Star 11,909

Please help support this community project with a donation:

Donate

Previous topic

Canvas: Designing Work-flows

Next topic

Daemonization

This Page

Show Source

Quick search

Go

YOUR AD HERE

Reach over 7 million devs each month when you advertise with Read the Docs.

Sponsored · Ads served ethically

This document describes the current stable version of Celery (4.2). For development docs, go here.

Workers Guide

Starting the worker

You can start the worker in the foreground by executing the command:

```
$ celery -A proj worker -l info
```

For a full list of available command-line options see **worker**, or simply do:

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

Daemonizing

You probably want to use a daemonization tool to start the worker in the background. See Daemonization for help starting the worker as a daemon using popular service managers.

You can start multiple workers on the same machine, but be sure to name each individual worker by specifying a node name with the **--hostname** argument:

```
$ celery -A proj worker --loglevel=INFO --concurrency=10 -n worker1@%h
$ celery -A proj worker --loglevel=INFO --concurrency=10 -n worker2@%h
$ celery -A proj worker --loglevel=INFO --concurrency=10 -n worker3@%h
```

The `hostname` argument can expand the following variables:

- `%h`: Hostname, including domain name.
- `%n`: Hostname only.
- `%d`: Domain name only.

If the current hostname is *george.example.com*, these will expand to:

Variable	Template	Result
`%h`	`worker1@%h`	*worker1@george.example.com*
`%n`	`worker1@%n`	*worker1@george*
`%d`	`worker1@%d`	*worker1@example.com*

Note for supervisor users:

The `%` sign must be escaped by adding a second one: *%%h*.

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. 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 being stuck in an infinite-loop or similar, you can use the **`KILL`** signal to force terminate the worker: but be aware that currently executing tasks will be lost (i.e., 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; make sure to do so manually. This command usually does the trick:

```
$ pkill -9 -f 'celery worker'
```

If you don't have the **pkill** command on your system, you can use the slightly longer version:

```
$ ps auxww | grep 'celery worker' | awk '{print $2}' | xargs kill -9
```

Restarting the worker

To restart the worker you should send the *TERM* signal and start a new instance. The easiest way to manage workers for development is by using *celery multi*:

```
$ celery multi start 1 -A proj -l info -c4 --pidfile=/var/run/celery/%n.
$ celery multi restart 1 --pidfile=/var/run/celery/%n.pid
```

For production deployments you should be using init-scripts or a process supervision system (see Daemonization).

Other than stopping, then starting the worker to restart, you can also restart the worker using the **`HUP`** signal. Note that the worker will be responsible for restarting itself so this is prone to problems and isn't recommended in production:

```
$ kill -HUP $pid
```

Note:

Restarting by **`HUP`** only works if the worker is running in the background as a daemon (it doesn't have a controlling terminal).

`HUP` is disabled on macOS 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`.

Variables in file paths

The file path arguments for `--logfile`, `--pidfile`, and `--statedb` can contain variables that the worker will expand:

Node name replacements

- `%p`: Full node name.
- `%h`: Hostname, including domain name.
- `%n`: Hostname only.
- `%d`: Domain name only.
- `%i`: Prefork pool process index or 0 if MainProcess.
- `%I`: Prefork pool process index with separator.

For example, if the current hostname is `george@foo.example.com` then these will expand to:

- `--logfile=%p.log` -> `george@foo.example.com.log`
- `--logfile=%h.log` -> `foo.example.com.log`
- `--logfile=%n.log` -> `george.log`
- `--logfile=%d.log` -> `example.com.log`

Prefork pool process index

The prefork pool process index specifiers will expand into a different filename depending on the process that'll eventually need to open the file.

This can be used to specify one log file per child process.

Note that the numbers will stay within the process limit even if processes exit or if autoscale/`maxtasksperchild`/time limits are used. That is, the number is the *process index* not the process count or pid.

- `%i` - Pool process index or 0 if MainProcess.

