Redis相关面试题

1.数据类型

1.Redis的常用数据类型有哪些？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆☆

Redis是典型的“键值型”数据库，不同数据类型其key结构一致，value有所差异。常见的类型有：string、hash、list、set、SortedSet等。

而基于以上5种基本数据类型，Redis又拓展了几种拓展类型，例如：BitMap、HyperLogLog、Geo等。

- String类型是Redis中最常见的数据类型，value与key一样都是Redis自定义的字符串结构，称为SDS。不过在保存数字、小字符串时因为采用INT和EMBSTR编码，内存结构紧凑，只需要申请一次内存分配，效率更高，更节省内存。

  而超过44字节的大字符串时则需要采用RAW编码，申请额外的SDS空间，需要两次内存分配，效率较低，内存占用也较高，但最大不超过512mb，因此建议单个value尽量不要超过44字节。

  String类型常用来做计数器、简单数据存储等。复杂数据建议采用其它数据结构。

- Hash结构，其value与Java中的HashMap类似，有是一个key-value结构。如果有一个对象需要被Redis缓存，而且将来可能有部分修改。建议用hash结构来存储这个对象的每一个字段和字段值。而不是作为一个JSON字符串存储到String类型中。因为Hash结构的每一个字段都可以单独做修改，而String的JSON串必须整体覆盖。

与Java中的hashMap不同的是，Redis中的Hash底层采用了渐进式rehash的算法，在做rehash时会创建一个新的hashtable，每次操作元素时移动一部分数据，只到所有数据迁移完成，再用新的HashTable来代替旧的，避免了因为rehash导致的阻塞，因此性能更高。

- List结构的value类型可以看做是一个双端链表，提供了一些命令便于我们从首尾操作元素。为了节省内存空间，底层采用了ZipList（压缩列表）来做基础存储。当压缩列表数据达到阈值（512）则会创建新的压缩列表。每个压缩列表作为一个双端链表的一个节点，最终形成一个QuickList结构。而且QuickList结构与一般的双端链表不同，他可以对中间不常用的ZipList节点做压缩以节省内存。

  List结构常用来模拟队列，实现任务排队这样的功能。

- Set结构的value与Java的Set类似，元素不可重复。Redis提供了求交集、并集等命令，可以帮助我们实现例如：好友列表、共同好友等功能。
  当存储元素是整数时，其底层默认采用IntSet结构，可以看做是一个有序的数组，结构紧凑，效率较高。而元素如果不是整数，或者元素量超过512这个阈值时则会转为hash表结构，内存占用会有大的增加。因此我们在使用Set结构时尽量采用数组存储，例如数值类型的id。而且元素数量尽量不要超过512，避免出现BigKey。

- SortedSet，也叫ZSet。其value就是一个有序的Set集合，元素唯一，并且会按照一个指定的score值排序。因此常用来做排行榜功能。

  SortedSet底层的利用Hash表保证元素的唯一性。利用跳表（SkipList）来保证元素的有序性，因此数据会有重复存储，内存占用较高，是一种典型的以空间换时间的设计。不建议在SortedSet中放入过多数据。

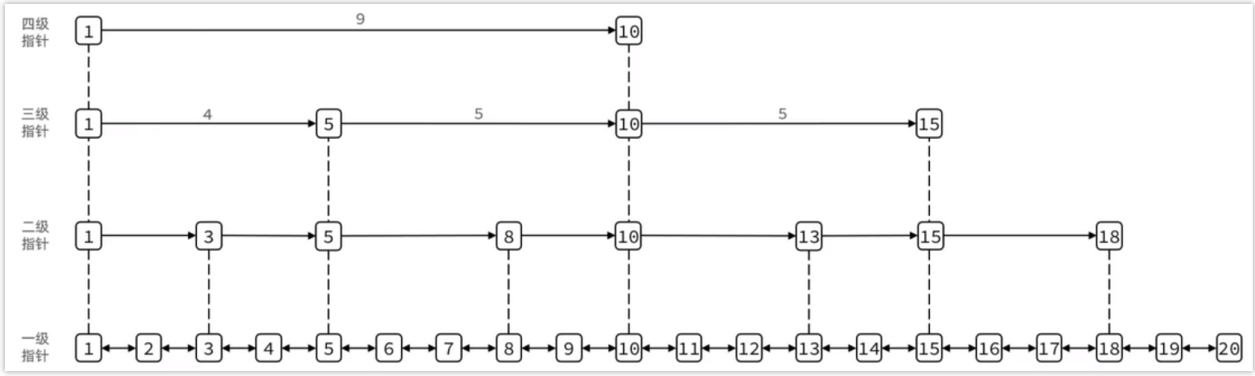
2.跳表你了解吗？

> 难易程度：☆☆☆☆
>
> 出现频率：☆☆☆

跳表(SkipList)首先是链表，但与传统的链表相比有几点差异：

- 跳表结合了链表和二分查找的思想
- 元素按照升序排列存储
- 节点可能包含多个指针，指针跨度不同

- 查找时从顶层向下，不断缩小搜索范围
- 整个查询的复杂度为 O (log n)



Redis数据类型Sorted Set使用了跳表作为其中一种数据结构

2.持久化

Redis的数据持久化策略有哪些？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆

在Redis中提供了两种数据持久化的方式：1、RDB 2、AOF

RDB：

定期更新，定期将Redis中的数据生成的快照同步到磁盘等介质上，磁盘上保存的就是Redis的内存快照

优点：数据文件的大小相比于aop较小，使用rdb进行数据恢复速度较快

缺点：比较耗时，存在丢失数据的风险

AOF：

将Redis所执行过的所有指令都记录下来，在下次Redis重启时，只需要执行指令就可以了

优点：数据丢失的风险大大降低了

缺点：数据文件的大小相比于rdb较大，使用aop文件进行数据恢复的时候速度较慢

> 你们的项目中的持久化是如何配置选择的？
>
> RDB+AOF

3.主从和集群

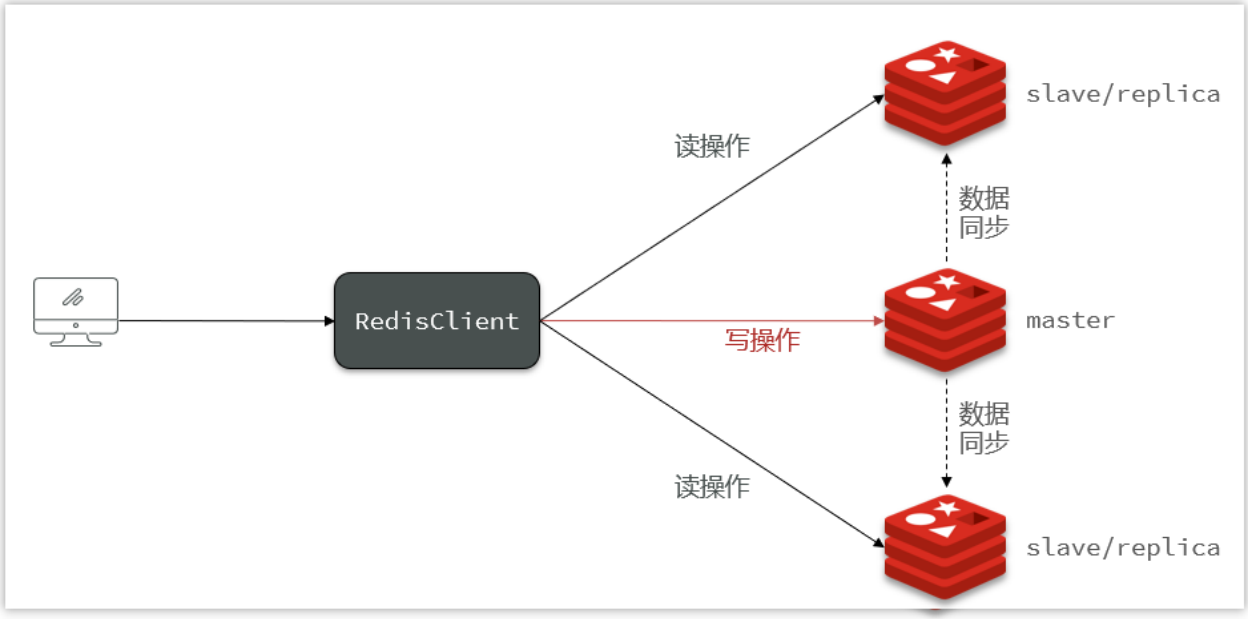
3.1 Redis集群有哪些方案, 知道嘛？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆

在Redis中提供的集群方案总共有三种：

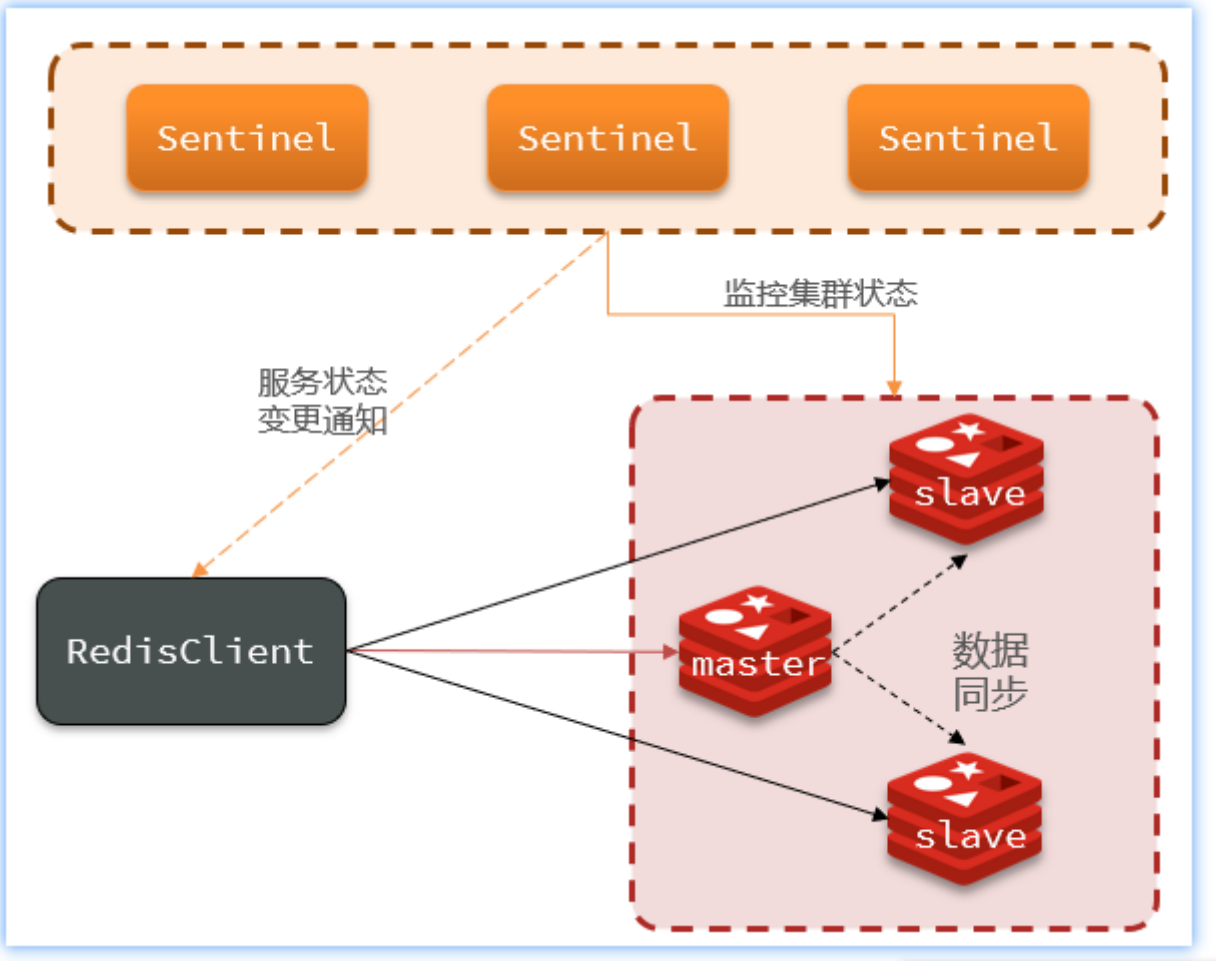
1、主从复制

- 保证高可用性
- 实现故障转移需要手动实现
- 无法实现海量数据存储



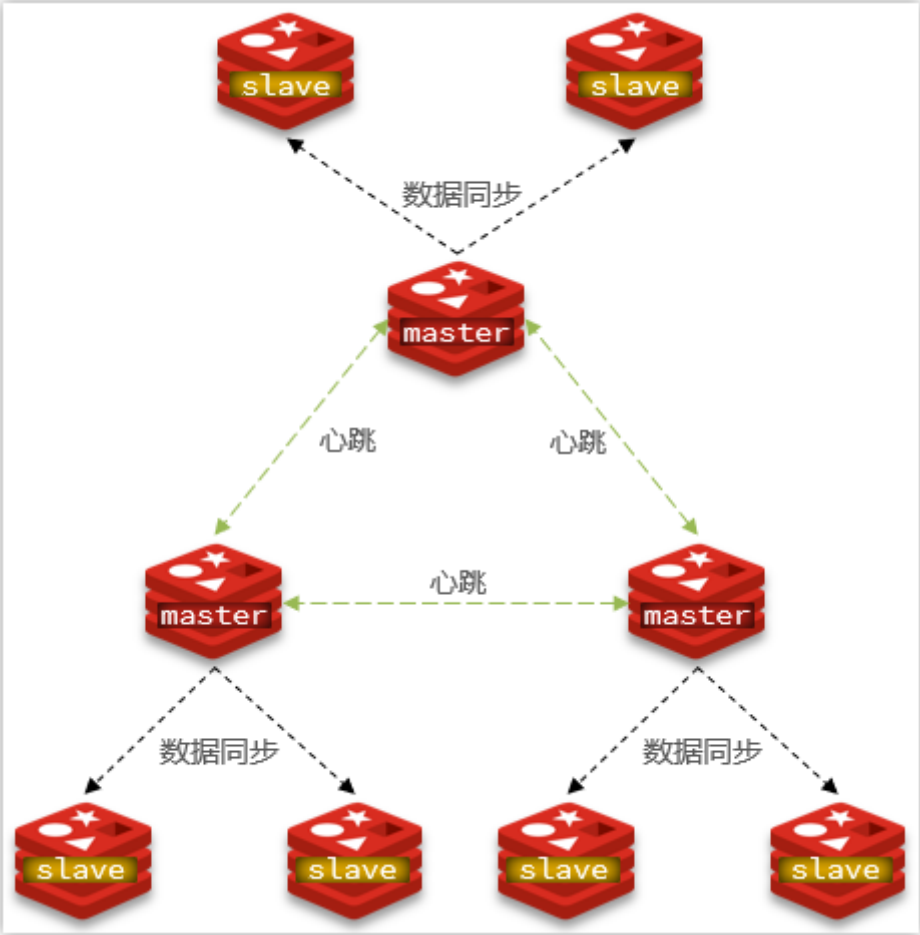
2、哨兵模式

- 保证高可用性
- 可以实现自动化的故障转移
- 无法实现海量数据存储



3、Redis分片集群

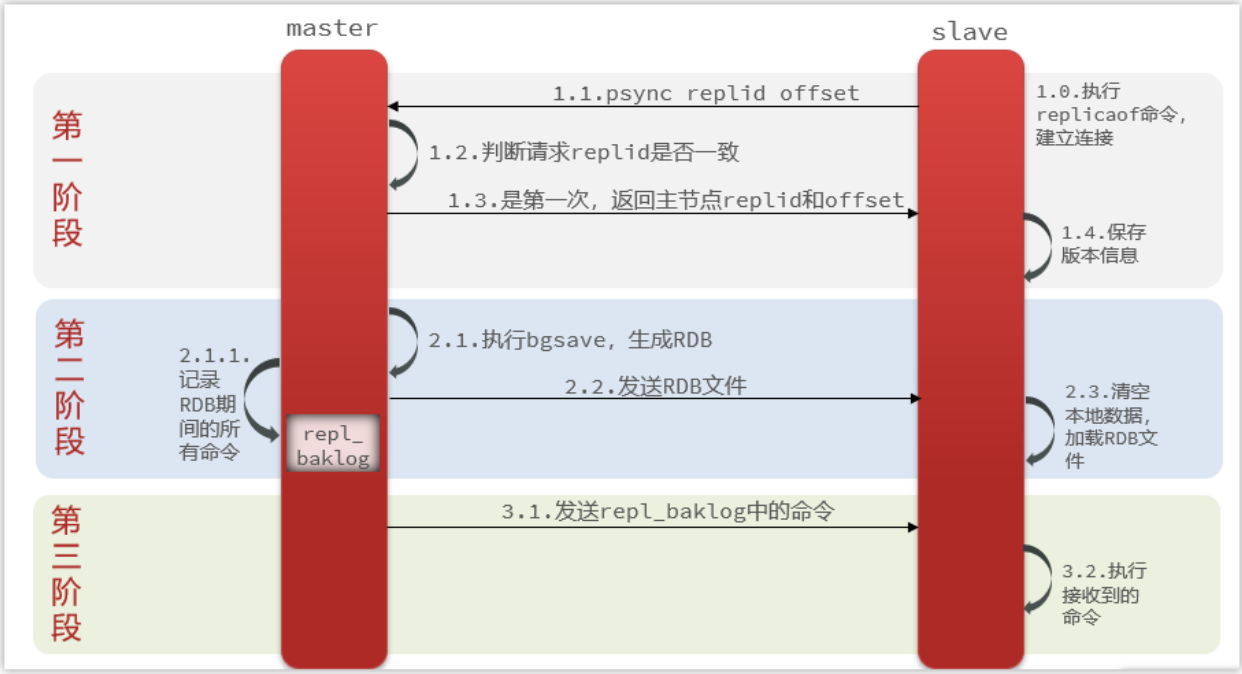
- 保证高可用性
- 可以实现自动化的故障转移
- 可以实现海量数据存储



3.2 什么是 Redis 主从同步？

> 难易程度：☆☆☆☆
>
> 出现频率：☆☆☆☆

主从第一次同步是**全量同步**



第一阶段，全量同步流程
1.从节点执行replicaof命令，发送自己的replid和offset给主节点
2.主节点判断从节点的replid与自己的是否一致，
3.如果不一致说明是第一次来，需要做全量同步，主节点返回自己的replid给从节点
4.主节点开始执行bgsave，生成rdb文件
5.主节点发送rdb文件给从节点，再发送的过程中
6.从节点接收rdb文件，清空本地数据，加载rdb文件中的数据
7.同步过程中，主节点接收到的新命令写入从节点的写缓冲区(repl_buffer)
8.从节点接收到缓冲区数据后写入本地，并记录最新数据对应的offset
9.后期采用增量同步

后期数据变化后，则执行**增量同步**

> 1.主节点会不断把自己接收到的命令记录在repl_baklog中，并修改offset
> 2.从节点向主节点发送psync命令，发送自己的offset和replid
> 3.主节点判断replid和offset与从节点是否一致
> 4.如果replid一致，说明是增量同步。然后判断offset是否一致
> 5.如果从节点offset小于主节点offset，并且在repl_baklog中能找到对应数据则将offset之间相差的数据发送给从节点
> 6.从节点接收到数据后写入本地，修改自己的offset与主节点一致

增量同步的风险

> repl_baklog大小有上限，写满后会覆盖最早的数据。如果slave断开时间过久，导致尚未备份的数据被覆盖，则无法基于log做增量同步，只能再次全量同步。
>
> repl_baklog可以在配置文件中进行修改存储大小

3.3 你们使用Redis是单点还是集群？哪种集群？(说说你们生产环境redis部署情况？)

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

一般部分服务做缓存用的Redis直接做主从（1主1从）加哨兵就可以了。单节点不超过10G内存，如果Redis内存不足则可以给不同服务分配独立的Redis主从节点。尽量不做分片集群。
原因：

- 维护起来比较麻烦
- 集群之间的心跳检测和数据通信会消耗大量的网络带宽
- 集群插槽分配不均和key的分批容易导致数据倾斜
- 客户端的route会有性能损耗
- 集群模式下无法使用lua脚本、事务

其他扩展：

- 一般的企业中redis存储超过100GB就是极少见的，一般只存热点数据
- 极端情况下，可以设置较大的内存。以阿里云为主，购买内存型服务器，目前最大的为：2048GB

