枫飞飞

Python 定时任务的实现方式

本文转载自:

https://lz5z.com/Python%E5%AE%9A%E6%97%B6%E4%BB%BB%E5%8A%A1%E7%9A%84%E5%AE%9E%E7%8E%B0%E6%96%B9%E5%BC%8F/

背景

目前所在的项目组需要经常执行一些定时任务，于是选择使用 Python 的定时器。

Python 实现定时任务

循环 sleep

这种方式最简单，在循环里面放入要执行的任务，然后 sleep 一段时间再执行

```
from datetime import datetime
import time
# 每n秒执行一次
def timer(n):
    while True:
        print(datetime.now().strftime("%Y-%m-%d %H:%M:%S"))
        time.sleep(n)
# 5s
timer(5)
```

这个方法的缺点是，只能执行固定间隔时间的任务，如果有定时任务就无法完成，比如早上六点半喊我起床。并且 sleep 是一个阻塞函数，也就是说 sleep 这一段时间，啥都不能做。

threading模块中的Timer

threading 模块中的 Timer 是一个非阻塞函数，比 sleep 稍好一点，不过依然无法喊我起床。

```
from datetime import datetime
from threading import Timer
# 打印时间函数
def printTime(inc):
    print(datetime.now().strftime("%Y-%m-%d %H:%M:%S"))
    t = Timer(inc, printTime, (inc,))
    t.start()
# 5s
printTime(5)
```

Timer 函数第一个参数是时间间隔（单位是秒），第二个参数是要调用的函数名，第三个参数是调用函数的参数(tuple)

使用sched模块

sched 模块是 Python 内置的模块，它是一个调度（延时处理机制），每次想要定时执行某任务都必须写入一个调度。

```
import sched
import time
from datetime import datetime
# 初始化sched模块的 scheduler 类
# 第一个参数是一个可以返回时间戳的函数，第二个参数可以在定时未到达之前阻塞。
```

```
schedule = sched.scheduler(time.time, time.sleep)
# 被周期性调度触发的函数
def printTime(inc):
    print(datetime.now().strftime("%Y-%m-%d %H:%M:%S"))
    schedule.enter(inc, 0, printTime, (inc,))
# 默认参数60s
def main(inc=60):
    # enter四个参数分别为：间隔事件、优先级（用于同时间到达的两个事件同时执行时定序）、被调用触发的函数，
    # 给该触发函数的参数（tuple形式）
    schedule.enter(0, 0, printTime, (inc,))
    schedule.run()
# 10s 输出一次
main(10)
```

sched 使用步骤如下：

（1）生成调度器：
s = sched.scheduler(time.time,time.sleep)
第一个参数是一个可以返回时间戳的函数，第二个参数可以在定时未到达之前阻塞。

（2）加入调度事件
其实有 enter、enterabs 等等，我们以 enter 为例子。
s.enter(x1,x2,x3,x4)
四个参数分别为：间隔事件、优先级（用于同时间到达的两个事件同时执行时定序）、被调用触发的函数，给触发函数的参数（注意：一定要以 tuple 给，如果只有一个参数就(xx,)）

（3）运行
s.run()
注意 sched 模块不是循环的，一次调度被执行后就 Over 了，如果想再执行，请再次 enter

APScheduler定时框架

终于找到了可以每天定时喊我起床的方式了

APScheduler是一个 Python 定时任务框架，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、并以 daemon 方式运行应用。

使用 APScheduler 需要安装

```
$ pip install apscheduler
```

首先来看一个周一到周五每天早上6点半喊我起床的例子

```
from apscheduler.schedulers.blocking import BlockingScheduler
from datetime import datetime
# 输出时间
def job():
    print(datetime.now().strftime("%Y-%m-%d %H:%M:%S"))
# BlockingScheduler
scheduler = BlockingScheduler()
scheduler.add_job(job, 'cron', day_of_week='1-5', hour=6, minute=30)
scheduler.start()
```

代码中的 BlockingScheduler 是什么呢?

BlockingScheduler是APScheduler中的调度器，APScheduler 中有两种常用的调度器，BlockingScheduler 和 BackgroundScheduler，当调度器是应用中唯一要运行的任务时，使用 BlockingSchedule，如果希望调度器在后台执行，使用 BackgroundScheduler。

1. BlockingScheduler: use when the scheduler is the only thing running in your process

2018年5月(38)
2018年4月(56)
2018年3月(16)
2018年2月(8)
2018年1月(37)
2017年12月(15)

最新评论

1. Re:进程和线程的区别

图片都挂了，看不见了

--ponxiaoming

2. Re:MYSQL的安全模式：sql safe updates介绍

按照楼主的贴子修改了模式

--今天你思考了嘛

3. Re:深入解析Mysql 主从同步延迟原理及解决方案

、、、测试格式

--李磊（leige）

4. Re:【面试】MyISAM 和 INNODB的区别是什么?

“MyISAM内存空间使用率比InnoDB低”怎么理解?

