Celery文档 阅读笔记

夏永锋

October 22, 2013

Contents

1 Get Started

1.1 Introduction to Celery

Celery是一个简单、灵活且可靠的分布式系统，用于处理数量巨大的消息，同时提供维护这样一个系统所需要的工具。Celery是一个消息队列，侧重于实时处理，但也支持任务调度。

1.1.1 What is a Task Queue?

任务队列是一种在多线程或多机器之间分配任务的机制。任务队列的输入是一个工作单元，也称为一个任务，然后专用的工作者进程持续地监听该队列，看是否有新的工作需要完成。Celery使用一个中间人（Broker）在客户端（消息生产者）和工作者（Worker，消息消费者）之间传递消息。客户端在对列中放入一个消息来开始一个任务，中间人将该消息分发到一个工作者。

一个Celery系统可以包含多个工作者和中间人，以实现高可用和水平扩展。

1.1.2 What do I need?

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.

1.2 Brokers

1.2.1 Using RabbitMQ

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//'
```

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 ".*" ".*" ".*"
```

1.2.2 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

Just configure the location of your Redis database:

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

1.3 First Steps with Celery

1.3.1 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.

tasks.py

```
from celery import Celery

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

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

1.3.2 Running the celery worker server

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

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

1.3.3 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:

```
>>> 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**.

1.3.4 Keeping Results

If you want to keep track of the tasks' states, Celery needs to store or send the states somewhere.

1.4 Next Steps

1.4.1 Using Celery in your application

Our Project

```
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 `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
```

1.4.2 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.

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.

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 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'
```

1.4.3 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)
```

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.

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
```

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

1.4.4 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
```

1.4.5 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.

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
```

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.

2 User Guide

2.1 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`).

2.1.1 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.

2.1.2 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.

2.2 Tasks

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(幂等的), 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..

2.2.1 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
```

2.2.2 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', )
```

2.2.3 Logging

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

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).

2.2.4 How it works

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.

2.3 Calling Tasks

2.3.1 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))
```

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.

2.3.2 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.

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)
```

2.3.3 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)
```

2.3.4 Serializers

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.

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')
```

2.3.5 Compression

Celery can compress the messages using either *gzip*, or *bzip2*.

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` setting.

Example specifying the compression used when calling a task:

```
>>> add.apply_async((2, 2), compression='zlib')
```

2.4 Works Guide