.utils.timeutils

This module contains various utilities related to dates and times.

exception celery.utils.timeutils.**AmbiguousTimeError**

class celery.utils.timeutils.**LocalTimezone**
Local time implementation taken from Python's docs.

Used only when pytz isn't available, and most likely inaccurate. If you're having trouble with this class, don't waste your time, just install pytz.

dst(*dt*)

tzname(*dt*)

utcoffset(*dt*)

celery.utils.timeutils.**delta_resolution**(*dt*, *delta*)
Round a datetime to the resolution of a timedelta.

If the timedelta is in days, the datetime will be rounded to the nearest days, if the timedelta is in hours the datetime will be rounded to the nearest hour, and so on until seconds which will just return the original datetime.

celery.utils.timeutils.**humanize_seconds**(*secs*, *prefix=''*, *sep=''*)
Show seconds in human form, e.g. 60 is "1 minute", 7200 is "2 hours".

Parameters **prefix** – Can be used to add a preposition to the output, e.g. 'in' will give 'in 1 second', but add nothing to 'now'.

celery.utils.timeutils.**is_naive**(*dt*)
Returns `True` if the datetime is naive (does not have timezone information).

celery.utils.timeutils.**localize**(*dt*, *tz*)
Convert aware datetime to another timezone.

celery.utils.timeutils.**make_aware**(*dt*, *tz*)
Sets the timezone for a datetime object.

celery.utils.timeutils.**maybe_iso8601**(*dt*)
Either datetime | str -> datetime or None -> None

celery.utils.timeutils.**maybe_make_aware**(*dt*, *tz=None*)

celery.utils.timeutils.**maybe_timedelta**(*delta*)
Coerces integer to timedelta if *delta* is an integer.

celery.utils.timeutils.**rate**(*rate*)
Parses rate strings, such as *"100/m"*, *"2/h"* or *"0.5/s"* and converts them to seconds.

celery.utils.timeutils.**remaining**(*start*, *ends_in*, *now=None*, *relative=False*, *debug=False*)
Calculate the remaining time for a start date and a timedelta.

e.g. "how many seconds left for 30 seconds after start?"

Parameters

- **start** – Start `datetime`.
- **ends_in** – The end delta as a `timedelta`.
- **relative** – If enabled the end time will be calculated using `delta_resolution()` (i.e. rounded to the resolution of *ends_in*).
- **now** – Function returning the current time and date, defaults to `datetime.utcnow()`.

celery.utils.timeutils.**timedelta_seconds**(*delta*)
Convert `datetime.timedelta` to seconds.

Doesn't account for negative values.

celery.utils.timeutils.**to_utc**(*dt*)
Converts naive datetime to UTC

celery.utils.timeutils.**weekday**(*name*)
Return the position of a weekday (0 - 7, where 0 is Sunday).

Example:

```
>>> weekday('sunday'), weekday('sun'), weekday('mon')
(0, 0, 1)
```

celery.utils.compat

- celery.utils.compat

celery.utils.compat

Compatibility implementations of features only available in newer Python versions.

celery.utils.compat.**chain_from_iterable**()
chain.from_iterable(iterable) –> chain object

Alternate chain() contructor taking a single iterable argument that evaluates lazily.

celery.utils.compat.**format_d**(*i*)

celery.utils.serialization

- celery.utils.serialization

celery.utils.serialization

Utilities for safely pickling exceptions.

exception celery.utils.serialization.**UnpickleableExceptionWrapper**(*exc_module*, *exc_cls_name*, *exc_args*, *text=None*)

Wraps unpickleable exceptions.

Parameters

- **exc_module** – see exc_module.
- **exc_cls_name** – see exc_cls_name.
- **exc_args** – see exc_args

Example

```
>>> try:
...     something_raising_unpickleable_exc()
>>> except Exception, e:
...     exc = UnpickleableException(e.__class__.__module__,
...                                 e.__class__.__name__,
...                                 e.args)
...     pickle.dumps(exc) # Works fine.
```

exc_args = None

The arguments for the original exception.

exc_cls_name = None

The name of the original exception class.

exc_module = None

The module of the original exception.

classmethod from_exception(*exc*)

restore()

celery.utils.serialization.**create_exception_cls**(*name*, *module*, *parent=None*)

Dynamically create an exception class.

celery.utils.serialization.**find_nearest_pickleable_exception**(*exc*, *loads=<built-in function loads>*, *dumps=<built-in function dumps>*)

With an exception instance, iterate over its super classes (by mro) and find the first super exception that is pickleable. It does not go below Exception (i.e. it skips Exception, BaseException and object). If that happens you should use UnpickleableException instead.

Parameters **exc** – An exception instance.

Returns the nearest exception if it's not Exception or below, if it is it returns None.

:rtype Exception:

celery.utils.serialization.**find_pickleable_exception**(*exc*, *loads=<built-in function loads>*, *dumps=<built-in function dumps>*)

With an exception instance, iterate over its super classes (by mro) and find the first super exception that is pickleable. It does not go below Exception (i.e. it skips Exception, BaseException and object). If that happens you should use UnpickleableException instead.

Parameters **exc** – An exception instance.

Returns the nearest exception if it's not Exception or below, if it is it returns None.

:rtype Exception:

celery.utils.serialization.**get_pickleable_etype**(*cls*, *loads=<built-in function loads>*, *dumps=<built-in function dumps>*)

celery.utils.serialization.**get_pickleable_exception**(*exc*)
Make sure exception is pickleable.

celery.utils.serialization.**get_pickled_exception**(*exc*)
Get original exception from exception pickled using `get_pickleable_exception()`.

celery.utils.serialization.**itermro**(*cls*, *stop*)

celery.utils.serialization.**subclass_exception**(*name*, *parent*, *module*)

celery.utils.serialization.**unwanted_base_classes = (<type 'exceptions.StandardError'>, <type 'exceptions.E**
List of base classes we probably don't want to reduce to.

celery.utils.threads

celery.utils.threads

Threading utilities.

class celery.utils.threads.**Local**

class celery.utils.threads.**LocalManager**(*locals=None*, *ident_func=None*)
Local objects cannot manage themselves. For that you need a local manager. You can pass a local manager multiple locals or add them later by appending them to *manager.locals*. Everytime the manager cleans up it, will clean up all the data left in the locals for this context.

The *ident_func* parameter can be added to override the default ident function for the wrapped locals.

cleanup()
Manually clean up the data in the locals for this context.

Call this at the end of the request or use *make_middleware()*.

get_ident()
Return the context identifier the local objects use internally for this context. You cannot override this method to change the behavior but use it to link other context local objects (such as SQLAlchemy's scoped sessions) to the Werkzeug locals.

celery.utils.threads.**LocalStack**
alias of `_LocalStack`

class celery.utils.threads.**bgThread**(*name=None*, ***kwargs*)

body()

on_crash(*msg*, **fmt*, ***kwargs*)

run()

stop()
Graceful shutdown.

celery.utils.threads.**release_local**(*local*)
Releases the contents of the local for the current context. This makes it possible to use locals without a manager.

Example:

```
>>> loc = Local()
>>> loc.foo = 42
>>> release_local(loc)
>>> hasattr(loc, 'foo')
False
```

With this function one can release `Local` objects as well as `StackLocal` objects. However it is not possible to release data held by proxies that way, one always has to retain a reference to the underlying local object in order to be able to release it.

New in version 0.6.1.

celery.utils.timer2

timer2

Scheduler for Python functions.

class celery.utils.timer2.**Entry**(*fun*, *args=None*, *kwargs=None*)

args

cancel()

cancelled

fun

kwargs

tref

class celery.utils.timer2.**Schedule**(*max_interval=None*, *on_error=None*, ***kwargs*)
ETA scheduler.

class Entry(*fun*, *args=None*, *kwargs=None*)

args

cancel()

cancelled

fun

kwargs

tref

Schedule.**apply_after**(*msecs*, *fun*, *args=()*, *kwargs={}*, *priority=0*)

Schedule.**apply_at**(*eta*, *fun*, *args=()*, *kwargs={}*, *priority=0*)

Schedule.**apply_entry**(*entry*)

Schedule.**apply_interval**(*msecs*, *fun*, *args=()*, *kwargs={}*, *priority=0*)

Schedule.**cancel**(*tref*)

Schedule.**clear**()

Schedule.**empty**()
Is the schedule empty?

Schedule.**enter**(*entry*, *eta=None*, *priority=0*)
Enter function into the scheduler.

Parameters
- **entry** – Item to enter.
- **eta** – Scheduled time as a `datetime.datetime` object.
- **priority** – Unused.

Schedule.**enter_after**(*msecs*, *entry*, *priority=0*, *time=<built-in function time>*)

Schedule.**handle_error**(*exc_info*)

Schedule.**info**()

Schedule.**on_error = None**

Schedule.**queue**
Snapshot of underlying datastructure.

Schedule.**schedule**

Schedule.**stop**()

class celery.utils.timer2.**Timer**(*schedule=None*, *on_error=None*, *on_tick=None*, *max_interval=None*, ***kwargs*)

class Entry(*fun*, *args=None*, *kwargs=None*)

args

cancel()

cancelled

fun

kwargs

tref

class Timer.**Schedule**(*max_interval=None*, *on_error=None*, ***kwargs*)
ETA scheduler.

class Entry(*fun*, *args=None*, *kwargs=None*)

args

cancel()

cancelled

fun

kwargs

tref

Timer.Schedule.**apply_after**(*msecs*, *fun*, *args=()*, *kwargs={}*, *priority=0*)

Timer.Schedule.**apply_at**(*eta*, *fun*, *args=()*, *kwargs={}*, *priority=0*)

Timer.Schedule.**apply_entry**(*entry*)

Timer.Schedule.**apply_interval**(*msecs*, *fun*, *args=()*, *kwargs={}*, *priority=0*)

Timer.Schedule.**cancel**(*tref*)

Timer.Schedule.**clear**()

Timer.Schedule.**empty**()
Is the schedule empty?

Timer.Schedule.**enter**(*entry*, *eta=None*, *priority=0*)
Enter function into the scheduler.

Parameters

- **entry** – Item to enter.
- **eta** – Scheduled time as a `datetime.datetime` object.
- **priority** – Unused.

Timer.Schedule.**enter_after**(*msecs*, *entry*, *priority=0*, *time=<built-in function time>*)

Timer.Schedule.**handle_error**(*exc_info*)

Timer.Schedule.**info**()

Timer.Schedule.**on_error = None**

Timer.Schedule.**queue**
Snapshot of underlying datastructure.

Timer.Schedule.**schedule**

Timer.Schedule.**stop**()

Timer.**apply_after**(**args*, ***kwargs*)

Timer.**apply_at**(**args*, ***kwargs*)

Timer.**apply_interval**(**args*, ***kwargs*)

Timer.**cancel**(*tref*)

Timer.**clear**()

Timer.**empty**()

Timer.**ensure_started**()

Timer.**enter**(*entry*, *eta*, *priority=None*)

Timer.**enter_after**(**args*, ***kwargs*)

Timer.**exit_after**(*msecs*, *priority=10*)

Timer.**next**()

Timer.**on_tick = None**

Timer.**queue**

Timer.**run**()

Timer.**running** = False

Timer.**stop**()

celery.utils.timer2.**to_timestamp**(*d*, *default_timezone=tzfile('/usr/share/zoneinfo/UTC')*)

celery.utils.imports

celery.utils.import

Utilities related to importing modules and symbols by name.

exception celery.utils.imports.**NotAPackage**

celery.utils.imports.**cwd_in_path**(**args*, ***kwds*)

celery.utils.imports.**find_module**(*module*, *path=None*, *imp=None*)
Version of imp.find_module() supporting dots.

celery.utils.imports.**import_from_cwd**(*module*, *imp=None*, *package=None*)
Import module, but make sure it finds modules located in the current directory.

Modules located in the current directory has precedence over modules located in *sys.path*.

celery.utils.imports.**instantiate**(*name*, **args*, ***kwargs*)
Instantiate class by name.

See symbol_by_name().

celery.utils.imports.**module_file**(*module*)

celery.utils.imports.**qualname**(*obj*)

celery.utils.imports.**reload_from_cwd**(*module*, *reloader=None*)

celery.utils.log

celery.utils.log

Logging utilities.

class celery.utils.log.**ColorFormatter**(*fmt=None*, *use_color=True*)

COLORS = {'blue': <bound method colored.blue of ''>, 'black': <bound method colored.black of ''>, 'yellow': <bound me
Loglevel -> Color mapping.

colors = {'DEBUG': <bound method colored.blue of ''>, 'CRITICAL': <bound method colored.magenta of ''>, 'WARN

format(*record*)

formatException(*ei*)

class celery.utils.log.**LoggingProxy**(*logger*, *loglevel=None*)
Forward file object to logging.Logger instance.

Parameters

- **logger** – The logging.Logger instance to forward to.
- **loglevel** – Loglevel to use when writing messages.

close()
When the object is closed, no write requests are forwarded to the logging object anymore.

closed = False

flush()
This object is not buffered so any flush() requests are ignored.

isatty()
Always returns False. Just here for file support.

loglevel = 40

mode = 'w'

name = None

write(*data*)
Write message to logging object.

writelines(*sequence*)
writelines(sequence_of_strings) -> None.

Write the strings to the file.

The sequence can be any iterable object producing strings. This is equivalent to calling write() for each string.

celery.utils.log.**ensure_process_aware_logger**()
Make sure process name is recorded when loggers are used.

celery.utils.log.**get_logger**(*name*)

celery.utils.log.**get_multiprocessing_logger**()

celery.utils.log.**get_task_logger**(*name*)

celery.utils.log.**mlevel**(*level*)

celery.utils.log.**reset_multiprocessing_logger**()

celery.utils.log.**set_in_sighandler**(*value*)

celery.utils.text

celery.utils.text

Text formatting utilities

celery.utils.text.**abbr**(*S*, *max*, *ellipsis='...'*)

celery.utils.text.**abbrtask**(*S*, *max*)

celery.utils.text.**dedent**(*s*, *n=4*, *sep='\n'*)

celery.utils.text.**dedent_initial**(*s*, *n=4*)

celery.utils.text.**ensure_2lines**(*s*, *sep='\n'*)

celery.utils.text.**fill_paragraphs**(*s*, *width*, *sep='\n'*)

celery.utils.text.**indent**(*t*, *indent=0*, *sep='\n'*)
Indent text.

celery.utils.text.**join**(*l*, *sep='\n'*)

celery.utils.text.**pluralize**(*n*, *text*, *suffix='s'*)

celery.utils.text.**pretty**(*value*, *width=80*, *nl_width=80*, *sep='\n'*, ***kw*)

celery.utils.text.**truncate**(*text*, *maxlen=128*, *suffix='...'*)
Truncates text to a maximum number of characters.

celery.utils.dispatch

celery.utils.dispatch.signal

Signal class.

class celery.utils.dispatch.signal.**Signal**(*providing_args=None*)
Base class for all signals

receivers
Internal attribute, holds a dictionary of
`{receiverkey (id): weakref(receiver)}` mappings.

connect(**args*, ***kwargs*)
Connect receiver to sender for signal.

Parameters

- **receiver** – A function or an instance method which is to receive signals. Receivers must be hashable objects.

  if weak is `True`, then receiver must be weak-referencable (more precisely `saferef.safe_ref()` must be able to create a reference to the receiver).

  Receivers must be able to accept keyword arguments.

  If receivers have a *dispatch_uid* attribute, the receiver will not be added if another receiver already exists with that *dispatch_uid*.
- **sender** – The sender to which the receiver should respond. Must either be of type `Signal`, or `None` to receive events from any sender.
- **weak** – Whether to use weak references to the receiver. By default, the module will attempt to use weak references to the receiver objects. If this parameter is false, then strong references will be used.
- **dispatch_uid** – An identifier used to uniquely identify a particular instance of a receiver. This will usually be a string, though it may be anything hashable.

disconnect(*receiver=None*, *sender=None*, *weak=True*, *dispatch_uid=None*)
Disconnect receiver from sender for signal.

If weak references are used, disconnect need not be called. The receiver will be removed from dispatch automatically.

Parameters

- **receiver** – The registered receiver to disconnect. May be none if *dispatch_uid* is specified.
- **sender** – The registered sender to disconnect.
- **weak** – The weakref state to disconnect.
- **dispatch_uid** – the unique identifier of the receiver to disconnect

send(*sender*, ***named*)

Send signal from sender to all connected receivers.

If any receiver raises an error, the error propagates back through send, terminating the dispatch loop, so it is quite possible to not have all receivers called if a raises an error.

Parameters

- **sender** – The sender of the signal. Either a specific object or `None`.
- ****named** – Named arguments which will be passed to receivers.

Returns a list of tuple pairs: *[(receiver, response), ...]*.

send_robust(*sender*, ***named*)

Send signal from sender to all connected receivers catching errors.

Parameters

- **sender** – The sender of the signal. Can be any python object (normally one registered with a connect if you actually want something to occur).
- ****named** – Named arguments which will be passed to receivers. These arguments must be a subset of the argument names defined in `providing_args`.

Returns a list of tuple pairs: *[(receiver, response), ...]*.

Raises DispatcherKeyError

if any receiver raises an error (specifically any subclass of `Exception`), the error instance is returned as the result for that receiver.

celery.utils.dispatch.saferef

"Safe weakrefs", originally from pyDispatcher.

Provides a way to safely weakref any function, including bound methods (which aren't handled by the core weakref module).

class celery.utils.dispatch.saferef.**BoundMethodWeakref**(*target*, *on_delete=None*)

'Safe' and reusable weak references to instance methods.

BoundMethodWeakref objects provide a mechanism for referencing a bound method without requiring that the method object itself (which is normally a transient object) is kept alive. Instead, the BoundMethodWeakref object keeps weak references to both the object and the function which together define the instance method.

key

the identity key for the reference, calculated by the class's `calculate_key()` method applied to the target instance method

deletion_methods

sequence of callable objects taking single argument, a reference to this object which will be called when *either* the target object or target function is garbage collected (i.e. when this object becomes invalid). These are specified as the on_delete parameters of `safe_ref()` calls.

weak_self

weak reference to the target object

weak_func

weak reference to the target function

_all_instances
class attribute pointing to all live BoundMethodWeakref objects indexed by the class's *calculate_key(target)* method applied to the target objects. This weak value dictionary is used to short-circuit creation so that multiple references to the same (object, function) pair produce the same BoundMethodWeakref instance.

classmethod calculate_key(*target*)
Calculate the reference key for this reference

Currently this is a two-tuple of the *id()*'s of the target object and the target function respectively.

class celery.utils.dispatch.saferef.**BoundNonDescriptorMethodWeakref**(*target*, *on_delete=None*)
A specialized BoundMethodWeakref, for platforms where instance methods are not descriptors.

It assumes that the function name and the target attribute name are the same, instead of assuming that the function is a descriptor. This approach is equally fast, but not 100% reliable because functions can be stored on an attribute named differenty than the function's name such as in:

```
>>> class A(object):
...     pass

>>> def foo(self):
...     return "foo"
>>> A.bar = foo
```

But this shouldn't be a common use case. So, on platforms where methods aren't descriptors (such as Jython) this implementation has the advantage of working in the most cases.

celery.utils.dispatch.saferef.**get_bound_method_weakref**(*target*, *on_delete*)
Instantiates the appropiate BoundMethodWeakRef, depending on the details of the underlying class method implementation.

celery.utils.dispatch.saferef.**safe_ref**(*target*, *on_delete=None*)
Return a *safe* weak reference to a callable target

Parameters
- **target** – the object to be weakly referenced, if it's a bound method reference, will create a BoundMethodWeakref, otherwise creates a simple weakref.ref.
- **on_delete** – if provided, will have a hard reference stored to the callable to be called after the safe reference goes out of scope with the reference object, (either a weakref.ref or a BoundMethodWeakref) as argument.

celery.platforms

celery.platforms

Utilities dealing with platform specifics: signals, daemonization, users, groups, and so on.

class celery.platforms.**DaemonContext**(*pidfile=None*, *workdir=None*, *umask=None*, *fake=False*, *after_chdir=None*, ***kwargs*)

close(**args*)

open()

redirect_to_null(*fd*)

exception celery.platforms.**LockFailed**
Raised if a pidlock can't be acquired.

celery.platforms.**PIDFile**
alias of Pidfile

class celery.platforms.**Pidfile**(*path*)
Pidfile

This is the type returned by create_pidlock().

TIP: Use the create_pidlock() function instead, which is more convenient and also removes stale pidfiles (when the process holding the lock is no longer running).

acquire()
Acquire lock.

is_locked()
Returns true if the pid lock exists.

path = None
Path to the pid lock file.

read_pid()
Reads and returns the current pid.

release(**args*)
Release lock.

remove()
Removes the lock.

remove_if_stale()
Removes the lock if the process is not running. (does not respond to signals).

write_pid()

class celery.platforms.**Signals**
Convenience interface to signals.

If the requested signal is not supported on the current platform, the operation will be ignored.

Examples:

```
>>> from celery.platforms import signals

>>> signals['INT'] = my_handler

>>> signals['INT']
my_handler

>>> signals.supported('INT')
True

>>> signals.signum('INT')
2

>>> signals.ignore('USR1')
>>> signals['USR1'] == signals.ignored
```

```
True

>>> signals.reset('USR1')
>>> signals['USR1'] == signals.default
True

>>> signals.update(INT=exit_handler,
...                TERM=exit_handler,
...                HUP=hup_handler)
```

default = 0

ignore(**signal_names*)
Ignore signal using SIG_IGN.

Does nothing if the platform doesn't support signals, or the specified signal in particular.

ignored = 1

reset(**signal_names*)
Reset signals to the default signal handler.

Does nothing if the platform doesn't support signals, or the specified signal in particular.

signum(*signal_name*)
Get signal number from signal name.

supported(*signal_name*)
Returns true value if signal_name exists on this platform.

update(*_d_=None*, ***sigmap*)
Set signal handlers from a mapping.

celery.platforms.**close_open_fds**(*keep=None*)

celery.platforms.**create_pidlock**(*pidfile*)
Create and verify pidfile.

If the pidfile already exists the program exits with an error message, however if the process it refers to is not running anymore, the pidfile is deleted and the program continues.

