17 开发实战：基于最小堆设计任务调度系统-慕课专栏

imooc.com/read/76/article/1913

堆 (*heap*)是一种特殊的树结构，能够快速定位 **最大值** 或 **最小值** ，是实现 **堆排序** 和 **优先队列** 的关键。优先队列主要应用在 **事件处理** 和 **任务调度** 等场景。接下来，我们以 **任务调度** 为例，抛砖引玉。

Python中的堆

Python 标准库内置了 **优先队列** 实现，这就是 *heapq* 模块。我们知道堆是一种满二叉树，可以保存于数组中；而 *list* 对象就是一种典型的动态数组结构！因此， *heapq* 将堆维护于 *list* 对象中，而不是提供一种新容器对象。相反，*heapq* 提供了几个关键操作函数，可直接操作 *list* 对象：

- *heapify* ，将 *list* 对象转化成堆(调整元素顺序以满足堆性质)；
- *heappush* ，将新元素压入堆中；
- *heappop* ，弹出堆顶元素；
- *etc*

创建一个列表对象并将其作为一个堆来使用：

```
heap = []
```

往堆中压入新元素，被压入元素对象必须 **可比较** ，自定义类需要实现 *__lt__* 等比较方法：

```
heappush(heap, item)
```

heapq 将 *list* 对象维护成 **最小堆** ，因此 **堆顶** (树的 **根节点**)即为最小值：

```
smallest = top = heap[0]
```

当然了，我们也可以将最小值从堆中弹出：

```
item = heappop(heap)
```

古典多线程调度

假设我们接到一个需求——设计定时任务执行系统。定时任务由 *JobItem* 类抽象，*executing_ts* 是任务执行时间：

```
class JobItem:

    def __init__(self, executing_ts, job):
        self.executing_ts = executing_ts
        self.job = job
```

初学者可能会想到最简单的多线程方案。系统需要同时处理多个定时任务，每个任务由一个线程来执行不就好了吗？这就是古典多线程模型，实例代码如下：

```
import time
from threading import Thread

def job_worker(job_item):

    time.sleep(job_item.executing_ts - time.time())


    process(job_item.job)

def add_job(job_item):

    Thread(target=job_worker, args=(job_item,)).start()
```

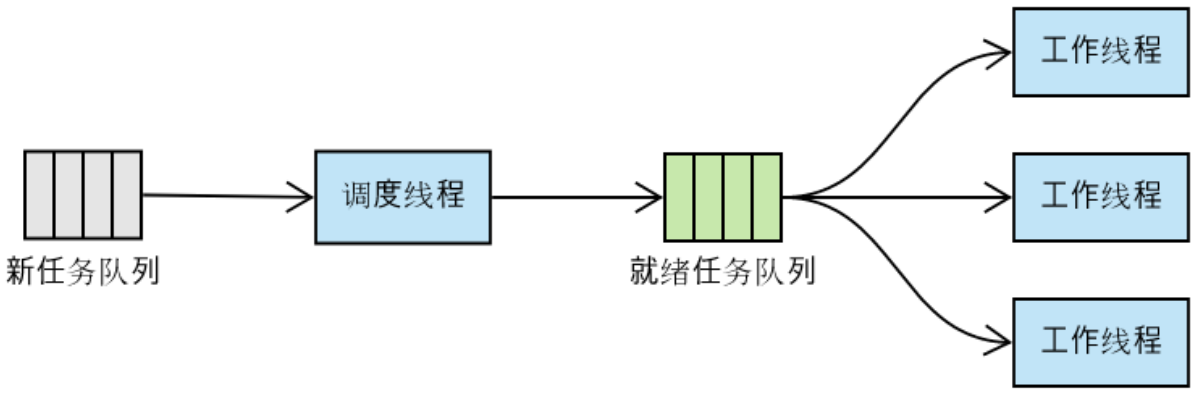
- *job_worker* ，工作线程执行函数，线程先睡眠等待执行时间到达(*6* 行)，然后调用 *process* 来执行(*8* 行)；
- *add_job* ，添加新定时任务时，启动一个新线程来处理；

