Redis Java客户端介绍

已有的客户端支持

Redis Java客户端有很多的开源产品比如Redission、Jedis、lettuce

差异

Jedis是Redis的Java实现的客户端，其API提供了比较全面的Redis命令的支持；

Redisson实现了分布式和可扩展的Java数据结构，和Jedis相比，功能较为简单，不支持字符串操作，不支持排序、事务、管道、分区等Redis特性。Redisson主要是促进使用者对Redis的关注分离，从而让使用者能够将精力更集中地放在处理业务逻辑上。

lettuce是基于Netty构建的一个可伸缩的线程安全的Redis客户端，支持同步、异步、响应式模式。多个线程可以共享一个连接实例，而不必担心多线程并发问题；

jedis-sentinel原理分析

原理

客户端通过连接到哨兵集群，通过发送Protocol.**SENTINEL_GET_MASTER_ADDR_BY_NAME** 命令，从哨兵机器中询问master节点的信息，拿到master节点的ip和端口号以后，再到客户端发起连接。连接以后，需要在客户端建立监听机制，当master重新选举之后，客户端需要重新连接到新的master节点

源码分析

```
private HostAndPort initSentinels(Set<String> sentinels, final String masterName) {

  HostAndPort master = null;
  boolean sentinelAvailable = false;

  log.info("Trying to find master from available Sentinels...");
// 有多个sentinels,遍历这些个sentinels
  for (String sentinel : sentinels) {
// host:port表示的sentinel地址转化为一个HostAndPort对象。
    final HostAndPort hap = HostAndPort.parseString(sentinel);
    log.fine("Connecting to Sentinel " + hap);
    Jedis jedis = null;
    try {
// 连接到sentinel
      jedis = new Jedis(hap.getHost(), hap.getPort());
// 根据masterName得到master的地址，返回一个list，host= list[0]，port =// list[1]
      List<String> masterAddr = jedis.sentinelGetMasterAddrByName(masterName);
      // connected to sentinel...
      sentinelAvailable = true;
```

```
      if (masterAddr == null || masterAddr.size() != 2) {
        log.warning("Can not get master addr, master name: " + masterName + ".
Sentinel: " + hap
            + ".");
        continue;
      }
// 如果在任何一个sentinel中找到了master，不再遍历sentinels
      master = toHostAndPort(masterAddr);
      log.fine("Found Redis master at " + master);
      break;
    } catch (JedisException e) {
      // resolves #1036, it should handle JedisException there's another chance
      // of raising JedisDataException
      log.warning("Cannot get master address from sentinel running @ " + hap + ".
Reason: " + e
          + ". Trying next one.");
    } finally {
      if (jedis != null) {
        jedis.close();
      }
    }
  }
// 到这里，如果master为null，则说明有两种情况，一种是所有的sentinels节点都down掉了，一种是master节
点没有被存活的sentinels监控到
  if (master == null) {
    if (sentinelAvailable) {
      // can connect to sentinel, but master name seems to not
      // monitored
      throw new JedisException("Can connect to sentinel, but " + masterName
          + " seems to be not monitored...");
    } else {
      throw new JedisConnectionException("All sentinels down, cannot determine where is
"
          + masterName + " master is running...");
    }
  }
//如果走到这里，说明找到了master的地址
  log.info("Redis master running at " + master + ", starting Sentinel listeners...");
//启动对每个sentinels的监听
为每个sentinel都启动了一个监听者MasterListener。MasterListener本身是一个线程，它会去订阅sentinel
上关于master节点地址改变的消息。
  for (String sentinel : sentinels) {
    final HostAndPort hap = HostAndPort.parseString(sentinel);
    MasterListener masterListener = new MasterListener(masterName, hap.getHost(),
hap.getPort());
    // whether MasterListener threads are alive or not, process can be stopped
    masterListener.setDaemon(true);
    masterListeners.add(masterListener);
    masterListener.start();
  }
  return master;
}
```

从哨兵节点获取master信息的方法

```
public List<String> sentinelGetMasterAddrByName(String masterName) {
  client.sentinel(Protocol.SENTINEL_GET_MASTER_ADDR_BY_NAME, masterName);
  final List<Object> reply = client.getObjectMultiBulkReply();
  return BuilderFactory.STRING_LIST.build(reply);
}
```

