

第1章 双十一缓存架构设计

目标1：掌握缓存的分类、缓存的淘汰算法、缓存的应用场景、缓存在微服务架构中的使用

目标2：理解双十一架构痛点

目标3：理解多级缓存架构流程

目标4：能实现Redis缓存的扩容、收容

目标5：能实现Redis集群哨兵策略

目标6：能在项目中操作Redis集群

目标7：实现基于Nginx配置浏览器缓存

目标8：实现Nginx代理缓存配置

缓存：

- 1.Redis
- 2.Nginx-拥有缓存【热点数据】
- 3.浏览器-静态资源
- 4.消息队列

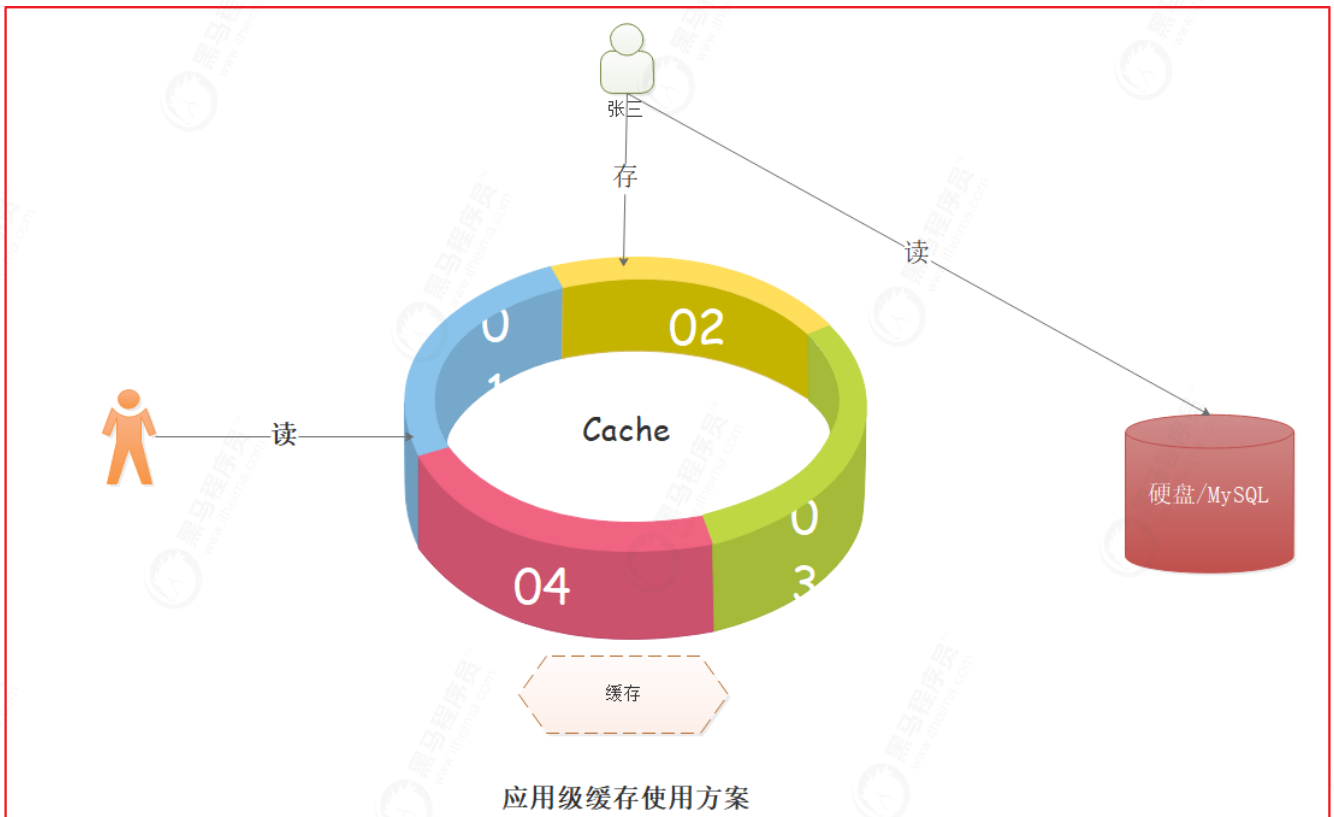
1 缓存的应用

缓存（cache），原始意义是指访问速度比一般随机存储器（RAM）快的一种高速存储器，设置缓存是为了更好的发挥计算机系统的高性能。

1.1 缓存分类

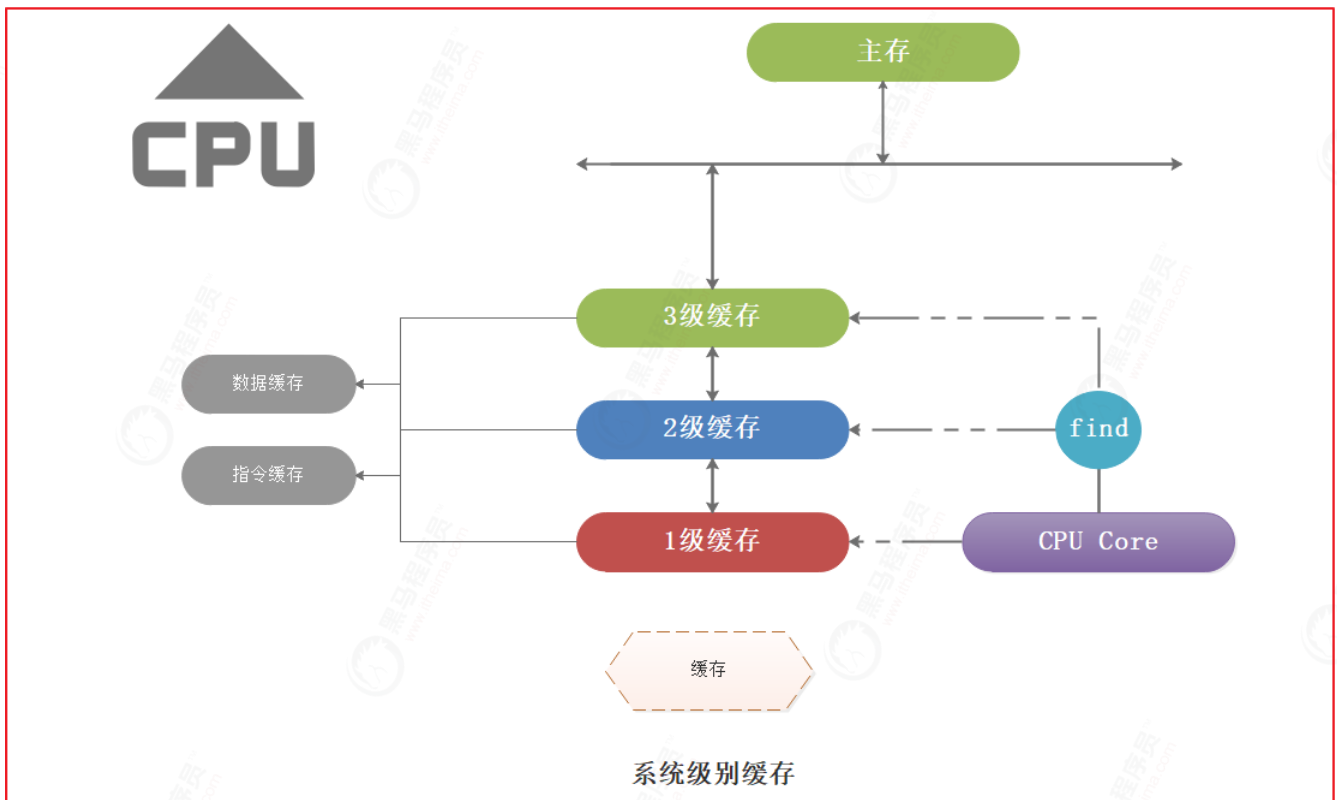
1)应用级缓存

应用级缓存也就是我们平时写的应用程序中所使用的缓存。在平时程序中一般是按照如下操作流程来实现缓存的操作，首先张三用户读取数据库，并将读取的数据存入到缓存中，其他用户读取的时候，直接从缓存中读取，而不用查询数据库，从而提高程序的执行速度和效率。



2)系统级别缓存

系统级别缓存是抛开我们应用程序之外硬件的缓存操作，例如某些CPU的缓存操作和如下图多级缓存流程类似，CPU在操作数据的时候，先读取1级缓存，1级缓存如果没有数据则读取2级缓存，2级缓存没有数据则读取3级缓存，3级缓存如果没有数据就直接从主存储器(存储指令和数据)读取数据。



1.2 缓存淘汰算法

数据缓存之后，为了避免缓存无限变大，我们需要对缓存进行管理，将一些不用的或者很少用的或者很久没更新的缓存淘汰掉，这里就需要用到一系列淘汰算法了，淘汰算法主要有如下几种：

