# 简介

Redis - 内存中的数据结构存储系统，可以用作数据库、缓存和消息中间件

特点：

* **高性能**： Redis 将所有数据集存储在内存中，可以在入门级 Linux 机器中每秒写（SET）11 万次，读（GET）8.1 万次。Redis 支持 Pipelining 命令，可一次发送多条命令来提高吞吐率，减少通信延迟。
* **持久化**：当所有数据都存在于内存中时，可以根据自上次保存以来经过的时间和/或更新次数，使用灵活的策略将更改异步保存在磁盘上。
* **多种数据结构**： Redis 支持各种类型的数据结构，例如字符串、散列、集合、列表、带有范围查询的有序集、位图、超级日志和带有半径查询的地理空间索引。
* **原子操作**：处理不同数据类型的 Redis 操作是原子操作，因此可以安全地 [SET](https://www.redis.com.cn/commands/set.html) 或 [INCR](https://www.redis.com.cn/commands/incr.html) 键，添加和删除集合中的元素等。
* **支持多种语言**：如 C、C++、Erlang、Go、Haskell、Java、JavaScript（Node.js)、Lua、Objective-C、Perl、PHP、Python、R、Ruby、Rust、Scala、Smalltalk 等。
* **主/从复制**： Redis 遵循非常简单快速的主/从复制。配置文件中只需要一行来设置它，而 Slave 在 Amazon EC2 实例上完成 10 MM key 集的初始同步只需要 21 秒。
* **支持分片**：与其他键值存储一样，跨多个 Redis 实例分发数据集非常容易。
* **可移植**： Redis 是用 C 编写的，适用于大多数 POSIX 系统，如 Linux、BSD、Mac OS X、Solaris 等。

zzm理解的使用场景：redis最早需求是为了将模块内部的数据移到模块外部处理，减少数据传输开销以及对服务端的压力，尤其是为了实现分布式场景的高可靠性。对于紧密耦合的模块使用redis，松耦合使用rpc，减少底层数据结构更改引发的改动。

# 安装

## 2.1 源码安装

|  |
| --- |
| # 下载所需版本  wget http://download.redis.io/releases/redis-6.0.8.tar.gz  tar xzf redis-6.0.8.tar.gz  cd redis-6.0.8  make  **# 下面是可选操作**  # 安装到指定目录  make PREFIX=/usr/local/redis install  # 修改环境变量  vim ~/.bashrc  # 在最后一行添上：  export PATH=/usr/local/redis/bin:$PATH  # 使环境变量配置生效  source ~/.bashrc  # 可下载tls进行测试  wget http://downloads.sourceforge.net/tcl/tcl8.6.1-src.tar.gz  tar xzvf tcl8.6.1-src.tar.gz  cd tcl8.6.1/unix/  ./configure  make  make install  # 运行测试  cd redis-6.0.8  make test |

**可调整下内存分配使用方式并使其生效**

|  |
| --- |
| #此参数可用的值为0,1,2  #0表示当用户空间请求更多的内存时，内核尝试估算出可用的内存  #1表示内核允许超量使用内存直到内存用完为止  #2表示整个内存地址空间不能超过swap+(vm.overcommit\_ratio)%的RAM值  echo "vm.overcommit\_memory=1">>/etc/sysctl.conf  sysctl -p |

**测试**

|  |
| --- |
| # 先启动服务端, 后面可以接配置文件  redis-server  # 另一个终端启动客户端, 验证是否成功连接  redis-cli |

## 2.2 修改配置

**修改redis配置方法**

|  |
| --- |
| **# 直接修改配置文件**  vim redis-6.0.8/redis.conf  **# 或使用 CONFIG SET 命令来修改配置**  # 例如 CONFIG GET CONFIG\_SETTING\_NAME  redis 127.0.0.1:6379> CONFIG GET loglevel |

**常用配置说明：**

1.Redis 默认不是以守护进程的方式运行，可以通过该配置项修改，使用 yes 启用守护进程

daemonize no

2.当 Redis 以守护进程方式运行时，Redis 默认会把 pid 写入 /var/run/redis.pid 文件，可以通过 pidfile 指定

pidfile /var/run/redis.pid

3.指定 Redis 监听端口

port 6379

4. 绑定的主机地址

bind 127.0.0.1

5.当客户端闲置多长时间后关闭连接，如果指定为 0，表示关闭该功能

timeout 300

6.指定日志记录级别，Redis 总共支持四个级别：debug、verbose、notice、warning，默认为 verbose

loglevel verbose

7.日志记录方式，默认为标准输出，如果配置 Redis 为守护进程方式运行，而这里又配置为日志记录方式为标准输出，则日志将会发送给 /dev/null

logfile stdout

8.设置数据库的数量，默认数据库为 0，可以使用 SELECT <dbid> 命令在连接上指定数据库 id

databases 16

9.指定在多长时间内，有多少次更新操作，就将数据同步到数据文件，可以多个条件配合

save <seconds> <changes>

Redis 默认配置文件中提供了三个条件：

save 900 1

​ save 300 10

​ save 60 10000

分别表示 90 0 秒（15 分钟）内有 1 个更改，300 秒（5 分钟）内有 10 个更改以及 60 秒内有 10000 个更改。

10.指定存储至本地数据库时是否压缩数据，默认为 yes，Redis 采用 LZF 压缩，如果为了节省 CPU 时间，可以关闭该选项，但会导致数据库文件变的巨大

rdbcompression yes

11.指定本地数据库文件名，默认值为 dump.rdb

dbfilename dump.rdb

12.指定本地数据库存放目录

dir ./

13.设置当本机为 slave 服务时，设置 master 服务的 IP 地址及端口，在 Redis 启动时，它会自动从 master 进行数据同步

slaveof <masterip> <masterport>

14.当 master 服务设置了密码保护时，slave 服务连接 master 的密码

masterauth <master-password>

15.设置 Redis 连接密码，如果配置了连接密码，客户端在连接 Redis 时需要通过 AUTH <password>命令提供密码，默认关闭

requirepass foobared

16.设置同一时间最大客户端连接数，Redis 可以同时打开的客户端连接数为 Redis 进程可以打开的最大文件描述符数，如果设置 maxclients 0，表示不作限制。

maxclients 128

17.指定 Redis 最大内存限制，Redis 在启动时会把数据加载到内存中，达到最大内存后，Redis 会先尝试清除已到期或即将到期的 Key，当此方法处理后，仍然到达最大内存设置，将无法再进行写入操作，但仍然可以进行读取操作。

maxmemory <bytes>

18.指定是否在每次更新操作后进行日志记录，Redis 在默认情况下是异步的把数据写入磁盘，如果不开启，可能会在断电时导致一段时间内的数据丢失。默认为 no

appendonly no

19.指定更新日志文件名，默认为 appendonly.aof

appendfilename appendonly.aof

20.指定更新日志条件，共有 3 个可选值：

no：表示等操作系统进行数据缓存同步到磁盘（快）

always：表示每次更新操作后手动调用 fsync() 将数据写到磁盘（慢，安全） everysec：表示每秒同步一次（折衷，默认值）

appendfsync everysec

21.指定是否启用虚拟内存机制，默认值为 no，简单的介绍一下，VM 机制将数据分页存放，由 Redis 将访问量较少的页即冷数据 swap 到磁盘上，访问多的页面由磁盘自动换出到内存中（在后面的文章我会仔细分析 Redis 的 VM 机制）

vm-enabled no

22.虚拟内存文件路径，默认值为 /tmp/redis.swap，不可多个 Redis 实例共享

vm-swap-file /tmp/redis.swap

23.将所有大于 vm-max-memory 的数据存入虚拟内存,无论 vm-max-memory 设置多小,所有索引数据都是内存存储的( Redis 的索引数据就是 keys ),也就是说,当 vm-max-memory 设置为 0 的时候,其实是所有 value 都存在于磁盘。默认值为 0

vm-max-memory 0

24.vm-page-size 是要根据存储的数据大小来设定的，作者建议如果存储很多小对象，page 大小最好设置为 32 或者 64 bytes；如果存储很大大对象，则可以使用更大的 page，如果不确定，就使用默认值

vm-page-size 32

25.设置 swap 文件中的 page 数量，由于页表（一种表示页面空闲或使用的 bitmap）是在放在内存中的，在磁盘上每 8 个 pages 将消耗 1byte 的内存。

