Python 并行分布式框架 Celery

Celery 是一个 基于 python 开发的分布式异步消息任务队列

Celery 由 Python 编写的简单、灵活、可靠的用来处理大量信息的分布式系统,它同时提供操作和维护分布式系统所需的工具。 该协议可以在任何语言实现。

Celery 专注于实时任务处理，支持任务调度。

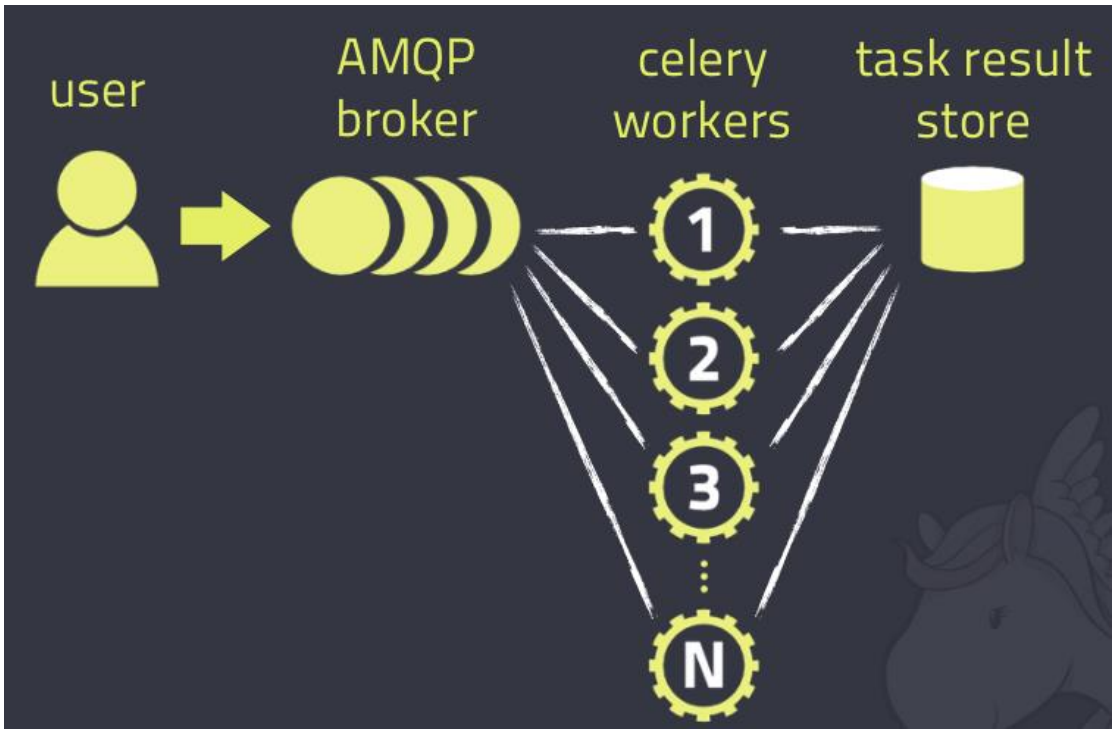
应用场景

- ✓ 你想对 100 台机器执行一条批量命令，可能会花很长时间 ，但你不想让你的程序等着结果返回，而是给你返回 一个任务 ID,你过一段时间只需要拿着这个任务 id 就可以拿到任务执行结果， 在任务执行时，你可以继续做其它的事情。
- ✓ 你想做一个定时任务，比如每天检测一下你们所有客户的资料，如果发现今天 是客户的生日，就给他发个短信祝福

解决方法

- ◆ 示例一的解决：将耗时的程序放到 celery 中执行
- ◆ 示例二的解决：使用 celery 定时执行

工作流程图



celery 的官方定义是分布式任务队列，celery 通过这个队列来实现跨线程或跨机器的作业分发。队列的输入是一个作业单元，被称为 task，通常是用一个函数定义，函数上方用 @app.task 装饰（也可带参数）。具体的 worker 会持续监控这个队列，看是不是需要新的作业。

celery 通过消息通信，一般使用一个叫 broker 的模块来实现 client 和 workers 的通信。当 client 需要初始化一个 task 时，它会向任务队列中添加一条消息，之后 broker 会负责通过一些算法将这条消息递送到合适的 worker。

worker 和 broker 的数量可以很多，这保证了该分布式架构的高可用和水平扩展性。

架构组成

Celery 的架构由三部分组成，消息中间件（message broker），任务执行单元（worker）和任务执行结果存储（task result store）组成。

Brokers

brokers 中文意思为中间人，在这里就是指任务队列本身，实现 client 和 workers 的消息通信。常见的 brokers 有 rabbitmq、redis、Zookeeper 等

