首先回顾一下Scrapy-Redis的去重机制。Scrapy-Redis将Request的指纹存储到了Redis集合中，每个指纹的长度为40，例如27adcc2e8979cdee0c9cecbbe8bf8ff51edefb61就是一个指纹，它的每一位都是16进制数。
我们计算一下用这种方式耗费的存储空间。每个十六进制数占用4 b，1个指纹用40个十六进制数表示，占用空间为20 B，1万个指纹即占用空间200 KB，1亿个指纹占用2 GB。当爬取数量达到上亿级别时，Redis的占用的内存就会变得很大，而且这仅仅是指纹的存储。Redis还存储了爬取队列，内存占用会进一步提高，更别说有多个Scrapy项目同时爬取的情况了。当爬取达到亿级别规模时，Scrapy-Redis提供的集合去重已经不能满足我们的要求。所以我们需要使用一个更加节省内存的去重算法Bloom Filter。

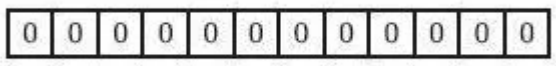
1 了解Bloom Filter

Bloom Filter，中文名称叫作布隆过滤器，是1970年由Bloom提出的，它可以被用来检测一个元素是否在一个集合中。Bloom Filter的空间利用效率很高，使用它可以大大节省存储空间。Bloom Filter使用位数组表示一个待检测集合，并可以快速地通过概率算法判断一个元素是否存在于这个集合中。利用这个算法我们可以实现去重效果。

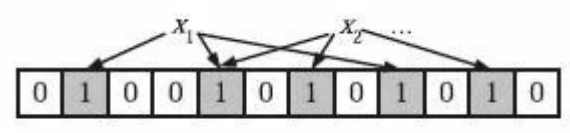
本节我们来了解Bloom Filter的基本算法，以及Scrapy-Redis中对接Bloom Filter的方法。

2 Bloom Filter的算法

在Bloom Filter中使用位数组来辅助实现检测判断。在初始状态下，我们声明一个包含m位的位数组，它的所有位都是0，如下图所示。



现在我们有了一个待检测集合，其表示为S={x1, x2, …, xn}。接下来需要做的就是检测一个x是否已经存在于集合S中。在Bloom Filter算法中，首先使用k个相互独立、随机的散列函数来将集合S中的每个元素x1, x2, …, xn映射到长度为m的位数组上，散列函数得到的结果记作位置索引，然后将位数组该位置索引的位置1。例如，我们取k为3，表示有三个散列函数，x1经过三个散列函数映射得到的结果分别为1、4、8，x2经过三个散列函数映射得到的结果分别为4、6、10，那么位数组的1、4、6、8、10这五位就会置为1，如下图所示。



如果有一个新的元素x，我们要判断x是否属于S集合，我们仍然用k个散列函数对x求映射结

果。如果所有结果对应的位数组位置均为1，那么x属于S这个集合；如果有一个不为1，则x不属于S集合。

例如，新元素x经过三个散列函数映射的结果为4、6、8，对应的位置均为1，则x属于S集合。如果结果为4、6、7，而7对应的位置为0，则x不属于S集合。

注意，这里m、n、k满足的关系是m>nk，也就是说位数组的长度m要比集合元素n和散列函数k的乘积还要大。

这样的判定方法很高效，但是也是有代价的，它可能把不属于这个集合的元素误认为属于这个集合。我们来估计一下这种方法的错误率。当集合S={x1, x2,..., xn} 的所有元素都被k个散列函数映射到m位的位数组中时，这个位数组中某一位还是0的概率是：

$$p' = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}.$$

散列函数是随机的，则任意一个散列函数选中这一位的概率为1/m，那么1-1/m就代表散列函数从未没有选中这一位的概率，要把S完全映射到m位数组中，需要做kn次散列运算，最后的概率就是1-1/m的kn次方。

