Celery

情境:

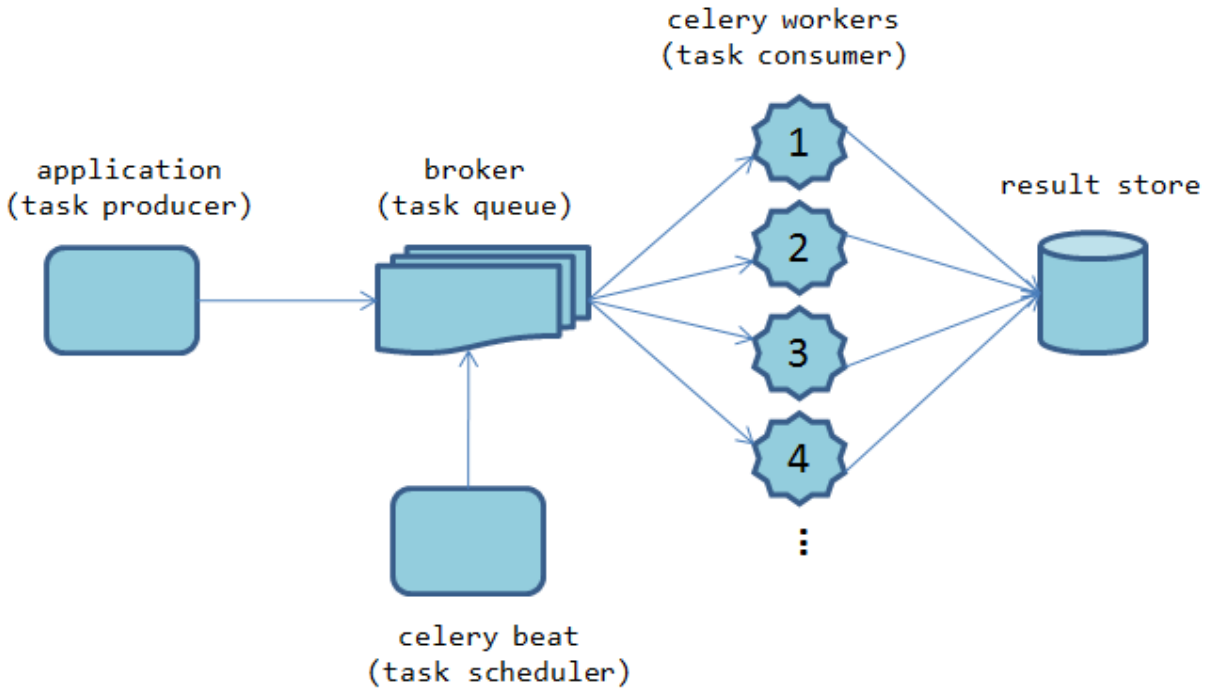
> 在Web 開發中，對新用戶的註冊，我們通常會給他發一封激活郵件，而發郵件是個 IO 阻塞式任務，如果直接把它放到應用當中，就需要等郵件發出去之後才能進行下一步操作，此時用戶只能等待再等待。更好的方式是在業務邏輯中觸發一個發郵件的異步任務，而主程序可以繼續往下運行。

1. **什麼是Celery**
   Celery 是一個用 Python 實作的分散式任務佇列 (Distributed Task Queue)，如果請求中有一些繁複又耗時間的任務 (如新增使用者、寄送電子郵件等等), 因為不能將訪客的連線給拉著不放, 進而導致伺服器無法處理新的請求, 為了避免這樣的情況發生, 我們通常會先將這類的任務送進去 queue 裡面, 啟動常駐的背景程式 (daemon) 來幫我們執行這些任務, 並且將任務的執行過程與結果記錄下來。

使用Celery的常見場景如下：

- **Web應用**
  當使用者觸發的一個操作需要較長時間才能執行完成時，可以把它作為任務交給Celery去非同步執行，執行完再返回給使用者。這段時間使用者不需要等待，提高了網站的整體吞吐量和響應時間。
- **定時任務**
  生產環境經常會跑一些定時任務。假如你有上千臺的伺服器、上千種任務，定時任務的管理很困難，Celery可以幫助我們快速在不同的機器設定不同種任務。
- **同步完成的附加工作**
  同步完成的附加工作都可以非同步完成。比如傳送簡訊/郵件、推送訊息、清理/設定快取等。