  Where `-n worker1@example.com -c2 -f %n-%i.log` will result in three log files:

  - `worker1-0.log` (main process)
  - `worker1-1.log` (pool process 1)
  - `worker1-2.log` (pool process 2)

- `%I` - Pool process index with separator.

  Where `-n worker1@example.com -c2 -f %n%I.log` will result in three log files:

  - `worker1.log` (main process)
  - `worker1-1.log` (pool process 1)
  - `worker1-2.log` (pool process 2)

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/prefork pool):

More pool processes are usually better, but there's a cut-off point where adding more pool processes affects performance in negative ways. There's 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.

pool support:	*prefork, eventlet, gevent,* blocking:*solo* (see note)
broker support:	*amqp, redis*

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 Management Command-line Utilities (inspect/control) for more information.

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's 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` 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:

```
>>> app.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:

```
>>> app.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:

```
>>> app.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()`**.

Commands

`revoke`: Revoking tasks

pool support:	all, terminate only supported by prefork
broker support:	*amqp, redis*
command:	**celery -A proj control revoke <task_id>**

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.

Note:

The terminate option is a last resort for administrators when a task is stuck. It's not for terminating the task, it's for terminating the process that's executing the task, and that process may have already started processing another task at the point when the signal is sent, so for this reason you must never call this programmatically.

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

```
>>> result.revoke()

>>> AsyncResult(id).revoke()

>>> app.control.revoke('d9078da5-9915-40a0-bfa1-392c7bde42ed')

>>> app.control.revoke('d9078da5-9915-40a0-bfa1-392c7bde42ed',
...                    terminate=True)

>>> app.control.revoke('d9078da5-9915-40a0-bfa1-392c7bde42ed',
...                    terminate=True, signal='SIGKILL')
```

Revoking multiple tasks

New in version 3.1.

The revoke method also accepts a list argument, where it will revoke several tasks at once.

Example

```
>>> app.control.revoke([
...    '7993b0aa-1f0b-4780-9af0-c47c0858b3f2',
...    'f565793e-b041-4b2b-9ca4-dca22762a55d',
...    'd9d35e03-2997-42d0-a13e-64a66b88a618',
])
```

The `GroupResult.revoke` method takes advantage of this since version 3.1.

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. When a worker starts up it will synchronize revoked tasks with other workers in the cluster.

The list of revoked tasks is in-memory so if all workers restart the list of revoked ids will also vanish. If you want to preserve this list between restarts you need to specify a file for these to be stored in by using the *–statedb* argument to **celery worker**:

```
$ celery -A proj worker -l info --statedb=/var/run/celery/worker.state
```

or if you use **celery multi** you want to create one file per worker instance so use the *%n* format to expand the current node name:

```
celery multi start 2 -l info --statedb=/var/run/celery/%n.state
```

See also Variables in file paths

Note that remote control commands must be working for revokes to work. Remote control commands are only supported by the RabbitMQ (amqp) and Redis at this point.

Time Limits

New in version 2.0.

pool support: *prefork/gevent*

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 isn’t catch-able and force terminates the task.

A single task can potentially run forever, if you have lots of tasks waiting for some event that’ll never happen you’ll 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:

```python
from myapp import app
from celery.exceptions import SoftTimeLimitExceeded

@app.task
def mytask():
    try:
        do_work()
    except SoftTimeLimitExceeded:
        clean_up_in_a_hurry()
```

Time limits can also be set using the `task_time_limit` / `task_soft_time_limit` settings.

Note:

Time limits don't currently work on platforms that don't support the `SIGUSR1` signal.

Changing time limits at run-time

New in version 2.3.

broker support: *amqp, redis*

There's 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:

```
>>> app.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 run-time

Example changing the rate limit for the *myapp.mytask* task to execute at most 200 tasks of that type every minute:

```
>>> app.control.rate_limit('myapp.mytask', '200/m')
```

The above doesn't specify a destination, so the change request will affect all worker instances in the cluster. If you only want to affect a specific list of workers you can include the `destination` argument:

```
>>> app.control.rate_limit('myapp.mytask', '200/m',
...            destination=['celery@worker1.example.com'])
```

Warning:

This won't affect workers with the `worker_disable_rate_limits` setting enabled.

