# 爬虫中的去重处理

## 课程概要

* 去重应用场景以及基本原理
* 基于信息摘要算法的去重
* 基于simhash算法的去重
* 布隆过滤器原理与实现
* 总结

## 1. 去重应用场景以及基本原理

### 1.1 爬虫中什么业务需要使用去重

* 防止发出重复的请求 数据
* 防止存储重复的数据 字符串, 文件, 图片, 都可以转化为二进制的数据

### 1.2 去重实现的基本原理

* 根据给定的**判断依据**和给定的**去重容器**，将**原始数据**逐一进行判断，判断去重容器中是否有该数据。如果没有那就把该数据对应的判断依据添加去重容器中，同时标记该数据是不重复数据；如果有就不添加，同时标记该数据是重复数据。
  + **判断依据**（原始数据、原始数据特征值）
  + **去重容器**（存储判断数据） set()

*## 给定一批数据, 进行去重*

*# 需求一：剔除重复数据 == set()*

data1 = [123, "123", 123, "456", "qwe", "qwe"]

*# 使用 for 循环 + if 判断*

*# 使用set进行去重*

result1 = list(set(data1))

*# [123, "123", "456", "qwe"]*

*# 需求二：剔除重复数据，同时规定 "100" 和 100是重复的 str() ==*

data2 = [123, "123", 123, "456", "qwe", "qwe"]

result2 = list(set([str(i) **for** i **in** data2]))

*# result2 = [123, "456", "qwe"]*

**class** **Test**(object):

**def** \_\_init\_\_(self, v):

self.v = v

t1 = Test(100)

t2 = Test(100)

t3 = Test(200)

t4 = t1

*# 需求三：剔除重复数据, 对象的去重 == set()*

data3 = [t1, t2, t3, t4]

result3 = [t1, t2, t3]

*# 需求四：剔除重复数据，规定：Test对象的 v 相同则为重复对象 t.v ==*

data4 = [t1, t2, t3, t4]

result4 = [t1, t3]

*# 需求五：剔除重复数据，规定：对象的类相同, 则表示是重复的对象 t.\_\_class\_\_ ==*

data5 = [t1, t2, t3, t4]

result5 = [t1]

*# 判断依据 (如何规定两个数据是重复数据)*

*# 去重容器 (存储判断原始数据的判断依据)*

*## 给定一批即将产生的数据, 进行去重. 数据是逐个产生的.*

*# 假设即将产生的数据为data1*

*# data1 = [123, "123", 123, "456", "qwe", "qwe"]*

*# 需求一：剔除重复数据 == set()*

result1 = [] *# 存储最终结果*

sign1 = [] *# 存储判断依据*

new\_data = None *# "123"*

**if** new\_data **not** **in** sign1:

sign1.append(new\_data)

result1.append(new\_data)

*# sign1: ["123", 123, "456", "qwe"]*

*# result1 : ["123", 123, "456", "qwe"]*

*# 需求二：剔除重复数据，同时规定"100"和100是重复的 str() ==*

result2 = []

sign2 = [] *# 存储判断依据, 数据的特征 str(原始数据) 去重容器*

new\_data = None *# "123"*

**if** str(new\_data) **not** **in** sign2:

sign2.append(str(new\_data))

result2.append(new\_data)

*# sign2: ["123", "456", "qwe"]*

*# result2 : [123, "456", "qwe"]*

**class** **Test**(object):

**def** \_\_init\_\_(self, v):

self.v = v

t1 = Test(100)

t2 = Test(100)

t3 = Test(200)

t4 = t1

*# 需求三：剔除重复数据 == set()*

data3 = [t1, t2, t3, t4]

set()

result3 = [t1, t2, t3]

*# 需求四：剔除重复数据，规定：Test对象的v相同则为重复对象 t.v ==*

data4 = [t1, t2, t3, t4]

sign4 = [100,200]

result4 = [t1, t3]

*# 需求五：剔除重复数据，规定：对象的类相同,则表示是重复的对象 t.\_\_class\_\_ ==*

data5 = [t1, t2, t3, t4]

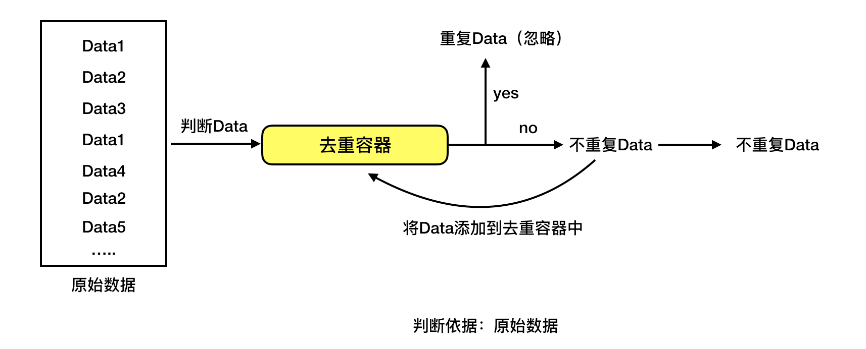
sign5 = [Test]

result5 = [t1]

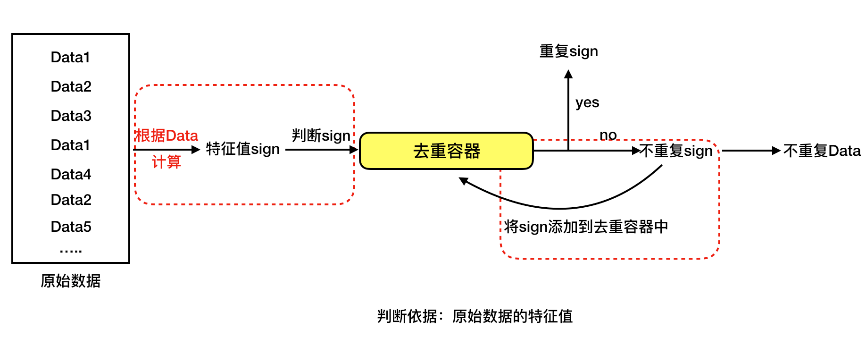
*# 判断依据 (如何规定两个数据是重复数据)*

*# 去重容器 (存储判断原始数据的判断依据)*

### 1.3 根据原始数据进行去重判断



### 1.4 根据原始数据的特征值进行去重判断



思考：为什么要用原始数据的特征值进行判断？

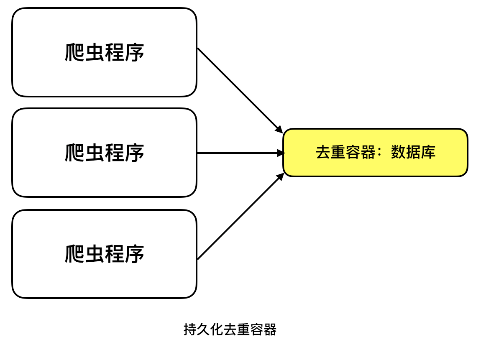
对于大数据, 节省存储空间

判断速度要比较快, md5, 16bytes, 16字节

### 1.5 临时去重容器与持久化去重容器

* **临时去重容器**指如利用list、set等编程语言的数据结构存储去重数据，一旦程序关闭或重启后，去重容器中的数据就被回收了。
  + 特点: 使用与实现简单方便；但无法共享、无法持久化
* **持久化去重容器**指如利用redis、mysql等数据库存储去重数据。
  + 特点: 持久化、共享；但使用与实现相对复杂