1)FIFO（先进先出）

数据最先存入缓存，那么也将最先被淘汰掉。

2)LRU（最不常用数据）

判断缓存最近被使用的时间，时间最远被使用的缓存优先淘汰。

3)LFU（最少使用）

一段时间内，被使用的次数最少的数据，优先淘汰掉。

1.3 缓存应用场景

1)频繁查询数据缓存

有一些数据经常被访问，而且变更频率较低，实时性要求不高的数据，可以把它存储到缓存中，每次读取数据直接读缓存即可，从而提升数据的加载速度和系统的性能。

2)列表排序分页数据

一些变更频率较低查询频次较高的列表、分页、排序数据，可以存入到Redis缓存，每次查询分页或者排序的时候，直接从Redis缓存中获取。

3)计数器

网站中用于统计访问频次、在线人数、商品抢购次数等，也可以使用缓存来实现。

4)详情内容

站点中，资讯内容、商品详情等较大变更频率又低的内容，可以采用缓存来加速数据的读取，降低IO操作。

5)分布式Session

实现会话共享的时候，可以使用Session来存储需要共享的会话，从而节省内存空间。

6)热点排名

我们可以使用ZSet来存储热数据，并实现热点数据的排名。

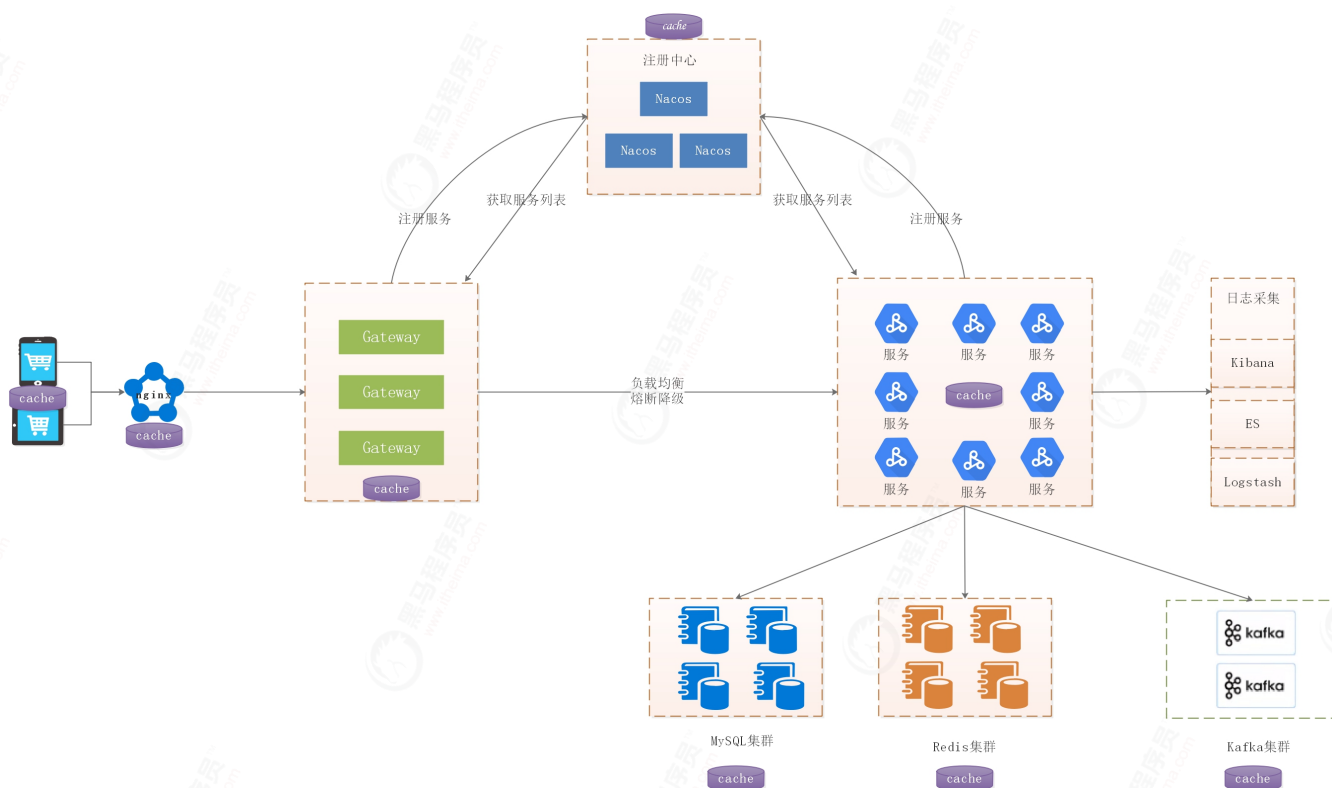
7)发布订阅

用Redis也可以实现发布与订阅，但不推荐，推荐用MQ。

8)分布式锁

可以使用Redisson结合Redis实现分布式锁，Redis实现的分布式锁效率极高，得到了市场的广泛使用。

1.4 微服务架构缓存的使用

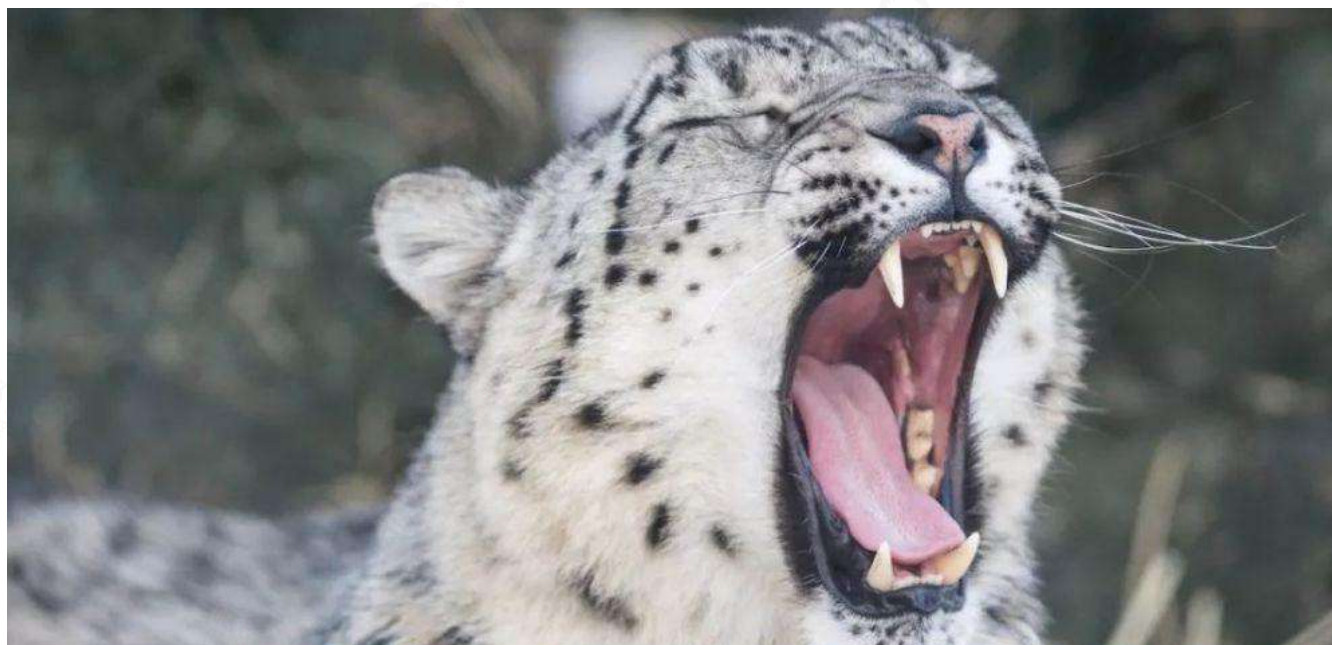


在微服务架构中，会大量使用到缓存，如上图：

1. 客户端缓存 (手机、PC)
2. Nginx缓存
3. 微服务网关限流令牌缓存
4. Nacos缓存服务列表、配置文件
5. 各大微服务自身也具有缓存
6. 数据库查询Query Cache
7. Redis集群缓存
8. Kafka也属于缓存

2 高并发站点缓存技术选型

双十一大促销活动有很多技术挑战，而最大的挑战之一就是高并发，而应对高并发的最有效手段之一就是分布式缓存，分布式缓存不仅仅是缓存要显示的数据这么简单，还可以在限流、队列削峰、高速读写、分布式锁等场景发挥重大作用。分布式缓存可以说是解决高并发场景的一头野兽。



2.1 双十一活动大促分析

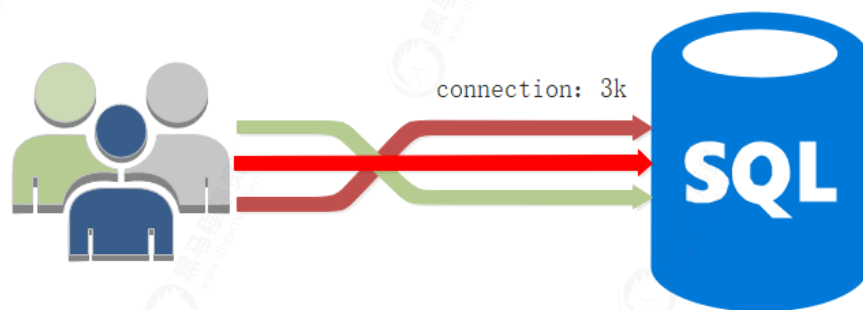
双十一无论抢红包、商品秒杀、抢优惠券，并发量都是超高，流量可以说是平时几倍乃至几十倍，对服务器造成的压力也几何数字增长，因此双十一必须要考虑各种场景的技术解决方案。

面对双十一巨大流量涌入，需要解决的问题我们做一个分析：

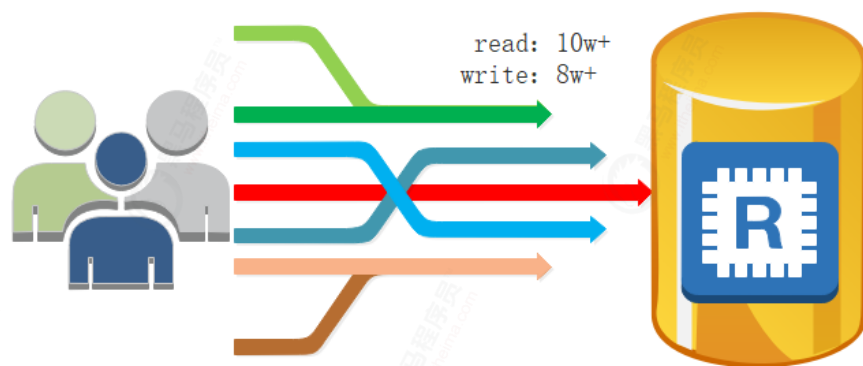
- 1、凌晨突然涌入的巨大流量。【队列削峰】【限流】
- 2、高并发场景秒杀、抢红包、抢优惠券，快速存取。【缓存】
- 3、高并发场景超卖、超额抢红包。
- 4、高并发场景重复抢单。【缓存解决】

2.2 高性能利器-缓存

传统关系型数据库能处理的并发链接一般都有限，如果优化的很优秀，并发链接量可高达几千，而一般公司对其优化的并不怎么乐观，很多公司的数据库并发链接量只控制在几百。单纯用数据库实现数据的高效存取是存在很大挑战和问题的，因此一定要找一种方式或者软件取代传统关系型数据库的存取功能。



针对传统关系型数据库存取性能瓶颈，我们可以采用高性能的非关系型数据库来取代传统关系型数据库，这里以当前主流的分布式缓存技术Redis为例，可以实现单机每秒读10万次以上，写每秒8万次以上的能力，远超关系型数据库，因此解决传统关系型数据库的存取能力，我们可以用非关系型数据库来优化甚至取代关系型数据库。



2.3 缓存性能对比

当前主流的缓存技术有 Memcached、Tair、Redis，这几款分布式缓存都各有优缺点，大家可以根据不同的使用场景来选择使用。

Memcached:

Memcache是老牌的内存缓存技术，对相关领域支持比较丰富，window和linux都可以使用，各种框架都支持使用，session的信息可以非常方便的保存到该Memcache中，每个key保存的数据量最大为1M，支持的数据类型比较单一，就是String类型，不支持持久化。

优点：高性能读写，单一的数据类型（key【String】-value【Object】），支持客户端式分布式集群，多线程读写性能高。

缺点：无持久化，服务端无法实现集群，跨机房数据同步困难，架构扩容复杂度高，不适合处理大规模数据量。

Redis：（推荐）

Redis支持比较多的数据类型(String/list/set/sortset/hash)，Redis支持集合计算的(set类型支持)，每个key最大数据存储量为1G，Redis是新兴的内存缓存技术，对各方面支持不完善，支持持久化操作。内存中的数据结构存储系统，可用作数据库、缓存和消息中间件。基本配合后端数据库使用，存放的只是用户当前频繁调去的数据。

优点：高性能读写，多数据类型支持，数据持久化，高可用架构，支持分布式分片集群，单线程读写性能极高（6.x版本支持多线程）。

缺点：多线程读写较Memcached慢，不适合处理大规模数据量。

Tair:

Tair是一个key/value结构数据的解决方案，它默认支持基于内存和文件的两种存储方式，分别与缓存和持久化存储对应。Tair的功能是get、put、delete以及批量接口，主要针对淘宝应用。

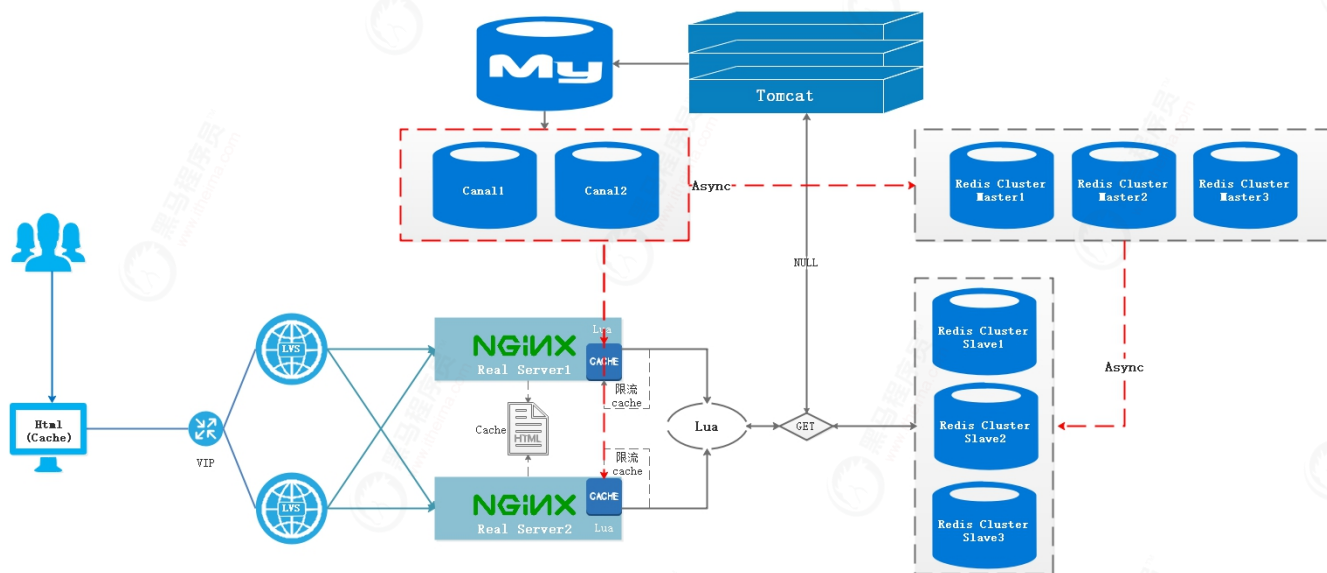
优点：高性能读写，支持三种存储引擎（ddb/rdb/lb），支持高可用，支持分布式分片集群，支撑了几乎所有淘宝业务的缓存。

缺点：单机情况下，读写性能较上两种较慢，适合处理大规模数据缓存。

3 双十一缓存架构设计

一谈到缓存架构，很多人想到的是Redis，但其实整套体系的缓存架构并非只有Redis，而应该是多个层面多个软件结合形成一套非常良性的缓存体系。比如咱们的缓存架构设计就涉及到了多个层面的缓存软件。

3.1 缓存架构设计



(缓存架构设计)

上图是黑马程序员专为双十一大促设计的缓存架构体系，该缓存架构体系借鉴了一线大厂京东缓存架构设计经验，并在京东设计方案上进行了优化，让它更符合一线厂家的需要。上述架构图综合了多种缓存和多层面的缓存设计，从前端页面缓存到代理服务器lvs和Nginx缓存，以及后端服务redis缓存，包括缓存数据同步等。

对上述架构，我们来个宏观解说：

- 1、HTML页面做缓存，浏览器端可以缓存HTML页面和其他静态资源，防止用户频繁刷新对后端造成巨大压力
- 2、Lvs实现记录不同协议以及不同用户请求链路缓存
- 3、Nginx这里会做HTML页面缓存配置以及Nginx自身缓存配置
- 4、数据查找这里用Lua取代了其他语言查找，提高了处理的性能效率，并发处理能力将大大提升
- 5、数据缓存采用了Redis集群+主从架构，并实现缓存读写分离操作
- 6、集成Canal实现数据库数据增量实时同步Redis

3.2 Redis集群高级应用

这里安装6个redis，配置如下：

Redis节点	IP	端口
redis-7001	172.18.0.2	7001
redis-7002	172.18.0.3	7002
redis-7003	172.18.0.4	7003
redis-7004	172.18.0.5	7004
redis-7005	172.18.0.6	7005
redis-7006	172.18.0.7	7006

小知识点(Redis6.0.5):

如果需要安装单机版，需要安装指定的插件

```
yum -y install gcc gcc-c++ libstdc++-devel
```

升级gcc

```
yum -y install centos-release-scl devtoolset-7
```

```
yum -y install devtoolset-9-gcc devtoolset-9-gcc-c++ devtoolset-9-binutils
```

切换gcc7

```
scl enable devtoolset-7 bash
```

单机版安装

解压redis-6.0.5.tar.gz

```
tar -xf redis-6.0.5.tar.gz
```

进入到redis-6.0.5目录

```
cd redis-6.0.5
```

编译安装

make

make install PREFIX=/usr/local/server/redis-cluster/redis

redis.conf拷贝到安装bin目录下

cp /usr/local/server/redis-cluster/redis-6.0.5/redis.conf /usr/local/server/redis-cluster/redis/bin/

修改redis.conf更改redis配置:

设置daemonize为yes 让redis支持后台启动

将protected-mode 改为no, 远程连接就不需要账号密码了

bind 192.168.211.141

注意安装Redis6.x的时候一定要切换到gcc7再安装, 否则会报如下错误:

```
server.c:5153:94: 错误: 'struct redisServer'没有名为'unixsocket'的成员
    serverLog(LL_NOTICE,"The server is now ready to accept connections at %s", server.unixsocket);
server.c:5154:19: 错误: 'struct redisServer'没有名为'supervised_mode'的成员
    if (server.supervised_mode == SUPERVISED_SYSTEMD) {
server.c:5155:24: 错误: 'struct redisServer'没有名为'masterhost'的成员
    if (!server.masterhost) {
server.c:5168:15: 错误: 'struct redisServer'没有名为'maxmemory'的成员
    if (server.maxmemory > 0 && server.maxmemory < 1024*1024) {
server.c:5168:39: 错误: 'struct redisServer'没有名为'maxmemory'的成员
    if (server.maxmemory > 0 && server.maxmemory < 1024*1024) {
server.c:5169:176: 错误: 'struct redisServer'没有名为'maxmemory'的成员
    serverLog(LL_WARNING,"WARNING: You specified a maxmemory value that is less than 1MB (current value is %llu bytes). Are you sure
want?", server.maxmemory);
```