--我在这儿

5. Re:Nginx安装及配置详解

分享一个性能媲美nginx的异步IO事件库libhv

--ithewei

阅读排行榜

1. Nginx安装及配置详解(102932)
2. sql中把字符串转化为数字的方法(87981)
3. Ubuntu终端命令--查看端口占用及关闭(36536)
4. Python 定时任务的实现方式(35233)
5. js的序列化和反序列化(31656)

评论排行榜

1. Nginx安装及配置详解(5)
2. 进程和线程的区别(2)
3. MYSQL的安全模式：sql safe updates介绍(2)
4. 【面试】MyISAM 和 INNODB的区别是什么?(1)
5. 深入解析Mysql 主从同步延迟原理及解决方案(1)

推荐排行榜

1. Nginx安装及配置详解(19)
2. Python程序 #!/usr/bin/python 的解释(4)
3. Python 定时任务的实现方式(3)
4. js的序列化和反序列化(3)
5. numpy中np.nan(pandas中NAN)(3)

2. BackgroundScheduler: use when you're not using any of the frameworks below, and want the scheduler to run in the background inside your application
3. AsyncIOScheduler: use if your application uses the asyncio module
4. GeventScheduler: use if your application uses gevent
5. TornadoScheduler: use if you're building a Tornado application
6. TwistedScheduler: use if you're building a Twisted application
7. QtScheduler: use if you're building a Qt application

APScheduler四个组件

APScheduler 四个组件分别为：触发器(trigger)，作业存储(job store)，执行器(executor)，调度器(scheduler)。

触发器(trigger)

包含调度逻辑，每一个作业有它自己的触发器，用于决定接下来哪一个作业会运行。除了他们自己初始配置意外，触发器完全是无状态的
APScheduler 有三种内建的 trigger:

```
date: 特定的时间点触发
interval: 固定时间间隔触发
cron: 在特定时间周期性地触发
```

作业存储(job store)

存储被调度的作业，默认的作业存储是简单地把作业保存在内存中，其他的作业存储是将作业保存在数据库中。一个作业的数据讲在保存在持久化作业存储时被序列化，并在加载时被反序列化。调度器不能分享同一个作业存储。
APScheduler 默认使用 MemoryJobStore，可以修改使用 DB 存储方案

执行器(executor)

处理作业的运行，他们通常通过在作业中提交制定的可调用对象到一个线程或者进城池来进行。当作业完成时，执行器将会通知调度器。
最常用的 executor 有两种：

```
ProcessPoolExecutor
ThreadPoolExecutor
```

调度器(scheduler)

通常在应用中只有一个调度器，应用的开发者通常不会直接处理作业存储、调度器和触发器，相反，调度器提供了处理这些的合适的接口。配置作业存储和执行器可以在调度器中完成，例如添加、修改和移除作业。

配置调度器

APScheduler提供了许多不同的方式来配置调度器，你可以使用一个配置字典或者作为参数关键字的方式传入。你也可以先创建调度器，再配置和添加作业，这样你可以在不同的环境中得到更大的灵活性。

下面来看一个简单的 BlockingScheduler 例子

```
1  from apscheduler.schedulers.blocking import BlockingScheduler
2  from datetime import datetime
3
4  def job():
5      print(datetime.now().strftime("%Y-%m-%d %H:%M:%S"))
6  # 定义BlockingScheduler
```

```
sched = BlockingScheduler()
sched.add_job(job, 'interval', seconds=5)
sched.start()
```

上述代码创建了一个 BlockingScheduler，并使用默认内存存储和默认执行器。(默认选项分别是 MemoryJobStore 和 ThreadPoolExecutor，其中线程池的最大线程数为10)。配置完成后使用 start() 方法来启动。