数据非常大时, 不能使用redis型的内存型数据库, 而要使用mysql型的磁盘型数据库

### 1.6 常用几种特殊的原始数据特征值计算

* 信息摘要hash算法（指纹)
* SimHash算法, 主要用于模糊文本的去重.
* 布隆过滤器方式, 主要用于海量数据的去重, 上亿级别

## 2. 基于信息摘要算法的去重

### 2.1 信息摘要hash算法介绍

* 信息摘要hash算法指可以将任意长度的文本、字节数据，通过一个算法得到一个固定长度的文本。 如MD5(128位)、SHA1(160位)等。
  + 特征：只要源文本不同，计算得到的结果，必然不同（摘要）。
  + 摘要：摘要算法主要用于比对信息源是否一致，因为只要源发生变化，得到的摘要必然不同；而且通常结果要比源短很多，所以称为“摘要”。
* 正因此，利用信息摘要算法能大大降低去重容器的存储空间使用率，并提高判断速度，且由于其强唯一性的特征，几乎不存在误判。
* 注意：
  + hash算法得出的结果其实本质上就是一串数值，如md5的128位指的是二进制的长度，十六进制的长度是32位。一个十六进制等于四个二进制。

MD5是 32位长度, sha1是40位长度的, 表示得出的结果的16进制的长度

md5, 128位, sha1, 160位是指的二进制位的长度.

**from** **hashlib** **import** md5

*# 创建一个md5对象*

m5 = md5()

*# 使用md5的update方法来生成某个python对象的md5值.*

*# md5.update()的对象必须是二进制的数据, 必须要把str字符串进行encode()转换为二进制的文本. encode默认使用utf-8的编码, 中文不支持ascii编码, 只能使用utf-8编码*

m5.update("sdfpoksodfsafs".encode())

**print**(m5.hexdigest())

*# fda6d7fb64dd99511046252a99c49a57*

*# 在字符串前加b, 字符串按照ascii码进行编码*

*# update相当于前后两个字符串进行拼接*

m5.update(b"sdfpoksodfsafs")

**print**(m5.hexdigest())

*# 2e303051141be2a210982eb928561935*

m5.update("sdfpoksodfsafs".encode("ascii"))

**print**(m5.hexdigest())

*# 552e374aff2e1f87e9b8341fcfae86fa*

*# 把16进制的"f"转换为10进制*

**print**(int("f",16))

*# 15*

*# 再把结果转换为2进制, 每一个16进制转换为2进制后是占4位的. 16进制的md5是32位长度的, 转换为2进制则是128位二进制位的. 1byte字节== 8bin比特位, 128比特位 bit = 128/8 = 16字节. sha1 == 20字节.*

**print**(bin(int("f",16)))

*# 0b1111*

### 2.2 信息摘要hash算法去重方案实现

* 普通内存版本 一般使用set()来完成, 临时的去重容器, 只在当前容器的生命周期内有效.
* Redis持久化版本, 速度快, 记录不会丢失.
* MySQL持久化版本, 记录不会丢失, 去重时效性要求不高, 节省内存. 数据量在几十万到上百万范围内, 使用mysql单表查询时速度了不会太慢.

#### 开发环境搭建

新建名为code的项目, 新建python-dev/Dockerfile, 进行开发环境的搭建.

# 配置基本的python开发环境

FROM python:alpine3.6

RUN apk add --no-cache gcc musl-dev

RUN apk add --no-cache libffi-dev && pip install ipython

RUN apk add --no-cache openssl-dev &&\

pip install redis &&\

pip install pymysql &&\

pip install sqlalchemy &&\

pip install six &&\

pip install w3lib

RUN pip install simhash

运行Dockerfile, 创建开发镜像.

#### 基类代码的实现

代码实现, 三种方法实现统一的接口, 通过传入不同的参数来调用不同的方法.

新建information\_summary\_filter的python程序包/模块, 因为三种方法的判断依据和具体的功能代码都不同, 可以新建3个不同的文件来完成相应的功能.

新建memory\_filter.py, 用于基于内存的去重.

新建redis\_filter.py, 用于基于redis的去重.

新建mysql\_filter.py, 用于基于mysql的去重.

由于使用统一的接口进行调用, 所以三个文件中的方法名必须要相同, 可以在\_\_init\_\_.py中建立一个基类, 基类中定义公有的方法, 在各个功能模块中继承并重写基类.

修改information\_summary\_filter/\_\_init\_\_.py文件

*# 基于信息摘要算法进行数据的去重判断和存储*

*# 1. 基于内存的存储*

*# 2. 基于redis的存储*

*# 3. 基于mysql的存储*

**import** **six**

**import** **hashlib**

**class** **BaseFilter**(object):

*'''基于信息摘要算法进行数据的去重判断和存储'''*

**def** \_\_init\_\_(self, hash\_func\_name="md5"):

*# 从hashlib模块中获取md5属性*

self.hash\_func = getattr(hashlib, hash\_func\_name)

*# 每一种去重方法都需要不同的数据结构来实现, 内存去重使用set集合来实现, mysql对应mysql的连接, redis对应redis的连接*

*# 这里如果使用None, 那么在每个子类中都要重写\_\_init\_\_方法, 上面的所有参数都要进行重写, 所以最好是定义一个方法, 在子类中重写同名的方法即可.*

*# self.storage = None*

self.storage = self.\_get\_storage()

*# 判断传入的数据是否是 bytes 二进制类型的字符串*

**def** \_safe\_data(self, data):

*'''*

*python2 str === python3 bytes*

*python2 uniocde === python3 str*

*:param data: 给定的原始数据*

*:return: 二进制类型的字符串数据*

*'''*

**if** six.PY3:

**if** isinstance(data, bytes):

**return** data

**elif** isinstance(data, str):

**return** data.encode()

**else**:

**raise** **Exception**("请提供一个字符串") *# 建议使用英文来描述, 代码可能在服务器上运行, 可能不支持中文*

**else**:

**if** isinstance(data, str):

**return** data

**elif** isinstance(data, unicode):

**return** data.encode()

**else**:

**raise** **Exception**("请提供一个字符串") *# 建议使用英文来描述*

*# 因为 save方法和is\_exists方法中都用到计算hash值的方法, 所以单独定义一个方法来计算hash值*

**def** \_get\_hash\_value(self, data):

*'''*

*根据给定的数据，返回的对应信息摘要hash值*

*:param data: 给定的原始数据（二进制类型的字符串数据）*

*:return: hash值*

*'''*

hash\_obj = self.hash\_func()

