

# 用 Celery 实现多台设备的协同工作

2021 年 2 月 19 日

## 1 引言

久而久之，我们会发现，随着计算任务量的增大，迟早有那么一天，单台主机不再具备能力处理如此大的计算任务，那么此时，将任务分发到多台设备上，不失为一个有吸引力的选择。具体来说，很多可并行化的任务都可以很方便地拆分开来并且分发到多个计算节点上，比方说前面我们提到过的神经网络、三维物体渲染、多个方程组的并行求解、元胞自动机，以及遗传算法都具有这种特性——可以很方便地并行化。除此之外，爬虫任务，也可以并行化，比如说我们有十万张页面要爬取，利用 Celery，我们可以把它们分发到 10 台甚至 100 台机器上，然后很方便地把爬取结果集中在一处，从而极大地提高了效率。与此同时，Celery 不仅能使多台设备协同工作，在单台设备上，我们也可以利用 Celery 将计算任务均等地分发到每一个 CPU 核心上。首先我们会介绍单机模式，然后介绍多机模式。

## 2 单机模式

在 Celery 中，具体的计算任务是由 Worker 执行的，而 Worker 与应用程序直接的通信是通过消息队列来进行，我们以 RabbitMQ 为例，首先安装它：

```
sudo apt install rabbitmq-server
```

安装好后通过命令

```
sudo lsof -i :5672
```

来查看 RabbitMQ 服务是否已经在运行。如果这里安装成功，则进入下一步，安装 Celery：

```
python3 -m pip install celery
```

这就装好了最基础的东西，接下来可以开始试运行了，首先创建一个目录，作为我们接下来的工作目录：

```
mkdir hello-celery
```

然后进入该目录，再创建一个项目文件夹，用来容纳任务代码和 Celery 启动代码：

```
1 cd hello-celery
2 mkdir proj
```

接下来进入 `proj` 目录，写 Celery 的启动代码，文件名为 `celery.py`

```
1 from celery import Celery
2
3 app = Celery('proj')
4 app.config_from_object('celeryconfig')
5
6 if __name__ == '__main__':
7     app.start()
```

其中 `__name__=='__main__'` 的判定是必不可少的，这个接下来我们就会知道，然后从 `proj` 目录返回上一层目录也就是 `hello-celery` 目录，在这里创建一个 `celeryconfig.py` 文件，写上：

```
1 # List of modules to import when the Celery worker starts.
2 imports = ('proj.tasks',)
3
4 # Broker settings.
5 broker_url = 'amqp://'
```

然后我们进入 `proj` 目录，创建一个 `tasks.py` 文件，并写上：

```
1 from .celery import app
2
3 @app.task
4 def add(x, y):
5     return x + y
```

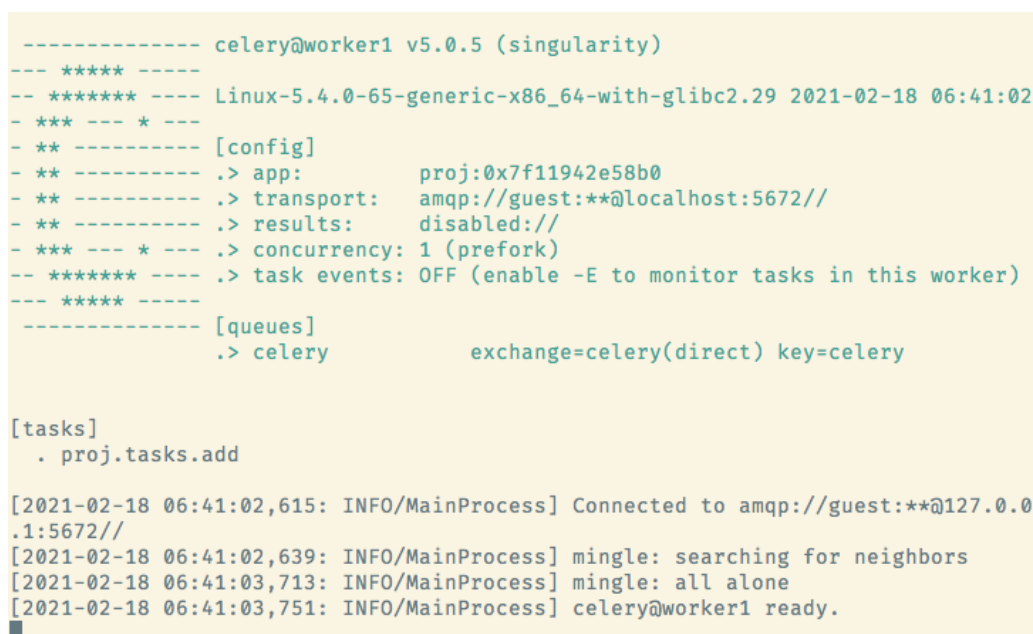
至此，整个目录结构是这样的：

```
hello-celery
├── celeryconfig.py
└── proj
    ├── celery.py
    └── tasks.py
```