一个不属于S的元素x如果误判定为在S中，那么这个概率就是k次散列运算得到的结果对应的位数组位置都为1，则误判概率为：

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k}$$

在给定m、n时，可以求出使得f最小化的k值为：

$$\frac{m}{n} \ln 2 \approx \frac{9m}{13n} \approx 0.7\frac{m}{n}$$

这里将误判概率归纳如下：

m/n	最优k	k=1	k=2	k=3	k=4	k=5	k=6	k=7	k=8
2	1.39	0.393	0.400						
3	2.08	0.283	0.237	0.253					
4	2.77	0.221	0.155	0.147	0.160				
5	3.46	0.181	0.109	0.092	0.092	0.101			
6	4.16	0.154	0.0804	0.0609	0.0561	0.0578	0.0638		
7	4.85	0.133	0.0618	0.0423	0.0359	0.0347	0.0364		
8	5.55	0.118	0.0489	0.0306	0.024	0.0217	0.0216	0.0229	
9	6.24	0.105	0.0397	0.0228	0.0166	0.0141	0.0133	0.0135	0.0145
10	6.93	0.0952	0.0329	0.0174	0.0118	0.00943	0.00844	0.00819	0.00846
11	7.62	0.0869	0.0276	0.0136	0.00864	0.0065	0.00552	0.00513	0.00509
12	8.32	0.08	0.0236	0.0108	0.00646	0.00459	0.00371	0.00329	0.00314
13	9.01	0.074	0.0203	0.00875	0.00492	0.00332	0.00255	0.00217	0.00199
14	9.7	0.0689	0.0177	0.00718	0.00381	0.00244	0.00179	0.00146	0.00129
15	10.4	0.0645	0.0156	0.00596	0.003	0.00183	0.00128	0.001	0.000852
16	11.1	0.0606	0.0138	0.005	0.00239	0.00139	0.000935	0.000702	0.000574
17	11.8	0.0571	0.0123	0.00423	0.00193	0.00107	0.000692	0.000499	0.000394
18	12.5	0.054	0.0111	0.00362	0.00158	0.000839	0.000519	0.00038	0.000275

表中第一列为m/n的值，第二列为最优k值，其后列为不同k值的误判概率。当k值确定时，随着m/n的增大，误判概率逐渐变小。当m/n的值确定时，当k越靠近最优K值，误判概率越小。误判概率总体来看都是极小的，在容忍此误判概率的情况下，大幅减小存储空间和判定速度是完全值得的。

接下来，我们将Bloom Filter算法应用到Scrapy-Redis分布式爬虫的去重过程中，以解决Redis内存不足的问题。

3. 对接Scrapy-Redis

实现Bloom Filter时，首先要保证不能破坏Scrapy-Redis分布式爬取的运行架构。我们需要修改Scrapy-Redis的源码，将它的去重类替换掉。同时，Bloom Filter的实现需要借助于一个位数组，既然当前架构还是依赖于Redis，那么位数组的维护直接使用Redis就好了。

首先实现一个基本的散列算法，将一个值经过散列运算后映射到一个m位数组的某一位上，代码如下：

```
class HashMap(object):
    def __init__(self, m, seed):
        self.m = m
        self.seed = seed        def hash(self, value):
        """
        Hash Algorithm
        :param value: Value
        :return: Hash Value
        """
```

```
        ret = 0
        for i in range(len(value)):
            ret += self.seed * ret + ord(value[i])                    return (self.m
```

这里新建了一个HashMap类。构造函数传入两个值，一个是m位数组的位数，另一个是种子值seed。不同的散列函数需要有不同的seed，这样可以保证不同的散列函数的结果不会碰撞。

在hash()方法的实现中，value是要被处理的内容。这里遍历了value的每一位，并利用ord()方法取到每一位的ASCII码值，然后混淆seed进行迭代求和运算，最终得到一个数值。这个数值的结果就由value和seed唯一确定。我们再将这个数值和m进行按位与运算，即可获取到m位数组的映射结果，这样就实现了一个由字符串和seed来确定的散列函数。当m固定时，只要seed值相同，散列函数就是相同的，相同的value必然会映射到相同的位置。所以如果想要构造几个不同的散列函数，只需要改变其seed就好了。以上内容便是一个简易的散列函数的实现。

