

第10章 Redis高可用

学习目标

- 了解Redis缓存集群方案
- 能够使用spring整合Redis集群环境

本章内容我们的主题为Redis，目前Redis在企业中的应用已经非常广泛，同时Redis也是面试中的重点内容。

1. Redis缓存相关问题

【目标】

- 1：了解Redis缓存穿透
- 2：了解Redis缓存雪崩
- 3：了解Redis缓存击穿

【路径】

- 1：缓存穿透描述，及解决方案
- 2：缓存雪崩描述，及解决方案
- 3：缓存击穿描述，及解决方案

【讲解】

1.1. 缓存穿透 【面试】

缓存穿透 是指查询一个数据库一定不存在的数据。

我们以前正常的使用Redis缓存的流程大致是：

第一次查询 先从jedis.get(setmealDetail{id})，如果不存在则查询数据库，获取后存入redis(setmealDetail{id})

如果套餐数据不存在，就不需要放入缓存

- 1、数据查询首先进行缓存查询 id=1setmealDetail_id = null
- 2、如果数据存在则直接返回缓存数据
- 3、如果数据不存在，就对数据库进行查询，并把查询到的数据放进缓存 1 null
- 4、如果数据库查询数据为空，则不放进缓存

例如我们的数据表中主键是自增产生的，所有的主键值都大于0。此时如果用户传入的参数为-1，会怎么样？这个-1，就是一定不存在的对象。程序就会每次都去查询数据库，而每次查询都是空，每次又都不会进行缓存。假如有人恶意攻击，就可以利用这个漏洞，对数据库造成压力，甚至压垮我们的数据库。

解决方案：为了防止有人利用这个漏洞恶意攻击我们的数据库，我们可以采取如下措施：

如果从数据库查询的对象为空，也放入缓存，key为用户提交过来的主键值，value为null，只是设定的缓存过期时间较短，比如设置为60秒。这样下次用户再根据这个key查询redis缓存就可以查询到值了（当然值为null），从而保护我们的数据库免遭攻击。到期就消失为了以后有真的数据也要返回真实数据

如果id有序，存入null值时，可以存入id的最大值（有数据），查询时判断查询id的范围是否大于最大值,直接返回null

1.2. 缓存雪崩【面试】

现象描述：缓存雪崩，是指在某一个时间段，缓存集中过期失效。在缓存集中失效的这个时间段对数据的访问查询，都落到了数据库上，对于数据库而言，就会产生周期性的压力波峰。

解决方案：为了避免缓存雪崩的发生，我们可以将缓存的数据设置不同的失效时间，这样就可以避免缓存数据在某个时间段集中失效。例如对于热门的数据（访问频率高的数据）可以缓存的时间长一些，对于冷门的数据可以缓存的时间短一些。甚至对于一些特别热门的数据可以设置永不过期(内存的开销)。还有其它第三方的缓存（Memory Cache, Jvm中的缓存: Spring Cache, Mybatis 二级缓存）

设置不同的失效时长 质数

延长有效期 expire key 时间

添加第三方缓存 重要的业务数据,高频访问的

1.3. 缓存击穿【面试】

现象描述：缓存击穿，是指一个key非常热点（例如双十一期间进行抢购的商品数据），在不停的扛着大并发，大并发集中对这一个点进行访问，当这个key在失效的瞬间，持续的大并发就穿破缓存，直接请求到数据库上，就像在一个屏障上凿开了一个洞。

解决方案：我们同样可以将这些热点数据设置永不过期就可以解决缓存击穿的问题了。或者阶段性的延长过期时间

【小结】

1：缓存穿透现象(查询不存在的数据)，及解决方案 存null, 设置较短的过期失效时间, 对有序的数据，设置null值的范围

2：缓存雪崩现象（大量的缓存在同一时失效），及解决方案 设置不同的失效时间,延长有效时间，辅以第三方缓存

3：缓存击穿现象(高并发的key失效瞬间)，及解决方案，延长过期时间甚至永久有效。定时或人工删除

1.4 Redis性能测试

1.4.1 目标

- ☐ 掌握Redis性能测试方法

1.4.2 路径

1. Redis的安装
2. redis-benchmark工具 使用说明
3. TPS、QPS、RT 讲解
4. 测算Redis性能

1.4.3 讲解

1.4.3.1 安装redis

在虚拟机（vm-redis、用户名root密码itcast），设置网络为net模式。装c++环境：

```
1 | yum install gcc-c++
```

安装Redis，依次执行以下命令：

```
1  #上传redis-4.0.14.tar.gz
2  put
3  # 解压
4  tar -zxf redis-4.0.14.tar.gz
5  # 进入解压目录
6  cd redis-4.0.14
7  # 编译
8  make
9  # 安装
10 make install PREFIX=/usr/local/redis
11 # 进入安装好的redis目录
12 cd /usr/local/redis/bin
13 # 复制配置文件
14 cp /root/redis-4.0.14/redis.conf ./
15
16 #设置环境变量
17 vi /etc/profile
18 #在文件的最后unset前添加
19 export REDIS_HOME=/usr/local/redis
20 export PATH=$PATH:$REDIS_HOME/bin
21 #使用环境变量生效
22 source /etc/profile
23
24 # 修改配置文件
25 vi redis.conf
26 # Redis后台启动
27 修改 daemonize 为 yes
28 # Redis服务器可以跨网络访问
29 修改 bind 为 0.0.0.0
30 # 开启aof持久化
31 appendonly yes
32
33 # 启动redis
34 redis-server redis.conf
35
```

1.4.3.2 redis-benchmark

redis-benchmark是官方自带的Redis性能测试工具，用来测试Redis在当前环境下的读写性能。我们在使用Redis的时候，服务器的硬件配置、网络状况、测试环境都会对Redis的性能有所影响，我们需要对Redis实时测试以确定Redis的实际性能。

使用语法：

```
1 | redis-benchmark [参数] [参数值]
```

参数说明：

选项	描述	默认值
-h	指定服务器主机名	127.0.0.1
-p	指定服务器端口	6379
-s	指定服务器 socket	
-c	指定并发连接数	50
-n	指定请求数	10000
-d	以字节的形式指定 SET/GET 值的数据大小	3
-k	1=keep alive 0=reconnect	1
-r	SET/GET/INCR 使用随机 key, SADD 使用随机值	
-P	通过管道传输 请求	1
-q	强制退出 redis。仅显示 query/sec 值	
--csv	以 CSV 格式输出	
-l	生成循环，永久执行测试	
-t	仅运行以逗号分隔的测试命令列表。	
-I	Idle 模式。仅打开 N 个 idle 连接并等待。	

常用的：-c -n -d -t

执行以下命令，测试性能：

```
1 | # 执行测试性能命令
2 | ./redis-benchmark -t set,get -n 100000
```

执行结果如下：

```
1 | ===== SET =====
2 | 100000 requests completed in 1.97 seconds
3 | 50 parallel clients
4 | 3 bytes payload
5 | keep alive: 1
6 |
7 | 95.32% <= 1 milliseconds
```

```
8 99.46% <= 2 milliseconds
9 99.67% <= 3 milliseconds
10 99.72% <= 4 milliseconds
11 99.84% <= 5 milliseconds
12 99.88% <= 6 milliseconds
13 99.90% <= 10 milliseconds
14 99.95% <= 18 milliseconds
15 99.97% <= 19 milliseconds
16 100.00% <= 19 milliseconds
17 50761.42 requests per second
18
19 ===== GET =====
20 100000 requests completed in 1.92 seconds
21 50 parallel clients
22 3 bytes payload
23 keep alive: 1
24
25 97.49% <= 1 milliseconds
26 99.95% <= 16 milliseconds
27 100.00% <= 16 milliseconds
28 52110.47 requests per second
```

在上面的测试结果中，我们关注GET结果最后一行 `52110.47 requests per second`，即每秒GET命令处理52110.47个请求，即QPS5.2万。但这里的数据都只是理想的测试数据，测出来的QPS不能代表实际生产的处理能力。

1.4.3.3 TPS、QPS、RT

在描述系统的高并发能力时，吞吐量（TPS）、QPS、响应时间（RT）经常提到，我们先了解这些概念：

- 响应时间RT
- 吞吐量TPS
- 每秒查询率QPS

响应时间(RT) Response Time

响应时间是指系统对请求作出响应的的时间。

直观上看，这个指标与人对软件性能的主观感受是非常一致的，因为它完整地记录了整个计算机系统处理请求的时间。由于一个系统通常会提供许多功能，而不同功能的业务逻辑也千差万别，因而不同功能的响应时间也不尽相同。