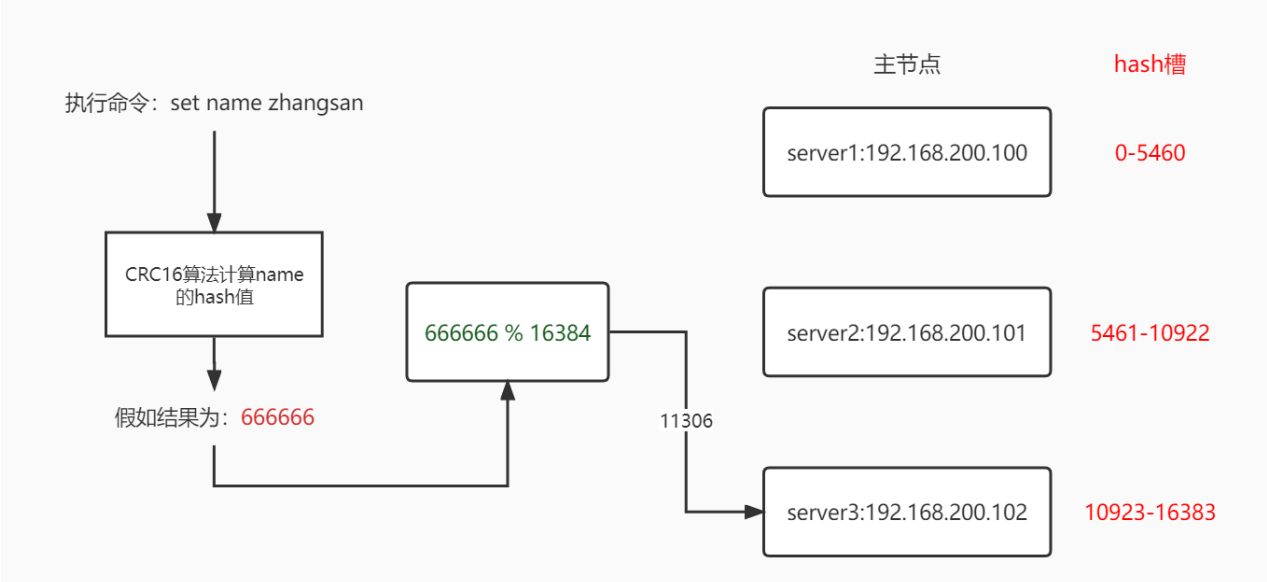
https://www.aliyun.com/product/ebm?spm=5176.19720258.J_2686872250.8.328576f4mDkIu9&scm=20140722.M_4603843.P_156.MO_871-ID_4603843-MID_4603843-CID_545-ST_5018-V_1

3.4 Redis分片集群中数据是怎么存储和读取的？

难易程度：☆☆☆

出现频率：☆☆☆

Redis 集群引入了哈希槽的概念，Redis 集群有 16384 个哈希槽，每个 key通过 CRC16 校验后对 16384 取模来决定放置哪个槽，集群的每个节点负责一部分 hash 槽。



上图是存值的流程，取值的流程类似

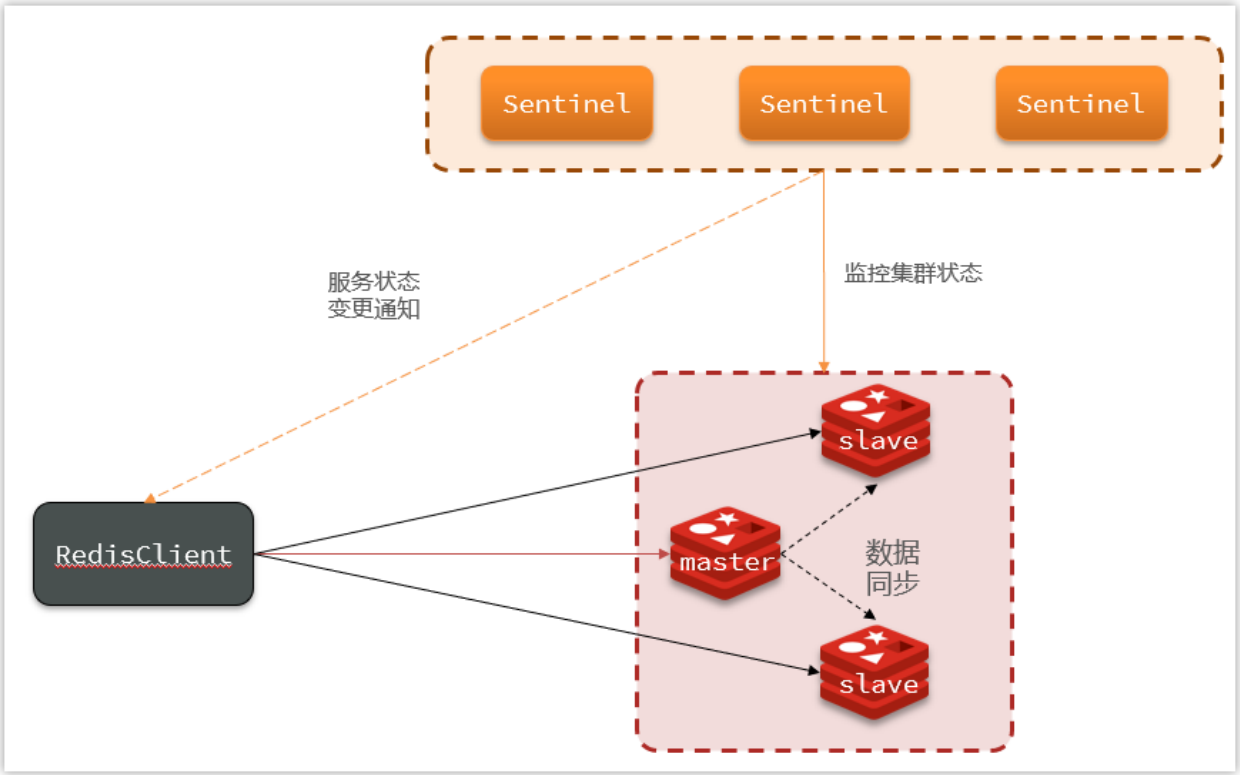
set {aaa}name zhangsan 计算hash是根据aaa计算的

3.5 redis集群脑裂?

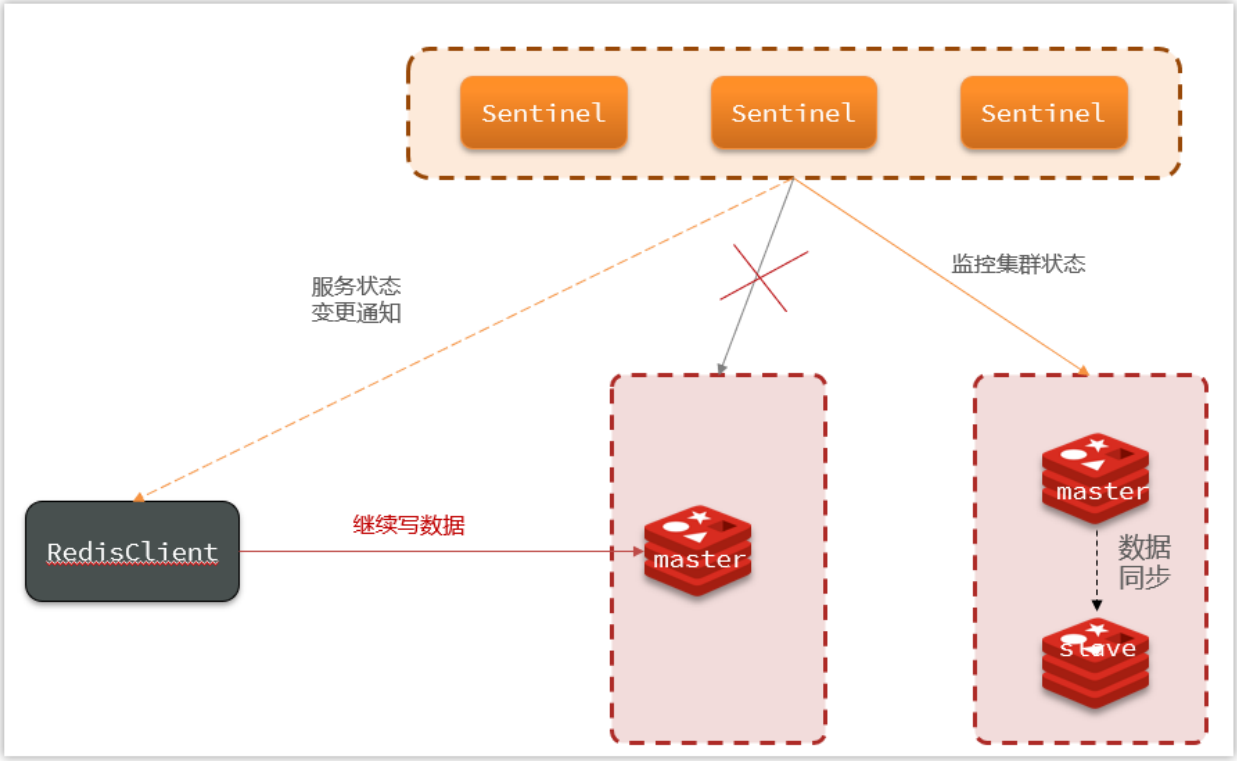
> 难易程度：☆☆☆☆
>
> 出现频率：☆☆☆

关于reids集群会由于网络等原因出现脑裂的情况，所谓的集群脑裂就是，由于redis master节点和redis salve节点和sentinel处于不同的网络分区，使得sentinel没有能够心跳感知到master，所以通过选举的方式提升了一个salve为master，这样就存在了两个master，就像大脑分裂了一样，这样会导致客户端还在old master那里写入数据，新节点无法同步数据，当网络恢复后，sentinel会将old master降为salve，这时再从新master同步数据，这会导致大量数据丢失。

正常情况：

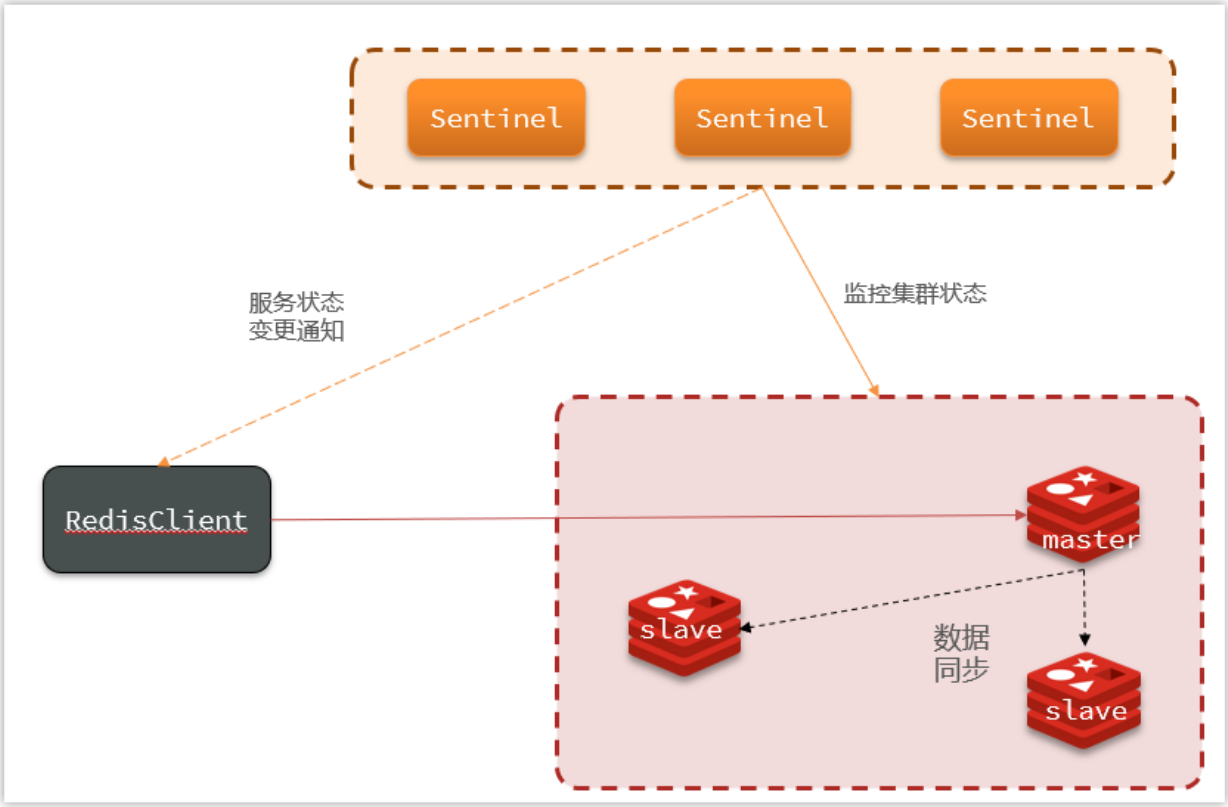


脑裂情况：



> 当哨兵与主节点由于网络抖动原因断开了链接，哨兵监控到之后，则会从剩余的从节点中选出一个作为主节点
>
> redis的客户端这个时候并没有是可以正常链接之前的maser(主节点)，并且可以正常写入数据