其中，`celeryconfig.py` 对应 `celery.py` 中的第 4 行，而 `proj` 目录则对于 `celery.py` 中的第 3 行，接下来我们启动一个 Worker，首先进入 `hello-celery` 目录，然后执行：

```
celery --app proj worker -l INFO -n worker1
```

参数 `--app proj` 表示在 `proj` 目录存放着 Celery 应用程式的初始化代码以及任务例程，而 `worker` 子命令表示创建一个 Worker，`-l INFO` 表示日志详细程度是 INFO，`-n worker1` 表示该 Worker 的名字叫做 `worker1`，成功启动 Worker 后，界面应该如图 2-1：



```
----- celery@worker1 v5.0.5 (singularity)
--- ***** -----
-- ***** --- Linux-5.4.0-65-generic-x86_64-with-glibc2.29 2021-02-18 06:41:02
- *** --- * ---
- ** ----- [config]
- ** ----- .> app:      proj:0x7f11942e58b0
- ** ----- .> transport:  amqp://guest:**@localhost:5672//
- ** ----- .> results:   disabled://
- *** --- * --- .> concurrency: 1 (prefork)
-- ***** --- .> task events: OFF (enable -E to monitor tasks in this worker)
--- ***** -----
----- [queues]
      .> celery          exchange=celery(direct) key=celery

[tasks]
  . proj.tasks.add

[2021-02-18 06:41:02,615: INFO/MainProcess] Connected to amqp://guest:**@127.0.0.1:5672//
[2021-02-18 06:41:02,639: INFO/MainProcess] mingle: searching for neighbors
[2021-02-18 06:41:03,713: INFO/MainProcess] mingle: all alone
[2021-02-18 06:41:03,751: INFO/MainProcess] celery@worker1 ready.
```

图 2-1: 一个 Celery Worker 成功启动

并且我们看到, Worker 已经成功连上了消息队列也就是本地的 `amqp://`, 接下来我们试着执行一个任务: 首先打开另外一个 Shell, 然后进入 `hello-celery` 目录, 然后打开 Python:

```
python3
```

然后依次输入

```
1 # 载入 hello-celery/proj/tasks.py 文件定义的 add 函数
2 from proj.tasks import add
3
4 # 让 Worker 计算 1 + 2
5 # add.apply_async((1, 2,))
```

如图 2-2:

```
Python 3.8.5 (default, Jul 28 2020, 12:59:40)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from proj.tasks import add
>>> add.apply_async((1,2,))
<AsyncResult: 886f27ae-d6e1-43b7-875d-efdc41ece274>
>>>
```

图 2-2: 发布任务

由于我们还没有配置 Result Backend (这个我们接下来会做), 所以还不能取出结果, 但是因为 Worker 的日志详细程度被设为 **INFO**, 所以我们还是可以看到计算结果 (图 2-3): 在 Worker 的界面:

```
[2021-02-18 06:48:26,442: INFO/MainProcess] Received task: proj.tasks.add[886f27ae-d6e1-43b7-875d-efdc41ece274]
[2021-02-18 06:48:26,468: INFO/ForkPoolWorker-1] Task proj.tasks.add[886f27ae-d6e1-43b7-875d-efdc41ece274] succeeded in 0.006891511002322659s: 3
```

图 2-3: Worker 收到并完成任务

在这个案例中, 当我们运行

```
add.apply_async((1,2,))
```

的时候, 用 `1,2` 作为参数去调用 `add` 函数这个命令通过消息队列 `amqp://` 传达到了 Worker 那里.

