面试增强计划——中间件面试

大明

中间件讲究的就是背背背

主要内容

- MYSQL 八股文
- Redis 八股文
- Kafka 八股文

MySQL 八股文

MySQL 八股文

MySQL 问的问题非常广泛，不过在八股文里面，最重要是掌握这几点：

- 索引原理与索引优化
- 多版本控制协议与事务

注意，一些比较简单的和面试比较少见的我这里都没有讲，但是大家面试前要去看看。

然后记住我们第一周的内容：你必须把自己打造成为一个擅长数据库和 SQL 性能优化的人。

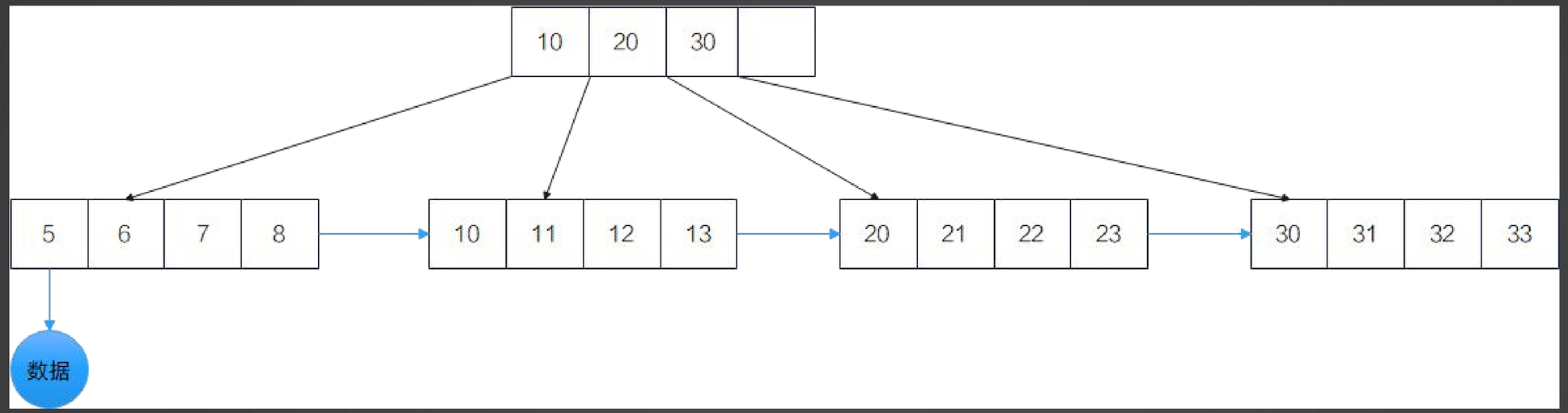
我个人认为比较 innodb 里面比较核心的一些东西，其中SQL执行过程比较少面

索引

要掌握：

- 索引在 MySQL innodb 引擎上的组织方式
- 索引的各种类型
- 数据库使用索引的各种异常情况
- 利用索引来优化查询性能

索引 —— 本质上是一个 B+ 树



在 MySQL 里面，分成所谓的聚簇索引和非聚簇索引。区别就在于叶子节点指向的数据：

- 聚簇索引的叶子节点里面存放的就是数据
- 非聚簇索引存放的就是主键 ID

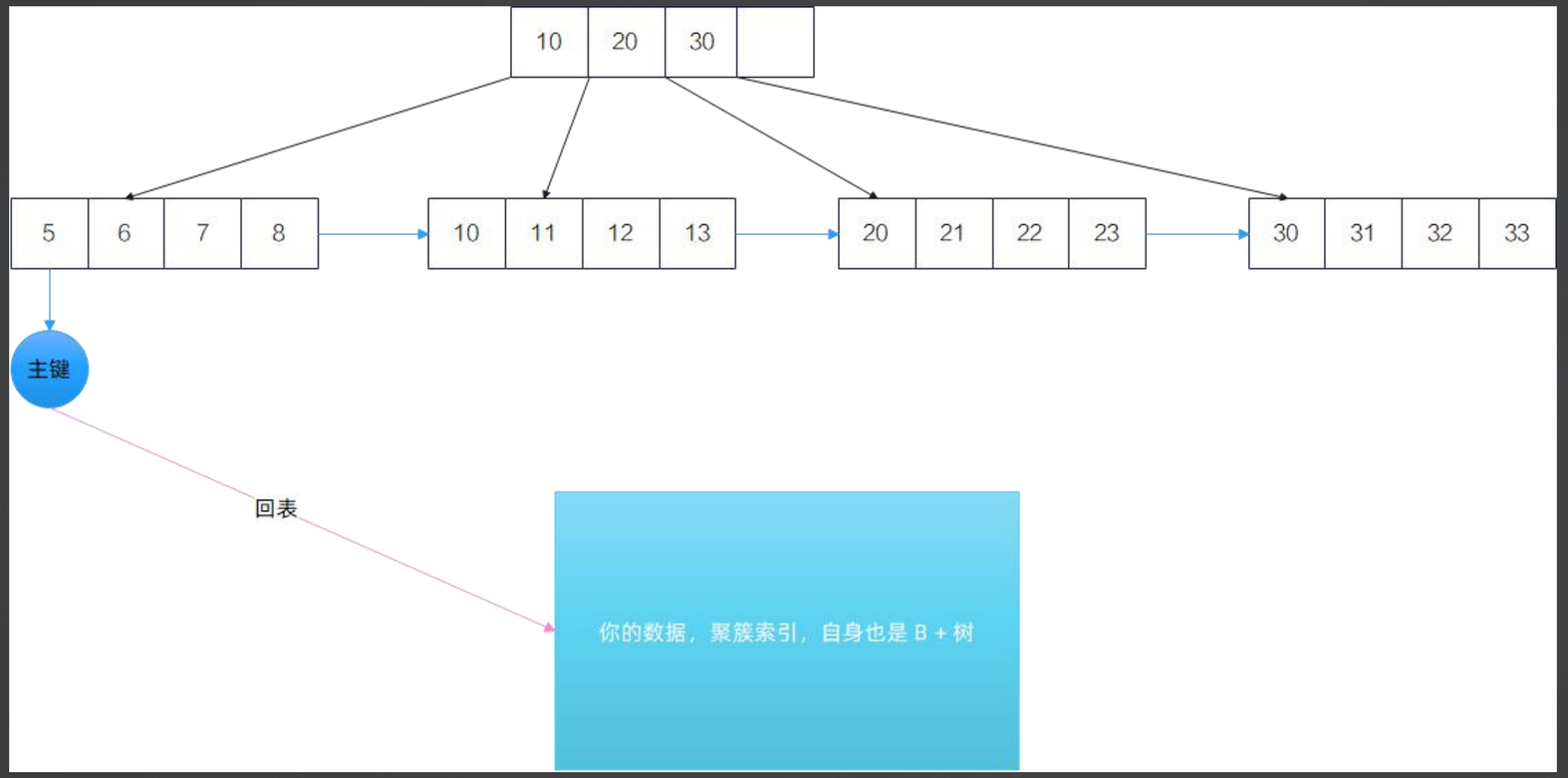
而非叶子节点，存放的就是 key，在索引里面就是索引对应的列的数据。

索引 —— 本质上是一个 B+ 树

从前面的那个图里面，尝试回答这些面试题：

- B+ 树的特征是什么？为什么用 B+ 树做索引？用 B 树行不行？二叉树呢？跳表呢？
- 什么是回表，为什么要回表？回表有什么缺点？怎么避免回表？
- 如果 ORDER BY 的列不是索引列，会发生什么？
- 为什么要用自增主键？不自增怎么办？分库分表可以用自增主键吗？
- 为什么没有命中索引（或者没有使用索引）的时候会使用表锁？

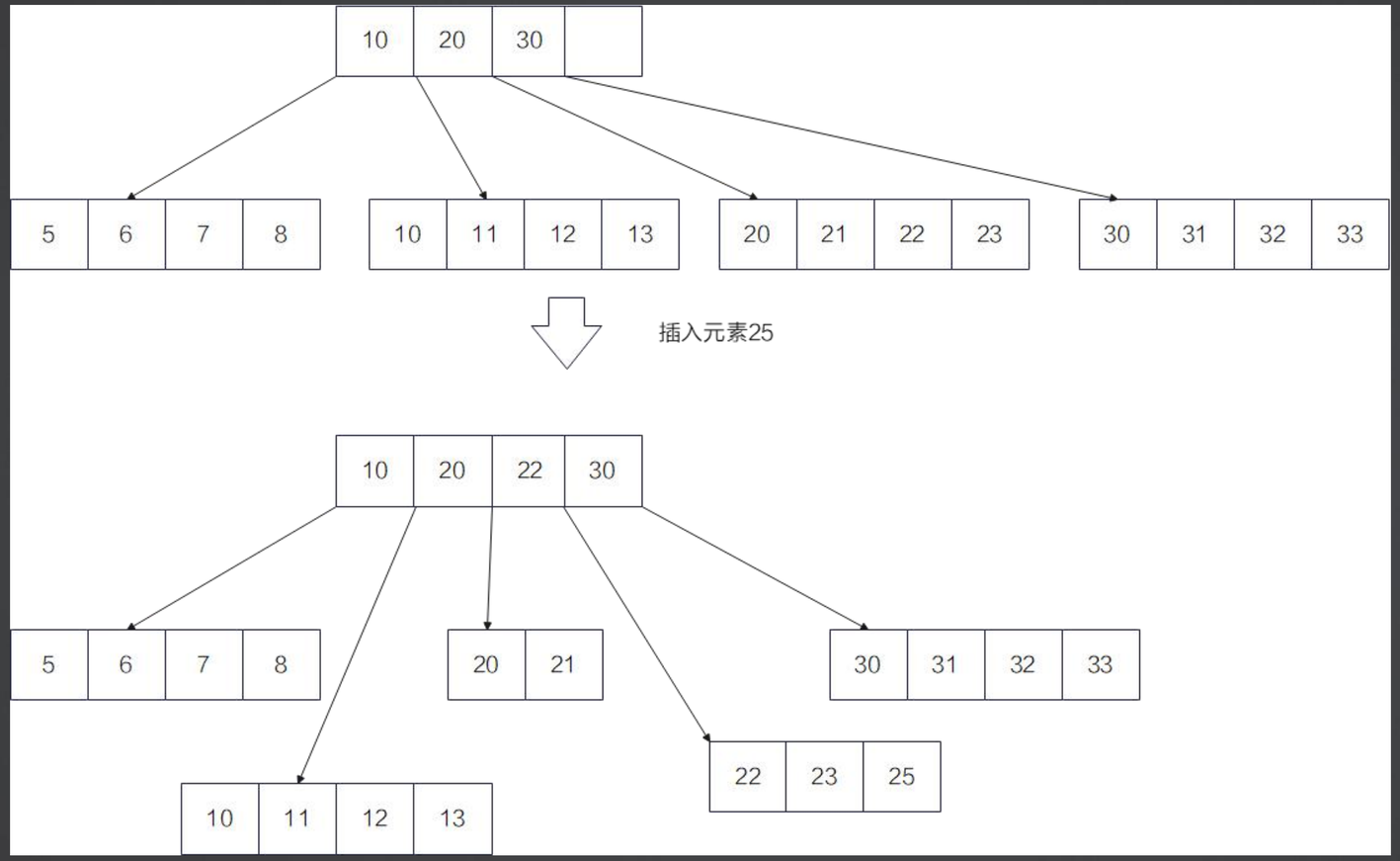
索引 —— 回表



所以你可以总结成一句话：但凡你的查询结果不能从索引本身拿到数据，那么就会回表。

回表就会引起磁盘 IO，磁盘 IO 就贼慢，所以你要避免回表

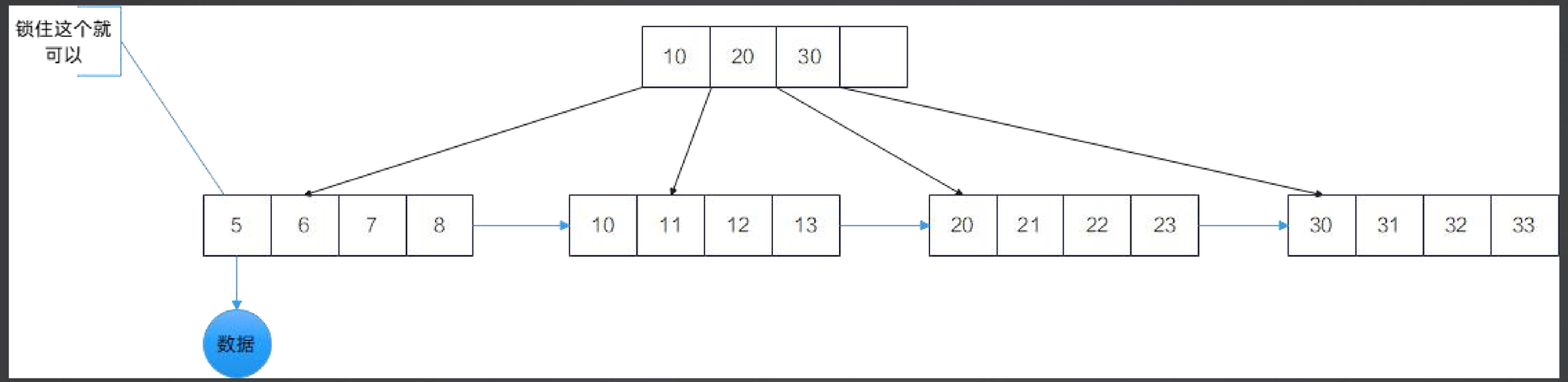
索引 —— 自增主键



主键如果不是自增，就会从树的中间插入，于是引起树节点分裂。树节点分离可以看做是数据页分裂，一分裂，就会有磁盘 IO 把原本的数据挪到别的位置。

自增主键会导致相邻的数据在磁盘上总是相邻的，能充分利用顺序 IO。

索引 —— 锁



其实如果你设计一个数据库的锁，你也没特别多的选择，比如说锁住对应的索引就可以了。

所以你可以得出另外一个推论：如果执行计划没有利用索引，那么也会使用对应的表锁。

索引 —— 数据库使用索引类似于多重循环

数据库利用索引的形式你可以看做是一个多重循环。例如说你在（A，B，C）上面创建了一个索引，那么数据库使用索引的时候类似于：

```
func findPK(as []A, bs []B, cs []C) {  no usages    ne
    for _, a := range a {
        for _, b := range bs {
            for _, c := range cs {

            }
        }
    }
}
```

索引

试试你能不能回答这些索引面试题：

- 索引的最左匹配原则是啥？
- 你可以用简单的话来描述 innodb 查询的时候是怎么利用索引的吗？
- 非聚簇索引是不是一定会回表？

索引 —— 最左匹配原则

最左前缀匹配原则是指，MySQL会按照联合索引创建的顺序，从左至右开始匹配，直到遇到了范围查询（<, <=, >, >=, between, like），再后面的查询条件就不会使用索引了。

可以理解为，在索引执行的多重循环中，但凡一层循环不是等值查询，那么内部的循环就不会再用到。
索引下推是例外，比如说 A = 123 AND B > 11 AND C like '%abc'

```
func findPK(as []A, bs []B, cs []C) {  no usages    ne
    for _, a := range a {
        for _, b := range bs {
            for _, c := range cs {

            }
        }
    }
}
```

假如 b 用了范围查询，那么 c 的循环就没用了

索引 —— 各种类型

所谓的各种类型，是指站在不同角度进行分类。比如说一个索引可以是联合索引，同时也是非聚簇索引。

- 聚簇索引和非聚簇索引：核心是叶子节点放的是数据本身，还是只是放了一个主键
- B + 树索引和哈希索引：innodb 引擎本身不支持哈希索引
- 联合索引（组合索引）和非联合索引：使用了多个列的就是联合索引
- 唯一索引和非唯一索引：
- 前缀索引：使用了列的一部分的索引。比如说在 varchar(128) 的字段上，值利用前32个字符创建索引；
- 全文索引
- 覆盖索引：其实是指你查询的列，都是某个索引的列。覆盖索引最大的好处就是不用回表；

索引 —— 使用索引的各种异常

面试题：

- 为什么明明创建了索引，但是查询就是没用？
- 为什么用了索引，查询还是很慢？
- 索引是不是越多越好？索引的列是不是越多越好？
- 使用索引有什么缺点？
- 查询索引本身会引起磁盘 IO 吗？
- NULL 对索引会有什么影响？
- 什么是索引下推

索引 —— 设计和优化索引

选择合适的列创建索引，可以遵循这些原则：

- 在 WHERE 条件里面经常出现的。比如说外键；
- 经常在 JOIN 关联条件里面出现的。可以忽略这一条，因为一般我们都不建议使用 JOIN 语句；
- 使用有很多不同值的列：所以类似于 Status 这种枚举的效果就不是很好；
- 不要使用很长的列：比如说 BLOB 这种，或者很长的 varchar。一定要用的话，创建前缀索引；