假如现在网络恢复了，哨兵发现主从中有两个主节点，则会强制一个主节点变为从节点，看下图



> 由于原来的主节点变成了从节点，则需要执行主从同步流程，清理数据（之前的主节点），同步新主节点中的数据，**在之前脑裂过程中，客户端写入的数据丢失**

解决方案：

redis中有两个配置参数：

- min-replicas-to-write 1 表示最少的salve节点为1个
- min-replicas-max-lag 5 表示数据复制和同步的延迟不能超过5秒

配置了这两个参数：如果发生脑裂：原master会在客户端写入操作的时候拒绝请求。这样可以避免大量数据丢失。

3.6 怎么保证redis的高并发高可用

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

- 主从+哨兵

- 集群

4.使用场景

4.1 项目中哪块使用了缓存？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆☆

结合自己简历上写的项目模块说明这个问题，要陈述出当时的场景

黑马头条：

- 用户行为数据
- 热点文章

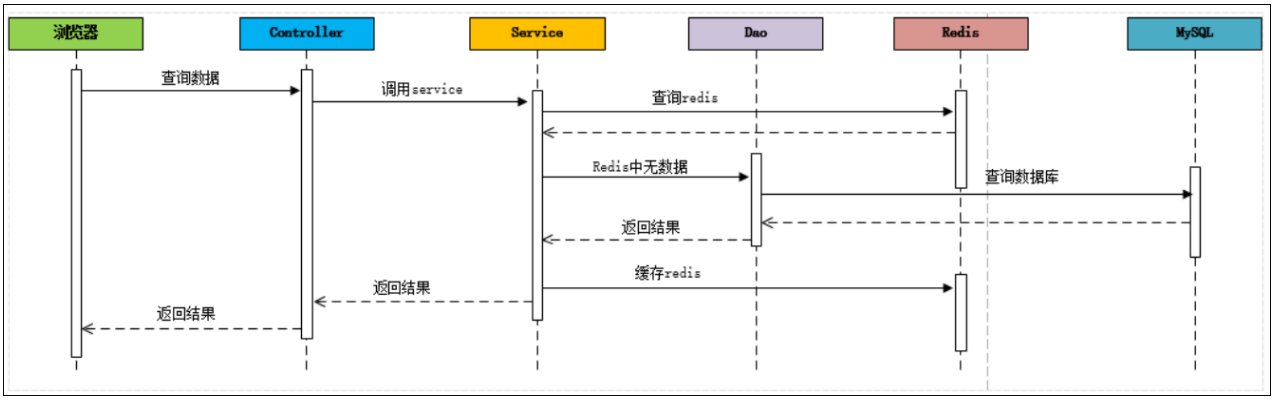
其他：

- 热点数据

4.2 什么是缓存穿透？怎么解决？

> 难易程度：☆☆☆☆
>
> 出现频率：☆☆☆☆☆

加入缓存以后的数据查询流程：



缓存穿透：

概述：指查询一个一定不存在的数据，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 DB 去查询，可能导致 DB 挂掉。

> get请求：api/v1/news/13

解决方案：

1、查询返回的数据为空，仍把这个空结果进行缓存，但过期时间会比较短

2、布隆过滤器：将所有可能存在的数据哈希到一个足够大的 bitmap 中，一个一定不存在的数据会被这个 bitmap 拦截掉，从而避免了对DB的查询

4.3 什么是缓存击穿？怎么解决？

> 难易程度：☆☆☆☆
>
> 出现频率：☆☆☆☆☆

概述：对于设置了过期时间的key，缓存在某个时间点过期的时候，恰好这时间点对这个Key有大量的并发请求过来，这些请求发现缓存过期一般都会从后端 DB 加载数据并回设到缓存，这个时候大并发的请求可能会瞬间把 DB 压垮。

解决方案：

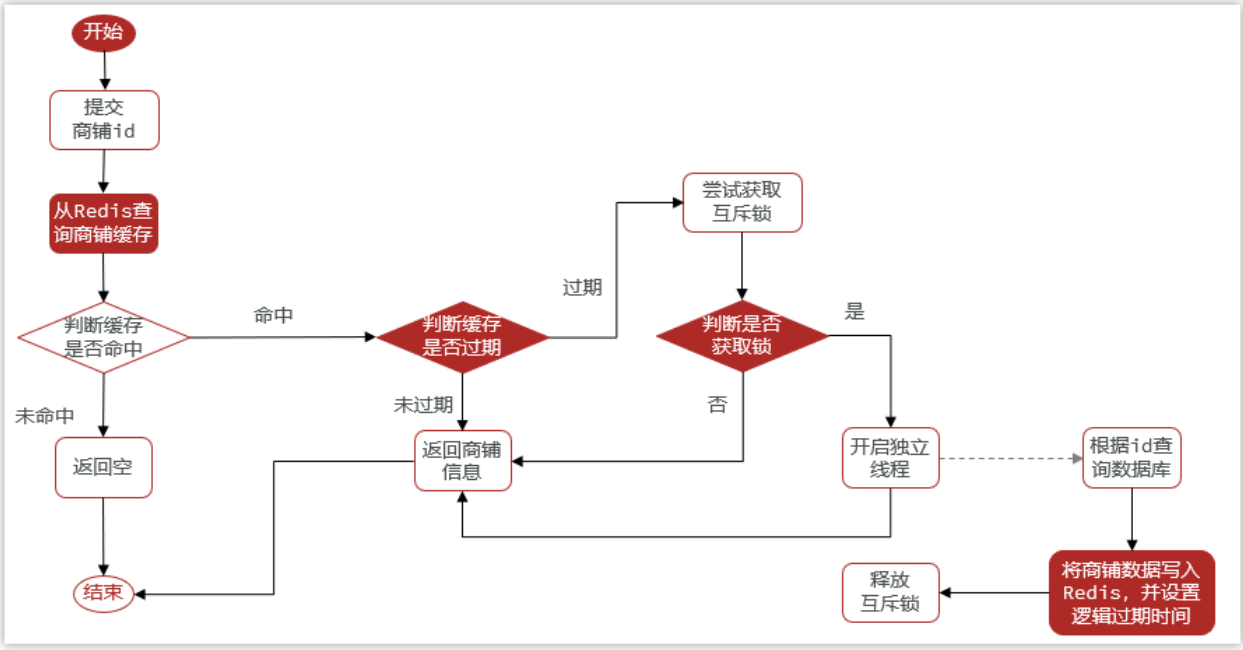
1、使用互斥锁：当缓存失效时，不立即去load db，先使用如 Redis 的 setnx 去设置一个互斥锁，当操作成功返回时再进行 load db的操作并回设缓存，否则重试get缓存的方法

2、可以设置当前key逻辑过期，大概是思路如下：

①：在设置key的时候，设置一个过期时间字段一块存入缓存中，不给当前key设置过期时间

②：当查询的时候，从redis取出数据后判断时间是否过期

③：如果过期则开通另外一个线程进行数据同步，当前线程正常返回数据，这个数据不是最新



两种方案对比：

解决方案	优点	缺点
互斥锁	没有额外的内存消耗 保证一致性 实现简单	线程需要等待，性能受影响 可能有死锁风险
逻辑过期	线程无需等待，性能较好	不保证一致性 有额外内存消耗 实现复杂

4.4 什么是缓存雪崩？怎么解决？

> 难易程度：☆☆☆☆
>
> 出现频率：☆☆☆☆☆

概述：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。与缓存击穿的区别：雪崩是很多key，击穿是某一个key缓存。

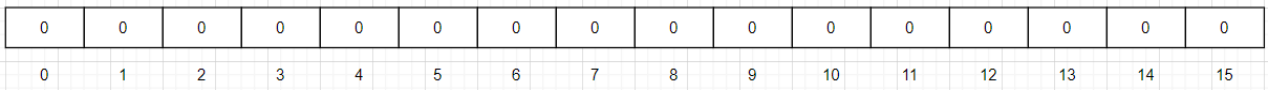
解决方案：

将缓存失效时间分散开，比如可以在原有的失效时间基础上增加一个随机值，比如1-5分钟随机，这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的事件。