这个方案虽然很简洁，但也很鸡肋。一方面，创建、销毁线程的开销很大；另一方面，由于线程需要占用不少资源，而系统能够支持的最大线程数相对有限。假设现在有成千上万的定时任务等待执行，系统能撑得住吗？

调度线程引入

采用多线程方案时，需要合理控制工作线程的 **个数** 。我们可以将执行时间已到达的任务放进一个 **就绪任务队列** ，然后启动若干个工作线程来执行就绪任务。新任务执行时间不定，可能有的是一分钟后执行，有的是一天后才执行。那么，问题就转变成——如何判断任务是否就绪？

这时，我们可以用另一个线程—— **调度线程** 来完成这个使命。调度线程不断接收新任务，并在任务到期时将其添加至就绪任务队列。如果我们用另一个队列来保存新任务，那么调度线程便是两个队列间的 **任务搬运工** ：



- **新任务队列** ，保存新任务，任务创建后即添加到这个队列；

- **就绪任务队列**，保存执行时间已到达的任务；
- **调度线程**，订阅 **新任务队列**，当任务时间到达时将其添加至 **就绪任务队列**（**搬运工**）；
- **工作线程**，从 **就绪任务队** 列取出任务并执行（**消费者**）；

借助 *queue* 模块，实现方案中的队列只需两行代码：

```
from queue import Queue


new_jobs = Queue()


ready_jobs = Queue()
```

这样一来，添加新任务时，只需将 *JobItem* 放入 **新任务队列** 即可：

```
def add_job(job_item):
    new_jobs.put(job_item)
```

工作线程执行逻辑也很简单，一个永久循环便搞定了：

```
def job_worker():
    while True:

        job_item = ready_jobs.get()


        process(job_item.job)
```