Jedis-cluster原理分析

连接方式

```
Set<HostAndPort> hostAndPorts=new HashSet<>();
HostAndPort hostAndPort=new HostAndPort("192.168.11.153",7000);
HostAndPort hostAndPort1=new HostAndPort("192.168.11.153",7001);
HostAndPort hostAndPort2=new HostAndPort("192.168.11.154",7003);
HostAndPort hostAndPort3=new HostAndPort("192.168.11.157",7006);
hostAndPorts.add(hostAndPort);
hostAndPorts.add(hostAndPort1);
hostAndPorts.add(hostAndPort2);
hostAndPorts.add(hostAndPort3);
JedisCluster jedisCluster=new JedisCluster(hostAndPorts,6000);
jedisCluster.set("mic","hello");
```

原理分析

程序启动初始化集群环境

1)、读取配置文件中的节点配置，无论是主从，无论多少个，只拿第一个，获取redis连接实例

2)、用获取的redis连接实例执行clusterNodes()方法，实际执行redis服务端cluster nodes命令，获取主从配置信息

3)、解析主从配置信息，先把所有节点存放到nodes的map集合中，key为节点的ip:port，value为当前节点的jedisPool

4)、解析主节点分配的slots区间段，把slot对应的索引值作为key，第三步中拿到的jedisPool作为value，存储在slots的map集合中

就实现了slot槽索引值与jedisPool的映射，这个jedisPool包含了master的节点信息，所以槽和几点是对应的，与redis服务端一致

从集群环境存取值

1)、把key作为参数，执行CRC16算法，获取key对应的slot值

2)、通过该slot值，去slots的map集合中获取jedisPool实例

3)、通过jedisPool实例获取jedis实例，最终完成redis数据存取工作

Redisson客户端的操作方式

redis-cluster连接方式

```
Config config=new Config();
config.useClusterServers().setScanInterval(2000).
        addNodeAddress("redis://192.168.11.153:7000",
                "redis://192.168.11.153:7001",
                "redis://192.168.11.154:7003","redis://192.168.11.157:7006");
RedissonClient redissonClient= Redisson.create(config);
RBucket<String> rBucket=redissonClient.getBucket("mic");
System.out.println(rBucket.get());
```

常规操作敏玲

```
getBucket-> 获取字符串对象;
getMap -> 获取map对象
getSortedSet->获取有序集合
getSet -> 获取集合
getList ->获取列表
```

redis实战

分布式锁的实现

关于锁，其实我们或多或少都有接触过一些，比如synchronized、 Lock这些，这类锁的目的很简单，在多线程环境下，对共享资源的访问造成的线程安全问题，通过锁的机制来实现资源访问互斥。那么什么是分布式锁呢？或者为什么我们需要通过Redis来构建分布式锁，其实最根本原因就是Score（范围），因为在分布式架构中，所有的应用都是进程隔离的，在多进程访问共享资源的时候我们需要满足互斥性，就需要设定一个所有进程都能看得到的范围，而这个范围就是Redis本身。所以我们才需要把锁构建到Redis中。

Redis里面提供了一些比较具有能够实现锁特性的命令，比如SETEX(在键不存在的情况下为键设置值)，那么我们可以基于这个命令来去实现一些简单的锁的操作

分布式锁实战

> 源代码已经上传到git->mic-vip目录下，请直接移步到git上下载

Redisson实现分布式锁

Redisson它除了常规的操作命令以外，还基于redis本身的特性去实现了很多功能的封装，比如分布式锁、原子操作、布隆过滤器、队列等等。我们可以直接利用这个api提供的功能去实现

```
Config config=new Config();
        config.useSingleServer().setAddress("redis://192.168.11.152:6379");
RedissonClient redissonClient=Redisson.create(config);

 RLock rLock=redissonClient.getLock("updateOrder");
        //最多等待100秒、上锁10s以后自动解锁
        if(rLock.tryLock(100,10,TimeUnit.SECONDS)){
            System.out.println("获取锁成功");
        }
```

原理分析

trylock

```
@Override
public boolean tryLock(long waitTime, long leaseTime, TimeUnit unit) throws InterruptedException {
    long time = unit.toMillis(waitTime);
    long current = System.currentTimeMillis();
    final long threadId = Thread.currentThread().getId();
    Long ttl = tryAcquire(leaseTime, unit, threadId);
    // lock acquired
    if (ttl == null) {
        return true;
    }

    time -= (System.currentTimeMillis() - current);
    if (time <= 0) {
        acquireFailed(threadId);
        return false;
    }
```

