分布式缓存中间件之redis

分布式缓存中间件之redis

说明书
版本 0.1
项目名称：

修订历史

版本号	作者	修订章节	修订原因	修订日期
0.1	甘建新	初稿	初稿	2017-07-18

目 录

概述

术语

需求背景

- 为提高系统性能;
- 减轻数据库的负载;
- 渠道限流；
- 网关商户限流；

架构

安装与部署

服务器	端口
192.168.180.92	7000
192.168.180.92	7001
192.168.180.92	7002
192.168.180.91	7003
192.168.180.91	7004
192.168.180.91	7005

在本文中仅介绍集群安装步骤。

安装gcc

yum install cpp
yum install binutils
yum install glibc
yum install glibc-kernheaders
yum install glibc-common
yum install glibc-devel
yum install gcc
yum install make
如果make继续报错，信息如下：error: jemalloc/jemalloc.h: No such file or directory
执行 make MALLOC=libc 就行

下载并解压

cd /opt/soft
wget http://download.redis.io/releases/redis-4.0.0.tar.gz
tar -zxvf redis-3.2.4.tar.gz

编译安装

cd redis-4.0.0
make && make install

将 redis-trib.rb 复制到 /usr/local/bin 目录下

cd src cp redis-trib.rb /usr/local/bin/

创建 Redis 节点

192.168.180.92

在/opt/soft/redis-4.0.0 目录下创建 redis_cluster 目录；
mkdir redis_cluster

在 redis_cluster 目录下，创建名为7000、7001、7002的目录，并将 redis.conf 拷贝到这三个目录中
mkdir 7000 7001 7002
cp redis.conf redis_cluster/7000
cp redis.conf redis_cluster/7001
cp redis.conf redis_cluster/7002
分别修改这三个配置文件，修改如下内容
port 7000 //端口7000,7002,7003
bind 本机ip //默认ip为127.0.0.1 需要改为其他节点机器可访问的ip 否则创建集群时无法访问对应的端口，无法创建集群
daemonize yes //redis后台运行
pidfile /var/run/redis_7000.pid //pidfile文件对应7000,7001,7002
cluster-enabled yes //开启集群 把注释#去掉
cluster-config-file nodes_7000.conf //集群的配置 配置文件首次启动自动生成 7000,7001,7002
cluster-node-timeout 15000 //请求超时 默认15秒，可自行设置
appendonly yes //aof日志开启 有需要就开启，它会每次写操作都记录一条日志

192.168.180.91

接着在另外一台机器上（192.168.180.91），的操作重复以上三步，只是把目录改为7003、7004、7005，对应的配置文件也按照这个规则修改即可

启动各个节点

192.168.180.92

./redis-server redis_cluster/7000/redis.conf
./redis-server redis_cluster/7001/redis.conf
./redis-server redis_cluster/7002/redis.conf

192.168.180.91

./redis-server redis_cluster/7003/redis.conf
./redis-server redis_cluster/7004/redis.conf
./redis-server redis_cluster/7005/redis.conf
批量脚本
for((i=1;i<=6;i++)); do /opt/soft/redis/bin/redis-server /usr/local/redis-cluster/700$i/redis.conf; done

检查 redis 启动情况

ps -ef| grep redis

```
[root@devbank91 redis_cluster]# ps -ef| grep redis
root       2327     1  0 Jul18 ?        00:10:57 src/redis-server 192.168.180.91:7003 [cluster]
root       2331     1  0 Jul18 ?        00:10:56 src/redis-server 192.168.180.91:7004 [cluster]
root       2335     1  0 Jul18 ?        00:10:38 src/redis-server 192.168.180.91:7005 [cluster]
root       5977  5939  0 21:29 pts/1    00:00:00 grep redis
```

netstat -tnlp | grep redis

```
[root@devbank91 redis_cluster]# netstat -tnlp | grep redis
tcp        0      0 192.168.180.91:17003        0.0.0.0:*                   LISTEN      2327/src/redis-serv
tcp        0      0 192.168.180.91:17004        0.0.0.0:*                   LISTEN      2331/src/redis-serv
tcp        0      0 192.168.180.91:17005        0.0.0.0:*                   LISTEN      2335/src/redis-serv
tcp        0      0 192.168.180.91:7003         0.0.0.0:*                   LISTEN      2327/src/redis-serv
tcp        0      0 192.168.180.91:7004         0.0.0.0:*                   LISTEN      2331/src/redis-serv
tcp        0      0 192.168.180.91:7005         0.0.0.0:*                   LISTEN      2335/src/redis-serv
```

安装 ruby

安装命令如下：
yum -y install ruby ruby-devel rubygems rpm-build
gem install redis

创建集群

Redis 官方提供了 redis-trib.rb 这个工具，就在解压目录的 src 目录中，第三步中已将它复制到 /usr/local/bin 目录中，可以直接在命令行中使用了。使用下面这个命令即可完成安装。
redis-trib.rb create --replicas 1 192.168.180.91:7003 192.168.180.91:7004 192.168.180.91:7005 192.168.180.92:7000 192.168.180.92:7001 192.168.180.92:7002

集群验证

./redis-cli -h 192.168.180.92 -p 7000
cluster info

```
[root@hdp2212 redis-4.0.0]# src/redis-cli -h 192.168.180.92 -p 7000
192.168.180.92:7000>
192.168.180.92:7000> cluster info
cluster_state:ok
cluster_slots_assigned:16384
cluster_slots_ok:16384
cluster_slots_pfail:0
cluster_slots_fail:0
cluster_known_nodes:6
cluster_size:3
cluster_current_epoch:6
cluster_my_epoch:4
cluster_stats_messages_ping_sent:572492
cluster_stats_messages_pong_sent:591756
cluster_stats_messages_meet_sent:4
cluster_stats_messages_sent:1164252
cluster_stats_messages_ping_received:591755
cluster_stats_messages_pong_received:572496
cluster_stats_messages_meet_received:1
cluster_stats_messages_received:1164252
```