This function will automatically install an atexit handler to release the lock at exit, you can skip this by calling _create_pidlock() instead.

Returns Pidfile.

Example:

```
pidlock = create_pidlock('/var/run/app.pid')
```

celery.platforms.**detached**(*logfile=None*, *pidfile=None*, *uid=None*, *gid=None*, *umask=0*, *workdir=None*, *fake=False*, ***opts*)
Detach the current process in the background (daemonize).

Parameters

- **logfile** – Optional log file. The ability to write to this file will be verified before the process is detached.
- **pidfile** – Optional pidfile. The pidfile will not be created, as this is the responsibility of the child. But the process will exit if the pid lock exists and the pid written is still running.
- **uid** – Optional user id or user name to change effective privileges to.
- **gid** – Optional group id or group name to change effective privileges to.
- **umask** – Optional umask that will be effective in the child process.

- **workdir** – Optional new working directory.
- **fake** – Don't actually detach, intented for debugging purposes.
- ****opts** – Ignored.

Example:

```
from celery.platforms import detached, create_pidlock

with detached(logfile='/var/log/app.log', pidfile='/var/run/app.pid',
              uid='nobody'):
    # Now in detached child process with effective user set to nobody,
    # and we know that our logfile can be written to, and that
    # the pidfile is not locked.
    pidlock = create_pidlock('/var/run/app.pid')

    # Run the program
    program.run(logfile='/var/log/app.log')
```

celery.platforms.**fileno**(*f*)

celery.platforms.**get_errno**(*n*)
Get errno for string, e.g. ENOENT.

celery.platforms.**get_fdmax**(*default=None*)
Returns the maximum number of open file descriptors on this system.

Parameters **default** – Value returned if there's no file descriptor limit.

celery.platforms.**ignore_errno**(**args*, ***kwds*)
Context manager to ignore specific POSIX error codes.

Takes a list of error codes to ignore, which can be either the name of the code, or the code integer itself:

```
>>> with ignore_errno('ENOENT'):
...     with open('foo', 'r'):
...         return r.read()

>>> with ignore_errno(errno.ENOENT, errno.EPERM):
...    pass
```

Parameters **types** – A tuple of exceptions to ignore (when the errno matches), defaults to Exception.

celery.platforms.**initgroups**(*uid*, *gid*)
Compat version of os.initgroups() which was first added to Python 2.7.

celery.platforms.**maybe_drop_privileges**(*uid=None*, *gid=None*)
Change process privileges to new user/group.

If UID and GID is specified, the real user/group is changed.

If only UID is specified, the real user is changed, and the group is changed to the users primary group.

If only GID is specified, only the group is changed.

celery.platforms.**maybe_fileno**(*f*)
Get object fileno, or None if not defined.

celery.platforms.**maybe_patch_concurrency**(*argv*, *short_opts=None*, *long_opts=None*)
With short and long opt alternatives that specify the command line option to set the pool, this makes sure that anything that needs to be patched is completed as early as possible. (e.g. eventlet/gevent monkey patches).

celery.platforms.**parse_gid**(*gid*)
Parse group id.

gid can be an integer (gid) or a string (group name), if a group name the gid is taken from the password file.

celery.platforms.**parse_uid**(*uid*)
Parse user id.

uid can be an integer (uid) or a string (user name), if a user name the uid is taken from the password file.

celery.platforms.**pyimplementation**()
Returns string identifying the current Python implementation.

celery.platforms.**set_mp_process_title**(*progname*, *info=None*, *hostname=None*)
Set the ps name using the multiprocessing process name.

Only works if `setproctitle` is installed.

celery.platforms.**set_process_title**(*progname*, *info=None*)
Set the ps name for the currently running process.

Only works if `setproctitle` is installed.

celery.platforms.**setgid**(*gid*)
Version of `os.setgid()` supporting group names.

celery.platforms.**setgroups**(*groups*)
Set active groups from a list of group ids.

celery.platforms.**setuid**(*uid*)
Version of `os.setuid()` supporting usernames.

celery.platforms.**shellsplit**(*s*)
Compat. version of `shlex.split()` that supports the `posix` option which was first added in Python 2.6.

Posix behavior will be disabled if running under Windows.

celery.platforms.**strargv**(*argv*)

celery._state

celery._state

This is an internal module containing thread state like the `current_app`, and `current_task`.

This module shouldn't be used directly.

celery._state.**current_app = <Celery __main__:0x7fb76929c510>**
Proxy to current app.

celery._state.**current_task = None**
Proxy to current task.

celery._state.**default_app = <Celery default:0x7fb766b79910>**
Global default app used when no current app.

celery._state.**get_current_app**()

celery._state.**get_current_task**()
Currently executing task.

celery._state.**get_current_worker_task**()
Currently executing task, that was applied by the worker.

This is used to differentiate between the actual task executed by the worker and any task that was called within a task (using `task.__call__` or `task.apply`)

celery._state.**set_default_app**(*app*)

2.15 History

This section contains historical change histories, for the latest version please visit *Change history*.

Release 3.0

Date July 10, 2014

2.15.1 Change history for Celery 2.5

2.5.5

release-date 2012-06-06 16:00 P.M BST

by Ask Solem

This is a dummy release performed for the following goals:

- Protect against force upgrading to Kombu 2.2.0
- Version parity with django-celery

2.5.3

release-date 2012-04-16 07:00 P.M BST

by Ask Solem

- A bug causes messages to be sent with UTC timestamps even though `CELERY_ENABLE_UTC` was not enabled (Issue #636).
- celerybeat: No longer crashes if an entry's args is set to None (Issue #657).
- Autoreload did not work if a module's `__file__` attribute was set to the modules '.pyc' file. (Issue #647).

- Fixes early 2.5 compatibility where __package__ does not exist (Issue #638).

2.5.2

release-date 2012-04-13 04:30 P.M GMT

by Ask Solem

News

- Now depends on Kombu 2.1.5.
- Django documentation has been moved to the main Celery docs.

  See *Django*.
- New `celeryd_init` signal can be used to configure workers by hostname.
- Signal.connect can now be used as a decorator.

  Example:

```
from celery.signals import task_sent

@task_sent.connect
def on_task_sent(**kwargs):
    print("sent task: %r" % (kwargs, ))
```

- Invalid task messages are now rejected instead of acked.

  This means that they will be moved to the dead-letter queue introduced in the latest RabbitMQ version (but must be enabled manually, consult the RabbitMQ documentation).
- Internal logging calls has been cleaned up to work better with tools like Sentry.

  Contributed by David Cramer.
- New method `subtask.clone()` can be used to clone an existing subtask with augmented arguments/options.

  Example:

```
>>> s = add.subtask((5, ))
>>> new = s.clone(args=(10, ), countdown=5})
>>> new.args
(10, 5)

>>> new.options
{"countdown": 5}
```

- Chord callbacks are now triggered in eager mode.

Fixes

- Programs now verifies that the pidfile is actually written correctly (Issue #641).

Hopefully this will crash the worker immediately if the system is out of space to store the complete pidfile.

In addition, we now verify that existing pidfiles contain a new line so that a partially written pidfile is detected as broken, as before doing:

```
echo -n "1" > celeryd.pid
```

would cause celeryd to think that an existing instance was already running (init has pid 1 after all).

- Fixed 2.5 compatibility issue with use of print_exception.

  Fix contributed by Martin Melin.

- Fixed 2.5 compatibility issue with imports.

  Fix contributed by Iurii Kriachko.

- All programs now fix up `__package__` when called as main.

  This fixes compatibility with Python 2.5.

  Fix contributed by Martin Melin.

- celeryctl can now be configured on the command line.

  Like with celeryd it is now possible to configure celery settings on the command line for celeryctl:

  ```
  $ celeryctl -- broker.pool_limit=30
  ```

- Version dependency for python-dateutil fixed to be strict.

  Fix contributed by Thomas Meson.

- `Task.__call__` is now optimized away in the task tracer rather than when the task class is created.

  This fixes a bug where a custom __call__ may mysteriously disappear.

- Autoreload's inotify support has been improved.

  Contributed by Mher Movsisyan.

- The Django broker documentation has been improved.

- Removed confusing warning at top of routing user guide.

2.5.1

release-date 2012-03-01 01:00 P.M GMT

by Ask Solem

Fixes

- Eventlet/Gevent: A small typo caused celeryd to hang when eventlet/gevent was used, this was because the environment was not monkey patched early enough.
- Eventlet/Gevent: Another small typo caused the mediator to be started with eventlet/gevent, which would make celeryd sometimes hang at shutdown.
- Mulitprocessing: Fixed an error occurring if the pool was stopped before it was properly started.
- Proxy objects now redirects `__doc__` and `__name__` so `help(obj)` works.

- Internal timer (timer2) now logs exceptions instead of swallowing them (Issue #626).
- celeryctl shell: can now be started with `--eventlet` or `--gevent` options to apply their monkey patches.

2.5.0

release-date 2012-02-24 04:00 P.M GMT

by Ask Solem

See *What's new in Celery 2.5*.

Since the changelog has gained considerable size, we decided to do things differently this time: by having separate "what's new" documents for major version changes.

Bugfix releases will still be found in the changelog.

2.15.2 Change history for Celery 2.4

2.4.5

release-date 2011-12-02 05:00 P.M GMT

by Ask Solem

- Periodic task interval schedules were accidentally rounded down, resulting in some periodic tasks being executed early.
- Logging of humanized times in the celerybeat log is now more detailed.
- New *Brokers* section in the Getting Started part of the Documentation

  This replaces the old *tut-otherqueues* tutorial, and adds documentation for MongoDB, Beanstalk and CouchDB.

2.4.4

release-date 2011-11-25 16:00 P.M GMT

by Ask Solem

Security Fixes

- [Security: CELERYSA-0001] Daemons would set effective id's rather than real id's when the `--uid`/`--gid` arguments to **celeryd-multi**, **celeryd_detach**, **celerybeat** and **celeryev** were used.

  This means privileges weren't properly dropped, and that it would be possible to regain supervisor privileges later.

Fixes

- Processes pool: Fixed rare deadlock at shutdown (Issue #523).

  Fix contributed by Ionel Maries Christian.

- Webhook tasks issued the wrong HTTP POST headers (Issue #515).

  The *Content-Type* header has been changed from `application/json` `application/x-www-form-urlencoded`, and adds a proper *Content-Length* header.

  Fix contributed by Mitar.

- Daemonization tutorial: Adds a configuration example using Django and virtualenv together (Issue #505).

  Contributed by Juan Ignacio Catalano.

- generic init scripts now automatically creates log and pid file directories (Issue #545).

  Contributed by Chris Streeter.

2.4.3

release-date 2011-11-22 18:00 P.M GMT

by Ask Solem

- Fixes module import typo in *celeryctl* (Issue #538).

  Fix contributed by Chris Streeter.

2.4.2

release-date 2011-11-14 12:00 P.M GMT

by Ask Solem

- Program module no longer uses relative imports so that it is possible to do `python -m celery.bin.name`.

2.4.1

release-date 2011-11-07 06:00 P.M GMT

by Ask Solem

- celeryctl inspect commands was missing output.
- processes pool: Decrease polling interval for less idle CPU usage.
- processes pool: MaybeEncodingError was not wrapped in ExceptionInfo (Issue #524).
- celeryd: would silence errors occuring after task consumer started.

- logging: Fixed a bug where unicode in stdout redirected log messages couldn't be written (Issue #522).

2.4.0

release-date 2011-11-04 04:00 P.M GMT

by Ask Solem

Important Notes

- Now supports Python 3.
- Fixed deadlock in worker process handling (Issue #496).

  A deadlock could occur after spawning new child processes because the logging library's mutex was not properly reset after fork.

  The symptoms of this bug affecting would be that the worker simply stops processing tasks, as none of the workers child processes are functioning. There was a greater chance of this bug occurring with `maxtasksperchild` or a time-limit enabled.

  This is a workaround for http://bugs.python.org/issue6721#msg140215.

  Be aware that while this fixes the logging library lock, there could still be other locks initialized in the parent process, introduced by custom code.

  Fix contributed by Harm Verhagen.

- AMQP Result backend: Now expires results by default.

  The default expiration value is now taken from the `CELERY_TASK_RESULT_EXPIRES` setting.

  The old `CELERY_AMQP_TASK_RESULT_EXPIRES` setting has been deprecated and will be removed in version 4.0.

  Note that this means that the result backend requires RabbitMQ 1.1.0 or higher, and that you have to disable expiration if you are running with an older version. You can do so by disabling the `CELERY_TASK_RESULT_EXPIRES` setting:

  ```
  CELERY_TASK_RESULT_EXPIRES = None
  ```

- Eventlet: Fixed problem with shutdown (Issue #457).
- Broker transports can be now be specified using URLs

  The broker can now be specified as an URL instead. This URL must have the format:

  ```
  transport://user:password@hostname:port/virtual_host
  ```

  for example the default broker is written as:

  ```
  amqp://guest:guest@localhost:5672//
  ```

  The scheme is required, so that the host is identified as an URL and not just a host name. User, password, port and virtual_host are optional and defaults to the particular transports default value.

  Note: Note that the path component (virtual_host) always starts with a forward-slash. This is necessary to distinguish between the virtual host `''` (empty) and `'/'`, which are both acceptable virtual host names.

A virtual host of `'/'` becomes:

amqp://guest:guest@localhost:5672//

and a virtual host of `''` (empty) becomes:

```
amqp://guest:guest@localhost:5672/
```

So the leading slash in the path component is **always required**.

In addition the `BROKER_URL` setting has been added as an alias to `BROKER_HOST`. Any broker setting specified in both the URL and in the configuration will be ignored, if a setting is not provided in the URL then the value from the configuration will be used as default.

Also, programs now support the *`-b|--broker`* option to specify a broker URL on the command line:

```
$ celeryd -b redis://localhost

$ celeryctl -b amqp://guest:guest@localhost//e
```

The environment variable `CELERY_BROKER_URL` can also be used to easily override the default broker used.

- The deprecated `celery.loaders.setup_loader()` function has been removed.
- The `CELERY_TASK_ERROR_WHITELIST` setting has been replaced by a more flexible approach (Issue #447).

  The error mail sending logic is now available as `Task.ErrorMail`, with the implementation (for reference) in `celery.utils.mail`.

  The error mail class can be sub-classed to gain complete control of when error messages are sent, thus removing the need for a separate white-list setting.

  The `CELERY_TASK_ERROR_WHITELIST` setting has been deprecated, and will be removed completely in version 4.0.

- Additional Deprecations

  The following functions has been deprecated and is scheduled for removal in version 4.0:

Old function	Alternative
celery.loaders.current_loader	*celery.current_app.loader*
celery.loaders.load_settings	*celery.current_app.conf*
celery.execute.apply	*Task.apply*
celery.execute.apply_async	*Task.apply_async*
celery.execute.delay_task	*celery.execute.send_task*

  The following settings has been deprecated and is scheduled for removal in version 4.0:

Old setting	Alternative
CELERYD_LOG_LEVEL	`celeryd --loglevel=`
CELERYD_LOG_FILE	`celeryd --logfile=`
CELERYBEAT_LOG_LEVEL	`celerybeat --loglevel=`
CELERYBEAT_LOG_FILE	`celerybeat --logfile=`
CELERYMON_LOG_LEVEL	`celerymon --loglevel=`
CELERYMON_LOG_FILE	`celerymon --logfile=`

News

- No longer depends on `pyparsing`.
- Now depends on Kombu 1.4.3.
- CELERY_IMPORTS can now be a scalar value (Issue #485).

  It is too easy to forget to add the comma after the sole element of a tuple, and this is something that often affects newcomers.

  The docs should probably use a list in examples, as using a tuple for this doesn't even make sense. Nonetheless, there are many tutorials out there using a tuple, and this change should be a help to new users.

  Suggested by jsaxon-cars.
- Fixed a memory leak when using the thread pool (Issue #486).

  Contributed by Kornelijus Survila.
- The statedb was not saved at exit.

  This has now been fixed and it should again remember previously revoked tasks when a `--statedb` is enabled.
- Adds `EMAIL_USE_TLS` to enable secure SMTP connections (Issue #418).

  Contributed by Stefan Kjartansson.
- Now handles missing fields in task messages as documented in the message format documentation.
  - Missing required field throws `InvalidTaskError`
  - Missing args/kwargs is assumed empty.

  Contributed by Chris Chamberlin.
- Fixed race condition in celery.events.state (celerymon/celeryev) where task info would be removed while iterating over it (Issue #501).
- The Cache, Cassandra, MongoDB, Redis and Tyrant backends now respects the `CELERY_RESULT_SERIALIZER` setting (Issue #435).

  This means that only the database (django/sqlalchemy) backends currently does not support using custom serializers.

  Contributed by Steeve Morin
- Logging calls no longer manually formats messages, but delegates that to the logging system, so tools like Sentry can easier work with the messages (Issue #445).

  Contributed by Chris Adams.
- `celeryd_multi` now supports a `stop_verify` command to wait for processes to shutdown.
- Cache backend did not work if the cache key was unicode (Issue #504).

  Fix contributed by Neil Chintomby.
- New setting `CELERY_RESULT_DB_SHORT_LIVED_SESSIONS` added, which if enabled will disable the caching of SQLAlchemy sessions (Issue #449).

  Contributed by Leo Dirac.
- All result backends now implements `__reduce__` so that they can be pickled (Issue #441).

  Fix contributed by Remy Noel

- celeryd-multi did not work on Windows (Issue #472).
- New-style `CELERY_REDIS_*` settings now takes precedence over the old `REDIS_*` configuration keys (Issue #508).

  Fix contributed by Joshua Ginsberg
- Generic celerybeat init script no longer sets *bash -e* (Issue #510).

  Fix contributed by Roger Hu.
- Documented that Chords do not work well with redis-server versions before 2.2.

  Contributed by Dan McGee.
- The `CELERYBEAT_MAX_LOOP_INTERVAL` setting was not respected.
- `inspect.registered_tasks` renamed to `inspect.registered` for naming consistency.

  The previous name is still available as an alias.

  Contributed by Mher Movsisyan
- Worker logged the string representation of args and kwargs without safe guards (Issue #480).
- RHEL init script: Changed celeryd startup priority.

  The default start / stop priorities for MySQL on RHEL are

  ```
  # chkconfig: - 64 36
  ```

  Therefore, if Celery is using a database as a broker / message store, it should be started after the database is up and running, otherwise errors will ensue. This commit changes the priority in the init script to

  ```
  # chkconfig: - 85 15
  ```

  which are the default recommended settings for 3-rd party applications and assure that Celery will be started after the database service & shut down before it terminates.

  Contributed by Yury V. Zaytsev.
- KeyValueStoreBackend.get_many did not respect the `timeout` argument (Issue #512).
- celerybeat/celeryev's –workdir option did not chdir before after configuration was attempted (Issue #506).
- After deprecating 2.4 support we can now name modules correctly, since we can take use of absolute imports.

  Therefore the following internal modules have been renamed:

  celery.concurrency.evlet -> celery.concurrency.eventlet celery.concurrency.evg -> celery.concurrency.gevent
- AUTHORS file is now sorted alphabetically.

  Also, as you may have noticed the contributors of new features/fixes are now mentioned in the Changelog.

2.15.3 Change history for Celery 2.3

2.3.4

release-date 2011-11-25 16:00 P.M GMT

by Ask Solem

Security Fixes

- [Security: CELERYSA-0001] Daemons would set effective id's rather than real id's when the `--uid`/`--gid` arguments to **celeryd-multi**, **celeryd_detach**, **celerybeat** and **celeryev** were used.

  This means privileges weren't properly dropped, and that it would be possible to regain supervisor privileges later.

Fixes

- Backported fix for #455 from 2.4 to 2.3.
- Statedb was not saved at shutdown.
- Fixes worker sometimes hanging when hard time limit exceeded.

2.3.3

release-date 2011-16-09 05:00 P.M BST

by Mher Movsisyan

- Monkey patching `sys.stdout` could result in the worker crashing if the replacing object did not define `isatty()` (Issue #477).
- `CELERYD` option in `/etc/default/celeryd` should not be used with generic init scripts.

2.3.2

release-date 2011-10-07 05:00 P.M BST

News

- Improved Contributing guide.

  If you'd like to contribute to Celery you should read the *Contributing Gudie*.

  We are looking for contributors at all skill levels, so don't hesitate!
- Now depends on Kombu 1.3.1
- `Task.request` now contains the current worker host name (Issue #460).

  Available as `task.request.hostname`.
- **It is now easier for app subclasses to extend how they are pickled.** (see `celery.app.AppPickler`).

Fixes

- *purge/discard_all* was not working correctly (Issue #455).
- The coloring of log messages didn't handle non-ASCII data well (Issue #427).
- [Windows] the multiprocessing pool tried to import `os.kill` even though this is not available there (Issue #450).
- Fixes case where the worker could become unresponsive because of tasks exceeding the hard time limit.
- The `task-sent` event was missing from the event reference.
- `ResultSet.iterate` now returns results as they finish (Issue #459).

  This was not the case previously, even though the documentation states this was the expected behavior.
- Retries will no longer be performed when tasks are called directly (using `__call__`).

  Instead the exception passed to `retry` will be re-raised.
- Eventlet no longer crashes if autoscale is enabled.

  growing and shrinking eventlet pools is still not supported.
- py24 target removed from `tox.ini`.

2.3.1

release-date 2011-08-07 08:00 P.M BST

Fixes

- The `CELERY_AMQP_TASK_RESULT_EXPIRES` setting did not work, resulting in an AMQP related error about not being able to serialize floats while trying to publish task states (Issue #446).

2.3.0

release-date 2011-08-05 12:00 P.M BST

tested cPython: 2.5, 2.6, 2.7; PyPy: 1.5; Jython: 2.5.2

Important Notes

- Now requires Kombu 1.2.1
- Results are now disabled by default.

  The AMQP backend was not a good default because often the users were not consuming the results, resulting in thousands of queues.

  While the queues can be configured to expire if left unused, it was not possible to enable this by default because this was only available in recent RabbitMQ versions (2.1.1+)

  With this change enabling a result backend will be a conscious choice, which will hopefully lead the user to read the documentation and be aware of any common pitfalls with the particular backend.

