Go 进阶训练营
第8课
分布式缓存 & 分布式事务答疑

大明

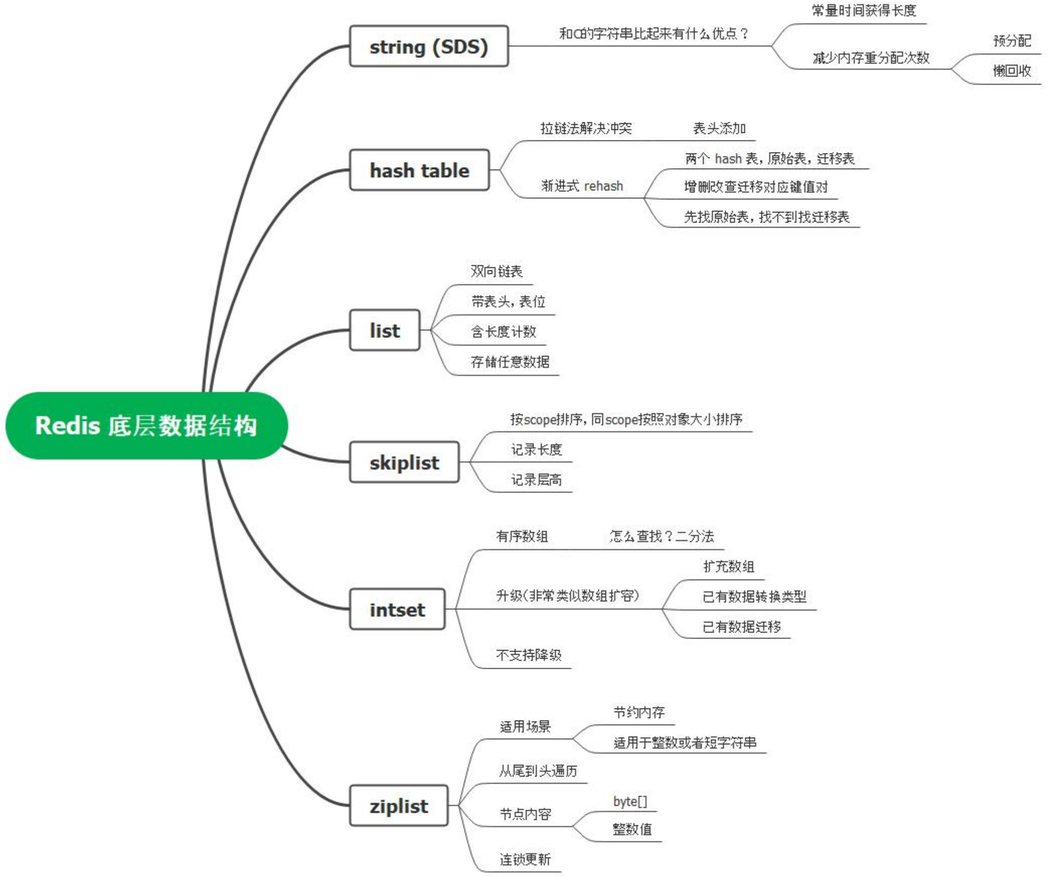
知识梳理 —— Redis基础

- Redis 数据结构
  - 底层数据结构
  - 值对象
- Redis 高性能、高可用

Redis 基础 —— 数据结构之底层实现

- 掌握每种底层实现的特点和优缺点（面试热点）
- 能从底层数据结构出发去分析实际问题

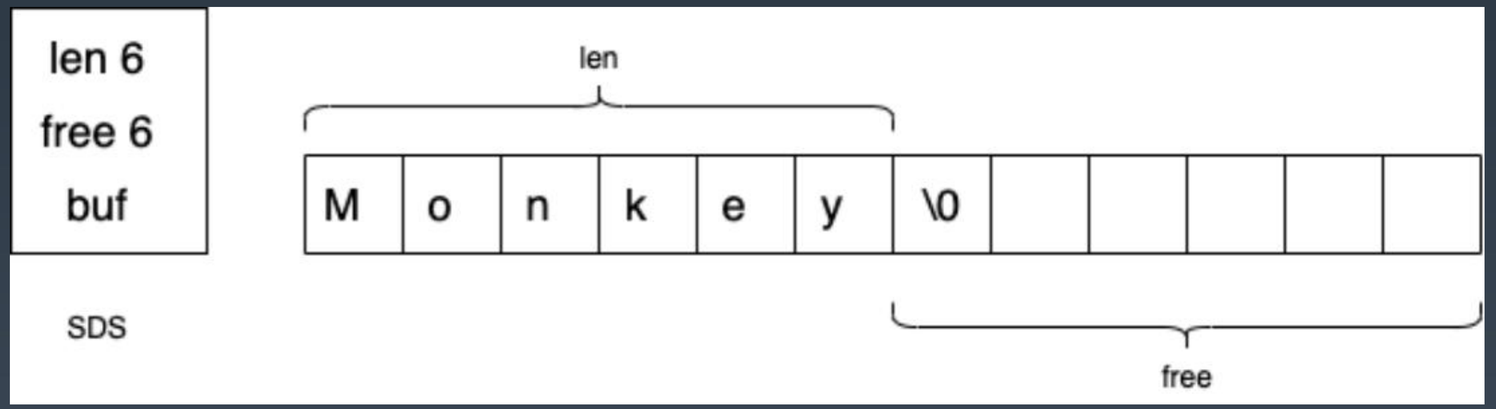
Redis 基础 —— 数据结构之底层实现



Redis 底层实现—— SDS

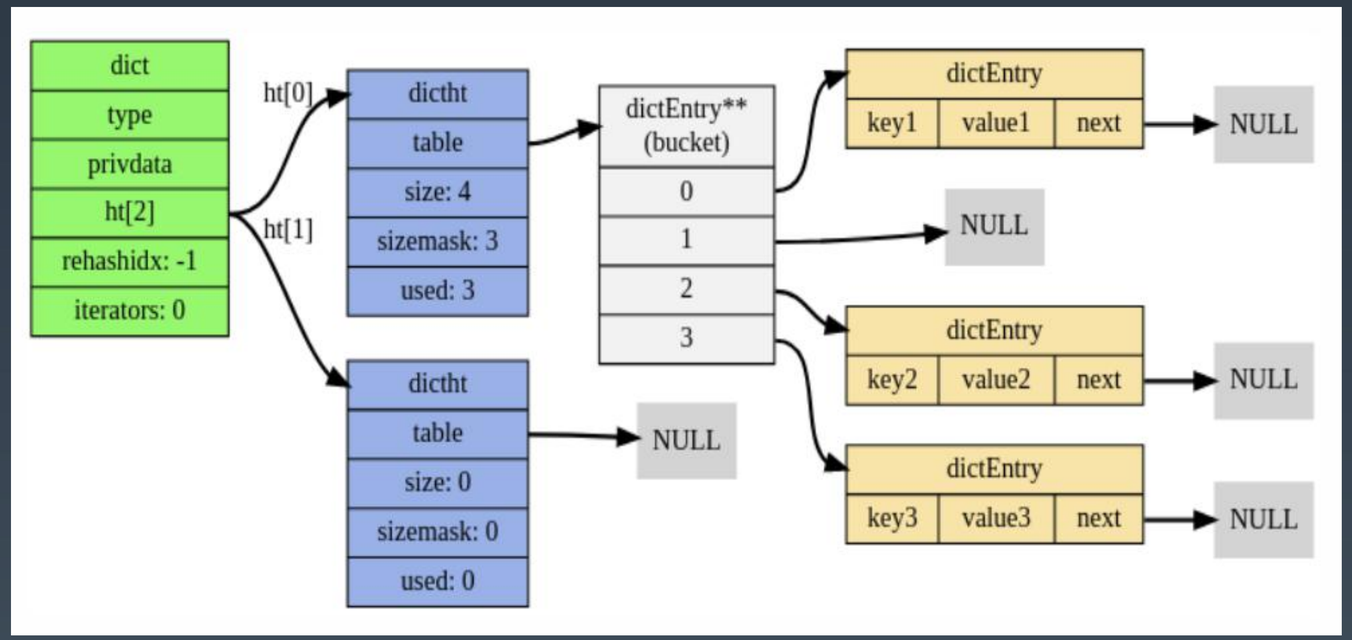
- 直接维持了字符串的长度
- 预分配，减少内存分配

- 预分配会带来额外的内存开销，但是大多数情况下不会成为一个问题



Redis 底层实现—— hash table

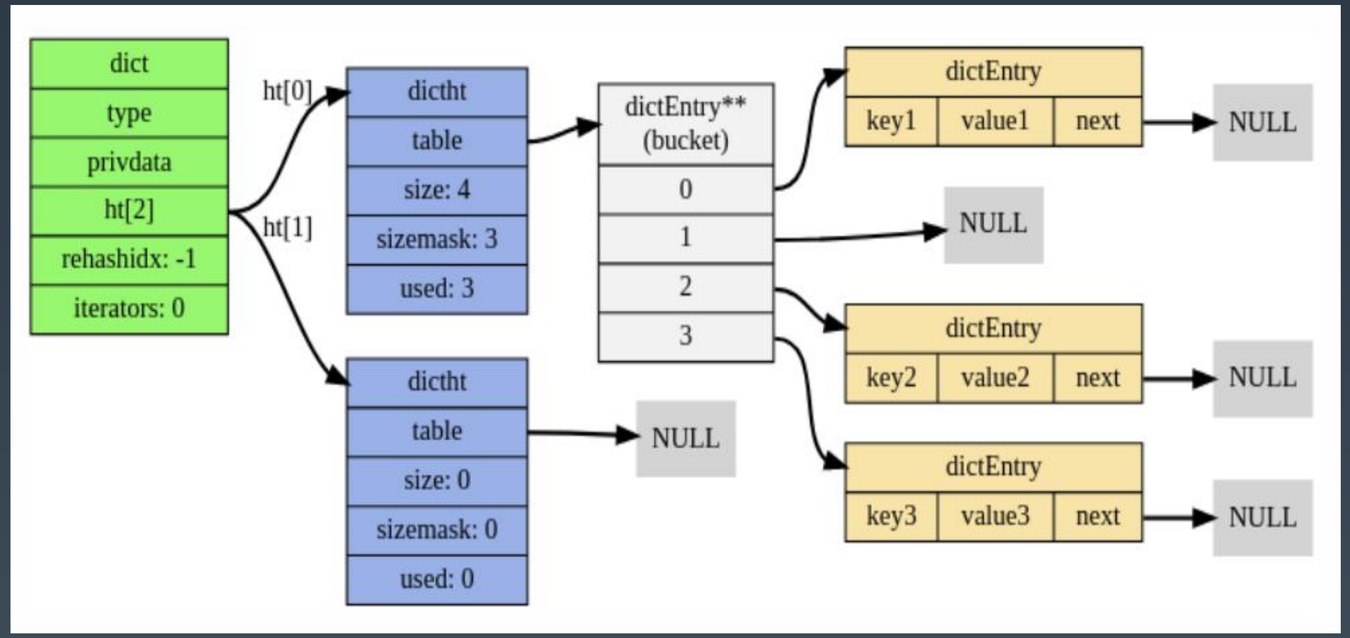
- 拉链法解决冲突：冲突节点放到头部
- 渐进式 rehash 扩容



Redis 底层实现—— rehash 过程

- 为 ht[1] 分配空间：扩容，则是当前长度向上第一个 2^n；缩容，则是当前长度向下 第一个 2^n；
- 将 ht[0] 中的元素重新计算分布到 ht[1]，改查会触发这个过程
- ht[0] 都迁移完之后，交换ht[0]和ht[1]

Redis 发现处于渐进式 rehash 过程中，会首先查找 ht[0]，其次查找 ht[1]



Redis 底层实现—— ziplist

- zlbytes：ziplist使用字节数
- zltail：最后元素的偏移量
- zllen：元素个数
- entry：元素
- zlend：结束标记
- 特点：内存是连续的，形似数组，可以理解为元素大小不定的数组

- 查： O(N)
- 增删：平均 O(N)，最坏 O(N^2)

Redis 底层实现—— ziplist

entry ：

- previous_entry_length：前一个 entry 的长度，一个字节或者五个字节
- encoding：数据类型和长度
- content：节点数据

previous_entry_len	encoding	content

个字节之间用空格隔开。

表 7-2　字节数组编码

编　　码	编码长度	content 属性保存的值
00bbbbbb	1 字节	长度小于等于 63 字节的字节数组
01bbbbbb xxxxxxxx	2 字节	长度小于等于 16 383 字节的字节数组
10______ aaaaaaaa bbbbbbbb cccccccc dddddddd	5 字节	长度小于等于 4 294 967 295 的字节数组

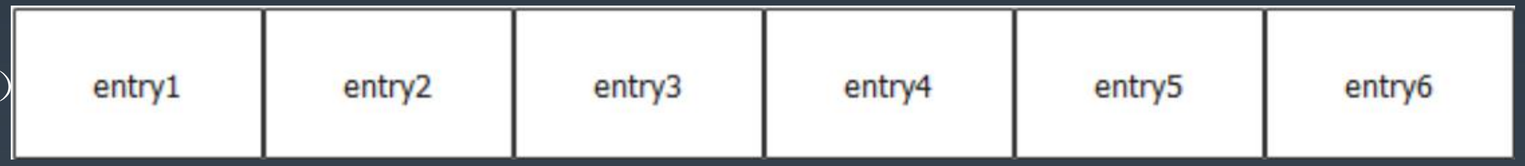
表 7-3　整数编码

编码	编码长度	content 属性保存的值
11000000	1 字节	int16_t 类型的整数
11010000	1 字节	int32_t 类型的整数
11100000	1 字节	int64_t 类型的整数
11110000	1 字节	24 位有符号整数
11111110	1 字节	8 位有符号整数
1111xxxx	1 字节	使用这一编码的节点没有相应的 content 属性，因为编码本身的 xxxx 四个位已经保存了一个介于 0 和 12 之间的值，所以它无须 content 属性