在讨论一个系统的响应时间时，通常是指该系统所有功能的**平均时间**或者所有功能的**最大响应时间**。

吞吐量TPS

吞吐量是指系统在单位时间(s秒)内处理请求的数量。每秒可以处理多少个请求

对于一个多用户的系统，如果只有一个用户使用系统的平均响应时间是 t ，当有 n 个用户使用，每个用户看到的响应时间通常并不是 $n \times t$ ，而往往比 $n \times t$ 小很多。这是因为在处理单个请求时，在每个时间点都可能有许多资源被闲置，当处理多个请求时，如果资源配置合理，每个用户看到的平均响应时间并不随用户数的增加而线性增加。

实际上，不同系统的平均响应时间随用户数增加而增长的速度也不大相同，这也是采用吞吐量来度量并发系统的性能的主要原因。一般而言，吞吐量是一个比较通用的指标，两个具有不同用户数和用户使用模式的系统，如果其最大吞吐量基本一致，则可以判断两个系统的处理能力基本一致。

```
1 1.服务器的并发为100,响应时间为100ms 1000ms/100ms每次 = 10次 每交可以并发处理100个请求, 因此1秒可以处理10次*100请求=1000个请求, 所以TPS为1000
2
3 2.TPS计算(1s):
4     1个并发    1000ms/100ms/次*1=10                    1个并发时, 可以处理10个请求
TPS:10
5     10个并发    1000ms/100ms*10=100                    10个并发时, 可以处理100请求
TPS:100
6     100个并发    1000/100*100=1000                    100个并发时, 可以处理1000请求 TPS:1000
7     200个并发    100个并发处理,100并发等待 吞吐量<1000
8                     (1000/100) / (200/100) 5 次请求 1000个请求
9     TPS, 是跟并发数有关系, 如果超出了的并发量处理, 会有所损耗 小于理论值
```

每秒查询率QPS

每秒查询率QPS是对一个特定的查询服务器在规定时间内所处理流量多少的衡量标准，在互联网中，经常用每秒查询率来衡量服务器的性能。对应fetches/sec，即每秒的响应请求数，也即是最大吞吐能力。

####1.4.3.4 测算Redis性能

在实际生产中，我们需要关心在应用场景中，redis能够处理的QPS是多少。我们需要估计生产的报文大小，使用benchmark工具指定-d数据块大小来模拟：

```
1 | ./redis-benchmark -t get -n 100000 -c 100 -d 2048
```

测试结果：

```
1  ===== GET =====
2  100000 requests completed in 2.33 seconds
3  100 parallel clients
4  2048 bytes payload
5  keep alive: 1
6
7  0.00% <= 1 milliseconds
8  51.74% <= 2 milliseconds
9  99.16% <= 3 milliseconds
10 99.57% <= 4 milliseconds
11 99.71% <= 5 milliseconds
12 99.83% <= 7 milliseconds
13 99.83% <= 8 milliseconds
14 99.86% <= 15 milliseconds
15 99.86% <= 16 milliseconds
16 99.90% <= 20 milliseconds
17 99.93% <= 21 milliseconds
18 99.94% <= 26 milliseconds
19 99.96% <= 29 milliseconds
20 99.97% <= 36 milliseconds
21 99.99% <= 37 milliseconds
22 100.00% <= 37 milliseconds
23 42955.32 requests per second
```

测得的QPS是4.2万

我们也可以使用redis客户端登陆到redis服务中，执行info命令查看redis的其他信息，执行命令：

```
1 # 使用Redis客户端
2 ./redis-cli
3 # 在客户端中执行info命令
4 127.0.0.1:6379> info
```

查看结果（摘取部分结果）：

```
1 connected_clients:101 #redis连接数
2
3 used_memory:8367424 # Redis 分配的内存总量
4 used_memory_human:7.98M
5 used_memory_rss:11595776 # Redis 分配的内存总量(包括内存碎片)
6 used_memory_rss_human:11.06M
7 used_memory_peak:8367424
8 used_memory_peak_human:7.98M #Redis所用内存的高峰值
9 used_memory_peak_perc:100.48%
```

1.4.4 小结

RT-每个请求的平均响应时间、或最大响应时间

TPS-吞吐量，跟可支持并发数与响应速度有关，还与真实的并发量有关（非线性）

QPS-每秒查询速度，衡量网络性能

衡量服务器的性能，测试时要带上并发量与数据大小

-d 数据大小， -c并发数， -n请求数， -t 执行的多个命令以逗号分割

2.Redis集群方案

单机Redis的读写速度非常快，能够支持大量用户的访问。虽然Redis的性能很高，但是对于大型网站来说，每秒需要获取的数据远远超过单台redis服务所能承受的压力，所以我们迫切需要一种方案能够解决单台Redis服务性能不足的问题。这就需要使用到Redis的集群了。Redis集群有多种方案，下面分别进行讲解。

2.1. 主从复制(读写分离)【重点】

2.1.1 目标

☐ 掌握Redis读写分离搭建

2.1.2 路径

1. Redis主从复制介绍
2. Redis主从复制实现
3. Redis同步原理

2.1.3 讲解

2.1.3.1 Redis主从复制介绍

在前面我们已经测试过，如果只有一台服务器，QPS是4.2万，而在大型网站中，可能要求更高的QPS，很明显，一台服务器就不能满足需要了。

- 1 Redis在知乎的规模：
- 2 机器内存总量约 70TB，实际使用内存约 40TB。
- 3 平均每秒处理约 1500 万次请求，峰值每秒约 2000 万次请求。 #QPS 2000万
- 4 每天处理约 1 万亿余次请求。
- 5 单集群每秒处理最高每秒约 400 万次请求。

我们可以对读写能力扩展，采用读写分离的方式解决性能瓶颈。运行新的服务器（称为从服务器），让从服务器与主服务器进行连接，然后主服务器发送数据副本，从服务器通过网络根据主服务器的数据副本进行准实时更新（具体的更新速度取决于网络带宽）。

这样我们就有额外的从服务器处理读请求，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` 语句配置即可

在主从复制模式下Redis节点分为两种角色：主节点(也称为master)和从节点(也称为slave)。这种模式集群是由一个主节点和多个从节点构成。

原则：Master会将数据同步到slave，而slave不会将数据同步到master。Slave启动时会连接master来同步数据。



这是一个典型的分布式读写分离模型。我们可以利用master来处理写操作，slave提供读操作。这样可以有效减少单个机器的并发访问数量。

要实现主从复制这种模式非常简单，主节点不用做任何修改，直接启动服务即可。从节点需要修改redis.conf配置文件，加入配置：`slaveof <主节点ip地址> <主节点端口号>`，例如master的ip地址为192.168.200.129，端口号为6379，那么slave只需要在redis.conf文件中配置`slaveof 192.168.200.129 6379`即可。

分别连接主节点和从节点，测试发现主节点的写操作，从节点立刻就能看到相同的数据。但是在从节点进行写操作，提示 `READONLY You can't write against a read only slave` 不能写数据到从节点。

现在我们就可以通过这种方式配置多个从节点进行读操作，主节点进行写操作，实现读写分离。

2.1.3.2 主从复制实现

在这里，我们把3个redis实现的主从复制搭在一台虚拟机里，真正产线环境时，每个redis应该独立在一台电脑上的ip运行

我们即将搭建的主从复制结构如下



image-20200701153617726


```
1 # 复制redis
2 cd /usr/local
3 cp redis redis01 -R
4 cd redis01/bin
5 # 修改配置
6 vi redis.conf
7 修改 port 为 6380
8 从的那台redis最后添加 slaveof 192.168.175.128[为你虚拟机的ip] 6379
9 # 清空持久化文件
10 rm -rf dump.rdb
11 rm -rf appendonly.aof
12 # 启动
13 ./redis-server redis.conf
```

```
1 # 启动客户端
2
3 ./redis-cli -p 6380
```

步骤同上，配置6381的服务器

6379

```
1 192.168.175.129:6379> info
2
3 Replication
4
5 role:master
6 connected_slaves:2
7 slave0:ip=192.168.175.129,port=6380,state=online,offset=588,lag=0
8 slave1:ip=192.168.175.129,port=6381,state=online,offset=588,lag=0
```

6380

```
1 192.168.175.129:6380> info
2
3 Replication
4
5 role:slave
6 master_host:192.168.175.129
7 master_port:6379
```

6381

```
1 192.168.175.129:6381> info
2
3 Replication
4
5 role:slave
6 master_host:192.168.175.129
7 master_port:6379
```

测试：

主节点

set name itcast：可以执行

get name：可以执行

从节点

set name itcast：报错(error) ERR wrong number of arguments for 'get' command

get name：可以执行

2.1.3.3 Redis同步原理【面试】

通过上面的例子，我们知道redis的主从复制，主服务器执行写操作命令，从服务器会通过主服务器的数据的变化，同步数据到从服务器。但是如果主服务器下线，从服务器无法连接主服务器，那么数据同步该如何拿到不能连接主服务器这段时间的命令呢？