切换语法:

```
sc1 enable devtoolset-7 bash
```

3.2.1 Redis版本特性介绍

Redis我们采用Redis6.0.5最新版本, 它有很多新特性, 我们这里对Redis每个版本的特性介绍一下:

Redis2.6

Redis2.6在2012年正是发布, 经历了17个版本, 到2.6.17版本, 相对于Redis2.4, 主要特性如下:

- 1) 服务端支持Lua脚本。
- 2) 去掉虚拟内存相关功能。
- 3) 放开对客户端连接数的硬编码限制。
- 4) 键的过期时间支持毫秒。
- 5) 从节点支持只读功能。
- 6) 两个新的位图命令: bitcount和bitop。
- 7) 增强了redis-benchmark的功能: 支持定制化的压测, CSV输出等功能。
- 8) 基于浮点数自增命令: incrbyfloat和hincrbyfloat。
- 9) redis-cli可以使用--eval参数实现Lua脚本执行。

- 10) shutdown命令增强。
- 11) 重构了大量的核心代码，所有集群相关的代码都去掉了，cluster功能将会是3.0版本最大的亮点。
- 12) info可以按照section输出，并且添加了一些统计项
- 13) sort命令优化

Redis2.8

Redis2.8在2013年11月22日正式发布，经历了24个版本，到2.8.24版本，相比于Redis2.6，主要特性如下：

- 1) 添加部分主从复制的功能，在一定程度上降低了由于网络问题，造成频繁全量复制生成RDB对系统造成的压力。
- 2) 尝试性的支持IPv6。
- 3) 可以通过config set命令设置maxclients。
- 4) 可以用bind命令绑定多个IP地址。
- 5) Redis设置了明显的进程名，方便使用ps命令查看系统进程。
- 6) config rewrite命令可以将config set持久化到Redis配置文件中。
- 7) 发布订阅添加了pubsub。
- 8) Redis Sentinel第二版，相比于Redis2.6的Redis Sentinel，此版本已经变成生产可用。

Redis3.0 (里程碑)

Redis3.0在2015年4月1日正式发布，相比于Redis2.8主要特性如下：

Redis最大的改动就是添加Redis的分布式实现Redis Cluster。

- 1) Redis Cluster: Redis的官方分布式实现。
- 2) 全新的embedded string对象编码结果，优化小对象内存访问，在特定的工作负载下载速度大幅提升。
- 3) Iru算法大幅提升。
- 4) migrate连接缓存，大幅提升键迁移的速度。
- 5) migrate命令两个新的参数copy和replace。
- 6) 新的client pause命令，在指定时间内停止处理客户端请求。
- 7) bitcount命令性能提升。
- 8) cinfo set设置maxmemory时候可以设置不同的单位（之前只能是字节）。
- 9) Redis日志小做调整：日志中会反应当前实例的角色（master或者slave）。
- 10) incr命令性能提升。

Redis3.2

Redis3.2在2016年5月6日正式发布，相比于Redis3.0主要特征如下：

- 1) 添加GEO相关功能。
- 2) SDS在速度和节省空间上都做了优化。
- 3) 支持用upstart或者systemd管理Redis进程。
- 4) 新的List编码类型：quicklist。
- 5) 从节点读取过期数据保证一致性。
- 6) 添加了hstrlen命令。
- 7) 增强了debug命令，支持了更多的参数。
- 8) Lua脚本功能增强。
- 9) 添加了Lua Debugger。
- 10) config set 支持更多的配置参数。
- 11) 优化了Redis崩溃后的相关报告。

- 12) 新的RDB格式，但是仍然兼容旧的RDB。
- 13) 加速RDB的加载速度。
- 14) spop命令支持个数参数。
- 15) cluster nodes命令得到加速。
- 16) jemalloc更新到4.0.3版本。

Redis4.0

可能出乎很多的意料，Redis3.2之后的版本是4.0，而不是3.4、3.6、3.8。
一般这种重大版本的升级也意味着软件或者工具本身发生了重大改革。下面是Redis4.0的新特性：

- 1) 提供了模块系统，方便第三方开发者拓展Redis的功能。
- 2) PSYNC2.0：优化了之前版本中，主从节点切换必然引起全量复制的问题。
- 3) 提供了新的缓存剔除算法：LFU (Last Frequently Used)，并对已有算法进行了优化。
- 4) 提供了非阻塞del和flushall/flushdb功能，有效解决删除了bigkey可能造成的Redis阻塞。
- 5) 提供了memory命令，实现对内存更为全面的监控统计。
- 6) 提供了交互数据库功能，实现Redis内部数据库的数据置换。
- 7) 提供了RDB-AOF混合持久化格式，充分利用了AOF和RDB各自优势。
- 8) Redis Cluster 兼容NAT和Docker。

Redis5.0

1. 新的Stream数据类型。5.0
2. 新的Redis模块API: Timers and Cluster API。
3. RDB现在存储LFU和LRU信息。
4. 集群管理器从Ruby (redis-trib.rb) 移植到C代码。可以在redis-cli中。查看`redis-cli -cluster help`了解更多信息。
5. 新sorted set命令: ZPOPMIN / MAX和阻塞变量。
6. 主动碎片整理V2。
7. 增强HyperLogLog实现。
8. 更好的内存统计报告。
9. 许多带有子命令的命令现在都有一个HELP子命令。
10. 客户经常连接和断开连接时性能更好。
11. 错误修复和改进。
12. jemalloc升级到5.1版

Redis6.0.5 (里程碑)

1. ACL用户权限控制功能
2. RESP3: 新的 Redis 通信协议
3. Cluster 管理工具
4. SSL 支持
5. IO多线程支持
6. 新的Module API
7. 新的 Expire 算法
8. Redis Cluster Proxy
9. Disque

3.2.2 Redis集群配置

现在安装对应的应用软件一般都推荐使用Docker容器，我们这里也不例外，直接使用Docker容器进行安装，安装大概要分这几个步骤：

- 1、创建redis-cluster.tpl配置Redis信息【端口、是否开启集群等】
- 2、创建redis.sh配置需要创建的redis信息
- 3、添加网络，redis集群使用该网络
- 4、执行redis.sh实现创建redis
- 5、执行redis-cli创建集群

配置Redis信息

创建 redis-cluster.tpl 配置Redis信息 (redis.conf)

```
#端口
port ${PORT}
#非保护模式
protected-mode no
#启用集群模式
cluster-enabled yes
cluster-config-file nodes.conf
#超时时间
cluster-node-timeout 5000
#集群各节点IP地址
cluster-announce-ip 192.168.211.141
#集群节点映射端口
cluster-announce-port ${PORT}
#集群总线端口
cluster-announce-bus-port 1${PORT}
#开启aof持久化策略
appendonly yes
#后台运行
#daemonize yes
#进程号存储
pidfile /var/run/redis_${PORT}.pid
#集群加密
#masterauth itheima
#requirepass itheima
```

Redis创建配置

创建 redis.sh 配置需要创建的 Redis 集群

```
#!/bin/bash
#在/usr/local/server/redis-cluster下生成conf和data目标，并生成配置信息
for port in `seq 7001 7006`;
```

```

do
    mkdir -p ./${port}/conf && PORT=${port} envsubst < ./redis-cluster.tpl >
    ./${port}/conf/redis.conf && mkdir -p ./${port}/data;
done
#创建6个redis容器
for port in `seq 7001 7006`;
do
    docker run -d -it -p ${port}:${port} -p 1${port}:1${port} -v /usr/local/server/redis-
cluster/${port}/conf/redis.conf:/usr/local/etc/redis/redis.conf -v /usr/local/server/redis-
cluster/${port}/data:/data --privileged=true --restart always --name redis-${port} --net
redis-net --sysctl net.core.somaxconn=1024 redis redis-server
/usr/local/etc/redis/redis.conf;
done
#查找ip
for port in `seq 7001 7006`;
do
    echo -n "$(docker inspect --format '{{ (index .NetworkSettings.Networks "redis-
net").IPAddress }}' "redis-${port}")":"${port}" ";
done
#换行
echo -e "\n"
#输入信息
read -p "请把输入要启动的docker容器名称, 默认redis-7001: " DOCKER_NAME
#判断是否为空
if [ ! $DOCKER_NAME ];
then DOCKER_NAME='redis-7001';
fi
#进入容器
docker exec -it redis-7001 /bin/bash

```

这里涉及到了一个网络的定义 `redis-net`，代码如下：

```
docker network create redis-net
```

移除脚本创建

创建 `stop.sh` 脚本，用于停止Redis容器，并且移除对应容器

```

#!/bin/bash
docker stop redis-7001 redis-7002 redis-7003 redis-7004 redis-7005 redis-7006

docker rm redis-7001 redis-7002 redis-7003 redis-7004 redis-7005 redis-7006

rm -rf 7001 7002 7003 7004 7005 7006

```

脚本授权：给 redis.sh 和 stop.sh 添加可执行权限：

```
chmod +x redis.sh
chmod +x stop.sh
```

执行脚本开始安装Redis节点，并进入到 /usr/local/bin 目录下执行 redis-cli 创建集群关联：

```
#执行集群几点创建
./redis.sh
```

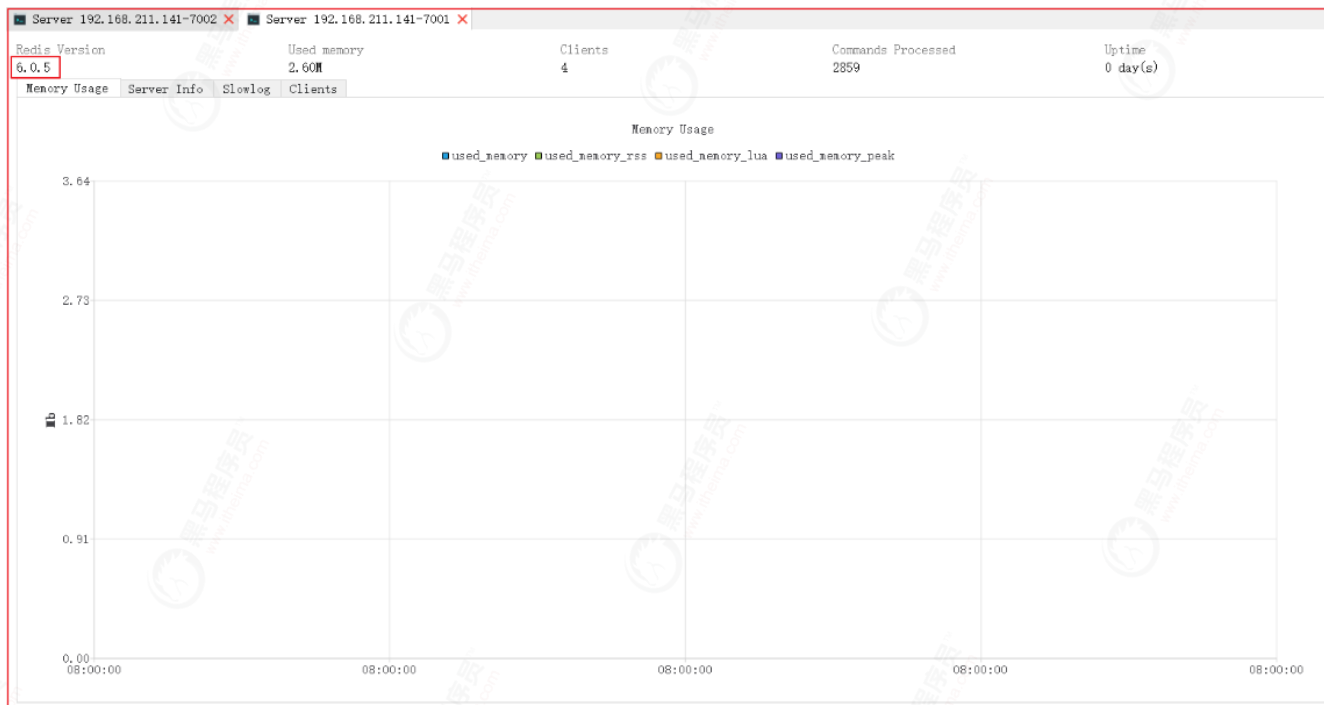
```
#进入到任意一个安装好的redis节点的bin目录，里面有个脚本对象redis-cli，然后执行集群创建
./redis-cli --cluster create 172.18.0.2:7001 172.18.0.3:7002 172.18.0.4:7003 172.18.0.5:7004
172.18.0.6:7005 172.18.0.7:7006 --cluster-replicas 1
```