Redis 底层实现—— ziplist

连锁更新 ：因为记录了前一个节点的长度，那么如果前一个节点的长度发生变化，会导致当前节点长度变化，又引起下一个节点的长度变化

- previous_entry_length：一个字节（<254长度）或者五个字节

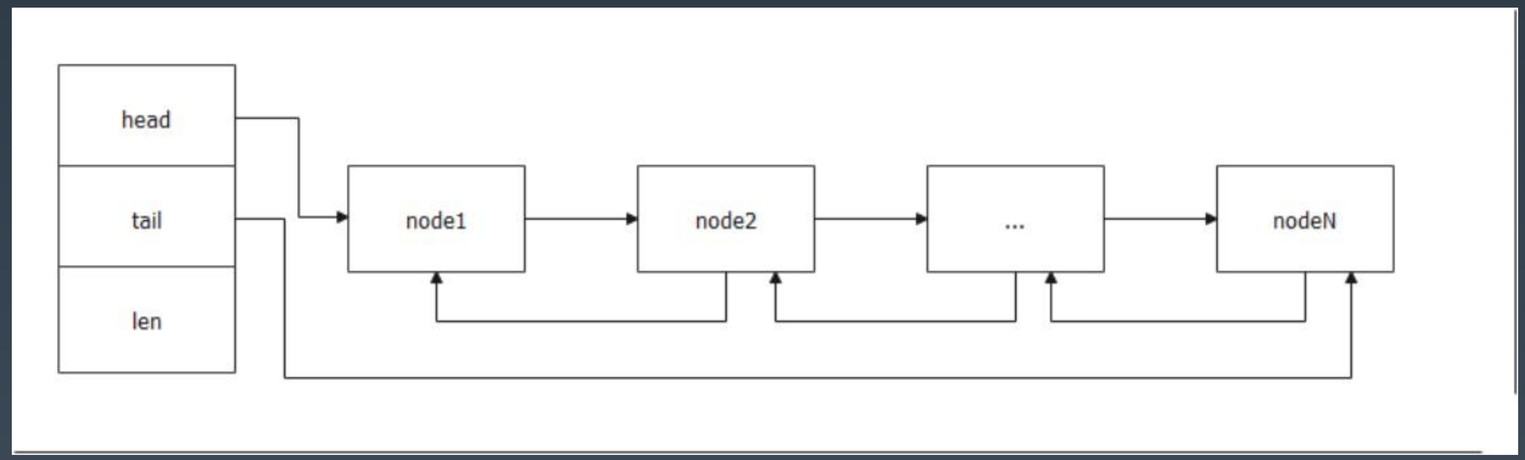


假设 entry 1 ~ 6 的长度都是 253。当entry1 之前插入了一个长度为 255的节点，那么entry1 的previous_entry_length要从一个字节扩展到五个字节，于是 entry1 长度变成了 257。

因为 entry1 此时长度已经超过 254，所以entry2 也要更新previous_entry_length为 五个字节，于是又引起了 entry3 更新

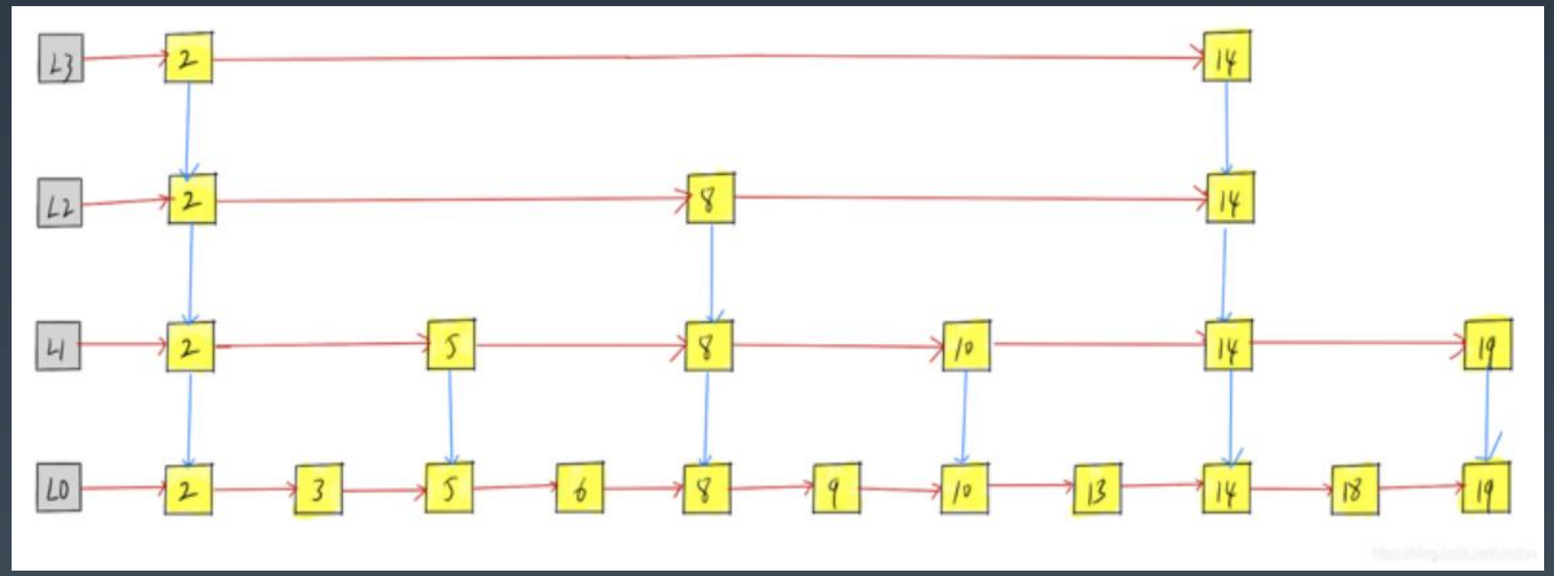
Redis 底层实现—— linkedlist

- 双向链表
- 直接维持了长度



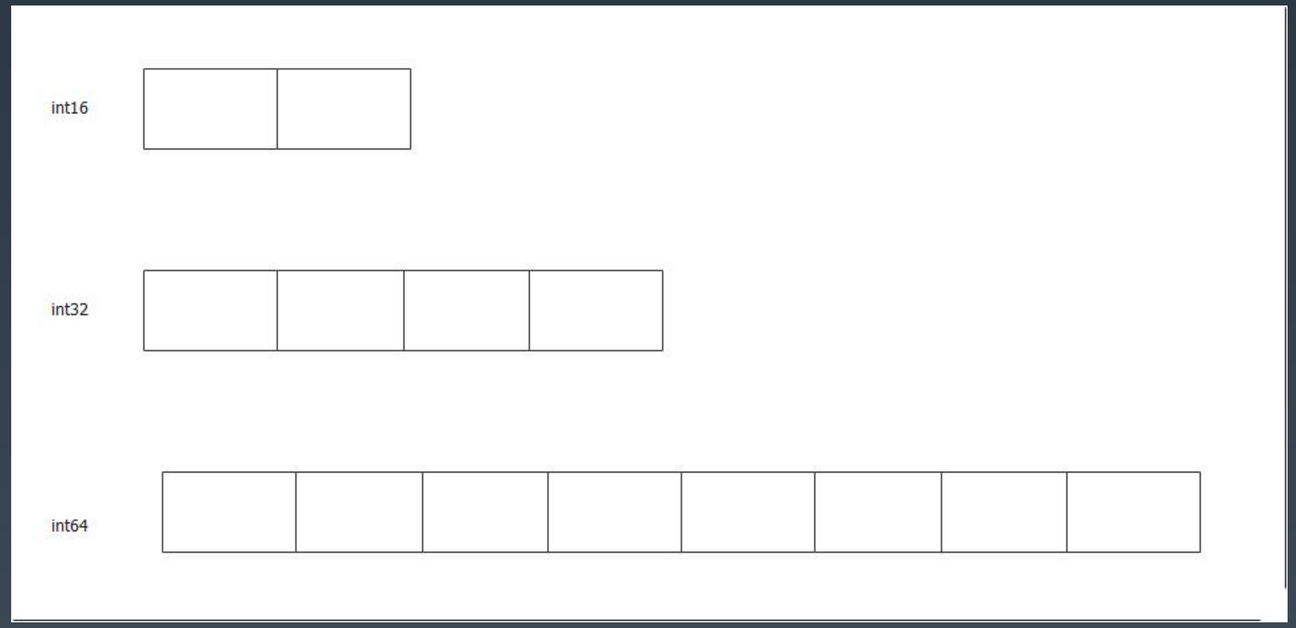
Redis 底层实现—— skiplist

- 跳表和平衡树性能相当，但是实现要简单很多
- 按照 scope 排序
- 记录了 header, tail, 高度（深度）和 长度



Redis 底层实现—— intset

- 用字节数组来存放数据
- 数据有序存放
- 可以存放 int16（2个字节）, int32（4个字节）, int64（八个字节） 三种数据
- 直接存放了元素个数



- 如果数字升级，那么需要全部元素升级一遍

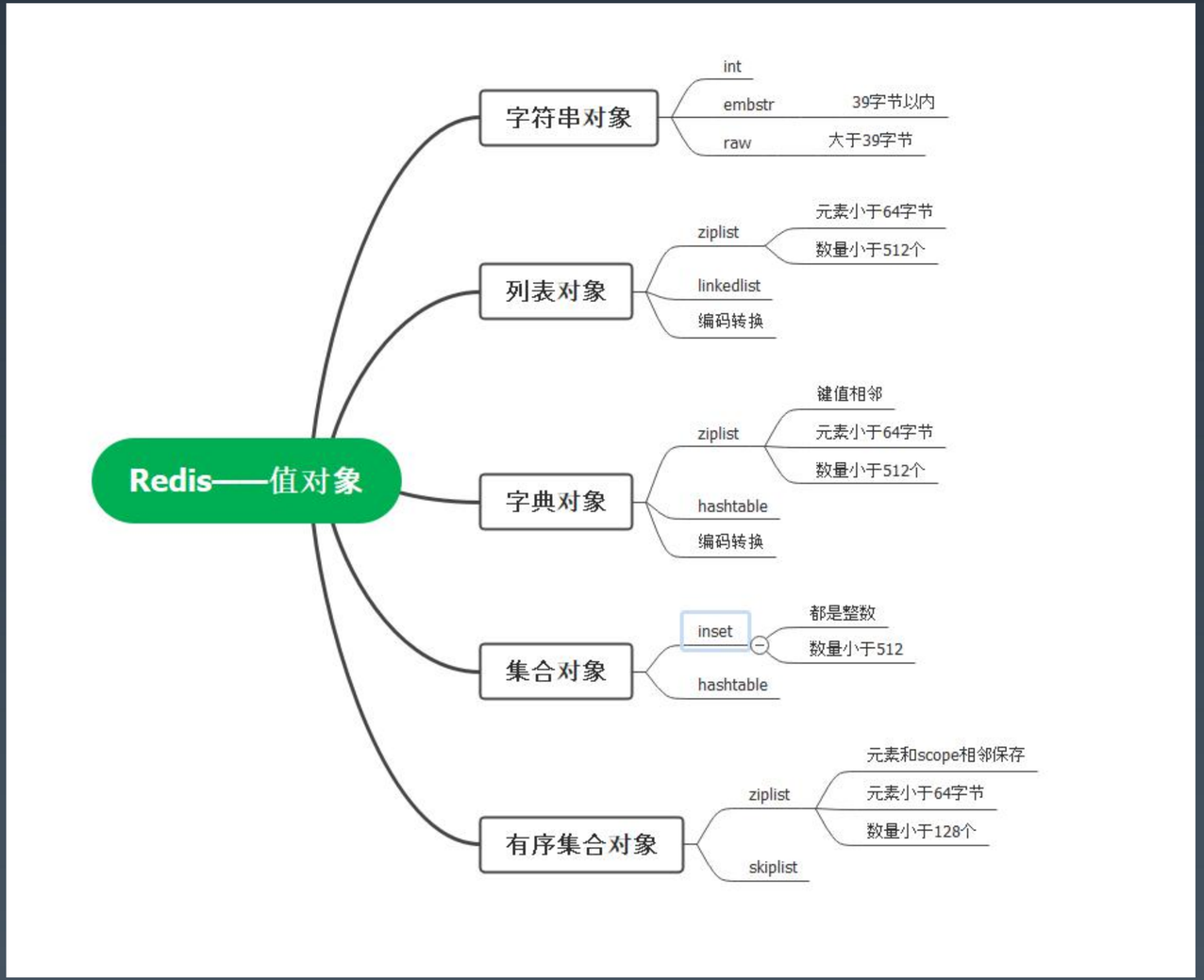
知识梳理 —— Redis基础

- Redis 数据结构
  - 底层数据结构
  - 值对象
- Redis 高性能、高可用
- Redis 和 Memcache 对比

Redis 基础 —— 数据结构之值对象

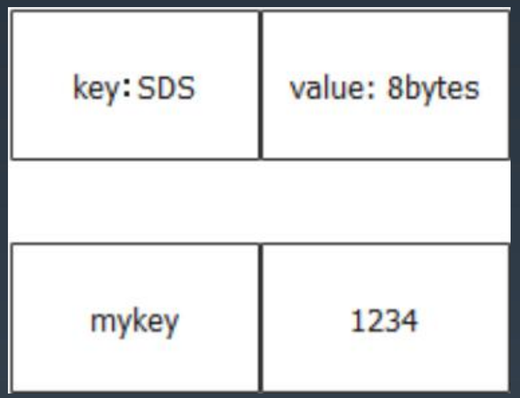
- 掌握不同类型的使用场景（面试热点）
- 掌握不同类型的底层实现是什么（面试热点）
- 掌握触发编码转换的条件

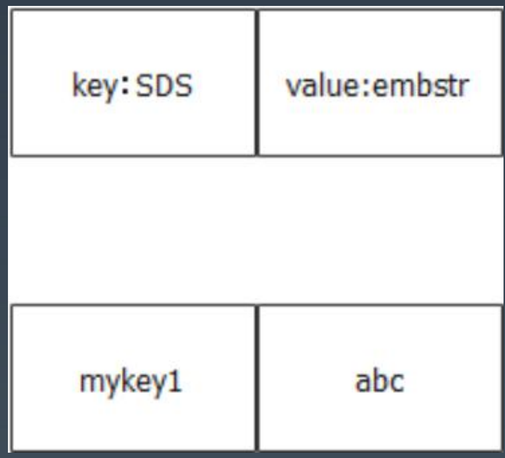
Redis 基础 —— 数据结构之值对象

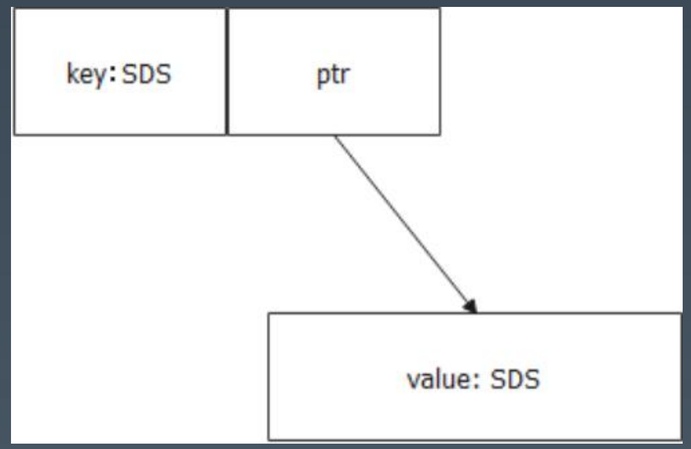


Redis 基础 —— 字符串

- int 编码：值可以用 64 位有符号整形表示，也就是 key-value 中的 value 不再是一个 redisObject
- embstr 编码：只读，长度小于 39 字节的字符串使用该编码。3.2 版本之后小于 44 字节的会采用该编码。key 和 value 共享连续64 字节，value 就是 embstr
- raw：其实就是 SDS 编码了







