**Skylark 爬虫产品后端文档说明书**

# 后端基本架构

1. **flask 框架基本功能介绍**

[Flask](http://flask.pocoo.org/)是一个Python编写的Web 微框架，让我们可以使用Python语言快速实现一个网站或Web服务。本文参考自[Flask官方文档](http://flask.pocoo.org/docs/0.12/) 。Flask是一个基于Python开发并且依赖jinja2模板和Werkzeug WSGI服务的一个微型框架，对于Werkzeug本质是Socket服务端，其用于接收http请求并对请求进行预处理，然后触发Flask框架，开发人员基于Flask框架提供的功能对请求进行相应的处理，并返回给用户 。

“微”(micro) 并不表示你需要把整个 Web 应用塞进单个 Python 文件(虽然确实可以)，也不意味着 Flask 在功能上有所欠缺。微框架中的“微”意味着 Flask 旨在保持核心简单而易于扩展。Flask 不会替你做出太多决策——比如使用何种数据库。而那些 Flask 所选择的——比如使用何种模板引擎——则很容易替换。

pip install flask

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route('/')

def hello\_world():

return 'Hello Flask!'

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

**调试模式**

我们修改代码中的输出，然后查看浏览器上是否有变化。如果你照做的话，可以看到什么变化都没有。其实Flask内置了调试模式，可以自动重载代码并显示调试信息。这需要我们开启调试模式，方法很简单，设置FLASK\_DEBUG环境变量，并将值设置为1。

然后再次运行程序，会看到有这样的输出。这时候如果再次修改代码，会发现这次Flask会自动重启。

一般debug模式会使服务器reload 一遍，这样的操作将会影响我们的后续功能（例如计划任务的执行次数），所以在此服务器中我们一般把服务器reload 的参数设置成false。

\* Restarting with stat

\* Debugger is active!

\* Debugger PIN: 157-063-180

\* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)

**路由（重要功能）：app.route(“/xxx”)**

在上面的例子里可以看到路由的使用。类似于Spring Web MVC的模式，路由通过使用Flask的app.route装饰器来设置，这类似Java的注解。在服务器的url后边直接加上代码中提前设计好的路由后缀，会直接进行该路由写好的代码。其中可以在参数中设置要接受的请求类型,其中包括了POST和GET两种，写好一个app.route(…)即意味对外提供了一个Restful的接口，一般而言当请求进入这个接口之后我们都会对接口含有的数据进行解析，然后根据解析的内容继续运行我们产品的业务逻辑，当前开发版本暂用json.loads和request.form来解析获得request内含的信息。在我们的后端中，接口有对前端适用的作为API服务的接口，也有对内的爬虫节点的接收爬虫数据和task字段信息的内部接口，由于flask强大的接口功能，我们可以灵活编写我们的接口代码和相关逻辑。

@app.route('/')

def index():

return 'Index Page'

@app.route('/hello')

def hello():

return 'Hello, World'

**路径变量**

如果希望获取/article/1这样的路径参数，就需要使用路径变量。路径变量的语法是/path/<converter:varname>。在路径变量前还可以使用可选的转换器，有以下几种转换器。

| 转换器 | 作用 |
| --- | --- |
| string | 默认选项，接受除了斜杠之外的字符串 |
| int | 接受整数 |
| float | 接受浮点数 |
| path | 和string类似，不过可以接受带斜杠的字符串 |
| any | 匹配任何一种转换器 |
| uuid | 接受UUID字符串 |

下面是Flask官方的例子:

@app.route('/user/<username>')

def show\_user\_profile(username):

# show the user profile for that user

return 'User %s' % username

@app.route('/post/<int:post\_id>')

def show\_post(post\_id):

# show the post with the given id, the id is an integer

return 'Post %d' % post\_id

**构造URL**

在Web程序中常常需要获取某个页面的URL，在Flask中需要使用url\_for('方法名')来构造对应方法的URL。下面是Flask官方的例子。

>>> from flask import Flask, url\_for

>>> app = Flask(\_\_name\_\_)

>>> @app.route('/')

... def index(): pass

...

>>> @app.route('/login')

... def login(): pass

...

>>> @app.route('/user/<username>')

... def profile(username): pass

...

>>> with app.test\_request\_context():

... print url\_for('index')

... print url\_for('login')

... print url\_for('login', next='/')

... print url\_for('profile', username='John Doe')

**HTTP方法**

如果需要处理具体的HTTP方法，在Flask 使用route装饰器的methods参数设置即可。

from flask import request

@app.route('/login', methods=['GET', 'POST'])

def login():

if request.method == 'POST':

do\_the\_login()

else:

show\_the\_login\_form()

**静态文件**

Web程序中常常需要处理静态文件，在Flask中需要使用url\_for函数并指定static端点名和文件名。在下面的例子中，实际的文件应放在static/文件夹下。

url\_for('static', filename='style.css')

**模板生成**

Flask默认使用[Jinja2](http://jinja.pocoo.org/docs/2.9/templates/)作为模板，Flask会自动配置Jinja 模板，所以我们不需要其他配置了。默认情况下，模板文件需要放在templates文件夹下。

使用 Jinja 模板，只需要使用render\_template函数并传入模板文件名和参数名即可。

from flask import render\_template

@app.route('/hello/')

@app.route('/hello/<name>')

def hello(name=None):

return render\_template('hello.html', name=name)

相应的模板文件如下。

<!doctype html>

<title>Hello from Flask</title>

{% if name %}

<h1>Hello {{ name }}!</h1>

{% else %}

<h1>Hello, World!</h1>

{% endif %}

**日志输出**

Flask 为我们预配置了一个 Logger，我们可以直接在程序中使用。这个Logger是一个标准的Python Logger，所以我们可以向标准Logger那样配置它，详情可以参考[官方文档](https://docs.python.org/library/logging.html)或者我的文章[Python 日志输出](http://www.jianshu.com/p/9884a660050f)。

app.logger.debug('A value for debugging')

app.logger.warning('A warning occurred (%d apples)', 42)

app.logger.error('An error occurred')

**处理请求**

在 Flask 中获取请求参数需要使用request等几个全局对象，但是这几个全局对象比较特殊，它们是 Context Locals ，其实就是 Web 上下文中局部变量的代理。虽然我们在程序中使用的是全局变量，但是对于每个请求作用域，它们都是互不相同的变量。理解了这一点，后面就非常简单了。

Request 对象

Request 对象是一个全局对象，利用它的属性和方法，我们可以方便的获取从页面传递过来的参数。

method属性会返回HTTP方法的类似，例如post和get。form属性是一个字典，如果数据是POST类型的表单，就可以从form属性中获取。下面是 Flask 官方的例子，演示了 Request 对象的method和form属性。后端初步在使用form表单进行POST请求，在后期若与VM后端进行对接时也将是以form表单形式接受外部的restful请求，下面是一个读取form表单请求的数据的代码实例。

from flask import request

@app.route('/login', methods=['POST', 'GET'])

def login():

error = None

if request.method == 'POST':

if valid\_login(request.form['username'],

request.form['password']):

return log\_the\_user\_in(request.form['username'])

else:

error = 'Invalid username/password'

# the code below is executed if the request method

# was GET or the credentials were invalid

return render\_template('login.html', error=error)

**内部链路的请求**

在后端内部其他节点的服务器传送数据或者发送报告job状态的请求中，我们同样适用POST请求，不过读取时使用 json.loads()来读取字典类型的数据。

@app.route("/cve\_data", methods=['GET', 'POST'])

def handleCVE\_data():

if request.method == 'POST':

data\_list = json.loads(request.data)

# logger.debug(data\_list)

logger.debug(type(data\_list))

logger.debug('this is the area of receiving cve data............')

for each in data\_list:

a=each['vul\_cveId']

b=each['vul\_cweId']

c=each['vul\_numOfExploits']

d=each['vul\_type']

e=each['vul\_publishedDate']

f=each['vul\_updateDate']

g=each['vul\_score']

h=each['vul\_accessLevel']

i=each['vul\_access']

j=each['vul\_complexity']

k=each['vul\_authentication']

l=each['vul\_conf']

m=each['vul\_integ']

n=each['vul\_avail']

o=each['vul\_des']

p=each['vul\_hash']

console\_db.cve\_insert\_data(a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p)

logger.debug('################################################')

logger.debug('all the cve first stage data have been inserted ')

logger.debug('################################################')

return jsonify({'msg':'successful.....'})

**重定向和错误**

redirect和abort函数用于重定向和返回错误页面。

from flask import abort, redirect, url\_for

@app.route('/')

def index():

return redirect(url\_for('login'))

@app.route('/login')

def login():

abort(401)

this\_is\_never\_executed()

默认的错误页面是一个空页面，如果需要自定义错误页面，可以使用errorhandler装饰器。

from flask import render\_template

@app.errorhandler(404)

def page\_not\_found(error):

return render\_template('page\_not\_found.html'), 404

1. **skylark 后端基本架构**

借助于flask框架强大的路由功能，我们将后端的功能分成双层次的节点分布，分别是 console，香港节点(hk\_node)和上海节点(sh\_node)。

console ：处理vm或其他前端平台的RESTful请求，行使数据库的CRUD任务操作，读取爬虫节点的数据，以及运行制造报告模版样式的脚本。当然，一些辅助性的脚本（诸如cve\_detail\_setup，cve的第二阶段）和一些与数据库有紧密联系的脚本（诸如Github search的脚本）。

hk\_node : 处理console发送的爬虫请求信息，下达爬虫任务并存储任务的基本信息，进行香港及海外网站的爬虫任务并将任务状态信息和爬取的数据发送到console节点。

sh\_node : 处理console发送的爬虫请求信息，下达爬虫任务并存储任务的基本信息，进行香港及海外网站的爬虫任务并将任务状态信息和爬取的数据发送到console节点。

1. **skylark APscheduler 基本介绍**

APScheduler 是python的一个任务调度框架。  
使用起来十分方便，提供了基于日期、固定时间间隔以及crontab类型的任务，并且可以持久化任务。在我们的服务器中一般使用BackgroundScheduler,使其能够在后台异步进行，不影响服务器的运作。

**Class apscheduler.job.Job(scheduler, id=None, \*\*kwargs)**

Contains the options given when scheduling callables and its current schedule and other state. This class should never be instantiated by the user.

一个job含有以下的变量：

|  |  |
| --- | --- |
|  | [**id**](https://docs.python.org/3/library/functions.html#id)**(**[**str**](https://docs.python.org/3/library/stdtypes.html#str)**)** – the unique identifier of this job  **name (**[**str**](https://docs.python.org/3/library/stdtypes.html#str)**)** – the description of this job  **func** – the callable to execute  **args (tuple|list)** – positional arguments to the callable  **kwargs (**[**dict**](https://docs.python.org/3/library/stdtypes.html#dict)) – keyword arguments to the callable  **coalesce (**[**bool**](https://docs.python.org/3/library/functions.html#bool)**)** – whether to only run the job once when several run times are due  **trigger**– the trigger object that controls the schedule of this job  **executor (**[**str**](https://docs.python.org/3/library/stdtypes.html#str)**)** – the name of the executor that will run this job  **misfire\_grace\_time (**[**int**](https://docs.python.org/3/library/functions.html#int)**)**–the time (in seconds) how much this job’s execution is allowed to be late  **max\_instances (**[**int**](https://docs.python.org/3/library/functions.html#int)**)** – the maximum number of concurrently executing instances allowed for this job  **next\_run\_time (**[**datetime.datetime**](https://docs.python.org/3/library/datetime.html#datetime.datetime)**)** – the next scheduled run time of this job |

**Note：**

The misfire\_grace\_time has some non-obvious effects on job execution. See the [Missed job executions and coalescing](http://apscheduler.readthedocs.io/en/latest/userguide.html#missed-job-executions) section in the documentation for an in-depth explanation.

modify(\*\*changes)

**基本方法介绍：**

[**modify\_job(\*\*changes 将属于job的variable重新定义输入可以将该任务进行改变)**](http://apscheduler.readthedocs.io/en/latest/modules/schedulers/base.html#apscheduler.schedulers.base.BaseScheduler.modify_job)

**pause()**

Temporarily suspend the execution of this job.暂停该任务

也等于[**pause\_job()**](http://apscheduler.readthedocs.io/en/latest/modules/schedulers/base.html#apscheduler.schedulers.base.BaseScheduler.pause_job)

**remove()**

Unschedules this job and removes it from its associated job store.直接在计划任务中去除该任务。

**也等于**[**remove\_job()**](http://apscheduler.readthedocs.io/en/latest/modules/schedulers/base.html#apscheduler.schedulers.base.BaseScheduler.remove_job)

**reschedule(trigger, \*\*trigger\_args)**

Shortcut for switching the trigger on this job.接收trigger对象，将该任务重新定义执行的规则或者时间。

也等于[**reschedule\_job()**](http://apscheduler.readthedocs.io/en/latest/modules/schedulers/base.html#apscheduler.schedulers.base.BaseScheduler.reschedule_job)

**resume()**

Resume the schedule of this job if previously paused.重新开始该任务。

也等于[**resume\_job()**](http://apscheduler.readthedocs.io/en/latest/modules/schedulers/base.html#apscheduler.schedulers.base.BaseScheduler.resume_job)

**Class apscheduler.triggers.cron.CronTrigger(year=None, month=None, day=None, week=None, day\_of\_week=None,**

**hour=None, minute=None, second=None, start\_date=None, end\_date=None, timezone=None, jitter=None)**

Triggers when current time matches all specified time constraints, similarly to how the UNIX cron scheduler works.

