2019
版

治愈系 Java 工程师面试指导课程

[Part7・NoSql 数据库和 Redis]

[本教程梳理了 NoSql 数据库和 Redis 的复习脉络，并且涵盖了 redis 的很多高级特性以及高频面试题；建议大家将目录结构打开，对照目录结构做一个复 习的、提纲准备面试。]

1. NoSql 介绍

1.1. 什么是 NoSql

传统的关系数据库在应付 web2.0 网站，特别是超大规模和高并发的 SNS 类型的 web2.0 纯动态网站已经显得力不从心，暴露了很多难以克服的问题，例如：

1、High performance - 对数据库高并发读写的需求

2、Huge Storage - 对海量数据的高效率存储和访问的需求

3、High Scalability && High Availability- 对数据库的高可扩展性和高可用性的需求

NoSQL 数据库的产生就是为了解决高并发、高可扩展、高可用、大数据存储问题而产生的数据库解决方案，就是 NoSql 数据库。

NoSQL，泛指非关系型的数据库，NoSQL 即 Not-Only SQL，它可以作为关系型数据库的良好补充。

1.2. Nosql 数据库分类

■ 键值(Key-Value)存储数据库

相关产品： Tokyo Cabinet/Tyrant、**Redis**、Voldemort、Berkeley DB、memcached

典型应用： 内容缓存，主要用于处理大量数据的高访问负载。

数据模型： 一系列键值对

优势： 快速查询

劣势： 存储的数据缺少结构化

■ 列存储数据库

相关产品：Cassandra, **HBase**, Riak

典型应用：分布式的文件系统 bson

数据模型：以列簇式存储，将同一列数据存在一起

优势：查找速度快，可扩展性强，更容易进行分布式扩展

劣势：功能相对局限

■ 文档型数据库

相关产品：CouchDB、**MongoDB**

典型应用：Web 应用（与 Key-Value 类似，Value 是结构化的）

数据模型： 一系列键值对

优势：数据结构要求不严格

劣势： 查询性能不高，而且缺乏统一的查询语法

■ 图形(Graph)数据库

相关数据库：Neo4J、InfoGrid、Infinite Graph

典型应用：社交网络

数据模型：图结构

优势：利用图结构相关算法。

劣势：需要对整个图做计算才能得出结果，不容易做分布式的集群方案。

尚硅谷Java工程师面试指导课程 - samuel

2. redis 基础知识

简单来说 redis 就是一个数据库，不过与传统数据库不同的是 redis 的数据是存在内存中的，所以存写速度非常快，因此 redis 被广泛应用于缓存方向。另外，redis 也经常用来做分布式锁。redis 提供了多种数据类型来支持不同的业务场景。除此之外，redis 支持事务 、持久化、LUA 脚本、LRU 驱动事件、多种集群方案。

2.1. 为什么要用 redis /为什么要用缓存

主要从“高性能”和“高并发”这两点来看待这个问题。

2.1.1. 高性能

假如用户第一次访问数据库中的某些数据。这个过程会比较慢，因为是从硬盘上读取的。将该用户访问的数据存在数缓存中，这样下一次再访问这些数据的时候就可以直接从缓存中获取了。操作缓存就是直接操作内存，所以速度相当快。如果数据库中的对应数据改变的之后，同步改变缓存中相应的数据即可！

2.1.2. 高并发

直接操作缓存能够承受的请求是远远大于直接访问数据库的，所以我们可以考虑把数据库中的部分数据转移到缓存中去，这样用户的一部分请求会直接到缓存这里而不用经过数据库。

2.2. 为什么要用 redis 而不用 map 做缓存?

缓存分为本地缓存和分布式缓存。以 Java 为例，使用自带的 map 或者 guava 实现的是本地缓存，最主要的特点是轻量以及快速，生命周期随着 jvm 的销毁而结束，并且在多实例的情况下，每个实例都需要各自保存一份缓存，缓存不具有一致性。

使用 redis 或 memcached 之类的称为分布式缓存，在多实例的情况下，各实例共用一份缓存数据，缓存具有一致性。缺点是需要保持 redis 或 memcached 服务的高可用，整个程序架构上较为复杂。

2.3. redis 和 memcached 的区别

对于 redis 和 memcached 我总结了下面四点。现在公司一般都是用 redis 来实现缓存，而且 redis 自身也越来越强大了！

治愈系Java工程师面试指导课程——samuel

对比参数	Redis	Memcached
类型	1、支持内存 2、非关系型数据库	1、支持内存 2、key-value键值对形式 3、缓存系统
数据存储类型	1、String 2、List 3、Set 4、Hash 5、Sort Set 【俗称ZSet】	1、文本型 2、二进制类型【新版增加】
查询【操作】类型	1、批量操作 2、事务支持【虽然是假的事务】 3、每个类型不同的CRUD	1、CRUD 2、少量的其他命令
附加功能	1、发布/订阅模式 2、主从分区 3、序列化支持 4、脚本支持【Lua脚本】	1、多线程服务支持
网络IO模型	1、单进程模式	2、多线程、非阻塞IO模式
事件库	自封装简易事件库AeEvent	贵族血统的LibEvent事件库
持久化支持	1、RDB 2、AOF	不支持

- redis 支持更丰富的数据类型（支持更复杂的应用场景）：Redis 不仅仅支持简单的 k/v 类型的数据，同时还提供 list，set，zset，hash 等数据结构的存储。memcache 支持简单的数据类型，String。

- Redis 支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用,而 Memecache 把数据全部存在内存之中。
- 集群模式：memcached 没有原生的集群模式，需要依靠客户端来实现往集群中分片写入数据；但是 redis 目前是原生支持 cluster 模式的.
- Memcached 是多线程，非阻塞 IO 复用的网络模型；Redis 使用单线程的多路 IO 复用模型。

2.4. redis 常见数据结构以及使用场景分析

2.4.1. String

常用命令: set,get,decr,incr,mget 等。
String 数据结构是简单的 key-value 类型，value 其实不仅可以是 String，也可以是数字。 常规 key-value 缓存应用；**常规计数：微博数，粉丝数**等。

2.4.2. Hash

常用命令： hget,hset,hgetall 等。
Hash 是一个 string 类型的 field 和 value 的映射表，hash 特别适合用于存储对象，后续操作的时候，你可以直接仅仅修改这个对象中的某个字段的值。 比如我们可以 Hash 数据结构来存储用户信息，商品信息等等。

2.4.3. List

常用命令: lpush,rpush,lpop,rpop,lrange 等

list 就是链表，Redis list 的应用场景非常多，也是 Redis 最重要的数据结构之一，比如微博的关注列表，粉丝列表，消息列表等功能都可以用 Redis 的 list 结构来实现。

Redis list 的实现为一个双向链表，即可以支持反向查找和遍历，更方便操作，不过带来了部分额外的内存开销。

另外可以通过 lrange 命令，就是从某个元素开始读取多少个元素，可以基于 list 实现分页查询，这个很棒的一个功能，基于 redis 实现简单的高性能分页，可以做类似微博那种下拉不断分页的东西（一页一页的往下走），性能高。

2.4.4. Set

常用命令： sadd,spop,smembers,sunion 等

set 对外提供的功能与 list 类似是一个列表的功能，特殊之处在于 set 是可以自动排重的。

当你需要存储一个列表数据，又不希望出现重复数据时，set 是一个很好的选择，并且 set 提供了判断某个成员是否在一个 set 集合内的重要接口，这个也是 list 所不能提供的。可以基于 set 轻易实现交集、并集、差集的操作。

比如：在微博应用中，可以将一个用户所有的关注人存在一个集合中，将其所有粉丝存在一个集合。Redis 可以非常方便的实现如共同关注、共同粉丝、共同喜好等功能。这个过程也就是求交集的过程，具体命令如下：sinterstore key1 key2 key3 将交集存在 key1 内

2.4.5. Sorted Set

常用命令： zadd,zrange,zrem,zcard 等

和 set 相比，sorted set 增加了一个权重参数 score，使得集合中的元素能够按 score 进行有序排列。

举例： 在直播系统中，实时排行信息包含直播间在线用户列表，各种礼物排行榜，弹幕消息（可以理解为按消息维度的消息排行榜）等信息，适合使用 Redis 中的 SortedSet 结构进行存储。

2.5. redis 设置过期时间

Redis 中有个设置时间过期的功能，即对存储在 redis 数据库中的值可以设置一个过期时间。作为一个缓存数据库，这是非常实用的。如我们一般项目中的 token 或者一些登录信息，尤其是**短信验证码**都是有时间限制的，按照传统的数据库处理方式，一般都是自己判断过期，这样无疑会严重影响项目性能。

我们 **set key 的时候，都可以给一个 expire time**，就是过期时间，通过过期时间我们可以指定这个 key 可以存活的时间。

如果假设你设置了一批 key 只能存活 1 个小时，那么接下来 1 小时后，redis 是怎么对这批 key 进行删除的？定期删除+惰性删除。

价值万Java工程师面试指导课程——samuel

2.5.1. 删除策略

- **定期删除**：redis 默认是每隔 100ms 就随机抽取一些设置了过期时间的 key，检查其是否过期，如果过期就删除。**注意这里是随机抽取的**。
- **惰性删除** ：定期删除可能会导致很多过期 key 到了时间并没有被删除掉。所以就有了惰性删除。假如你的过期 key，靠定期删除没有被删除掉，还停留在内存里，除非你的系统去查一下那个 key，才会被 redis 给删除掉。这就是所谓的惰性删除！
- **存在问题及解决方法**