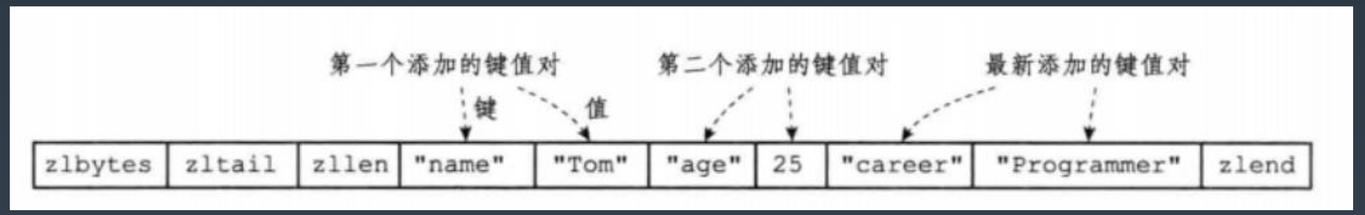
Redis 基础 —— 列表对象

- ziplist 编码 和 linkedlist 编码

- 使用 ziplist 的条件：
  - 列表对象保存的所有字符串元素的长度都小于 64 字节；
  - 列表对象保存的元素数量小于 512 个；

Redis 基础 —— 字典对象

- ziplist 编码 和 hashtable 编码



- 使用 ziplist 的条件：
  - 列表对象保存的所有键和值的长度都小于 64 字节；
  - 列表对象保存的元素数量小于 512 个；

Redis 基础 —— 集合对象

- intset 编码 和 hashtable 编码

- 使用 intset 的条件：
  - 都是整数；
  - 元素数量小于 512 个；

因为 Hash 结构内存利用率不高，所以很多时候我们会考虑用有序数组的结构来实现类似的功能

Redis 基础 —— 有序集合对象

- ziplist 编码 和 skiplist 编码
- 额外维护了一个哈希结构，用于在o(1)内找到 scope
- 所以有两种组合： ziplist + 哈希结构 或者 skiplist + 哈希结构

- 使用 ziplist 的条件：
  - 所有元素长度小于64字节；
  - 元素数量小于 128个；

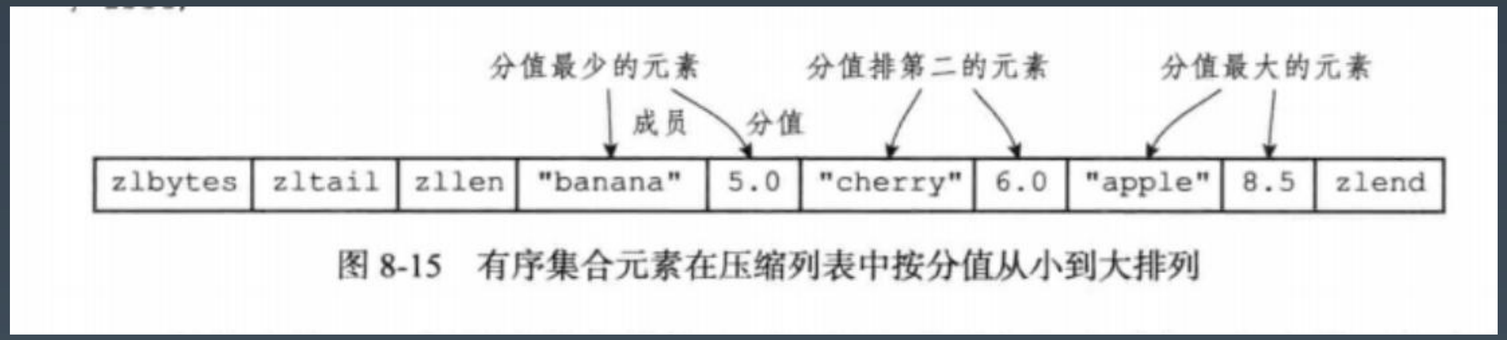


图 8-15　有序集合元素在压缩列表中按分值从小到大排列

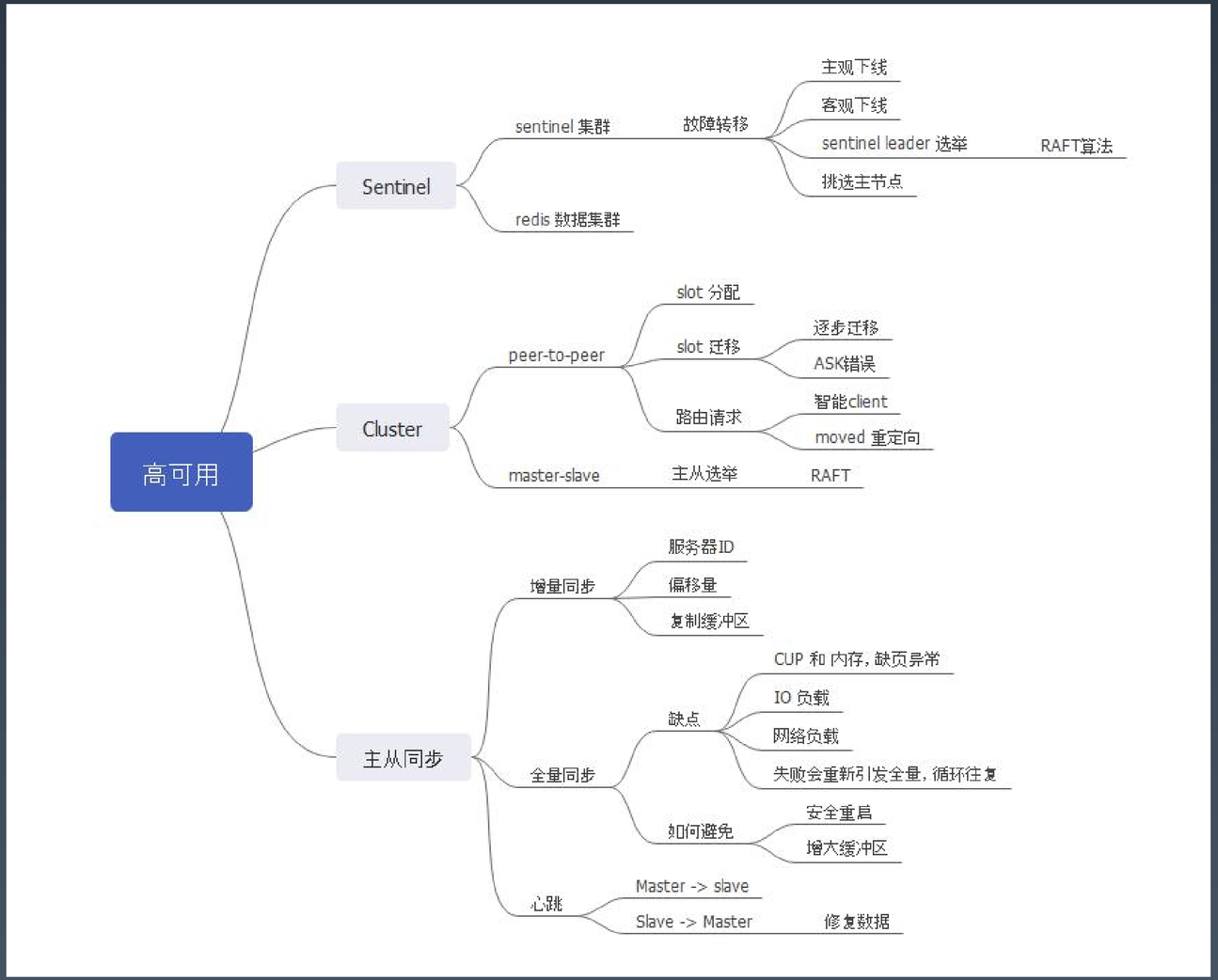
知识梳理 —— Redis基础

- Redis 数据结构
  - 底层数据结构
  - 值对象
- Redis 高性能、高可用
  - Redis Sentinel
  - Redis Cluster
  - Redis 主从同步
  - Redis 持久化
  - Redis IO 模型

Redis高可用

Redis 高可用有两种模式，Sentinel 和 Cluster：

- Sentinel 本质上是主从模式，与一般的主从模式不同的是，主节点的选举，不是从节点完成的，而是通过 Sentinel 来监控整个集群模式，发起主从选举。因此本质上 Redis Sentinel 有两个集群，一个是 Redis 数据集群，一个是哨兵（Sentinel）集群。
- Redis Cluster 集成了对等模式和主从模式。Redis Cluster 由多个节点组成，每个节点都可以是一个主从集群。Redis 将 key 映射为 16384 个槽（slot），均匀分配在所有节点上
- 两种模式下的主从同步都有全量同步和增量同步两种



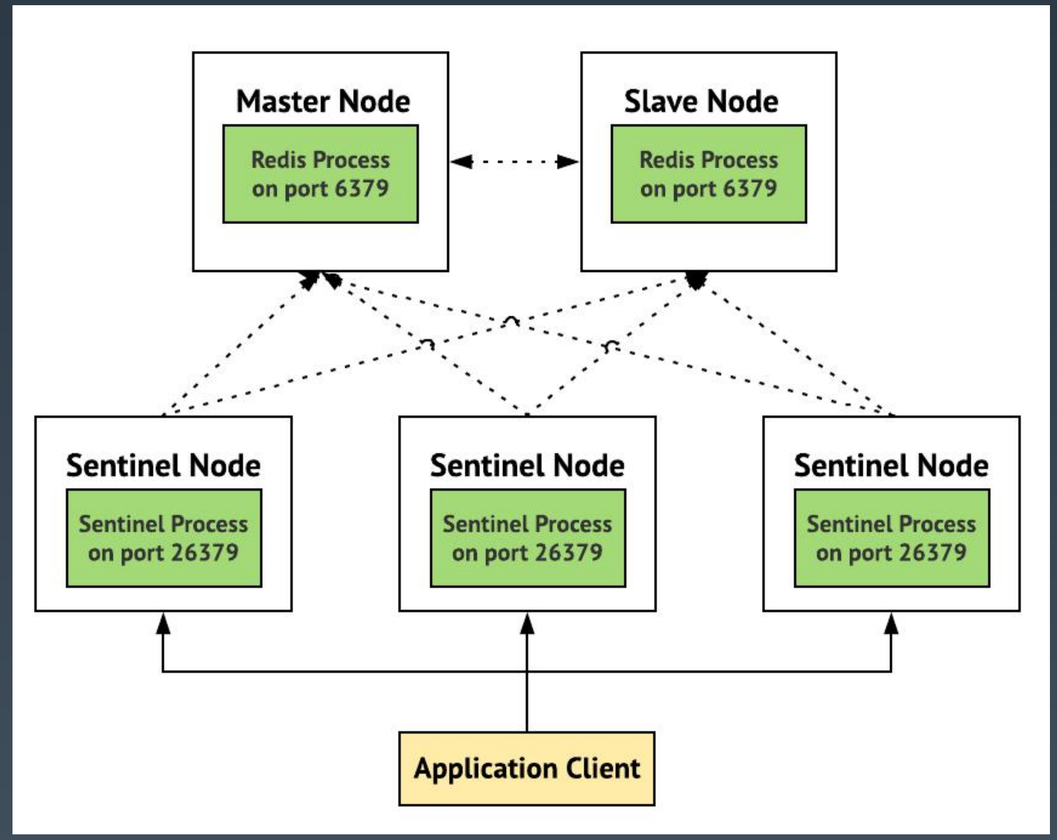
知识梳理 —— Redis基础

- Redis 数据结构
  - 底层数据结构
  - 值对象
- Redis 高性能、高可用
  - Redis Sentinel
  - Redis Cluster
  - Redis 主从同步
  - Redis 持久化
  - Redis IO 模型

Redis高可用——Sentinel 监控

三个步骤：主观下线 -> 客观下线 -> 主节点故障转移：

1. 首先 Sentinel 获取了主从结构的信息，而后向所有的节点发送心跳检测，如果这个时候发现某个节点没有回复，就把它标记为主观下线
2. 如果这个节点是主节点，那么 Sentinel 就询问别的 Sentinel 节点主节点信息。如果大多数都 Sentinel 都认为主节点已经下线了，就认为主节点已经客观下线
3. 当主节点已经客观下线，就要步入故障转移阶段。故障转移分成两个步骤，一个是 Sentinel 要选举一个 leader，另外一个步骤是 Sentinel leader 挑一个主节点