--cluster-replicas 表示有一个主有几个slave。

```
[root@localhost bin]# ./redis-cli --cluster create 172.18.0.2:7001 172.18.0.3:7002 172.18.0.4:7003 172.18.0.5:7004 172.18.0.6:7005 172.18.0.7:7006 --cluster-replicas 1
>>> Performing hash slots allocation on 6 nodes...
Master[0] -> Slots 0 - 5460
Master[1] -> Slots 5461 - 10922
Master[2] -> Slots 10923 - 16383
Adding replica 172.18.0.6:7005 to 172.18.0.2:7001
Adding replica 172.18.0.7:7006 to 172.18.0.3:7002
Adding replica 172.18.0.5:7004 to 172.18.0.4:7003
M: 80a69bb8af3737bce2913b2952b4456430a89eb3 172.18.0.2:7001
slots:[0-5460] (5461 slots) master
M: c9687b2ebec8b99ee14fcb885b5c3439c58827f 172.18.0.3:7002
slots:[5461-10922] (5462 slots) master
M: 612e4af8eae48426938ce65d12a7d7376b0b37e3 172.18.0.4:7003
slots:[10923-16383] (5461 slots) master
S: 29f3f5c42732084481812d4da36d74b1201723f8 172.18.0.5:7004
replicas 612e4af8eae48426938ce65d12a7d7376b0b37e3
S: c69fc0a0562b07b91f319a98a208108cb9a8e726 172.18.0.6:7005
replicas 80a69bb8af3737bce2913b2952b4456430a89eb3
S: e822b7752a461f50bd0c97e00df9b6a978608bf5 172.18.0.7:7006
replicas c9687b2ebec8b99ee14fcb885b5c3439c58827f
Can I set the above configuration? (type 'yes' to accept): yes
>>> 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 172.18.0.2:7001)
M: 80a69bb8af3737bce2913b2952b4456430a89eb3 172.18.0.2:7001
slots:[0-5460] (5461 slots) master
1 additional replica(s)
S: e822b7752a461f50bd0c97e00df9b6a978608bf5 192.168.211.141:7006
slots: (0 slots) slave
replicas c9687b2ebec8b99ee14fcb885b5c3439c58827f
M: 612e4af8eae48426938ce65d12a7d7376b0b37e3 192.168.211.141:7003
slots:[10923-16383] (5461 slots) master
1 additional replica(s)
S: c69fc0a0562b07b91f319a98a208108cb9a8e726 192.168.211.141:7005
slots: (0 slots) slave
replicas 80a69bb8af3737bce2913b2952b4456430a89eb3
M: c9687b2ebec8b99ee14fcb885b5c3439c58827f 192.168.211.141:7002
slots:[5461-10922] (5462 slots) master
1 additional replica(s)
S: 29f3f5c42732084481812d4da36d74b1201723f8 192.168.211.141:7004
slots: (0 slots) slave
replicas 612e4af8eae48426938ce65d12a7d7376b0b37e3
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
[root@localhost bin]#
```

安装好了后，我们链接Redis效果如下：



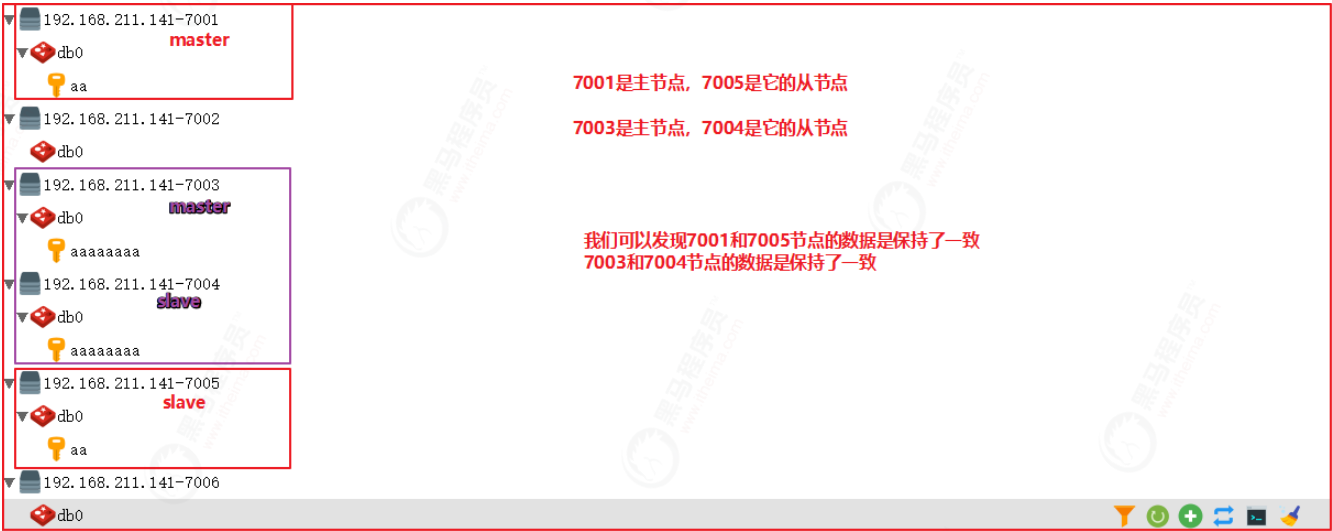
3.2.3 主从复制

我们先来了解下主从同步，如下图：



- 1、Slave服务启动，主动连接Master，并发送SYNC命令，请求初始化同步；
- 2、Master收到SYNC后，执行BGSAVE命令生成RDB文件，并缓存该时间段内的写命令；
- 3、Master完成RDB文件后，将其发送给所有Slave服务器；
- 4、Slave服务器接收到RDB文件后，删除内存中旧的缓存数据，并装载RDB文件；
- 5、Master在发送完RDB后，即刻向所有Slave服务器发送缓存中的写命令；

我们可以用 DesktopManager 链接Redis集群，并测试数据，往redis集群添加2条数据，可以明显看到主从效果，效果如下：



上面是我们集群时随机创建的从节点，如果手动给指定节点添加从节点也是可以实现的，我们这里实现给 7001 添加一个从节点。

3.2.4 集群扩容收容

上面虽然创建了主从复制，但如果手动给节点添加一个从节点，有可能添加从节点，也有可能添加从节点，这是我们想要干的。接下来我们给集群节点添加指定的从节点。

我们安装 7007、7008、7009 几个Redis节点，然后将 7007 和 7008 作为主节点，添加到集群中，7009 作为从节点添加到集群中。

Redis节点	IP	端口
redis-7007	172.18.0.8	7007
redis-7008	172.18.0.9	7008
redis-7009	172.18.0.10	7009

基于Docker安装Redis这里编写了一个脚本，安装脚本 redis-port.sh 如下：

```
#!/bin/bash
#在/usr/local/server/redis-cluster下生成conf和data目标，并生成配置信息
#换行
echo -e "\n"
#输入信息
read -p "请输入容器端口： " DOCKER_PORT

#输入端口赋值
port=$DOCKER_PORT;
```

```
echo -e "$port"
```

#创建配置文件

```
mkdir -p ./${port}/conf && PORT=${port} envsubst < ./redis-cluster.tpl >
./${port}/conf/redis.conf && mkdir -p ./${port}/data;
```

#创建redis容器

```
docker run -d -it -p ${port}:${port} -p 1${port}:1${port} -v /usr/local/server/redis-
cluster/${port}/conf/redis.conf:/usr/local/etc/redis/redis.conf -v /usr/local/server/redis-
cluster/${port}/data:/data --privileged=true --restart always --name redis-${port} --net
redis-net --sysctl net.core.somaxconn=1024 redis redis-server
/usr/local/etc/redis/redis.conf;
```

#查找ip

```
echo -n "启动$(docker inspect --format '{{ (index .NetworkSettings.Networks "redis-
net").IPAddress }}' "redis-${port}")":"${port}" 成功! ";
echo -e "\n"
```

我们执行 `redis-port.sh` 脚本，实现 7007,7008,7009 节点安装。

添加脚本执行权限：

```
chmod +x redis-port.sh
```

查看集群状态

主节点查看：

```
./redis-cli -p 7001 cluster nodes|grep master
```

主节点状态信息如下：

```
[root@localhost bin]# ./redis-cli -p 7001 cluster nodes|grep master
80a69bb8af3737bce2913b2952b4456430a89eb3 192.168.211.141:7001@17001 myself,master - 0 1593930465000 1 connected 0-5460
612e4af8eae48426938ce65d12a7d7376b0b37e3 192.168.211.141:7003@17003 master - 0 1593930466060 3 connected 10923-16383
c9687b2ebec8b99ee14fcb885b5c3439c58827f 192.168.211.141:7002@17002 master - 0 1593930465557 2 connected 5461-10922
[root@localhost bin]#
```

从节点查看：

```
./redis-cli -p 7001 cluster nodes|grep slave
```

```
[root@localhost bin]# ./redis-cli -p 7001 cluster nodes|grep slave
e822b7752a461f50bd0c97e00df9b6a978608bf5 192.168.211.141:7006@17006 slave c9687b2ebec8b99ee14fcb885b5c3439c58827f 0 1593930580557 6 connected
c69fc0a0562b07b91f319a98a208108cb9a8e726 192.168.211.141:7005@17005 slave 80a69bb8af3737bce2913b2952b4456430a89eb3 0 1593930579852 1 connected
29f3f5c42732084481812d4da36d74b1201723f8 192.168.211.141:7004@17004 slave 612e4af8eae48426938ce65d12a7d7376b0b37e3 0 1593930578544 4 connected
[root@localhost bin]#
```

从上面信息我们可以看出集群关系：

Master: 7001	Slave: 7005
Master: 7002	Slave: 7006
Master: 7003	Slave: 7004

3.2.4.1 添加集群主节点

我们需要给集群节点添加一个主节点，我们需要将 192.168.211.141:7007 节点添加到 192.168.211.141:7001 节点所在的集群中，并且添加后作为主节点，添加命令行如下：

```
./redis-cli --cluster add-node 192.168.211.141:7007 192.168.211.141:7001
```

命令说明：

将192.168.211.141:7007节点添加到192.168.211.141:7001节点所在的集群中

执行命令后，效果如下：

```
[root@localhost redis-cluster]# cd redis/bin
[root@localhost bin]# ./redis-cli --cluster add-node 192.168.211.141:7007 192.168.211.141:7001
>>> Adding node 192.168.211.141:7007 to cluster 192.168.211.141:7001
>>> Performing Cluster Check (using node 192.168.211.141:7001)
M: 80a69bb8af3737bce2913b2952b4456430a89eb3 192.168.211.141:7001
  slots:[0-5460] (5461 slots) master
  1 additional replica(s)
S: e822b7752a461f50bd0c97e00df9b6a978608bf5 192.168.211.141:7006
  slots: (0 slots) slave
  replicates c9687b2ebec8b99ee14fcb885b5c3439c58827f
M: 612e4af8eae48426938ce65d12a7d7376b0b37e3 192.168.211.141:7003
  slots:[10923-16383] (5461 slots) master
  1 additional replica(s)
S: c69fc0a0562b07b91f319a98a208108cb9a8e726 192.168.211.141:7005
  slots: (0 slots) slave
  replicates 80a69bb8af3737bce2913b2952b4456430a89eb3
M: c9687b2ebec8b99ee14fcb885b5c3439c58827f 192.168.211.141:7002
  slots:[5461-10922] (5462 slots) master
  1 additional replica(s)
S: 29f3f5c42732084481812d4da36d74b1201723f8 192.168.211.141:7004
  slots: (0 slots) slave
  replicates 612e4af8eae48426938ce65d12a7d7376b0b37e3
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
>>> Send CLUSTER MEET to node 192.168.211.141:7007 to make it join the cluster.
[OK] New node added correctly.
[root@localhost bin]#
```

此时我们可以进入到集群节点之一查看此时的集群状态，我们进入到 7002 节点中输入 `cluster node` 查询，操作方法如下：

进入容器

```
docker exec -it redis-7002 /bin/bash
```

进入到redis-cli脚本目录

```
cd /usr/local/bin
```

登录7002节点

```
./redis-cli -p 7002 -c
```

查询集群状态

```
cluster nodes
```

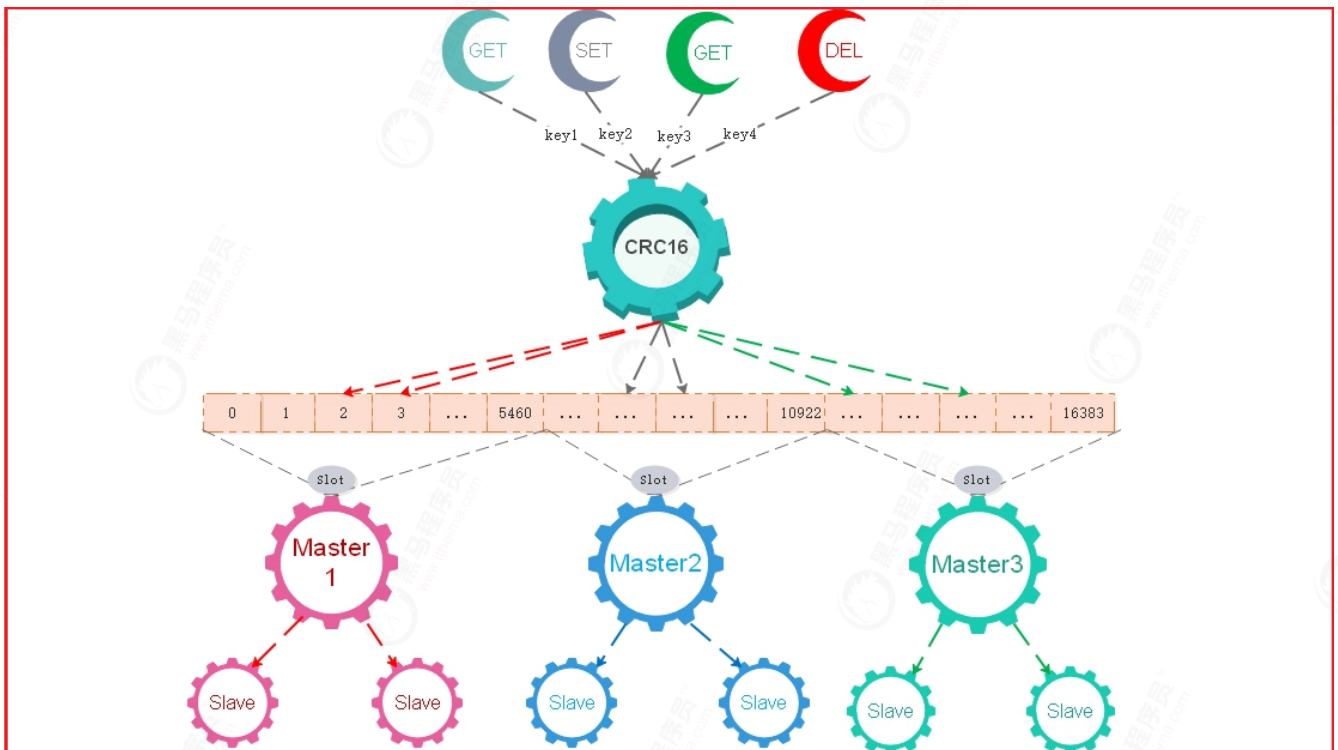
效果如下：

```
[root@localhost bin]# ./redis-cli -p 7002 -c
127.0.0.1:7002> cluster nodes
e822b7752a461f50bd0c97e00df9b6a978608bf5 192.168.211.141:7006@17006 slave c9687b2ebec8b99ee14fcb885b5c3439c58827f 0 1593930787000 6 connected
443096af2ff8c1e89f1160faed4f6a02235822a7 192.168.211.141:7007@17007 master - 0 1593930787000 0 connected
29f3f5c42732084481812d4da36d74b1201723f8 192.168.211.141:7004@17004 slave 612e4af8eae48426938ce65d12a7d7376b0b37e3 0 1593930788628 4 connected
c9687b2ebec8b99ee14fcb885b5c3439c58827f 192.168.211.141:7002@17002 myself,master - 0 1593930786000 2 connected 5461-10922
80a69bb8af3737bce2913b2952b4456430a89eb3 192.168.211.141:7001@17001 master - 0 1593930787621 1 connected 0-5460
c69fc0a0562b07b91f319a98a208108cb9a8e726 192.168.211.141:7005@17005 slave 80a69bb8af3737bce2913b2952b4456430a89eb3 0 1593930787621 5 connected
612e4af8eae48426938ce65d12a7d7376b0b37e3 192.168.211.141:7003@17003 master - 0 1593930788000 3 connected 10923-16383
127.0.0.1:7002>
```