但是仅仅通过设置过期时间还是有问题的。我们想一下：如果定期删除漏掉了很多过期 key，然后你也没及时去查，也就没走惰性删除，此时会怎么样？如果大量过期 key 堆积在内存里，导致 redis 内存块耗尽了。怎么解决这个问题呢？-------内存淘汰策略

2.5.2. redis 内存淘汰机制。

redis 提供 6 种数据淘汰策略：

1. volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰
2. volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰
3. volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰
4. allkeys-lru：当内存不足以容纳新写入数据时，在键空间中，移除最近最少使用的 key（这个是最常用的）.
5. allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰
6. no-eviction：禁止驱逐数据，也就是说当内存不足以容纳新写入数据时，新写入操作会报错。这个应该没人使用吧！

2.6. redis 持久化机制

很多时候我们需要持久化数据也就是将内存中的数据写入到硬盘里面，大部分原因是为了之后重用数据（比如重启机器、机器故障之后回复数据），或者是为了防止系统故障而将数据备份到一个远程位置。Redis 支持两种不同的持久化操作。Redis 的一种持久化方式叫快照（snapshotting，RDB）,另一种方式是只追加文件（append-only file,AOF）.

2.6.1. 快照（snapshotting）持久化（RDB）

Redis 可以通过创建快照来获得存储在内存里面的数据在某个时间点上的副本。Redis 创建快照之后，可以对快照进行备份，可以将快照复制到其他服务器从而创建具有相同数据的服务器副本（Redis 主从结构，主要用来提高 Redis 性能），还可以将快照留在原地以便重启服务器的时候使用。

RDB 持久化是 Redis 默认采用的持久化方式，在 redis.conf 配置文件中默认有此下配置：

沁意系Java工程师面试指导课程——samuel

```
save 900 1                  #在900秒(15分钟)之后，如果至少有1个key发生变化，Redis就会自动触发BGSAVE命令
创建快照。

save 300 10                 #在300秒(5分钟)之后，如果至少有10个key发生变化，Redis就会自动触发BGSAVE命令
创建快照。

save 60 10000           #在60秒(1分钟)之后，如果至少有10000个key发生变化，Redis就会自动触发BGSAVE命令
创建快照。
```

默认每隔 5 分钟创建一个快照副本，这种方式占用空间大，而且会丢失间隔时间 5 分钟之内的数据，但是他适合做备份，恢复时，可以根据需要恢复任意间隔时间点的数据

2.6.2. AOF（append-only file）持久化

与 RDB 持久化相比，AOF 持久化 的实时性更好，因此已成为主流的持久化方案。默认情况下 Redis 没有开启 AOF（append only file）方式的持久化，可以通过 appendonly 参数开启：

```
appendonly yes
```

开启 AOF 持久化后每执行一条会更改 Redis 中的数据的命令，Redis 就会将该命令写入硬盘中的 AOF 文件。AOF 文件的

在 Redis 的配置文件中存在三种不同的 AOF 持久化方式，它们分别是：

- appendfsync always #每次有数据修改发生时都会写入 AOF 文件,这样会严重降低 Redis 的速度
- appendfsync everysec #每秒钟同步一次，显示地将多个写命令同步到硬盘
- appendfsync no #让操作系统决定何时进行同步
- 为了兼顾数据和写入性能，用户可以考虑 appendfsync everysec 选项 ，让 Redis 每秒同步一次 AOF 文件，Redis 性能几乎没受到任何影响。而且这样即使出现系统崩溃，用户最多只会丢失一秒之内产生的数据。当硬盘忙于执行写入操作的时候，Redis 还会优雅的放慢自己的速度以便适应硬盘的最大写入速度。

2.6.3. Redis 4.0 对于持久化机制的优化

Redis 4.0 开始支持 RDB 和 AOF 的混合持久化(**默认关闭，可以通过配置项 aof-use-rdb-preamble 开启**)。如果把混合持久化打开，AOF 重写的时候就直接把 RDB 的内容写到 AOF 文件开头。这样做的好处是可以结合 RDB 和 AOF 的优点, 快速加载同时避免丢失过多的数据。当然缺点也是有的， AOF 里面的 RDB 部分是压缩格式不再是 AOF 格式，可读性较差。

2.6.4. 补充内容：AOF 重写

AOF 重写可以产生一个新的 AOF 文件，这个新的 AOF 文件和原有的 AOF 文件所保存的数据库状态一样，但体积更小。AOF 重写是一个有歧义的名字，该功能是通过读取数据库中的键值对来实现的，程序无须对现有 AOF 文件进行任何读入、分析或者写入操作。

在执行 BGREWRITEAOF 命令时，Redis 服务器会维护一个 AOF 重写缓冲区，该缓冲区会在子进程创建新 AOF 文件期间，记录服务器执行的所有写命令。当子进程完成创建新 AOF 文件的工作之后，服务器会将重写缓冲区中的所有内容追加到新 AOF 文件的末尾，使得新旧两个 AOF 文件所保存的数据库状态一致。最后，

服务器用新的 AOF 文件替换旧的 AOF 文件，以此来完成 AOF 文件重写操作

2.7. redis 事务

Redis 通过 MULTI、EXEC、WATCH 等命令来实现事务(transaction)功能。事务提供了一种将多个命令请求打包，然后一次性、按顺序地执行多个命令的机制，并且在事务执行期间，服务器不会中断事务而改去执行其他客户端的命令请求，它会将事务中的所有命令都执行完毕，然后才去处理其他客户端的命令请求。

在传统的关系式数据库中，常常用 ACID 性质来检验事务功能的可靠性和安全性。在 Redis 中，事务总是具有原子性（Atomicity)、一致性(Consistency)和隔离性（Isolation），并且当 Redis 运行在某种特定的持久化模式下时，事务也具有持久性（Durability)。

redis 对事务的支持是指可以一次执行多个命令，本质是一组命令的集合。一个事务中的所有命令都会序列化，按顺序地串行化执行而不会被其它命令插入，不许加塞。

涉及的命令:

multi : 开启一个事务

exec: 执行一个事务

discard:取消事务，如果被 watch 监控 取消 watch

watch: 监控一个 key 或 多个 key 是否发送变化

unwatch: 取消监控

在项目中，我们经常配合使用 watch + multi + exec + discard 达到一个乐观锁的目的处理页面,比如在秒杀项目中我们的预减库存操作就是利用这个方式，用 watch 监控一个商品库存，开启事务尝试修改库存，如果在我们修改期间我们的这条数据被其它人修改过，那么这个事务就会提交不成功，达到一个乐观锁的目的。 redis 处理事务的机制稍弱，需要我们在代码中多加控制。

3. Redis 使用过程中遇到的具体问题

3.1. 缓存雪崩和缓存穿透问题解决方案

3.1.1. 缓存雪崩

- 简介：缓存同一时间大面积的失效，所以，后面的请求都会落到数据库上，造成数据库短时间内承受大量请求而崩掉。
- 解决办法
  - 事前：尽量保证整个 redis 集群的高可用性，发现机器宕机尽快补上。选择合适的内存淘汰策略。
  - 事中：本地 ehcache 缓存 + hystrix 限流&降级，避免 MySQL 崩掉
  - 事后：利用 redis 持久化机制保存的数据尽快恢复缓存

沟壑无Java工程师面试指导课程——samuel

3.1.2. 缓存穿透

- 简介：一般是黑客故意去请求缓存中不存在的数据，导致所有的请求都落到数据库上，造成数据库短时间内承受大量请求而崩掉。
- 解决办法： 有很多种方法可以有效地解决缓存穿透问题，最常见的则是采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的 bitmap 中，一个一定不存在的数据会被 这个 bitmap 拦截掉，从而避免了对底层存储系统的查询压力。另外也有一个更为简单粗暴的方法（我们采用的就是这种），如果一个查询返回的数据为空（不管是数 据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。

3.2. 如何解决 Redis 的并发竞争 Key 问题

- 所谓 Redis 的并发竞争 Key 的问题也就是多个系统同时对一个 key 进行操作，但是最后执行的顺序和我们期望的顺序不同，这样也就导致了结果的不同！

- 解决方案：分布式锁（zookeeper 和 redis 都可以实现分布式锁）。（如果不存在 Redis 的并发竞争 Key 问题，不要使用分布式锁，这样会影响性能）
  基于 zookeeper 临时有序节点可以实现的分布式锁。大致思想为：每个客户端对某个方法加锁时，在 zookeeper 上的与该方法对应的指定节点的目录下，生成一个唯一的瞬时有序节点。 判断是否获取锁的方式很简单，只需要判断有序节点中序号最小的一个。 当释放锁的时候，只需将这个瞬时节点删除即可。同时，其可以避免服务宕机导致的锁无法释放，而产生的死锁问题。完成业务流程后，删除对应的子节点释放锁。
  在实践中，当然是从以可靠性为主。所以首推 Zookeeper。

