

Redis基础

什么是Redis

Redis是Remote Dictionary Service，是一种基于键值对的**NoSQL**数据库

- 支持多种数据结构，如string、hash、list、set、zset、bitmaps、hyperloglog、geo等
- 所有数据存放在内存中，读写性能非常出色
- 可以将所有数据持久化到硬盘上，发生断电或机器故障时，内存中的数据不会丢失
- 还提供了键过期、发布订阅、事务、流水线、Lua 脚本等附加功能

Redis可以用来做什么

1. 缓存

- 由于 Redis 的数据存储在内存中，所以读写速度非常快，远超基于磁盘存储的数据库。使用 Redis 缓存可以极大地提高应用的响应速度和吞吐量。

2. 排行榜 / 计数器

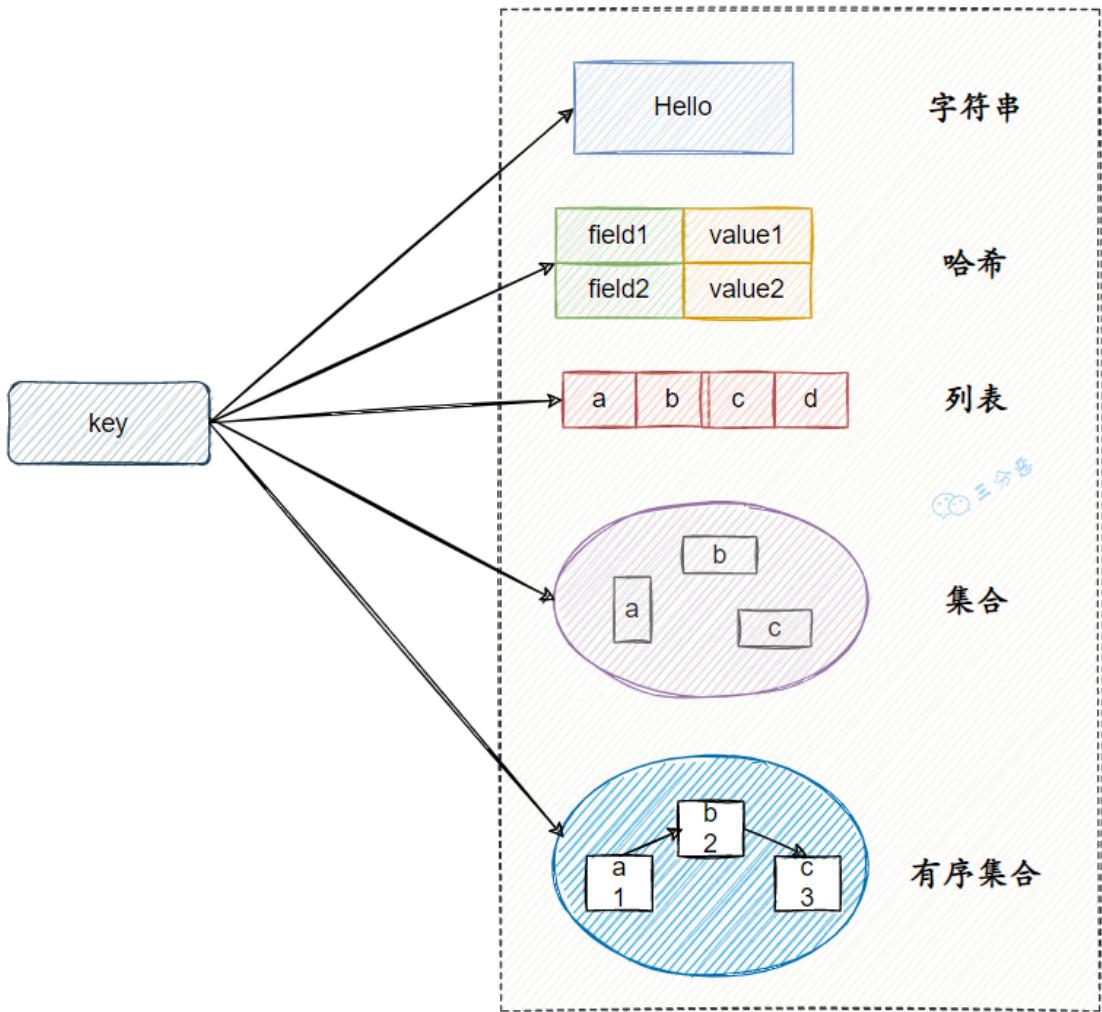
- Redis 的 ZSet 非常适合用来实现排行榜的功能，可以根据 score（分值）进行排序，实时展示用户的活跃度。
- 同时 Redis 的原子递增操作可以用来实现计数器功能

3. 分布式锁

- 可以实现分布式锁，用来控制跨多个进程的资源访问

Redis有哪些数据类型

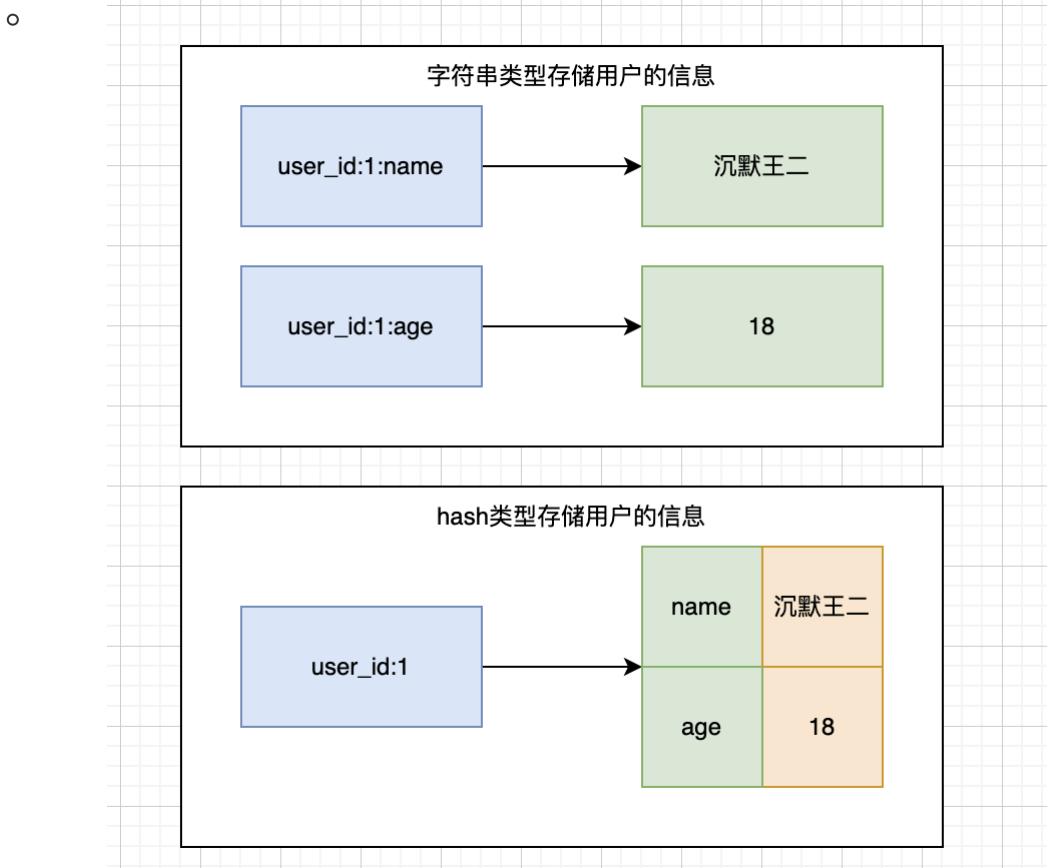
Redis一共有八种数据类型



1. string

2. hash

- 格式: Key => ({field1, value1}, {field2, value2})
- 什么时候使用hash类型而不是用string类型存储



- 使用 hash 比使用 string 更便于进行序列化，我们可以将一个用户对象序列化，然后作为一个 value 存储在 Redis 中，存取更加便捷。

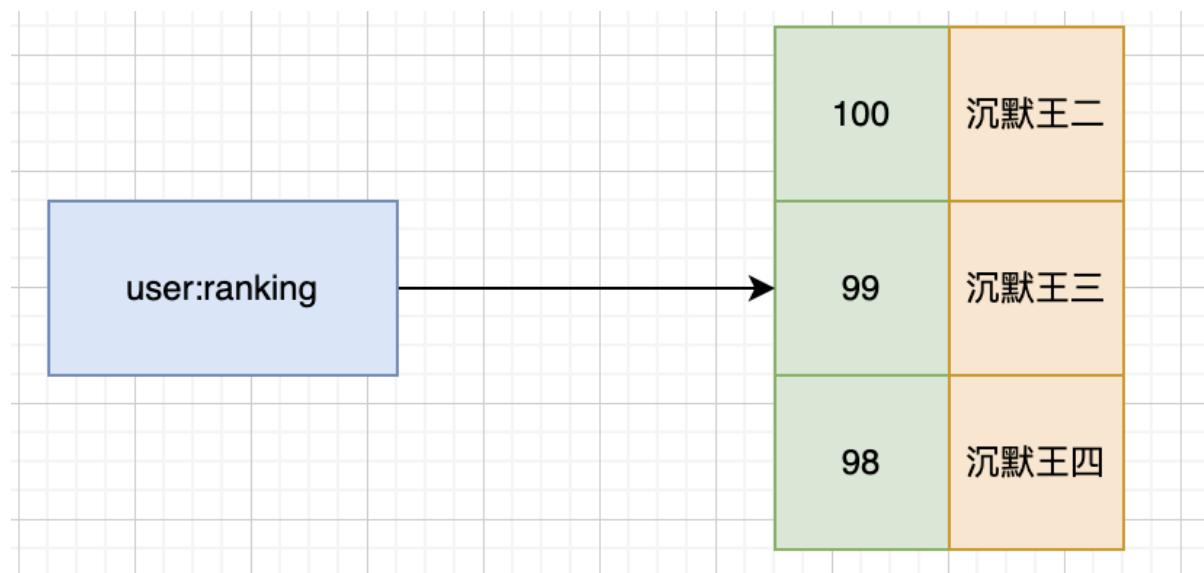
3. list

- list 是一个简单的字符串列表，按照插入顺序排序。可以添加一个元素到列表的头部（左边）或者尾部（右边）。

4. set

- Set 是一个无序集合，元素是唯一的，不允许重复。

5. zset



- Zset 是有序集合，比 set 多了一个排序属性 score。

6. bitmap

7. HyperLogLog

Redis为什么快

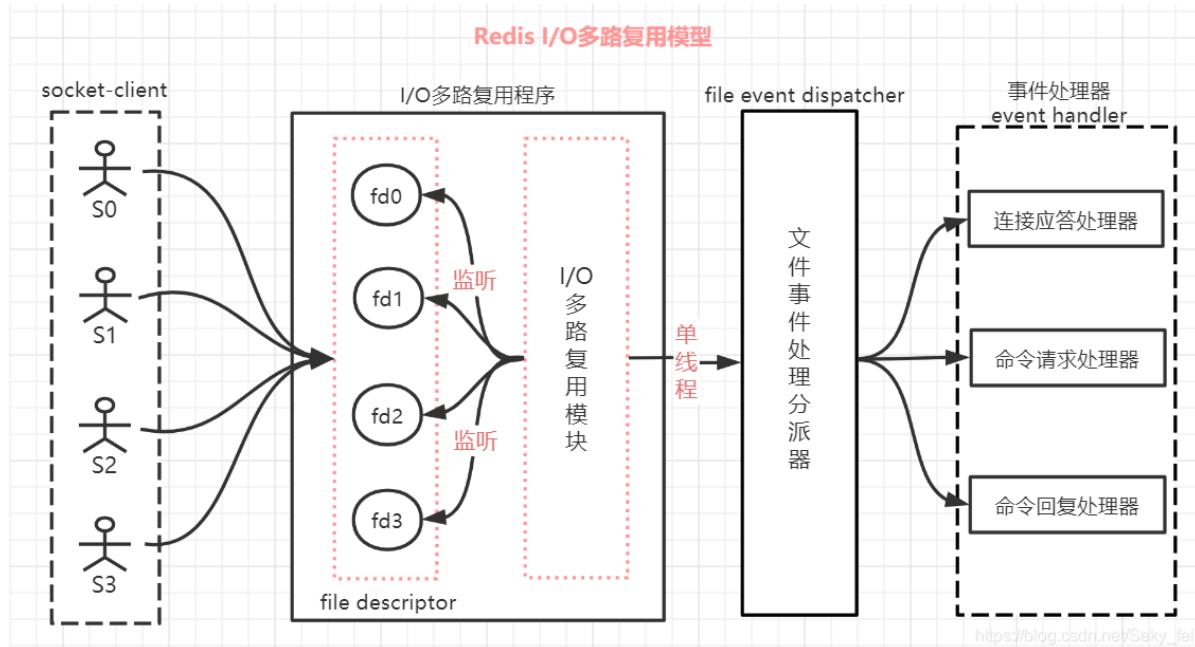
1. 基于内存的数据存储

- 将数据存储在内存当中，使得数据的读写操作避开了磁盘 I/O

2. 单线程模型

- 使用单线程模型来处理客户端的请求，这意味着在任何时刻只有一个命令在执行。这样就避免了线程切换和锁竞争带来的消耗

3. IO多路复用

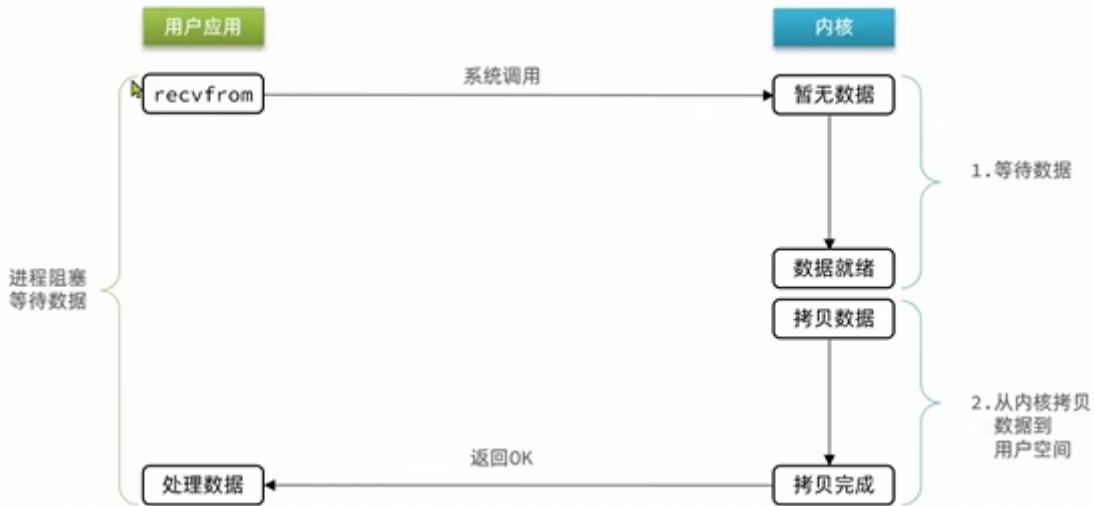


- 基于 Linux 的 select/epoll 机制。该机制允许内核中同时存在多个监听套接字和已连接套接字，内核会一直监听这些套接字上的连接请求或者数据请求，一旦有请求到达，就会交给 Redis 处理，就实现了所谓的 Redis 单个线程处理多个 IO 读写的请求。

4. 高效的数据结构

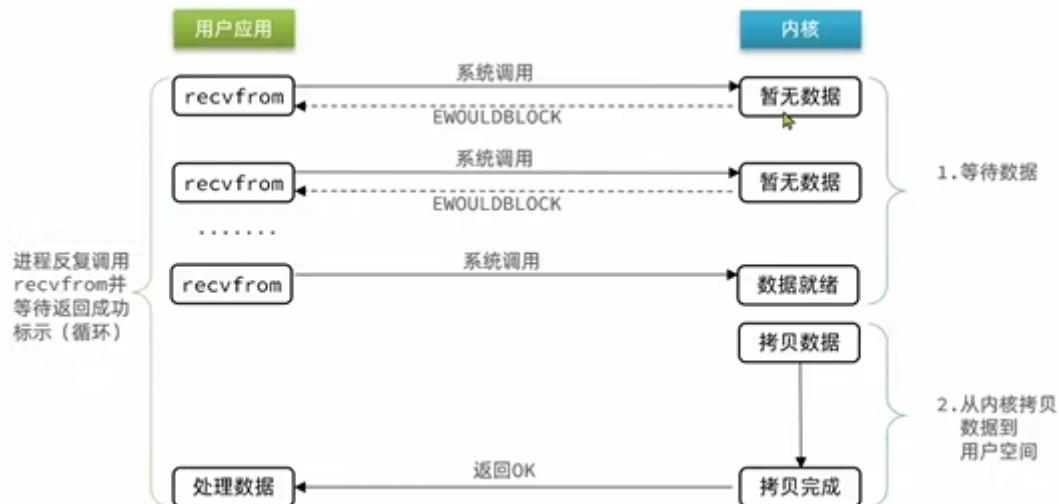
什么是阻塞IO

阻塞IO就是等待数据就绪（从磁盘拷贝到用户的缓冲区）和读取数据（从用户的缓冲区）两个阶段都必须阻塞等待



什么是非阻塞IO

非阻塞IO的recvfrom操作会立即返回结果而不是阻塞用户进程，第二阶段数据拷贝的时候还是阻塞的



能说一下IO多路复用吗

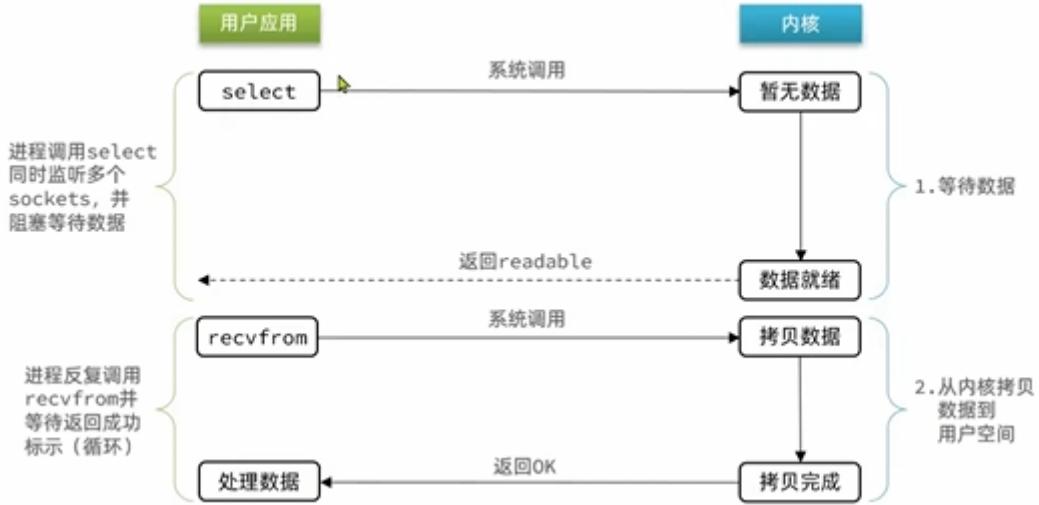
无论是阻塞IO还是非阻塞IO，用户应用在第一阶段都需要调用recvfrom来获取数据，差别在于无数据时的处理方案：