查看集群节点
cluster nodes

```
192.168.180.92:7000> cluster nodes
804a03cb887989cfb890955919c8a982fdf0c0e6 192.168.180.91:7003@17003 master - 0 1500992202149 1 connected 5461-10922
22efad6569a739567c7ee480254e37b54eade20d 192.168.180.92:7002@17002 slave 804a03cb887989cfb890955919c8a982fdf0c0e6 0 1500992203151 6 connected
149898ab6a74e3476058cfeaea5267e7c5e7aaab 192.168.180.91:7004@17004 slave cc1f883d92680d7f40150c40de59a51266441ffa 0 1500992202000 4 connected
fdfc0c230374cfda8c967813 1cfe68811dbfa5e2 192.168.180.91:7005@17005 slave 7c2d6fc9dba7ddf4d5c8ed0e2c6ca64c9ac2ce54 0 1500992202000 5 connected
7c2d6fc9dba7ddf4d5c8ed0e2c6ca64c9ac2ce54 192.168.180.92:7001@17001 master - 0 1500992204153 5 connected 10923-16383
cc1f883d92680d7f40150c40de59a51266441ffa 192.168.180.92:7000@17000 myself,master - 0 1500992201000 4 connected 0-5460
192.168.180.92:7000>
```

基准测试

1K数据
[root@hdp2212 redis-4.0.0]# src/redis-benchmark -h 192.168.180.92 -p 7000 -t set,lpush -c 1000 -q -d 1024
SET: 130890.05 requests per second
LPUSH: 62150.41 requests per second
1M数据
[root@hdp2212 redis-4.0.0]# src/redis-benchmark -h 192.168.180.92 -p 7000 -t set,lpush -c 1000 -q -d 10240
SET: 69396.25 requests per second
LPUSH: 17652.25 requests per second
循环压测
for i in {1..10}
do
src/redis-benchmark -h 192.168.180.92 -p 7000 -t set,lpush -c 9000 -q -d 10240
done

停止服务

停止所有：pkill redis-server
停止单个服务：kill -15 pid
批量停服务：shutdown_all.sh
for((i=1;i<=6;i++)); do /opt/soft/redis/bin/redis-cli -c -h 192.168.180.91 -p 700$i shutdown; done

功能介绍

数据类型

Redis支持五种数据类型：string（字符串），hash（哈希），list（列表），set（集合）及zset(sorted set：有序集合)。

String（字符串）

string类型是二进制安全的。意思是redis的string可以包含任何数据。比如jpg图片或者序列化的对象 。
string类型是Redis最基本的数据类型，一个键最大能存储512MB。
命令：
SET：将键key设定为指定的"字符串"值；
GET：返回key的value。如果key不存在，返回特殊值nil。如果key的value不是string，就返回错误，因为GET只处理string类型的values。

```
192.168.180.91:7003> SET name "runoob"
OK
192.168.180.91:7003> GET name
"runoob"
192.168.180.91:7003>
```

Hash（哈希）

Redis hash 是一个键名对集合。
Redis hash是一个string类型的field和value的映射表，hash特别适合用于存储对象。
每个 hash 可以存储 2^{32} -1 键值对（40多亿）。
命令：
HMSET：设置 key 指定的哈希集中指定字段的值。
如果 key 指定的哈希集不存在，会创建一个新的哈希集并与 key 关联。
如果字段在哈希集中存在，它将被重写。
HGET：返回 key 指定的哈希集中该字段所关联的值；
HGETALL：返回 key 指定的哈希集中所有的字段和值。返回值中，每个字段名的下一个是它的值，所以返回值的长度是哈希集大小的两倍。

```
192.168.180.91:7003> HMSET user:1 username runoob password runoob points 200
OK
192.168.180.91:7003> HGET user:1 username
"runoob"
192.168.180.91:7003> HGETALL user:1
1) "username"
2) "runoob"
3) "password"
4) "runoob"
5) "points"
6) "200"
```

List（列表）

Redis 列表是简单的字符串列表，按照插入顺序排序。你可以添加一个元素到列表的头部（左边）或者尾部（右边）。
列表最多可存储 2^{32} - 1 元素 (4294967295, 每个列表可存储40多亿)。

```
192.168.180.92:7001> lpush runoob redis
(integer) 1
192.168.180.92:7001> lpush runoob mongodb
(integer) 2
192.168.180.92:7001> lpush runoob rabitmq
(integer) 3
192.168.180.92:7001> lrange runoob 0 10
1) "rabitmq"
2) "mongodb"
3) "redis"
192.168.180.92:7001>
```

Set（集合）

Redis的Set是string类型的无序集合。
集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是O(1)。

```
192.168.180.92:7001> sadd runoob redis
(integer) 1
192.168.180.92:7001> sadd runoob mongodb
(integer) 1
192.168.180.92:7001> sadd runoob rabitmq
(integer) 1
192.168.180.92:7001> sadd runoob rabitmq
(integer) 0
192.168.180.92:7001> smembers runoob
1) "mongodb"
2) "rabitmq"
3) "redis"
```

zset(sorted set：有序集合)

Redis zset 和 set 一样也是string类型元素的集合,且不允许重复的成员。
不同的是每个元素都会关联一个double类型的分数。redis正是通过分数来为集合中的成员进行从小到大的排序。
zset的成员是唯一的,但分数(score)却可以重复。

```
192.168.180.92:7001> del runoob
(integer) 1
192.168.180.92:7001> zadd runoob 0 redis
(integer) 1
192.168.180.92:7001> zadd runoob 0 mongodb
(integer) 1
192.168.180.92:7001> zadd runoob 0 rabitmq
(integer) 1
192.168.180.92:7001> zadd runoob 0 rabitmq
(integer) 0
192.168.180.92:7001> ZRANGEBYSCORE runoob 0 1000
1) "mongodb"
2) "rabitmq"
3) "redis"
```

客户端命令

客户端进入：./redis-cli --help
所有命令参照https://redis.io/commands

Redis HyperLogLog基数统计

hyperloglog只会根据输入元素来计算基数，而不会存储元素本身，所以不能像集合那样返回各个元素本身。
什么是基数?
比如数据集 {1, 3, 5, 7, 5, 7, 8}， 那么这个数据集的基数集为 {1, 3, 5 ,7, 8}, 基数(不重复元素)为5。
基数估计就是在误差可接受的范围内，快速计算基数。
基本命令：
PFADD：添加指定元素到 HyperLogLog 中；
PFCOUNT：返回给定 HyperLogLog 的基数估算值；
PFMERGE：将多个 HyperLogLog 合并为一个 HyperLogLog。