初始化一个cron trigger所能够使用的参数如下所示：

|  |  |
| --- | --- |
|  | **year (int|str)** – 4-digit year  **month (int|str)** – month (1-12)  **day (int|str)** – day of the (1-31)  **week (int|str)** – ISO week (1-53)  **day\_of\_week (int|str)** – number or name of weekday (0-6 or mon,tue,wed,thu,fri,sat,sun)  **hour (int|str)** – hour (0-23)  **minute (int|str)** – minute (0-59)  **second (int|str)** – second (0-59)  **start\_date (datetime|str)** – earliest possible date/time to trigger on (inclusive)  **end\_date (datetime|str**) – latest possible date/time to trigger on (inclusive)  **timezone (datetime.tzinfo|str)** – time zone to use for the date/time calculations (defaults to scheduler timezone)  **jitter (int|None)** – advance or delay the job execution by jitter seconds at most. |

**Note**

weekday 的第一天永远是Monday

**Introduction Cron Trigger 基本介绍**

This is the most powerful of the built-in triggers in APScheduler. You can specify a variety of different expressions on each field, and when determining the next execution time, it finds the earliest possible time that satisfies the conditions in every field. This behavior resembles the “Cron” utility found in most UNIX-like operating systems.

You can also specify the starting date and ending dates for the cron-style schedule through “the start\_date” and “end\_date” parameters, respectively. They can be given as a date/datetime object or text (in the [ISO 8601](https://en.wikipedia.org/wiki/ISO_8601) format).

Unlike with crontab expressions, you can omit fields that you don’t need. Fields greater than the least significant explicitly defined field default to \* while lesser fields default to their minimum values except for week and day\_of\_week which default to \*.

For example, day=1, minute=20 is equivalent to

“year='\*', month='\*', day=1, week='\*', day\_of\_week='\*', hour='\*', minute=20, second=0”.

The job will then execute on the first day of every month on every year at 20 minutes of every hour. The code examples below should further illustrate this behavior.

**Note**

The behavior for omitted fields was changed in APScheduler 2.0. Omitted fields previously always defaulted to \*.

Expression types

The following table lists all the available expressions for use in the fields from year to second. Multiple expression can be given in a single field, separated by commas.

| Expression | Field | Description |
| --- | --- | --- |
| \* | Any | Fire on every value |
| \*/a | any | Fire every a values, starting from the minimum |
| a-b | any | Fire on any value within the a-b range (a must be smaller than b) |
| a-b/c | any | Fire every c values within the a-b range |
| xth y | day | Fire on the x -th occurrence of weekday y within the month |
| last x | day | Fire on the last occurrence of weekday x within the month |
| last | day | Fire on the last day within the month |
| x,y,z | any | Fire on any matching expression; can combine any number of any of the above expressions |

**Note**

The month and day\_of\_week fields accept abbreviated English month and weekday names (jan – dec and mon – sun) respectively.

**Daylight saving time behavior**

The cron trigger works with the so-called “wall clock” time. Thus, if the selected time zone observes DST (daylight saving time), you should be aware that it may cause unexpected behavior with the cron trigger when entering or leaving DST. When switching from standard time to daylight saving time, clocks are moved either one hour or half an hour forward, depending on the time zone. Likewise, when switching back to standard time, clocks are moved one hour or half an hour backward. This will cause some time periods to either not exist at all, or be repeated. If your schedule would have the job executed on either one of these periods, it may execute more often or less often than expected. This is not a bug. If you wish to avoid this, either use a timezone that does not observe DST, for instance UTC. Alternatively, just find out about the DST switch times and avoid them in your scheduling.

For example, the following schedule may be problematic:

**# In the Europe/Helsinki timezone, this will not execute at all on the last sunday morning of March**

**# Likewise, it will execute twice on the last sunday morning of October**

sched.add\_job(job\_function, 'cron', hour=3, minute=30)

**Examples（代码范例）:**

from apscheduler.schedulers.blocking import BlockingScheduler

def job\_function():

print "Hello World"

sched = BlockingScheduler()

**# Schedules job\_function to be run on the third Friday**

**# of June, July, August, November and December at 00:00, 01:00, 02:00 and 03:00**

**# 使用 job\_function 作为这个apscheduler的callable，如果有参数，还有一个args的数组参数，可以将我们的参数写成一个数组装在add\_job（）中。**

sched.add\_job(job\_function, 'cron', month='6-8,11-12', day='3rd fri', hour='0-3')

sched.start()

You can use start\_date and end\_date to limit the total time in which the schedule runs:

**# Runs from Monday to Friday at 5:30 (am) until 2014-05-30 00:00:00：**

sched.add\_job(job\_function, 'cron', day\_of\_week='mon-fri', hour=5, minute=30, end\_date='2014-05-30')

**使用装饰器来增加任务，不过一般很少用:**

@sched.scheduled\_job('cron', id='my\_job\_id', day='last sun')

def some\_decorated\_task():

print("I am printed at 00:00:00 on the last Sunday of every month!")

**Class apscheduler.triggers.interval.IntervalTrigger(weeks=0, days=0, hours=0, minutes=0, seconds=0,**

**start\_date=None, end\_date=None, timezone=None, jitter=None)**

Triggers on specified intervals, starting on start\_date if specified, datetime.now() + interval otherwise.

按照时间区间来执行的另外一个trigger，以下是他所使用的参数列表：

|  |  |
| --- | --- |
|  | **weeks (**[**int**](https://docs.python.org/3/library/functions.html#int)**)** – number of weeks to wait  **days (**[**int**](https://docs.python.org/3/library/functions.html#int)**)** – number of days to wait  **hours (**[**int**](https://docs.python.org/3/library/functions.html#int)**)** – number of hours to wait  **minutes (**[**int**](https://docs.python.org/3/library/functions.html#int)**)** – number of minutes to wait  **seconds (**[**int**](https://docs.python.org/3/library/functions.html#int)**)** – number of seconds to wait  **start\_date (datetime|str)** – starting point for the interval calculation  **end\_date (datetime|str)** – latest possible date/time to trigger on  **timezone (datetime.tzinfo|str)** – time zone to use for the date/time calculations  **jitter (int|None)** – advance or delay the job execution by jitter seconds at most. |

**Introduction**

This method schedules jobs to be run periodically, on selected intervals.

You can also specify the starting date and ending dates for the schedule through the start\_date and end\_date parameters, respectively. They can be given as a date/datetime object or text (in the [ISO 8601](https://en.wikipedia.org/wiki/ISO_8601) format).

If the start date is in the past, the trigger will not fire many times retroactively but instead calculates the next run time from the current time, based on the past start time.

**Examples:**

from datetime import datetime

from apscheduler.schedulers.blocking import BlockingScheduler

def job\_function():

print("Hello World")

sched = BlockingScheduler()

**# Schedule job\_function to be called every two hours**

sched.add\_job(job\_function, 'interval', hours=2)

sched.start()

**You can use start\_date and end\_date to limit the total time in which the schedule runs:**

**# The same as before, but starts on 2010-10-10 at 9:30 and stops on 2014-06-15 at 11:00**

sched.add\_job(job\_function, 'interval', hours=2, start\_date='2010-10-10 09:30:00', end\_date='2014-06-15 11:00:00')

The [scheduled\_job()](http://apscheduler.readthedocs.io/en/latest/modules/schedulers/base.html#apscheduler.schedulers.base.BaseScheduler.scheduled_job) decorator works nicely too:

from apscheduler.scheduler import BlockingScheduler

@sched.scheduled\_job('interval', id='my\_job\_id', hours=2)

def job\_function():

print("Hello World")

1. **skylark 基本模式的代码说明和拓展思维**

Skylark 的基本功能包括任务监控，定时邮件发送，对外Restful API以及管理员界面进行任务指派和调整。

Skylark 的代码文档将会根据业务逻辑和功能分解为不同的行为模式，以一个行为模式为最基本单位，将代码中的方法按照行为模式的先后顺序进行解读和说明。

**行为模式1 – 对外的API接口读取任务指派的相关信息和字段，根据字段的信息来决定任务的执行时间的区间（开始时间，结束时间）以及执行频率和使用的爬取的脚本。同时对任务进行初始化，并将任务的基本信息存入数据库中。**

任务派发的基本信息对应字段分别由以下映射表所呈现：

| 任务字段 | 对应字段（与数据库的列名一致） |
| --- | --- |
| 任务owner | task\_owner |
| 开始时间 | task\_startTime |
| 结束时间 | task\_endTime |
| 执行频次 | task\_sequence |
| 爬虫脚本ID | task\_crawlerId |

以上字段能够基本决定一个任务的基本信息：定时由何时开始执行脚本，何时结束，任务的拥有者以及执行哪个脚本。详细的字段说明请参考CTI服务的数据结构说明书。

此类字段我们将其作为初始化的字段，由前端发送到flask 的固定接口（在此我们暂定一个接口负责一个脚本也就是url/cve将执行cve这一个爬虫任务，所以能够进入cve接口也就一定会是“cve”的task\_crawlerId，后续的开发人员若进行拓展，可以将flask 的app.route接口作为一个比较广泛的接口，例如url/compliance，在此接口的代码逻辑里再根据task\_crawlerId 来派发任务。）

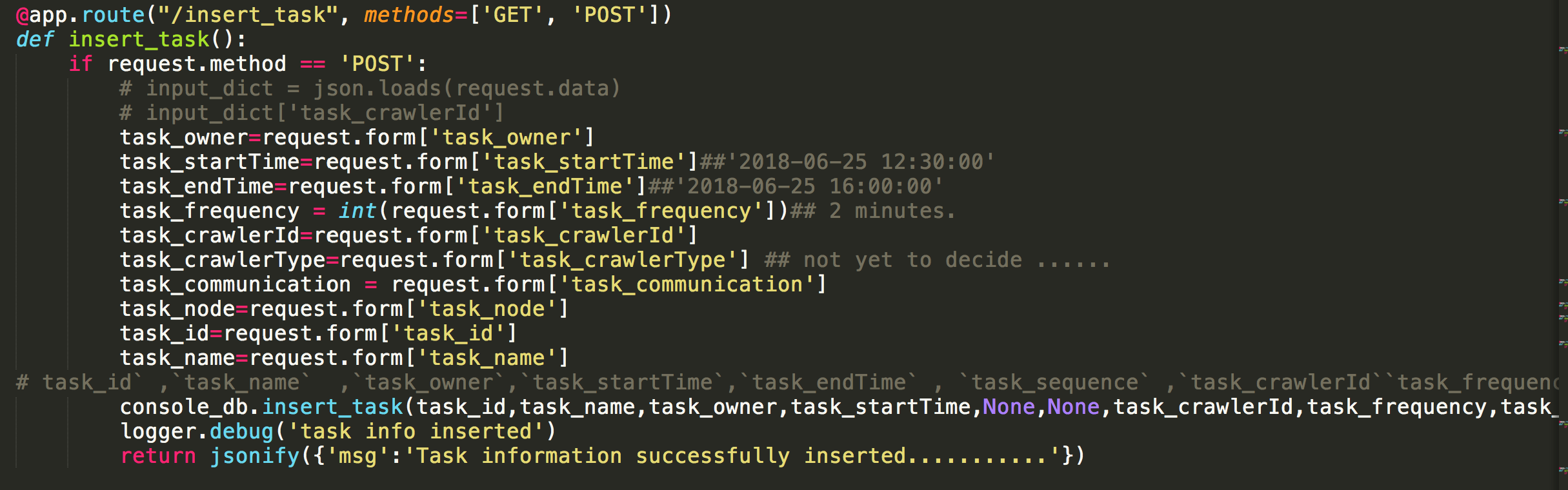
因此，前端的用户可以发送POST请求，将关键的字段信息装载在POST请求的form-data (json格式的数据，以取字典键值的方式进行信息读取),在开发阶段我们使用linux command 直接curl服务器进行请求因此读取数据选用json.loads的方式来读取字段信息，如果用form-data进行发送可以选用request.form[‘字段’]进行读取。由于console节点进行派发任务，所以字段信息取出来之后我们将会再重新包装成一个字典(在此例子中我们原封不动地将信息复制重新包装，但是后续如果对再派发的字典进行拓展这样的写法会比较好，因此我们选用比较麻烦的方式来重写这个字典)，然后再将该字典放到POST请求中发送到爬虫节点。