Workers

Celery 提供的任务执行的单元，worker 并发的运行在分布式的系统节点中。

Result Stores / backend

存储 Worker 执行的任务的结果，常见的 backend 有 AMQP、redis、Memcached 甚至常用的数据都可以。

Tasks

想在队列中执行的任务，一般由用户、触发器或其他操作将任务入队，然后交由 workers 进行处理。

安装 Celery 和 redis

这里我们用 redis 当做 celery 的 broker 和 backend。

以及 python 的 redis 支持:

```
apt-get install redis-server
pip install redis
pip install celery
```

```
(env2) E:\Python\代码\scrapy\scrapyd_test>pip install celery
Collecting celery
  Downloading https://files.pythonhosted.org/packages/e8/58/2a0b1067
ab2c12131b5c089dfc579467c76402475c5231095e36a43b749c/celery-4.2.1-py
2.py3-none-any.whl (401kB)
    7% |███▏                            | 30kB 34kB/s eta 0:00:11
```

如果 win10 下，安装:

pip install eventlet

```
(env2) E:\Python\代码\scrapy\scrapyd_test> pip install eventlet
Collecting eventlet
  Downloading https://files.pythonhosted.org/packages/86/7e/96e1412f
96eeb2f2eca9342dcc4d5bc9305880a448b603b0a8e54439b71c/eventlet-0.24.1
-py2.py3-none-any.whl (219kB)
```

pip install gevent

Celery 的基本用法

编写 task

定义 tasks.py 文件:

设置 broker，backend，编写 task

```
#tasks.py
from celery import Celery
uri = 'redis://:123@127.0.0.1:6379/2'
app = Celery('tasks',  backend=uri, broker=uri) #配置好 celery 的 backend 和 broker

@app.task  #普通函数装饰为 celery task
def add(x, y):
    return x + y
```

装饰器 app.task 实际上是将一个正常的函数修饰成了一个 celery task 对象

运行 worker

语法：

celery -A proj.task worker --loglevel=info

-A 是指对应的应用程序，其参数任务文件名

worker 任务角色，，表明当前机器的身份；此时，就是启动了一个 worker

--loglevel=info 任务日志级别

在 tasks.py 所在目录下运行：

#celery -A tasks worker -l info -P eventlet （win10 下）
#celery -A tasks worker --loglevel=info

-P eventlet：通过协程实现并发

触发任务脚本

在脚本中调用被装饰成 task 的函数：

```
#trigger.py
from tasks import add
result = add.delay(4, 4) #不要直接 add(4, 4)，这里需要用 celery 提供的接口 delay 进行调用
while not result.ready():
    time.sleep(1)
print 'task done: {0}'.format(result.get())
```

```
[tasks]
  . tasks.add

[2018-10-20 17:22:03,160: INFO/MainProcess] Connected to redis://:**@127.0.0.1:6379/2
[2018-10-20 17:22:03,174: INFO/MainProcess] mingle: searching for neighbors
[2018-10-20 17:22:04,196: INFO/MainProcess] mingle: all alone
[2018-10-20 17:22:04,210: INFO/MainProcess] pidbox: Connected to redis://:**@127.0.0.1:6379/2.
[2018-10-20 17:22:04,217: INFO/MainProcess] celery@QIKUWWW-JJ3BDIB ready.
[2018-10-20 17:22:11,721: INFO/MainProcess] Received task: tasks.add[81659d6b-19c8-4669-84fe-b8781f3d7c54]
[2018-10-20 17:22:11,725: INFO/MainProcess] Task tasks.add[81659d6b-19c8-4669-84fe-b8781f3d7c54] succeeded in 0.0s: 8
```

```
D:\test\vritualenv\env2\Scripts\python.exe E:/Python/代码/scrapy/15_celery/demo1/trigger.py
task done: 8

进程已结束,退出代码0
```

定时/周期任务

celery 支持定时任务，设定好任务的执行时间，celery 就会定时自动帮你执行， 这个定时任务模块叫 **celery beat**

Celery 中启动定时任务有两种方式，（1）在配置文件中指定；（2）在程序中指定。

在配置文件中指定

新建 Celery 配置文件

#celery_config.py

```
from datetime import timedelta
from celery.schedules import crontab

CELERYBEAT_SCHEDULE = {
    'ptask': {
        'task': 'tasks.period_task',
        'schedule': timedelta(seconds=5),
    },
}

CELERY_RESULT_BACKEND = 'redis://:123@127.0.0.1:6379/2'
CELERY_TIMEZONE = 'Asia/Shanghai'
```