vm-pages 134217728

26.设置访问 swap 文件的线程数,最好不要超过机器的核数,如果设置为 0,那么所有对 swap 文件的操作都是串行的，可能会造成比较长时间的延迟。默认值为 4

vm-max-threads 4

27.设置在向客户端应答时，是否把较小的包合并为一个包发送，默认为开启

glueoutputbuf yes

28.指定在超过一定的数量或者最大的元素超过某一临界值时，采用一种特殊的哈希算法

hash-max-zipmap-entries 64

29.指定是否激活重置哈希，默认为开启（后面在介绍 Redis 的哈希算法时具体介绍）

activerehashing yes

30.指定包含其它的配置文件，可以在同一主机上多个 Redis 实例之间使用同一份配置文件，而同时各个实例又拥有自己的特定配置文件

include /path/to/local.conf

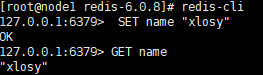
# 常用指令

## 数据类型相关

1.String（字符串）

string是redis最基本的类型，一个key对应一个value。

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 ，一个键最大能存储512MB。



在以上实例中我们使用了 Redis 的 SET 和 GET 命令。键为 name，对应的值为 xlosy。

2.Hash（哈希）

Redis hash 是一个键值(key=>value)对集合。

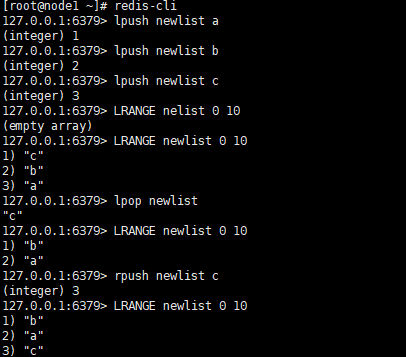
Redis hash 是一个 string 类型的 field 和 value 的映射表，hash 特别适合用于存储对象。



实例中我们使用了 Redis HMSET, HGET 命令，HMSET 设置了两个 field=>value 对, HGET 获取对应 field 对应的 value。

3.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）。

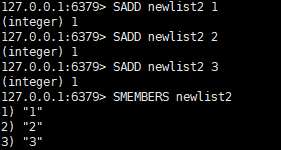


列表最多可存储 232 - 1 元素 (4294967295, 每个列表可存储40多亿)。

4.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)。



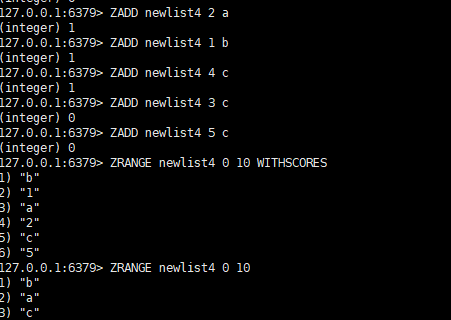
集合中最大的成员数为 232 - 1(4294967295, 每个集合可存储40多亿个成员)。

5.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。

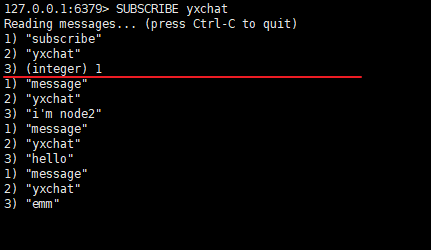
不同的是每个元素都会关联一个double类型的分数。redis正是通过分数来为集合中的成员进行从小到大的排序。

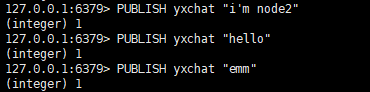
zset的成员是唯一的,但分数(score)却可以重复。



## 发布与订阅

同时开两个客户端，一个订阅频道，另一个负责推送消息





其他指令：

PSUBSCRIBE pattern [pattern …]

订阅一个或多个符合给定模式的频道

PUBSUB subcommand [argument [argument …]]

查看订阅与发布系统状态

PUBLISH channel message

将信息发送到指定的频道

PUNSUBSCRIBE [pattern [pattern …]]

退订所有给定模式的频道

SUBSCRIBE channel [channel …]

订阅一个或多个频道

UNSUBSCRIBE [channel [channel …]]

退订频道

## 事务

Redis 事务可以一次执行多个命令， 单个 Redis 命令的执行是原子性的，但 Redis 没有在事务上增加任何维持原子性的机制，所以 Redis 事务的执行并不是原子性的。事务带有以下三个重要的保证：

批量操作在发送 EXEC 命令前被放入队列缓存

收到 EXEC 命令后进入事务执行，事务中**任意命令执行失败，其余的命令依然被执行**

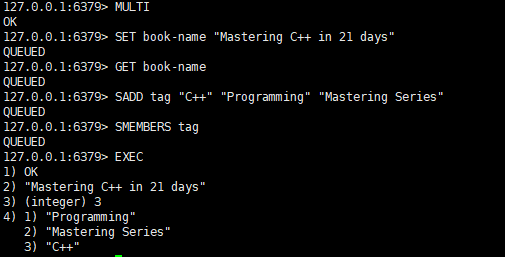
在事务执行过程，**其他客户端提交的命令请求不会插入到事务执行命令序列中**

一个事务从开始到执行会经历以下三个阶段：

开始事务

命令入队

执行事务



其他相关指令：

DISCARD

取消事务，放弃执行事务块内的所有命令

EXEC

执行所有事务块内的命令

MULTI

标记一个事务块的开始

UNWATCH

取消 WATCH 命令对所有 key 的监视

5 WATCH key [key …]

监视一个(或多个) key ，如果在事务执行之前这个(或这些) key 被其他命令所改动，那么事务将被打断

## 脚本

EVAL script numkeys key [key …] arg [arg …]

执行 Lua 脚本

EVALSHA sha1 numkeys key [key …] arg [arg …]

执行 Lua 脚本

SCRIPT EXISTS script [script …]

查看指定的脚本是否已经被保存在缓存当中

SCRIPT FLUSH

从脚本缓存中移除所有脚本

SCRIPT KILL

杀死当前正在运行的 Lua 脚本

SCRIPT LOAD script

将脚本 script 添加到脚本缓存中，但并不立即执行这个脚本

## 连接

AUTH password

验证密码是否正确

ECHO message

打印字符串

PING

查看服务是否运行

QUIT

关闭当前连接

SELECT index

切换到指定的数据库

## 服务器相关

BGREWRITEAOF

异步执行一个 AOF（AppendOnly File） 文件重写操作

BGSAVE

在后台异步保存当前数据库的数据到磁盘

CLIENT KILL [ip:port] [ID client-id]

关闭客户端连接

CLIENT LIST

获取连接到服务器的客户端连接列表

CLIENT GETNAME

获取连接的名称

CLIENT PAUSE timeout

在指定时间内终止运行来自客户端的命令

CLIENT SETNAME connection-name

设置当前连接的名称

CLUSTER SLOTS

获取集群节点的映射数组

COMMAND

获取 Redis 命令详情数组

COMMAND COUNT

获取 Redis 命令总数

COMMAND GETKEYS

获取给定命令的所有键

TIME

返回当前服务器时间

COMMAND INFO command-name [command-name …]

获取指定 Redis 命令描述的数组

CONFIG GET parameter

获取指定配置参数的值

CONFIG REWRITE

对启动 Redis 服务器时所指定的 redis.conf 配置文件进行改写

CONFIG SET parameter value

修改 redis 配置参数，无需重启

CONFIG RESETSTAT

重置 INFO 命令中的某些统计数据

DBSIZE

返回当前数据库的 key 的数量

DEBUG OBJECT key

获取 key 的调试信息

DEBUG SEGFAULT

让 Redis 服务崩溃

FLUSHALL

删除所有数据库的所有key

FLUSHDB

删除当前数据库的所有key

INFO [section]

获取 Redis 服务器的各种信息和统计数值

LASTSAVE

返回最近一次 Redis 成功将数据保存到磁盘上的时间，以 UNIX 时间戳格式表示

MONITOR

实时打印出 Redis 服务器接收到的命令，调试用

ROLE

返回主从实例所属的角色

SAVE

同步保存数据到硬盘

SHUTDOWN [NOSAVE] [SAVE]

异步保存数据到硬盘，并关闭服务器

SLAVEOF host port

将当前服务器转变为指定服务器的从属服务器(slave server)

SLOWLOG subcommand [argument]