主从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**。

Redis数据库持久化有两种方式：RDB全量持久化和AOF增量持久化。

数据同步步骤：

1. redis2.8版本之前使用旧版复制功能SYNC，这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。单个key可以存储最大512M的值
- 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. 2.8之后使用PSYNC，具有完整重同步和部分重同步两种模式部分重同步两种模式。

第一种完整重同步：

1560235618330

第二种部分重同步：

1560236056210

功能由以下三个部分构成：

- 1) 主服务的复制偏移量（replication offset）和从服务器的复制偏移量。
- 2) 主服务器的复制积压缓冲区（replication backlog），默认大小为1M。
- 3) 服务器的运行ID，用于存储服务器标识：

如果从服务器断线重新连接，获取主服务器的运行ID与重接后的主服务器运行ID进行对比，

判断是不是原来的主服务器，从而决定是执行部分重同步(aof)，还是执行完整重同步(rdb)。

2.1.4 小结

1. 搭建就照的步骤即可, 或使用rw脚本
 - 从服务器的配置文件里添加slaveof 主的ip 端口
2. redis数据同步有2种方式:
 - RDB的全量, 主服务切换为另一个服务器时发生, 主与从不可用(阻塞)
 - AOF增量, 主服务收到的写操作, 处理后, 把写的命令发送给从服务, 从服务暂时断网重连后也会发生AOF同步
3. 手工主从切换
 - 把主shutdown
 - 把一个从执行slaveof no one 新主
 - 另一个从执行slaveof 新主ip 新主的端口

2.2. 哨兵sentinel【重点】

我们现在已经给Redis实现了主从复制, 可将主节点数据同步给从节点, 实现了读写分离, 提高Redis的性能。但是现在还存在一个问题, 就是在主从复制这种模式下只有一个主节点, 一旦主节点宕机, 就无法再进行写操作了。也就是说主从复制这种模式没有实现高可用。那么什么是高可用呢? 如何实现高可用呢?

2.2.1. 高可用介绍

高可用(HA)是分布式系统架构设计中必须考虑的因素之一, 它是通过架构设计减少系统不能提供服务的时间。保证高可用通常遵循下面几点:

1. 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。2个以上
2. 通过架构设计而保证系统高可用的, 其核心准则是: 冗余。
3. 实现自动故障转移。

2.2.2. Redis sentinel介绍

sentinel(哨兵)是用于监控redis集群中Master状态的工具, 其本身也是一个独立运行的进程, 是Redis的高可用解决方案, sentinel哨兵模式已经被集成在redis2.4之后的版本中。

sentinel可以监视一个或者多个redis master服务, 以及这些master服务的所有从服务; 当某个master服务下线时, 自动将该master下的某个从服务升级为master服务替代已下线的master服务继续处理请求, 并且其余从节点开始从新的主节点复制数据。

在redis安装完成后, 会有一个redis-sentinel的文件, 这就是启动sentinel的脚本文件, 同时还有一个sentinel.conf文件, 这个是sentinel的配置文件。

sentinel工作模式:



注意: 可能有些同学会有疑问, 现在我们已经基于sentinel实现了高可用, 但是如果sentinel挂了怎么办呢? 其实sentinel本身也可以实现集群, 也就是说sentinel也是高可用的。

2.2.3. Redis sentinel使用

1. 重新准备一个主从复制的集群6379主, 6380从, 6380从
2. 配置sentinel
3. 启动sentinel

2.2.3.1. 配置sentinel

Sentinel在redis的安装包中有，我们直接使用就可以了，但是先需要修改配置文件，执行命令：

```
1 cd /usr/local/redis/
2
3 # 复制sentinel配置文件
4 cp /usr/local/redis/sentinel.conf sentinel01.conf
5
6 # 修改配置文件：
7 vi sentinel01.conf
```

在sentinel01.conf配置文件中添加：

```
1 # 外部可以访问
2 bind 0.0.0.0
3 sentinel monitor mymaster 127.0.0.1 6379 1
4 sentinel down-after-milliseconds mymaster 10000
5 sentinel failover-timeout mymaster 60000
6 sentinel parallel-syncs mymaster 1
7
```

注意：如果有sentinel monitor mymaster 127.0.0.1 6379 2配置，则注释掉，因为我们现在只有1个哨兵Sentinel节点。

```
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
```

如果以上4个配置设置成新值，需要将之前的值最好全部注释掉，否则可能会有问题。

参数说明：

- sentinel monitor mymaster 192.168.200.129 6379 1
mymaster 主节点名,可以任意起名，但必须和后面的配置保持一致。
192.168.200.128 6379 主节点连接地址。
1 将主服务器判断为失效需要投票，这里设置至少需要 1个 Sentinel 同意。
- sentinel down-after-milliseconds mymaster 10000
设置Sentinel认为服务器已经断线所需的毫秒数。
- sentinel failover-timeout mymaster 60000
设置failover（故障转移）的过期时间。当failover开始后，在此时间内仍然没有触发任何failover操作，当前sentinel 会认为此次failover失败。
- sentinel parallel-syncs mymaster 1
设置在执行故障转移时，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，表示同时进行同步的从服务器越少，那么完成故障转移所需的时间就越长。

2.2.3.2. 启动sentinel

配置文件修改后，执行以下命令，启动sentinel：

```
1 [root@itheima redis]# ./bin/redis-sentinel ./sentinel01.conf
2 或者
3 [root@itheima redis]# /user/local/redis/bin/redis-sentinel sentinel01.conf
```

效果如下：



可以看到，6379是主服务，6380和6381是从服务。

2.2.3.3. 测试sentinel

我们在6379执行shutdown，关闭主服务，Sentinel提示如下：

```
1 +sdown master mymaster 192.168.200.129 6379 #主节点宕机
2 +odown master mymaster 192.168.200.129 6379 #quorum 1/1
3 +new-epoch 1
4 +try-failover master mymaster 192.168.200.129 6379 #尝试故障转移
5 +vote-for-leader 00a6933e0cfa2b1bf0c3aab0d6b7a1a6455832ec 1 #选举领导
6 +elected-leader master mymaster 192.168.200.129 6379
7 +failover-state-select-slave master mymaster 192.168.200.129 6379 #故障转移选
  择从服务
8 +selected-slave slave 192.168.200.129:6380 192.168.200.129 6380 @ mymaster
  192.168.200.129 6379
9 #故障转移状态发送 发送到6380
10 +failover-state-send-slaveof-noone slave 192.168.200.129:6380
  192.168.200.129 6380 @ mymaster 192.168.200.129 6379
11 +failover-state-wait-promotion slave 192.168.200.129:6380 192.168.200.129
  6380 @ mymaster 192.168.200.129 6379
12 +promoted-slave slave 192.168.200.129:6380 192.168.200.129 6380 @ mymaster
  192.168.200.129 6379
13 +failover-state-reconf-slaves master mymaster 192.168.200.129 6379
14 +slave-reconf-sent slave 192.168.200.129:6381 192.168.200.129 6381 @
  mymaster 192.168.200.129 6379
15 +slave-reconf-inprog slave 192.168.200.129:6381 192.168.200.129 6381 @
  mymaster 192.168.200.129 6379
16 +slave-reconf-done slave 192.168.200.129:6381 192.168.200.129 6381 @
  mymaster 192.168.200.129 6379
17 +failover-end master mymaster 192.168.200.129 6379 #故障转移结束，原来的主服务是
  6379
18 +switch-master mymaster 192.168.200.129 6379 192.168.200.129 6380 #转换主服
  务，由原来的6379转为现在的6380
19 +slave slave 192.168.200.129:6381 192.168.200.129 6381 @ mymaster
  192.168.200.129 6380
20 +slave slave 192.168.200.129:6379 192.168.200.129 6379 @ mymaster
  192.168.200.129 6380
21 +sdown slave 192.168.200.129:6379 192.168.200.129 6379 @ mymaster
  192.168.200.129 6380
```

根据提示信息，我们可以看到，6379故障转移到了6380，通过投票选择6380为新的主服务器。

在6380执行info

```
1 # Replication
2 role:master
3 connected_slaves:1
4 slave0:ip=127.0.0.1,port=6381,state=online,offset=80531,lag=1
```

在6381执行info

```
1 # Replication
2 role:slave
3 master_host:127.0.0.1
4 master_port:6380
5 master_link_status:up
```

故障转移如下图：



此时：6380是主节点，可以进行写操作。

但是如果6379重新启动，此时6380还是主节点，而6379是从节点。

####2.2.3.4. 原理【面试】

Sentinel主要是监控服务器的状态，并决定是否进行故障转移。如何进行故障转移在前面的部分已经给大家演示过人工的操作，那么Sentinel是如何判断服务是否下线呢，主要分为主观下线和客观下线：

