Python定时任务框架APScheduler（译）

Feb 19, 2016

§ 简介

`APScheduler` (以下简称APS)框架可以让用户定时执行或者周期性执行Python任务。既可以添加任务也可以删除任务，还可以将任务存储在数据库中。当APS重启之后，还会继续执行之前设置的任务。
APS是跨平台的，注意APS既不是守护进程也不是服务，更不是命令行程序。APS是进程内的调度器，也就是说它的实现原理是在进程内产生内置的阻塞来创建定时服务，以便在预定的时间内执行某个任务。
APS支持以下三种定时任务：

- `crontab` 类型任务
- 固定时间间隔任务
- 基于日期时间的一次性任务

你可以使用上面任意一种定时任务或者组合任务。
APS支持以下面几种方式保存：

- 内存
- SQLAlchemy
- MongoDB
- Redis

APS也可以整合进其他几个Python框架：

- asyncio
- gevent
- Tornado
- Twisted
- Qt

§ 安装

使用pip进行安装：

```
PYTHON

$ pip install apscheduler
```

如果你没有安装pip，可以通过下面链接进行安装get-pip.py
如果不能通过pip下载，也可以直接下载APS安装包，解压再进行安装

```
PYTHON
```

```
$ python setup.py install
```

§ 基础概念

APS由以下几部分组成：

- 触发器（triggers）
- 任务仓库（job stores）
- 执行器（executors）
- 调度器（schedulers）

触发器包含了所有定时任务逻辑，每个任务都有一个对应的触发器，触发器决定任何的何时执行，初始配置情况下，触发器是无状态的。

任务仓库保存要执行的任务，其中一个默认的任务仓库将任务保存在内存中，而另外几个任务仓库将任务保存在数据库中。在将任务保存到任务仓库前，会对任务执行序列化操作，当重新读取任务时，再执行反序列化操作。除了默认的任务仓库，其他任务仓库都不会在内存中保存任务，而是作为任务保存、加载、更新以及搜索的一个中间件。任务仓库在定时器之间不能共享。

执行器用来执行定时任务，它只是将要执行的任务放在新的线程或者线程池中运行。执行完毕之后，再通知定时器。

调度器将其它几个组件联系在一起，一般在应用中只有一个调度器，程序开发者不会直接操作触发器、任务仓库或执行器，相反，调度器提供了这个接口。任务仓库以及执行器的配置都是通过调度器来实现的。

§ 选择合适的调度器、执行器以及任务仓库

选择调度器是根据我们的开发环境与实际应用来决定的，下面是一些常用调度器：

- BlockingScheduler：适合于只在进程中运行单个任务的情况
- BackgroundScheduler: 适合于要求任何在程序后台运行的情况
- `AsyncIOScheduler`： 适合于使用asyncio框架的情况
- `GeventScheduler` : 适合于使用gevent框架的情况
- TornadoScheduler: 适合于使用Tornado框架的应用
- `TwistedScheduler` : 适合使用Twisted框架的应用
- `QtScheduler` : 适合使用QT的情况

而任务仓库，如果是非持久任务，使用 `MemoryStore` 就可以了，如果是持久性任务，那么久需要根据编程环境进行选择了。

大多数情况下，执行器选择 `ThreadPoolExecutor` 就可以了，但是如果涉及到比较耗CPU的任务，就可以选择 `ProcessPoolExecutor` ，以充分利用多核CPU。，当然也可以同时使用两个执行器。

可以在相应的API中找到对应的任务仓库以及执行器。

§ 配置任务调度器

APS提供了多种调度器配置方法，既可以使用配置字典，也可以直接传递配置参数给调度器，还可以先初始化调度器，添加完任务之后，最后再来配置调度器。
相关的配置参数可以参考API文档。
举个简单例子：

PYTHON

```
from apscheduler.schedulers.background import BackgroundScheduler
scheduler = BackgroundScheduler()
# Initialize the rest of the application here, or before the scheduler initialization
```