  The default backend is now a dummy backend (`celery.backends.base.DisabledBackend`). Saving state is simply an noop operation, and AsyncResult.wait(), .result, .state, etc. will raise a `NotImplementedError` telling the user to configure the result backend.

  For help choosing a backend please see *Result Backends*.

  If you depend on the previous default which was the AMQP backend, then you have to set this explicitly before upgrading:

  ```
  CELERY_RESULT_BACKEND = "amqp"
  ```

  Note: For django-celery users the default backend is still `database`, and results are not disabled by default.

- The Debian init scripts have been deprecated in favor of the generic-init.d init scripts.

  In addition generic init scripts for celerybeat and celeryev has been added.

News

- Automatic connection pool support.

  The pool is used by everything that requires a broker connection. For example calling tasks, sending broadcast commands, retrieving results with the AMQP result backend, and so on.

  The pool is disabled by default, but you can enable it by configuring the `BROKER_POOL_LIMIT` setting:

  ```
  BROKER_POOL_LIMIT = 10
  ```

  A limit of 10 means a maximum of 10 simultaneous connections can co-exist. Only a single connection will ever be used in a single-thread environment, but in a concurrent environment (threads, greenlets, etc., but not processes) when the limit has been exceeded, any try to acquire a connection will block the thread and wait for a connection to be released. This is something to take into consideration when choosing a limit.

  A limit of `None` or 0 means no limit, and connections will be established and closed every time.

- Introducing Chords (taskset callbacks).

  A chord is a task that only executes after all of the tasks in a taskset has finished executing. It's a fancy term for "taskset callbacks" adopted from Cω).

It works with all result backends, but the best implementation is currently provided by the Redis result backend.

Here's an example chord:

```
>>> chord(add.subtask((i, i))
...         for i in xrange(100))(tsum.subtask()).get()
9900
```

Please read the *Chords section in the user guide*, if you want to know more.

- Time limits can now be set for individual tasks.

  To set the soft and hard time limits for a task use the `time_limit` and `soft_time_limit` attributes:

```
import time

@task(time_limit=60, soft_time_limit=30)
def sleeptask(seconds):
    time.sleep(seconds)
```

  If the attributes are not set, then the workers default time limits will be used.

  New in this version you can also change the time limits for a task at runtime using the `time_limit()` remote control command:

```
>>> from celery.task import control
>>> control.time_limit("tasks.sleeptask",
...                    soft=60, hard=120, reply=True)
[{'worker1.example.com': {'ok': 'time limits set successfully'}}]
```

  Only tasks that starts executing after the time limit change will be affected.

Note: Soft time limits will still not work on Windows or other platforms that do not have the `SIGUSR1` signal.

- **Redis backend configuration directive names changed to include the** `CELERY_` prefix.

Old setting name	Replace with
REDIS_HOST	*CELERY_REDIS_HOST*
REDIS_PORT	*CELERY_REDIS_PORT*
REDIS_DB	*CELERY_REDIS_DB*
REDIS_PASSWORD	*CELERY_REDIS_PASSWORD*

  The old names are still supported but pending deprecation.

- PyPy: The default pool implementation used is now multiprocessing if running on PyPy 1.5.

- celeryd-multi: now supports "pass through" options.

  Pass through options makes it easier to use celery without a configuration file, or just add last-minute options on the command line.

  Example use:

```
$ celeryd-multi start 4  -c 2  -- broker.host=amqp.example.com \
                                  broker.vhost=/               \
                                  celery.disable_rate_limits=yes
```

- celerybeat: Now retries establishing the connection (Issue #419).
- celeryctl: New `list bindings` command.

  Lists the current or all available bindings, depending on the broker transport used.
- Heartbeat is now sent every 30 seconds (previously every 2 minutes).
- `ResultSet.join_native()` and `iter_native()` is now supported by the Redis and Cache result backends.

  This is an optimized version of `join()` using the underlying backends ability to fetch multiple results at once.
- Can now use SSL when sending error e-mails by enabling the `EMAIL_USE_SSL` setting.
- `events.default_dispatcher()`: Context manager to easily obtain an event dispatcher instance using the connection pool.
- Import errors in the configuration module will not be silenced anymore.
- ResultSet.iterate: Now supports the `timeout`, `propagate` and `interval` arguments.
- `with_default_connection` -> `with default_connection`
- TaskPool.apply_async: Keyword arguments `callbacks` and `errbacks` has been renamed to `callback` and `errback` and take a single scalar value instead of a list.
- No longer propagates errors occurring during process cleanup (Issue #365)
- Added `TaskSetResult.delete()`, which will delete a previously saved taskset result.
- Celerybeat now syncs every 3 minutes instead of only at shutdown (Issue #382).
- Monitors now properly handles unknown events, so user-defined events are displayed.
- Terminating a task on Windows now also terminates all of the tasks child processes (Issue #384).
- celeryd: `-I|--include` option now always searches the current directory to import the specified modules.
- Cassandra backend: Now expires results by using TTLs.
- Functional test suite in `funtests` is now actually working properly, and passing tests.

Fixes

- celeryev was trying to create the pidfile twice.
- celery.contrib.batches: Fixed problem where tasks failed silently (Issue #393).
- Fixed an issue where logging objects would give "<Unrepresentable", even though the objects were.
- `CELERY_TASK_ERROR_WHITE_LIST` is now properly initialized in all loaders.
- celeryd_detach now passes through command line configuration.
- Remote control command `add_consumer` now does nothing if the queue is already being consumed from.

2.15.4 Change history for Celery 2.2

2.2.8

release-date 2011-11-25 16:00 P.M GMT

by Ask Solem

Security Fixes

- [Security: CELERYSA-0001] Daemons would set effective id's rather than real id's when the `--uid`/`--gid` arguments to **celeryd-multi**, **celeryd_detach**, **celerybeat** and **celeryev** were used.

  This means privileges weren't properly dropped, and that it would be possible to regain supervisor privileges later.

2.2.7

release-date 2011-06-13 16:00 P.M BST

- New signals: `after_setup_logger` and `after_setup_task_logger`

  These signals can be used to augment logging configuration after Celery has set up logging.

- Redis result backend now works with Redis 2.4.4.
- celeryd_multi: The `--gid` option now works correctly.
- celeryd: Retry wrongfully used the repr of the traceback instead of the string representation.
- App.config_from_object: Now loads module, not attribute of module.

- Fixed issue where logging of objects would give "<Unrepresentable: ...>"

2.2.6

release-date 2011-04-15 16:00 P.M CEST

Important Notes

- Now depends on Kombu 1.1.2.
- Dependency lists now explicitly specifies that we don't want python-dateutil 2.x, as this version only supports py3k.

  If you have installed dateutil 2.0 by accident you should downgrade to the 1.5.0 version:

  ```
  pip install -U python-dateutil==1.5.0
  ```

  or by easy_install:

  ```
  easy_install -U python-dateutil==1.5.0
  ```

Fixes

- The new `WatchedFileHandler` broke Python 2.5 support (Issue #367).
- Task: Don't use `app.main` if the task name is set explicitly.
- Sending emails did not work on Python 2.5, due to a bug in the version detection code (Issue #378).
- Beat: Adds method `ScheduleEntry._default_now`

  This method can be overridden to change the default value of `last_run_at`.
- An error occurring in process cleanup could mask task errors.

  We no longer propagate errors happening at process cleanup, but log them instead. This way they will not interfere with publishing the task result (Issue #365).
- Defining tasks did not work properly when using the Django `shell_plus` utility (Issue #366).
- **`AsyncResult.get` did not accept the `interval` and `propagate`** arguments.
- **celeryd: Fixed a bug where celeryd would not shutdown if a** `socket.error` was raised.

2.2.5

release-date 2011-03-28 06:00 P.M CEST

Important Notes

- Now depends on Kombu 1.0.7

News

- Our documentation is now hosted by Read The Docs (http://docs.celeryproject.org), and all links have been changed to point to the new URL.
- Logging: Now supports log rotation using external tools like logrotate.d (Issue #321)

  This is accomplished by using the `WatchedFileHandler`, which re-opens the file if it is renamed or deleted.
- ***tut-otherqueues* now documents how to configure Redis/Database result** backends.
- gevent: Now supports ETA tasks.

  But gevent still needs `CELERY_DISABLE_RATE_LIMITS=True` to work.
- TaskSet User Guide: now contains TaskSet callback recipes.
- Eventlet: New signals:
  - `eventlet_pool_started`
  - `eventlet_pool_preshutdown`
  - `eventlet_pool_postshutdown`
  - `eventlet_pool_apply`

  See `celery.signals` for more information.
- New `BROKER_TRANSPORT_OPTIONS` setting can be used to pass additional arguments to a particular broker transport.
- celeryd: `worker_pid` is now part of the request info as returned by broadcast commands.
- TaskSet.apply/Taskset.apply_async now accepts an optional `taskset_id` argument.
- The taskset_id (if any) is now available in the Task request context.
- SQLAlchemy result backend: taskset_id and taskset_id columns now have a unique constraint. (Tables need to recreated for this to take affect).
- Task Userguide: Added section about choosing a result backend.
- Removed unused attribute `AsyncResult.uuid`.

Fixes

- multiprocessing.Pool: Fixes race condition when marking job with `WorkerLostError` (Issue #268).

  The process may have published a result before it was terminated, but we have no reliable way to detect that this is the case.

  So we have to wait for 10 seconds before marking the result with WorkerLostError. This gives the result handler a chance to retrieve the result.
- multiprocessing.Pool: Shutdown could hang if rate limits disabled.

  There was a race condition when the MainThread was waiting for the pool semaphore to be released. The ResultHandler now terminates after 5 seconds if there are unacked jobs, but no worker processes left to start them (it needs to timeout because there could still be an ack+result that we haven't consumed from the result queue. It is unlikely we will receive any after 5 seconds with no worker processes).
- celerybeat: Now creates pidfile even if the `--detach` option is not set.

- eventlet/gevent: The broadcast command consumer is now running in a separate greenthread.

  This ensures broadcast commands will take priority even if there are many active tasks.

- Internal module `celery.worker.controllers` renamed to `celery.worker.mediator`.
- celeryd: Threads now terminates the program by calling `os._exit`, as it is the only way to ensure exit in the case of syntax errors, or other unrecoverable errors.
- Fixed typo in `maybe_timedelta` (Issue #352).
- celeryd: Broadcast commands now logs with loglevel debug instead of warning.
- AMQP Result Backend: Now resets cached channel if the connection is lost.
- Polling results with the AMQP result backend was not working properly.
- Rate limits: No longer sleeps if there are no tasks, but rather waits for the task received condition (Performance improvement).
- ConfigurationView: `iter(dict)` should return keys, not items (Issue #362).
- celerybeat: PersistentScheduler now automatically removes a corrupted schedule file (Issue #346).
- Programs that doesn't support positional command line arguments now provides a user friendly error message.
- Programs no longer tries to load the configuration file when showing `--version` (Issue #347).
- Autoscaler: The "all processes busy" log message is now severity debug instead of error.
- celeryd: If the message body can't be decoded, it is now passed through `safe_str` when logging.

  This to ensure we don't get additional decoding errors when trying to log the failure.

- `app.config_from_object`/`app.config_from_envvar` now works for all loaders.
- Now emits a user-friendly error message if the result backend name is unknown (Issue #349).
- `celery.contrib.batches`: Now sets loglevel and logfile in the task request so `task.get_logger` works with batch tasks (Issue #357).
- celeryd: An exception was raised if using the amqp transport and the prefetch count value exceeded 65535 (Issue #359).

  The prefetch count is incremented for every received task with an ETA/countdown defined. The prefetch count is a short, so can only support a maximum value of 65535. If the value exceeds the maximum value we now disable the prefetch count, it is re-enabled as soon as the value is below the limit again.

- cursesmon: Fixed unbound local error (Issue #303).
- eventlet/gevent is now imported on demand so autodoc can import the modules without having eventlet/gevent installed.
- celeryd: Ack callback now properly handles `AttributeError`.
- `Task.after_return` is now always called *after* the result has been written.
- Cassandra Result Backend: Should now work with the latest `pycassa` version.
- multiprocessing.Pool: No longer cares if the putlock semaphore is released too many times. (this can happen if one or more worker processes are killed).
- SQLAlchemy Result Backend: Now returns accidentally removed `date_done` again (Issue #325).
- Task.request contex is now always initialized to ensure calling the task function directly works even if it actively uses the request context.

- Exception occuring when iterating over the result from `TaskSet.apply` fixed.
- eventlet: Now properly schedules tasks with an ETA in the past.

2.2.4

release-date 2011-02-19 12:00 AM CET

Fixes

- celeryd: 2.2.3 broke error logging, resulting in tracebacks not being logged.
- AMQP result backend: Polling task states did not work properly if there were more than one result message in the queue.
- `TaskSet.apply_async()` and `TaskSet.apply()` now supports an optional `taskset_id` keyword argument (Issue #331).
- The current taskset id (if any) is now available in the task context as `request.taskset` (Issue #329).
- SQLAlchemy result backend: *date_done* was no longer part of the results as it had been accidentally removed. It is now available again (Issue #325).
- SQLAlchemy result backend: Added unique constraint on *Task.id* and *TaskSet.taskset_id*. Tables needs to be recreated for this to take effect.
- Fixed exception raised when iterating on the result of `TaskSet.apply()`.
- Tasks Userguide: Added section on choosing a result backend.

2.2.3

release-date 2011-02-12 04:00 P.M CET

Fixes

- Now depends on Kombu 1.0.3
- Task.retry now supports a `max_retries` argument, used to change the default value.
- *multiprocessing.cpu_count* may raise `NotImplementedError` on platforms where this is not supported (Issue #320).
- Coloring of log messages broke if the logged object was not a string.
- Fixed several typos in the init script documentation.
- A regression caused *Task.exchange* and *Task.routing_key* to no longer have any effect. This is now fixed.
- Routing Userguide: Fixes typo, routers in `CELERY_ROUTES` must be instances, not classes.
- **celeryev** did not create pidfile even though the *`--pidfile`* argument was set.
- Task logger format was no longer used. (Issue #317).

  The id and name of the task is now part of the log message again.
- A safe version of `repr()` is now used in strategic places to ensure objects with a broken `__repr__` does not crash the worker, or otherwise make errors hard to understand (Issue #298).

- Remote control command `active_queues`: did not account for queues added at runtime.

  In addition the dictionary replied by this command now has a different structure: the exchange key is now a dictionary containing the exchange declaration in full.

- The `-Q` option to **celeryd** removed unused queue declarations, so routing of tasks could fail.

  Queues are no longer removed, but rather *app.amqp.queues.consume_from()* is used as the list of queues to consume from.

  This ensures all queues are available for routing purposes.

- celeryctl: Now supports the *inspect active_queues* command.

2.2.2

release-date 2011-02-03 04:00 P.M CET

Fixes

- Celerybeat could not read the schedule properly, so entries in `CELERYBEAT_SCHEDULE` would not be scheduled.
- Task error log message now includes *exc_info* again.
- The *eta* argument can now be used with *task.retry*.

  Previously it was overwritten by the countdown argument.

- celeryd-multi/celeryd_detach: Now logs errors occuring when executing the *celeryd* command.
- daemonizing tutorial: Fixed typo `--time-limit 300` -> `--time-limit=300`
- Colors in logging broke non-string objects in log messages.
- `setup_task_logger` no longer makes assumptions about magic task kwargs.

2.2.1

release-date 2011-02-02 04:00 P.M CET

Fixes

- Eventlet pool was leaking memory (Issue #308).
- Deprecated function `celery.execute.delay_task` was accidentally removed, now available again.
- `BasePool.on_terminate` stub did not exist
- **celeryd detach: Adds readable error messages if user/group name does not** exist.
- Smarter handling of unicode decod errors when logging errors.

2.2.0

release-date 2011-02-01 10:00 AM CET

Important Notes

- Carrot has been replaced with Kombu

  Kombu is the next generation messaging framework for Python, fixing several flaws present in Carrot that was hard to fix without breaking backwards compatibility.

  Also it adds:

  - First-class support for virtual transports; Redis, Django ORM, SQLAlchemy, Beanstalk, MongoDB, CouchDB and in-memory.
  - Consistent error handling with introspection,
  - The ability to ensure that an operation is performed by gracefully handling connection and channel errors,
  - Message compression (zlib, bzip2, or custom compression schemes).

  This means that *ghettoq* is no longer needed as the functionality it provided is already available in Celery by default. The virtual transports are also more feature complete with support for exchanges (direct and topic). The Redis transport even supports fanout exchanges so it is able to perform worker remote control commands.

- Magic keyword arguments pending deprecation.

  The magic keyword arguments were responsibile for many problems and quirks: notably issues with tasks and decorators, and name collisions in keyword arguments for the unaware.

  It wasn't easy to find a way to deprecate the magic keyword arguments, but we think this is a solution that makes sense and it will not have any adverse effects for existing code.

  The path to a magic keyword argument free world is:

  - the *celery.decorators* module is deprecated and the decorators can now be found in *celery.task*.
  - The decorators in *celery.task* disables keyword arguments by default
  - All examples in the documentation have been changed to use *celery.task*.

  This means that the following will have magic keyword arguments enabled (old style):

  ```
  from celery.decorators import task

  @task()
  def add(x, y, **kwargs):
      print("In task %s" % kwargs["task_id"])
      return x + y
  ```

  And this will not use magic keyword arguments (new style):

  ```
  from celery.task import task

  @task()
  def add(x, y):
      print("In task %s" % add.request.id)
      return x + y
  ```

  In addition, tasks can choose not to accept magic keyword arguments by setting the *task.accept_magic_kwargs* attribute.

Deprecation

Using the decorators in `celery.decorators` emits a `PendingDeprecationWarning` with a helpful message urging you to change your code, in version 2.4 this will be replaced with a `DeprecationWarning`, and in version 4.0 the `celery.decorators` module will be removed and no longer exist.

Similarly, the *task.accept_magic_kwargs* attribute will no longer have any effect starting from version 4.0.

- The magic keyword arguments are now available as *task.request*

  This is called *the context*. Using thread-local storage the context contains state that is related to the current request.

  It is mutable and you can add custom attributes that will only be seen by the current task request.

  The following context attributes are always available:

Magic Keyword Argument	Replace with
kwargs["task_id"]	*self.request.id*
kwargs["delivery_info"]	*self.request.delivery_info*
kwargs["task_retries"]	*self.request.retries*
kwargs["logfile"]	*self.request.logfile*
kwargs["loglevel"]	*self.request.loglevel*
kwargs["task_is_eager	*self.request.is_eager*
NEW	*self.request.args*
NEW	*self.request.kwargs*

  In addition, the following methods now automatically uses the current context, so you don't have to pass *kwargs* manually anymore:

  - *task.retry*
  - *task.get_logger*
  - *task.update_state*

- Eventlet support.

  This is great news for I/O-bound tasks!

  To change pool implementations you use the `-P|--pool` argument to **celeryd**, or globally using the `CELERYD_POOL` setting. This can be the full name of a class, or one of the following aliases: *processes*, *eventlet*, *gevent*.

  For more information please see the *Concurrency with Eventlet* section in the User Guide.

  Why not gevent?

  For our first alternative concurrency implementation we have focused on Eventlet, but there is also an experimental gevent pool available. This is missing some features, notably the ability to schedule ETA tasks.

  Hopefully the gevent support will be feature complete by version 2.3, but this depends on user demand (and contributions).

- Python 2.4 support deprecated!

  We're happy^H^H^H^H^Hsad to announce that this is the last version to support Python 2.4.

You are urged to make some noise if you're currently stuck with Python 2.4. Complain to your package maintainers, sysadmins and bosses: tell them it's time to move on!

Apart from wanting to take advantage of with-statements, coroutines, conditional expressions and enhanced try blocks, the code base now contains so many 2.4 related hacks and workarounds it's no longer just a compromise, but a sacrifice.

If it really isn't your choice, and you don't have the option to upgrade to a newer version of Python, you can just continue to use Celery 2.2. Important fixes can be backported for as long as there is interest.

- *celeryd*: Now supports Autoscaling of child worker processes.

  The `--autoscale` option can be used to configure the minimum and maximum number of child worker processes:

```
--autoscale=AUTOSCALE
     Enable autoscaling by providing
     max_concurrency,min_concurrency.  Example:
       --autoscale=10,3 (always keep 3 processes, but grow to
      10 if necessary).
```

- Remote Debugging of Tasks

  `celery.contrib.rdb` is an extended version of pdb that enables remote debugging of processes that does not have terminal access.

  Example usage:

```
    from celery.contrib import rdb
    from celery.task import task

    @task()
    def add(x, y):
        result = x + y
        rdb.set_trace()  # <- set breakpoint
        return result


:func:`~celery.contrib.rdb.set_trace` sets a breakpoint at the current
location and creates a socket you can telnet into to remotely debug
your task.

The debugger may be started by multiple processes at the same time,
so rather than using a fixed port the debugger will search for an
available port, starting from the base port (6900 by default).
The base port can be changed using the environment variable
:envvar:`CELERY_RDB_PORT`.

By default the debugger will only be available from the local host,
to enable access from the outside you have to set the environment
variable :envvar:`CELERY_RDB_HOST`.

When `celeryd` encounters your breakpoint it will log the following
information::

    [INFO/MainProcess] Got task from broker:
        tasks.add[d7261c71-4962-47e5-b342-2448bedd20e8]
    [WARNING/PoolWorker-1] Remote Debugger:6900:
        Please telnet 127.0.0.1 6900.  Type `exit` in session to continue.
```

```
    [2011-01-18 14:25:44,119: WARNING/PoolWorker-1] Remote Debugger:6900:
        Waiting for client...

If you telnet the port specified you will be presented
with a ``pdb`` shell:

.. code-block:: bash

    $ telnet localhost 6900
    Connected to localhost.
    Escape character is '^]'.
    > /opt/devel/demoapp/tasks.py(128)add()
    -> return result
    (Pdb)

Enter ``help`` to get a list of available commands,
It may be a good idea to read the `Python Debugger Manual`_ if
you have never used `pdb` before.
```

- Events are now transient and is using a topic exchange (instead of direct).