配置中 schedule 就是间隔执行的时间，这里可以用 datetime.timedelta 或者 crontab 甚至太阳系经纬度坐标进行间隔时间配置

如果定时任务涉及到 datetime 需要在配置中加入时区信息，否则默认是以 utc 为准。例如中国可以加上：

```
CELERY_TIMEZONE = 'Asia/Shanghai'
```

增加周期执行的任务

#tasks.py

```
from celery import Celery
from celery import Task

app = Celery('tasks', backend='redis://:123@127.0.0.1:6379/2',
broker='redis://:123@127.0.0.1:6379/2')
app.config_from_object('celery_config')


@app.task(bind=True)
def period_task(self):
    print('period task done: {0}'.format(self.request.id))
```

当装饰器@app.task 添加 bind=True 时,被修饰的函数第一个参数被作为任务对象,通过

self 可获取任务的上下文

运行 worker:

celery -A tasks worker -l info -P eventlet

运行 beat，启动任务调度器:

celery -A tasks beat

在程序中指定

```
# tasks.py
from celery import Celery
from celery.schedules import crontab

uri = 'redis://:123@127.0.0.1:6379/2'
app = Celery('tasks', broker=uri)
# 每分钟执行一次
c1 = crontab()
```

```
# 每天凌晨十二点执行
c2 = crontab(minute=0, hour=0)
# 每十五分钟执行一次
crontab(minute='*/15')
# 每周日的每一分钟执行一次
crontab(minute='*',hour='*', day_of_week='sun')
# 每周三，五的三点，七点和二十二点没十分钟执行一次
crontab(minute='*/10',hour='3,17,22', day_of_week='thu,fri')


@app.task
def send(message):
    return message


app.conf.beat_schedule = {
    'send-every-10-seconds': {
        'task': 'tasks.send',
        'schedule': 10.0,
        #'schedule':c1,
        'args': ('Hello World', )
    },
}
```

上面的示例配置了一个每十秒执行一次的周期任务，任务为 tasks.send，参数为 'Hello World' 。

这种配置的方式可以支持多个参数

task： 指定任务的名字

schedule：设定任务的调度方式，可以是一个表示秒的整数，也可以是一个 timedelta 对象，或者是一个 crontab 对象（后面介绍），总之就是设定任务如何重复执行

args： 任务的参数列表

kwargs：任务的参数字典

options：所有 apply_async 所支持的参数

运行 worker：

celery -A tasks worker -l info -P eventlet

运行 beat，启动任务调度器：

celery -A tasks beat

定时配置参考方式

Example	Meaning
crontab()	每分钟执行
crontab(minute=0, hour=0)	每天0点执行
crontab(minute=0, hour='*/3')	每3小时执行: midnight, 3am, 6am, 9am, noon, 3pm, 6pm, 9pm.
crontab(minute=0, hour='0, 3, 6, 9, 12, 15, 18, 21')	同上
crontab(minute='*/15')	每15分钟执行
crontab(day_of_week='sunday')	周天的每分钟执行
crontab(minute='*', hour='*', day_of_week='sun')	同上
crontab(minute='*/10', hour='3, 17, 22', day_of_week='thu, fri')	周三、五，3-4 am, 5-6 pm, and 10-11 pm，每10分钟执行
crontab(minute=0, hour='*/2, */3')	每小时/2和每小时/3，执行
crontab(minute=0, hour='*/3, 8-17')	每小时/3，8am-5pm，执行
crontab(0, 0, day_of_month='2')	Execute on the second day of every month.
crontab(0, 0, day_of_month='2-30/3')	Execute on every even numbered day.
crontab(0, 0, day_of_month='1-7, 15-21')	Execute on the first and third weeks of the month.
crontab(0, 0, day_of_month='11', month_of_year='5')	Execute on the eleventh of May every year.
crontab(0, 0, month_of_year='*/3')	Execute on the first month of every quarter.

案例：分布式爬取全国城市的气温

实现爬取的 worker

这里定义 task 是根据传入的 url 抓取当前页面里面的城市气温，这需要在 worker 里实现，

代码如下：

#weather_worker.py

```
from celery import Celery
from lxml import etree
import requests

uri1 = 'redis://:123@127.0.0.1:6379/3'
uri2 = 'redis://:123@127.0.0.1:6379/4'
app = Celery('tasks',  backend=uri1, broker=uri2) #配置好 celery 的 backend 和 broker

@app.task
def crawl(location, url):
    headers = {
        "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/68.0.3440.106 Safari/537.36"
    }
    response = requests.get(url, headers=headers)
    html = etree.HTML(response.content)
    #print(etree.tostring(html, pretty_print=True).decode())
    temperature = html.xpath('//dd[@class="weather"]/p/b/text()')[0] + '℃'
    print(location,temperature)
    return [location,temperature]
```