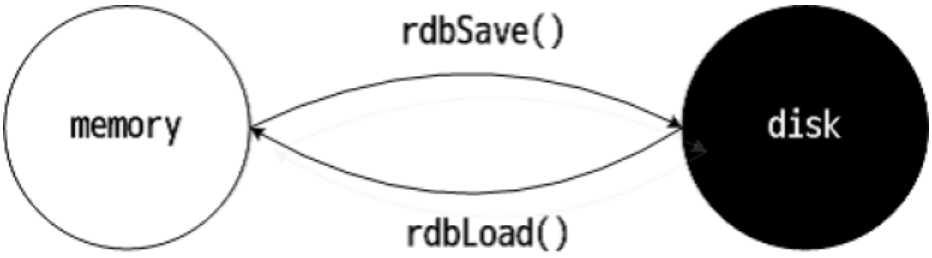
持久化与数据备份恢复

Redis 提供两种持久化方式：RDB 和 AOF。Redis 允许两者结合，也允许两者同时关闭。
AOF 持久化和 RDB 持久化的最主要区别在于，前者记录了数据的变更，而后者是保存了数据本身。

RDB

RDB 可以定时备份内存中的数据集。服务器启动的时候，可以从 RDB 文件中恢复数据集。

持久化运作机制



Redis 支持两种方式进行 RDB 持久化：当前进程执行和后台执行（BGSAVE）。RDB BGSAVE 策略是 fork 出一个子进程，把内存中的数据集整个 dump 到硬盘上。两个场景举例：

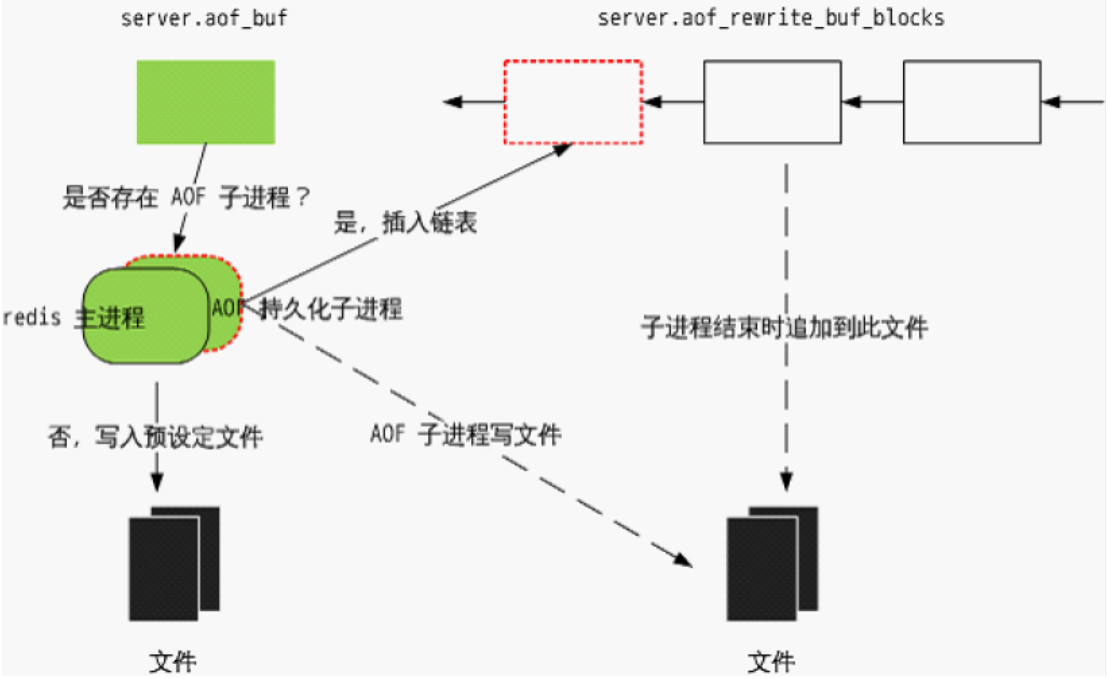
- Redis 服务器初始化过程中，设定了定时事件，每隔一段时间就会触发持久化操作； 进入定时事件处理程序中，就会 fork 产生子进程执行持久化操作。
- Redis 服务器预设了 save 指令，客户端可要求服务器进程中断服务，执行持久化操作。

AOF

AOF(append only file)
可以记录服务器的所有写操作。在服务器重新启动的时候，会把所有的写操作重新执行一遍，从而实现数据备份。当写操作集过大（比原有的数据集还大），Redis 会重写写操作集。

持久化运作机制

和 redis RDB 持久化运作机制不同，redis AOF 有后台执行和边服务边备份两种方式。



后台执行

fork 一个子进程，主进程仍进行服务，子进程执行AOF 持久化，数据被dump 到磁盘上。与 RDB 不同的是，后台子进程持久化过程中，主进程会记录期间的所有数据变更（主进程还在服务），并存储在 server.aof_rewrite_buf_blocks 中；后台子进程结束后，Redis 更新缓存追加到 AOF 文件中。

边服务边备份

Redis 服务器会把所有的数据变更存储在 server.aof_buf 中，并在特定时机将更新缓存写入预设定的文件（server.aof_filename）。特定时机有三种：

- 进入事件循环之前
- Redis 服务器定时程序 serverCron() 中
- 停止 AOF 策略的 stopAppendOnly() 中

细说更新缓存

每一次数据变更记录都会写入 server.aof_buf 中，同时如果后台子进程在持久化，变更记录还会被写入 server.server.aof_rewrite_buf_blocks 中。server.aof_buf 会在特定时期写入指定文件，server.server.aof_rewrite_buf_blocks 会在后台持久化结束后追加到文件。
Redis 源码中是这么实现的：propagate()->~~feedAppendOnlyFile()~~->aofRewriteBufferAppend()；
后台持久化的数据首先会被写入"temp-rewriteaof-bg-%d.aof"，其中"%d"是AOF 子进程 id；待 AOF 子进程结束后，"temp-rewriteaof-bg-%d.aof"会被以追加的方式打开，继而写入 server.aof_rewrite_buf_blocks 中的更新缓存，最后"temp-rewriteaof-bg-%d.aof"文件被命名为 server.aof_filename，所以之前的名为 server.aof_filename 的文件会被删除，也就是说边服务边备份写入的文件会被删除。边服务边备份的数据会被一直写入到 server.aof_filename文件中。
因此，确实会产生两个文件，但是最后都会变成 server.aof_filename 文件。

