

# 导航

## Redis

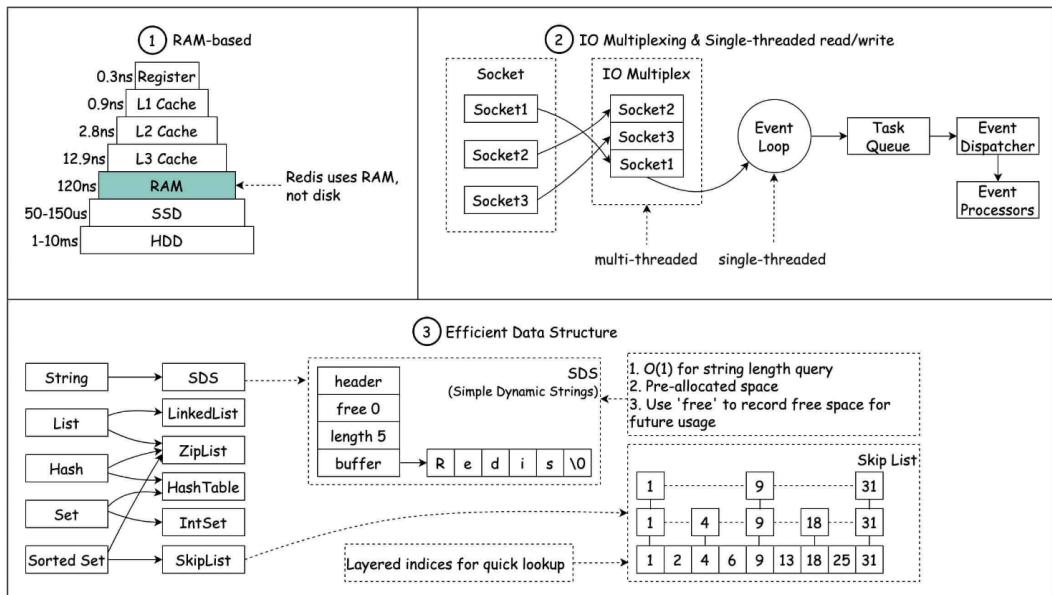
标签 C 端查询 redis 平均在 0.8ms 【Hbase 平均在 1ms】



## Redis 为什么这么快？

Redis 内部做了非常多的性能优化，比较重要的有下面 3 点：

1. Redis 基于内存，内存的访问速度是磁盘的上千倍；
2. Redis 基于 Reactor 模式设计开发了一套高效的事件处理模型，主要是单线程事件循环和 IO 多路复用 (Redis 线程模式后面会详细介绍到)；
3. Redis 内置了多种优化过后的数据类型 / 结构实现，性能非常高。



## Redis 5 种基本数据类型&底层结构实现

博客：<https://blog.csdn.net/a745233700/article/details/113449889>

数据类型	说明
String	一种二进制安全的数据类型，可以用来存储任何类型的数据比如字符串、整数、浮点数、图片（图片的 base64 编码或者解码或者图片的路径）、序列化后的对象。
List	Redis 的 List 的实现为一个双向链表，即可以支持反向查找和遍历，更方便操作，不过带来了部分额外的内存开销。
Hash	一个 String 类型的 field-value（键值对）的映射表，特别适合用于存储对象，后续操作的时候，你可以直接修改这个对象中的某些字段的值。
Set	无序集合，集合中的元素没有先后顺序但都唯一，有点类似于 Java 中的 <code>HashSet</code> 。
Zset	和 Set 相比，Sorted Set 增加了一个权重参数 <code>score</code> ，使得集合中的元素能够按 <code>score</code> 进行有序排列，还可以通过 <code>score</code> 的范围来获取元素的列表。有点像是 Java 中 <code>HashMap</code> 和 <code>TreeSet</code> 的结合体。

## RedisObject【所有数据结构对外的统一封装】

Redis 内部所有存储的数据都使用 `redisObject` 来封装。`redisObject` 是 Redis 对象系统的核心，它包含了类型、编码、引用计数、内存回收等元数据，以及指向实际数据的指针。

Redis 的五种基本数据结构：String（字符串）、List（列表）、Set（集合）、Hash（散列）、Zset（有序集合），都是通过 `redisObject` 进行封装的。每种数据类型在 `redisObject` 中都有对应的 `type` 值来表示。此外，`redisObject` 的 `encoding` 字段记录了每种值对象在底层的结构实现编码，例如 hash 在数据量较少时可能使用 `ziplist` 编码，而在数据量较大时则使用 `dict` 编码。

因此，可以说 `redisObject` 是 Redis 实现其数据结构的关键部分，它提供了一种统一的方式来处理不同类型的数据，并提供了丰富的元数据和操作接口，使得 Redis 能够高效地管理和操作数据。

## String

### 底层实现方式：动态字符串 SDS 或者 Long

SDS 优点：动态扩容，小于 1M，每次扩容为当前字符串大小的 2 倍，大于 1M，每次扩容 1M。

SDS 缺点：内存重分配频繁，在元素频繁增减时可能有性能问题。以及固定的空间预分配，可能会造成一定的内存浪费。

1、当你保存数据为整数时，String 会使用 8 字节的 Long 类型方式来保存，这种保存方式通常也叫作 int 编码方式。

2、当你保存数据包含字符时，String 类型就会用简单动态字符串（Simple Dynamic String，SDS）结构体来保存

## Hash

### 底层实现方式：压缩列表 ziplist 或者 哈希表

当数据量较少的情况下，hash 底层会使用压缩列表 `ziplist` 进行存储数据，但是受限于 `hash-max-ziplist-entries` 和 `hash-max-ziplist-value` 这两个配置项，超过了就会转成哈希表。

### 【为什么会转换成哈希表？因为当 ziplist 很大会造成如下问题】

- 涉及内存重新分配和拷贝操作导致降低性能：因为 `ziplist` 的长度是固定的，无法动态扩展。每次添加或修改数据时，都涉及内存的 `realloc` 操作【内存重新分配和拷贝】，特别是当 `ziplist` 长度很长的时候，一次 `realloc` 可能会导致大批量的数据拷贝，进一步降低性能。

- 当ziplist数据项过多的时候，在它上面查找指定的数据项就会性能变得很低，因为ziplist上的查找需要进行遍历。

## List

### Redis3.2之前的底层实现方式：压缩列表ziplist或者双向循环链表linkedlist

当list存储的数据量较少时，会使用ziplist存储数据，也就是同时满足下面两个条件：

- 列表中数据个数少于512个
- list中保存的每个元素的长度小于64字节

当不能同时满足上面两个条件的时候，list就通过双向循环链表linkedlist来实现了

### Redis3.2及之后的底层实现方式：quicklist

Redis的quicklist是Redis 3.2版本以后针对链表和压缩列表进行改造的一种数据结构，它是zipList和linkedList的混合体，由多个节点(Node)组成的双向链表，每个节点都是一个ziplist(压缩列表)。这种混合结构结合了ziplist的内存高效性和链表的快速插入删除特性，使得quicklist既能高效地处理大量数据，又能保持较好的内存利用率。

## Set

### 底层实现方式：有序整数集合intset或者哈希表

当存储的数据同时满足下面这样两个条件的时候，Redis就采用整数集合intset来实现set这种数据类型：

- 存储的数据都是整数
- 存储的数据元素个数小于512个

当不能同时满足这两个条件的时候，Redis就使用哈希表来存储集合中的数据

## Sorted Set

解释：Sorted set相比set多了一个权重参数score，集合中的元素能够按score进行排列。可以做排行榜应用，取TOP N操作。另外，sorted set可以用来做延时任务。最后一个应用就是可以做范围查找。

### 底层实现方式：压缩列表ziplist或者zset

- 当数据比较少时，即元素个数要小于128个&&集合中每个数据大小都小于64字节，用ziplist存储，ziplist存储了多个【元素和对应分数】
- 数据较多时，用dict+skiplist两种存储结构进行存储，

Dict：key为元素，value为分数，所以根据一个key能快速定位到他的分数。

Skiplist：按照分数有序的从小到大保存所有的集合元素，每个元素包含元素和分数，在按照分数范围取值的时候，平均时间复杂度是O(logN)，最坏是O(N)。

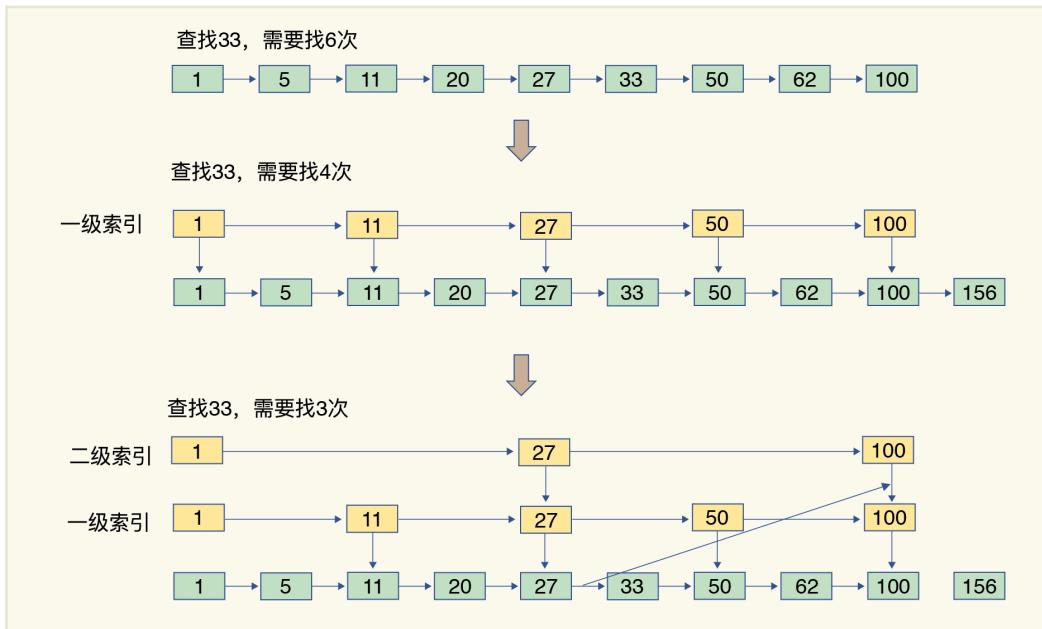
<https://www.cnblogs.com/xuwc/p/14016461.html>

## 底层实现之跳表

### (1) 什么是跳表：

跳表是一种可以进行二分查找的有序链表，采用空间换时间的设计思路，跳表在原有的有序链表上面增加了多级索引(例如每两个节点就提取一个节点到上一级)，通过索引来实现快速查找。跳表是一种动态数据结构，支持快速的插入、删除、查找操作，时间复杂度都为O(logn)，空间复杂度为O(n)。跳表非常灵活，可以通过改变索引构建策略，有效平衡执行效率和内存消耗。