申请锁，返回还剩余的锁的过期时间

表示申请锁成功

tryAcquire

```java
private <T> RFuture<Long> tryAcquireAsync(long leaseTime, TimeUnit unit, final long threadId) {
    if (leaseTime != -1) {
        return tryLockInnerAsync(leaseTime, unit, threadId, RedisCommands.EVAL_LONG);
    }
    RFuture<Long> ttlRemainingFuture = tryLockInnerAsync(commandExecutor.getConnectionManager().
    ttlRemainingFuture.addListener(new FutureListener<Long>() {
        @Override
        public void operationComplete(Future<Long> future) throws Exception {
            if (!future.isSuccess()) {
                return;
            }

            Long ttlRemaining = future.getNow();
            // lock acquired
            if (ttlRemaining == null) {
                scheduleExpirationRenewal(threadId);
            }
        }
    });
    return ttlRemainingFuture;
}
```

针对leaseTime（过期时间）是否设置过来做不同的转发处理

tryLockInnerAsync

```java
<T> RFuture<T> tryLockInnerAsync(long leaseTime, TimeUnit unit, long threadId, RedisStrictCommand<T> command) {
    internalLockLeaseTime = unit.toMillis(leaseTime);

    return commandExecutor.evalWriteAsync(getName(), LongCodec.INSTANCE, command,
              script: "if (redis.call('exists', KEYS[1]) == 0) then " +
                  "redis.call('hset', KEYS[1], ARGV[2], 1); " +
                  "redis.call('pexpire', KEYS[1], ARGV[1]); " +
                  "return nil; " +
              "end; " +
              "if (redis.call('hexists', KEYS[1], ARGV[2]) == 1) then " +
                  "redis.call('hincrby', KEYS[1], ARGV[2], 1); " +
                  "redis.call('pexpire', KEYS[1], ARGV[1]); " +
                  "return nil; " +
              "end; " +
              "return redis.call('pttl', KEYS[1]);",
                Collections.<~>singletonList(getName()), internalLockLeaseTime, getLockName(threadId));
}
```

通过lua脚本来实现加锁的操作

\1. 判断lock键是否存在，不存在直接调用hset存储当前线程信息并且设置过期时间,返回nil，告诉客户端直接获取到锁。

\2. 判断lock键是否存在，存在则将重入次数加1，并重新设置过期时间，返回nil，告诉客户端直接获取到锁。

\3. 被其它线程已经锁定，返回锁有效期的剩余时间，告诉客户端需要等待。

unlock

```
protected RFuture<Boolean> unlockInnerAsync(long threadId) {
    return commandExecutor.evalWriteAsync(getName(), LongCodec.INSTANCE, RedisCommands.EVAL_BOOLEAN,
            script: "if (redis.call('exists', KEYS[1]) == 0) then " +
                "redis.call('publish', KEYS[2], ARGV[1]); " +
                "return 1; " +
            "end;" +
            "if (redis.call('hexists', KEYS[1], ARGV[3]) == 0) then " +
                "return nil;" +
            "end; " +
            "local counter = redis.call('hincrby', KEYS[1], ARGV[3], -1); " +
            "if (counter > 0) then " +
                "redis.call('pexpire', KEYS[1], ARGV[2]); " +
                "return 0; " +
            "else " +
                "redis.call('del', KEYS[1]); " +
                "redis.call('publish', KEYS[2], ARGV[1]); " +
                "return 1; "+
            "end; " +
            "return nil;",
            Arrays.<~>asList(getName(), getChannelName()), LockPubSub.unlockMessage, internalLockLeaseTime,

}
```