创建联合索引，确定索引的顺序：

- 选择性高的在前面；
- 经常用作范围查询（也就是会中断索引使用的）放在后面；

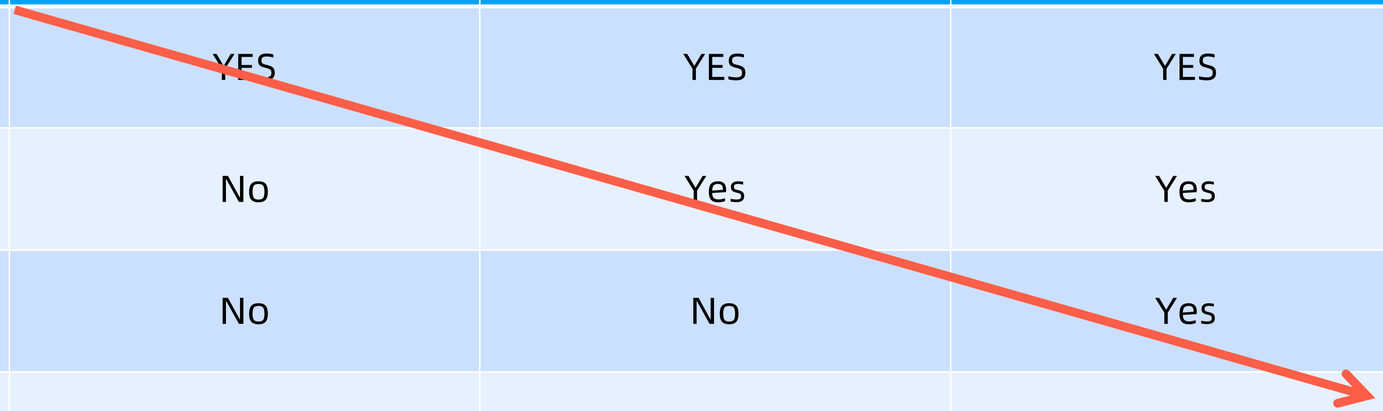
MVCC —— 多版本控制协议

多版本控制协议是 innodb 用于实现并发控制的协议。前置知识：

- 事务隔离级别
- redo log 和 undo log
- Read View

MySQL 隔离级别与读异常

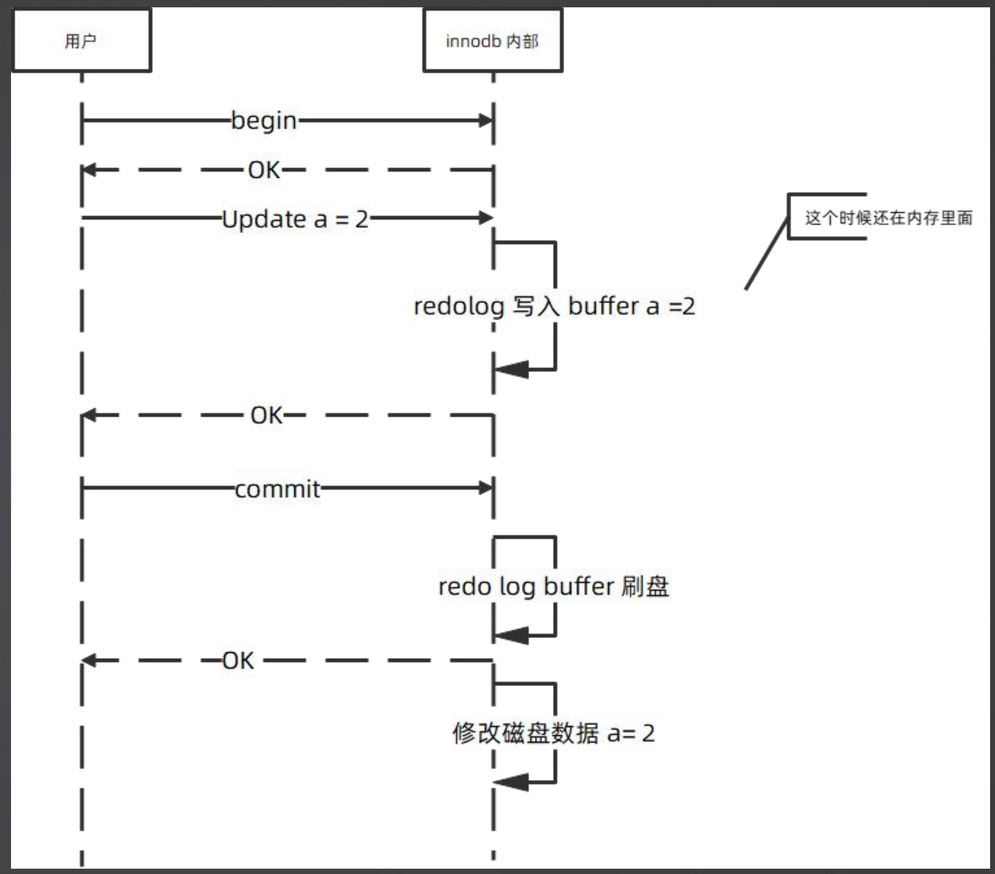
隔离级别	脏读	不可重复读	幻读
未提交读	YES	YES	YES
已提交读	No	Yes	Yes
可重复度	No	No	Yes
序列化	No	No	No



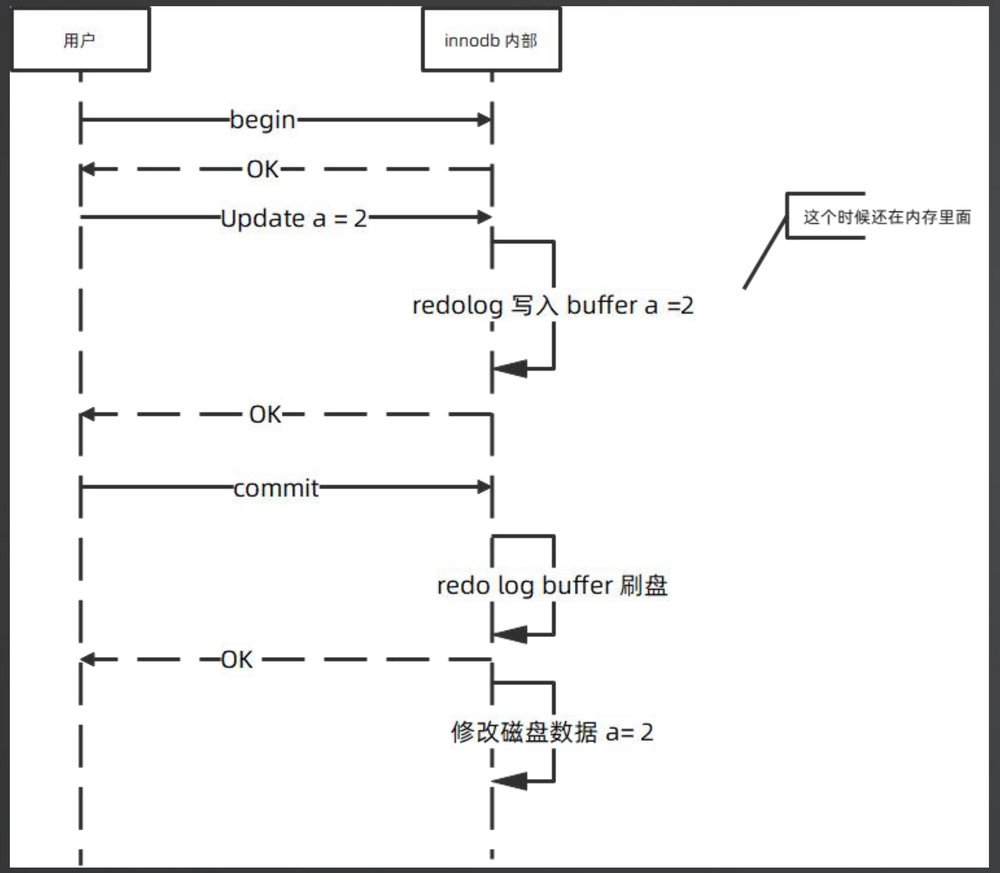
- 未提交读，永远都是读最新的数据；
- 已提交读和可重复读被 MVCC 控制，通过 Read View 来实现
- 序列化直接咔嚓一把锁，加锁成功之后读最新的（必然是已经提交的，也必然是可重复读的）

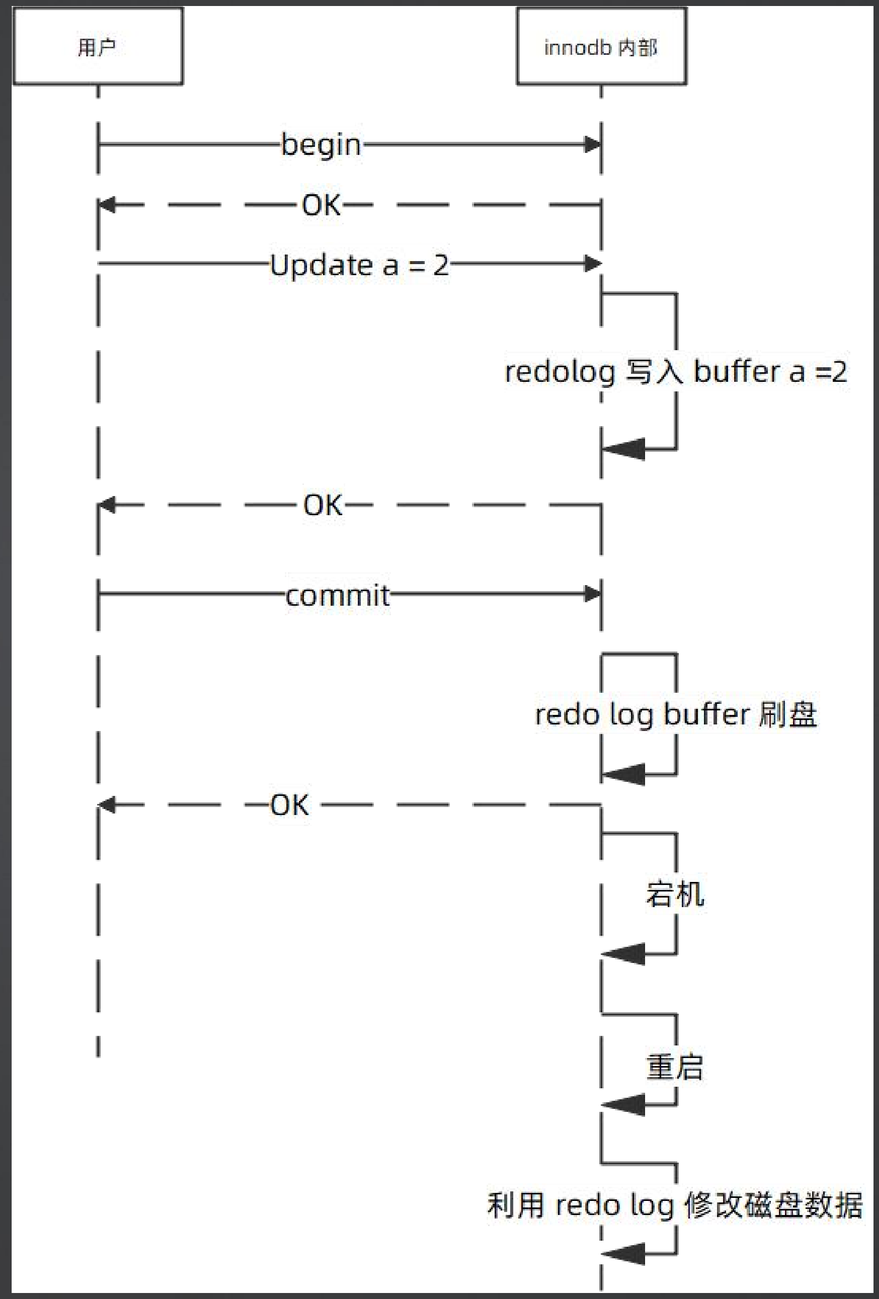
redo log

- redo log： 你可以看做就是你的语句本身。比如说你在事务里面执行一个 INSERT，它会先把 INSERT 语句写入 redo log。万一事务提交的时候数据库宕机了，重启的时候就会利用 redo log 来保证事务提交成功。



redo log 和恢复


用户
innodb 内部
begin
OK
Update a = 2
这个时候还在内存里面
redolog 写入 buffer a =2
OK
commit
redo log buffer 刷盘
OK
修改磁盘数据 a= 2

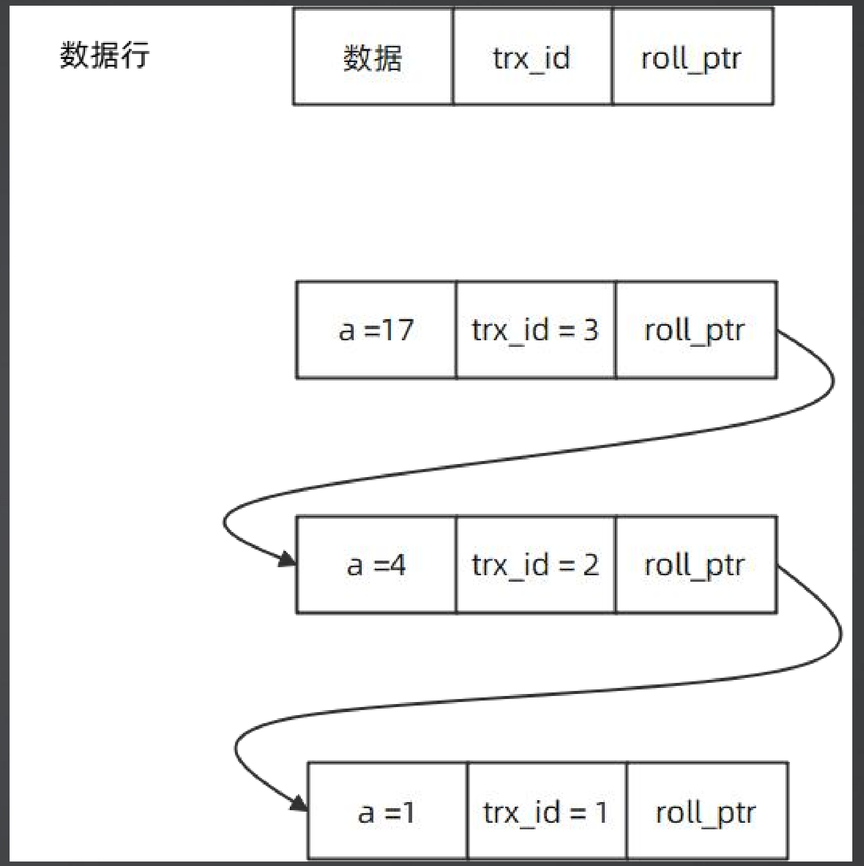

用户
innodb 内部
begin
OK
Update a = 2
redolog 写入 buffer a =2
OK
commit
redo log buffer 刷盘
OK
宕机
重启
利用 redo log 修改磁盘数据

undo log

- undo log： 你可以看做是一个相反的语句。比如说你在事务里面执行一个 INSERT，那么对应的 undo log 就是一个 DELELTE 语句。
  - INSERT -> DELETE
  - UPDATE -> UPDATE
  - DELETE -> INSERT

undo log 主要用于回滚，所以它需要将所有的历史变化记录下来，才能准确回滚数据。

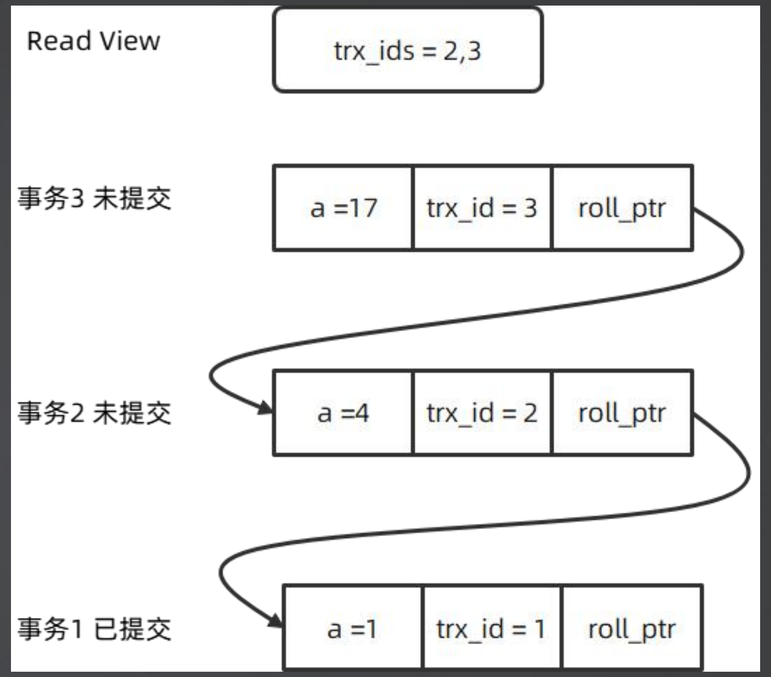
所以要理解 undo log 就得弄明白每一条数据的 trx_id 和 roll_ptr 的含义。



当前对数据修改的历史记录全部组织在一起，这个也叫做回滚段

Read View

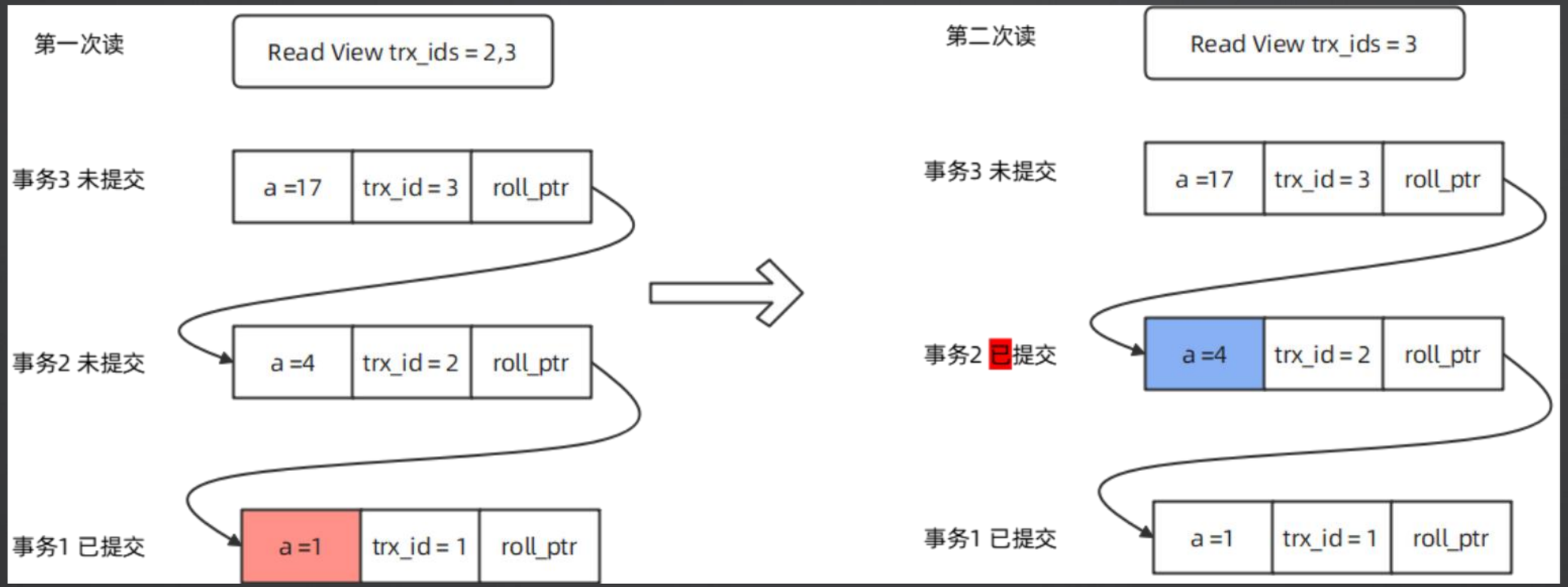
Read View 的核心是它会记录所有当前活跃事务（已经begin，但是还没提交的事务）的 ID，叫做 trx_ids。