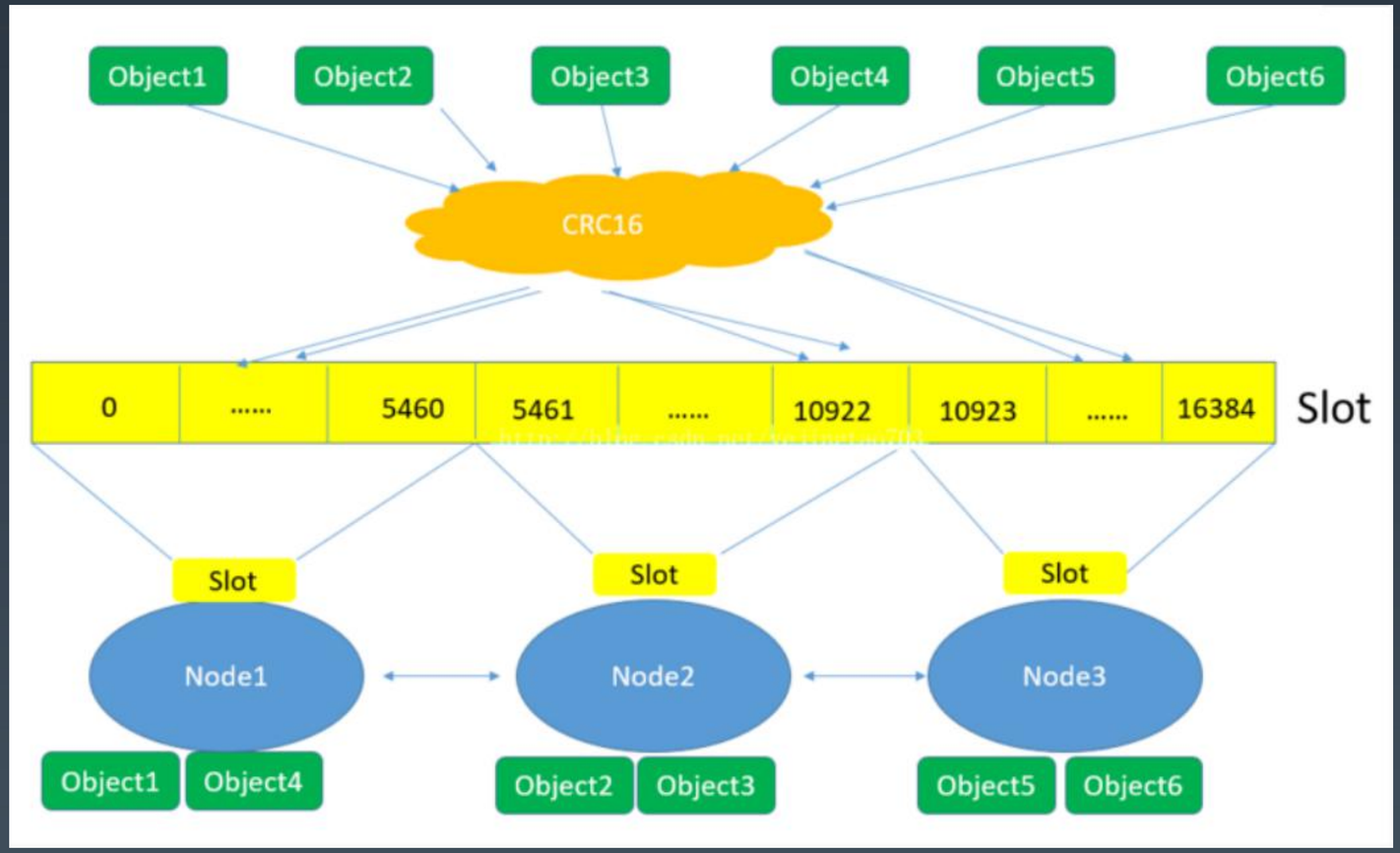


知识梳理 —— Redis基础

- Redis 数据结构
  - 底层数据结构
  - 值对象
- Redis 高性能、高可用
  - Redis Sentinel
  - Redis Cluster
  - Redis 主从同步
  - Redis 持久化
  - Redis IO 模型

Redis高可用——Redis Cluster

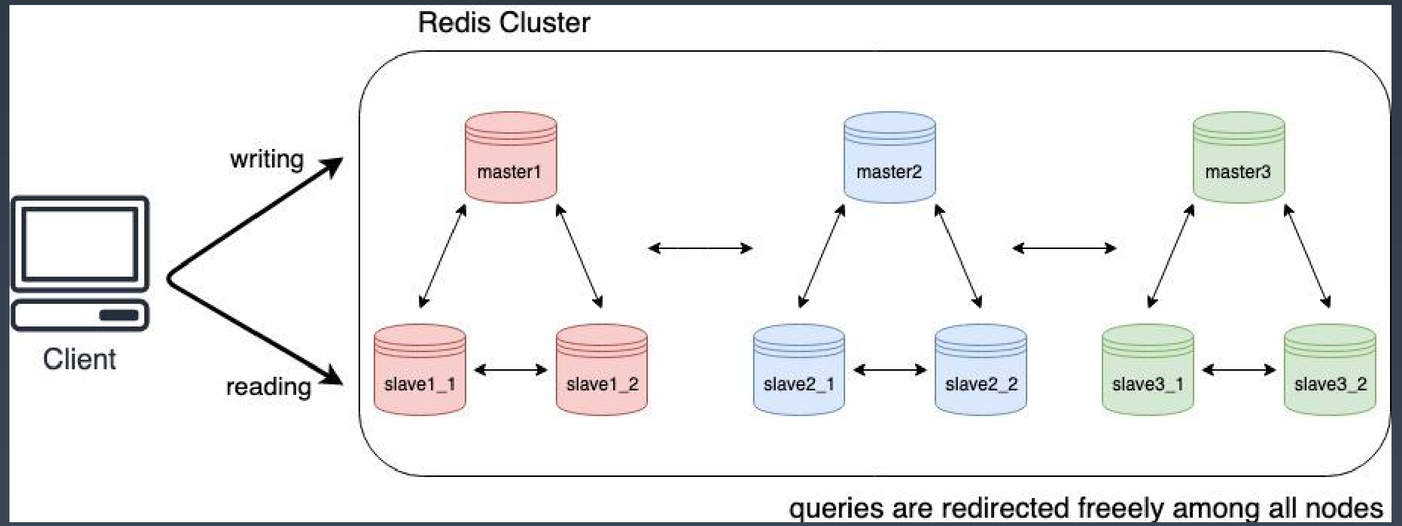
Redis Cluster 主要是利用 key 的哈希值，将其分成 16384 个槽，而后每个槽被分配到不同的主从集群上



Redis高可用——Redis Cluster

Redis Cluster 是peer-to-peer，每个节点都能提供读写服务。如果客户端请求的某个 key 不在该服务器上，该服务器就会返回一个 move 错误，让客户端再一次请求正确的服务器。

因此有所谓的智能客户端直接维护了槽到节点的映射关系。

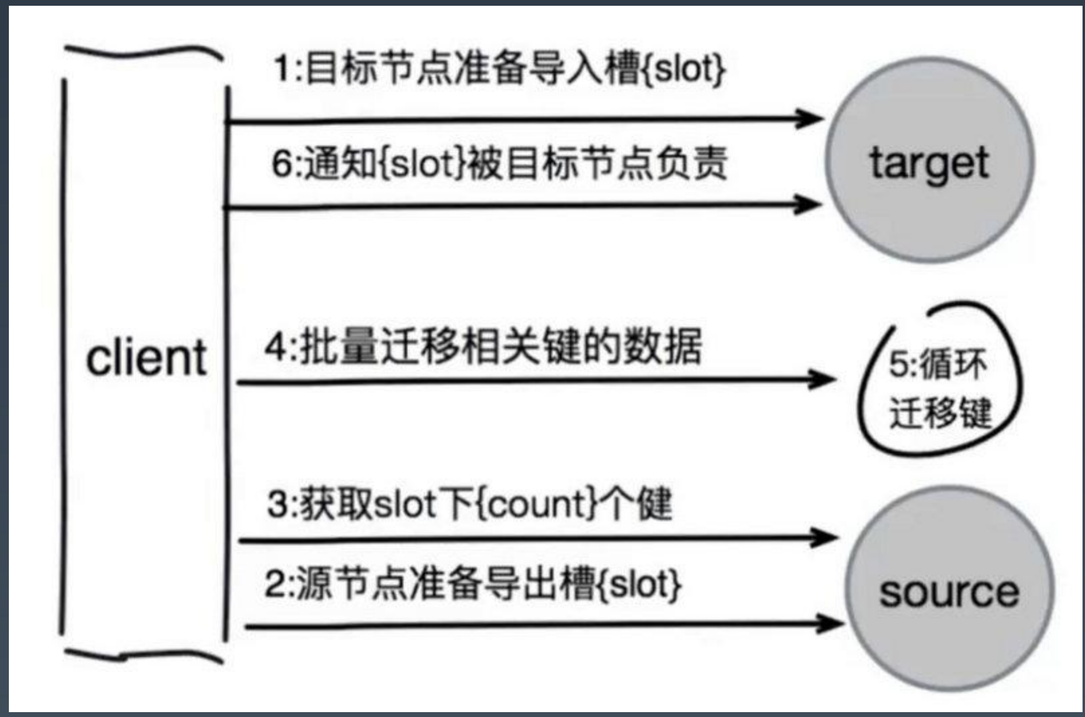


Redis高可用——Redis Cluster

重分片的时候，会触发槽迁移，也就是把一部分数据挪到另外一个部分。

这个步骤是渐进式的

在迁移过程中，一个槽的部分 key 可能在源节点，一部分在目标节点。因此如果请求过来，打到源节点，源节点发现已经迁移了，就会返回一个 ASK 错误，这个错误会引导客户端直接去访问目标节点。



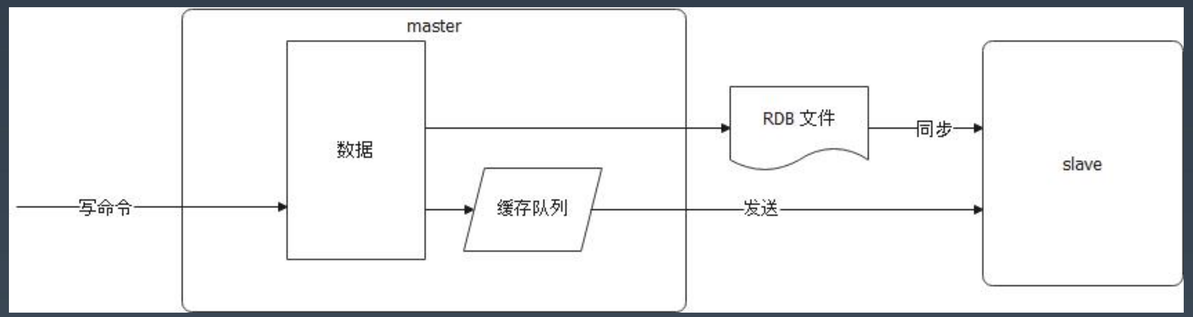
知识梳理 —— Redis基础

- Redis 数据结构
  - 底层数据结构
  - 值对象
- Redis 高性能、高可用
  - Redis Sentinel
  - Redis Cluster
  - Redis 主从同步
  - Redis 持久化
  - Redis IO 模型

Redis高可用——主从同步

主从同步有全量同步和增量同步两种，全量同步：

- 从服务器发起同步，主服务器开启 BG SAVE，生成 BG SAVE 过程中的写命令也会被放入一个缓冲队列；
- 主节点生成 RDB 文件之后，将 RDB 发给从服务器；
- 从服务器接收文件，清空本地数据，再入 RDB 文件；（这个过程会忽略已经过期的 key，参考过期部分的讨论）
- 主节点将缓冲队列命令发送给从节点，从节点执行这些命令；
- 从节点重写 AOF；
- 主节点源源不断发送新的命令；

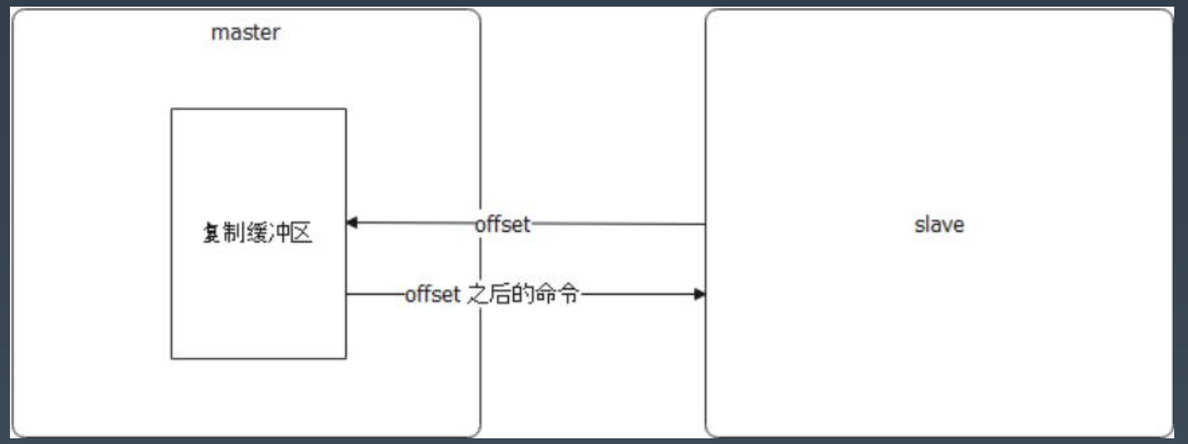


Redis高可用——增量同步

- 全量同步非常重，资源消耗很大
- 大多数情况下，从服务器上是存在大部分数据的，只是短暂失去了连接
- 如果这个时候又发起全量同步，那么很容易陷入到无休止的全量同步之中。

增量同步的依赖于三个东西：

1. 服务器ID：用于标识 Redis 服务器ID；
2. 复制偏移量：主服务器用于标记它已经发出去多少；从服务用于标记它已经接收多少（从服务器的比较关键）；
3. 复制缓冲区：主服务器维护的一个 1M 的FIFO队列，近期执行的写命令保存在这里；



知识梳理 —— Redis基础

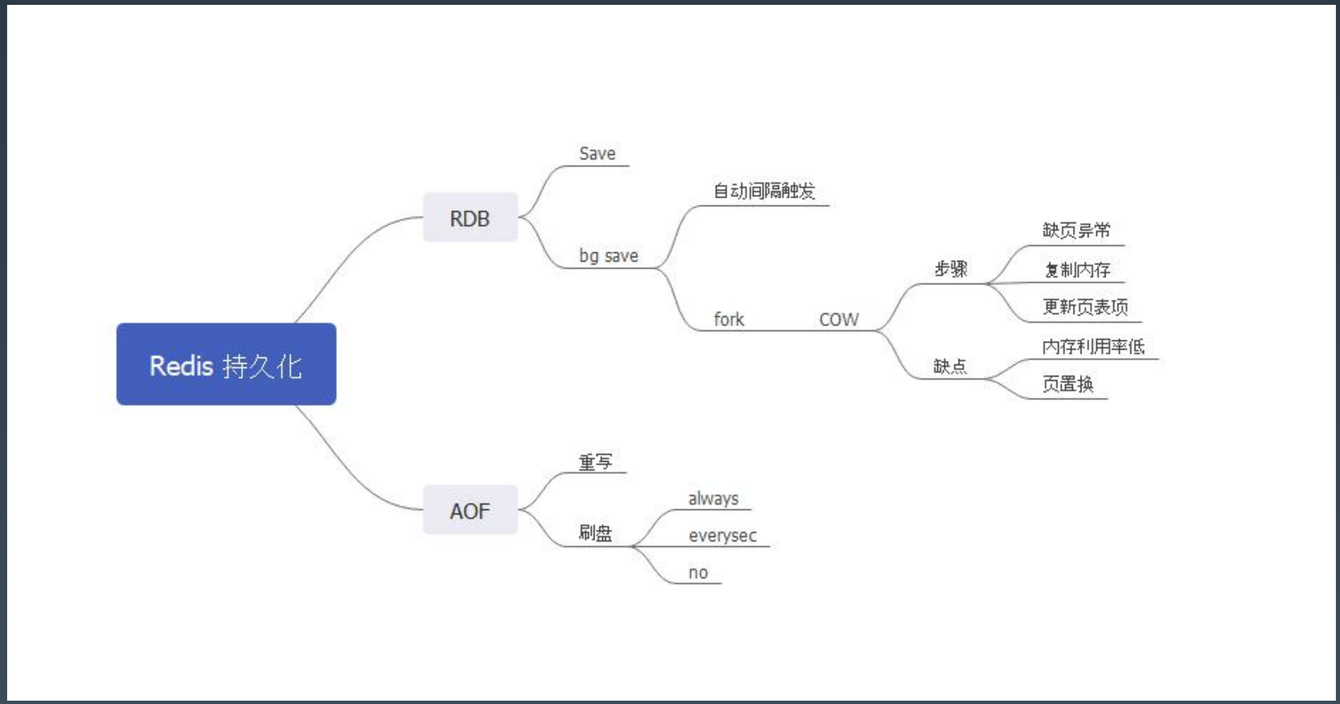
- Redis 数据结构
  - 底层数据结构
  - 值对象
- Redis 高性能、高可用
  - Redis Sentinel
  - Redis Cluster
  - Redis 主从同步
  - Redis 持久化
  - Redis IO 模型