- 阻塞IO阻塞等待数据
- 非阻塞IO使CPU空转

如果服务端处理客户端Socket请求时，在单线程情况下，只能一次处理每一个Socket，如果正在处理的Socket恰好未就绪，线程就会被阻塞，哪怕其它的Socket就绪了也必须等待。

文件描述符FD：是一个从0开始递增的无符号整数，用来关联Linux中的一个文件。在Linux中，一切皆文件，例如常规文件、视频、硬件设备等，当然也包括网络套接字Socket

IO多路复用：利用单个线程来同时监听多个FD，并在某个FD可读、可写时得到通知，从而避免无效的等待，充分利用CPU资源



select、poll和epoll有什么区别

IO 多路复用是一种高效管理多个 IO 事件的技术，通过单线程监控多个文件描述符 (fd)，实现高并发的 IO 操作。

select和poll只会通知用户进程有FD就绪，但不确定具体是哪个FD，需要用户进程逐个遍历FD来确认
epoll则会在通知用户进程FD就绪的同时，把已就绪的FD写入用户空间

select

select是Linux中最早的I/O多路复用实现方案：

```

// 定义类型别名 __fd_mask, 本质是 long int
typedef long int __fd_mask;

/* fd_set 记录要监听的fd集合，及其对应状态 */
typedef struct {
    // fds_bits是long类型数组，长度为 1024/32 = 32
    // 共1024个bit位，每个bit位代表一个fd，0代表未就绪，1代表就绪
    __fd_mask fds_bits[__FD_SETSIZE / __NFBITS];
    // ...
} fd_set;

// select函数，用于监听多个fd的集合
int select(
    int nfds, // 要监视的fd_set的最大fd + 1
    fd_set *readfds, // 要监听读事件的fd集合
    fd_set *writefds, // 要监听写事件的fd集合
    fd_set *exceptfds, // // 要监听异常事件的fd集合
    // 超时时间, null-永不超时; 0-不阻塞等待; 大于0-固定等待时间
    struct timeval *timeout
);

```



- 需要将整个fd_set从用户空间拷贝到内核空间，select结束后还要再次拷贝回用户空间
- select无法直接得知哪个fd就绪，需要遍历整个fd_set
- fd_set监听的fd数量不能超过1024

poll

IO流程：

- ① 创建pollfd数组，向其中添加关注的fd信息，数组大小自定义
- ② 调用poll函数，将pollfd数组拷贝到内核空间，转链表存储，无上限
- ③ 内核遍历fd，判断是否就绪
- ④ 数据就绪或超时后，拷贝pollfd数组到用户空间，返回就绪fd数量n
- ⑤ 用户进程判断n是否大于0
- ⑥ 大于0则遍历pollfd数组，找到就绪的fd

```
// pollfd 中的事件类型
#define POLLIN    //可读事件
#define POLLOUT   //可写事件
#define POLLERR   //错误事件
#define POLLNVAL  //fd未打开

// pollfd结构
struct pollfd {
    int fd;           /* 要监听的fd */
    short int events; /* 要监听的事件类型：读、写、异常 */
    short int revents; /* 实际发生的事件类型 */
};

// poll函数
int poll(
    struct pollfd *fds, // pollfd数组，可以自定义大小
    nfds_t nfd, // 数组元素个数
    int timeout // 超时时间
);
```

- poll模式对select模式做了简单改进，但性能提升不影响
- 使用动态数组管理fd，监听数量无上限（唯一的改进）
- 监听的FD越多，每次遍历消耗的时间也越久，性能反而会下降

epoll

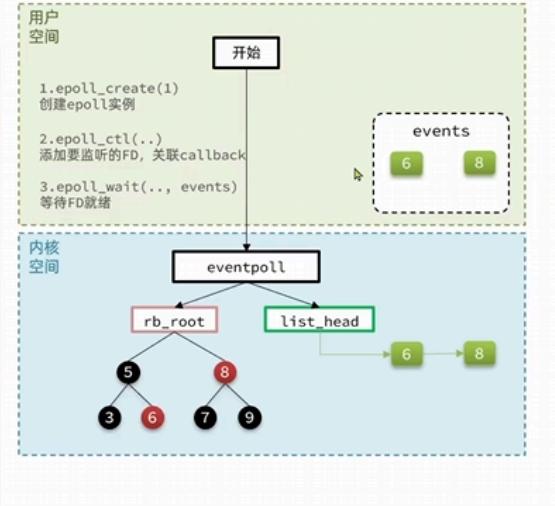
epoll模式是对select和poll的改进，它提供了三个函数：

```
struct eventpoll {
    ...
    struct rb_root rbr; // 一颗红黑树，记录要监听的FD
    struct list_head rdlist; // 一个链表，记录就绪的FD
    ...
};

// 1.会在内核创建eventpoll结构体，返回对应的句柄epfd
int epoll_create(int size);

// 2.将一个FD添加到epoll的红黑树中，并设置ep_poll_callback
// callback触发时，就把对应的FD加入到rdlist这个就绪列表中
int epoll_ctl(
    int epfd, // epoll实例的句柄
    int op, // 要执行的操作，包括：ADD、MOD、DEL
    int fd, // 要监听的FD
    struct epoll_event *event // 要监听的事件类型：读、写、异常等
);

// 3.检查rdlist列表是否为空，不为空则返回就绪的FD的数量
int epoll_wait(
    int epfd, // eventpoll实例的句柄
    struct epoll_event *events, // 空event数组，用于接收就绪的FD
    int maxevents, // events数组的最大长度
    int timeout // 超时时间，-1用不超时；0不阻塞；大于0为阻塞时间
);
```



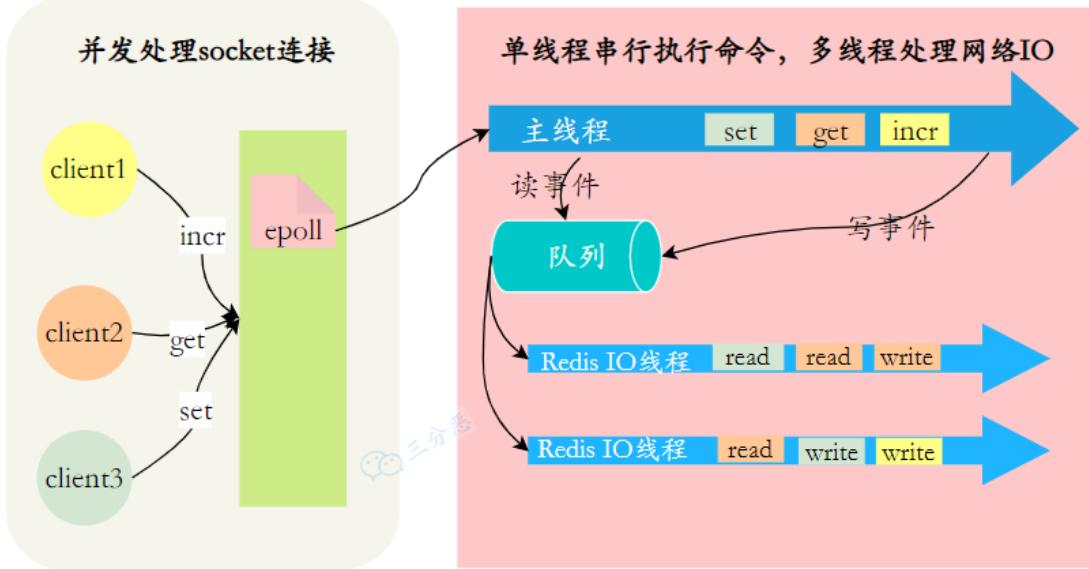
- 内核空间只拷贝就绪的FD到用户空间的指定位置
- 用户空间只需遍历就绪的fd，不用遍历所有FD

这样，整个过程只在进行 select、poll、epoll 这些调用的时候才会阻塞，收发客户消息是不会阻塞的，整个进程或者线程就被充分利用起来，这就是事件驱动，所谓的 reactor 模式。

Redis6.0使用多线程是怎么回事

单线程模型意味着 Redis 在大量 IO 请求时，无法充分利用多核 CPU 的优势。

Redis 6.0以前，网络 I/O 由主线程处理 → 主线程既要处理命令，又要等待 socket 读写，性能受限



- Redis主线程监听epoll事件
- Redis IO线程池（多个Redis IO线程）负责读取socket数据，并放到主线程的队列中
- Redis主线程（仍然是单线程，保证原子性）解析请求，并执行命令，将执行结果写回队列
- 响应客户端Redis IO 线程负责将队列中的结果写入（write）socket

SADD操作的时间复杂度是多少

- Redis 的 Set 底层使用哈希表（dict）实现。
- 插入元素到哈希表的时间复杂度是 $O(1)$ ，因为哈希表直接计算哈希值，然后插入到对应的 bucket 中，不需要遍历整个集合。
- 多个元素插入需要 $O(N)$ ，因为需要执行 N 次 $O(1)$ 的插入操作。

单线程Redis的QPS是多少

Redis 的 QPS (Queries Per Second, 每秒查询率) 受多种因素影响，包括硬件配置（如 CPU、内存、网络带宽）、数据模型、命令类型、网络延迟等。

根据官方的基准测试，一个普通服务器的 Redis 实例通常可以达到**每秒数万到几十万的 QPS**。

可以通过 `redis-benchmark` 命令进行基准测试：

```
1 | redis-benchmark -h 127.0.0.1 -p 6379 -c 50 -n 10000
```

- `-h`：指定 Redis 服务器的地址，默认是 127.0.0.1。
- `-p`：指定 Redis 服务器的端口，默认是 6379。
- `-c`：并发连接数，即同时有多少个客户端在进行测试。
- `-n`：请求总数，即测试过程中总共要执行多少个请求。

持久化

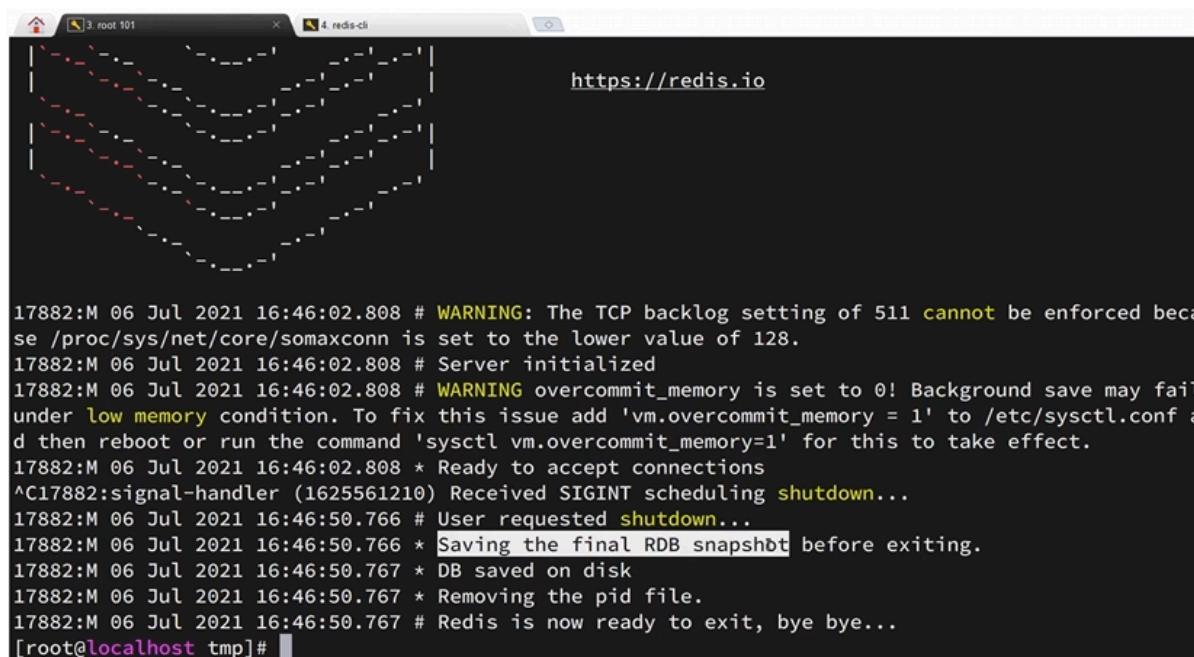
什么是RDB

RDB全称Redis Database Backup file (Redis数据备份文件)，也被叫做Redis数据快照。

简单来说就是把内存中的所有数据都记录到磁盘中，当Redis实例故障重启后，从磁盘读取快照文件，恢复数据。

```
[root@localhost ~]# redis-cli
127.0.0.1:6379> save  #由Redis主进程来执行RDB，会阻塞所有命令
ok
127.0.0.1:6379> bgsave #开启子进程执行RDB，避免主进程受到影响
Background saving started
```

- Redis主动停机时会自动执行一次RDB



```
https://redis.io

17882:M 06 Jul 2021 16:46:02.808 # WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net/core/somaxconn is set to the lower value of 128.
17882:M 06 Jul 2021 16:46:02.808 # Server initialized
17882:M 06 Jul 2021 16:46:02.808 # WARNING overcommit_memory is set to 0! Background save may fail under low memory condition. To fix this issue add 'vm.overcommit_memory = 1' to /etc/sysctl.conf and then reboot or run the command 'sysctl vm.overcommit_memory=1' for this to take effect.
17882:M 06 Jul 2021 16:46:02.808 * Ready to accept connections
^C17882:signal-handler(1625561210) Received SIGINT scheduling shutdown...
17882:M 06 Jul 2021 16:46:50.766 # User requested shutdown...
17882:M 06 Jul 2021 16:46:50.766 * Saving the final RDB snapshot before exiting.
17882:M 06 Jul 2021 16:46:50.767 * DB saved on disk
17882:M 06 Jul 2021 16:46:50.767 * Removing the pid file.
17882:M 06 Jul 2021 16:46:50.767 # Redis is now ready to exit, bye bye...
[root@localhost tmp]#
```

- Redis内部有触发RDB的机制可以在redis.conf文件中找到，格式如下：

```
# 900秒内，如果至少有1个key被修改，则执行bgsave ， 如果是save "" 则表示禁用RDB
save 900 1
save 300 10
save 60 10000
```

- RDB的其它配置也可以在redis.conf文件中设置：

```
# 是否压缩 ,建议不开启，压缩也会消耗cpu，磁盘的话不值钱
rdbcompression yes

# RDB文件名称
dbfilename dump.rdb

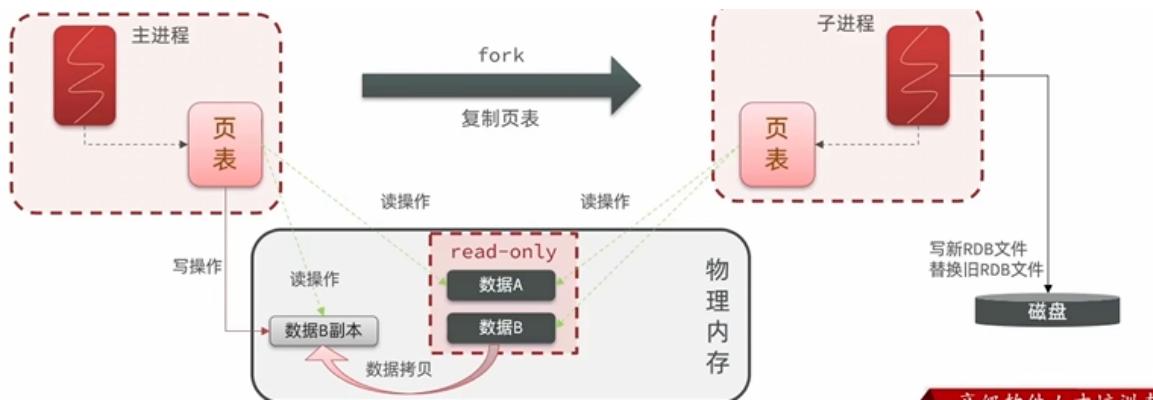
# 文件保存的路径目录
dir ./
```

RDB在什么场景下会自动触发

1. redis.conf文件中的配置
2. 当Redis服务器通过SHUTDOWN命令正常关闭时，如果没有禁用RDB持久化，Redis会自动执行一次RDB持久化，确保数据在下次启动时能够恢复
3. 在Redis复制场景中，当一个Redis实例被配置为从节点并且与主节点建立连接时，它可能会根据配置接收主节点的RDB文件来初始化数据集。这个过程中，主节点会在后台自动触发RDB持久化，然后将生成的RDB文件发送给从节点。

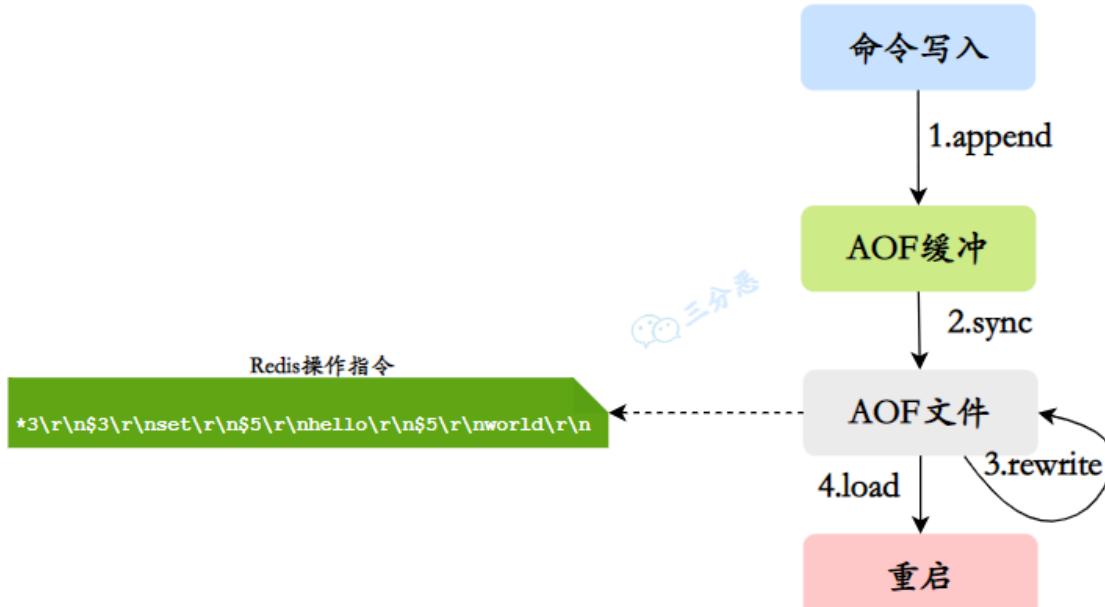
bgsave原理是什么

bgsave开始时会fork主进程得到子进程，子进程共享主进程的内存数据，完成fork后读取内存数据并写入RDB文件



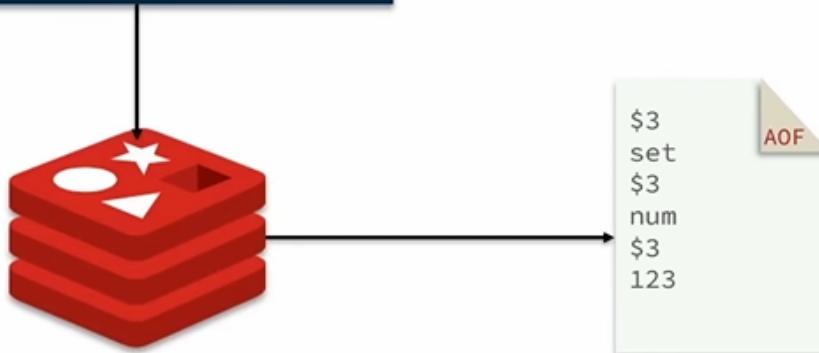
- 主进程将自己的页表（虚拟内存到物理内存的映射）拷贝给子进程
- 子进程异步写入RDB文件的过程中主进程有可能会修改数据
 - 需要写入时，必须先对原始数据进行拷贝，在拷贝上写

