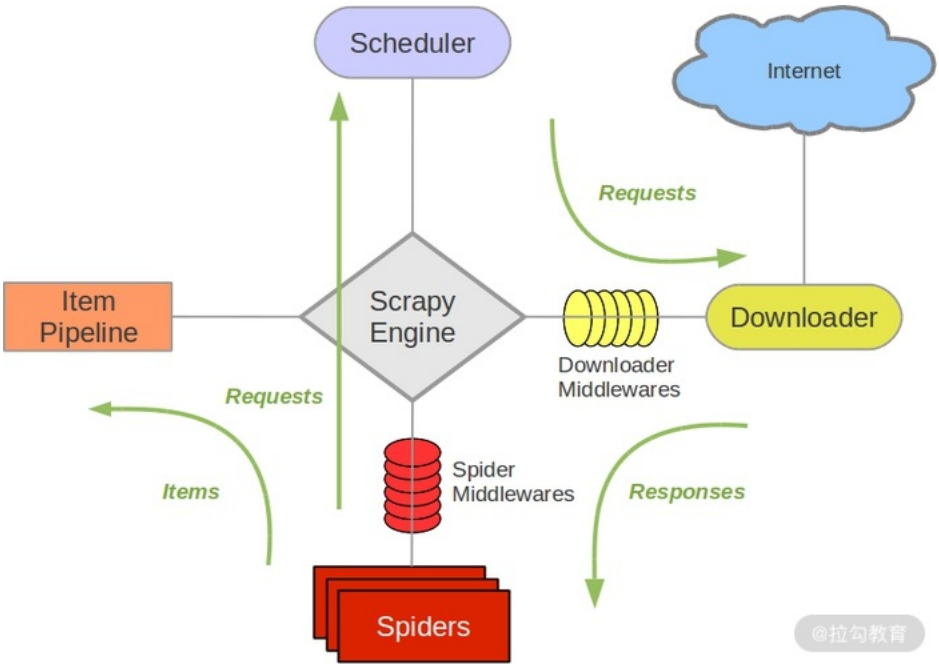
我们在前面几节课了解了 Scrapy 爬虫框架的用法。但这些框架都是在同一台主机上运行的，爬取效率比较低。如果能够实现多台主机协同爬取，那么爬取效率必然会成倍增长，这就是分布式爬虫的优势。

接下来我们就来了解一下分布式爬虫的基本原理，以及 Scrapy 实现分布式爬虫的流程。

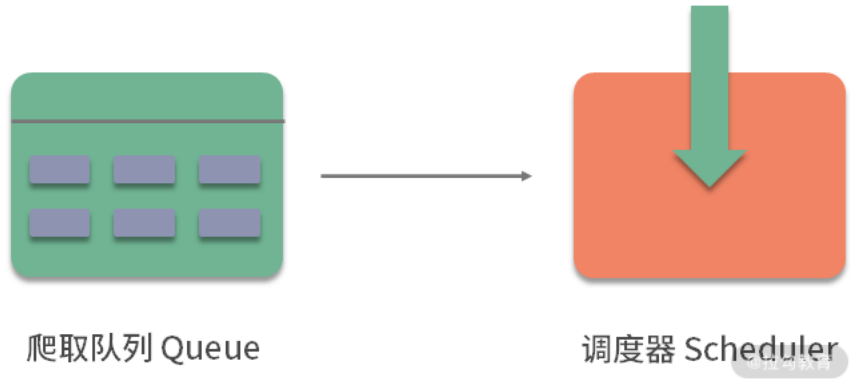
我们在前面已经实现了 Scrapy 基本的爬虫功能，虽然爬虫是异步加多线程的，但是我们却只能在一台主机上运行，所以爬取效率还是有限的，而分布式爬虫则是将多台主机组合起来，共同完成一个爬取任务，这将大大提高爬取的效率。