Redis高可用——Redis持久化

Redis 的持久化机制分成两种，RDB 和 AOF。RDB 也是主从全量同步里的 RDB。

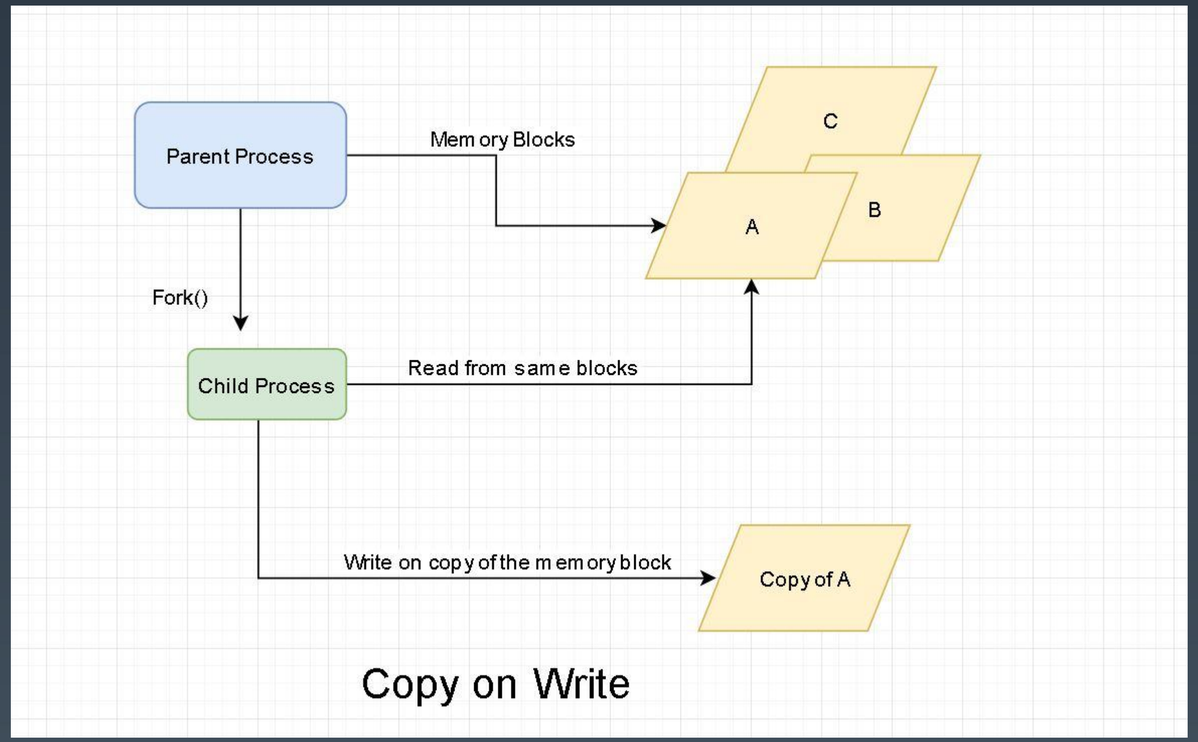
RDB 可以理解为是一个快照，直接把 Redis 内存中的数据以快照的形式保存下来。因为这个过程很消耗资源，所以分成 SAVE 和 BG SAVE 两种。BG SAVE的核心是利用 fork 和 COW 机制。

AOF 是将 Redis 的命令逐条保留下来，而后通过重放这些命令来复原。我们可以通过重写 AOF 来减少资源消耗。



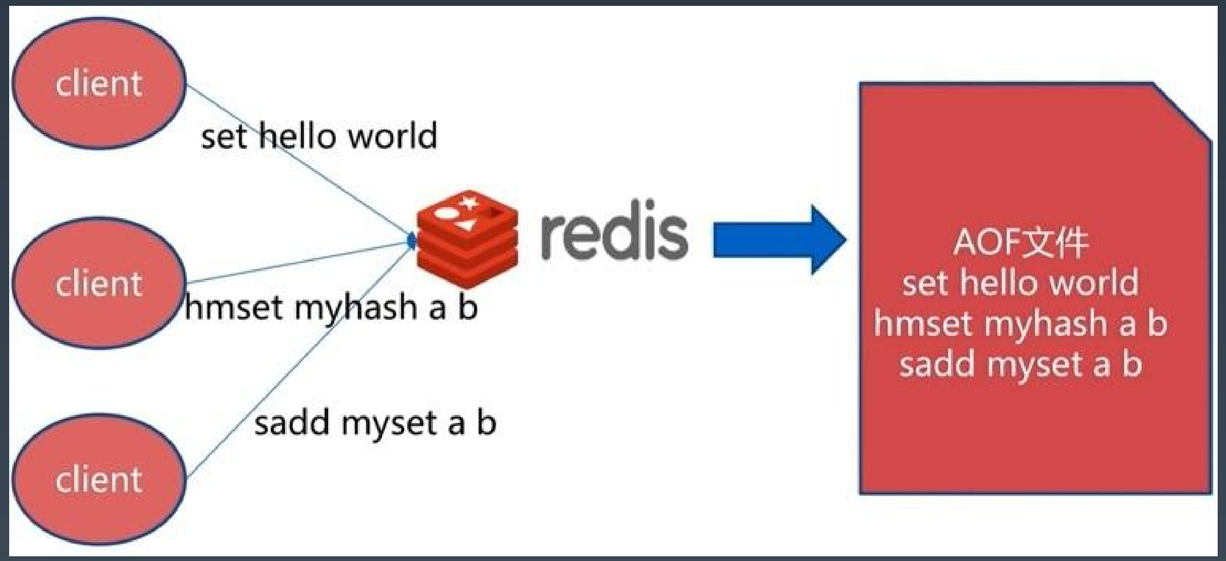
Redis高可用——Redis BG Save

- 利用`fork`系统调用，复制出来一个子进程
- 子进程尝试将数据写入文件。这个时候，子进程和主进程是共享内存的，当主进程发生写操作，那么就会复制一份内存，这就是所谓的 COW。
- COW 的核心是利用缺页异常，操作系统在捕捉到缺页异常之后，发现他们共享内存了，就会复制出来一份。



Redis高可用—— AOF

- 逐条记录命令
- AOF 刷新磁盘的时机
  - always: 每次都刷盘
  - everysec: 每秒，这意味着一般情况下会丢失一秒钟的数据。而实际上，考虑到硬盘阻塞（见后面**使用 everysec 输盘策略有什么缺点），那么可能丢失两秒的数据。
  - no: 由操作系统决定
- 可以通过重写来合并 AOF 文件



Redis高可用—— AOF 刷盘对比

AOF 刷新磁盘的时机

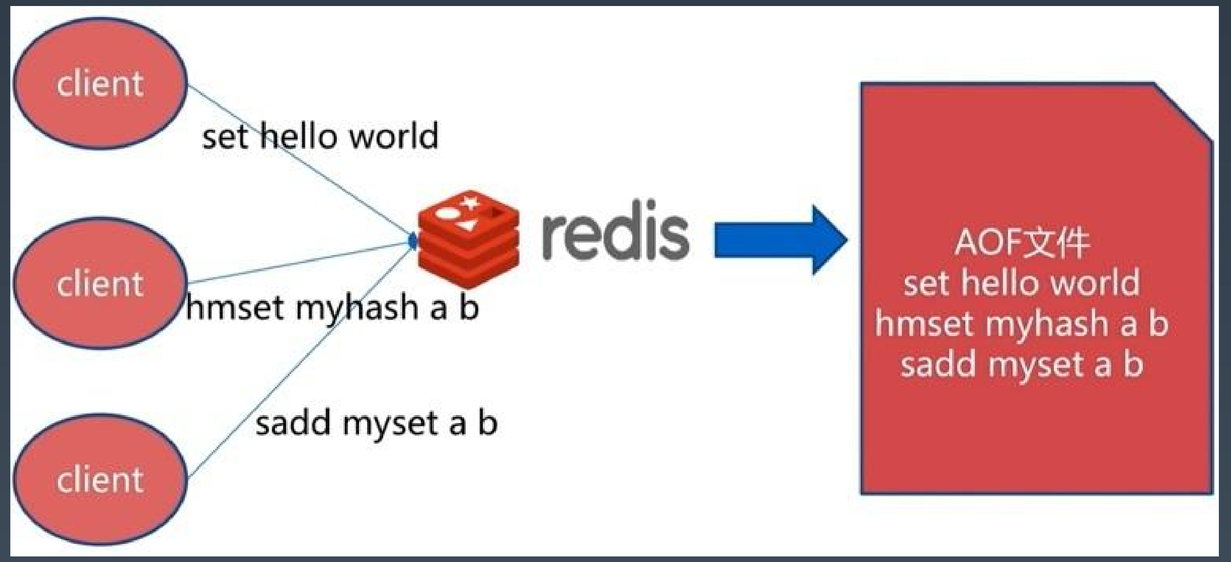
- always: 每次都刷盘
- everysec: 每秒，这意味着一般情况下会丢失一秒钟的数据。而实际上，考虑到硬盘阻塞（见后面**使用 everysec 输盘策略有什么缺点），那么可能丢失两秒的数据。
- no: 由操作系统决定

MySQL redo log 刷盘：

- 写到 log buffer , 每秒刷新；
- 实时刷新；
- 写到 OS cache, 每秒刷新

MySQL bin log 刷盘：

- 系统自由判断
- commit刷盘
- 每N个事务刷盘



写入语义

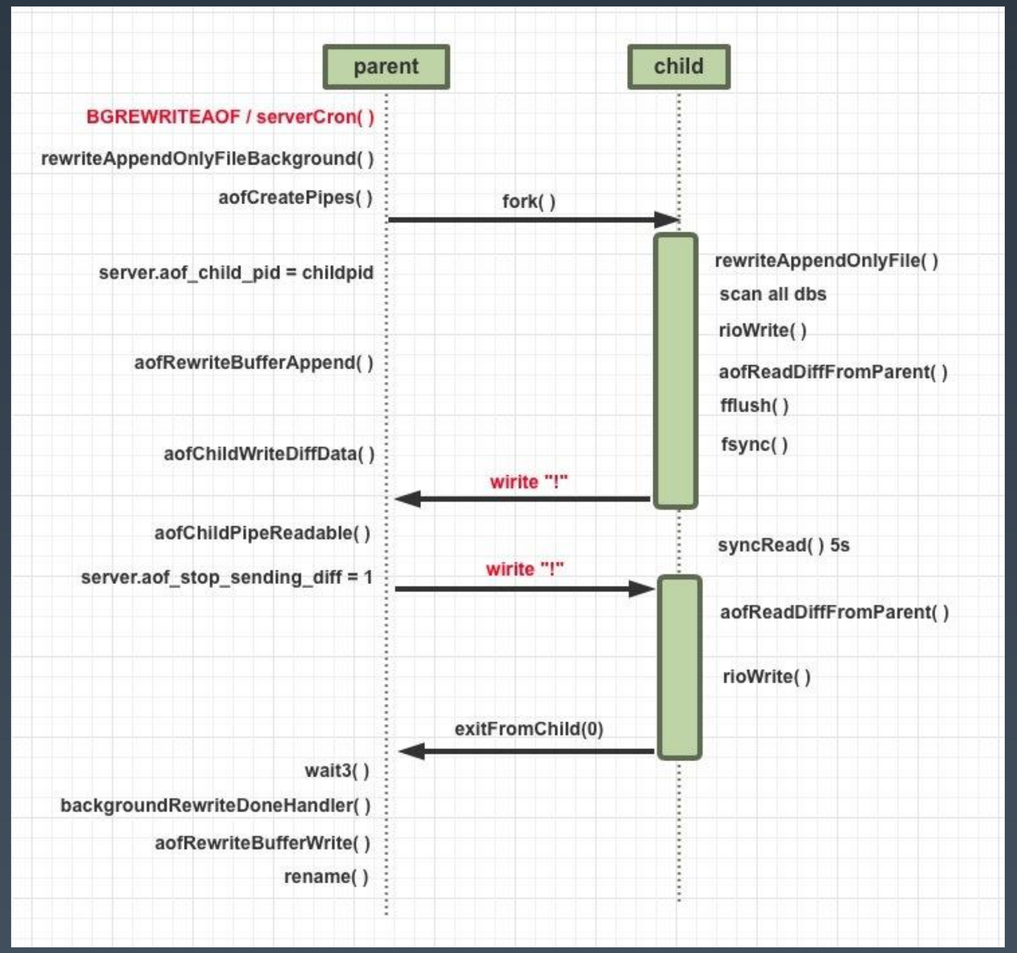
1. 中间件写到日志缓存就认为写入了；
2. 中间件写入到系统缓存（page cache）就认为写入了；
3. 中间件强制刷新到磁盘（发起了 fsync）就认为写入了；