- 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
 - 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，那么 Sentinel 就会将这个服务器标记为主观下线
- 客观下线
 - 概念：

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 ODOWN。（一个 Sentinel 可以通过向另一个 Sentinel 发送命令来询问对方是否认为给定的服务器已下线）
 - 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），而是使用了流言传播（Gossip）：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。
 - 注意点：

客观下线条件只适用于主服务器，对于其他类型的 Redis 实例，Sentinel 在将它们判断为下线前不不需要进行协商，所以从服务器或者其他 Sentinel 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 推选，并对失效的主服务器执行自动故障迁移操作。

####2.2.3.5 小结

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转移操作, 它会将失效主服务器的其中一个从服务器升级为新的主服务器, 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。**选举的依据**依次是: 网络连接正常->5秒内回复过 INFO命令->10*down-after-milliseconds内与主连接过的->从服务器优先级->复制偏移量->运行id 较小的。选出之后通过slaveif no one将该从服务器升为新主服务器

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。

- 从服务器优先级: redis.conf中的slave-priority配置项, 数值越小, 优先级越高
- 复制偏移量: master_repl_offset是主节点的复制偏移量, slaveX中的offset即对应从节点的复制偏移量, 两者的差值即主从的延迟量



- 运行时id



3.3.4 互联网冷备和热备

- 冷备
 - 概念:

冷备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:
 - 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“某一时间点上”的恢复
 - 在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, 采用归档模式(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
 - 难于维护, 要非常仔细小心

2.3. Redis内置集群cluster

2.3.1. Redis cluster介绍

Redis Cluster是Redis的内置集群，在Redis3.0推出的实现方案。在Redis3.0之前是没有这个内置集群的。Redis Cluster是无中心节点的集群架构，依靠Gossip协议协同自动化修复集群的状态。

Redis cluster在设计的时候，就考虑到了去中心化，去中间件，也就是说，集群中的每个节点都是平等的关系，都是对等的，每个节点都保存各自的数据和整个集群的状态。每个节点都和其他所有节点连接，而且这些连接保持活跃，这样就保证了我们只需要连接集群中的任意一个节点，就可以获取到其他节点的数据。

Redis cluster集群架构图如下：



2.3.2. 哈希槽方式分配数据【面试】

需要注意的是，这种集群模式下集群中每个节点保存的数据并不是所有的数据，而只是一部分数据。那么数据是如何合理的分配到不同的节点上的呢？

Redis 集群是采用一种叫做 哈希槽 (hash slot) 的方式来分配数据的。redis cluster 默认分配了 16384 个slot，当我们set一个key 时，会用 CRC16 算法来取模得到所属的 slot，然后将这个key 分到哈希槽区间的节点上，具体算法就是： $CRC16(key) \% 16384$ 。

假设现在有3个节点已经组成了集群，分别是：A, B, C 三个节点，它们可以是一台机器上的三个端口，也可以是三台不同的服务器。那么，采用 哈希槽 (hash slot) 的方式来分配16384个slot 的话，它们三个节点分别承担的slot 区间是：

- 节点A覆盖0 - 5460
- 节点B覆盖5461 - 10922
- 节点C覆盖10923 - 16383

那么，现在要设置一个key ,比如叫 my_name：

```
1 | set my_name itcast
```

按照redis cluster的哈希槽算法： $CRC16('my_name') \% 16384 = 2412$ 。那么就会把这个key 的存储分配到 节点A 上了。

2.3.3. Redis cluster的主从模式

redis cluster 为了保证数据的高可用性，加入了主从模式，一个主节点对应一个或多个从节点，主节点提供数据存取，从节点则是从主节点拉取数据备份，当这个主节点挂掉后，就会在这些从节点中选取一个来充当主节点，从而保证集群不会挂掉。

redis cluster加入了主从模式后的效果如下：



2.3.4. Redis cluster搭建

2.3.4.1. 准备Redis节点

为了保证可以进行选举，需要至少3个主节点。

每个主节点都需要至少一个从节点,所以需要至少3个从节点。

一共需要6台redis服务器，我们这里使用6个redis实例，端口号为7001~7006。

先准备一个干净的redis环境，复制原来的bin文件夹，清理后作为第一个redis节点，具体命令如下：

```
1 # 进入redis安装目录
2 cd /usr/local
3 # 复制redis
4 mkdir cluster
5 cp -R ./redis ./cluster/node1
6 # 删除持久化文件
7 cd cluster/node1/bin
8 rm -rf dump.rdb
9 rm -rf appendonly.aof
10 # 删除原来的配置文件
11 rm -rf redis.conf
12 # 复制新的配置文件
13 cp /root/redis-4.0.14/redis.conf ./
14 # 修改配置文件
15 vi redis.conf
```

集群环境redis节点的配置文件如下：放置到文件的最后。

```
1 # 不能设置密码，否则集群启动时会连接不上
2 # Redis服务器可以跨网络访问
3 bind 0.0.0.0
4 # 修改端口号（6个机器，分别配置7001,7002,7003,7004,7005,7006）
5 port 7001
6 # Redis后台启动
7 daemonize yes
8 # 开启aof持久化
9 appendonly yes
10 # 开启集群
11 cluster-enabled yes
12 # 集群的配置 配置文件首次启动自动生成（可以不配置）
13 cluster-config-file nodes_7001.conf
14 # 请求超时（可以不配置）
15 cluster-node-timeout 5000
```

回到cluster目录，第一个redis节点node1准备好之后，再分别复制5份，

```
1 cp -R node1/ node2
2 cp -R node1/ node3
3 cp -R node1/ node4
4 cp -R node1/ node5
5 cp -R node1/ node6
```

修改六个节点的端口号为7001~7006，修改bin/redis.conf配置文件即可

```
1 vi node2/bin/redis-conf
2 vi node3/bin/redis-conf
3 vi node4/bin/redis-conf
4 vi node5/bin/redis-conf
5 vi node6/bin/redis-conf
```

编写启动节点的脚本：

```
1 | vi start-all.sh
```

内容为：

```
1 | cd node1/bin
2 | ./redis-server redis-conf
3 | cd ..
4 | cd ..
5 | cd node2/bin
6 | ./redis-server redis-conf
7 | cd ..
8 | cd ..
9 | cd node3/bin
10 | ./redis-server redis-conf
11 | cd ..
12 | cd ..
13 | cd node4/bin
14 | ./redis-server redis-conf
15 | cd ..
16 | cd ..
17 | cd node5/bin
18 | ./redis-server redis-conf
19 | cd ..
20 | cd ..
21 | cd node6/bin
22 | ./redis-server redis-conf
23 | cd ..
24 | cd ..
```

```
1 | [root@itheima cluster]# ll
2 | 总用量 28
3 | drwxr-xr-x. 2 root root 4096 12月 9 22:00 node1
4 | drwxr-xr-x. 2 root root 4096 12月 9 22:02 node2
5 | drwxr-xr-x. 2 root root 4096 12月 9 22:03 node3
6 | drwxr-xr-x. 2 root root 4096 12月 9 22:03 node4
7 | drwxr-xr-x. 2 root root 4096 12月 9 22:03 node5
8 | drwxr-xr-x. 2 root root 4096 12月 9 22:04 node6
9 | -rw-r--r--. 1 root root 246 12月 9 22:07 start-all.sh
```

不过查看start-all.sh文件，发现-rw-r--r--表示没有权限，需要设置权限

设置脚本的权限，并启动：

```
1 | chmod 744 start-all.sh
2 | ./start-all.sh
```

使用命令 `ps -ef | grep redis` 查看效果如下：



以上的6个节点并不是集群，如何将6个节点设置为集群呢？

2.3.4.2. 启动Redis集群

redis集群的管理工具使用的是ruby脚本语言，安装集群需要ruby环境，先安装ruby环境：

```
1 # 安装ruby
2 yum -y install ruby ruby-devel rubygems rpm-build
3
4 # 升级ruby版本，redis4.0.14集群环境需要2.2.2以上的ruby版本
5 yum install centos-release-scl-rh
6 yum install rh-ruby23 -y
7 scl enable rh-ruby23 bash
8
9 # 查看ruby版本
10 ruby -v
```

