

第一章 分布式缓存课程介绍

笔记请见资料里面的PPT

第二章 分布式锁实现方案分析

第1集 redis分布式锁的使用场景

简介：分布式锁是BATJ最常见的Redis面试题，对分布式锁的掌握，多场景下不同版本分布式锁的掌握显得尤为关键

- 分布式锁是什么
 - 分布式锁是控制分布式系统或不同系统之间共同访问共享资源的一种锁实现
 - 如果不同的系统或同一个系统的不同主机之间共享了某个资源时，往往通过互斥来防止彼此干扰。
- 分布式锁设计目的

可以保证在分布式部署的应用集群中，同一个方法在同一操作只能被一台机器上的一个线程执行。

- 设计要求
 - 这把锁要是一把可重入锁（避免死锁）
 - 这把锁有高可用的获取锁和释放锁功能
 - 这把锁获取锁和释放锁的性能要好...
- 分布式锁实现方案分析
 - 获取锁的时候，使用 setnx(SETNX key val:当且仅当 key 不存在时，set 一个 key 为 val 的字符串，返回 1;
 - 若 key 存在，则什么都不做，返回 【0】加锁，锁的 value 值为当前占有锁服务器内网IP编号拼接任务标识
 - 在释放锁的时候进行判断。并使用 expire 命令为锁添加一个超时时间，超过该时间则自动释放锁。
 - 返回1则成功获取锁。还设置一个获取的超时时间，若超过这个时间则放弃获取锁。
setex (key,value,expire) 过期以秒为单位
 - 释放锁的时候，判断是不是该锁（即Value为当前服务器内网IP编号拼接任务标识），若是该锁，则执行 delete 进行锁释放

第2集 手把手进行Redis分布式锁的实现

简介：springboot定时任务讲解，手把手带你进行Redis分布式锁源码分析，高性能的分布式锁现在代码实现

- springboot定时任务配置，定时任务配置步骤
 - 1、开启springboot注解@EnableScheduling
 - 2、执行方法加注解 @Scheduled(cron="0/10 * * * * *") 规律：秒分时日月年
 - 2、本地服务器集群部署同一套代码？
 - 面临问题分析
 - jar包远程copy，scp 源 账户号@目标机器:路径，如 scp
 - 可单机部署多个节点也可部署在不同虚拟机上面，如果是单机部署多节点要保持端口不一致，如设置8080、8081、8082等
 - 查看集群里面所有机器是否都启动成功
- ```
netstat -anp |grep 8080或者ps -ef|grep 8080
nohup java -jar jar包名称 &
```

## 第3集 本地服务器集群部署同一套代码

---

**简介：单机多节点部署讲解，图解后端性能演变**

- 服务器log日志打印，springboot已经logback
    - 1、引入logback-spring.xml
    - 2、配置路径,在application.properties加入路径引用
- ```
logging.config=classpath:logback-spring.xml
logging.path=/data/java/weblog/xdclassredis
```
- 修改启动端口
 - nohup持久化启动方式
 - nohup的意思是忽略SIGHUP信号，所以当运行nohup a.jar的时候，关闭shell, 那么a.jar进程还是存在的（对SIGHUP信号免疫）

```
nohup java -jar jar名 &
```

- nohup xdclass-mobile-redis-0.0.1-SNAPSHOT.jar &启动项目
- 可单机部署多个节点也可部署在不同虚拟机上面，如果是单机部署多节点要保持端口不一致，如设置8080、8081、8082等
- 查看集群里面所有机器是否都启动成功 lsof -i:8080 ,如果没有安装lsof的话先执行

```
yum install lsof
```

第4集 互联网大厂面试题之深入剖析TCP三次握手

简介：互联网动向分析，TCP为什么要握三次手

- 市场动向分析
 - 市场背景：

对于这几年的互联网市场，越来越多的市场需求导致人才输出渠道更加丰富，五花八门的培训机构培训，学校对互联网人才的培养同样会显得越来越重视
 - 问题分析：

那么，越来越多的人的出现必将给我们学员带来更大的挑战，怎么增大在互联网市场的竞争力
 - 解决方案：从宏观的角度来讲我们首先必须增加自我个体的价值和特色
 - 总结：

在学习框架知识的同时，我们应该注重高级知识的学习，让高级篇幅成为自己的特色，从而在压力巨大的市场中脱颖而出。作为网络知识的一大环节，TCP知识的学习将会给你的知识体系带来特色。显然，TCP知识以及成为大厂选拔人才中常常会涉猎到的加分项。

- 查看本机TCP连接状态

- 查看当前机器的连接数

```
netstat -n | awk '/^tcp/ {++S[$NF]} END {for(a in S) print a, S[a}]'
```

- 图解分析TCP三次握手协议
- 为什么要三次握手，不能像http或者UDP一样直接传输

主要是为了防止已失效的连接请求报文段突然又传到了B,因而报文错乱问题 假定A发出的第一个连接请求报文段并没有丢失，而是在某些网络结点长时间滞留了，一直延迟到连接释放 以后的某个时间才到达B，本来这是一个早已失效的报文段。但B收到此失效的连接请求报文段后，就误认为 是A又发出一次新的连接请求，于是就向A发出确认报文段，同意建立连接。假定不采用三次握手，那么只要B发出确认，新的连接就建立了，这样一直等待A发来数据，B的许多资源就这样白白浪费了。

第5集 互联网大厂面试题之深入剖析TCP四次挥手

大厂面试TCP网络调度原理和TCP四次挥手设计

- 面试题：你知道TCP四次挥手是什么吗？为什么要进行四次挥手
 - 确保数据能够完整传输
 - - 当被动方收到主动方的FIN报文通知时，它仅仅表示主动方没有数据再发送给被动方了。
 - 但未必被动方所有的数据都完整的发送给了主动方，所以被动方不会马上关闭SOCKET,它可能还需要发送一些数据给主动方后，再发送FIN报文给主动方，告诉主动方同意关闭连接
 - 所以这里的ACK报文和FIN报文多数情况下都是分开发送的。
 - 模拟流程

A:“喂，我不说了（FIN）。”A->FIN_WAIT1

B:“我知道了(ACK)。等下，上一句还没说完。Ba la ba la... (传输数据)”B->CLOSE_WAIT | A->FIN_WAIT2

B:”好了，说完了，我也不说了（FIN）。”B->LAST_ACK

A:”我知道了（ACK）。”A->TIME_WAIT | B->CLOSED

A等待2MSL,保证B收到了消息,否则重说一次”我知道了”,A->CLOSED

- 图解分析TCP四次挥手协议
 - TCP前面10种状态切换
- TCP第11种状态CLOSING 状态概念
 - 这种状态在实际情况中应该很少见，属于一种比较罕见的例外状态。正常情况下，当一方发送FIN报文后，按理来说是应该先收到（或同时收到）对方的ACK报文，再收到对方的FIN报文。但是CLOSING状态表示一方发送FIN报文后，并没有收到对方的ACK报文，反而却也收到了对方的FIN报文。什么情况下会出现此种情况呢？那就是当双方几乎在同时close()一个SOCKET的话，就出现了双方同时发送FIN报文的情况，这是就会出现CLOSING 状态，表示双方都正在关闭SOCKET连接。
- netstat -anp|grep 8080