当前对数据修改的历史记录全部组织在一起，这个也叫做回滚段

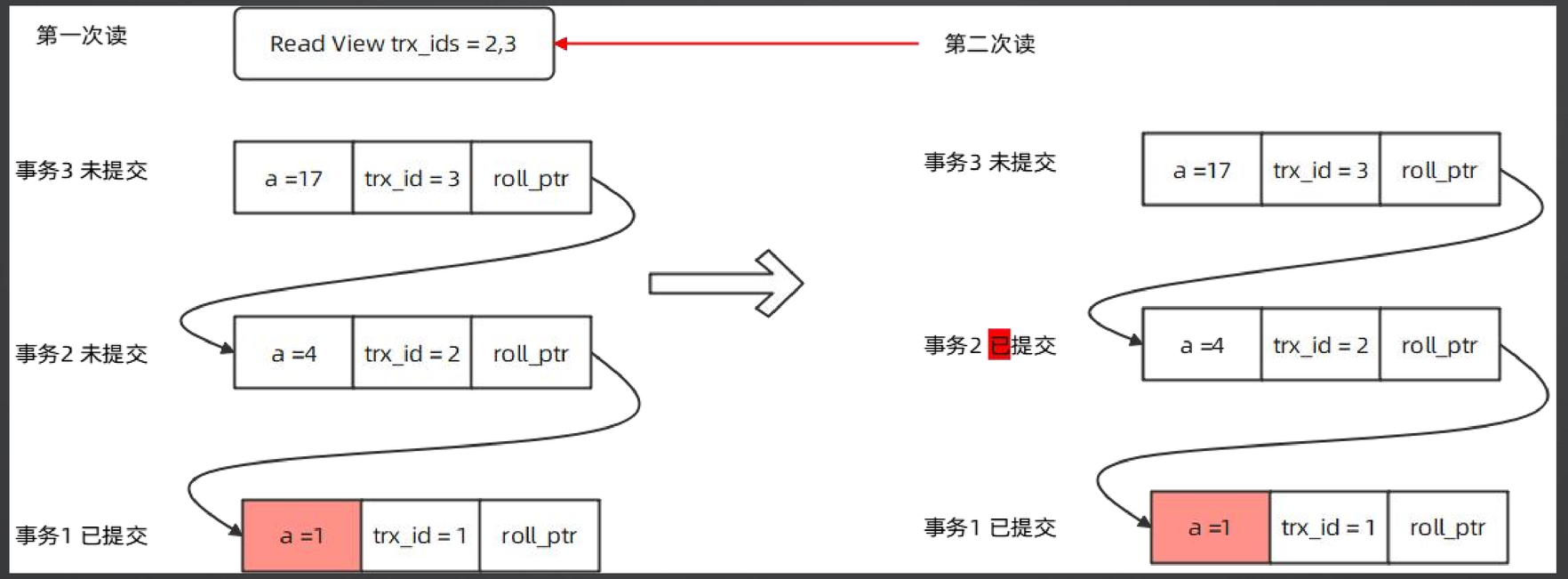
Read View - Read Committed

在 RC 的隔离级别之下，Read View 每次都会重新生成一个



Read View - Read Repeatable

在 RR 的隔离级别之下，Read View 用的是一直都是同一个



MVCC 面试题

- 什么是 MVCC
- 什么是 redo log？有什么用？
- 什么是 undo log？有什么用？
- 什么是 bin log？有什么用？
- MySQL 的隔离级别有什么？
- 为什么事务提交了，数据还是丢了？

锁

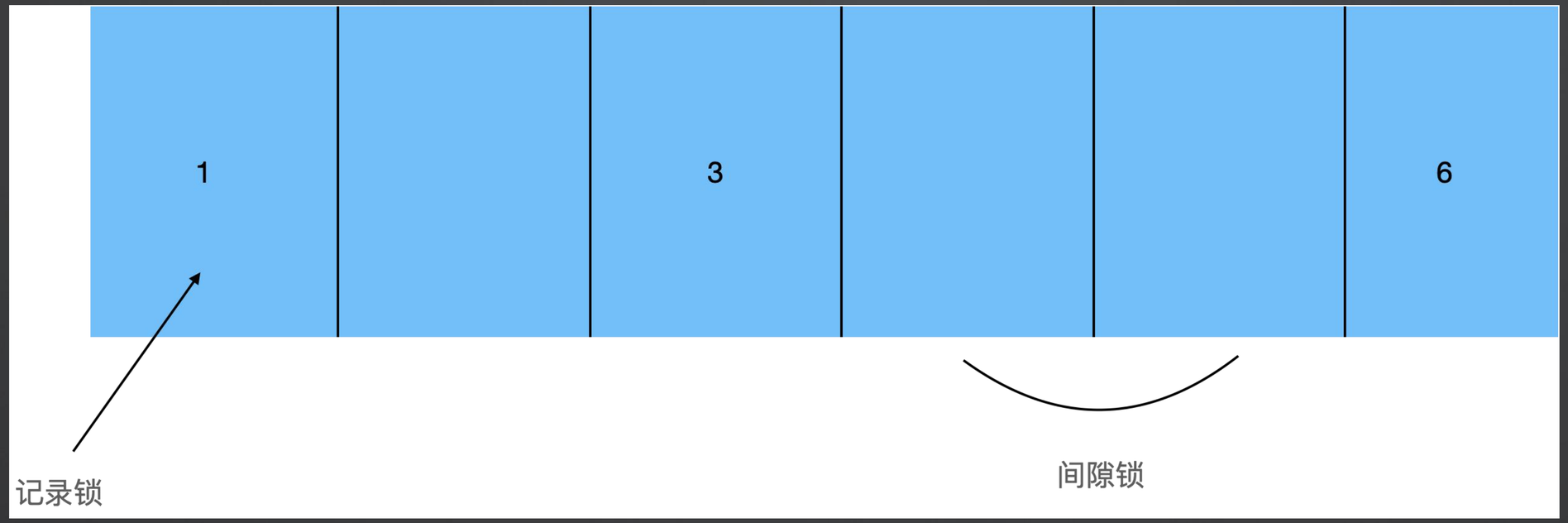
MySQL 的锁有点像索引，可以从不同的角度来进行分类：

- 共享锁与排它锁：共享锁就是可以多个人拿着，类似读锁；排他锁类似写锁；
- 意向锁：意向共享锁和意向排它锁。简单理解就是在加锁前对其它事务打个招呼，说我需要什么锁；
- 记录锁、间隙锁和临界锁
- 行锁和表锁

另外所谓乐观锁，实际上是你多加一个列，比如说叫做 version，然后执行更新或者查找的时候 WHERE 里面带上版本号。比如说 Update xxx set a = 1 WHERE id = 10 AND version = 123

记录锁、间隙锁和临界锁

- 记录锁：锁住一行，所以叫做记录锁，也是行锁；
- 间隙锁：锁住记录之间的间隔，或者索引之前的范围，或者所以之后的范围。只在重复读级别产生，（可以在前面隔离级别的地方提）
- 临键锁（Next key lock）:记录锁和间隙锁的组合，即锁住记录，又锁住了间隙



锁面试题

- innodb 引擎的 RR 级别解决了幻读吗？解决了（官方说法），利用了临键锁
- 各种锁的概念、含义和使用

Redis 八股文

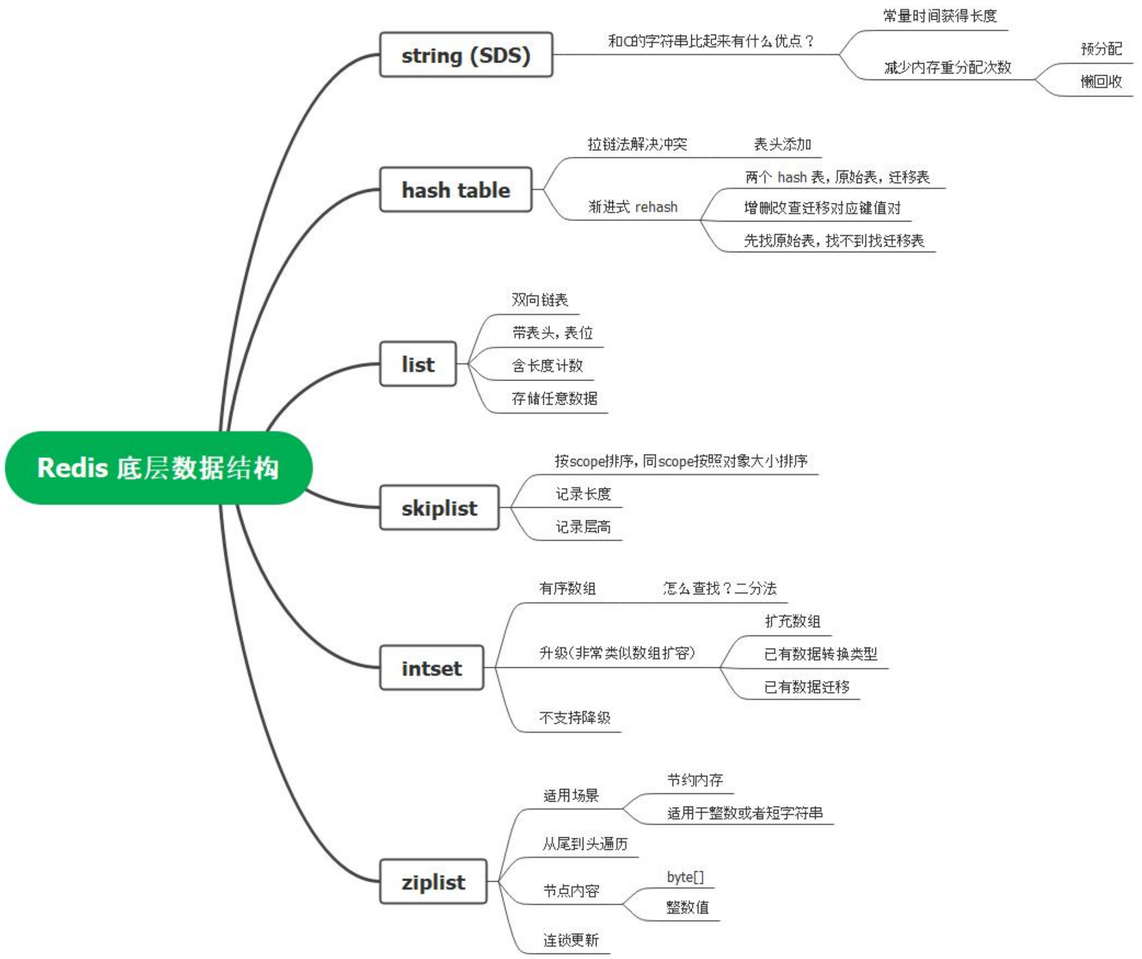
面试要点

- 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

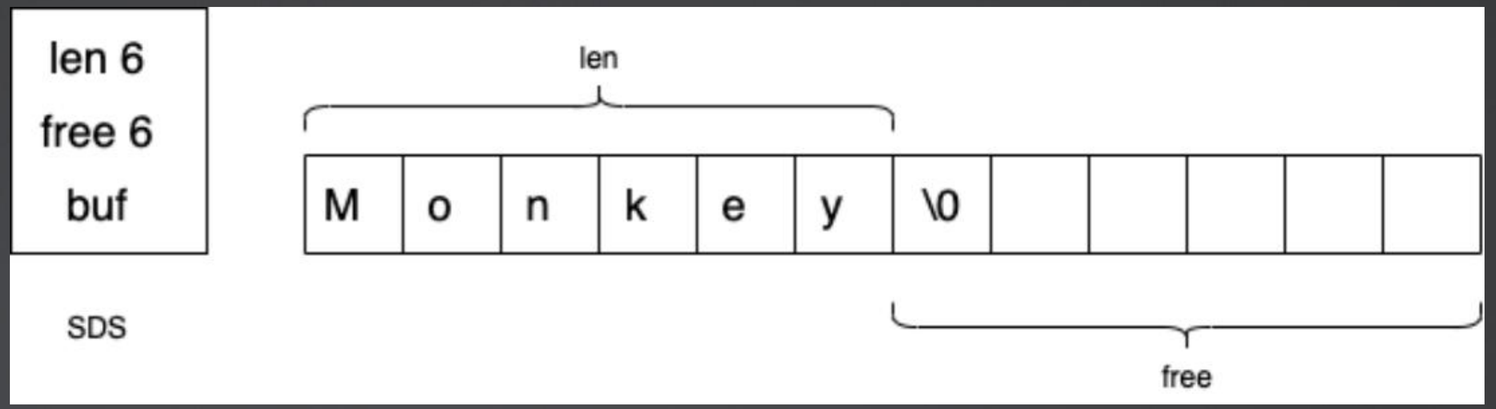
Redis 基础 —— 数据结构之底层实现



Redis 底层实现—— SDS

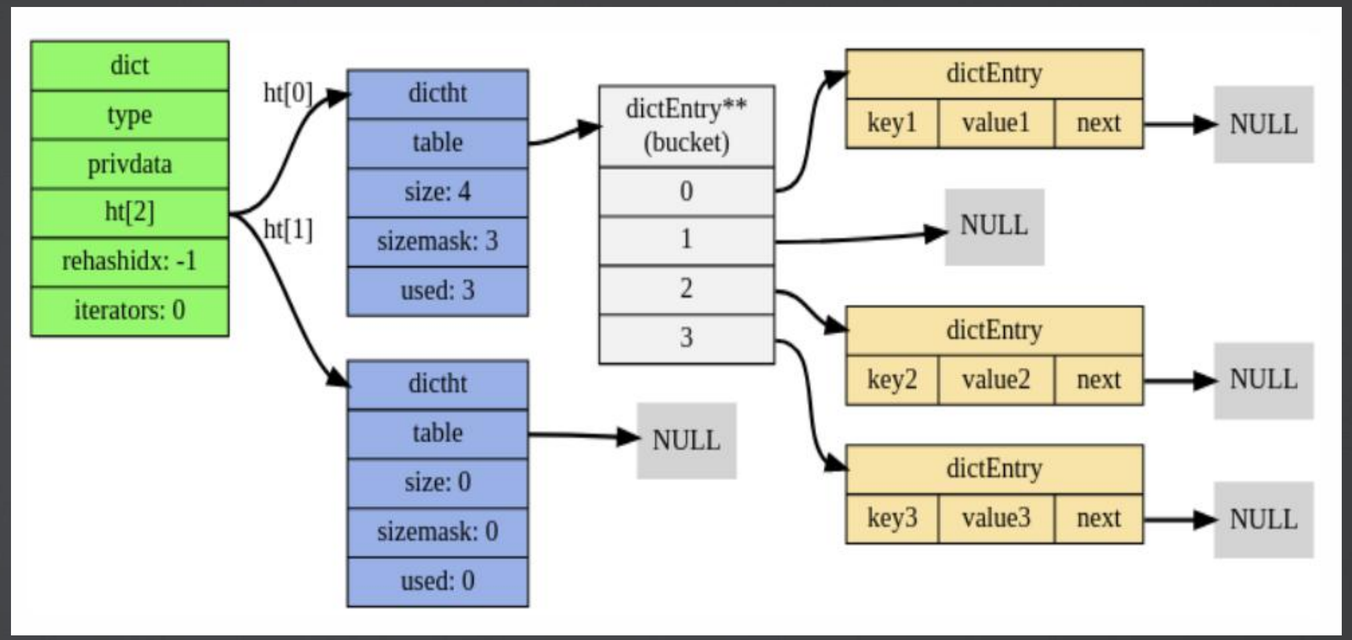
- 直接维持了字符串的长度
- 预分配，减少内存分配

- 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



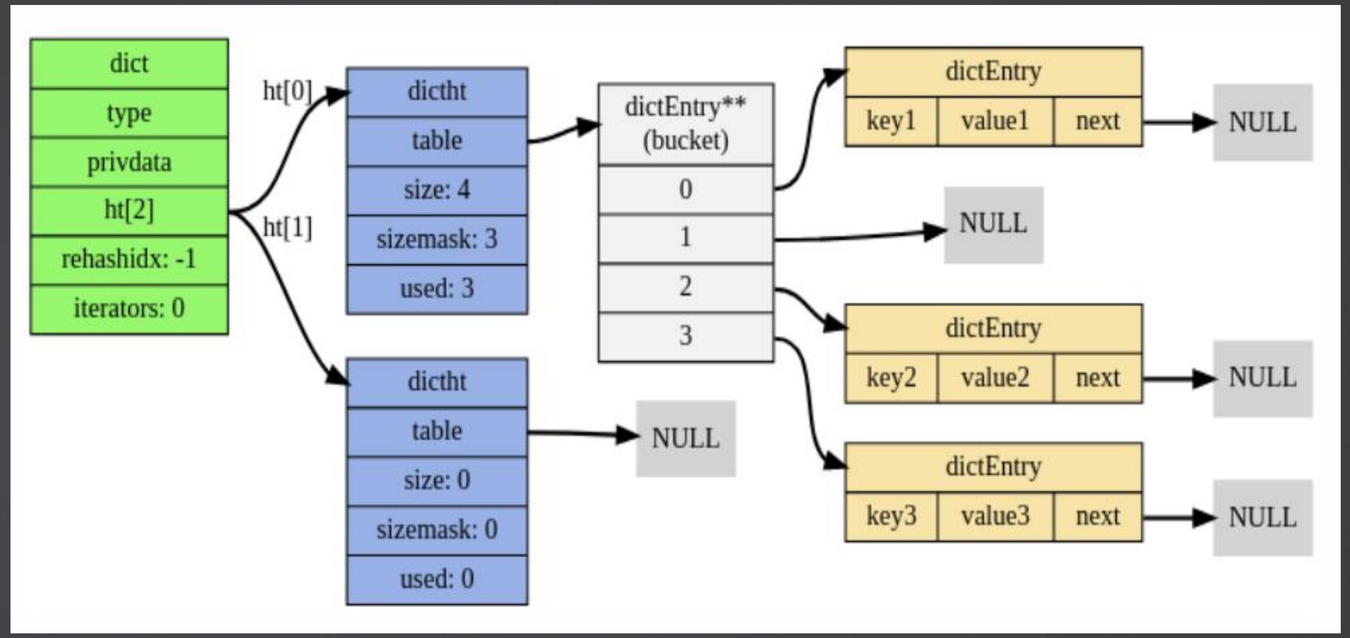
Redis 底层实现—— hash table

- 拉链法解决冲突：冲突节点放到头部
- 渐进式 rehash 扩容：对比 go map 的扩容



Redis 底层实现—— rehash 过程

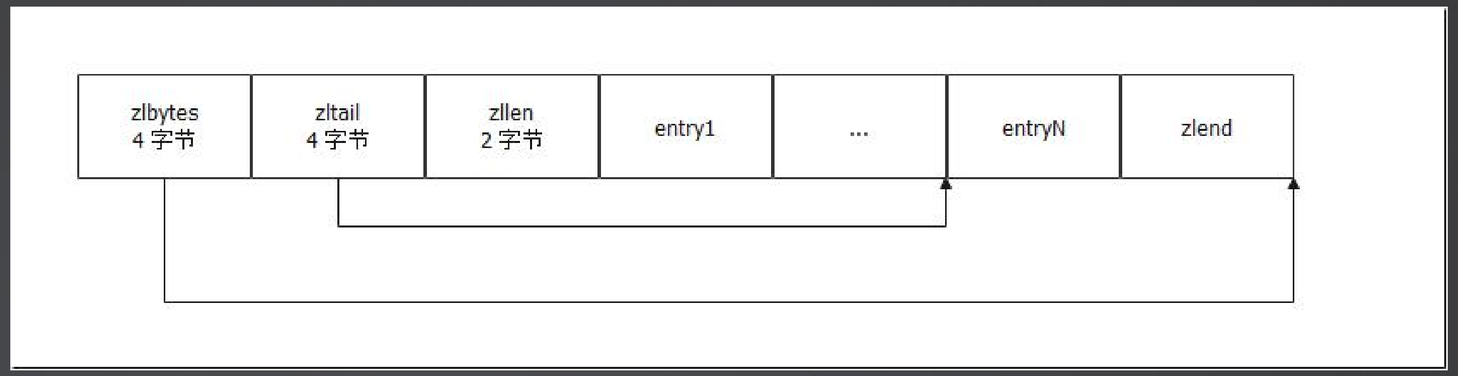
- 为 ht[1] 分配空间：扩容，则是当前长度向上第一个 2^n；缩容，则是当前长度向下 第一个 2^n；
- 将 ht[0] 中的元素重新计算分布到 ht[1]，改查会触发这个过程
- ht[0] 都迁移完之后，交换ht[0]和 ht[1]