4.5 什么是布隆过滤器？

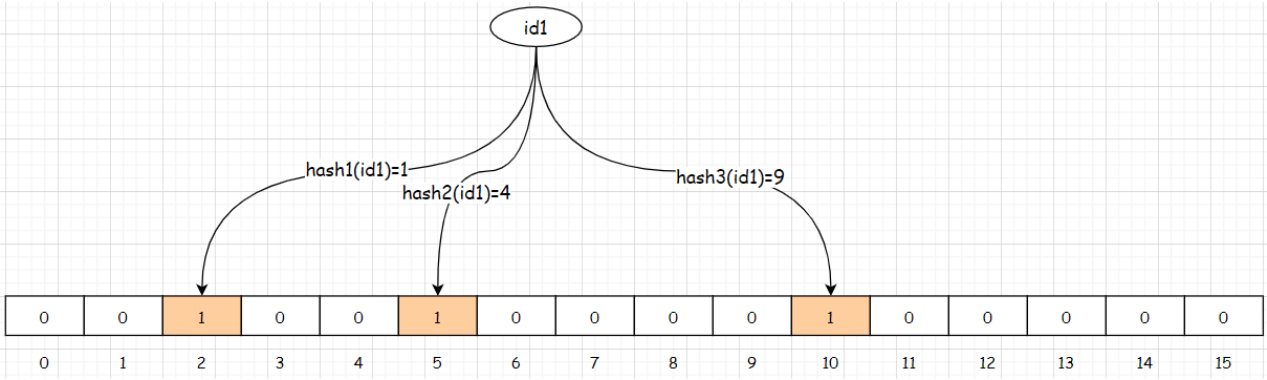
> 难易程度：☆☆☆☆
>
> 出现频率：☆☆☆☆☆

概述：布隆过滤器（Bloom Filter）是1970年由布隆提出的。它实际上由一个很长的二进制向量(二进制数组)和一系列随机映射函数(hash函数)。



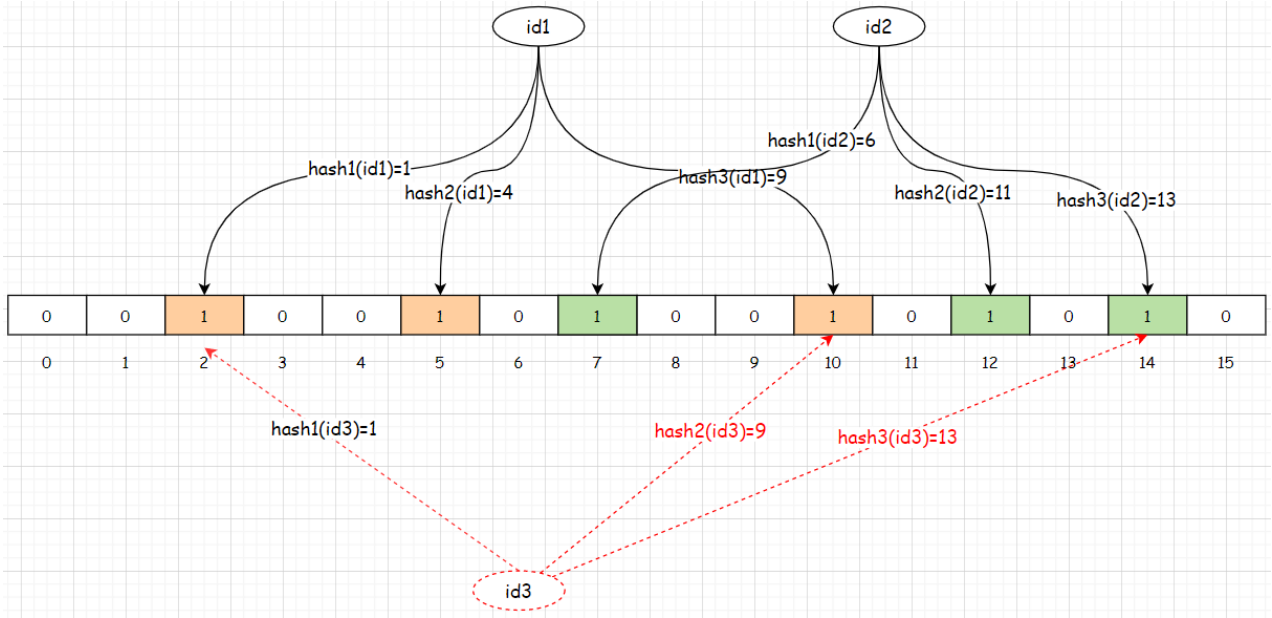
作用：布隆过滤器可以用于检索一个元素是否在一个集合中。

添加元素：将商品的id(id1)存储到布隆过滤器



假设当前的布隆过滤器中提供了三个hash函数，此时就使用三个hash函数对id1进行哈希运算，运算结果分别为：1、4、9那么就会数组中对应的位置数据**更改为1**。

判断数据是否存在：使用相同的hash函数对数据进行哈希运算，得到哈希值。然后判断该哈希值所对应的数组位置是否都为1，如果不都是则说明该数据**肯定不存在**。如果是说明该数据**可能**存在，因为哈希运算可能就会存在重复的情况。如下图所示：



假设添加完id1和id2数据以后，布隆过滤器中数据的存储方式如上图所示，那么此时要判断id3对应的数据在布隆过滤器中是否存在，按照上述的判断规则应该是存在，但是id3这个数据在布隆过滤器中压根就不存在，这种情况就属于误判。

误判率：数组越小误判率就越大，数组越大误判率就越小，但是同时带来了更多的内存消耗。

删除元素：布隆布隆器不支持数据的删除操作，因为如果支持删除那么此时就会影响判断不存在的结果。

使用布隆过滤器：在redis的框架redisson中提供了布隆过滤器的实现，使用方式如下所示：

pom.xml文件

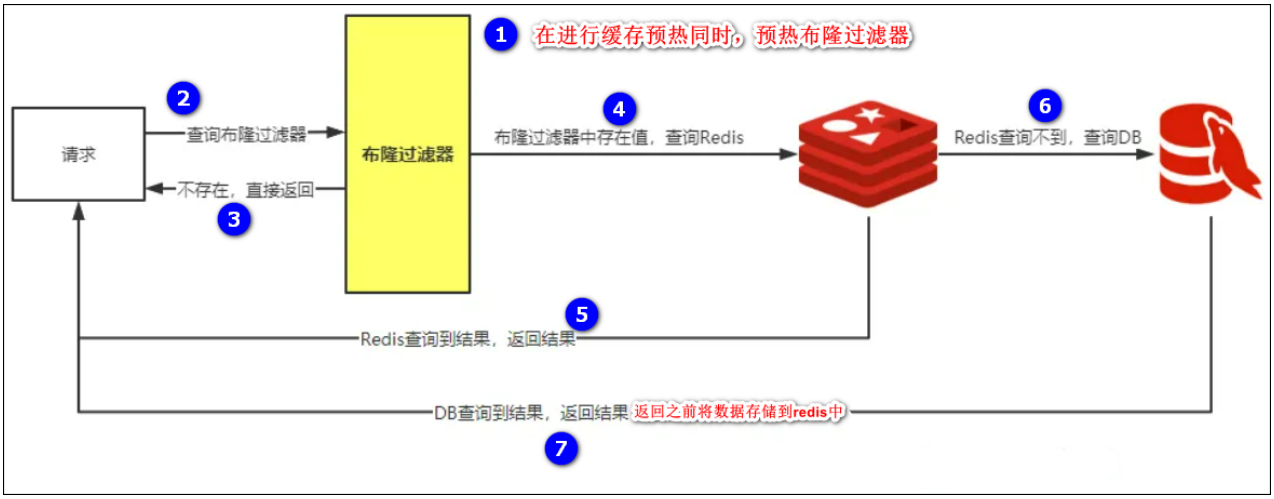
```
<dependency>
    <groupId>org.redisson</groupId>
    <artifactId>redisson</artifactId>
    <version>3.13.6</version>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

测试代码：

```
import org.redisson.Redisson;
import org.redisson.api.RBloomFilter;
import org.redisson.api.RedissonClient;
import org.redisson.config.Config;

public class Application {

    public static void main(String[] args) {
        //链接redis,
        Config config = new Config();

 config.useSingleServer().setAddress("redis://192.168.200.130:6379")
.setPassword("leadnews");
        //创建redisson客户端
        RedissonClient redissonClient = Redisson.create(config);
        //创建布隆过滤器
        RBloomFilter<String> bloomFilter =
redissonClient.getBloomFilter("bloom-filter");

        int size = 10000;

        //初始化数据
//          initData(bloomFilter, size);
        //测试误判率
        int count = getData(bloomFilter, size);
        System.out.println("总的误判条数为: " + count);

    }
```

```
    /**
     * 测试误判率
     * @param bloomFilter
     * @param size
     * @return
     */
    private static int getData(RBloomFilter<String> bloomFilter, int
size) {
        int count = 0 ;      // 记录误判的数据条数
        for(int x = size; x < size * 2 ; x++) {
            if(bloomFilter.contains("add" + x)) {
                count++ ;
            }
        }
        return count;
    }

    /**
     * 初始化数据
     * @param bloomFilter
     * @param size
     */
    private static void initData(RBloomFilter<String> bloomFilter,
int size) {
        //第一个参数：布隆过滤器存储的元素个数
        //第一个参数：误判率
        bloomFilter.tryInit(size,0.01);
        //在布隆过滤器初始化数据
        for(int x = 0; x < size; x++) {
            bloomFilter.add("add" + x) ;
        }
        System.out.println("初始化完成...");
    }
}
```

Redis中使用布隆过滤器防止缓存穿透流程图如下所示：



4.6 redis双写问题?

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆☆

同步方案：

普通缓存，一般采用更新时删除缓存，查询时建立缓存的延迟更新方案。

异步方案：

1、使用消息队列进行缓存同步：更改代码加入异步操作缓存的逻辑代码(数据库操作完毕以后，将要同步的数据发送到MQ中，MQ的消费者从MQ中获取数据，然后更新缓存)

2、使用阿里巴巴旗下的canal组件实现数据同步：不需要更改业务代码，部署一个canal服务。canal服务把自己伪装成mysql的一个从节点，当mysql数据更新以后，canal会读取binlog数据，然后在通过canal的客户端获取到数据，更新缓存即可。

5 Redis分布式锁

5.1 Redis分布式锁如何实现？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆☆

Redis实现分布式锁主要利用Redis的**setnx**命令。setnx是SET if not exists(如果不存在，则 SET)的简写。

```
127.0.0.1:6379> setnx lock value1 #在键lock不存在的情况下，将键key的值设置为value1
(integer) 1
127.0.0.1:6379> setnx lock value2 #试图覆盖lock的值，返回0表示失败
(integer) 0
127.0.0.1:6379> get lock #获取lock的值，验证没有被覆盖
"value1"
127.0.0.1:6379> del lock #删除lock的值，删除成功
(integer) 1
127.0.0.1:6379> setnx lock value2 #再使用setnx命令设置，返回0表示成功
(integer) 1
127.0.0.1:6379> get lock #获取lock的值，验证设置成功
"value2"
```

上面这几个命令就是最基本的用来完成分布式锁的命令。

加锁：使用`setnx key value`命令，如果key不存在，设置value(加锁成功)。如果已经存在lock(也就是有客户端持有锁了)，则设置失败(加锁失败)。

解锁：使用`del`命令，通过删除键值释放锁。释放锁之后，其他客户端可以通过`setnx`命令进行加锁。