3.3. 如何保证缓存与数据库双写时的数据一致性？

你只要用缓存，就可能会涉及到缓存与数据库双存储双写，你只要是双写，就一定会有数据一致性的问题，那么你如何解决一致性问题？

一般来说，就是如果你的系统不是严格要求缓存+数据库必须一致性的话，最好不要做这个方案。如果不得不做这种方案时，那么可以使用读请求和写请求串行化，串到一个内存队列里去，这样就可以保证一定不会出现不一致的情况串行化之后，就会导致系统的吞吐量会大幅度的降低，用比正常情况下多几倍的机器去支撑线上的一个请求。

4. 主从复制（读写分离）

主从复制的好处有 2 点：
1、 避免 redis 单点故障
2、 构建读写分离架构，满足读多写少的应用场景

分享Java工程师面试指导课程——samuel

4.1. 复制的过程原理

1、 当从库和主库建立 MS 关系后，会向主数据库发送 SYNC 命令；
2、 主库接收到 SYNC 命令后会开始在后台保存快照（RDB 持久化过程），并将期间接收到的写命令缓存起来；
3、 当快照完成后，主 Redis 会将快照文件和所有缓存的写命令发送给从 Redis；
4、 从 Redis 接收到后，会载入快照文件并且执行收到的缓存的命令；
5、 之后，主 Redis 每当接收到写命令时就会将命令发送从 Redis，从而保证数据的一致；

4.2. 无磁盘复制

通过前面的复制过程我们了解到，主库接收到 SYNC 的命令时会执行 RDB 过程，即使在配置文件中禁用 RDB 持久化也会生成，那么如果主库所在的服务器磁盘 IO 性能较差，那么这个复制过程就会出现瓶颈，庆幸的是，Redis 在 2.8.18 版本开始实现了无磁盘复制功能（不过该功能还是处于试验阶段）。

原理：
Redis 在与从数据库进行复制初始化时将不会将快照存储到磁盘，而是直接通过网络发送给从数据库，避免了 IO 性能差问题。

开启无磁盘复制：repl-diskless-sync yes

4.3. 复制架构中出现宕机情况，怎么办？

如果在主从复制架构中出现宕机的情况，需要分情况看：
1、 从 Redis 宕机
   a) 这个相对而言比较简单，在 Redis 中从库重新启动后会自动加入到主从架构中，自动完成同步数据；
   b) 问题？ 如果从库在断开期间，主库的变化不大，从库再次启动后，主库依然会将所有的数据做 RDB 操作吗？还是增量更新？（从库有做持久化的前提下）
      i. 不会的，因为在 Redis2.8 版本后就实现了，主从断线后恢复的情况下实现增量复制。
2、 主 Redis 宕机
   a) 这个相对而言就会复杂一些，需要以下 2 步才能完成
      i. 第一步，在从数据库中执行 SLAVEOF NO ONE 命令，断开主从关系并且提升为主库继续服务；
      ii. 第二步，将主库重新启动后，执行 SLAVEOF 命令，将其设置为其他库的从库，这时数据就能更新回来；
   b) 这个手动完成恢复的过程其实是比较麻烦的并且容易出错，有没有好办法解决呢？当前有的，Redis 提高的哨兵（sentinel）的功能。

沿途学Java工程师面试指导课程 samuel

5. 哨兵（sentinel）

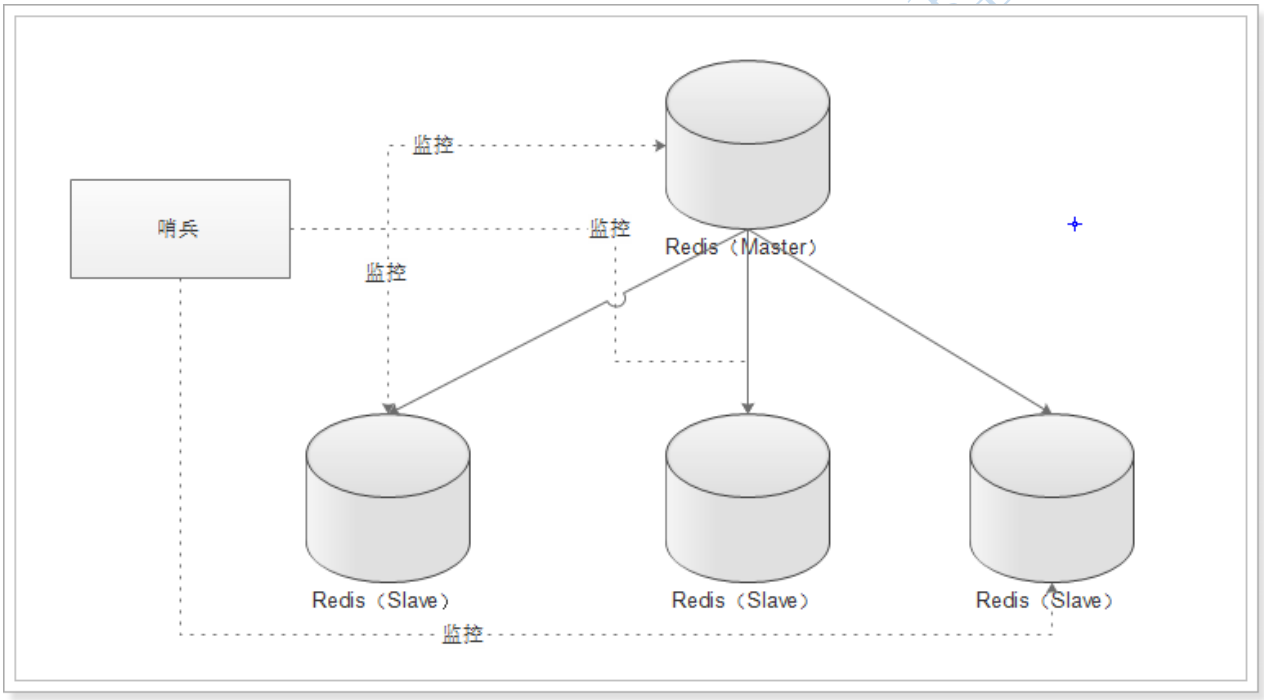
5.1. 什么是哨兵

顾名思义，哨兵的作用就是对 Redis 的系统的运行情况的监控，它是一个独立进程。它的功能有 2 个：

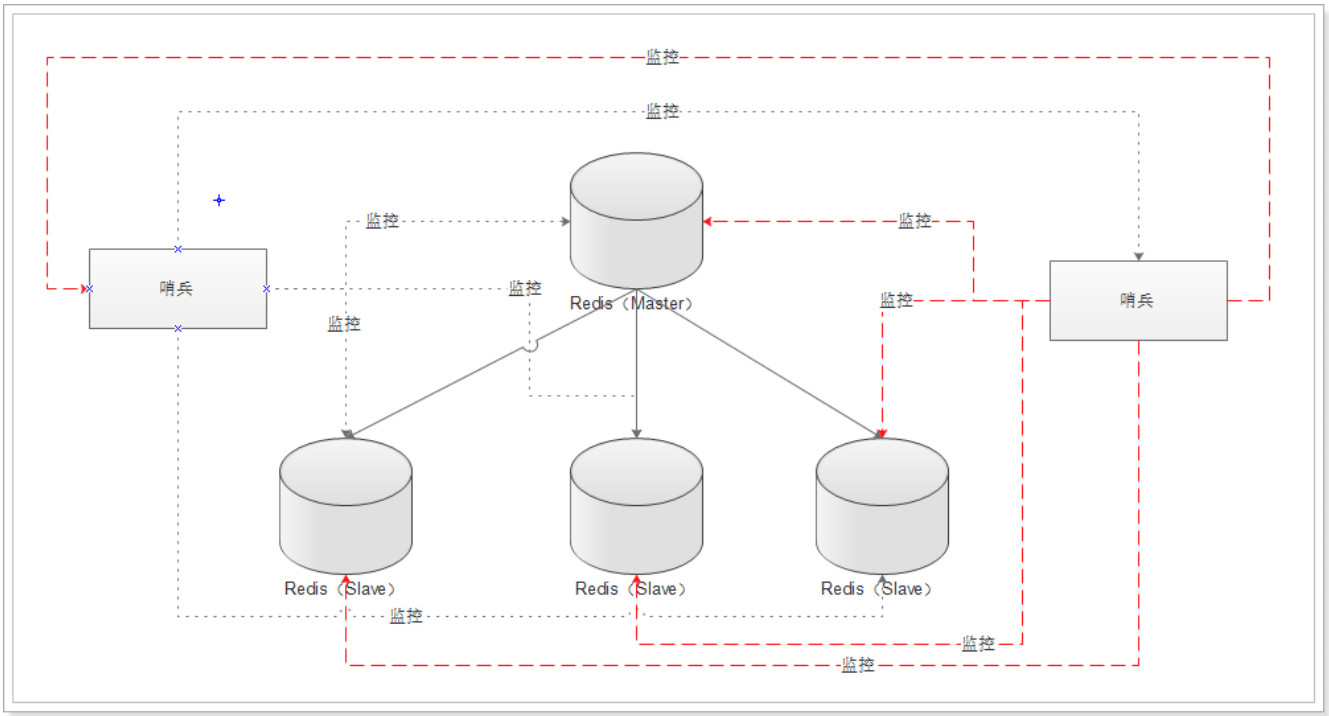
1、 监控主数据库和从数据库是否运行正常；
2、 主数据出现故障后自动将从数据库转化为主数据库；