小D课堂 愿景：“让编程不在难学，让技术与生活更加有趣” 更多教程请访问

第三章 分布式锁面试如此多娇，引无数英雄尽折腰

第1集 高级篇幅之Redis分布式锁实现源码讲解

简介：剖析分布式锁实现分析，手把手进行实战步骤讲解

- Redis分布锁步骤分解图讲解
- linux查看历史命令 history|grep scp
- 手把手进行实战步骤讲解
 - 分布锁满足两个条件，一个是加有效时间的锁，一个是高性能解锁
 - 采用redis命令setnx (set if not exist)、setex (set expire value) 实现
 - 【千万记住】解锁流程不能遗漏，否则导致任务执行一次就永不过期
 - 将加锁代码和任务逻辑放在try，catch代码块，将解锁流程放在finally
 - 通过scp方式将项目打包jar包放到集群里面的服务器
 - 通过nohup启动jar包

scp 本机文件路径 daniel@172.16.244.145:/usr/local/jar_file

nohup java -jar jar包名称 &

第2集 高级篇幅之Redis分布式锁可能出现的问题

简介：剖析分布式锁setnx、setex的缺陷，在setnx和setex中间发生了服务down机，实战模拟演练服务宕机情况

- Redis分布式锁思考，图解分布式锁setnx、setex的缺陷
 - 从Redis宕机讲解分布式锁执行的异常场景流程
 - 从Server服务宕机讲解分布式锁执行的异常场景流程
- 实战演练Server服务宕机情况
 - 先kill调集群里面的其他节点的java进程，
 - 在执行job的时候将进程kill掉
 - 通过ps -ef|grep java看到进程的pid
 - 重启服务，看服务是否每次都获取锁失败
- 一步步分析解决问题方案
 - 怎么一次性执行过一条命令而不会出现问题，采用Lua脚本
 - Redis从2.6之后支持setnx、setex连用

第3集 Lua脚本讲解之Redis分布式锁

简介：手把手进行Lua脚本讲解和setnx、setex命令连用讲解

- Lua简介
 - 从 Redis 2.6.0 版本开始，通过内置的 Lua 解释器，可以使用 EVAL 命令对 Lua 脚本进行求值。
 - Redis 使用单个 Lua 解释器去运行所有脚本，并且，Redis 也保证脚本会以原子性(atomic)的方式执行：当某个脚本正在运行的时候，不会有其他脚本或 Redis 命令被执行。这和使用 MULTI / EXEC 包围的事务很类似。在其他别的客户端看来，脚本的效果(effect)要么是可见的(visible)，要么就是已完成的(already completed)。
- Lua脚本配置流程
 - 1、在resource目录下面新增一个后缀名为.lua结尾的文件
 - 2、编写lua脚本
 - 3、传入lua脚本的key和arg
 - 4、调用redisTemplate.execute方法执行脚本
- Lua脚本结合RedisTemplate实战演练
- Lua脚本其他工作场景剖析和演练
- lua eval <http://doc.redisfans.com/script/eval.html>

第4集 实战操作RedisConnection实现分布式锁

简介：RedisConnection实现分布锁的方式，采用redisTemplate操作redisConnection 实现setnx和setex两个命令连用

- redisTemplate本身有没通过valueOperation实现分布式锁
 - 问题探索：Spring Data Redis提供了与Java客户端包的集成服务，比如Jedis, JRedis等 通过getNativeConnection的方式可以解决问题吗？
- Spring Data Redis提供了与Java客户端包的集成服务，比如Jedis, JRedis等
 - 代码演示
 - /**

```
    * 重写redisTemplate的set方法
    * <p>
    * 命令 SET resource-name anystring NX EX max-lock-time 是一种在 Redis 中实现锁
    的简单方法。
    * <p>
    * 客户端执行以上的命令：
    * <p>
    * 如果服务器返回 OK ，那么这个客户端获得锁。
    * 如果服务器返回 NIL ，那么客户端获取锁失败，可以在稍后再重试。
    *
    * @param key    锁的key
    * @param value  锁里面的值
    * @param seconds 过去时间 ( 秒 )
    * @return
    */
    private String set(final String key, final String value, final long seconds)
    {
        Assert.isTrue(!StringUtils.isEmpty(key), "key不能为空");
        return redisTemplate.execute(new RedisCallback<String>() {
            @Override
            public String doInRedis(RedisConnection connection) throws
            DataAccessException {
                Object nativeConnection = connection.getNativeConnection();
                String result = null;
                if (nativeConnection instanceof JedisCommands) {
                    result = ((JedisCommands) nativeConnection).set(key, value,
                    NX, EX, seconds);
                }

                if (!StringUtils.isEmpty(lockKeyLog) &&
                !StringUtils.isEmpty(result)) {
                    logger.info("获取锁{}的时间：{}", lockKeyLog,
                    System.currentTimeMillis());
                }

                return result;
            }
        });
    }
}
```

- 为什么新版本的spring-data-redis会报class not can not be case错误

```
io.lettuce.core.RedisAsyncCommandsImpl cannot be cast to
redis.clients.jedis.JedisCommands
```

- 探索spring-data-redis升级

- 官网api分析 <https://docs.spring.io/spring-data/redis/docs/1.5.0.RELEASE/api/> <https://docs.spring.io/spring-data/redis/docs/2.0.13.RELEASE/api/>
- 源码改造

```
public Boolean doInRedis(RedisConnection connection) throws DataAccessException {  
    RedisConnection redisConnection =  
        redisTemplate.getConnectionFactory().getConnection();  
    return redisConnection.set(key.getBytes(), getHostIp().getBytes(),  
        Expiration.seconds(expire), RedisStringCommands.SetOption.ifAbsent());  
}
```

第5集 实战操作采用lua脚本做高可用分布式锁的优化

简介：高可用分布式锁的优化点分析

- 解锁的流程分析

当某个锁需要持有的时间小于锁超时时间时会出现两个进程同时执行任务的情况，这时候如果进程没限制只有占有这把锁的人才能解锁的原则就会出现，A解了B的锁。

- 采用lua脚本做解锁流程优化讲解

第6集 经验分享面试技巧分析之支付宝分布式锁

简介：分析支付宝2018分布式锁面试题

- 问题1：1、什么是分布式锁？

- 首先，为了确保分布式锁可用，我们至少要确保锁的实现同时满足以下三个条件：

- 互斥性。在任意时刻，只有一个客户端能持有锁。
- 不会发生死锁。即使有一个客户端在持有锁的期间崩溃而没有主动解锁，也能保证后续其他客户端能加锁。
- 解铃还须系铃人。加锁和解锁必须是同一个客户端，客户端自己不能把别人加的锁给解了

- ◦ 从 Redis 2.6.0 版本开始，通过内置的 Lua 解释器，可以使用 EVAL 命令对 Lua 脚本进行求值。
- Redis 使用单个 Lua 解释器去运行所有脚本，并且，Redis 也保证脚本会以原子性(atomic)的方式执行：当某个脚本正在运行的时候，不会有其他脚本或 Redis 命令被执行。这和使用 MULTI / EXEC 包围的事务很类似。在其他别的客户端看来，脚本的效果(effect)要么是不可见的(not visible)，要么就是已完成的(already completed)。

- 问题2：怎么实现分布式锁

- 实现分布式锁的方案大概有两种

- 采用lua脚本操作分布式锁
- 采用setnx、setex命令连用的方式实现分布式锁

- 问题3：

- 解锁需要注意什么

解铃还须系铃人。加锁和解锁必须是同一个客户端，客户端自己不能把别人加的锁给解了



小点课堂 愿景：“让编程不在难学，让技术与生活更加有趣” 更多教程请访问

第四章 分布式锁面试如此多娇，引无数英雄尽折腰

第1集 不可不知的Redis高性能读写分离

简介：玩转Redis读写分离

- 海量并发性能瓶颈处理
- 对读写能力进行扩展，采用读写分离方式解决性能瓶颈
运行一些额外的服务器，让它们与主服务器进行连接，然后将主服务器发送的数据副本并通过网络进行准实时的更新（具体的更新速度取决于网络带宽）通过将读请求分散到不同的服务器上面进行处理，用户可以从新添加的从服务器上获得额外的读查询处理能力
- redis已经发现了这个读写分离场景特别普遍，自身集成了读写分离供用户使用。我们只需在redis的配置文件里面加上一条，【slaveof host port】语句
- 配置过程，启动多个redis节点，修改节点里面的redis.conf配置文件
- 可能遇到的问题？
 - 服务器下线导致数据丢失，slave下线之后怎么保证数据的同步？