此时我们包含请求基本信息的POST请求已经发送到第二层的爬虫节点(爬虫节点url/cve\_order,也就是hk\_main的主脚本)中，在这个例子中将是HK节点，我们将会把传送过来的字典信息读取出来，但是这些只是一些决定任务的请求的，我们将开始由这些信息定制一个task，首先是task\_id,我们将根据uuid包内的方法来制作属于这个task的主键，并将当前的时间戳记录下来作为创建时间。

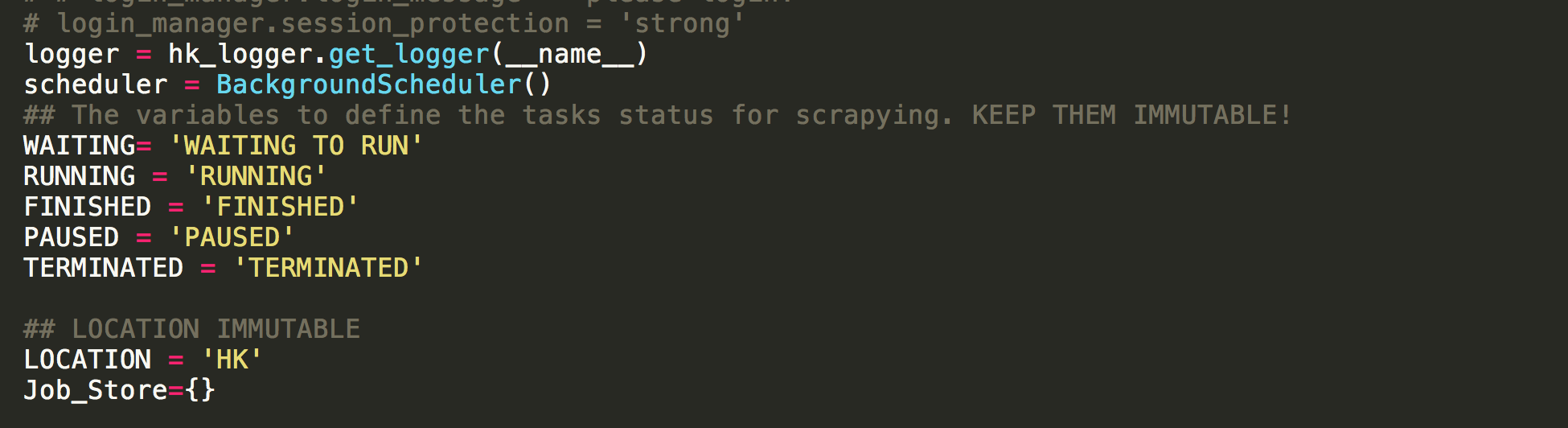
(created\_time = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")，标准化其格式)

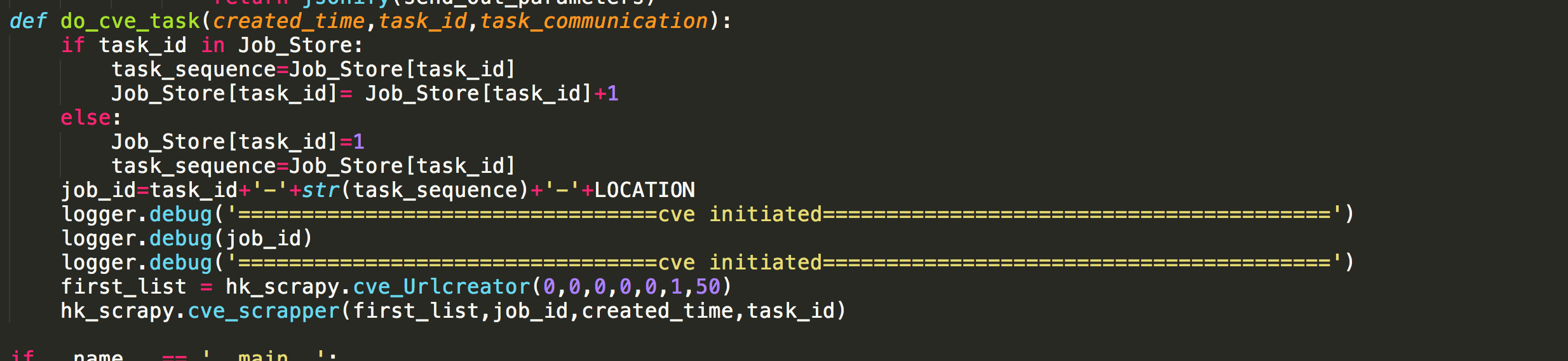
在这个爬虫节点的接口方法内，我们再取出来这个字典并再次重新包装成一个全新的字典，这次的新字典将会包含created\_time和task\_id两个字段的信息，此时该字典包含的信息涵盖了一个task 的全部基本信息(后续待业务逻辑丰满了之后也需要将该字典的字段进行补充，例如task\_url和task\_modelid等，按照此代码逻辑进行补充)，也就意味着我已经创建了这个task，他将重新发送POST请求回到console 节点进行存库，并且执行apscheduler 的计划任务的下达，我们在add\_job的方法内增添了这个计划任务的id，就直接使用我们刚刚做出来的task\_id来作为唯一识别的id。调用的方法是我们自己写的一个do\_cve\_task(建议 apscheduler 的callable方法一般是do\_xxx\_task)，我们的do\_cve\_task将会如第三幅图所示，增加这个task的执行批次，制成job\_id并传到具体的爬虫脚本的方法内。

此时将装好task全部基本信息的字典传送给console 界面的一个接口(app.route(“/insert\_task”), 也就是console-url/insert\_task),我们的console将会对该任务的基本信息进行处理，取出来之后再进行标准化的格式处理(例如将持续时间简化成标准格式) 。这是一个标准化的接口，后续的开发人员仅需装载好标准化的字典放到POST的请求中，console的接口会直接解析后存库。



当到定点的开始时间后，do\_cve\_task将会被执行，此时job就会被生成。我们的做法是在hk\_main增加一个全局的字典，同样存每个task作为该字典的键，对应的值将会是他的执行批次。后续的拓展建议是写进环境配置中或者存储在txt 文档中。那么第一次我们在初始化这个task之后会给他制造一个sequence 作为1，在do\_cve\_task中我们将先将把Job\_store中的task所对应的批次赋值给task\_sequence，同时制造出job\_id(task-task\_sequence-LOCATION),再将job\_id 传人具体的爬虫脚本中。





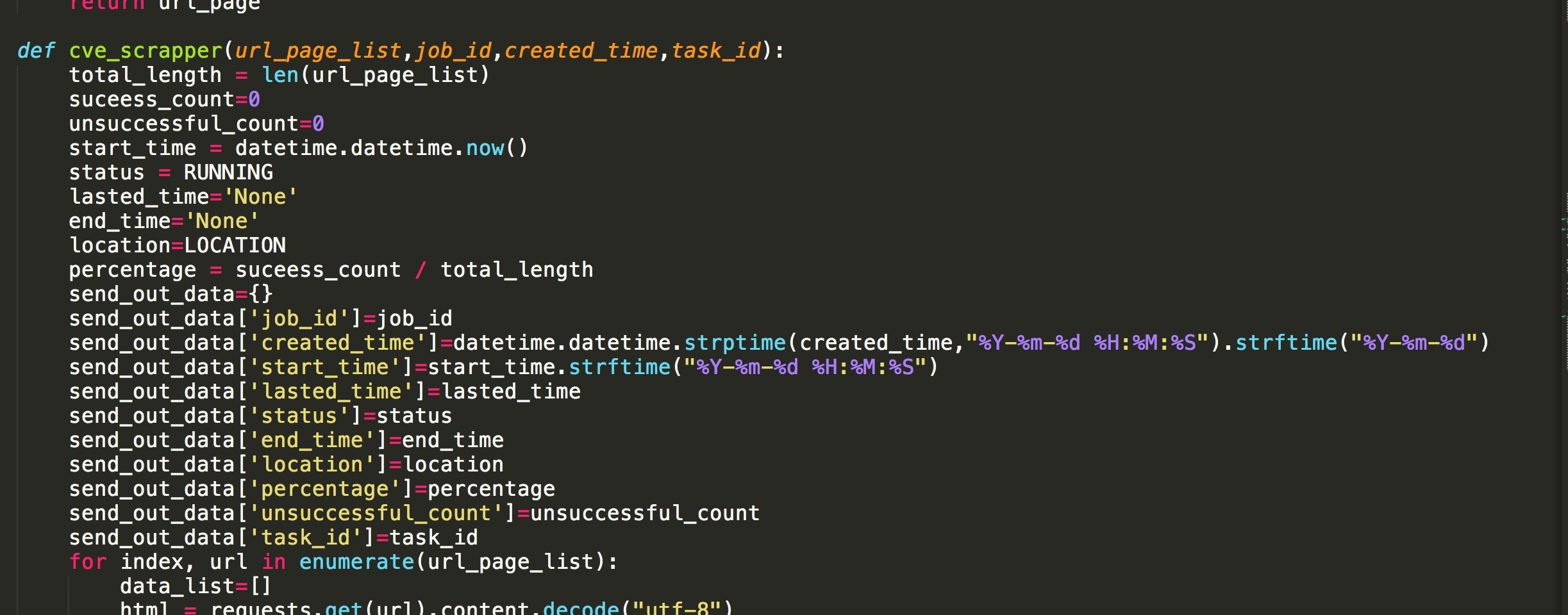
至此我们就完成了一个任务的下达，并且将一些必须的字段信息进行传送和制造，在console和hk节点中进行联调。Console 拥有对外的接口和接收task信息的接口，hk\_main则有实际接收task任务下达的接口。

**行为模式2 –爬虫的job进行网页爬取并将数据传输给console节点，同时内嵌job的监控的代码逻辑**

延续行为模式1的断点，当时间节点到了之后，此时我们会开始我们的job，也就是具体的爬虫任务，那么需要内嵌一些监控job的代码。一般而言，我们的监控代码需要将原有的爬虫脚本的代码进行修改，也需要根据其逻辑来监控他的进度，所以每一个脚本的代码写法是一个隐形的艺术，我们一般会用try: 和except Exception as e:的逻辑来判断单次loop是否成功。一般而言爬虫的脚本都是有在for loop中进行，这也就天然提供了一个监控进度的机制，我们一般会选一个loop的次数作为总的进度数，走完一个loop视作成功的loop加一次。