下载符合环境要求的gem，下载地址如下：

<https://rubygems.org/gems/redis-4.1.0>

课程资料中已经提供了redis-4.1.0.gem，直接上传安装即可，安装命令：

```
1 # 使用sftp上传redis-4.1.0.gem
2 [root@itheima ~]# ll
3 总用量 185996
4 -rw-----. 1 root root      1438 5月 14 2019 anaconda-ks.cfg
5 -rw-r--r--. 1 root root     27338 5月 14 2019 install.log
6 -rw-r--r--. 1 root root      7572 5月 14 2019 install.log.syslog
7 -rw-r--r--. 1 root root 188607817 7月 28 2018 jdk-8u181-linux-i586.tar.gz
8 drwxrwxr-x. 6 root root      4096 3月 19 2019 redis-4.0.14
9 -rw-r--r--. 1 root root   1740967 11月 12 15:44 redis-4.0.14.tar.gz
10 -rw-r--r--. 1 root root     55808 11月 12 15:44 redis-4.1.0.gem
11 # 安装
12 gem install redis-4.1.0.gem
```

进入redis安装目录，使用redis自带的集群管理脚本，执行命令：

```
1 # 进入redis安装包
2 cd /root//redis-4.0.14/src/
3 # 查看集群管理脚本
4 ll *.rb
5 # 使用集群管理脚本启动集群，下面命令中的1表示为每个主节点创建1个从节点（可在文本中打开编辑，再复制）
6 ./redis-trib.rb create --replicas 1 192.168.175.128:7001 192.168.175.128:7002
192.168.175.128:7003 192.168.175.128:7004 192.168.175.128:7005
192.168.175.128:7006
```

效果如下：

```
1 >>> Creating cluster
2 >>> Performing hash slots allocation on 6 nodes...
3 Using 3 masters:
4 192.168.200.129:7001
5 192.168.200.129:7002
6 192.168.200.129:7003
```

```

7 Adding replica 192.168.200.129:7005 to 192.168.200.129:7001
8 Adding replica 192.168.200.129:7006 to 192.168.200.129:7002
9 Adding replica 192.168.200.129:7004 to 192.168.200.129:7003
10 >>> Trying to optimize slaves allocation for anti-affinity
11 [WARNING] Some slaves are in the same host as their master
12 M: f0094f14b59c023acd38098336e2adcd3d434497 192.168.200.129:7001
13 slots:0-5460 (5461 slots) master
14 M: 0eba44418d7e88f4d819f89f90da2e6e0be9c680 192.168.200.129:7002
15 slots:5461-10922 (5462 slots) master
16 M: ac16c5545d9b099348085ad8b3253145912ee985 192.168.200.129:7003
17 slots:10923-16383 (5461 slots) master
18 S: edc7a799e1cfd75e4d80767958930d86516ffc9b 192.168.200.129:7004
19 replicates ac16c5545d9b099348085ad8b3253145912ee985
20 S: cbd415973b3e85d6f3ad967441f6bcb5b7da506a 192.168.200.129:7005
21 replicates f0094f14b59c023acd38098336e2adcd3d434497
22 S: 40fdde45b16e1ac85c8a4c84db75b43978d1e4d2 192.168.200.129:7006
23 replicates 0eba44418d7e88f4d819f89f90da2e6e0be9c680
24 Can I set the above configuration? (type 'yes' to accept): yes #注意选择为yes
25 >>> Nodes configuration updated
26 >>> Assign a different config epoch to each node
27 >>> Sending CLUSTER MEET messages to join the cluster
28 Waiting for the cluster to join..
29 >>> Performing Cluster Check (using node 192.168.200.129:7001)
30 M: f0094f14b59c023acd38098336e2adcd3d434497 192.168.200.129:7001
31 slots:0-5460 (5461 slots) master
32 1 additional replica(s)
33 M: ac16c5545d9b099348085ad8b3253145912ee985 192.168.200.129:7003
34 slots:10923-16383 (5461 slots) master
35 1 additional replica(s)
36 S: cbd415973b3e85d6f3ad967441f6bcb5b7da506a 192.168.200.129:7005
37 slots: (0 slots) slave
38 replicates f0094f14b59c023acd38098336e2adcd3d434497
39 S: 40fdde45b16e1ac85c8a4c84db75b43978d1e4d2 192.168.200.129:7006
40 slots: (0 slots) slave
41 replicates 0eba44418d7e88f4d819f89f90da2e6e0be9c680
42 M: 0eba44418d7e88f4d819f89f90da2e6e0be9c680 192.168.200.129:7002
43 slots:5461-10922 (5462 slots) master
44 1 additional replica(s)
45 S: edc7a799e1cfd75e4d80767958930d86516ffc9b 192.168.200.129:7004
46 slots: (0 slots) slave
47 replicates ac16c5545d9b099348085ad8b3253145912ee985
48 [OK] All nodes agree about slots configuration.
49 >>> Check for open slots...
50 >>> Check slots coverage...
51 [OK] All 16384 slots covered.

```

2.3.5. 使用Redis集群

按照redis cluster的特点，它是去中心化的，每个节点都是对等的，所以连接哪个节点都可以获取和设置数据。

查询集群信息：

```
1 | redis-cli -p 7001 cluster nodes
```



使用redis的客户端连接redis集群，命令如下：

```
1 # 进入到bin目录
2 cd /usr/local/cluster/node1/bin
3 # 启动客户端程序
4 ./redis-cli -h 192.168.175.128 -p 7001 -c
```

其中-c 一定要加，这个是redis集群连接时，进行节点跳转的参数。

连接到集群后可以设置一些值，可以看到这些值根据前面提到的哈希槽方式分散存储在不同的节点上了。

测试一：在7001的节点上添加数据：

```
1 192.168.175.128:7001> set key01 value01
2 -> Redirected to slot [13770] located at 127.0.0.1:7003
3 OK
4 127.0.0.1:7003> set key02 value02
5 -> Redirected to slot [1449] located at 127.0.0.1:7001
6 OK
7 127.0.0.1:7001> set key03 value03
8 -> Redirected to slot [5512] located at 127.0.0.1:7002
9 OK
10 127.0.0.1:7002> set key04 value04
11 OK
12 127.0.0.1:7002>
```

发现根据slot槽不同，数据存放的主节点也是不同的

测试二：在7004的节点上查询数据：

```
1 cd node4
2 ./redis-cli -h 192.168.175.128 -p 7004 -c
```

```
1 192.168.175.128:7004> get key01
2 -> Redirected to slot [13770] located at 127.0.0.1:7003
3 "value01"
4 127.0.0.1:7003> get key02
5 -> Redirected to slot [1449] located at 127.0.0.1:7001
6 "value02"
7 127.0.0.1:7001> get key03
8 -> Redirected to slot [5512] located at 127.0.0.1:7002
9 "value03"
10 127.0.0.1:7002> get key04
11 "value04"
12 127.0.0.1:7002>
```

发现，当取数据的时候，会从不同的节点取值。

测试三：关闭7001端口，7004会自动升级为主节点

第一步：断开7001

```
1 [root@itheima node1]# ./redis-cli -h 192.168.175.128 -p 7001 -c
2 192.168.175.128:7001> shutdown
3 not connected>
```

第二步：查看

```
1 [root@itheima node1]# ps -ef | grep redis
2 root      9084      1  0 22:27 ?        00:00:02 ./redis-server 0.0.0.0:7002
3 [cluster]
4 root      9089      1  0 22:27 ?        00:00:02 ./redis-server 0.0.0.0:7003
5 [cluster]
6 root      9091      1  0 22:27 ?        00:00:02 ./redis-server 0.0.0.0:7004
7 [cluster]
8 root      9096      1  0 22:27 ?        00:00:02 ./redis-server 0.0.0.0:7005
9 [cluster]
10 root      9104      1  0 22:27 ?        00:00:02 ./redis-server 0.0.0.0:7006
11 [cluster]
12 root      9361    9308  0 23:14 pts/6    00:00:00 grep  redis
```

第三步：查看7004，看是否升级为主节点

```
1 192.168.175.128:7004> info
2
3 Replication
4
5 role:master
6 connected_slaves:0
```

测试四：重新启动7001，此时7004已经是主节点，而7001变成从节点。

```
1 [root@itheima node1]# ./redis-server redis.conf
```

查看7001

```
1 192.168.175.128:7001> info
2 Replication
3 role:slave
4 master_host:127.0.0.1
5 master_port:7004
```

查看7004

```
1 192.168.175.128:7004> info
2 Replication
3 role:master
4 connected_slaves:1
5 slave0:ip=127.0.0.1,port=7001,state=online,offset=2230,lag=1
```

注意：

1: 需要每次启动linux, 都执行集群

```
1 进入redis安装包
2
3 cd /root/redis-4.0.14/src/
4
5 查看集群管理脚本
6
7 ll *.rb
8
9 使用集群管理脚本启动集群, 下面命令中的1表示为每个主节点创建1个从节点
10 ./redis-trib.rb create --replicas 1 192.168.175.128:7001
    192.168.175.128:7002 192.168.175.128:7003 192.168.175.128:7004
    192.168.175.128:7005 192.168.175.128:7006
```

