1、缓存问题解决方案

1.1 缓存穿透

缓存没有，数据库也没有，业务系统访问压根就不存在的数据，导致每次访问都将压力挂到了数据库服务器上导致服务崩溃，一般来说都是恶意访问导致

解决方案：

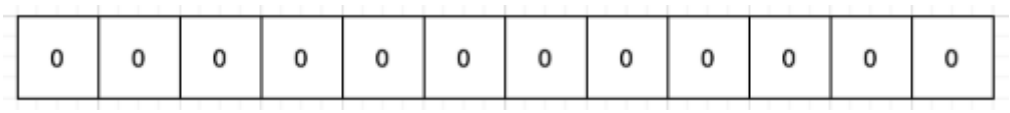
1、缓存空数据

第一，空值做了缓存，意味着缓存层中存了更多的键，需要更多的内存空间 (如果是攻击，问题更严重)，比较有效的方法是针对这类数据设置一个较短的过期时间，让其自动剔除。
第二，缓存层和存储层的数据会有一段时间窗口的不一致，可能会对业务有一定影响。例如过期时间设置为 5 分钟，如果此时存储层添加了这个数据，那此段时间就会出现缓存层和存储层数据的不一致，此时可以利用消息系统或者其他方式清除掉缓存层中的空对象。

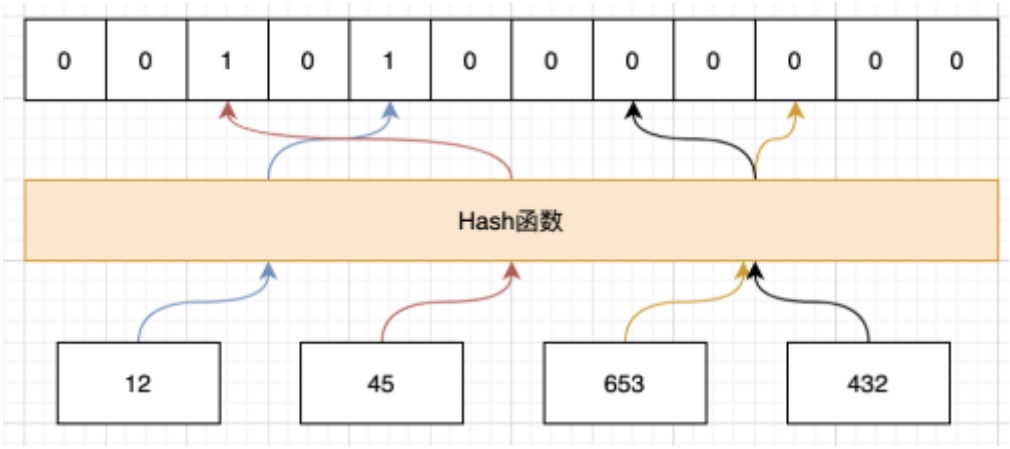
2、BloomFilter（布隆过滤器）：在缓存之前再加一道屏障，里面存储目前redis数据库中存在的所有key