恢复

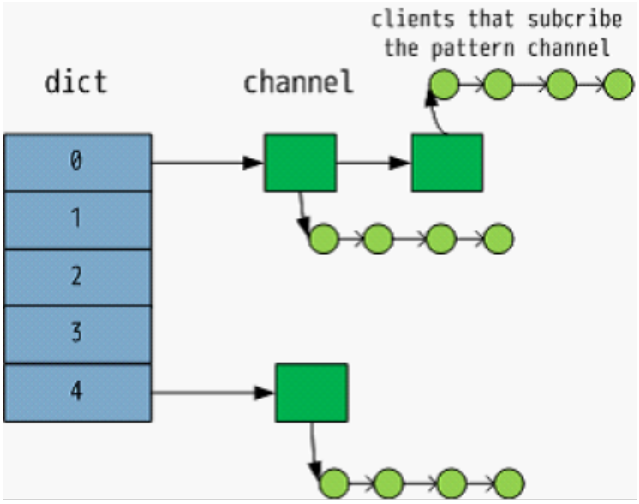
AOF 的数据恢复过程设计很巧妙，它模拟一个 Redis 的服务过程。Redis 首先虚拟一个客户端，读取 AOF 文件恢复 Redis 命令和参数；接着过程就和服务客户端一样执行命令相应的函数，从而恢复数据，这样做的目的无非是提高代码的复用率。这些过程主要在 loadAppendOnlyFile() 中实现。

Redis 发布订阅

Redis 提供两个订阅模式：频道（channel）订阅和 glob-style 模式（pattern）频道订阅。

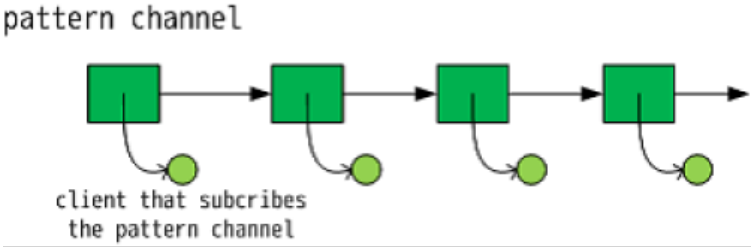
频道订阅

容易理解，即CA（client A）向服务器订阅了频道 news，当 CB 向 news 发布消息的时候，CA 便能收到。
频道订阅是一个 dict，每个 channel 被哈希进相应的桶，每个 channel 对应一个 clients，clients 都订阅了此 channel。当有消息发布的时候，检索 channel，遍历 clients，发布消息。



glob-style 模式（pattern）频道订阅

需要先解释什么是 glob-style？举一个简单的例子，rm *.jpg ，linux 下这条命令删除当前目录下所有 jpg 图片，所用到的是 glob-style 模式匹配，你可以将他理解为某种 style 的正则表达式;
举例，CA（client A）向服务器订阅了频道*.news：
当 CB 向 China.news 发布消息的时候，CA 能收到,
当 CB 向 America.news 发布消息的时候，CA 能收到,
当 CB 向 AV.news 发布消息的时候，CA 便能收到。
模式频道订阅是一个 list。当有消息发布的时候，channel 与 glob-style pattern 匹配，发布消息。



Redis 事务

Redis 事务简述
MULTI，EXEC，DISCARD，WATCH 四个命令是 Redis 事务的四个基础命令。其中：

- MULTI，告诉 Redis 服务器开启一个事务。注意，只是开启，而不是执行；
- EXEC，告诉 Redis 开始执行事务；
- DISCARD，告诉 Redis 取消事务；
- WATCH，监视某一个键值对，它的作用是在事务执行之前如果监视的键值被修改，事务会被取消。

原子性，即一个事务中的所有操作，要么全部完成，要么全部不完成，不会结束在中间某个环节。Redis 事务不支持原子性，最明显的是 Redis 不支持回滚操作。
一致性，在事务开始之前和事务结束以后，数据库的完整性没有被破坏。这一点，Redis 事务能够保证。
隔离性，当两个或者多个事务并发访问（此处访问指查询和修改的操作）数据库的同一数据时所表现出的相互关系。Redis 不存在多个事务的问题，因为 Redis 是单进程单线程的工作模式。
持久性，在事务完成以后，该事务对数据库所作的更改便持久地保存在数据库之中，并且是完全的。Redis 提供两种持久化的方式，即 RDB 和 AOF。RDB 持久化只备份当前内存中的数据集，事务执行完毕时，其数据还在内存中，并未立即写入到磁盘，所以 RDB 持久化不能保证 Redis 事务的持久性。再来讨论 AOF 持久化，Redis AOF 有后台执行和边服务边备份两种方式。后台执行和 RDB

持久化类似，只能保存当前内存中的数据集；边备份边服务的方式中，因为 Redis 只是每间隔 2s
才进行一次备份，因此它的持久性也是不完整的！

Redis 与 Lua 脚本

Redis 脚本使用 Lua 解释器来执行脚本。 Reids 2.6 版本通过内嵌支持 Lua 环境。执行脚本的常用命令为 EVAL。

语法

Eval 命令的基本语法如下：

```
redis 127.0.0.1:6379> EVAL script numkeys key [key ...] arg [arg ...]
```

实例

以下实例演示了 redis 脚本工作过程：

```
redis 127.0.0.1:6379> EVAL "return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}" 2 key1 key2 first second
1) "key1"
2) "key2"
3) "first"
4) "second"
```