实现爬取的 client

client 需要把 task 所需要的 url 发送到消息队列里，也就是这里的'redis://:''@127.0.0.1/4'。

client 最多会一次性发送 1000 条消息到队列，然后阻塞自己。同时各个 worker 会检测这

个队列，根据消息决定需要执行什么内容，并把结果写入 backend。

#execute_tasks.py

```
class Client:
```

```
    def __init__(self):
        self.urls = []
        self.base_url = 'https://www.tianqi.com'

    def getUrls(self):
        url = 'https://www.tianqi.com/chinacity.html'
        headers = {
            "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/68.0.3440.106 Safari/537.36"
        }
        response = requests.get(url, headers=headers)
        html = etree.HTML(response.content)
        ls = html.xpath('//div[@class="citybox"]//a')
        # ls2 = html.xpath('//div[@class="citybox"]//a/text()')
        for item in ls:
            url = self.base_url + item.xpath('@href')[0]
            location = item.text
            self.urls.append((location, url))
        print(self.urls)

    def task_manage(self):
        for url in self.urls:
            pass
            crawl.delay(url[0], url[1])
            # app.send_task('aqicn.crawl', args=(url[0],url[1],))


if __name__ == "__main__":
    client = Client()
    client.getUrls()
    client.task_manage()
```

这里的 crawl.delay(url[0],url[1])就是发送消息到队列的操作，推送 task 有两种写法，代码中注释的部分就是第二种。

部署

可以把两个 python 文件部署到不同的主机上，代表 worker

运行爬虫

在各主机的 worker 上执行：

celery -A weather_worker worker -l info -P gevent -c 20

-A 指明 app 所在的模块，这里是 **weather_worker**;

worker 表明当前机器的身份；

-l 是 log 的等级；

-P 是表明并发的方式，默认是单线程，这里使用的是协程；

-c 表示协程的数量，这里是 20。协程数量不能太高，不然会出现大量的请求失败，导致没有数据。

client 只要直接执行：py execute_tasks.py。

之后你就会看到两台 worker 开始不断按照接受 task，执行 task，返回结果的顺序运行，直到队列空。

这里即使 task 的代码进入异常流程了，也只会把异常信息保存下来，程序不会终止。

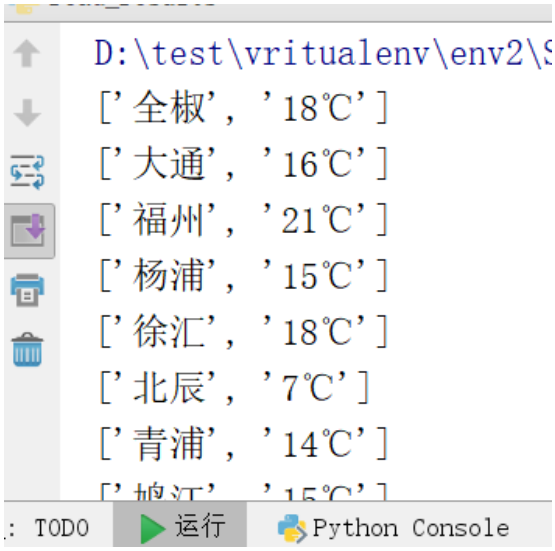
获取结果

写一段代码从作为 backend 的 redis 中将结果取出来，代码如下：

```
#read_results.py

import redis
import json

r = redis.Redis(host='127.0.0.1',password='123',port=6379,db=3)

keys = r.keys()

for key in keys:
```

```
res = r.get(key)
res = json.loads(res.decode('utf-8'))
print (res.get('result'))
```

结果：



附录：Python 协程实现

python 网络库也有基于协程的实现，知名的方案有：

Tornado 协程

依赖与 Tornado 的 IOLoop，所以不能单独拿出来使用。

Greenlet

真正的协程，在使用过程中通过 switch 来中断当前执行的函数，切换到另一个 greenlet，在其它的 geenlet 中调用 switch 会激活之前被挂起的协程。

Eventlet

Eventlet 在 Greenlet 的基础上实现了自己的 GreenThread，实际上就是 greenlet 类的扩展封装，而与 Greenlet 的不同是，Eventlet 实现了自己调度器称为 Hub，Hub 类似于 Tornado 的 IOLoop，是单实例的。在 Hub 中有一个 event loop，根据不同的事件来切换到对应的 GreenThread。

Gevent

Gevent 的 2 架马车，libev 与 Greenlet。不同于 Eventlet 的用 python 实现的 hub 调度，Gevent 通过 Cython 调用 libev 来实现一个高效的 event loop 调度循环。