基于 Redis 的三种分布式爬虫策略

爬虫是偏 IO 型的任务，分布式爬虫的实现难度比分布式计算和分布式存储简单得多。

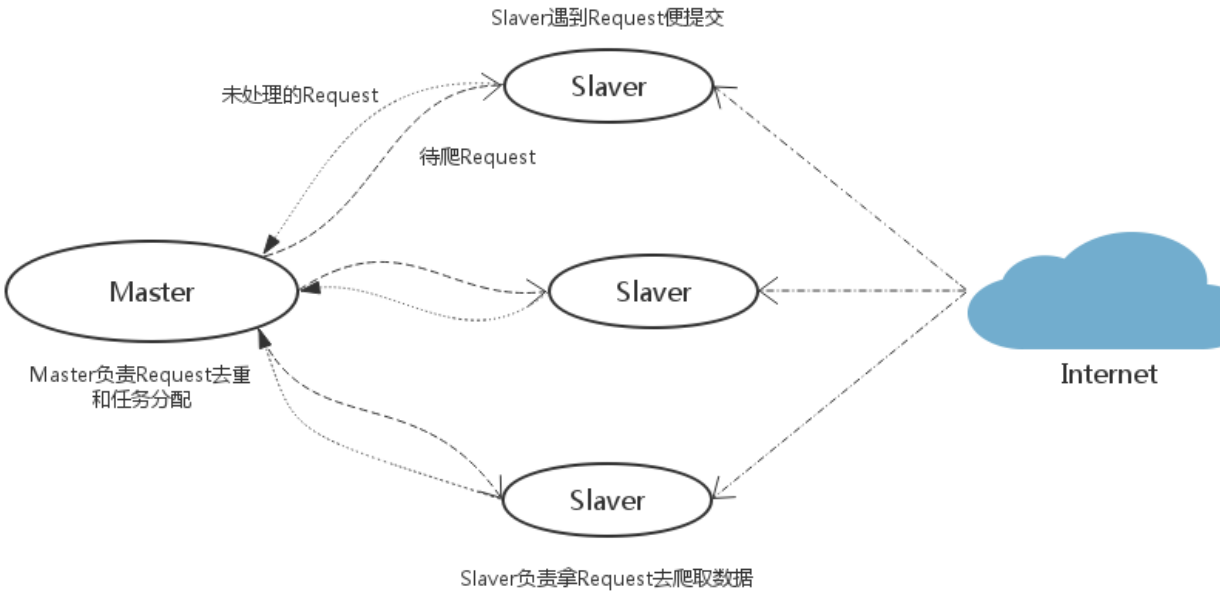
个人以为分布式爬虫需要考虑的点主要有以下几个：

- 爬虫任务的统一调度
- 爬虫任务的统一去重
- 存储问题
- 速度问题
- 足够“健壮”的情况下实现起来越简单/方便越好
- 最好支持“断点续爬”功能

Python 分布式爬虫比较常用的应该是 scrapy 框架加上 Redis 内存数据库，中间的调度任务等用 scrapy-redis 模块实现。

此处简单介绍一下基于 Redis 的三种分布式策略，其实它们之间还是很相似的，只是为适应不同的网络或爬虫环境作了一些调整而已（如有错误欢迎留言拍砖）。

【策略一】



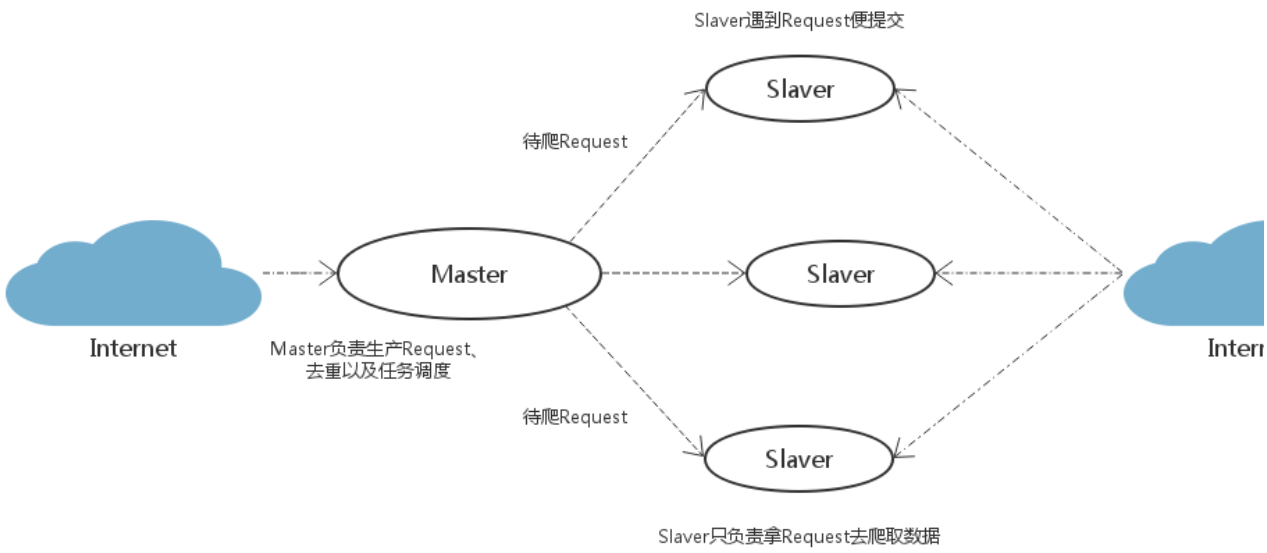
Slaver 端从 Master 端拿任务（Request/url/ID）进行数据抓取，在抓取数据的同时也生成新任务，并将任务抛给 Master。Master 端只有一个 Redis 数据库，负责对 Slaver 提交的任务进行去重、加入待爬队列。