什么是AOF



AOF全称为Append Only File (追加文件) , Redis处理的每一个写命令都会记录在AOF文件, 可以看作是命令日志文件。

```
[root@localhost redis-6.2.4]# redis-cli  
127.0.0.1:6379> set num 123  
OK
```



- AOF默认是关闭的, 需要修改redis.conf配置文件来开启AOF

```
# 是否开启AOF功能, 默认是no  
appendonly yes  
# AOF文件的名称  
appendfilename "appendonly.aof"
```

- AOF的命令记录的频率也可以通过redis.conf文件来配置

```
# 表示每执行一次写命令, 立即记录到AOF文件  
appendfsync always  
# 写命令执行完先放入AOF缓冲区, 然后表示每隔1秒将缓冲区数据写到AOF文件, 是默认方案  
appendfsync everysec  
# 写命令执行完先放入AOF缓冲区, 由操作系统决定何时将缓冲区内容写回磁盘  
appendfsync no
```

配置项	刷盘时机	优点	缺点
Always	同步刷盘	可靠性高, 几乎不丢数据	性能影响大
everysec	每秒刷盘	性能适中	最多丢失1秒数据
no	操作系统控制	性能最好	可靠性较差, 可能丢失大量数据

- 因为是记录命令, AOF文件会比RDB文件大得多, 而且AOF会记录对同一个key的多次写操作, 但只有最后一次写操作才有意义。通过执行bgrewriteaof命令, 可以让AOF文件执行重写功能, 用最少的命令达到相同效果。



- Redis也会在触发阈值时自动去重写AOF文件, 阈值也可以在redis.conf中配置

```
# AOF文件比上次文件 增长超过多少百分比则触发重写  
auto-aof-rewrite-percentage 100  
# AOF文件体积最小多大以上才触发重写  
auto-aof-rewrite-min-size 64mb
```

AOF重写期间命令可能会写入两次，会造成什么影响

AOF 重写期间，Redis 会将新的写命令同时写入旧的 AOF 文件和重写缓冲区。这样会带来额外的磁盘开销。

但可以防止在 AOF 重写尚未完成时，Redis 发生崩溃，导致数据丢失。即使重写失败，旧的 AOF 文件仍然是完整的。

当重写完成后，会通过原子操作将新的 AOF 文件替换旧的 AOF 文件。

RDB和AOF各自有什么优缺点

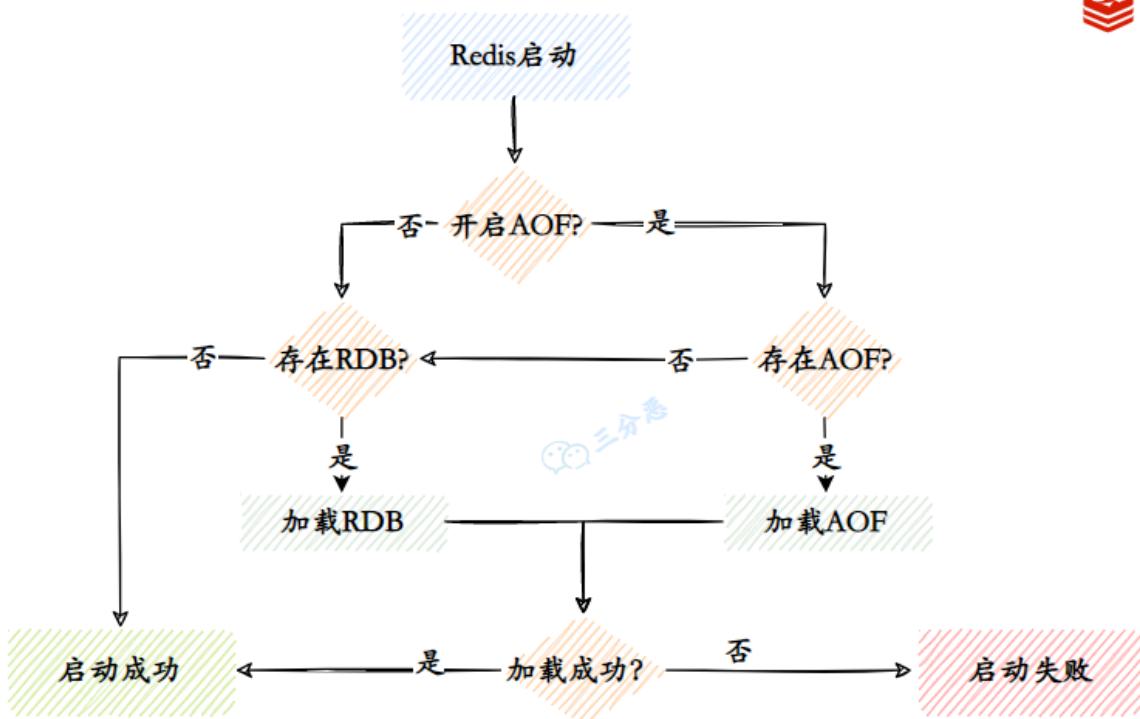
RDB非常适合备份数据，比如在夜间进行备份，然后将 RDB 文件复制到远程服务器。但可能会丢失最后一次持久化后的数据。

AOF最大优点是灵活，实时性好，可以设置不同的 fsync 策略，如每秒同步一次，每次写入命令就同步，或者完全由操作系统来决定何时同步。但 AOF 文件往往比较大，恢复速度慢，因为它记录了每个写操作。

RDB和AOF如何选择

	RDB	AOF
持久化方式	定时对整个内存做快照	记录每一次执行的命令
数据完整性	不完整，两次备份之间会丢失	相对完整，取决于刷盘策略
文件大小	会有压缩，文件体积小	记录命令，文件体积很大
宕机恢复速度	很快	慢
数据恢复优先级	低，因为数据完整性不如AOF	高，因为数据完整性更高
系统资源占用	高，大量CPU和内存消耗	低，主要是磁盘IO资源 但AOF重写时会占用大量CPU和内存资源
使用场景	可以容忍数分钟的数据丢失，追求更快的启动速度	对数据安全性要求较高常见

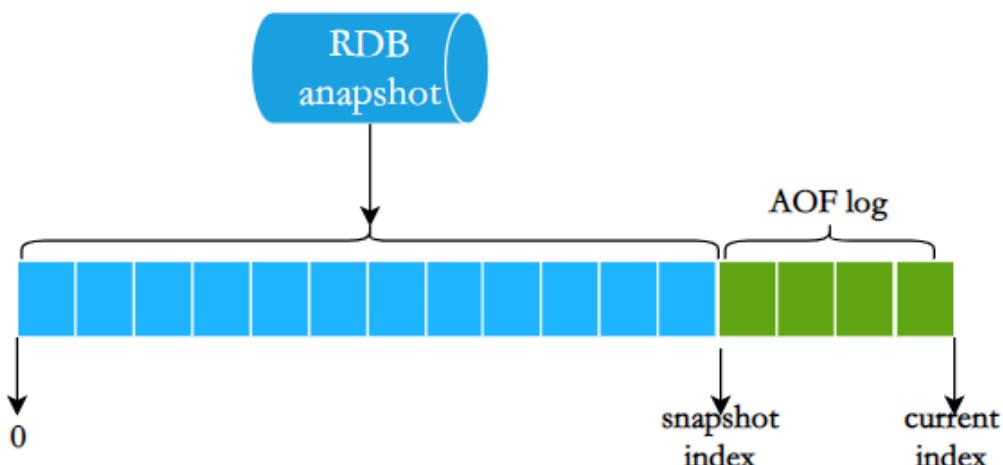
Redis的数据恢复



1. AOF 开启且存在 AOF 文件时，优先加载 AOF 文件。
2. AOF 关闭或者 AOF 文件不存在时，加载 RDB 文件。

Redis4.0的混合持久化了解吗

在 Redis 4.0 版本中，混合持久化模式会在 AOF 重写的时候同时生成一份 RDB 快照，然后将这份快照作为 AOF 文件的一部分，最后再附加新的写入命令。



这样，当需要恢复数据时，Redis 先加载 RDB 文件来恢复到快照时刻的状态，然后应用 RDB 之后记录的 AOF 命令来恢复之后的数据更改，既快又可靠。

高可用

redis的部署方式有哪些

除了单机部署外，Redis还可以通过主从复制、哨兵模式和集群模式来实现高可用。

主从复制

- 允许一个 Redis 服务器（主节点）将数据复制到一个或多个 Redis 服务器（从节点）。这种方式可以实现读写分离，适合读多写少的场景

哨兵模式

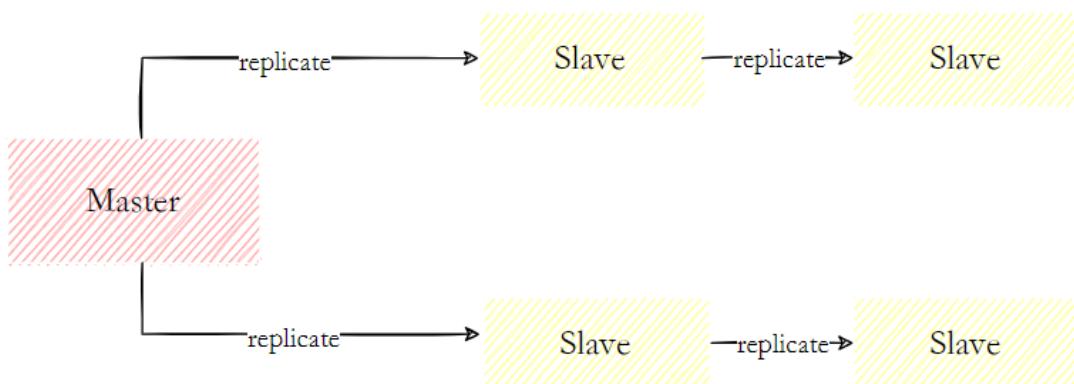
- 用于监控主节点和从节点的状态，实现自动故障转移。
- 如果主节点发生故障，哨兵可以自动将一个从节点升级为新的主节点

集群模式

- Redis 集群通过分片的方式存储数据，每个节点存储数据的一部分，用户请求可以并行处理。
- 集群模式支持自动分区、故障转移，并且可以在不停机的情况下进行节点增加或删除。

主从复制了解吗

Redis 主从复制



主节点负责处理所有的写操作，并将这些操作异步复制到从节点。从节点主要用于读取操作，以分担主节点的压力和提高读性能。

作用：

1. 数据冗余：

主从复制实现了数据的热备份，是持久化之外的一种数据冗余方式。

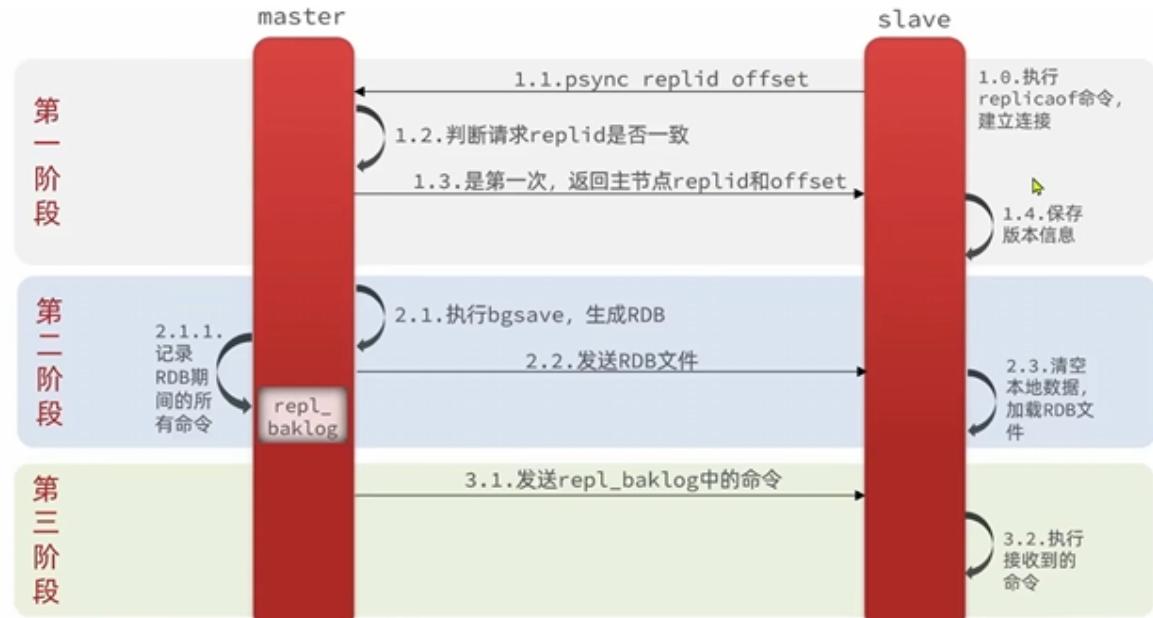
2. 故障恢复：

- 如果主节点挂掉了，可以将一个从节点提升为主节点，从而实现故障的快速恢复。
- 如果从节点挂掉了，主节点不受影响，但应该尽快修复并重启挂掉的从节点，使其重新加入集群并从主节点同步数据。

3. 负载均衡：在主从复制的基础上，配合读写分离，可以由主节点提供写服务，由从节点提供读服务（即写 Redis 时连接主节点，读 Redis 时连接从节点），分担服务器负载。

Redis数据同步原理

主从第一次同步是全量同步：



但如果 slave 重启后同步，则执行增量同步：

主从第一次同步是全量同步，但如果 slave 重启后同步，则执行增量同步



`repl_baklog` 大小有上限，写满后会覆盖最早的数据。如果 slave 断开时间过久，导致尚未备份的数据被覆盖，则无法基于 log 做增量同步，只能再次全量同步

master如何判断slave是不是第一次来同步数据

Replication Id：简称 `replid`，是数据集的标记，`id`一致则说明是同一数据集。每一个 `master` 都有唯一的 `replid`，`slave` 则会继承 `master` 节点的 `replid`

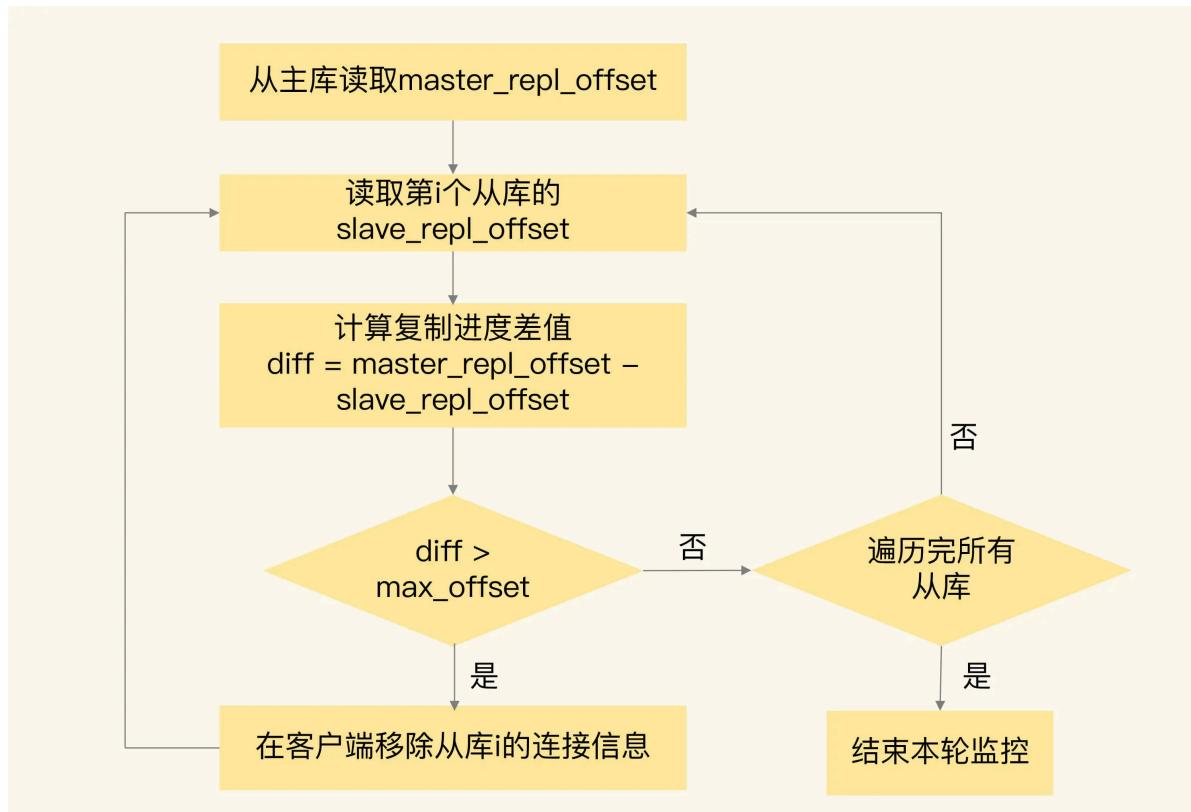
offset：偏移量，随着记录在 `repl_baklog` 中的数据增多而增大，`slave` 完成同步时也会记录当前同步的 `offset`。如果 `slave` 的 `offset` 小于 `master` 的 `offset`，说明 `slave` 数据落后于 `master`，需要更新。

master 通过 `replid` 是否一致来判断是否是第一次同步

主从复制出现数据不一致怎么办

Redis采用异步复制，主库的写入不会立即同步到从库，而是异步进行。

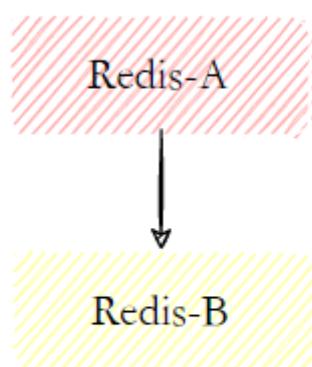
解决方法：



- 通过 Redis 的 `INFO replication` 命令监控主从节点的复制进度，及时发现和处理复制延迟。
- 获取主节点的 `master_repl_offset` 和从节点的 `slave_repl_offset`，计算两者的差值。如果差值超过预设的阈值，采取措施（如停止从节点的数据读取）以减少读到不一致数据的情况。