第2集 你知道Redis读写分离是怎么做数据同步的吗？

简介：一起探索redis读写分离数据同步的问题，通过图解方式解析数据同步的概念和意义

- 进行复制中的主从服务器双方的数据库将保存相同的数据，概念上将这种现象称作“数据库状态一致”
RDB 全量持久化 AOF append only if 增量持久化
- redis2.8版本之前使用旧版复制功能SYNC
 - SYNC是一个非常耗费资源的操作
 - 主服务器需要执行BGSAVE命令来生成RDB文件，这个生成操作会耗费主服务器大量的CPU、内存和磁盘读写资源
 - 主服务器将RDB文件发送给从服务器，这个发送操作会耗费主从服务器大量的网络带宽和流量，并对主服务器响应命令
 - 请求的时间产生影响：接收到RDB文件的从服务器在载入文件的过程是阻塞的，无法处理命令请求
- 2.8之后使用PSYNC

- PSYNC命令具有完整重同步（full resynchronization）和部分重同步（partial resynchronization1）两种模式
 - 部分重同步功能由以下三个部分构成：
 - 主服务的复制偏移量（replication offset）和从服务器的复制偏移量
 - 主服务器的复制积压缓冲区（replication backlog），默认大小为1M
 - 服务器的运行ID(run ID),用于存储服务器标识，如从服务器断线重新连接，取到主服务器的运行ID与重接后的主服务器运行ID进行对比，从而判断是执行部分重同步还是执行完整重同步

第3集 面试分享高频大厂面试题之分布式系统高可用

简介：作为曾经的阿里金牌面试官，daniel老师有话要说；redis高可用的探索，剖析高可用解决方案

- 高可用的概念？
 - 高可用HA（High Availability）是分布式系统架构设计中必须考虑的因素之一，它通常是指，通过设计减少系统不能提供服务的时间。
- 通过三大要点解释高可用:

 - 单点是系统高可用的大敌，应该尽量在系统设计的过程中避免单点
 - 保证系统高可用，架构设计的核心准则是：冗余。
 - 每次出现故障需要人工介入恢复势必会增加系统的不可服务时间,实现自动故障转移
 - 重启服务，看服务是否每次都获取锁失败

- 分布式高可用经典架构环节分析
 - 【客户端层】到【反向代理层】的高可用，是通过反向代理层的冗余来实现的。以nginx为例：有两台nginx，一台对线上提供服务，另一台冗余以保证高可用，常见的实践是keepalived存活探测
 - 【反向代理层】到【web应用】的高可用，是通过站点层的冗余来实现的。假设反向代理层是nginx，nginx.conf里能够配置多个web后端，并且nginx能够探测到多个后端的存活性。
 - 自动故障转移：当web-server挂了的时候，nginx能够探测到，会自动的进行故障转移，将流量自动迁移到其他的web-server，整个过程由nginx自动完成，对调用方是透明的。
 - 【服务层】到【缓存层】的高可用，是通过缓存数据的冗余来实现的。redis天然支持主从同步，redis官方也有sentinel哨兵机制，来做redis的存活性检测。
 - 【服务层】到【数据库层】的高可用，数据库层用“主从同步，读写分离”架构，所以数据库层的高可用，又分为“读库高可用”与“写库高可用”两类。
 - 读库采用冗余的方式实现高可用，写库采用keepalived存活探测 binlog进行同步



小D课堂 愿景：“让编程不在难学，让技术与生活更加有趣”

第五章 Redis高可用架构Sentinel

第1集 灾备切换Sentinel的使用

简介：互联网服务灾备故障转移，sentinel的配置

- Redis主从复制可将主节点数据同步给从节点，从节点此时有两个作用：

- 一旦主节点宕机，从节点作为主节点的备份可以随时顶上来。
- 扩展主节点的读能力，分担主节点读压力。

但是问题来了：

- 一旦主节点宕机，从节点晋升成主节点，同时需要修改应用方的主节点地址，还需要命令所有从节点去复制新的主节点，整个过程需要人工干预。

- redis主节点挂掉之后应该怎么操作？命令模拟

```
slaveof no one          # 取消主备，变更为主节点
```

```
slaveof 新host 新节点  # 将其他节点设置为新主节点的备份节点
```

- Sentinel正是实现了这个功能
- 开启Sentinel配置 3主3从 3主6从

```
sentinel monitor mymaster 127.0.0.1 6379 1
sentinel down-after-milliseconds mymaster 10000
sentinel failover-timeout mymaster 60000
sentinel parallel-syncs mymaster 1
```

- 命令讲解

- sentinel monitor mymaster 127.0.0.1 6379 1 名称为mymaster的主节点名，1表示将这个主服务器判断为失效至少需要 1个 Sentinel 同意（只要同意 Sentinel 的数量不达标，自动故障迁移就不会执行）
- down-after-milliseconds 选项指定了 Sentinel 认为服务器已经断线所需的毫秒数
- failover-timeout 过期时间，当failover开始后，在此时间内仍然没有触发任何failover操作，当前sentinel 将会认为此次failover失败
- parallel-syncs 选项指定了在执行故障转移时，最多可以有多少个从服务器同时对新的主服务器进行同步，这个数字越小，完成故障转移所需的时间就越长。
- 如果从服务器被设置为允许使用过期数据集，那么你可能不希望所有从服务器都在同一时间向新的主服务器发送同步请求，因为尽管复制过程的绝大部分步骤都不会阻塞从服务器，但从服务器在载入主服务器发来的 RDB 文件时，仍然会造成从服务器在一段时间内不能处理命令请求：如果全部从服务器一起对新的主服务器进行同步，那么就可能会导致所有从服务器在短时间内全部不可用的情况出现。

- 启动所有主从上的sentinel

- 前提是它们各自的server已成功启动 `cd /usr/local/redis/src/redis-sentinel /etc/redis/sentinel.conf`

- info Replication 查看节点信息
- shutdown主节点看服务是否正常

第2集 互联网高可用灾备以及Sentinel三大任务讲解

简介：互联网冷备和热备讲解，Sentinel是怎么工作的？Sentinel三大工作任务是什么？

- Sentinel三大工作任务

- 监控（Monitoring）：Sentinel 会不断地检查你的主服务器和从服务器是否运作正常。

- 提醒（Notification）：当被监控的某个 Redis 服务器出现问题时，Sentinel 可以通过 API 向管理员或者其他应用程序发送通知。
- 自动故障迁移（Automatic failover）：当一个主服务器不能正常工作时，Sentinel 会开始一次自动故障迁移操作，它会将失效主服务器的其中一个从服务器升级为主服务器，并让失效主服务器的其他从服务器改为复制新的主服务器；当客户端试图连接失效的主服务器时，集群也会向客户端返回新主服务器的地址，使得集群可以使用新主服务器代替失效服务器。
- 互联网冷备和热备讲解，冷备和热备的特点分析
 - 冷备
 - 概念：冷备份发生在数据库已经正常关闭的情况下，当正常关闭时会提供给我们一个完整的数据库
 - 优点：
 - 是非常快速的备份方法（只需拷文件）
 - 低度维护，高度安全
 - 缺点：
 - 单独使用时，只能提供到“某一时间点上”的恢复
 - 再实施备份的全过程中，数据库必须要作备份而不能作其他工作。也就是说，在冷备份过程中，数据库必须是关闭状态
 - 热备
 - 概念：热备份是在数据库运行的情况下，采用archive log mode方式备份数据库的方法
 - 优点：
 - 备份的时间短
 - 备份时数据库仍可使用
 - 可达到秒级恢复
 - 缺点
 - 若热备份不成功，所得结果不可用于时间点的恢复
 - 困难于维护，所以要非凡仔细小心