## 2.1 保存计算结果

由于没有配置 Result backend, 所以 Worker 没有地方保存计算结果. 这一小节我们演示如何用 MongoDB 作为 Backend, 首先添加 MongoDB 的软件仓库的公钥:

```
1 sudo apt install gnupg
2 wget -q0 - https://www.mongodb.org/static/pgp/server-4.4.asc | sudo apt-key add -
```

然后添加软件源, 打开

```
/etc/apt/sources.list.d/mongodb-org-4.4.list
```

文件, 然后在最后一行加上

```
deb [ arch=amd64,arm64 ] https://repo.mongodb.org/apt/ubuntu focal/mongodb-org/4.4 multiverse
```

然后更新软件源并安装 MongoDB

```
sudo apt update
sudo apt install -y mongodb-org
systemctl start mongod
apt install python3-pip
python3 -m pip install -U pip
python3 -m pip install pymongo
```

我们可以通过运行

```
sudo lsof -i :27017
```

来判断 MongoDB 是否已经启动。默认配置下的 MongoDB 监听的是本地的 27017 端口，并且无需用户名和密码就可登录（当然仅限于本地）。然后我们再看 `celeryconfig.py` 文件最后加上一行：

```
result_backend = 'mongodb://'
```

并且重新启动 Worker，再次在 `hello-celery` 目录下启动 Python 并发布任务，此时应该可以取得计算结果了（图 2-4）：

```
Python 3.8.5 (default, Jul 28 2020, 12:59:40)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from proj.tasks import add
>>> result = add.apply_async((2,3,))
>>> result.get()
5
>>> █
```

图 2-4: 取得计算结果

可以单独指定启动 Worker 时的 `-c` 参数来设置一个 Worker 的处理队列中的子进程的个数，例如 `celery --app proj worker -l INFO -n worker1 -c 4`

就设置该 Worker 的子进程个数为 4，当然实际上这个数字不用我们自己设定，它默认会是 CPU 的核心个数。

### 3 多机模式

有时候我们会遇到这样的情形：要写爬虫工具爬取很多张页面，并且先爬取哪个页面后爬取哪个页面是无关紧要的，如果是在本地运行，要好几个小时，如果人不能一直守着电脑，还要通过设置确保本地的机器不会自动休眠或者锁屏，有的机器锁屏时会断开网络连接，所以我们就想把这么多的页面抓取任务分发到在云端始终运行着的多台服务器上去，这样一来我们就无需蹲守在电脑前确保电脑不会意外关机、休眠或者锁屏了。如果是笔记本电脑，始终开着就牺牲了它的便携性，除非我们能确保它能在合上盖子后继续运行。

我们打算做出这样的效果：在云端有若干服务器始终运行，第一台运行 MongoDB 和 RabbitMQ，并且面向公网提供服务，其余的服务器连接到第一台服务器的 MongoDB 和 RabbitMQ，然后本地 Celery 应用程序也连接到第一台服务器的 MongoDB 和 RabbitMQ，本地不用运行 Worker。这样配置好后，本地的任务就可以被分发到云端的这些机器以被同时执行。我们主要要做的是配置 MongoDB 和 RabbitMQ 的公网访问，至于 Celery，只要把同一份配置文件 `celeryconfig.py` 多处复制就可以了。

### 3.1 MongoDB 的远程访问

首先把正在运行的 `mongod` 进程关掉:

```
systemctl stop mongod
```

然后我们以无访问控制模式启动 MongoDB

```
mongod --fork --port 27017 --dbpath /var/lib/mongod
```

然后连接到 MongoDB

```
mongo --port 27017
```

创建一个超级管理员账户:

```
1 use admin
2 db.createUser(
3   {
4     user: "admin",
5     pwd: passwordPrompt(),
6     roles: [
7       { role: "userAdminAnyDatabase", db: "admin" },
8       "readWriteAnyDatabase"
9     ]
10  }
11 )
```