管理 redis 的慢日志

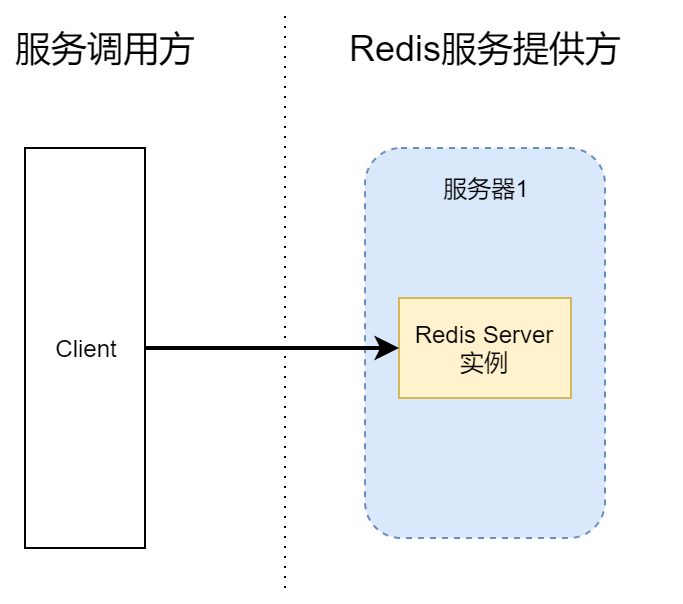
SYNC

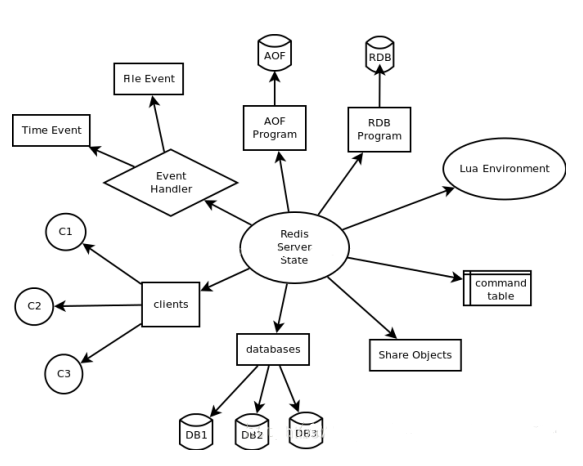
用于复制功能(replication)的内部命令

# 架构原理

## 4.1 整体架构(三种模式）

**单机版server：**





File Event: 处理文件事件(在多个客户端中实现多路复用，接受它们发来的命令请求（读事 件），并将命令的执行结果返回给客户端（写事件）)

Time Event: 定时事件(更新统计信息，清理过期数据，附属节点同步，定期持久化等)

AOF: 命令日志（写操作）的数据持久化

RDB：实际的数据持久化

Lua Environment : Lua 脚本的运行环境. 为了让 Lua 环境符合 Redis 脚本功能的需求， Redis 对 Lua 环境进行了一系列的修改， 包括添加函数库、更换随机函数、 保护全局变量， 等等

Command table(命令表)：在执行命令时，根据字符来查找相应命令的实现函数。

Share Objects（对象共享）：主要存储常见的值：a.各种命令常见的返回值，例如返回值OK、 ERROR、WRONGTYPE等字符；b. 小于 redis.h/REDIS\_SHARED\_INTEGERS (默认1000)的所有整数。通过预分配的一些常见的值对象，并在多个数据结构 之间共享对象，程序避免了重复分配的麻烦。也就是说，这些常见的值在内存 中只有一份。

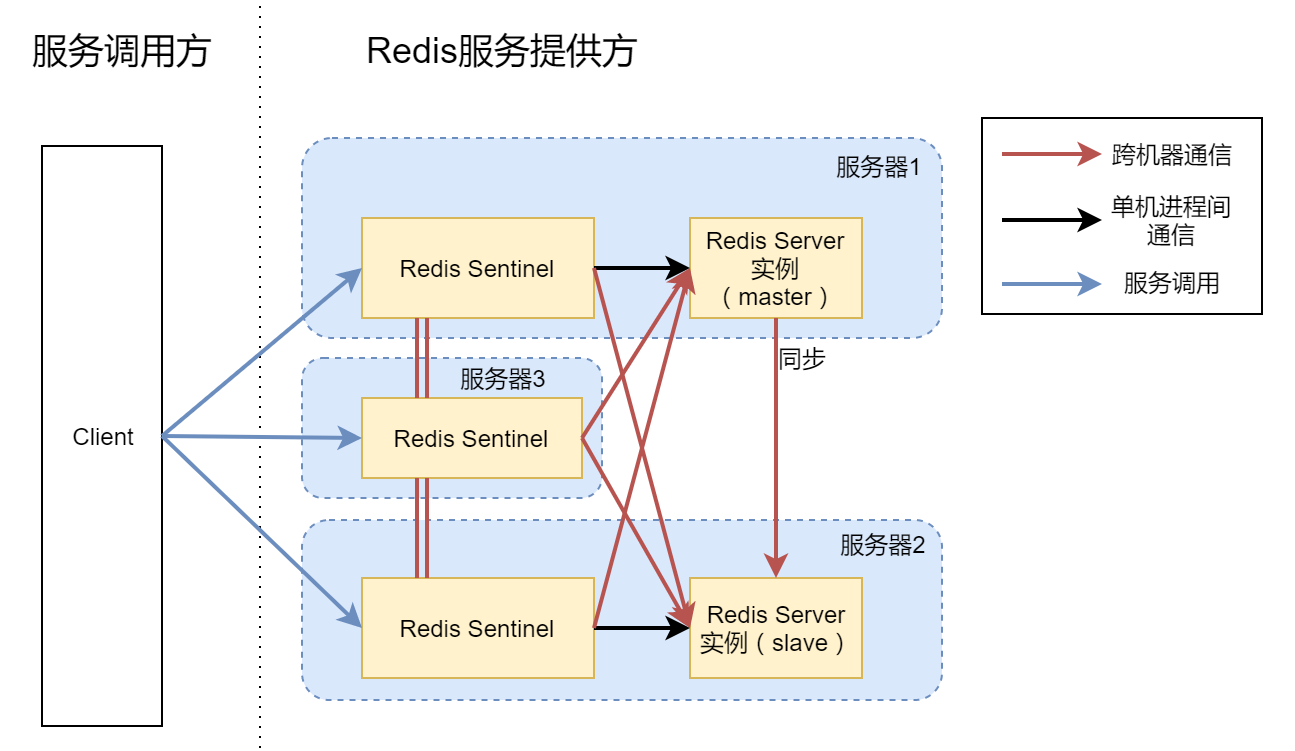
Databases：Redis数据库是真正存储数据的地方。当然，数据库本身也是存储在内存中的