2. **Celery的架構**



Celery的架構由三部分組成，訊息中介軟體（message broker），任務執行單元（worker）和任務執行結果儲存（task result store）組成。而任務進來的方式則有兩種: 週期性的定時任務由celery beat發送，而異步任務是呼叫了Celery提供的API、函式或者裝飾器而產生任務並交給任務佇列處理。

- **Broker**
  接受任務生產者傳送過來的任務訊息，存進佇列再按序分發給任務消費方（通常是訊息佇列或者資料庫）。Celery本身不提供訊息服務，但是可以方便的和第三方提供的訊息中介軟體整合。包括: RabbitMQ, Redis, MongoDB, Django ORM, IronMQ 等等。
- **Worker**
  Worker是Celery提供的任務執行的單元，worker併發的執行在分散式的系統節點中，是執行任務的消費者，通常會在多臺伺服器執行多個消費者來提高執行效率。
- **Result store**
  用來儲存Worker執行的任務的結果，Celery支援以不同方式儲存任務的結果，包括AMQP, Redis，memcached, MongoDB，SQLAlchemy, Django ORM，Apache Cassandra, IronCache

使任務進入Celery的兩種元件:

> Celery Beat：任務排程器，Beat程序會讀取配置檔案的內容，週期性地將配置中到期需要執行的任務傳送給任務佇列。
>
> Producer：呼叫了Celery提供的API、函式或者裝飾器而產生任務並交給任務佇列處理的都是任務生產者。

3. **使用範例**

- **選擇Broker**：
  Celery需要接收與發送訊息，因此需要Message broker，有多種選擇但較建議使用RabbitMQ，相較於其他套件RabbitMQ支援監控、遠端控制等功能，且安裝方便，因此以rabbitmq作為broker，因此輸入指令安裝rabbitmq

```
$ sudo apt-get install rabbitmq-server
```

> 檢查是否安裝成功，並確認狀態：

```
$ sudo rabbitmqctl status
```

- **安裝 Celery**：
  Celery包含在Python Package Index中，因此可以透過PIP指令安裝

```
$ sudo pip install celery
```

使用 Celery 實現異步任務主要包含三個步驟：

- **Create a Celery instance**
  實例作為一個入口，裡面包含定義Task以及安排Worker等等，也可以導入任何需要的module

```
#!/usr/bin/env python2
# -*- coding:utf-8 -*-
app = celery.Celery('task', broker='amqp://localhost/', backend='rpc://localhost/')
@app.task
def add(x, y):
    return x + y
@app.task
def multiply(x,y):
    return x * y
```

> Celery(celery application's name, broker='', backend='')
> Borker：若是使用RabbitMQ則輸入'amqp://IP'，範例中為求方便使用本機位置
> Backend：儲存Task狀態及結果，範例中使用RPC，故須輸入'rpc://IP'
> @app.task：可被Celery調度的任務，定義Task前皆須標明

- **啟動 Celery Worker**
  輸入下方code，以啟動Worker：

```
$ celery -A task.app worker --loglevel=info
```

> -A：指定Celery instance的位置，Celery 會自動在該文件中尋找 Celery 實例，也可以自己指定。
> -l info：記錄級別，預設為WARNING，選項有DEBUG, INFO, WARNING, ERROR, CRITICAL, FATAL，可以簡寫為"-l info"

- 應用程序調用異步任務
  由於borker中設置位置為本機，因此打開另一個CMD後執行Python，導入所需的module後便可利用delay()、apply_async()來調度任務：

```
import task
import celery
r1 = task.add.delay(4, 4)
r1.ready()      #確認執行狀況
r1.get()        #取得結果
r2 = task.multiply.apply_async(args(5,5))
```

> 輸入完畢後將會在Worker上看到它執行輸入的任務，可以在Worker的介面或者Backend上看到執行結果。

- 保留結果
  如果想要追蹤Task的狀態，Celery需要Backend作為儲存或傳送訊息的地方
  有許多種類可選擇：
  SQLAlchemy/Django ORM, Memcached, Redis, RPC (RabbitMQ/AMQP)，或是自定義。