\1. 如果lock键不存在，发消息说锁已经可用，发送一个消息

\2. 如果锁不是被当前线程锁定，则返回nil

\3. 由于支持可重入，在解锁时将重入次数需要减1

\4. 如果计算后的重入次数>0，则重新设置过期时间

\5. 如果计算后的重入次数<=0，则发消息说锁已经可用

管道模式

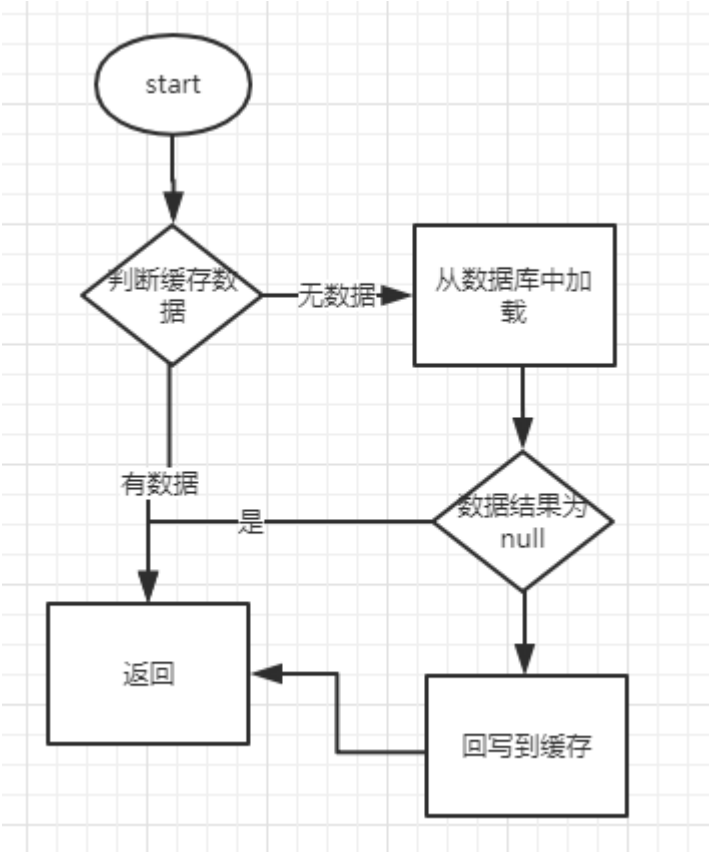
Redis服务是一种C/S模型，提供请求 - 响应式协议的TCP服务，所以当客户端发起请求，服务端处理并返回结果到客户端，一般是以阻塞形式等待服务端的响应，但这在批量处理连接时延迟问题比较严重，所以Redis为了提升或弥补这个问题，引入了管道技术：可以做到服务端未及时响应的时候，客户端也可以继续发送命令请求，做到客户端和服务端互不影响，服务端并最终返回所有服务端的响应，大大提高了C/S模型交互的响应速度上有了质的提高

使用方法

```
Jedis jedis=new Jedis("192.168.11.152",6379);
Pipeline pipeline=jedis.pipelined();
for(int i=0;i<1000;i++){
    pipeline.incr("test");
}
pipeline.sync();
```

Redis的应用架构

对于读多写少的高并发场景，我们会经常使用缓存来进行优化。比如说支付宝的余额展示功能，实际上99%的时候都是查询，1%的请求是变更（除非是土豪，每秒钟都有收入在不断更改余额），所以，我们在这样的场景下，可以加入缓存，用户->余额



Redis缓存与数据一致性问题

那么基于上面的这个出发点，问题就来了，当用户的余额发生变化的时候，如何更新缓存中的数据，也就是说。

\1. 我是先更新缓存中的数据再更新数据库的数据；

\2. 还是修改数据库中的数据再更新缓存中的数据

这就是我们经常会在面试遇到的问题，数据库的数据和缓存中的数据如何达到一致性？首先，可以肯定的是，redis中的数据和数据库中的数据不可能保证事务性达到统一的，这个是毫无疑问的，所以在实际应用中，我们都是基于当前的场景进行权衡降低出现不一致问题的出现概率

更新缓存还是让缓存失效

更新缓存表示数据不但会写入到数据库，还会同步更新缓存； 而让缓存失效是表示只更新数据库中的数据，然后删除缓存中对应的key。那么这两种方式怎么去选择？这块有一个衡量的指标。

\1. 如果更新缓存的代价很小，那么可以先更新缓存，这个代价很小的意思是我不需要很复杂的计算去获得最新的余额数字。

\2. 如果是更新缓存的代价很大，意味着需要通过多个接口调用和数据查询才能获得最新的结果，那么可以先淘汰缓存。淘汰缓存以后后续的请求如果在缓存中找不到，自然去数据库中检索。

先操作数据库还是先操作缓存?