Redis高可用—— AOF 重写

重写 AOF 整体类似于 RDB。

它并不是读已经写好的 AOF 文件，然后合并。而是类似于 RDB，直接fork出来一个子进程，子进程按照当前内存数据生成一个 AOF 文件。

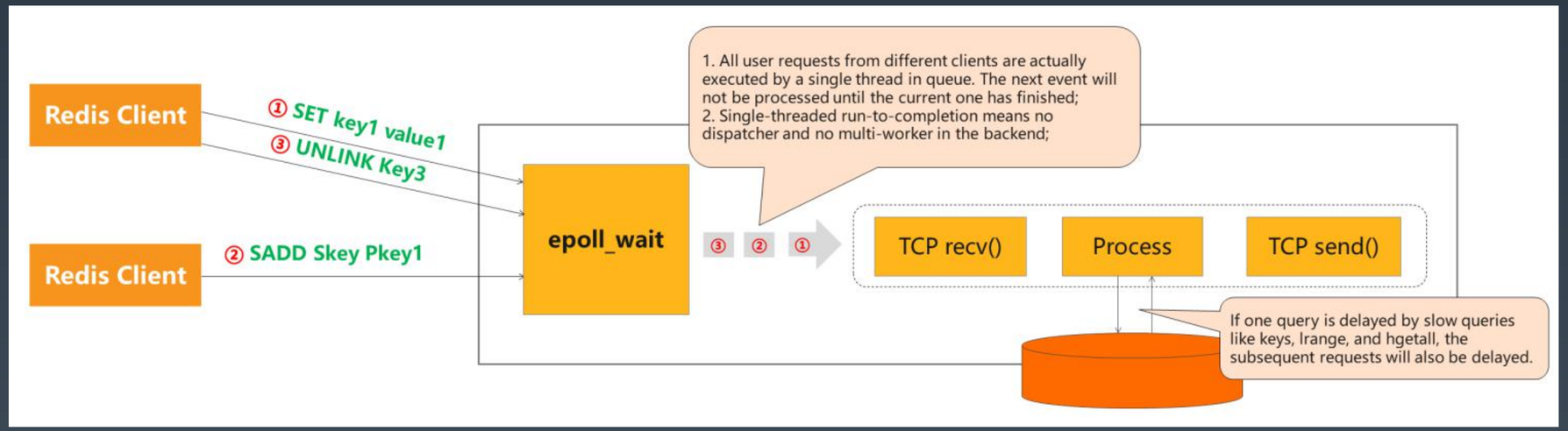
在这个过程中，Redis 还在源源不断执行命令，这部分命令将会被写入一个 AOF 的缓存队列里面。当子进程写完 AOF 之后，发一个信号给主进程，主进程负责把缓冲队列里面的数据写入到新 AOF。而后用新的 AOF 替换掉老的 AOF。



知识梳理 —— Redis基础

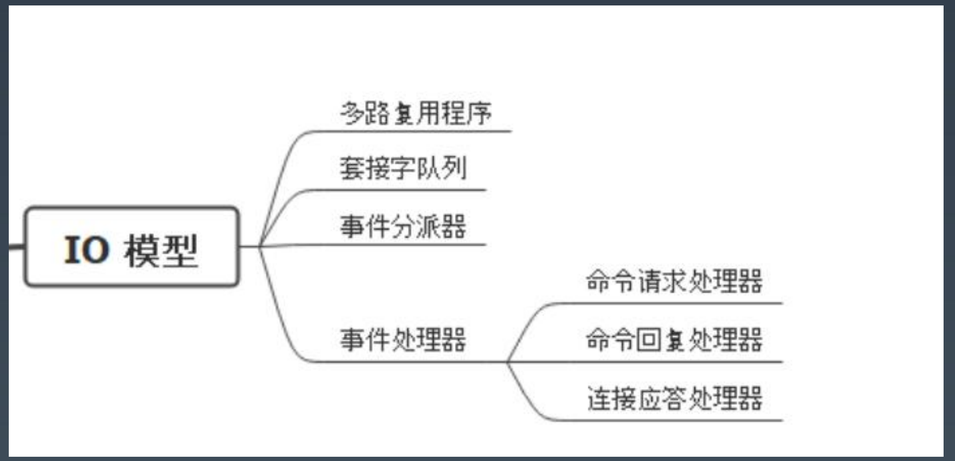
- Redis 数据结构
  - 底层数据结构
  - 值对象
- Redis 高性能、高可用
  - Redis Sentinel
  - Redis Cluster
  - Redis 主从同步
  - Redis 持久化
  - Redis IO 模型

Redis性能 —— Redis IO 模型



Redis性能 —— Redis IO 模型

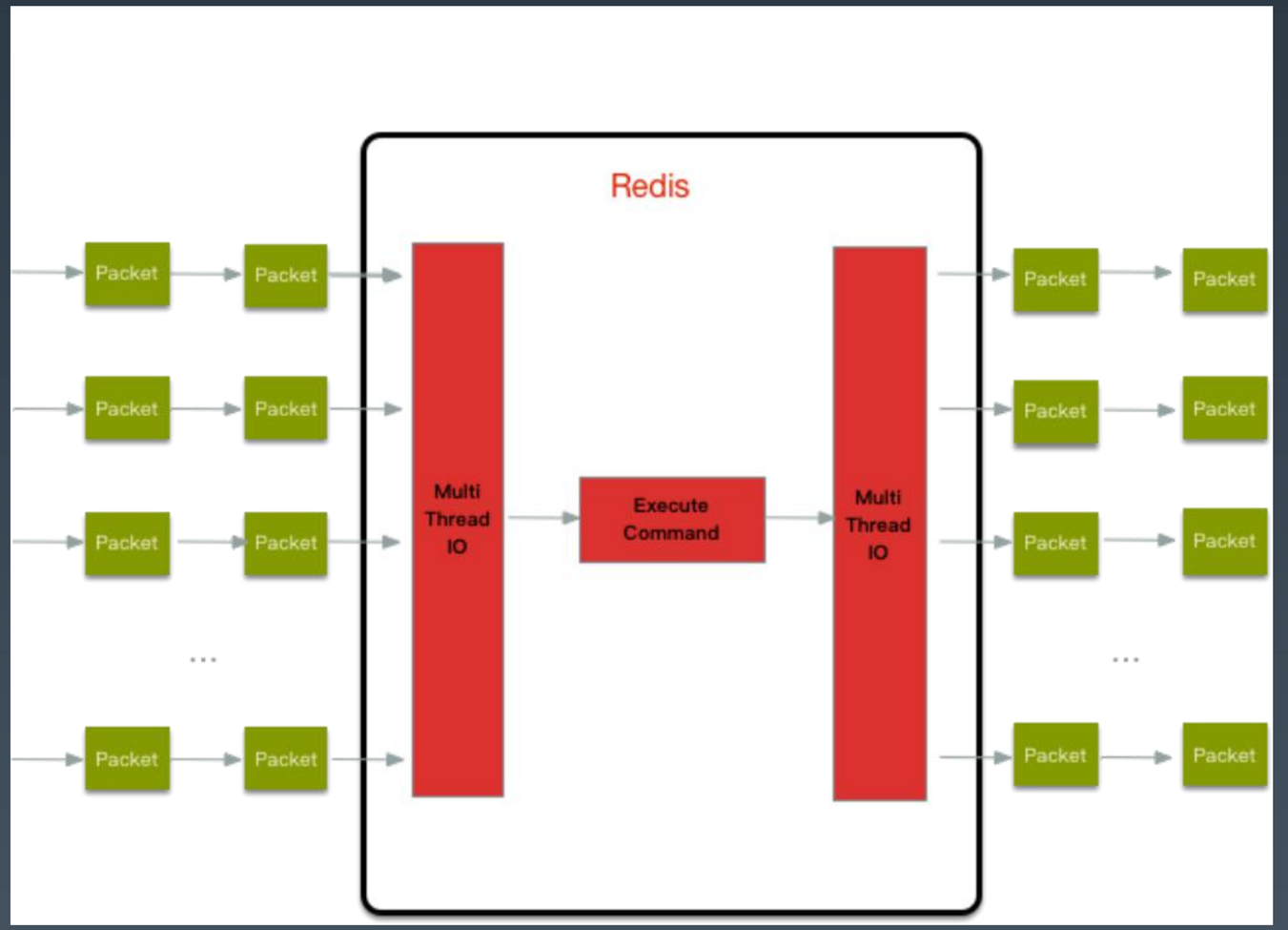
多路复用程序会监听不同套接字的事件，当某个事件，比如发来了一个请求，那么多路复用程序就把这个套接字丢过去套接字队列，事件分派器从队列里边找到套接字，丢给对应的事件处理器处理。



Redis性能 —— Redis IO 模型

两端读写数据和解析协议，都是多线程的

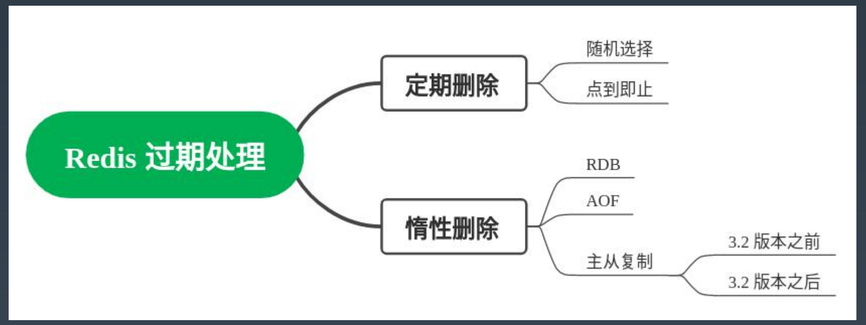
命令的执行，是单线程的



Redis基础——过期处理

定期删除和懒惰删除：

- 定期删除是指 Redis 会定期遍历数据库，检查过期的 key 并且执行删除。它的特点是随机检查，点到即止。它并不会一次遍历全部过期 key，然后删除，而是在规定时间内，能删除多少就删除多少。这是为了平衡 CPU 开销和内存消耗。
- 懒惰删除是指如果在访问某个 key 的时候，会检查其过期时间，如果已经过期，则会删除该键值对



如果 Redis 开启了持久化和主从同步，那么 Redis 的过期处理要复杂一些。

- 在 RDB 之下，加载 RDB 会忽略已经过期的 key；（RDB 不读）
- 在 AOF 之下，重写 AOF 会忽略已经过期的 key；（AOF 不写）
- 主从同步之下，从服务器等待主服务器的删除命令；（从服务器啥也不干）

面试题

https://github.com/flycash/interview-baguwen/tree/main/redis

补充：负载均衡

- 算法
- 负载均衡算法在微服务下的额外约束
- 负载均衡的业务相关性：本地缓存与热点

负载均衡的本质

就是一个如何挑选的问题。所有算法都是——我挑一个我觉得很 OK 的服务提供者

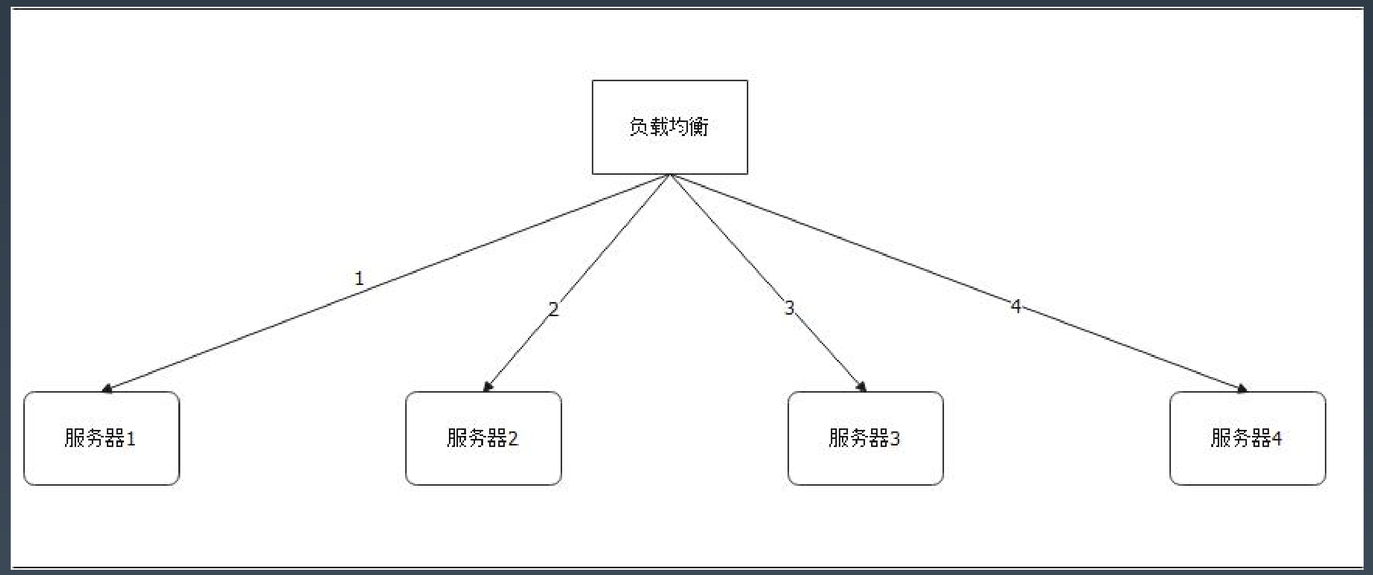
负载均衡——算法

- 轮询
- 加权轮询
- 随机
- 加权随机
- 哈希
- 最小连接数
- 最小活跃数（最少请求数）

负载均衡——轮询

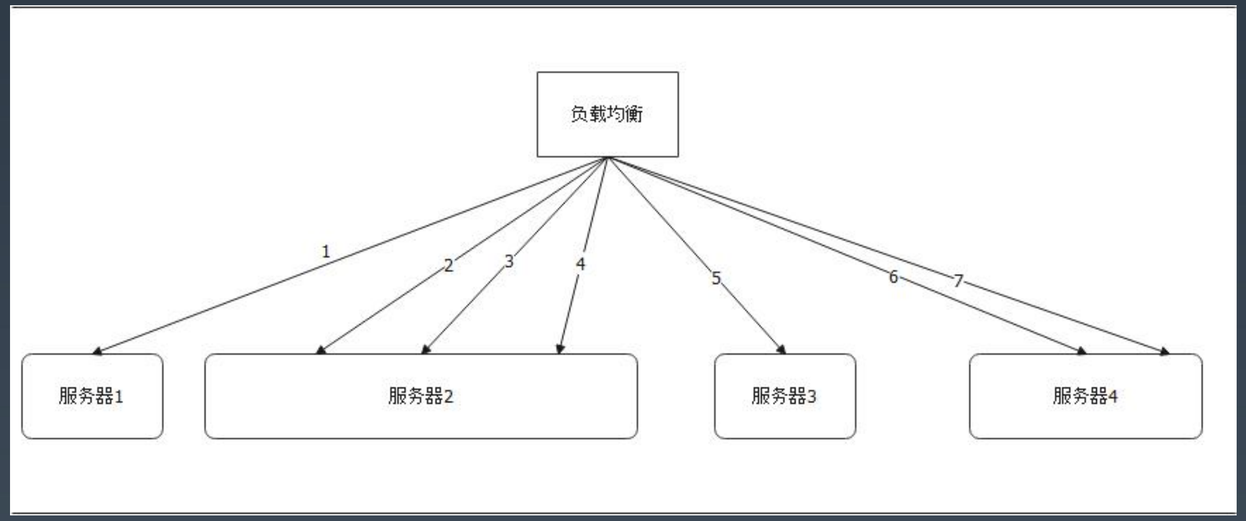
- 排排坐分果果
- 这种负载均衡算法有一些假设：
  - 所有服务器的处理能力是一样的；
  - 所有请求所需的资源也是一样的；