在讲布隆过滤器之前我们思考下：

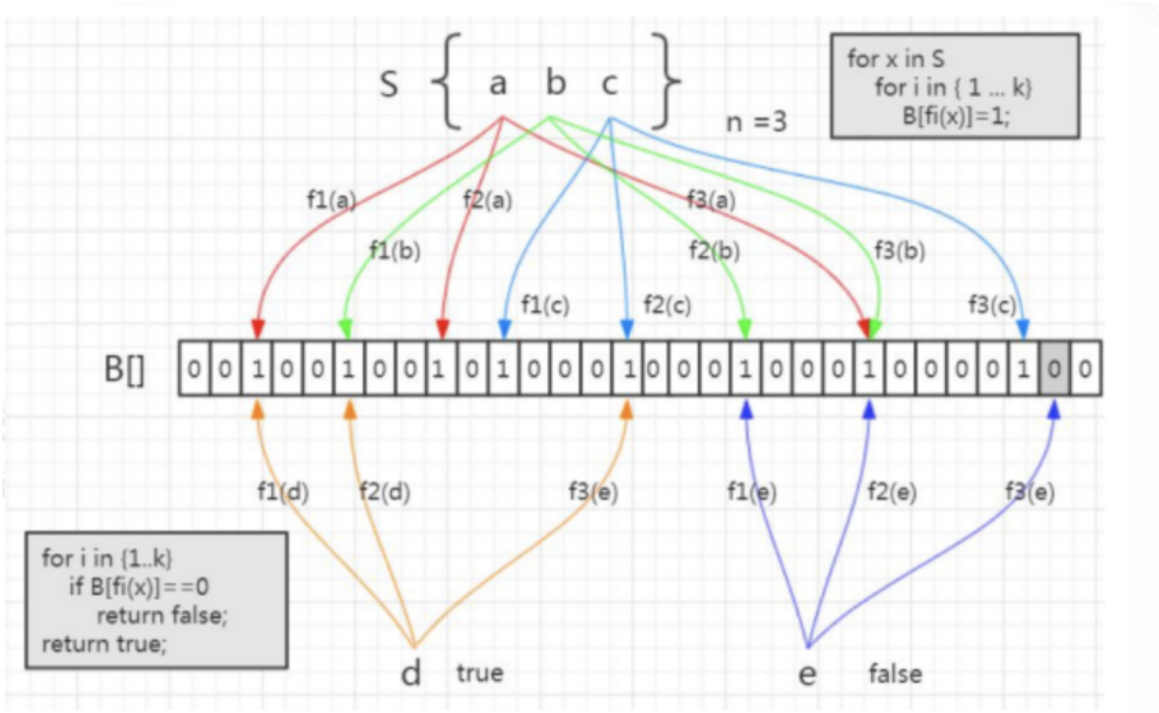
有些同学可能会在想，我使用集合行不行呢：比如在海量元素中（例如 10 亿无序、不定长、不重复）快速 判断一个元素是否存在？好，我们最简单的想法就是把这么多数据放到数据结构里去，比如List、Map、Tree，一搜不就出来了吗，比如map.get(),我们假设一个元素1个字节的字段，10亿的数据大概需要 900G 的内存空间，这个对于普通的服务器来说是承受不了的，当然面试官也不希望听到你这个答案，因为太笨了吧，我们肯定是要用一种好的方法，巧妙的方法来解决，这里引入一种节省空间的数据结构 --- **位数组** ，它是一个有序的数组，只有两个值，0 和 1。0代表不存在，1代表存在。



有了这个厉害的东西，现在我们还需要一个映射关系，你总得知道某个元素在哪个位置上吧，然后在去看这个位置上是0还是1，怎么解决这个问题呢，那就要用到哈希函数，用哈希函数有两个好处，第一是哈希函数无论输入值的长度是多少，得到的输出值长度是固定的，第二是他的分布是均匀的，如果全挤的一块去那还怎么区分，比如MD5、SHA-1这些就是常见的哈希算法。



布隆過濾器原理：



集合里面有3个元素, 要把它存到布隆过滤器里面去，应该怎么做呢？首先是a元素,，这里我们用3次计算，b、c元素也是一样.

元素都存进去以后，现在我要来判断一个元素在这个容器中是否存在，就要使用同样的三个函数(这个地方三个函数只是我们约定的，当然也可能是四个，五个。。。)进行计算。

比如d元素，我用第一个函数f1 计算，发现这个位置上是1，没问题, 第二个位置也是1，第三个位置上也是1。

如果经过三次计算得到的下标位置值都是1，这种情况下, 能不能确定d元素一定在这个容器里面呢? 实际上是不能的. 比如这张图里面，这三个位置分别是把a、b、c 存进去的时候置成1， 所以即使d 元素之前没有存进去, 也会得到三个1，判断返回true

所以 这个是布隆过滤器的一个很重要的特性，因为哈希碰撞是不可避免的，所以它会存在一定的误判率。这种把本来不存在布隆过滤器中的元素误判为存在的情况，我们把它叫做 假阳性(False Positive Probability，FPP)

我们再来看另一个元素e, 我们要判断它在容器中是否存在， 一样的要用这三个函数去计算，第一个位置是1，第二个位置是1，第三个位置是0

e元素是不是一定不在这个容器里面呢？可以确定一定不存在，如果说当时已经把e元素存到布隆过滤器里面去了，那么这三个位置肯定都是1，不可能会出现0。

布隆过滤器的特点，从容器的角度来说：

- 如果布隆过滤器判断元素在集合中存在, 不一定存在.
- 如果布隆过滤器判断不存在, 则一定不存在.