```
public boolean tryLock(long timeoutSec) {
    // 获取线程标示
    String threadId = ID_PREFIX + Thread.currentThread().getId();
    // 获取锁
    Boolean success = stringRedisTemplate.opsForValue()
        .setIfAbsent(KEY_PREFIX + name, threadId, timeoutSec,
TimeUnit.SECONDS);
    return Boolean.TRUE.equals(success);
}
```

5.2 Redis实现分布式锁如何合理的控制锁的有效时长？

> 难易程度：☆☆☆☆
>
> 出现频率：☆☆☆☆☆

有效时间设置多长，假如我的业务操作比有效时间长？我的业务代码还没执行完就自动给我解锁了，不就完蛋了吗。

解决方案：

1、第一种：程序员自己去把握，预估一下业务代码需要执行的时间，然后设置有效期时间比执行时间长一些，保证不会因为自动解锁影响到客户端业务代码的执行。

2、第二种：给锁续期。

锁续期实现思路：当加锁成功后，同时开启守护线程，默认有效期是用户所设置的，然后每隔10秒就会给锁续期到用户所设置的有效期，只要持有锁的客户端没有宕机，就能保证一直持有锁，直到业务代码执行完毕由客户端自己解锁，如果宕机了自然就在有效期失效后自动解锁。

上述的第二种解决方案可以使用redis官方所提供的Redisson进行实现。

使用步骤如下：

1、加入依赖

```
<dependency>
    <groupId>org.redisson</groupId>
    <artifactId>redisson</artifactId>
    <version>3.13.6</version>
</dependency>
```

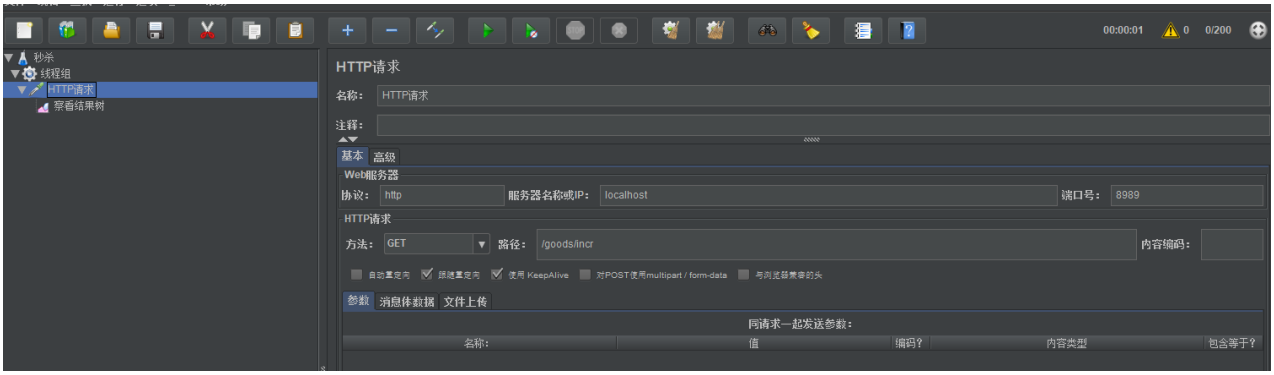
2、定义配置类

```
@Configuration
public class RedisConfig {

    @Bean
    public RedissonClient redissonClient(){
        Config config = new Config();

 config.useSingleServer().setAddress("redis://192.168.200.130:6379")
.setPassword("leadnews");
        return Redisson.create(config);
    }
}
```

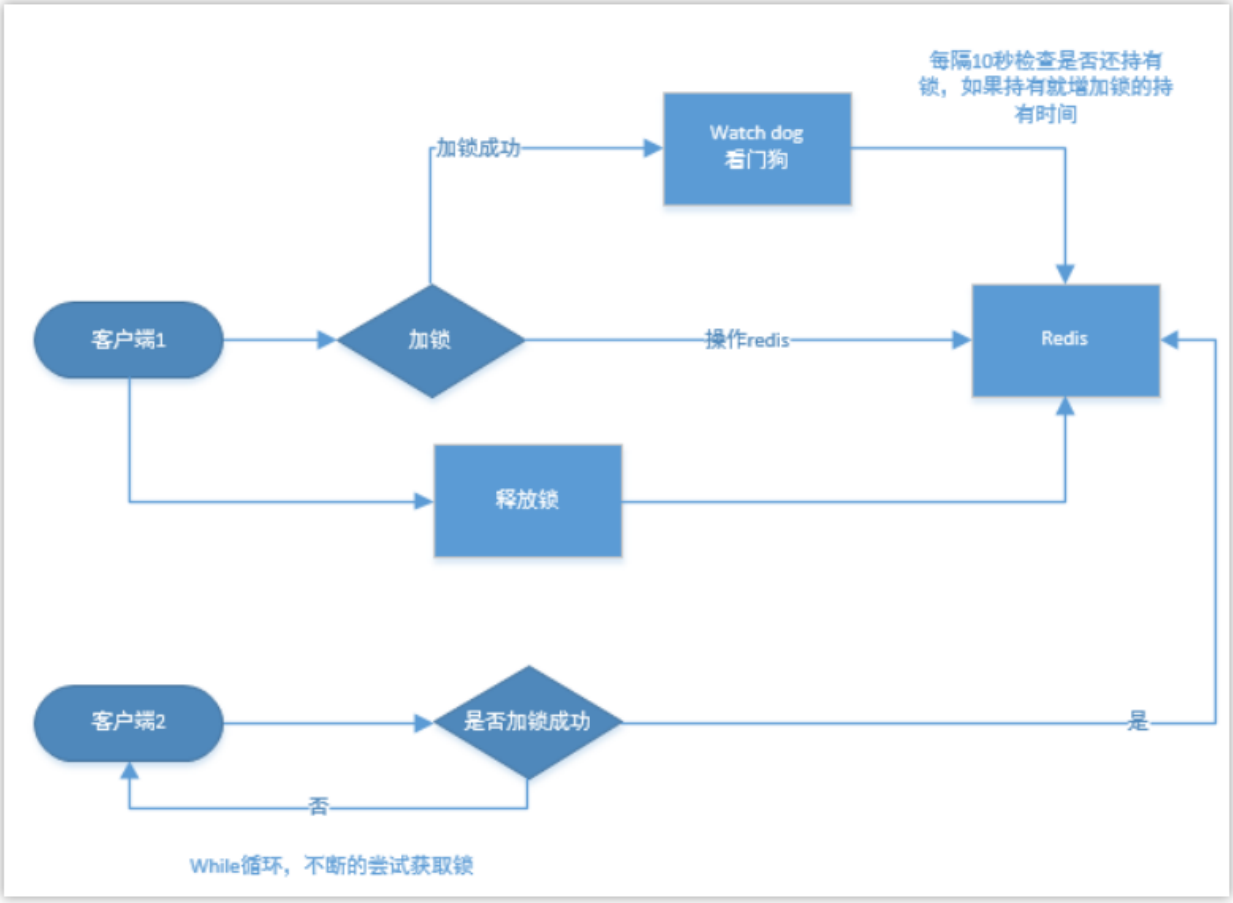
3、业务代码加入分布式锁

```
public void redisLock() throws InterruptedException {

    RLock lock = redissonClient.getLock("anyLock");
    try {
        //第一个参数30s:表示尝试获取分布式锁，并且最大的等待获取锁的时间
为30s
        //第二个参数10s:表示上锁之后，10s内操作完毕将自动释放锁
        boolean isLock = lock.tryLock(30, 30, TimeUnit.SECONDS);
        String num = redisTemplate.opsForValue().get("num");
        Integer intNum = Integer.parseInt(num);
        if (intNum == null || intNum <= 0) {
            throw new RuntimeException("商品已抢完");
        }
        if(isLock){
            intNum = intNum - 1;
            redisTemplate.opsForValue().set("num",
intNum.toString());

 System.out.println(redisTemplate.opsForValue().get("num"));
        }
    } finally {
        //释放锁
        lock.unlock();
    }

}
```

4、Jmeter测试



5.Redisson分布式锁原理（重要）



> 注意：Redisson的watchDog不是每10秒做一次续期，而是每隔(releaseTime / 3)的时间做一次续期。也就是锁自动释放时间的1/3，默认的锁释放时间是30秒，因此默认每隔10秒续期。

redisson实现的分布式锁是可重入的

```
public void add1(){

    boolean isLock = lock.tryLock(30, 10, TimeUnit.SECONDS);
```

```
    //执行业务
    add2();

    //释放锁
    lock.unlock();

}

public void add2(){

    boolean isLock = lock.tryLock(30, 10, TimeUnit.SECONDS);
    //执行业务

    //释放锁
    lock.unlock();

}
```

5.3 你的项目中哪里用到了分布式锁？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆☆

黑马头条：

- 集群情况下定时任务（分布式锁）

立可得项目：

- 下单（分布式锁）

其他情况：

- 缓存
- 秒杀
- 幂等性场景

6.其他

6.1 Redis的数据过期策略有哪些？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆

数据删除策略：Redis中可以对数据设置数据的有效时间，数据的有效时间到了以后，就需要将数据从内存中删除掉。而删除的时候就需要按照指定的规则进行删除，这种删除规则就被称之为数据的删除策略。

Redis中数据的删除策略：

① 惰性删除

概述：设置该key过期时间后，我们不去管它，当需要该key时，我们在检查其是否过期，如果过期，我们就删掉它，反之返回该key。

- 优点：对CPU友好，我们只会在使用该键时才会进行过期检查，对于很多用不到的key不用浪费时间进行过期检查。
- 缺点：对内存不友好，如果一个键已经过期，但是一直没有使用，那么该键就会一直存在内存中，如果数据库中有很多这种使用不到的过期键，这些键便永远不会被删除，内存永远不会释放。

> set name zhangsan 10
>
> get name 发现name过期了，直接删除key

② 定期删除

概述：每隔一段时间，我们就对一些key进行检查，删除里面过期的key(从一定数量的数据库中取出一定数量的随机键进行检查，并删除其中的过期键)。

- 优点：可以通过限制删除操作执行的时长和频率来减少删除操作对 CPU 的影响。另外定期删除，也能有效释放过期键占用的内存。
- 缺点：难以确定删除操作执行的时长和频率。

如果执行的太频繁，定期删除策略变得和定时删除策略一样，对CPU不友好。如果执行的太少，那又和惰性删除一样了，过期键占用的内存不会及时得到释放。

另外最重要的是，在获取某个键时，如果某个键的过期时间已经到了，但是还没执行定期删除，那么就会返回这个键的值，这是业务不能忍受的错误。

定期清理的两种模式：

- SLOW模式是定时任务，执行频率默认为10hz，每次不超过25ms，以通过修改配置文件redis.conf 的 **hz** 选项来调整这个次数
- FAST模式执行频率不固定，每次事件循环会尝试执行，但两次间隔不低于2ms，每次耗时不超过1ms

Redis的过期删除策略：**惰性删除 + 定期删除**两种策略进行配合使用定期删除函数的运行频率

6.2 Redis的数据淘汰策略有哪些？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆

数据的淘汰策略：当Redis中的内存不够用时，此时在向Redis中添加新的key，那么Redis就会按照某一种规则将内存中的数据删除掉，这种数据的删除规则被称之为内存的淘汰策略。

常见的数据淘汰策略：