客户端连接限制

maxclients 的默认值是 10000，可以在 redis.conf 中对这个值进行修改。

Redis 管道技术

Redis 管道技术可以在服务端未响应时，客户端可以继续向服务端发送请求，并最终一次性读取所有服务端的响应。

Redis 分区

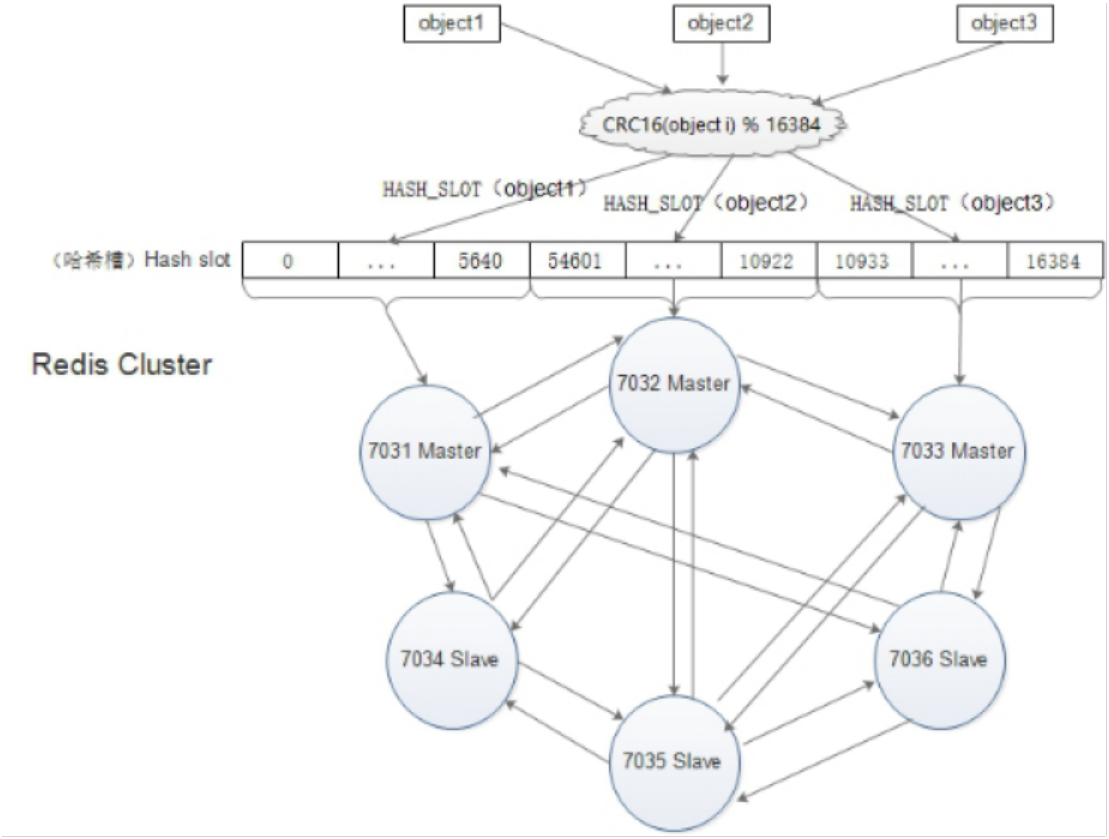
槽(slot)

- redis存取key的时候，都要定位相应的槽(slot)。
- Redis 集群键分布算法使用数据分片（sharding）而非一致性哈希（consistency hashing）来实现： 一个 Redis 集群包含 16384 个哈希槽（hash slot），
  它们的编号为0、1、2、3.....16382、16383，这个槽是一个逻辑意义上的槽，实际上并不存在。redis中的每个key都属于这 16384 个哈希槽的其中一个，存取key时都要进行key->slot的映射计算。

键-槽映射算法

HASH_SLOT（key）= CRC16(key) % 16384
其中 CRC16(key) 语句用于计算键 key 的 CRC16 校验和 。key经过公式计算后得到所对应的哈希槽，而哈希槽被某个主节点管理，从而确定key

在哪个主节点上存取，这也是redis将数据均匀分布到各个节点上的基础。



集群分区好处

无论是memcached的一致性哈希算法，还是redis的集群分区，最主要的目的都是在移除、添加一个节点时对已经存在的缓存数据的定位影响尽可能的降到最小。redis将哈希槽分布到不同节点的做法使得用户可以很容易地向集群中添加或者删除节点， 比如说：

- 1）、如果用户将新节点 D 添加到集群中， 那么集群只需要将节点 A 、B 、 C 中的某些槽移动到节点 D 就可以了。
- 2)、与此类似， 如果用户要从集群中移除节点 A ， 那么集群只需要将节点 A 中的所有哈希槽移动到节点 B 和节点 C ，然后再移除空白（不包含任何哈希槽）的节点 A 就可以了。

因为将一个哈希槽从一个节点移动到另一个节点不会造成节点阻塞， 所以无论是添加新节点还是移除已存在节点，
又或者改变某个节点包含的哈希槽数量， 都不会造成集群下线，从而保证集群的可用性。

集群操作

查看集群信息

cluster info 查看集群状态，槽分配，集群大小等，cluster nodes也可查看主从节点。

新增节点

新增节点配置文件

执行下面的脚本创建脚本配置文件
[root@localhost redis-cluster]# mkdir /usr/local/redis-cluster/7037 && cp /usr/local/redis-cluster/7031/redis.conf /usr/local/redis-cluster/7037/redis.conf && sed -i 's/7031/7037/g' /usr/local/redis-cluster/7037/redis.conf

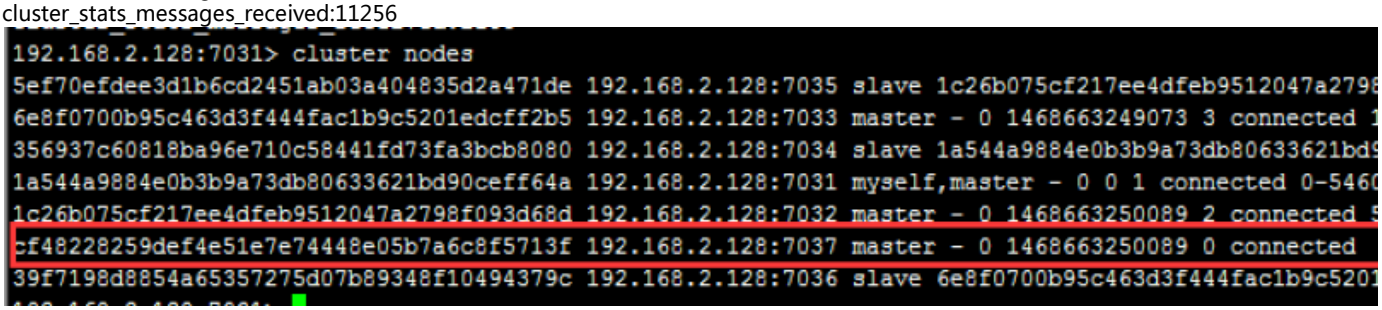
启动新增节点

[root@localhost bin]# /usr/local/redis/bin/redis-server /usr/local/redis-cluster/7037/redis.conf