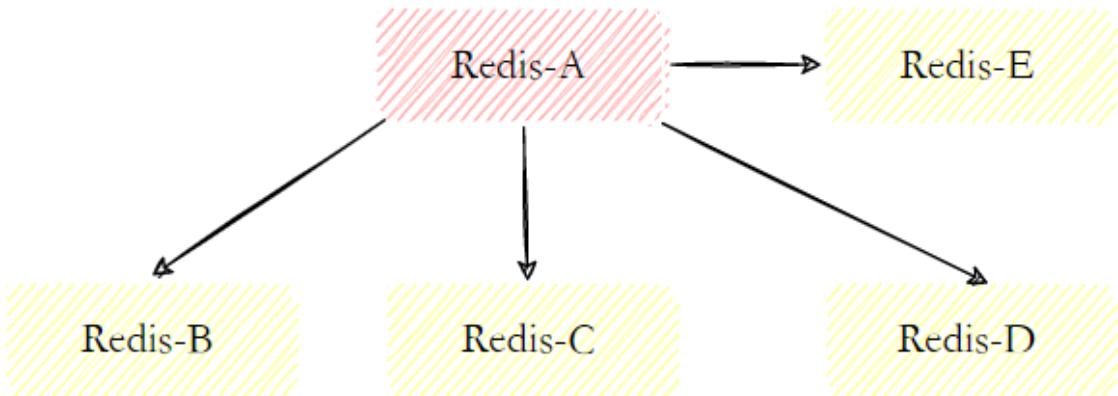
Redis主从有几种常见的拓扑结构

1. 一主一从

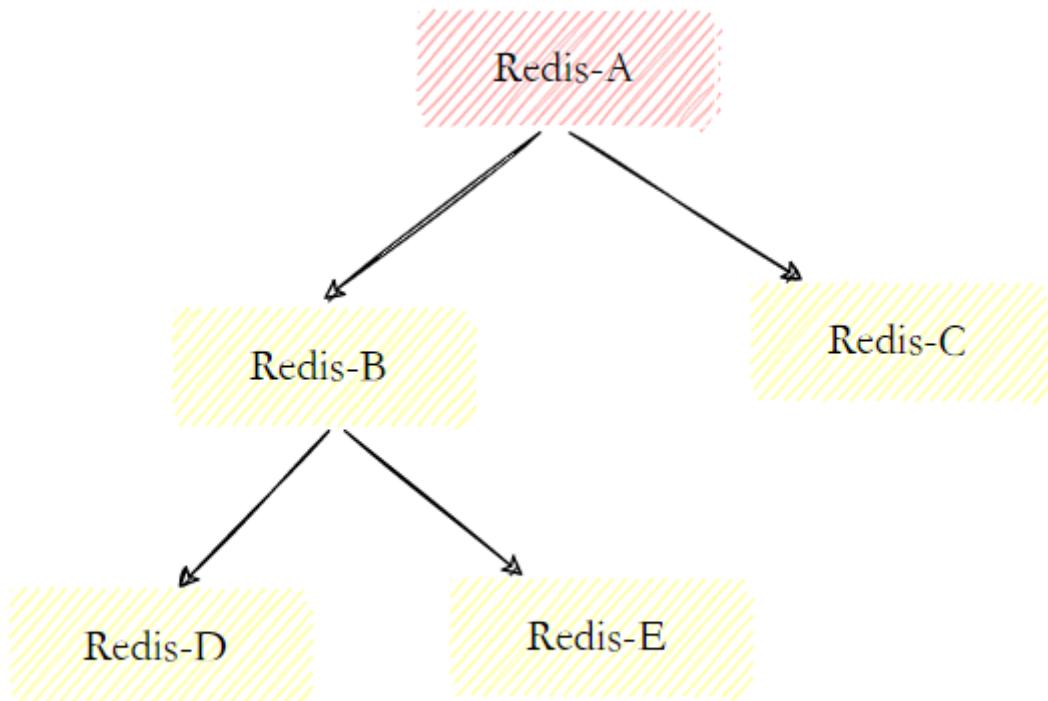


一主一从结构

2. 一主多从



一主多从（星形）结构



树状主从结构

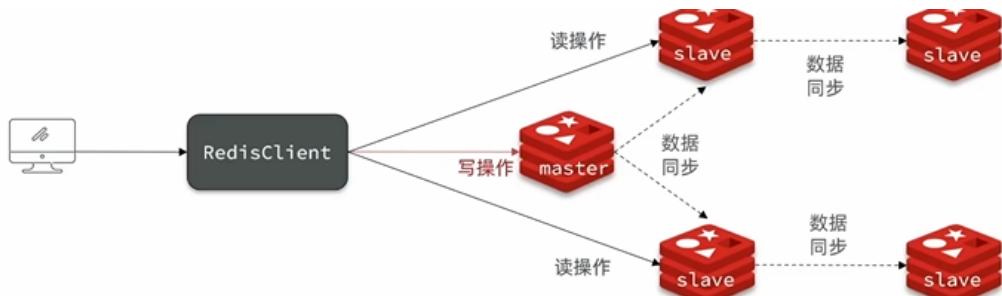
脑裂问题了解吗

Redis 的脑裂问题是指出在主从模式或集群模式下，由于网络分区或节点故障，可能导致系统中出现多个主节点，从而引发数据不一致、数据丢失等问题。

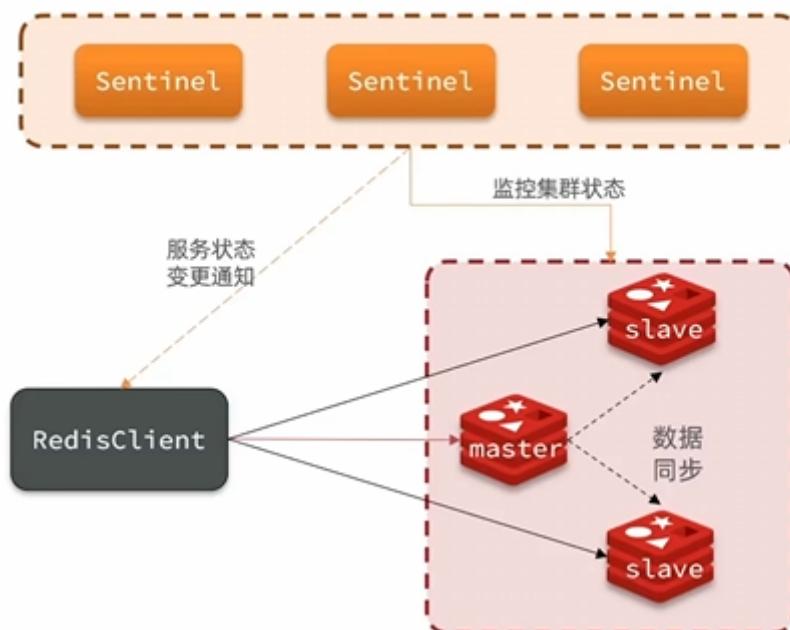
可以通过 Sentinel 模式和 Cluster 模式中的投票机制和强制下线机制来解决。

主从复制如何优化

- 可以在master启用无磁盘复制，避免全量同步时的磁盘IO
- Redis单节点上的内存占用不要太大
- 适当提高repl_baklog的大小
- 限制一个master上的slave节点数量，如果实在太多slave，可以采用主-从-从链式结构减轻master压力



Redis的哨兵模式是什么



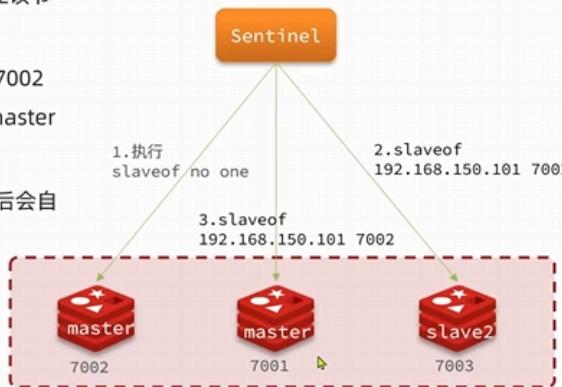
Sentinel基于心跳机制监测服务状态，每隔1秒向集群的每个实例发送ping命令

- 主观下线：某sentinel节点发现某实例未在规定时间响应
- 客观下线：若超过指定数量的sentinel都认为该实例主观下线

哨兵模式如何实现故障转移

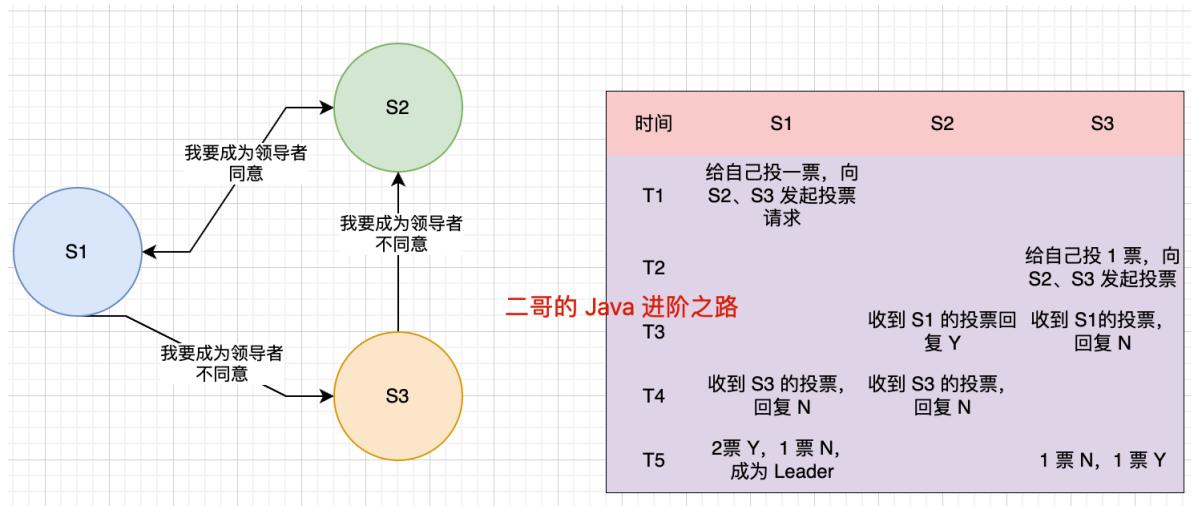
当选中了其中一个slave为新的master后（例如slave1），故障的转移的步骤如下：

- sentinel给备选的slave1节点发送slaveof no one命令，让该节点成为master
- sentinel给所有其它slave发送slaveof 192.168.150.101 7002命令，让这些slave成为新master的从节点，开始从新的master上同步数据。
- 最后，sentinel将故障节点标记为slave，当故障节点恢复后会自动成为新的master的slave节点



领导者Sentinel节点选举了解吗？

Redis 使用 Raft 算法实现领导者选举的：当主节点挂掉后，新的主节点是由剩余的从节点发起选举后晋升的。



1. 每个在线的 Sentinel 节点都有资格成为领导者，当它确认主节点下线时候，会向其他哨兵节点发送命令，表明希望由自己来执行主从切换，并让所有其他哨兵进行投票。

这个投票过程称为“Leader 选举”。候选者会给自己先投 1 票，然后向其他 Sentinel 节点发送投票的请求。

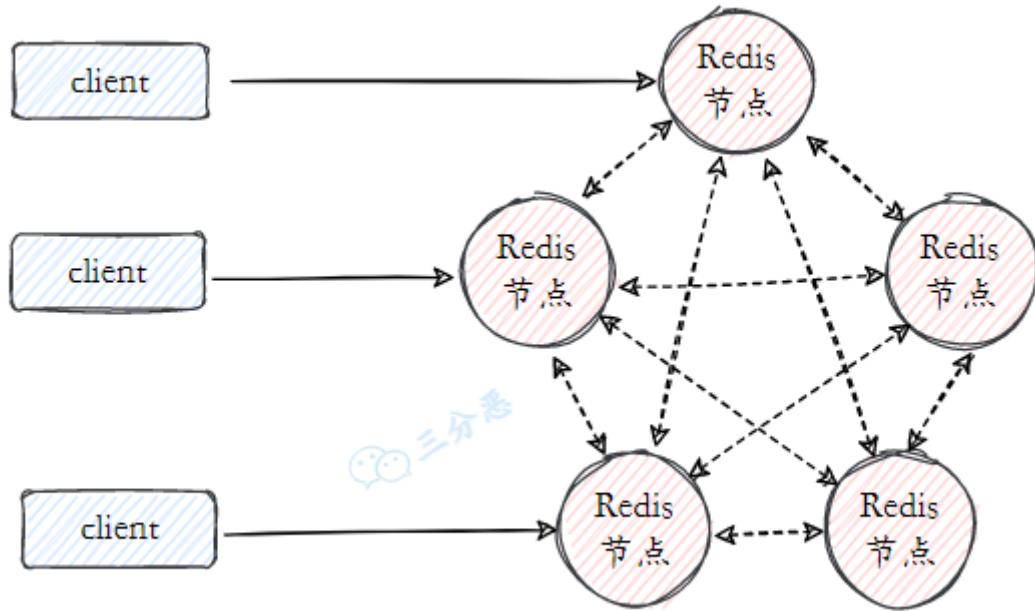
2. 收到请求的 Sentinel 节点会进行判断，如果候选者的日志与自己的日志一样新，任期号也小于自己，且之前没有投票过，就会同意投票，回复 Y。否则回复 N。
3. 候选者收到投票后会统计支持自己的得票数，如果候选者获得了集群中超过半数节点的投票支持（即多数原则），它将成为新的主节点。

新的主节点在确立后，会向其他从节点发送心跳信号，告诉它们自己已经成为主节点，并将其他节点的状态重置为从节点。

4. 如果多个节点同时成为候选人，并且都有可能获得足够的票数，这种情况下可能会出现选票分裂。也就是没有候选者获得超过半数的选票，那么这次选举就会失败，所有候选者都会再次发起选举。

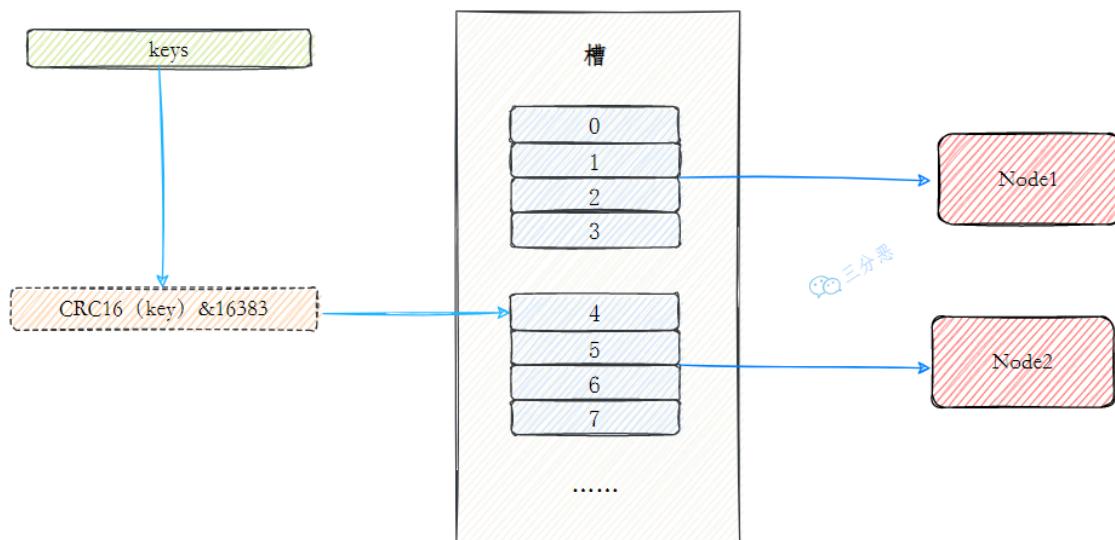
为了防止无限制的选举失败，每个节点都会有一个选举超时时间，且是随机的。

Redis分片集群是什么



- 集群中有多个master，每个master保存不同数据
- 每个master都可以有多个slave节点
- master之间通过ping检测彼此健康状态
- 客户端请求可以访问集群任意节点，最终都会被转发到正确节点

散列插槽

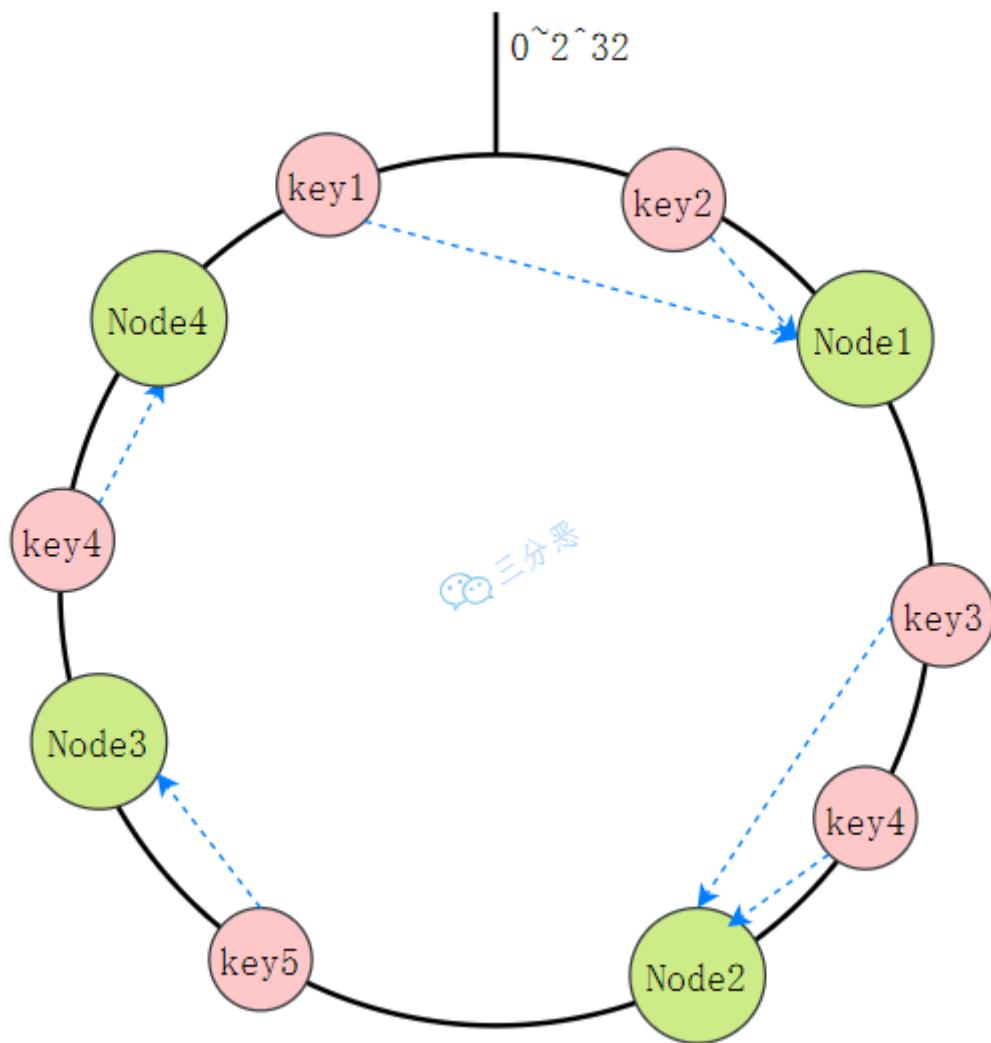


CRC16 是一种哈希算法，它可以将任意长度的输入数据映射为一个 16 位的哈希值。

- Redis会把每一个master节点映射到0~16383共16384个插槽上

```
M: f5fc58defbebb957e47fb0d8327a09dc4f1678f5 192.168.150.101:7001
  slots:[0-5460] (5461 slots) master
M: afaaa70d6528fc72490e0f3f7b32731a12c12bb8 192.168.150.101:7002
  slots:[5461-10922] (5462 slots) master
M: 1c00e5f9e158b169f199f15884ab43bc433b1a06 192.168.150.101:7003
  slots:[10923-16383] (5461 slots) master
```

