异步， celery ， rabbitmq

1.如何在多台机器上部署celery和rabbitmq

假设有机器A, B

(1)

在机器B上安装并且启动rabbitmq， 默认自带guest用户(只能localhost登录)， 自己需要再添加一个用户admin

命令：

rabbitmqctl start_app 启动服务

rabbitmqctl list_users 列出所有用户

rabbitmqctl add_user admin admin 添加admin用户

rabbitmqctl set_permissions -p / admin .* .* .* 为admin添加权限

之后在B机器上可通过http://127.0.0.1:15672访问rabbitmq控制台

(2)

在机器A上建立文件

tasks.py

```
-------------------------------------------------------------------------------
-----------
# -*- coding:utf-8 -*-
from celery import Celery

app = Celery('tasks',  broker='amqp://admin:admin@B机器IP:5672',
backend='amqp://admin:admin@B机器IP:5672') #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
-------------------------------------------------------------------------------
-----------
```

在机器B上建立文件

tasks.py

```
-------------------------------------------------------------------------------
-----------
# -*- coding:utf-8 -*-
from celery import Celery

app = Celery('tasks',  broker='amqp://guest:guest@localhost:5672',
```

backend='amqp://guest:guest@localhost:5672') #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
```

注意这里机器A上的add函数执行加法， 而机器B上的add执行乘法。

（3）

在机器A上tasks.py所在的文件夹执行命令行指令

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

在机器B上tasks.py所在的文件夹执行命令行指令

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

（4）

在机器A或机器B上执行指令

```
from tasks import add
result = add.delay(4, 4)
result.get()
```

默认不作修改的情况下两个celery工作节点轮流获取任务， 所以重复执行这个指令会出现8， 16， 8， 16循环地输出， 让两个输出不同只是为了验证是在不同的机器上进行部署，实际生产环境中函数应该是一样的以实现分布式。

==========

2.celery的其他方法

（1）

```python
# tasks.py
class MyTask(Task):
    def on_success(self, retval, task_id, args, kwargs):
        print 'task done: {0}'.format(retval)
        return super(MyTask, self).on_success(retval, task_id, args, kwargs)

    def on_failure(self, exc, task_id, args, kwargs, einfo):
        print 'task fail, reason: {0}'.format(exc)
        return super(MyTask, self).on_failure(exc, task_id, args, kwargs, einfo)

@app.task(base=MyTask)
def add(x, y):
    return x + y

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

```
    return x + y
```

通过继承并修改Task类的on_success和on_failure方法， 可以实现add函数成功和失败时输出日志的修改。

（2）

```
# tasks.py
from celery.utils.log import get_task_logger

logger = get_task_logger(__name__)
@app.task(bind=True)
def add(self, x, y):
    logger.info(self.request.__dict__)
    return x + y
```

通过获取默认日志记录器和绑定实例方法的方式可以输出request中的参数

（3）

实际场景中得知任务状态是很常见的需求，对于 Celery 其内建任务状态有如下几种：

参数	说明
PENDING	任务等待中
STARTED	任务已开始
SUCCESS	任务执行成功
FAILURE	任务执行失败
RETRY	任务将被重试
REVOKED	任务取消

```
# tasks.py
from celery import Celery
import time

@app.task(bind=True)
def test_mes(self):
    for i in xrange(1, 11):
        time.sleep(0.1)
        self.update_state(state="PROGRESS", meta={'p': i*10})
    return 'finish'
```

```
# trigger.py
from task import add,test_mes
import sys

def pm(body):
    res = body.get('result')
    if body.get('status') == 'PROGRESS':
        sys.stdout.write('\r任务进度: {0}%'.format(res.get('p')))
        sys.stdout.flush()
    else:
        print '\r'
        print res
r = test_mes.delay()
print r.get(on_message=pm, propagate=False)
```

在命令行中执行python trigger.py 输出test_mes函数执行过程的情况。

（4）

周期/定时任务

新建 Celery 配置文件 celery_config.py:

```
# 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://localhost:6379/0'
```

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，具体可以参考这里
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'
```

然后在 tasks.py 中增加要被周期执行的任务：

```
# tasks.py
app = Celery('tasks', backend='redis://localhost:6379/0', broker='redis://localhost:6379/0
')
app.config_from_object('celery_config')

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

然后重新运行 worker，接着再运行 beat：

```
celery -A task beat
```

（5）
链式任务

有些任务可能需由几个子任务组成，此时调用各个子任务的方式就变的很重要，尽量不要以同步阻塞的方式调用子任务，而是用异步回调的方式进行链式任务的调用：
错误示范

```
@app.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)

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

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

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

```

正确示范1

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

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

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

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

正确示范2

```
fetch_page.apply_async(args=[url], link=[parse_page.s(), store_page_info.s
(url)])
```

链式任务中前一个任务的返回值默认是下一个任务的输入值之一（ 不想让返回值做默认参数可以用 si() 或者 s(immutable=True) 的方式调用 ）。

这里的 s() 是方法 celery.signature() 的快捷调用方式，signature 具体作用就是生成一个包含调用任务及其调用参数与其他信息的对象，个人感觉有点类似偏函数的概念：先不执行任务，而是把任务与任务参数存起来以供其他地方调用。