```
noeviction                          # 不删除任何数据，内存不足直接报错
(默认策略)
volatile-lru                        # 挑选最近最久使用的数据淘汰(举例:
key1是在3s之前访问的，key2是在9s之前访问的，删除的就是key2)
volatile-lfu                        # 挑选最近最少使用数据淘汰 （举
例：key1最近5s访问了4次，key2最近5s访问了9次， 删除的就是key1)
volatile-ttl                        # 挑选将要过期的数据淘汰
volatile-random                     # 任意选择数据淘汰
allkeys-lru                         # 挑选最近最少使用的数据淘汰
allkeys-lfu                         # 挑选最近使用次数最少的数据淘汰
allkeys-random                      # 任意选择数据淘汰，相当于随机
```

> - **LRU**（**L**east **R**ecently **U**sed）最少最近使用。用当前时间减去最后一次访问时间，这个值越大则淘汰优先级越高。
> - **LFU**（**L**east **F**requently **U**sed）最少频率使用。会统计每个key的访问频率，值越小淘汰优先级越高。
> - **volatile**：表示设置了带过期时间的key
> - **allkeys**：表示所有的key

缓存淘汰策略常见配置项

```
maxmemory-policy noeviction         # 配置淘汰策略
maxmemory ?mb                       # 最大可使用内存，即占用物理内存的
比例，默认值为0，表示不限制。生产环境中根据需求设定，通常设置在50%以上。
maxmemory-samples count             # 设置redis需要检查key的个数
```

使用建议：

1. 优先使用 allkeys-lru 策略。充分利用 LRU 算法的优势，把最近最常访问的数据留在缓存中。如果业务有明显的冷热数据区分，建议使用。
2. 如果业务中数据访问频率差别不大，没有明显冷热数据区分，建议使用 allkeys-random，随机选择淘汰。
3. 如果业务中有置顶的需求，可以使用 volatile-lru 策略，同时置顶数据不设置过期时间，这些数据就一直不被删除，会淘汰其他设置过期时间的数据。
4. 如果业务中有短时高频访问的数据，可以使用 allkeys-lfu 或 volatile-lfu 策略。

6.3 数据库有1000万数据 ,Redis只能缓存20w数据, 如何保证Redis中的数据都是热点数据？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

使用 allkeys-lru （挑选最近最少使用的数据淘汰）淘汰策略，那留下来的都是经常访问的热点数据

6.4 Redis的内存用完了会发生什么？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆

主要看数据淘汰策略是什么？如果是默认的配置，则直接报错

6.5 你们用过Redis的事务吗？事务的命令有哪些？

> 难易程度：☆☆☆☆
>
> 出现频率：☆☆

Redis的事务与传统事务不同。传统事务一般ACID事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。

Redis中的事务：Redis事务的本质是一组命令的集合。事务支持一次执行多个命令，一个事务中所有命令都会被序列化。在事务执行过程，会按照顺序串行化执行队列中的命令，其他客户端提交的命令请求不会插入到事务执行命令序列中。

总结说：Redis事务就是一次性、顺序性、排他性的执行一个队列中的一系列命令。Reids中，单条命令式原子性执行的，但事务不保证原子性，且没有回滚。

事务相关的命令：

1、MULTI：用来组装一个事务

2、EXEC：执行一个事物

3、DISCARD：取消一个事务

4、WATCH：用来监视一些key，一旦这些key在事务执行之前被改变，则取消事务的执行

5、UNWATCH：取消 WATCH 命令对所有key的监视

如下所示：

6.6 Redis是单线程的，但是为什么还那么快？

> 难易程度：☆☆☆
>
> 出现频率：☆☆☆☆

Redis总体快的原因：

1、完全基于内存的，C语言编写

2、采用单线程，避免不必要的上下文切换可竞争条件

3、数据简单，数据操作也相对简单

4、使用多路I/O复用模型，非阻塞IO

bgsave 在后台执行rdb的保存，不影响主线程的正常使用，不会产生阻塞

bgrewriteaof 在后台执行aof文件的保存，不影响主线程的正常使用，不会产生阻塞

7 面试现场

7.1 数据类型

面试官：Redis的常用数据类型有哪些？

候选人：

嗯，这个有很多，不过我们开发比较常见的有5种，分别是：

Redis是典型的“键值型”数据库，不同数据类型其key结构一致，value有所差异。常见的类型有：string、hash、list、set、SortedSet等。

当然基于以上5种基本数据类型，Redis又拓展了几种拓展类型，例如：BitMap、HyperLogLog、Geo等。

----------增强回答----------

- String类型是Redis中最常见的数据类型，value与key一样都是Redis自定义的字符串结构，称为SDS。不过在保存数字、小字符串时因为采用INT和EMBSTR编码，内存结构紧凑，只需要申请一次内存分配，效率更高，更节省内存。

而超过44字节的大字符串时则需要采用RAW编码，申请额外的SDS空间，需要两次内存分配，效率较低，内存占用也较高，但最大不超过512mb，因此建议单个value尽量不要超过44字节。

String类型常用来做计数器、简单数据存储等。复杂数据建议采用其它数据结构。

- Hash结构，其value与Java中的HashMap类似，有是一个key-value结构。如果有一个对象需要被Redis缓存，而且将来可能有部分修改。建议用hash结构来存储这个

对象的每一个字段和字段值。而不是作为一个JSON字符串存储到String类型中。因为Hash结构的每一个字段都可以单独做修改，而String的JSON串必须整体覆盖。

与Java中的hashMap不同的是，Redis中的Hash底层采用了渐进式rehash的算法，在做rehash时会创建一个新的hashtable，每次操作元素时移动一部分数据，只到所有数据迁移完成，再用新的HashTable来代替旧的，避免了因为rehash导致的阻塞，因此性能更高。

- List结构的value类型可以看做是一个双端链表，提供了一些命令便于我们从首尾操作元素。为了节省内存空间，底层采用了ZipList（压缩列表）来做基础存储。当压缩列表数据达到阈值（512）则会创建新的压缩列表。每个压缩列表作为一个双端链表的一个节点，最终形成一个QuickList结构。而且QuickList结构与一般的双端链表不同，他可以对中间不常用的ZipList节点做压缩以节省内存。

List结构常用来模拟队列，实现任务排队这样的功能。

- Set结构的value与Java的Set类似，元素不可重复。Redis提供了求交集、并集等命令，可以帮助我们实现例如：好友列表、共同好友等功能。
  当存储元素是整数时，其底层默认采用IntSet结构，可以看做是一个有序的数组，结构紧凑，效率较高。而元素如果不是整数，或者元素量超过512这个阈值时则会转为hash表结构，内存占用会有大的增加。因此我们在使用Set结构时尽量采用数组存储，例如数值类型的id。而且元素数量尽量不要超过512，避免出现BigKey。
- SortedSet，也叫ZSet。其value就是一个有序的Set集合，元素唯一，并且会按照一个指定的score值排序。因此常用来做排行榜功能。

SortedSet底层的利用Hash表保证元素的唯一性。利用跳表（SkipList）来保证元素的有序性，因此数据会有重复存储，内存占用较高，是一种典型的以空间换时间的设计。不建议在SortedSet中放入过多数据。

面试官：跳表你了解吗？

候选人：

嗯，这个知道一些~~

Redis数据类型Sorted Set使用了跳表作为其中一种数据结构，跳表结合了链表和二分查找的思想

它的底层是一个有序的双向列表，但不同于其他双向列表的是，它有多个指针，指针的跨度也不同，查找时从顶层向下，不断缩小范围。效率相对比较高，它的时间复杂度为 O (log n) 。

7.2 持久化

面试官：Redis的数据持久化策略有哪些？

候选人：

嗯~~，在Redis中提供了两种数据持久化的方式：1、RDB 2、AOF

RDB：

定期更新，定期将Redis中的数据生成的快照同步到磁盘等介质上，磁盘上保存的就是Redis的内存快照

优点：数据文件的大小相比于aop较小，使用rdb进行数据恢复速度较快

缺点：比较耗时，存在丢失数据的风险

AOF：

将Redis所执行过的所有指令都记录下来，在下次Redis重启时，只需要执行指令就可以了

优点：数据丢失的风险大大降低了

缺点：数据文件的大小相比于rdb较大，使用aop文件进行数据恢复的时候速度较慢

面试官：好的，那在你们的项目中采用了哪种持久化策略？

候选人：

嗯，是这样，我们项目中为了能够预防更少的数据丢失，两种方式都用了，RDB+AOF

7.3 主从和集群

面试官：Redis集群有哪些方案, 知道嘛？

候选人：

嗯~~，在Redis中提供的集群方案总共有三种：主从复制、哨兵模式、Redis分片集群

1、主从复制

- 保证高可用性
- 实现故障转移需要手动实现
- 无法实现海量数据存储

2、哨兵模式

- 保证高可用性
- 可以实现自动化的故障转移
- 无法实现海量数据存储

3、Redis分片集群

- 保证高可用性
- 可以实现自动化的故障转移
- 可以实现海量数据存储

面试官：什么是 Redis 主从同步？

候选人：

这个主要指的是redis的主节点向从节点同步数据的过程。一共分为了两个阶段

第一阶段，全量同步流程
1.从节点执行replicaof命令，发送自己的replid和offset给主节点
2.主节点判断从节点的replid与自己的是否一致，
3.如果不一致说明是第一次来，需要做全量同步，主节点返回自己的replid给从节点
4.主节点开始执行bgsave，生成rdb文件
5.主节点发送rdb文件给从节点，再发送的过程中
6.从节点接收rdb文件，清空本地数据，加载rdb文件中的数据
7.同步过程中，主节点接收到的新命令写入从节点的写缓冲区(repl_buffer)
8.从节点接收到缓冲区数据后写入本地，并记录最新数据对应的offset
9.后期采用增量同步

第二阶段，增量同步数据

1.主节点会不断把自己接收到的命令记录在repl_baklog中，并修改offset
2.从节点向主节点发送psync命令，发送自己的offset和replid
3.主节点判断replid和offset与从节点是否一致
4.如果replid一致，说明是增量同步。然后判断offset是否一致
5.如果从节点offset小于主节点offset，并且在repl_baklog中能找到对应数据
  则将offset之间相差的数据发送给从节点
6.从节点接收到数据后写入本地，修改自己的offset与主节点一致