- ① 跳表的删除操作：除了要删除原始链表中的节点，还要删除索引中的节点。
- ② 插入元素后，索引的动态更新：不停的往跳表里面插入数据，如果不更新索引，就有可能出现某两个索引节点之间的数据非常多的情况，甚至退化成单链表。针对这种情况，我们在添加元素的时候，通过一个随机函数，同时选择将这个数据插入到部分索引层。比如随机函数生成了值 K，那我们就将这个结点添加到第一级到第 K 级这 K 级的索引中



如果我们要在链表中查找 33 这个元素，只能从头开始遍历链表，查找 6 次，直到找到 33 为止。此时，复杂度是  $O(N)$ ，查找效率很低。

为了提高查找速度，我们来增加一级索引：从第一个元素开始，每两个元素选一个出来作为索引。这些索引再通过指针指向原始的链表。例如，从前两个元素中抽取元素 1 作为一级索引，从第三、四个元素中抽取元素 11 作为一级索引。此时，我们只需要 4 次查找就能定位到元素 33 了。

如果我们还想再快，可以再增加二级索引：从一级索引中，再抽取部分元素作为二级索引。例如，从一级索引中抽取 1、27、100 作为二级索引，二级索引指向一级索引。这样，我们只需要 3 次查找，就能定位到元素 33 了。

可以看到，这个查找过程就是在多级索引上跳来跳去，最后定位到元素。这也正好符合“跳”表的叫法。当数据量很大时，跳表的查找复杂度就是  $O(\log N)$ 。

PS:索引动态生成规则==>如下图，每次插入数据时，根据概率 P 百分之 50 概率，来决定是不是要上一层索引插入这个值，直到概率为 false 或者达到设定的最高多级索引层数。

## 一、新增索引层的核心机制

### 1. 概率决定层级（核心原理）

跳表通过随机函数（如抛硬币）决定新节点是否晋升到更高层索引：

- **晋升规则：**新节点插入后，从第0层开始，以概率  $p$ （通常  $p = 0.5$ ）向上晋升：
  - 若随机结果为“成功”，则在当前层生成索引节点，并继续尝试向更高层晋升；
  - 若失败，则停止晋升。
- **示例：**若连续成功3次，则新节点会出现在第0、1、2、3层索引中。

### 2. 最高层动态扩展

当新节点的晋升层数 超过当前跳表最高层 时，需要新增索引层：

- 例如：当前最高层为3层，若新节点晋升到第4层，则需创建第4层索引（原最高层+1）；
- 此时，跳表新增一个仅包含头节点和尾节点的空层，新节点作为该层首个数据节点<sup>3 5</sup>。

## 二、新增索引层的具体触发条件

场景	触发条件	操作
新节点晋升超过当前最高层	随机晋升后的层数 $k >$ 当前最大层数 $\text{maxLevel}$	创建第 $k$ 层索引，更新 $\text{maxLevel} = k$
最高层索引密度过低	最高层节点数过少（如仅剩头节点），导致高层索引失去意义	删除空层，降低 $\text{maxLevel}$ （非插入时触发，但属于层级 <sup>5</sup> ）
跳表初始化	首次插入数据时	创建第0层索引（基础层）

## 三、设计考量与优化

### 1. 概率参数 $p$ 的影响

- $p = 0.5$  时，每层索引节点数约为下一层的一半，保证查询复杂度为  $O(\log n)$ ；
- 若  $p$  过高，索引层过密，浪费空间；若  $p$  过低，索引稀疏，查询效率下降。

### 2. 空间与时间的权衡

- 新增索引层：提升查询速度，但增加内存占用；
- 删除空层：减少内存开销，但可能因频繁删增导致性能波动<sup>5</sup>。

### 3. 最大层级限制

实际实现中会设置  $\text{maxLevel}$  上限（如32层），避免极端情况下层级过深<sup>4</sup>。

# Redis的缓存失效策略&内存淘汰策略

## Redis的缓存失效策略

### 定时删除

- 当设置键值对时，可以为其指定一个过期时间。一旦到达这个时间点，Redis会自动删除相应的键。
- 此策略适用于对数据时效性有严格要求的应用，如会话管理、临时验证码存储等。
- 在高吞吐量环境下，频繁的定时检查和删除操作可能导致性能瓶颈。

### 惰性删除

- 在每次访问键时，Redis会检查该键是否已过期。如果过期，则删除并返回空

值。

- 该策略减少了主动检查的开销，适用于非即时敏感的过期需求或大规模数据集。
- 但可能导致过期数据暂时占用内存。

#### 定期删除

- Redis 会周期性地检查并删除一部分已过期的键。
- 这种策略结合了定时删除的及时性和惰性删除的低开销特点。
- 适用于大多数常规场景，有效避免了内存中过期数据的累积，同时保持了系统的高效运行。

## Redis 的内存淘汰策略

- LRU (The Least Recently Used) 是一种基于数据访问时间的缓存淘汰策略。它认为最近被访问的数据在未来被再次访问的可能性更高，因此，当缓存满了需要移除数据时，LRU 会选择最久未使用的数据进行移除。
  - LFU (Least frequently used) 是一种基于数据访问频率的缓存淘汰策略。它认为访问频率较低的数据在未来被再次访问的可能性也较低，因此，当缓存满了需要移除数据时，LFU 会选择访问频率最低的数据进行移除。
- 

## String 还是 Hash 存储对象数据更好呢？

- String 存储的是序列化后的对象数据，存放的是整个对象。Hash 是对对象的每个字段单独存储，可以获取部分字段的信息，也可以修改或者添加部分字段，节省网络流量。如果对象中某些字段需要经常变动或者经常需要单独查询对象中的个别字段信息，Hash 就非常适合。
  - String 存储相对来说更加节省内存，缓存相同数量的对象数据，String 消耗的内存约是 Hash 的一半。并且，存储具有多层嵌套的对象时也方便很多。如果系统对性能和资源消耗非常敏感的话，String 就非常适合。【hash 结构，每条数据都会带一个 dictEntry 的结构体，dictEntry 结构中有三个 8 字节的指针，分别指向 key、value 以及下一个 dictEntry，三个指针共 24 字节，以及 jemalloc 在分配内存时会多分配一点】
- 

## Redis 持久化机制之 AOF&RDB

Redis 的持久化主要有两大机制：AOF (Append Only File) 日志和 RDB 快照

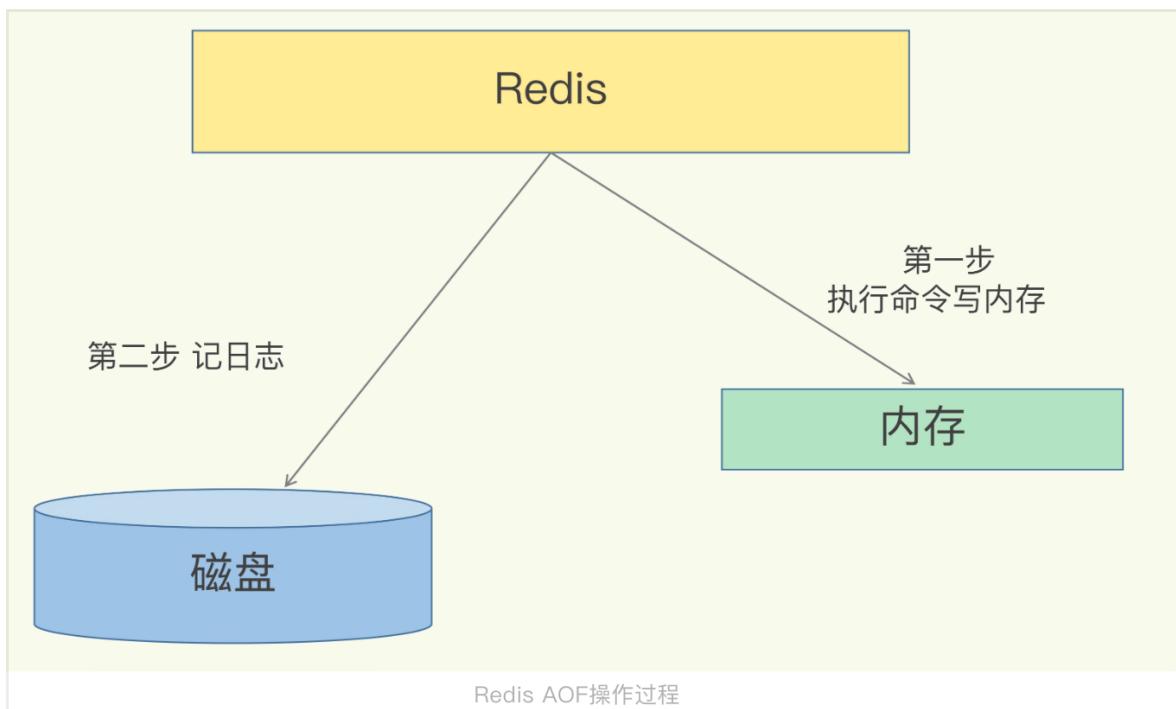
### WAL

redo log buffer 叫做重做日志缓冲，最大的作用就是可以实现崩溃恢复，这是因为在 InnoDB 中，在持久化数据之前，即提交事务的时候，会先将数据的物理修改内容写入 redo log 日志中，然后再在适当的时间点将脏页数据写入磁盘的表空间中。这种策略称为 WAL (Write Ahead Log) 预写日志策略。这样一旦内存崩溃了，MySQL 重启之后，也可以从 redo log 中重新加载回事务的数据到内存中，实现了数据的持久化，保证了数据的可靠性。WAL 里所谓的写前日志，应该是相对落盘而

言的吧？而不是相对于内存而言。Mysql 一样也是先写内存里的

### mysql和redis日志顺序区别

说到日志，我们比较熟悉的是数据库的写前日志（Write Ahead Log, WAL），也就是说，在实际写数据前，先把修改的数据记到日志文件中，以便故障时进行恢复。不过，AOF 日志正好相反，它是写后日志，“写后”的意思是 Redis 是先执行命令，把数据写入内存，然后才记录日志，如下图所示：



传统数据库的日志，例如 redo log（重做日志），记录的是修改后的数据，而 AOF 里记录的是 Redis 收到的每一条命令，这些命令是以文本形式保存的。

### AOF写后日志的好处（保证记录正确+不会阻塞当前写操作）

为了避免额外的检查开销，Redis 在向 AOF 里面记录日志的时候，并不会先去对这些命令进行语法检查。所以，如果先记日志再执行命令的话，日志中就有可能记录了错误的命令，Redis 在使用日志恢复数据时，就可能会出错。

而写后日志这种方式，就是先让系统执行命令，只有命令能执行成功，才会被记录到日志中，否则，系统就会直接向客户端报错。所以，Redis 使用写后日志这一方式的一大好处是，可以避免出现记录错误命令的情况。