上面的代码生成一个后台调度器，使用默认名为 `default` 的 `MemoryJobStore` ，以及默认名为 `default` 的 `ThreadPoolExecutor` ，最大线程数为10。

现在假设你想使用两个任务仓库以及两个执行器，并且还想调整下任务的默认参数。可以使用下面三种方法，所完成的功能是一样的。

方法1：

PYTHON

```
from pytz import utc
from apscheduler.schedulers.background import BackgroundScheduler
from apscheduler.jobstores.mongodb import MongoDBJobStore
from apscheduler.jobstores.sqlalchemy import SQLAlchemyJobStore
from apscheduler.executors.pool import ThreadPoolExecutor, ProcessPoolExecutor
jobstores = {
    'mongo': MongoDBJobStore(),
    'default': SQLAlchemyJobStore(url='sqlite:///jobs.sqlite')
}
executors = {
    'default': ThreadPoolExecutor(20),
    'processpool': ProcessPoolExecutor(5)
}
job_defaults = {
    'coalesce': False,
    'max_instances': 3
}
scheduler = BackgroundScheduler(jobstores=jobstores, executors=executors, job_defaults=job_defaults, time
```

方法2：

PYTHON

```
from apscheduler.schedulers.background import BackgroundScheduler
# The "apscheduler." prefix is hard coded
scheduler = BackgroundScheduler({
    'apscheduler.jobstores.mongo': {
         'type': 'mongodb'
    },
    'apscheduler.jobstores.default': {
        'type': 'sqlalchemy',
        'url': 'sqlite:///jobs.sqlite'
    },
    'apscheduler.executors.default': {
        'class': 'apscheduler.executors.pool:ThreadPoolExecutor',
        'max_workers': '20'
    },
    'apscheduler.executors.processpool': {
        'type': 'processpool',
        'max_workers': '5'
    },
    'apscheduler.job_defaults.coalesce': 'false',
    'apscheduler.job_defaults.max_instances': '3',
```

```
    'apscheduler.timezone': 'UTC',
})
```

方法3：

PYTHON

```
from pytz import utc
from apscheduler.schedulers.background import BackgroundScheduler
from apscheduler.jobstores.sqlalchemy import SQLAlchemyJobStore
from apscheduler.executors.pool import ProcessPoolExecutor
jobstores = {
    'mongo': {'type': 'mongodb'},
    'default': SQLAlchemyJobStore(url='sqlite:///jobs.sqlite')
}
executors = {
    'default': {'type': 'threadpool', 'max_workers': 20},
    'processpool': ProcessPoolExecutor(max_workers=5)
}
job_defaults = {
    'coalesce': False,
    'max_instances': 3
}
scheduler = BackgroundScheduler()
# .. do something else here, maybe add jobs etc.
scheduler.configure(jobstores=jobstores, executors=executors, job_defaults=job_defaults, timezone=utc)
```

§ 启动调度器

使用 `start()` 方法启动调度器， `BlockingScheduler` 需要在初始化之后才能执行 `start()` ,对于其他的Scheduler，调用 `start()` 方法都会直接返回，然后可以继续执行后面的初始化操作。

调度器启动之后，就不能更改它的配置了。

§ 添加任务

有两种添加任务的办法：

- 调用 `add_job()`
- 使用 `scheduled_job()` 修饰器

第一个方法是使用最多的，因为调用它会返回一个 `apscheduler.job.Job` 实例，后续可以对它进行修改或者删除，而使用修饰器添加的任务添加之后就不能进行修改。

在调度器中设置定时任务，如果任务添加的时候，调度器还没有启动，那么任务只是暂时放到调度器中，当调度器启动之后重新计算第一次执行时间。

需要注意的是，可以使用执行器或者任务仓库来序列化任务，但是这个任务必须满足两个条件：

1. 调用对象必须全局可访问
2. 调用对象的参数必须也可以序列化。

在所有内置的任务仓库中，只有 `MemoryJobStore` 不能序列化对象；在所有内置的执行器中，只有 `ProcessPoolExecutors` 会序列化任务。