关闭 MongoDB 服务器

```
db.adminCommand( { shutdown: 1 } )
```

去修改配置文件 `/etc/mongod.conf`, 置:

```
1 security:
2   authorization: enabled
```

再次启动 MongoDB:

```
mongod --config /etc/mongod.conf --fork --auth --dbpath /var/lib/mongod
```

执行

```
lsof -i :27017
```

确保 MongoDB 已经启动. 现在我们可以以管理员的身份登录:

```
mongo "mongodb://name:pwd@127.0.0.1:27017/?authSource=admin"
```

其中 `name` 替换为 `admin` 是用户名, `pwd` 替换为你设置的密码, `authSource` 参数的值 `admin` 也就对应着创建用户的那段代码

```
1   roles: [
2     { role: "userAdminAnyDatabase", db: "admin" },
3     "readWriteAnyDatabase"
4   ]
```

成功连接到 MongoDB 之后我们创建一个 Celery 专用的数据库, 只有「使用」它就可以了, 如果不存在则 MongoDB 就会自动创建它:

```
use celery
```

然后创建用户并且授予它 `celery` 库的读写权限和 `reporting` 库的读权限：

```
1 db.createUser(  
2   {  
3     user: "celery",  
4     pwd: "blue",  
5     roles: [  
6       { role: "readWrite", db: "celery" },  
7       { role: "read", db: "reporting" }  
8     ]  
9   }  
10 )
```

然后我们按 Control-D 退出 `mongo`，并且使用这个 `celery` 账户登录，实际上用户名和数据库名不一定要相同，我们创建用户的时候都取 `celery` 只是为了方便：

```
mongo "mongodb://celery:blue@127.0.0.1/?authSource=celery"
```

如果没有报错，就说明用户创建成功了，可以进行下面的操作。

现在我们发现我们仍然是通过 `127.0.0.1` 这个环回地址连接到 MongoDB 数据库，所以我们要让 Mongo 绑定公网的 IP 地址，为此，可以修改配置文件 `/etc/mongod.conf`，找到并且修改：

```
1 net:  
2   port: 27017  
3   bindIp: 0.0.0.0
```

然后保存退出，并且重启 MongoDB：

```
pkill mongod  
mongod --auth --fork --config /etc/mongod.conf --dbpath /var/lib/mongodb
```

然后我们可以试着从本地连接它：

```
1 mongo "mongodb://celery:blue@ipaddr/?authSource=celery"
```

并且将其中的 `ipaddr` 替换为服务器的公网 IP。此时已经可以远程连接 MongoDB 了，但是密码是明文传递的，所以不安全，所以我们需要为服务器配置 TLS 证书，首先整一个域名，然后安装 `acme.sh` 为这个域名签证书：

```
1 apt install socat  
2 curl https://get.acme.sh | sh -s email=my@example.com  
3 acme.sh --issue -d mongodb.example.com --standalone
```

只是注意要将 `example.com` 替换为自己的域名（尤其是邮箱那部分）。`acme.sh` 列出了证书的路径就说明签发成功了，域名签发成功以后我们修改配置文件：

```
1 net:  
2   tls:  
3     mode: requireTLS  
4     certificateKeyFile: /etc/ssl/mongodb.pem  
5     port: 27017  
6     bindIp: 0.0.0.0
```

将其中的路径换成 `acme.sh` 列出的证书路径，然后创建这个 `mongodb.pem`：

```
1 cd ~/.acme.sh/domain.com  
2 cat fullchain.cer domain.com.key | tee /etc/ssl/mongodb.pem
```

然后重启 MongoDB:

```
pkill mongod
mongod --auth --fork --config /etc/mongod.conf --dbpath /var/lib/mongodb
```

然后在本地试着用 TLS 的方式连接:

```
1 mongo "mongodb://celery:password@domain.com/?authSource=celery&tls=true"
```

如果没有报错, 那么就算成功配置 MongoDB 了.