第3集 Redis高可用Sentinel故障转移原理

简介：Sentinel是怎么工作的？

- 主观下线：
 - 概念主观下线（Subjectively Down，简称 SDOWN）指的是单个 Sentinel 实例对服务器做出的下线判断
 - 主管下线特点：
 - 如果一个服务器没有在 master-down-after-milliseconds 选项所指定的时间内，对向它发送 PING 命令的 Sentinel 返回一个有效回复（valid reply），那么 Sentinel 就会将这个服务器标记为主观下线
 - 服务器对 PING 命令的有效回复可以是以下三种回复的其中一种：
 - 返回 +PONG。
 - 返回 -LOADING 错误。
 - 返回 -MASTERDOWN 错误。
- 客观下线
 - 客观下线概念：
 - 指的是多个 Sentinel 实例在对同一个服务器做出 SDOWN 判断，并且通过 SENTINEL is-master-down-by-addr 命令互相交流之后，得出的服务器下线判断。（一个 Sentinel 可以通过

向另一个 Sentinel 发送 SENTINEL is-master-down-by-addr 命令来询问对方是否认为给定的服务器已下线。)

- 客观下线特点：
 - 从主观下线状态切换到客观下线状态并没有使用严格的法定人数算法 (strong quorum algorithm)，而是使用了流言协议：如果 Sentinel 在给定的时间范围内，从其他 Sentinel 那里接收到了足够数量的主服务器下线报告，那么 Sentinel 就会将主服务器的状态从主观下线改变为客观下线。如果之后其他 Sentinel 不再报告主服务器已下线，那么客观下线状态就会被移除。
- 客观下线注意点：
 - 客观下线条件只适用于主服务器：对于任何其他类型的 Redis 实例，Sentinel 在将它们判断为下线前不需要进行协商，所以从服务器或者其他 Sentinel 永远不会达到客观下线条件。只要一个 Sentinel 发现某个主服务器进入了客观下线状态，这个 Sentinel 就可能会被其他 Sentinel 推出，并对失效的主服务器执行自动故障迁移操作。

第4集 sentinel整合Springboot实战

简介：Redis高可用整合springboot讲解

- 思考？redis高可用sentinel整合springboot的步骤

图解高可用架构代理的概念

- 引入yaml配置文件

```
redis:
  sentinel:
    master: redis_master_group1      #mymaster
    nodes: 172.16.244.133:26379
```

- pom文件

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

- 启动springboot服务
- 登上服务器看配置是否有效



第六章 Redis内置高可用集群

第1集 Redis集群搭建这回事

简介：Redis集群搭建实战，赠送Redis图文搭建教程

- 安装redis

- 处理步骤

```
cd /usr/local/  
wget http://download.redis.io/releases/redis-4.0.6.tar.gz  
tar -zxvf redis-4.0.6.tar.gz  
cd redis-4.0.6  
make && make install
```

- 新建集群文件夹

- 处理步骤

```
cd /usr/local/  
mkdir redis_cluster  
cd redis_cluster  
mkdir 7000 7001 7002 7003 7004 7005  
cp /usr/local/redis-4.0.6/redis.conf /usr/local/redis_cluster/7000
```

- 修改redis_cluster/7000到redis_cluster/7005文件夹下面的Redis.conf

- 处理步骤

```
daemonize    yes                //redis后台运行  
port 7000                //端口7000,7002,7003  
cluster-enabled yes        //开启集群 把注释#去掉  
cluster-config-file nodes.conf //集群的配置 配置文件首次启动自动生成 7000,7001,7002  
cluster-node-timeout 5000    //请求超时 设置5秒够了  
appendonly yes            //aof日志开启 有需要就开启，它会每次写操作都记录一条日志  
bind 127.0.0.1 172.16.244.144(此处为自己内网的ip地址，centos7下面采用ip addr来查看，其他系统试一下  
ifconfig查看，ip为)
```

- 在其他节点也修改完Redis.conf

- 处理步骤

```
cp /usr/local/redis_cluster/7000/redis.conf /usr/local/redis_cluster/7001  
cp /usr/local/redis_cluster/7000/redis.conf /usr/local/redis_cluster/7002  
cp /usr/local/redis_cluster/7000/redis.conf /usr/local/redis_cluster/7003  
cp /usr/local/redis_cluster/7000/redis.conf /usr/local/redis_cluster/7004  
cp /usr/local/redis_cluster/7000/redis.conf /usr/local/redis_cluster/7005
```

- 启动所有redis节点cd redis-server所在的路径

- 处理步骤

```
cp /usr/local/redis-4.0.6/src/redis-server /usr/local/ redis-cluster
```

```
cd /usr/local/redis_cluster/7000 ../redis-server ./redis.conf
```

```
cd /usr/local/redis-cluster/7001 ../redis-server ./redis.conf
```

```
cd /usr/local/redis-cluster/7002 ../redis-server ./redis.conf
```

```
cd /usr/local/redis-cluster/7003 ../redis-server ./redis.conf
```

```
cd /usr/local/redis-cluster/7004 ../redis-server ./redis.conf
```

```
cd /usr/local/redis-cluster/7005 ../redis-server ./redis.conf
```

- 创建集群

- 前面已经准备好了搭建集群的redis节点，接下来我们要把这些节点都串连起来搭建集群。官方提供了一个工具：redis-trib.rb(/usr/local/redis-4.0.6/src/redis-trib.rb) 看后缀就知道这鸟东西不能直接执行，它是用ruby写的一个程序，所以我们还得安装ruby.

```
yum -y install ruby ruby-devel rubygems rpm-build
```

```
gem install redis
```

- 如果gem install redis发现报错

```
curl -L get.rvm.io | bash -s stable
```

```
source /usr/local/rvm/scripts/rvm
```

```
rvm list known
```

```
rvm install 2.3.3
```

```
rvm use 2.3.3
```

```
ruby --version
```

```
gem install redis
```

- 开启集群工作

```
cd /usr/local/redis-4.0.6/src
```

```
./redis-trib.rb create --replicas 1 127.0.0.1:7000 127.0.0.1:7001 \  
127.0.0.1:7002 127.0.0.1:7003 127.0.0.1:7004 127.0.0.1:7005
```

- 测试集群是否正常

```
./redis-cli -c -p 7000
```

- 如果搭建失败，请用此命令将所有启动的redis server一个个关闭掉

```
./redis-cli -p 7000 shutdown
```

第2集 Redis集群不得不说的这点事

简介：Redis集群特点介绍

- Redis 集群的数据分片
 - 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,
 - 举个例子,比如当前集群有3个节点,那么:
 - 节点 A 约包含 0 到 5500号哈希槽.
 - 节点 B 约包含5501 到 11000 号哈希槽.
 - 节点 C 约包含11001 到 16384号哈希槽.
 - 查看集群信息`redis-cli -p 7000 cluster nodes | grep master`
 - 这种结构很容易添加或者删除节点. 比如如果我想新添加个节点D, 我需从节点 A, B, C中得部分槽到D上. 如果我想移除节点A,需将A中的槽移到B和C节点上,然后将没有任何槽的A节点从集群中移除即可. 由于从一个节点将哈希槽移动到另一个节点并不会停止服务,所以无论添加删除或者改变某个节点的哈希槽的数量都不会造成集群不可用的状态.
 - 从Redis宕机讲解分布式锁执行的异常场景流程
 - 从Server服务宕机讲解分布式锁执行的异常场景流程
- Redis 集群的主从复制模型
 - 为了使在部分节点失败或者大部分节点无法通信的情况下集群仍然可用,所以集群使用了主从复制模型.每个节点都会有N-1个复制品. 在我们例子中具有A, B, C三个节点的集群,在没有复制模型的情况下,如果节点B失败了,那么整个集群就会以为缺少5501-11000这个范围的槽而不可用.Redis集群做主从备份解决了这个问题
- Redis 一致性保证
 - 主节点对命令的复制工作发生在返回命令回复之后, 因为如果每次处理命令请求都需要等待复制操作完成的话, 那么主节点处理命令请求的速度将极大地降低 —— 我们必须在性能和一致性之间做出权衡. 注意: Redis 集群可能会在将来提供同步写的方法. Redis 集群另外一种可能会丢失命令的情况是集群出现了网络分区, 并且一个客户端与至少包括一个主节点在内的少数实例被孤立.