2: 需要每次启动linux, 都执行关闭防火墙, 否则客户端无法连接

```
1 # centos7 64位
2 systemctl stop firewalld
3 systemctl iptables stop
```

3: 启动redis集群

```
1 [root@itheima cluster]# ./start-all.sh
```

【小结】

1: 主从复制Replication

从的redis上配置 slaveof 主的ip 端口

同步原理

rdb: 全量, 2.8以前, 或切换新主时

aof: 增量, 主写操作后, 就会同步到从

2: 哨兵Sentinel

(1) 高可用介绍 某一时刻系统出问题了, 功能依然可用 原则: 冗余 自动故障转移

(2) Redis sentinel介绍

哨兵, 监控集群中的redis节点, 主节点: 挂, 自动选举一个新的节点为主节点

选举原理:

(3) Redis sentinel使用

- 先准备主从复制集群, 配置sentinel
- 启动sentinel
- 测试sentinel

3: Redis内置集群cluster (推荐) 不要超过200个节点

(1) Redis cluster介绍

(2) 哈希槽方式分配数据 $\text{crc16}(\text{key})\%16384$ =槽的位置->redis所在=转到这个redis上执行命令

(3) Redis cluster的主从模式

(4) Redis cluster搭建集群

- 准备Redis节点
- 启动Redis集群

(5) 使用Redis集群

2.4 一致性哈希算法

2.4.1 目标

- ☐ 理解一致性哈希算法

2.4.2 路径

1. Redis内置集群缺点
2. 一致性哈希算法

2.4.3 讲解

2.4.3.1 Redis内置集群缺点

我们已经学完了Redis内置集群，是不是这种方式就足够我们使用了呢？在这里，我们要对redis集群现在使用的情况进行分析。

1. 集群使用现状

Redis Cluster内置集群，在Redis3.0才推出的实现方案。在3.0之前是没有这个内置集群的。

但是在3.0之前，有很多公司都有自己的一套Redis高可用集群方案。虽然现在有内置集群，但是因为历史原因，很多公司都没有切换到内置集群方案，而其原理就是集群方案的核心，这也是很多大厂为什么要问原理的原因。

2. 网络通信问题

Redis Cluster是无中心节点的集群架构，依靠Gossip协议（谣言传播）协同自动化修复集群的状态。

但Gossip有消息延时和消息冗余的问题，在集群节点数量过多的时候，节点之间需要不断进行PING/PANG通讯，不必须要的流量占用了大量的网络资源。虽然Redis4.0对此进行了优化，但这个问题仍然存在。

3. 数据迁移问题

Redis Cluster可以进行节点的动态扩容缩容，在扩缩容的时候，就需要进行数据迁移。

而Redis 为了保证迁移的一致性，迁移所有操作都是同步操作，执行迁移时，两端的Redis 均会进入时长不等的阻塞状态。对于小Key，该时间可以忽略不计，但如果一旦Key的内存使用过大，严重的时候会触发集群内的故障转移，造成不必要的切换。

以上原因说明只是学习Redis Cluster并不够，我们还需要学习新的集群方案。

- 1 **Gossip** 的缺陷
- 2 - 消息的延迟
- 3 由于 **Gossip** 协议中，节点只会随机向少数几个节点发送消息，消息最终是通过多个轮次的散播而到达全网的。
- 4 因此使用 **Gossip** 协议会造成不可避免的消息延迟。不适合用在对实时性要求较高的场景下。
- 5 - 消息冗余
- 6 **Gossip** 协议规定，节点会定期随机选择周围节点发送消息，而收到消息的节点也会重复该步骤。
- 7 因此存在消息重复发送给同一节点的情况，造成了消息的冗余，同时也增加了收到消息的节点的处理压力。
- 8 而且，由于是定期发送而且不反馈，因此即使节点收到了消息，还是会反复收到重复消息，加重了消息的冗余。
- 9 <https://www.cnblogs.com/rjzheng/p/11430592.html>

2.4.3.2 一致性哈希算法【面试】

#####2.4.3.2.1 分片介绍

在前面我们讲了内置的集群因为一些原因，在节点数量过多的时候，并不能满足我们的要求，哪还有什么新的集群方案呢？我们在这里讲解使用twemproxy实现hash分片的Redis集群方案，这个方案也是知乎2000万QPS场景所使用的方案。



上图我们看到twemproxy主要的角色是代理服务器的作用，是对数据库进行分片操作。twemproxy的分片保证需要存储的数据散列存放在集群的节点上，尽量做到平均分布。如何实现呢，这里就涉及到一致性哈希算法，这个算法是分布式系统中常用的算法。

2.4.3.2.2 传统哈希方案

传统方案是使用对象的哈希值，对节点个数取模，再映射到相应编号的节点，这种方案在节点个数变动时，绝大多数对象的映射关系会失效而需要迁移。

- 1 **Hash**，一般翻译做散列、杂凑，或音译为哈希，是把任意长度的输入，通过散列算法变换成固定长度的输出，该输出就是散列值。散列值的空间通常远小于输入的空间，不同的输入可能会散列成相同的输出，所以不可能从散列值来确定唯一的输入值。
- 2 简单的说就是一种将任意长度的消息压缩到某一固定长度的消息摘要的函数。

正常有3个节点，取3的模，分配数据，效果如下图：



如果节点挂了一个，那么就需要进行数据迁移，把数据分配到剩下的两个节点上，如下图：




可以看到原本存在Master1上的key3，需要迁移到Master3上，而Master1始终是正常的，这就造成了没有必要的数据迁移，浪费资源，所以我们需要采取另一种方式，一致性哈希算法。

#####2.4.3.2.3 一致性哈希算法

一致性哈希算法(Consistent Hashing Algorithm)是一种分布式算法，常用于负载均衡。twemproxy也选择这种算法，解决将key-value均匀分配到众多 server上的问题。它可以取代传统的取模操作，解决了取模操作应对增删 Server的问题。

步骤

1. 先用hash算法将对应的节点ip哈希到一个具有 2^{32} (2147483647)次方个桶的空间中，即 $0 \sim (2^{32}) - 1$ 的数字空间。现在我们可以将这些数字头尾相连，连接成一个闭合的环形：



2. 当用户在客户端进行请求时候，首先根据key计算路由hash值，然后看hash值落到了hash环的哪个地方，根据hash值在hash环上的位置顺时针找距离最近的节点：



3. 当新增节点的时候，和之前的做法一样，只需要把受到影响的数据迁移到新节点即可

新增Master4节点：



4. 当移除节点的时候，和之前的做法一样，把移除节点的数据，迁移到顺时针距离最近的节点

移除Master2节点：



从上面的步骤可以看出，当节点个数变动时，使用哈希一致性映射关系失效的对象非常少，迁移成本也非常小。那么判断一个哈希算法好坏的指标有哪些呢？以下列出了3个指标：

- 平衡性 (Balance)：

平衡性是指哈希的结果能够尽可能分散到不同的缓存服务器上去，这样可以使得所有的服务器得到利用。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

- 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，哈希的结果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量避免尽量降低分散性。而一致性hash具有很低的分散性。

2.4.3.2.4 虚拟节点

一部分节点下线之后，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。

在刚才的例子当中，如果Master3节点也挂掉，那么一致性hash倾斜就很明显了：



可以看到，理论上Master1需要存储25%的数据，而Master4要存储75%的数据。

上面这个例子中，我们可以对已有的两个节点创建虚拟节点，每个节点创建两个虚拟节点。那么实际的Master1节点就变成了两个虚拟节点Master1-1和Master1-2，而另一个实际的Master4节点就变成了两个虚拟节点Master4-1和Master4-2，这个时候数据基本均衡了：

1560356902393

2.4.4 小结

1. 通过一个哈希环，在环上分布很多存储节点($2^{32}-1$)，存储数据时，对key进行hash运算(hash算法)，放在顺时针方向上最近的节点。尽量让数据分布均匀
2. 如何判断一个算法是否均衡：平衡性(数据分布是否均匀)，单调性（保证原有不变，新的进入新的节点），分散性(当节点减少时，是否会变成分布不均匀)
3. 使用虚拟节点（原有的节点虚拟出新的节点）解决哈希倾斜

Redis集群总结

1. 主从复制
2. sentinel 工作中需要搭建的【重点】，一般企业
3. cluster写的并发数更高时

3.项目中使用Redis集群环境【了解】

spring boot RedisTemplate

应用场景：移动端系统，发送验证码，使用redis集群方式存储

【目标】

- 1：项目中使用Redis集群
- 2：spring配置Redis集群

【路径】

- 1：坐标环境
- 2：配置文件
- 3：RedisTemplate的使用
 - 发送验证码
 - 清理垃圾图片

【讲解】

3.1. 坐标环境

第一步：导入spring-data-redis坐标

health_parent父工程

```

1 <properties>
2     <spring.data.redis>2.1.6.RELEASE</spring.data.redis>
3 </properties>
4 <dependency>
5     <groupId>org.springframework.data</groupId>
6     <artifactId>spring-data-redis</artifactId>
7     <version>${spring.data.redis}</version>
8 </dependency>