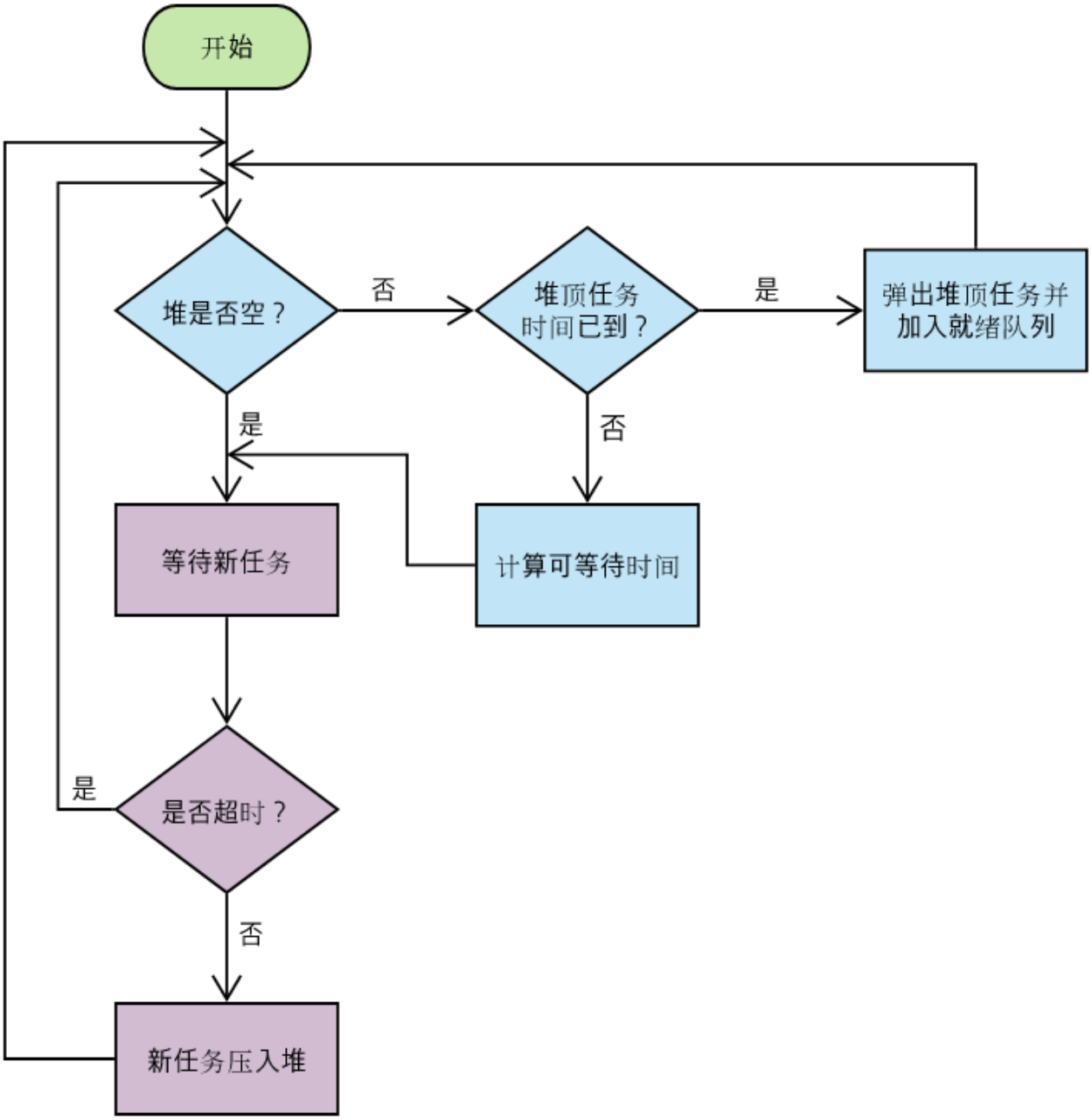
开始划重点了—— **调度线程** 的实现！

由于就绪任务一定是所有任务中执行时间最小的，因此可以用一个 **最小堆** 来维护任务集。我们希望任务按执行时间排序，因此需要为 *JobItem* 编写相关比较方法：

```
from functools import total_ordering

@total_ordering
class JobItem:

    def __init__(self, executing_ts, job):
        self.executing_ts = executing_ts
        self.job = job


    def __eq__(self, other):
        return self.executing_ts == other.executing_ts

    def __lt__(self, other):
        return self.executing_ts < other.executing_ts
```

注意到，我们只实现了 *__eq__* 和 *__lt__* 魔术方法，*__gt__* 等其他比较方法均由 *total_ordering* 装饰器代劳。

调度线程只需从新任务队列中取任务并压入最小堆，与此同时检查堆顶任务执行时间是否到达。由于线程需要同时处理两件不同的事情，初学者可能要慌了。不打紧，我们先画一个流程图梳理一下执行逻辑：



线程主体逻辑是一个永久循环，每次循环时：

1. 先检查堆顶任务，如果执行时间已到，则移到就绪任务队列并进入下次循环；
2. 等待新任务队列，如有新任务到达，则压入堆中并进入下次循环；
3. 特别注意，等待新任务时不能永久阻塞，需要根据当前堆顶任务计算等待时间；
4. 等待超时便进入下次循环再次检查堆顶任务，因此堆中任务不会被耽搁；

理清执行逻辑后，调度线程实现便没有任何难度了，代码如下：

```
from heapq import heappush, heappop

def job_dispatcher():

    heap = []

    while True:

        wait_time = 1


        if heap:

            job_item = heap[0]


            now = time.time()
            executing_ts = job_item.executing_ts
            if now >= executing_ts:

                heappop(heap)
                ready_jobs.put(job_item)


                continue
            else:

                wait_time = executing_ts - now

        try:



            job_item = new_queue.get(timeout=wait_time)



            heappush(heap, job_item)
        except:
            pass
```