- 手把手测试故障转移

```
redis-cli -p 7000 debug segfault
redis-cli -p 7001 cluster nodes | grep master
```

第3集 Redis集群分片重哈希

简介：玩转Redis集群节点分片重哈希

- 采用SSH连接远程服务器
 - ssh命令安装过程：<https://blog.csdn.net/DanielAntony/article/details/87997574>
- 集群重新分片
 - 手动处理slot节点槽重新分片
 - `./redis-trib.rb reshard 127.0.0.1:7000`

- 你想移动多少个槽(从1到16384)? all
- 添加一个新的主节点


```
./redis-trib.rb add-node 127.0.0.1:7006 127.0.0.1:7000
```
- 添加一个新的从节点


```
./redis-trib.rb add-node --slave 127.0.0.1:7006 127.0.0.1:7000
```
- 移除一个节点


```
./redis-trib.rb del-node 127.0.0.1:7000 <node-id>
```

 第一个参数是任意一个节点的地址,第二个节点是你想要移除的节点地址。
 - 移除主节点【先确保节点里面没有slot】
 - 使用同样的方法移除主节点,不过在移除主节点前,需要确保这个主节点是空的. 如果不是空的,需要将这个节点的数据重新分片到其他主节点上.
 - 替代移除主节点的方法是手动执行故障恢复,被移除的主节点会作为一个从节点存在,不过这种情况下不会减少集群节点的数量,也需要重新分片数据.
 - 移除从节点 直接移除成功

第4集 Redis集群整合Springboot实战

简介：Redis集群整合springboot讲解

- 介绍
 - 大多数用户可能会使用RedisTemplate它及其相应的包org.springframework.data.redis.core- 由于其丰富的功能集,该模板实际上是Redis模块的中心类。该模板为Redis交互提供了高级抽象。虽然RedisConnection提供了接受和返回二进制值(byte数组)的低级方法,但模板负责序列化和连接管理,使用户无需处理这些细节。
- 引入spring-data-redis pom依赖


```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```
- 引入redistemplate
 - 引入bean redisTemplate的使用,类型于: monogoTemplate、jdbcTemplate数据库连接工具
 - 编写redisTemplate类,设置redisConnectFactory
- 配置yml配置文件

第七章 Redis高可用集群扩展

第1集 Redis集群知多少

简介：一起来进行Redis集群探索

- 图解探索【是不是只学习master cluster就行了】
- 分析cluster集群方式原理
 - 水平切分于垂直切分相比，相对来说稍微复杂一些。因为要将同一个表中的不同数据拆分到不同的数据库中。分片是一种基于数据库分成若干片段的传统概念扩容技术，它将数据库分割成多个碎片并将这些碎片放置在不同的服务器上。
 - 垂直切分的最大特点就是规则简单，实施也更为方便，尤其适合各业务之间的耦合度非常低，相互影响很小，业务逻辑非常清晰的系统。按照业务维度将不同数据放入不同的表
- 故障转移
 - 通过gossip节点信息同步实现，实现slave作为master的备份节点，并实现故障剔除master节点采用slave替换，并在替换完之后将结果通知到其他节点。
- Twitter推特公司twemproxy服务端分片和客户端分片集群解决方案

第2集 一致性Hash算法

简介：一起来进行Redis集群探索

2的32次方进行hash取模 0到2的32次方-1

- jedis分布式之 ShardedJedisPool（一致性Hash分片算法）
- 概念：

分布式系统中负载均衡的问题时候可以使用Hash算法让固定的一部分请求落到同一台服务器上，这样每台服务器固定处理一部分请求（并维护这些请求的信息），起到负载均衡的作用
- 做法：
 - hash环上顺时针从整数0开始，一直到最大正整数，我们根据四个ip计算的hash值肯定会落到这个hash环上的某一个点，至此我们把服务器的四个ip映射到了一致性hash环
 - 当用户在客户端进行请求时候，首先根据hash(用户id)计算路由规则（hash值），然后看hash值落到了hash环的那个地方，根据hash值在hash环上的位置顺时针找距离最近的ip作为路由ip

- 当用户在客户端进行请求时候，首先根据hash(用户id)计算路由规则（hash值），然后看hash值落到了hash环的那个地方，根据hash值在hash环上的位置顺时针找距离最近的ip作为路由ip.
- 一致性hash的特性
 - 单调性(Monotonicity)，单调性是指如果已经有一些请求通过哈希分派到了相应的服务器进行处理，又有新的服务器加入到系统中时候，应保证原有的请求可以被映射到原有的或者新的服务器中去，而不会被映射到原来的其它服务器上去。
 - 分散性(Spread)：分布式环境中，客户端请求时候可能不知道所有服务器的存在，可能只知道其中一部分服务器，在客户端看来他看到的部分服务器会形成一个完整的hash环。如果多个客户端都把部分服务器作为一个完整hash环，那么可能会导致，同一个用户的请求被路由到不同的服务器进行处理。这种情况显然是应该避免的，因为它不能保证同一个用户的请求落到同一个服务器。所谓分散性是指上述情况发生的严重程度。好的哈希算法应尽量避免尽量降低分散性。一致性hash具有很低的分散性
 - 平衡性(Balance)：平衡性也就是说负载均衡，是指客户端hash后的请求应该能够分散到不同的服务器上去。一致性hash可以做到每个服务器都进行处理请求，但是不能保证每个服务器处理的请求的数量大致相同
- 虚拟节点

第3集 一致性Hash算法虚拟节点

简介：通过虚拟节点倾斜解决方案，均匀一致性hash

- 出现问题分析：

部门hash节点下架之后，虽然剩余机器都在处理请求，但是明显每个机器的负载不均衡，这样称为一致性hash的倾斜，虚拟节点的出现就是为了解决这个问题。
- 增设虚拟节点

当物理机器数目为A，虚拟节点为B的时候，实际hash环上节点个数为A*B，将A节点分部为A1,A2,A3;将A1、A2、A3平均分布在各个位置，使A服务的节点尽量均匀分配在各个角落
- 每台服务器负载相对均衡

当某个节点挂了之后，其数据均衡的分布给相邻的顺时针后面的一个节点上面，故所有数据比之前所述一致性hash相对均衡

第4集 twemproxy实现hash分片

简介：经典服务端分片twemproxy特性讲解

codis

- 概念：

Twemproxy，也叫nutcracker。是一个twitter开源的一个redis和memcache快速/轻量级代理服务器; Twemproxy是一个快速的单线程代理程序，支持Memcached 和Redis。Redis代理中间件twemproxy是一种利用中间件做分片的技术。twemproxy处于客户端和服务器的中间，将客户端发来的请求，进行一定的处理后（sharding），再转发给后端真正的redis服务器
- 作用：

Twemproxy通过引入一个代理层，可以将其后端的多台Redis或Memcached实例进行统一管理与分配，使应用程序只需要在Twemproxy上进行操作，而不用关心后面具体有多少个真实的Redis或Memcached存储
- 特性：

- 支持失败节点自动删除
 - 可以设置重新连接该节点的时间
 - 可以设置连接多少次之后删除该节点
- 减少了客户端直接与服务器连接的连接数量
 - 自动分片到后端多个redis实例上
- 多种哈希算法
 - md5 , crc16 , crc32 , crc32a , fnv1_64 , fnv1a_64 , fnv1_32 , fnv1a_32 , hsieh , murmur , jenkins
- 多种分片算法
 - ketama (一致性hash算法的一种实现) , modula , random



小D课堂 愿景："让编程不在难学，让技术与生活更加有趣"

第八章 大厂面试题之Redis缓存持久化策略

第1集 Redis RDB持久化原理

简介：rdb持久化方案配置讲解，redis的开发者是怎么实现rdb的

- rdb持久化配置