### AOF写后日志的潜在风险

1. 刚执行完命令如果宕机就容易丢失数据（如果用做缓存可从数据库恢复数据，如果用做数据库一旦宕机因为日志丢失了就无法恢复数据则会造成数据丢失）

2. **AOF 日志由主线程中执行**，如果在把日志文件写入磁盘时，磁盘写压力大，就会导致写盘很慢，进而导致后续的操作也无法执行了。

### AOF写回日志到磁盘到三种策略

AOF 机制给我们提供了三个选择，也就是 AOF 配置项 appendfsync 的三个可选值。

1. Always, 同步写回：每个写命令执行完，立马同步地将日志写回磁盘；（缺点：“同步写回”可以做到基本不丢数据，但是它在每一个写命令后都有一个慢速的

落盘操作，不可避免地会影响主线程性能；）

2. Everysec，每秒写回：每个写命令执行完，只是先把日志写到 AOF 文件的内存缓冲区，每隔一秒把缓冲区中的内容写入磁盘；（缺点：虽然“操作系统控制的写回”在写完缓冲区后，就可以继续执行后续的命令，但是落盘的时机已经不在 Redis 手中了，只要 AOF 记录没有写回磁盘，一旦宕机对应的数据就丢失了；“每秒写回”采用一秒写回一次的频率，避免了“同步写回”的性能开销，虽然减少了对系统性能的影响，但是如果发生宕机，上一秒内未落盘的命令操作仍然会丢失。所以，这只能算是，在避免影响主线程性能和避免数据丢失两者间取了个折中。）
3. No，操作系统控制的写回：每个写命令执行完，只是先把日志写到 AOF 文件的内存缓冲区，由操作系统决定何时将缓冲区内容写回磁盘。

配置项	写回时机	优点	缺点
Always	同步写回	可靠性高，数据基本不丢失	每个写命令都要落盘，性能影响较大
Everysec	每秒写回	性能适中	宕机时丢失1秒内的数据
No	操作系统控制的写回	性能好	宕机时丢失数据较多

总结一下就是：想要获得高性能，就选择 No 策略；如果想要得到高可靠性保证，就选择 Always 策略；如果允许数据有一点丢失，又希望性能别受太大影响的话，那么就选择 Everysec 策略。

## AOF 重写机制

针对同一个 key 多次操作的日志只取当前最新的日志，然后将所有 key 都按照这样的形式汇集成一条日志命令，这样会大大缩小日志文件大小。

### AOF 重写会阻塞吗？

和 AOF 日志由主线程写回不同，**重写过程是由后台子进程 bgrewriteaof 来完成的**，这也是为了避免阻塞主线程，导致数据库性能下降。并且重写可以降低磁盘空间大小缓解磁盘压力，提高 AOF 写后日志的主线程写入速度，降低主线程阻塞可能性。

### 重写过程：一个拷贝，两处日志

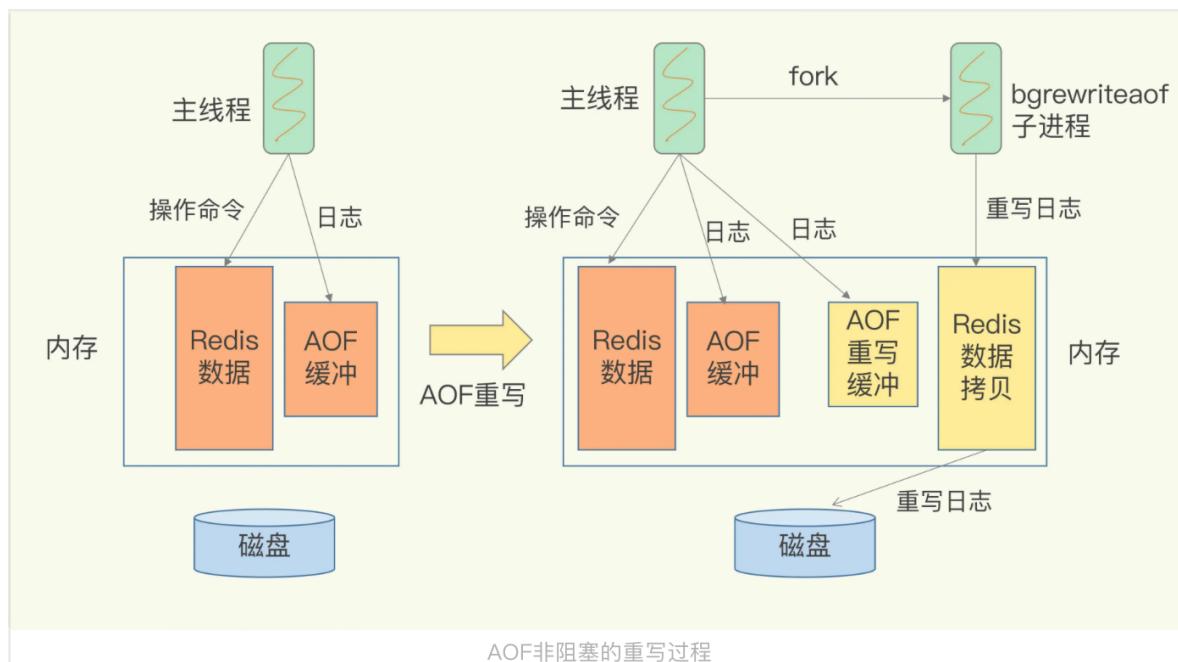
#### AOF 重写机制原理

根据 Redis 进程内的数据生成一个新的 AOF 文件，只包含当前有效和存在的数据的写入命令，而不是历史上所有的写入命令。AOF 重写机制是通过 fork 出一个子进程来完成的，子进程会扫描 Redis 的数据库，并将每个键值对转换为相应的写入命令，**然后写入到一个临时文件中**。在子进程进行 AOF 重写的过程中，主进程还会继续接收和处理客户端的请求，如果有新的写操作发生，主进程会将这些写操作追加到一个缓冲区中，并通过管道通知子进程【**即新的写操作，主进程会同时向 AOF 缓冲和 AOF 重写缓冲写入数据**】。**子进程在完成 AOF 重写后，会将缓冲区中的写操作也追加到临时文件中**，然后向主进程发送信号，通知主进程可以切换到新的 AOF 文件了。主进程在收到子进程的信号后，会用临时文件替换旧的 AOF 文件，并关闭旧的 AOF 文件【**AOF 缓冲区有两个，一个在老 AOF 主线程，一个在子线程重写日志里，都是用于接收新写入的命令**】。

拷贝：每次执行重写时，主线程 fork 出后台的 bgrewriteaof 子进程，并且会拷贝一份最新数据给子进程记入重写日志

日志一：主线程未阻塞还在处理新的请求，当有新的请求进来时，会把新的数据放入到 AOF 缓冲区

日志二：新请求的操作也会放入到 AOF 重写缓冲



AOF 缓冲 (AOF Buffer) 和 AOF 重写缓冲 (AOF Rewrite Buffer) 在 Redis 的 AOF 持久化机制中各自扮演着不同的角色。

### 1. AOF 缓冲 (AOF Buffer):

- 它是 AOF 持久化策略中的一个关键组件，负责暂存待写入 AOF 文件的命令。
- 当 Redis 执行写操作时，这些写命令首先会被追加到 AOF 缓冲区中，而不是直接写入磁盘。
- AOF 缓冲区的内容会根据特定的写回策略（如 always、everysec、no）被定期或同步地写入到 AOF 文件中。
- 它的主要目的是提高性能，通过减少磁盘 I/O 操作次数来降低写操作的延迟。

### 2. AOF 重写缓冲 (AOF Rewrite Buffer):

- AOF 重写是 Redis 为了减小 AOF 文件大小和提高恢复速度而进行的一个过程。
- 当 AOF 重写触发时（例如，AOF 文件大小达到一定阈值），Redis 会创建一个子进程来负责重写 AOF 文件。
- 在 AOF 重写期间，所有新的写命令不会直接写入正在被重写的 AOF 文件，而是被追加到 AOF 重写缓冲区中。
- AOF 重写缓冲区确保了在重写过程中，新的写命令不会丢失，并在重写完成后被追加到新的 AOF 文件中。

总结：

- 1、AOF 缓冲区用于提高性能，通过减少磁盘 I/O 操作次数来降低写操作的延迟。
- 2、AOF 重写缓冲区是在 AOF 重写过程中用于暂存新的写命令的缓冲区，以确保数据的一致性和完整性。

## Redis 持久化机制之 RDB

### RDB 为什么恢复数据那么快？

和 AOF (AOF 恢复数据的话需要顺序、逐一重新执行操作命令带来的低效性能问题) 相比，RDB 记录的是某一时刻的数据，并不是操作，所以，在做数据恢复时，我们可以直接把 RDB 文件读入内存，很快地完成恢复【快照是二进制文件，通常较小，恢复速度快】。

### RDB 缺点

- 1、容易丢数据 (生成 RDB 期间 redis 宕机)
- 2、**数据很多时，主线程 fork () 需要 copy 全量当前数据，容易对主线程造成阻塞压力【生成 RDB 文件是由 bgsave 异步执行的】**

PS: fork 为什么会阻塞主线程？

### 多线程进程中的 fork，为线程分配的内存

多线程进程中的 fork 是指在多线程程序中使用 fork 系统调用创建子进程的操作。fork 系统调用会创建一个与父进程完全相同的子进程，包括代码、数据、堆栈等。但是，由于多线程程序中存在多个线程共享同一进程的资源，因此在使用 fork 创建子进程时需要特别注意线程的内存分配。

在多线程程序中，每个线程都有自己的栈空间，用于存储局部变量和函数调用的上下文信息。当使用 fork 创建子进程时，子进程会复制父进程的整个地址空间，包括所有线程的栈空间。这意味着子进程会继承父进程中所有线程的栈空间，包括栈中的数据和状态。

由于多线程程序中存在多个线程共享同一进程的堆空间和全局变量，因此在使用 fork 创建子进程时，子进程会继承父进程中的堆空间和全局变量。这意味着子进程可以访问和修改父进程中的堆空间和全局变量。

需要注意的是，由于子进程继承了父进程中所有线程的栈空间、堆空间和全局变量，因此在子进程中对这些资源的修改可能会影响到其他线程和进程。为了避免潜在的问题，通常在使用 fork 创建子进程后，子进程会立即调用 exec 函数族中的某个函数来加载新的程序，以替换掉原有的代码和数据。

总结起来，多线程进程中的 fork 会复制父进程的整个地址空间，包括所有线程的栈空间、堆空间和全局变量。在使用 fork 创建子进程后，需要注意子进程对这些资源的修改可能会影响到其他线程和进程，因此通常会立即调用 exec 函数族中的某个函数来加载新的程序。