### 3.2 RabbitMQ 的远程访问

首先我们要给 RabbitMQ 服务器创建一个用户:

```
rabbitmqctl add_user "celery"
```

然后创建一个「虚拟主机」:

```
1 rabbitmqctl add_vhost celery
```

然后授予 celery 用户在 celery 虚拟主机上的全部权限:

```
1 rabbitmqctl set_permissions -p "celery" "celery" ".*" ".*" ".*"
```

默认情况下 RabbitMQ 会监听所有网卡的 5672 端口, 所以不用像 MongoDB 那样设置监听 0.0.0.0 了. 接下来我们试着验证可以远程连接 RabbitMQ, 首先安装它的 Python 客户端 pika:

```
1 python3 -m pip install pika
```

然后创建一个 send.py 文件:

```
1 #!/Library/Frameworks/Python.framework/Versions/3.8/bin/python3
2 import pika
3
4 credentials = pika.PlainCredentials('celery', 'blue')
5 connection = pika.BlockingConnection(
6     pika.ConnectionParameters(
7         host='ipaddr',
8         port=5672,
9         virtual_host='celery',
10        credentials=credentials
11    )
12 )
13 channel = connection.channel()
14
15 channel.queue_declare(queue='hello')
16
17 channel.basic_publish(exchange='', routing_key='hello', body='Hello World!')
18 print(" [x] Sent 'Hello World!'")
19 connection.close()
```

以及一个 receive.py 文件:

```
1 #!/Library/Frameworks/Python.framework/Versions/3.8/bin/python3
2 import pika, sys, os
3
4 def main():
```

```

5
6 credentials = pika.PlainCredentials('celery', 'blue')
7 connection = pika.BlockingConnection(
8     pika.ConnectionParameters(
9         host='ipaddr',
10        port=5672,
11        virtual_host='celery',
12        credentials=credentials
13    )
14 )
15
16 channel = connection.channel()
17
18 channel.queue_declare(queue='hello')
19
20 def callback(ch, method, properties, body):
21     print("[x] Received %r" % body)
22
23 channel.basic_consume(
24     queue='hello',
25     on_message_callback=callback,
26     auto_ack=True
27 )
28
29 print('[*] Waiting for messages. To exit press CTRL+C')
30 channel.start_consuming()
31
32 if __name__ == '__main__':
33     try:
34         main()
35     except KeyboardInterrupt:
36         print('Interrupted')
37         try:
38             sys.exit(0)
39         except SystemExit:
40             os._exit(0)

```

将其中的 ipaddr 替换为 RabbitMQ 服务器的 IP 地址，然后在一个终端先运行：

```
python3 receive.py
```

再在另一个终端运行：

```
python3 send.py
```

此时应该会看到接受终端出现：

```

[*] Waiting for messages. To exit press CTRL+C
[x] Received b'Hello World!'

```

这时 RabbitMQ 的远程连接已经基本配置成功了。接下来同样地我们也要为 RabbitMQ 服务器配置 TLS，否则密码也会明文发送。首先修改 `/etc/rabbitmq/rabbitmq.conf`，写入：

```

1 listeners.ssl.default = 5671
2 ssl_options.cacertfile = /etc/ssl/rabbitmq/ca.cer

```



```
3 ssl_options.certfile = /etc/ssl/rabbitmq/fullchain.cer
4 ssl_options.keyfile = /etc/ssl/rabbitmq/domain.key
5 ssl_options.verify = verify_none
```

将原来 acme 目录下的文件复制到/etc/ssl/rabbitmq 下即可，然后重启 RabbitMQ 使配置生效：

```
1 rabbitmqctl shutdown
2 rabbitmq-server -detached
3 rabbitmqctl start_app
```

新建一个 ssl\_test.py 文件：

```
1 import pika
2 import ssl
3
4 credentials = pika.PlainCredentials('celery', 'blue')
5 context = ssl.SSLContext(ssl.PROTOCOL_TLSv1_1)
6 context.verify_mode = ssl.CERT_REQUIRED
7 context.load_default_certs()
8 ssl_options = pika.SSLOptions(context, server_hostname='domain.com')
9 conn_params = pika.ConnectionParameters(
10     host='domain.com',
11     port=5671,
12     virtual_host='celery',
13     credentials=credentials,
14     ssl_options=ssl_options
15 )
16
17 with pika.BlockingConnection(conn_params) as conn:
18     ch = conn.channel()
19     ch.queue_declare("foobar")
20     ch.basic_publish("", "foobar", "Hello, world!")
21     print(ch.basic_get("foobar"))
```