如果你需要在应用中使用持久性任务，那就必须给任务定义一个ID，并设置 `replace_existing=True` ,否则每次重启应用都会返回一个新的任务。

如果要立即执行任务，只需要在添加任务的时候省略 `trigger` 参数。

§ 删除任务

当从调度器中删除任务的时候，就会从相关联的任务仓库中删除任务，后面就不会再执行了，有两种方法来删除任务。

1. 调用 `remove_job()` ，参数为job ID以及任务仓库名
2. 调用 `remove()`

第二种方法用起来更加方便，但是需要先保存添加任务时返回的实例对象；而通过 `scheduled_job()` 添加的任务，只能使用第一种方法进行删除。

如果任务执行完毕，它会自动被删除。

例如：

```
PYTHON
job = scheduler.add_job(myfunc, 'interval', minutes=2)
job.remove()
```

同样的，如果显式使用任务ID，则使用下面的方法：

```
CODE
scheduler.add_job(myfunc, 'interval', minutes=2, id='my_job_id')
scheduler.remove_job('my_job_id')
```

§ 暂停与恢复任务

暂停与恢复任务也很简单，可以直接操作任务实例或者调度器来实现，当任务暂停时，它的运行时间会被重置，暂停期间不会计算进去，重启之后又算进去。

暂停任务可以使用以下两种方法：

- `apscheduler.job.Job.pause()`
- `apscheduler.schedulers.base.BaseScheduler.pause_job()`

恢复任务可以使用以下两种方法：

- `apscheduler.job.Job.resume()`
- `apscheduler.schedulers.BaseScheduler.resume_job()`

§ 获取任务列表

可以使用 `get_jobs()` 方法来获取当前正在处理的任务列表，如果只是想获取某个任务仓库中的任务列表，可以使用任务仓库名作为参数传入。

APS还提供了一个 `print_jobs()` 方法来打印格式化的任务列表。

§ 修改任务

使用 `apscheduler.job.Job.modify()` 或者 `modify_job()` 方法可以修改任务的属性，你可以修改任务的任意属性，除了 `id` 。

例如：

PYTHON

```
job.modify(max_instances=6, name='Alternate name')
```

如果你想重新调度某个任务，例如改变它的触发器，则可以使用 `apscheduler.job.Job.reschedule()` 或者 `reschedule_job()` 方法，这个两个方法都可以为任务重新创建一个触发器，并重新计算任务的运行时间。

例如：

PYTHON

```
scheduler.reschedule_job('my_job_id', trigger='cron', minute='*/5')
```

§ 关闭调度器

使用下面的方法关闭调度器：

PYTHON

```
scheduler.shutdown()
```

默认情况下，scheduler会关闭它的任务仓库以及执行器，并等待所有正在执行的任务执行完。如果你不想等待，可以这样操作：

PYTHON

```
scheduler.shutdown(wait=False)
```

§ 限制任务实例并发执行的个数

默认情况下，只允许同时执行一个任务实例，通过 `max_instances` 参数来设置允许并发执行任务的个数。

§ 错失任务的合并

有时候调度器可能无法按时执行某个任务，最常见的情况就是当某个持久性任务保存在任务仓库中时，调度器关闭之后再重启，但是任务需要在重启之前被执行，这时这个任务就被错过了。调度器会根据任务 `misfire_grace_time` 参数的配置来决定错过的任务还要不要继续执行。这样做的话可能导致重启调度器之后会连续执行好几个任务。

§ 调度器事件

可以给调度器添加事件监听器，调度器事件只有在某些情况下才会被触发，并且可以携带某些有用的信息。通过给 `add_listener()` 传递合适的 `mask` 参数，可以只监听几种特定的事件类型。

例如：

```python
def my_listener(event):
    if event.exception:
        print('The job crashed :(')
    else:
        print('The job worked :)')
scheduler.add_listener(my_listener, EVENT_JOB_EXECUTED | EVENT_JOB_ERROR)
```

#python

Prev Next