3.2.4.2 哈希槽分配

```
[root@localhost bin]# ./redis-cli -p 7002 -c
127.0.0.1:7002> cluster nodes
e822b7752a461f50bd0c97e00df9b6a978608bf5 192.168.211.141:7006@17006 slave c9687b2ebec8b99ee14fcb885b5c3439c58827f 0 1593930787000 6 connected
443096af2ff8c1e89f1160faed4f6a02235822a7 192.168.211.141:7007@17007 master - 0 1593930787000 0 connected 没有分配哈希槽???????
29f3f5c42732084481812d4da36d74b1201723f8 192.168.211.141:7004@17004 slave 612e4af8eae48426938ce65d12a7d7376b0b37e3 0 1593930788628 4 connected
c9687b2ebec8b99ee14fcb885b5c3439c58827f 192.168.211.141:7002@17002 myself,master - 0 1593930786000 2 connected 5461-10922
80a69bb8af3737bce2913b2952b4456430a89eb3 192.168.211.141:7001@17001 master - 0 1593930787621 1 connected 0-5460
c69fc0a0562b07b91f319a98a208108cb9a8e726 192.168.211.141:7005@17005 slave 80a69bb8af3737bce2913b2952b4456430a89eb3 0 1593930787621 5 connected
612e4af8eae48426938ce65d12a7d7376b0b37e3 192.168.211.141:7003@17003 master - 0 1593930788000 3 connected 10923-16383
127.0.0.1:7002>
```

从上面的集群节点信息我们可以看出一个问题，其他主节点都有一串数字范围，而刚才添加的 7007 节点没有这段数字范围，这和Redis集群原理有关系，我们来讲一下集群的原理，然后实现新增节点哈希槽(这段数字范围)的分配。



(Redis集群原理图)

Redis原理:

- (1) Redis Cluster 特性之一是引入了槽的概念。一个redis集群包含 16384 个哈希槽。
- (2) 集群时，会将16384个哈希槽分别分配给每个Master节点，每个Master节点占16384个哈希槽中的一部分。
- (3) 执行GET/SET/DEL时，都会根据key进行操作，Redis通过CRC16算法对key进行计算得到该key所属Redis节点。
- (4) 根据key去指定Redis节点操作数据。

从原理上分析，因为之前 7001,7002,7003 已经瓜分了16384个哈希槽，所以再增加一个新节点是没有剩余哈希槽分配的，所以新增的 7007 节点没有分配到哈希槽。我们只能重新分配哈希槽，才能让新增节点分配到一定的哈希槽，重新分配哈希槽后，我们还要考虑之前其他Redis节点中的数据迁移。

重新分配Hash槽

我们将 7001,7002,7003 中的 100 个哈希槽挪给 7007，命令如下：

```
./redis-cli --cluster reshard 192.168.211.141:7001 --cluster-from
c9687b2ebec8b99ee14fcbb885b5c3439c58827f,80a69bb8af3737bce2913b2952b4456430a89eb3,612e4af8eae48426938ce65d12a7d7376b0b37e3 --cluster-to 443096af2ff8c1e89f1160faed4f6a02235822a7 --
cluster-slots 100
```

命令说明：

将节点

```
c9687b2ebec8b99ee14fcbb885b5c3439c58827f
80a69bb8af3737bce2913b2952b4456430a89eb3
612e4af8eae48426938ce65d12a7d7376b0b37e3
```

中的100个哈希槽移动到

```
443096af2ff8c1e89f1160faed4f6a02235822a7
```

中

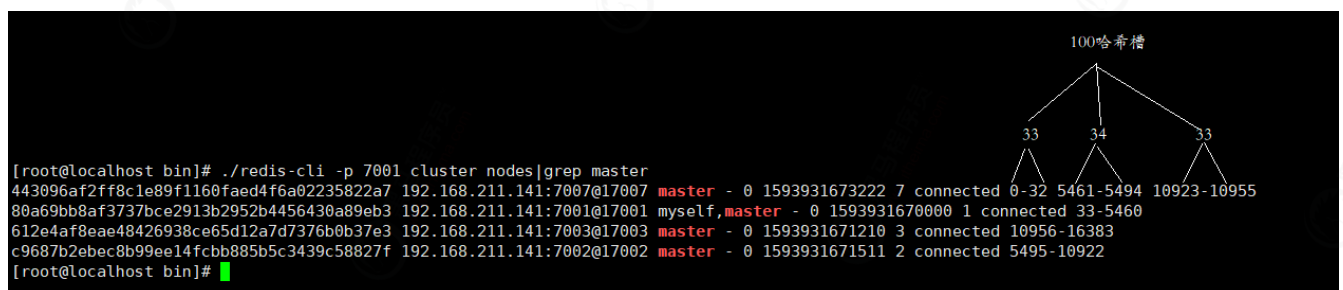
参数说明：

```
--cluster-from: 表示slot目前所在的节点的node ID，多个ID用逗号分隔
--cluster-to: 表示需要新分配节点的node ID
--cluster-slots: 分配的slot数量
```

将100个哈希槽挪给7007后，我们查询下节点信息：

```
./redis-cli -p 7001 cluster nodes|grep master
```

效果如下：



3.2.4.3 添加集群从节点

我们需要往集群中给 7007 节点添加一个从节点 7008，添加从节点的主要目的是提高高可用，防止主节点宕机后该节点无法提供服务。添加从节点命令如下：

```
./redis-cli --cluster add-node 192.168.211.141:7008 192.168.211.141:7007 --cluster-slave --cluster-master-id 443096af2ff8c1e89f1160faed4f6a02235822a7
```

命令说明：

将192.168.211.141:7008节点添加到192.168.211.141:7007对应的集群中，并且加入的节点为从节点，对应的主节点id是443096af2ff8c1e89f1160faed4f6a02235822a7

参数说明：

add-node：后面的分别跟着新加入的slave和slave对应的master
cluster-slave：表示加入的是slave节点
--cluster-master-id：表示slave对应的master的node ID

执行命令后，效果如下：

```
[root@localhost bin]# ./redis-cli --cluster add-node 192.168.211.141:7008 192.168.211.141:7007 --cluster-slave --cluster-master-id 443096af2ff8c1e89f1160faed4f6a02235822a7
>>> Adding node 192.168.211.141:7008 to cluster 192.168.211.141:7007
>>> Performing Cluster Check (using node 192.168.211.141:7007)
M: 443096af2ff8c1e89f1160faed4f6a02235822a7 192.168.211.141:7007
  slots:[0-32],[5461-5494],[10923-10955] (100 slots) master
  S: e822b7752a461f50bd0c97e00df9b6a978608bf5 192.168.211.141:7006
    slots: (0 slots) slave
    replicates c9687b2ebec8b99ee14fcb885b5c3439c58827f
M: 612e4af8eae48426938ce65d12a7d7376b0b37e3 192.168.211.141:7003
  slots:[10956-16383] (5428 slots) master
  1 additional replica(s)
S: 29f3f5c42732084481812d4da36d74b1201723f8 192.168.211.141:7004
  slots: (0 slots) slave
  replicates 612e4af8eae48426938ce65d12a7d7376b0b37e3
S: c69fc0a0562b07b91f319a98a208108cb9a8e726 192.168.211.141:7005
  slots: (0 slots) slave
  replicates 80a69bb8af3737bce2913b2952b4456430a89eb3
M: c9687b2ebec8b99ee14fcb885b5c3439c58827f 192.168.211.141:7002
  slots:[5495-10922] (5428 slots) master
  1 additional replica(s)
M: 80a69bb8af3737bce2913b2952b4456430a89eb3 192.168.211.141:7001
  slots:[33-5460] (5428 slots) master
  1 additional replica(s)
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
>>> Send CLUSTER MEET to node 192.168.211.141:7008 to make it join the cluster.
Waiting for the cluster to join

>>> Configure node as replica of 192.168.211.141:7007.
[OK] New node added correctly.
[root@localhost bin]#
```

集群信息查看：

```
[root@localhost bin]# ./redis-cli -p 7001 -c
127.0.0.1:7001> cluster nodes
98be71d8d6eff6bd40947fa0441e5a821dce20ae 192.168.211.141:7008@17008 slave 443096af2ff8c1e89f1160faed4f6a02235822a7 0 1593933008562 7 connected
443096af2ff8c1e89f1160faed4f6a02235822a7 192.168.211.141:7007@17007 master - 0 1593933008000 7 connected 0-32 5461-5494 10923-10955
80a69bb8af3737bce2913b2952b4456430a89eb3 192.168.211.141:7001@17001 myself,master - 0 1593933006000 1 connected 33-5460
e822b7752a461f50bd0c97e00df9b6a978608bf5 192.168.211.141:7006@17006 slave c9687b2ebec8b99ee14fcb885b5c3439c58827f 0 1593933007254 6 connected
612e4af8eae48426938ce65d12a7d7376b0b37e3 192.168.211.141:7003@17003 master - 0 1593933008260 3 connected 10956-16383
c69fc0a0562b07b91f319a98a208108cb9a8e726 192.168.211.141:7005@17005 slave 80a69bb8af3737bce2913b2952b4456430a89eb3 0 1593933007556 1 connected
c9687b2ebec8b99ee14fcb885b5c3439c58827f 192.168.211.141:7002@17002 master - 0 1593933008260 2 connected 5495-10922
29f3f5c42732084481812d4da36d74b1201723f8 192.168.211.141:7004@17004 slave 612e4af8eae48426938ce65d12a7d7376b0b37e3 0 1593933007556 4 connected
127.0.0.1:7001>
```

3.2.4.4 缩容

1)数据迁移

2)哈希槽迁移

3)从节点删除

4)主节点删除

在真实生产环境中，我们也会跟着我们的业务和环境执行缩容处理，比如双十一过后，流量没有那么大了，我们往往会缩容处理，服务器开销。

Redis实现缩容，需要哈希槽重新分配，将需要移除的节点所分配的所有哈希槽值分配给其他需要运行工作的节点，还需要移除该节点的从节点，然后再删除该节点。

移除从节点

移除 7007 的从节点 7008，命令如下：

```
./redis-cli --cluster del-node 192.168.211.141:7008 98be71d8d6eff6bd40947fa0441e5a821dce20ae
```

参数说明：

del-node:删除节点，后面跟着slave节点的 ip:port 和node ID

删除后，我们再来查看集群节点，此时再无7008节点。

```
[root@localhost bin]# ./redis-cli -p 7001 -c
127.0.0.1:7001> cluster nodes
443096af2ff8c1e89f1160faed4f6a02235822a7 192.168.211.141:7007@17007 master - 0 1593934756512 7 connected 0-32 5461-5494 10923-10955
80a69bb8af3737bce2913b2952b4456430a89eb3 192.168.211.141:7001@17001 myself,master - 0 1593934757000 1 connected 33-5460
e822b7752a461f50bd0c97e00df9b6a978608bf5 192.168.211.141:7006@17006 slave c9687b2ebec8b99ee14fcb885b5c3439c58827f 0 1593934757000 6 connected
612e4af8eae48426938ce65d12a7d7376b0b37e3 192.168.211.141:7003@17003 master - 0 1593934757822 3 connected 10956-16383
c69fc0a0562b07b91f319a98a208108cb9a8e726 192.168.211.141:7005@17005 slave 80a69bb8af3737bce2913b2952b4456430a89eb3 0 1593934756511 1 connected
c96f7b2ebec8b99ee14fcb885b5c3439c58827f 192.168.211.141:7002@17002 master - 0 1593934757520 2 connected 5495-10922
29f3f5c42732084481812d4da36d74b1201723f8 192.168.211.141:7004@17004 slave 612e4af8eae48426938ce65d12a7d7376b0b37e3 0 1593934757000 4 connected
127.0.0.1:7001>
```

迁移Master的Slot

我们需要将 7007 节点的哈希槽迁移到 7001,7002,7003 节点上，仍然用上面用过的 redis-cli --cluster reshard 语法，命令如下：

第1次迁移：

```
./redis-cli --cluster reshard 192.168.211.141:7007 --cluster-from
443096af2ff8c1e89f1160faed4f6a02235822a7 --cluster-to
80a69bb8af3737bce2913b2952b4456430a89eb3 --cluster-slots 33 --cluster-yes
```

命令说明：

将192.168.211.141:7007节点所在集群中443096af2ff8c1e89f1160faed4f6a02235822a7节点的33个哈希槽迁移给80a69bb8af3737bce2913b2952b4456430a89eb3节点，不回显需要迁移的slot，直接迁移。