  The *CELERYD_EVENT_EXCHANGE*, *CELERYD_EVENT_ROUTING_KEY*, *CELERYD_EVENT_EXCHANGE_TYPE* settings are no longer in use.

  This means events will not be stored until there is a consumer, and the events will be gone as soon as the consumer stops. Also it means there can be multiple monitors running at the same time.

  The routing key of an event is the type of event (e.g. *worker.started*, *worker.heartbeat*, *task.succeeded*, etc. This means a consumer can filter on specific types, to only be alerted of the events it cares about.

  Each consumer will create a unique queue, meaning it is in effect a broadcast exchange.

  This opens up a lot of possibilities, for example the workers could listen for worker events to know what workers are in the neighborhood, and even restart workers when they go down (or use this information to optimize tasks/autoscaling).

  Note: The event exchange has been renamed from "celeryevent" to "celeryev" so it does not collide with older versions.

  If you would like to remove the old exchange you can do so by executing the following command:

  ```
  $ camqadm exchange.delete celeryevent
  ```

- *celeryd* now starts without configuration, and configuration can be specified directly on the command line.

  Configuration options must appear after the last argument, separated by two dashes:

  ```
  $ celeryd -l info -I tasks -- broker.host=localhost broker.vhost=/app
  ```

- Configuration is now an alias to the original configuration, so changes to the original will reflect Celery at runtime.

- *celery.conf* has been deprecated, and modifying *celery.conf.ALWAYS_EAGER* will no longer have any effect.

  The default configuration is now available in the `celery.app.defaults` module. The available configuration options and their types can now be introspected.

- Remote control commands are now provided by *kombu.pidbox*, the generic process mailbox.

- Internal module *celery.worker.listener* has been renamed to *celery.worker.consumer*, and *.CarrotListener* is now *.Consumer*.
- Previously deprecated modules *celery.models* and *celery.management.commands* have now been removed as per the deprecation timeline.
- **[Security: Low severity] Removed *celery.task.RemoteExecuteTask* and** accompanying functions: *dmap*, *dmap_async*, and *execute_remote*.

  Executing arbitrary code using pickle is a potential security issue if someone gains unrestricted access to the message broker.

  If you really need this functionality, then you would have to add this to your own project.
- [Security: Low severity] The *stats* command no longer transmits the broker password.

  One would have needed an authenticated broker connection to receive this password in the first place, but sniffing the password at the wire level would have been possible if using unencrypted communication.

News

- The internal module *celery.task.builtins* has been removed.
- The module *celery.task.schedules* is deprecated, and *celery.schedules* should be used instead.

  For example if you have:

  ```
  from celery.task.schedules import crontab
  ```

  You should replace that with:

  ```
  from celery.schedules import crontab
  ```

  The module needs to be renamed because it must be possible to import schedules without importing the *celery.task* module.
- The following functions have been deprecated and is scheduled for removal in version 2.3:
  - *celery.execute.apply_async*

    Use *task.apply_async()* instead.
  - *celery.execute.apply*

    Use *task.apply()* instead.
  - *celery.execute.delay_task*

    Use *registry.tasks[name].delay()* instead.
- Importing *TaskSet* from *celery.task.base* is now deprecated.

  You should use:

  ```
  >>> from celery.task import TaskSet
  ```

  instead.
- New remote control commands:
  - *active_queues*

Returns the queue declarations a worker is currently consuming from.

- Added the ability to retry publishing the task message in the event of connection loss or failure.

  This is disabled by default but can be enabled using the `CELERY_TASK_PUBLISH_RETRY` setting, and tweaked by the `CELERY_TASK_PUBLISH_RETRY_POLICY` setting.

  In addition *retry*, and *retry_policy* keyword arguments have been added to *Task.apply_async*.

  Note: Using the *retry* argument to *apply_async* requires you to handle the publisher/connection manually.

- Periodic Task classes (*@periodic_task*/*PeriodicTask*) will *not* be deprecated as previously indicated in the source code.

  But you are encouraged to use the more flexible `CELERYBEAT_SCHEDULE` setting.

- Built-in daemonization support of celeryd using *celeryd-multi* is no longer experimental and is considered production quality.

  See *Generic init scripts* if you want to use the new generic init scripts.

- Added support for message compression using the `CELERY_MESSAGE_COMPRESSION` setting, or the *compression* argument to *apply_async*. This can also be set using routers.

- ***celeryd*: Now logs stacktrace of all threads when receiving the** *SIGUSR1* signal. (Does not work on cPython 2.4, Windows or Jython).

  Inspired by https://gist.github.com/737056

- Can now remotely terminate/kill the worker process currently processing a task.

  The *revoke* remote control command now supports a *terminate* argument Default signal is *TERM*, but can be specified using the *signal* argument. Signal can be the uppercase name of any signal defined in the `signal` module in the Python Standard Library.

  Terminating a task also revokes it.

  Example:

```
>>> from celery.task.control import revoke

>>> revoke(task_id, terminate=True)
>>> revoke(task_id, terminate=True, signal="KILL")
>>> revoke(task_id, terminate=True, signal="SIGKILL")
```

- *TaskSetResult.join_native*: Backend-optimized version of *join()*.

  If available, this version uses the backends ability to retrieve multiple results at once, unlike *join()* which fetches the results one by one.

  So far only supported by the AMQP result backend. Support for memcached and Redis may be added later.

- Improved implementations of *TaskSetResult.join* and *AsyncResult.wait*.

  An *interval* keyword argument have been added to both so the polling interval can be specified (default interval is 0.5 seconds).

  A *propagate* keyword argument have been added to *result.wait()*, errors will be returned instead of raised if this is set to False.

> **Warning:** You should decrease the polling interval when using the database result backend, as frequent polling can result in high database load.

- The PID of the child worker process accepting a task is now sent as a field with the `task-started` event.
- The following fields have been added to all events in the worker class:
  - *sw_ident*: Name of worker software (e.g. celeryd).
  - *sw_ver*: Software version (e.g. 2.2.0).
  - *sw_sys*: Operating System (e.g. Linux, Windows, Darwin).
- For better accuracy the start time reported by the multiprocessing worker process is used when calculating task duration.

  Previously the time reported by the accept callback was used.
- ***celerybeat*: New built-in daemonization support using the *–detach*** option.
- ***celeryev*: New built-in daemonization support using the *–detach*** option.
- *TaskSet.apply_async*: Now supports custom publishers by using the *publisher* argument.
- Added `CELERY_SEND_TASK_SENT_EVENT` setting.

  If enabled an event will be sent with every task, so monitors can track tasks before the workers receive them.
- ***celerybeat*: Now reuses the broker connection when calling** scheduled tasks.
- The configuration module and loader to use can now be specified on the command line.

  For example:

```
$ celeryd --config=celeryconfig.py --loader=myloader.Loader
```

- Added signals: *beat_init* and *beat_embedded_init*
  - `celery.signals.beat_init`

    Dispatched when **celerybeat** starts (either standalone or embedded). Sender is the `celery.beat.Service` instance.
  - `celery.signals.beat_embedded_init`

    Dispatched in addition to the `beat_init` signal when **celerybeat** is started as an embedded process. Sender is the `celery.beat.Service` instance.
- Redis result backend: Removed deprecated settings *REDIS_TIMEOUT* and *REDIS_CONNECT_RETRY*.
- CentOS init script for **celeryd** now available in *extra/centos*.
- Now depends on *pyparsing* version 1.5.0 or higher.

  There have been reported issues using Celery with pyparsing 1.4.x, so please upgrade to the latest version.
- Lots of new unit tests written, now with a total coverage of 95%.

Fixes

- *celeryev* Curses Monitor: Improved resize handling and UI layout (Issue #274 + Issue #276)
- AMQP Backend: Exceptions occurring while sending task results are now propagated instead of silenced.

celeryd will then show the full traceback of these errors in the log.

- AMQP Backend: No longer deletes the result queue after successful poll, as this should be handled by the `CELERY_AMQP_TASK_RESULT_EXPIRES` setting instead.
- AMQP Backend: Now ensures queues are declared before polling results.
- Windows: celeryd: Show error if running with *-B* option.

  Running celerybeat embedded is known not to work on Windows, so users are encouraged to run celerybeat as a separate service instead.
- Windows: Utilities no longer output ANSI color codes on Windows
- camqadm: Now properly handles Ctrl+C by simply exiting instead of showing confusing traceback.
- Windows: All tests are now passing on Windows.
- Remove bin/ directory, and *scripts* section from setup.py.

  This means we now rely completely on setuptools entrypoints.

Experimental

- Jython: celeryd now runs on Jython using the threaded pool.

  All tests pass, but there may still be bugs lurking around the corners.
- PyPy: celeryd now runs on PyPy.

  It runs without any pool, so to get parallel execution you must start multiple instances (e.g. using **celeryd-multi**).

  Sadly an initial benchmark seems to show a 30% performance decrease on pypy-1.4.1 + JIT. We would like to find out why this is, so stay tuned.
- `PublisherPool`: Experimental pool of task publishers and connections to be used with the *retry* argument to *apply_async*.

  The example code below will re-use connections and channels, and retry sending of the task message if the connection is lost.

```
from celery import current_app

# Global pool
pool = current_app().amqp.PublisherPool(limit=10)

def my_view(request):
    with pool.acquire() as publisher:
        add.apply_async((2, 2), publisher=publisher, retry=True)
```

2.15.5 Change history for Celery 2.1

2.1.4

release-date 2010-12-03 12:00 P.M CEST

Fixes

- Execution options to *apply_async* now takes precedence over options returned by active routers. This was a regression introduced recently (Issue #244).
- *celeryev* curses monitor: Long arguments are now truncated so curses doesn't crash with out of bounds errors. (Issue #235).
- *celeryd*: Channel errors occurring while handling control commands no longer crash the worker but are instead logged with severity error.
- SQLAlchemy database backend: Fixed a race condition occurring when the client wrote the pending state. Just like the Django database backend, it does no longer save the pending state (Issue #261 + Issue #262).
- Error email body now uses *repr(exception)* instead of *str(exception)*, as the latter could result in Unicode decode errors (Issue #245).
- Error email timeout value is now configurable by using the `EMAIL_TIMEOUT` setting.
- *celeryev*: Now works on Windows (but the curses monitor won't work without having curses).
- Unit test output no longer emits non-standard characters.
- *celeryd*: The broadcast consumer is now closed if the connection is reset.
- *celeryd*: Now properly handles errors occurring while trying to acknowledge the message.
- ***TaskRequest.on_failure* now encodes traceback using the current filesystem** encoding. (Issue #286).
- *EagerResult* can now be pickled (Issue #288).

Documentation

- Adding *Contributing*.
- Added *Optimizing*.

- Added *Security* section to the FAQ.

2.1.3

release-date 2010-11-09 05:00 P.M CEST

- Fixed deadlocks in *timer2* which could lead to *djcelerymon/celeryev -c* hanging.
- *EventReceiver*: now sends heartbeat request to find workers.

  This means **celeryev** and friends finds workers immediately at startup.
- celeryev cursesmon: Set screen_delay to 10ms, so the screen refreshes more often.
- Fixed pickling errors when pickling `AsyncResult` on older Python versions.
- celeryd: prefetch count was decremented by eta tasks even if there were no active prefetch limits.

2.1.2

release-data TBA

Fixes

- celeryd: Now sends the `task-retried` event for retried tasks.
- celeryd: Now honors ignore result for `WorkerLostError` and timeout errors.
- celerybeat: Fixed `UnboundLocalError` in celerybeat logging when using logging setup signals.
- celeryd: All log messages now includes *exc_info*.

2.1.1

release-date 2010-10-14 02:00 P.M CEST

Fixes

- Now working on Windows again.

  Removed dependency on the pwd/grp modules.
- snapshots: Fixed race condition leading to loss of events.
- celeryd: Reject tasks with an eta that cannot be converted to a time stamp.

  See issue #209
- concurrency.processes.pool: The semaphore was released twice for each task (both at ACK and result ready).

  This has been fixed, and it is now released only once per task.
- docs/configuration: Fixed typo *CELERYD_TASK_SOFT_TIME_LIMIT* -> `CELERYD_TASK_SOFT_TIME_LIMIT`.

  See issue #214
- control command *dump_scheduled*: was using old .info attribute
- **celeryd-multi: Fixed *set changed size during iteration* bug** occurring in the restart command.

- celeryd: Accidentally tried to use additional command line arguments.

  This would lead to an error like:

  got multiple values for keyword argument 'concurrency'.

  Additional command line arguments are now ignored, and does not produce this error. However – we do reserve the right to use positional arguments in the future, so please do not depend on this behavior.

- celerybeat: Now respects routers and task execution options again.
- celerybeat: Now reuses the publisher instead of the connection.
- Cache result backend: Using `float` as the expires argument to *cache.set* is deprecated by the memcached libraries, so we now automatically cast to `int`.
- unit tests: No longer emits logging and warnings in test output.

News

- Now depends on carrot version 0.10.7.
- Added `CELERY_REDIRECT_STDOUTS`, and `CELERYD_REDIRECT_STDOUTS_LEVEL` settings.

  `CELERY_REDIRECT_STDOUTS` is used by **celeryd** and **celerybeat**. All output to *stdout* and *stderr* will be redirected to the current logger if enabled.

  `CELERY_REDIRECT_STDOUTS_LEVEL` decides the log level used and is `WARNING` by default.

- Added `CELERYBEAT_SCHEDULER` setting.

  This setting is used to define the default for the -S option to **celerybeat**.

  Example:

  ```
  CELERYBEAT_SCHEDULER = "djcelery.schedulers.DatabaseScheduler"
  ```

- Added Task.expires: Used to set default expiry time for tasks.
- New remote control commands: *add_consumer* and *cancel_consumer*.

  add_consumer(queue, exchange, exchange_type, routing_key, **options)
  Tells the worker to declare and consume from the specified declaration.

  cancel_consumer(*queue_name*)
  Tells the worker to stop consuming from queue (by queue name).

  Commands also added to **celeryctl** and `inspect`.

  Example using celeryctl to start consuming from queue "queue", in exchange "exchange", of type "direct" using binding key "key":

  ```
  $ celeryctl inspect add_consumer queue exchange direct key
  $ celeryctl inspect cancel_consumer queue
  ```

  See *celery: Command Line Management Utility* for more information about the **celeryctl** program.

  Another example using `inspect`:

```
>>> from celery.task.control import inspect
>>> inspect.add_consumer(queue="queue", exchange="exchange",
...                      exchange_type="direct",
...                      routing_key="key",
...                      durable=False,
...                      auto_delete=True)

>>> inspect.cancel_consumer("queue")
```

- celerybeat: Now logs the traceback if a message can't be sent.
- celerybeat: Now enables a default socket timeout of 30 seconds.
- README/introduction/homepage: Added link to Flask-Celery.

2.1.0

release-date 2010-10-08 12:00 P.M CEST

Important Notes

- Celery is now following the versioning semantics defined by semver.

  This means we are no longer allowed to use odd/even versioning semantics By our previous versioning scheme this stable release should have been version 2.2.

- Now depends on Carrot 0.10.7.
- No longer depends on SQLAlchemy, this needs to be installed separately if the database result backend is used.
- django-celery now comes with a monitor for the Django Admin interface. This can also be used if you're not a Django user. (Update: Django-Admin monitor has been replaced with Flower, see the Monitoring guide).
- If you get an error after upgrading saying: *AttributeError: 'module' object has no attribute 'system'*,

  Then this is because the *celery.platform* module has been renamed to *celery.platforms* to not collide with the built-in `platform` module.

  You have to remove the old `platform.py` (and maybe `platform.pyc`) file from your previous Celery installation.

  To do this use **python** to find the location of this module:

  ```
  $ python
  >>> import celery.platform
  >>> celery.platform
  <module 'celery.platform' from '/opt/devel/celery/celery/platform.pyc'>
  ```

  Here the compiled module is in `/opt/devel/celery/celery/`, to remove the offending files do:

  ```
  $ rm -f /opt/devel/celery/celery/platform.py*
  ```

News

- Added support for expiration of AMQP results (requires RabbitMQ 2.1.0)

  The new configuration option `CELERY_AMQP_TASK_RESULT_EXPIRES` sets the expiry time in seconds (can be int or float):

  ```
  CELERY_AMQP_TASK_RESULT_EXPIRES = 30 * 60   # 30 minutes.
  CELERY_AMQP_TASK_RESULT_EXPIRES = 0.80      # 800 ms.
  ```

- celeryev: Event Snapshots

  If enabled, **celeryd** sends messages about what the worker is doing. These messages are called "events". The events are used by real-time monitors to show what the cluster is doing, but they are not very useful for monitoring over a longer period of time. Snapshots lets you take "pictures" of the clusters state at regular intervals. This can then be stored in a database to generate statistics with, or even monitoring over longer time periods.

  django-celery now comes with a Celery monitor for the Django Admin interface. To use this you need to run the django-celery snapshot camera, which stores snapshots to the database at configurable intervals.

  To use the Django admin monitor you need to do the following:

  1. Create the new database tables:

     ```
     $ python manage.py syncdb
     ```

  2. Start the django-celery snapshot camera:

     ```
     $ python manage.py celerycam
     ```

  3. Open up the django admin to monitor your cluster.

  The admin interface shows tasks, worker nodes, and even lets you perform some actions, like revoking and rate limiting tasks, and shutting down worker nodes.

  There's also a Debian init.d script for `celeryev` available, see *Running the worker as a daemon* for more information.

  New command line arguments to celeryev:

  - *`-c|--camera`*: Snapshot camera class to use.
  - *`--logfile|-f`*: Log file
  - *`--loglevel|-l`*: Log level
  - *`--maxrate|-r`*: Shutter rate limit.
  - *`--freq|-F`*: Shutter frequency

  The *`--camera`* argument is the name of a class used to take snapshots with. It must support the interface defined by `celery.events.snapshot.Polaroid`.

  Shutter frequency controls how often the camera thread wakes up, while the rate limit controls how often it will actually take a snapshot. The rate limit can be an integer (snapshots/s), or a rate limit string which has the same syntax as the task rate limit strings (*"200/m"*, *"10/s"*, *"1/h"*, etc).

For the Django camera case, this rate limit can be used to control how often the snapshots are written to the database, and the frequency used to control how often the thread wakes up to check if there's anything new.

The rate limit is off by default, which means it will take a snapshot for every *--frequency* seconds.

- `broadcast()`: Added callback argument, this can be used to process replies immediately as they arrive.
- celeryctl: New command line utility to manage and inspect worker nodes, apply tasks and inspect the results of tasks.

  See also:

  The *celery: Command Line Management Utility* section in the *User Guide*.

  Some examples:

```
$ celeryctl apply tasks.add -a '[2, 2]' --countdown=10

$ celeryctl inspect active
$ celeryctl inspect registered_tasks
$ celeryctl inspect scheduled
$ celeryctl inspect --help
$ celeryctl apply --help
```

- Added the ability to set an expiry date and time for tasks.

  Example:

```
>>> # Task expires after one minute from now.
>>> task.apply_async(args, kwargs, expires=60)
>>> # Also supports datetime
>>> task.apply_async(args, kwargs,
...                  expires=datetime.now() + timedelta(days=1)
```

  When a worker receives a task that has been expired it will be marked as revoked (`TaskRevokedError`).

- Changed the way logging is configured.

  We now configure the root logger instead of only configuring our custom logger. In addition we don't hijack the multiprocessing logger anymore, but instead use a custom logger name for different applications:

Application	Logger Name
celeryd	"celery"
celerybeat	"celery.beat"
celeryev	"celery.ev"

  This means that the *loglevel* and *logfile* arguments will affect all registered loggers (even those from 3rd party libraries). Unless you configure the loggers manually as shown below, that is.

  Users can choose to configure logging by subscribing to the :signal:`~celery.signals.setup_logging` signal:

```
from logging.config import fileConfig
from celery import signals

@signals.setup_logging.connect
```

```
def setup_logging(**kwargs):
    fileConfig("logging.conf")
```

If there are no receivers for this signal, the logging subsystem will be configured using the *--loglevel*/*--logfile* argument, this will be used for *all defined loggers*.

Remember that celeryd also redirects stdout and stderr to the celery logger, if manually configure logging you also need to redirect the stdouts manually:

```
from logging.config import fileConfig
from celery import log

def setup_logging(**kwargs):
     import logging
     fileConfig("logging.conf")
     stdouts = logging.getLogger("mystdoutslogger")
     log.redirect_stdouts_to_logger(stdouts, loglevel=logging.WARNING)
```

- celeryd: Added command line option *-I*/*--include*:

  A comma separated list of (task) modules to be imported.

  Example:

  ```
  $ celeryd -I app1.tasks,app2.tasks
  ```

- celeryd: now emits a warning if running as the root user (euid is 0).
- `celery.messaging.establish_connection()`: Ability to override defaults used using keyword argument "defaults".
- celeryd: Now uses *multiprocessing.freeze_support()* so that it should work with **py2exe**, **PyInstaller**, **cx_Freeze**, etc.
- celeryd: Now includes more metadata for the `STARTED` state: PID and host name of the worker that started the task.

  See issue #181

- subtask: Merge additional keyword arguments to *subtask()* into task keyword arguments.

  e.g.:

  ```
  >>> s = subtask((1, 2), {"foo": "bar"}, baz=1)
  >>> s.args
  (1, 2)
  >>> s.kwargs
  {"foo": "bar", "baz": 1}
  ```

  See issue #182.