**主从server + sentinel（哨兵）架构：**

没有降低master写压力，仅仅降低了master的读压力，此架构模式中，客户端可以直接与slave交互，获取数据，但不能向slave写数据

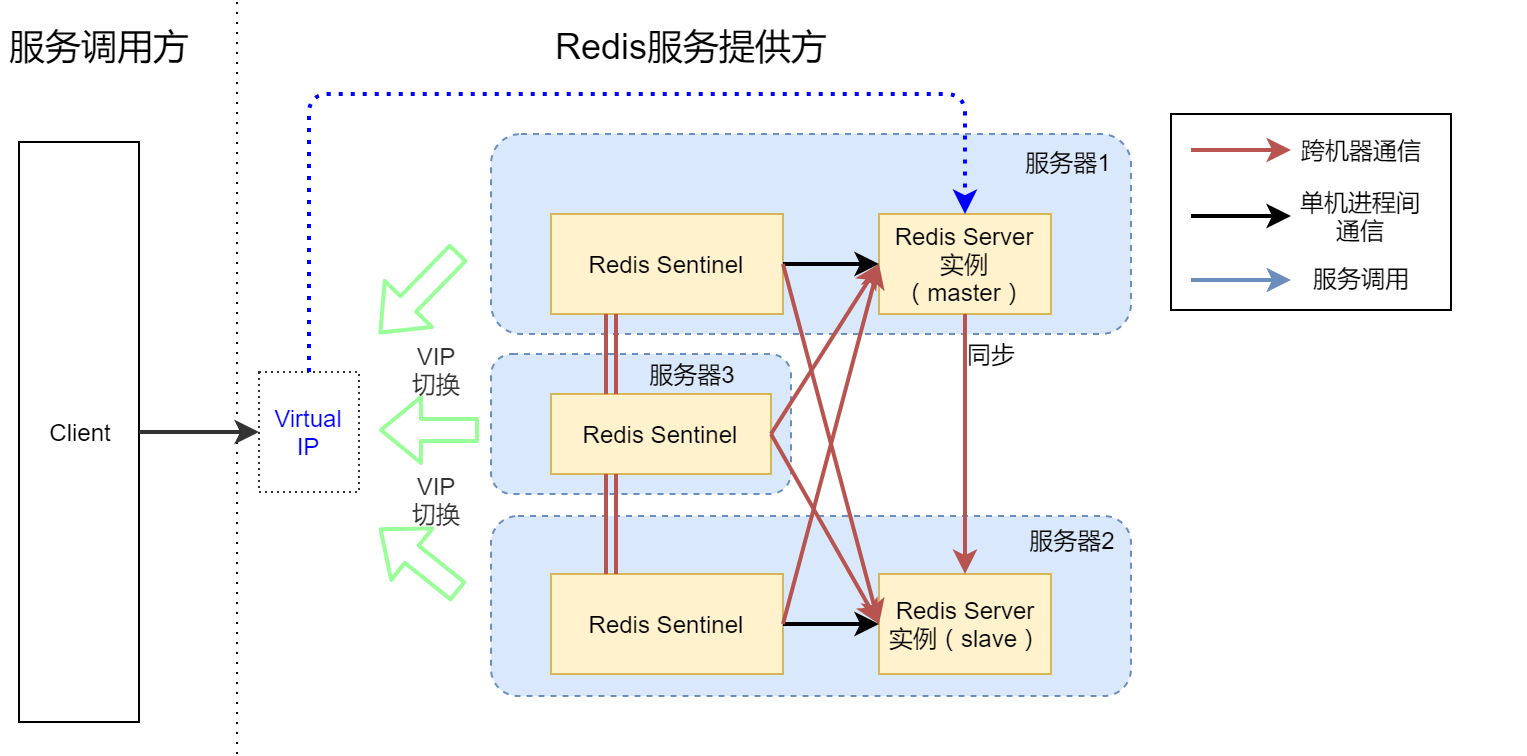
注意：Redis的设定是**只有当超过50%的Sentinel进程可以连通并投票选取新的master**时，才会真正发生主从切换。**当主节点出现故障时，由Redis Sentinel自动完成故障发现和转移，并通知应用方**，实现高可用性

sentinel数目一般为奇数，且大于2，如图所示：



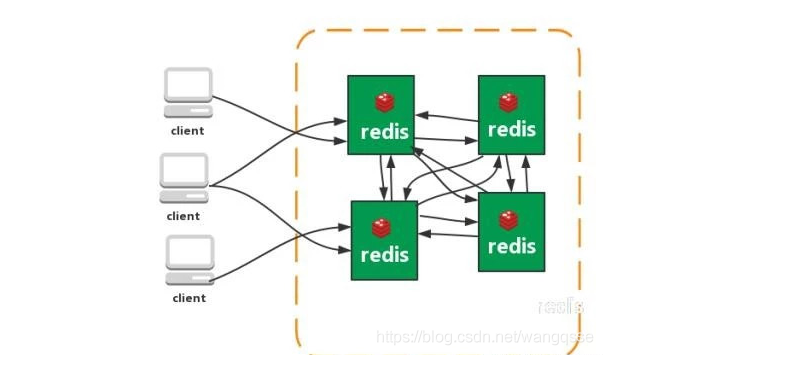
如果资源充足也可以把服务器3上面也开启一个Redis Server，形成1 master + 2 slave的架构

另外，对外可以用**虚拟ip**提升易用性：



**cluster（直连型）：**

redis3.0后支持的**去中心架构**



Redis Cluster特点

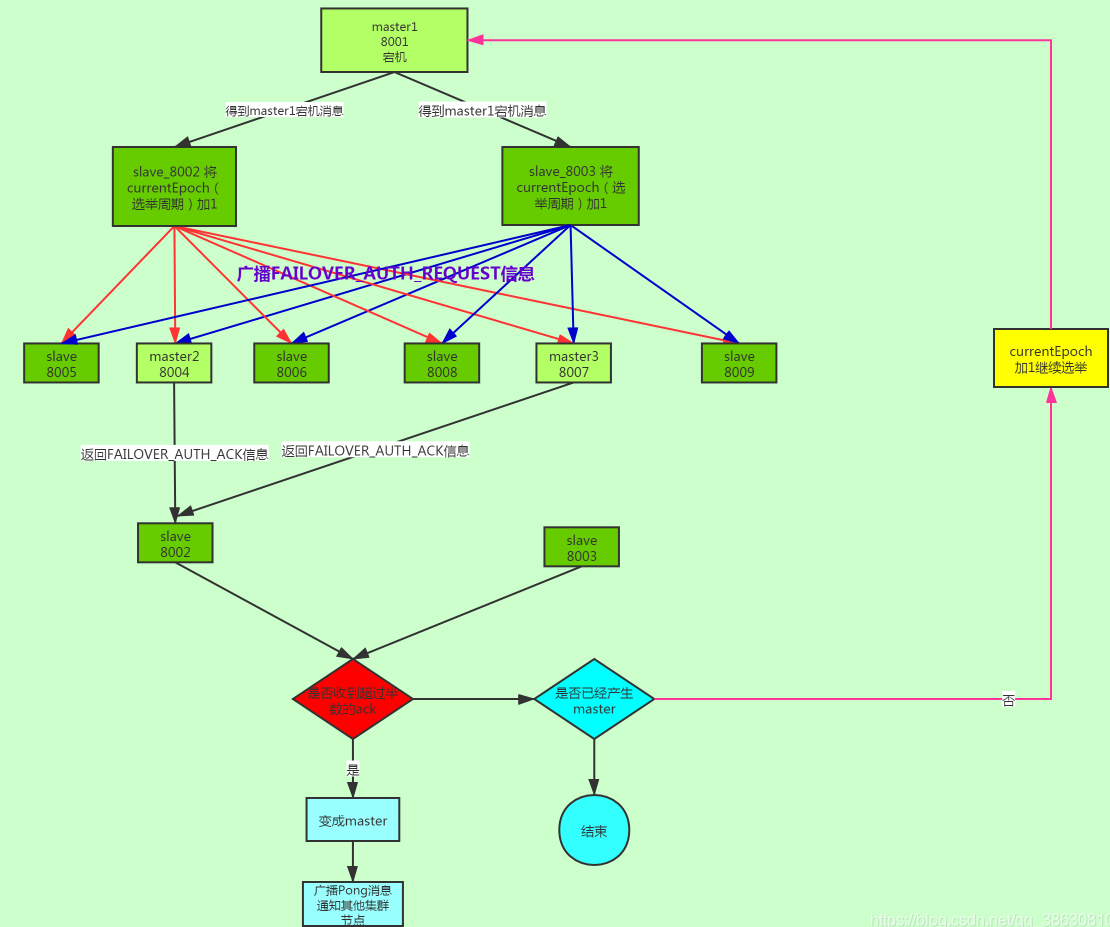
**多主多从，去中心化**：**从节点作为备用，复制主节点，不做读写操作，不提供服务**

不支持处理多个key：因为数据分散在多个节点，在数据量大高并发的情况下会影响性能；

**支持动态扩容节点**：Rerdis Cluster最大的优点之一；

**节点之间相互通信，相互选举**，不再依赖sentinel：准确来说是主节点之间相互“监督”，保证及时故障转移.

选举流程：



当slave发现自己的master变为FAIL状态时，便尝试进行Failover，以期成为新的master。由于挂掉的master可能会有多个slave，从而存在多个slave竞争成为master节点的过程， 其过程如下：

1.slave发现自己的master变为FAIL

2.将自己记录的集群currentEpoch加1，并广播FAILOVER\_AUTH\_REQUEST信息

3.其他节点收到该信息，只有master响应，判断请求者的合法性，并发送FAILOVER\_AUTH\_ACK，对每一个epoch只发送一次ack

4.尝试failover的slave收集FAILOVER\_AUTH\_ACK

5.超过半数后变成新Master

6.广播Pong通知其他集群节点。

## 4.2 持久化

RDB

简而言之，就是在不同的时间点，将redis存储的数据生成快照并存储到磁盘等介质上，可以将快照复制到其他服务器从而创建具有相同数据的服务器副本。如果系统发生故障，将会丢失最后一次创建快照之后的数据。如果数据量大，保存快照的时间会很长。