*# update的对象必须是二进制的字符串数据, 先使用一个方法进行判断, 如果不是bytes类型的, 就先转换为bytes类型.*

*# python3 bytes python2 str*

hash\_obj.update(self.\_safe\_data(data))

hash\_value = hash\_obj.hexdigest()

**return** hash\_value

**def** save(self, data):

*'''*

*根据data计算出对应的指纹进行存储*

*:param data: 给定的原始数据, 数据类型为二进制类型的字符串数据*

*:return: 存储的结果, 存储成功还是失败*

*'''*

*# 计算hash value*

hash\_value = self.\_get\_hash\_value(data)

*# 因为3种方法具体保存时的逻辑是不同的, 所以可以定义一个新的方法\_save, 子类在继承时重写\_save方法即可.*

**return** self.\_save(hash\_value)

**def** \_save(self, hash\_value):

*'''*

*存储对应的hash值（交给对应的子类去继承）*

*:param hash\_value: 通过信息摘要算法求出的hash值*

*:return: 存储的结果*

*'''*

**pass**

**def** is\_exists(self, data):

*'''*

*判断给定的数据对应的指纹是否存在*

*:param data: 给定的原始数据*

*:return: True or False*

*'''*

hash\_value = self.\_get\_hash\_value(data)

*# 同样, 定义一个新的方法, 交给子类继承并重写*

**return** self.\_is\_exists(hash\_value)

**def** \_is\_exists(self, hash\_value):

*'''*

*判断对应的hash值是否已经存在（交给对应的子类去继承）*

*:param hash\_value: 通过信息摘要算法求出的hash值*

*:return: 判断的结果（True or False）*

*'''*

**pass**

**def** \_get\_storage(self):

*'''*

*返回对应的一个存储对象（交给对应的子类去继承）*

*:return:*

*'''*

**pass**

#### 基于内存去重的代码实现

新建information\_summary\_filter/memory\_filter.py文件, 实现基于内存的去重方法.

*# 基于python中的 集合数据结构进行去重判断依据的存储*

**from** **.** **import** BaseFilter

**class** **MemoryFilter**(BaseFilter):

*'''基于python中的 集合数据结构进行去重判断依据的存储'''*

**def** \_get\_storage(self):

*# 基于内存的去重方法使用python中的集合来实现, 返回一个空集合*

**return** set()

**def** \_save(self, hash\_value):

*'''*

*利用set进行存储*

*:param hash\_value:*

*:return:*

*'''*

*# self.storage = self.\_get\_storage(), 这里重写了\_get\_storage()方法, 就使用的是 set(), 相当于set().add()方法*

**return** self.storage.add(hash\_value)

**def** \_is\_exists(self, hash\_value):

**if** hash\_value **in** self.storage:

**return** True

**return** False

在项目根目录中新建info\_filter\_demo.py, 对去重方法进行测试.

**from** **information\_summary\_filter.memory\_filter** **import** MemoryFilter

*# 创建对象*

filter = MemoryFilter()

*# 需要进行去重的数据*

data = ["111", "qwe", '222', "333" ,"111", "qwe", "中文", "qwer"]

*# 进行去重*

**for** d **in** data:

**if** filter.is\_exists(d):

**print**("发现重复的数据: ", d)

**else**:

filter.save(d)

**print**("保存去重的数据：", d)

右键运行info\_filter\_demo.py, 就会看到执行的结果.

#### 基于redis去重的代码实现

新建information\_summary\_filter/redis\_filter.py, 实现基于redis的去重.

*# 基于redis的持久化存储的去重判断依据的实现*

**import** **redis**

**from** **.** **import** BaseFilter

**class** **RedisFilter**(BaseFilter):

*'''基于redis的持久化存储的去重判断依据的实现'''*

**def** \_get\_storage(self):

*'''返回一个redis连接对象'''*

*# 因为save方法和is\_exists方法都是要频繁执行的, 每次执行都要新建并关闭tcp类型的redis连接, 为了减轻服务器的压力, 创建一个redis连接池*

*# 为了在创建redis连接时用户可以自定义连接的地址, 端口和数据, 可以在\_\_init\_\_方法中定义相应的参数.*

pool = redis.ConnectionPool(host=self.redis\_host, port=self.redis\_port, db=self.redis\_db)

client = redis.StrictRedis(connection\_pool=pool)

**return** client

**def** \_save(self, hash\_value):

*'''*

*利用redis的无序集合进行存储*

*:param hash\_value:*

*:return:*

*'''*

*# redis是key-value类型的数据库, 所以使用sadd添加数据时要指定key和value*

**return** self.storage.sadd(self.redis\_key, hash\_value)

**def** \_is\_exists(self, hash\_value):

*'''判断redis对应的无序集合中是否有对应的判断依据'''*

**return** self.storage.sismember(self.redis\_key, hash\_value)

修改information\_summary\_filter/\_\_init\_\_.py, 添加redis的实例化参数

**class** **BaseFilter**(object):

*'''基于信息摘要算法进行数据的去重判断和存储'''*

**def** \_\_init\_\_(self,

hash\_func\_name="md5",

redis\_host="localhost",

redis\_port=6379,

redis\_db=0,

redis\_key="filter",

mysql\_url=None,

mysql\_table\_name="filter"):

*# 注意, 因为在子类的\_get\_storage()方法中调用了redis, mysql的连接参数, 所以连接参数都要放在\_get\_storage()方法的前面, 否则就会出错.*

self.redis\_host = redis\_host

self.redis\_port = redis\_port

self.redis\_db = redis\_db

*# redis是key-value的数据库, 想要向集合中保存数据, 必须要指定对应的key, 在实例化时可以自定义redis\_key*

self.redis\_key = redis\_key

*# 从hashlib模块中获取md5属性*

self.hash\_func = getattr(hashlib, hash\_func\_name)

*# 每一种去重方法都需要不同的数据结构来实现, 内存去重使用set集合来实现, mysql对应mysql的连接, redis对应redis的连接*

*# 这里如果使用None, 那么在每个子类中都要重写\_\_init\_\_方法, 上面的所有参数都要进行重写, 所以最好是定义一个方法, 在子类中重写同名的方法即可.*

*# self.storage = None*

self.storage = self.\_get\_storage()

修改info\_filter\_demo.py, 对redis\_filter进行测试. 由于是基于redis的去重, 所以需要有redis服务器, 在pycharm中先基于redis:alpine镜像创建redis容器, 查看容器的ip地址, 在实例化redis\_filter时添加此ip地址.

*# python3*

*# from information\_summary\_filter import MemoryFilter*

**from** **information\_summary\_filter** **import** RedisFilter

*# filter = MemoryFilter()*

filter = RedisFilter(redis\_host="172.17.0.2")

data = ["111", "qwe", '222', "333" ,"111", "qwe", "中文", "qwer"]

**for** d **in** data:

**if** filter.is\_exists(d):

**print**("发现重复的数据: ", d)

**else**:

filter.save(d)

**print**("保存去重的数据：", d)

运行info\_filter\_demo.py, 就会看到去重的结果. 再次运行, 因为redis服务器是一直运行的, 所有的数据都会变成重复数据. 如果redis容器没有使用数据卷, 数据是保存在容器中的, 如果删除容器, 再次创建并运行容器, 原来保存的数据就会丢失, 所以再次运行时就会重新操作. 所以想要redis进行持久化的存储, 一定要在创建redis时绑定对应的数据卷.