```

health_common子工程

```

1 <dependency>
2     <groupId>org.springframework.data</groupId>
3     <artifactId>spring-data-redis</artifactId>
4 </dependency>

```

3.2. 配置文件

第一步：配置redis

health_mobile项目的resources下

配置redis.properties

```

1 ###redis集群推送任务信息缓存###
2 spring.redis.cluster.nodes1.host=192.168.175.129
3 spring.redis.cluster.nodes1.port=7001
4 spring.redis.cluster.nodes2.host=192.168.175.129
5 spring.redis.cluster.nodes2.port=7002
6 spring.redis.cluster.nodes3.host=192.168.175.129
7 spring.redis.cluster.nodes3.port=7003
8 spring.redis.cluster.nodes4.host=192.168.175.129
9 spring.redis.cluster.nodes4.port=7004
10 spring.redis.cluster.nodes5.host=192.168.175.129
11 spring.redis.cluster.nodes5.port=7005
12 spring.redis.cluster.nodes6.host=192.168.175.129
13 spring.redis.cluster.nodes6.port=7006
14 ## Redis数据库索引(默认为0)
15 spring.redis.database=0
16 ## 连接超时时间(毫秒)
17 spring.redis.timeout=60000
18 ## 最大重试次数
19 spring.redis.maxRedirects=3
20 ## 连接池最大连接数(使用负值表示没有限制)如果是集群就是每个ip的连接数
21 spring.redis.pool.max-active=500
22 ## 连接池最大阻塞等待时间(使用负值表示没有限制)
23 spring.redis.pool.max-wait=-1
24 ## 连接池中的最大空闲连接
25 spring.redis.pool.max-idle=300
26 ## 连接池中的最小空闲连接
27 spring.redis.pool.min-idle=100

```

第二步：配置spring-redis.xml

```

1 <?xml version="1.0" encoding="UTF-8"?>

```

```

2  <beans xmlns="http://www.springframework.org/schema/beans"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xmlns:context="http://www.springframework.org/schema/context"
5      xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"
6      xmlns:mvc="http://www.springframework.org/schema/mvc"
7      xsi:schemaLocation="http://www.springframework.org/schema/beans
8                          http://www.springframework.org/schema/beans/spring-
beans.xsd
9                          http://www.springframework.org/schema/mvc
10                         http://www.springframework.org/schema/mvc/spring-mvc.xsd
11                         http://code.alibabatech.com/schema/dubbo
12                         http://code.alibabatech.com/schema/dubbo/dubbo.xsd
13                         http://www.springframework.org/schema/context
14                         http://www.springframework.org/schema/context/spring-
context.xsd">
15     <!--加载redis.properties常量配置-->
16     <context:property-placeholder location="classpath:redis.properties">
</context:property-placeholder>
17
18
19     <!-- redis集群开始 -->
20     <!-- redis template definition -->
21     <bean id="redisTemplate"
class="org.springframework.data.redis.core.RedisTemplate">
22         <property name="connectionFactory" ref="jedisConnectionFactory" />
23         <property name="keySerializer">
24             <bean
class="org.springframework.data.redis.serializer.StringRedisSerializer" />
25             </property>
26             <property name="valueSerializer">
27                 <bean
class="org.springframework.data.redis.serializer.JdkSerializationRedisSerial
izer" />
28                 </property>
29                 <property name="hashKeySerializer">
30                     <bean
class="org.springframework.data.redis.serializer.StringRedisSerializer" />
31                     </property>
32                     <property name="hashValueSerializer">
33                         <bean
class="org.springframework.data.redis.serializer.JdkSerializationRedisSerial
izer" />
34                         </property>
35                     </bean>
36
37                 <!-- Spring-redis连接池管理工厂 -->
38                 <bean id="jedisConnectionFactory"
class="org.springframework.data.redis.connection.jedis.JedisConnectionFactory"
destroy-method="destroy">
39                     <constructor-arg ref="redisClusterConfiguration" />
40                     <constructor-arg ref="jedisPoolConfig" />
41                     <!-- Redis数据库索引(默认为0) -->
42                     <property name="database" value="${spring.redis.database}"/>
43                 </bean>
44
45                 <!-- 集群配置 -->
46                 <bean id="redisClusterConfiguration"
class="org.springframework.data.redis.connection.RedisClusterConfiguration">

```

```

47         <property name="clusterNodes">
48             <set>
49                 <ref bean="clusterRedisNodes1"/>
50                 <ref bean="clusterRedisNodes2"/>
51                 <ref bean="clusterRedisNodes3"/>
52                 <ref bean="clusterRedisNodes4"/>
53                 <ref bean="clusterRedisNodes5"/>
54                 <ref bean="clusterRedisNodes6"/>
55             </set>
56         </property>
57         <property name="maxRedirects" value="${spring.redis.maxRedirects}"
/>
58     </bean>
59     <!-- 集群节点 -->
60     <bean id="clusterRedisNodes1"
class="org.springframework.data.redis.connection.RedisNode">
61         <constructor-arg value="${spring.redis.cluster.nodes1.host}" />
62         <constructor-arg value="${spring.redis.cluster.nodes1.port}"
type="int" />
63     </bean>
64     <bean id="clusterRedisNodes2"
class="org.springframework.data.redis.connection.RedisNode">
65         <constructor-arg value="${spring.redis.cluster.nodes2.host}" />
66         <constructor-arg value="${spring.redis.cluster.nodes2.port}"
type="int" />
67     </bean>
68     <bean id="clusterRedisNodes3"
class="org.springframework.data.redis.connection.RedisNode">
69         <constructor-arg value="${spring.redis.cluster.nodes3.host}" />
70         <constructor-arg value="${spring.redis.cluster.nodes3.port}"
type="int" />
71     </bean>
72     <bean id="clusterRedisNodes4"
class="org.springframework.data.redis.connection.RedisNode">
73         <constructor-arg value="${spring.redis.cluster.nodes4.host}" />
74         <constructor-arg value="${spring.redis.cluster.nodes4.port}"
type="int" />
75     </bean>
76     <bean id="clusterRedisNodes5"
class="org.springframework.data.redis.connection.RedisNode">
77         <constructor-arg value="${spring.redis.cluster.nodes5.host}" />
78         <constructor-arg value="${spring.redis.cluster.nodes5.port}"
type="int" />
79     </bean>
80     <bean id="clusterRedisNodes6"
class="org.springframework.data.redis.connection.RedisNode">
81         <constructor-arg value="${spring.redis.cluster.nodes6.host}" />
82         <constructor-arg value="${spring.redis.cluster.nodes6.port}"
type="int" />
83     </bean>
84     <!-- 集群节点 -->
85     <!--&lt;t;!&ndash; redis集群结束 &ndash;&gt;-->
86
87     <bean id="jedisPoolConfig" class="redis.clients.jedis.JedisPoolConfig">
88         <property name="maxTotal" value="${spring.redis.pool.max-active}" />
89         <property name="maxIdle" value="${spring.redis.pool.max-idle}" />
90         <property name="minIdle" value="${spring.redis.pool.min-idle}" />

```

```

91         <property name="maxwaitMillis" value="${spring.redis.pool.max-wait}"
    />
92         <property name="testOnBorrow" value="true" />
93         <property name="testOnReturn" value="true"/>
94     </bean>
95 </beans>

```

第三步: springmvc.xml

引入spring-redis.xml

```

1 <import resource="classpath:spring-redis.xml"></import>

```

3.3. RedisTemplate的使用

3.3.1 发送验证码

第一步: 修改ValidateCodeMobileController, 用于存放Redis数据

```

1 @Autowired
2 RedisTemplate redisTemplate;
3
4 @RequestMapping(value = "/send4Order")
5 public Result send4Order(String telephone){
6     ...
7     redisTemplate.opsForValue().set(telephone+
RedisMessageConstant.SENDTYPE_ORDER,code4.toString(),5, TimeUnit.MINUTES);
8 }
9
10 @RequestMapping(value = "/send4Login")
11 public Result send4Login(String telephone){
12     ...
13     redisTemplate.opsForValue().set(telephone+
RedisMessageConstant.SENDTYPE_LOGIN,code4.toString(),5, TimeUnit.MINUTES);
14 }

```

第二步: 修改OrderMobileController, 用于获取Redis数据

```

1 @Autowired
2 RedisTemplate redisTemplate;
3
4 // 提交预约的保存
5 @RequestMapping(value = "/submit")
6 public Result submit(@RequestBody Map map){
7     ...
8     String redisvalidateCode = (String)
redisTemplate.opsForValue().get(telephone +
RedisMessageConstant.SENDTYPE_ORDER);
9 }