```
# 时间策略，表示900s内如果有1条是写入命令，就触发产生一次快照，可以理解为就进行一次备份
save 900 1
save 300 10 # 表示300s内有10条写入，就产生快照
save 60 10000
# redis servercron 类似于linux的crontab，默认每隔100毫秒执行一次

# 文件名称
dbfilename dump.rdb

# 如果持久化出错，主进程是否停止写入
stop-writes-on-bgsave-error yes

# 是否压缩
rdbcompression yes

# 导入时是否检查
rdbchecksum yes

# 文件保存路径
dir /usr/local/redis-4.0.6
```

- save的含义

实际生产环境每个时段的读写请求肯定不是均衡的，为此redis提供一种根据key单位时间操作次数来触发一次备份到磁盘，我们可以自由定制什么情况下触发备份，此功能起到平衡性能与数据安全的作用

- 在Redis中RDB持久化的触发分为两种：自己手动触发与Redis定时触发

- 针对RDB方式的持久化，手动触发可以使用：

- save：会阻塞当前Redis服务器，直到持久化完成，线上应该禁止使用。
- bgsave：该触发方式会fork一个子进程，由子进程负责持久化过程，因此阻塞只会发生在fork子进程的时候

- 而自动触发的场景主要是有以下几点
 - 根据我们的 `save m n` 配置规则自动触发
 - 从节点全量复制时，主节点发送rdb文件给从节点完成复制操作，主节点会触发 `bgsave`
 - 执行 `debug reload` 时
 - 执行 `shutdown` 时，如果没有开启aof，也会触发
- 禁用RDB
 - 只需要在save的最后一行写上：`save ""`

第2集 Redis AOF持久化原理

简介：rdb持久化方案配置讲解，redis的开发者是怎么实现rdb的

- AOF持久化配置

```
# 是否开启aof
appendonly yes

# 文件名称
appendfilename "appendonly.aof"

# 同步方式
appendfsync everysec

# aof重写期间是否同步
no-appendfsync-on-rewrite no

# 重写触发配置
auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb

# 加载aof时如果有错如何处理
aof-load-truncated yes # yes表示如果aof尾部文件出问题，写log记录并继续执行。no表示提示写入等待修复后写入

# 文件重写策略
aof-rewrite-incremental-fsync yes
```

- `appendfsync` 同步模式有三种模式，一般情况下都采用 **everysec** 配置，在数据和安全里面做平衡性选择，最多损失1s的数据
 - `always`：把每个写命令都立即同步到aof，很慢，但是很安全
 - `everysec`：每秒同步一次，是折中方案
 - `no`：redis不处理交给OS来处理，非常快，但是也最不安全
- AOF的整个流程大体来看可以分为两步，一步是命令的实时写入（如果是 `appendfsync everysec` 配置，会有1s损耗），第二步是对aof文件的重写。
 - 步骤：
 - 命令写入=》追加到aof_buf=》通过时间事件调用flushAppendOnlyFile函数同步到aof磁盘
 - 原因：
 - 实时写入磁盘会带来非常高的磁盘IO，影响整体性能

- AOF持久化的效率和安全性分析

always：每个时间事件循环都将AOF_BUF缓冲区的所有内容写入到AOF文件，并且同步AOF文件，这是最安全的方式，但磁盘操作和阻塞延迟，是IO开支较大。

everysec：每秒同步一次，性能和安全都比较中庸的方式，也是redis推荐的方式。如果遇到物理服务器故障，有可能导致最近一秒内aof记录丢失(可能为部分丢失)。

no：redis并不直接调用文件同步，而是交给操作系统来处理，操作系统可以根据buffer填充情况/通道空闲时间等择机触发同步；这是一种普通的文件操作方式。性能较好，在物理服务器故障时，数据丢失量会因OS配置有关。处于no模式下的flushAppendOnlyFile调用无须执行同步操作

第3集 Redis两种持久化方案对比

简介：rdb和aof两种持久化方案的优缺点分析，以及如何选择

- Redis提供了不同的持久性选项：
 - RDB持久性以指定的时间间隔执行数据集的时间点快照。
 - AOF持久性记录服务器接收的每个写入操作，将在服务器启动时再次播放，重建原始数据集。使用与Redis协议本身相同的格式以追加方式记录命令。当Redis太大时，Redis能够重写日志背景。
- RDB的优缺点
 - 优点：
 - RDB最大限度地提高了Redis的性能，父进程不需要参与磁盘I/O
 - 与AOF相比，RDB允许使用大数据集更快地重启
 - 缺点：
 - 如果您需要在Redis停止工作时（例如断电后）将数据丢失的可能性降至最低，则RDB并不好
 - RDB经常需要fork（）才能使用子进程持久存储在磁盘上。如果数据集很大，Fork（）可能会非常耗时
- AOF的优缺点
 - 优点：
 - 数据更加安全
 - 当Redis AOF文件太大时，Redis能够在后台自动重写AOF `## INCRE 1 执行10万 = INCREBY10万执行一次`
 - AOF以易于理解和解析的格式一个接一个地包含所有操作的日志 `# flushdb类似于rm -rf /*`
 - 缺点：
 - AOF文件通常比同一数据集的等效RDB文件大
 - 根据确切的fsync策略，AOF可能比RDB慢
- RDB 和 AOF ,我应该用哪一个？一般来说,如果想达到足以媲美 PostgreSQL 的数据安全性，你应该同时使用两种持久化功能。如果你非常关心你的数据,但仍然可以承受数分钟以内的数据丢失，那么你可以只使用RDB持久化。有很多用户都只使用 AOF 持久化，但我们并不推荐这种方式：因为定时生成 RDB 快照（snapshot）非常便于进行数据库备份，并且 RDB 恢复数据集的速度也要比 AOF 恢复的速度要快
- 在线上我们到底该怎么做？
 - RDB持久化与AOF持久化同步使用
 - 如果Redis中的数据并不是特别敏感或者可以通过其它方式重写生成数据，可以关闭持久化，如果丢失数据可以通过其它途径补回；
 - 自己制定策略定期检查Redis的情况，然后可以手动触发备份、重写数据；

- 采用集群和主从同步

第4集 Redis过期key清除策略

简介：redis过期key清除策略分析

- Redis如何淘汰过期的keys：set name daniel 3600
 - 惰性删除：
 - 概念：当一些客户端尝试访问它时，key会被发现并主动的过期
 - 放任键过期不管，但是每次从键空间中获取键时，都检查取得的键是否过期，如果过期的话，就删除该键
 - 特点：**CPU友好**，但如果一个key不再使用，那么它会一直存在于内存中，造成浪费
 - 定时删除：
 - 概念：设置键的过期时间的同时，创建一个定时器（timer），让定时器在键的过期时间来临时，立即执行对键的删除操作
 - 定期删除：
 - 隔一段时间，程序就对数据库进行一次检查，删除里面的过期键，至于要删除多少过期键，以及要检查多少个数据库，则由算法决定。即设置一个定时任务，比如10分钟删除一次过期的key；间隔小则占用CPU,间隔大则浪费内存
 - 例如Redis每秒处理：
 1. 测试随机的20个keys进行相关过期检测。
 2. 删除所有已经过期的keys。
 3. 如果有多于25%的keys过期，重复步骤1。

Redis服务器实际使用的是惰性删除和定期删除两种策略：通过配合使用这两种删除策略，服务器可以很好地在合理使用CPU时间和避免浪费内存空间之间取得平衡。

惰性删除策略是怎么实现？通过`expireIfNeeded`函数，当我们操作key的时候进行判断key是否过期

定期删除策略是怎么实现的？通过`activeExpireCycle`函数，`serverCron`函数执行时，`activeExpireCycle`函数就会被调用，规定的时间里面分多次遍历服务器的`expires`字典随机检查一部分key的过期时间，并删除其中的过期key




小D课堂 愿景："让编程不在难学，让技术与生活更加有趣"

第九章 你知道微信红包实现原理吗？

第1集 微信红包原理业务探讨

简介：微信红包业务探讨分析

- 技术不只有面试，redis到底有什么实战应用
 - 面试只是表达自己的知识，而开发实战才是show time
- 小D课堂带你进入软件开发流程剖析，图解分析流程

- 如果上司给一个任务，让我们在实现微信抢红包这个功能，我们该怎么做？
 - 业务思考，实现方式千百种，不追求方法复制，只追求推导过程的思考总结
 - 功能点探索
 - 新建红包：在DB、cache各新增一条记录
 - 抢红包：请求访问cache，剩余红包个数大于0则可拆开红包
 - key：1，value：20 string decr原子减，每次减1，而decreby减指定数量2
 - 拆红包：20个红包里面有500块，key：1,value：50000(以分为单位) decreby 548，decreby 1055，decreby 2329
 - 请求访问cache，剩余红包个数大于0则继续，同时获取可抢红包数与金额
 - 计算金额（从1分到剩余平均值2倍之间随机数，如果不是最后一个红包，剩余金额预留最少1分给cas更新失败，最后一位拿红包的人）
 - cas更新数据库（更新红包计数表记录【剩余红包个数、剩余红包金额】、插入领取记录）
 - 查看红包记录：用户进来直接查DB即可

第2集 微信红包数据库表设计

简介：微信红包业务探讨分析

- 红包实战数据库表设计
 - 数据库表设计
 - 业务梳理
- 红包流水表