5.2. 原理

单个哨兵的架构：



多个哨兵的架构：



多个哨兵，不仅同时监控主从数据库，而且哨兵之间互为监控。

5.3. 环境

当前处于一主多从的环境中：

```
127.0.0.1:6379> INFO replication
# Replication
role:master
connected_slaves:2
slave0:ip=127.0.0.1,port=6380,state=online,offset=5279,lag=0
slave1:ip=127.0.0.1,port=6381,state=online,offset=5279,lag=0
master_repl_offset:5279
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:2
repl_backlog_histlen:5278
127.0.0.1:6379>
```

5.4. 配置哨兵

启动哨兵进程首先需要创建哨兵配置文件：

vim sentinel.conf
输入内容：
sentinel monitor e3Master 127.0.0.1 6379 1

说明：
e3Master：监控主数据的名称，自定义即可，可以使用大小写字母和“.-_”符号
127.0.0.1：监控的主数据库的 IP
6379：监控的主数据库的端口
1：最低通过票数

启动哨兵进程：
redis-sentinel ./sentinel.conf

```
[root@taotao2 redis]# redis-sentinel ./sentinel.conf
2989:X 05 Jun 19:56:55.053 * Increased maximum number of open files to 10032 (it was originally set to 1024).
                _._
           _.-``__ ''-._
      _.-``    `.  `_.  ''-._           Redis 3.0.1 (00000000/0) 64 bit
  .-`` .-```.  ```\/    _.,_ ''-._
 (    '      ,       .-`  | `,    )     Running in sentinel mode
 |`-._`-...-` __...-.``-._|'` _.-'|     Port: 26379
 |    `-._   `._    /     _.-'    |     PID: 2989
  `-._    `-._  `-./  _.-'    _.-'
 |`-._`-._    `-.__.-'    _.-'_.-'|
 |    `-._`-._        _.-'_.-'    |           http://redis.io
  `-._    `-._`-.__.-'_.-'    _.-'
 |`-._`-._    `-.__.-'    _.-'_.-'|
 |    `-._`-._        _.-'_.-'    |
  `-._    `-._`-.__.-'_.-'    _.-'
      `-._    `-.__.-'    _.-'
          `-._        _.-'
              `-.__.-'

2989:X 05 Jun 19:56:55.053 # Sentinel runid is 9059917216012421e8e89a4aa02f15b75346d2b7
2989:X 05 Jun 19:56:55.053 # +monitor master taotaoMaster 127.0.0.1 6379 quorum 1
2989:X 05 Jun 19:56:55.055 * +slave slave 127.0.0.1:6380 127.0.0.1 6380 @ taotaoMaster 127.0.0.1 6379
2989:X 05 Jun 19:56:55.066 * +slave slave 127.0.0.1:6381 127.0.0.1 6381 @ taotaoMaster 127.0.0.1 6379
```

由上图可以看到：
1、 哨兵已经启动，它的 id 为 9059917216012421e8e89a4aa02f15b75346d2b7
2、 为 master 数据库添加了一个监控
3、 发现了 2 个 slave（由此可以看出，哨兵无需配置 slave，只需要指定 master，哨兵会自动发现 slave）

5.5. 从数据库宕机

```
[root@taotao2 ~]# ps -ef|grep redis
root      2615     1  0 18:43 ?        00:00:04 redis-server *:6379
root      2626     1  0 18:44 ?        00:00:04 redis-server *:6380
root      2863     1  0 19:32 ?        00:00:01 redis-server *:6381
root      2989  1428  0 19:56 pts/0    00:00:01 redis-sentinel *:26379 [sentinel]
root      3076  3056  0 20:06 pts/1    00:00:00 grep redis
[root@taotao2 ~]#
```

kill 掉 2826 进程后，30 秒后哨兵的控制台输出：

2989:X 05 Jun 20:09:33.509 # +sdown slave 127.0.0.1:6380 127.0.0.1 6380 @ e3Master 127.0.0.1 6379

说明已经监控到 slave 宕机了，那么，如果我们将 3380 端口的 redis 实例启动后，会自动加入到主从复制吗？

治愈系Java工程师面试指导课程——samuel

2989:X 05 Jun 20:13:22.716 * +reboot slave 127.0.0.1:6380 127.0.0.1 6380 @ e3Master 127.0.0.1 6379
2989:X 05 Jun 20:13:22.788 # -sdown slave 127.0.0.1:6380 127.0.0.1 6380 @ e3Master 127.0.0.1 6379

可以看出，slave 从新加入到了主从复制中。-sdown：说明是恢复服务。

```
127.0.0.1:6379> INFO replication
# Replication
role:master
connected_slaves:2
slave0:ip=127.0.0.1,port=6381,state=online,offset=83514,lag=1
slave1:ip=127.0.0.1,port=6380,state=online,offset=83514,lag=1
master_repl_offset:83514
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:2
repl_backlog_histlen:83513
```

5.6. 主库宕机

哨兵控制台打印出如下信息：

2989:X 05 Jun 20:16:50.300 # +sdown master e3Master 127.0.0.1 6379 说明 master 服务已经宕机
2989:X 05 Jun 20:16:50.300 # +odown master e3Master 127.0.0.1 6379 #quorum 1/1
2989:X 05 Jun 20:16:50.300 # +new-epoch 1
2989:X 05 Jun 20:16:50.300 # +try-failover master e3Master 127.0.0.1 6379 开始恢复故障
2989:X 05 Jun 20:16:50.304 # +vote-for-leader 9059917216012421e8e89a4aa02f15b75346d2b7 1 投票选举哨兵 leader，现在就一个哨兵所以 leader 就自己
2989:X 05 Jun 20:16:50.304 # +elected-leader master e3Master 127.0.0.1 6379 选中 leader
2989:X 05 Jun 20:16:50.304 # +failover-state-select-slave master e3Master 127.0.0.1 6379 选中其中的一个 slave 当做 master
2989:X 05 Jun 20:16:50.357 # +selected-slave slave 127.0.0.1:6381 127.0.0.1 6381 @ e3Master 127.0.0.1 6379 选中 6381
2989:X 05 Jun 20:16:50.357 * +failover-state-send-slaveof-noone slave 127.0.0.1:6381 127.0.0.1 6381 @ e3Master 127.0.0.1 6379 发送 slaveof no one 命令
2989:X 05 Jun 20:16:50.420 * +failover-state-wait-promotion slave 127.0.0.1:6381 127.0.0.1 6381 @ e3Master 127.0.0.1 6379 等待升级 master
2989:X 05 Jun 20:16:50.515 # +promoted-slave slave 127.0.0.1:6381 127.0.0.1 6381 @ e3Master 127.0.0.1 6379 升级 6381 为 master
2989:X 05 Jun 20:16:50.515 # +failover-state-reconf-slaves master e3Master 127.0.0.1 6379
2989:X 05 Jun 20:16:50.566 * +slave-reconf-sent slave 127.0.0.1:6380 127.0.0.1 6380 @ e3Master 127.0.0.1 6379
2989:X 05 Jun 20:16:51.333 * +slave-reconf-inprog slave 127.0.0.1:6380 127.0.0.1 6380 @ e3Master 127.0.0.1 6379
2989:X 05 Jun 20:16:52.382 * +slave-reconf-done slave 127.0.0.1:6380 127.0.0.1 6380 @ e3Master 127.0.0.1 6379
2989:X 05 Jun 20:16:52.438 # +failover-end master e3Master 127.0.0.1 6379 故障恢复完成
2989:X 05 Jun 20:16:52.438 # +switch-master e3Master 127.0.0.1 6379 127.0.0.1 6381 主数据库从 6379 转变为

沧海云 Java 工程师面试指导课程——samuel

6381

2989:X 05 Jun 20:16:52.438 * +slave slave 127.0.0.1:6380 127.0.0.1 6380 @ e3Master 127.0.0.1 6381 添加 6380 为 6381 的从库

2989:X 05 Jun 20:16:52.438 * +slave slave 127.0.0.1:6379 127.0.0.1 6379 @ e3Master 127.0.0.1 6381 添加 6379 为 6381 的从库

2989:X 05 Jun 20:17:22.463 # +sdown slave 127.0.0.1:6379 127.0.0.1 6379 @ e3Master 127.0.0.1 6381 发现 6379 已经宕机，等待 6379 的恢复

```
[root@taotao2 6380]# redis-cli -p 6381
127.0.0.1:6381> INFO replication
# Replication
role:master
connected_slaves:1
slave0:ip=127.0.0.1,port=6380,state=online,offset=69815,lag=1
master_repl_offset:69952
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:2
repl_backlog_histlen:69951
127.0.0.1:6381>
```

可以看出，目前，6381 位 master，拥有一个 slave 为 6380.