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

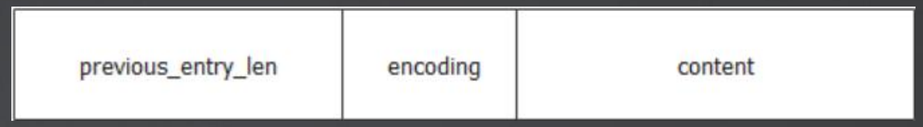


- 查： O(N)
- 增删：平均 O(N)，最坏 O(N^2)

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



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

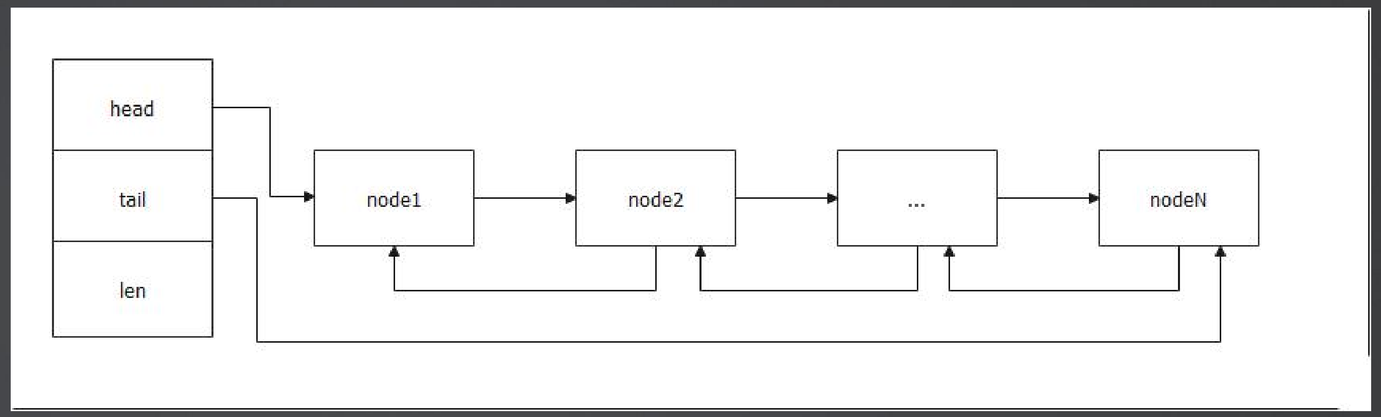
- previous_entry_length：一个字节（<254长度）或者五个字节

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。

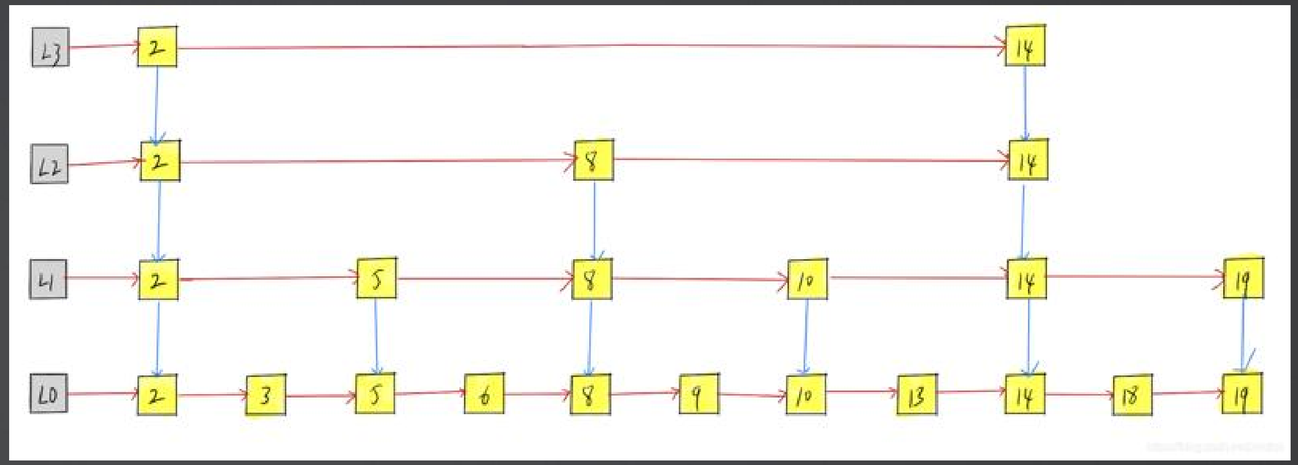
Redis 底层实现—— linkedlist

- 双向链表
- 直接维持了长度



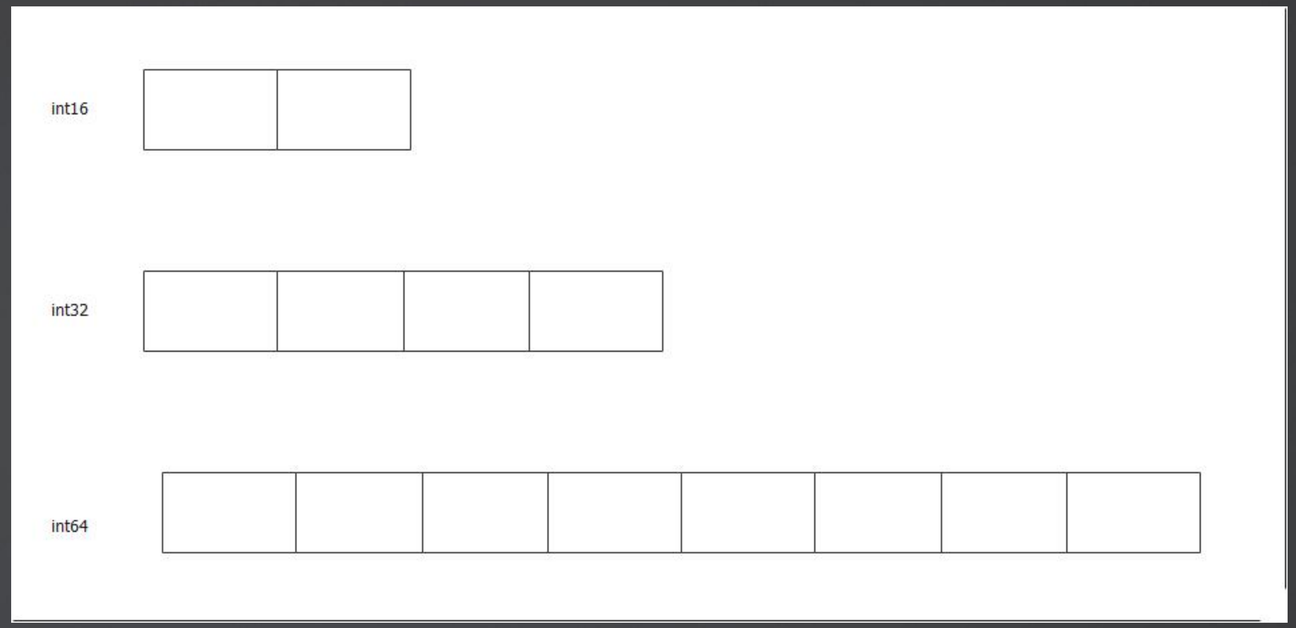
Redis 底层实现—— skiplist

- 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

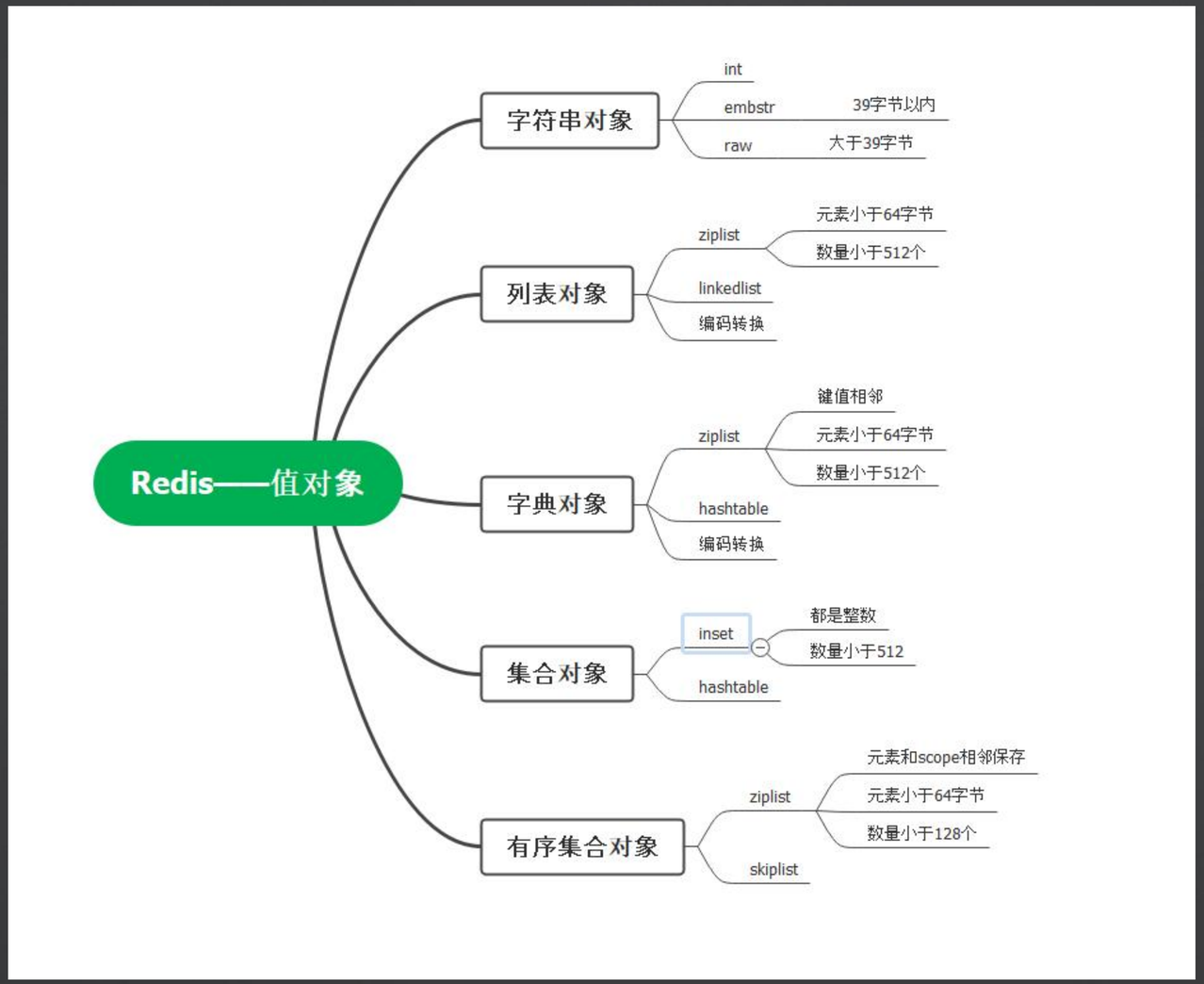


- 如果数字升级，那么需要全部元素升级一遍（因为 intset 要求每个元素字节数必须一样）
- intset 这种思想，就是建立在数据量小的情况下，遍历比 Hash 之类的结构，在查找方面更快

Redis 基础 —— 数据结构之值对象

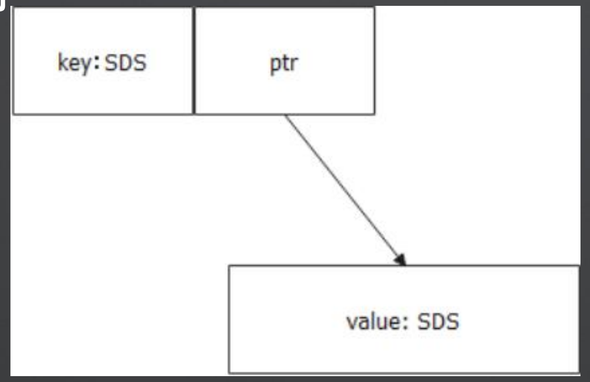
- 掌握不同类型的使用场景（面试热点）
- 掌握不同类型的底层实现是什么（面试热点）
- 掌握触发编码转换的条件