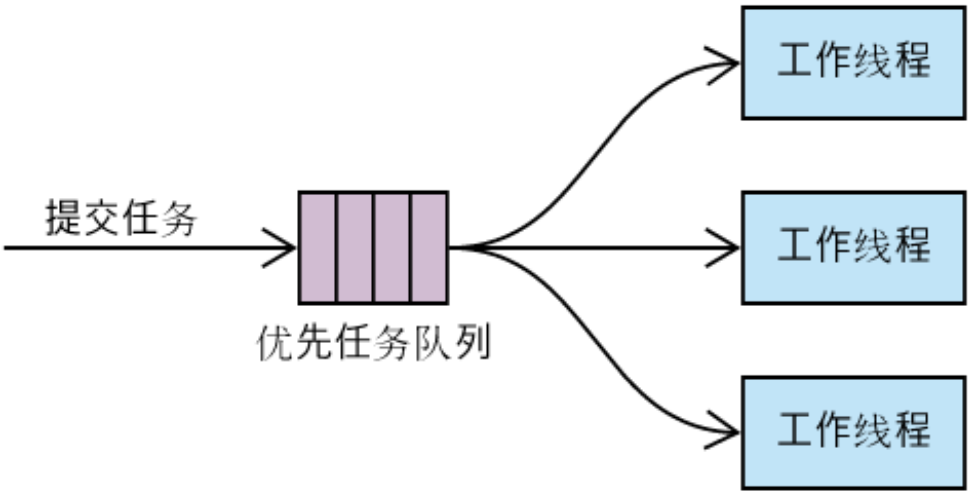
请结合代码注释并对照流程图阅读，不再重复讲解。

PriorityQueue

用过 *PriorityQueue* 的同学可能会提出这样的解决方案：

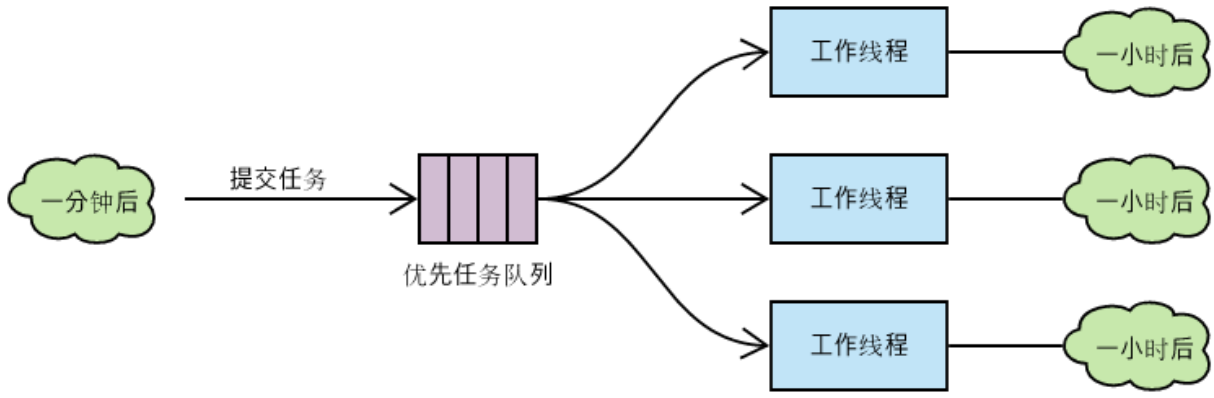
```
import time
from queue import PriorityQueue

jobs = PriorityQueue()

def job_worker():
    while True:

        job_item = jobs.get()


        now = time.time()
        executing_ts = job_item.executing_ts
        if executing_ts > now:
            time.sleep(executing_ts - now)


        process(job_item.job)

def add_job(job_item):
    jobs.put(job_item)
```