3、RDB 针对主线程后面进来的修改 / 写操作进行 copy-on-write 时也会占用大量的资源 (需要存储额外的存储元数据信息 8 字节左右，因为 cow 时主线程会创建副本来自同步后面进来的写操作数据)

### RDB 的写时复制 COW 的底层实现机制

Redis 在使用 RDB 方式进行持久化时，会用到写时复制机制。我在第 5 节课讲写时复制的时候，着重介绍了写时复制的效果：bgsave 子进程相当于复制了原始数据，而主线程仍然可以修改原来的数据。

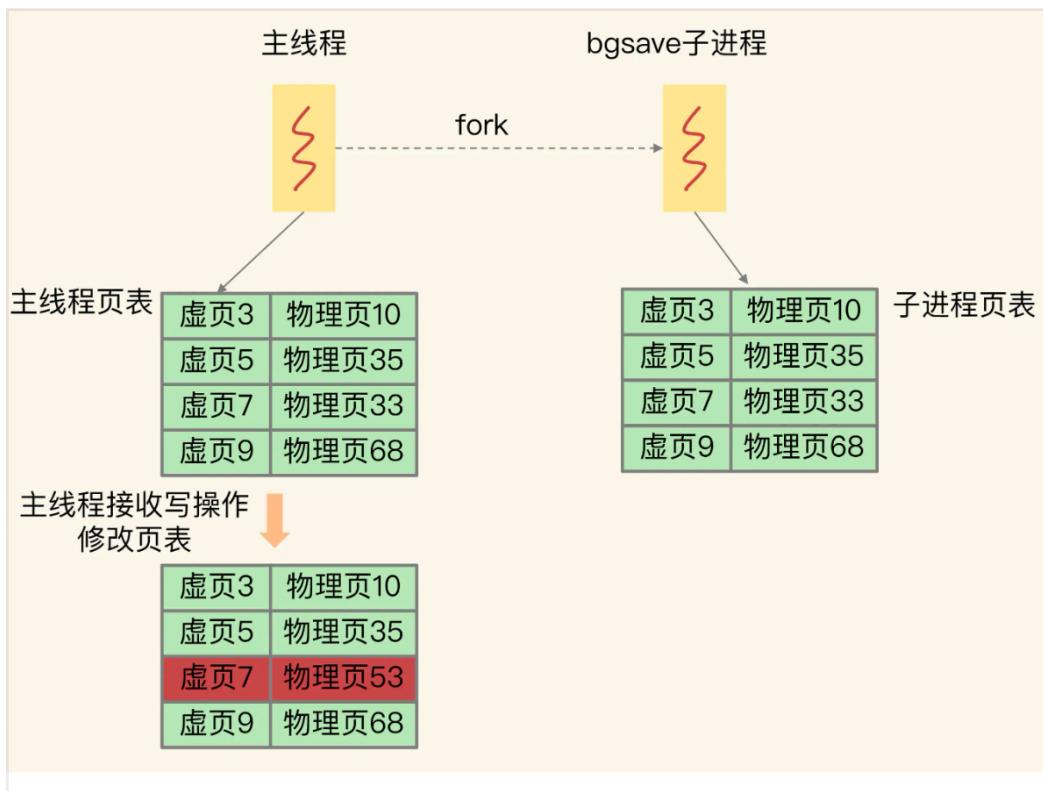
今天，我再具体讲一讲写时复制的底层实现机制。

对 Redis 来说，主线程 fork 出 bgsave 子进程后，bgsave 子进程实际是复制了主线程的页表。这些页表中，就保存了在执行 bgsave 命令时，主线程的所有数据块在内存中的物理地址。这样一来，bgsave 子进程生成 RDB 时，就可以根据页表读取这些数据，再写入磁盘中。如果此时，主线程接收到新写或修改操作，那么，主线程会使用写时复制机制。具体来说，写时复制就是指，主线程在有写操作时，才会把这个新写或修改后的数据写入到一个新的物理地址中，并修改自己的页表映射。

我来借助下图中的例子，具体展示一下写时复制的底层机制。

bgsave 子进程复制主线程的页表以后，假如主线程需要修改虚页 7 里的数据，那么，主线程就需要新分配一个物理页（假设是物理页 53），然后把修改后的虚页 7 里的数据写到物理页 53 上，而虚页 7 里原来的数据仍然保存在物理页 33 上。这个时候，虚页 7 到物理页 33 的映射关系，仍然保留在 bgsave 子进程中。所以，bgsave 子进程可以无误地把虚页 7 的原始数据写入 RDB 文件。

\*\*简单来说，就是在写时复制的过程中，因为 bgsave 想要复制当前物理页的数据，所以物理页的数据不会变动；那么新的修改会落在一个新分配的物理页上，等到 bgsave 结束后再将变动数据保存回原来的物理页上；



## 快速恢复：RDB+AOF

### 什么是混合持久化？

重启 Redis 时，我们很少使用 RDB 来恢复内存状态，因为会丢失大量数据。我们通常使用 AOF 日志重放，但是重放 AOF 日志性能相对 RDB 来说要慢很多，这样在 Redis 实例很大的情况下，启动需要花费很长的时间。Redis 4.0 为了解决这个问题，带来了一个新的持久化选项——混合持久化。

如果开启了混合持久化，AOF 在重写时，不再是单纯将内存数据转换为 RESP 命令写入 AOF 文件，而是将重写这一刻之前的内存做 RDB 快照处理，并且将 RDB 快照内容和增量的 AOF 修改内存数据的命令存在一起，都写入新的 AOF 文件，新的文件一开始不叫 `appendonly.aof`，等到重写完新的 AOF 文件才会进行改名，覆盖原有的 AOF 文件，完成新旧两个 AOF 文件的替换。

于是在 Redis 重启的时候，可以先加载 RDB 的内容，然后再重放增量 AOF 日志就可以完全替代之前的 AOF 全量文件重放，因此重启效率大幅得到提升。



## 开启配置

aof-use-rdb-preamble 修改为 yes【redis 5.0 默认开启】

### 存储数据方式

混合持久化也是 AOF 文件，例如在发生 AOF 重写时，通过存量数据 RDB+ 增量数据 AOF 的结合方式存储在 AOF 文件中。

### Redis 崩溃重启恢复数据

如果 AOF 和 RDB 都开启了，优先走 AOF 恢复【混合持久化是 Redis 自动处理的过程，无需人工干预，且 redis 5.0 后默认开启，所以 AOF 一般来说就含有 RDB+AOF】。

---

## Redis 的渐进式 Rehash

### 压缩列表

压缩列表的设计不是为了查询的，而是为了减少内存的使用和内存的碎片化。比如一个列表中的只保存 int，结构上还需要两个额外的指针 prev 和 next，每添加一个结点都这样。而压缩列表是将这些数据集合起来只需要一个 prev 和 next【本质上存储的是一个数组，内存空间连续，减少内存碎片化】。

### 压缩列表可以节省内存

压缩列表：Redis 底层数据结构，是一种非常节省内存的结构。

#### 压缩列表的数据结构：

1. 表头三个字段 zlbytes【列表长度】、zltail【列表尾的偏移量】、zllen【列表中的 entry 个数】。
2. 列表中的 entry【压缩列表之所以能节省内存，就在于它是用一系列连续的 entry 保存数据】。
3. 表尾一个字段 zlend【列表结束】。

#### entry 的元数据组成：

1. prev\_len：表示前一个 entry 的长度。（1）取值1字节时：表示上一个 entry 的长

度小于 254 字节。(2) 取值 5 字节时：表示上一个 entry 长度大于等于 254 字节。

2. len：自身长度，4 字节。

3. encoding：编码方式，1 字节。

4. content：实际数据。

Redis 基于压缩列表实现了 List、Hash、Sorted Set 这样的集合类型，最大好处就是节省了 dicEntry 的开销。当使用 String 类型时，一个键值对就有一个 dicEntry，要用 32 字节空间。但采用集合类型时，一个 key 就对应一个集合的数据，能保存的数据多了很多，但也只用了一个 dicEntry【避免指针开销】，这样就节省了内存。

但是如果压缩列表超过如下两个配置的任一阈值就会转成哈希表保存（一旦转了就不会转回压缩列表），那么就没那么高效了【哈希列表会有多个 dicEntry】

hash-max-ziplist-entries：表示用压缩列表保存时哈希集合中的最大元素个数。

hash-max-ziplist-value：表示用压缩列表保存时哈希集合中单个元素的最大长度。

\*\*【使用 Hash、List、Sorted Set 等集合类型写入，没超过阈值底层会默认使用压缩列表存储，否则就会采用哈希表存储】

压缩列表实际上类似于一个数组，数组中的每一个元素都对应保存一个数据。和数组不同的是，压缩列表在表头有三个字段 zlbytes、zltail 和 zllen，分别表示列表长度、列表尾的偏移量和列表中的 entry 个数；压缩列表在表尾还有一个 zlend，表示列表结束。

## 压缩列表的查找

zlbytes	zltail	zllen	entry1	entry2	...	entryN	zlend
---------	--------	-------	--------	--------	-----	--------	-------

在压缩列表中，如果我们要查找定位第一个元素和最后一个元素，可以通过表头三个字段的长度直接定位，复杂度是 O(1)。而查找其他元素时，就没有这么高效了，只能逐个查找，此时的复杂度就是 O(N) 了。

**PS：之所以这里讲压缩列表 ziplist，是因为下面的讲“什么时候触发 rehash”的编码转换概念**

Redis 什么时候做 rehash?