接下来，我们恢复 6379 查看状态：

2989:X 05 Jun 20:35:32.172 # -sdown slave 127.0.0.1:6379 127.0.0.1 6379 @ e3Master 127.0.0.1 6381 6379 已经恢复服务

2989:X 05 Jun 20:35:42.137 * +convert-to-slave slave 127.0.0.1:6379 127.0.0.1 6379 @ e3Master 127.0.0.1 6381 将 6379 设置为 6381 的 slave

```
127.0.0.1:6381> INFO replication
# Replication
role:master
connected_slaves:2
slave0:ip=127.0.0.1,port=6380,state=online,offset=82845,lag=0
slave1:ip=127.0.0.1,port=6379,state=online,offset=82845,lag=0
master_repl_offset:82845
```

5.7. 配置多个哨兵

vim sentinel.conf

输入内容：

sentinel monitor e3Master 127.0.0.1 6381 2

sentinel monitor e3Master2 127.0.0.1 6381 1

3451:X 05 Jun 21:05:56.083 # +sdown master e3Master2 127.0.0.1 6381

3451:X 05 Jun 21:05:56.083 # +odown master e3Master2 127.0.0.1 6381 #quorum 1/1
3451:X 05 Jun 21:05:56.083 # +new-epoch 1
3451:X 05 Jun 21:05:56.083 # +try-failover master e3Master2 127.0.0.1 6381
3451:X 05 Jun 21:05:56.086 # +vote-for-leader 3f020a35c9878a12d2b44904f570dc0d4015c2ba 1
3451:X 05 Jun 21:05:56.086 # +elected-leader master e3Master2 127.0.0.1 6381
3451:X 05 Jun 21:05:56.086 # +failover-state-select-slave master e3Master2 127.0.0.1 6381
3451:X 05 Jun 21:05:56.087 # +sdown master e3Master 127.0.0.1 6381
3451:X 05 Jun 21:05:56.189 # +selected-slave slave 127.0.0.1:6380 127.0.0.1 6380 @ e3Master2 127.0.0.1 6381
3451:X 05 Jun 21:05:56.189 * +failover-state-send-slaveof-noone slave 127.0.0.1:6380 127.0.0.1 6380 @ e3Master2 127.0.0.1 6381
3451:X 05 Jun 21:05:56.252 * +failover-state-wait-promotion slave 127.0.0.1:6380 127.0.0.1 6380 @ e3Master2 127.0.0.1 6381
3451:X 05 Jun 21:05:57.145 # +promoted-slave slave 127.0.0.1:6380 127.0.0.1 6380 @ e3Master2 127.0.0.1 6381
3451:X 05 Jun 21:05:57.145 # +failover-state-reconf-slaves master e3Master2 127.0.0.1 6381
3451:X 05 Jun 21:05:57.234 * +slave-reconf-sent slave 127.0.0.1:6379 127.0.0.1 6379 @ e3Master2 127.0.0.1 6381
3451:X 05 Jun 21:05:58.149 * +slave-reconf-inprog slave 127.0.0.1:6379 127.0.0.1 6379 @ e3Master2 127.0.0.1 6381
3451:X 05 Jun 21:05:58.149 * +slave-reconf-done slave 127.0.0.1:6379 127.0.0.1 6379 @ e3Master2 127.0.0.1 6381
3451:X 05 Jun 21:05:58.203 # +failover-end master e3Master2 127.0.0.1 6381
3451:X 05 Jun 21:05:58.203 # +switch-master e3Master2 127.0.0.1 6381 127.0.0.1 6380
3451:X 05 Jun 21:05:58.203 * +slave slave 127.0.0.1:6379 127.0.0.1 6379 @ e3Master2 127.0.0.1 6380
3451:X 05 Jun 21:05:58.203 * +slave slave 127.0.0.1:6381 127.0.0.1 6381 @ e3Master2 127.0.0.1 6380

6. 集群

即使有了主从复制，每个数据库都要保存整个集群中的所有数据，容易形成木桶效应。

使用 Jedis 实现了分片集群，是由客户端控制哪些 key 数据保存到哪个数据库中，如果在水平扩容时就必须手动进行数据迁移，而且需要将整个集群停止服务，这样做非常不好的。

Redis3.0 版本的一大特性就是集群（Cluster），接下来我们一起学习集群。

6.1. hash 槽的分配

通过 cluster nodes 命令可以查看当前集群的信息：

```
192.168.56.102:6381> cluster nodes
4a9b8886ba5261e82597f5590fcdb49ea47c4c6c 192.168.56.102:6380 master - 0 1433550616849 2 connected 5461-10922
9dec39b8441c55658572b290f413200eaa76691f 192.168.56.102:6379 master - 0 1433550615337 1 connected 0-5460
b22a802b9025b7460f81cfc72f5967cfcca390a5 192.168.56.102:6381 myself,master - 0 0 3 connected 10923-16383
192.168.56.102:6381>
```

谷粒学院Java工程师面授指导课程-samuel

该信息反映出了集群中的每个节点的 id、身份、连接数、插槽数等。

当我们执行 set abc 123 命令时，redis 是如何将数据保存到集群中的呢？执行步骤：
1、 接收命令 set abc 123
2、 通过 key（abc）计算出插槽值，然后根据插槽值找到对应的节点。（abc 的插槽值为：7638）
3、 重定向到该节点执行命令[连接客户端 redise-cli.sh -c]

整个 Redis 提供了 16384 个插槽，也就是说集群中的每个节点分得的插槽数总和为 16384。
./redis-trib.rb 脚本实现了是将 16384 个插槽平均分配给了 N 个节点。

注意：如果插槽数有部分是没有指定到节点的，那么这部分插槽所对应的 key 将不能使用。

6.2. hash 槽和 key 的关系

计算 key 的插槽值：
key 的有效部分使用 CRC16 算法计算出哈希值，再将哈希值对 16384 取余，得到插槽值。

什么是有效部分？
1、 如果 key 中包含了{符号，且在{符号后存在}符号，并且{和}之间至少有一个字符，则有效部分是指{和}之间的部分；
   a) key={hello}_e3 的有效部分是 hello
2、 如果不满足上一条情况，整个 key 都是有效部分；
   a) key=hello_e3 的有效部分是全部

6.3. 故障转移

如果集群中的某一节点宕机会出现什么状况？我们这里假设 6381 宕机。

```
[root@taotao2 src]# ps -ef|grep redis
root      1456     1  0 14:31 ?        00:00:04 redis-server *:6379 [cluster]
root      1465     1  0 14:31 ?        00:00:04 redis-server *:6381 [cluster]
root      1469     1  0 14:31 ?        00:00:05 redis-server *:6382 [cluster]
root      1589  1430  0 14:58 pts/0    00:00:00 grep redis
[root@taotao2 src]#
[root@taotao2 src]# kill -9 1465
```

```
[root@taotao2 src]# ps -ef|grep redis
root      1456     1  0 14:31 ?        00:00:04 redis-server *:6379 [cluster]
root      1469     1  0 14:31 ?        00:00:05 redis-server *:6382 [cluster]
root      1592  1430  0 14:59 pts/0    00:00:00 grep redis
[root@taotao2 src]#
```

我们尝试连接下集群，并且查看集群信息，发现 6381 的节点断开连接：

samuel 工程师面试指导课程

```
[root@taotao2 src]# redis-cli
127.0.0.1:6379>
127.0.0.1:6379>
127.0.0.1:6379>
127.0.0.1:6379> cluster nodes
82ed0d63cfa6d19956dca833930977a87d6ddf74 192.168.56.102:6382 master - 0 1433574010425 4 connected 0-2041 5461-12964
b22a802b9025b7460f81cfc72f5967cfcca390a5 192.168.56.102:6381 master,fail - 1433573958671 1433573957967 3 disconnected 12965-16383
9dec39b8441c55658572b290f413200eaa76691f 192.168.56.102:6379 myself,master - 0 0 1 connected 2042-5460
127.0.0.1:6379>
```

我们尝试执行 set 命令，结果发现无法执行：

```
127.0.0.1:6379>
127.0.0.1:6379> set abc 1
(error) CLUSTERDOWN The cluster is down
127.0.0.1:6379>
```