那么我们需要用一个字典来初始化我们这个job的任务状态信息，在其中加入标准化的字段。

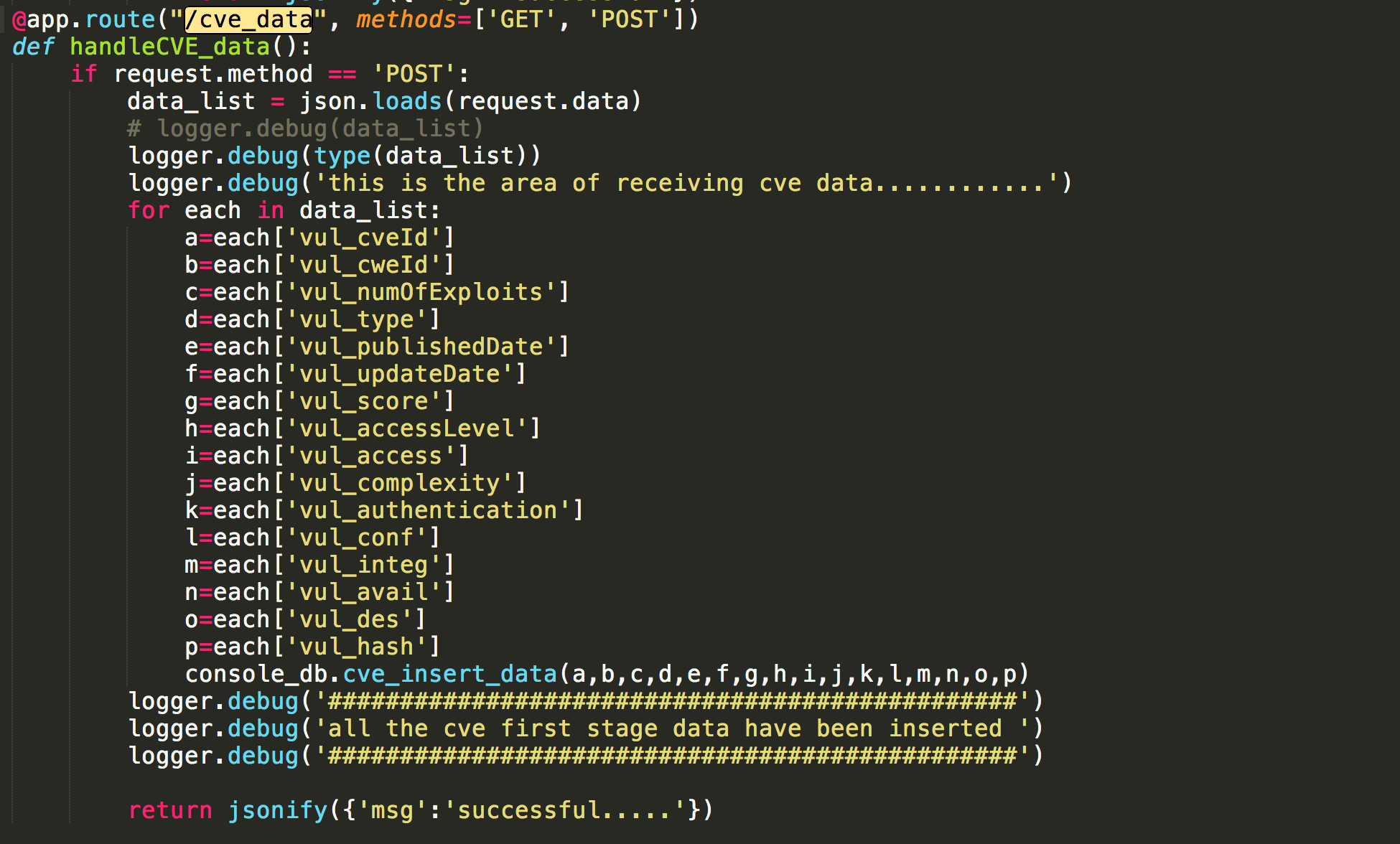
值得注意的是datetime object转换成标准化格式的字符串的时机，和变量赋值的写法，这是出于后续我们计算job持续时间时仍需要用start\_time – end\_time来获得timedelta 的object。



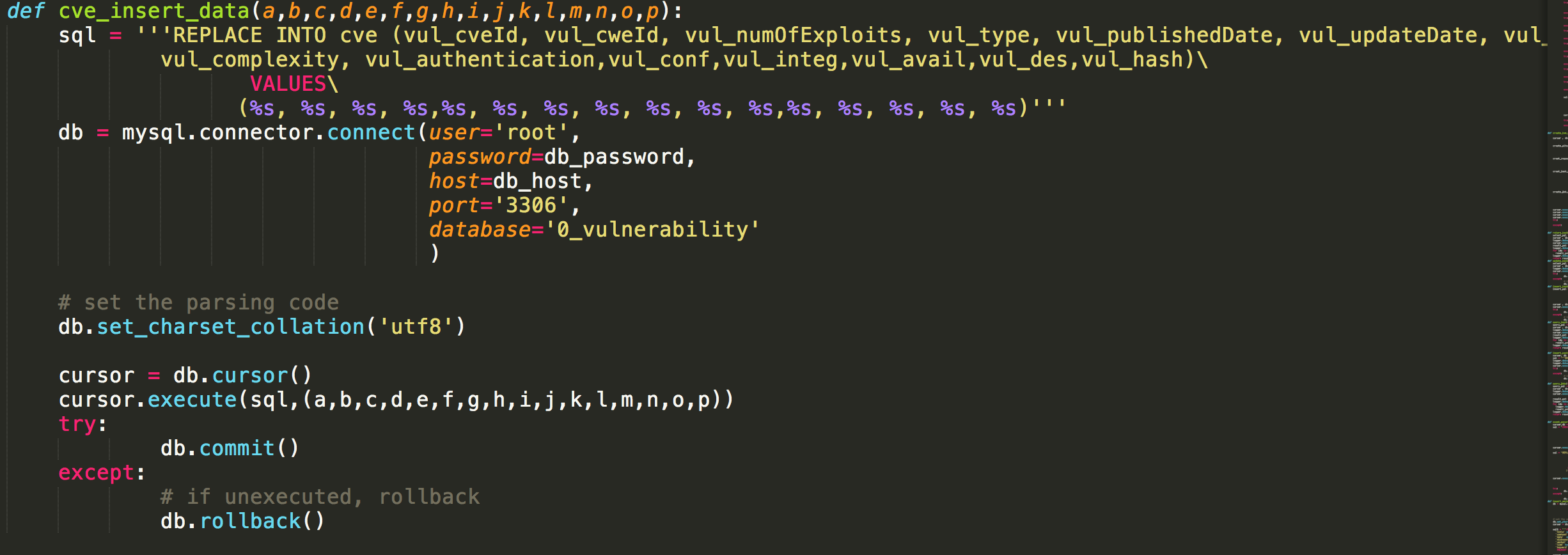
同时，当爬虫的代码爬取完网页的数据之后，我们一般的做法是在更大范围的loop中先实例化一个数组(如果每个小loop是爬一条数据)，然后在小范围的loop中(也就是我们爬取一条数据的loop中)将爬取的数据装载成一个字典，按照POST的请求传输数据的形式再传送回给console节点。

由下图我们可以看到爬取的数据将会被被装载成字典，加入到大for loop的数组中，最后我们大for loop结束的时候以json.dumps()的形式传输回给console 节点。

那么此时console 节点就需要再自己写一个接口，来处理我们的爬取数据。在这里我们传输的接口叫做cve\_data。



同样，我们将传输过来的数据，重新json.loads为数组，再逐个处理。此时使用console\_db我们写好的方法进行存库。



**重要提示-如何拆解原生的爬虫脚本：当前版本仅将功能搭建完成，后续的开发人员需要重复性地将原生写好的爬虫脚本拆解好，基本的方法是，把原生的爬虫脚本的数据库操作，写成console 节点的一个方法，再在console 的接收data 的接口中进行调用，而装在无论是HK或者SH节点的脚本则会用字典来取代数据库操作，把数据装进字典中。再通过HTTP 请求进行传输。**

值得注意的是console 里含有cve的第二阶段的代码，他根据第一阶段的代码来读取数据，再将数据筛选处理，与数据库连接紧密，所以我们直接将其制作成一个方法，装载在console 节点，选择节点也是根据脚本的基本类型来决定的。#Black ART

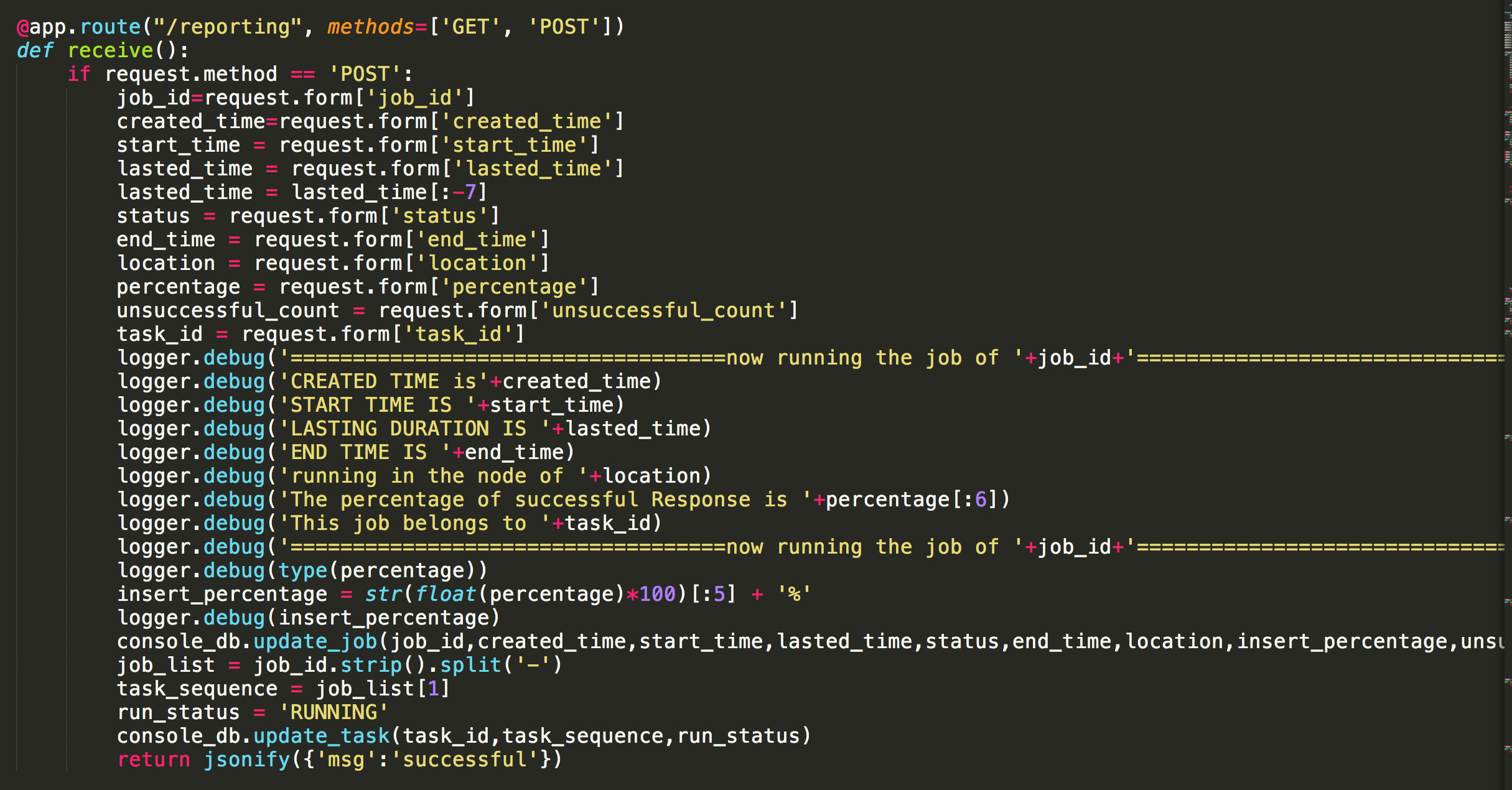
回到我们的HK节点，hk\_scrapy中仍在进行爬虫，此时每走一个loop 就会更新一次percentage，按照成功次数/总次数来计算，在走完所有的loop之后我们依然会对job的状态信息进行一次清算，在大loop的外面，爬虫任务已经结束，那么此时，lasted\_time和end\_time 可以根据当前时间戳来计算，而percentage作为最后的结果也能得到，job的status 将会变成“FINISHED”。