- 容易触发 rehash：hash-max-ziplist-entries 和 hash-max-ziplist-value 这两个参数控制哈希表单个元素的内部 ziplist 编码，一旦超过 Redis 会将该 ziplist 节点内部的所有元素转换为哈希表（hash table）节点，而不再使用 ziplist 进行存储，这个过程被称为编码转换，原本在 ziplist 中的所有元素都会被迁移到新的哈希表中，每个元素都会成为一个独立的哈希表节点。这样，哈希表就能够容纳更多的元素和更大的值，同时也提高了查找和插入操作的效率，因为哈希表的查找和插入时间复杂度通常是 O(1)，而 ziplist 的查找和插入时间复杂度与元素数量成正比（时间复杂度为 O(n)）。因此频繁发生编码转换，使得哈希表元素变多，容易触发 rehash。默认情况下，hash-max-ziplist-entries 的默认值是 512，而 hash-max-ziplist-value 的默认值是 64。
- Redis 会使用装载因子（load factor）来判断是否需要做 rehash。装载因子的计算方式是，哈希表中所有 entry 的个数除以哈希桶个数。Redis 会根据装载因子的两种情况，来触发 rehash 操作：

情况一：装载因子 $\geq 1$ ，同时，哈希表被允许进行 rehash；

情况二：装载因子 $\geq 5$ 。

在第一种情况下，如果装载因子等于 1，同时我们假设，所有键值对是平均分布在哈希表的各个桶中的，那么，此时，哈希表可以不用链式哈希，因为一个哈希桶正好保存了一个键值对。但是，如果此时再有新的数据写入，哈希表就要使用链式哈希了，这会对查询性能产生影响。在进行 RDB 生成和 AOF 重写时，哈希表的 rehash 是被禁止的，这是为了避免对 RDB 和 AOF 重写造成影响。如果此时，Redis 没有在生成 RDB 和重写 AOF，那么，就可以进行 rehash。否则的话，再有数据写入时，哈希表就要开始使用查询较慢的链式哈希了。在第二种情况下，也就是装载因子大于等于 5 时，就表明当前保存的数据量已经远远大于哈希桶的个数，哈希桶里会有大量的链式哈希存在，性能会受到严重影响，此时，就立马开始做 rehash【前提是还没有生成 AOF 和 RDB】。刚刚说的是触发 rehash 的情况，如果装载因子小于 1，或者装载因子大于 1 但是小于 5，同时哈希表暂时不被允许进行 rehash（例如，实例正在生成 RDB 或者重写 AOF），此时，哈希表是不会进行 rehash 操作的。

### 高效 rehash->渐进式 rehash

Redis 默认使用了两个全局哈希表：哈希表 1 和哈希表 2。一开始，当你刚插入数据时，默认使用哈希表 1，此时的哈希表 2 并没有被分配空间。随着数据逐步增多，Redis 开始执行 rehash，这个过程分为三步：

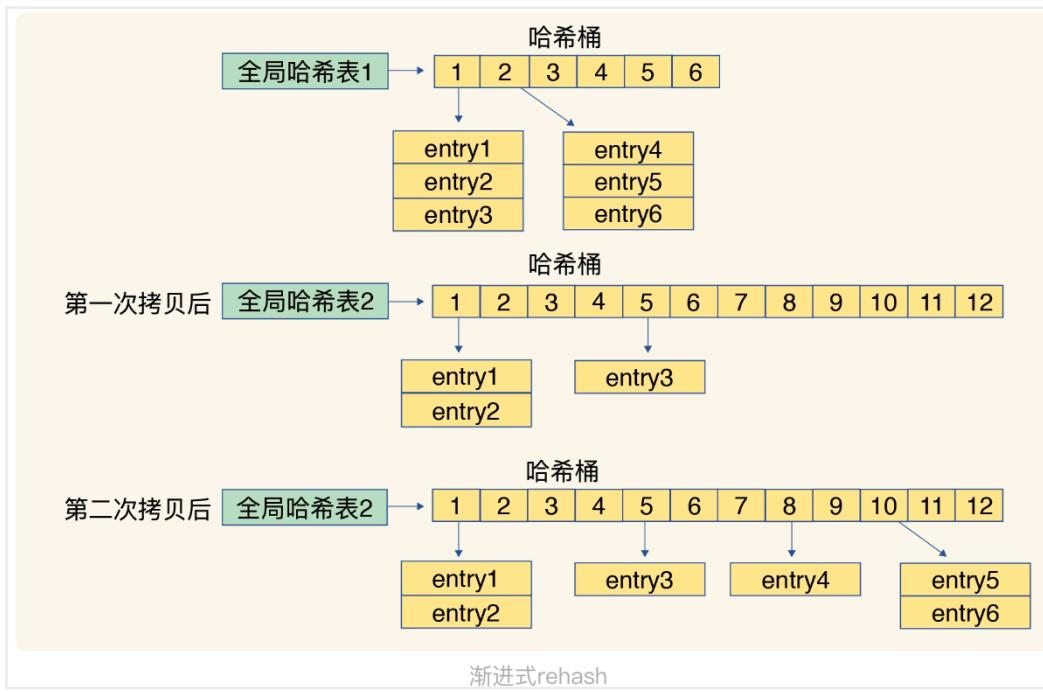
- 1、给哈希表 2 分配更大的空间，例如是当前哈希表 1 大小的两倍；
- 2、把哈希表 1 中的数据重新映射并拷贝到哈希表 2 中；
- 3、释放哈希表 1 的空间。

到此，我们就可以从哈希表 1 切换到哈希表 2，用增大的哈希表 2 保存更多数据，而原来的哈希表 1 留作下一次 rehash 扩容备用。

这个过程看似简单，但是第二步涉及大量的数据拷贝，如果一次性把哈希表 1 中的数据都迁移完，会造成 Redis 线程阻塞，无法服务其他请求。此时，Redis 就无法快速访问数据了。

为了避免这个问题，Redis 采用了渐进式 rehash。

简单来说就是在第二步拷贝数据时，Redis 仍然正常处理客户端请求，每处理一个请求时，从哈希表 1 中的第一个索引位置开始，顺带着将这个索引位置上的所有 entries 拷贝到哈希表 2 中；等处理下一个请求时，再顺带拷贝哈希表 1 中的下一个索引位置的 entries。如下图所示：



这样就巧妙地把一次性大量拷贝的开销，分摊到了多次处理请求的过程中，避免了耗时操作，保证了数据的快速访问。

### Rehash过程中哈希表1只减不增

在进行渐进式 rehash 的过程中，会同时使用 哈希表1 和 哈希表2 两个哈希表，所以在渐进式 rehash 进行期间，字典的删除 (delete)、查找 (find)、更新 (update) 等操作会在两个哈希表上进行：比如说，要在 Redis 里面查找一个键的话，程序会先在 哈希表1 里面进行查找，如果没找到的话，就会继续到 哈希表2 里面进行查找，诸如此类。另外，在渐进式 rehash 执行期间，新添加到字典的键值对一律会被保存到 哈希表2 里面，而 哈希表1 则不再进行任何添加操作：这一措施保证了哈希表1 包含的键值对数量会只减不增，并随着 rehash 操作的执行而最终变成空表。

一次请求一个 entrys，那后续如果再也没有请求来的时候，余下的 entrys 是怎么处理的呢？

是就留在 hash1 中了还是有定时任务后台更新过去呢 答案：渐进式 rehash 执行时，除了根据键值对的操作来进行数据迁移，Redis 本身还会有一个定时任务在执行 rehash，如果没有键值对操作时，这个定时任务会周期性地（例如每 100ms 一次）搬移一些数据到新的哈希表中，这样可以缩短整个 rehash 的过程。

## Redis 线程模型

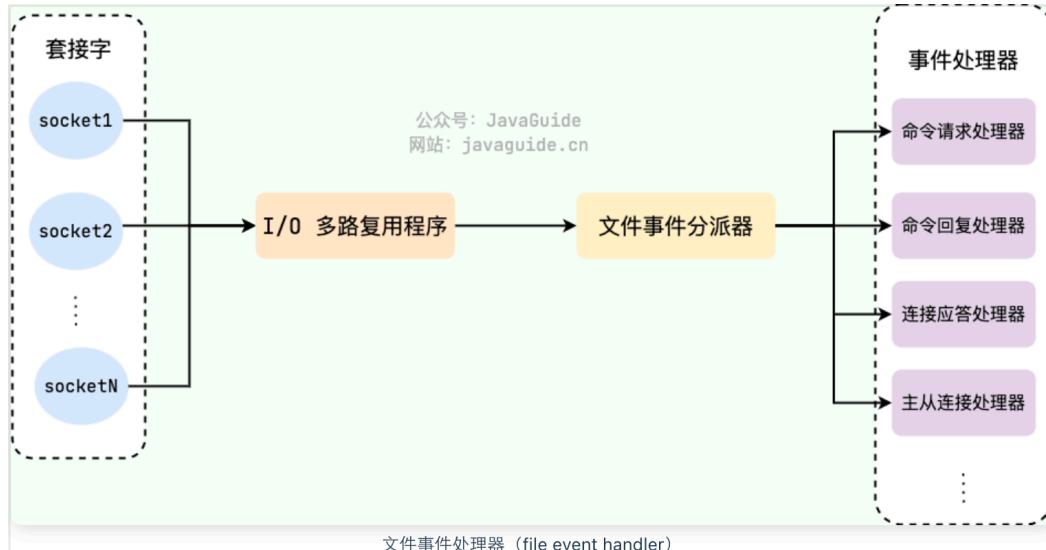
Redis 基于 Reactor 模式设计开发了一套高效的事件处理模型（Netty 的线程模型也基于 Reactor 模式，Reactor 模式不愧是高性能 IO 的基石），这套事件处理模型对应的是 Redis 中的文件事件处理器（file event handler）。由于文件事件处理器（file event handler）是单线程方式运行的，所以我们一般都说 Redis 是单线程模型。

**多路复用好处：**I/O 多路复用技术的使用让 Redis 不需要额外创建多余的线程来监

听客户端的大量连接，降低了资源的消耗（和 NIO 中的 Selector 组件很像）。

文件事件处理器（file event handler）主要是包含 4 个部分：

- 多个 socket（客户端连接）
- I/O 多路复用程序（支持多个客户端连接的关键）
- 文件事件分派器（将 socket 关联到相应的事件处理器）
- 事件处理器（连接应答处理器、命令请求处理器、命令断处理器）



## IO 多路复用

IO 多路复用支持三种模式【select、poll、epoll】

Select【同步 IO 机制】

底层使用 bitmap 标识 fd，将已连接的 Socket 都放到一个文件描述符集合【默认监听 1024 个文件描述符】，其中 select 模式下包含了针对文件描述符集合的 2 次遍历和 2 次拷贝，

- 1、拷贝 1：将文件描述符集合从用户态拷贝到内核态
- 2、遍历 1：内核态遍历产生网络事件的 Socket 并且标记可读或可写
- 3、拷贝 2：将内核态遍历的数据拷贝传回用户态
- 4、遍历 2：用户态遍历产生网络事件的 Socket 并且标记可读或可写（用户态为何还要遍历可以看下图解释）
- 5、处理数据

弊端：这种方式的 I/O 多路复用，不适合太大规模的并发量，因为随着并发量上来。性能会越来越差

PS：Redis 中确实可能存在没有网络事件产生的文件描述符。这些文件描述符可能属于已关闭或未连接的客户端连接，或者它们可能没有被注册到事件监听器中。然而，对于活跃的连接，当客户端与 Redis 进行通信时，与之关联的文件描述符会产生相应的网络事件