PriorityQueue 是 *Python* 标准库提供的优先队列，位于 *queue* 模块，接口与 *Queue* 一致。*PriorityQueue* 底层基于 *heapq* 实现 *get* 优先返回 **最小值** 。此外，*PriorityQueue* 是线程安全的，因此可以在多线程环境中使用：



这个多线程模型很简洁，但这个场景中存在一个 **致命问题** 。

假设我们先往队列提交 *3* 个 *1* 小时后执行的定时任务，这 *3* 个任务均被工作线程接收并处理。由于任务时间未到，工作线程先睡眠等待，这似乎问题不大。试想这时再提交一个 *1* 分钟后执行的定时任务会怎样？



由于工作线程正在睡眠，新任务需要等到工作线程唤醒，也就是一小时后才有机会被执行！

优先队列设计思路揭秘

当然了，我们可以对 *PriorityQueue* 进行一定的改造，使得 *get* 方法阻塞到堆顶任务到期才返回。想要改造 *PriorityQueue* ，必须知道它的实现方式。只要有源码，一切都好说：

```
>>> import queue
>>> print(queue.__file__)
/usr/local/Cellar/python/3.7.3/Frameworks/Python.framework/Versions/3.7/lib/python3.7/queu
e.py
```

这里不打算花大量篇幅介绍 *PriorityQueue* 的源码，鼓励亲们自行研究，必有收获。现在，我们再造一个轮子——从零开始设计自己 *PriorityQueue* 类，以此领会优先队列的设计思路，并探索新解决方案。

首先，我们模仿 *Queue* 实现优先队列类 *PriorityQueue* ，将 *get* 、 *put* 方法代理到 *heappush* 和 *heappop* ：

```
from heapq import heappush, heappop

class PriorityQueue:

    def __init__(self):
        self.queue = []

    def get(self):
        if self.queue:
            return heappop(self.queue)

    def put(self, item):
        heappush(self.queue, item)
```

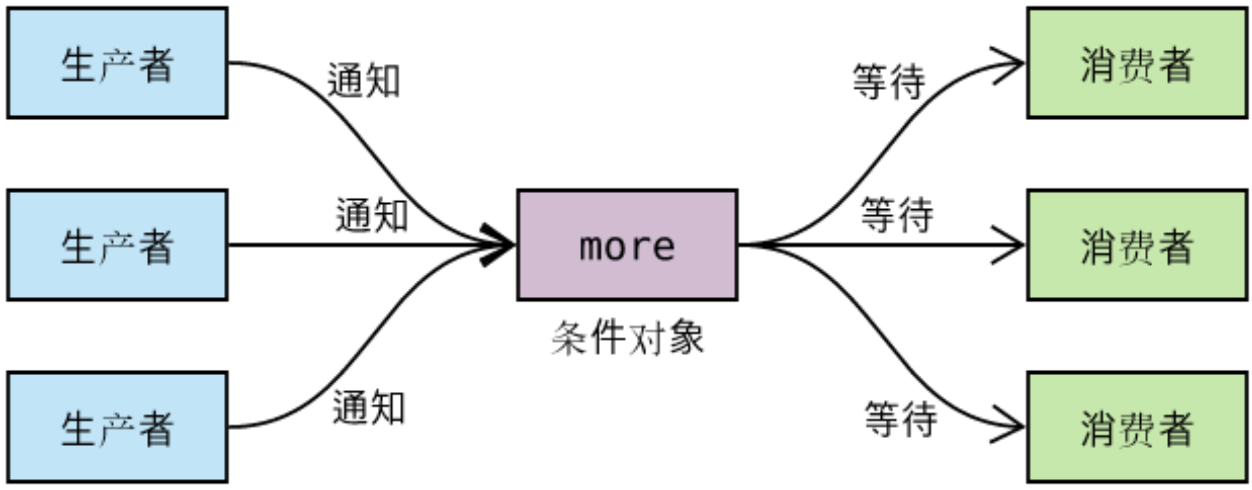
由于 *PriorityQueue* 可能同时被多个线程访问，因此必须考虑 **线程安全性** 。由于 *heappush* 和 *heappop* 函数并不是线程安全的，需要给这 *self.queue* 加上一一个 **互斥锁** 。因此，我们引入 *threading* 模块的 *Lock* 对象：

```
from  threading import Lock
from heapq import heappush, heappop

class PriorityQueue:

    def __init__(self):
        self.queue = []
        self.mutex = Lock()

    def get(self):
        with self.mutex:
            if self.queue:
                return heappop(self.queue)

    def put(self, item):
        with self.mutex:
            heappush(self.queue, item)
```