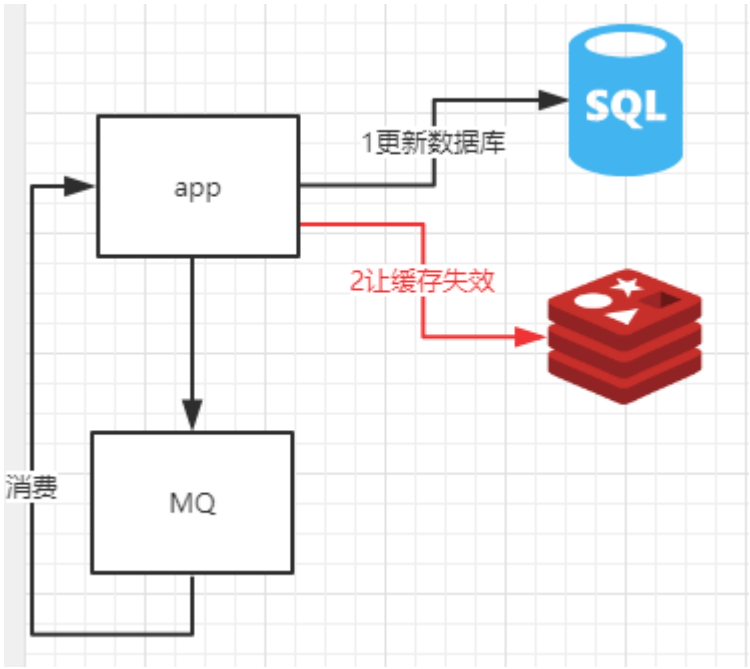
当客户端发起事务类型请求时，假设我们以让缓存失效作为缓存的的处理方式，那么又会存在两个情况，

\1. 先更新数据库再让缓存失效

\2. 先让缓存失效，再更新数据库

前面我们讲过，更新数据库和更新缓存这两个操作，是无法保证原子性的，所以我们需要根据当前业务的场景的容忍性来选择。也就是如果出现不一致的情况下，哪一种更新方式对业务的影响最小，就先执行影响最小的方案

最终一致性的解决方案



关于缓存雪崩的解决方案

当缓存大规模渗透在整个架构中以后，那么缓存本身的可用性讲决定整个架构的稳定性。那么接下来我们来讨论下缓存在应用过程中可能会导致的问题。

缓存雪崩

缓存雪崩是指设置缓存时采用了相同的过期时间，导致缓存在某一个时刻同时失效，或者缓存服务器宕机宕机导致缓存全面失效，请求全部转发到了DB层面，DB由于瞬间压力增大而导致崩溃。缓存失效导致的雪崩效应对底层系统的冲击是很大的。

解决方式

\1. 对缓存的访问，如果发现从缓存中取不到值，那么通过加锁或者队列的方式保证缓存的单进程操作，从而避免失效时并发请求全部落到底层的存储系统上；但是这种方式会带来性能上的损耗

\2. 将缓存失效的时间分散，降低每一个缓存过期时间的重复率

\3. 如果是因为缓存服务器故障导致的问题，一方面需要保证缓存服务器的高可用、另一方面，应用程序中可以采用多级缓存

缓存穿透

缓存穿透是指查询一个根本不存在的数据，缓存和数据源都不会命中。出于容错的考虑，如果从数据层查不到数据则不写入缓存，即数据源返回值为 null 时，不缓存 null。缓存穿透问题可能会使后端数据源负载加大，由于很多后端数据源不具备高并发性，甚至可能造成后端数据源宕掉

解决方式

\1. 如果查询数据库也为空，直接设置一个默认值存放到缓存，这样第二次到缓冲中获取就有值了，而不会继续访问数据库，这种办法最简单粗暴。比如，”key”，“&&”。

在返回这个&&值的时候，我们的应用就可以认为这是不存在的key，那我们的应用就可以决定是否继续等待继续访问，还是放弃掉这次操作。如果继续等待访问，过一个时间轮询点后，再次请求这个key，如果取到的值不再是&&，则可以认为这时候key有值了，从而避免了透传到数据库，从而把大量的类似请求挡在了缓存之中。

\2. 根据缓存数据Key的设计规则，将不符合规则的key进行过滤

采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的BitSet中，不存在的数据将会被拦截掉，从而避免了对底层存储系统的查询压力