从元素的角度来说:

- 如果元素实际存在, 布隆过滤器一定判断存在
- 如果元素实际不存在,布隆过滤器可能判断存在

利用第二个特性, 我们是不是就可以解决持续从数据库查询不存在的值的问题呢?

布隆过滤器：适用于数据命中不高，数据相对固定实时性低（通常是数据集较大）的应用场景，代码维护较为复杂，但是缓存空间占用少

总结:

布隆过滤器添加元素

- 将要添加的元素给k个哈希函数
- 得到对应于位数组上的k个位置
- 将这k个位置设为1/0

布隆过滤器查询元素

- 将要查询的元素给k个哈希函数
- 得到对应于位数组上的k个位置
- 如果k个位置有一个为0，则肯定不在集合中
- 如果k个位置全部为1，则可能在集合中

Guava实现布隆过滤器案例:

```
@Test
    public void testBloomFilter() {
        //插入多少数据
        int insertions = 1000000;
        //期望的误判率
        double fpp = 0.03;
        long start = System.currentTimeMillis();
        //初始化一个存储string数据的布隆过滤器,默认误判率是0.03
        BloomFilter<String> bf =
BloomFilter.create(Funnels.stringFunnel(Charsets.UTF_8), insertions, fpp);

        //用于存放所有实际存在的key，判断是否真实存在
        Set<String> sets = new HashSet<String>(insertions);
        //用于存放所有实际存在的key，用于取出
        List<String> lists = new ArrayList<String>(insertions);
        //插入随机字符串
        for (int i = 0; i < insertions; i++) {
            String uuid = UUID.randomUUID().toString();
            bf.put(uuid);
            sets.add(uuid);
            lists.add(uuid);
        }
        int rightNum = 0;
        int wrongNum = 0;
        for (int i = 0; i < 10000; i++) {
            // 0-10000之间，可以被100整除的数有100个（100的倍数）
            String data = i % 100 == 0 ? lists.get(i / 100) :
UUID.randomUUID().toString();
```

```
            //这里用了might,看上去不是很自信，所以如果布隆过滤器判断存在了,我们还要去sets中验证下
            if (bf.mightContain(data)) {
                if (sets.contains(data)) {
                    rightNum++;
                    continue;
                }
                wrongNum++;
            }
        }
        long end = System.currentTimeMillis();
        logger.info("the general total time is:" + (end - start));
        BigDecimal percent = new BigDecimal(wrongNum).divide(new BigDecimal(9900), 2, RoundingMode.HALF_UP);
        BigDecimal bingo = new BigDecimal(9900 - wrongNum).divide(new BigDecimal(9900), 2, RoundingMode.HALF_UP);
        System.out.println("在100W个元素中，判断100个实际存在的元素，布隆过滤器认为存在的个数：" + rightNum);
        System.out.println("在100W个元素中，判断9900个实际不存在的元素，误认为存在的个数：" + wrongNum + "，命中率：" + bingo + "，误判率：" + percent);
    }
```

以下为误判率系数表：

k	p	r
1	0.5	1.442695041
2	0.25	2.885390082
3	0.125	4.328085123
4	0.0625	5.770780164
5	0.03125	7.213475204
6	0.015625	8.656170245
7	0.0078125	10.09886529
8	0.00390625	11.54156033
9	0.001953125	12.98425537
10	0.000976563	14.42695041