然后用运行它：

```
python3 ssl_example.py
```

输出大概是：

```
(<Basic.GetOk(['delivery_tag=1', ... , <BasicProperties>, b'Hello, world!']>
```

这说明客户端可以通过 TLS 连接到服务器。

### 3.3 启动 Worker

现在我们已经在一台服务器上安装好了 RabbitMQ 和 MongoDB，并且都分别为它们启用了 TLS 并且创建了用户，如果我们能在这台机器上启动一个 Worker，并且通过 TLS 加用户名与密码的方式与 MQ 和 DB 连接，那么在其他机器上启用 Celery Worker 也是类似的，只不过就不再需要再配置 MQ 和 DB 了，也就是说前面做的这些安装 RabbitMQ、安装 MongoDB、创建用户和启用 TLS 只需要做一次。

现在我们在本地创建一个文件夹叫做 `hello-celery`，结构类似于：

```
hello-celery
├── celeryconfig.py
└── proj
    ├── celery.py
    └── tasks.py
```

这里的 `celery.py` 文件以及 `tasks.py` 文件与前面的一样，要改的只有 `celeryconfig.py`：

```
1 # List of modules to import when the Celery worker starts.
2 imports = ('proj.tasks',)
3
4 # Broker settings.
5 broker_url = 'amqps://celery:blue@domain.com/celery'
6
7 # Using the database to store task state and results.
8 result_backend = 'mongodb://celery:blue@domain.com/?authSource=celery&tls=true'
```

将其中的 `celery:blue` 换成自己设定的用户名和密码，将 `domain.com` 换成自己设定的域名。现在如果 `cd` 到本地的 `hello-celery` 目录，并且执行：

```
celery --app proj inspect active
```

那么本地的 Celery 应用程式就会试图跟远端的 MongoDB 服务器和 RabbitMQ 服务器进行通信，相当于问：是否已有 Worker 正在运行，自然是没的，因为我们还没有启动 Worker，那么为了启动 Worker，我们只需将这份代码（也就是 `hello-celery` 整个目录）复制到任何一个能联网的服务器上，并在上面执行：

```
celery --app proj worker -l INFO
```

就可以启动一个 Worker。

现在我们可以可以在 `/.ssh/config` 中配置我们的各个服务器的连接方式：

```
1 IdentityFile /path/to/id_rsa
2
3 Host remote-server-1
4 Hostname ipaddr1
5 User ubuntu
6 Port 2333
7
8 Host remote-server-2
9 Hostname ipaddr2
10 User ubuntu
11 Port 2333
12
13 Host remote-server-3
14 Hostname ipaddr3
15 User ubuntu
16 Port 2333
```

注意将其中的 `ipaddr{1,2,3}` 换成自己的服务器的地址或者可解析的域名，然后我们创建一个文件名为 `duplicate.py` 用于将 Celery 应用程式文件分发到各个主机上：

```
1 import os
```

```
2
3 for i in range(1, 4):
4     os.system(f"rsync -aPuvz hello-celery remote-server-{i}:~")
```

再创建一个名为 `workers.py` 的文件用于批量启动或者停止 Worker:

```
1 import os
2 import sys
3
4 action = ""
5 if sys.argv[1] == "start":
6     action = "start"
7 elif sys.argv[1] == "stop":
8     action = "stop"
9 elif sys.argv[1] == "restart":
10    action = "restart"
11 else:
12    sys.exit(1)
13
14 for i in range(1, 4):
15    ssh_command = f"ssh remote-server-{i}"
16    change_cwd = "cd hello-celery"
17
18    launch_worker = f"celery multi {action} w{i} -A proj -l INFO"
19    command = f"{ssh_command} \"{change_cwd} && {launch_worker}\""
20    os.system(command)
```