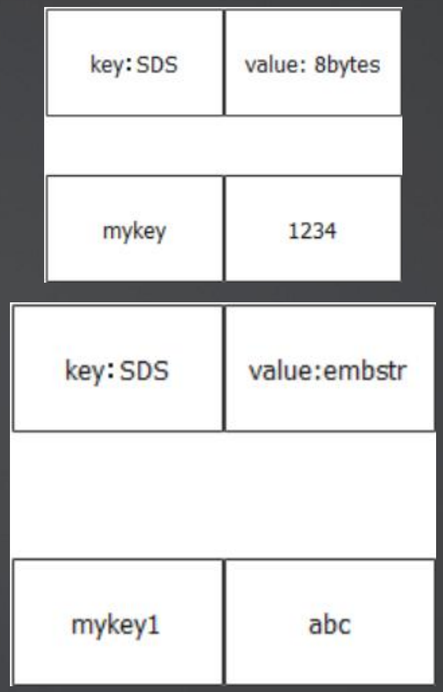
Redis 基础 —— 数据结构之值对象



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





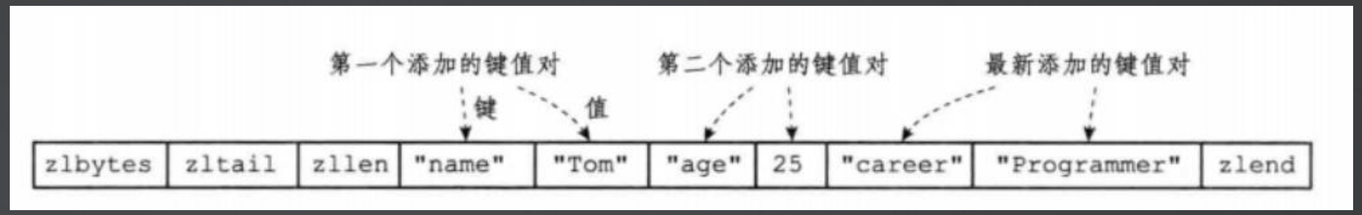
Redis 基础 —— 列表对象

- 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；

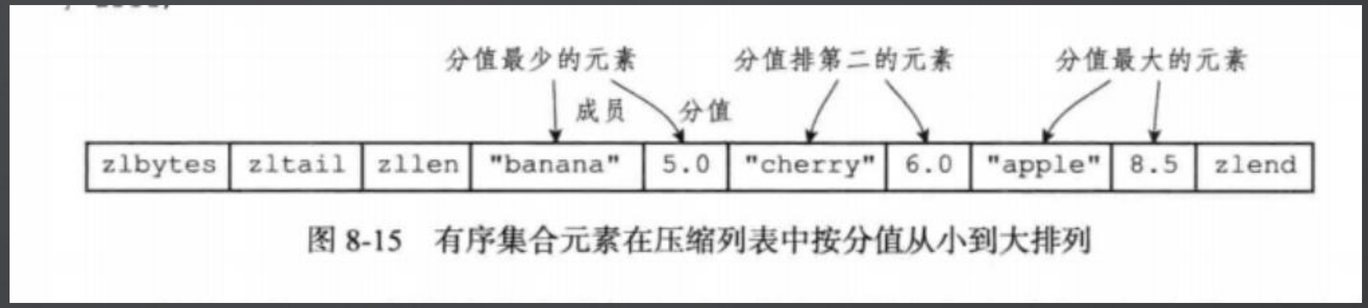


图 8-15　有序集合元素在压缩列表中按分值从小到大排列

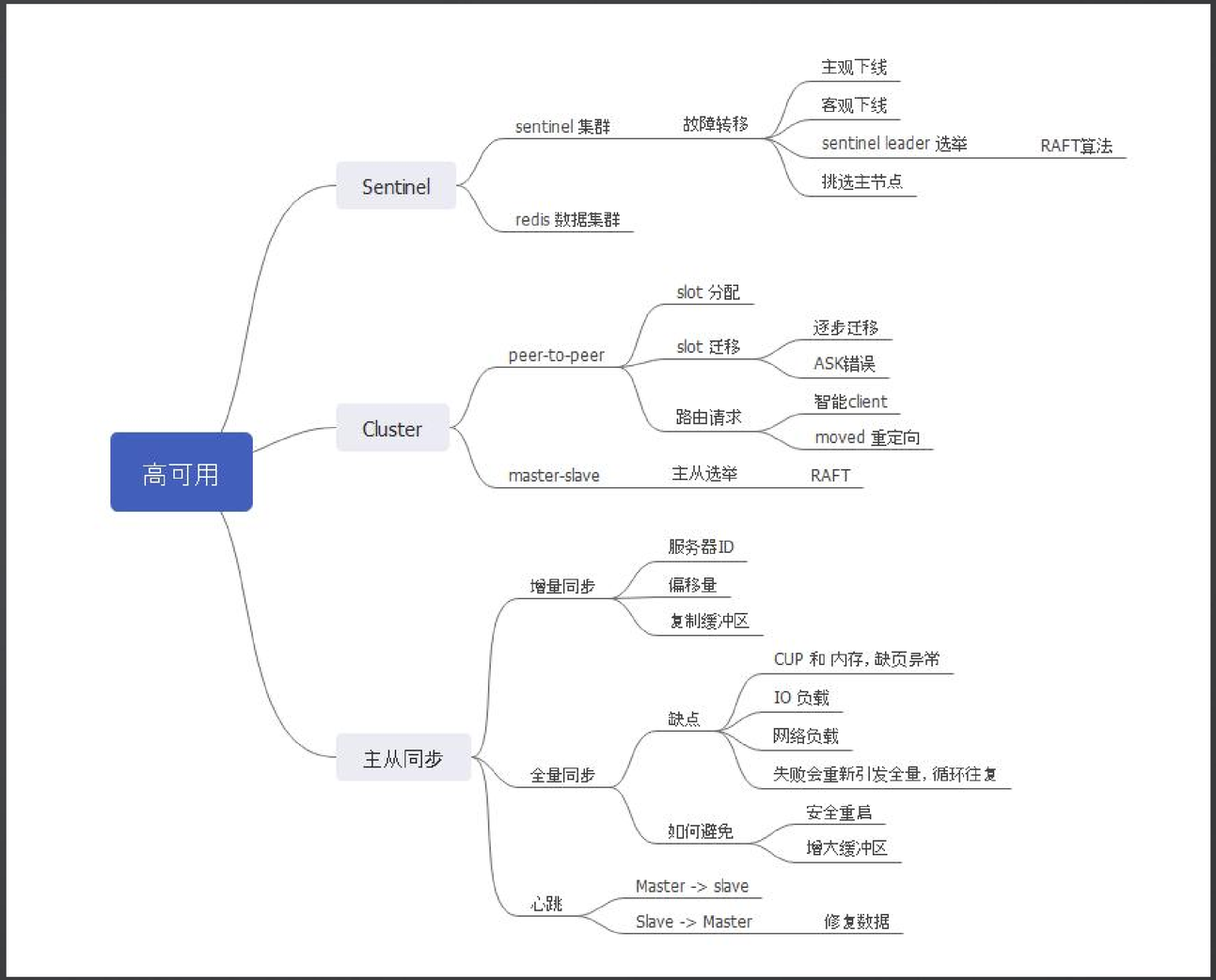
知识梳理 —— Redis 基础

- Redis 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



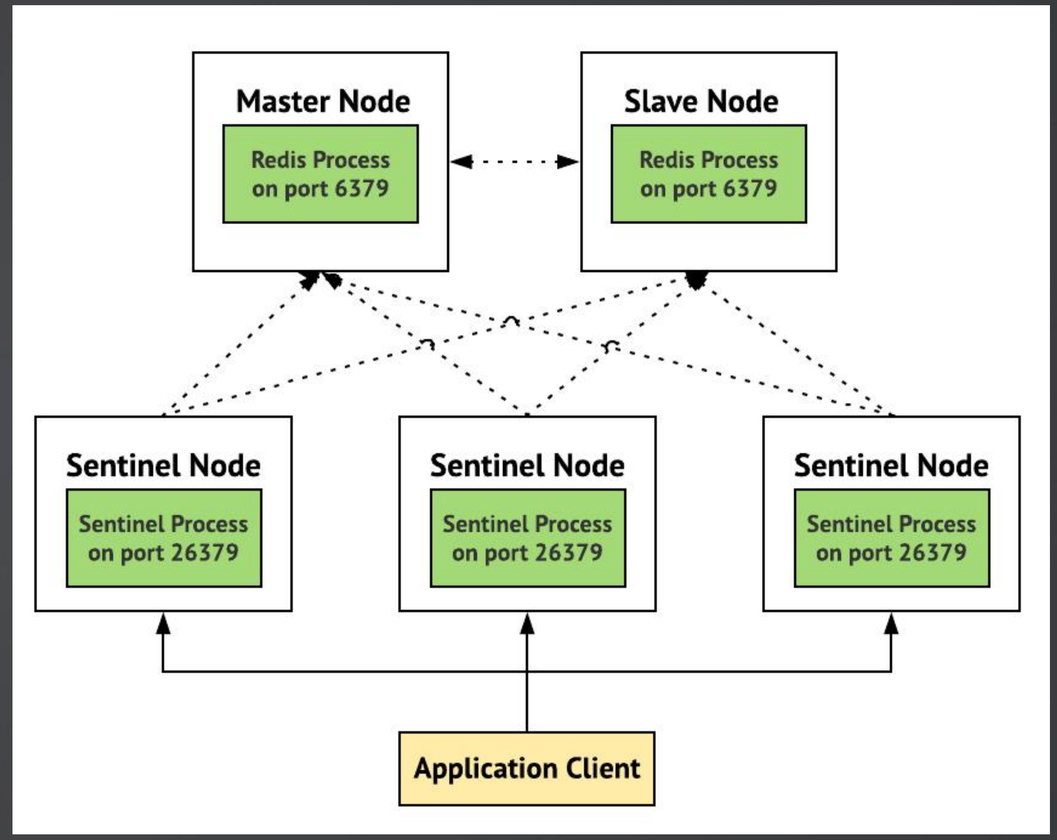
知识梳理 —— Redis 基础

- Redis 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

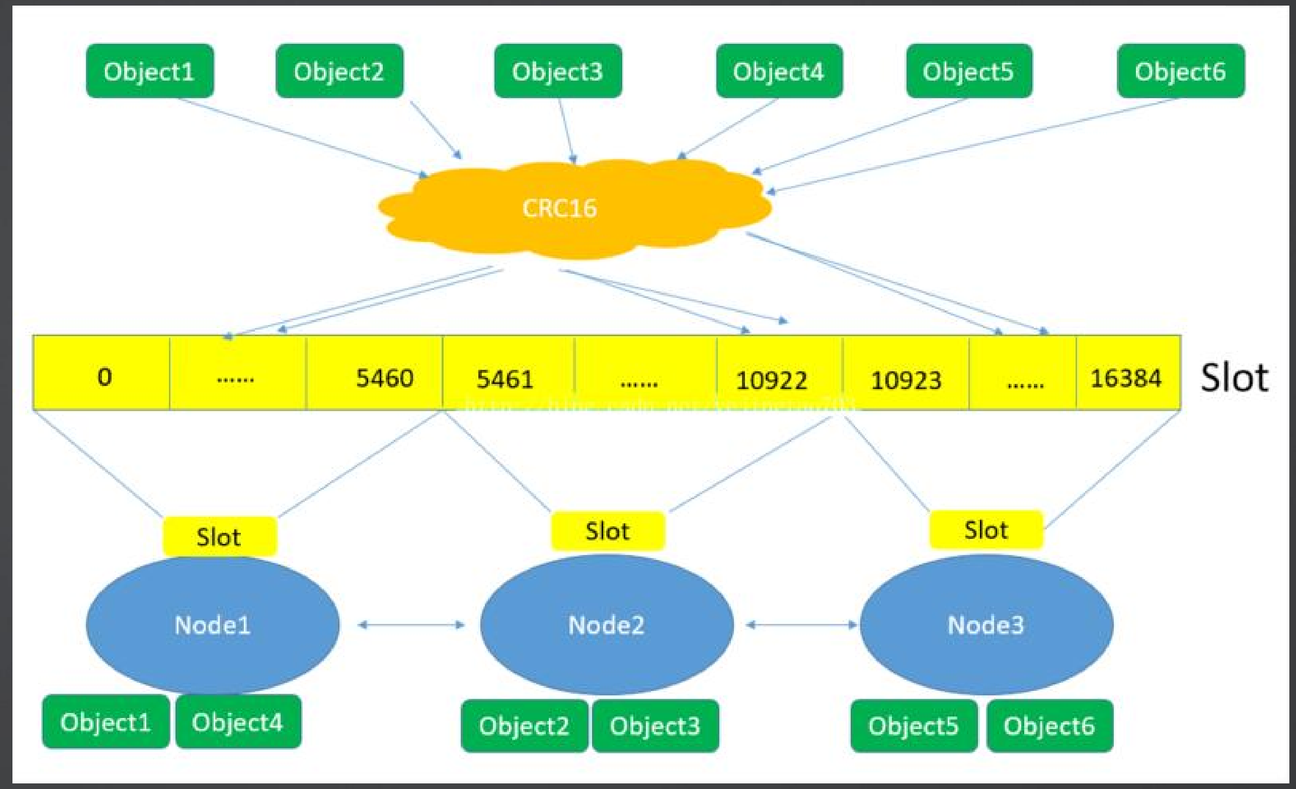


知识梳理 —— Redis 基础

- Redis 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

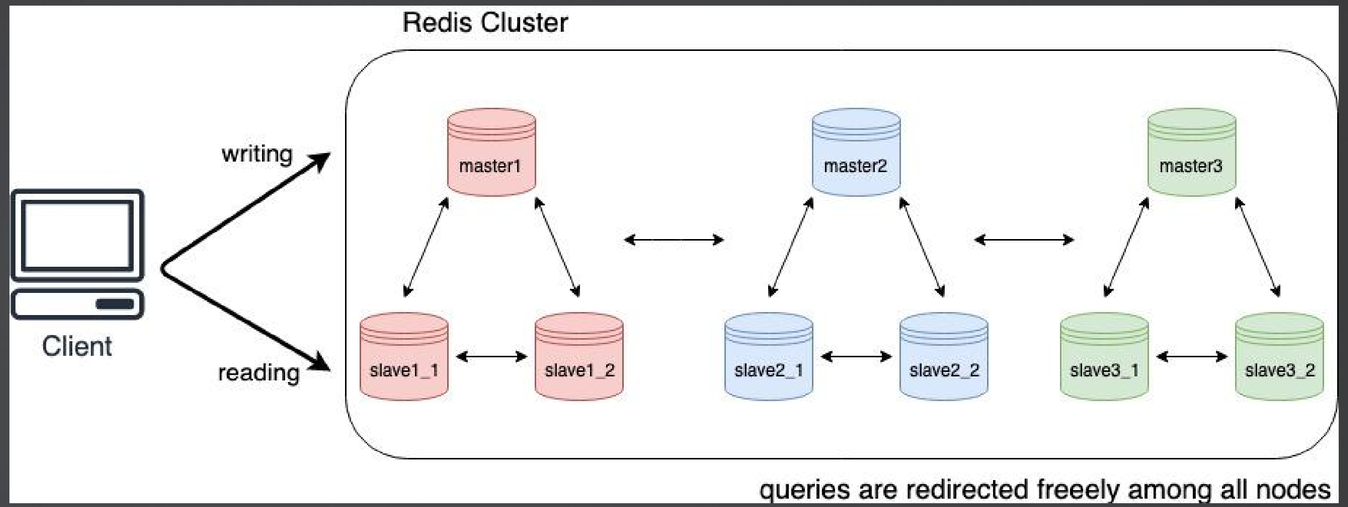
Redis Cluster 主要是利用 key 的哈希值，将其分成 16384 个槽，而后每个槽被分配到不同的主从集群上



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。

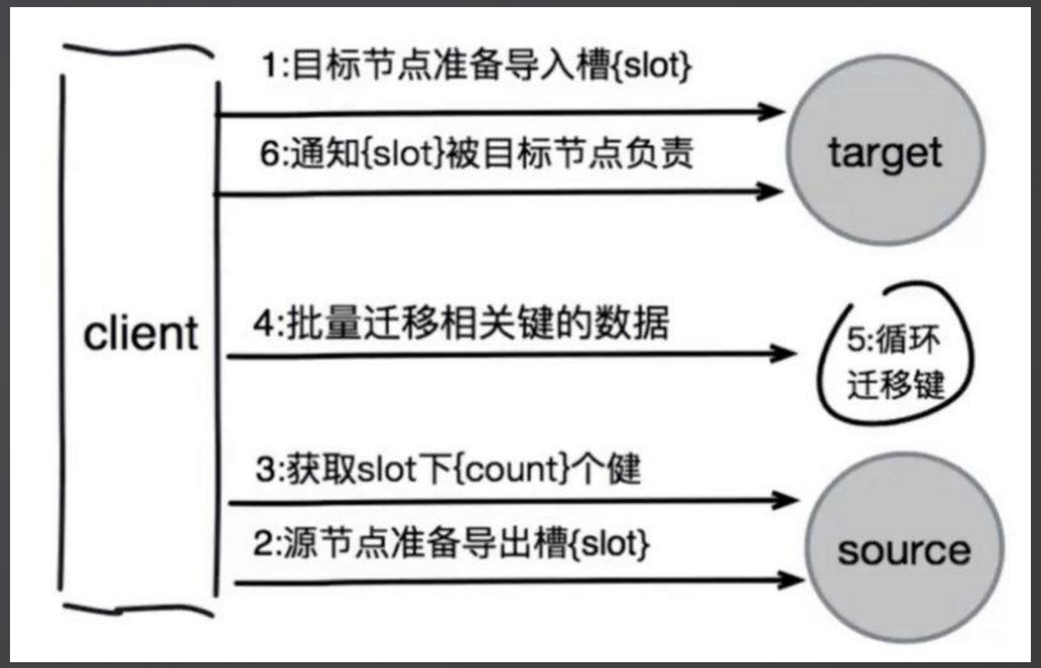


Redis 高可用——Redis Cluster

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。



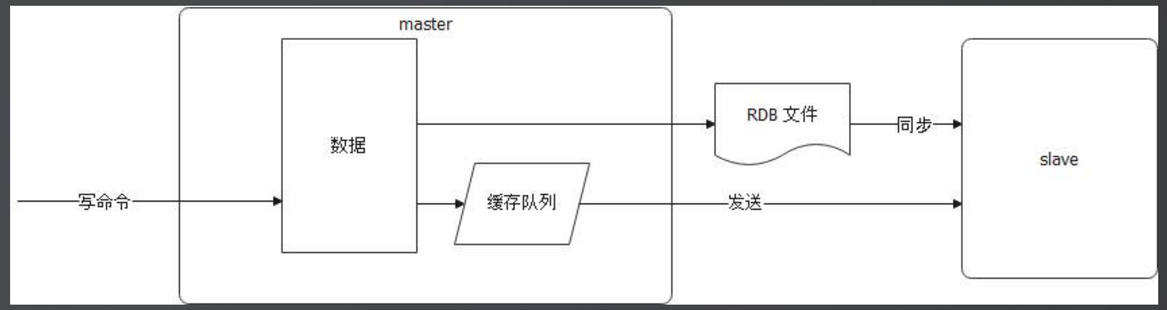
知识梳理 —— Redis 基础

- Redis 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

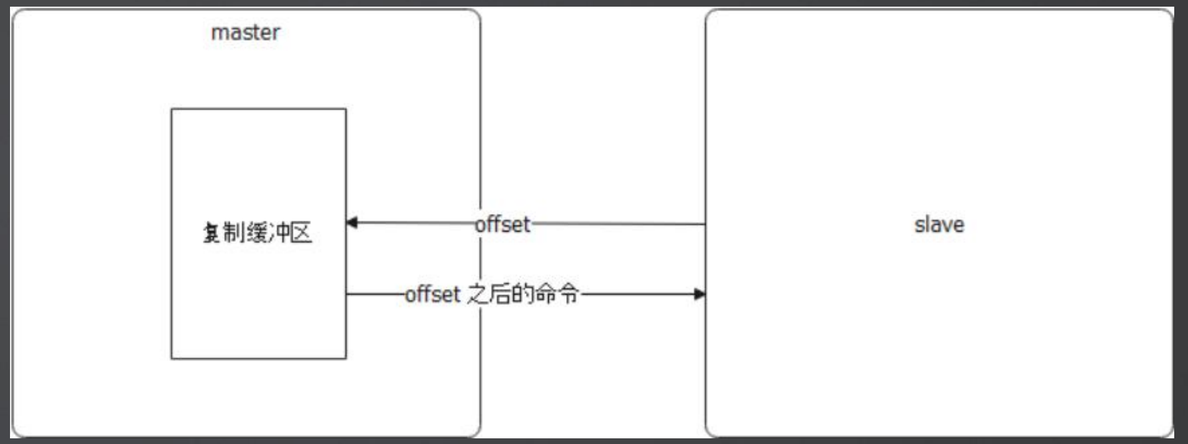
主从同步有全量同步和增量同步两种，全量同步：

- 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，载入 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

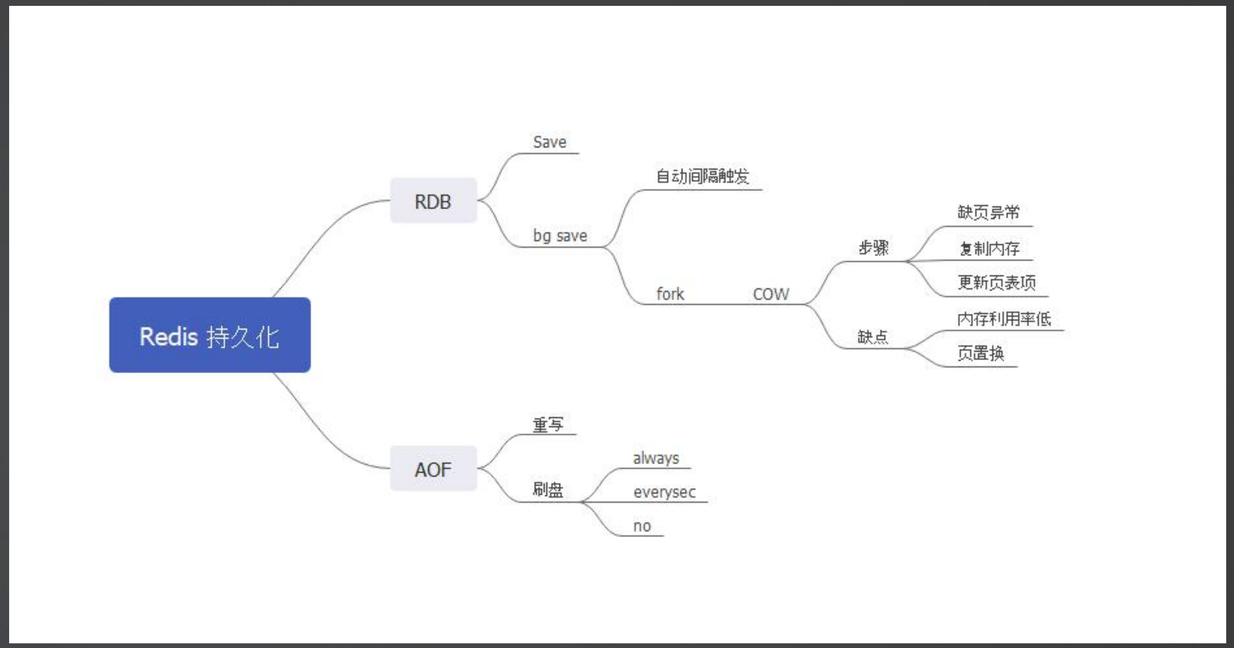
- Redis 高性能、高可用
  - Redis Sentinel
  - Redis Cluster
  - Redis 主从同步
  - Redis 持久化
  - Redis IO 模型

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。

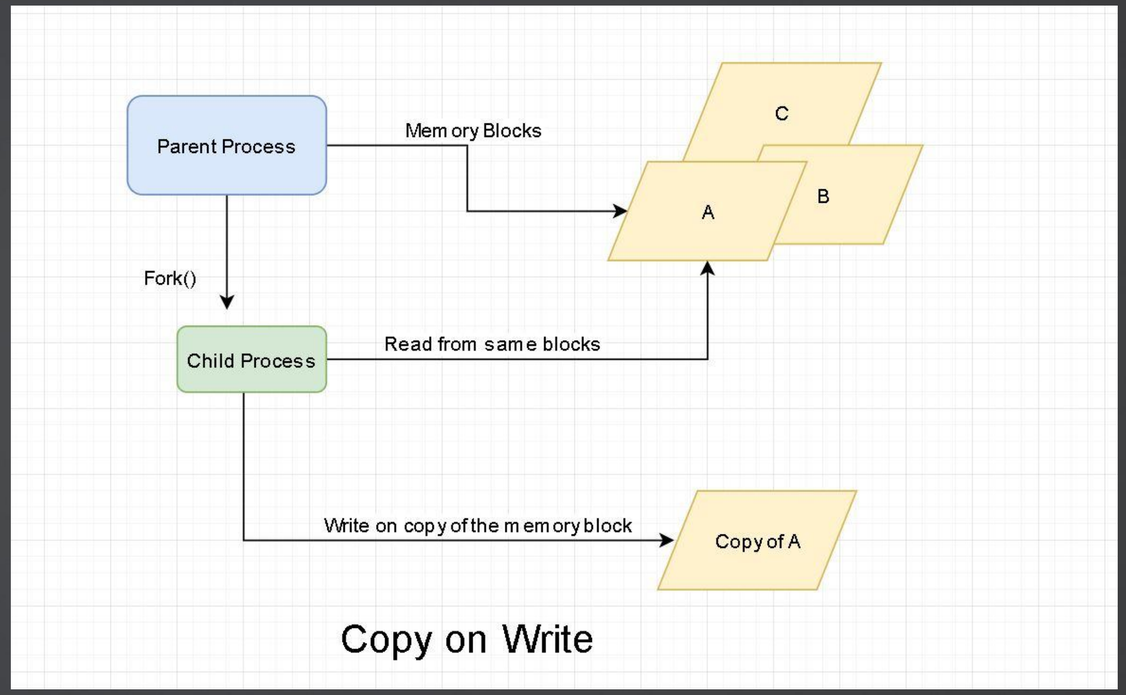
RDB 可以理解为是一个快照，直接把 Redis 内存中的数据以快照的形式保存下来。因为这个过程很消耗资源，所以分成 SAVE 和 BG SAVE 两种。BG SAVE的核心是利用 fork 和 COW 机制。