AOF

将redis执行过的所有写指令记录下来，在下次redis重新启动时，只要把这些写指令从前到后再重复执行一遍，就可以实现数据恢复了。将写命令添加到 AOF 文件（append only file）末尾。

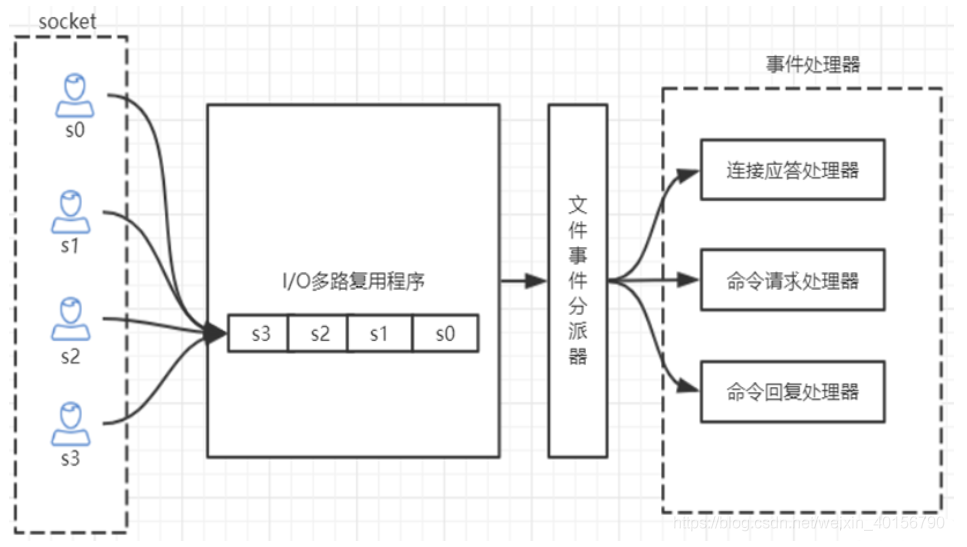
## 4.3 事件模型

Redis服务器是一个事件驱动程序，主要处理以下两类事件：

文件事件（file event）：Redis服务器通过套接字与客户端进行连接，而文件事件就是对套接字操作的抽象，可以将其理解为IO事件，Redis将产生事件套接字放入一个就绪队列中，即redisServer.aeEventLoop.fired数组，然后在aeProcessEvents会依次分派给文件事件处理器处理。Redis中文件事件包括：客户端的连接、命令请求、数据回复、连接断开等，当上述事件发生时，会造成相应的描述符可读可写，再调用相应类型的文件事件处理器。

时间事件（time event）：时间事件包含定时事件和周期性事件，Redis将其放入一个单向无序链表中，每当时间事件执行器运行时，就遍历链表，查找已经到达的时间事件，调用相应的处理器。定时事件：让一段程序在指定的时间之后执行一次，比如让程序X在当前时间30ms之后执行一次；周期事件：让一段程序每隔指定时间就执行一次，比如让程序Y每隔30ms就执行一次

io多路复用



对比多线程：

io多路复用节约资源，但它要求每个任务时间短，不能阻塞太久，并且要尽量占满线程。**因此要特别注意阻塞问题**。

内存溢出控制策略：

当所用内存达到maxmemory上限则会触发溢出控制策略。配置具体策略是maxmemory-policy参数控制，redis支持6种策略。

1）noeviction：默认的策略，拒绝写入新数据，不删任何数据，此时只响应读操作。（会返回错误信息（error）OOM command not allowed when used memory）

2）volatile-lru：根据LRU算法删除超时的键，如果没有可删除的键则回退到noeviction策略

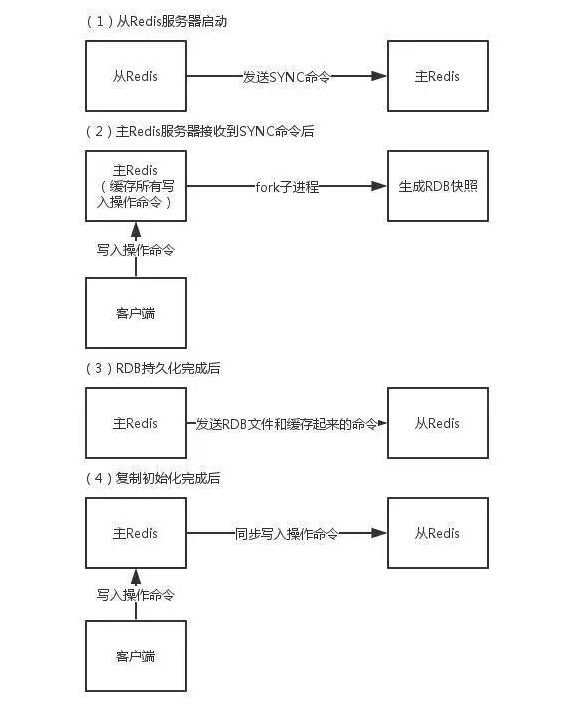
3）allkeys-lru：根据LRU算法删除键（不管键有没有设置过期时间）

4）allkeys-random：随机删除键

5）volatile-random：随机删除过期键

1. volatile-ttl：删除最近将要过期的键，没有则回退到noeviction策略

写入的场景



# 源码阅读

**使用的源码为6.0.8版本**

目录：

src: 包含用 C 编写的 Redis 实现

tests: 包含在 Tcl 中实现的单元测试

deps: 包含Redis使用的库。编译Redis所需的一切都在这个目录中，值得注意的是 deps 包含 jemalloc 的副本，它是 Linux 下 Redis 的默认分配器。

## 数据结构

### 5.1.1 string

源码所在文件 sds.h 和 sds.c

sds字符串根据字符串的长度，划分了五种结构体sdshdr5、sdshdr8、sdshdr16、sdshdr32、sdshdr64,分别对应的类型为SDS\_TYPE\_5、SDS\_TYPE\_8、SDS\_TYPE\_16、SDS\_TYPE\_32、SDS\_TYPE\_64

|  |
| --- |
| struct \_\_attribute\_\_ ((\_\_packed\_\_)) sdshdr5 {  unsigned char flags; /\* 低3位用来存储类型，高5位用来存储长度 \*/  char buf[];  };  struct \_\_attribute\_\_ ((\_\_packed\_\_)) sdshdr8 {  uint8\_t len; /\* 字符串在buf中实际占用的字节数（不包括'\0'）\*/  uint8\_t alloc; /\* 去除头长度和结束符'\0'后的总长度 \*/  unsigned char flags; /\* 低位的3个bit位用来表示结构类型，其余5个bit位未使用 \*/  char buf[];  };  struct \_\_attribute\_\_ ((\_\_packed\_\_)) sdshdr16 {  uint16\_t len; /\* 字符串在buf中实际占用的字节数（不包括'\0'）\*/  uint16\_t alloc; /\* excluding the header and null terminator \*/  unsigned char flags; /\* 低位的3个bit位用来表示结构类型，其余5个bit位未使用 \*/  char buf[];  };  ...... |