#### 基于mysql去重的代码实现

新建information\_summary\_filter/mysql\_filter.py

*# 基于mysql的去重判断依据的存储*

*# 这里使用的sqlalchemy关系型数据的ORM来连接和操作mysql数据库的, 原因有二, 1. 如果使用pymysql或mysqldb来连接数据库, 就要结合原生的sql语句. 这时用户的精力可能就会转向sql语句的书写和优化, 而不是去重代码的实现了. 2. pymysql和mysqlbdb对数据库的连接是单个的长连接, 如果整个程序运行的时间比较长, 建立的长连接没有使用的话, mysql数据库可能会强制关闭建立的连接, 此时再进行数据库的操作时就会出现异常. 而使用ORM连接mysql数据库则会自动维护连接池.*

**from** **sqlalchemy** **import** create\_engine, Column, Integer, String

**from** **sqlalchemy.orm** **import** sessionmaker

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

**from** **.** **import** BaseFilter

*# 创建Base对象*

Base = declarative\_base()

*# 定义mysql数据表结构*

**class** **Filter**(Base):

*# 定义数据表的表名*

\_\_tablename\_\_ = "filter"

*# 定义数据表的字段, 根据实际中的需要添加字段*

id = Column(Integer, primary\_key=True)

hash\_value = Column(String(40), index=True, unique=True)

**class** **MySQLFilter**(BaseFilter):

*'''基于mysql的去重判断依据的存储'''*

**def** \_get\_storage(self):

*'''返回一个mysql连接对象(sqlalchemy的数据库连接对象)'''*

*# 使用create引擎, 传入mysql的连接信息, 得到连接对象engine, 可以把engine当成是mysql的客户端来使用. 修改修改information\_summary\_filter/\_\_init\_\_.py, 添加self.mysql\_url的参数*

engine = create\_engine(self.mysql\_url)

*# 创建mysql数据表, 如果存在就忽略*

Base.metadata.create\_all(engine)

*# save操作是基于session对话的形式来完成的, 所以\_get\_storage返回一个Session的类*

Session = sessionmaker(engine)

**return** Session

**def** \_save(self, hash\_value):

*'''*

*利用redis的无序集合进行存储*

*:param hash\_value:*

*:return:*

*'''*

*# 实例化session对象. self.storage()调用self.\_get\_storage(), 最终返回的是Session类*

session = self.storage()

*# 实例化mysql数据表, 保存数据到数据表中, 主键id会自动进行设置, 所以此时不用传入*

filter = self.Filter(hash\_value=hash\_value)

*# 使用session.add方法添加一个对象*

session.add(filter)

*# session提交*

session.commit()

session.close()

**def** \_is\_exists(self, hash\_value):

*'''判断redis对应的无序集合中是否有对应的判断依据'''*

session = self.storage()

ret = session.query(Filter).filter\_by(hash\_value=hash\_value).first()

session.close()

*# 如果找到了, 就返回True, 否则返回None*

**if** ret **is** None:

**return** False

**return** True

修改information\_summary\_filter/\_\_init\_\_.py, 定义mysql的初始化设置

**class** **BaseFilter**(object):

*'''基于信息摘要算法进行数据的去重判断和存储'''*

**def** \_\_init\_\_(self,

hash\_func\_name="md5",

redis\_host="localhost",

redis\_port=6379,

redis\_db=0,

redis\_key="filter",

mysql\_url=None,

mysql\_table\_name="filter"):

*# 注意, 因为在子类的\_get\_storage()方法中调用了redis, mysql的连接参数, 所以连接参数都要放在\_get\_storage()方法的前面, 否则就会出错.*

self.redis\_host = redis\_host

self.redis\_port = redis\_port

self.redis\_db = redis\_db

*# redis是key-value的数据库, 想要向集合中保存数据, 必须要指定对应的key, 在实例化时可以自定义redis\_key*

self.redis\_key = redis\_key

self.mysql\_url = mysql\_url

*# 从hashlib模块中获取md5属性*

self.hash\_func = getattr(hashlib, hash\_func\_name)

*# 每一种去重方法都需要不同的数据结构来实现, 内存去重使用set集合来实现, mysql对应mysql的连接, redis对应redis的连接*

*# 这里如果使用None, 那么在每个子类中都要重写\_\_init\_\_方法, 上面的所有参数都要进行重写, 所以最好是定义一个方法, 在子类中重写同名的方法即可.*

*# self.storage = None*

self.storage = self.\_get\_storage()

mysql数据库也是使用docker来创建的, 在pycharm中基于mysql:5.7来创建mysql的容器, 查看github的docker library中的mysql的说明信息. 可以在创建容器时设置mysql的参数.

<https://github.com/docker-library/docs/tree/master/mysql#environment-variables>

在pycharm中的创建容器的对话框中选择选择 "Environment variables" 中进行设置,

MYSQL\_ROOT\_PASSWORD: password

MYSQL\_DATABASE: data

查看创建的mysql容器的ip地址, 为172.17.0.4

修改info\_filter\_demo.py, 对mysql\_filter进行测试.

*# python3*

**from** **information\_summary\_filter** **import** MemoryFilter

**from** **information\_summary\_filter** **import** RedisFilter

**from** **information\_summary\_filter** **import** MySQLFilter

*# filter = MemoryFilter()*

*# filter = RedisFilter(redis\_host="172.17.0.2")*

*# sqlalchemy ORM使用的是pymysql与mysql数据库进行的连接, 需要定义mysql\_url的变量*

*# mysql\_url = "mysql+pymysql://root:password@localhost:3306/db\_name?charset=utf8"*

mysql\_url = "mysql+pymysql://root:password@172.17.0.4:3306/data?charset=utf8"

filter = MySQLFilter(mysql\_url=mysql\_url)

data = ["111", "qwe", '222', "333" ,"111", "qwe", "中文", "qwer"]

**for** d **in** data:

**if** filter.is\_exists(d):

**print**("发现重复的数据: ", d)

**else**:

filter.save(d)

**print**("保存去重的数据：", d)

运行info\_filter\_demo.py文件, 就能正常进行去重和数据的保存了.

**在爬虫中, 对于请求的去重, 更多的是使用redis, 而对于数据的去重, 更多的是使用mysql**

##### 对程序的导入方法进行改进

在使用3种去重过滤器时, 直接从information\_summary\_filter这个包中导入3种去重过滤器. 即在info\_filter\_demo.py中, 使用 from information\_summary\_filter import MemoryFilter, RedisFilter, MysqlFilter.

修改information\_summary\_filter/\_\_init\_\_.py, 在最后面添加如下3条语句.