接下来我们再实现Bloom Filter。Bloom Filter里面需要用到k个散列函数，这里要对这几个散列函数指定相同的m值和不同的seed值，构造如下：

```
BLOOMFILTER_HASH_NUMBER = 6BLOOMFILTER_BIT = 30class BloomFilter(object):
    def __init__(self, server, key, bit=BLOOMFILTER_BIT, hash_number=BLOOMFILTER
        """
        Initialize BloomFilter
        :param server: Redis Server
        :param key: BloomFilter Key
        :param bit: m = 2 ^ bit
        :param hash_number: the number of hash function
        """
        # default to 1 << 30 = 10,7374,1824 = 2^30 = 128MB, max filter 2^30/hash
        self.m = 1 << bit
        self.seeds = range(hash_number)
        self.maps = [HashMap(self.m, seed) for seed in self.seeds]
        self.server = server
        self.key = key
```

由于我们需要亿级别的数据的去重，即前文介绍的算法中的n为1亿以上，散列函数的个数k大约取10左右的量级。而m>kn，这里m值大约保底在10亿，由于这个数值比较大，所以这里用移位操作来实现，传入位数bit，将其定义为30，然后做一个移位操作1<<30，相当于2的30次方，等于1073741824，量级也是恰好在10亿左右，由于是位数组，所以这个位数组占用的大小就是2^30 b=128 MB。开头我们计算过Scrapy-Redis集合去重的占用空间大约在2 GB左右，可见Bloom Filter的空间利用效率极高。

随后我们再传入散列函数的个数，用它来生成几个不同的seed。用不同的seed来定义不同的散列函数，这样我们就可以构造一个散列函数列表。遍历seed，构造带有不同seed值的HashMap对象，然后将HashMap对象保存成变量maps供后续使用。

另外，server就是Redis连接对象，key就是这个m位数组的名称。

接下来，我们要实现比较关键的两个方法：一个是判定元素是否重复的方法exists()，另一个是添加元素到集合中的方法insert()，实现如下：

```
def exists(self, value):
    """
    if value exists
    :param value:
    :return:
    """
    if not value:                return False
    exist = 1
    for map in self.maps:
        offset = map.hash(value)
        exist = exist & self.server.getbit(self.key, offset)        return exist
    """
    add value to bloom
    :param value:
    :return:
    """
    for f in self.maps:
        offset = f.hash(value)
        self.server.setbit(self.key, offset, 1)
```

首先看下insert()方法。Bloom Filter算法会逐个调用散列函数对放入集合中的元素进行运算，得到在m位位数组中的映射位置，然后将位数组对应的位置置1。这里代码中我们遍历了初始化好的散列函数，然后调用其hash()方法算出映射位置offset，再利用Redis的setbit()方法将该位置1。

在exists()方法中，我们要实现判定是否重复的逻辑，方法参数value为待判断的元素。我们首先定义一个变量exist，遍历所有散列函数对value进行散列运算，得到映射位置，用getbit()方法取得该映射位置的结果，循环进行与运算。这样只有每次getbit()得到的结果都为1时，最后的exist才为True，即代表value属于这个集合。如果其中只要有一次getbit()得到的结果为0，即m位数组中有对应的0位，那么最终的结果exist就为False，即代表value不属于这个集合。

Bloom Filter的实现就已经完成了，我们可以用一个实例来测试一下，代码如下：

```
conn = StrictRedis(host='localhost', port=6379, password='foobared')
bf = BloomFilter(conn, 'testbf', 5, 6)
bf.insert('Hello')
bf.insert('World')
result = bf.exists('Hello')
print(bool(result))
result = bf.exists('Python')
print(bool(result))
```

这里首先定义了一个Redis连接对象，然后传递给Bloom Filter。为了避免内存占用过大，这里传的位数bit比较小，设置为5，散列函数的个数设置为6。

调用insert()方法插入Hello和World两个字符串，随后判断Hello和Python这两个字符串是否存在，最后输出它的结果，运行结果如下：

```
TrueFalse
```

很明显，结果完全没有问题。这样我们就借助Redis成功实现了Bloom Filter的算法。

接下来继续修改Scrapy-Redis的源码，将它的dupefilter逻辑替换为Bloom Filter的逻辑。这里主要是修改RFPDupeFilter类的request_seen()方法，实现如下：

```
def request_seen(self, request):
    fp = self.request_fingerprint(request)        if self.bf.exists(fp):
    self.bf.insert(fp)            return False
```