```
CREATE TABLE `red_packet_info` (  
  `id` int(11) NOT NULL AUTO_INCREMENT,  
  `red_packet_id` bigint(11) NOT NULL DEFAULT 0 COMMENT '红包id,采用timestamp+5位随机数',  
  `total_amount` int(11) NOT NULL DEFAULT 0 COMMENT '红包总金额,单位分',  
  `total_packet` int(11) NOT NULL DEFAULT 0 COMMENT '红包总个数',  
  `remaining_amount` int(11) NOT NULL DEFAULT 0 COMMENT '剩余红包金额,单位分',  
  `remaining_packet` int(11) NOT NULL DEFAULT 0 COMMENT '剩余红包个数',  
  `uid` int(20) NOT NULL DEFAULT 0 COMMENT '新建红包用户的用户标识',  
  `create_time` timestamp COMMENT '创建时间',  
  `update_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP  
  COMMENT '更新时间',  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8mb4 COMMENT='红包信息表,新建一个红包插入一条记录';
```

- 红包记录表

```
CREATE TABLE `red_packet_record` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `amount` int(11) NOT NULL DEFAULT '0' COMMENT '抢到红包的金额',
  `nick_name` varchar(32) NOT NULL DEFAULT '0' COMMENT '抢到红包的用户的用户名',
  `img_url` varchar(255) NOT NULL DEFAULT '0' COMMENT '抢到红包的用户的头像',
  `uid` int(20) NOT NULL DEFAULT '0' COMMENT '抢到红包用户的用户标识',
  `red_packet_id` bigint(11) NOT NULL DEFAULT '0' COMMENT '红包id,采用timestamp+5位随机数',
  `create_time` timestamp COMMENT '创建时间',
  `update_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP COMMENT '更新时间',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8mb4 COMMENT='抢红包记录表,抢一个红包插入一条记录';
```

- 开发过程中记录表信息，养成上线前梳理好上线流程的好习惯
- 将库表导入数据库
- 通过mybatis generator生成代码

第3集 发红包接口实现

简介：微信红包业务探讨分析

- 发红包功能接口开发
 - 新增一条红包记录
 - 往mysql里面添加一条红包记录
 - 往redis里面添加一条红包数量记录 decr
 - 往redis里面添加一条红包金额记录 decreby
- 抢红包功能接口开发
 - 抢红包功能属于原子减操作
 - 当大小小于0时原子减失败
- 当红包个数为0时后面进来的用户全部抢红包失败，并不会进入拆红包环节
- 抢红包功能扩展设计
 - 将红包ID的请求放入请求队列中，如果发现超过红包的个数，直接返回
 - 类推出token令牌和秒杀设计原理
- 注意点
 - 抢到红包不到能拆成功
 - 2014年的红包一点开就知道金额，分两次操作，先抢到金额，然后再转账。2015年后的红包的拆和抢是分离的，需要点两次，因此会出现抢到红包了，但点开告知红包已经被领完的状况。进入到第一个页面不代表抢到，只表示当时红包还有。

第4集 抢红包接口实现

简介：微信红包业务探讨分析

- 抢红包功能接口开发
 - 在抢红包这里并不能保证用户已经能领到这个红包
 - 抢红包只是做了一个判断，判断当前是否还有红包
 - 有红包则返回可以领
 - 没红包则返回不可以领
- 拆红包功能接口开发
 - 拆红包才是用户能领导红包
 - 这时候要先减redis里面的金额和红包数量
 - 减完金额再入库

第5集 技术角度分析以后我们应该怎么抢红包？

简介：微信红包业务探讨之拆红包原理，以及以后我们应该怎么抢红包

- 微信红包设计算法分析
 - 玩法：微信金额是拆的时候实时算出来，不是预先分配的，采用的是纯内存计算，不需要预算空间存储
 - 分配：
 - 发100块钱，总共10个红包，那么平均值是10块钱一个，那么发出来的红包的额度在0.01元~20元之间波动
 - 当前面4个红包总共被领了30块钱时，剩下70块钱，总共6个红包，那么这7个红包的额度在： $0.01 \sim (70 \div 6 \times 2) = 23.33$ 之间波动
 - 这样算下去，可能会超过最开始的全部金额，因此到了最后面如果不够这么算，那么会采取如下算法：保证剩余用户能拿到最低1分钱即可
 - 存储：数据库会累加已经领取的个数与金额，插入一条领取记录。入账则是后台异步操作
 - 转账：通过财付通往红包所有者账户转账，过程通过是异步操作

第6集 抢红包项目总结

简介：微信红包业务探讨之拆红包原理，以及以后我们应该怎么抢红包

- take all操作
 - 入库转账时需要保证红包个数和红包剩余金额正确
- 高并发处理：红包如何计算被抢完？
 - cache会抵抗无效请求，将无效的请求过滤掉，实际进入到后台的量不大。cache记录红包个数，原子操作进行个数递减，到0表示被抢光
- 性能扩展
 - 多主sharding，水平扩展机器
 - 数据库层面sharding分片
 - redis层面sharding分片技术
- 业务能动性，从发展的角度来看待业务
- 观察总结，技术赋能业务



第十章 20w年薪必须掌握的Redis实战

第1集 寻衅滋事？先过了我这道关

简介：布隆过滤器是什么，一定要用吗？

- 黑客流量攻击：故意访问不存在的数据，导致程序不断访问DB数据库的数据
- 黑客安全拦截：当黑客访问不存在的缓存时迅速返回避免缓存及DB挂掉
- 思考：如果让你实现这个功能你会怎么做？key：10000 10001 10002 10003 大集合，key是否在集合里面
- 温故而知新：分析java常用数据结构复习 set map key,value list 有序get[0]、get[1]；
- list.contains(key)遍历数据，进行equals()比较，性能小
- set.contains(key) hashCode比较，性能较高，64位 1G
- map.get(key) hashCode比较，性能还行
- 概念：
 - **布隆过滤器**（英语：Bloom Filter）是1970年由布隆提出的。它实际上是一个很长的**二进制**向量和一系列随机**映射函数**。布隆过滤器可以用于检索一个元素是否在一个集合中。它的优点是空间效率和查询时间都远远超过一般的算法，缺点是有一定的误识别率和删除困难。
- 优点：
 - 相比于其它的数据结构，布隆过滤器在空间和时间方面都有巨大的优势。布隆过滤器存储空间和插入/查询时间都是常数（ $O(k)$ ）。另外，散列函数相互之间没有关系，方便由硬件并行实现。布隆过滤器不需要存储元素本身，在某些对保密要求非常严格的场合有优势
- 缺点
 - 但是布隆过滤器的缺点和优点一样明显。误算率是其中之一。随着存入的元素数量增加，误算率随之增加。但是如果元素数量太少，则使用散列表足矣

第2集 了解布隆过滤器

简介：布隆过滤器是什么，一定要用吗？

- 布隆过滤器的其他使用场景
- 网页爬虫对URL的去重，避免爬取相同的URL地址；
反垃圾邮件，从数十亿个垃圾邮件列表中判断某邮箱是否垃圾邮箱（同理，垃圾短信）；
缓存击穿，将已存在的缓存放到布隆中，当黑客访问不存在的缓存时迅速返回避免缓存及DB挂掉。
- 布隆过滤器实现原理图解

第3集 谷歌布隆过滤器实现会员转盘抽奖

简介：抽奖程序功能需求分析，谷歌实现布隆过滤器，谷歌布隆过滤器的局限性

- 需求分析步骤
 - 互联网功能需求分析
 - 这是一个抽奖程序，只针对会员用户有效
 - 抽离出功能所有api
 - 制定存储方案
 - 性能优化方案分析
- 成型互联网产品用户量上千万，日常百万，怎么做到高性能非会员过滤
- 这是一个布隆过滤器的经典使用场景
- 通过google布隆过滤器存储会员数据实战
 - 程序启动时将数据放入内存中
 - google自动创建布隆过滤器
 - 用户ID进来之后判断是否是会员
- 代码演练
- ```
CREATE TABLE `sys_user` (
 `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
 `user_name` varchar(11) CHARACTER SET utf8mb4 DEFAULT NULL COMMENT '用户名',
 `image` varchar(11) CHARACTER SET utf8mb4 DEFAULT NULL COMMENT '用户头像',
 PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=11 DEFAULT CHARSET=utf8;
```

## 第4集 goole布隆过滤器与Redis布隆过滤器

简介：布隆过滤器两种实现方案的优缺点分析

- google布隆过滤器的缺陷与思考
  - 基于内存布隆过滤器有什么特点
  - 内存级别产物
  - 重启即失效
  - 本地内存无法用在分布式场景
  - 不支持大数据量存储
- 需求分析步骤
  - 互联网功能需求分析
    - 这是一个抽奖程序，只针对会员用户有效
  - 抽离出功能所有api
  - 制定存储方案
  - 性能优化方案分析
- Redis布隆过滤器
  - 可扩展性Bloom过滤器
    - 一旦Bloom过滤器达到容量，就会在其上创建一个新的过滤器
  - 不存在重启即失效或者定时任务维护的成本
    - 基于goole实现的布隆过滤器需要启动之后初始化布隆过滤器
  - 缺点：
    - 需要网络IO,性能比基于内存的过滤器低
- 选择:  
优先基于数据量进行考虑

## 第5集 Redis布隆过滤器安装

简介：

- Redis布隆过滤器安装过程 自己构建一个bitMap
  - git在centos7下面的安装
    - 1、安装git，直接使用yum安装即可：  
  
`yum -y install git`
    - 2、创建git用户，git用户可以正常通过ssh使用git，但无法登录shell，因为我们为git用户指定的git-shell每次一登录就自动退出。  
  
`useradd -m -d /home/git -s /usr/bin/git-shell git`

### 3、初始化git仓库