此时我们的console也有一个标准化的统一的接口来接收我们的job信息，称为/reporting。

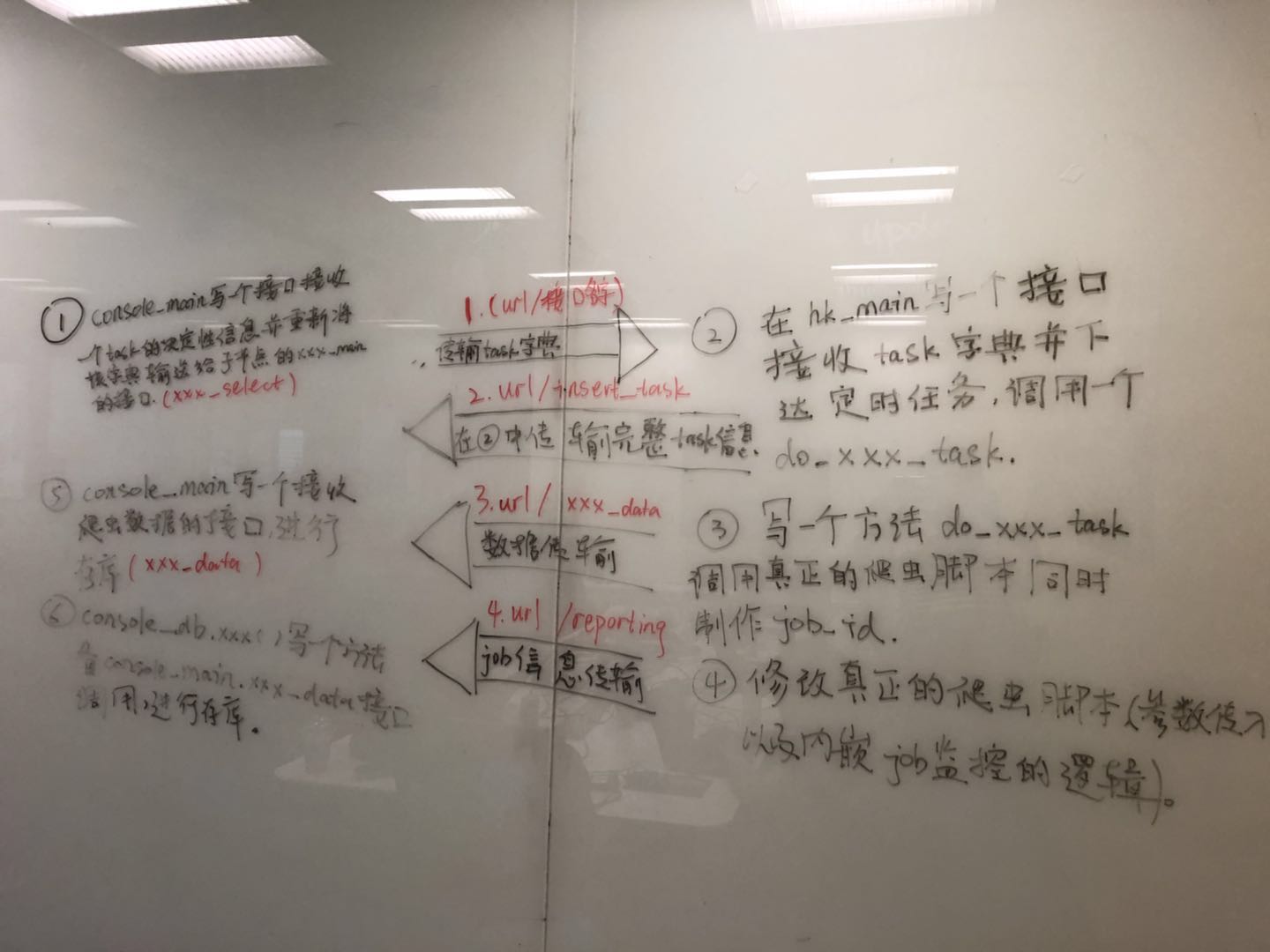
我们会在for loop 的里面，以及任务完成之后，不停地发送POST请求来更新任务状态信息。个人建议是不要太过频繁的发送无意义的更新请求，在重要的进度变化发生之后，再发送请求给reporting的接口。由此总结的规律是，insert\_task接口会被接受一次，存进库内的task表中，而reporting作为对job的接口，则会被调用多次。

此时类似于接收task 字段信息的接口，我们会对传过来的字典进行处理后存库，使用的是console\_db中写好的方法来进行存库。

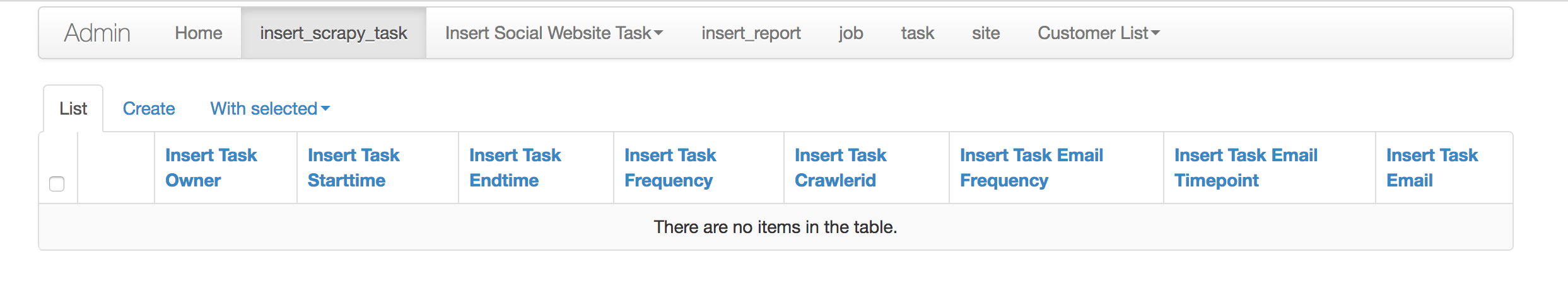
同时根据job\_id撕裂后的结果，我们还能对已经存库的task的进行批次的更新，或者状态的更新。后续的开发人员若希望拓展代码逻辑可以遵循这样的逻辑继续改写方法，或者调整reporting 的固化模块。



**至此，一个cve的任务，就此完成，从RESTFUL 接口获得决定任务的信息到最后定点完成，包括了数据爬取传输和任务状态监控的所有代码逻辑，在console 与HK 的两个节点的交互中完成。**



**行为模式3–从管理员界面下发任务，听取创建的字段之后根据字段信息来下发任务进行爬虫，并同时设定发送邮件的时间间隔和时间点。**

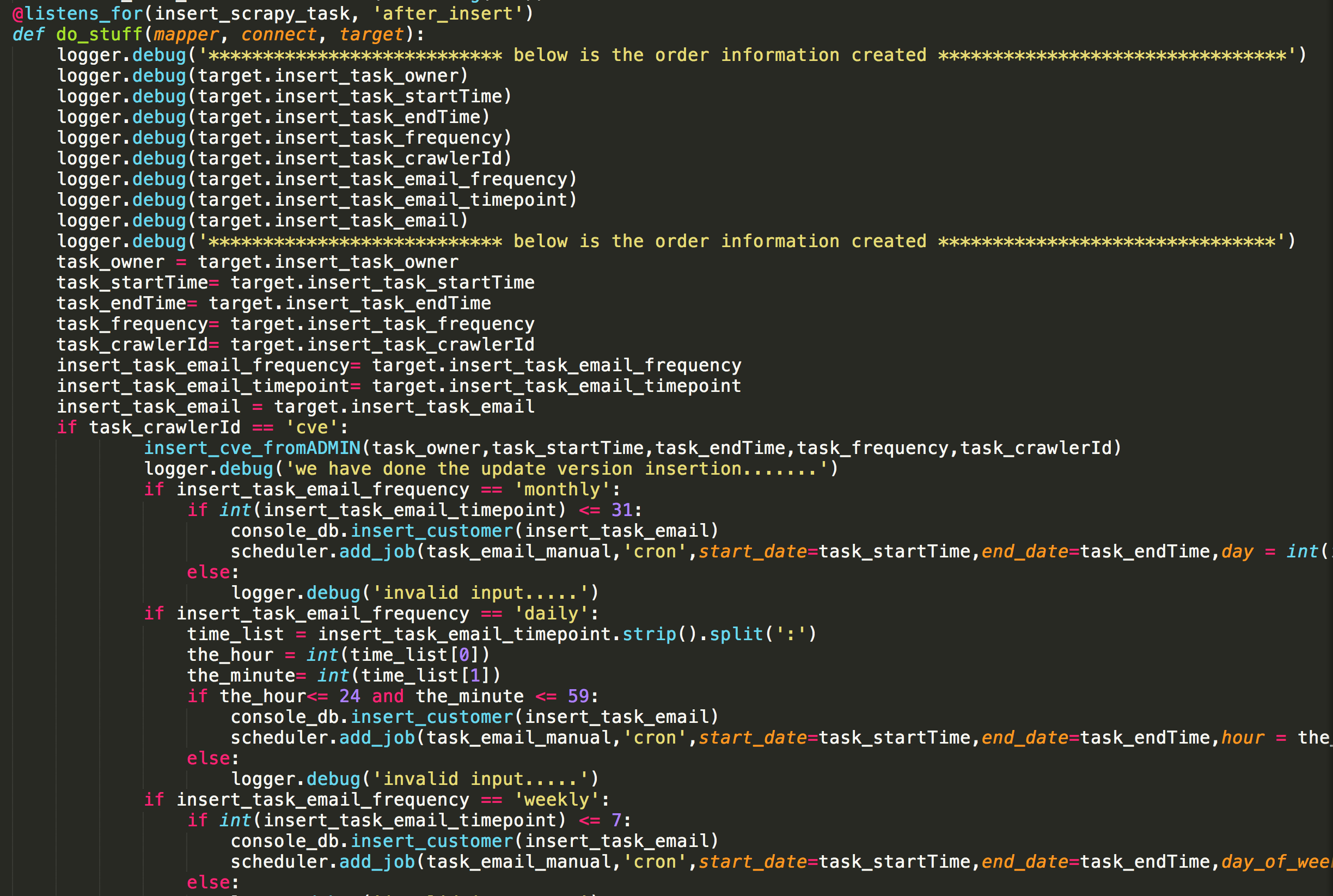
****

****

往往管理员也需要有权限来直接下达任务，我们的做法是在SQLAlchemy中直接创建一个叫做insert\_scrapy\_task的列表来的进行任务的下达，类似于对外的RESTFUL接口，在这里的各个字段都将决定一个任务的下发，创建一个insert\_scrapy\_task模块的几大步骤分别为：