\_\_attribute\_\_ ((\_\_packed\_\_)) 告诉编译分配的是紧凑内存，而不是字节对齐的方式。

len表示字符串已使用的长度,buf长度

alloc表示字符串的容量

flags表示字符串类型标记SDS\_TYPE\_5、SDS\_TYPE\_8、SDS\_TYPE\_16、SDS\_TYPE\_32、SDS\_TYPE\_64

buf[]表示柔性数组。在分配内存的时候会指向字符串的内容

### 5.1.2 list

|  |
| --- |
| typedef struct listNode {  struct listNode \*prev; //prev指针，指向前一个节点  struct listNode \*next;//next指针，指向下一个节点  void \*value;  } listNode;  //迭代器  typedef struct listIter {  listNode \*next;  int direction; //迭代器方向  } listIter;  typedef struct list {  listNode \*head; //head指针指向链表头部  listNode \*tail; //head指针指向链表头部  void \*(\*dup)(void \*ptr); //自定义节点值的复制函数，如果不定义，默认策略的复制操作会让原链表和新链表共享同一个数据域  void (\*free)(void \*ptr);//自定义节点free操作  int (\*match)(void \*ptr, void \*key);//search操作的时候比较两个value是否相等，默认策略是比较两个指针的值  unsigned long len; //记录链表的长度，获取长度操作可以O(1)返回  } list;  //**ziplist**（特殊编码的双向链表，内存紧凑，每个节点记录上一个元素的长度）：  struct entry {  int<var> prevlen; # 前一个 entry 的字节长度  int<var> encoding; # 元素类型编码  optional byte[] content; # 元素内容  }  struct ziplist<T> {  int32 zlbytes;  int32 zltail\_offset;  int16 zllength;  T[] entries;  int8 zlend;  }  **//quicklist**（ziplist+linkedList）  typedef struct quicklistNode {  struct quicklistNode \*prev;  struct quicklistNode \*next;  unsigned char \*zl;  unsigned int sz; /\* ziplist size in bytes \*/  unsigned int count : 16; /\* count of items in ziplist \*/  unsigned int encoding : 2; /\* RAW==1 or LZF==2 \*/  unsigned int container : 2; /\* NONE==1 or ZIPLIST==2 \*/  unsigned int recompress : 1; /\* was this node previous compressed? \*/  unsigned int attempted\_compress : 1; /\* node can't compress; too small \*/  unsigned int extra : 10; /\* more bits to steal for future usage \*/  } quicklistNode;  typedef struct quicklist {  quicklistNode \*head;  quicklistNode \*tail;  unsigned long count; /\* total count of all entries in all ziplists \*/  unsigned long len; /\* number of quicklistNodes \*/  int fill : 16; /\* fill factor for individual nodes \*/  unsigned int compress : 16; /\* depth of end nodes not to compress;0=off \* /  } quicklist; |

### 5.1.3 dict

|  |
| --- |
| typedef struct dictht { //hash表结构定义，每一个dict有两个，在rehash过程中会用到两个。  dictEntry \*\*table; // 哈希表节点指针数组（俗称桶，bucket）  unsigned long size; // 指针数组的大小  unsigned long sizemask; // 指针数组的长度掩码，用于计算索引值  unsigned long used; // 哈希表现有的节点数量  } dictht;  typedef struct dictEntry {  void \*key; //键  union {  void \*val;  uint64\_t u64;  int64\_t s64;  double d;  } v; //值  struct dictEntry \*next;//在存在hash冲突的时候，用来链接具有相同hash值得key/val对，如下图所示。  } dictEntry;  typedef struct dict {  dictType \*type;  void \*privdata;  dictht ht[2]; //两个hash表  long rehashidx; /\*记录 rehash 进度的标志，值为 -1 表示 rehash 未进行\*/  unsigned long iterators; /\* 当前正在运作的安全迭代器数量\*/  } dict; |

### 5.1.4 set

|  |
| --- |
| typedef struct intset {  uint32\_t encoding;  uint32\_t length;  int8\_t contents[];  } intset; |

### 5.1.5 zset

|  |
| --- |
| /\* ZSETs use a specialized version of **Skiplists** \*/  typedef struct zskiplistNode {  sds ele;  double score;  struct zskiplistNode \* backward;  struct zskiplistLevel {  struct zskiplistNode \* forward;  unsigned long span;  } level[];  } zskiplistNode;  typedef struct zskiplist {  struct zskiplistNode \* header, \* tail;  unsigned long length;  int level;  } zskiplist;  typedef struct zset {  dict \* dict;  zskiplist \* zsl;  } zset ; |

### 5.1.6 使用总结

String，redis对于KV的操作效率很高，在实际应用中，这种应用方式也是最广泛直接的，比如多个进程共享一个数据，实现分布式锁的基本数据内容，它是二进制安全的，所以这就可以存储任意格式的数据。

hash，设计它的目的是为了高效和存储和操作数据。比如快速命中用户、商品，从而进一步访问更详细的信息。其还可以进行分段存储，这样可以利用较少的位来存储更复杂的数据。

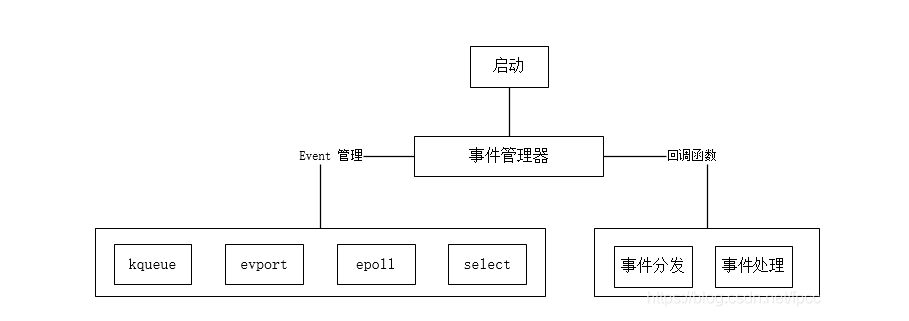
list，列表类型和普通的链表没什么区别，其实它就是链表，而且是双向的。它的应用就很广泛了，比如形成一个队列，维护一个域范围等。

set，集合，重点是可以去重。也就是相同的数据无法再次插入修改。当然支持一些集合的基本操作，如交、并集等。最典型的就画像时做某些特征的求共同部分和总体。

zset，有序集合，最典型就是权重配比，并依照配比进行操作。

## 服务端启动流程

整体框架：



不同平台使用不同的网络编程模型，不过都支持select

网络数据分为接收连接和收发数据两类，通过注册不同事件的回调函数来完成处理

启动入口为server.c main函数，只看重要方法：

1. 首先是一系列配置初始化以及配置文件载入
2. 调用initServer用来初始化服务器，创建事件循环处理结构体的实例，设置服务端处理 socket 连接事件的函数及时间事件的函数，具体工作如下：

2.1 初始化 server 服务端的各项配置属性，包括定时任务执行频率，客户端链表结构，Slave 节点链表结构等

2.2 调用 aeCreateEventLoop函数创建事件循环处理的结构体，完成初始化

2.3 调用 listenToPort函数绑定服务端 Socket 监听端口

2.4 初始化 redis 的 db 数据结构 redisDb

2.5 调用 aeCreateTimeEvent 函数指定**定时事件处理函数**为 serverCron

2.6调用aeCreateFileEvent函数指定不同类型的**连接处理函数**，如acceptTcpHandler，将事件的读处理指针进行赋值

2.7调用 aeSetBeforeSleepProc、aeSetAfterSleepProc函数设置每次事件处理之前需要进行的操作为 beforeSleep，处理后的操作为aftersleep