```
mkdir -p /data/git
cd /data/git
git init --bare project1.git
chown git.git project1.git -R
```

### 4、创建免密钥

```
cd /home/git
mkdir .ssh
chmod 700 .ssh
touch .ssh/authorized_keys
chmod 600 .ssh/authorized_keys
chown git.git .ssh -R
```

#### ◦ 您应该首先下载并编译模块：

```
$ git clone git://github.com/RedisLabsModules/rebloom
$ cd rebloom
$ make
```

#### ◦ 将Rebloom加载到Redis中，在redis.conf里面添加

```
loadmodule /path/to/rebloom.so
```

#### ◦ 命令实战

```
BF.ADD bloom redis
BF.EXISTS bloom redis
BF.EXISTS bloom nonxist
```

## 第6集 Redis布隆过滤器与springboot的整合探索

### 简介：基于lua脚本实现springboot和布隆过滤器的整合

- 通过普通命令无法实现springboot整合布隆过滤器
- 查找github开源框架的流程
- 分析开源框架的实现原理
- 通过lua脚本自己实现布隆过滤器

- 编写两个lua脚本
  - 添加数据到指定名称的布隆过滤器
  - 从指定名称的布隆过滤器获取key是否存在的脚本

## 第7集 秒杀系统需求分析

简介：采用大厂需求分析步骤对秒杀需求功能分析

- 功能核心点
  - 经典互联网商品抢购秒杀功能
- 功能api
  - 商品秒杀接口
- 数据落地存储方案
  - 通过分布式redis减库存
  - DB存最终订单信息数据
- api性能调优
  - 性能瓶颈在高并发秒杀
  - 技术难题在于超卖问题

## 第8集 秒杀系统功能步骤梳理

简介：后端秒杀功能步骤梳理

- 利用 Redis 缓存incr拦截流量
  - 首先通过数据控制模块，提前将秒杀商品缓存到读写分离 Redis，并设置秒杀开始标记如下：  

```
"skuId_start": 0 //开始标记0表示秒杀开始
"skuId_count": 10000 //总数
"skuId_access": 12000 //接受抢购数
```
  - 秒杀开始前，服务集群读取 goodsId\_Start 为 0，直接返回未开始。
  - 服务时间不一致可能导致流量倾斜
  - 数据控制模块将 goodsId\_start 改为1，标志秒杀开始。
  - 当接受下单数达到 sku\_count\*1.2 后，继续拦截所有请求，商品剩余数量为 0
- 利用Redis缓存加速库存扣量  

```
"skuId_booked": 10000 //总数0开始10000 通过incr扣减库存，返回抢购成功
```
- 将用户订单数据写入mq
- 监听mq入库

## 第9集 秒杀系统功能api实战(上)

简介：后端秒杀网关流量拦截层功能开发

- 先判断秒杀是否已经开始
  - 初始化时将key SECKILL\_START\_1 value 0\_1554046102存入数据库中
- 利用 Redis 缓存incr拦截流量
  - 缓存拦截流量代码编写
  - 用incr方法原子加
  - 通过原子加判断当前skuld\_access是否达到最大值
  - 思考：是否需要保证获取到值的时候和incr值两个命令的原子性
    - 保证原子性的方式，采用lua脚本
    - 采用lua脚本方式保证原子性带来缺点，性能有所下降
    - 不保证原子性缺点，放入请求量可能大于skuld\_access

## 第10集 秒杀系统功能api实战(中)

简介：后端秒杀信息校验层功能开发布隆过滤器实现重复购买拦截

- 订单信息校验层
  - 校验当前用户是否已经买过这个商品
    - 需要存储用户的uid
    - 存数据库效率太低
    - 存Redis value方式数据太大
    - 存布隆过滤器性能高且数据量小
- 校验完通过直接返回抢购成功

## 第11集 秒杀系统功能api实战(下)

简介：后端秒杀信息校验层功能开发lua脚本实现库存扣除

- 库存扣除成功，获取当前最新库存
- 如果库存大于0，即马上进行库存扣除，并且访问抢购成功给用户
- 考虑原子性问题
  - 保证原子性的方式，采用lua脚本
  - 采用lua脚本方式保证原子性带来缺点，性能有所下降

- 不保证原子性缺点，放入请求量可能大于预期值
  - 当前扣除库存场景必须保证原子性，否则会导致超卖
- 返回抢购结果
  - 抢购成功
  - 库存没了，抢购失败



---

## 第十一章 分布式缓存课程回顾与展望

笔记请见资料里面的PPT