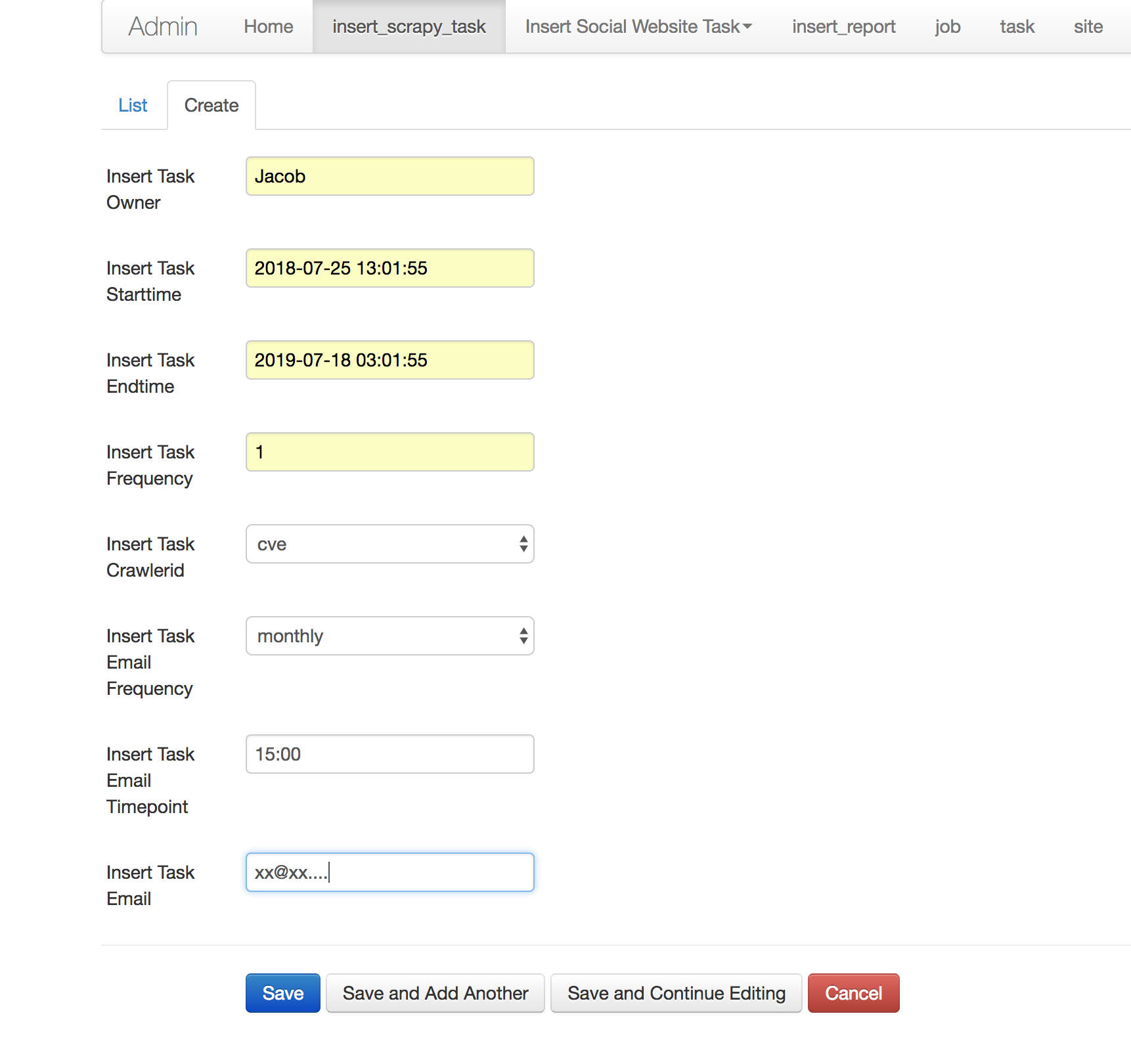
1. FORM(Optional 可写可不写在这个需要输入字段和下拉菜单等需求的情况下，我们将创建属于自己的form，待后续的ModelView模块进行复写使用。)
2. Db 的Model，需要在console main 中将数据库中的表直接可视化出来，主键我们一般将owner 和starttime组成复合主键，如此便可以直接下达任务。
3. ModelView，装载这个db Model 的一个视图模块，内置有很多参数，如果后续开发人员有更多复杂的需求须查看flask admin的model 的文档和相关的方法。
4. event.listens\_for的函数，此听取机制是在insert\_scrapy\_task创建单条数据之后进行代码逻辑的实现，具体方法是根据model(也就是我们的target)的相关字段来决定我们的代码逻辑。







**需要注意的是，listens\_for需要放置在db.Model后，ModelView的前面，否则报错。**

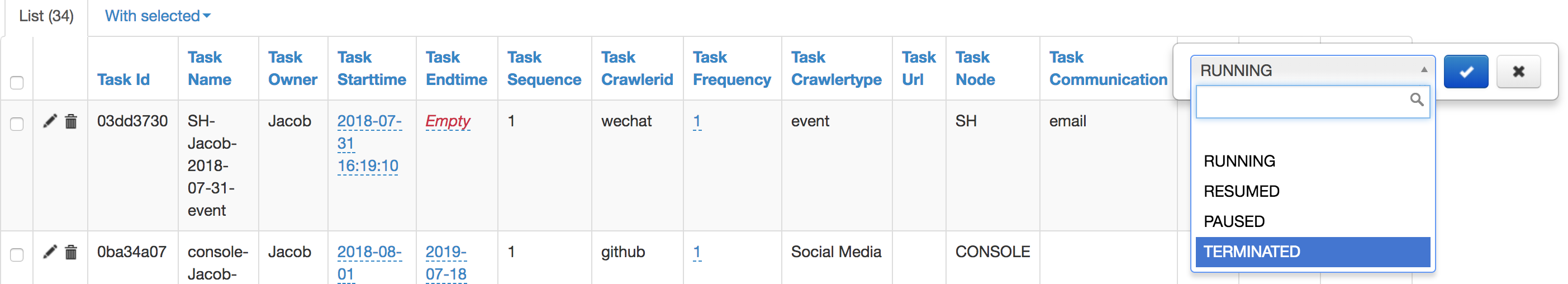


创建了任务的相关字段信息之后，我们会根据管理界面的要求来创建，当前版本的功能搭建基本完毕，但是更多拓展的代码逻辑将根据实际需求来进行拓展。(比如time point 需要以xx:xx 的数字格式，任何时间节点均以20180-07-01 xx:xx:xx的格式进行输入，当前版本对于这里的输入暂时要求比较严苛，但是后续可以继续各种容错机制的优化)

在写好了listens\_for的代码逻辑之后，我们将会遵循api接口的同样的模式将装载好的字典POST发送到爬虫节点，仅仅再重新写一个装载字典的方法以便定时计划器进行调用，并按照行为模式1和2继续进行同样的爬虫任务，我们也可以根据同样的逻辑进行拓展。

由上图可见在属于不同的频率的代码逻辑中我们会要求用户输入不同的timepoint, 例如在monthly则输入每个月的几号接受邮件，而daily则要求用户输入当天几分几秒接收到邮件，weekly 则要求用户输入一周的第几天接收到邮件，如果选择了monthly或者weekly则默认在早上9点进行邮件的发送。

**行为模式4– 根据管理界面进行任务调控和相关字段调整**

****

管理员的权限往往会有直接更高任务的状态和相关字段的权利，我们可以根据下拉菜单来变换任务的基本信息，由此可以达到远程操控任务的目的。

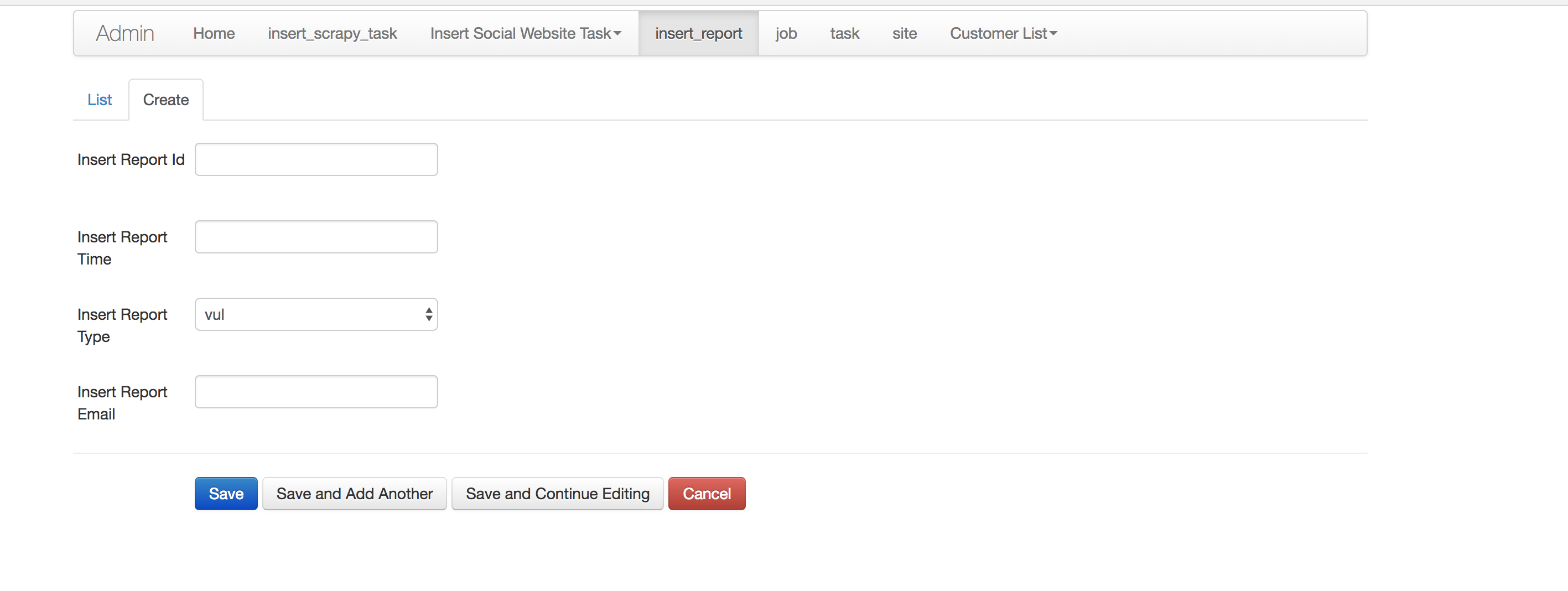
在设计好task的form 模块之后，除了RUNNING还有各种状态，以及直接更改时间执行的时间戳的字符串。

在更新字段之后，我们的listens\_for会听取update的行为，并且进一步根据这一条数据的内容进行代码逻辑的实现，在此的after\_update的代码中我们的做法是新的全部可能更改的字段直接装载成字典，通过POST请求发送给各个节点中的edit\_task接口。

由此我们通过把task更新过的字典进行传输，到达存储了有以该task\_id为任务id的apscheduler的节点，来到这个节点我们可以直接更改这个任务的基本状态，当前版本示范了如何更新开始的时间，当然也可以更新结束时间，直接加入end\_time即可。后续进行拓展可以参考apscheduler的document使用modify或者reschedule来对任务进行调整。同时我们也可以对任务进行状态上的调整，例如task会被暂停，终止，重新开始，如果传过来的new\_status不是RUNNING那么说明我们对于task的任务有所调整，此时的代码逻辑就会根据具体需求来调整任务的状态。

此时通过apscheduler的API我们便可以很方便地对任务进行远程操控。

**行为模式5– 根据管理界面进行单次邮件发送**

****

同理我们可以在事先创建的insert\_report 的表中进行数据插入，再以听取的方法中直接根据创建的单条数据来决定发送时间和发送的目标邮箱。

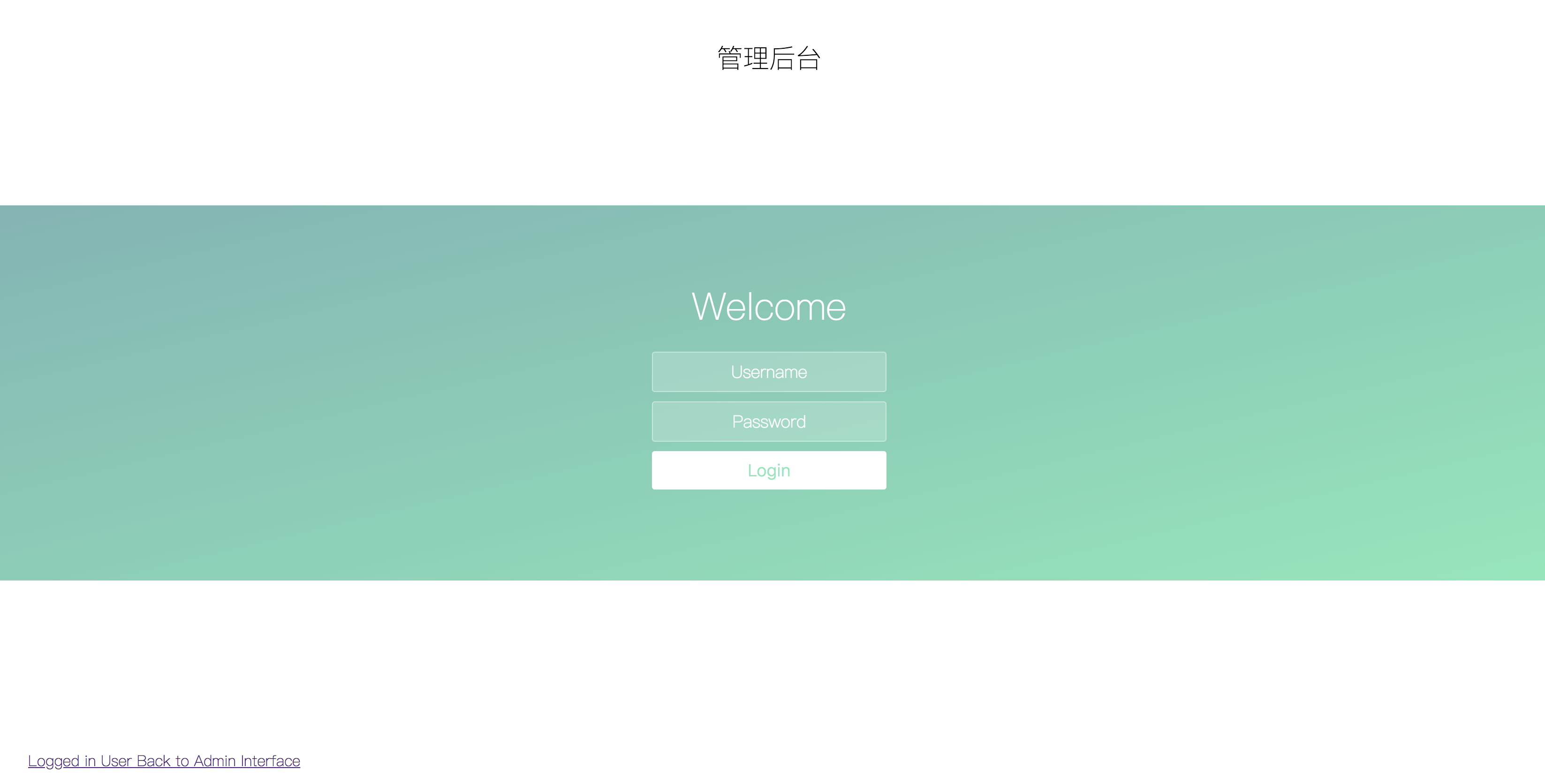
此时我们运用的是console\_email中的mail\_sender2，作为单独单词发送邮件的功能。