Max tasks per child setting

New in version 2.0.

pool support: *prefork*

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 `--max-tasks-per-child` argument or using the `worker_max_tasks_per_child` setting.

Max memory per child setting

New in version 4.0.

pool support: *prefork*

With this option you can configure the maximum amount of resident memory 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 `--max-memory-per-child` argument or using the `worker_max_memory_per_child` setting.

Autoscaling

New in version 2.2.

pool support: *prefork, 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 `worker_autoscaler` setting.

Queues

A worker instance can consume from any number of queues. By default it will consume from all queues defined in the `task_queues` setting (that if not specified falls back to the default queue named `celery`).

You can specify what queues to consume from at start-up, by giving a comma separated list of queues to the `-Q` option:

```
$ celery -A proj worker -l info -Q foo,bar,baz
```

If the queue name is defined in **`task_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 **`task_create_missing_queues`** option).

You can also tell the worker to start and stop consuming from a queue at run-time 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 -A proj 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 -A proj control add_consumer foo -d celery@worker1.local
```

The same can be accomplished dynamically using the **`app.control.add_consumer()`** method:

```
>>> app.control.add_consumer('foo', reply=True)
[{u'worker1.local': {u'ok': u"already consuming from u'foo'"}}]

>>> app.control.add_consumer('foo', reply=True,
...                          destination=['worker1@example.com'])
[{u'worker1.local': {u'ok': u"already consuming from u'foo'"}}]
```

By now we've only shown examples using automatic queues, If you need more control you can also specify the exchange, routing_key and even other options:

```
>>> app.control.add_consumer(
...     queue='baz',
...     exchange='ex',
...     exchange_type='topic',
...     routing_key='media.*',
...     options={
...         'queue_durable': False,
...         'exchange_durable': False,
...     },
...     reply=True,
...     destination=['w1@example.com', 'w2@example.com'])
```

Queues: Canceling 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 -A proj 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 -A proj control cancel_consumer foo -d celery@worker1.local
```

You can also cancel consumers programmatically using the **app.control.cancel_consumer()** method:

```
>>> app.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 -A proj inspect active_queues
[...]
```

Like all other remote control commands this also supports the **--destination** argument used to specify the workers that should reply to the request:

```
$ celery -A proj inspect active_queues -d celery@worker1.local
[...]
```

This can also be done programmatically by using the **app.control.inspect.active_queues()** method:

```
>>> app.control.inspect().active_queues()
[...]

>>> app.control.inspect(['worker1.local']).active_queues()
[...]
```

Inspecting workers

app.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 **app.control** interface.

```
>>> # Inspect all nodes.
>>> i = app.control.inspect()

>>> # Specify multiple nodes to inspect.
>>> i = app.control.inspect(['worker1.example.com',
                            'worker2.example.com'])

>>> # Specify a single node to inspect.
>>> i = app.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 have been received, but are 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': '{}'}]}]
```

Statistics

The remote control command `inspect stats` (or **`stats()`**) will give you a long list of useful (or not so useful) statistics about the worker:

```
$ celery -A proj inspect stats
```

The output will include the following fields:

- `broker`

  Section for broker information.

  - `connect_timeout`

Timeout in seconds (int/float) for establishing a new connection.

  - `heartbeat`

    Current heartbeat value (set by client).

  - `hostname`

    Node name of the remote broker.

  - `insist`

    No longer used.

  - `login_method`

    Login method used to connect to the broker.

  - `port`

    Port of the remote broker.

  - `ssl`

    SSL enabled/disabled.

  - `transport`

    Name of transport used (e.g., `amqp` or `redis`)

  - `transport_options`

    Options passed to transport.