- 那么，为什么大多数时候它运作效果都很好呢？



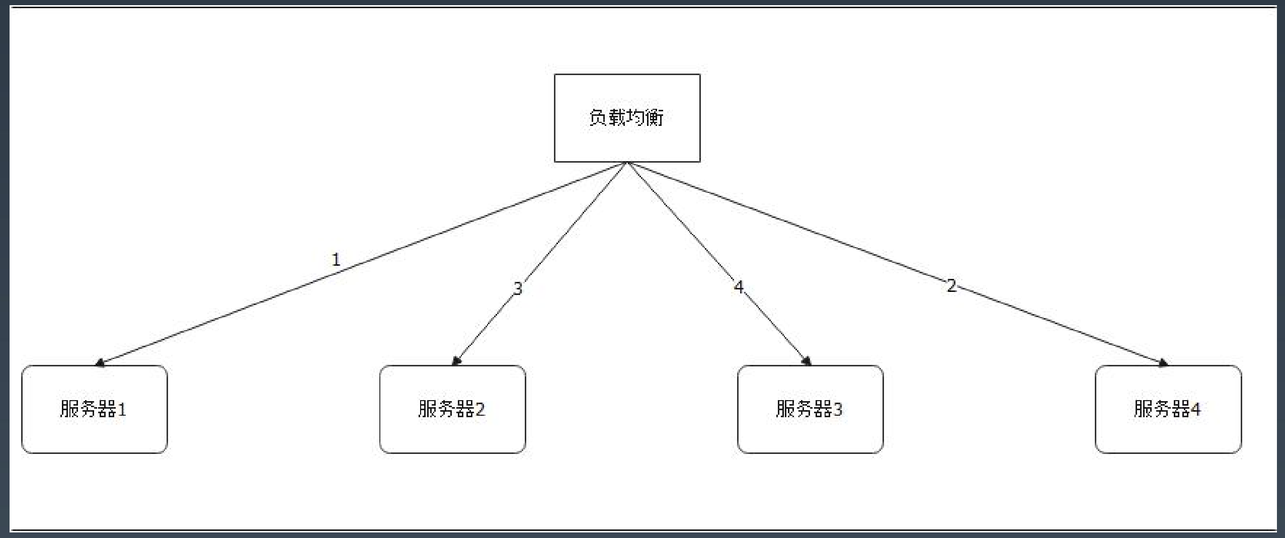
负载均衡——加权轮询

- 也是排排坐分果果，但是按照权重来分。即权重大的多分，权重少的少分
- 这种负载均衡算法有一些假设：
  - 用权重来代表服务器的处理能力；
  - 所有请求所需的资源也是一样的；



负载均衡——随机

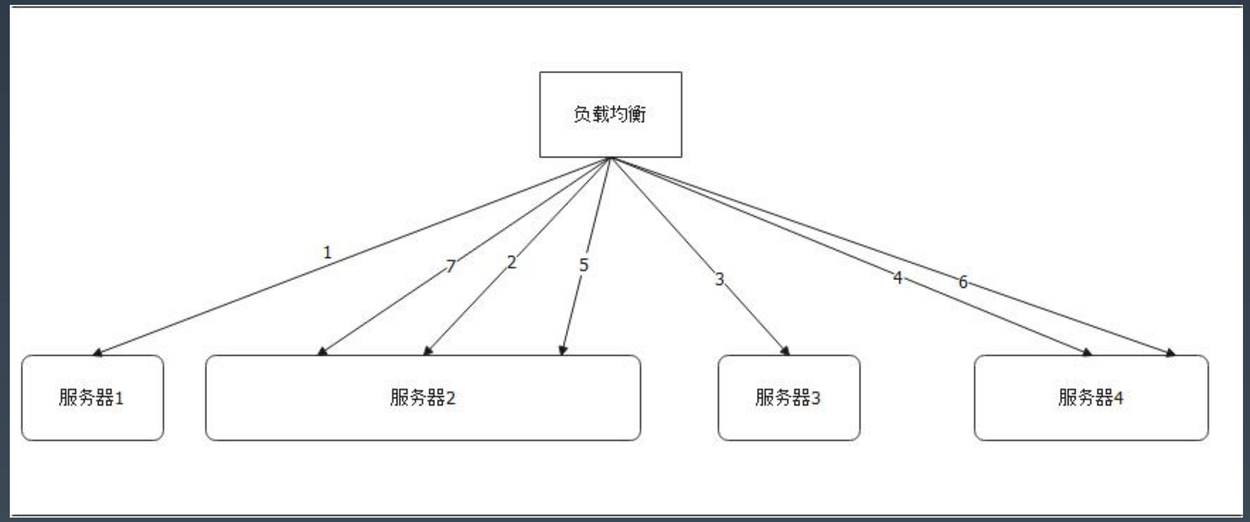
- 闭着眼睛瞎选
- 它在轮询的两个假设上，还有一个假设：
  - 所有服务器的处理能力是一样的；
  - 所有请求所需的资源也是一样的；
  - 每台服务器被随机到的概率一样，因而大量请求的情况下，服务器之间的负载会一样



- 相比之下，轮询的可控性更强。但是大多数时候可以认为，它们效果差不多

负载均衡——加权随机

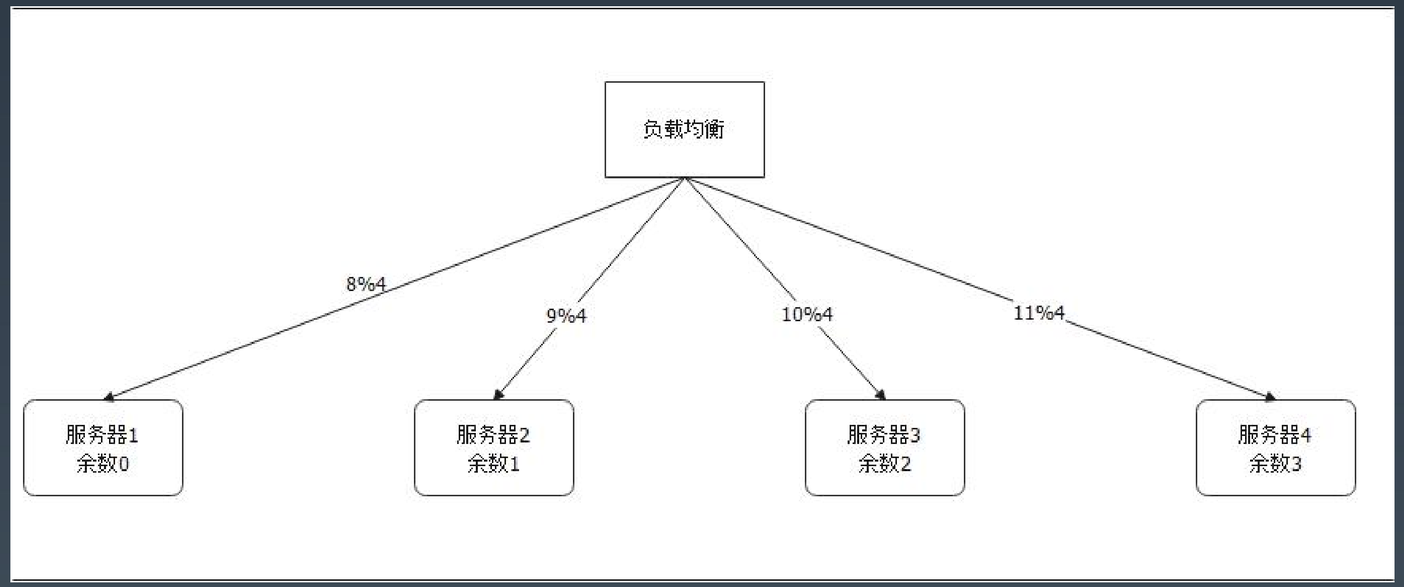
- 根据权重来确定选中概率
- 两个假设：
  - 用权重来代表服务器的处理能力；
  - 所有请求所需的资源也是一样的；



负载均衡——哈希

- 这种负载均衡算法有一些假设：
  - 所有服务器的处理能力是一样的；
  - 所有请求所需的资源也是一样的；
  - 哈希值是均匀的；

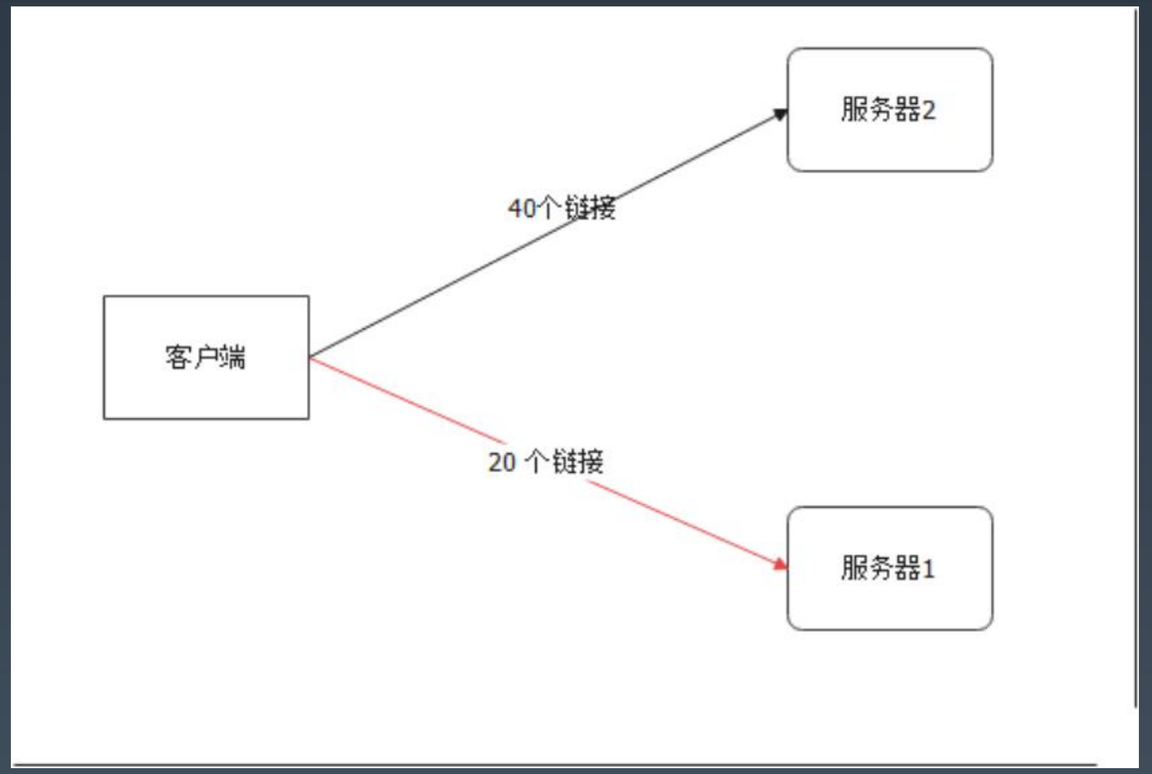
- 哈希值不均匀会导致请求堆积在一个地方。



负载均衡——最小连接数

- 假设：
  - 用连接数来代表服务器负载
  - 所有服务器的处理能力是一样的
  - 请求所需资源都一样

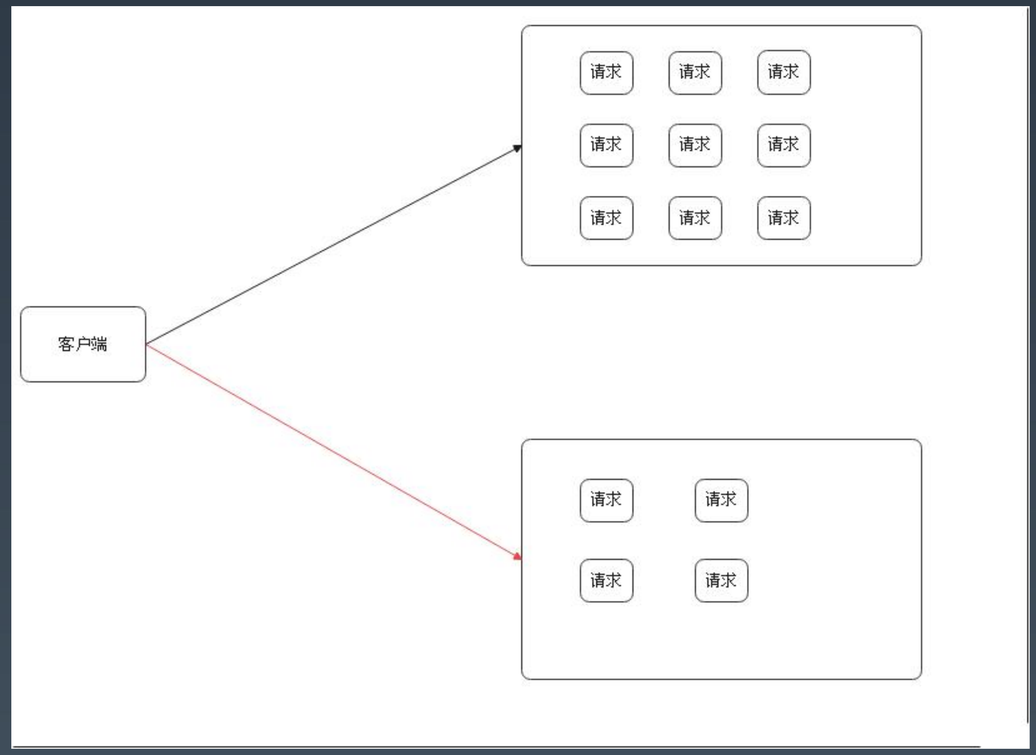
- 可能会短时间内把所有的请求都发过去同一台服务器
- 连接复用的情况下，连接数不能很好代表服务器的负载



负载均衡——最少活跃数

- 假设：
  - 用服务器上的请求数量来代表负载
  - 所有服务器的处理能力是一样的
  - 请求所需资源都一样

- 可能会短时间内把所有的请求都发过去同一台服务器



负载均衡总结

- 要不要考虑服务器处理能力？
  - 轮询，随机，哈希，最小连接数，最少活跃数都没考虑
- 选择什么指标来表达服务器当前负载？
  - （加权）轮询、（加权）随机、哈希什么都没选，依赖于统计
  - 选择连接数、请求数、响应时间、错误数……
- 是不是所有的请求所需资源都是一样的？显然不是
  - 大商家品类极多，大买家订单极多
  - 不考虑请求消耗资源的负载均衡容易出现偶发性的打爆某一台实例的情况