效果如下：

```
[root@localhost bin]# ./redis-cli --cluster reshard 192.168.211.141:7007 --cluster-from 443096af2ff8c1e89f1160faed4f6a02235822a7 --cluster-to 80a69bb8af3737bce2913b2952b4456430a89eb3 --cluster-slots 33 --cluster-yes
>>> Performing Cluster Check (using node 192.168.211.141:7007)
M: 443096af2ff8c1e89f1160faed4f6a02235822a7 192.168.211.141:7007
slots:[0-32],[5461-5494],[10923-10955] (100 slots) master
S: e822b7752a461f50bd0c97e00df9b6a978608bf5 192.168.211.141:7006
slots: (0 slots) slave
replicates c9687b2ebec8b99ee14fcb885b5c3439c58827f
M: 612e4af8eae48426938ce65d12a7d7376b0b37e3 192.168.211.141:7003
slots:[10956-16383] (5428 slots) master
1 additional replica(s)
S: 29f3f5c42732084481812d4da36d74b1201723f8 192.168.211.141:7004
slots: (0 slots) slave
replicates 612e4af8eae48426938ce65d12a7d7376b0b37e3
S: c69fc0a0562b07b91f319a98a208108cb9a8e726 192.168.211.141:7005
slots: (0 slots) slave
replicates 80a69bb8af3737bce2913b2952b4456430a89eb3
M: c9687b2ebec8b99ee14fcb885b5c3439c58827f 192.168.211.141:7002
slots:[5495-10922] (5428 slots) master
1 additional replica(s)
M: 80a69bb8af3737bce2913b2952b4456430a89eb3 192.168.211.141:7001
slots:[33-5460] (5428 slots) master
1 additional replica(s)
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.

Ready to move 33 slots.
Source nodes:
M: 443096af2ff8c1e89f1160faed4f6a02235822a7 192.168.211.141:7007
slots:[0-32],[5461-5494],[10923-10955] (100 slots) master
Destination node:
M: 80a69bb8af3737bce2913b2952b4456430a89eb3 192.168.211.141:7001
slots:[33-5460] (5428 slots) master
1 additional replica(s)
```

查看集群节点：

```
[root@localhost bin]# ./redis-cli -p 7001 -c
127.0.0.1:7001> cluster nodes
443096af2ff8c1e89f1160faed4f6a02235822a7 192.168.211.141:7007@17007 master - 0 1593935229155 7 connected 5461-5494 10923-10955
80a69bb8af3737bce2913b2952b4456430a89eb3 192.168.211.141:7001@17001 myself,master - 0 1593935225000 8 connected 0-5460 0-52的哈希槽已迁移到7001
e822b7752a461f50bd0c97e00df9b6a978608bf5 192.168.211.141:7006@17006 slave c9687b2ebec8b99ee14fcb885b5c3439c58827f 0 1593935228000 6 connected
612e4af8eae48426938ce65d12a7d7376b0b37e3 192.168.211.141:7003@17003 master - 0 1593935227140 3 connected 10956-16383
c69fc0a0562b07b91f319a98a208108cb9a8e726 192.168.211.141:7005@17005 slave 80a69bb8af3737bce2913b2952b4456430a89eb3 0 1593935228147 8 connected
c9687b2ebec8b99ee14fcb885b5c3439c58827f 192.168.211.141:7002@17002 master - 0 1593935227542 2 connected 5495-10922
29f3f5c42732084481812d4da36d74b1201723f8 192.168.211.141:7004@17004 slave 612e4af8eae48426938ce65d12a7d7376b0b37e3 0 1593935227542 4 connected
127.0.0.1:7001>
```

我们再次迁移其他哈希槽到其他节点，将剩余的哈希槽迁移到7002和7003去。

第2次迁移：

```
./redis-cli --cluster reshard 192.168.211.141:7007 --cluster-from
443096af2ff8c1e89f1160faed4f6a02235822a7 --cluster-to
c9687b2ebec8b99ee14fcb885b5c3439c58827f --cluster-slots 34 --cluster-yes
```

第3次迁移：

```
./redis-cli --cluster reshard 192.168.211.141:7007 --cluster-from
443096af2ff8c1e89f1160faed4f6a02235822a7 --cluster-to
612e4af8eae48426938ce65d12a7d7376b0b37e3 --cluster-slots 33 --cluster-yes
```

集群状态查询：

```
[root@localhost bin]# ./redis-cli -p 7001 -c
127.0.0.1:7001> cluster nodes
443096af2ff8c1e89f1160faed4f6a02235822a7 192.168.211.141:7007@17007 master - 0 1593935612000 7 connected 再次迁移槽
80a69bb8af3737bce2913b2952b4456430a89eb3 192.168.211.141:7001@17001 myself,master - 0 1593935611000 8 connected 0-5460
e822b7752a461f50bd0c97e00df9b6a978608bf5 192.168.211.141:7006@17006 slave c9687b2ebec8b99ee14fcb885b5c3439c58827f 0 1593935611000 9 connected
612e4af8eae48426938ce65d12a7d7376b0b37e3 192.168.211.141:7003@17003 master - 0 1593935612581 10 connected 10923-16383
c69fc0a0562b07b91f319a98a208108cb9a8e726 192.168.211.141:7005@17005 slave 80a69bb8af3737bce2913b2952b4456430a89eb3 0 1593935611000 8 connected
c9687b2ebec8b99ee14fcb885b5c3439c58827f 192.168.211.141:7002@17002 master - 0 1593935612581 9 connected 5461-10922
29f3f5c42732084481812d4da36d74b1201723f8 192.168.211.141:7004@17004 slave 612e4af8eae48426938ce65d12a7d7376b0b37e3 0 1593935611000 10 connected
127.0.0.1:7001>
```

删除节点命令如下:

```
./redis-cli --cluster del-node 192.168.211.141:7007 443096af2ff8c1e89f1160faed4f6a02235822a7
```

效果如下：

```
[root@localhost bin]# ./redis-cli --cluster del-node 192.168.211.141:7007 443096af2ff8c1e89f1160faed4f6a02235822a7
>>> Removing node 443096af2ff8c1e89f1160faed4f6a02235822a7 from cluster 192.168.211.141:7007
>>> Sending CLUSTER FORGET messages to the cluster...
>>> Sending CLUSTER RESET SOFT to the deleted node.
[root@localhost bin]#
```

集群节点查看:

```

root@localhost bin# ./redis-cli -p 7001 -c
127.0.0.1:7001> cluster nodes
08a69bb8af3737bce2913b2952b4456430a89eb3 192.168.211.141:7001@17001 myself,master - 0 1593935795000 8 connected 0-5460
e822b7752a461f5b0dc9f7e00df9a769f8680b5 192.168.211.141:7006@17006 slave c9687b2ebec8b99e541fcb88b55c3439c58827f 0 1593935797000 9 connected
612a748faee4a26938ce65d12a7d7376bb37e3 192.168.211.141:7003@17003 master - 0 1593935797634 10 connected 10923-16383
c96f0a0562b07b91f319a9a208108cb9a8e726 192.168.211.141:7005@17005 slave 08a69bb8af3737bce2913b2952b4456430a89eb3 0 1593935796529 8 connected
c9687b2ebec8b99e541fcb88b55c3439c58827f 192.168.211.141:7002@17002 master - 0 1593935797533 9 connected 5461-10922
29f3f5c42732084481812d4da36d74b1201723f8 192.168.211.141:7004@17004 slave 612e4af8ae48426938ce65d12a7d7376bb37e3 0 1593935796529 10 connected
127.0.0.1:7001>

```

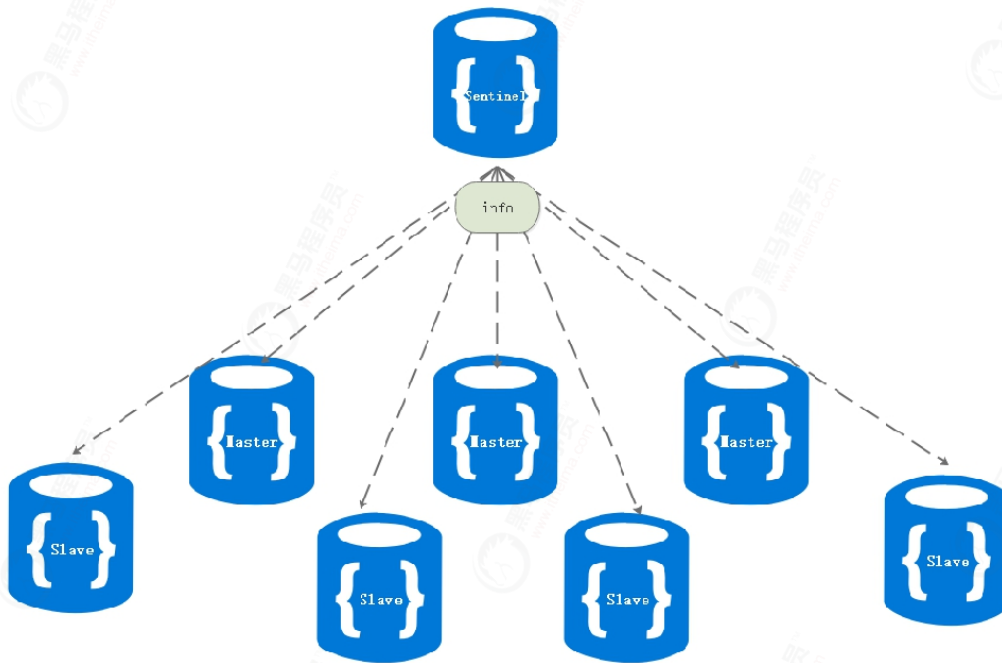
哨兵是用来解决redis高可用性的，可以监控集群中主从的变化，然后进行故障转移。

一套合理的监控机制是Sentinel节点判定节点不可达的重要保证，Redis Sentinel通过三个定时监控任务完成对各个节点发现和监控。

周期10秒监控

每隔10秒，每个Sentinel节点会向主节点和从节点发送info命令获取最新的主从结构，该命令有3个作用：

- 1、通过向主节点执行info命令，获取从节点的信息，这也是为什么 Sentinel节点不需要显式配置监控从节点
- 2、当有新的从节点加入时都可以立刻感知出来
- 3、节点不可达或者故障转移后，可以通过info命令实时更新节点结构信息

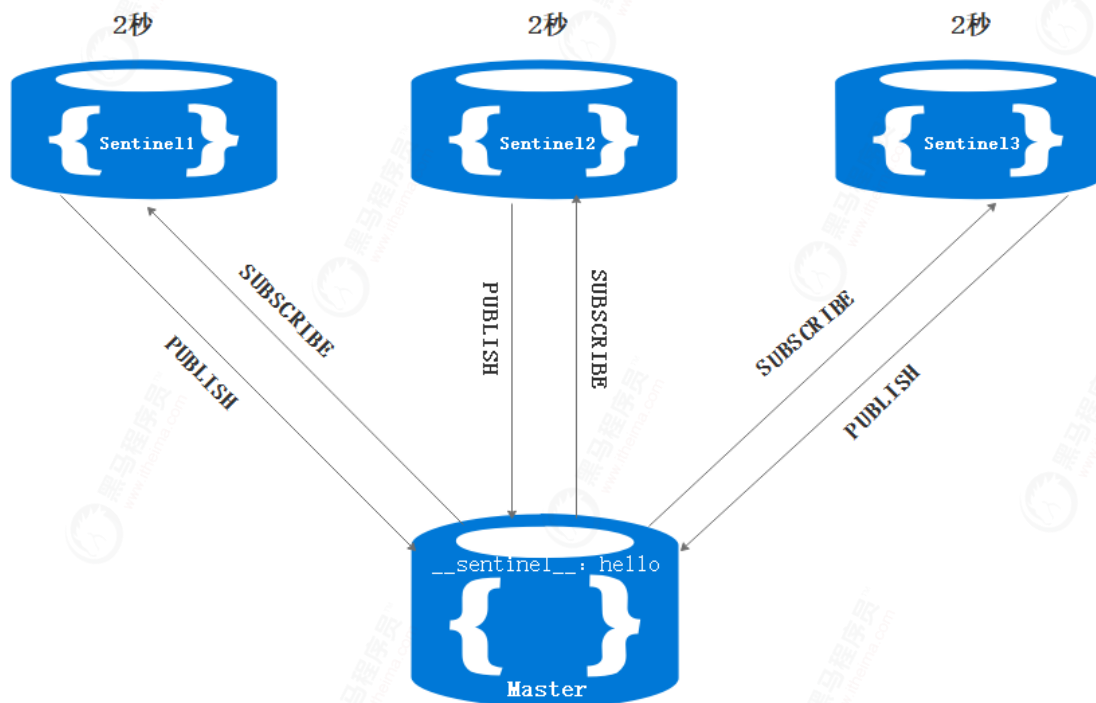


周期2秒监控

每隔2秒，每个Sentinel节点会向Redis数据节点的 `__sentinel__:hello` 频道上发送该Sentinel节点对于主节点的判断以及当前Sentinel节点的信息，同时每个Sentinel节点也会订阅该频道，来了解其他Sentinel节点以及它们对主节点的判断。

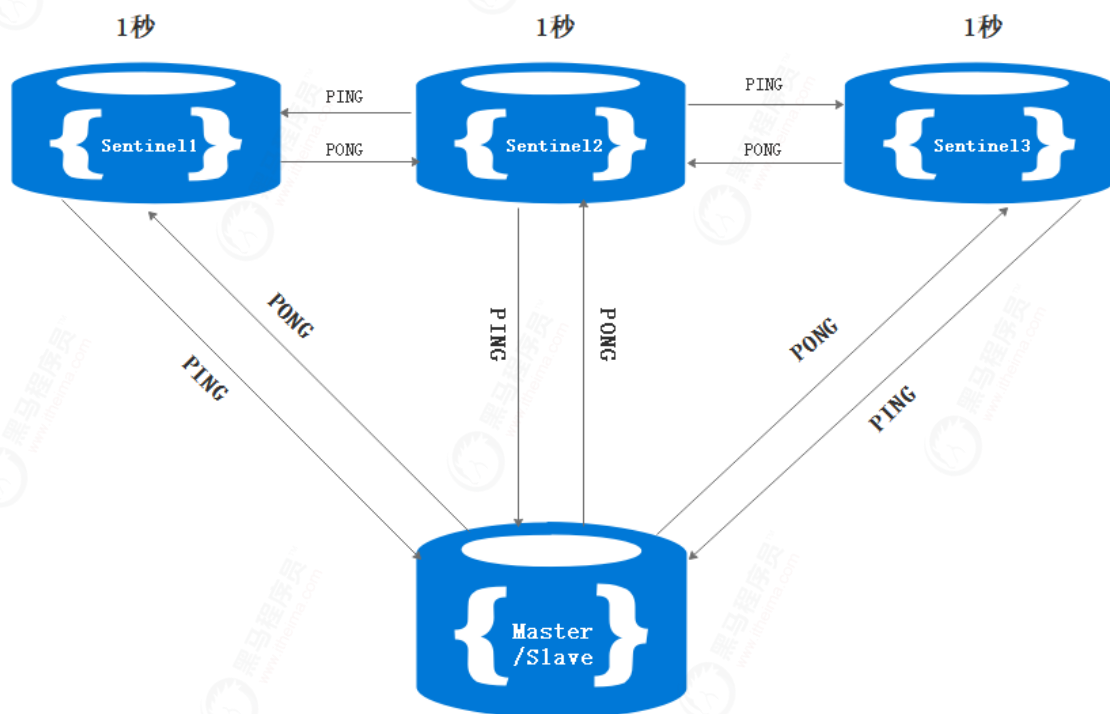
该定时任务主要有2个作用：

- 1、发现新的Sentinel节点：通过订阅主节点的 `__sentinel__:hello` 了解其他的Sentinel节点信息，如果是新加入的Sentinel节点，将该Sentinel节点信息保存起来，并与该Sentinel节点创建连接。
- 2、Sentinel节点之间交换主节点的状态，作为后面客观下线以及领导者选举的依据。