如果想要显式设置 job store(使用mongo存储)和 executor 可以这样写：

```
from datetime import datetime
from pymongo import MongoClient
from apscheduler.schedulers.blocking import BlockingScheduler
from apscheduler.jobstores.memory import MemoryJobStore
from apscheduler.jobstores.mongodb import MongoDBJobStore
from apscheduler.executors.pool import ThreadPoolExecutor, ProcessPoolExecutor
# MongoDB 参数
host = '127.0.0.1'
port = 27017
client = MongoClient(host, port)
# 输出时间
def job():
    print(datetime.now().strftime("%Y-%m-%d %H:%M:%S"))
# 存储方式
jobstores = {
    'mongo': MongoDBJobStore(collection='job', database='test', client=client),
    'default': MemoryJobStore()
}
executors = {
    'default': ThreadPoolExecutor(10),
    'processpool': ProcessPoolExecutor(3)
}
job_defaults = {
    'coalesce': False,
    'max_instances': 3
}
scheduler = BlockingScheduler(jobstores=jobstores, executors=executors, job_defaults=job_defaults)
scheduler.add_job(job, 'interval', seconds=5, jobstore='mongo')
scheduler.start()
```

在运行程序5秒后，第一次输出时间。
在 MongoDB 中可以看到 job 的状态

对 job 的操作

添加 job

添加job有两种方式：

1. add_job()
2. scheduled_job()

第二种方法只适用于应用运行期间不会改变的 job，而第一种方法返回一个 apscheduler.job.Job 的实例，可以用来改变或者移除 job。

```
from apscheduler.schedulers.blocking import BlockingScheduler
sched = BlockingScheduler()
# 装饰器
@sched.scheduled_job('interval', id='my_job_id', seconds=5)
def job_function():
    print("Hello World")
# 开始
sched.start()
```

@sched.scheduled_job() 是 Python 的装饰器。

移除 job

移除 job 也有两种方法：

1. remove_job()
2. job.remove()

remove_job 使用 jobID 移除
job.remove() 使用 add_job() 返回的实例

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

暂停和恢复 job

暂停一个 job：

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

恢复一个 job：

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

希望你还记得 apscheduler.job.Job 是 add_job() 返回的实例

获取 job 列表

获得可调度 job 列表，可以使用get_jobs() 来完成，它会返回所有的 job 实例。

也可以使用print_jobs() 来输出所有格式化的 job 列表。

修改 job

除了 jobID 之外 job 的所有属性都可以修改，使用
apscheduler.job.Job.modify() 或者 modify_job() 修改一个 job 的属性

```
job.modify(max_instances=6, name='Alternate name')
modify_job('my_job_id', trigger='cron', minute='*/5')
```

关闭 job

默认情况下调度器会等待所有的 job 完成后，关闭所有的调度器和作业存储。将 wait 选项设置为 False 可以立即关闭。

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

scheduler 事件

scheduler 可以添加事件监听器，并在特殊的时间触发。

```
def my_listener(event):
    if event.exception:
        print('The job crashed :(')
    else:
        print('The job worked :)')
# 添加监听器
scheduler.add_listener(my_listener, EVENT_JOB_EXECUTED | EVENT_JOB_ERROR)
```

trigger 规则

date

最基本的一种调度，作业只会执行一次。它的参数如下：

- run_date (datetime|str) – the date/time to run the job at
- timezone (datetime.tzinfo|str) – time zone for run_date if it doesn't have one already

```
from datetime import date
from apscheduler.schedulers.blocking import BlockingScheduler
sched = BlockingScheduler()
def my_job(text):
    print(text)
# The job will be executed on November 6th, 2009
sched.add_job(my_job, 'date', run_date=date(2009, 11, 6), args=['text'])
sched.add_job(my_job, 'date', run_date=datetime(2009, 11, 6, 16, 30, 5), args=['text'])
sched.add_job(my_job, 'date', run_date='2009-11-06 16:30:05', args=['text'])
# The 'date' trigger and datetime.now() as run_date are implicit
sched.add_job(my_job, args=['text'])
sched.start()
```

cron

- 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)

表达式：

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

```
from apscheduler.schedulers.blocking import BlockingScheduler

def job_function():
    print("Hello World")
# BlockingScheduler
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
sched.add_job(job_function, 'cron', month='6-8,11-12', day='3rd fri', hour='0-3')
# 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')
sched.start()
```

interval

参数:

- weeks (int) – number of weeks to wait
- days (int) – number of days to wait
- hours (int) – number of hours to wait
- minutes (int) – number of minutes to wait
- seconds (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

```
from datetime import datetime
from apscheduler.schedulers.blocking import BlockingScheduler

def job_function():
    print("Hello World")
# BlockingScheduler
sched = BlockingScheduler()
# Schedule job_function to be called every two hours
sched.add_job(job_function, 'interval', hours=2)
# 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')
sched.start()
```


枫飞飞
关注 - 8
粉丝 - 72
+加关注

3 0

« 上一篇: 常见数据结构的查找、插入、删除时间复杂度
» 下一篇: redis持久化（persistence）

posted on 2019-06-12 17:27 枫飞飞 阅读(35233) 评论(0) 编辑 收藏