Redis的一致性哈希是什么

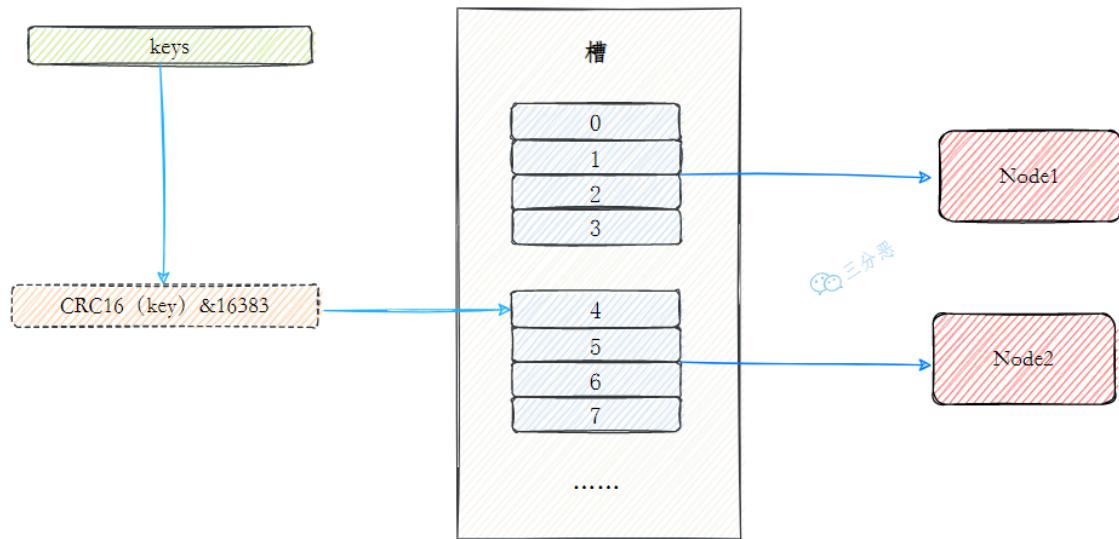


- 将哈希值空间组织成一个环，数据项和节点都映射到这个环上。数据项由其哈希值直接映射到环上，然后顺时针分配到遇到的第一个节点。从而来减少节点变动时数据迁移的量。

存在的问题：

- 节点在圆环上分布不平均，会造成部分缓存节点的压力较大
- 当某个节点故障时，这个节点所要承担的所有访问都会被顺移到另一个节点上，会对后面这个节点造成压力。

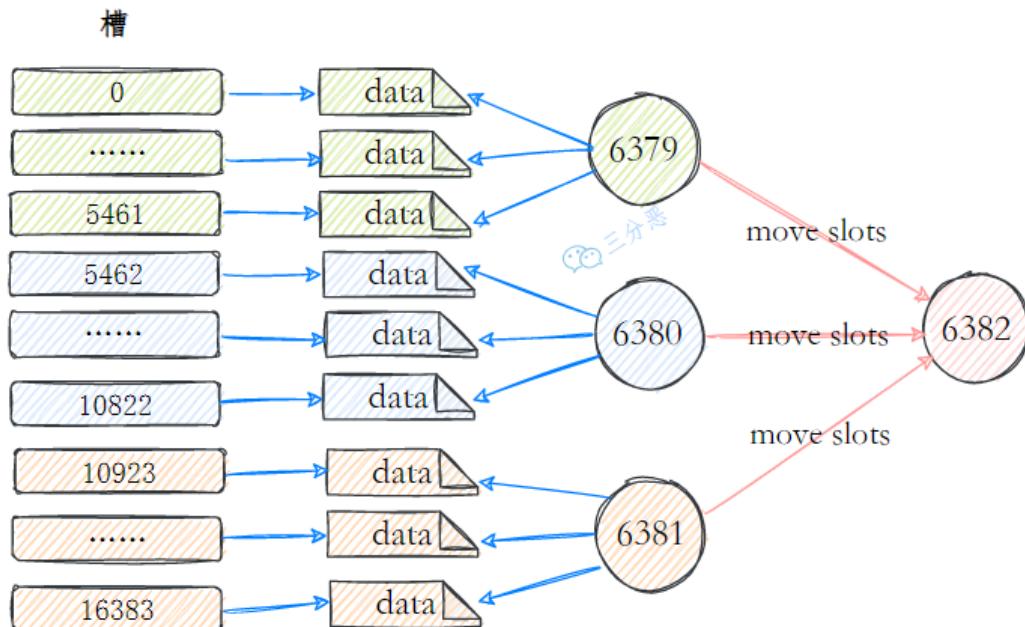
什么是Redis的虚拟槽分区



- 在虚拟槽（也叫哈希槽）分区中，槽位的数量是固定的（例如 Redis Cluster 有 16384 个槽），每个键通过哈希算法（比如 CRC16）映射到这些槽上，每个集群节点负责管理一定范围内的槽。
- 这种分区可以灵活地将槽（以及槽中的数据）从一个节点迁移到另一个节点，从而实现平滑扩容和缩容；数据分布也更加均匀，Redis Cluster 采用的正是这种分区方式。

能保证扩容后，大部分数据停留在扩容前的位置，只有少部分数据需要迁移到新的槽上。

Redis集群如何保证扩容过程中数据正常访问插入



当需要扩容时，新的节点被添加到集群中，集群会自动执行数据迁移，以重新分布哈希槽到新的节点。数据迁移的过程可以确保在扩容期间数据的正常访问和插入。

当数据正在迁移时，客户端请求可能被路由到原有节点或新节点。Redis Cluster 会根据哈希槽的映射关系判断请求应该被路由到哪个节点，并在必要时进行重定向。

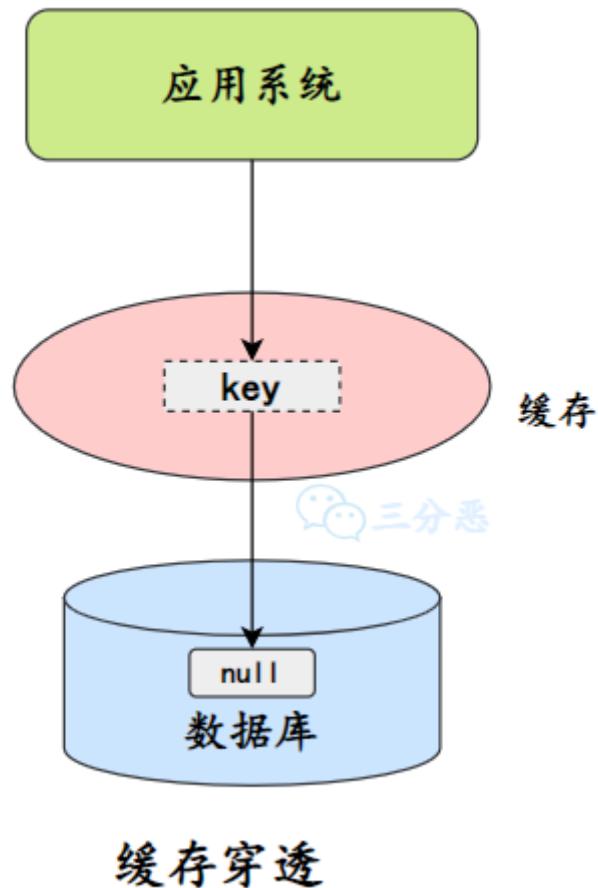
如果请求被路由到正在迁移数据的哈希槽，Redis Cluster 会返回一个 MOVED 响应，指示客户端重新路由请求到正确的目标节点。这种机制也就保证了数据迁移过程中的最终一致性。

缓存设计

缓存穿透、缓存雪崩、缓存击穿了解吗

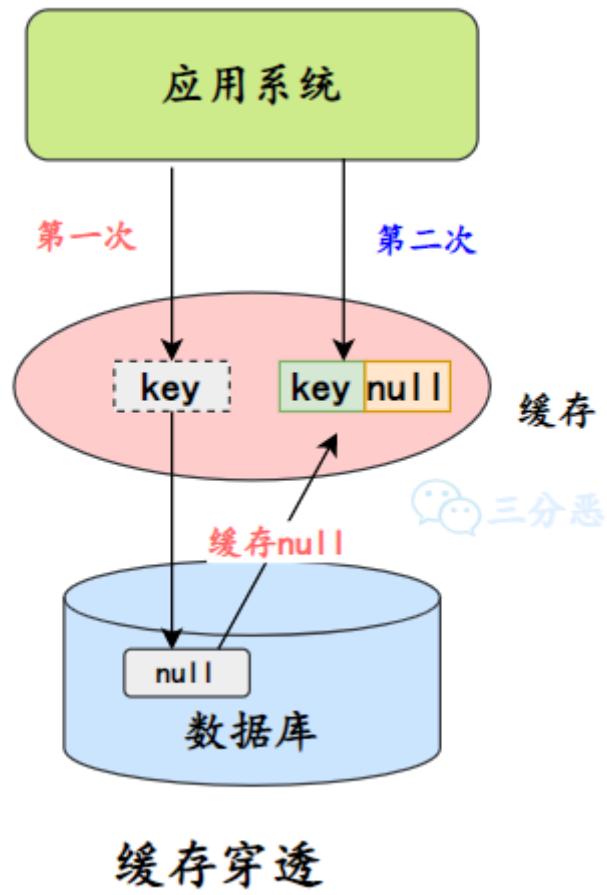
缓存穿透

查询不存在的数据，由于缓存没有命中，请求每次都会穿过缓存去查询数据库。如果这种查询非常频繁，会给数据库造成很大压力



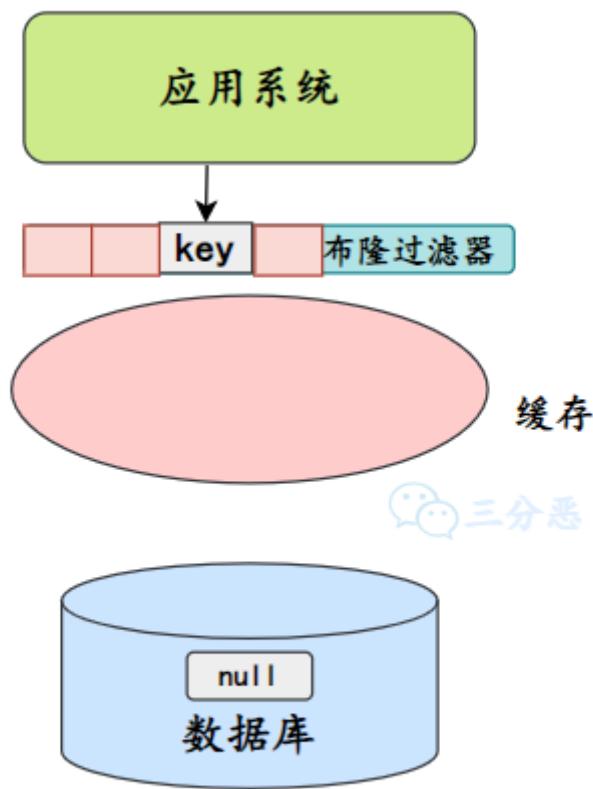
解决方法：

- 缓存空值/默认值



缓存穿透

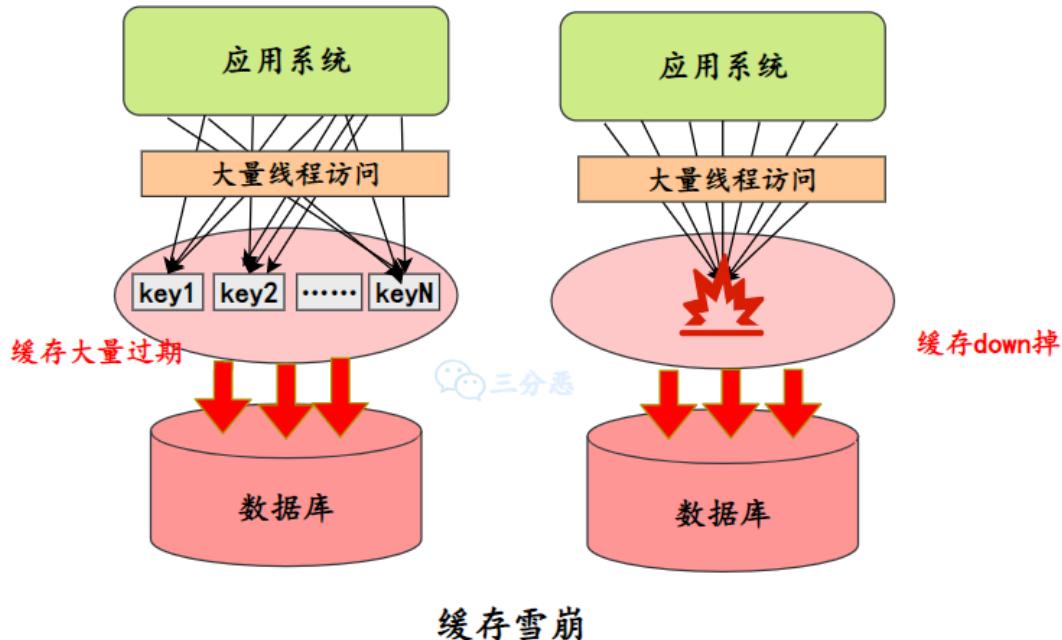
- 布隆过滤器



缓存穿透

缓存雪崩

在某一个时间点，由于大量的缓存数据同时过期或缓存服务器突然宕机了，导致所有的请求都落到了数据库上（比如 MySQL），从而对数据库造成巨大压力，甚至导致数据库崩溃的现象。

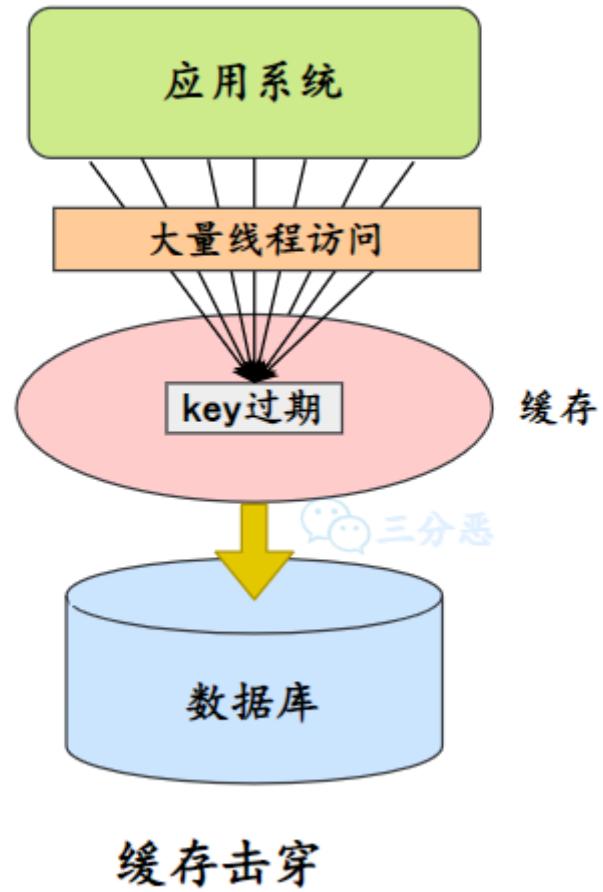


解决方法：

- 给不同的Key的TTL添加随机值
- 利用Redis集群提高服务的可用性
- 给缓存业务添加降级限流策略，防止在缓存失效时数据库被打垮

缓存击穿

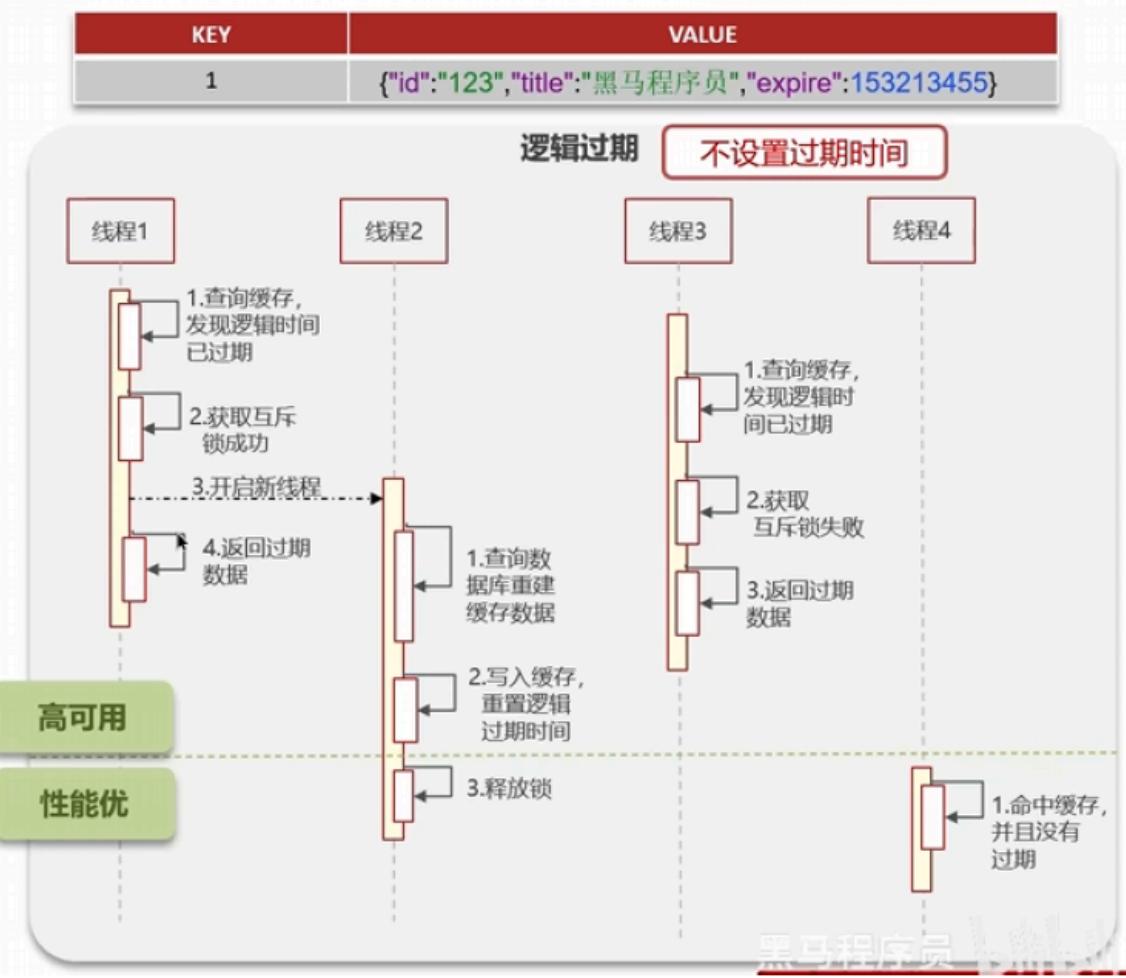
缓存击穿是指某一个或少数几个数据被高频访问，当这些数据在缓存中过期的那一刻，大量请求就会直接到达数据库，导致数据库瞬间压力过大。