1. aeMain启动事件处理循环，开始接受客户端连接并处理客户端命令，主要循环调用aeProcessEvents处理事件，处理函数之前第二步已经设置过了

## 客户端启动流程

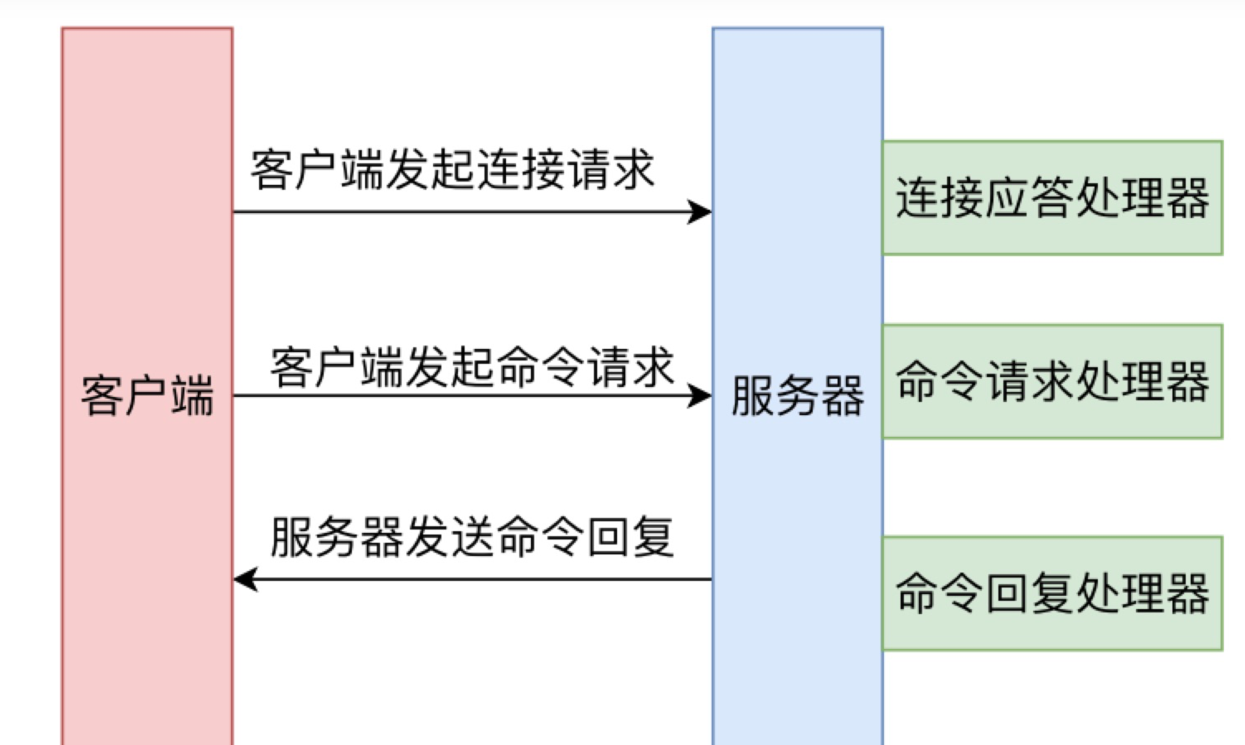
参考文件redis-cli.c：

1. 从main函数开始，首先是给config这个全局结构体设置默认值
2. 调用parseOptions处理参数，继续填充修改config，剩下的参数用于非交互模式的命令执行
3. 调用parseEnv判断是否需要验证权限
4. 根据先前输入的参数确定以哪种模式启动，如cluster、slave等。
5. 最后如果没有其他参数且不带eval命令，则先用cliConnect连接服务器，再调用repl（该函数用来向服务器端发送命令并且接收到的结果返回）进入交互模式，如果带eval命令进入非交互模式

## 连接流程

1. 上面提到客户端进入交互模式前会调用cliConnect连接服务端，参数为0代表强制重连，客户端到这里暂不深入调用关系
2. 服务端当连接请求到来时，会在事件轮询时轮询出**可读事件**，这样就会触发启动过程中设置的处理函数，例如acceptTcpHandler
3. acceptTcpHandler接着调用anetTcpAccept（底层会调用 socket 的 accept 方法)获取cfd，调用 connCreateAcceptedSocket创建 Socket，再通过acceptCommonHandler函数对新建的 Socket 进行处理
4. acceptCommonHandler将调用createClient函数创建client对象，该函数**设置了真正的读事件处理函数**readQueryFromClient，并将新建的client加入客户端链表队尾，用来维护连接，

## 命令执行流程



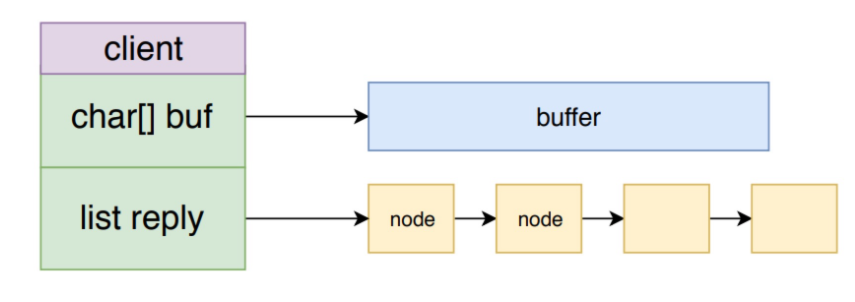
客户端

repl函数循环接收处理用户指令，首先调用linenoise函数读取用户输入的命令

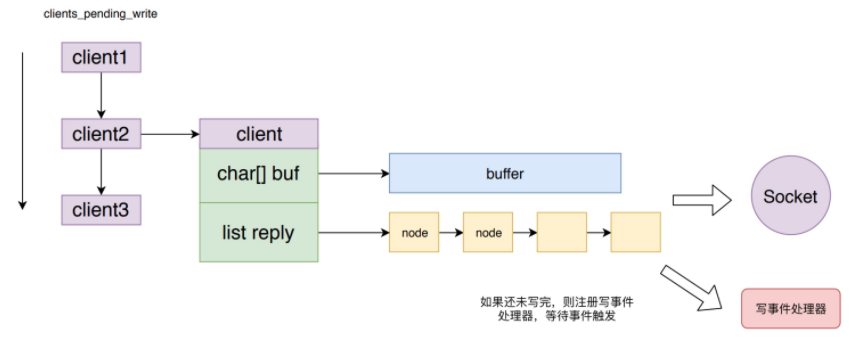
1. 再通过issueCommandRepeat，重复调用cliSendCommand发送命令给服务端
2. 最后通过cliReadReply接收服务器端返回的结果，调用cliFormatReplyRaw()函数将结果进行编码并返回

服务端

1. 当轮询到读事件后，调用 readQueryFromClient（底层调用read方法）将数据保存到client对象结构的输入缓冲区中，随即调用processInputBuffer处理缓冲区信息
2. 解析信息，生成指令，然后调用processCommandAndResetClient执行指令，在所有的 redisCommand 执行的最后，一般都会调用 addReply 方法进行结果返回，将执行结果放在client的输出缓冲区，输出缓冲区结构如下



1. 在每次调用aeProcessEvents时，都会调用beforesleep函数，进一步调用handleClientsWithPendingWrites，**将数据从缓存写到socket中**，如果没写完，调用 aeCreateFileEvent 方法来注册一个**写数据事件处理器** sendReplyToClient，等待 Redis 事件机制的再次调用：



其目的在于对于返回数据较少的客户端，不用进行写事件处理函数的注册，也不需要麻烦的关联写数据事件来触发

## 哨兵

哨兵的任务：

1. 判断节点的主观下线状态
2. 判断主节点的客观下线状态
3. 故障转移，选择从节点晋升（超时重新选择），让其他从节点同步新的主节点

哨兵进程定时任务

|  |
| --- |
| void sentinelTimer(void) {  //检测是不是TILT模式  sentinelCheckTiltCondition();  //处理所有哨兵实例  **sentinelHandleDictOfRedisInstances**(sentinel.masters);  //执行脚本队列中的任务  sentinelRunPendingScripts();  //检查终止的脚本，成功执行则删除，否则重试一定次数  sentinelCollectTerminatedScripts();  //杀死超时的脚本任务  sentinelKillTimedoutScripts();  /\* We continuously change the frequency of the Redis "timer interrupt"  \* in order to desynchronize every Sentinel from every other.  \* This non-determinism avoids that Sentinels started at the same time  \* exactly continue to stay synchronized asking to be voted at the  \* same time again and again (resulting in nobody likely winning the  \* election because of split brain voting). \*/  server.hz = CONFIG\_DEFAULT\_HZ + rand() % CONFIG\_DEFAULT\_HZ;  } |

sentinelHandleDictOfRedisInstances对每个哨兵实例调用sentinelHandleRedisInstance进行处理，**相当于哨兵进程的主体**，对于master实例，最后会做错误检测和错误迁移，将slave提升为master：

|  |
| --- |
| /\* Perform scheduled operations for the specified Redis instance. \*/  void sentinelHandleRedisInstance(sentinelRedisInstance \*ri) {  /\* ========== MONITORING HALF ============ \*/  /\* Every kind of instance \*/  sentinelReconnectInstance(ri);  sentinelSendPeriodicCommands(ri);  /\* ============== ACTING HALF ============= \*/  /\* We don't proceed with the acting half if we are in TILT mode.  \* TILT happens when we find something odd with the time, like a  \* sudden change in the clock. \*/  if (sentinel.tilt) {  if (mstime()-sentinel.tilt\_start\_time < SENTINEL\_TILT\_PERIOD) return;  sentinel.tilt = 0;  sentinelEvent(LL\_WARNING,"-tilt",NULL,"#tilt mode exited");  }  /\* Every kind of instance \*/  sentinelCheckSubjectivelyDown(ri);  /\* Masters and slaves \*/  if (ri->flags & (SRI\_MASTER|SRI\_SLAVE)) {  /\* Nothing so far. \*/  }  /\* Only masters \*/  if (ri->flags & SRI\_MASTER) {  sentinelCheckObjectivelyDown(ri);  if (sentinelStartFailoverIfNeeded(ri))  sentinelAskMasterStateToOtherSentinels(ri,SENTINEL\_ASK\_FORCED);  **sentinelFailoverStateMachine**(ri);  sentinelAskMasterStateToOtherSentinels(ri,SENTINEL\_NO\_FLAGS);  } |