分布式爬虫架构

在了解分布式爬虫架构之前，首先回顾一下 Scrapy 的架构，如图所示。

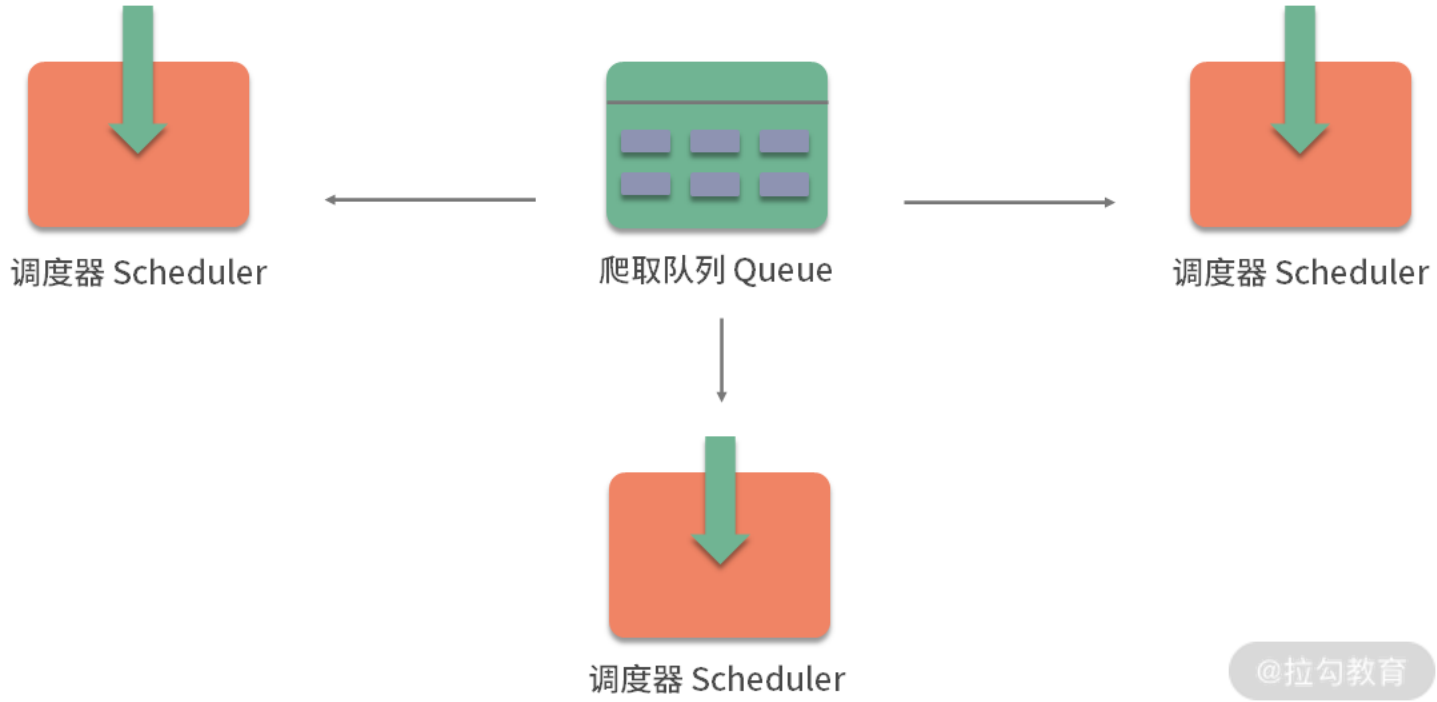


Scrapy 单机爬虫中有一个本地爬取队列 Queue，这个队列是利用 deque 模块实现的。如果新的 Request 生成就会放到队列里面，随后 Request 被 Scheduler 调度。之后，Request 交给 Downloader 执行爬取，简单的调度架构如图所示。



如果两个 Scheduler 同时从队列里面获取 Request，每个 Scheduler 都会有其对应的 Downloader，那么在带宽足够、正常爬取且不考虑队列存取压力的情况下，爬取效率会有什么变化呢？没错，爬取效率会翻倍。

这样，Scheduler 可以扩展多个，Downloader 也可以扩展多个。而爬取队列 Queue 必须始终为一个，也就是所谓的共享爬取队列。这样才能保证 Scheduer 从队列里调度某个 Request 之后，其他 Scheduler 不会重复调度此 Request，就可以做到多个 Schduler 同步爬取。这就是分布式爬虫的基本雏形，简单调度架构如图所示。



我们需要做的就是在多台主机上同时运行爬虫任务协同爬取，而协同爬取的前提就是共享爬取队列。这样各台主机就不需要维护各自的爬取队列了，而是从共享爬取队列存取 Request。但是各台主机还有各自的 Scheduler 和 Downloader，所以调度和下载功能是分别完成的。如果不考虑队列存取性能消耗，爬取效率还是可以成倍提高的。

维护爬取队列

那么如何维护这个队列呢？我们首先需要考虑的就是性能问题，那什么数据库存取效率高呢？这时我们自然想到了基于内存存储的 Redis，而且 Redis 还支持多种数据结构，例如列表 List、集合 Set、有序集合 Sorted Set 等，存取的操作也非常简单，所以在这里我们采用 Redis 来维护爬取队列。

这几种数据结构存储实际各有千秋，分析如下：

- 列表数据结构有 lpush、lpop、rpush、rpop 方法，所以我们可以用它实现一个先进先出的爬取队列，也可以实现一个先进后出的栈式爬取队列。
- 集合的元素是无序且不重复的，这样我们就可以非常方便地实现一个随机排序的不重复的爬取队列。
- 有序集合带有分数表示，而 Scrapy 的 Request 也有优先级的控制，所以我们用有序集合就可以实现一个带优先级调度的队列。