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。



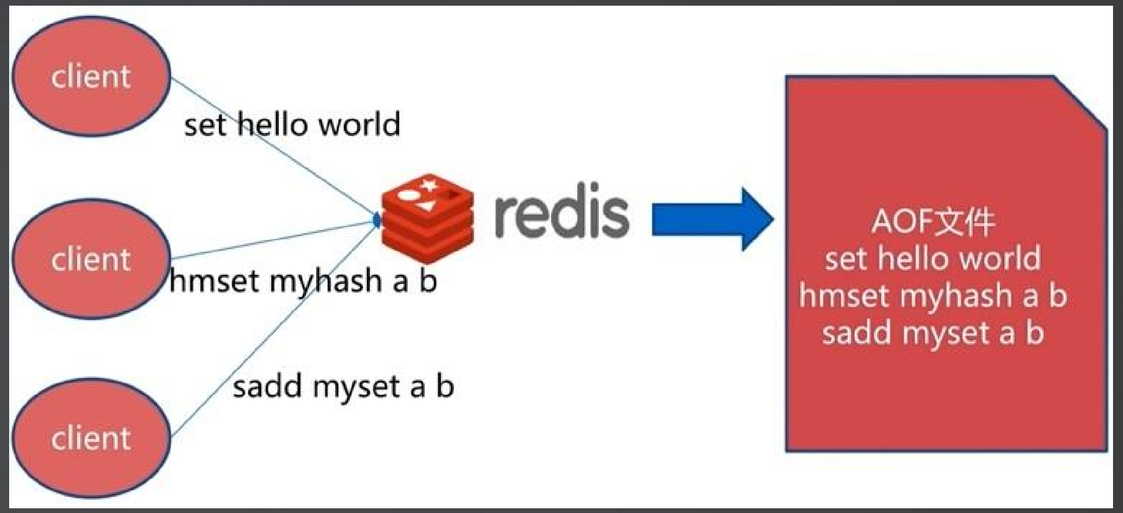
Redis 高可用——Redis BG Save

- 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



Redis 高可用—— AOF 刷盘对比

AOF 刷新磁盘的时机

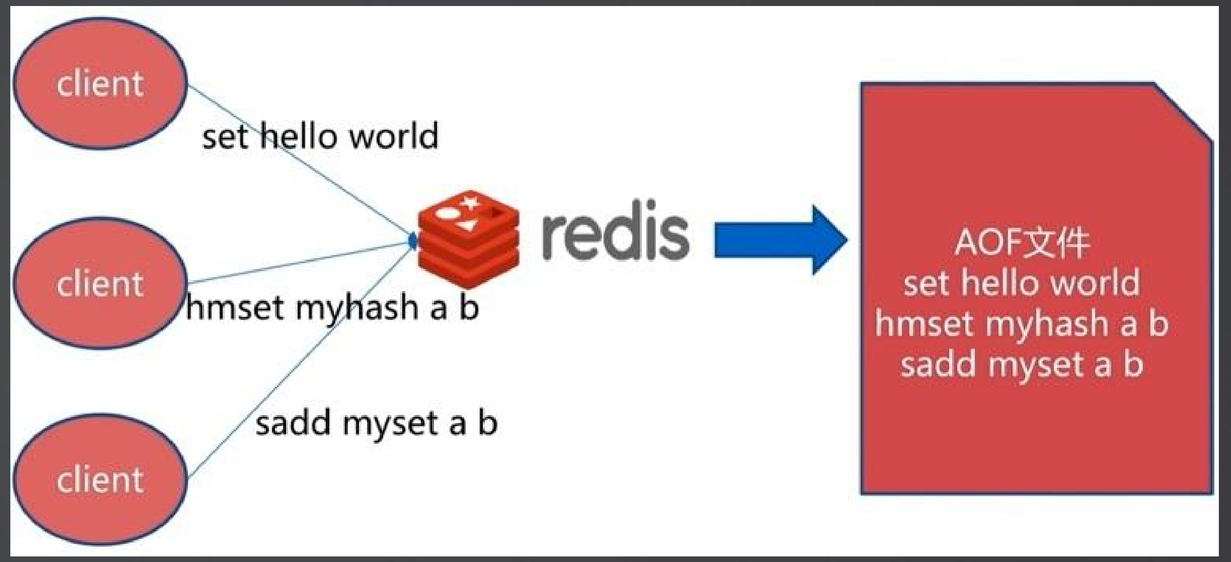
- always: 每次都刷盘
- everysec: 每秒，这意味着一般情况下会丢失一秒钟的数据。而实际上，考虑到硬盘阻塞（见后面**使用 everysec 输盘策略有什么缺点），那么可能丢失两秒的数据。
- no: 由操作系统决定

MySQL redo log 刷盘：

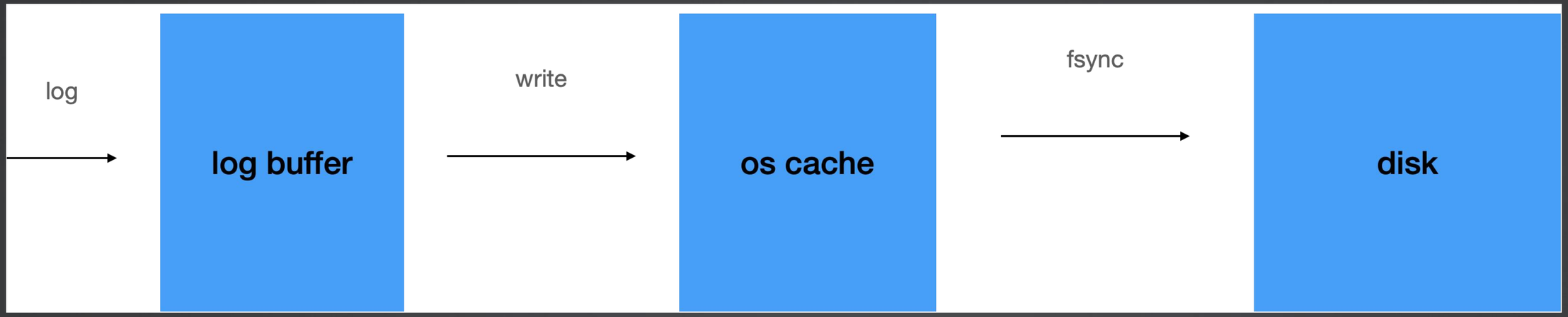
- 写到 log buffer , 每秒刷新；
- 实时刷新；
- 写到 OS cache, 每秒刷新

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



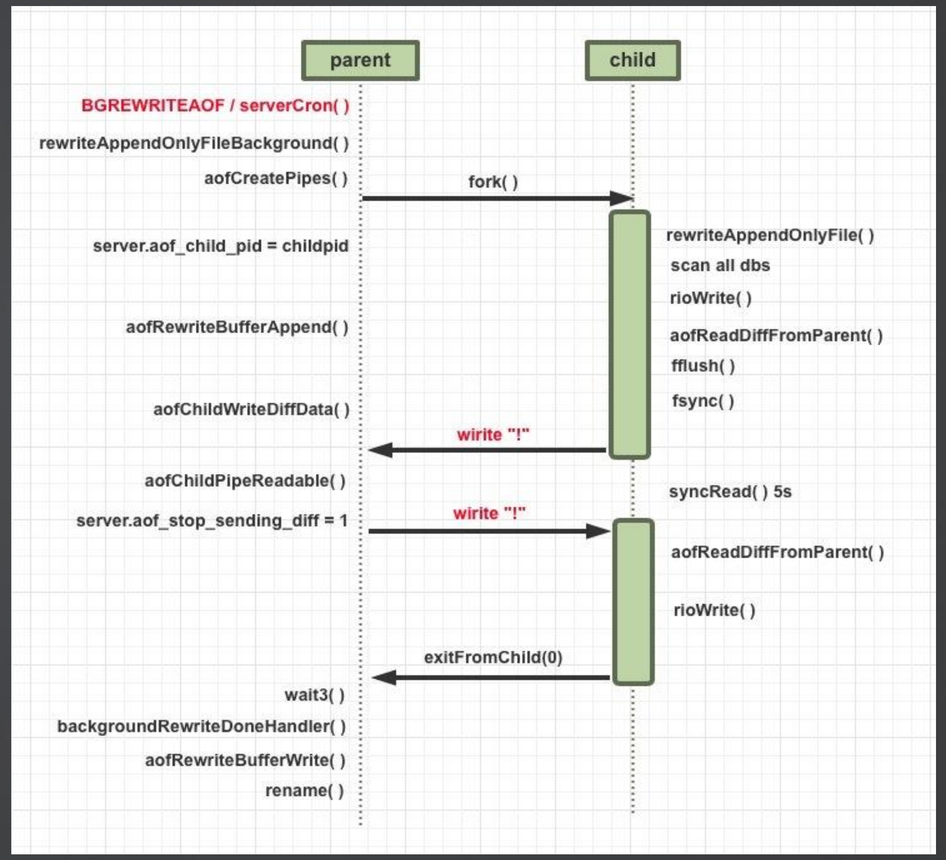
1. 中间件写到日志缓存就认为写入了；
2. 中间件写入到系统缓存（page cache）就认为写入了；
3. 中间件强制刷新到磁盘（发起了 fsync）就认为写入了；

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。

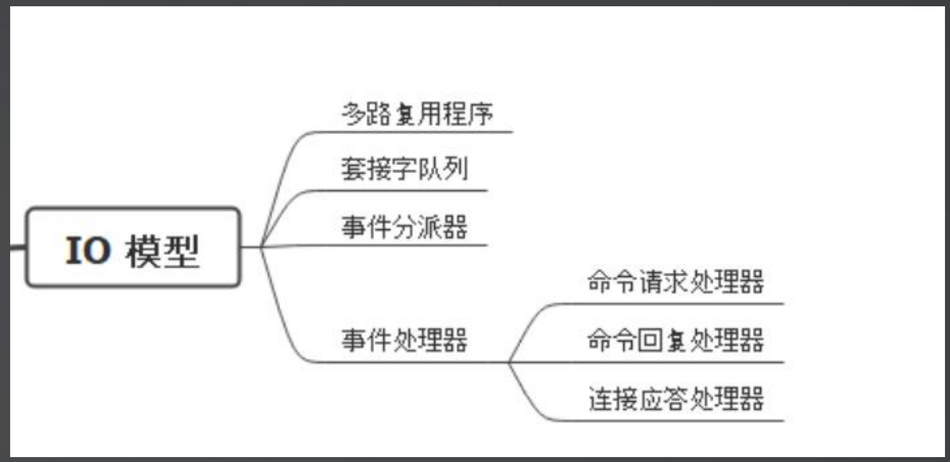


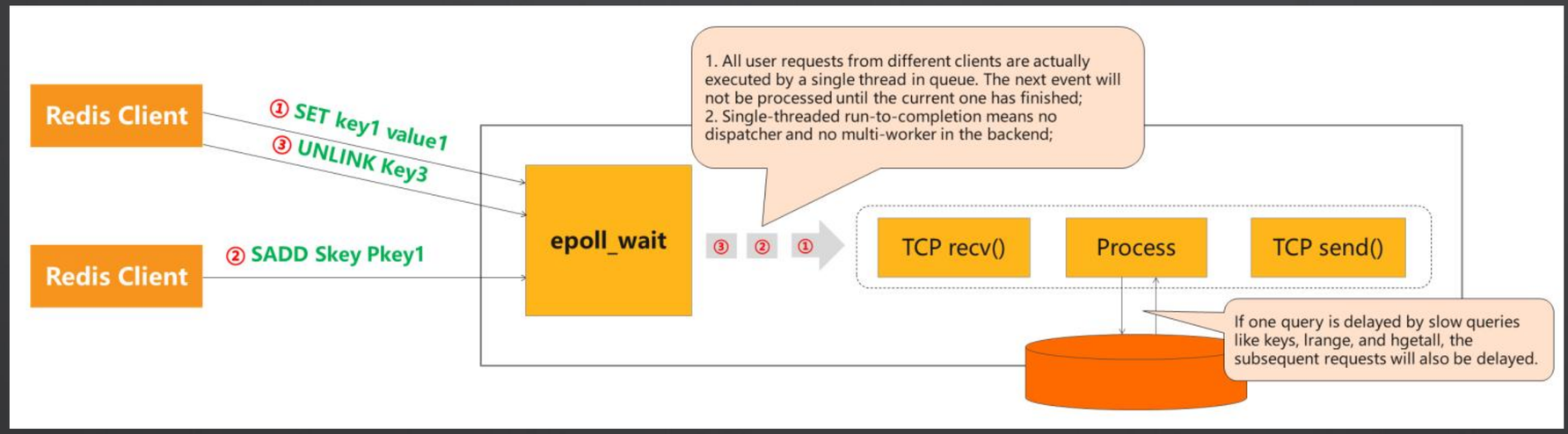
知识梳理 —— Redis 基础

- Redis 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

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。

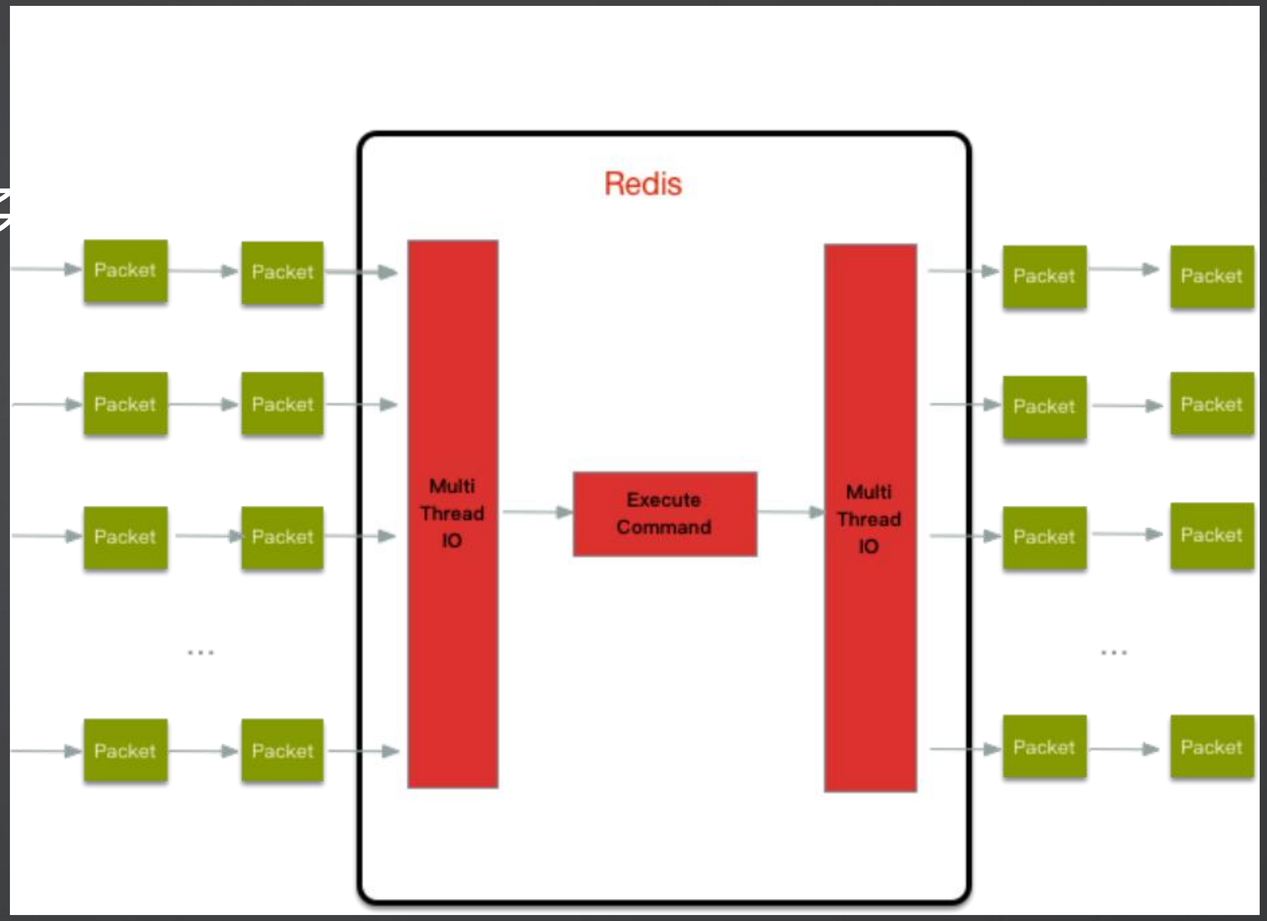




Redis 性能 —— Redis IO 模型

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

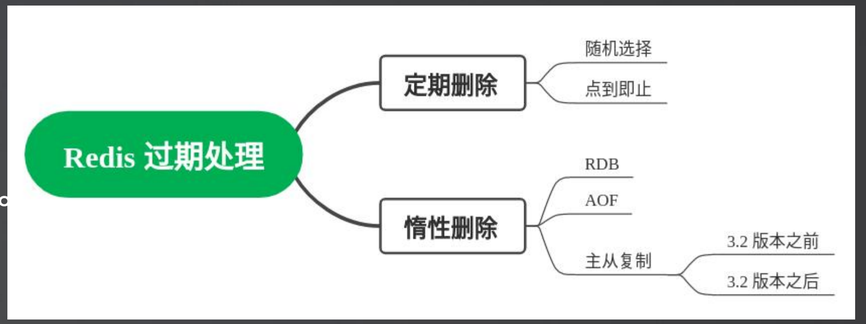


Redis 基础——过期处理

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。

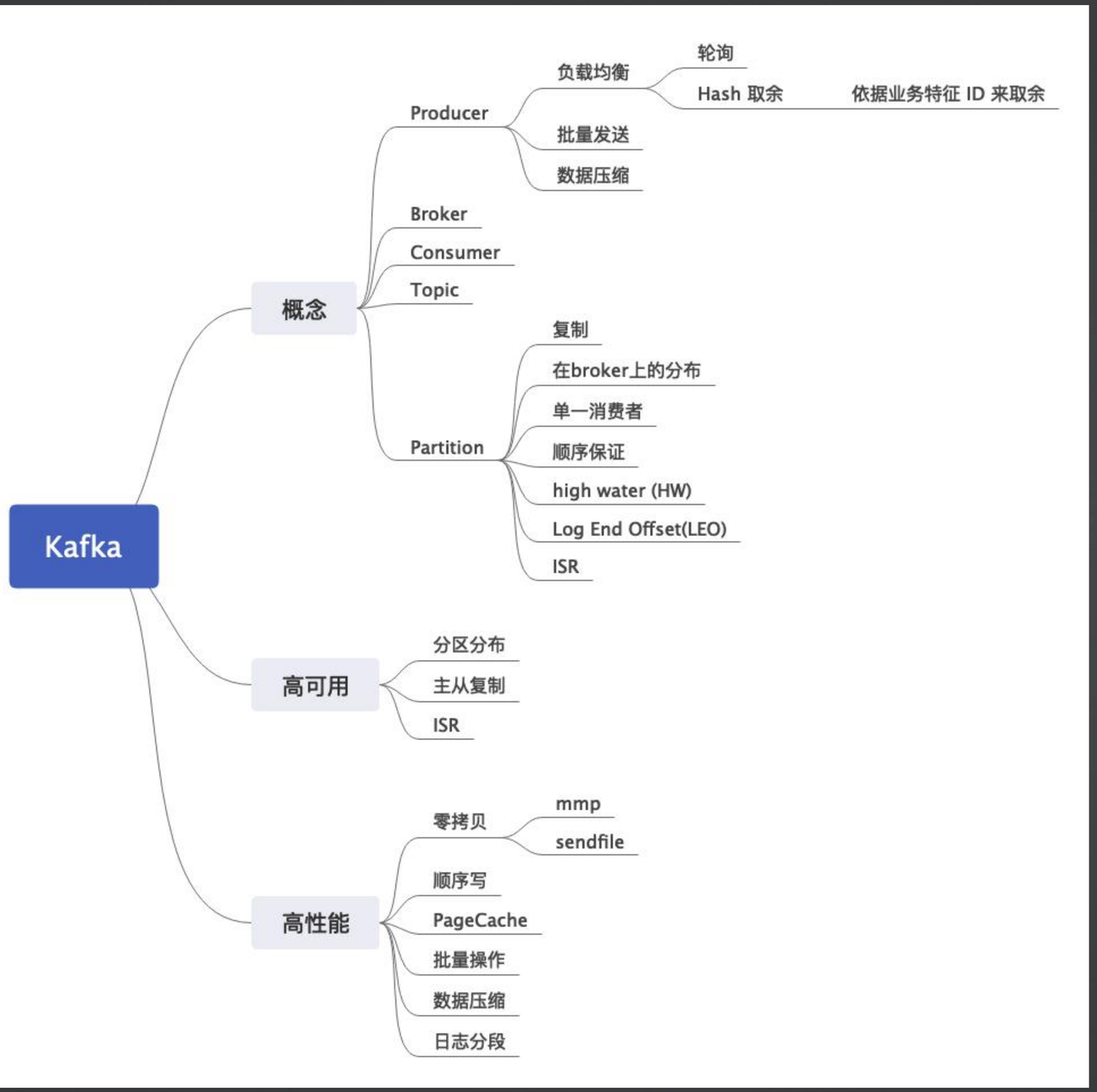
懒惰删除是指如果在访问某个 key 的时候，会检查其过期时间，如果已经过期，则会删除该键值对