缓存击穿

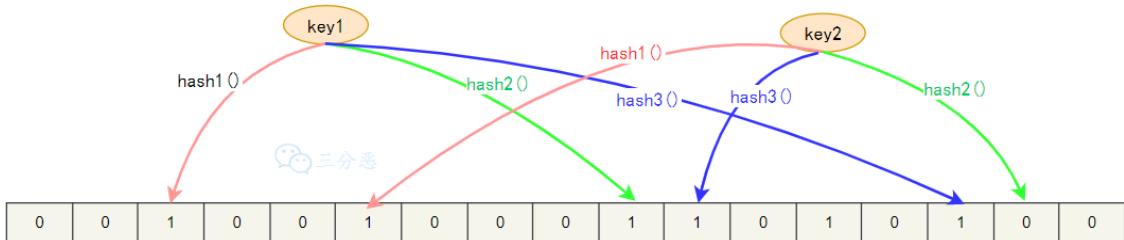
解决方法：

- 互斥锁：只有一个请求会去数据库查询，写入缓存，后面的请求直接从缓存中获取数据（强一致）
- 逻辑过期：不设置过期时间



布隆过滤器是什么

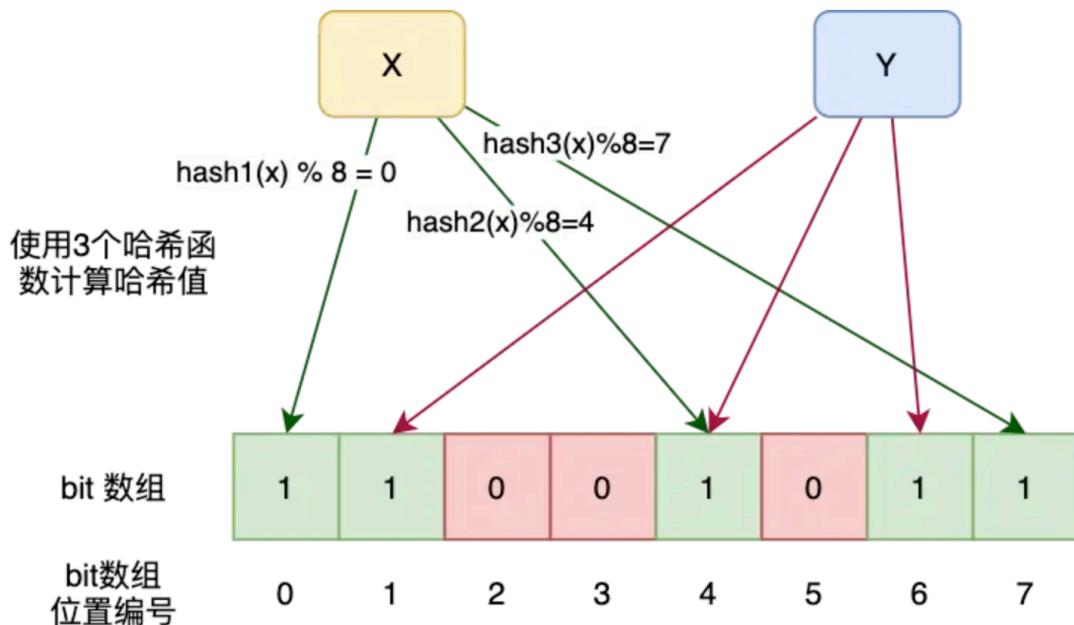
布隆过滤器是一种空间效率极高的概率型数据结构，用于快速检查一个元素是否存在于一个集合中。



布隆过滤器由一个长度为 m 的位数组和 k 个哈希函数组成。

- 开始时，布隆过滤器的每个位都被设置为 0。
- 当一个元素被添加到过滤器中时，它会被 k 个哈希函数分别计算得到 k 个位置，然后将位数组中对应的位设置为 1。
- 当检查一个元素是否存在于过滤器中时，同样使用 k 个哈希函数计算位置，如果任一位置的位为 0，则该元素肯定不在过滤器中；如果所有位置的位都为 1，则该元素可能在过滤器中。

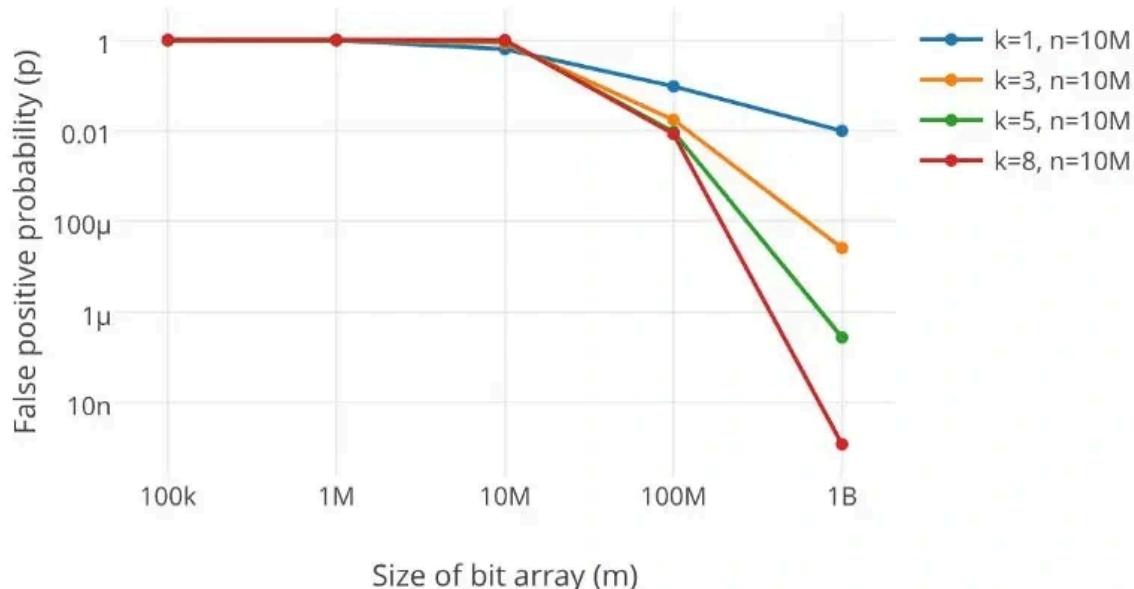
存在误判：元素不存在但布隆过滤器判断为存在



布隆过滤器的误判率与什么因素有关

布隆过滤器的误判率取决于以下几个因素：

- 位数组的大小 (m)：位数组的大小决定了可以存储的标志位数量。如果位数组过小，那么哈希碰撞的几率就会增加，从而导致更高的误判率。
- 哈希函数的数量 (k)：哈希函数的数量决定了每个元素在位数组中标记的位数。哈希函数越多，碰撞的概率也会相应变化。如果哈希函数太少，则过滤器很快会变得不精确；如果太多，误判率也会升高，效率下降。
- 存入的元素数量 (n)：存入的元素越多，哈希碰撞的几率越大，从而导致更高的误判率。



布隆过滤器支持删除吗

布隆过滤器不支持删除元素，因为多个元素可能哈希到一个布隆过滤器的同一个位置，如果直接删除该位置的元素，则会影响其他元素的判断。

如何保证缓存和数据库的数据一致性

采用先写MySQL，再删除Redis的方式来保证缓存和数据库的一致性。

The diagram illustrates the sequence of events for two requests, A and B, interacting with Redis and MySQL:

- Request A:** Issues a query to Redis. Redis returns cached data (10). MySQL is updated to 11.
- Request B:** Issues a query to Redis. Redis returns cached data (10). Request B then deletes the Redis cache (10).
- Request B (Continued):** Issues a query to Redis. Redis returns "query cache miss". Request B then issues a query to MySQL, which returns 11. Request B then writes back to Redis (11).

Notes:

- For the first query from Request B, Redis has data (10) and MySQL has data (11), resulting in inconsistency.
- For the second query from Request B, Redis has no data (miss) and MySQL has data (11), resulting in consistency.

理论知识:

- 不好的方案:**
 - 先写 MySQL, 再写 Redis
 - 先写 Redis, 再写 MySQL
 - 先删除 Redis, 再写 MySQL
- 好的方案:**
 - 先删除 Redis, 再写 MySQL, ...
 - 先写 MySQL, 再删除 Redis
 - 先写 MySQL, 通过 Binlog, ...

几种方案比较:

- 项目实战
 - 数据更新
 - 数据获取
 - 测试用例

为什么要先更新数据库，再删除缓存

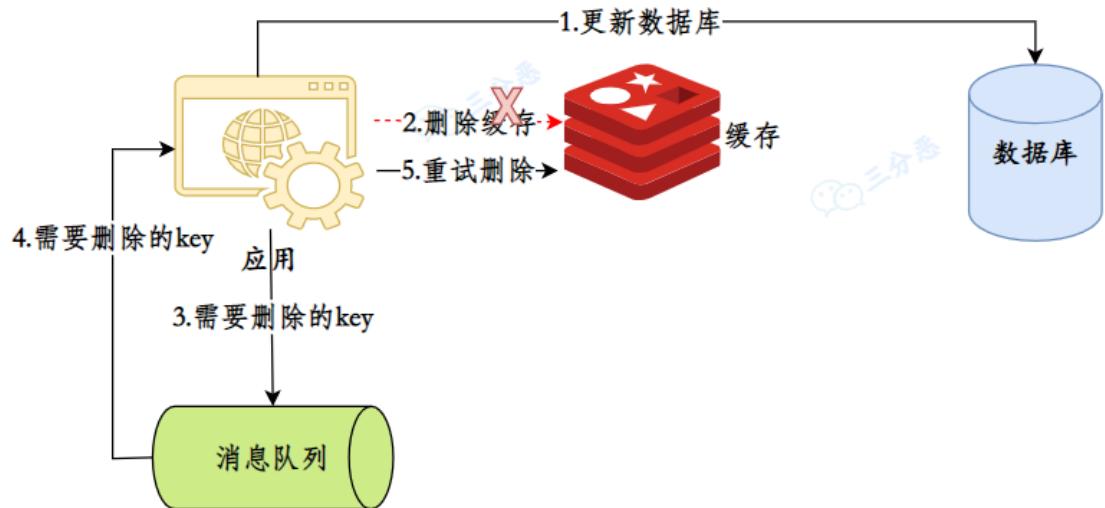
因为更新数据库的速度比删除缓存的速度要慢得多。

如果先删除缓存，再更新数据库，会造成这样的情况：

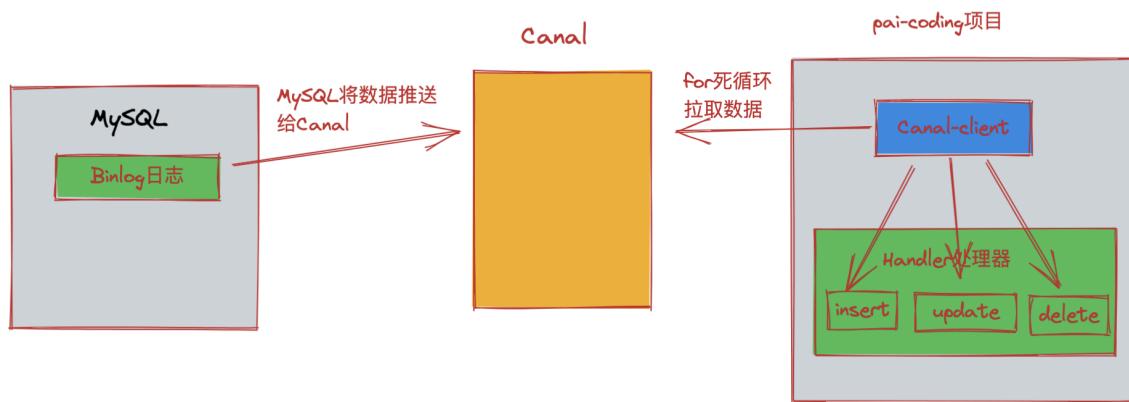
- 另一个请求过来发现缓存中不存在，数据库又没有完成更新，就会从数据库中读取旧的数据并写入缓存，这时写入缓存的是旧的数据，不一致了

如果对一致性要求很高该怎么办

1. 引入消息队列保证缓存被删除

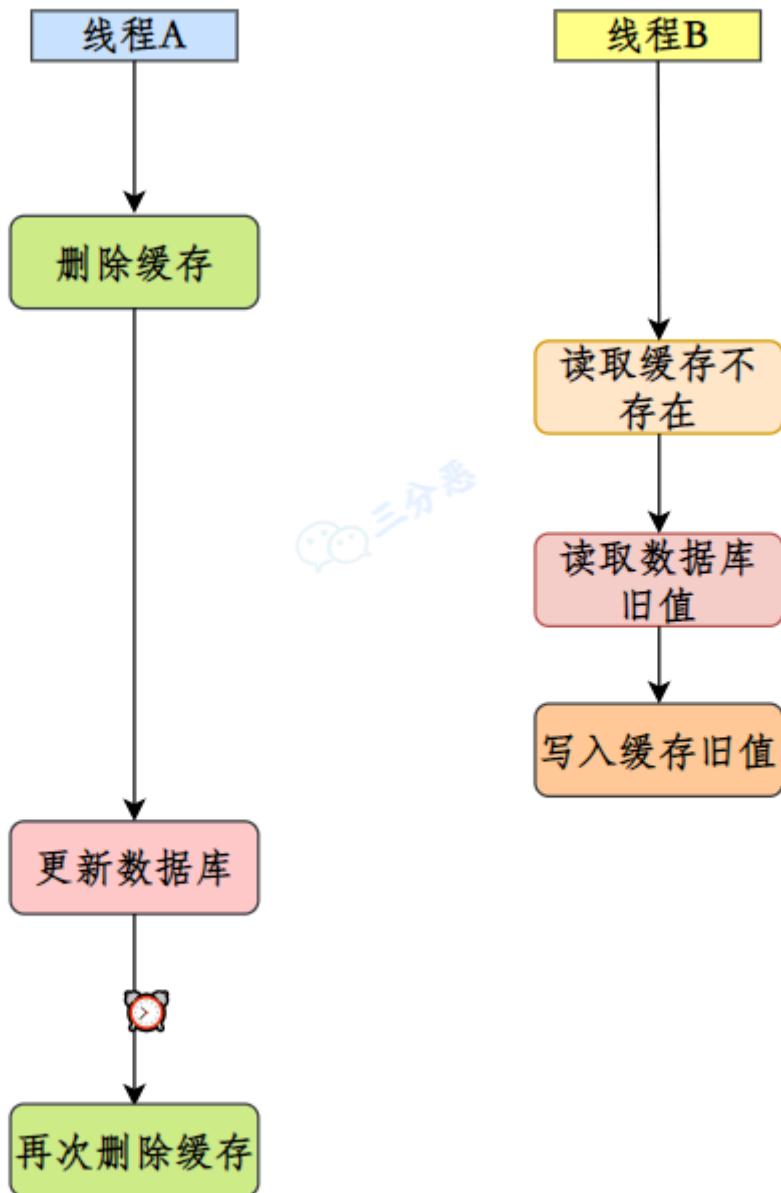


2. 数据库订阅+消息队列保证缓存被删除



3. 延时双删防止脏数据

在第一次删除缓存之后，过一段时间之后，再次删除缓存。



这种方式的延时时间需要仔细考量和测试。

4. 设置缓存过期时间兜底

这是一个朴素但有用的兜底策略，给缓存设置一个合理的过期时间，即使发生了缓存和数据库的数据不一致问题，也不会永远不一致下去，缓存过期后，自然就一致了。

如何保证本地缓存和分布式缓存的一致性

1. 设置本地缓存的过期时间

- 当本地缓存过期时，就从Redis缓存中去同步

2. 使用Redis的Pub/Sub机制

- 当Redis缓存发生变化时，发布一个消息，本地缓存订阅这个消息，然后删除对应的本地缓存

3. Redis缓存发生变化时，引用消息队列去更新本地缓存

热key导致的问题

所谓的热 key，就是指在很短时间内被频繁访问的键。

1. Redis单点压力过大，可能导致雪崩
2. 热 key 在 Redis 分片集群下，可能集中在某个节点

怎么处理热key

对热key的处理，最关键的是对热key的监控：

1. 客户端
 - 在客户端设置全局字典（key 和调用次数），每次调用 Redis 命令时，使用这个字典进行记录。
2. Redis服务端
 - 使用 monitor 命令统计热点 key 是很多开发和运维人员首先想到的方案，monitor 命令可以监控到 Redis 执行的所有命令。

只要监控到了热key，对热key的处理就简单了：

- 把热key打散到不同的服务器，降低压力
 - 给热 Key 加上前缀或者后缀

```
1 // N 为 Redis 实例个数，M 为 N 的 2倍
2 const M = N * 2
3 //生成随机数
4 random = GenRandom(0, M)
5 //构造备份新 Key
6 bakHotKey = hotKey + "_" + random
7 data = redis.GET(bakHotKey)
8 if data == NULL {
9     data = redis.GET(hotKey)
10    if data == NULL {
11        data = GetFromDB()
12        // 可以利用原子锁来写入数据保证数据一致性
13        redis.SET(hotKey, data, expireTime)
14        redis.SET(bakHotKey, data, expireTime + GenRandom(0, 5))
15    } else {
16        redis.SET(bakHotKey, data, expireTime + GenRandom(0, 5))
17    }
18 }
```

- 加入二级缓存，当出现热key后，把热key加载到本地缓存中，后续针对这些热 Key 的请求，直接从本地缓存中读取。

缓存预热怎么做

缓存预热是指在系统启动时，提前将一些预定义的数据加载到缓存中，以避免在系统运行初期由于缓存未命中（cache miss）导致的性能问题。

- 项目启动时自动加载
- 定时预热

热点key重建是什么

发的时候一般使用“缓存+过期时间”的策略，既可以加速数据读写，又保证数据的定期更新，这种模式基本能够满足绝大部分需求。

但是有两个问题如果同时出现，可能就会出现比较大的问题：

- 当前 key 是一个热点 key（例如一个热门的娱乐新闻），并发量非常大。
- 重建缓存不能在短时间完成，可能是一个复杂计算，例如复杂的 SQL、多次 IO、多个依赖等。在缓存失效的瞬间，有大量线程来重建缓存，造成后端负载加大，甚至可能会让应用崩溃。

解决方法：

- 互斥锁
 - 只允许一个线程重建缓存，其他线程等待重建缓存的线程执行完，重新从缓存获取数据即可。
- 永远不过期
 - 从缓存层面来看，确实没有设置过期时间，所以不会出现热点 key 过期后产生的问题，也就是“物理”不过期。
 - 从功能层面来看，为每个 value 设置一个逻辑过期时间，当发现超过逻辑过期时间后，会使用单独的线程去构建缓存。

无底洞问题了解吗，如何解决

通常来说添加节点使得缓存集群性能应该更强了，但事实并非如此。键值数据库由于通常采用哈希函数将 key 映射到各个节点上，造成 key 的分布与业务无关，但是由于数据量和访问量的持续增长，造成需要添加大量节点做水平扩容，导致键值分布到更多的节点上，所以**批量操作**通常需要从不同节点上获取，相比于单机批量操作只涉及一次网络操作，分布式批量操作会涉及**多次网络时间**。

如何优化：

- 使用 Hash Tag 保证批量 Key 落在同一个 Redis 节点
- 利用本地缓存减少 redis 访问
- 降低接入成本，例如客户端使用长连/连接池、NIO 等