  - `uri_prefix`

    Some transports expects the host name to be a URL.

    ```
    redis+socket:///tmp/redis.sock
    ```

    In this example the URI-prefix will be `redis`.

  - `userid`

    User id used to connect to the broker with.

  - `virtual_host`

    Virtual host used.

- `clock`

  Value of the workers logical clock. This is a positive integer and should be increasing every time you receive statistics.

- `pid`

  Process id of the worker instance (Main process).

- `pool`

  Pool-specific section.

  - `max-concurrency`

    Max number of processes/threads/green threads.

- `prefetch_count`

    - `max-tasks-per-child`

        Max number of tasks a thread may execute before being recycled.

    - `processes`

        List of PIDs (or thread-id's).

    - `put-guarded-by-semaphore`

        Internal

    - `timeouts`

        Default values for time limits.

    - `writes`

        Specific to the prefork pool, this shows the distribution of writes to each process in the pool when using async I/O.

- `prefetch_count`

    Current prefetch count value for the task consumer.

- `rusage`

    System usage statistics. The fields available may be different on your platform.

    From *getrusage(2)*:

    - `stime`

        Time spent in operating system code on behalf of this process.

    - `utime`

        Time spent executing user instructions.

    - `maxrss`

        The maximum resident size used by this process (in kilobytes).

    - `idrss`

        Amount of non-shared memory used for data (in kilobytes times ticks of execution)

    - `isrss`

        Amount of non-shared memory used for stack space (in kilobytes times ticks of execution)

    - `ixrss`

        Amount of memory shared with other processes (in kilobytes times ticks of execution).

    - `inblock`

        Number of times the file system had to read from the disk on behalf of this process.

    - `oublock`

        Number of times the file system has to write to disk on behalf of this process.

  - `majflt`

    Number of page faults that were serviced by doing I/O.

  - `minflt`

    Number of page faults that were serviced without doing I/O.

  - `msgrcv`

    Number of IPC messages received.

  - `msgsnd`

    Number of IPC messages sent.

  - `nvcsw`

    Number of times this process voluntarily invoked a context switch.

  - `nivcsw`

    Number of times an involuntary context switch took place.

  - `nsignals`

    Number of signals received.

  - `nswap`

    The number of times this process was swapped entirely out of memory.

- `total`

  Map of task names and the total number of tasks with that type the worker has accepted since start-up.

Additional Commands

Remote shutdown

This command will gracefully shut down the worker remotely:

```
>>> app.control.broadcast('shutdown') # shutdown all workers
>>> app.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:

```
>>> app.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 the 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**.

```
>>> app.control.enable_events()
>>> app.control.disable_events()
```

Writing your own remote control commands

There are two types of remote control commands:

- Inspect command

  Does not have side effects, will usually just return some value found in the worker, like the list of currently registered tasks, the list of active tasks, etc.

- Control command

  Performs side effects, like adding a new queue to consume from.

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 increments the task prefetch count:

```
from celery.worker.control import control_command

@control_command(
    args=[('n', int)],
    signature='[N=1]',  # <- used for help on the command-line.
)
def increase_prefetch_count(state, n=1):
    state.consumer.qos.increment_eventually(n)
    return {'ok': 'prefetch count incremented'}
```

Make sure you add this code to a module that is imported by the worker: this could be the same module as where your Celery app is defined, or you can add the module to the `imports` setting.

Restart the worker so that the control command is registered, and now you can call your command using the **celery control** utility:

```
$ celery -A proj control increase_prefetch_count 3
```

You can also add actions to the **celery inspect** program, for example one that reads the current prefetch count:

```
from celery.worker.control import inspect_command

@inspect_command
def current_prefetch_count(state):
    return {'prefetch_count': state.consumer.qos.value}
```

After restarting the worker you can now query this value using the **celery inspect** program:

```
$ celery -A proj inspect current_prefetch_count
```

© Copyright 2009-2018, Ask Solem & contributors.