*# 预先导入3个子模块. 由于在这3个子模块中使用了BaseFilter, 所以要放在最后面*

**from** **.mysql\_filter** **import** MySQLFilter

**from** **.memory\_filter** **import** MemoryFilter

**from** **.redis\_filter** **import** RedisFilter

修改info\_filter\_demo.py, 把导包的方式修改为

**from** **information\_summary\_filter** **import** MemoryFilter

**from** **information\_summary\_filter** **import** RedisFilter

**from** **information\_summary\_filter** **import** MySQLFilter

##### 对mysql\_filter代码进行改进

在mysql\_filter.py中的Filter类中定义了进行去重的表名为 "filter", 此时用户不能动态的修改使用的数据表的表名, 如果有多个网站的爬虫, 想要对每个网站的爬虫使用不同的数据表进行去重, 此时需要基于MySQLFilter创建多个对象, 但这些对象全部都使用 filter 这个数据表来进行去重, 就不能达到对不同的爬虫使用不同的数据表进行去重的目的.

修改information\_summary\_filter/\_\_init\_\_.py, 添加初始化的mysql\_table\_name字段.

**class** **BaseFilter**(object):

*'''基于信息摘要算法进行数据的去重判断和存储'''*

**def** \_\_init\_\_(self,

hash\_func\_name="md5",

redis\_host="localhost",

redis\_port=6379,

redis\_db=0,

redis\_key="filter",

mysql\_url=None,

mysql\_table\_name="filter"):

*# 注意, 因为在子类的\_get\_storage()方法中调用了redis, mysql的连接参数, 所以连接参数都要放在\_get\_storage()方法的前面, 否则就会出错.*

self.redis\_host = redis\_host

self.redis\_port = redis\_port

self.redis\_db = redis\_db

*# redis是key-value的数据库, 想要向集合中保存数据, 必须要指定对应的key, 在实例化时可以自定义redis\_key*

self.redis\_key = redis\_key

self.mysql\_url = mysql\_url

*# 设置去重时使用的mysql的数据表*

self.mysql\_table\_name = mysql\_table\_name

*# 从hashlib模块中获取md5属性*

self.hash\_func = getattr(hashlib, hash\_func\_name)

*# 每一种去重方法都需要不同的数据结构来实现, 内存去重使用set集合来实现, mysql对应mysql的连接, redis对应redis的连接*

*# 这里如果使用None, 那么在每个子类中都要重写\_\_init\_\_方法, 上面的所有参数都要进行重写, 所以最好是定义一个方法, 在子类中重写同名的方法即可.*

*# self.storage = None*

self.storage = self.\_get\_storage()

修改information\_summary\_filter/mysql\_filter.py, 添加mysql\_table\_name的代码.

*# 基于mysql的去重判断依据的存储*

*# 这里是使用ORM来连接mysql数据库的, 如果使用pymysql或mysqldb来连接数据库, 就要结合原生的sql语句. pymysql和mysqlbdb对数据库的连接是单个的长连接, 如果整个程序运行的时间比较长, 建立的长连接没有使用的话, mysql数据库可能会强制关闭建立的连接, 此时再进行数据库的操作时就会出现异常. 而使用ORM连接mysql数据库则会自动维护连接池.*

**from** **sqlalchemy** **import** create\_engine, Column, Integer, String

**from** **sqlalchemy.orm** **import** sessionmaker

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

**from** **.** **import** BaseFilter

Base = declarative\_base()

*# # 建立mysql数据表*

*# class Filter(Base):*

*# \_\_tablename\_\_ = "filter"*

*# id = Column(Integer, primary\_key=True)*

*# hash\_value = Column(String(40), index=True, unique=True)*

**class** **MySQLFilter**(BaseFilter):

*'''基于mysql的去重判断依据的存储'''*

*# 重写父类BaseFilter的\_\_init\_\_方法, 定义使用的数据表*

**def** \_\_init\_\_(self, \*args, \*\*kwargs):

*# # 因为在information\_summary\_filter/\_\_init\_\_.py中定义了mysql\_table\_name的变量, 实例化MySQLFilter对象时如果传入了这个变量, 想要使用这个变量来创建对应的mysql数据表, 就要把类放在\_\_init\_\_中进行创建, 但不符合python的习惯. 可以使用type来动态的创建类.*

*# class Filter(Base):*

*# \_\_tablename\_\_ = kwargs["mysql\_table\_name"]*

*# id = Column(Integer, primary\_key=True)*

*# hash\_value = Column(String(40), index=True, unique=True)*

*# # 创建数据表*

*# self.table = Filter()*

*# 使用type来动态的创建类, type(创建的类的类名, 元组形式的继承的父类, 包含属性/方法的字典(名称和值)), 只要在类进行初始化时传入了 "mysql\_table\_name" 这个参数, 就会动态的创建对应的同名的类和数据表, 从而达到代码复用的目的.*

Filter = type(

kwargs["mysql\_table\_name"],

(Base,),

dict(

\_\_tablename\_\_=kwargs["mysql\_table\_name"],

id = Column(Integer, primary\_key=True),

hash\_value = Column(String(40), index=True, unique=True)

)

)

*# 创建数据表*

self.table = Filter()

BaseFilter.\_\_init\_\_(self, \*args, \*\*kwargs)

**def** \_get\_storage(self):

*'''返回一个mysql连接对象(sqlalchemy的数据库连接对象)'''*

*# 可以把engine当成是mysql的客户端来使用*

engine = create\_engine(self.mysql\_url)

Base.metadata.create\_all(engine) *# 创建表、如果存在就忽略*

*# save操作是基于session对话的形式来完成的, 所以\_get\_storage返回一个Session的类*

Session = sessionmaker(engine)

**return** Session

**def** \_save(self, hash\_value):

*'''*

*利用redis的无序集合进行存储*

*:param hash\_value:*

*:return:*

*'''*

*# 实例化session对象*

session = self.storage()

*# 主键id会自动进行设置, 所以此时不用传入*

*# filter = self.Filter(hash\_value=hash\_value)*

filter = self.table(hash\_value=hash\_value)

*# 使用session.add方法添加一个对象*

session.add(filter)

*# session提交*

session.commit()

session.close()

**def** \_is\_exists(self, hash\_value):

*'''判断redis对应的无序集合中是否有对应的判断依据'''*

session = self.storage()

*# ret = session.query(Filter).filter\_by(hash\_value=hash\_value).first()*

ret = session.query(self.table).filter\_by(hash\_value=hash\_value).first()

session.close()