## Poll【同步 IO 机制】

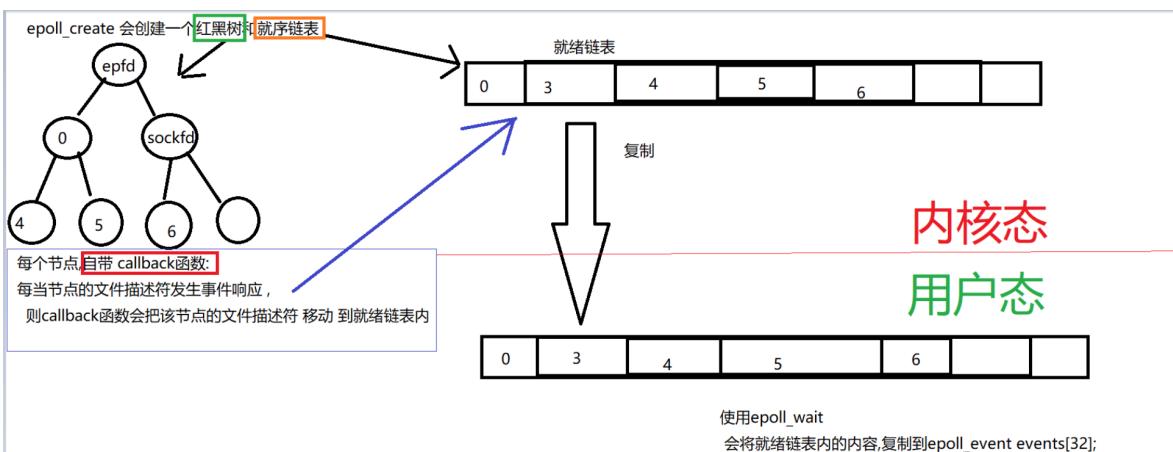
基本原理和 select 一致, 同样需要针对文件描述符在内核态和用户态之间进行 2 次遍历和 2 次拷贝, 只不过底层使用数组, 所以文件描述符个数无限制。

## Epoll【异步 IO 机制】

- 1、epoll\_create 创建红黑树节点 + 就绪链表
- 2、epoll\_ctl 将接收到的文件描述符存入到红黑树以及注册对应的回调函数例如读写
- 3、当内核监听到文件描述符发生事件响应, 回调函数会将文件描述符移动到就绪链表
- 4、一旦就绪链表产生文件描述符, epoll\_wait 就会通知 Redis 将内核态的数据复制到用户态

**【水平触发模式 (Redis 默认): 就绪链表产生文件描述符就会通知 Redis 将内核态的数据复制到用户态】**

**【边缘触发模式: 一定时间内触发一次就绪链表上的文件描述符通知 Redis 将内核态的数据复制到用户态】**



总结

select、poll都需要将有关文件描述符的数据结构拷贝进内核，最后再拷贝出来，涉及到用户态和内核态的转换，非常影响性能。  
而 epoll 模式下，在内核态和用户态都可以访问的共享空间创建红黑树和就绪链表，那么内核态只需和共享空间打交道，所以 epoll 模式下只需一次的用户态内核态转换，不像 select 和 poll 需要 2 次转换。并且 epoll 是基于回调机制的，无需遍历所有的文本描述符集合，效率极高。

---

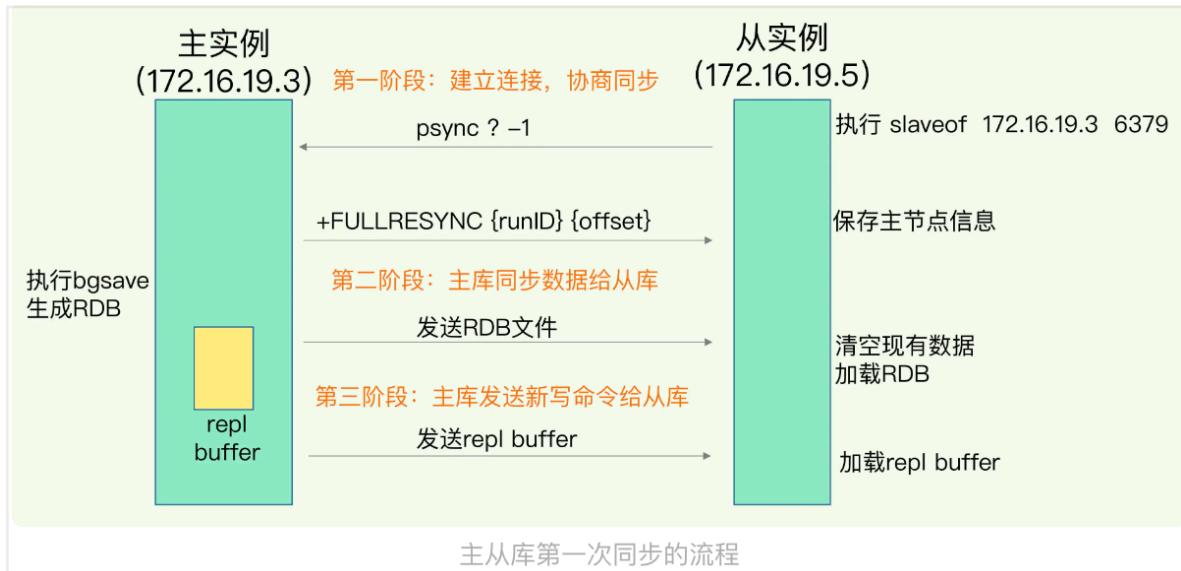
---

## 数据同步：主从库如何实现数据一致？【主从模式】

redis 高可用：1、数据尽可能少的丢失--AOF 和 RDB 2、服务尽可能少中断-- 多个 redis 副本

Redis 读写分离：主库负责写，从库负责读，如果不分离且从库也可以写操作，那为了维护数据一致性协同所有主从库要加锁会导致开销很大。

主从数据同步：通过 replicaof 命令可以设置某库成为某库的从库。下图为第一次主从同步的三个阶段：



ps：第三阶段指的是当在执行第二阶段时接收到新的写命令存储在 replication buffer，第三阶段就是同步 replication buffer 里面的数据到从库，这样完成了主从数据复制了。

\*\* replication buffer：主库内存中专门用于记录生成 rdb 后接收到新的写操作命令。

\*\*这三个阶段预示着首次的全量复制完成

### 主从级联模式分担全量复制时的主库压力

如果从库很多，主库会忙于 fork 子进程生成 rdb 文件进行全量同步。fork 这个操作会阻塞主线程处理正常请求，从而导致主库响应请求变慢。并且大量传输 rdb 会占用主库网络带宽，同样会给主库的资源使用带来压力。所以可以通过“主-从-从”模式去

让从库同步给别的从库。

\*\*基于长连接的命令传播

主从库间网络断了怎么办：

采用增量复制-网络断的期间，把主库收到的命令同步给从库。无需全量复制

repl\_backlog\_buffer：环形缓冲区，主库会记录自己写到的位置，从库则会记录自己已经读到的位置。

master\_repl\_offset：主库偏移量

slave\_repl\_offset：从库偏移量

解释：在主从网络没断开的时，主从库偏移量保持一致。当网络断了且连接恢复后，从库会发一个psync命令给主库重新建立链接（此时主库偏移量>从库偏移量，因为主库偏移量一直在递增），然后把自己从库的偏移量告诉主库。这样主库通过自己的偏移量和从库的偏移量之间的差距命令操作同步给同库就好了。

ps:由于repl\_backlog\_buffer是一个环形，可能存在从库读取速度远低于主库写的速度，这样容易造成从库未读取的数据被主库写的操作覆盖了，可以通过调整repl\_backlog\_size这个参数达到控制环形缓冲空间大小，这样在遇到高峰期时主库写操作不会太快覆盖整个环形并且远领先于从库读取速度。\*\*如果确实发生了，会进行一次全量复制（耗时）

总结：Redis 的主从库同步的基本原理，有三种模式：全量复制（第一次主从三个阶段连接的全量复制同步）、基于长连接的命令传播（主从级联模式分担主库压力），以及增量复制（主从之间网络断了）。

### replication buffer 和 repl\_backlog\_buffer 的区别

总的来说，replication buffer 是主从库在进行全量复制时，主库上用于和从库连接的客户端的buffer，而repl\_backlog\_buffer 是为了支持从库增量复制，主库上用于持续保存写操作的一块专用buffer。

Redis 主从库在进行复制时，当主库要把全量复制期间的写操作命令发给从库时，主库会先创建一个客户端，用来连接从库，然后通过这个客户端，把写操作命令发给从库。在内存中，主库上的客户端就会对应一个buffer，这个buffer 就被称为replication buffer。Redis 通过client\_buffer 配置项来控制这个buffer 的大小。主库会给每个从库建立一个客户端，所以replication buffer 不是共享的，而是每个从库都有一个对应的客户端。

repl\_backlog\_buffer（从库共享）是一块专用buffer，在Redis 服务器启动后，开始一直接收写操作命令，这是所有从库共享的。主库和从库会各自记录自己的复制进度，所以，不同的从库在进行恢复时，会把自己的复制进度(slave\_repl\_offset) 发给主库，主库就可以和它独立同步。

---

## 哨兵机制：主库挂了，如何不间断服务？【哨兵模式】

哨兵：特殊模式下的 Redis 进程；作用：1、监控 2、选主（选择主库）3、通知

监控：通过PING来监控主从

选主：主库挂了，在从库中按一定的机制选择一个新主库

通知：通知其他从库（然后会把主库信息发给所有的从库，让从库执行replicaof 和新主库建立连接）和客户端新的主库信息（哨兵会把新主库的连接信息通知给客户

端，让它们把请求操作发到新主库上)

## 监控

当一个哨兵监测到主库 PING 命令响应超时会判定为“主观下线”(由于网络不稳定等因素可能造成单个哨兵误判)，只有多个哨兵对主库监测都认为“主观下线”，主库才会被标记为“客观下线”。

“客观下线”的标准：当有 N 个哨兵实例时，最好要有  $N/2 + 1$  个实例判断主库为“主观下线”，才能最终判定主库为“客观下线”。

## 选主

规则：从库优先级、从库复制进度以及从库 ID 号(按照规则顺序匹配，满足则自动选举为主库，不满足看下一个规则，而不是任意满足其中一个条件就当选主库)

从库优先级：在设置从库实例内存配置的时候，配置最高的选举为主库

从库复制进度：最大的从库 slave\_repl\_offset(说明这个从库数据同步主库速度是最快的)的选举为主库

从库 ID 号：ID 号越小的选举为主库(最小的说明最早连接上，数据也就最全，能活到现在也就最稳定，毕竟久经考验)

## 通知

客户端通过订阅哨兵提供的消息订阅频道(哨兵 leader 执行完主从切换后会通知其他所有哨兵，然后其他哨兵发布频道消息通知对应监听的从库与新主库建立连接；并且哨兵 leader 也会通知客户端与新主库建立连接)，详见第 8 章

---

---

# 哨兵集群：哨兵挂了，主从库还能切换吗？【哨兵模式】

哨兵之间如何建立连接形成集群？--基于 pub/sub 机制的哨兵集群组成

- 1、所有哨兵需配置主库信息(IP+端口信息，即建立连接) && 订阅主库频道“\_\_sentinel\_\_:hello”。
- 2、发布自己的 IP+端口信息到主库频道(供后面进来的哨兵获取信息建立网络连接)
- 3、从主库频道获取其他哨兵 IP+端口信息
- 4、通过 IP+端口同别的哨兵建立网络连接
- 5、哨兵集群形成(即多个 redis 实例，哨兵也是实例，只不过用途于监控通知之类的)