此时同样调用了apscheduler的add\_job来进行定时任务的调度和下达，然后写一个具体的执行方法，在此email\_manual直接把发送两封邮件的代码运行一遍(先跑一遍console\_xxx后跑console\_xxx\_brief的脚本)。

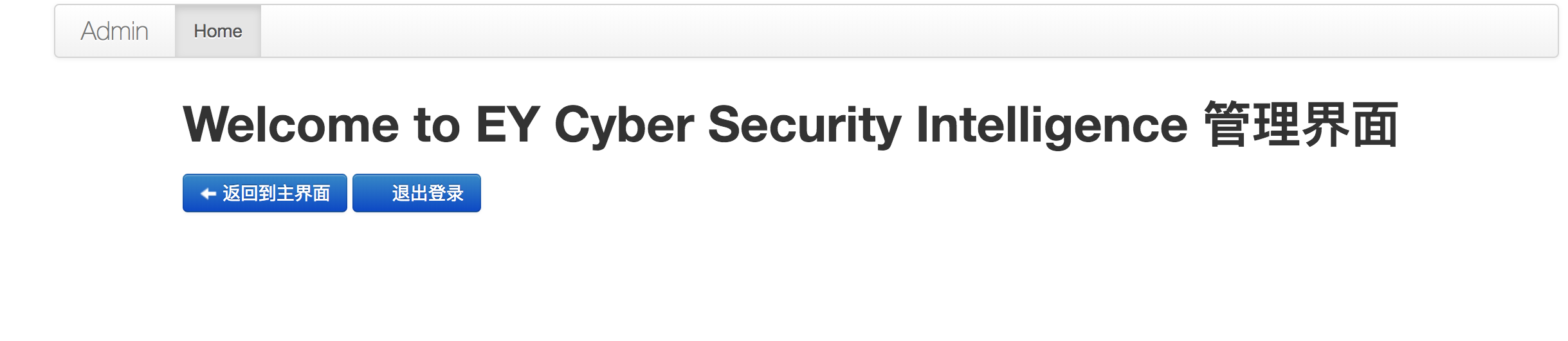
在发送email的过程中，有一个filename参数我们是后台直接写好，暂定我们会写'test\_vul\_word0705.html' 作为漏洞的报告内容，每次跑console\_vul和console\_vul\_brief后均会把存储在云端本地的html文件覆盖，更新，最后我们直接读取本地的'test\_vul\_word0705.html'，作为最新的报告内容。(在此我们是将内容不断地更新本地的这个'test\_vul\_word0705.html'文件的内容，通过脚本运行之后覆盖本地的文件，我们暂时通过这样的做法来获取当前邮件报告的内容)。行业是同样的道理，文件改为'test\_industry\_word0705\_3.html'。

**行为模式6– 根据管理界面进行管理员权限登录**

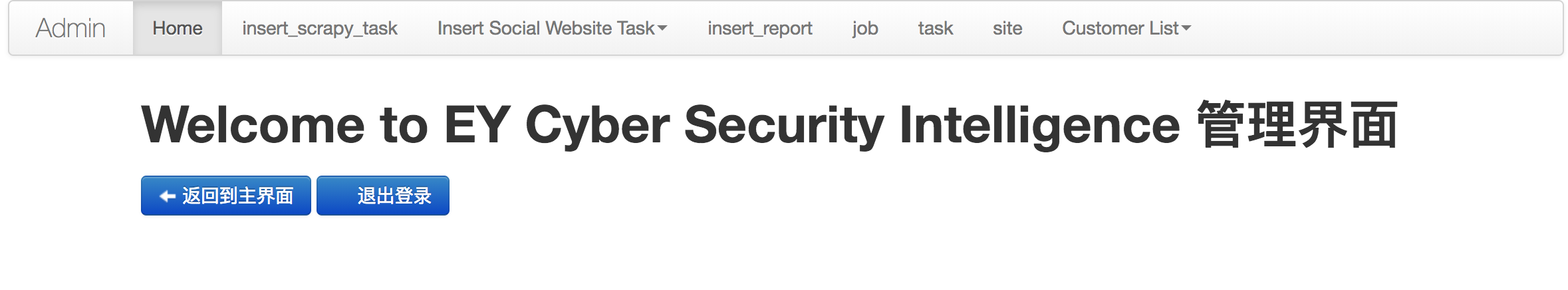
在flask admin所拓展的管理员界面中，我们初步的跳转是在login 界面，此时如果没有管理员进行登录，那么则需要输入账号密码的字段，进行登录。



若直接输入错误字段或者不输入，企图点击左下角的直接跳转admin界面，所有列表都被屏蔽。



若输入正确的管理员账号密码，那么mysql中的可视化界面则会直接呈现给管理员。



此时无论回到login主界面，或者是进行任何一个模块的操作，管理员的登录状态都会被记住，并且在login界面点击左下的链接即可直接回到admin界面，无需重输密码。只需在退出去前点击退出登录，该管理员的登录状态就会失效，模块将对当前状态的用户进行屏蔽。

**Flask Admin 的基本操作**

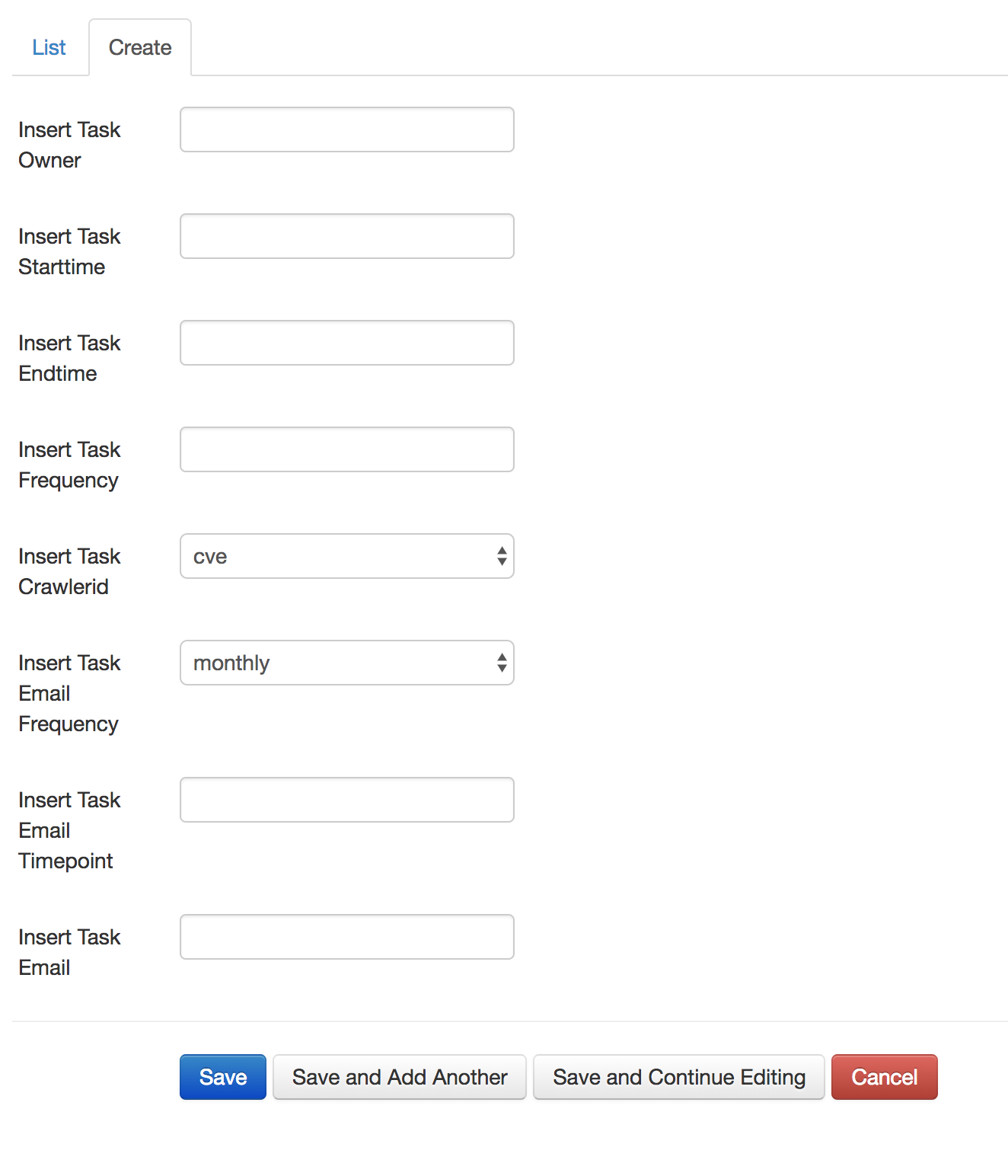
Flask admin 的基本操作是主要基于flask\_sqlalchemy对mysql中的数据表进行映射，并且通过listens\_for的听取机制对创建的数据进行代码逻辑的操作。并且有job 和task 以及site这些进行管理员参看查看信息的表，也有insert\_xxx\_task的表通过管理员创建任务直接下达任务到各节点。