添加节点到集群

现在已经添加了新增一个节点所需的配置文件，但是这个这点还没有添加到集群中，现在让它成为集群中的一个主节点
[root@localhost redis-cluster]# cd /usr/local/redis/bin/
[root@localhost bin]# ./redis-trib.rb add-node 192.168.2.128:7037 192.168.2.128:7036
>>> Adding node 192.168.2.128:7037 to cluster 192.168.2.128:7036
>>> Performing Cluster Check (using node 192.168.2.128:7036)
S: 2c8d72f1914f9d6052065f7e9910cc675c3c717b 192.168.2.128:7036
slots: (0 slots) slave
replicates 6dbb4aa323864265c9507cf336ef7d3b95ea8d1b
M: 6dbb4aa323864265c9507cf336ef7d3b95ea8d1b 192.168.2.128:7033
slots:10923-16383 (5461 slots) master
1 additional replica(s)
S: 791a7924709bfd7ef5c36d9b9c838925e41e3c2e 192.168.2.128:7034
slots: (0 slots) slave
replicates d9e3c78a7c49689c29ab67a8a17be9d95cb08452
M: d9e3c78a7c49689c29ab67a8a17be9d95cb08452 192.168.2.128:7031
slots:0-5460 (5461 slots) master
1 additional replica(s)
M: 69b63d8db629fa8a689dd1ed25ed941c076d4111 192.168.2.128:7032
slots:5461-10922 (5462 slots) master
1 additional replica(s)
S: e669a91866225279aafcac29bf07b826eb5be91c 192.168.2.128:7035
slots: (0 slots) slave
replicates 69b63d8db629fa8a689dd1ed25ed941c076d4111
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
>>> Send CLUSTER MEET to node 192.168.2.128:7037 to make it join the cluster.
[OK] New node added correctly.
[root@localhost bin]#

./redis-trib.rb add-node 命令中，7037 是新增的主节点，7036 是集群中已有的从节点。再来看看集群信息
192.168.2.128:7031> cluster info
cluster_state:ok
cluster_slots_assigned:16384
cluster_slots_ok:16384
cluster_slots_pfail:0
cluster_slots_fail:0
cluster_known_nodes:7
cluster_size:3
cluster_current_epoch:6
cluster_my_epoch:1
cluster_stats_messages_sent:11256
cluster_stats_messages_received:11256



分配槽

从添加主节点输出信息和查看集群信息中可以看出，我们已经成功的向集群中添加了一个主节点，但是这个主节还没有成为真正的主节点，因为还没有分配槽（slot），也没有从节点，现在要给它分配槽（slot）
[root@localhost bin]# ./redis-trib.rb reshard 192.168.2.128:7031
>>> Performing Cluster Check (using node 192.168.2.128:7031)
M: 1a544a9884e0b3b9a73db80633621bd90ceff64a 192.168.2.128:7031
.....
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
How many slots do you want to move (from 1 to 16384)? 1024
What is the receiving node ID?

系统提示要移动多少个配槽（slot）,并且配槽（slot）要移动到哪个节点，任意输入一个数，如1024，再输入新增节点的ID cf48228259def4e51e7e74448e05b7a6c8f5713f.
What is the receiving node ID? cf48228259def4e51e7e74448e05b7a6c8f5713f
Please enter all the source node IDs.
Type 'all' to use all the nodes as source nodes for the hash slots.
Type 'done' once you entered all the source nodes IDs.
Source node #1:
然后提示要从哪几个节点中移除1024个槽（slot），这里输入'all'表示从所有的主节点中随机转移，凑够1024个哈希槽，然后就开始从新分配槽（slot）了。从新分配完后再次查看集群节点信息

```
192.168.2.128:7031> cluster nodes
5ef70efdee3d1b6cd2451ab03a404835d2a471de 192.168.2.128:7035 slave 1c26b075cf217ee4dfeb9512047a2798f093d
6e8f0700b95c463d3f444fac1b9c5201edcff2b5 192.168.2.128:7033 master - 0 1468666643210 3 connected 11264-
356937c60818ba96e710c58441fd73fa3bcb8080 192.168.2.128:7034 slave 1a544a9884e0b3b9a73db80633621bd90ceff
1a544a9884e0b3b9a73db80633621bd90ceff64a 192.168.2.128:7031 myself,master - 0 0 1 connected 341-5460
1c26b075cf217ee4dfeb9512047a2798f093d68d 192.168.2.128:7032 master - 0 1468666644220 2 connected 5803-1
cf48228259def4e51e7e74448e05b7a6c8f5713f 192.168.2.128:7037 master - 0 1468666643716 7 connected 0-340
39f7198d8854a65357275d07b89348f10494379c 192.168.2.128:7036 slave 6e8f0700b95c463d3f444fac1b9c5201edcff
```

可见，0-340 5461-5802 10923-11263的槽（slot）被分配给了新增加的节点。三个加起来刚好1024个槽（slot）。

指定从节点

现在从节点7036的主节点是7033，现在我们要把他变为新增加节点（7037）的从节点，需要登录7036的客户端
[root@localhost bin]# /usr/local/redis/bin/redis-cli -c -h 192.168.2.128 -p 7036
192.168.2.128:7036> cluster replicate cf48228259def4e51e7e74448e05b7a6c8f5713f
OK
再来查看集群节点信息

```
192.168.2.128:7036> cluster nodes
5ef70efdee3d1b6cd2451ab03a404835d2a471de 192.168.2.128:7035 slave 1c26b075cf217ee4dfeb9512047a2798
1a544a9884e0b3b9a73db80633621bd90ceff64a 192.168.2.128:7031 master - 0 1468667889919 1 connected 3
356937c60818ba96e710c58441fd73fa3bcb8080 192.168.2.128:7034 slave 1a544a9884e0b3b9a73db80633621bd9
cf48228259def4e51e7e74448e05b7a6c8f5713f 192.168.2.128:7037 master - 0 1468667888911 7 connected 0
1c26b075cf217ee4dfeb9512047a2798f093d68d 192.168.2.128:7032 master - 0 1468667888408 2 connected 5
39f7198d8854a65357275d07b89348f10494379c 192.168.2.128:7036 myself,slave cf48228259def4e51e7e74448
6e8f0700b95c463d3f444fac1b9c5201edcff2b5 192.168.2.128:7033 master - 0 1468667890423 3 connected 1
```

可见，7036成为了新增节点7037的从节点。
==========让其成为从节点========== [root@localhost bin]# redis-cli -c -p 7037 cluster replicate cc0a527378f4fe60d1147d1785097f7b8a3457d5

删除节点

指定删除节点的ID即可，如下
[root@localhost bin]#
./redis-trib.rb del-node 192.168.2.128:7037 'a56461a171334560f16652408c2a45e629d268f6'
>>> Removing node a56461a171334560f16652408c2a45e629d268f6 from cluster 192.168.2.128:7037
>>> Sending CLUSTER FORGET messages to the cluster...
>>> SHUTDOWN the node.
[root@localhost bin]#

异常问题：

- 删除节点报is not empty! Reshard data away and try again.异常的意思就是因为当前节点不为空(有slots槽)，需要重新分区(即把当前节点的槽分出去)才可以删除节点。

具体措施：重复执行上面(4)分配槽

- 在How many slots do you want to move (from 1 to 16384)?