- celeryd: Now emits a warning if there is already a worker node using the same name running on the same virtual host.
- AMQP result backend: Sending of results are now retried if the connection is down.
- **AMQP result backend: *result.get()*: Wait for next state if state is not** in `READY_STATES`.
- TaskSetResult now supports subscription.

```
>>> res = TaskSet(tasks).apply_async()
>>> res[0].get()
```

- Added *Task.send_error_emails* + *Task.error_whitelist*, so these can be configured per task instead of just by the global setting.
- Added *Task.store_errors_even_if_ignored*, so it can be changed per Task, not just by the global setting.
- The crontab scheduler no longer wakes up every second, but implements *remaining_estimate* (*Optimization*).
- **celeryd: Store `FAILURE` result if the `WorkerLostError` exception occurs (worker process disappeared).**
- celeryd: Store `FAILURE` result if one of the **TimeLimitExceeded* exceptions occurs.
- Refactored the periodic task responsible for cleaning up results.
  - **The backend cleanup task is now only added to the schedule if `CELERY_TASK_RESULT_EXPIRES` is set.**
  - If the schedule already contains a periodic task named "celery.backend_cleanup" it won't change it, so the behavior of the backend cleanup task can be easily changed.
  - The task is now run every day at 4:00 AM, rather than every day since the first time it was run (using crontab schedule instead of *run_every*)
  - **Renamed *celery.task.builtins.DeleteExpiredTaskMetaTask* -> `celery.task.builtins.backend_cleanup`**
  - The task itself has been renamed from "celery.delete_expired_task_meta" to "celery.backend_cleanup"

  See issue #134.
- Implemented *AsyncResult.forget* for sqla/cache/redis/tyrant backends. (Forget and remove task result).

  See issue #184.
- `TaskSetResult.join`: Added 'propagate=True' argument.

  When set to `False` exceptions occurring in subtasks will not be re-raised.
- Added *Task.update_state(task_id, state, meta)* as a shortcut to *task.backend.store_result(task_id, meta, state)*.

  The backend interface is "private" and the terminology outdated, so better to move this to `Task` so it can be used.
- timer2: Set *self.running=False* in `stop()` so it won't try to join again on subsequent calls to *stop()*.
- Log colors are now disabled by default on Windows.
- *celery.platform* renamed to `celery.platforms`, so it doesn't collide with the built-in `platform` module.
- Exceptions occurring in Mediator+Pool callbacks are now caught and logged instead of taking down the worker.
- Redis result backend: Now supports result expiration using the Redis *EXPIRE* command.
- unit tests: Don't leave threads running at tear down.
- celeryd: Task results shown in logs are now truncated to 46 chars.
- ***Task.__name__* is now an alias to *self.__class__.__name__*.** This way tasks introspects more like regular functions.
- *Task.retry*: Now raises `TypeError` if kwargs argument is empty.

  See issue #164.

- timedelta_seconds: Use *timedelta.total_seconds* if running on Python 2.7
- `TokenBucket`: Generic Token Bucket algorithm
- `celery.events.state`: Recording of cluster state can now be paused and resumed, including support for buffering.

  `State.`**`freeze`**`(`*buffer=True*`)`
  Pauses recording of the stream.

  If *buffer* is true, events received while being frozen will be buffered, and may be replayed later.

  `State.`**`thaw`**`(`*replay=True*`)`
  Resumes recording of the stream.

  If *replay* is true, then the recorded buffer will be applied.

  `State.`**`freeze_while`**`(`*fun*`)`
  With a function to apply, freezes the stream before, and replays the buffer after the function returns.
- `EventReceiver.capture` Now supports a timeout keyword argument.
- celeryd: The mediator thread is now disabled if `CELERY_RATE_LIMITS` is enabled, and tasks are directly sent to the pool without going through the ready queue (*Optimization*).

Fixes

- Pool: Process timed out by *TimeoutHandler* must be joined by the Supervisor, so don't remove it from the internal process list.

  See issue #192.
- *TaskPublisher.delay_task* now supports exchange argument, so exchange can be overridden when sending tasks in bulk using the same publisher

  See issue #187.
- celeryd no longer marks tasks as revoked if `CELERY_IGNORE_RESULT` is enabled.

  See issue #207.
- AMQP Result backend: Fixed bug with *result.get()* if `CELERY_TRACK_STARTED` enabled.

  result.get() would stop consuming after receiving the `STARTED` state.
- Fixed bug where new processes created by the pool supervisor becomes stuck while reading from the task Queue.

  See http://bugs.python.org/issue10037
- Fixed timing issue when declaring the remote control command reply queue

  This issue could result in replies being lost, but have now been fixed.
- Backward compatible *LoggerAdapter* implementation: Now works for Python 2.4.

  Also added support for several new methods: *fatal*, *makeRecord*, *_log*, *log*, *isEnabledFor*, *addHandler*, *removeHandler*.

Experimental

- celeryd-multi: Added daemonization support.

  celeryd-multi can now be used to start, stop and restart worker nodes:

  ```
  $ celeryd-multi start jerry elaine george kramer
  ```

  This also creates PID files and log files (`celeryd@jerry.pid`, ..., `celeryd@jerry.log`. To specify a location for these files use the *–pidfile* and *–logfile* arguments with the *%n* format:

  ```
  $ celeryd-multi start jerry elaine george kramer \
                  --logfile=/var/log/celeryd@%n.log \
                  --pidfile=/var/run/celeryd@%n.pid
  ```

  Stopping:

  ```
  $ celeryd-multi stop jerry elaine george kramer
  ```

  Restarting. The nodes will be restarted one by one as the old ones are shutdown:

  ```
  $ celeryd-multi restart jerry elaine george kramer
  ```

  Killing the nodes (**WARNING**: Will discard currently executing tasks):

  ```
  $ celeryd-multi kill jerry elaine george kramer
  ```

  See *celeryd-multi help* for help.

- celeryd-multi: *start* command renamed to *show*.

  celeryd-multi start will now actually start and detach worker nodes. To just generate the commands you have to use *celeryd-multi show*.

- celeryd: Added *–pidfile* argument.

  The worker will write its pid when it starts. The worker will not be started if this file exists and the pid contained is still alive.

- Added generic init.d script using *celeryd-multi*

  http://github.com/celery/celery/tree/master/extra/generic-init.d/celeryd

Documentation

- Added User guide section: Monitoring
- Added user guide section: Periodic Tasks

  Moved from *getting-started/periodic-tasks* and updated.

- tutorials/external moved to new section: “community”.
- References has been added to all sections in the documentation.

  This makes it easier to link between documents.

2.15.6 Change history for Celery 2.0

2.0.3

release-date 2010-08-27 12:00 P.M CEST

Fixes

- celeryd: Properly handle connection errors happening while closing consumers.
- celeryd: Events are now buffered if the connection is down, then sent when the connection is re-established.
- No longer depends on the `mailer` package.

  This package had a name space collision with *django-mailer*, so its functionality was replaced.
- Redis result backend: Documentation typos: Redis doesn't have database names, but database numbers. The default database is now 0.
- `inspect`: *registered_tasks* was requesting an invalid command because of a typo.

  See issue #170.
- `CELERY_ROUTES`: Values defined in the route should now have precedence over values defined in `CELERY_QUEUES` when merging the two.

  With the follow settings:

  ```
  CELERY_QUEUES = {"cpubound": {"exchange": "cpubound",
                                "routing_key": "cpubound"}}

  CELERY_ROUTES = {"tasks.add": {"queue": "cpubound",
                                 "routing_key": "tasks.add",
                                 "serializer": "json"}}
  ```

  The final routing options for *tasks.add* will become:

  ```
  {"exchange": "cpubound",
   "routing_key": "tasks.add",
   "serializer": "json"}
  ```

This was not the case before: the values in `CELERY_QUEUES` would take precedence.

- Worker crashed if the value of `CELERY_TASK_ERROR_WHITELIST` was not an iterable
- `apply()`: Make sure *kwargs["task_id"]* is always set.
- *AsyncResult.traceback*: Now returns `None`, instead of raising `KeyError` if traceback is missing.
- `inspect`: Replies did not work correctly if no destination was specified.
- Can now store result/metadata for custom states.
- celeryd: A warning is now emitted if the sending of task error emails fails.
- celeryev: Curses monitor no longer crashes if the terminal window is resized.

  See issue #160.
- celeryd: On OS X it is not possible to run *os.exec** in a process that is threaded.

  This breaks the SIGHUP restart handler, and is now disabled on OS X, emitting a warning instead.

  See issue #152.
- `celery.execute.trace`: Properly handle *raise(str)*, which is still allowed in Python 2.4.

  See issue #175.
- Using urllib2 in a periodic task on OS X crashed because of the proxy auto detection used in OS X.

  This is now fixed by using a workaround. See issue #143.
- Debian init scripts: Commands should not run in a sub shell

  See issue #163.
- Debian init scripts: Use the absolute path of celeryd to allow stat

  See issue #162.

Documentation

- getting-started/broker-installation: Fixed typo

  set_permissions "" -> set_permissions "."*.
- Tasks User Guide: Added section on database transactions.

  See issue #169.
- Routing User Guide: Fixed typo *"feed": -> {"queue": "feeds"}*.

  See issue #169.
- Documented the default values for the `CELERYD_CONCURRENCY` and `CELERYD_PREFETCH_MULTIPLIER` settings.
- Tasks User Guide: Fixed typos in the subtask example
- celery.signals: Documented worker_process_init.
- Daemonization cookbook: Need to export DJANGO_SETTINGS_MODULE in */etc/default/celeryd*.
- Added some more FAQs from stack overflow
- Daemonization cookbook: Fixed typo *CELERYD_LOGFILE/CELERYD_PIDFILE*

to *CELERYD_LOG_FILE* / *CELERYD_PID_FILE*

Also added troubleshooting section for the init scripts.

2.0.2

release-date 2010-07-22 11:31 A.M CEST

- Routes: When using the dict route syntax, the exchange for a task could disappear making the task unroutable.

  See issue #158.

- Test suite now passing on Python 2.4
- No longer have to type *PYTHONPATH*=. to use celeryconfig in the current directory.

  This is accomplished by the default loader ensuring that the current directory is in *sys.path* when loading the config module. *sys.path* is reset to its original state after loading.

  Adding the current working directory to *sys.path* without the user knowing may be a security issue, as this means someone can drop a Python module in the users directory that executes arbitrary commands. This was the original reason not to do this, but if done *only when loading the config module*, this means that the behavior will only apply to the modules imported in the config module, which I think is a good compromise (certainly better than just explicitly setting *PYTHONPATH*=. anyway)

- Experimental Cassandra backend added.
- celeryd: SIGHUP handler accidentally propagated to worker pool processes.

  In combination with 7a7c44e39344789f11b5346e9cc8340f5fe4846c this would make each child process start a new celeryd when the terminal window was closed :/

- celeryd: Do not install SIGHUP handler if running from a terminal.

  This fixes the problem where celeryd is launched in the background when closing the terminal.

- celeryd: Now joins threads at shutdown.

  See issue #152.

- Test tear down: Don't use *atexit* but nose's *teardown()* functionality instead.

  See issue #154.

- Debian init script for celeryd: Stop now works correctly.
- Task logger: *warn* method added (synonym for *warning*)
- Can now define a white list of errors to send error emails for.

  Example:

  ```
  CELERY_TASK_ERROR_WHITELIST = ('myapp.MalformedInputError')
  ```

  See issue #153.

- celeryd: Now handles overflow exceptions in *time.mktime* while parsing the ETA field.
- LoggerWrapper: Try to detect loggers logging back to stderr/stdout making an infinite loop.
- Added `celery.task.control.inspect`: Inspects a running worker.

  Examples:

```
# Inspect a single worker
>>> i = inspect("myworker.example.com")

# Inspect several workers
>>> i = inspect(["myworker.example.com", "myworker2.example.com"])

# Inspect all workers consuming on this vhost.
>>> i = inspect()

### Methods

# Get currently executing tasks
>>> i.active()

# Get currently reserved tasks
>>> i.reserved()

# Get the current eta schedule
>>> i.scheduled()

# Worker statistics and info
>>> i.stats()

# List of currently revoked tasks
>>> i.revoked()

# List of registered tasks
>>> i.registered_tasks()
```

- Remote control commands *dump_active/dump_reserved/dump_schedule* now replies with detailed task requests.

  Containing the original arguments and fields of the task requested.

  In addition the remote control command *set_loglevel* has been added, this only changes the log level for the main process.

- Worker control command execution now catches errors and returns their string representation in the reply.
- Functional test suite added

  `celery.tests.functional.case` contains utilities to start and stop an embedded celeryd process, for use in functional testing.

2.0.1

release-date 2010-07-09 03:02 P.M CEST

- multiprocessing.pool: Now handles encoding errors, so that pickling errors doesn't crash the worker processes.
- The remote control command replies was not working with RabbitMQ 1.8.0's stricter equivalence checks.

  If you've already hit this problem you may have to delete the declaration:

  ```
  $ camqadm exchange.delete celerycrq
  ```

  or:

  ```
  $ python manage.py camqadm exchange.delete celerycrq
  ```

- A bug sneaked in the ETA scheduler that made it only able to execute one task per second(!)

  The scheduler sleeps between iterations so it doesn't consume too much CPU. It keeps a list of the scheduled items sorted by time, at each iteration it sleeps for the remaining time of the item with the nearest deadline. If there are no eta tasks it will sleep for a minimum amount of time, one second by default.

  A bug sneaked in here, making it sleep for one second for every task that was scheduled. This has been fixed, so now it should move tasks like hot knife through butter.

  In addition a new setting has been added to control the minimum sleep interval; `CELERYD_ETA_SCHEDULER_PRECISION`. A good value for this would be a float between 0 and 1, depending on the needed precision. A value of 0.8 means that when the ETA of a task is met, it will take at most 0.8 seconds for the task to be moved to the ready queue.

- Pool: Supervisor did not release the semaphore.

  This would lead to a deadlock if all workers terminated prematurely.

- Added Python version trove classifiers: 2.4, 2.5, 2.6 and 2.7
- Tests now passing on Python 2.7.
- Task.__reduce__: Tasks created using the task decorator can now be pickled.
- setup.py: nose added to *tests_require*.
- Pickle should now work with SQLAlchemy 0.5.x
- New homepage design by Jan Henrik Helmers: http://celeryproject.org
- New Sphinx theme by Armin Ronacher: http://docs.celeryproject.org/
- Fixed "pending_xref" errors shown in the HTML rendering of the documentation. Apparently this was caused by new changes in Sphinx 1.0b2.
- Router classes in `CELERY_ROUTES` are now imported lazily.

  Importing a router class in a module that also loads the Celery environment would cause a circular dependency. This is solved by importing it when needed after the environment is set up.

- `CELERY_ROUTES` was broken if set to a single dict.

  This example in the docs should now work again:

  ```
  CELERY_ROUTES = {"feed.tasks.import_feed": "feeds"}
  ```

- *CREATE_MISSING_QUEUES* was not honored by apply_async.
- New remote control command: *stats*

  Dumps information about the worker, like pool process ids, and total number of tasks executed by type.

  Example reply:

  ```
  [{'worker.local':
       'total': {'tasks.sleeptask': 6},
       'pool': {'timeouts': [None, None],
                'processes': [60376, 60377],
                'max-concurrency': 2,
                'max-tasks-per-child': None,
                'put-guarded-by-semaphore': True}}]
  ```

- New remote control command: *dump_active*

  Gives a list of tasks currently being executed by the worker. By default arguments are passed through repr in case there are arguments that is not JSON encodable. If you know the arguments are JSON safe, you can pass the argument *safe=True*.

  Example reply:

```
>>> broadcast("dump_active", arguments={"safe": False}, reply=True)
[{'worker.local': [
    {'args': '(1,)',
     'time_start': 1278580542.6300001,
     'name': 'tasks.sleeptask',
     'delivery_info': {
         'consumer_tag': '30',
         'routing_key': 'celery',
         'exchange': 'celery'},
     'hostname': 'casper.local',
     'acknowledged': True,
     'kwargs': '{}',
     'id': '802e93e9-e470-47ed-b913-06de8510aca2',
    }
]}]
```

- Added experimental support for persistent revokes.

  Use the *-S*|*–statedb* argument to celeryd to enable it:

```
$ celeryd --statedb=/var/run/celeryd
```

  This will use the file: */var/run/celeryd.db*, as the *shelve* module automatically adds the *.db* suffix.

2.0.0

release-date 2010-07-02 02:30 P.M CEST

Foreword

Celery 2.0 contains backward incompatible changes, the most important being that the Django dependency has been removed so Celery no longer supports Django out of the box, but instead as an add-on package called django-celery.

We're very sorry for breaking backwards compatibility, but there's also many new and exciting features to make up for the time you lose upgrading, so be sure to read the *News* section.

Quite a lot of potential users have been upset about the Django dependency, so maybe this is a chance to get wider adoption by the Python community as well.

Big thanks to all contributors, testers and users!

Upgrading for Django-users

Django integration has been moved to a separate package: django-celery.

- To upgrade you need to install the django-celery module and change:

```
INSTALLED_APPS = "celery"
```

to:

```
INSTALLED_APPS = "djcelery"
```

- If you use *mod_wsgi* you need to add the following line to your *.wsgi* file:

```
import os
os.environ["CELERY_LOADER"] = "django"
```

- The following modules has been moved to django-celery:

Module name	Replace with
celery.models	*djcelery.models*
celery.managers	*djcelery.managers*
celery.views	*djcelery.views*
celery.urls	*djcelery.urls*
celery.management	*djcelery.management*
celery.loaders.djangoapp	*djcelery.loaders*
celery.backends.database	*djcelery.backends.database*
celery.backends.cache	*djcelery.backends.cache*

Importing `djcelery` will automatically setup Celery to use Django loader. loader. It does this by setting the `CELERY_LOADER` environment variable to *"django"* (it won't change it if a loader is already set.)

When the Django loader is used, the "database" and "cache" result backend aliases will point to the `djcelery` backends instead of the built-in backends, and configuration will be read from the Django settings.

Upgrading for others

Database result backend The database result backend is now using SQLAlchemy instead of the Django ORM, see Supported Databases for a table of supported databases.

The *DATABASE_** settings has been replaced by a single setting: `CELERY_RESULT_DBURI`. The value here should be an SQLAlchemy Connection String, some examples include:

```
# sqlite (filename)
CELERY_RESULT_DBURI = "sqlite:///celerydb.sqlite"

# mysql
CELERY_RESULT_DBURI = "mysql://scott:tiger@localhost/foo"

# postgresql
CELERY_RESULT_DBURI = "postgresql://scott:tiger@localhost/mydatabase"

# oracle
CELERY_RESULT_DBURI = "oracle://scott:tiger@127.0.0.1:1521/sidname"
```

See SQLAlchemy Connection Strings for more information about connection strings.

To specify additional SQLAlchemy database engine options you can use the `CELERY_RESULT_ENGINE_OPTIONS` setting:

```
# echo enables verbose logging from SQLAlchemy.
CELERY_RESULT_ENGINE_OPTIONS = {"echo": True}
```

Cache result backend The cache result backend is no longer using the Django cache framework, but it supports mostly the same configuration syntax:

```
CELERY_CACHE_BACKEND = "memcached://A.example.com:11211;B.example.com"
```

To use the cache backend you must either have the pylibmc or python-memcached library installed, of which the former is regarded as the best choice.

The support backend types are *memcached://* and *memory://*, we haven't felt the need to support any of the other backends provided by Django.

Backward incompatible changes

- Default (python) loader now prints warning on missing *celeryconfig.py* instead of raising `ImportError`.

  celeryd raises `ImproperlyConfigured` if the configuration is not set up. This makes it possible to use *–help* etc., without having a working configuration.

  Also this makes it possible to use the client side of celery without being configured:

  ```
  >>> from carrot.connection import BrokerConnection
  >>> conn = BrokerConnection("localhost", "guest", "guest", "/")
  >>> from celery.execute import send_task
  >>> r = send_task("celery.ping", args=(), kwargs={}, connection=conn)
  >>> from celery.backends.amqp import AMQPBackend
  >>> r.backend = AMQPBackend(connection=conn)
  >>> r.get()
  'pong'
  ```

- The following deprecated settings has been removed (as scheduled by the *Celery Deprecation Timeline*):

Setting name	Replace with
CELERY_AMQP_CONSUMER_QUEUES	*CELERY_QUEUES*
CELERY_AMQP_EXCHANGE	*CELERY_DEFAULT_EXCHANGE*
CELERY_AMQP_EXCHANGE_TYPE	*CELERY_DEFAULT_EXCHANGE_TYPE*
CELERY_AMQP_CONSUMER_ROUTING_KEY	*CELERY_QUEUES*
CELERY_AMQP_PUBLISHER_ROUTING_KEY	*CELERY_DEFAULT_ROUTING_KEY*

- The *celery.task.rest* module has been removed, use `celery.task.http` instead (as scheduled by the *Celery Deprecation Timeline*).
- It's no longer allowed to skip the class name in loader names. (as scheduled by the *Celery Deprecation Timeline*):

  Assuming the implicit *Loader* class name is no longer supported, if you use e.g.:

  ```
  CELERY_LOADER = "myapp.loaders"
  ```

  You need to include the loader class name, like this:

  ```
  CELERY_LOADER = "myapp.loaders.Loader"
  ```

- `CELERY_TASK_RESULT_EXPIRES` now defaults to 1 day.

Previous default setting was to expire in 5 days.

- AMQP backend: Don't use different values for *auto_delete*.

  This bug became visible with RabbitMQ 1.8.0, which no longer allows conflicting declarations for the auto_delete and durable settings.

  If you've already used celery with this backend chances are you have to delete the previous declaration:

  ```
  $ camqadm exchange.delete celeryresults
  ```

- Now uses pickle instead of cPickle on Python versions <= 2.5

  cPickle is broken in Python <= 2.5.

  It unsafely and incorrectly uses relative instead of absolute imports, so e.g.:

  ```
  exceptions.KeyError
  ```

  becomes:

  ```
  celery.exceptions.KeyError
  ```

  Your best choice is to upgrade to Python 2.6, as while the pure pickle version has worse performance, it is the only safe option for older Python versions.

News

- **celeryev**: Curses Celery Monitor and Event Viewer.

  This is a simple monitor allowing you to see what tasks are executing in real-time and investigate tracebacks and results of ready tasks. It also enables you to set new rate limits and revoke tasks.