什么情况？集群不可用了？？ 这集群也太弱了吧？？

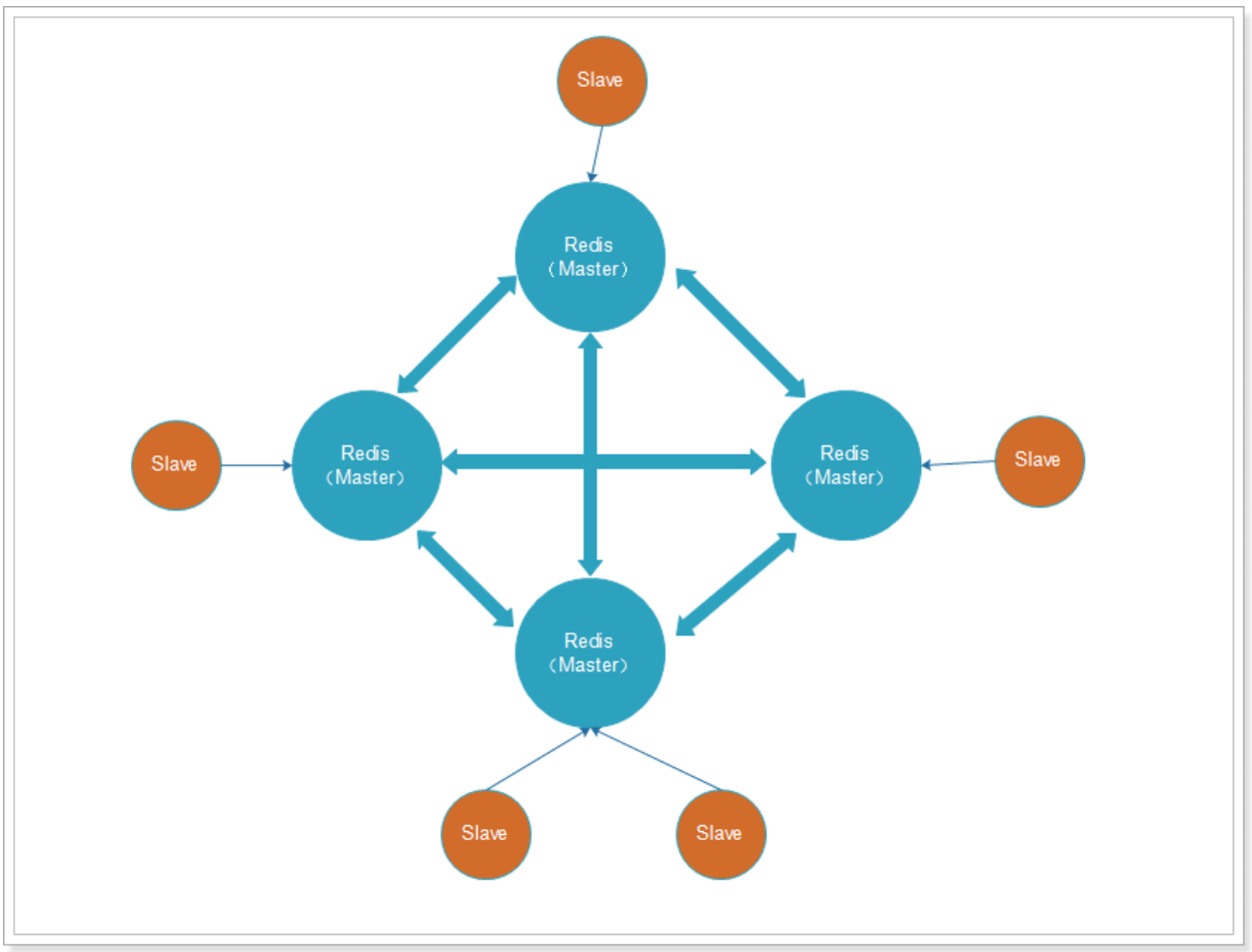
6.3.1. 故障机制

1、 集群中的每个节点都会定期的向其它节点发送 PING 命令，并且通过有没有收到回复判断目标节点是否下线；
2、 集群中每一秒就会随机选择 5 个节点，然后选择其中最久没有响应的节点放 PING 命令；
3、 如果一定时间内目标节点都没有响应，那么该节点就认为目标节点**疑似下线**；
4、 当集群中的节点超过半数认为该目标节点疑似下线，那么该节点就会被标记为**下线**；
5、 当集群中的任何一个节点下线，就会导致插槽区有空档，不完整，那么该集群将不可用；
6、 如何解决上述问题？
   a) 在 Redis 集群中可以使用主从模式实现某一个节点的高可用
   b) 当该节点（master）宕机后，集群会将该节点的从数据库（slave）转变为（master）继续完成集群服务；

6.3.2. 集群中的主从复制架构

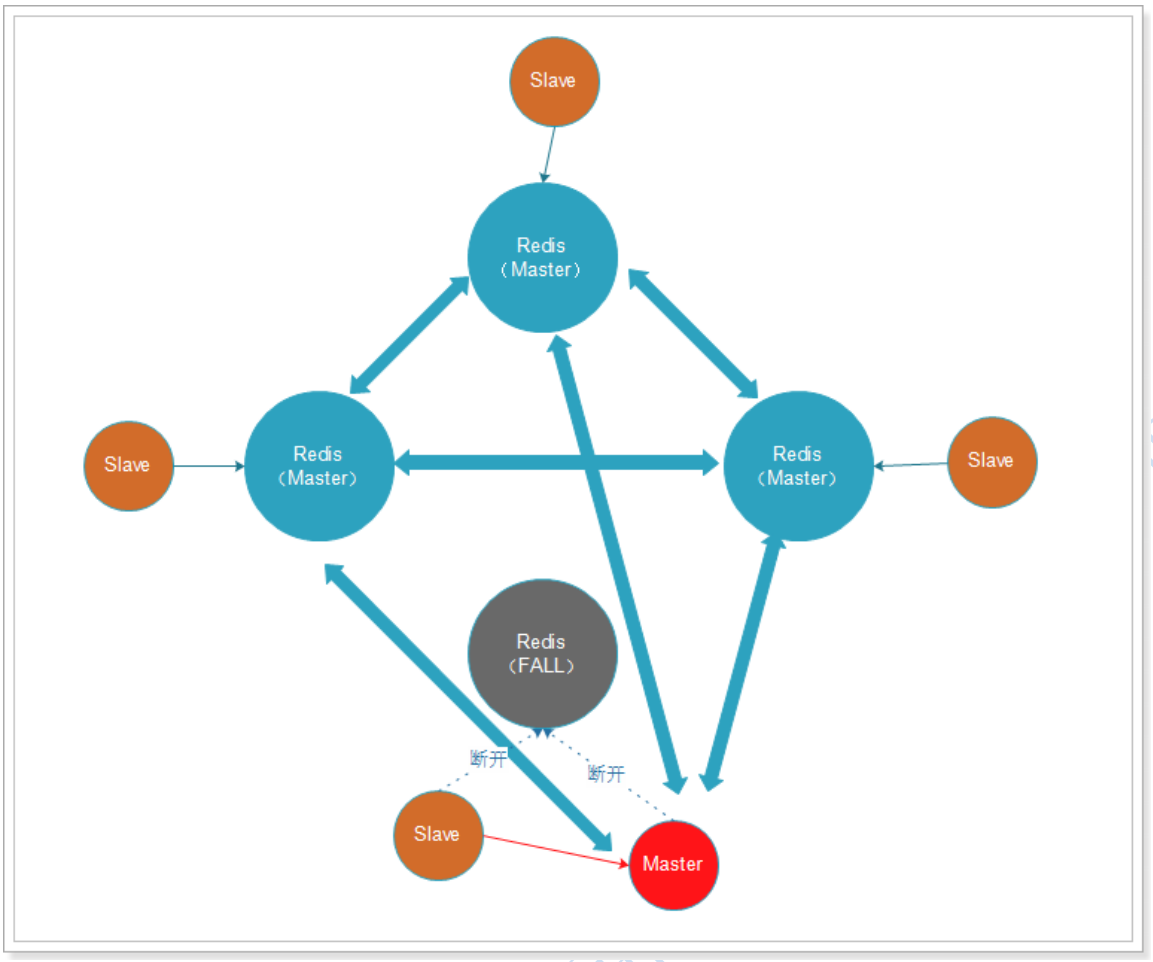
架构：

治愈系Java工程师面试指导课程——samuel



出现故障：

治愈系Java工程师面



6.3.3. 创建主从集群

需要启动 6 个 redis 实例，分别是：
6379（主） 6479（从）
6380（主） 6480（从）
6381（主） 6481（从）

```
drwxr-xr-x. 2 root root  4096 6月   6 15:50 6379
drwxr-xr-x. 2 root root  4096 6月   6 15:50 6380
drwxr-xr-x. 2 root root  4096 6月   6 15:50 6381
drwxr-xr-x. 2 root root  4096 6月   6 15:52 6479
drwxr-xr-x. 2 root root  4096 6月   6 15:47 6480
drwxr-xr-x. 2 root root  4096 6月   6 15:48 6481
```

启动 redis 实例：
cd 6379/ && redis-server ./redis.conf && cd ..
cd 6380/ && redis-server ./redis.conf && cd ..
cd 6381/ && redis-server ./redis.conf && cd ..
cd 6479/ && redis-server ./redis.conf && cd ..
cd 6480/ && redis-server ./redis.conf && cd ..

cd 6481/ && redis-server ./redis.conf && cd ..

```
[root@taotao2 6481]# ps -ef|grep redis
root      1858     1  0 15:50 ?        00:00:00 redis-server *:6379 [cluster]
root      1862     1  0 15:50 ?        00:00:00 redis-server *:6380 [cluster]
root      1864     1  0 15:50 ?        00:00:00 redis-server *:6381 [cluster]
root      1910     1  0 15:57 ?        00:00:00 redis-server *:6479 [cluster]
root      1935     1  0 15:58 ?        00:00:00 redis-server *:6480 [cluster]
root      1943     1  0 15:59 ?        00:00:00 redis-server *:6481 [cluster]
root      1947  1430  0 15:59 pts/0    00:00:00 grep redis
```