基本上我们的模块除了job, task, site是供任务状态的跟踪监控，和对相关站点的视察。

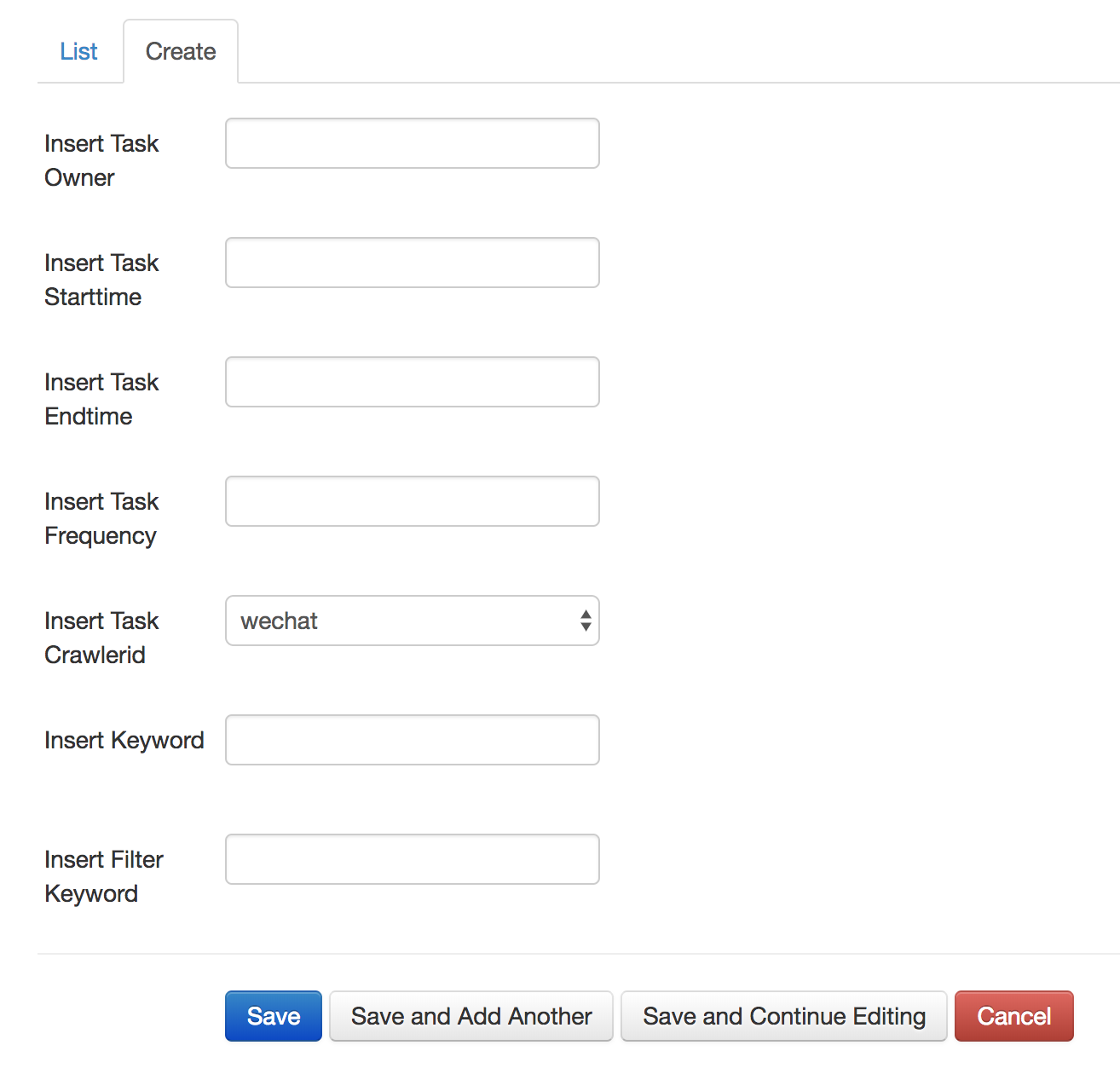
具有代码逻辑的模块分为以下几类：

**insert scrapy task:**

表达我们希望下达爬虫任务，在前文的行为模式中我们已经详细解释了从管理员界面下达爬虫任务的具体操作，以及对于输入字段的具体要求。



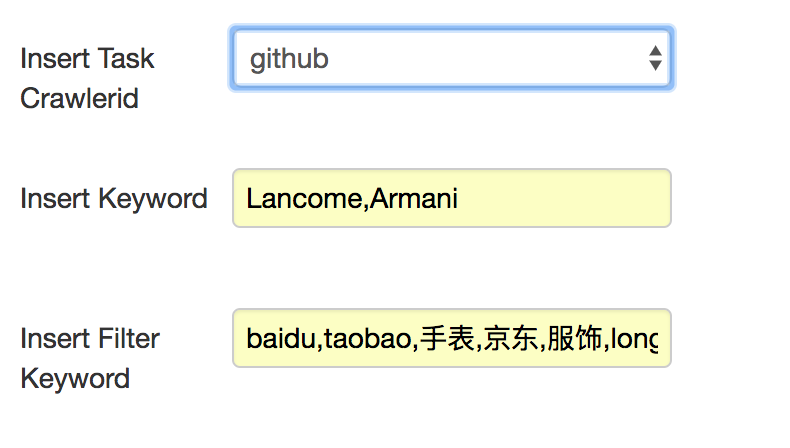
**insert social task:**



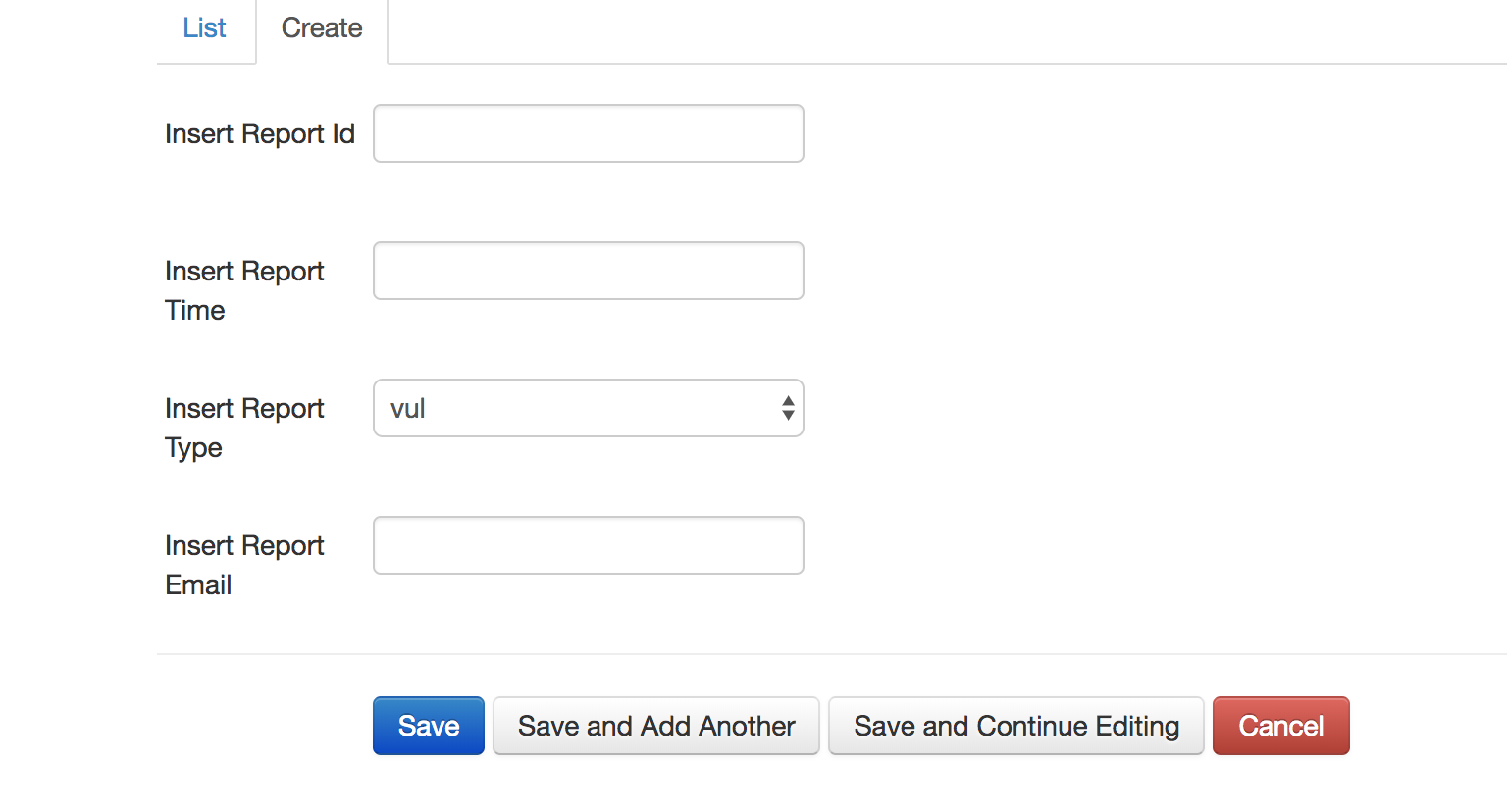
对于第二类的网站爬取的脚本我们另置了一个模块，暂时将wechat分为insert soaicl task和github自己分为insert github task 的模块。两个模块在create 中都会共用一样的输入框，但是在下达wechat 任务的时候需要点击wechat 的social 模块进行任务创建，此时对于wechat 任务而言，Insert filter keywords是没意义的那么可以直接无视，后台不会有相应的听取机制。其他字段则与insert task 一致，只需要管理员输入相应的决定性信息即可。

同理，若需要创建github 任务则需要在此点击上面的insert github task模块，根据具体要求输入要求的字段信息。

在此，keywords 的数量若大于一，统一用英文的逗号进行分开。

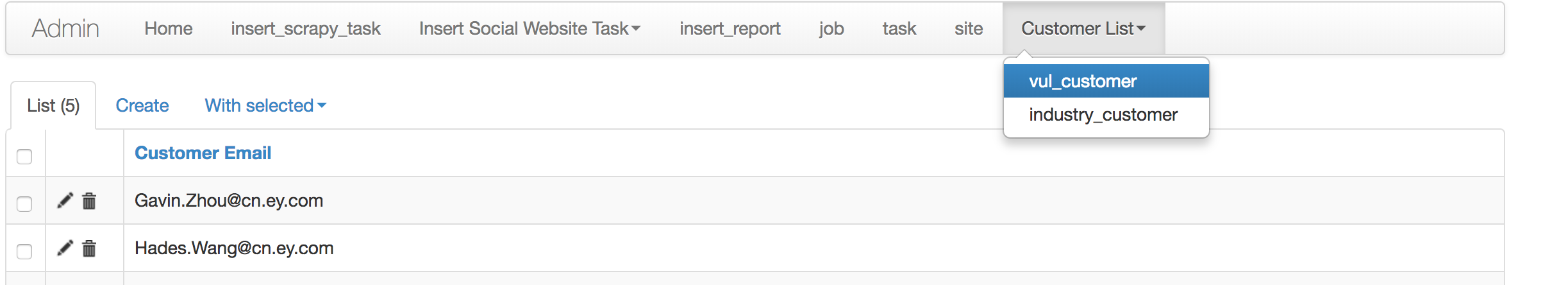


**insert report :**



该模块是为了单次直接发送报告给某个用户，并不会直接影响默认客户的清单，是一个一次性的操作。当前版本暂时会要求用户随机输入一个insert report id， 该id 与report id 并无直接关系，仅仅是为了数据库的插入数据所需的一个主键，后续开发人员若需要改善，可以将某两个列作为主键，重新创建表。

**Customer List:**



目前的两个默认客户清单中，对于发送email 任务的代码逻辑而言，一种报告对应一个清单，当发送该种报告的时候，那一类型的清单中的所有邮箱地址就会被作为收件人，因此，管理员可以在该表直接进行CRUD操作，或者在insert scrapy task 中进行用户的插入。

**Trivial Command**

Restful command line

在skylark开发初始阶段，为了调试对外的Restful API接口，我们会用linux command line 来进行手动curl 的操作，以模拟从外界直接发送Restful 请求调用接口，因此下列linux command line 展示了如何将一个task 的决定性信息放在一个HTTP 请求中从而达到调用API的目的。

######################################## standard Restful input ########################################

######################################## standard Restful input ########################################

Environment Variable Settings

同时我们的脚本中会有一些敏感机密信息，例如连接数据库，SMTP服务器，和OSS服务器的账号密码等字段，该类信息我们将一律存放在 ~/.bsahrc中进行环境变量的存储。基本的操作是

vim ~/.bashrc ## 进入脚本进行修改

进行环境变量的设置。

Wq退出

source ~/.bashrc ## source进行激活

No Hang Up Linux Command Line

nohup python sh\_main.py > myout.file 2>&1 &

## 运用nohup进行离线启动，只要当前路径在含有main的文件夹中。如此一来即使离开ssh的linux界面服务器依然能够在后台运行。

nohup python console\_main.py > myout.file 2>&1 &

nohup python hk\_main.py > myout.file 2>&1 &

## 将 python 进程杀死，从而停止后台运作。

kill -9 $(ps -aux | grep console\_main | awk '{ print $2 }')

kill -9 $(ps -aux | grep hk\_main | awk '{ print $2 }')

kill -9 $(ps -aux | grep sh\_main | awk '{ print $2 }')

**对于后续开发的建议与展望**

flask 作为一个微型框架虽然被称为micro framework， 但是他的拓展性和与数据库模版可视化的性能都是python 主流框架中最好的之一。如果服务器需要面对较高的并发量，则需要进一步进行gunicorn 或者celery+redis的异步特性，从而增强服务器的并发量。

以下是一些关键的reference网站以供参考：

<https://blog.csdn.net/huobanjishijian/article/details/51470898>

<https://blog.csdn.net/qiqiyingse/article/details/79487514>

<https://www.jianshu.com/p/69b3b8e39bf4>

<https://www.cnblogs.com/jessicaDuan/p/7827618.html>

<https://www.cnblogs.com/baby123/p/6477429.html>

<https://github.com/pyecharts/pyecharts-snapshot>

<https://blog.csdn.net/englishsname/article/details/51029767>

<https://apscheduler.readthedocs.io/en/latest/index.html>

若部署文档中有遗漏的package 未部署，点击 pip install xxx即可。