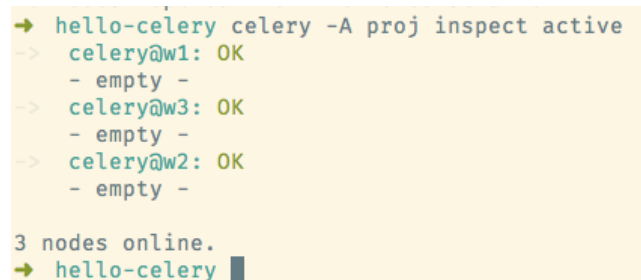
这时我们可以先后执行:

```
1 python3 duplicate.py
2 python3 workers.py start
```

来实现 Workers 的批量启动. 启动完成后再本地执行:

```
celery --app proj inspect active
```

来查看已经启动好了的 Workers (图 3-1):



```
→ hello-celery celery -A proj inspect active
-> celery@w1: OK
- empty -
-> celery@w3: OK
- empty -
-> celery@w2: OK
- empty -

3 nodes online.
→ hello-celery █
```

图 3-1: 启动好了的 Workers

另外, 可以通过:

```
celery control --help
```

来查看更多可以远端执行的针对 Worker 的控制指令.

## 4 更好的部署方式

我们在上文中给出的部署方式适合一种常见的情形：手头有多台闲置着的 VPS，并且这些 VPS 很有可能是到处分布着的，并且对计算任务的实时性要求不高，所以我们可以忍受公网的高延迟再加上 TLS 验证的时间和性能开销。但是如果是对于任务和计算结果的传输时延要求比较严格的话，最好还是不要用这种方式。下面我们来介绍几种方案，但不具体涉及，仅提供思路：

### 4.1 基于云平台的 VPC 私网部署

现在很多云平台 (AWS, Azure, Aliyun, etc.) 都提供私有网络的服务，私有网络 (VPC) 它可以提供更好的安全隔离以及更快的内网传输，也就是说如果两台设备是在一个私网内，它们之间的数据传输就可以很快，延迟也可以很低，因为不需要走拥堵繁忙的公网路线，这就好像是坐地铁去上班和坐直升飞机去上班的区别。于是我们可以把全部的 Worker, MongoDB 服务器和 RabbitMQ 服务器都搭建在同一个私网内，这样一来不需要配置用户名和密码，也不需要 TLS，只要把特定网段加入白名单就可以了，并且发布任务也不是在本地发布，而是在私网的其中一台机器上发布，这样一来 Worker 和 Celery 应用程式之间的通信延迟就可以很低。

### 4.2 基于本地局域网的部署

与基于云平台 VPC 的方案是类似的，只不过搬到了本地，并且事实上延迟可能更低，但是毕竟毕竟不是每个人都是专业的运维人员，而且而外的可靠性和可用性保障也需要增加投入，所以感觉并不比云平台的方案好多少。但是如果家里头有多台闲置设备的话，也是可以做这种尝试。较旧的设备可能能耗比不高，也就是说有可能会增加更多的电费支出，但是获得的算力却不见增加多少。

## 5 其他想说的

如果是专业的科学计算情形、机器学习/深度学习，以及大规模的数据分析和数据处理，基于 Celery 的实现方案似乎还不是很多，并且，已经知道的是，高性能计算对于节点与节点之间的通信延迟和数据传输带宽要求严格，所以，对于这类情形，我还是推荐像 Spark, Hadoop, Hive, CUDA 等成熟方案。Celery 作为一个任务队列，它的作用主要是把多个任务分发到多个节点上，并且提供一定程度上的 Fail-Safe 措施，即如果单台设备突然 Go offline 了，那么 Assign 到它上面的还没有 ACK 的任务可以被重新 Assign 给其他 Active 节点，从而提供一种 High Availability，从这个角度看呢，它更类似于一种可编程的、更加智能的负载均衡 (Load Balancing)，或者说是简化版本的消息队列 (Message Queue)，事实上 Celery 正是基于消息队列的，它相当于做了更高的抽象，使得消息队列的一个功能子集——任务队列的使用变得简单。Hadoop 能做 Map-Reduce，而 Celery 也能做，这正好揭示了 Map-Reduce 的灵活实现方式，以及 Celery 广阔的应用情景。