这些不同的队列我们需要根据具体爬虫的需求灵活选择。

怎样去重

Scrapy 有自动去重功能，它的去重使用了 Python 中的集合。这个集合记录了 Scrapy 中每个 Request 的指纹，这个指纹实际上就是 Request 的散列值。我们可以看看 Scrapy 的源代码，如下所示：

```
import hashlib
def request_fingerprint(request, include_headers=None):
    if include_headers:
        include_headers = tuple(to_bytes(h.lower())
                                for h in sorted(include_headers))
    cache = _fingerprint_cache.setdefault(request, {})
    if include_headers not in cache:
        fp = hashlib.sha1()
        fp.update(to_bytes(request.method))
        fp.update(to_bytes(canonicalize_url(request.url)))
        fp.update(request.body or b'')
        if include_headers:
            for hdr in include_headers:
                if hdr in request.headers:
                    fp.update(hdr)
                    for v in request.headers.getlist(hdr):
                        fp.update(v)
        cache[include_headers] = fp.hexdigest()
    return cache[include_headers]
```

request_fingerprint 就是计算 Request 指纹的方法，其方法内部使用的是 hashlib 的 sha1 方法。计算的字段包括 Request 的 Method、URL、Body、Headers 这几部分内容，这里只要有一点不同，那么计算的结果就不同。计算得到的结果是加密后的字符串，也就是指纹。每个 Request 都有独有的指纹，指纹就是一个字符串，判定字符串是否重复比判定 Request 对象是否重复容易得多，所以指纹可以作为判定 Request 是否重复的依据。

那么我们如何判定是否重复呢？Scrapy 是这样实现的，如下所示：

```
def __init__(self):
    self.fingerprints = set()
```

```
def request_seen(self, request):
    fp = self.request_fingerprint(request)
    if fp in self.fingerprints:
        return True
    self.fingerprints.add(fp)
```

在去重的类 RFPDupeFilter 中，有一个 request_seen 方法，这个方法有一个参数 request，它的作用就是检测该 Request 对象是否重复。这个方法调用 request_fingerprint 获取该 Request 的指纹，检测这个指纹是否存在于 fingerprints 变量中，而 fingerprints 是一个集合，集合的元素都是不重复的。如果指纹存在，那么就返回 True，说明该 Request 是重复的，否则将这个指纹加入集合中。如果下次还有相同的 Request 传递过来，指纹也是相同的，那么这时指纹就已经存在于集合中，Request 对象就会直接判定为重复。这样去重的目的就实现了。

Scrapy 的去重过程就是，利用集合元素的不重复特性来实现 Request 的去重。

对于分布式爬虫来说，我们肯定不能再使用每个爬虫各自的集合来去重了。因为这样还是每台主机单独维护自己的集合，不能做到共享。多台主机如果生成了相同的 Request，只能各自去重，各个主机之间就无法做到去重了。

那么要实现多台主机去重，这个指纹集合也需要是共享的，Redis 正好有集合的存储数据结构，我们可以利用 Redis 的集合作为指纹集合，那么这样去重集合也是共享的。每台主机新生成 Request 之后，会把该 Request 的指纹与集合比对，如果指纹已经存在，说明该 Request 是重复的，否则将 Request 的指纹加入这个集合中即可。利用同样的原理不同的存储结构我们也可以实现分布式 Reqeust 的去重。

防止中断

在 Scrapy 中，爬虫运行时的 Request 队列放在内存中。爬虫运行中断后，这个队列的空间就被释放，此队列就被销毁了。所以一旦爬虫运行中断，爬虫再次运行就相当于全新的爬取过程。

要做到中断后继续爬取，我们可以将队列中的 Request 保存起来，下次爬取直接读取保存数据即可获取上次爬取的队列。我们在 Scrapy 中指定一个爬取队列的存储路径即可，这个路径使用 JOB_DIR 变量来标识，我们可以用如下命令来实现：

```
scrapy crawl spider -s JOBDIR=crawls/spider
```

更加详细的使用方法可以参见官方文档，链接为：https://doc.scrapy.org/en/latest/topics/jobs.html。
在 Scrapy 中，我们实际是把爬取队列保存到本地，第二次爬取直接读取并恢复队列即可。那么在分布式架构中我们还用担心这个问题吗？不需要。因为爬取队列本身就是用数据库保存的，如果爬虫中断了，数据库中的 Request 依然是存在的，下次启动就会接着上次中断的地方继续爬取。

所以，当 Redis 的队列为空时，爬虫会重新爬取；当 Redis 的队列不为空时，爬虫便会接着上次中断之处继续爬取。

架构实现

我们接下来就需要在程序中实现这个架构了。首先需要实现一个共享的爬取队列，还要实现去重功能。另外，还需要重写一个 Scheduer 的实现，使之可以从共享的爬取队列存取 Request。

幸运的是，已经有人实现了这些逻辑和架构，并发布成了叫作 Scrapy-Redis 的 Python 包。

在下一节，我们便看看 Scrapy-Redis 的源码实现，以及它的详细工作原理。