*# 如果找到了, 就返回True, 否则返回None*

**if** ret **is** None:

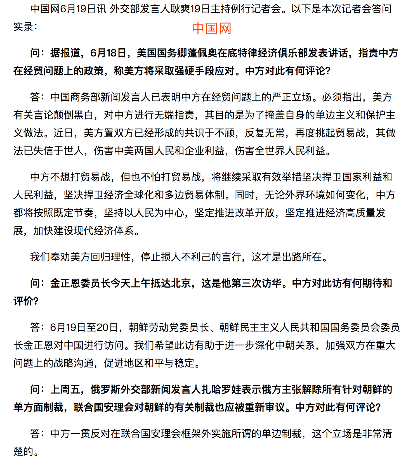
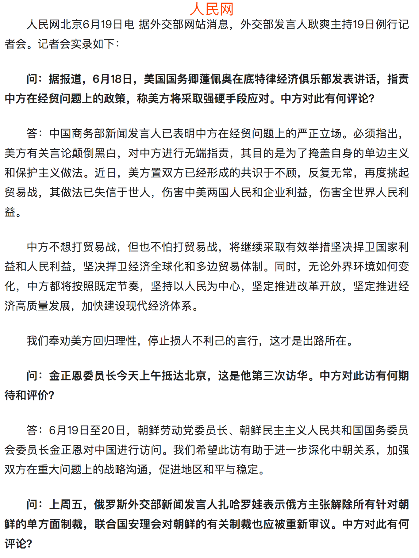
**return** False

**return** True

## 3. 基于simhash算法的去重

### 3.1 Simhash介绍以及应用场景

* Simhash算法是一种局部敏感哈希算法，能实现**相似**文本内容的去重。比如下列两篇新闻数据

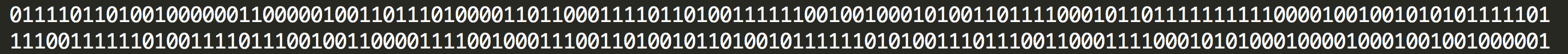


### 3.2 Simhash的特征

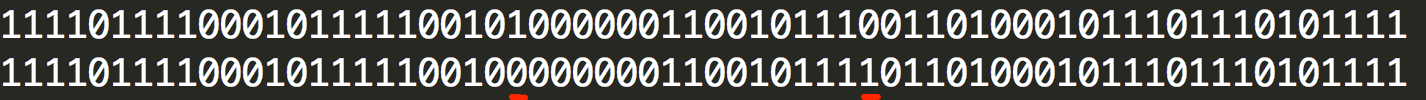
* 与信息摘要算法的区别：
  + 信息摘要算法：如果原始内容只相差一个字节，所产生的签名也很可能差别很大。 ==
  + Simhash算法：如果原始内容只相差一个字节，所产生的签名差别非常小。
* Simhash值比对：通过两者的simhash值的二进制位的差异来表示原始文本内容的差异。差异个数又被称为海明距离。
* 注意：
  + Simhash对长文本500字+比较适用，短文本可能偏差较大
  + 在google的论文给出的数据中，64位simhash值，在海明距离为3的情况下，可认为两篇文档是相似的或者是重复的。当然这个值只是参考值，针对自己的应用可能有不同的测试取值

### 3.3 Simhash值的比对

* Python实现的[simhash](https://github.com/leonsim/simhash)算法。该模块得出的simhash值长度正是64位
* 如对比前面列出的人民网和中国网的两篇相似新闻（以下值仅供参考）
* 128位MD5值：



* 64位simhash值



simhash算法不是对比值是否完全相等, 而是逐位的比较, 值相等的表示原文本相同, 值不等的表示原文本不同. 在不同的个数小于等于3的情况下, 可以认为二者是相似文本或重复文本.

### 3.4 Simhash模块的使用

<https://github.com/leonsim/simhash>

pip install simhash

**import** **re**

**from** **simhash** **import** Simhash, SimhashIndex

**def** get\_features(s):

width = 3

*# 全部转换为小写字母*

s = s.lower()

*# 把非字母数字的字符转换为空白字符, 相当于去掉空白字符和标点符号*

s = re.sub(r'[^\w]+', '', s)

*# 分词操作, 根据设置的长度width对目标字符串进行分词操作*

**return** [s[i:i + width] **for** i **in** range(max(len(s) - width + 1, 1))]

*# Simhash需要一个列表格式的参数*

**print**('**%x**' % Simhash(get\_features('How are you? I am fine. Thanks.')).value)

**print**('**%x**' % Simhash(get\_features('How are u? I am fine. Thanks.')).value)

**print**('**%x**' % Simhash(get\_features('How r you?I am fine. Thanks.')).value)

*# 求文本之间的海明距离*

*# 如果是传入的字符串, 就把字符串作为一个整体进行对比, 如果传入的是列表, 就把列表中的每一个元素作为一部分进行对比.*

**from** **simhash** **import** Simhash

**print**(Simhash('aa').distance(Simhash('bb')))

*# 30*

**print**(Simhash('aa').distance(Simhash('aa')))

*# 0*

**print**(Simhash(['aa','a']).distance(Simhash(['aa','b'])))

*# 10*

**print**(Simhash('aaa').distance(Simhash('aab')))

*# 30*

*# 计算simhash索引*

*# Use the Simhash Index*

*# 可以使用simhash索引来查询最近邻的重复的对象*

*# Use the SimhashIndex to query near duplicates objects in a very efficient way.*

data = {

"1": u'How are you? I Am fine. blar blar blar blar blar Thanks.',

"2": u'How are you i am fine. blar blar blar blar blar than',

"3": u'This is simhash test.',

*# index.add('4', s1)的作用相当于把s1中的文本添加到了4的位置.*

*# "4": u'How are you i am fine. blar blar blar blar blar thank'*

}

data = {

*# 把键由原来的1,2,3 修改为 key1, key2, key3, 得到的结果是key1, key2, key3, 也就是得到原数据中的哪个键对应的值与s1的内容比较接近*

"key1": u'How are you? I Am fine. blar blar blar blar blar Thanks.',

"key2": u'How are you i am fine. blar blar blar blar blar than',

"key3": u'This is simhash test.',

*# "4": u'How are you i am fine. blar blar blar blar blar thank'*

}

objs = [

(str(k), Simhash(get\_features(v))) **for** k, v **in** data.items()

]

**print**(objs)

index = SimhashIndex(objs, k=4) *# k相当于海明距离*

**print**(index.bucket\_size()) *# 11, 海明距离越大, 重复的对象越多*

s1 = Simhash(get\_features(u'How are you i am fine. blar blar blar blar blar thank'))

*# 在index中保存着data中4个值对应的simhash值, 用index中保存的值与s1进行比对, 看s1与index中的哪个比较接近. 得到的结果是1*

**print**(index.get\_near\_dups(s1)) *# ['1'] ['key1']*

*# 把s1添加到index中的4位置, 然后再求出哪两个相同. 得到的结果是1和4.*

index.add('4', s1)