现在多线程导致的 **竞争态** 已经消除了，但还存在另一个问题：当队列为空时，消费者线程取不到任务。由于消费者并不知道队列何时有新任务，因此只能不断轮询，浪费大量 *CPU* 时间：

```
while True:
    result = queue.get()
    if result is None:
        continue
```

如果资源不可用时，消费者线程能够先睡眠，等生产者线程准备好资源后再唤醒就完美了！事实上，操作系统有这样的 **线程同步** 工具，这就是 **条件变量** 。*Python* 将条件变量封装为 *Condition* 对象，位于 *threading* 模块内。借助 *Condition* 对象，可以轻松实现线程等待和唤醒：



- 条件对象与资源保护锁配合使用；
- 资源未准备好，消费者调用 *wait* 方法等待生产者，*wait* 释放互斥锁；
- 生产者准备好资源，调用 *notify* 方法通知消费者；
- 消费者被唤醒，*wait* 方法重新拿到互斥锁并返回，消费者再次检查资源状态；

引入 *Condition* 后的代码如下，请结合注释阅读，不再赘述：

```
class PriorityQueue:

    def __init__(self):
        self.queue = []

        self.mutex = Lock()

        self.more = Condition(self.mutex)

    def get(self):

        with self.mutex:
            while True:
                if not self.queue:


                    self.more.wait()




                    continue

                return heappop(self.queue)

    def put(self, item):
        with self.mutex:

            heappush(self.queue, item)

            self.more.notify()
```

至此，我们已经山寨了一个 *PriorityQueue* ，麻雀虽小五脏俱全！

接下来，我们必须改造 *PriorityQueue* ，保证任务只有到期后才能弹出。因此，*get* 方法需要检查堆顶任务，未到期则等待。等待期间可能有执行时间更早的任务提交，这时线程必须停止等待重新检查堆顶。

```
class PriorityQueue:

    def __init__(self):
        self.queue = []

        self.mutex = Lock()

        self.more = Condition(self.mutex)

    def get(self):

        with self.mutex:
            while True:
                if not self.queue:


                    self.more.wait()




                    continue


                job_item = self.queue[0]


                now = time.time()
                executing_ts = job_item.executing_ts
                if executing_ts > now:

                    self.more.wait(executing_ts - now)

                    continue


                heappop(self.queue)
                return job_item

    def put(self, job_item):
        with self.mutex:

            heappush(self.queue, job_item)

            self.more.notify()
```

至此，我们完全实现了先前的设想！学习源码后，一路开挂，没有什么是不可能的！

最后，谈一谈重复造轮子的问题。

在实际工程项目中，最好不要重复造轮子。我们可以站在巨人的肩膀上，调研并选用业内成熟的解决方案，保证项目的 稳定性 、 可维护性 以及 开发效率 。

在学习的过程中，鼓励重复造轮子。我们不应该被类库蒙蔽双眼，沦为 *API* 调用侠。而想要了解一个类库的秘密，最佳途径是模仿着自己实现一个。带着好奇心，保持学习的心态，不断模仿，便总有超越的一天。