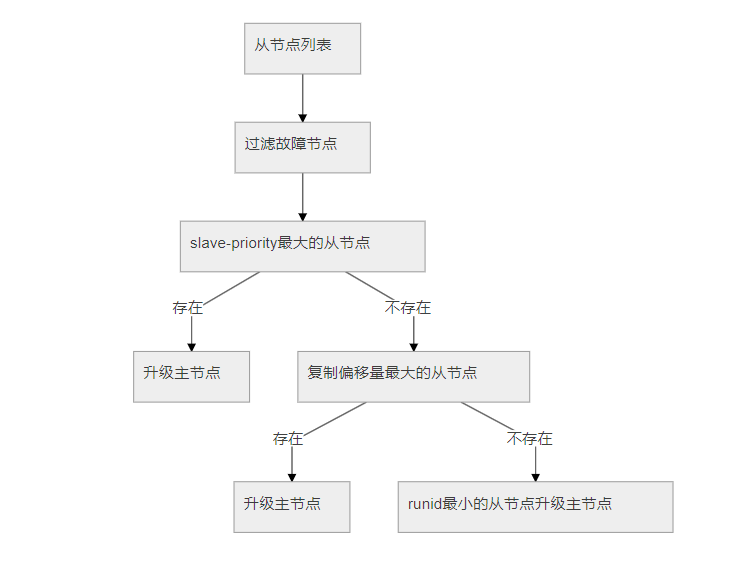
错误迁移的状态机

|  |
| --- |
| void sentinelFailoverStateMachine(sentinelRedisInstance \*ri) {  serverAssert(ri->flags & SRI\_MASTER);  if (!(ri->flags & SRI\_FAILOVER\_IN\_PROGRESS)) return;  switch(ri->failover\_state) {  case SENTINEL\_FAILOVER\_STATE\_WAIT\_START:  **sentinelFailoverWaitStart**(ri);  break;  case SENTINEL\_FAILOVER\_STATE\_SELECT\_SLAVE:  **sentinelFailoverSelectSlave**(ri);  break;  case SENTINEL\_FAILOVER\_STATE\_SEND\_SLAVEOF\_NOONE:  **sentinelFailoverSendSlaveOfNoOne**(ri);  break;  case SENTINEL\_FAILOVER\_STATE\_WAIT\_PROMOTION:  **sentinelFailoverWaitPromotion**(ri);  break;  case SENTINEL\_FAILOVER\_STATE\_RECONF\_SLAVES:  **sentinelFailoverReconfNextSlave**(ri);  break;  }  } |

哨兵对于其所监控的所有主节点，及其属下的所有从节点，都会建立两个TCP连接。一个用于发送命令，一个用于订阅其HELLO频道。而哨兵对于监控同一主节点的其他哨兵实例，只建立一个命令连接。

选举流程:

1. 选举一个Sentinel节点作为Leader
2. Sentinel Leader从redis从节点中选择一个redis节点作为主节点



## 5.7 数据库

默认情况下，客户端使用0号数据库，客户端可以使用select命令，切换其要使用的数据库。在redis.h中定义的redisClient结构，就代表redis客户端，其定义如下

|  |
| --- |
| typedef struct redisClient {  ...  redisDb \*db;//代表数据库  ...  } redisClient; |

数据库redisDb定义在redis.h中，它的定义如下：

|  |
| --- |
| typedef struct redisDb {  dict \*dict; /\*该数据库中，保存所有键值对的字典\*/  dict \*expires; /\* Timeout of keys with a timeout set \*/  ...  int id; /\* Database ID \*/  ...  } redisDb; |

常见相关接口

|  |
| --- |
| 查找键值对  robj \*lookupKey(redisDb \*db, robj \*key)  设置键值对  void dbAdd(redisDb \*db, robj \*key, robj \*val)  删除键值对  int dbDelete(redisDb \*db, robj \*key)  设置生存时间  void setExpire(redisDb \*db, robj \*key, long long when)  删除生存时间的设置  int removeExpire(redisDb \*db, robj \*key) |

AOF、RDB和复制功能对过期键的处理：

a：RDB功能处理过期键的策略如下：

执行SAVE命令或BGSAVE命令创建一个新的RDB文件时，程序会对数据库中的键进行检查，已过期的键不会被保存到新的RDB文件中；

如果服务器开启了RDB功能，那么启动Redis服务器时，将会载入RDB文件。如果服务器以主服务器模式运行，则在载人RDB文件时，程序会对文件中保存的键进行检查，过期键则会被忽略而不被载入；

如果服务器以从服务器模式运行，则在载人RDB文件时，文件中保存的所有键，不论是否过期，都会被载人到数据库中。不过，因为主从服务器在进行数据同步时，从服务器的数据库就会被清空，所以一般来讲，过期键对载人RDB文件的从服务器不会造成影响。

b：AOF功能处理过期键的策略如下：

当服务器以AOF持久化模式运行时，如果数据库中的某个键已经过期，但它还没有被惰性删除或定期删除，则AOF文件不会因为这个过期键而产生任何影响；

当过期键被惰性删除或定期删除之后，程序会向AOF文件追加一条DEL命令，显式地记录该键已被删除。这是通过propagateExpire函数实现的，而该函数会在expireIfNeeded中被调用。

在执行AOF重写的过程中，程序会对数据库中的键进行检查，已过期的键不会被保存到重写后的AOF文件中。

c：复制功能处理过期键的策略如下：

当服务器运行在复制模式下时，从服务器的过期键删除动作由主服务器控制：

主服务器在删除一个过期键之后，会显式地向所有从服务器发送一个DEL命令，告知从服务器删除这个过期键。这是通过propagateExpire函数实现的，而该函数会在expireIfNeeded中被调用。

从服务器在执行客户端发送的读命令时，即使碰到过期键也不会将过期键删除，而是继续像处理未过期的键一样来处理过期键。从服务器只有在接到主服务器发来的DEL命令之后，才会删除过期键。

通过由主服务器来控制从服务器统一地删除过期键，可以保证主从服务器数据的一致性。

# C语言接口hiredis

**主要接口**

|  |
| --- |
| /\*连接redis\*/  redisContext \*redisConnect(const char \*ip, int port);    /\*redis数据库操作 如SET GET AUTH等\*/  void \*redisCommand(redisContext \*c, const char \*format, ...);  /\*释放redisCommand返回得到的结果结构体\*/  void freeReplyObject(void \*reply);  /\*断开连接\*/  void redisFree(redisContext \*c); |

# Lua脚本

在 Redis 中，执行 Lua 语言是原子性的，也就说 Redis 执行 Lua 的时候是不会被中断的，具备原子性，这个特性有助于 Redis 对并发数据一致性的支持。

Redis 支持两种方法运行脚本，一种是直接输入一些 Lua 语言的程序代码；另外一种是将 Lua 语言编写成文件。

在实际应用中，一些简单的脚本可以采取第一种方式，对于有一定逻辑的一般采用第二种方式。

命令格式为：

EVAL luascript numkeys key [key ...] arg [arg ...]

其中：

EVAL  命令的关键字。

luascript Lua 脚本。

numkeys  指定的Lua脚本需要处理键的数量，其实就是 key数组的长度。

key 传递给Lua脚本零到多个键，空格隔开，在Lua 脚本中通过 KEYS[INDEX]来获取对应的值，其中1 <= INDEX <= numkeys。

arg是传递给脚本的零到多个附加参数，空格隔开，在Lua脚本中通过ARGV[INDEX]来获取对应的值，其中1 <= INDEX <= numkeys。

# 经典问题

如何用Redis高效实现12306的复杂售票业务

新浪微博突发事件如何做好Redis缓存的高可用

高并发场景缓存穿透&失效&雪崩如何解决

Redis高并发场景热点缓存如何重建

Redis缓存与数据库双写不一致如何解决

Redis底层ZSet跳表是如何设计与实现的

Redis底层ZSet实现压缩列表和跳表如何选择

Redis 6.0 多线程模型比单线程优化在哪里了

下一步计划：

**查看其他公司怎么使用redis，应用场景和使用技巧**