**print**(index.get\_near\_dups(s1)) *# ['4', '1'] ['key1']*

*# simhash的比对是基于二进制位的比对, 在比对时必须要已经具有全部的基础上才能得到较好的结果.*

*# 比对一般只能在内存中进行, 想要实现持久化保存, 需要把index进行序列化, 再保存到硬盘中.*

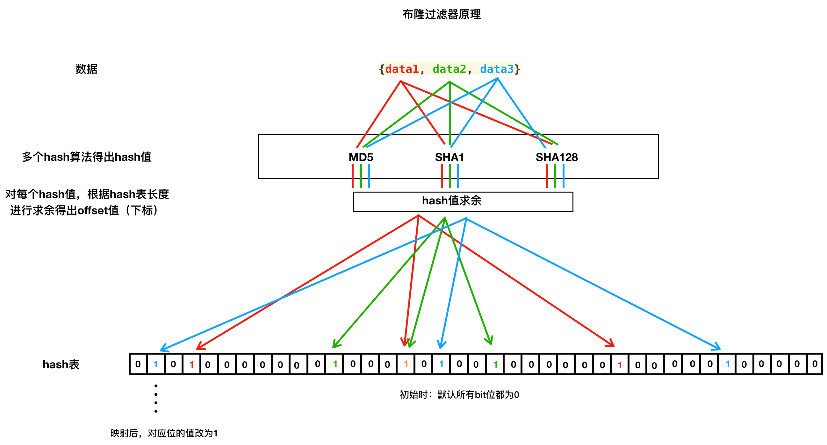
*# 序列化工具： 将一个对象转换为二进制的一个数据 pickle序列化工具, pickle.dumps, pickle.loads*

*# 反序列化：二进制--> 对象*

## 4. 布隆过滤器原理与实现

### 4.1 布隆过滤器（bloomfilter）原理

海量数据的去重. 信息摘要算法得到的结果是16字节, 布隆过滤器得到的结果是3比特位到1字节之间.



条件1: 多个(>=3)hash算法求出hash值, 如信息摘要算法, simhash算法等, 由原始的数据, 根据一定的算法, 得到一个唯一的特征值.

条件2: hash表. 对每个hash值, 根据hash表长度进行求余得到offset值(下标)

hash表是一个有序的表, 类似于python中的列表, 在使用hash表时要先确定它的长度, hash表中只能存放1, 0这两种数据, 相当于hash表的每个元素都是一个比特位, 占用的空间都是一个比特位.

1M大小的hash表, 1M=1024Kb = 210 Kb = 220 b = 8 \* 220 bit位

256M大小的hash表, 256 \* 1024 \*1024 \* 8 = 21 4748 3648

from hashlib import md5

# 求md5值

m5 = md5(b"a").hexdigest()

# 对hash表的长度求余, 任何值对hash表的长度求余得到的结果必定小于hash表的长度, 也就是在0-2147483648之间

int(m5, 16) % (256 \*1024 \* 1024 \* 8)

# hash表初始值全部为0, 把某个对象对hash表长度求余的结果作为下标, 把hash表中此处的值设置为1, 有多个hash函数, 就会进行多次映射. 把hash表多个位上的值设置为1. 不同的数据可能会对hash表中的同一个位置进行映射, 但由于设置的值都是1, 所以不同数据的映射之间不会影响. hash映射的过程就相当于是对判断依据的保存.

条件3: 判断数据是否重复. 判断时, 只读取数据, 不更改数据. 只要多个位上同时是1, 就表示数据已经存在. 有一定的机率会误判.

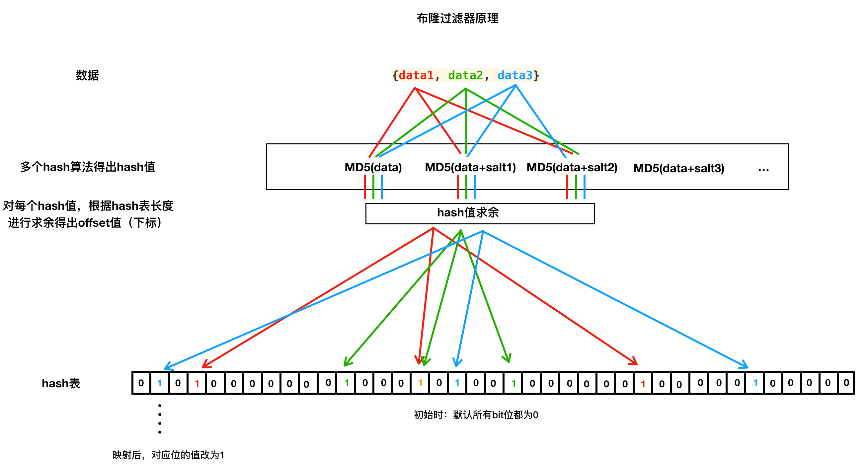
### 4.2 布隆过滤器（bloomfilter）原理 \*

减小误判率的方法:

1. 增加hash算法的个数, 误判率就会降低. 3-10个之间.

2. 增加hash表的长度.

3. 如果全部使用python内置的hash算法, 可能不能满足hash算法个数的要求, 可以使用某一种hash算法, 但计算多次, 每次计算都预先对数据进行一定的处理, 如加上固定的值. 这样也能每次得到不同的hash值, 相当于使用不同的hash算法.



### 4.3 布隆过滤器实现

* Python实现的内存版布隆过滤器[pybloom](https://github.com/jaybaird/python-bloomfilter)

<https://github.com/jaybaird/python-bloomfilter>

* 手动实现的redis版布隆过滤器

*# 布隆过滤器 redis版本实现*

**import** **hashlib**

**import** **redis**

**import** **six**

*# 1. 多个hash函数的实现和求值*

*# 2. hash表实现和实现对应的映射和判断*

**class** **MultipleHash**(object):

*"""根据提供的原始数据，和预定义的多个salt，生成多个hash函数值"""*

**def** \_\_init\_\_(self, salts, hash\_func\_name="md5"):

*# getattr(x,'y')等价于x.y*

self.hash\_func = getattr(hashlib, hash\_func\_name)

*# salts类似于['1','2','3','4']的结构, 同一个项目中salts的值不能修改, 对所有的数据, 对映射和判断都要使用相同的salts*

**if** len(salts) < 3:

**raise** **Exception**("请至少提供3个salt")

self.salts = salts

**def** get\_hash\_values(self, data):

*"""根据提供的原始数据, 返回多个hash函数值"""*

hash\_values = []

*# 遍历salts,取出来每个salt, 与原始数据进行拼接, 生成新的字符串*

**for** salt **in** self.salts:

*# 调用self.hash\_func(), 创建出hash\_obj*

hash\_obj = self.hash\_func()

*# 对每个salt,都更新hash\_obj*

hash\_obj.update(self.\_safe\_data(data))

hash\_obj.update(self.\_safe\_data(salt))

*# 求出来hash值*

ret = hash\_obj.hexdigest()

*# 以10进制的形式添加到hash\_values的列表中*

hash\_values.append(int(ret, 16))

**return** hash\_values

**def** \_safe\_data(self, data):

*"""*

*python2 str === python3 bytes*

*python2 uniocde === python3 str*

*:param data: 给定的原始数据*

*:return: 二进制类型的字符串数据*

*"""*

**if** six.PY3:

**if** isinstance(data, bytes):

**return** data

**elif** isinstance(data, str):

**return** data.encode()

**else**:

**raise** **Exception**("请提供一个字符串") *# 建议使用英文来描述*

**else**:

**if** isinstance(data, str):

**return** data

**elif** isinstance(data, unicode):

**return** data.encode()

**else**:

**raise** **Exception**("请提供一个字符串") *# 建议使用英文来描述*

**class** **BloomFilter**(object):

*"""*

*功能: 基于redis实现BloomFilter的hash表*

*方法: 保存/映射和判断hash值*

*http://redis.cn/commands.html, 搜索bit, 得到所有与bit相当的操作.*

*SETBIT key offset value 设置bit位的值, GETBIT key offset 获取某个bit位的值.*

*hash表使用前需要设定长度, 由于redis中bit的数据结构是基于Strings的数据类型的, 一个Strings最多可以使用512M的内存, 所以bit数据结构也最多可以使用512M的内存, 对应于42亿左右长度的hash表.*

*代码中需要设置hash表的长度, 如256M对应的hash表长度.*

*"""*

*# 数据是基于redis进行操作的, 所以要实例化redis的参数*

**def** \_\_init\_\_(self, salts, redis\_host="localhost", redis\_port=6379, redis\_db=0, redis\_key="bloomfilter"):

self.redis\_host = redis\_host

self.redis\_port = redis\_port

self.redis\_db = redis\_db

self.redis\_key = redis\_key

self.client = self.\_get\_redis\_client()

self.multiple\_hash = MultipleHash(salts)

**def** \_get\_redis\_client(self):

*"""返回一个redis连接对象"""*

pool = redis.ConnectionPool(host=self.redis\_host, port=self.redis\_port, db=self.redis\_db)

client = redis.StrictRedis(connection\_pool=pool)

**return** client

**def** save(self, data):

*"""把数据的hash值映射到hash表"""*

*# 返回列表形式的多个hash值.*

hash\_values = self.multiple\_hash.get\_hash\_values(data)

*# 遍历hash值列表, 求出偏移量*

**for** hash\_value **in** hash\_values:

offset = self.\_get\_offset(hash\_value)

*# 使用redis的setbit方法向redis中保存bit类型的值. setbit(key,offset,value)*

*# 这里value的值只能是1*

self.client.setbit(self.redis\_key, offset, 1)

*# 保存成功时返回True*

**return** True

**def** is\_exists(self, data):

*"""判断数据的hash值在hash表中是否存在"""*

hash\_values = self.multiple\_hash.get\_hash\_values(data)

**for** hash\_value **in** hash\_values:

offset = self.\_get\_offset(hash\_value)

*# 使用getbit来获取对应位置的值, 如果是0, 表示数据不存在, 如果是1, 表示存在.*

v = self.client.getbit(self.redis\_key, offset)

*# 只要有一个0, 就表示不存在, 就可以返回了*

**if** v == 0:

**return** False

*# 当循环执行结束后, 还是没有退出, 表示for循环中的v没有得到0, 全部是1, 就表示值是存在的.*

**return** True

**def** \_get\_offset(self, hash\_value):

*"""根据hash值计算偏移量"""*

*# 2\*\*8 = 256 # 256M的bit位作为hash表的长度*

*# 2\*\*20 = 1024 \* 1024*

*# 2\*\*31 = 256M, 大小不能超过2\*\*32, 也就是占用的空间不能超过512M. redis中每一个key对应的值所占的空间最大为 512M*

*# 256 \** 1024 *\* 1024 \* 8 = (2\*\*8 \* 2\*\*20 \* 2\*\*3) 代表hash表的长度 同一项目中不能更改*

**return** hash\_value % (2\*\*8 \* 2\*\*20 \* 2\*\*3)

**if** \_\_name\_\_ == '\_\_main\_\_':

*# 对多个hash函数进行测试*

*# mh = MultipleHash(["1","2","3","4"])*

*# print(mh.get\_hash\_values("adbdgadfasfasdfjilasdfkasdf"))*

*# 对redis版布隆过滤器进行测试*

data = ["asdfasdf", "123", "123", "456","asf", "asf"]

bm = BloomFilter(salts=["1","2","3","4"],redis\_host="172.17.0.2")

**for** d **in** data:

**if** **not** bm.is\_exists(d):

bm.save(d)

**print**("映射数据成功： ", d)

**else**:

**print**("发现重复数据：", d)

基于信息摘要算法的3种过滤器与布隆过滤器的封装

在项目根目录中新建filter\_class的python程序包, 把information\_summary\_filter和bloomfilter.py都拖动到filter\_class这个程序包中. 在使用时, 直接从filter\_class这个包中导入4种过滤器即可.

修改filter\_class/\_\_init\_\_.py, 定义函数作为4种过滤器的统一接口.

**def** get\_filter\_class(cls\_name):

*'''根据传入的 cls\_name 参数返回对应的过滤器的类对象'''*

**if** cls\_name == "bloom":

**from** **.bloomfilter** **import** BloomFilter

**return** BloomFilter

**elif** cls\_name == "memory":

**from** **.information\_summary\_filter** **import** MemoryFilter

**return** MemoryFilter

**elif** cls\_name == "mysql":

**from** **.information\_summary\_filter** **import** MySQLFilter

**return** MySQLFilter

**elif** cls\_name == "redis":

**from** **.information\_summary\_filter** **import** RedisFilter

**return** RedisFilter

在使用时, 从filter\_class中导入get\_filter\_class方法就可以了.

from filter\_class import get\_filter\_class

get\_filter\_class("bloom")

在information\_summary\_filter的过滤器中, 在使用save方法保存指纹之后, 有可能会需要获取到保存的指纹的信息, 所以可以使用return返回hash\_value的值.

修改information\_summary\_filter/\_\_init\_\_.py

**def** save(self, data):

*'''*

*根据data计算出对应的指纹进行存储*

*:param data: 给定的原始数据, 数据类型为二进制类型的字符串数据*

*:return: 存储的结果, 存储成功还是失败*

*'''*

*# 计算hash value*

hash\_value = self.\_get\_hash\_value(data)

*# 因为3种方法具体保存时的逻辑是不同的, 所以可以定义一个新的方法\_save, 子类在继承时重写\_save方法即可.*

self.\_save(hash\_value)

**return** hash\_value

有时候会需要获取bloomfilter中数据的偏移量offset的值, 修改bloomfilter.py, 在save方法中返回offset的值.

**def** save(self, data):

*"""将原始数据在hash表中一一映射, 并返回对应的偏移量."""*

*# 返回列表形式的多个hash值.*

hash\_values = self.multiple\_hash.get\_hash\_values(data)

offsets = []

*# 遍历hash值列表, 求出偏移量*

**for** hash\_value **in** hash\_values:

offset = self.\_get\_offset(hash\_value)

offsets.append(offset)

*# 使用redis的setbit方法向redis中保存bit类型的值. setbit(key,offset,value)*

*# 这里value的值只能是1*

self.client.setbit(self.redis\_key, offset, 1)

*# 保存成功时返回True*

**return** offsets

## 4. 总结