布隆过滤器

布隆过滤器是Burton Howard Bloom在1970年提出来的，一种空间效率极高的概率型算法和数据结构，主要用来判断一个元素是否在集合中存在。因为他是一个概率型的算法，所以会存在一定的误差，如果传入一个值去布隆过滤器中检索，可能会出现检测存在的结果但是实际上可能是不存在的，但是肯定不会出现实际上不存在然后反馈存在的结果。因此，Bloom Filter不适合那些“零错误”的应用场合。而在能容忍低错误率的应用场合下，Bloom Filter通过极少的错误换取了存储空间的极大节省。

bitmap

所谓的Bit-map就是用一个bit位来标记某个元素对应的Value，通过Bit为单位来存储数据，可以大大节省存储空间.所以我们可以通过一个int型的整数的32比特位来存储32个10进制的数字，那么这样所带来的好处是内存占用少、效率很高（不需要比较和位移）比如我们要存储5**(101)**、3**(11)**四个数字，那么我们申请int型的内存空间，会有32个比特位。这四个数字的二进制分别对应

从右往左开始数，比如第一个数字是5，对应的二进制数据是101, 那么从右往左数到第5位，把对应的二进制数据存储到32个比特位上。

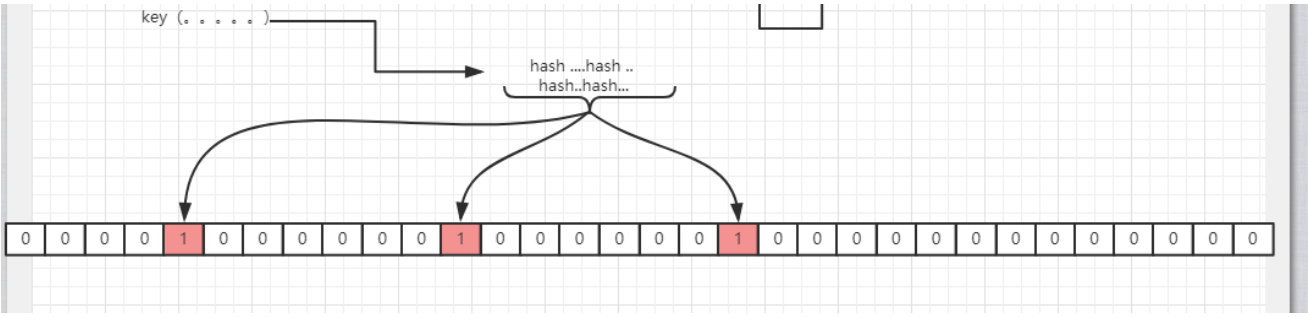
第一个5就是 00000000000000000000000000101000

输入3时候 00000000000000000000000000001100

布隆过滤器原理

有了对位图的理解以后，我们对布隆过滤器的原理理解就会更容易了，仍然以前面提到的40亿数据为案例，假设这40亿数据为某邮件服务器的黑名单数据，邮件服务需要根据邮箱地址来判断当前邮箱是否属于垃圾邮件。原理如下

假设集合里面有3个元素{x, y, z}，哈希函数的个数为3。首先将位数组进行初始化，将里面每个位都设置位0。对于集合里面的每一个元素，将元素依次通过3个哈希函数进行映射，每次映射都会产生一个哈希值，这个值对应位数组上面的一个点，然后将位数组对应的位置标记为1。查询W元素是否存在集合中的时候，同样的方法将W通过哈希映射到位数组上的3个点。如果3个点的其中有一个点不为1，则可以判断该元素一定不存在集合中。反之，如果3个点都为1，则该元素可能存在集合中



接下来按照该方法处理所有的输入对象，每个对象都可能把bitMap中一些白位置涂黑，也可能会遇到已经涂黑的位置，遇到已经为黑的让他继续为黑即可。处理完所有的输入对象之后，在bitMap中可能已经有相当多的位置已经被涂黑。至此，一个布隆过滤器生成完成，这个布隆过滤器代表之前所有输入对象组成的集合。

如何去判断一个元素是否存在bit array中呢？ 原理是一样，根据k个哈希函数去得到的结果，如果所有的结果都是1，表示这个元素可能（假设某个元素通过映射对应下标为4，5，6这3个点。虽然这3个点都为1，但是很明显这3个点是不同元素经过哈希得到的位置，因此这种情况说明元素虽然不在集合中，也可能对应的都是1）存在。 如果一旦发现其中一个比特位的元素是0，表示这个元素一定不存在

至于k个哈希函数的取值为多少，能够最大化的降低错误率（因为哈希函数越多，映射冲突会越少），这个地方就会涉及到最优的哈希函数个数的一个算法逻辑