一般用默认的误判率(0.03)即可，兼顾内存和性能

1.2 缓存击穿

主要体现在：热点数据过了有效时间，此刻有大量请求会落在数据库上，从而可能会导致数据库崩溃

解决方案：

1、互斥锁

只允许一个线程重建缓存，其他线程等待重建缓存的线程执行完，重新从缓存获取数据---》可能存在死锁

```
@Test
    public void testLock() {
        for (int i = 0; i < 100; i++) {
            get("001");
        }
    }

    private String get(String key) {
        JedisPool jedisPool = (JedisPool) context.getBean("jedisPool");
        Jedis jedis = jedisPool.getResource();
        String value = jedis.get(key);
        System.out.println("redis的值" + value);
        if (value == null) { //代表缓存值过期
            //设置一个临时key_mutex，用于阻塞相同的请求！设置10S的超时，防止del操作失败的时候，下次缓存过期一直不能load db
            if (jedis.setnx(key_mutex + key, key_mutex + key) == 1) {  //代表设置成功
                jedis.expire(key_mutex + key, 10);
                value = "去数据库查询出来的值";
                jedis.set(key, value);
                jedis.expire(key, 5 * 60);
                jedis.del(key_mutex);
            } else {  //这个时候代表同时候的其他线程已经load ，并且第一个大爷db并回设到缓存了，这时候重试获取缓存值即可
                try {
                    Thread.sleep(50);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                get(key);  //重试
            }
        } else {
            return value;
        }
        return "";
    }
```

无非是使用setNx阻塞相同的请求！

2、热点数据永不过期

业界主流的做法又分两种：

无非就是对该数据不设置过期时间，但是一定要注意该数据更新的同时，必须对缓存数据进行更新，这种问题在于热点数据过多的话会一致占据内存

1.3 缓存雪崩

因某种原因发生了宕机或者数据在同一时间批量失效，那么原本被缓存抵挡的海量查询请求就会像疯狗一样涌向数据库。此时数据库如果抵挡不了这巨大的压力，它就会崩溃。

解决方案：

1、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得缓存数据库中---Redis Cluster

2、尽可能使缓存数据不在同一时间过期，比如使用随机时间

3、热点数据永不过期

4、最后没办法的情况下，使用服务熔断降级、隔离限流等手段，比如采用netflix的hystrix

最后：无论是缓存穿透，缓存击穿还是缓存雪崩，都建议使用队列来排队、拒绝大量请求涌入和分布式互斥锁来避免后端数据服务被冲击，防止已有的数据出现问题

2、批量查询优化

经常会有这样一种业务逻辑，就是需要根据Redis中Key的规则，模糊查询对应的数据，当数据量少时，利用常规的命令也能满足需求，但是数据量大时，就会导致堵塞，就算是采用不堵塞的函数，如果数据需要显示的话，显示结果的时间也比较慢，用户体验不好。

2.1 pipeline管道模式

```
 @Test
    public void testPipelineQuery(){
        /**
         * 数据库操作
         */
        final long start = System.currentTimeMillis();
        JdbcTemplate jdbcTemplate = (JdbcTemplate)
context.getBean("jdbcTemplate");
        NamedParameterJdbcTemplate namedParameterJdbcTemplate =
                new NamedParameterJdbcTemplate(jdbcTemplate);
        Object[] args = new Object[10000];
        for (int i = 0; i < 10000; i++) {
            args[i] = i;
        }
        MapSqlParameterSource parameters = new MapSqlParameterSource();
        parameters.addValue("ids",Arrays.asList(args));
        List<Map<String, Object>> maps
                = namedParameterJdbcTemplate.queryForList("select * from
redis_batch where id in (:ids)",parameters);
        System.out.println(maps);
        long end = System.currentTimeMillis();
        System.out.println("the general total time is:" + (end-start));

        /**
         * 普通redis查询（反而比不上数据库的批量查询）
         */
        long start0 = System.currentTimeMillis();
        for (int i = 0; i < 10000; i++) {
            jedis.get(String.valueOf(i));
        }
        long end0 = System.currentTimeMillis();
```

```
        System.out.println("the general total time is:" + (end0 - start0));

        /**
         * 查询所有的value
         */
        Pipeline pipe = jedis.pipelined(); // 先创建一个 pipeline 的链接对象（管道，
替换了jedis的单步操作）
        long start1 = System.currentTimeMillis();
        for (int i = 0; i < 10000; i++) {
            pipe.get(String.valueOf(i));
        }
        List<Object> list = pipe.syncAndReturnAll();
        long end1 = System.currentTimeMillis();
        System.out.println("the general total time is:" + (end1 - start1));

        /**
         * 查询所有的key/value
         */
        Map<String,Response<String>> map = new HashMap();
        long start2 = System.currentTimeMillis();
        for (int i = 0; i < 10000; i++) {
            map.put(String.valueOf(i),pipe.get(String.valueOf(i)));
        }
        pipe.sync();
        long end2 = System.currentTimeMillis();
        System.out.println("the general total time is:" + (end2 - start2));
        for (Map.Entry<String, Response<String>> responseEntry : map.entrySet())
{
            Response<String> sResponse=
(Response<String>)responseEntry.getValue();
            /*System.out.println(new String(responseEntry.getKey())
                    +"-----"+new String(sResponse.get()).toString());*/
        }
    }
```