周期1秒监控

每隔1秒，每个Sentinel节点会向主节点、从节点、其余Sentinel节点发送一条ping命令做一次心跳检测，来确认这些节点当前是否可达，从而实现检查每个节点的健康状态。



Redis-Sentinel是用于管理Redis集群,主要执行如下任务:

1、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;

2、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;

3、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;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。

执行流程

哨兵来实现Redis高可用有这5个流程:

- 1、判断主节点是否是主观下线sdown。(主观是指当前节点判断的结果, sdown表示该节点判断该Redis宕机了)
- 2、一旦主节点的主观下线到了一定数量,哨兵群进行客观下线odown的判断。(客观指也有其他Sentinel节点也判断该Redis宕机了,并且数量达到了设置的值)
- 3、哨兵群选举一个Leader,准备对客观下线的节点进行故障转移。
- 4、Leader选择一个slave a,其他哨兵确认。
- 5、其他slave成为a的slave。

状态说明

节点状态分为: ok、主观下线、客观下线。

正常(ok): 就是节点在线,能够正常响应哨兵的检测和命令。

主观下线(sdown): 指单个哨兵,发现主节点在down-after-milliseconds时间内无正确响应,做出的状态判断。

客观下线(odown): 指多个哨兵对一个主节点做了sdown判断后,互相使用is-master-down-by-addr命令交流后,做出的节点已经下线的判断。

3.3.2 Sentinel搭建

我们这里搭建3个哨兵,采用Docker的方式搭建。

节点	IP	端口	监听节点
Sentinel1	192.168.211.141	8001	7001
Sentinel2	192.168.211.141	8002	7002
Sentinel3	192.168.211.141	8003	7003

拷贝 `/usr/local/server/redis-cluster` 下的单节点 `redis` 到 `/usr/local/server/redis-cluster/sentinel/` 目录下, 分别拷贝3份, 命名为 `sentinel1`, `sentinel2`, `sentinel3`, 并且修改每个目录下的配置文件 `sentinel.conf`, 修改 `port`、监听节点信息、后台运行, 配置如下:

第1台的 `sentinel.conf`:

```
port 8001
sentinel monitor mymaster 192.168.211.141 7001 2
daemonize yes
```

第2台的 `sentinel.conf`:

```
port 8002
sentinel monitor mymaster 192.168.211.141 7002 2
daemonize yes
```

第3台的 `sentinel.conf`:

```
port 8003
sentinel monitor mymaster 192.168.211.141 7003 2
daemonize yes
```

参数说明:

```
sentinel monitor mymaster 192.168.211.141 7003 2
```

表示执行Sentinel监控, `mymaster`给当前监控取个名字, 可以随意取, `192.168.211.141 7003`, 表示监听该主节点, `2` 表示有2个sentinel主观 (sdown) 认为该节点宕机了, 则该节点就编程下线状态 (odown)

启动: (到每台sentinel的bin目录下执行启动)

```
./redis-sentinel ./sentinel.conf
```

3.3.3 Sentinel集群讲解

启动Sentinel后, 会有很多日志信息:

```

05:39:26.099 * +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 172.18.0.1 port 8003
05:39:26.634 * +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 192.168.211.141 port
05:39:26.644 * +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 172.18.0.1 port 8002
05:39:28.097 * +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 192.168.211.141 port
05:39:28.120 * +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 172.18.0.1 port 8003
05:39:28.122 * +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 192.168.211.141 port
05:39:28.657 * +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 192.168.211.141 port
05:39:28.658 * +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 172.18.0.1 port 8002
05:39:28.659 * +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 192.168.211.141 port
05:39:29.760 * +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 172.18.0.1 port 8003
05:39:30.151 * +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 192.168.211.141 port
05:39:30.162 * +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 172.18.0.1 port 8003
05:39:30.164 * +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 192.168.211.141 port
05:39:30.694 * +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 172.18.0.1 port 8002
05:39:30.697 * +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 192.168.211.141 port
05:39:32.201 * +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 172.18.0.1 port 8003
05:39:32.211 * +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 192.168.211.141 port
05:39:32.222 * +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 172.18.0.1 port 8003
05:39:32.225 * +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 192.168.211.141 port
05:39:32.753 * +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 172.18.0.1 port 8002

```

我们可以测试下故障自动转移，停掉 redis-7001 节点，可以看到如下日志：

```

* +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 172.18.0.1 port 8002 for
* +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 172.18.0.1 port 8003 for
* +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 192.168.211.141 port 8003
* +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 192.168.211.141 port 8002
* +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 172.18.0.1 port 8002 for
* +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 172.18.0.1 port 8003 for
* +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 192.168.211.141 port 8003
* +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 172.18.0.1 port 8003 for
* +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 192.168.211.141 port 8002
* +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 172.18.0.1 port 8002 for
# +sdown master mymaster 192.168.211.141 7001
* +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 192.168.211.141 port 8003
* +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 172.18.0.1 port 8003 for
* +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 192.168.211.141 port 8002
* +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 172.18.0.1 port 8002 for
* +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 192.168.211.141 port 8003
* +sentinel-address-switch master mymaster 192.168.211.141 7001 ip 192.168.211.141 port 8002

```

此时哨兵这边会出现如下日志，表示7001节点客观上宕机次数-1，并且将该节点添加为7005的从节点：

```

24187:X 06 Jul 2020 03:18:42.100 # -sdown slave 192.168.211.141:7001 192.168.211.141 7001 @ redis-7001 172.18.0.1 7005
24187:X 06 Jul 2020 03:18:51.842 * +slave slave 172.18.0.1:7001 172.18.0.1 7001 @ redis-7001 172.18.0.1 7005

```

日志信息表：

```

+reset-master <instance details> : 主服务器已被重置。
+slave <instance details> : 一个新的从服务器已经被 Sentinel 识别并关联。
+failover-state-reconf-slaves <instance details> : 故障转移状态切换到了 reconf-slaves 状态。
+failover-detected <instance details> : 另一个 Sentinel 开始了一次故障转移操作，或者一个从服务器转换成了主服务器。
+slave-reconf-sent <instance details> : 领头 (leader) 的 Sentinel 向实例发送了 SLAVEOF 命令，为实例设置新的主服务器。
+slave-reconf-inprog <instance details> : 实例正在将自己设置为指定主服务器的从服务器，但相应的同步过程仍未完成。
+slave-reconf-done <instance details> : 从服务器已经成功完成对新主服务器的同步。
-dup-sentinel <instance details> : 对给定主服务器进行监视的一个或多个 Sentinel 已经因为重复出现而被移除——当 Sentinel 实例重启的时候，就会出现这种情况。
+sentinel <instance details> : 一个监视给定主服务器的新 Sentinel 已经被识别并添加。

```

+sdown <instance details> : 给定的实例现在处于主观下线状态。
 -sdown <instance details> : 给定的实例已经不再处于主观下线状态。
 +odown <instance details> : 给定的实例现在处于客观下线状态。
 -odown <instance details> : 给定的实例已经不再处于客观下线状态。
 +new-epoch <instance details> : 当前的纪元 (epoch) 已经被更新。
 +try-failover <instance details> : 一个新的故障迁移操作正在执行中, 等待被大多数 Sentinel 选中 (waiting to be elected by the majority) 。
 +elected-leader <instance details> : 赢得指定纪元的选举, 可以进行故障迁移操作了。
 +failover-state-select-slave <instance details> : 故障转移操作现在处于 select-slave 状态 -- Sentinel 正在寻找可以升级为主服务器的从服务器。
 no-good-slave <instance details> : Sentinel 操作未能找到适合进行升级的从服务器。Sentinel 会在一段时间之后再次尝试寻找合适的从服务器来进行升级, 又或者直接放弃执行故障转移操作。
 selected-slave <instance details> : Sentinel 顺利找到适合进行升级的从服务器。
 failover-state-send-slaveof-noone <instance details> : Sentinel 正在将指定的从服务器升级为主服务器, 等待升级功能完成。
 failover-end-for-timeout <instance details> : 故障转移因为超时而中止, 不过最终所有从服务器都会开始复制新的主服务器 (slaves will eventually be configured to replicate with the new master anyway) 。
 failover-end <instance details> : 故障转移操作顺利完成。所有从服务器都开始复制新的主服务器了。
 +switch-master <master name> <oldip> <oldport> <newip> <newport> : 配置变更, 主服务器的 IP 和地址已经改变。这是绝大多数外部用户都关心的信息。
 +tilt : 进入 tilt 模式。
 -tilt : 退出 tilt 模式。

项目中链接Redis集群或者Redis哨兵, 只需要配置一下 application.yml即可:

```

spring:
  #Redis
  redis:
    #timeout: 50000
    sentinel:
      nodes: 192.168.211.141:8001,192.168.211.141:8002,192.168.211.141:8003
    #cluster:
    # nodes:
    192.168.211.141:7001,192.168.211.141:7002,192.168.211.141:7003,192.168.211.141:7004,192.168.
    211.141:7005,192.168.211.141:7006
  
```

4 Nginx缓存

为了提升网站的整体性能, 我们一般会采用缓存, 从宏观层面来说, 会采用浏览器缓存和后端缓存, Nginx处于Web网站的服务最外层, 而且支持浏览器缓存配置和后端数据缓存, 用它来做部分数据缓存, 效率更高。

Web缓存是可以自动保存常见文档副本的HTTP 设备。当Web请求抵达缓存时, 如果本地有“已缓存的”副本, 就可以从本地设备而不是服务器中提取这个文档。

4.1 OpenResty安装

OpenResty® 是一个基于 Nginx 与 Lua 的高性能 web 平台，其内部集成了大量精良的 Lua 库、第三方模块以及大多数的依赖项。用于方便地搭建能够处理超高并发、扩展性极高的动态 web 应用、web 服务和动态网关。

OpenResty 通过lua脚本扩展 nginx 功能，可提供负载均衡、请求路由、安全认证、服务鉴权、流量控制与日志监控等服务。

OpenResty® 通过汇聚各种设计精良的 Nginx 模块（主要由 OpenResty 团队自主开发），从而将 Nginx 有效地变成一个强大的通用 web 应用平台。这样，web 开发人员和系统工程师可以使用 Lua 脚本语言调动 Nginx 支持的各种 C 以及 Lua 模块，快速构造出足以胜任 10k 乃至 1000k 以上单机并发连接的高性能 web 应用系统。

关于 OpenResty 的学习，大家可以参考：<http://openresty.org/cn/>

安装依赖库

```
yum install wget libtermcap-devel ncurses-devel libevent-devel readline-devel pcre-devel gcc  
openssl openssl-devel perl
```

下载安装包

```
wget https://openresty.org/download/openresty-1.11.2.5.tar.gz
```

解压安装

```
tar -xf openresty-1.11.2.5.tar.gz  
  
cd openresty-1.11.2.5  
  
./configure --prefix=/usr/local/openresty --with-luajit --without-http_redis2_module --with-  
http_stub_status_module --with-http_v2_module --with-http_gzip_static_module --with-  
http_sub_module --add-module=/usr/local/server/nginx_cache_purge-2.3/  
  
make && make install
```

安装完成后，在 `/usr/local/openresty/nginx` 目录下是安装好的nginx。

4.2 浏览器缓存

客户端侧缓存一般指的是浏览器缓存、app缓存等等，目的就是加速各种静态资源的访问，降低服务器压力。

我们通过配置Nginx设置网页缓存信息，从而降低用户对服务器频繁访问造成的巨大压力。我们先配置一个案例，再基于案例去讲解Nginx缓存。

4.2.1 Nginx Web缓存配置

nginx 提供了 expires、etag、if-modified-since 指令来进行浏览器缓存控制。我们使用 expires 来配置 Nginx对网页的缓存。

```
语法: expires [modified] time;  
默认值: expires off;  
上下文: http, server, location, if in location
```

1)上传html

将1.html上传到服务器的 /usr/local/server/html 目录下。

2)配置nginx

修改 /usr/local/openresty/nginx/conf/nginx.conf 文件, 配置如下:

```
server {  
    listen      80;  
    server_name localhost;  
  
    location / {  
        #静态文件路径  
        root    /usr/local/server/html;  
        #缓存10秒  
        expires 10s;  
    }  
}
```

过期时间配置说明

```
expires 30s;    #30秒  
expires 30m;    #30分钟  
expires 2h;     #2个小时  
expires 30d;    #30天
```