负载均衡

- 算法
- 负载均衡算法在微服务下的额外约束
- 负载均衡的业务相关性：本地缓存与热点

负载均衡算法在微服务下的额外约束

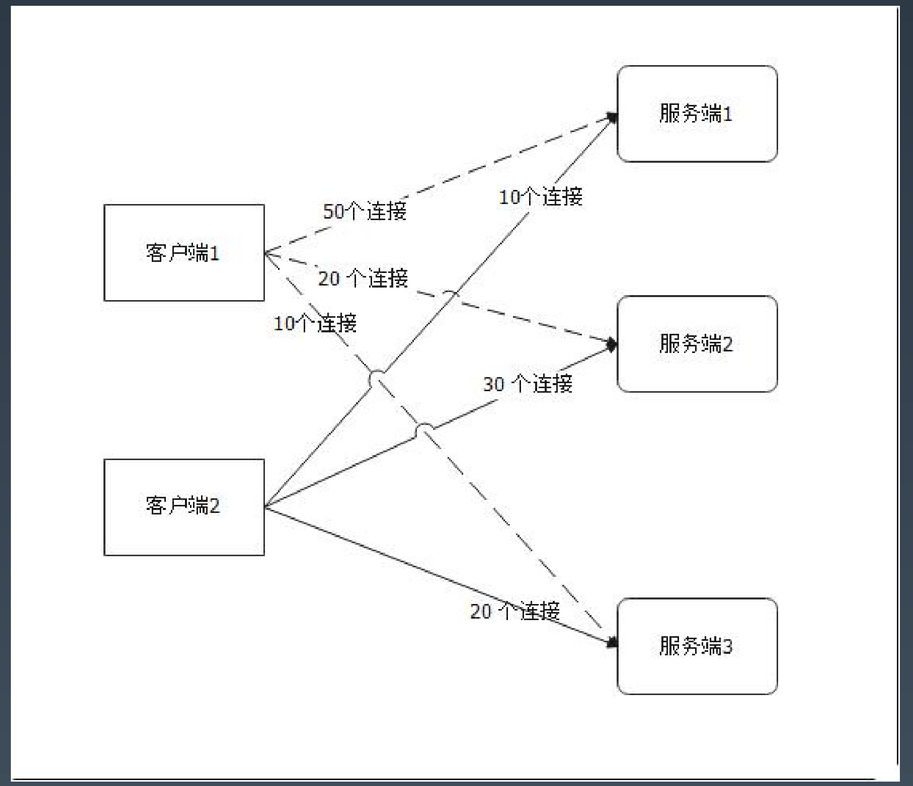
- 微服务客户端不同于网关，它不具备全局信息
  - 最小连接数负载均衡：客户端只能知道自己和服务器之间有多少链接
  - 最少活跃数负载均衡：客户端只能知道自己发过去的请求，还有多少个没有返回

- 微服务框架很少设计为要获得全局信息。难点在于：
  - 这本质上是一个分布式一致性问题
  - 即便借助于注册中心交换信息，会导致注册中心频繁通知客户端

负载均衡算法在微服务下的额外约束

- 在缺乏全局信息的情况下，客户端会选择服务端1作为服务提供者

- 在微服务中选择负载均衡算法，这种需要全局信息的算法可能抖动会比较厉害
- 那么为什么它们运作得还是很好呢？因为请求数量多了，慢慢会收敛都一种比较均匀的状态



负载均衡

- 算法
- 负载均衡算法在微服务下的额外约束
- 负载均衡的业务相关性：本地缓存与热点

负载均衡的业务相关性

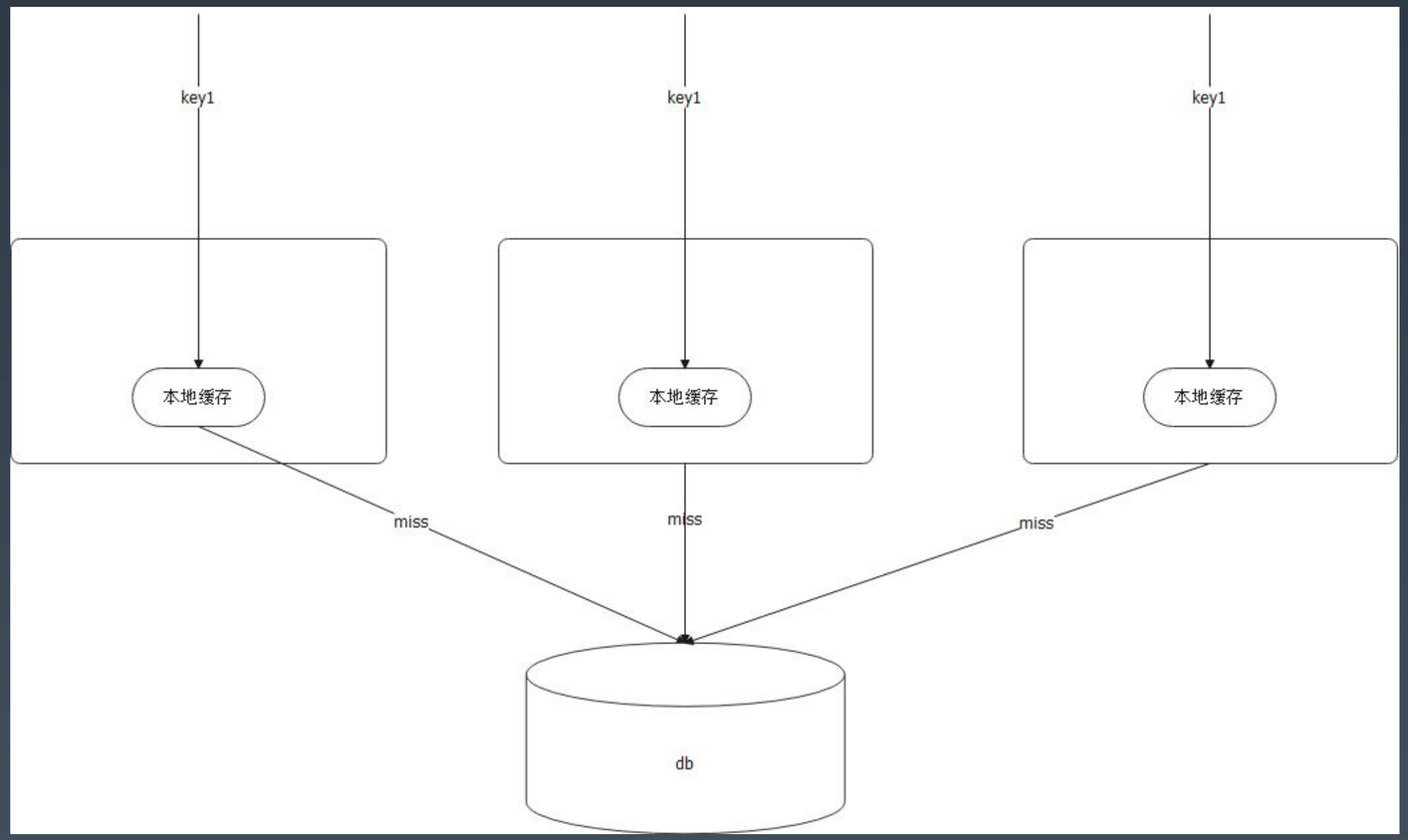
- 业务相关负载均衡：根据业务的某些特征来进行负载均衡。典型的如根据用户 ID 来进行哈希负载均衡。

- 但是往往一个请求所需多少资源和业务是强相关的，于是容易出现热点问题，或者大请求永远落在一部分机器上两个问题

- 业务相关的负载均衡可以结合本地缓存，避免本地缓存同步的问题

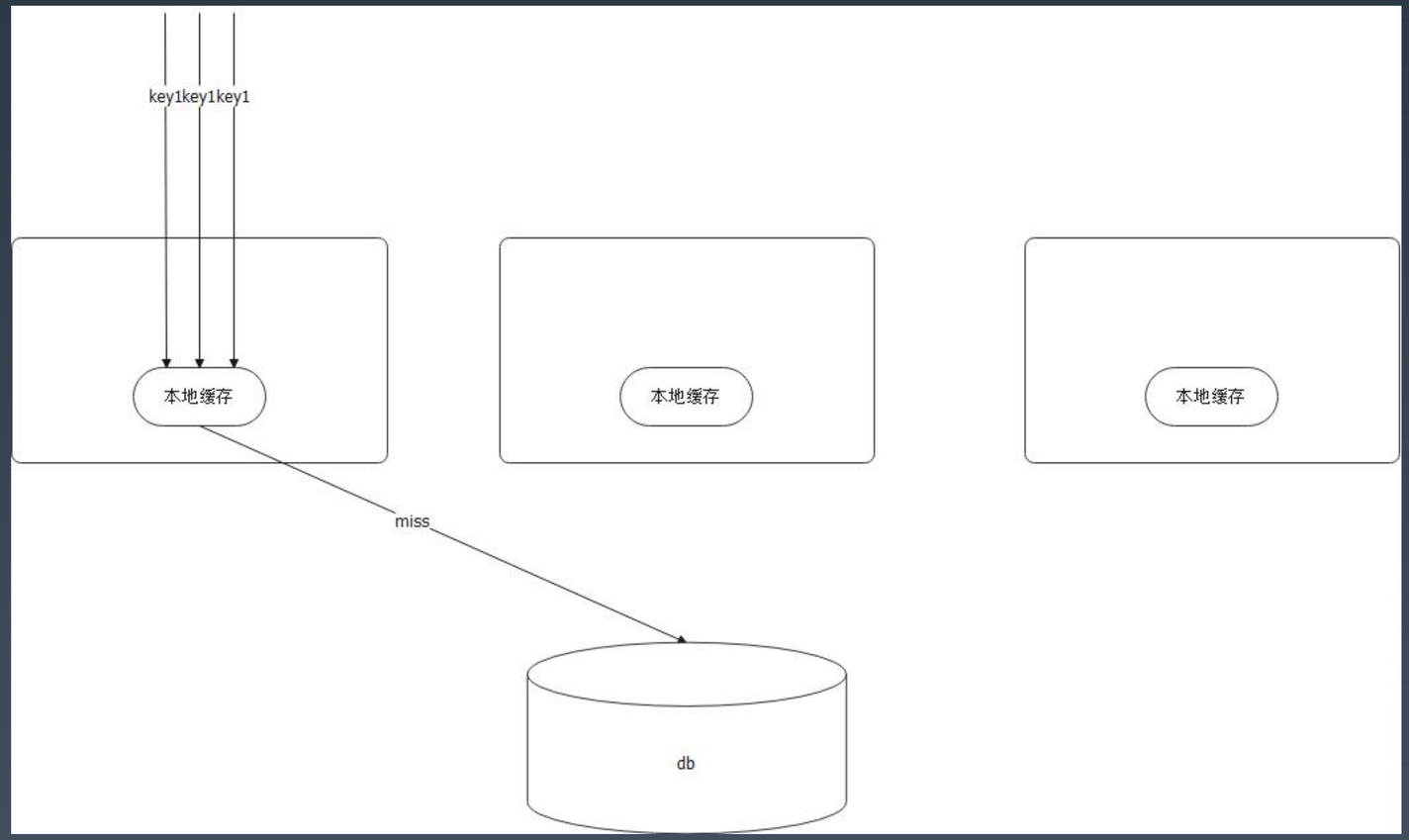
负载均衡与本地缓存

- 业务无关的负载均衡和本地缓存搭配，效果很差：
  - 缓存命中率低
  - 缓存占据内存更大
- 业务相关负载均衡，例如根据用户 ID 来做负载均衡（分库分表大部分都可以看做是业务相关的），存在热点问题



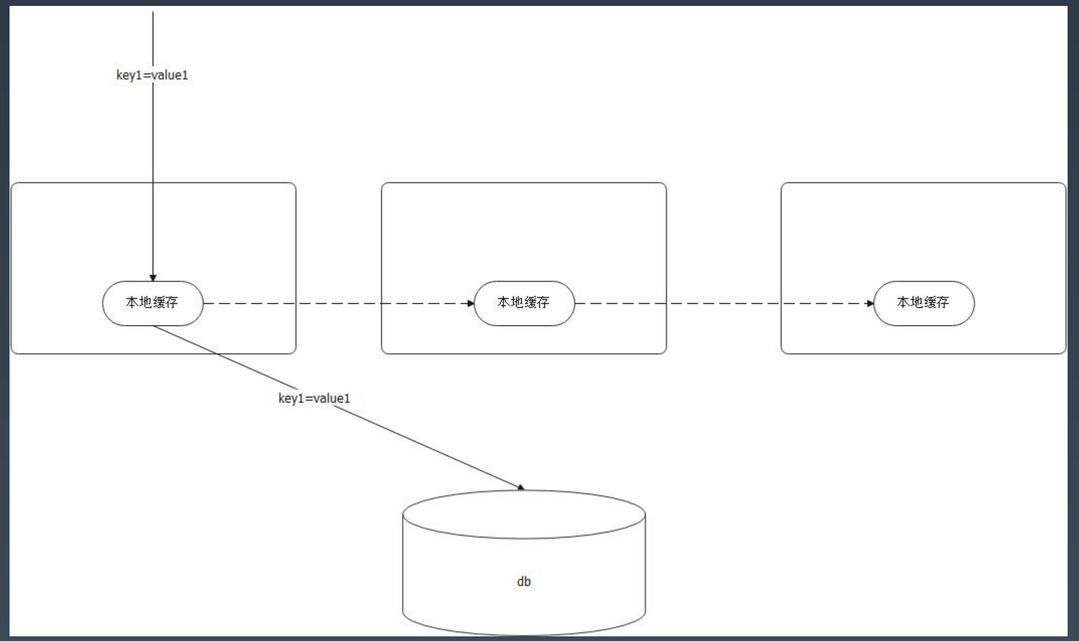
负载均衡与本地缓存

- 业务无关的负载均衡和本地缓存搭配，效果很差：
  - 缓存命中率低
  - 缓存占据内存更大
- 业务相关负载均衡，例如根据用户 ID 来做负载均衡（分库分表大部分都可以看做是业务相关的），存在热点问题



负载均衡与本地缓存更新

- 业务无关的负载均衡条件下，数据被更新之后，通知其它实例更新数据
- 本地缓存中间件具备这种能力，其它实例上的本地缓存会同步更新



负载均衡与本地缓存更新

- 业务无关的负载均衡条件下，数据被更新之后，通知其它实例更新数据。允许不一致的话，可以等待其它实例过期
- 本地缓存中间件具备这种能力，其它实例上的本地缓存会同步更新
- 本地缓存订阅数据变更