优点： scrapy-redis 默认使用的就是这种策略，我们实现起来很简单，因为任务调度等工作 scrapy-redis 都已经帮我们做好了，我们只需要继承 RedisSpider、指定 redis_key 就行了。

缺点： scrapy-redis 调度的任务是 Request 对象，里面信息量比较大（不仅包含 url，还有 callback 函数、headers 等信息），导致的结果就是会降低爬虫速

度、而且会占用 Redis 大量的存储空间。当然我们可以重写方法实现调度 url 或者用户 ID。

【策略二】



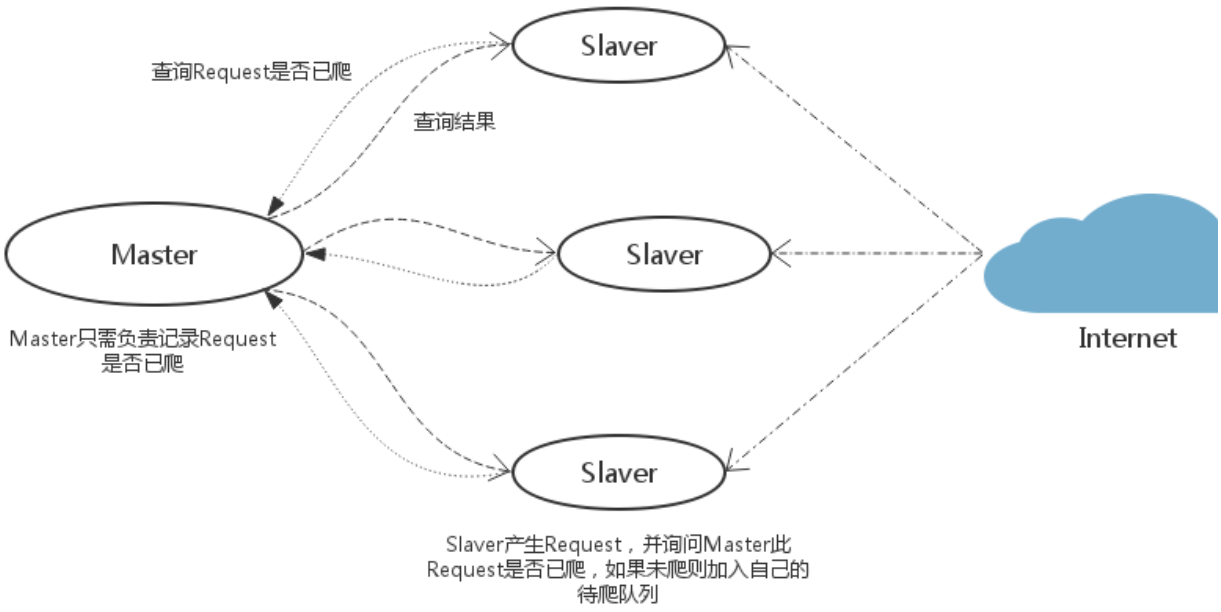
这是对策略的一种优化改进：在 Master 端跑一个程序去生成任务（Request/url/ID）。Master 端负责的是生产任务，并把任务去重、加入到待爬队列。Slaver 只管从 Master 端拿任务去爬。

优点： 将生成任务和抓取数据分开，分工明确，减少了 Master 和 Slaver 之间的数据交流；Master 端生成任务还有一个好处就是：可以很方便地重写判重策略（当数据量大时优化判重的性能和速度还是很重要的）。