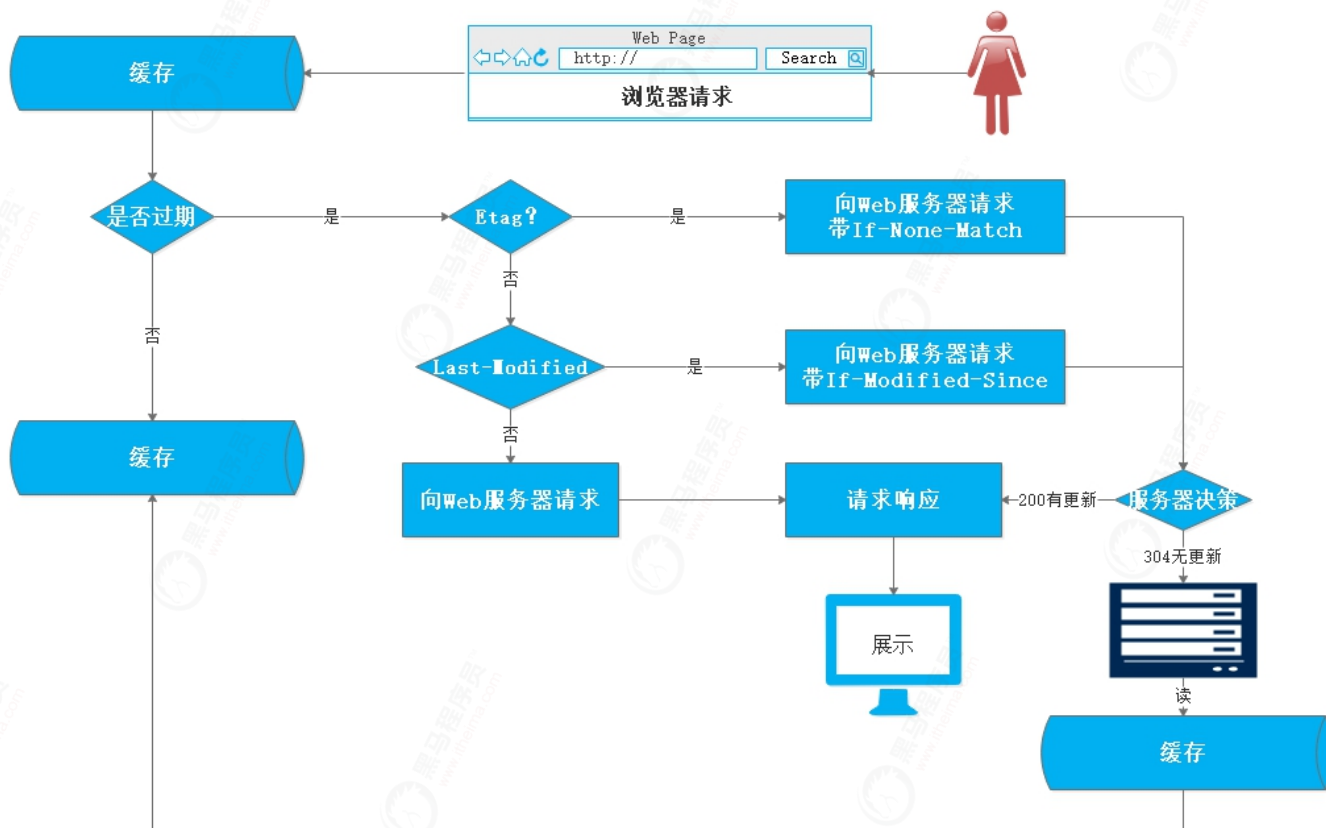
第一次请求 <http://192.168.211.141/1.html>



第二次请求 <http://192.168.211.141/1.html>



4.2.2 Http缓存控制头



参数说明：

HTTP 中最基本的缓存机制，涉及到的 HTTP 头字段，包括 Cache-Control, Last-Modified, If-Modified-Since, Etag, If-None-Match 等。

Last-Modified/If-Modified-Since

Etag是服务端的一个资源的标识，在 HTTP 响应头中将其传送到客户端。所谓的服务端资源可以是一个web页面，也可以是JSON或XML等。服务器单独负责判断记号是什么及其含义，并在HTTP响应头中将其传送到客户端。比如，浏览器第一次请求一个资源的时候，服务端给予返回，并且返回了Etag: "50b1c1d4f775c61:df3" 这样的字样给浏览器，当浏览器再次请求这个资源的时候，浏览器会将If-None-Match: w/"50b1c1d4f775c61:df3" 传输给服务端，服务端拿到该ETAG，对比资源是否发生变化，如果资源未发生改变，则返回304HTTP状态码，不返回具体的资源。

Last-Modified：标示这个响应资源的最后修改时间。web服务器在响应请求时，告诉浏览器资源的最后修改时间。

If-Modified-Since：当资源过期时（使用Cache-Control标识的max-age），发现资源具有 Last-Modified 声明，则再次向web服务器请求时带上头。

If-Modified-Since，表示请求时间。web服务器收到请求后发现头 If-Modified-Since 则与被请求资源的最后修改时间进行比对。若最后修改时间较新，说明资源有被改动过，则响应整片资源内容（写在响应消息包体内），HTTP 200；若最后修改时间较旧，说明资源无新修改，则响应 HTTP 304（无需包体，节省浏览），告知浏览器继续使用所保存的 cache。

Pragma行是为了兼容 HTTP1.0，作用与 Cache-Control: no-cache 是一样的

Etag/If-None-Match

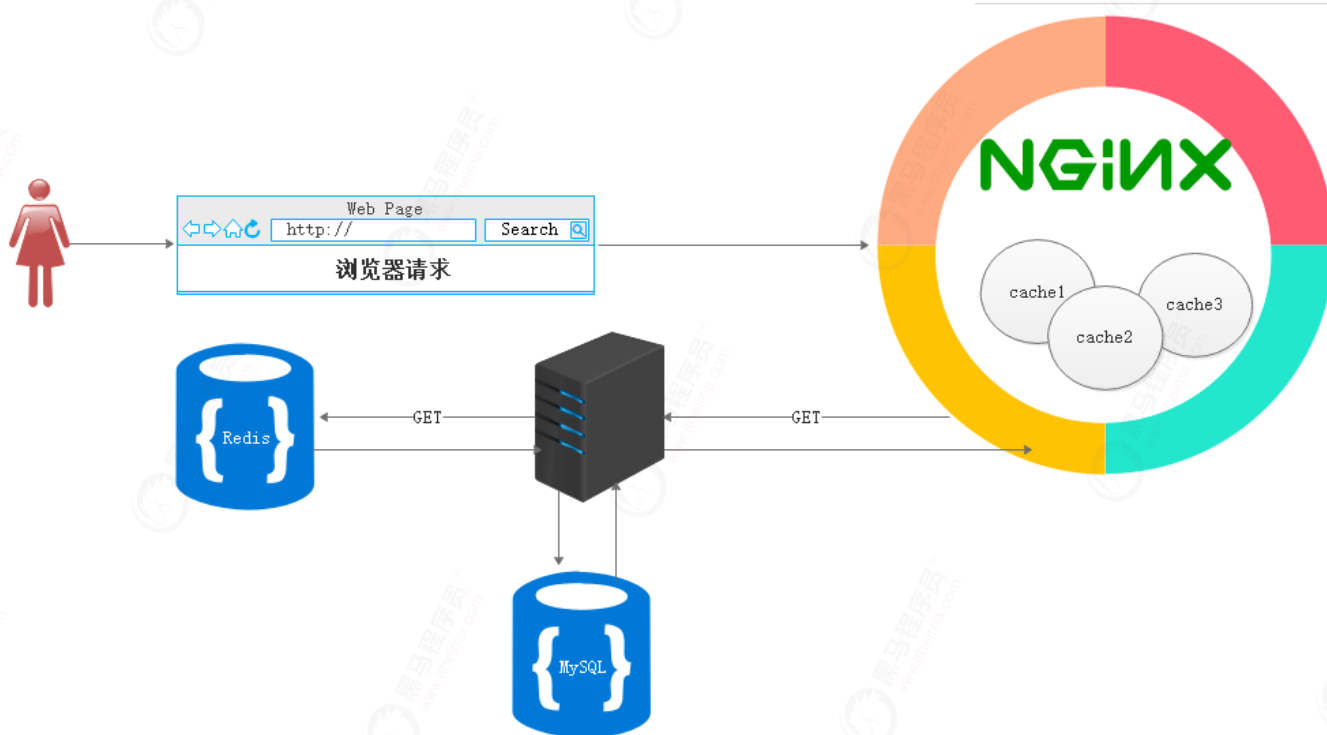
Etag：web服务器响应请求时，告诉浏览器当前资源在服务器的唯一标识（生成规则由服务器决定），如果给定URL中的资源修改，则一定要生成新的Etag值。

If-None-Match : 当资源过期时 (使用Cache-Control标识的max-age) , 发现资源具有Etag声明, 则再次向web服务器请求时带上头 If-None-Match (Etag的值)。web服务器收到请求后发现头 If-None-Match 则与被请求资源的相应校验串进行比对, 决定返回200或304。

Etag:

Last-Modified 标注的最后修改只能精确到秒级, 如果某些文件在1秒钟以内, 被修改多次的话, 它将不能准确标注文件的修改时间, 如果某些文件会被定期生成, 当有时内容并没有任何变化, 但 Last-Modified 却改变了, 导致文件没法使用缓存有可能存在服务器没有准确获取文件修改时间, 或者与代理服务器时间不一致等情形 Etag是服务器自动生成或者由开发者生成的对应资源在服务器端的唯一标识符, 能够更加准确的控制缓存。Last-Modified 与 ETag 是可以一起使用的, 服务器会优先验证 ETag , 一致的情况下, 才会继续比对 Last-Modified , 最后才决定是否返回304。

4.3 代理缓存



用户如果请求获取的数据不是需要后端服务器处理返回, 如果我们需要对数据做缓存来提高服务器的处理能力, 我们可以按照如下步骤实现:

- 1、请求Nginx, Nginx将请求路由给后端服务
- 2、后端服务查询Redis或者MySQL, 再将返回结果给Nginx
- 3、Nginx将结果存入到Nginx缓存, 并将结果返回给用户
- 4、用户下次执行同样请求, 直接在Nginx中获取缓存数据

4.3.1 proxy_cache

proxy_cache 是用于 proxy 模式 (一般也可称为反代) 的缓存功能, proxy_cache 在 Nginx 配置的 http 段、server 段 (location 段) 中分别写入不同的配置。http 段中的配置用于定义 proxy_cache 空间, server 段中的配置用于调用 http 段中的定义, 启用对 server 的缓存功能。

属性使用说明

proxy_cache_path:

Example

```
proxy_cache_path /usr/local/openresty/nginx/cache levels=1:2
keys_zone=openresty_cache:10m max_size=10g inactive=60m use_temp_path=off;
```

【作用】指定缓存存储的路径，缓存存储在/usr/local/openresty/nginx/cache目录

【levels=1:2】设置一个两级目录层次结构存储缓存，在单个目录中包含大量文件会降低文件访问速度，因此我们建议对大多数部署使用两级目录层次结构。如果 levels 未包含该参数，Nginx 会将所有文件放在同一目录中。

【keys_zone=openresty_cache:10m】设置共享内存区域，用于存储缓存键和元数据，例如使用计时器。拥有内存中的密钥副本，Nginx 可以快速确定请求是否是一个 HIT 或 MISS 不必转到磁盘，从而大大加快了检查速度。1 MB 区域可以存储大约 8,000 个密钥的数据，因此示例中配置的 10 MB 区域可以存储大约 80,000 个密钥的数据。

【max_size=10g】设置缓存大小的上限。它是可选的；不指定值允许缓存增长以使用所有可用磁盘空间。当缓存大小达到限制时，一个称为缓存管理器的进程将删除最近最少使用的缓存，将大小恢复到限制之下的文件。

【inactive=60m】指定项目在未被访问的情况下可以保留在缓存中的时间长度。在此示例中，缓存管理器进程会自动从缓存中删除 60 分钟未请求的文件，无论其是否已过期。默认值为 10 分钟 (10m)。非活动内容与过期内容不同。Nginx 不会自动删除缓存 header 定义为已过期内容（例如 Cache-Control:max-age=120）。过期（陈旧）内容仅在指定时间内未被访问时被删除。访问过期内容时，Nginx 会从原始服务器刷新它并重置 inactive 计时器。

【use_temp_path=off】表示NGINX会将临时文件保存在缓存数据的同一目录中。这是为了避免在更新缓存时，磁盘之间互相复制响应数据，我们一般关闭该功能。

proxy_cache:

设置是否开启对后端响应的缓存，如果开启的话，参数值就是zone的名称，比如:proxy_cache openresty_cache;

proxy_cache_valid:

针对不同的response code设定不同的缓存时间，如果不设置code，默认为200,301,302,也可以用any指定所有code

Example:

【proxy_cache_valid 200 304 10s;】所有200/304响应的数据都缓存10秒。

【proxy_cache_valid any 1m;】所有请求响应的值都缓存1分钟。

proxy_cache_min_uses:

指定在多少次请求之后才缓存响应内容,这里表示将缓存内容写入到磁盘。

Example:

【proxy_cache_min_uses 3;】同一个请求达到了3次, 才将缓存写入磁盘。

proxy_cache_lock:

默认不开启, 开启的话则每次只能有一个请求更新相同的缓存, 其他请求要么等待缓存有数据要么限时等待锁释放;nginx 1.1.12才开始有。

proxy_cache_key:

缓存文件的唯一key, 可以根据它实现对缓存文件的清理操作

4.3.2 缓存操作

我们在 nginx.conf 中添加如下配置:

```
#缓存配置
proxy_cache_path /usr/local/openresty/nginx/cache levels=1:2 keys_zone=openresty_cache:10m
max_size=10g inactive=60m use_temp_path=off;

server {
    listen      80;
    server_name localhost;

    #html配置
    location ~ /\.html {
        #静态文件路径
        root /usr/local/server/html;
        #缓存10秒
        expires 10s;
    }

    #非html配置
    location / {
        #启用缓存openresty_cache
        proxy_cache openresty_cache;
        #针对指定请求缓存
        #proxy_cache_methods GET;
        #设置指定请求会缓存
        proxy_cache_valid 200 304 10s;
        #最少请求1次才会缓存
        proxy_cache_min_uses 3;
        #如果并发请求, 只有第1个请求会去服务器获取数据
        #proxy_cache_lock on;
    }
}
```

```
#唯一的key
proxy_cache_key $host$uri$is_args$args;
proxy_pass http://myip:18081;
}
}
```

此时 `/usr/local/openresty/nginx/cache` 目录下只有1个temp文件夹。

我们执行3次请求 `<http://192.168.211.141/user/wangwu>`，可以发现此时多了一些其他目录，这些目录就是存储每个请求对应的缓存。

```
[root@localhost cache]# ls
```

```
0 1 4 8 9 a f temp
```

```
[root@localhost cache]#
```

点击进去，还可以看到第2级目录