  Screenshot:

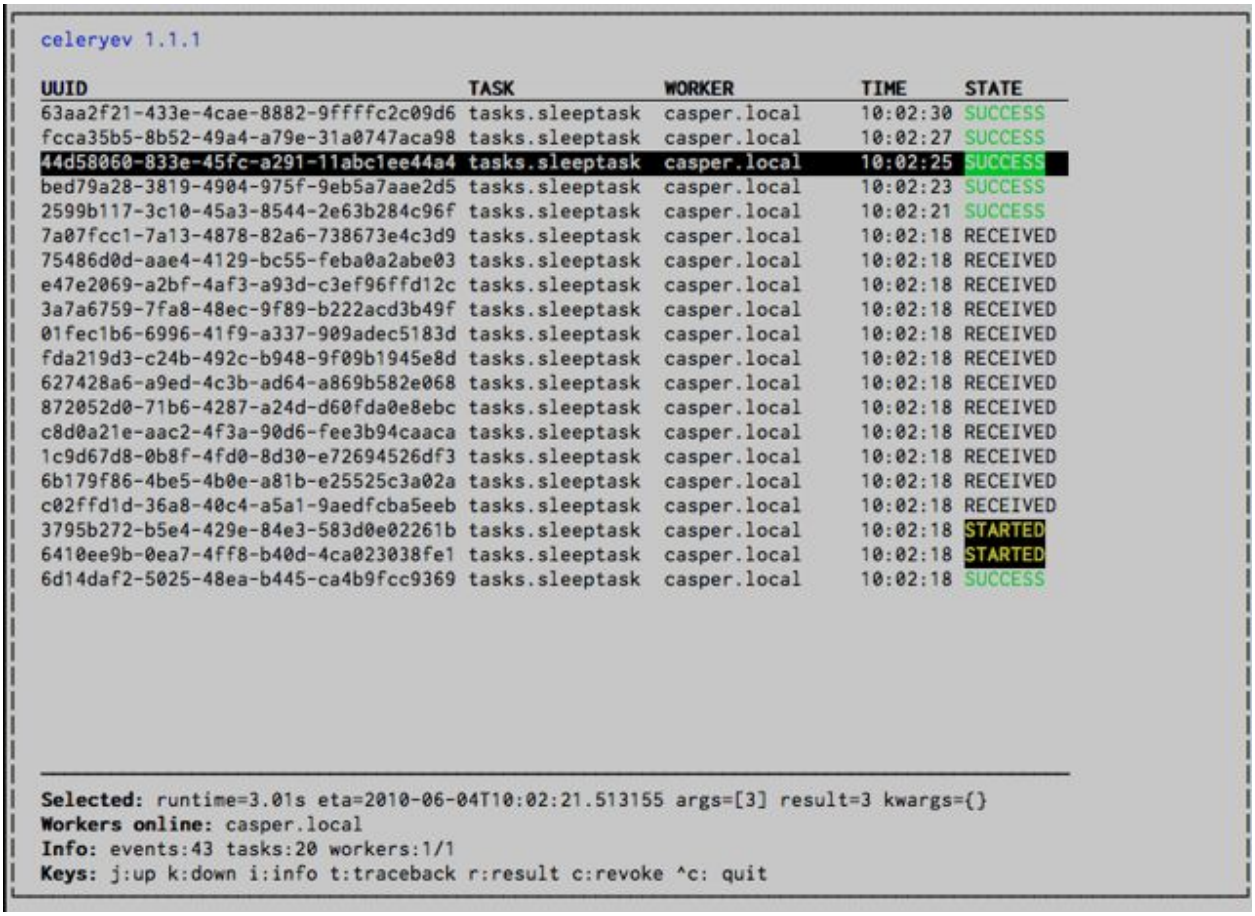
  If you run *celeryev* with the *-d* switch it will act as an event dumper, simply dumping the events it receives to standard out:

  ```
  $ celeryev -d
  -> celeryev: starting capture...
  casper.local [2010-06-04 10:42:07.020000] heartbeat
  casper.local [2010-06-04 10:42:14.750000] task received:
      tasks.add(61a68756-27f4-4879-b816-3cf815672b0e) args=[2, 2] kwargs={}
      eta=2010-06-04T10:42:16.669290, retries=0
  casper.local [2010-06-04 10:42:17.230000] task started
      tasks.add(61a68756-27f4-4879-b816-3cf815672b0e) args=[2, 2] kwargs={}
  casper.local [2010-06-04 10:42:17.960000] task succeeded:
      tasks.add(61a68756-27f4-4879-b816-3cf815672b0e)
      args=[2, 2] kwargs={} result=4, runtime=0.782663106918

  The fields here are, in order: *sender hostname*, *timestamp*, *event type* and
  *additional event fields*.
  ```

- AMQP result backend: Now supports *.ready()*, *.successful()*, *.result*, *.status*, and even responds to changes in task state

- New user guides:

  - *Workers Guide*


celeryev 1.1.1
UUID TASK WORKER TIME STATE
63aa2f21-433e-4cae-8882-9ffffc2c09d6 tasks.sleeptask casper.local 10:02:30 SUCCESS
fcca35b5-8b52-49a4-a79e-31a0747aca98 tasks.sleeptask casper.local 10:02:27 SUCCESS
44d58060-833e-45fc-a291-11abc1ee44a4 tasks.sleeptask casper.local 10:02:25 SUCCESS
bed79a28-3819-4904-975f-9eb5a7aae2d5 tasks.sleeptask casper.local 10:02:23 SUCCESS
2599b117-3c10-45a3-8544-2e63b284c96f tasks.sleeptask casper.local 10:02:21 SUCCESS
7a07fcc1-7a13-4878-82a6-738673e4c3d9 tasks.sleeptask casper.local 10:02:18 RECEIVED
75486d0d-aae4-4129-bc55-feba0a2abe03 tasks.sleeptask casper.local 10:02:18 RECEIVED
e47e2069-a2bf-4af3-a93d-c3ef96ffd12c tasks.sleeptask casper.local 10:02:18 RECEIVED
3a7a6759-7fa8-48ec-9f89-b222acd3b49f tasks.sleeptask casper.local 10:02:18 RECEIVED
01fec1b6-6996-41f9-a337-909adec5183d tasks.sleeptask casper.local 10:02:18 RECEIVED
fda219d3-c24b-492c-b948-9f09b1945e8d tasks.sleeptask casper.local 10:02:18 RECEIVED
627428a6-a9ed-4c3b-ad64-a869b582e068 tasks.sleeptask casper.local 10:02:18 RECEIVED
872052d0-71b6-4287-a24d-d60fda0e8ebc tasks.sleeptask casper.local 10:02:18 RECEIVED
c8d0a21e-aac2-4f3a-90d6-fee3b94caaca tasks.sleeptask casper.local 10:02:18 RECEIVED
1c9d67d8-0b8f-4fd0-8d30-e72694526df3 tasks.sleeptask casper.local 10:02:18 RECEIVED
6b179f86-4be5-4b0e-a81b-e25525c3a02a tasks.sleeptask casper.local 10:02:18 RECEIVED
c02ffd1d-36a8-40c4-a5a1-9aedfcba5eeb tasks.sleeptask casper.local 10:02:18 RECEIVED
3795b272-b5e4-429e-84e3-583d0e02261b tasks.sleeptask casper.local 10:02:18 STARTED
6410ee9b-0ea7-4ff8-b40d-4ca023038fe1 tasks.sleeptask casper.local 10:02:18 STARTED
6d14daf2-5025-48ea-b445-ca4b9fcc9369 tasks.sleeptask casper.local 10:02:18 SUCCESS
Selected: runtime=3.01s eta=2010-06-04T10:02:21.513155 args=[3] result=3 kwargs={}
Workers online: casper.local
Info: events:43 tasks:20 workers:1/1
Keys: j:up k:down i:info t:traceback r:result c:revoke ^c: quit

  - *Canvas: Designing Workflows*
  - *Routing Tasks*

- celeryd: Standard out/error is now being redirected to the log file.
- `billiard` has been moved back to the celery repository.

Module name	celery equivalent
billiard.pool	*celery.concurrency.processes.pool*
billiard.serialization	*celery.serialization*
billiard.utils.functional	*celery.utils.functional*

  The `billiard` distribution may be maintained, depending on interest.

- now depends on `carrot` >= 0.10.5
- now depends on `pyparsing`
- celeryd: Added *–purge* as an alias to *–discard*.
- celeryd: Ctrl+C (SIGINT) once does warm shutdown, hitting Ctrl+C twice forces termination.
- Added support for using complex crontab-expressions in periodic tasks. For example, you can now use:

```
>>> crontab(minute="*/15")
```

  or even:

```
>>> crontab(minute="*/30", hour="8-17,1-2", day_of_week="thu-fri")
```

  See *Periodic Tasks*.

- celeryd: Now waits for available pool processes before applying new tasks to the pool.

  This means it doesn't have to wait for dozens of tasks to finish at shutdown because it has applied prefetched tasks without having any pool processes available to immediately accept them.

  See issue #122.

- New built-in way to do task callbacks using `subtask`.

  See *Canvas: Designing Workflows* for more information.

- TaskSets can now contain several types of tasks.

  `TaskSet` has been refactored to use a new syntax, please see *Canvas: Designing Workflows* for more information.

  The previous syntax is still supported, but will be deprecated in version 1.4.

- TaskSet failed() result was incorrect.

  See issue #132.

- Now creates different loggers per task class.

  See issue #129.

- Missing queue definitions are now created automatically.

  You can disable this using the `CELERY_CREATE_MISSING_QUEUES` setting.

  The missing queues are created with the following options:

```
CELERY_QUEUES[name] = {"exchange": name,
                       "exchange_type": "direct",
                       "routing_key": "name}
```

This feature is added for easily setting up routing using the *-Q* option to *celeryd*:

```
$ celeryd -Q video, image
```

See the new routing section of the User Guide for more information: *Routing Tasks*.

- New Task option: *Task.queue*

  If set, message options will be taken from the corresponding entry in `CELERY_QUEUES`. *exchange*, *exchange_type* and *routing_key* will be ignored

- Added support for task soft and hard time limits.

  New settings added:

  - `CELERYD_TASK_TIME_LIMIT`

    Hard time limit. The worker processing the task will be killed and replaced with a new one when this is exceeded.

  - `CELERYD_TASK_SOFT_TIME_LIMIT`

    Soft time limit. The `SoftTimeLimitExceeded` exception will be raised when this is exceeded. The task can catch this to e.g. clean up before the hard time limit comes.

  New command line arguments to celeryd added: *–time-limit* and *–soft-time-limit*.

  What's left?

  This won't work on platforms not supporting signals (and specifically the *SIGUSR1* signal) yet. So an alternative the ability to disable the feature all together on nonconforming platforms must be implemented.

  Also when the hard time limit is exceeded, the task result should be a *TimeLimitExceeded* exception.

- Test suite is now passing without a running broker, using the carrot in-memory backend.

- Log output is now available in colors.

Log level	Color
DEBUG	Blue
WARNING	Yellow
CRITICAL	Magenta
ERROR	Red

  This is only enabled when the log output is a tty. You can explicitly enable/disable this feature using the `CELERYD_LOG_COLOR` setting.

- Added support for task router classes (like the django multi-db routers)

  - New setting: `CELERY_ROUTES`

  This is a single, or a list of routers to traverse when sending tasks. Dictionaries in this list converts to a `celery.routes.MapRoute` instance.

  Examples:

```
>>> CELERY_ROUTES = {"celery.ping": "default",
                     "mytasks.add": "cpu-bound",
                     "video.encode": {
                         "queue": "video",
                         "exchange": "media"
                         "routing_key": "media.video.encode"}}


>>> CELERY_ROUTES = ("myapp.tasks.Router",
                     {"celery.ping": "default})
```

Where *myapp.tasks.Router* could be:

```
class Router(object):

    def route_for_task(self, task, args=None, kwargs=None):
        if task == "celery.ping":
            return "default"
```

route_for_task may return a string or a dict. A string then means it's a queue name in CELERY_QUEUES, a dict means it's a custom route.

When sending tasks, the routers are consulted in order. The first router that doesn't return *None* is the route to use. The message options is then merged with the found route settings, where the routers settings have priority.

Example if `apply_async()` has these arguments:

```
>>> Task.apply_async(immediate=False, exchange="video",
...                  routing_key="video.compress")
```

and a router returns:

```
{"immediate": True,
 "exchange": "urgent"}
```

the final message options will be:

```
immediate=True, exchange="urgent", routing_key="video.compress"
```

(and any default message options defined in the `Task` class)

- New Task handler called after the task returns: `after_return()`.
- **`ExceptionInfo` now passed to** `on_retry()`/`on_failure()` as einfo keyword argument.
- celeryd: Added CELERYD_MAX_TASKS_PER_CHILD / *--maxtasksperchild*

  Defines the maximum number of tasks a pool worker can process before the process is terminated and replaced by a new one.
- Revoked tasks now marked with state REVOKED, and *result.get()* will now raise TaskRevokedError.
- `celery.task.control.ping()` now works as expected.
- *apply(throw=True)* / CELERY_EAGER_PROPAGATES_EXCEPTIONS: Makes eager execution re-raise task errors.
- New signal: ~celery.signals.worker_process_init: Sent inside the pool worker process at init.

- celeryd `-Q` option: Ability to specify list of queues to use, disabling other configured queues.

  For example, if `CELERY_QUEUES` defines four queues: *image*, *video*, *data* and *default*, the following command would make celeryd only consume from the *image* and *video* queues:

  ```
  $ celeryd -Q image,video
  ```

- celeryd: New return value for the *revoke* control command:

  Now returns:

  ```
  {"ok": "task $id revoked"}
  ```

  instead of *True*.

- celeryd: Can now enable/disable events using remote control

  Example usage:

  ```
  >>> from celery.task.control import broadcast
  >>> broadcast("enable_events")
  >>> broadcast("disable_events")
  ```

- Removed top-level tests directory. Test config now in celery.tests.config

  This means running the unit tests doesn't require any special setup. *celery/tests/__init__* now configures the `CELERY_CONFIG_MODULE` and `CELERY_LOADER` environment variables, so when *nosetests* imports that, the unit test environment is all set up.

  Before you run the tests you need to install the test requirements:

  ```
  $ pip install -r requirements/test.txt
  ```

  Running all tests:

  ```
  $ nosetests
  ```

  Specifying the tests to run:

  ```
  $ nosetests celery.tests.test_task
  ```

  Producing HTML coverage:

  ```
  $ nosetests --with-coverage3
  ```

  The coverage output is then located in *celery/tests/cover/index.html*.

- celeryd: New option *–version*: Dump version info and exit.

- `celeryd-multi`: Tool for shell scripts to start multiple workers.

  Some examples:

  ```
  # Advanced example with 10 workers:
  #   * Three of the workers processes the images and video queue
  #   * Two of the workers processes the data queue with loglevel DEBUG
  #   * the rest processes the default' queue.
  $ celeryd-multi start 10 -l INFO -Q:1-3 images,video -Q:4,5:data
      -Q default -L:4,5 DEBUG
  ```

```
# get commands to start 10 workers, with 3 processes each
$ celeryd-multi start 3 -c 3
celeryd -n celeryd1.myhost -c 3
celeryd -n celeryd2.myhost -c 3
celeryd -n celeryd3.myhost -c 3

# start 3 named workers
$ celeryd-multi start image video data -c 3
celeryd -n image.myhost -c 3
celeryd -n video.myhost -c 3
celeryd -n data.myhost -c 3

# specify custom hostname
$ celeryd-multi start 2 -n worker.example.com -c 3
celeryd -n celeryd1.worker.example.com -c 3
celeryd -n celeryd2.worker.example.com -c 3

# Additionl options are added to each celeryd',
# but you can also modify the options for ranges of or single workers

# 3 workers: Two with 3 processes, and one with 10 processes.
$ celeryd-multi start 3 -c 3 -c:1 10
celeryd -n celeryd1.myhost -c 10
celeryd -n celeryd2.myhost -c 3
celeryd -n celeryd3.myhost -c 3

# can also specify options for named workers
$ celeryd-multi start image video data -c 3 -c:image 10
celeryd -n image.myhost -c 10
celeryd -n video.myhost -c 3
celeryd -n data.myhost -c 3

# ranges and lists of workers in options is also allowed:
# (-c:1-3 can also be written as -c:1,2,3)
$ celeryd-multi start 5 -c 3  -c:1-3 10
celeryd-multi -n celeryd1.myhost -c 10
celeryd-multi -n celeryd2.myhost -c 10
celeryd-multi -n celeryd3.myhost -c 10
celeryd-multi -n celeryd4.myhost -c 3
celeryd-multi -n celeryd5.myhost -c 3

# lists also works with named workers
$ celeryd-multi start foo bar baz xuzzy -c 3 -c:foo,bar,baz 10
celeryd-multi -n foo.myhost -c 10
celeryd-multi -n bar.myhost -c 10
celeryd-multi -n baz.myhost -c 10
celeryd-multi -n xuzzy.myhost -c 3
```

- The worker now calls the result backends *process_cleanup* method *after* task execution instead of before.
- AMQP result backend now supports Pika.

2.15.7 Change history for Celery 1.0

1.0.6

release-date 2010-06-30 09:57 A.M CEST

- RabbitMQ 1.8.0 has extended their exchange equivalence tests to include *auto_delete* and *durable*. This broke the AMQP backend.

  If you've already used the AMQP backend this means you have to delete the previous definitions:

  ```
  $ camqadm exchange.delete celeryresults
  ```

  or:

  ```
  $ python manage.py camqadm exchange.delete celeryresults
  ```

1.0.5

release-date 2010-06-01 02:36 P.M CEST

Critical

- SIGINT/Ctrl+C killed the pool, abruptly terminating the currently executing tasks.

  Fixed by making the pool worker processes ignore `SIGINT`.

- Should not close the consumers before the pool is terminated, just cancel the consumers.

  See issue #122.

- Now depends on `billiard` >= 0.3.1
- celeryd: Previously exceptions raised by worker components could stall startup, now it correctly logs the exceptions and shuts down.
- celeryd: Prefetch counts was set too late. QoS is now set as early as possible, so celeryd can't slurp in all the messages at start-up.

Changes

- `celery.contrib.abortable`: Abortable tasks.

  Tasks that defines steps of execution, the task can then be aborted after each step has completed.

- `EventDispatcher`: No longer creates AMQP channel if events are disabled
- Added required RPM package names under *[bdist_rpm]* section, to support building RPMs from the sources using setup.py
- Running unit tests: `NOSE_VERBOSE` environment var now enables verbose output from Nose.
- `celery.execute.apply()`: Pass log file/log level arguments as task kwargs.

  See issue #110.

- celery.execute.apply: Should return exception, not `ExceptionInfo` on error.

  See issue #111.

- Added new entries to the *FAQs*:

- Should I use retry or acks_late?
- Can I call a task by name?

1.0.4

release-date 2010-05-31 09:54 A.M CEST

- Changelog merged with 1.0.5 as the release was never announced.

1.0.3

release-date 2010-05-15 03:00 P.M CEST

Important notes

- Messages are now acknowledged *just before* the task function is executed.

  This is the behavior we've wanted all along, but couldn't have because of limitations in the multiprocessing module. The previous behavior was not good, and the situation worsened with the release of 1.0.1, so this change will definitely improve reliability, performance and operations in general.

  For more information please see http://bit.ly/9hom6T

- Database result backend: result now explicitly sets *null=True* as *django-picklefield* version 0.1.5 changed the default behavior right under our noses :(

  See: http://bit.ly/d5OwMr

  This means those who created their celery tables (via syncdb or celeryinit) with picklefield versions >= 0.1.5 has to alter their tables to allow the result field to be *NULL* manually.

  MySQL:

  ```
  ALTER TABLE celery_taskmeta MODIFY result TEXT NULL
  ```

  PostgreSQL:

  ```
  ALTER TABLE celery_taskmeta ALTER COLUMN result DROP NOT NULL
  ```

- Removed *Task.rate_limit_queue_type*, as it was not really useful and made it harder to refactor some parts.
- Now depends on carrot >= 0.10.4
- Now depends on billiard >= 0.3.0

News

- AMQP backend: Added timeout support for *result.get()* / *result.wait()*.
- New task option: *Task.acks_late* (default: `CELERY_ACKS_LATE`)

Late ack means the task messages will be acknowledged **after** the task has been executed, not *just before*, which is the default behavior.

Note: This means the tasks may be executed twice if the worker crashes in mid-execution. Not acceptable for most applications, but desirable for others.

- Added crontab-like scheduling to periodic tasks.

  Like a cron job, you can specify units of time of when you would like the task to execute. While not a full implementation of cron's features, it should provide a fair degree of common scheduling needs.

  You can specify a minute (0-59), an hour (0-23), and/or a day of the week (0-6 where 0 is Sunday, or by names: sun, mon, tue, wed, thu, fri, sat).

  Examples:

```
from celery.schedules import crontab
from celery.decorators import periodic_task

@periodic_task(run_every=crontab(hour=7, minute=30))
def every_morning():
    print("Runs every morning at 7:30a.m")

@periodic_task(run_every=crontab(hour=7, minute=30, day_of_week="mon"))
def every_monday_morning():
    print("Run every monday morning at 7:30a.m")

@periodic_task(run_every=crontab(minutes=30))
def every_hour():
    print("Runs every hour on the clock. e.g. 1:30, 2:30, 3:30 etc.")
```

  Note: This a late addition. While we have unittests, due to the nature of this feature we haven't been able to completely test this in practice, so consider this experimental.

- *TaskPool.apply_async*: Now supports the *accept_callback* argument.
- *apply_async*: Now raises `ValueError` if task args is not a list, or kwargs is not a tuple (Issue #95).
- *Task.max_retries* can now be *None*, which means it will retry forever.
- Celerybeat: Now reuses the same connection when publishing large sets of tasks.
- Modified the task locking example in the documentation to use *cache.add* for atomic locking.
- Added experimental support for a *started* status on tasks.

  If *Task.track_started* is enabled the task will report its status as "started" when the task is executed by a worker.

  The default value is *False* as the normal behaviour is to not report that level of granularity. Tasks are either pending, finished, or waiting to be retried. Having a "started" status can be useful for when there are long running tasks and there is a need to report which task is currently running.

  The global default can be overridden by the `CELERY_TRACK_STARTED` setting.

- User Guide: New section *Tips and Best Practices*.

  Contributions welcome!

Remote control commands

- Remote control commands can now send replies back to the caller.

  Existing commands has been improved to send replies, and the client interface in *celery.task.control* has new keyword arguments: *reply*, *timeout* and *limit*. Where reply means it will wait for replies, timeout is the time in seconds to stop waiting for replies, and limit is the maximum number of replies to get.