Redis运维

Redis报内存不足怎么处理

- 修改redis.conf的maxmemory参数
- 修改内存淘汰策略，及时释放内存空间
- 使用集群模式，进行横向扩容

Redis的key的过期策略有哪些

1. 惰性删除

- 时间到了并不会立即删除
- 如果访问key的时候发现过期了才删除
- 如果没有访问可能一直不会删除

2. 定期删除

- 每个隔一段时间，Redis都会随机检查一批key，并删除其中过期的key

定期删除会增加Redis的CPU消耗，惰性删除会增加Redis的内存占用

Redis有哪些内存淘汰策略

- **noeviction**: 默认策略，不进行任何数据淘汰，当内存不足时，直接返回OOM错误
- **allkeys-lru**: 从所有键中，使用LRU算法淘汰最久未被访问的键
- **allkeys-lfu**: 从所有键中，使用LFU算法淘汰访问频率最低的键的键
- **volatile-lru**: 从设置了过期的键中，使用LRU算法淘汰最久未被访问的键
- **volatile-ttl**: 从设置了过期时间的键中淘汰即将过期的键

Redis阻塞怎么解决

首先找到Redis阻塞的原因，再进行解决：

- 慢查询、慢删除（BigKey）
 - 使用 `redis-cli --bigkeys` 查找big key
 - 分批查询big key，避免HGETALL
- RDB / AOF触发
 - 异步RDB快照，避免阻塞主进程
 - AOF在重写的时候会占大量的CPU和内存资源，导致服务load过高，出现短暂服务暂停现象。

- 限制RDB和AOF触发条件
- 主从同步阻塞
 - 改为异步复制

大key问题了解吗

大key指的是单个key存储的数据量过大，导致查询、删除、迁移等操作变慢

删除大key

- 使用 UNLINK 命令安全地删除大 Key，该命令能够以非阻塞的方式，逐步地清理传入的大 Key。

压缩和拆分key

- 将一个大 key 分为不同的部分，记录每个部分的 key，使用 multiget 等操作实现事务读取。
- 当 value 是 list/set 等集合类型时，根据预估的数据规模来进行分片，不同的元素计算后分到不同的片

假如Redis里有一亿个key，其中有10w个key是以某个固定的已知的前缀开头的，如何将它们全部找出来？

1. Redis的KEY命令

```
1 | KEYS pattern
```

`pattern`：支持通配符匹配，如：

- `*`：匹配所有 Key
- `user:*`：匹配 `user:` 前缀的 Key
- `session:*:active`：匹配 `session:xxx:active`

一次性扫描所有Key，大数据量时会导致Redis阻塞，不推荐！

2. SCAN命令

```
1 | SCAN cursor [MATCH pattern] [COUNT count]
```

- `cursor`：游标，从 `0` 开始，每次返回部分 Key
- `MATCH pattern`：按模式匹配 Key
- `COUNT count`：每次返回的 Key 数量（仅建议值，不是硬性限制）

非阻塞，逐步扫描

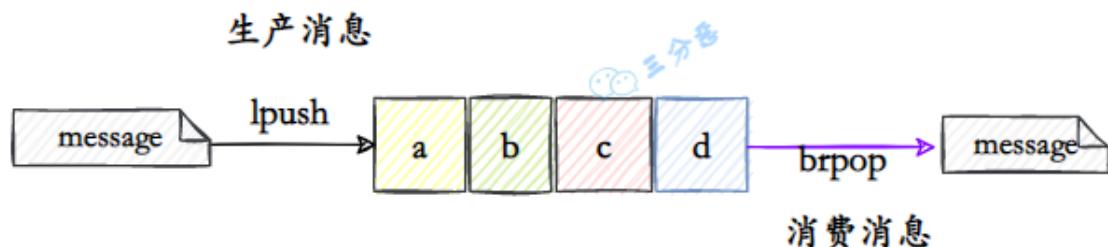
如果Redis客户端宕机服务器如何感知？

- 每个客户端在 Redis 中维护一个特定的键（称为心跳键），用于表示客户端的健康状态。该键具有一个设置的超时时间，例如 10 秒。
- 客户端定期（如每 5 秒）更新这个心跳键的超时时间，保持它的存活状态，通常通过 SET 命令重设键的过期时间。
- Redis 服务端定期检查这个心跳键。如果发现该键已超时并被 Redis 自动删除，说明客户端可能已宕机。

Redis应用

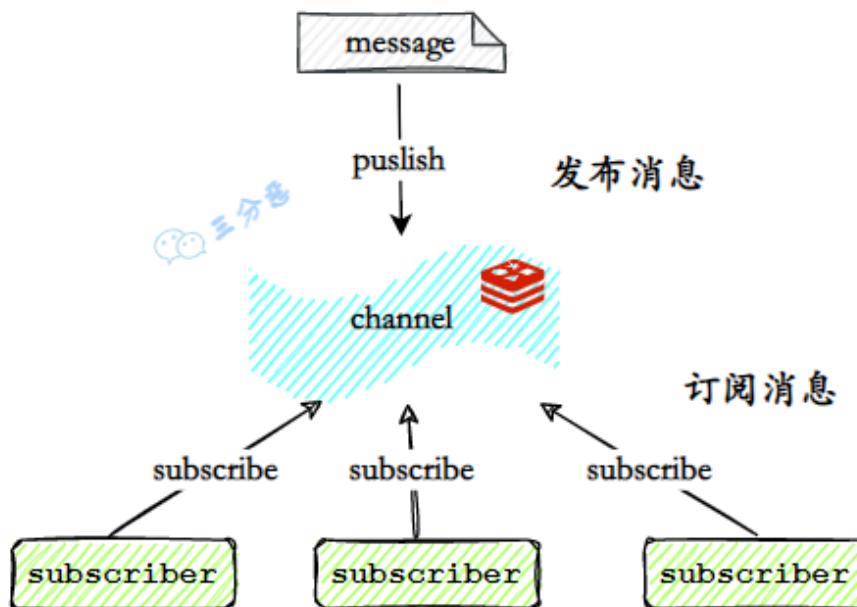
Redis如何实现异步队列

1. 使用 list 作为队列，lpush 生产消息，brpop 消费消息



- brpop 是 rpop 的阻塞版本，list 为空的时候，它会一直阻塞，直到 list 中有值或者超时。
- 这种方式只能实现一对一对的消息队列。

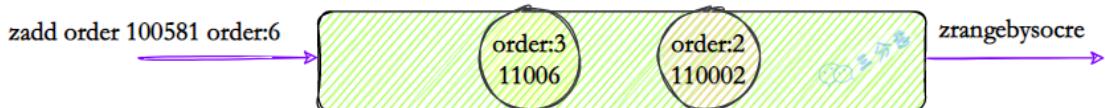
2. 使用 Redis 的 pub/sub 来进行消息的发布/订阅



- 发布/订阅模式可以 1: N 的消息发布/订阅。发布者将消息发布到指定的频道频道 (channel)，订阅相应频道的客户端都能收到消息。
- 但它不保证订阅者一定能收到消息，也不进行消息的存储。

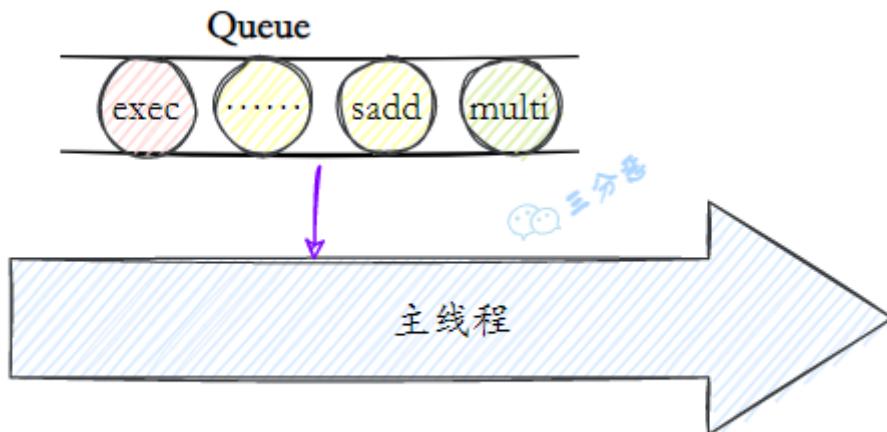
Redis如何实现延时队列

使用 Redis 的 zset (有序集合) 来实现延时队列



- 将任务添加到 zset 中，score 为任务的执行时间戳，value 为任务的内容。
- 定期（例如每秒）从 zset 中获取 score 小于当前时间戳的任务，然后执行任务。
- 任务执行后，从 zset 中删除任务。

Redis支持事务吗



- 使用 `MULTI` 命令开始一个事务。从这个命令执行之后开始，所有的后续命令都不会立即执行，而是被放入一个队列中。在这个阶段，Redis 只是记录下了这些命令。
- 使用 `EXEC` 命令触发事务的执行。一旦执行了 `EXEC`，之前 `MULTI` 后队列中的所有命令会被原子地 (atomic) 执行。这里的“原子”意味着这些命令要么全部执行，要么在 `EXEC` 之前错误全部不执行，在 `EXEC` 之后错误不支持回滚。
- 如果在执行 `EXEC` 之前决定不执行事务，可以使用 `DISCARD` 命令来取消事务。这会清空事务队列并退出事务状态。
- `WATCH` 命令用于实现乐观锁。`WATCH` 命令可以监视一个或多个键，如果在执行事务的过程中（即在执行 `MULTI` 之后，执行 `EXEC` 之前），被监视的键被其他命令改变了，那么当执行 `EXEC` 时，事务将被取消，并且返回一个错误。

Redis事务为什么不支持回滚

Redis 是一个基于内存的数据存储系统，其设计重点是实现高性能。事务回滚需要额外的资源和时间来管理和执行，这与 Redis 的设计目标相违背。因此，Redis 选择不支持事务回滚。

Redis的ACID特性如何体现

原子性

- Redis 事务不支持回滚，一旦 EXEC 命令被调用，所有命令都会被执行，即使有些命令可能执行失败。

二、为什么Lua脚本可以保证原子性？

原子性在并发编程中，和在数据库中两种不同的概念。

在数据库中，事务的ACID中原子性指的是“要么都执行要么都回滚”。在 [并发编程](#) 中，原子性指的是“操作不可拆分、不被中断”。

Redis 既是一个数据库，又是一个支持并发编程的系统，所以，他的原子性有两种。所以，我们需要明确清楚，在问“Lua脚本保证 Redis原子性”的时候，指的到底是哪个原子性。

Lua脚本可以保证原子性，因为Redis会将Lua脚本封装成一个单独的事务，而这个单独的事务会在Redis客户端运行时，由Redis服务器自行处理并完成整个事务，如果在这个过程中有其他客户端请求的时候，Redis将会把它暂存起来，等到 Lua 脚本处理完毕后，才会再把被暂存的请求恢复。

这样就可以保证整个脚本是作为一个整体执行的，中间不会被其他命令插入。但是，如果命令执行过程中命令产生错误，事务是不会回滚的，会影响后续命令的执行。

也就是说，Redis保证以原子方式执行Lua脚本，但是不保证脚本中所有操作要么都执行或者都回滚。

那也就意味着，Redis中Lua脚本的执行，可以保证并发编程中不可拆分、不被中断的这个原子性，但是没有保证数据库ACID中要么都执行要么都回滚的这个原子性。

一致性

- Redis 事务中，所有命令会依次执行，但并不支持部分失败后的自动回滚。因此 Redis 在事务层面 **并不能保证一致性**，我们必须通过程序逻辑来进行优化。

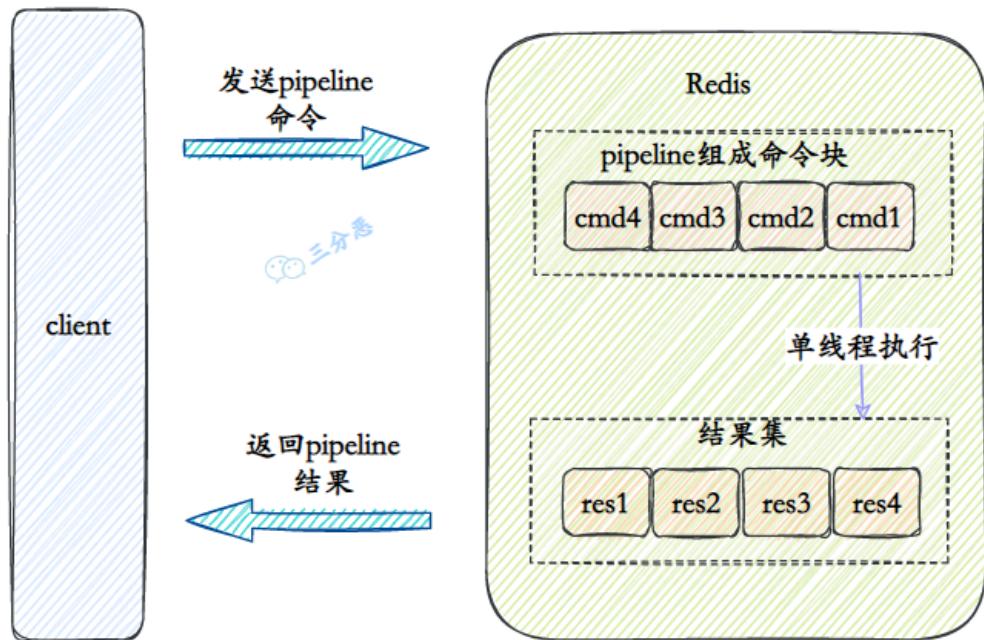
隔离性

- Redis 事务在一定程度上提供了隔离性，事务中的命令会按顺序执行，不会被其他客户端的命令插入。

持久性

- Redis 的持久性依赖于其持久化机制（如 RDB 和 AOF），而不是事务本身。

Redis的管道Pipeline了解吗



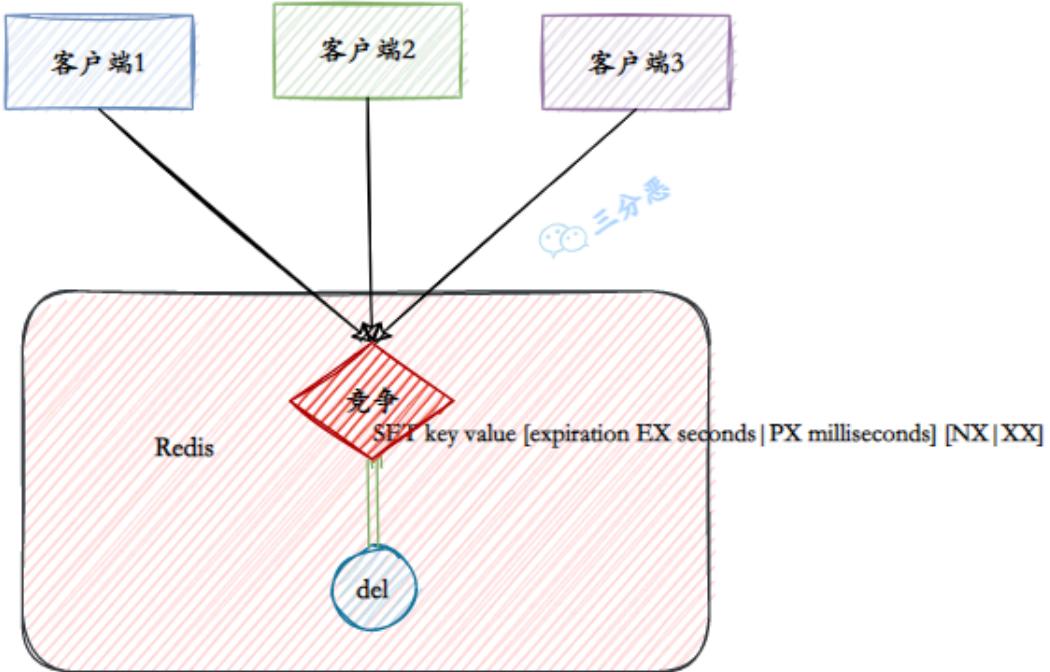
Pipeline 是 Redis 提供的一种优化手段，允许客户端一次性向服务器发送多个命令，而不必等待每个命令的响应，从而减少网络延迟。它的工作原理类似于批量操作，即多个命令一次性打包发送，Redis 服务器依次执行后再将结果一次性返回给客户端。

有了 Pipeline 后，流程变为：

发送命令1、命令2、命令3..... -> 服务器处理 -> 一次性返回所有结果。

Redis实现分布式锁了解吗

可以使用 Redis 的 SET 命令实现分布式锁。同时添加过期时间，以防止死锁的发生。

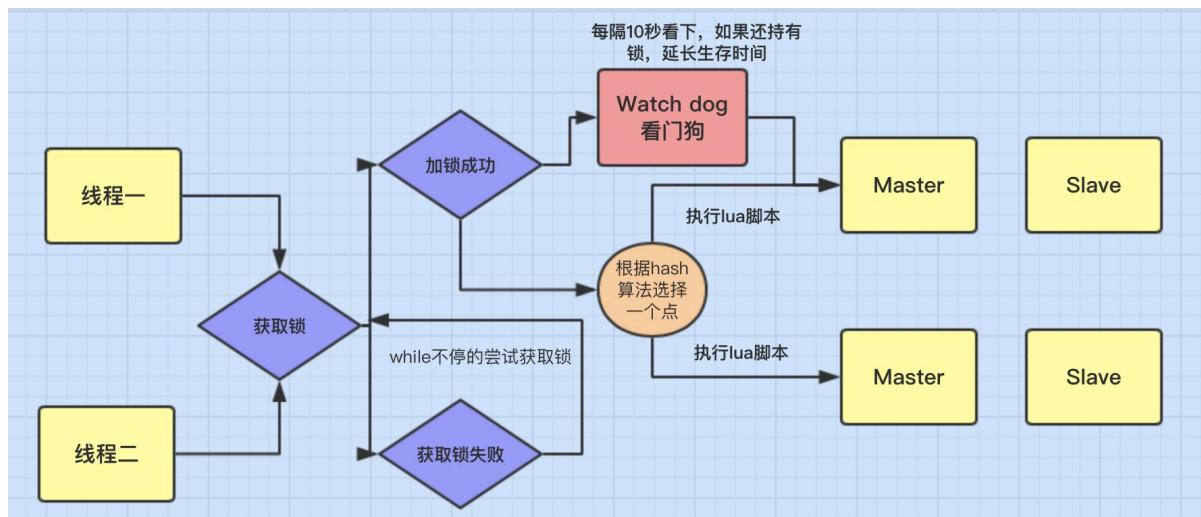


- 使用 `setnx` 创建分布式锁时，虽然设置过期时间可以避免死锁问题，但可能存在这样的问题：线程 A 获取锁后开始任务，如果任务执行时间超过锁的过期时间，锁会提前释放，导致线程 B 也获取了锁并开始执行任务。这会破坏锁的独占性，导致并发访问资源，进而造成数据不一致。

可以引入**锁的自动续约机制**，在任务执行过程中定期续期，确保锁在任务完成之前不会过期。

Redisson是怎么实现分布式锁的

Redisson提供的分布式锁是支持锁自动续期的，也就是说，如果线程在锁到期之前还没执行完，那么Redisson会自动给锁续期



- 看门狗启动后，每隔 10 秒会刷新锁的过期时间，将其延长到 30 秒，确保在锁持有期间不会因为过期而释放。
- 当任务执行完成时，客户端调用 `unlock()` 方法释放锁，看门狗也随之停止。