指要移动多少个配槽，输入要删除节点现有槽数量；

- 在What is the receiving node ID?

指分配给谁，输入节点id，需要删除的节点除外；

- 在Source node #1:

指从哪几个节点中移除1024个槽，此时输入需要删除节点的id。
注：删除从节点，则不会出现上面的异常，因为从节点没有哈希槽slots

集群操作小结

从上面过程可以看出，添加节点、分配槽、删除节点的过程，不用停止集群，不阻塞集群的其他操作。命令小结：

```
#向集群中添加节点，7037是新增节点，7036是集群中已有的节点
./redis-trib.rb add-node 192.168.2.128:7037 192.168.2.128:7036
#重新分配槽
./redis-trib.rb reshard 192.168.2.128:7031
#指定当前节点的主节点
cluster replicate cf48228259def4e51e7e74448e05b7a6c8f5713f
#删除节点
./redis-trib.rb del-node 192.168.2.128:7037 'a56461a171334560f16652408c2a45e629d268f6'
```

Java使用redis

spring-data-redis

http://docs.spring.io/spring-data/redis/docs/current/api/org/springframework/data/redis/core/RedisTemplate.html

pom.xml

```
<spring.version>4.3.10.RELEASE</spring.version>
<redis.client>2.9.0</redis.client>
<spring-data-redis-version>1.8.4.RELEASE</spring-data-redis-version>

<!-- https://mvnrepository.com/artifact/org.springframework.data/spring-data-redis -->
<dependency>
<groupId>org.springframework.data</groupId>
<artifactId>spring-data-redis</artifactId>
<version>${spring-data-redis-version}</version>
</dependency>
<!-- https://mvnrepository.com/artifact/redis.clients/jedis -->
<dependency>
<groupId>redis.clients</groupId>
<artifactId>jedis</artifactId>
<version>${redis.client}</version>
</dependency>
```

redis.properties

```
#cluster configuration
redis.host1=192.168.180.92
redis.port1=7000
redis.host2=192.168.180.92
redis.port2=7001
redis.host3=192.168.180.92
redis.port3=7002
redis.host4=192.168.180.91
redis.port4=7000
redis.host5=192.168.180.91
redis.port5=7001
redis.host6=192.168.180.91
redis.port6=7002
redis.maxRedirects=3
redis.maxIdle=100
redis.maxTotal=600
```

applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="
_http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.2.xsd_
http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.2.xsd">

<description>Jedis Cluster Configuration集群</description>
<!-- 加载配置属性文件 按需加载 -->
<context:property-placeholder
ignore-unresolvable="false" location="classpath:redis-cluster.properties" />
<bean id="redisClusterConfiguration"
class="org.springframework.data.redis.connection.RedisClusterConfiguration">
<property name="maxRedirects" value="${redis.maxRedirects}"></property>
<property name="clusterNodes">
<set>
<bean class="org.springframework.data.redis.connection.RedisClusterNode">
<constructor-arg name="host" value="${redis.host1}"></constructor-arg>
<constructor-arg name="port" value="${redis.port1}"></constructor-arg>
</bean>
<bean class="org.springframework.data.redis.connection.RedisClusterNode">
<constructor-arg name="host" value="${redis.host2}"></constructor-arg>
<constructor-arg name="port" value="${redis.port2}"></constructor-arg>
</bean>
<bean class="org.springframework.data.redis.connection.RedisClusterNode">
<constructor-arg name="host" value="${redis.host3}"></constructor-arg>
<constructor-arg name="port" value="${redis.port3}"></constructor-arg>
</bean>
<bean class="org.springframework.data.redis.connection.RedisClusterNode">
<constructor-arg name="host" value="${redis.host4}"></constructor-arg>
<constructor-arg name="port" value="${redis.port4}"></constructor-arg>
</bean>
<bean class="org.springframework.data.redis.connection.RedisClusterNode">
<constructor-arg name="host" value="${redis.host5}"></constructor-arg>
<constructor-arg name="port" value="${redis.port5}"></constructor-arg>
</bean>
<bean class="org.springframework.data.redis.connection.RedisClusterNode">
<constructor-arg name="host" value="${redis.host6}"></constructor-arg>
<constructor-arg name="port" value="${redis.port6}"></constructor-arg>
</bean>
</set>
</property>
</bean>
<bean id="jedisPoolConfig" class="redis.clients.jedis.JedisPoolConfig">
<property name="maxIdle" value="${redis.maxIdle}" />
<property name="maxTotal" value="${redis.maxTotal}" />
</bean>
<bean id="jeidsConnectionFactory"
class="org.springframework.data.redis.connection.jedis.JedisConnectionFactory">
<constructor-arg ref="redisClusterConfiguration" />
<constructor-arg ref="jedisPoolConfig" />
</bean>
<bean id="redisTemplate" class="org.springframework.data.redis.core.RedisTemplate">
<property name="connectionFactory" ref="jeidsConnectionFactory" />
</bean>
</beans>
```

详细代码

```
/**
```

- Kjtpay.com Inc. Copyright (c) 2012-2017 All Rights Reserved.