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）

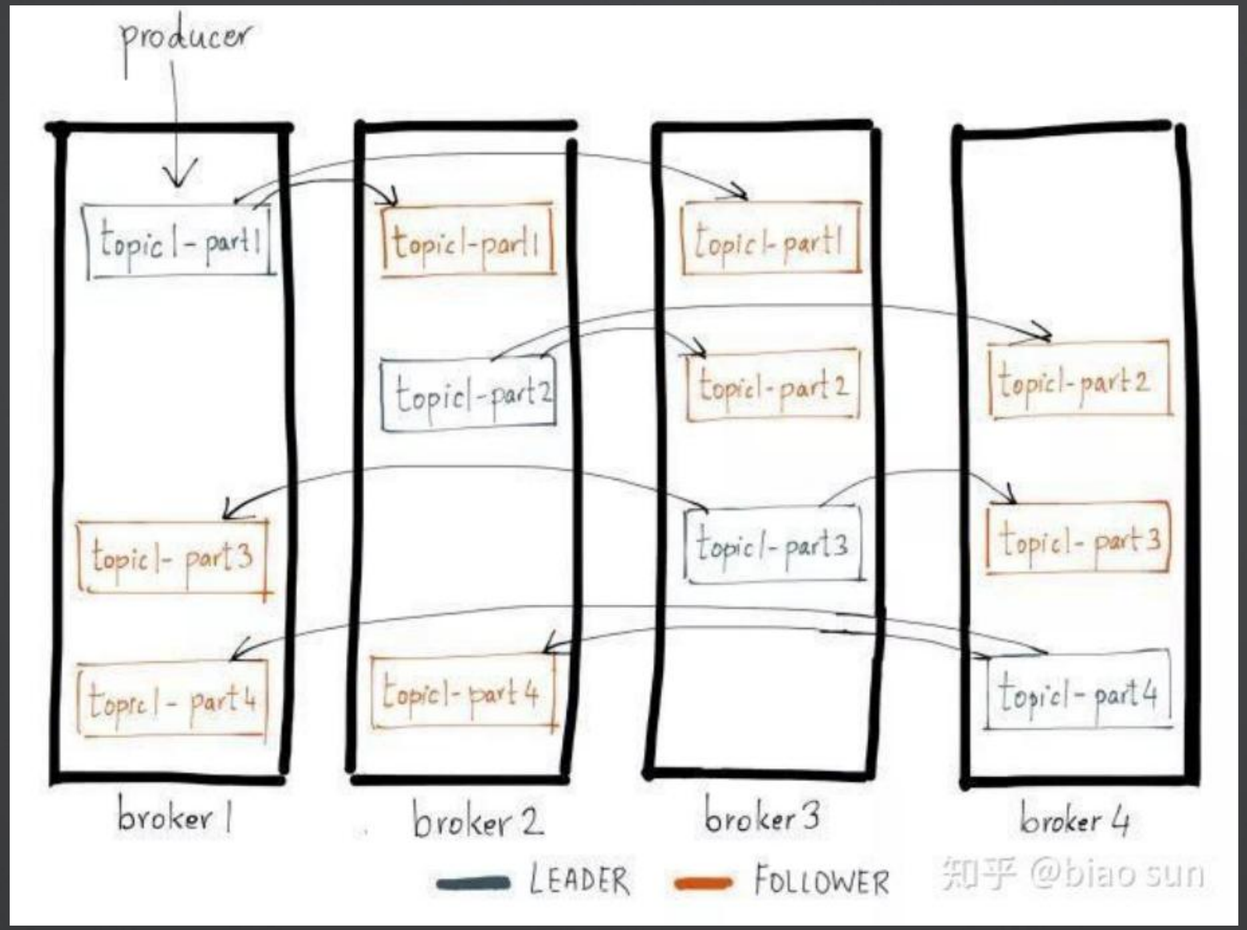
Kafka 八股文


Kafka
概念
Producer
负载均衡
轮询
Hash 取余
依据业务特征 ID 来取余
批量发送
数据压缩
Broker
Consumer
Topic
Partition
复制
在broker上的分布
单一消费者
顺序保证
high water (HW)
Log End Offset(LEO)
ISR
高可用
分区分布
主从复制
ISR
高性能
零拷贝
mmp
sendfile
顺序写
PageCache
批量操作
数据压缩
日志分段

Kafka 面试要点

- 你是否了解 Kafka？回答 kafka 的基本结构，也就是从 partition, broker 等几个基本概念开始回答；
- 你用消息队列做过什么？
- Kafka 的高性能是如何保证的？核心是零拷贝，Page Cache，顺序写，批量操作，数据压缩，日志分段存储；
- Kafka 的 ISR 是如何工作的？核心是理解Kafka 如何维护 ISR，什么情况下会导致一个 partition 进去（或者出来）ISR
- Kafka 的负载均衡策略有哪些？列举策略，要注意分析优缺点。更进一步可以讨论更加宽泛的负载均衡的做法，和 RPC 之类的负载均衡结合做对比
- 为什么 Kafka 的从 Partition 不能读取？违背直觉的问题，关键是要协调偏移量的代价太大；
- 为什么 Kafka 在消费者端采用了拉（PULL）模型？注意和 PUSH 模型做对比。最好是能够举一个适用 PUSH 的例子
- 分区过多会引起什么问题？又是一个违背直觉的问题，核心在于顺序写
- 如何确定合适的分区数量？
- 如何解决 Topic 的分区数量过多的问题？
- 如何保证消息有序性？方案有什么缺点？抓住核心，相关的消息要确保发送到同一个分区，例如 ID 为1的永远发到分区1
- Kafka能不能重复消费？当然可以。但是要强调，一般的消费者都要考虑幂等的问题
- 如何保证消息消费的幂等性？就是去重，简单就是数据库唯一索引，高级就是布隆过滤器 + 唯一索引
- 如何保证只发送（或者只消费）一次？属实没必要，做好消费幂等简单多了
- Rebalance 发生时机，rebalance 过程，rebalance 有啥影响？如何避免 rebalance？核心把 rebalance 的过程背下来
- 消息积压怎么办？没啥好办法，也就是加快消费，合并消息

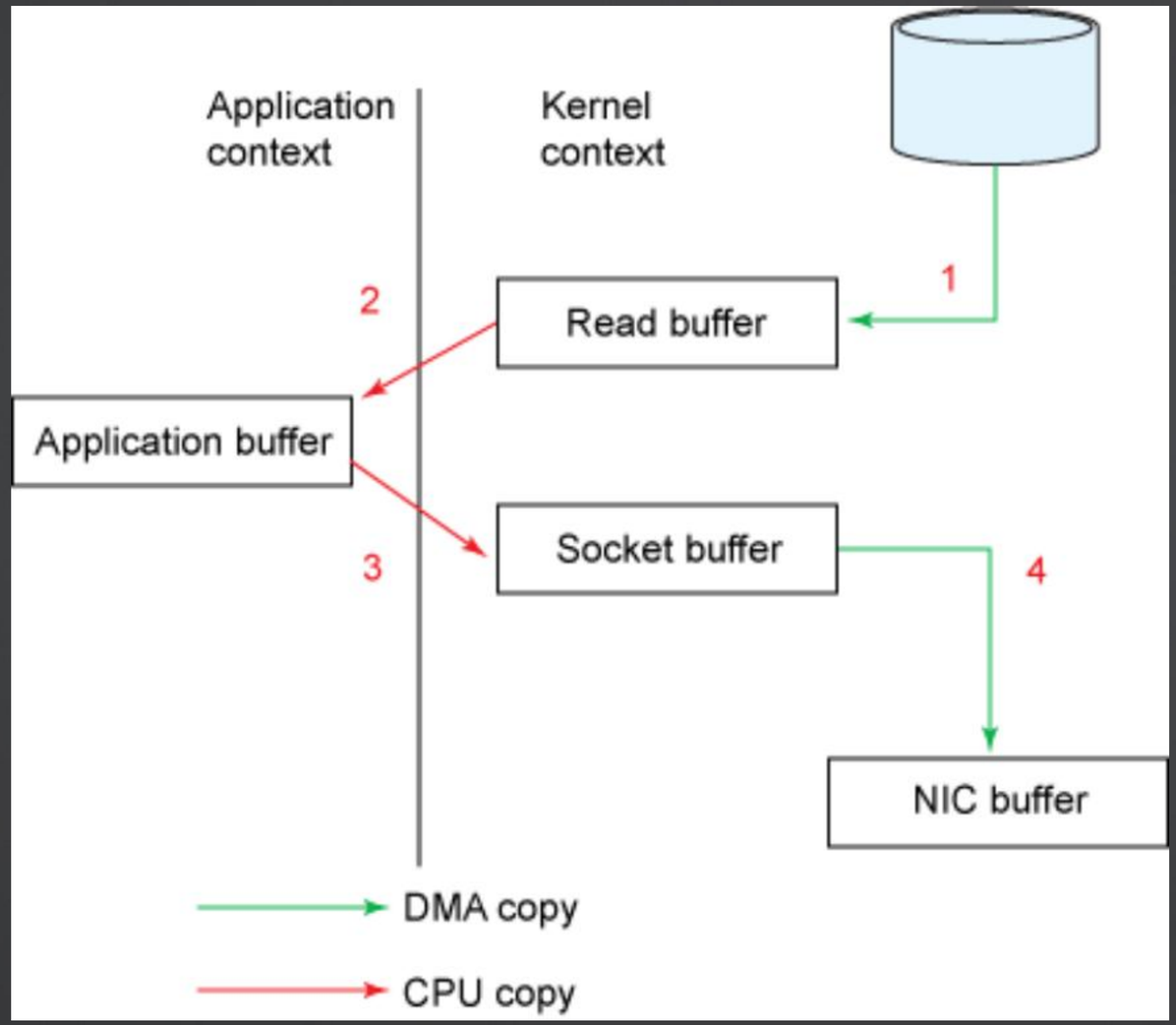
介绍一下 Kafka


producer
topic1-part1
topic1-part2
topic1-part3
topic1-part4
topic1-part1
topic1-part2
topic1-part4
topic1-part1
topic1-part2
topic1-part3
topic1-part2
topic1-part3
topic1-part4
broker 1
broker 2
broker 3
broker 4
LEADER
FOLLOWER
知乎 @biao sun

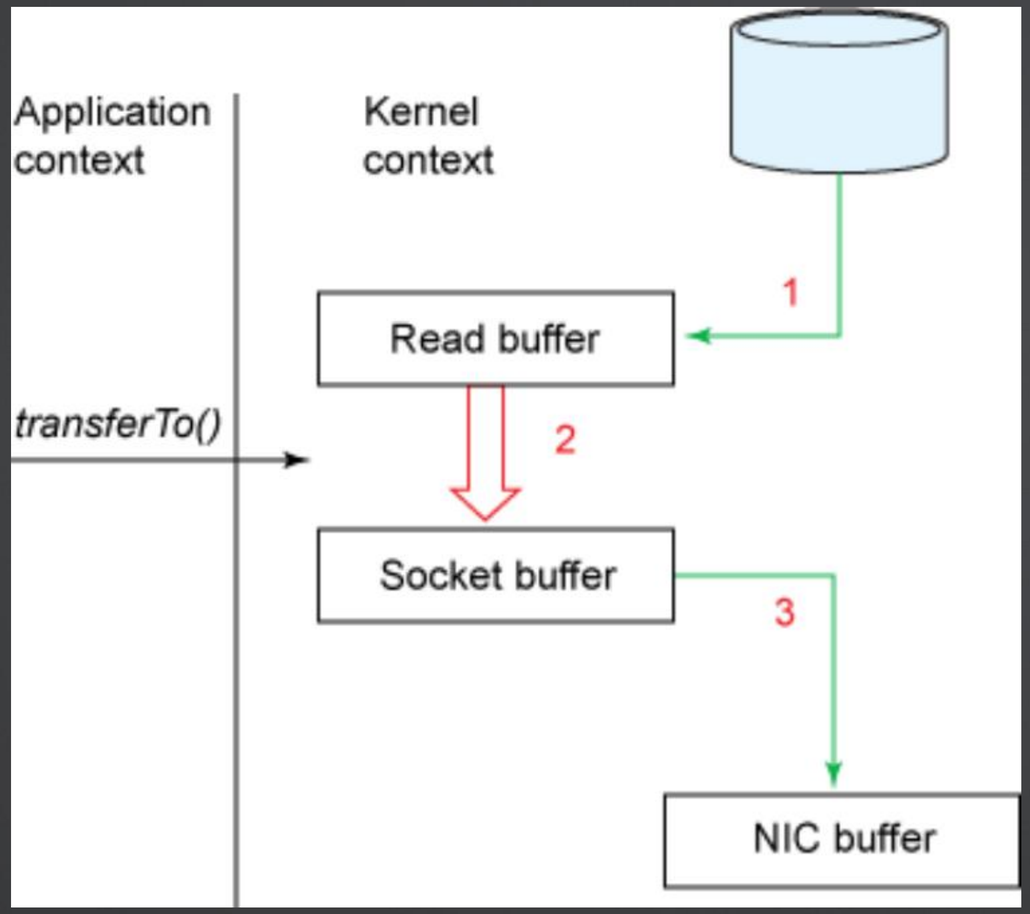
Kafka 为什么高性能

- 1. 零拷贝。在 Linux 上 Kafka 使用了两种手段，mmp (内存映射，一般我都记成妈卖批，哈哈哈) 和 sendfile，前者用于解决 Producer 写入数据，后者用于 Consumer 读取数据；
- 2. 顺序写：Kafka 的数据，可以看做是 AOF （append only file），它只允许追加数据，而不允许修改已有的数据。（后面是亮点）该手段也在数据库如 MySQL，Redis上很常见，这也是为什么我们一般说 Kafka 用机械硬盘就可以了。有人做过实验（的确有，你们可以找找，我已经找不到链接了），机械磁盘 Kafka 和 SSD Kafka 在性能上差距不大；
- 3. Page Cache：Kafka 允许落盘的时候，是写到 Page Cache的时候就返回，还是一定要刷新到磁盘（主要就是mmp之后要不要强制刷新磁盘），类似的机制在 MySQL, Redis上也是常见，（简要评价一下两种方式的区别）如果写到 Page Cache 就返回，那么会存在数据丢失的可能。
- 4. 批量操作：包括 Producer 批量发送，也包括 Broker 批量落盘。批量能够放大顺序写的优势，比如说 Producer 还没攒够一批数据发送就宕机，就会导致数据丢失；
- 5. 数据压缩：Kafka 提供了数据压缩选项，采用数据压缩能减少数据传输量，提高效率；
- 6. 日志分段存储：Kafka 将日志分成不同的段，只有最新的段可以写，别的段都只能读。同时为每一个段保存了偏移量索引文件和时间戳索引文件，采用二分法查找数据，效率极高。同时 Kafka 会确保索引文件能够全部装入内存，以避免读取索引引发磁盘 IO。（这里有一点很有意思，就是在 MySQL 上，我们也会尽量说把索引大小控制住，能够在内存装下，在讨论数据库磁盘 IO 的时候，我们很少会计算索引无法装入内存引发的磁盘 IO，而是只计算读取数据的磁盘 IO）