缺点： 像 QQ 或者新浪微博这种网站，发送一个请求，返回的内容里面可能包

含几十个待爬的用户 ID，即几十个新爬虫任务。但有些网站一个请求只能得到一两个新任务，并且返回的内容里也包含爬虫要抓取的目标信息，如果将生成任务和抓取任务分开反而会降低爬虫抓取效率。毕竟带宽也是爬虫的一个瓶颈问题，我们要秉着发送尽量少的请求为原则，同时也是为了减轻网站服务器的压力，要做一只有道德的 Crawler。所以，视情况而定。

【策略三】



Master 中只有一个集合，它只有查询的作用。Slaver 在遇到新任务时询问 Master 此任务是否已爬，如果未爬则加入 Slaver 自己的待爬队列中，Master 把此任务记为已爬。它和策略一比较像，但明显比策略一简单。策略一的简单是因为有 scrapy-redis 实现了 scheduler 中间件，它并不适用于非 scrapy 框架的爬虫。

优点：实现简单，非 scrapy 框架的爬虫也适用。Master 端压力比较小，Master 与 Slaver 的数据交流也不大。

缺点：“健壮性”不够，需要另外定时保存待爬队列以实现“断点续爬”功能。各 Slaver 的待爬任务不通用。

结语：

如果把 Slaver 比作工人，把 Master 比作工头。策略一就是工人遇到新任务都上报给工头，需要干活的时候就去工头那里领任务；策略二就是工头去找新任务，工人只管从工头那里领任务干活 ;策略三就是工人遇到新任务时询问工头此任务是否有人做了，没有的话工人就将此任务加到自己的“行程表”。

我们在爬大型网站的时候，需要处理上千万乃至上亿的 url 的去重。如果采用 python 的自带 set,或者 redis 的 set,那就需要占用很大的内存。如果存入将 url 存入数据库去重，那速度又会变慢。这种量级以上的去重，一般是采用 BloomFilter，但是如果机器 down 机了，那 BloomFilter 在内存的数据中的数据，就没了。我们知道 redis 的数据既可以存在内存中，也可以存在硬盘中。如果能将 BloomFilter 和 redis 结合起来，那就非常棒了。

```
# encoding=utf-8
import redis
from hashlib import md5

class SimpleHash(object):
    def __init__(self, cap, seed):
        self.cap = cap
        self.seed = seed

    def hash(self, value):
        ret = 0
        for i in range(len(value)):
            ret += self.seed * ret + ord(value[i])
        return (self.cap - 1) &amp;amp; ret

class BloomFilter(object):
    def __init__(self, host='localhost', port=6379, db=0, blockNum=1,
key='bloomfilter'):
        """
        :param host: the host of Redis
        :param port: the port of Redis
        :param db: witch db in Redis
        :param blockNum: one blockNum for about 90,000,000; if you
have more strings for filtering, increase it.
        :param key: the key's name in Redis
        """
        self.server = redis.Redis(host=host, port=port, db=db)
```

```
        # <<表示二进制向左移动位数，比如 2<<2,2 的二进制表示 000010，向左移 2 位，就是 001000，就是十进制的 8
        self.bit_size = 1 <<31   # Redis 的 String 类型最大容量为 512M，现使用 256M
        self.seeds = [5, 7, 11, 13, 31, 37, 61]
        self.key = key
        self.blockNum = blockNum
        self.hashfunc = []
        for seed in self.seeds:
            self.hashfunc.append(SimpleHash(self.bit_size, seed))

    def isContains(self, str_input):
        if not str_input:
            return False
        m5 = md5()
        m5.update(str_input)
        str_input = m5.hexdigest()
        ret = True
        name = self.key + str(int(str_input[0:2], 16) % self.blockNum)
        for f in self.hashfunc:
            loc = f.hash(str_input)
            ret = ret &amp;amp; self.server.getbit(name, loc)
        return ret

    def insert(self, str_input):
        m5 = md5()
        m5.update(str_input)
        str_input = m5.hexdigest()
        name = self.key + str(int(str_input[0:2], 16) % self.blockNum)
        for f in self.hashfunc:
            loc = f.hash(str_input)
```

```
            self.server.setbit(name, loc, 1)


if __name__ == '__main__':
    """ 第一次运行时会显示 not exists!，之后再运行会显示 exists! """
    bf = BloomFilter()
    if bf.isContains('http://www.baidu.com'):   # 判断字符串是否存在
        print 'exists!'
    else:
        print 'not exists!'
        bf.insert('http://www.baidu.com')
```