  By default, it will wait for as many replies as possible for one second.

  - rate_limit(task_name, destination=all, reply=False, timeout=1, limit=0)

    Worker returns *{"ok": message}* on success, or *{"failure": message}* on failure.

    ```
    >>> from celery.task.control import rate_limit
    >>> rate_limit("tasks.add", "10/s", reply=True)
    [{'worker1': {'ok': 'new rate limit set successfully'}},
     {'worker2': {'ok': 'new rate limit set successfully'}}]
    ```

  - ping(destination=all, reply=False, timeout=1, limit=0)

    Worker returns the simple message *"pong"*.

    ```
    >>> from celery.task.control import ping
    >>> ping(reply=True)
    [{'worker1': 'pong'},
     {'worker2': 'pong'},
    ```

  - revoke(destination=all, reply=False, timeout=1, limit=0)

    Worker simply returns *True*.

    ```
    >>> from celery.task.control import revoke
    >>> revoke("419e46eb-cf6a-4271-86a8-442b7124132c", reply=True)
    [{'worker1': True},
     {'worker2'; True}]
    ```

- You can now add your own remote control commands!

  Remote control commands are functions registered in the command registry. Registering a command is done using `celery.worker.control.Panel.register()`:

  ```
  from celery.task.control import Panel

  @Panel.register
  def reset_broker_connection(panel, **kwargs):
      panel.consumer.reset_connection()
      return {"ok": "connection re-established"}
  ```

  With this module imported in the worker, you can launch the command using *celery.task.control.broadcast*:

  ```
  >>> from celery.task.control import broadcast
  >>> broadcast("reset_broker_connection", reply=True)
  [{'worker1': {'ok': 'connection re-established'},
   {'worker2': {'ok': 'connection re-established'}}]
  ```

TIP You can choose the worker(s) to receive the command by using the *destination* argument:

```
>>> broadcast("reset_broker_connection", destination=["worker1"])
[{'worker1': {'ok': 'connection re-established'}]
```

- New remote control command: *dump_reserved*

  Dumps tasks reserved by the worker, waiting to be executed:

```
>>> from celery.task.control import broadcast
>>> broadcast("dump_reserved", reply=True)
[{'myworker1': [<TaskRequest ....>]}]
```

- New remote control command: *dump_schedule*

  Dumps the workers currently registered ETA schedule. These are tasks with an *eta* (or *countdown*) argument waiting to be executed by the worker.

```
>>> from celery.task.control import broadcast
>>> broadcast("dump_schedule", reply=True)
[{'w1': []},
 {'w3': []},
 {'w2': ['0. 2010-05-12 11:06:00 pri0 <TaskRequest
            {name:"opalfeeds.tasks.refresh_feed_slice",
             id:"95b45760-4e73-4ce8-8eac-f100aa80273a",
             args:"(<Feeds freq_max:3600 freq_min:60
                         start:2184.0 stop:3276.0>,)",
             kwargs:"{'page': 2}"}>']},
 {'w4': ['0. 2010-05-12 11:00:00 pri0 <TaskRequest
            {name:"opalfeeds.tasks.refresh_feed_slice",
             id:"c053480b-58fb-422f-ae68-8d30a464edfe",
             args:"(<Feeds freq_max:3600 freq_min:60
                         start:1092.0 stop:2184.0>,)",
             kwargs:"{\'page\': 1}"}>',
       '1. 2010-05-12 11:12:00 pri0 <TaskRequest
            {name:"opalfeeds.tasks.refresh_feed_slice",
             id:"ab8bc59e-6cf8-44b8-88d0-f1af57789758",
             args:"(<Feeds freq_max:3600 freq_min:60
                         start:3276.0 stop:4365>,)",
             kwargs:"{\'page\': 3}"}>']}]
```

Fixes

- Mediator thread no longer blocks for more than 1 second.

  With rate limits enabled and when there was a lot of remaining time, the mediator thread could block shutdown (and potentially block other jobs from coming in).

- Remote rate limits was not properly applied (Issue #98).
- Now handles exceptions with Unicode messages correctly in *TaskRequest.on_failure*.
- Database backend: *TaskMeta.result*: default value should be *None* not empty string.

1.0.2

release-date 2010-03-31 12:50 P.M CET

- Deprecated: CELERY_BACKEND, please use CELERY_RESULT_BACKEND instead.
- We now use a custom logger in tasks. This logger supports task magic keyword arguments in formats.

  The default format for tasks (CELERYD_TASK_LOG_FORMAT) now includes the id and the name of tasks so the origin of task log messages can easily be traced.

  Example output::

  [2010-03-25 13:11:20,317: INFO/PoolWorker-1] [tasks.add(a6e1c5ad-60d9-42a0-8b24-9e39363125a4)] Hello from add

  To revert to the previous behavior you can set:

```
CELERYD_TASK_LOG_FORMAT = """
    [%(asctime)s: %(levelname)s/%(processName)s] %(message)s
""".strip()
```

- Unit tests: Don't disable the django test database tear down, instead fixed the underlying issue which was caused by modifications to the *DATABASE_NAME* setting (Issue #82).
- Django Loader: New config CELERY_DB_REUSE_MAX (max number of tasks to reuse the same database connection)

  The default is to use a new connection for every task. We would very much like to reuse the connection, but a safe number of reuses is not known, and we don't have any way to handle the errors that might happen, which may even be database dependent.

  See: http://bit.ly/94fwdd

- celeryd: The worker components are now configurable: CELERYD_POOL, CELERYD_CONSUMER, CELERYD_MEDIATOR, and CELERYD_ETA_SCHEDULER.

  The default configuration is as follows:

```
CELERYD_POOL = "celery.concurrency.processes.TaskPool"
CELERYD_MEDIATOR = "celery.worker.controllers.Mediator"
CELERYD_ETA_SCHEDULER = "celery.worker.controllers.ScheduleController"
CELERYD_CONSUMER = "celery.worker.consumer.Consumer"
```

  The CELERYD_POOL setting makes it easy to swap out the multiprocessing pool with a threaded pool, or how about a twisted/eventlet pool?

  Consider the competition for the first pool plug-in started!

- Debian init scripts: Use *-a* not *&&* (Issue #82).
- Debian init scripts: Now always preserves *$CELERYD_OPTS* from the */etc/default/celeryd* and */etc/default/celerybeat*.
- celery.beat.Scheduler: Fixed a bug where the schedule was not properly flushed to disk if the schedule had not been properly initialized.
- celerybeat: Now syncs the schedule to disk when receiving the SIGTERM and SIGINT signals.
- Control commands: Make sure keywords arguments are not in Unicode.
- ETA scheduler: Was missing a logger object, so the scheduler crashed when trying to log that a task had been revoked.

- management.commands.camqadm: Fixed typo *camqpadm* -> *camqadm* (Issue #83).
- PeriodicTask.delta_resolution: Was not working for days and hours, now fixed by rounding to the nearest day/hour.
- Fixed a potential infinite loop in *BaseAsyncResult.__eq__*, although there is no evidence that it has ever been triggered.
- celeryd: Now handles messages with encoding problems by acking them and emitting an error message.

1.0.1

release-date 2010-02-24 07:05 P.M CET

- Tasks are now acknowledged early instead of late.

  This is done because messages can only be acknowledged within the same connection channel, so if the connection is lost we would have to refetch the message again to acknowledge it.

  This might or might not affect you, but mostly those running tasks with a really long execution time are affected, as all tasks that has made it all the way into the pool needs to be executed before the worker can safely terminate (this is at most the number of pool workers, multiplied by the `CELERYD_PREFETCH_MULTIPLIER` setting.)

  We multiply the prefetch count by default to increase the performance at times with bursts of tasks with a short execution time. If this doesn't apply to your use case, you should be able to set the prefetch multiplier to zero, without sacrificing performance.

  Note: A patch to `multiprocessing` is currently being worked on, this patch would enable us to use a better solution, and is scheduled for inclusion in the *2.0.0* release.

- celeryd now shutdowns cleanly when receiving the `SIGTERM` signal.
- celeryd now does a cold shutdown if the `SIGINT` signal is received (Ctrl+C), this means it tries to terminate as soon as possible.
- Caching of results now moved to the base backend classes, so no need to implement this functionality in the base classes.
- Caches are now also limited in size, so their memory usage doesn't grow out of control.

  You can set the maximum number of results the cache can hold using the `CELERY_MAX_CACHED_RESULTS` setting (the default is five thousand results). In addition, you can refetch already retrieved results using *backend.reload_task_result* + *backend.reload_taskset_result* (that's for those who want to send results incrementally).

- *celeryd* now works on Windows again.

  Warning: If you're using Celery with Django, you can't use *project.settings* as the settings module name, but the following should work:

  ```
  $ python manage.py celeryd --settings=settings
  ```

- Execution: *.messaging.TaskPublisher.send_task* now incorporates all the functionality apply_async previously did.

  Like converting countdowns to eta, so `celery.execute.apply_async()` is now simply a convenient front-end to `celery.messaging.TaskPublisher.send_task()`, using the task classes default options.

Also `celery.execute.send_task()` has been introduced, which can apply tasks using just the task name (useful if the client does not have the destination task in its task registry).

Example:

```
>>> from celery.execute import send_task
>>> result = send_task("celery.ping", args=[], kwargs={})
>>> result.get()
'pong'
```

- *camqadm*: This is a new utility for command line access to the AMQP API.

  Excellent for deleting queues/bindings/exchanges, experimentation and testing:

  ```
  $ camqadm
  1> help
  ```

  Gives an interactive shell, type *help* for a list of commands.

  When using Django, use the management command instead:

  ```
  $ python manage.py camqadm
  1> help
  ```

- Redis result backend: To conform to recent Redis API changes, the following settings has been deprecated:
  - *REDIS_TIMEOUT*
  - *REDIS_CONNECT_RETRY*

  These will emit a *DeprecationWarning* if used.

  A *REDIS_PASSWORD* setting has been added, so you can use the new simple authentication mechanism in Redis.

- The redis result backend no longer calls *SAVE* when disconnecting, as this is apparently better handled by Redis itself.
- If *settings.DEBUG* is on, celeryd now warns about the possible memory leak it can result in.
- The ETA scheduler now sleeps at most two seconds between iterations.
- The ETA scheduler now deletes any revoked tasks it might encounter.

  As revokes are not yet persistent, this is done to make sure the task is revoked even though it's currently being hold because its eta is e.g. a week into the future.

- The *task_id* argument is now respected even if the task is executed eagerly (either using apply, or `CELERY_ALWAYS_EAGER`).
- The internal queues are now cleared if the connection is reset.
- New magic keyword argument: *delivery_info*.

  Used by retry() to resend the task to its original destination using the same exchange/routing_key.

- Events: Fields was not passed by *.send()* (fixes the UUID key errors in celerymon)
- Added *–schedule/-s* option to celeryd, so it is possible to specify a custom schedule filename when using an embedded celerybeat server (the *-B/–beat*) option.
- Better Python 2.4 compatibility. The test suite now passes.
- task decorators: Now preserve docstring as *cls.__doc__*, (was previously copied to *cls.run.__doc__*)

- The *testproj* directory has been renamed to *tests* and we're now using *nose* + *django-nose* for test discovery, and *unittest2* for test cases.
- New pip requirements files available in `requirements`.
- TaskPublisher: Declarations are now done once (per process).
- Added *Task.delivery_mode* and the `CELERY_DEFAULT_DELIVERY_MODE` setting.

  These can be used to mark messages non-persistent (i.e. so they are lost if the broker is restarted).
- Now have our own *ImproperlyConfigured* exception, instead of using the Django one.
- Improvements to the Debian init scripts: Shows an error if the program is not executable. Does not modify *CELERYD* when using django with virtualenv.

1.0.0

release-date 2010-02-10 04:00 P.M CET

Backward incompatible changes

- Celery does not support detaching anymore, so you have to use the tools available on your platform, or something like Supervisord to make celeryd/celerybeat/celerymon into background processes.

  We've had too many problems with celeryd daemonizing itself, so it was decided it has to be removed. Example startup scripts has been added to the *extra/* directory:

  - Debian, Ubuntu, (start-stop-daemon)

    extra/debian/init.d/celeryd extra/debian/init.d/celerybeat

  - Mac OS X launchd

    extra/mac/org.celeryq.celeryd.plist extra/mac/org.celeryq.celerybeat.plist extra/mac/org.celeryq.celerymon.plist

  - Supervisord (http://supervisord.org)

    extra/supervisord/supervisord.conf

  In addition to *–detach*, the following program arguments has been removed: *–uid*, *–gid*, *–workdir*, *–chroot*, *–pidfile*, *–umask*. All good daemonization tools should support equivalent functionality, so don't worry.

  Also the following configuration keys has been removed: *CELERYD_PID_FILE*, *CELERYBEAT_PID_FILE*, *CELERYMON_PID_FILE*.
- Default celeryd loglevel is now *WARN*, to enable the previous log level start celeryd with *–loglevel=INFO*.
- Tasks are automatically registered.

  This means you no longer have to register your tasks manually. You don't have to change your old code right away, as it doesn't matter if a task is registered twice.

  If you don't want your task to be automatically registered you can set the *abstract* attribute

```
class MyTask(Task):
    abstract = True
```

By using *abstract* only tasks subclassing this task will be automatically registered (this works like the Django ORM).

If you don't want subclasses to be registered either, you can set the *autoregister* attribute to *False*.

Incidentally, this change also fixes the problems with automatic name assignment and relative imports. So you also don't have to specify a task name anymore if you use relative imports.

- You can no longer use regular functions as tasks.

  This change was added because it makes the internals a lot more clean and simple. However, you can now turn functions into tasks by using the *@task* decorator:

  ```
  from celery.decorators import task

  @task()
  def add(x, y):
      return x + y
  ```

  See also:

  Tasks for more information about the task decorators.

- The periodic task system has been rewritten to a centralized solution.

  This means *celeryd* no longer schedules periodic tasks by default, but a new daemon has been introduced: *celerybeat*.

  To launch the periodic task scheduler you have to run celerybeat:

  ```
  $ celerybeat
  ```

  Make sure this is running on one server only, if you run it twice, all periodic tasks will also be executed twice.

  If you only have one worker server you can embed it into celeryd like this:

  ```
  $ celeryd --beat # Embed celerybeat in celeryd.
  ```

- The supervisor has been removed.

  This means the *-S* and *–supervised* options to *celeryd* is no longer supported. Please use something like http://supervisord.org instead.

- *TaskSet.join* has been removed, use *TaskSetResult.join* instead.
- The task status *"DONE"* has been renamed to *"SUCCESS"*.
- *AsyncResult.is_done* has been removed, use *AsyncResult.successful* instead.
- The worker no longer stores errors if *Task.ignore_result* is set, to revert to the previous behaviour set `CELERY_STORE_ERRORS_EVEN_IF_IGNORED` to *True*.
- The statistics functionality has been removed in favor of events, so the *-S* and –statistics‘ switches has been removed.
- The module *celery.task.strategy* has been removed.
- *celery.discovery* has been removed, and it's *autodiscover* function is now in *celery.loaders.djangoapp*. Reason: Internal API.
- The `CELERY_LOADER` environment variable now needs loader class name in addition to module name,

E.g. where you previously had: *"celery.loaders.default"*, you now need *"celery.loaders.default.Loader"*, using the previous syntax will result in a *DeprecationWarning*.

- Detecting the loader is now lazy, and so is not done when importing *celery.loaders*.

  To make this happen *celery.loaders.settings* has been renamed to *load_settings* and is now a function returning the settings object. *celery.loaders.current_loader* is now also a function, returning the current loader.

  So:

  ```
  loader = current_loader
  ```

  needs to be changed to:

  ```
  loader = current_loader()
  ```

Deprecations

- The following configuration variables has been renamed and will be deprecated in v2.0:
  - CELERYD_DAEMON_LOG_FORMAT -> CELERYD_LOG_FORMAT
  - CELERYD_DAEMON_LOG_LEVEL -> CELERYD_LOG_LEVEL
  - CELERY_AMQP_CONNECTION_TIMEOUT -> CELERY_BROKER_CONNECTION_TIMEOUT
  - CELERY_AMQP_CONNECTION_RETRY -> CELERY_BROKER_CONNECTION_RETRY
  - CELERY_AMQP_CONNECTION_MAX_RETRIES -> CELERY_BROKER_CONNECTION_MAX_RETRIES
  - SEND_CELERY_TASK_ERROR_EMAILS -> CELERY_SEND_TASK_ERROR_EMAILS
- The public API names in celery.conf has also changed to a consistent naming scheme.
- We now support consuming from an arbitrary number of queues.

  To do this we had to rename the configuration syntax. If you use any of the custom AMQP routing options (queue/exchange/routing_key, etc.), you should read the new FAQ entry: *Can I send some tasks to only some servers?*.

  The previous syntax is deprecated and scheduled for removal in v2.0.

- *TaskSet.run* has been renamed to *TaskSet.apply_async*.

  TaskSet.run has now been deprecated, and is scheduled for removal in v2.0.

News

- Rate limiting support (per task type, or globally).
- New periodic task system.
- Automatic registration.
- New cool task decorator syntax.
- celeryd now sends events if enabled with the *-E* argument.

Excellent for monitoring tools, one is already in the making (http://github.com/celery/celerymon).

Current events include: `worker-heartbeat`, task-[received/succeeded/failed/retried], `worker-online`, `worker-offline`.

- You can now delete (revoke) tasks that has already been applied.
- You can now set the hostname celeryd identifies as using the *–hostname* argument.
- Cache backend now respects the `CELERY_TASK_RESULT_EXPIRES` setting.
- Message format has been standardized and now uses ISO-8601 format for dates instead of datetime.
- *celeryd* now responds to the `SIGHUP` signal by restarting itself.
- Periodic tasks are now scheduled on the clock.

  I.e. *timedelta(hours=1)* means every hour at :00 minutes, not every hour from the server starts. To revert to the previous behaviour you can set *PeriodicTask.relative = True*.

- Now supports passing execute options to a TaskSets list of args, e.g.:

```
>>> ts = TaskSet(add, [([2, 2], {}, {"countdown": 1}),
...                    ([4, 4], {}, {"countdown": 2}),
...                    ([8, 8], {}, {"countdown": 3})])
>>> ts.run()
```

- Got a 3x performance gain by setting the prefetch count to four times the concurrency, (from an average task round-trip of 0.1s to 0.03s!).

  A new setting has been added: `CELERYD_PREFETCH_MULTIPLIER`, which is set to *4* by default.

- Improved support for webhook tasks.

  celery.task.rest is now deprecated, replaced with the new and shiny `celery.task.http`. With more reflective names, sensible interface, and it's possible to override the methods used to perform HTTP requests.

- The results of task sets are now cached by storing it in the result backend.

Changes

- Now depends on carrot >= 0.8.1
- New dependencies: billiard, python-dateutil, django-picklefield
- No longer depends on python-daemon
- The *uuid* distribution is added as a dependency when running Python 2.4.
- Now remembers the previously detected loader by keeping it in the `CELERY_LOADER` environment variable.

  This may help on windows where fork emulation is used.

- ETA no longer sends datetime objects, but uses ISO 8601 date format in a string for better compatibility with other platforms.
- No longer sends error mails for retried tasks.
- Task can now override the backend used to store results.
- Refactored the ExecuteWrapper, *apply* and `CELERY_ALWAYS_EAGER` now also executes the task callbacks and signals.
- Now using a proper scheduler for the tasks with an ETA.

This means waiting eta tasks are sorted by time, so we don't have to poll the whole list all the time.

- Now also imports modules listed in CELERY_IMPORTS when running with django (as documented).
- Log level for stdout/stderr changed from INFO to ERROR
- ImportErrors are now properly propagated when autodiscovering tasks.
- You can now use *celery.messaging.establish_connection* to establish a connection to the broker.
- When running as a separate service the periodic task scheduler does some smart moves to not poll too regularly.

  If you need faster poll times you can lower the value of CELERYBEAT_MAX_LOOP_INTERVAL.
- You can now change periodic task intervals at runtime, by making *run_every* a property, or subclassing *PeriodicTask.is_due*.
- The worker now supports control commands enabled through the use of a broadcast queue, you can remotely revoke tasks or set the rate limit for a task type. See celery.task.control.
- The services now sets informative process names (as shown in *ps* listings) if the setproctitle module is installed.
- NotRegistered now inherits from KeyError, and *TaskRegistry.__getitem__'+'pop* raises *NotRegistered* instead
- You can set the loader via the CELERY_LOADER environment variable.
- You can now set CELERY_IGNORE_RESULT to ignore task results by default (if enabled, tasks doesn't save results or errors to the backend used).
- celeryd now correctly handles malformed messages by throwing away and acknowledging the message, instead of crashing.

Bugs

- Fixed a race condition that could happen while storing task results in the database.

Documentation

- Reference now split into two sections; API reference and internal module reference.

0.8.4

release-date 2010-02-05 01:52 P.M CEST

- Now emits a warning if the –detach argument is used. –detach should not be used anymore, as it has several not easily fixed bugs related to it. Instead, use something like start-stop-daemon, Supervisord or launchd (os x).
- Make sure logger class is process aware, even if running Python >= 2.6.
- Error emails are not sent anymore when the task is retried.

0.8.3

release-date 2009-12-22 09:43 A.M CEST

- Fixed a possible race condition that could happen when storing/querying task results using the database backend.

- Now has console script entry points in the setup.py file, so tools like Buildout will correctly install the programs celeryd and celeryinit.

0.8.2

release-date 2009-11-20 03:40 P.M CEST

- QOS Prefetch count was not applied properly, as it was set for every message received (which apparently behaves like, "receive one more"), instead of only set when our wanted value changed.

0.8.1

release-date 2009-11-16 05:21 P.M CEST

Very important note

This release (with carrot 0.8.0) enables AMQP QoS (quality of service), which means the workers will only receive as many messages as it can handle at a time. As with any release, you should test this version upgrade on your development servers before rolling it out to production!