哨兵是如何知道从库的 IP 地址和端口的呢？

通过发送 INFO 命令给主库获取所有从库信息(这样哨兵可以针对主库以及所有从库建立连接并且监控)

客户端(服务器)如何感知哨兵集群在监控、选主、切换这个过程中发生的各种事件？

客户端通过订阅哨兵提供的消息订阅频道，即每个哨兵实例也提供 pub/sub 机制。哨兵可提供的重点频道如下：

事件	相关频道
主库下线事件	+sdown (实例进入“主观下线”状态)
	-sdown (实例退出“主观下线”状态)
	+odown (实例进入“客观下线”状态)
	-odown (实例退出“客观下线”状态)
从库重新配置事件	+slave-reconf-sent (哨兵发送SLAVEOF命令重新配置从库)
	+slave-reconf-inprog (从库配置了新主库，但尚未进行同步)
	+slave-reconf-done (从库配置了新主库，且和新主库完成同步)
新主库切换	+switch-master (主库地址发生变化)

### 由哪个哨兵执行主从切换？

- 1、任一哨兵实例判断主库“主观下线”后，就会向其他哨兵发送is-master-down-by-addr 命令，其他哨兵根据自身和主库连接情况做出 Y/N 回应，Y 相当于赞成票，N 相当于反对票。
- 2、当得到的返回 Y 票数超过哨兵配置文件中的 quorum 配置项设定（Y 票数包括本身这一票），即当前哨兵认定主库为“客观下线”。
- 3、当前哨兵如果认定为“客观下线”就有资格进入执行切换主从（选主）的“Leader 选举”。
- 4、成为 Leader 的哨兵，要满足两个条件：第一，拿到半数以上的赞成票；第二，拿到的票数同时还需要大于等于哨兵配置文件中的 quorum 值。
- 5、参与选举的哨兵，会默认给自己投一票。投过 Y 票的哨兵不再支持再给别的哨兵返回 Y，只会返回 N。所以优先发起选举的哨兵，是最有概率成为 leader 的（越早请求别的哨兵越容易占领票数的市场份额），因为投过 Y 票的人也不能参与选举了。
- 6、如果当下由于网络传输堵塞等原因无法获取半数以上的票数导致没有选举出 Leader，那么哨兵集群会等一段时间再次发起“Leader 选举”（之后网络可能变好了，也大大提升了选举成功概率）。
- 7、PS：哨兵选举至少 3 个实例以上，哨兵对接收到第一个选举投票 say yes，对之后的 say no。如果配置 2 个实例很可能一直无法进行主从切换。
- 8、哨兵 leader 负责实际的主从切换，即由它来完成新主库的选择以及通知从库与客户端（只有哨兵 leader 才能操作主从切换）。

---

## 切片集群：数据增多了，是该加内存还是加实例？【切片模式】

### 如何保存更多数据？

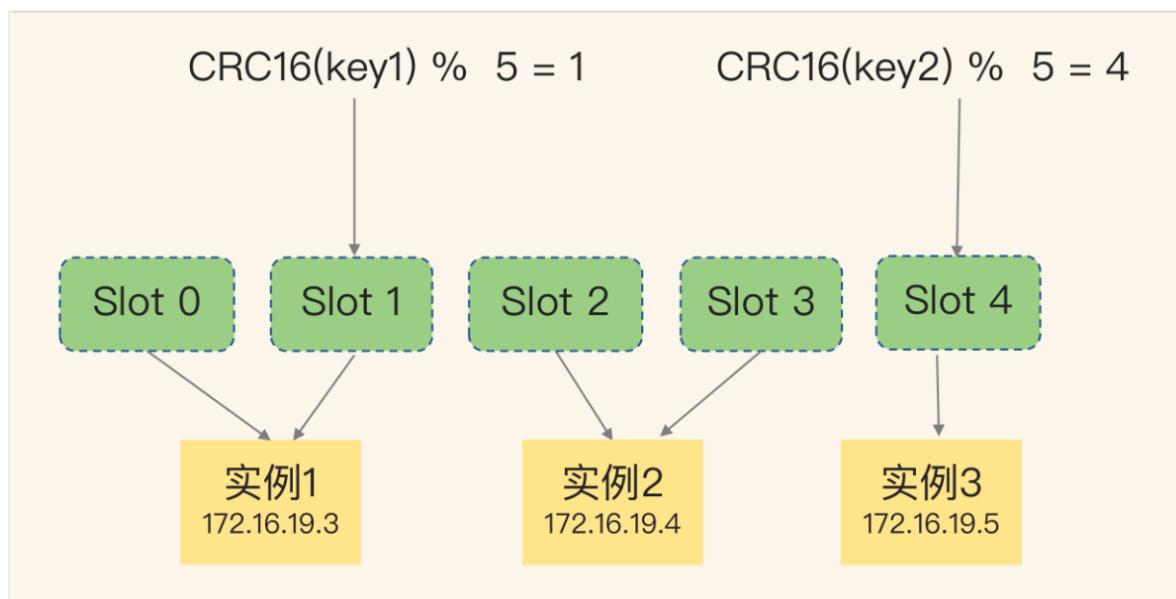
- 1、纵向扩展：增加内存 + 硬盘配置（如果不要求持久化 RDB 这个方案比较好）

优点：扩展使用简单；缺点：受硬件限制，大小受限，以及 RDB 持久化 fork 进程时会比较长且易阻塞主进程

2、横向扩展（即 Redis 切片集群，有点像 MySQL 的分库分表）：增加 redis 实例不受硬件限制，大小几乎无限。不影响性能；缺点：复杂度增加

### 数据切片和实例的对应分布关系

通过 cluster create 命令创建集群，redis 会根据 CRC16 算法创建 16384 个槽，如果集群中有 N 个实例，那么，每个实例上的槽个数为  $16384/N$  个（即一实例对应多槽，手动分配哈希槽时需要把 16384 都分配完不然无法正常工作）。



### 客户端如何定位数据？

流程：客户端请求键值对 -> 计算键所对应的哈希槽 -> 请求哈希槽对应的实例

### 客户端和集群实例刚建立连接后如何知道别的实例所拥有的哈希槽信息？

在集群刚创建的时候，每个实例只知道自己被分配了哪些哈希槽，是不知道其他实例拥有的哈希槽信息的。

但是通过 gossip 协议实现无中心化通信，不同 Redis 实例会把自己的哈希槽信息发给和它相连接的其它实例，来完成哈希槽分配信息的扩散。当实例之间相互连接后，每个实例就有所有哈希槽的映射关系了。并且把所有槽分别对应的实例信息缓存到客户端，这样客户端就能知道每个槽对应的实例直接发送请求。

### 集群中的实例和哈希槽的对应关系并不是一成不变的

#### 造成原因

- 1、在集群中，实例有新增或删除，Redis 需要重新分配哈希槽
- 2、为了负载均衡，Redis 需要把哈希槽在所有实例上重新分布一遍

#### 造成问题现象

实例之间还可以通过相互传递消息，获得最新的哈希槽分配信息，但是，客户端是无法主动感知这些变化的。这就会导致，它缓存的分配信息和最新的分配信息就不一致了

#### 解决方案

Redis Cluster 方案提供了一种重定向机制。(类似于 HTTP 将客户端重定向一样，客户端请求了一个已经不存在或移走的资源，服务端需要通过 header Location 信息告诉客户端去请求某个新地址。这里 redis 也差不多是这样的原理)

### 重定向时出现的两种情况：1、哈希槽及携带的数据已迁移到别的 redis 实例；2、迁移中

1、请求时发现当前实例无对应哈希槽，实例会返回 moved 命令结果（携带新实例地址），则客户端会重新请求新实例地址。然后客户端更新本地哈希槽信息缓存。

2、实例返回 ASK 命令，这个命令意思是：让这个实例允许执行客户端接下来发送的命令。然后，客户端再向这个实例发送 GET 命令，以读取数据。

ASK 命令两层含义：

第一，表明 Slot 数据还在迁移中；

第二，ASK 命令把客户端所请求数据的最新实例地址返回给客户端（若请求的请求的哈希槽已经迁移到别的实例中时，ASK 会返回最新实例地址，反之直接返回当前实例）。

PS：和 MOVED 命令不同，ASK 命令并不会更新客户端缓存的哈希槽分配信息。这也就是说，ASK 命令的作用只是让客户端能给新实例发送一次请求，而不像 MOVED 命令那样，会更改本地缓存，让后续所有命令都发往新实例。

### MOVED 和 ASK 区别

因为只是部分数据已经迁移到了新的实例，所以下次请求仍然需要先请求旧的实例，旧的实例上没有的再请求新实例，直至都迁移完成后，不需要再和旧的实例交互了，就返回 MOVED（返回 MOVED 就会更新客户端哈希槽缓存）。MOVED 和 ASK 有点临时重定向和永久重定向的味道。

PS：多实例主集群分别分配 1 个及以上个哈希槽（主集群会分配完所有的哈希槽），并且每个主实例写数据时只写自己所管辖的哈希槽。每个从集群也会分配 1 个及以上的哈希槽，当根据 key 计算出对应的槽位时直接请求对应的从实例去进行查询。不过从实例接管的哈希槽要属于对应主实例所管辖的范围内。并且通过 gossip 协议去监控主从的存活情况，当发现有某一个主实例挂了，下面的从实例就会选举成为新主实例（这一切行为通过 gossip 网络协议进行）。

---

---

## \*\*Redis 三种部署模式（总结）

Redis 三种部署模式：主从模式、哨兵模式、集群模式

参考博客：<https://zhuanlan.zhihu.com/p/651365147>

### 主从模式

主从复制的优缺点

优点：

- 1、配置简单，易于实现。
- 2、实现数据冗余，提高数据可靠性。
- 3、读写分离，提高系统性能。

缺点：

- 1、主节点故障时，需要手动切换到从节点，故障恢复时间较长。

2、主节点承担所有写操作，可能成为性能瓶颈。

3、无法实现数据分片，受单节点内存限制。

### 主从复制场景应用

1、数据备份和容灾恢复：通过从节点备份主节点的数据，实现数据冗余。

2、读写分离：将读操作分发到从节点，减轻主节点压力，提高系统性能。

3、在线升级和扩展：在不影响主节点的情况下，通过增加从节点来扩展系统的读取能力。

### 总结：

主从复制模式适合数据备份、读写分离和在线升级等场景，但在主节点故障时需要手动切换，不能自动实现故障转移。如果对高可用性要求较高，可以考虑使用哨兵模式或 Cluster 模式。

## 哨兵模式

哨兵模式是在主从复制基础上加入了哨兵节点，实现了自动故障转移。哨兵节点是一种特殊的 Redis 节点，它会监控主节点和从节点的运行状态。当主节点发生故障时，哨兵节点会自动从从节点中选举出一个新的主节点，并通知其他从节点和客户端，实现故障转移。