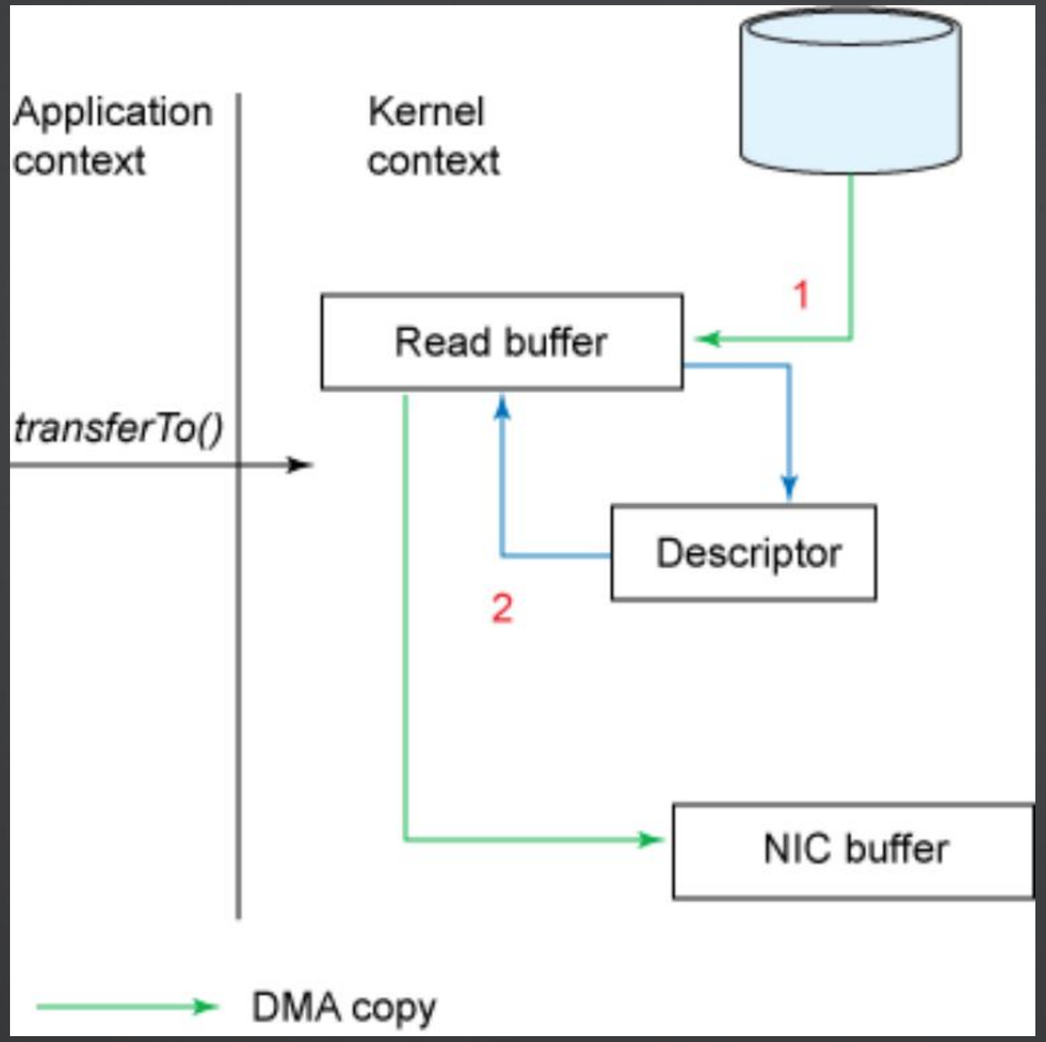
Kafka 为什么高性能——零拷贝



Kafka 为什么高性能——零拷贝



Kafka 为什么高性能——零拷贝



Kafka 为什么高性能

- 1. 零拷贝。在 Linux 上 Kafka 使用了两种手段，mmp (内存映射，一般我都记成妈卖批，哈哈哈) 和 sendfile，前者用于解决 Producer 写入数据，后者用于 Consumer 读取数据；
- 2. 顺序写：Kafka 的数据，可以看做是 AOF （append only file），它只允许追加数据，而不允许修改已有的数据。（后面是亮点）该手段也在数据库如 MySQL，Redis上很常见，这也是为什么我们一般说 Kafka 用机械硬盘就可以了。有人做过实验（的确有，你们可以找找，我已经找不到链接了），机械磁盘 Kafka 和 SSD Kafka 在性能上差距不大；
- 3. Page Cache：Kafka 允许落盘的时候，是写到 Page Cache的时候就返回，还是一定要刷新到磁盘（主要就是mmp之后要不要强制刷新磁盘），类似的机制在 MySQL, Redis上也是常见，（简要评价一下两种方式的区别）如果写到 Page Cache 就返回，那么会存在数据丢失的可能。
- 4. 批量操作：包括 Producer 批量发送，也包括 Broker 批量落盘。批量能够放大顺序写的优势，比如说 Producer 还没攒够一批数据发送就宕机，就会导致数据丢失；
- 5. 数据压缩：Kafka 提供了数据压缩选项，采用数据压缩能减少数据传输量，提高效率；
- 6. 日志分段存储：Kafka 将日志分成不同的段，只有最新的段可以写，别的段都只能读。同时为每一个段保存了偏移量索引文件和时间戳索引文件，采用二分法查找数据，效率极高。同时 Kafka 会确保索引文件能够全部装入内存，以避免读取索引引发磁盘 IO。（这里有一点很有意思，就是在 MySQL 上，我们也会尽量说把索引大小控制住，能够在内存装下，在讨论数据库磁盘 IO 的时候，我们很少会计算索引无法装入内存引发的磁盘 IO，而是只计算读取数据的磁盘 IO）

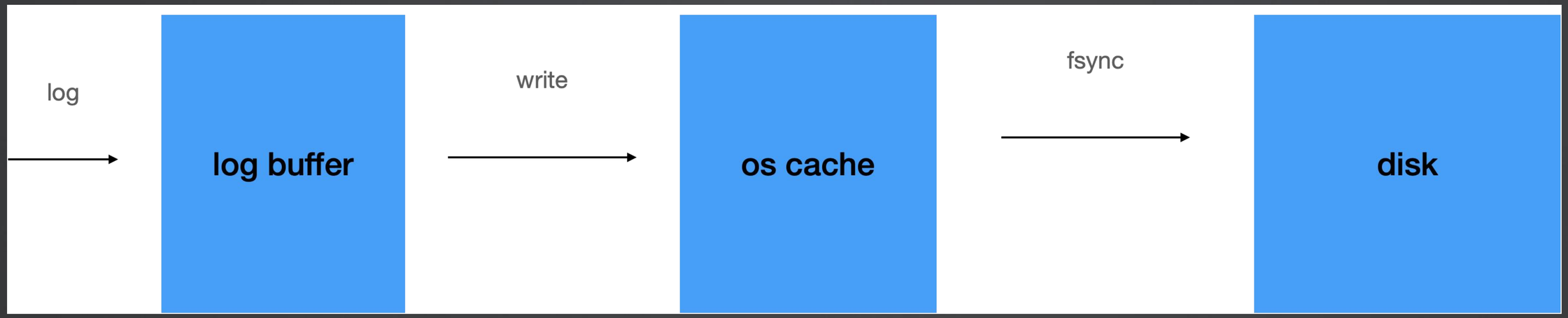
顺序写，AOF，WAL

- MySQL redolog, undolog, binlog
- Redis AOF 持久化机制
- Kafka

Kafka 为什么高性能

- 1. 零拷贝。在 Linux 上 Kafka 使用了两种手段，mmp (内存映射，一般我都记成妈卖批，哈哈哈) 和 sendfile，前者用于解决 Producer 写入数据，后者用于 Consumer 读取数据；
- 2. 顺序写：Kafka 的数据，可以看做是 AOF （append only file），它只允许追加数据，而不允许修改已有的数据。（后面是亮点）该手段也在数据库如 MySQL，Redis上很常见，这也是为什么我们一般说 Kafka 用机械硬盘就可以了。有人做过实验（的确有，你们可以找找，我已经找不到链接了），机械磁盘 Kafka 和 SSD Kafka 在性能上差距不大；
- 3. Page Cache：Kafka 允许落盘的时候，是写到 Page Cache的时候就返回，还是一定要刷新到磁盘（主要就是mmp之后要不要强制刷新磁盘），类似的机制在 MySQL, Redis上也是常见，（简要评价一下两种方式的区别）如果写到 Page Cache 就返回，那么会存在数据丢失的可能。
- 4. 批量操作：包括 Producer 批量发送，也包括 Broker 批量落盘。批量能够放大顺序写的优势，比如说 Producer 还没攒够一批数据发送就宕机，就会导致数据丢失；
- 5. 数据压缩：Kafka 提供了数据压缩选项，采用数据压缩能减少数据传输量，提高效率；
- 6. 日志分段存储：Kafka 将日志分成不同的段，只有最新的段可以写，别的段都只能读。同时为每一个段保存了偏移量索引文件和时间戳索引文件，采用二分法查找数据，效率极高。同时 Kafka 会确保索引文件能够全部装入内存，以避免读取索引引发磁盘 IO。（这里有一点很有意思，就是在 MySQL 上，我们也会尽量说把索引大小控制住，能够在内存装下，在讨论数据库磁盘 IO 的时候，我们很少会计算索引无法装入内存引发的磁盘 IO，而是只计算读取数据的磁盘 IO）

write, page cache, flush



write, page cache, flush

- MySQL redolog undolog binlog
- Redis AOF
- Kafka

write, page cache, flush

- 写到自己的缓存（应用缓存）
- 写到 page cache
- 刷新策略：
  - 交给操作系统——MySQL，Redis，Kafka
  - 按秒——Mysql，Redis
  - 强制——MySQL，Redis，Kafka

Kafka 为什么高性能

- 1. 零拷贝。在 Linux 上 Kafka 使用了两种手段，mmp (内存映射，一般我都记成妈卖批，哈哈哈) 和 sendfile，前者用于解决 Producer 写入数据，后者用于 Consumer 读取数据；
- 2. 顺序写：Kafka 的数据，可以看做是 AOF （append only file），它只允许追加数据，而不允许修改已有的数据。（后面是亮点）该手段也在数据库如 MySQL，Redis上很常见，这也是为什么我们一般说 Kafka 用机械硬盘就可以了。有人做过实验（的确有，你们可以找找，我已经找不到链接了），机械磁盘 Kafka 和 SSD Kafka 在性能上差距不大；
- 3. Page Cache：Kafka 允许落盘的时候，是写到 Page Cache的时候就返回，还是一定要刷新到磁盘（主要就是mmp之后要不要强制刷新磁盘），类似的机制在 MySQL, Redis上也是常见，（简要评价一下两种方式的区别）如果写到 Page Cache 就返回，那么会存在数据丢失的可能。
- 4. 批量操作：包括 Producer 批量发送，也包括 Broker 批量落盘。批量能够放大顺序写的优势，比如说 Producer 还没攒够一批数据发送就宕机，就会导致数据丢失；
- 5. 数据压缩：Kafka 提供了数据压缩选项，采用数据压缩能减少数据传输量，提高效率；
- 6. 日志分段存储：Kafka 将日志分成不同的段，只有最新的段可以写，别的段都只能读。同时为每一个段保存了偏移量索引文件和时间戳索引文件，采用二分法查找数据，效率极高。同时 Kafka 会确保索引文件能够全部装入内存，以避免读取索引引发磁盘 IO。（这里有一点很有意思，就是在 MySQL 上，我们也会尽量说把索引大小控制住，能够在内存装下，在讨论数据库磁盘 IO 的时候，我们很少会计算索引无法装入内存引发的磁盘 IO，而是只计算读取数据的磁盘 IO）

Kafka 为什么高性能

- 1. 零拷贝。在 Linux 上 Kafka 使用了两种手段，mmp (内存映射，一般我都记成妈卖批，哈哈哈) 和 sendfile，前者用于解决 Producer 写入数据，后者用于 Consumer 读取数据；
- 2. 顺序写：Kafka 的数据，可以看做是 AOF （append only file），它只允许追加数据，而不允许修改已有的数据。（后面是亮点）该手段也在数据库如 MySQL，Redis上很常见，这也是为什么我们一般说 Kafka 用机械硬盘就可以了。有人做过实验（的确有，你们可以找找，我已经找不到链接了），机械磁盘 Kafka 和 SSD Kafka 在性能上差距不大；
- 3. Page Cache：Kafka 允许落盘的时候，是写到 Page Cache的时候就返回，还是一定要刷新到磁盘（主要就是mmp之后要不要强制刷新磁盘），类似的机制在 MySQL, Redis上也是常见，（简要评价一下两种方式的区别）如果写到 Page Cache 就返回，那么会存在数据丢失的可能。
- 4. 批量操作：包括 Producer 批量发送，也包括 Broker 批量落盘。批量能够放大顺序写的优势，比如说 Producer 还没攒够一批数据发送就宕机，就会导致数据丢失；
- 5. 数据压缩：Kafka 提供了数据压缩选项，采用数据压缩能减少数据传输量，提高效率；
- 6. 日志分段存储：Kafka 将日志分成不同的段，只有最新的段可以写，别的段都只能读。同时为每一个段保存了偏移量索引文件和时间戳索引文件，采用二分法查找数据，效率极高。同时 Kafka 会确保索引文件能够全部装入内存，以避免读取索引引发磁盘 IO。（这里有一点很有意思，就是在 MySQL 上，我们也会尽量说把索引大小控制住，能够在内存装下，在讨论数据库磁盘 IO 的时候，我们很少会计算索引无法装入内存引发的磁盘 IO，而是只计算读取数据的磁盘 IO）

Kafka 为什么高性能

- 1. 零拷贝。在 Linux 上 Kafka 使用了两种手段，mmp (内存映射，一般我都记成妈卖批，哈哈哈) 和 sendfile，前者用于解决 Producer 写入数据，后者用于 Consumer 读取数据；
- 2. 顺序写：Kafka 的数据，可以看做是 AOF （append only file），它只允许追加数据，而不允许修改已有的数据。（后面是亮点）该手段也在数据库如 MySQL，Redis上很常见，这也是为什么我们一般说 Kafka 用机械硬盘就可以了。有人做过实验（的确有，你们可以找找，我已经找不到链接了），机械磁盘 Kafka 和 SSD Kafka 在性能上差距不大；
- 3. Page Cache：Kafka 允许落盘的时候，是写到 Page Cache的时候就返回，还是一定要刷新到磁盘（主要就是mmp之后要不要强制刷新磁盘），类似的机制在 MySQL, Redis上也是常见，（简要评价一下两种方式的区别）如果写到 Page Cache 就返回，那么会存在数据丢失的可能。
- 4. 批量操作：包括 Producer 批量发送，也包括 Broker 批量落盘。批量能够放大顺序写的优势，比如说 Producer 还没攒够一批数据发送就宕机，就会导致数据丢失；
- 5. 数据压缩：Kafka 提供了数据压缩选项，采用数据压缩能减少数据传输量，提高效率；
- 6. 日志分段存储：Kafka 将日志分成不同的段，只有最新的段可以写，别的段都只能读。同时为每一个段保存了偏移量索引文件和时间戳索引文件，采用二分法查找数据，效率极高。同时 Kafka 会确保索引文件能够全部装入内存，以避免读取索引引发磁盘 IO。（这里有一点很有意思，就是在 MySQL 上，我们也会尽量说把索引大小控制住，能够在内存装下，在讨论数据库磁盘 IO 的时候，我们很少会计算索引无法装入内存引发的磁盘 IO，而是只计算读取数据的磁盘 IO）

Kafka 的 ISR 是如何工作的

ISR 是分区同步的概念。Kafka 为每个主分区维护了一个 ISR，处于 ISR 的分区意味着与主分区保持了同步（所以主分区也在 ISR 里面）。

当 Producer 写入消息的时候，需要等 ISR 里面分区的确认，当 ISR 确认之后，就被认为消息已经提交成功了。ISR 里面的分区会定时从主分区里面拉取数据，如果长时间未拉取，或者数据落后太多，分区会被移出 ISR。ISR 里面分区已经同步的偏移量被称为 LEO（Log End Offset），最小的 LEO 称为 HW，也就是**消费者可以消费的最新消息**。

（高水位，high water，这个用木桶来比喻就很生动，ISR 里面的分区已同步消息就是木板，高水位就取决于最短的那个木板，也就是同步最落后的）

当主分区挂掉的时候，会从 ISR 里面选举一个新的主分区出来。

Kafka 的 ISR 是如何工作的 ——进一步解释

我们在 Producer 里面可以控制 ACK 机制。Producer 可以配置成三种：

- 1. Producer 发出去就算成功；
- 2. Producer 发出去，主分区写入本地磁盘就算成功；
- 3. Producer 发出去，ISR 所有的分区都写入磁盘，就算成功；

其性能依次下降，但是可靠性依次上升。

因为 ISR 里面包含了主分区，也就是说，如果整个 ISR 只有主分区，那么全部写入就退化为主分区写入。所以在可靠性要求非常高的情况下，我们要求 ISR 中分区不能少于三个（有人说两个）。该参数可以在 Broker 中配置（min.insync.replicas）

Kafka 的 ISR 是如何工作的 ——扩展到主从同步

ISR 的同步机制和其它中间件机制也是类似的，在涉及主从同步的时候都要在性能和可靠性之间做取舍。通常的选项都是：

1. 主写入就认为成功
2. 主写入，至少一个从写入就认为成功；
3. 主写入，大部分从库写入就认为成功（一般“大部分”是可以配置的，从这个意义上来说，2和3可以合并为一点）；
4. 主写入，所有从库写入就认为成功；

为什么 Kafka 的从 Partition 不能读取？

首先是 Kafka 自身的限制，即 Kafka 强制要求一个 Partition 只能有一个 Consumer，因此 Consumer 天然只需要消费主 Partition 就可以。

那么假如说 Kafka 放开这种限制，比如说有多个 Consumer，分别从主 Partition 和从 Partition 上读取数据，那么会出现一个问题：即偏移量如何同步的问题。例如一个 Consumer 从 Partition A 读取了 0- 100 的消息，那么另外一个 Consumer 从 Partition B 上读取，就只能读取 100 之后的数据。那么 Kafka 就需要在不同的 Partition 之间协调这个已读取偏移量。而这是分布式一致性的问题，难以解决。

MySQL 的主从模式比起来，并没有这种问题，即 MySQL 不需要进行类似偏移量的协商。

而从另外一个角度来说，Kafka 的读取压力是远小于 MySQL 的，毕竟一个 Topic，是不会有特别多的消费者的。并且 Kafka 也不需要支持复杂查询，所以完全没必要读取从 Partition 的数据。

Kafka 面试要点

- 你是否了解 Kafka？回答 kafka 的基本结构，也就是从 partition, broker 等几个基本概念开始回答；
- 你用消息队列做过什么？
- Kafka 的高性能是如何保证的？核心是零拷贝，Page Cache，顺序写，批量操作，数据压缩，日志分段存储；
- Kafka 的 ISR 是如何工作的？核心是理解Kafka 如何维护 ISR，什么情况下会导致一个 partition 进去（或者出来）ISR
- Kafka 的负载均衡策略有哪些？列举策略，要注意分析优缺点。更进一步可以讨论更加宽泛的负载均衡的做法，和 RPC 之类的负载均衡结合做对比
- 为什么 Kafka 的从 Partition 不能读取？违背直觉的问题，关键是要协调偏移量的代价太大；
- 为什么 Kafka 在消费者端采用了拉（PULL）模型？注意和 PUSH 模型做对比。最好是能够举一个适用 PUSH 的例子
- 分区过多会引起什么问题？又是一个违背直觉的问题，核心在于顺序写
- 如何确定合适的分区数量？
- 如何解决 Topic 的分区数量过多的问题？
- 如何保证消息有序性？方案有什么缺点？抓住核心，相关的消息要确保发送到同一个分区，例如 ID 为1的永远发到分区1
- Kafka能不能重复消费？当然可以。但是要强调，一般的消费者都要考虑幂等的问题
- 如何保证消息消费的幂等性？就是去重，简单就是数据库唯一索引，高级就是布隆过滤器 + 唯一索引
- 如何保证只发送（或者只消费）一次？属实没必要，做好消费幂等简单多了
- Rebalance 发生时机，rebalance 过程，rebalance 有啥影响？如何避免 rebalance？核心把 rebalance 的过程背下来
- 消息积压怎么办？没啥好办法，也就是加快消费，合并消息

Q & A

THANKS