Important changes

- If you're using Python < 2.6 and you use the multiprocessing backport, then multiprocessing version 2.6.2.1 is required.
- All AMQP_* settings has been renamed to BROKER_*, and in addition AMQP_SERVER has been renamed to BROKER_HOST, so before where you had:

```
AMQP_SERVER = "localhost"
AMQP_PORT = 5678
AMQP_USER = "myuser"
AMQP_PASSWORD = "mypassword"
AMQP_VHOST = "celery"
```

You need to change that to:

```
BROKER_HOST = "localhost"
BROKER_PORT = 5678
BROKER_USER = "myuser"
BROKER_PASSWORD = "mypassword"
BROKER_VHOST = "celery"
```

- Custom carrot backends now need to include the backend class name, so before where you had:

```
CARROT_BACKEND = "mycustom.backend.module"
```

you need to change it to:

```
CARROT_BACKEND = "mycustom.backend.module.Backend"
```

where *Backend* is the class name. This is probably *"Backend"*, as that was the previously implied name.

- New version requirement for carrot: 0.8.0

Changes

- Incorporated the multiprocessing backport patch that fixes the *processName* error.
- Ignore the result of PeriodicTask's by default.
- Added a Redis result store backend
- Allow /etc/default/celeryd to define additional options for the celeryd init script.
- MongoDB periodic tasks issue when using different time than UTC fixed.
- Windows specific: Negate test for available os.fork (thanks miracle2k)
- Now tried to handle broken PID files.
- Added a Django test runner to contrib that sets *CELERY_ALWAYS_EAGER = True* for testing with the database backend.
- Added a `CELERY_CACHE_BACKEND` setting for using something other than the django-global cache backend.
- Use custom implementation of functools.partial (curry) for Python 2.4 support (Probably still problems with running on 2.4, but it will eventually be supported)
- Prepare exception to pickle when saving `RETRY` status for all backends.
- SQLite no concurrency limit should only be effective if the database backend is used.

0.8.0

release-date 2009-09-22 03:06 P.M CEST

Backward incompatible changes

- Add traceback to result value on failure.

  Note: If you use the database backend you have to re-create the database table *celery_taskmeta*.

  Contact the *Mailing list* or *IRC* channel for help doing this.

- Database tables are now only created if the database backend is used, so if you change back to the database backend at some point, be sure to initialize tables (django: *syncdb*, python: *celeryinit*).

  Note: This is only applies if using Django version 1.1 or higher.

- Now depends on *carrot* version 0.6.0.
- Now depends on python-daemon 1.4.8

Important changes

- Celery can now be used in pure Python (outside of a Django project).

  This means celery is no longer Django specific.

  For more information see the FAQ entry *Is Celery for Django only?*.

- Celery now supports task retries.

  See *Retrying* for more information.

- We now have an AMQP result store backend.

  It uses messages to publish task return value and status. And it's incredibly fast!

  See issue #6 for more info!

- AMQP QoS (prefetch count) implemented:

  This to not receive more messages than we can handle.

- Now redirects stdout/stderr to the celeryd log file when detached

- **Now uses *inspect.getargspec* to only pass default arguments** the task supports.

- **Add Task.on_success, .on_retry, .on_failure handlers**

  See `celery.task.base.Task.on_success()`, `celery.task.base.Task.on_retry()`, `celery.task.base.Task.on_failure()`,

- ***celery.utils.gen_unique_id*: Workaround for** http://bugs.python.org/issue4607

- **You can now customize what happens at worker start, at process init, etc.,** by creating your own loaders. (see `celery.loaders.default`, `celery.loaders.djangoapp`, `celery.loaders`.)

- Support for multiple AMQP exchanges and queues.

  This feature misses documentation and tests, so anyone interested is encouraged to improve this situation.

- celeryd now survives a restart of the AMQP server!

  Automatically re-establish AMQP broker connection if it's lost.

  New settings:

  - **AMQP_CONNECTION_RETRY** Set to *True* to enable connection retries.
  - **AMQP_CONNECTION_MAX_RETRIES.** Maximum number of restarts before we give up. Default: *100*.

News

- **Fix an incompatibility between python-daemon and multiprocessing,** which resulted in the *[Errno 10] No child processes* problem when detaching.
- **Fixed a possible DjangoUnicodeDecodeError being raised when saving pickled** data to Django's memcached cache backend.
- Better Windows compatibility.
- **New version of the pickled field (taken from** http://www.djangosnippets.org/snippets/513/)
- **New signals introduced: *task_sent*, *task_prerun* and** *task_postrun*, see `celery.signals` for more information.
- ***TaskSetResult.join* caused *TypeError* when *timeout=None*.** Thanks Jerzy Kozera. Closes #31
- ***views.apply* should return *HttpResponse* instance.** Thanks to Jerzy Kozera. Closes #32
- ***PeriodicTask*: Save conversion of *run_every* from *int*** to *timedelta* to the class attribute instead of on the instance.

- **Exceptions has been moved to *celery.exceptions*, but are still** available in the previous module.
- **Try to rollback transaction and retry saving result if an error happens** while setting task status with the database backend.
- jail() refactored into `celery.execute.ExecuteWrapper`.
- *views.apply* now correctly sets mime-type to "application/json"
- *views.task_status* now returns exception if state is `RETRY`
- ***views.task_status* now returns traceback if state is `FAILURE`** or `RETRY`
- Documented default task arguments.
- Add a sensible __repr__ to ExceptionInfo for easier debugging
- **Fix documentation typo .. *import map* -> .. *import dmap*.** Thanks to mikedizon

0.6.0

release-date 2009-08-07 06:54 A.M CET

Important changes

- **Fixed a bug where tasks raising unpickleable exceptions crashed pool** workers. So if you've had pool workers mysteriously disappearing, or problems with celeryd stopping working, this has been fixed in this version.
- Fixed a race condition with periodic tasks.
- **The task pool is now supervised, so if a pool worker crashes,** goes away or stops responding, it is automatically replaced with a new one.
- **Task.name is now automatically generated out of class module+name, e.g.** *"djangotwitter.tasks.UpdateStatusesTask"*. Very convenient. No idea why we didn't do this before. Some documentation is updated to not manually specify a task name.

News

- Tested with Django 1.1
- New Tutorial: Creating a click counter using carrot and celery
- **Database entries for periodic tasks are now created at *celeryd*** startup instead of for each check (which has been a forgotten TODO/XXX in the code for a long time)
- **New settings variable: `CELERY_TASK_RESULT_EXPIRES`** Time (in seconds, or a *datetime.timedelta* object) for when after stored task results are deleted. For the moment this only works for the database backend.
- ***celeryd* now emits a debug log message for which periodic tasks** has been launched.
- **The periodic task table is now locked for reading while getting** periodic task status. (MySQL only so far, seeking patches for other engines)
- **A lot more debugging information is now available by turning on the** *DEBUG* log level (*–loglevel=DEBUG*).
- Functions/methods with a timeout argument now works correctly.

- **New: *celery.strategy.even_time_distribution*:** With an iterator yielding task args, kwargs tuples, evenly distribute the processing of its tasks throughout the time window available.
- Log message *Unknown task ignored...* now has log level *ERROR*
- **Log message *"Got task from broker"* is now emitted for all tasks, even if** the task has an ETA (estimated time of arrival). Also the message now includes the ETA for the task (if any).
- **Acknowledgement now happens in the pool callback. Can't do ack in the job** target, as it's not pickleable (can't share AMQP connection, etc.)).
- Added note about .delay hanging in README
- Tests now passing in Django 1.1
- Fixed discovery to make sure app is in INSTALLED_APPS
- **Previously overridden pool behavior (process reap, wait until pool worker** available, etc.) is now handled by *multiprocessing.Pool* itself.
- Convert statistics data to Unicode for use as kwargs. Thanks Lucy!

0.4.1

release-date 2009-07-02 01:42 P.M CET

- Fixed a bug with parsing the message options (*mandatory*, *routing_key*, *priority*, *immediate*)

0.4.0

release-date 2009-07-01 07:29 P.M CET

- Adds eager execution. *celery.execute.apply*‘|‘*Task.apply* executes the function blocking until the task is done, for API compatibility it returns an *celery.result.EagerResult* instance. You can configure celery to always run tasks locally by setting the `CELERY_ALWAYS_EAGER` setting to *True*.
- Now depends on *anyjson*.
- 99% coverage using python *coverage* 3.0.

0.3.20

release-date 2009-06-25 08:42 P.M CET

- New arguments to *apply_async* (the advanced version of *delay_task*), *countdown* and *eta*;

```
>>> # Run 10 seconds into the future.
>>> res = apply_async(MyTask, countdown=10);


>>> # Run 1 day from now
>>> res = apply_async(MyTask,
...                   eta=datetime.now() + timedelta(days=1))
```

- Now unlinks stale PID files
- Lots of more tests.
- Now compatible with carrot >= 0.5.0.

- **IMPORTANT** The *subtask_ids* attribute on the *TaskSetResult* instance has been removed. To get this information instead use:

```
>>> subtask_ids = [subtask.id for subtask in ts_res.subtasks]
```

- *Taskset.run()* now respects extra message options from the task class.
- Task: Add attribute *ignore_result*: Don't store the status and return value. This means you can't use the *celery.result.AsyncResult* to check if the task is done, or get its return value. Only use if you need the performance and is able live without these features. Any exceptions raised will store the return value/status as usual.
- Task: Add attribute *disable_error_emails* to disable sending error emails for that task.
- Should now work on Windows (although running in the background won't work, so using the *–detach* argument results in an exception being raised.)
- Added support for statistics for profiling and monitoring. To start sending statistics start *celeryd* with the *–statistics option. Then after a while you can dump the results by running 'python manage.py celerystats*. See *celery.monitoring* for more information.
- The celery daemon can now be supervised (i.e. it is automatically restarted if it crashes). To use this start celeryd with the –supervised' option (or alternatively *-S*).
- views.apply: View calling a task. Example

```
http://e.com/celery/apply/task_name/arg1/arg2//?kwarg1=a&kwarg2=b
```

> **Warning:** Use with caution! Do not expose this URL to the public without first ensuring that your code is safe!

- Refactored *celery.task*. It's now split into three modules:
  - celery.task

    Contains *apply_async*, *delay_task*, *discard_all*, and task shortcuts, plus imports objects from *celery.task.base* and *celery.task.builtins*
  - celery.task.base

    Contains task base classes: *Task*, *PeriodicTask*, *TaskSet*, *AsynchronousMapTask*, *ExecuteRemoteTask*.
  - celery.task.builtins

    Built-in tasks: *PingTask*, *DeleteExpiredTaskMetaTask*.

0.3.7

release-date 2008-06-16 11:41 P.M CET

- **IMPORTANT** Now uses AMQP's *basic.consume* instead of *basic.get*. This means we're no longer polling the broker for new messages.
- **IMPORTANT** Default concurrency limit is now set to the number of CPUs available on the system.
- **IMPORTANT** *tasks.register*: Renamed *task_name* argument to *name*, so

```
>>> tasks.register(func, task_name="mytask")
```

has to be replaced with:

```
>>> tasks.register(func, name="mytask")
```

- The daemon now correctly runs if the pidlock is stale.
- Now compatible with carrot 0.4.5
- Default AMQP connection timeout is now 4 seconds.
- *AsyncResult.read()* was always returning *True*.
- Only use README as long_description if the file exists so easy_install doesn't break.
- *celery.view*: JSON responses now properly set its mime-type.
- *apply_async* now has a *connection* keyword argument so you can re-use the same AMQP connection if you want to execute more than one task.
- Handle failures in task_status view such that it won't throw 500s.
- Fixed typo *AMQP_SERVER* in documentation to *AMQP_HOST*.
- Worker exception emails sent to administrators now works properly.
- No longer depends on *django*, so installing *celery* won't affect the preferred Django version installed.
- Now works with PostgreSQL (psycopg2) again by registering the *PickledObject* field.
- *celeryd*: Added *–detach* option as an alias to *–daemon*, and it's the term used in the documentation from now on.
- Make sure the pool and periodic task worker thread is terminated properly at exit. (So *Ctrl-C* works again).
- Now depends on *python-daemon*.
- Removed dependency to *simplejson*
- Cache Backend: Re-establishes connection for every task process if the Django cache backend is memcached/libmemcached.
- Tyrant Backend: Now re-establishes the connection for every task executed.

0.3.3

release-date 2009-06-08 01:07 P.M CET

- The *PeriodicWorkController* now sleeps for 1 second between checking for periodic tasks to execute.

0.3.2

release-date 2009-06-08 01:07 P.M CET

- celeryd: Added option *–discard*: Discard (delete!) all waiting messages in the queue.
- celeryd: The *–wakeup-after* option was not handled as a float.

0.3.1

release-date 2009-06-08 01:07 P.M CET

- The *PeriodicTask* worker is now running in its own thread instead of blocking the *TaskController* loop.
- Default *QUEUE_WAKEUP_AFTER* has been lowered to *0.1* (was *0.3*)

0.3.0

release-date 2009-06-08 12:41 P.M CET

Warning: This is a development version, for the stable release, please see versions 0.2.x.

VERY IMPORTANT: Pickle is now the encoder used for serializing task arguments, so be sure to flush your task queue before you upgrade.

- **IMPORTANT** TaskSet.run() now returns a celery.result.TaskSetResult instance, which lets you inspect the status and return values of a taskset as it was a single entity.
- **IMPORTANT** Celery now depends on carrot >= 0.4.1.
- The celery daemon now sends task errors to the registered admin emails. To turn off this feature, set *SEND_CELERY_TASK_ERROR_EMAILS* to *False* in your *settings.py*. Thanks to Grégoire Cachet.
- You can now run the celery daemon by using *manage.py*:

  ```
  $ python manage.py celeryd
  ```

  Thanks to Grégoire Cachet.
- Added support for message priorities, topic exchanges, custom routing keys for tasks. This means we have introduced *celery.task.apply_async*, a new way of executing tasks.

  You can use *celery.task.delay* and *celery.Task.delay* like usual, but if you want greater control over the message sent, you want *celery.task.apply_async* and *celery.Task.apply_async*.

  This also means the AMQP configuration has changed. Some settings has been renamed, while others are new:

  ```
  CELERY_AMQP_EXCHANGE
  CELERY_AMQP_PUBLISHER_ROUTING_KEY
  CELERY_AMQP_CONSUMER_ROUTING_KEY
  CELERY_AMQP_CONSUMER_QUEUE
  CELERY_AMQP_EXCHANGE_TYPE
  ```

  See the entry *Can I send some tasks to only some servers?* in the *FAQ* for more information.
- Task errors are now logged using log level *ERROR* instead of *INFO*, and stacktraces are dumped. Thanks to Grégoire Cachet.
- Make every new worker process re-establish it's Django DB connection, this solving the "MySQL connection died?" exceptions. Thanks to Vitaly Babiy and Jirka Vejrazka.
- **IMPORTANT** Now using pickle to encode task arguments. This means you now can pass complex python objects to tasks as arguments.
- Removed dependency to *yadayada*.
- Added a FAQ, see *docs/faq.rst*.
- Now converts any Unicode keys in task *kwargs* to regular strings. Thanks Vitaly Babiy.
- Renamed the *TaskDaemon* to *WorkController*.
- *celery.datastructures.TaskProcessQueue* is now renamed to *celery.pool.TaskPool*.
- The pool algorithm has been refactored for greater performance and stability.

0.2.0

release-date 2009-05-20 05:14 P.M CET

- Final release of 0.2.0
- Compatible with carrot version 0.4.0.
- Fixes some syntax errors related to fetching results from the database backend.

0.2.0-pre3

release-date 2009-05-20 05:14 P.M CET

- *Internal release*. Improved handling of unpickleable exceptions, *get_result* now tries to recreate something looking like the original exception.

0.2.0-pre2

release-date 2009-05-20 01:56 P.M CET

- Now handles unpickleable exceptions (like the dynamically generated subclasses of *django.core.exception.MultipleObjectsReturned*).

0.2.0-pre1

release-date 2009-05-20 12:33 P.M CET

- It's getting quite stable, with a lot of new features, so bump version to 0.2. This is a pre-release.
- *celery.task.mark_as_read()* and *celery.task.mark_as_failure()* has been removed. Use *celery.backends.default_backend.mark_as_read()*, and *celery.backends.default_backend.mark_as_failure()* instead.

0.1.15

release-date 2009-05-19 04:13 P.M CET

- The celery daemon was leaking AMQP connections, this should be fixed, if you have any problems with too many files open (like *emfile* errors in *rabbit.log*, please contact us!

0.1.14

release-date 2009-05-19 01:08 P.M CET

- Fixed a syntax error in the *TaskSet* class. (No such variable *TimeOutError*).

0.1.13

release-date 2009-05-19 12:36 P.M CET

- Forgot to add *yadayada* to install requirements.
- Now deletes all expired task results, not just those marked as done.

- Able to load the Tokyo Tyrant backend class without django configuration, can specify tyrant settings directly in the class constructor.
- Improved API documentation
- Now using the Sphinx documentation system, you can build the html documentation by doing:

```
$ cd docs
$ make html
```

and the result will be in *docs/.build/html*.

0.1.12

release-date 2009-05-18 04:38 P.M CET

- *delay_task()* etc. now returns *celery.task.AsyncResult* object, which lets you check the result and any failure that might have happened. It kind of works like the *multiprocessing.AsyncResult* class returned by *multiprocessing.Pool.map_async*.
- Added dmap() and dmap_async(). This works like the *multiprocessing.Pool* versions except they are tasks distributed to the celery server. Example:

```
>>> from celery.task import dmap
>>> import operator
>>> dmap(operator.add, [[2, 2], [4, 4], [8, 8]])
>>> [4, 8, 16]
```

```
>>> from celery.task import dmap_async
>>> import operator
>>> result = dmap_async(operator.add, [[2, 2], [4, 4], [8, 8]])
>>> result.ready()
False
>>> time.sleep(1)
>>> result.ready()
True
>>> result.result
[4, 8, 16]
```

- Refactored the task metadata cache and database backends, and added a new backend for Tokyo Tyrant. You can set the backend in your django settings file. E.g.:

```
CELERY_RESULT_BACKEND = "database"; # Uses the database
CELERY_RESULT_BACKEND = "cache"; # Uses the django cache framework
CELERY_RESULT_BACKEND = "tyrant"; # Uses Tokyo Tyrant
TT_HOST = "localhost"; # Hostname for the Tokyo Tyrant server.
TT_PORT = 6657; # Port of the Tokyo Tyrant server.
```

0.1.11

release-date 2009-05-12 02:08 P.M CET

- The logging system was leaking file descriptors, resulting in servers stopping with the EMFILES (too many open files) error. (fixed)

0.1.10

release-date 2009-05-11 12:46 P.M CET

- Tasks now supports both positional arguments and keyword arguments.
- Requires carrot 0.3.8.
- The daemon now tries to reconnect if the connection is lost.

0.1.8

release-date 2009-05-07 12:27 P.M CET

- Better test coverage
- More documentation
- celeryd doesn't emit *Queue is empty* message if *settings.CELERYD_EMPTY_MSG_EMIT_EVERY* is 0.

0.1.7

release-date 2009-04-30 01:50 P.M CET

- Added some unit tests
- Can now use the database for task metadata (like if the task has been executed or not). Set *settings.CELERY_TASK_META*
- Can now run *python setup.py test* to run the unit tests from within the *tests* project.
- Can set the AMQP exchange/routing key/queue using *settings.CELERY_AMQP_EXCHANGE*, *settings.CELERY_AMQP_ROUTING_KEY*, and *settings.CELERY_AMQP_CONSUMER_QUEUE*.

0.1.6

release-date 2009-04-28 02:13 P.M CET

- Introducing *TaskSet*. A set of subtasks is executed and you can find out how many, or if all them, are done (excellent for progress bars and such)
- Now catches all exceptions when running *Task.__call__*, so the daemon doesn't die. This doesn't happen for pure functions yet, only *Task* classes.
- *autodiscover()* now works with zipped eggs.
- celeryd: Now adds current working directory to *sys.path* for convenience.
- The *run_every* attribute of *PeriodicTask* classes can now be a *datetime.timedelta()* object.
- celeryd: You can now set the *DJANGO_PROJECT_DIR* variable for *celeryd* and it will add that to *sys.path* for easy launching.
- Can now check if a task has been executed or not via HTTP.
- You can do this by including the celery *urls.py* into your project,

```
>>> url(r'^celery/$', include("celery.urls"))
```

then visiting the following url,:

```
http://mysite/celery/$task_id/done/
```

this will return a JSON dictionary like e.g:

```
>>> {"task": {"id": $task_id, "executed": true}}
```

- *delay_task* now returns string id, not *uuid.UUID* instance.
- Now has *PeriodicTasks*, to have *cron* like functionality.
- Project changed name from *crunchy* to *celery*. The details of the name change request is in *docs/name_change_request.txt*.

0.1.0

release-date 2009-04-24 11:28 A.M CET

- Initial release

2.16 Glossary

ack Short for *acknowledged*.

acknowledged Workers acknowledge messages to signify that a message has been handled. Failing to acknowledge a message will cause the message to be redelivered. Exactly when a transaction is considered a failure varies by transport. In AMQP the transaction fails when the connection/channel is closed (or lost), but in Redis/SQS the transaction times out after a configurable amount of time (the `visibility_timeout`).

apply Originally a synonym to *call* but used to signify that a function is executed by the current process.

calling Sends a task message so that the task function is *executed* by a worker.

context The context of a task contains information like the id of the task, it's arguments and what queue it was delivered to. It can be accessed as the tasks `request` attribute. See *Context*

executing Workers *execute* task *requests*.

idempotent Idempotence is a mathematical property that describes a function that can be called multiple times without changing the result. Practically it means that a function can be repeated many times without unintented effects, but not necessarily side-effect free in the pure sense (compare to *nullipotent*).

nullipotent describes a function that will have the same effect, and give the same result, even if called zero or multiple times (side-effect free). A stronger version of *idempotent*.

request Task messages are converted to *requests* within the worker. The request information is also available as the task's *context* (the `task.request` attribute).

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*

[AOC1] Breshears, Clay. Section 2.2.1, “The Art of Concurrency”. O’Reilly Media, Inc. May 15, 2009. ISBN-13 978-0-596-52153-0.

C