2.2 LUA脚本

2.2.1 LUA简介

Lua 是一种轻量小巧的脚本语言，用标准C语言编写并以源代码形式开放， 其设计目的是为了嵌入应用程序中，从而为应用程序提供灵活的扩展和定制功能。

Redis中使用Lua的好处

- 减少网络开销。可以将多个请求通过脚本的形式一次发送，减少网络时延
- 原子操作。redis会将整个脚本作为一个整体执行，中间不会被其他命令插入。因此在编写脚本的过程中无需担心会出现竞态条件，无需使用事务。
- 复用。客户端发送的脚步会永久存在redis中，这样，其他客户端可以复用这一脚本而不需要使用代码完成相同的逻辑。

2.2.1 lua 安装和helloworld

```
curl -R -O http://www.lua.org/ftp/lua-5.3.0.tar.gz
tar zxf lua-5.3.0.tar.gz
cd lua-5.3.0
make linux test
make install


-------------------------
[root@ydt1 lua-5.3.0]# lua
Lua 5.1.4  Copyright (C) 1994-2008 Lua.org, PUC-Rio
> print("Hello World!")
Hello World!
```

2.2.2 lua批量查询

可惜lua脚本长度有限制!

```
@Test
    public void test() {
        /*jedis.eval("redis.call('set','lua','hello lua')");*/
        /*System.out.println(jedis.eval("return redis.call('get','lua')"));*/
        long start0 = System.currentTimeMillis();
        StringBuffer luaScript = new StringBuffer("return redis.call('mget'");
        List<String> keys = new ArrayList<String>();
        for (int i = 0; i < 247; i++) {
            keys.add(i + "");
            luaScript.append(",KEYS[" + (i + 1) + "]");
        }
        luaScript.append(")");
        List<String> values = new ArrayList<String>();
        Object getResult = null;
        for (int i = 0; i < 40; i++) {
            getResult = jedis.eval(luaScript.toString(), keys, values);
        }
        long end0 = System.currentTimeMillis();
        System.out.println("the general total time is:" + (end0 - start0));
        List<String> retList = (List<String>) getResult;
        for (String ret : retList) {
            System.out.println(ret);
        }
    }
```

2.3 pipeline与lua区别

lua

Redis在2.6版引入了对Lua的支持

- 使用Lua可以非常明显的提升Redis的效率。
- 只要脚本所对应的函数曾经在 Lua 里面定义过， 那么即使用户不知道脚本的内容本身， 也可以直接通过脚本的 SHA1 校验和来调用脚本所对应的函数， 从而达到执行脚本的目的 —— 这就是 EVALSHA 命令的实现原理
- 当 Lua 脚本里本身有调用 Redis 命令时（执行 `redis.call` 或者 `redis.pcall` ）， Redis 和 Lua 脚本之间的数据交互会更复杂一些。

redis pipeline

pipeline引入，降低了多次命令-应答之间的网络交换次数，并不能缩小redis对每个命令的处理时间

什么时候使用pipeline，什么时候使用lua

- 当多个redis命令之间没有依赖、顺序关系（例如第二条命令依赖第一条命令的结果）时，建议使用pipline；
- 如果命令之间有依赖或顺序关系时，pipline就无法使用，此时可以考虑才用lua脚本的方式来使用。

redis执行lua脚本好处

- 减少网络开销，本来多次网络请求的操作，可以用一个请求完成，原来多次请求的逻辑均放在redis服务器上完成。使用lua，减少了网络往返时延；
- 原子操作：redis会将整个脚本作为一个整体执行，不会被其他命令插入。
- 复用：客户端发送的脚本会永久存储在redis中，意味着其他客户端可以复用这一脚本而无需使用代码完成同样逻辑。