创建集群，指定了从库数量为 1，创建顺序为主库（3 个）、从库（3 个）：
./redis-trib.rb create --replicas 1 192.168.56.102:6379 192.168.56.102:6380 192.168.56.102:6381 192.168.56.102:6479 192.168.56.102:6480 192.168.56.102:6481

```
[root@taotao2 6481]# cd /usr/local/src/redis/redis-3.0.1/src/
[root@taotao2 src]# ./redis-trib.rb create --replicas 1 192.168.56.102:6379 192.168.56.102:6380 192.168.56.102:6381 192.168.56.102:6479 192.168.56.102:6480 192.168.56.102:6481
>>> Creating cluster
Connecting to node 192.168.56.102:6379: OK
Connecting to node 192.168.56.102:6380: OK
Connecting to node 192.168.56.102:6381: OK
Connecting to node 192.168.56.102:6479: OK
Connecting to node 192.168.56.102:6480: OK
Connecting to node 192.168.56.102:6481: OK
>>> Performing hash slots allocation on 6 nodes...
Using 3 masters:
192.168.56.102:6379
192.168.56.102:6380
192.168.56.102:6381
Adding replica 192.168.56.102:6479 to 192.168.56.102:6379
Adding replica 192.168.56.102:6480 to 192.168.56.102:6380
Adding replica 192.168.56.102:6481 to 192.168.56.102:6381
M: e901103802fe41256217971350d56ea9f831e222 192.168.56.102:6379
   slots:0-5460 (5461 slots) master
M: 4f4ffa2d77f27a94e830eaede0601cd207462a08 192.168.56.102:6380
   slots:5461-10922 (5462 slots) master
M: 615bd451b7ec656e5088ece832eae87faac3f97a 192.168.56.102:6381
   slots:10923-16383 (5461 slots) master
S: 6fd1dd8d6ea33bdee5513b5127b8bbe6a5cc7bf8 192.168.56.102:6479
   replicates e901103802fe41256217971350d56ea9f831e222
S: 3fc265c57ba04c45203e538adf04234548d28884 192.168.56.102:6480
   replicates 4f4ffa2d77f27a94e830eaede0601cd207462a08
S: dbfabf223345d7cbd68ba9a829fedf8f6a5aec2a 192.168.56.102:6481
   replicates 615bd451b7ec656e5088ece832eae87faac3f97a
Can I set the above configuration? (type 'yes' to accept):
```


治愈系Java工程师—samuel

```
>>> Nodes configuration updated
>>> Assign a different config epoch to each node
>>> Sending CLUSTER MEET messages to join the cluster
Waiting for the cluster to join..
>>> Performing Cluster Check (using node 192.168.56.102:6379)
M: e901103802fe41256217971350d56ea9f831e222 192.168.56.102:6379
   slots:0-5460 (5461 slots) master
M: 4f4ffa2d77f27a94e830eaede0601cd207462a08 192.168.56.102:6380
   slots:5461-10922 (5462 slots) master
M: 615bd451b7ec656e5088ece832eae87faac3f97a 192.168.56.102:6381
   slots:10923-16383 (5461 slots) master
M: 6fd1dd8d6ea33bdee5513b5127b8bbe6a5cc7bf8 192.168.56.102:6479
   slots: (0 slots) master
   replicates e901103802fe41256217971350d56ea9f831e222
M: 3fc265c57ba04c45203e538adf04234548d28884 192.168.56.102:6480
   slots: (0 slots) master
   replicates 4f4ffa2d77f27a94e830eaede0601cd207462a08
M: dbfabf223345d7cbd68ba9a829fedf8f6a5aec2a 192.168.56.102:6481
   slots: (0 slots) master
   replicates 615bd451b7ec656e5088ece832eae87faac3f97a
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
[root@taotao2 src]#
```

创建成功！查看集群信息：

```
[root@taotao2 src]# redis-cli cluster nodes
4f4ffa2d77f27a94e830eaede0601cd207462a08 192.168.56.102:6380 master - 0 1433577866269 2 connected 546
3fc265c57ba04c45203e538adf04234548d28884 192.168.56.102:6480 slave 4f4ffa2d77f27a94e830eaede0601cd207
6fd1dd8d6ea33bdee5513b5127b8bbe6a5cc7bf8 192.168.56.102:6479 slave e901103802fe41256217971350d56ea9f8
dbfabf223345d7cbd68ba9a829fedf8f6a5aec2a 192.168.56.102:6481 slave 615bd451b7ec656e5088ece832eae87faa
615bd451b7ec656e5088ece832eae87faac3f97a 192.168.56.102:6381 master - 0 1433577864254 3 connected 109
e901103802fe41256217971350d56ea9f831e222 192.168.56.102:6379 myself,master - 0 0 1 connected 0-5460
[root@taotao2 src]#
```

6.3.4. 测试

```
[root@taotao2 src]# redis-cli -c
127.0.0.1:6379> set abc 123
-> Redirected to slot [7638] located at 192.168.56.102:6380
OK
192.168.56.102:6380> get abc
"123"
192.168.56.102:6380>
```

保存、读取数据 OK！

查看下 6480 的从库数据：

```
[root@taotao2 src]# redis-cli -c -p 6480
127.0.0.1:6480> keys *
1) "abc"
127.0.0.1:6480> get abc
-> Redirected to slot [7638] located at 192.168.56.102:6380
"123"
192.168.56.102:6380>
```

看到从 6480 查看数据也是被重定向到 6380.

说明集群一切运行 OK！

6.3.5. 测试集群中 slave 节点宕机

我们将 6480 节点 kill 掉，查看情况。

```
[root@taotao2 src]# ps -ef|grep redis
root      1858     1  0 15:50 ?        00:00:00 redis-server *:6379 [cluster]
root      1862     1  0 15:50 ?        00:00:00 redis-server *:6380 [cluster]
root      1864     1  0 15:50 ?        00:00:01 redis-server *:6381 [cluster]
root      1910     1  0 15:57 ?        00:00:00 redis-server *:6479 [cluster]
root      1935     1  0 15:58 ?        00:00:00 redis-server *:6480 [cluster]
root      1943     1  0 15:59 ?        00:00:00 redis-server *:6481 [cluster]
root      2031  1430  0 16:13 pts/0    00:00:00 grep redis
[root@taotao2 src]# kill -9 1935
[root@taotao2 src]# ps -ef|grep redis
root      1858     1  0 15:50 ?        00:00:00 redis-server *:6379 [cluster]
root      1862     1  0 15:50 ?        00:00:00 redis-server *:6380 [cluster]
root      1864     1  0 15:50 ?        00:00:01 redis-server *:6381 [cluster]
root      1910     1  0 15:57 ?        00:00:00 redis-server *:6479 [cluster]
root      1943     1  0 15:59 ?        00:00:00 redis-server *:6481 [cluster]
root      2033  1430  0 16:13 pts/0    00:00:00 grep redis
[root@taotao2 src]#
```

查看集群情况：

```
[root@taotao2 src]# redis-cli -c cluster nodes
4f4ffa2d77f27a94e830eaede0601cd207462a08 192.168.56.102:6380 master - 0 1433578465276 2 connected 5461-10922
3fc265c57ba04c45203e538adf04234548d2888 192.168.56.102:6480 slave,fail 4f4ffa2d77f27a94e830eaede0601cd207462a08 1433578423043 1433578421433 5 disconnected
6fd1dd8d6ea33bdee5513b5127b8bbe6a5cc7bf8 192.168.56.102:6479 slave e901103802fe41256217971350d56ea9f831e222 0 1433578464265 4 connected
dbfabf223345d7cbd68ba9a829fedf8f6a5aec2a 192.168.56.102:6481 slave 615bd451b7ec656e5088ece832eae87faac3f97a 0 1433578465782 6 connected
615bd451b7ec656e5088ece832eae87faac3f97a 192.168.56.102:6381 master - 0 1433578463759 3 connected 10923-16383
e901103802fe41256217971350d56ea9f831e222 192.168.56.102:6379 myself,master - 0 0 1 connected 0-5460
[root@taotao2 src]#
```

发现 6480 节点不可用。

那么整个集群可用吗？

```
[root@taotao2 src]# redis-cli
127.0.0.1:6379>
[root@taotao2 src]# redis-cli -c
127.0.0.1:6379> get abc
-> Redirected to slot [7638] located at 192.168.56.102:6380
"123"
192.168.56.102:6380> set abc 456
OK
192.168.56.102:6380> get abc
"456"
192.168.56.102:6380>
```

发现集群可用，可见从数据库宕机不会影响集群正常服务。

恢复 6480 服务：

```
[root@taotao2 6480]# redis-cli cluster nodes
4f4ffa2d77f27a94e830eaede0601cd207462a08 192.168.56.102:6380 master - 0 1433578691033 2 connected 5461-10922
3fc265c57ba04c45203e538adf04234548d28884 192.168.56.102:6480 slave 4f4ffa2d77f27a94e830eaede0601cd207462a08 0 1433578690029 5 connected
6fd1dd8d6ea33bdee5513b5127b8bbe6a5cc7bf8 192.168.56.102:6479 slave e901103802fe41256217971350d56ea9f831e222 0 1433578689024 4 connected
dbfabf223345d7cbd68ba9a829fedf8f6a5aec2a 192.168.56.102:6481 slave 615bd451b7ec656e5088ece832eae87faac3f97a 0 1433578690029 6 connected
615bd451b7ec656e5088ece832eae87faac3f97a 192.168.56.102:6381 master - 0 1433578691033 3 connected 10923-16383
e901103802fe41256217971350d56ea9f831e222 192.168.56.102:6379 myself,master - 0 0 1 connected 0-5460
[root@taotao2 6480]#
```