```

第三步: 修改LoginMobileController, 用于获取Redis数据

```

1  @Autowired
2  private RedisTemplate redisTemplate;
3
4  // 登录校验
5  @RequestMapping(value = "/check")
6  public Result submit(@RequestBody Map map, HttpServletResponse response){
7      ...
8      String redisValidateCode = (String)
        redisTemplate.opsForValue().get(telephone +
        RedisMessageConstant.SENDTYPE_LOGIN);
9  }

```

启动healthmobile_web，使用手机号登陆页面，获取验证码功能完成测试



3.3.1 清理垃圾图片【不用了】

第一步：配置文件

1：复制redis.properties和spring-redis.xml放置到health_web中



2：复制redis.properties和spring-redis.xml放置到health_service中，将spring-redis.xml改名为applicationContext-redis.xml



3：复制redis.properties和spring-redis.xml放置到health_jobs中，将spring-redis.xml改名为applicationContext-redis.xml



第二步：修改SetmealServiceImpl.java，用于存放Redis数据，保存数据库的同时，存放图片名称

```

1  @Autowired
2  RedisTemplate redisTemplate;
3
4  // 保存套餐数据
5  @Override
6  public void add(Setmeal setmeal, Integer[] checkgroupIds) {
7      // 1: 新增套餐，向套餐表中添加1条数据
8      setmealDao.add(setmeal);
9      // 2: 新增套餐和检查组的中间表，想套餐和检查组的中间表中插入多条数据
10     if(checkgroupIds!=null && checkgroupIds.length>0){
11         setSetmealAndCheckGroup(setmeal.getId(),checkgroupIds);
12     }
13     // 3: 向Redis中集合的key值为setmealPicDbResources下保存数据，数据为图片的名称
14
15     // jedisPool.getResource().sadd(RedisConstant.SETMEAL_PIC_DB_RESOURCES,setmeal.getImg());
16
17     redisTemplate.opsForSet().add(RedisConstant.SETMEAL_PIC_DB_RESOURCES,setmeal.getImg());

```



```

16
17     //新增套餐后需要重新生成静态页面
18     generateMobileStaticHtml();
19 }
20
21
22 //编辑套餐，同时需要更新和检查组的关联关系
23 @Override
24 public void edit(Setmeal setmeal, Integer[] checkgroupIds) {
25     // 使用套餐id，查询数据库对应的套餐，获取数据库存放的img
26     Setmeal setmeal_db = setmealDao.findById(setmeal.getId());
27     String img = setmeal_db.getImg();
28     // 如果页面传递的图片名称和数据库存放的图片名称不一致，说明图片更新，需要删除七牛云之
    前数据库的图片
29     if(setmeal.getImg()!=null && !setmeal.getImg().equals(img)){
30         qiniuUtils.deleteFileFromQiniu(img);
31         //将图片名称从Redis中删除，key值为setmealPicDbResources
32
33         //jedisPool.getResource().srem(RedisConstant.SETMEAL_PIC_DB_RESOURCES,img);
34
35         redisTemplate.opsForSet().remove(RedisConstant.SETMEAL_PIC_DB_RESOURCES,img
    );
36         //将图片名称从Redis中删除，key值为setmealPicResources
37
38         //jedisPool.getResource().srem(RedisConstant.SETMEAL_PIC_RESOURCES,img);
39
40         redisTemplate.opsForSet().remove(RedisConstant.SETMEAL_PIC_RESOURCES,img);
41     }
42
43     //1: 根据套餐id删除中间表数据（清理原有关联关系）
44     setmealDao.deleteAssociation(setmeal.getId());
45     //2: 向中间表(t_setmeal_checkgroup)插入数据（建立套餐和检查组关联关系）
46     setSetmealAndCheckGroup(setmeal.getId(),checkgroupIds);
47     //3: 更新套餐基本信息
48     setmealDao.edit(setmeal);
49 }
50
51 // 删除套餐
52 @Override
53 public void deleteById(Integer id) {
54     // 使用套餐id，查询数据库对应的套餐，获取数据库存放的img
55     Setmeal setmeal_db = setmealDao.findById(id);
56
57     // 使用套餐id，查询套餐和检查组中间表
58     Integer count = setmealDao.findSetmealAndCheckGroupCountBySetmealId(id);
59     // 存在数据
60     if(count>0){
61         throw new RuntimeException("当前套餐和检查组之间存在关联关系，不能删除");
62     }
63     // 删除套餐
64     setmealDao.deleteById(id);
65
66     // 获取存放的图片信息
67     String img = setmeal_db.getImg();
68     // 需要先删除七牛云之前数据库的图片
69     if(img!=null && !"".equals(img)){
70         qiniuUtils.deleteFileFromQiniu(img);
71         //将图片名称从Redis中删除，key值为setmealPicDbResources

```

```

68 //jedisPool.getResource().srem(RedisConstant.SETMEAL_PIC_DB_RESOURCES,img);
69
70     redisTemplate.opsForSet().remove(RedisConstant.SETMEAL_PIC_DB_RESOURCES,img);
71 };
72     //将图片名称从Redis中删除, key值为setmealPicResources
73
74 //jedisPool.getResource().srem(RedisConstant.SETMEAL_PIC_RESOURCES,img);
75
76     redisTemplate.opsForSet().remove(RedisConstant.SETMEAL_PIC_RESOURCES,img);
77 }
78 }

```

修改SetmealController.java, 用于存放Redis数据, 上传图片的同时, 存放图片名称

```

1      @Autowired
2      RedisTemplate redisTemplate;
3
4      // 套餐图片的上传(springmvc的文件上传)
5      @RequestMapping(value = "/upload")
6      public Result upload(MultipartFile imgFile){
7          try {
8              // 之前的文件名
9              String fileName = imgFile.getOriginalFilename();
10             // 使用UUID的形式(时间戳), 保证文件名唯一
11             fileName =
12             UUID.randomUUID().toString()+fileName.substring(fileName.lastIndexOf("."));
13             // 七牛云上上传
14             QiniuUtils.upload2Qiniu(imgFile.getBytes(),fileName);
15             // 同时向Redis中的集合的key=setmealPicResource, 存放数据, 数据存放文件
16             名
17             //
18             jedisPool.getResource().sadd(RedisConstant.SETMEAL_PIC_RESOURCES,fileName);
19
20             redisTemplate.opsForSet().add(RedisConstant.SETMEAL_PIC_RESOURCES,fileName);
21             ;
22             // 响应文件名
23             return new Result(true,
24             MessageConstant.PIC_UPLOAD_SUCCESS,fileName);
25         } catch (Exception e) {
26             e.printStackTrace();
27             return new Result(false, MessageConstant.PIC_UPLOAD_FAIL);
28         }
29     }
30 }

```

第三步: 修改health_jobs, 使用定时任务, 清除垃圾图片。

```

1 // 任务类
2 public class ClearImgJob {
3
4     //      @Autowired
5     //      JedisPool jedisPool;
6
7     @Autowired
8     RedisTemplate redisTemplate;

```

```

9
10 // 任务类执行的方法
11 public void executeJob(){
12     //计算setmealPicResources集合与setmealPicDbResources集合的差值，清理图片
13     // Set<String> set =
jedisPool.getResource().sdiff(RedisConstant.SETMEAL_PIC_RESOURCES,
RedisConstant.SETMEAL_PIC_DB_RESOURCES);
14     Set<String> set =
redisTemplate.opsForSet().difference(RedisConstant.SETMEAL_PIC_RESOURCES,
RedisConstant.SETMEAL_PIC_DB_RESOURCES);
15     Iterator<String> iterator = set.iterator();
16     while(iterator.hasNext()){
17         String picName = iterator.next();
18         System.out.println("删除的图片名称: "+picName);
19         // 1: 删除七牛云的数据
20         QiniuUtils.deleteFileFromQiniu(picName);
21         // 2:: 删除key值为ssetmealPicResources 的redis的数据
22
//jedisPool.getResource().srem(RedisConstant.SETMEAL_PIC_RESOURCES,picName)
;
23
redisTemplate.opsForSet().remove(RedisConstant.SETMEAL_PIC_RESOURCES,picName
e);
24     }
25
26 }
27 }

```

启动health_job，使用定时任务清理垃圾图片。

小结

- 1: 坐标环境 spring-data-redis
- 2: 配置文件 比较繁琐，将来与springboot整合时变得很简单
- 3: RedisTemplate的使用 opsForValue (bit, String) opsForSet 操作集合
 - 发送验证码
 - 生成静态页面

项目描述:

开发环境和使用技术:

责任描述:

技术描述: