redis内容扩展

1.Pipeline

注意：使用Pipeline的操作是非原子操作

2.GEO

GEOADD locations 116.419217 39.921133 beijin

GEOPOS locations beijin

GEODIST locations tianjin beijin km 计算距离

GEORADIUSBYMEMBER locations beijin 150 km 通过距离计算城市

注意：没有删除命令 它的本质是zset （type locations）

所以可以使用zrem key member 删除元素

zrange key 0 -1 表示所有 返回指定集合中所有value

3.hyperLogLog

Redis 在 2.8.9 版本添加了 HyperLogLog 结构。

Redis HyperLogLog 是用来做基数统计的算法，HyperLogLog 的优点是，在输入元素的数量或者体积非常非常大时，计算基数所需的空间总是固定 的、并且是很小的

在 Redis 里面，每个 HyperLogLog 键只需要花费 12 KB 内存，就可以计算接近 2^64 个不同元素的基 数。这和计算基数时，元素越多耗费内存就越多的集合形成鲜明对比。

PFADD 2017_03_06:taibai 'yes' 'yes' 'yes' 'yes' 'no'

PFCOUNT 2017_03_06:taibai 统计有多少不同的值

1.PFADD 2017_09_08:taibai uuid9 uuid10 uu11

2.PFMERGE 2016_03_06:taibai 2017_09_08:taibai 合并

注意：本质还是字符串 ， 有容错率，官方数据是0.81%

4.bitmaps

setbit taibai 500000 0

getbit taibai 500000

bitcount taibai

Bitmap本质是string，是一串连续的2进制数字（0或1），每一位所在的位置为偏移(offset)。 string（Bitmap）最大长度是512 MB，所以它们可以表示2 ^ 32=4294967296个不同的位。

缓存几大问题

1.缓存粒度控制

通俗来讲，缓存粒度问题就是我们在使用缓存时，是将所有数据缓存还是缓存部分数据?

数据类型	通用性	空间占用（内存空间+网络码率）	代码维护
全部数据	高	大	简单
部分数据	低	小	较为复杂

缓存粒度问题是一个容易被忽视的问题，如果使用不当，可能会造成很多无用空间的浪费，可能会造成网络带宽的浪费，可能会造成代码通用性较差等情况，必须学会综合数据通用性、空间占用比、代码维护性 三点评估取舍因素权衡使用。

2.缓存穿透问题

缓存穿透是指查询一个一定不存在的数据，由于缓存不命中，并且出于容错考虑， 如果从存储层查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到存储层去查询，失去了缓存的意义。

可能造成原因:

1.业务代码自身问题 2.恶意攻击。爬虫等等

危害

对底层数据源压力过大，有些底层数据源不具备高并发性。 例如mysql一般来说单台能够扛1000-QPS就已经很不错了

解决方案

1.缓存空对象

```
public class NullValueResultDO implements Serializable{
     private static final long serialVersionUID = -6550539547145486005L;
}

public class UserManager {
     UserDAO userDAO;
     LocalCache localCache;

     public UserDO getUser(String userNick) {
          Object object = localCache.get(userNick);
          if(object != null) {
               if(object instanceof NullValueResultDO) {
                    return null;
               }

               return (UserDO)object;
```

```
            } else {
                User user = userDAO.getUser(userNick);
                if(user != null) {
                    localCache.put(userNick,user);
                } else {
                    localCache.put(userNick, new NullValueResultDO());
                }
                return user;
            }
        }
}
```

2.布隆过滤器

3.缓存击穿.热点key重建缓存问题

缓存击穿是指缓存中没有但数据库中有的数据（一般是缓存时间到期），这时由于并发用户特别多，同时读缓存没读到数据，又同时去数据库去取数据，引起数据库压力瞬间增大，造成过大压力

我们知道，使用缓存，如果获取不到，才会去数据库里获取。但是如果是热点 key，访问量非常的大，数据库在重建缓存的时候，会出现很多线程同时重建的情况。因为高并发导致的大量热点的 key 在重建还没完成的时候，不断被重建缓存的过程，由于大量线程都去做重建缓存工作，导致服务器拖慢的情况。

解决方案

1.互斥锁

第一次获取缓存的时候，加一个锁，然后查询数据库，接着是重建缓存。这个时候，另外一个请求又过来获取缓存，发现有个锁，这个时候就去等待，之后都是一次等待的过程，直到重建完成以后，锁解除后再次获取缓存命中。

```
public String getKey(String key){
    String value = redis.get(key);
    if(value == null){
        String mutexKey = "mutex:key:"+key; //设置互斥锁的key
        if(redis.set(mutexKey,"1","ex 180","nx")){ //给这个key上一把锁，ex表示只有一个线程能执行，过期时间为180秒
          value = db.get(key);
          redis.set(key,value);
          redis.delete(mutexKety);
  }else{
        // 其他的线程休息100毫秒后重试
        Thread.sleep(100);
        getKey(key);
  }
 }
 return value;
}
```

互斥锁的优点是思路非常简单，具有一致性，但是互斥锁也有一定的问题，就是大量线程在等待的问题。存在死锁的可能性。

4.缓存雪崩问题

缓存雪崩是指机器宕机或在我们设置缓存时采用了相同的过期时间，导致缓存在某一时刻同时失效，请求全部转发到DB，DB瞬时压力过重雪崩。

1：在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个key只允许一个线程查询数据和写缓存，其他线程等待。

2：做二级缓存，A1为原始缓存，A2为拷贝缓存，A1失效时，可以访问A2，A1缓存失效时间设置为短期，A2设置为长期

3：不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀。

4：如果缓存数据库是分布式部署，将热点数据均匀分布在不同搞得缓存数据库中。