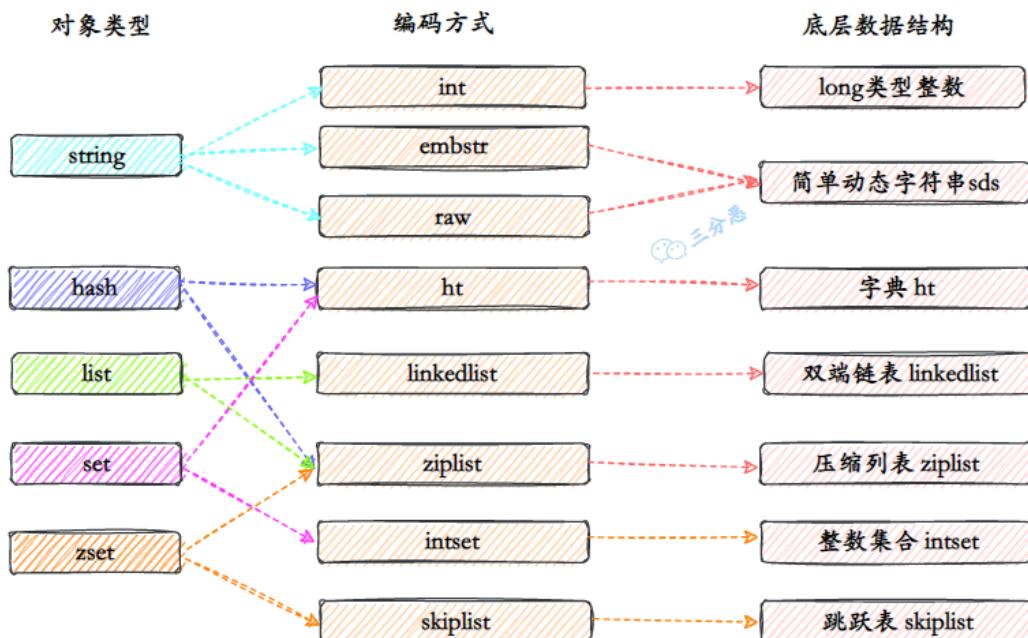
什么是Red Lock

- 会在指定的 Redis 实例上逐一尝试获取锁，至少需要多数（如 5 个实例中的 3 个）实例成功获取锁，才能认为整个锁获取成功
- 如果指定了锁的持有时间（leaseTime），在成功获取锁后，Redlock 会为锁进行续期，以防止锁在操作完成之前意外失效。

底层结构

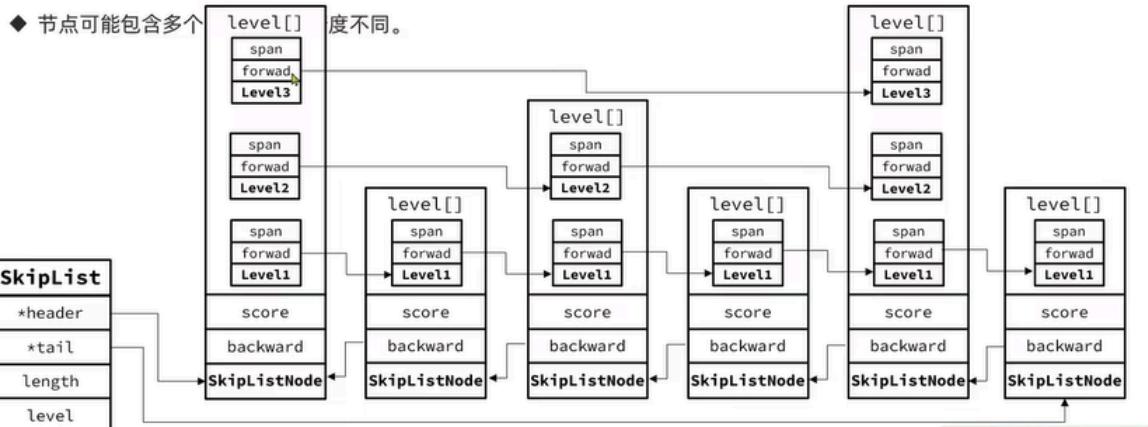
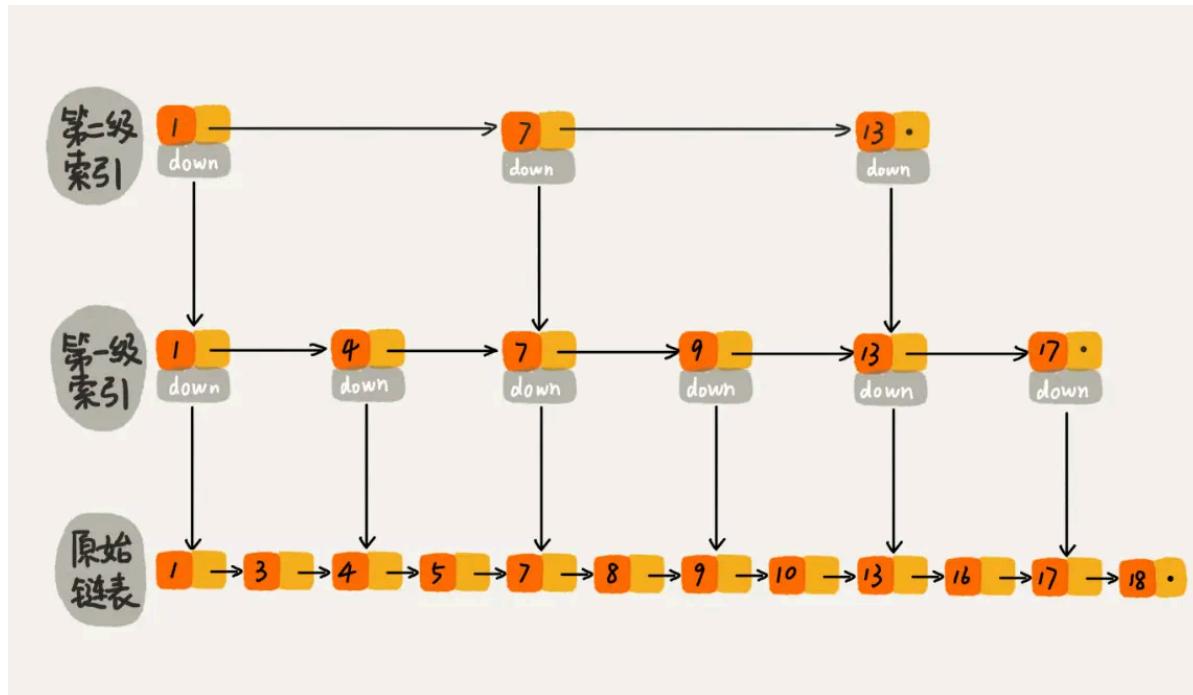
说说Redis的底层数据结构

Redis 的底层数据结构有动态字符串(sds)、链表(list)、字典(ht)、跳跃表(skiplist)、整数集合(intset)、压缩列表(ziplist) 等。



什么是跳表

- 底层是链表
- 元素按照升序排列存储
- 节点可能包含多个指针，指针跨度不同
- 查找、插入、删除的时间复杂度都是 $O(\log n)$ 的一种数据结构



什么是RedisObject

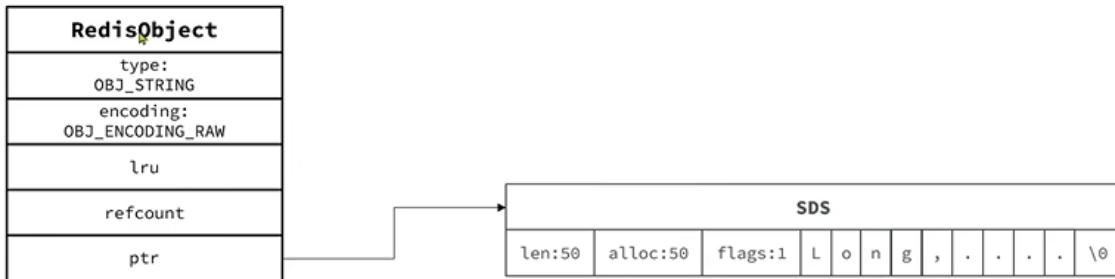
Redis中的任意数据类型的键和值都会被封装为一个RedisObject，也叫做Redis对象



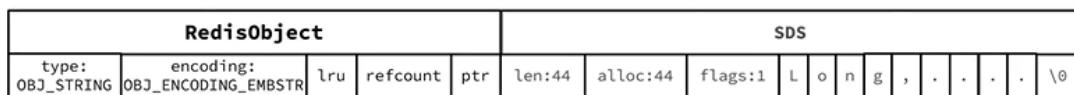
Redis头部（除了ptr的部分）占了16个字节

Redis的String数据类型的底层结构

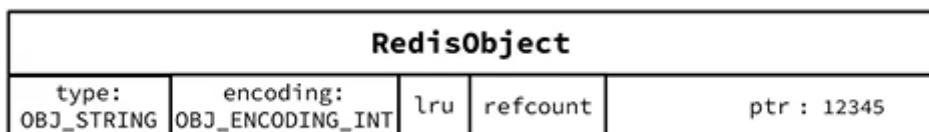
- 基本编码方式是RAW，基于简单动态字符串（SDS）实现，存储上限为512mb



- 如果存储的SDS长度小于44字节，则会采用**EMBSTR**编码，此时object head与SDS是一段连续空间，申请内存时只需要调用一次内存分配函数，效率更高



- 如果数据是数字，则会采用**INT**编码，直接使用ptr存储数据



Redis的SDS和C中字符串相比有什么优势

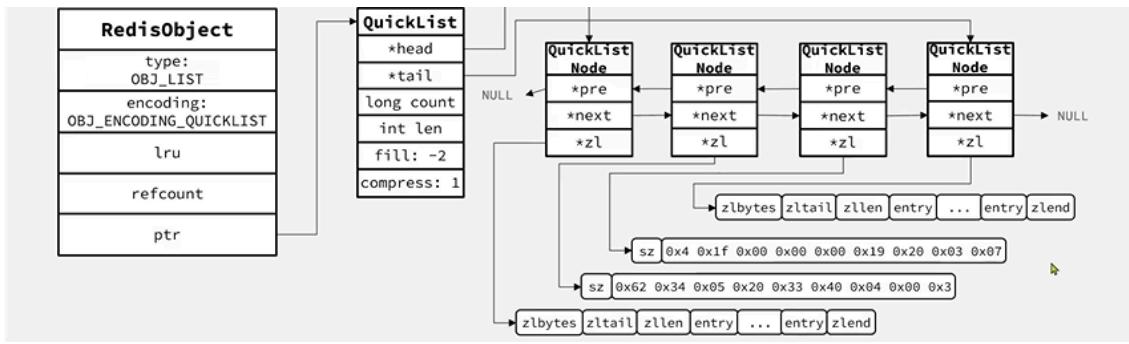
- 获取字符串长度复杂度高**：因为 C 不保存数组的长度，每次都需要遍历一遍整个字符数组，时间复杂度为 $O(n)$ ；
- 不能杜绝 缓冲区溢出/内存泄漏 的问题**
- 只能保存 ascii 码**

Redis的优势：

- 多增加 len 表示当前字符串的长度**：这样就可以直接获取长度了，复杂度 $O(1)$ ；
- 自动扩展空间**：当 SDS 需要对字符串进行修改时，首先借助于 `len` 和 `alloc` 检查空间是否满足修改所需的要求，如果空间不够的话，SDS 会自动扩展空间，避免了像 C 字符串操作中的溢出情况；
- 二进制安全**：C 语言字符串只能保存 `ascii` 码，对于图片、音频等信息无法保存，SDS 是二进制安全的，写入什么读取就是什么，不做任何过滤和限制；

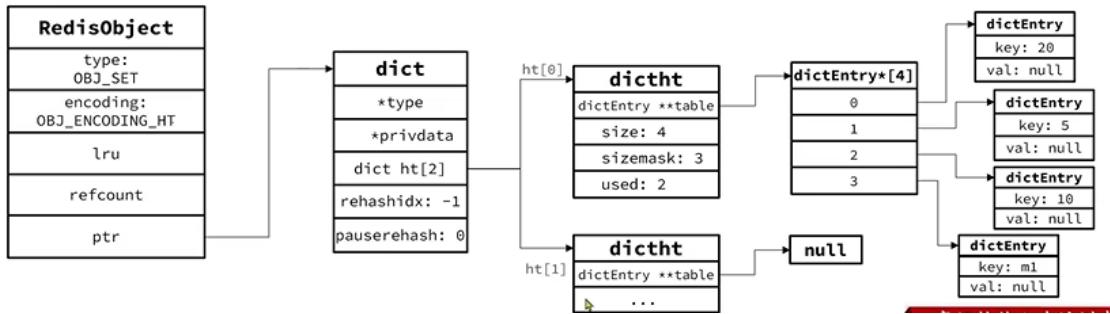
Redis的List数据类型的底层结构

- 在3.2版本之后，Redis统一采用**QuickList**来实现List，QuickList是LinkedList和ZipList的结合，每个LinkerdList的节点是ZipList

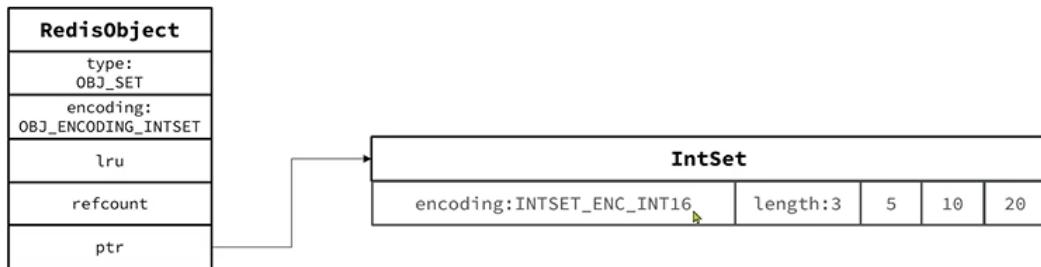


Redis的Set数据类型的底层结构

- 为了查询效率和唯一性，set采用HT编码 (Dict)， Dict中的key用来存储元素， value统一为null



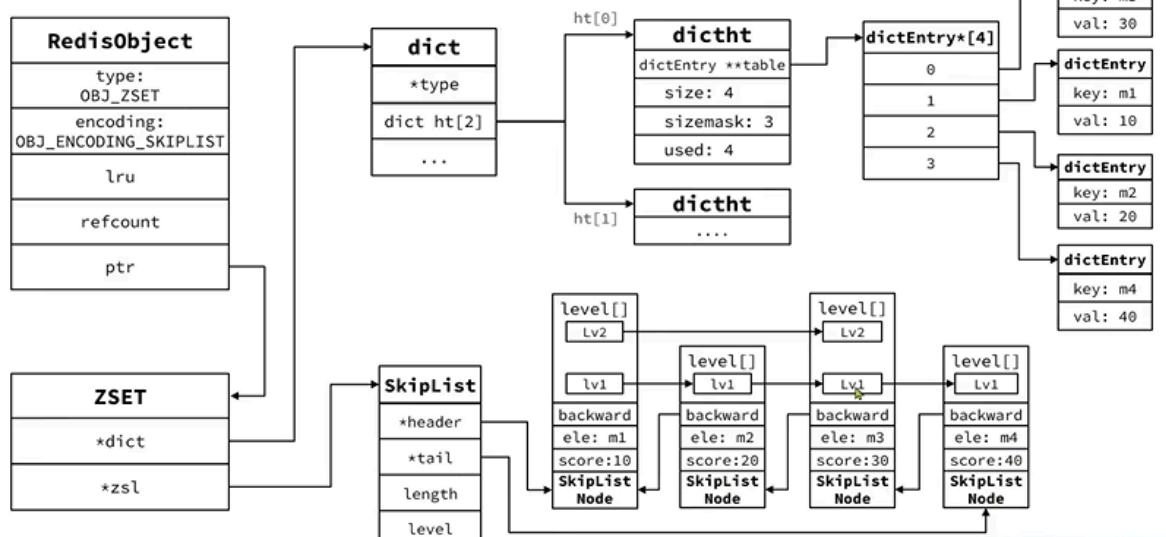
- 当存储的所有数据都是整数，并且元素数量不超过阈值时，Set会采用IntSet编码



Redis的ZSet数据类型的底层结构

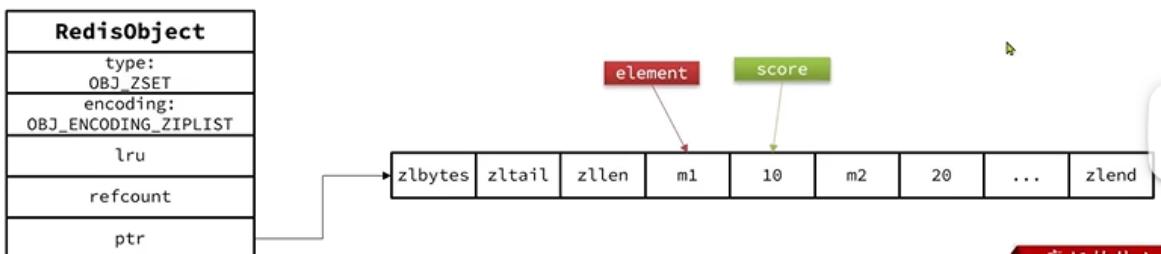
- SkipList：可以排序，并且可以同时存储score和ele值
- HT(Dict)：可以键值存储，并且可以根据key找value

ZSet



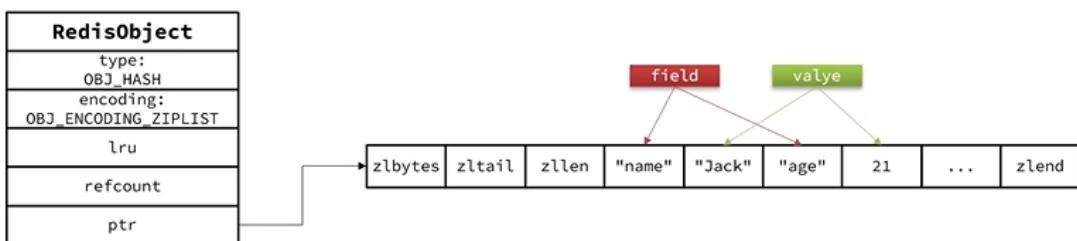
- 当元素数量不多时，HT和SkipList的优势不明显，而且更耗内存，因此会采用ZipList来节省内存

- ZipList是连续内存，因此score和element是紧挨在一起的两个entry，element在前，score在后
- score越小越接近队首，score越大越接近队尾，按照score值升序排列

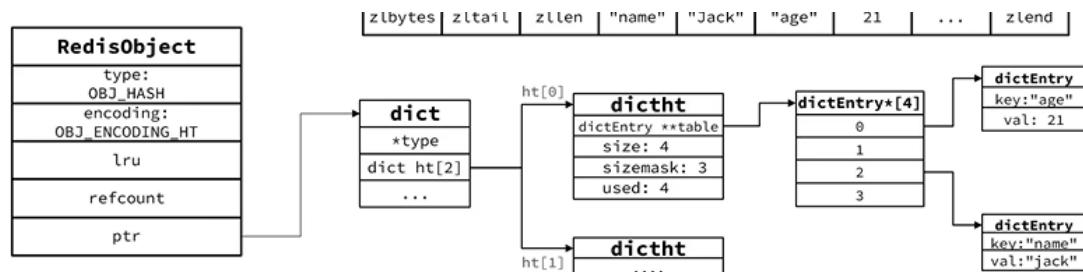


Redis的Hash数据类型的底层结构

- Hash结构默认采用ZipList编码，用以节省内存



- 当数据量较大时，Hash结构会转为HT编码，也就是Dict



参考资料

黑马程序员

- P40 - P43 (从穿透看到击穿)
- P97 - P100 (持久化)
- P155 - P159 (数据结构)