测试 6480 中的数据：

```
[root@taotao2 6480]# redis-cli -c -p 6480
127.0.0.1:6480> keys *
1) "abc"
127.0.0.1:6480> get abc
-> Redirected to slot [7638] located at 192.168.56.102:6380
"456"
192.168.56.102:6380>
```

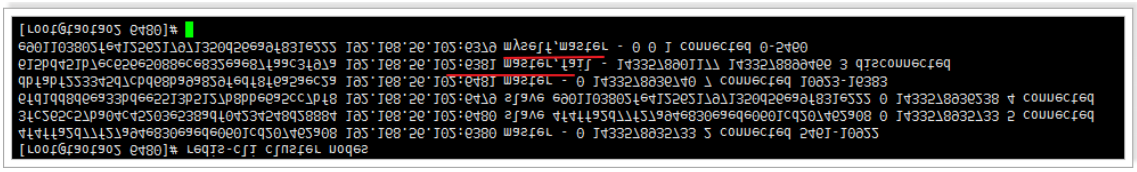
看到已经更新成最新数据。

6.3.6. 测试集群中 master 宕机

假设 6381 宕机：

```
[root@taotao2 6480]# ps -ef|grep redis
root      1858     1  0 15:50 ?        00:00:01 redis-server *:6379 [cluster]
root      1862     1  0 15:50 ?        00:00:01 redis-server *:6380 [cluster]
root      1864     1  0 15:50 ?        00:00:01 redis-server *:6381 [cluster]
root      1910     1  0 15:57 ?        00:00:01 redis-server *:6479 [cluster]
root      1943     1  0 15:59 ?        00:00:01 redis-server *:6481 [cluster]
root      2061     1  0 16:17 ?        00:00:00 redis-server *:6480 [cluster]
root      2077  1430  0 16:21 pts/0    00:00:00 grep redis
[root@taotao2 6480]# kill -9 1864
```

查看集群情况：

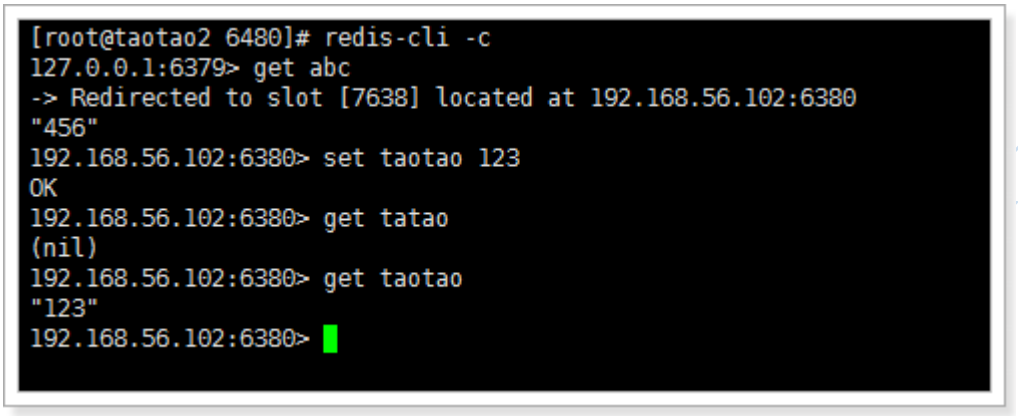


发现：

1、6381 节点失效不可用

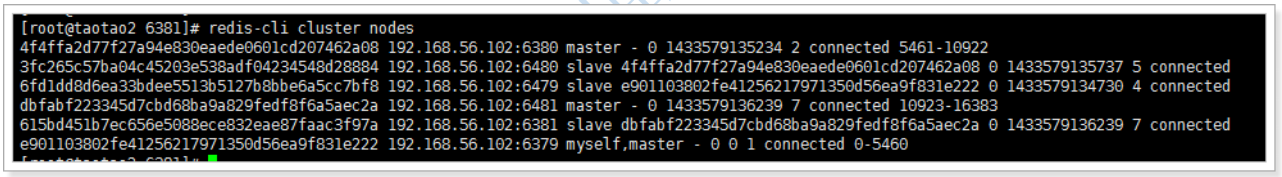
2、6481 节点从 slave 转换为 master

测试集群是否可用：



集群可用。

恢复 6381：



发现：

1、6381 节点可用

2、6481 依然是主节点

3、6381 成为 6481 的从数据库

6.4. 使用集群需要注意的事项

1、多键的命令操作（如 MGET、MSET），如果每个键都位于同一个节点，则可以正常支持，否则会提示错误。

2、集群中的节点只能使用 0 号数据库，如果执行 SELECT 切换数据库会提示错误。

7. Redis 分布式锁【了解】

基于 Redis 实现分布式锁的方式名叫 *Redlock*，此种方式比原先的单节点的方法更安全。它可以保证以下特性：

1. 安全特性：互斥访问，即永远只有一个 client 能拿到锁
2. 避免死锁：最终 client 都可能拿到锁，不会出现死锁的情况，即使原本锁住某资源的 client crash 了或者出现了网络分区
3. 容错性：只要大部分 Redis 节点存活就可以正常提供服务

7.1. 怎么在单节点上实现分布式锁

SET resource*name myr*andom_value NX PX 30000

主要依靠上述命令，该命令仅当 Key 不存在时（NX 保证）set 值，并且设置过期时间 3000ms （PX 保证），值 my*random*value 必须是所有 client 和所有锁请求发生期间唯一的，释放锁的逻辑是：

```
lua if redis.call("get",KEYS[1]) == ARGV[1] then return redis.call("del",KEYS[1]) else return 0 end
```

上述实现可以避免释放另一个 client 创建的锁，如果只有 del 命令的话，那么如果 client1 拿到 lock1 之后因为某些操作阻塞了很长时间，此时 Redis 端 lock1 已经过期了并且已经被重新分配给了 client2，那么 client1 此时再去释放这把锁就会造成 client2 原本获取到的锁被 client1 无故释放了，但现在为每个 client 分配一个 unique 的 string 值可以避免这个问题。至于如何去生成这个 unique string，方法很多随意选择一种就行了。

7.2. Redlock 算法

算法很易懂，起 5 个 master 节点，分布在不同的机房尽量保证可用性。为了获得锁，client 会进行如下操作：

1. 得到当前的时间，微秒单位
2. 尝试顺序地在 5 个实例上申请锁，当然需要使用相同的 key 和 random value，这里一个 client 需要合理设置与 master 节点沟通的 timeout 大小，避免长时间和一个 fail 了的节点浪费时间
3. 当 client 在大于等于 3 个 master 上成功申请到锁的时候，且它会计算申请锁消耗了多少时间，这部分消耗的时间采用获得锁的当下时间减去第一步获得的时间戳得到，如果锁的持续时长（lock validity time）比流逝的时间多的话，那么锁就真正获取到了。
4. 如果锁申请到了，那么锁真正的 lock validity time 应该是 origin（lock validity time） - 申请锁期间流逝的时间
5. 如果 client 申请锁失败了，那么它就会在少部分申请成功锁的 master 节点上执行释放锁的操作，重置状态

咨询尽在Java工程师面试指导课程——samuel

7.3. 失败重试

如果一个 client 申请锁失败了，那么它需要稍等一会在重试避免多个 client 同时申请锁的情况，最好的情况是一个 client 需要几乎同时向 5 个 master 发起锁申请。另外就是如果 client 申请锁失败了它需要尽快在它曾经申请到锁的 master 上执行 unlock 操作，便于其他 client 获得这把锁，避免这些锁过期造成的时间浪费，当然如果这时候网络分区使得 client 无法联系上这些 master，那么这种浪费就是不得不付出的代价了。

7.4. 放锁

放锁操作很简单，就是依次释放所有节点上的锁就行了

7.5. 性能、崩溃恢复和 fsync

如果我们的节点没有持久化机制，client 从 5 个 master 中的 3 个处获得了锁，然后其中一个重启了，这是注意 **整个环境中又出现了 3 个 master 可供另一个 client 申请同一把锁！** 违反了互斥性。如果我们开启了 AOF 持久化那么情况会稍微好转一些，因为 Redis 的过期机制是语义层面实现的，所以在 server 挂了的时候时间依旧在流逝，重启之后锁状态不会受到污染。但是考虑断电之后呢，AOF 部分命令没来得及刷回磁盘直接丢失了，除非我们配置刷回策略为 fsnyc = always，但这会损伤性能。解决这个问题的方法是，当一个节点重启之后，我们规定在 max TTL 期间它是不可用的，这样它就不会干扰原本已经申请到的锁，等到它 crash 前的那部分锁都过期了，环境不存在历史锁了，那么再把这个节点加进来正常工作。

治愈系Java工程师面试指导课程——samuel