```
*/
package com.kjtpay.redis.cluster;
import java.util.HashMap;
import java.util.Map;
import javax.annotation.Resource;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

```
import org.springframework.data.redis.core.HashOperations;
import org.springframework.data.redis.core.ListOperations;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.core.SetOperations;
import org.springframework.data.redis.core.ValueOperations;
import org.springframework.data.redis.core.ZSetOperations;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
/**
```

- TODO
  *
- @author ganjianxin
- @version $Id: SpringClusterTest.java, v 0.1 2017年7月21日 下午6:13:09 ganjianxin Exp $

```
*/
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext-cluster.xml")
public class SpringClusterTest {
Logger logger = LoggerFactory.getLogger(SpringClusterTest.class);
@Resource(name = "redisTemplate")
public RedisTemplate<String, Object> redisTemplate;

/**
```

- @throws Exception

```
*/
@Test
public void testValue() throws Exception {
// 添加一个 key
ValueOperations<String, Object> value = redisTemplate.opsForValue();
value.set("lp", "hello word");
// 获取 这个 key 的值
logger.info("value:{}", value.get("lp"));
}
/**
```

- @throws Exception

```
*/
@Test
public void testHash() throws Exception {
// 添加 一个 hash集合
HashOperations<String, Object, Object> hash = redisTemplate.opsForHash();
Map<String, Object> map = new HashMap<String, Object>();
map.put("name", "lp");
map.put("age", "26");
hash.putAll("lpMap", map);
// 获取 map
logger.info("hash:{}", hash.entries("lpMap"));
}
/**
```

- @throws Exception

```
*/
@Test
public void testList() throws Exception {
// 添加 一个 list 列表
ListOperations<String, Object> list = redisTemplate.opsForList();
list.rightPush("lpList", "lp");
list.rightPush("lpList", "26");
// 输出 list
logger.info("list:{}", list.range("lpList", 0, 1));
}
/**
```

- @throws Exception

```
*/
@Test
public void testSet() throws Exception {
// 添加 一个 set 集合
SetOperations<String, Object> set = redisTemplate.opsForSet();
set.add("lpSet", "lp");
set.add("lpSet", "26");
set.add("lpSet", "178cm");
// 输出 set 集合
logger.info("set:{}", set.members("lpSet"));
}
/**
```

```
@throws Exception
*/
@Test
public void testZset() throws Exception {
// 添加有序的 set 集合
ZSetOperations<String, Object> zset = redisTemplate.opsForZSet();
zset.add("lpZset", "lp", 0);
zset.add("lpZset", "26", 1);
zset.add("lpZset", "178cm", 2);
// 输出有序 set 集合
logger.info("zset:{}", zset.rangeByScore("lpZset", 0, 2));
}
}
```

redis监控工具

待续；

redis和memcache区别

网络IO 模型

Memcached

- Memcached 是多线程，非阻塞IO 复用的网络模型，分为监听主线程和worker 子线程；
- 监听线程监听网络连接，接受请求后，将连接描述字pipe 传递给worker 线程，进行读写IO, 网络层使用libevent 封装的事件库；
- 多线程模型可以发挥多核作用，但是引入了cache coherency和锁的问题，比如，Memcached 最常用的stats 命令，实际Memcached 所有操作都要对这个全局变量加锁，进行计数等工作，带来了性能损耗；

Redis

- Redis 使用单线程的IO 复用模型，自己封装了一个简单的AeEvent 事件处理框架，主要实现了epoll、kqueue 和select，对于单纯只有IO 操作来说，单线程可以将速度优势发挥到最大；
- 但是Redis 也提供了一些简单的计算功能，比如排序、聚合等，对于这些操作，单线程模型实际会严重影响整体吞吐量，CPU 计算过程中，整个IO 调度都是被阻塞住的；

内存管理方面

Memcached

- 使用预分配的内存池的方式，使用slab 和大小不同的chunk 来管理内存，Item 根据大小选择合适的chunk存储，内存池的方式可以省去申请/释放内存的开销，并且能减小内存碎片产生，但这种方式也会带来一定程度上的空间浪费，并且在内存仍然有很大空间时，新的数据也可能会被剔除；

Redis

- Redis 使用现场申请内存的方式来存储数据，并且很少使用free-list 等方式来优化内存分配，会在一定程度上存在内存碎片，Redis 根据存储命令参数，会把带过期时间的数据单独存放在一起，并把它们称为临时数据，非临时数据是永远不会被剔除的，即便物理内存不够，导致swap 也不会剔除任何非临时数据(但会尝试剔除部分临时数据)，这点上Redis 更适合作为存储而不是cache；

数据一致性问题

- Memcached 提供了cas 命令，可以保证多个并发访问操作同一份数据的一致性问题；
- Redis 没有提供cas 命令，并不能保证这点，不过Redis 提供了事务的功能，可以保证一串命令的原子性，中间不会被任何操作打断；

存储方式

Memcache

只支持简单的key-value 存储

Redis

Redis 除key/value 之外，还支持list,set,sorted set,hash 等众多数据结构

关于不同语言的客户端支持

主流语言都支持

Redis 和Memcached 的集群实现机制对比

Memcached

- Memcached 本身并不支持分布式，因此只能在客户端通过像一致性哈希这样的分布式算法来实现Memcached 的分布式存储；
- 客户端向Memcached 集群发送数据之前，首先会通过内置的分布式算法计算出该条数据的目标节点；

Redis

- 支持服务器端构建分布式存储Redis Cluster；

总体对比：

性能

- Redis 只使用单核，而Memcached 可以使用多核，所以平均每一个核上Redis 在存储小数据时比Memcached 性能更高。而在100k以上的数据中，Memcached 性能要高于Redis，虽然Redis
  最近也在存储大数据的性能上进行优化，但是比起Memcached，还是稍有逊色；

内存使用率

- 使用简单的key-value 存储的话，Memcached 的内存利用率更高，而如果Redis 采用hash 结构来做key-value
  存储，由于其组合式的压缩，其内存利用率会高于Memcached;

数据操作

- Redis 相比Memcached 来说，拥有更多的数据结构和并支持更丰富的数据操作，通常在Memcached
  里，你需要将数据拿到客户端来进行类似的修改再set 回去。这大大增加了网络IO 的次数和数据体积。在Redis
  中，这些复杂的操作通常和一般的GET/SET 一样高效。所以，如果需要缓存能够支持更复杂的结构和操作，那么Redis 会是不错的选择

Redis 具有Memcached 不具备的功能

- 数据持久化；
- 事务；
- 发布/订阅；
- Lua 脚本；
- 内置服务器端集群模块；

redis内存优化

http://blog.csdn.net/u013256816/article/details/51133134

参考资料

http://redis.io/
http://redisbook.com/
https://github.com/huangz1990/redis-3.0-annotated
http://wiki.jikexueyuan.com/project/redis