当然repl_baklog的大小是有上限的，我们也可以通过配置修改大小

面试官：你们使用Redis是单点还是集群？哪种集群？(说说你们生产环境redis部署情况？)

候选人：

嗯~~，是这样的，因为我们项目使用缓存的业务量不是超级大，所以采用了哨兵模式，机器一共有5台，主从2台（1主1从），哨兵3台。

面试官：你感觉集群模式有什么问题吗？

候选人：

嗯~~~，集群模式是可以解决大数据量的存储，但也存在一些问题，比如

- 维护起来比较麻烦
- 集群之间的心跳检测和数据通信会消耗大量的网络带宽
- 集群插槽分配不均和key的分批容易导致数据倾斜
- 客户端的route会有性能损耗
- 集群模式下无法使用lua脚本、事务

面试官：Redis分片集群中数据是怎么存储和读取的？

候选人：

嗯~，在redis集群中是这样的

Redis 集群引入了哈希槽的概念，有 16384 个哈希槽，集群中每个主节点绑定了一定范围的哈希槽范围， key通过 CRC16 校验后对 16384 取模来决定放置哪个槽，通过槽找到对应的节点进行存储。

取值的逻辑是一样的

面试官：redis集群脑裂?

候选人：

嗯！ 这个在项目很少见，原因是这样的，我们现在用的是redis的哨兵模式集群的

有的时候由于网络等原因可能会出现脑裂的情况，就是说，由于redis master节点和redis salve节点和sentinel处于不同的网络分区，使得sentinel没有能够心跳感知到master，所以通过选举的方式提升了一个salve为master，这样就存在了两个master，就像大脑分裂了一样，这样会导致客户端还在old master那里写入数据，新节点无法同步数据，当网络恢复后，sentinel会将old master降为salve，这时再从新master同步数据，这会导致old master中的大量数据丢失。

关于解决的话，我记得在redis的配置中可以设置

第一可以设置最少的salve节点个数，

第二可以设置数据复制和同步的延迟时间

面试官：怎么保证redis的高并发高可用

候选人：

参考回答一：

嗯，由于我们的存入redis的业务量是有限的，为了能够承受突发的高并发请求，我们公司采用的主从架构和哨兵模式

参考回答二：

嗯，由于我们的存入redis的业务量很大，为了能够承受住高并发请求，我们公司采用的集群模式

7.4使用场景

面试官：项目中哪块使用了缓存？

本文作者：结合自己简历上写的项目模块说明这个问题，要陈述出当时的场景

候选人：

嗯，有很多的~~

我负责的黑马头条项目（你不能写），在app端的用户的行为数据我们是存储到了缓存中的，就是redis。

还有就是在自媒体端，自媒体人发布文章的时候可以设置一个未来时间定时发布文章，我们采用的redis的zset作为延迟队列用的。

还有一块是，在app端的热点文章，因为这块内容相对请求数量比较多，我们会把一些热点数据缓存到redis。减轻一大部分访问的压力。

我们的很多的定时任务，都是晚上自动跑，但是我们的服务都是集群，为了防止每个集群都执行这个定时任务，我们当时使用了redis实现的分布式来限制的（setnx）

立可得项目使用场景：

- 工单排行榜
- 下单（分布式锁）

面试官：什么是缓存穿透？怎么解决？

候选人：

嗯~~，我想一下

缓存穿透是指查询一个一定不存在的数据，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 DB 去查询，可能导致 DB 挂掉。这种情况大概率是遭到了攻击。

解决方案的话，我们通常都会用布隆过滤器来解决它

面试官：好的，你能介绍一下布隆过滤器吗？

候选人：

嗯，是这样~

布隆过滤器主要是用于检索一个元素是否在一个集合中。我们当时使用的是redisson实现的布隆过滤器。

它的底层主要是先去初始化一个比较大数组，里面存放的二进制0或1。在一开始都是0，当一个key来了之后经过3次hash计算，模于数组长度找到数据的下标然后把数组中原来的0改为1，这样的话，三个数组的位置就能标明一个key的存在。查找的过程也是一样的。

当然是有缺点的，布隆过滤器有可能会产生一定的误判，我们一般可以设置这个误判率，大概不会超过5%，其实这个误判是必然存在的，要不就得增加数组的长度，其实已经算是很划分了，5%以内的误判率一般的项目也能接受，不至于高并发下压倒数据库。

面试官：什么是缓存击穿？怎么解决？

候选人：

嗯！！

缓存击穿的意思是对于设置了过期时间的key，缓存在某个时间点过期的时候，恰好这时间点对这个Key有大量的并发请求过来，这些请求发现缓存过期一般都会从后端 DB 加载数据并回设到缓存，这个时候大并发的请求可能会瞬间把 DB 压垮。

解决方案有两种方式：

第一可以使用互斥锁：当缓存失效时，不立即去load db，先使用如 Redis 的 setnx 去设置一个互斥锁，当操作成功返回时再进行 load db的操作并回设缓存，否则重试get缓存的方法

第二种方案可以设置当前key逻辑过期，大概是思路如下：

①：在设置key的时候，设置一个过期时间字段一块存入缓存中，不给当前key设置过期时间

②：当查询的时候，从redis取出数据后判断时间是否过期

③：如果过期则开通另外一个线程进行数据同步，当前线程正常返回数据，这个数据不是最新

当然两种方案各有利弊：

如果选择数据的强一致性，建议使用分布式锁的方案，性能上可能没那么高，锁需要等，也有可能产生死锁的问题

如果选择key的逻辑删除，则优先考虑的高可用性，性能比较高，但是数据同步这块做不到强一致。

面试官：什么是缓存雪崩？怎么解决？

候选人：

嗯！！

缓存雪崩意思是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。与缓存击穿的区别：雪崩是很多key，击穿是某一个key缓存。

解决方案主要是可以将缓存失效时间分散开，比如可以在原有的失效时间基础上增加一个随机值，比如1-5分钟随机，这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的事件。

面试官：redis双写问题?

候选人：

嗯！我们当时采用的阿里巴巴旗下的canal组件实现数据同步：不需要更改业务代码，部署一个canal服务。canal服务把自己伪装成mysql的一个从节点，当mysql数据更新以后，canal会读取binlog数据，然后在通过canal的客户端获取到数据，更新缓存即可。

7.5 Redis分布式锁

面试官：Redis分布式锁如何实现？

候选人：

嗯，在redis中提供了一个命令setnx(SET if not exists)

由于redis的单线程的，用了命令之后，只能有一个客户端对某一个key设置值，在没有过期或删除key的时候是其他客户端是不能设置这个key的

面试官：好的，那你如何控制Redis实现分布式锁有效时长呢？

候选人：

嗯，的确，redis的setnx指令不好控制这个问题，我们当时采用的redis的一个框架redisson实现的。

在redisson中需要手动加锁，并且可以控制锁的失效时间和等待时间，当锁住的一个业务还没有执行完成的时候，在redisson中引入了一个看门狗机制，就是说每隔一段时间就检查当前业务是否还持有锁，如果持有就增加加锁的持有时间，当业务执行完成之后需要使用释放锁就可以了

还有一个好处就是，在高并发下，一个业务有可能会执行很快，先客户1持有锁的时候，客户2来了以后并不会马上拒绝，它会自选不断尝试获取锁，如果客户1释放之后，客户2就可以马上持有锁，性能也得到了提升。

面试官：好的，redisson实现的分布式锁是可重入的吗？

候选人：

嗯，是可以重入的。这样做是为了避免死锁的产生。这个重入其实在内部就是判断是否是当前线程持有的锁，如果是当前线程持有的锁就会计数，如果释放锁就会在计算上减一。

面试官：你的项目中哪里用到了分布式锁 ？

候选人：

本文作者：这个也要根据自己简历上的项目进行描述

常见的业务：集群情况下定时任务（分布式锁）、秒杀、幂等性场景

7.6 其他

面试官：Redis的数据过期策略有哪些？

候选人：

嗯~，在redis中提供了两种数据过期删除策略

第一种是惰性删除，在设置该key过期时间后，我们不去管它，当需要该key时，我们在检查其是否过期，如果过期，我们就删掉它，反之返回该key。

第二种是 定期删除，就是说每隔一段时间，我们就对一些key进行检查，删除里面过期的key

定期清理的两种模式：

- SLOW模式是定时任务，执行频率默认为10hz，每次不超过25ms，以通过修改配置文件redis.conf 的 **hz** 选项来调整这个次数
- FAST模式执行频率不固定，每次事件循环会尝试执行，但两次间隔不低于2ms，每次耗时不超过1ms

Redis的过期删除策略：**惰性删除 + 定期删除**两种策略进行配合使用定期删除函数的运行频率。

面试官：Redis的数据淘汰策略有哪些？

候选人：

嗯，这个在redis中提供了很多种，默认是noeviction，不上任何数据，内部不足直接报错

当然在线上的redis都会进行设置数据淘汰策略：比较常见是

对有所key的操作有三个：

- allkeys-lru 挑选最近最少使用的数据淘汰
- allkeys-lfu 挑选最近使用次数最少的数据淘汰
- allkeys-random 随机选择数据淘汰

对设置了过期key的操作有四个：

- volatile-lru 挑选最近最久使用的数据淘汰
- volatile-lfu 挑选最近最少使用数据淘汰
- volatile-ttl 挑选将要过期的数据淘汰
- volatile-random 任意选择数据淘汰

我们在项目通常设置的allkeys-lru，挑选最近最少使用的数据淘汰，把一些经常访问的key留在redis中

面试官：数据库有1000万数据 ,Redis只能缓存20w数据, 如何保证Redis中的数据都是热点数据？

候选人：

嗯，我想一下~~

可以使用 allkeys-lru （挑选最近最少使用的数据淘汰）淘汰策略，那留下来的都是经常访问的热点数据

面试官：Redis的内存用完了会发生什么？

候选人：

嗯~，这个要看redis的数据淘汰策略是什么，如果是默认的配置，redis内存用完以后则直接报错。我们当时设置的 allkeys-lru 策略。把最近最常访问的数据留在缓存中。

面试官：你们用过Redis的事务吗？事务的命令有哪些？

候选人：

嗯，这个没在项目中用过，不过我知道一些。

Redis的事务与传统事务不同。传统事务一般ACID事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行

Redis事务的本质是一组命令的集合。事务支持一次执行多个命令，一个事务中所有命令都会被序列化。在事务执行过程，会按照顺序串行化执行队列中的命令，其他客户端提交的命令请求不会插入到事务执行命令序列中。单条命令式原子性执行的，但事务不保证原子性，且没有回滚。

关于事务的命令有：

1、MULTI：用来组装一个事务

2、EXEC：执行一个事物

3、DISCARD：取消一个事务

4、WATCH：用来监视一些key，一旦这些key在事务执行之前被改变，则取消事务的执行

5、UNWATCH：取消 WATCH 命令对所有key的监视

面试官：Redis是单线程的，但是为什么还那么快？

候选人：

嗯，这个有几个原因吧~~~

1、完全基于内存的，C语言编写

2、采用单线程，避免不必要的上下文切换可竞争条件

3、使用多路I/O复用模型，非阻塞IO

例如：bgsave 和 bgrewriteaof 都是在后台执行操作，不影响主线程的正常使用，不会产生阻塞