利用request_fingerprint()方法获取Request的指纹，调用Bloom Filter的exists()方法判定该指纹是否存在。如果存在，则说明该Request是重复的，返回True，否则调用Bloom Filter的insert()方法将该指纹添加并返回False。这样就成功利用Bloom Filter替换了Scrapy-Redis的集合去重。

对于Bloom Filter的初始化定义，我们可以将__init__()方法修改为如下内容：

```
def __init__(self, server, key, debug, bit, hash_number):
    self.server = server
    self.key = key
    self.debug = debug
    self.bit = bit
    self.hash_number = hash_number
    self.logdupes = True
    self.bf = BloomFilter(server, self.key, bit, hash_number)
```

其中bit和hash_number需要使用from_settings()方法传递，修改如下：

```
@classmethoddef from_settings(cls, settings):
    server = get_redis_from_settings(settings)
    key = defaults.DUPEFILTER_KEY % {'timestamp': int(time.time())}
    debug = settings.getbool('DUPEFILTER_DEBUG', DUPEFILTER_DEBUG)
    bit = settings.getint('BLOOMFILTER_BIT', BLOOMFILTER_BIT)
    hash_number = settings.getint('BLOOMFILTER_HASH_NUMBER', BLOOMFILTER_HASH_NU
```

其中，常量DUPEFILTER_DEBUG和BLOOMFILTER_BIT统一定义在defaults.py中，默认如下：

```
BLOOMFILTER_HASH_NUMBER = 6BLOOMFILTER_BIT = 30
```

现在，我们成功实现了Bloom Filter和Scrapy-Redis的对接。

为了方便使用，本节的代码已经打包成一个Python包并发布到PyPi，链接为https://pypi.python.org/pypi/scrapy-redis-bloomfilter，可以直接使用ScrapyRedisBloomFilter，不需要自己实现一遍。

我们可以直接使用pip来安装，命令如下：

```
pip3 install scrapy-redis-bloomfilter
```

使用的方法和Scrapy-Redis基本相似，在这里说明几个关键配置。

```
# 去重类，要使用Bloom Filter请替换DUPEFILTER_CLASSDUPEFILTER_CLASS = "scrapy_redis_
```

- DUPEFILTER_CLASS是去重类，如果要使用Bloom Filter，则DUPEFILTER_CLASS需要修改为该包的去重类。
- BLOOMFILTER_HASH_NUMBER是Bloom Filter使用的散列函数的个数，默认为6，可以根据去重量级自行修改。
- BLOOMFILTER_BIT即前文所介绍的BloomFilter类的bit参数，它决定了位数组的位数。如果BLOOMFILTER_BIT为30，那么位数组位数为2的30次方，这将占用Redis 128 MB的存储空间，去重量级在1亿左右，即对应爬取量级1亿左右。如果爬取量级在10亿、20亿甚至100亿，请务必将此参数对应调高。

源代码附有一个测试项目，放在tests文件夹，该项目使用了ScrapyRedisBloomFilter来去重，Spider的实现如下

```
from scrapy import Request, Spiderclass TestSpider(Spider):
    name = 'test'
    base_url = 'https://www.baidu.com/s?wd='

    def start_requests(self):
        for i in range(10):
            url = self.base_url + str(i)                        yield Request(ur
        for i in range(100):
            url = self.base_url + str(i)                        yield Request(ur
        self.logger.debug('Response of ' + response.url)
```

start_requests()方法首先循环10次，构造参数为0$_{9}$的URL，然后重新循环了100次，构造了参数为0^{99}的URL。那么这里就会包含10个重复的Request，我们运行项目测试一下：

```
scrapy crawl test
```

```
{'bloomfilter/filtered': 10, 'downloader/request_bytes': 34021, 'downloader/requ
```

最后统计的第一行的结果：

```
'bloomfilter/filtered': 10,
```

这就是Bloom Filter过滤后的统计结果，它的过滤个数为10个，也就是它成功将重复的10个Reqeust识别出来了，测试通过。
以上内容便是Bloom Filter的原理及对接实现，Bloom Filter的使用可以大大节省Redis内存。在数据量大的情况下推荐此方案。