### 优点：

1、自动故障转移，提高系统的高可用性。

2、具有主从复制模式的所有优点，如数据冗余和读写分离。

### 缺点：

1、配置和管理相对复杂。

2、依然无法实现数据分片，受单节点内存限制。

### 场景应用：

1、通过自动故障转移，确保服务的持续可用。

2、数据备份和容灾恢复：在主从复制的基础上，提供自动故障转移功能。

### 总结：

哨兵模式在主从复制模式的基础上实现了自动故障转移，提高了系统的高可用性。

然而，它仍然无法实现数据分片。如果需要实现数据分片和负载均衡，可以考虑使用 Cluster 模式。

## Redis-Cluster 集群

Redis 的哨兵模式虽然已经可以实现高可用，读写分离，但是存在几个方面的不足：

- 哨兵模式下每台 Redis 服务器都存储相同的数据，很浪费内存空间；数据量太大，主从同步时严重影响了 master 性能。
- 哨兵模式是中心化的集群实现方案，每个从机和主机的耦合度很高，master 容机到 slave 选举 master 恢复期间服务不可用。
- 哨兵模式始终只有一个 Redis 主机来接收和处理写请求，写操作还是受单机瓶颈影响，没有实现真正的分布式架构。

redis cluster主要是针对海量数据+高并发+高可用的场景，海量数据，如果你的数据量很大，那么建议就用 redis cluster，数据量不是很大时，使用 sentinel就够了。redis cluster的性能和高可用性均优于哨兵模式。Redis Cluster采用虚拟哈希槽分区而非一致性 hash 算法，预先分配一些卡槽，所有的键根据哈希函数映射到这些槽内，每一个分区内的 master 节点负责维护一部分槽以及槽所映射的键值数据。

### 与其它集群模式的区别

- 相比较 sentinel 模式，多个 master 节点保证主要业务（比如 master 节点主要负责写）稳定性，不需要搭建多个 sentinel 实例监控一个 master 节点；

- 相比较一主多从的模式，不需要手动切换，具有自我故障检测，故障转移的特点；
- 相比较其他两个模式而言，对数据进行分片（sharding），不同节点存储的数据是不一样的；
- 从某种程度上来说，Sentinel模式主要针对高可用（HA），而Cluster模式是不仅针对大数据量，高并发，同时也支持HA。

## 得物用阿里云的Redis切片模式

阿里云RDS属于proxy代理模式（负责key的槽位定点-代替客户端去计算、监控、选主、切换、发布/订阅等，替代了部分客户端功能+gossip通信模式）。比如得物的集群版-双副本（即切片集群），都是一主一从模式（即双副本模式），写和读都是master控制，slave只负责当master挂了的时候去做切主使用。并且每个主同样也是负责多个哈希槽，等等后面的概念和原生redis差不多了【切片集群可以将大量数据分摊到多个实例上，就算有大量请求进来，也会摊到多个实例去执行，因此读写速度也提高了很多，如果慢就加实例】。

The screenshot shows the Alibaba Cloud RDS Redis instance management interface. On the left, there's a sidebar with various settings like Instance Information, Performance Monitoring, Alert Settings, and so on. The main area is titled 'prd-商品-spu基础数据...' and shows '运行中' (Running). It has tabs for '服务可用区信息' (Service Availability Zone Information) and '备可用区: 杭州 可用区K'. Below these are sections for '数据节点' (Data Nodes) and '角色' (Role). Two shards are listed:

节点ID	角色	可用区
r-bp124e49c93cdbf4-db-0	master slave	杭州 可用区I
r-bp124e49c93cdbf4-db-1	master slave	杭州 可用区I

## Redis6.0之前为什么不使用多线程？

- 单线程编程容易并且更容易维护；
- Redis的性能瓶颈不在CPU，主要在内存和网络；
- 多线程就会存在死锁、线程上下文切换等问题，甚至会影响性能（多线程虽然会帮助我们更充分地利用CPU资源，但是操作系统上线程的切换也不是免费的，线程切换其实会带来额外的开销，例如保存线程1的执行上下文&&加载线程2的执行上下文）。

PS: Redis4.0后（特定场景的异步，例如删除，不过还不是多线程），加入了UNLINK（如果删除的数量没超过64个元素还是默认调同步Del进行删除）、FLUSHALL ASYNC、FLUSHDB ASYNC等非阻塞的删除操作。因为当删除的数据很多会占用主线程资源，所以异步删除会先删除元数据命名空间里面的key（客户端保存的键值对+对应机器实例），真正的删除操作会在后台异步执行。

## Redis6.0 之后为何引入了多线程? (默认关闭)

Redis6.0 引入多线程主要是为了提高网络 IO 读写性能, 因为这个算是 Redis 中的一个性能瓶颈 (Redis 的瓶颈主要受限于内存和网络)。

I/O 线程仅仅是读取和解析客户端命令而不会真正去执行命令, 客户端命令的执行最终还是要在主线程上完成。

缺陷: 在 Redis 的多线程方案中, I/O 线程任务仅仅是通过 socket 读取客户端请求命令并解析, 却没有真正去执行命令, 所有客户端命令最后还需要回到主线程去执行, 因此对多核的利用率并不算高, 而且每次主线程都必须在分配完任务之后轮询等待所有 I/O 线程, 完成任务之后才能继续执行其他逻辑。

性能提升: 测试数据表明, Redis 在使用多线程模式之后性能大幅提升, 达到了一倍。

Redis 多线程网络模型全面揭秘 : <https://segmentfault.com/a/1190000039223696>

---

## Redis 阻塞原因汇总

Redis 常见阻塞原因总结: <https://javaguide.cn/database/redis/redis-common-blocking-problems-summary.html>

- 1、O(n) 命令
  - 2、SAVE 创建 RDB 快照
  - 3、AOF (AOF 记录日志是在 Redis 主线程中进行的)
  - 4、大 Key
  - 5、清空数据库 (执行 Del 操作, 除非超过 64 个元素才会异步)
  - 6、集群扩容
  - 7、Swap (内存交换) -- 当内存不足的时候, 把一部分硬盘空间虚拟成内存使用, 从而解决内存容量不足的情况。因此, Swap 分区的作用就是牺牲硬盘, 增加内存, 解决 VPS 内存不够用或者爆满的问题。  
Swap 对于 Redis 来说是非常致命的, Redis 保证高性能的一个重要前提是所有的数据在内存中。如果操作系统把 Redis 使用的部分内存换出硬盘, 由于内存与硬盘的读写速度差几个数量级, 会导致发生交换后的 Redis 性能急剧下降。
  - 8、CPU 竞争 (Redis 是典型的 CPU 密集型应用, 不建议和其他多核 CPU 密集型服务部署在一起。当其他进程过度消耗 CPU 时, 将严重影响 Redis 的吞吐量。)
  - 9、网络问题 (连接拒绝、网络延迟, 网卡软中断等网络问题也可能会导致 Redis 阻塞)
- 

## Redis 优化手段汇总

### 1、使用批量操作减少网络传输

一个 Redis 命令的执行可以简化为以下 4 步: 1.发送命令、2.命令排队、3.命令执行、4.返回结果--批量操作可以减少网络传输, 但是在分片集群场景下, 例如 mget 时候也不能保证所有 key 都在一个 hash slot, 如果不在一个 hash 槽的话客户端会发起多次请求对应的槽位, 还会产生网络传输

## 2、大量 key 集中过期问题

Redis 采用的是 定期删除 + 惰性 / 懒汉式删除 策略 来释放 key 的使用内存

定期删除是主线程执行 (容易阻塞客户端请求), 惰性删除是异步单独子线程执行 (开启 lazy-free)

不过无论如何, 在设置 key 的时候设置随机过期时间, 尽量避免同一时间段全部过期。

## 3、避免 Redis bigkey (大 Key)

### 影响

- 客户端超时阻塞: 由于 Redis 执行命令是单线程处理, 然后在操作大 key 时会比较耗时, 那么就会阻塞 Redis, 从客户端这一视角看, 就是很久很久都没有响应。
- 网络阻塞: 每次获取大 key 产生的网络流量较大, 如果一个 key 的大小是 1 MB, 每秒访问量为 1000, 那么每秒会产生 1000MB 的流量, 这对于普通千兆网卡的服务器来说是灾难性的。
- 工作线程阻塞: 如果使用 del 删除大 key 时, 会阻塞工作线程, 这样就没办法处理后续的命令。

### 解决

- 分割 bigkey: 将一个 bigkey 分割为多个小 key。例如, 将一个含有上万字段数量的 Hash 按照一定策略 (比如二次哈希) 拆分为多个 Hash。
- 手动清理: Redis 4.0+ 可以使用 UNLINK 命令来异步删除一个或多个指定的 key。Redis 4.0 以下可以考虑使用 SCAN 命令结合 DEL 命令来分批次删除。
- 采用合适的数据结构: 例如, 文件二进制数据不使用 String 保存、使用 HyperLogLog 统计页面 UV、Bitmap 保存状态信息 (0/1)。
- 开启 lazy-free (惰性删除 / 延迟释放): lazy-free 特性是 Redis 4.0 开始引入的, 指的是让 Redis 采用异步方式延迟释放 key 使用的内存, 将该操作交给单独的子线程处理, 避免阻塞主线程。

## 4、Redis hotkey (热 Key)

### 危害

处理 hotkey 会占用大量的 CPU 和带宽, 可能会影响 Redis 实例对其他请求的正常处理。此外, 如果突然访问 hotkey 的请求超出了 Redis 的处理能力, Redis 就会直接宕机。这种情况下, 大量请求将落到后面的数据上, 可能会导致数据库崩溃。因此, hotkey 很可能成为系统性能的瓶颈点, 需要单独对其进行优化, 以确保系统的高可用性和稳定性。

### 解决

- 读写分离: 主节点处理写请求, 从节点处理读请求。
- 使用 Redis Cluster: 将热点数据分散存储在多个 Redis 节点上。
- 二级缓存: hotkey 采用二级缓存的方式进行处理, 将 hotkey 存放一份到 JVM 本地内存中 (可以用 Caffeine)。

## 5、避免慢查询命令 (时间复杂度为 O(n): keys、lrange 等命令)

## 6、Redis 内存碎片

Redis 内存碎片虽然不会影响 Redis 性能, 但是会增加内存消耗。

## 解决

- jemalloc (内存分配器) 专门针对内存碎片问题做了优化，一般不会存在过度碎片化的问题。
  - 直接通过 config set 命令将 activedefrag 配置项设置为 yes 即可。
  - 重启节点可以做到内存碎片重新整理
- 
- 

## Redis 生产问题 (重要)

### 缓存穿透

什么是缓存穿透：缓存穿透说简单点就是大量请求的 key 是不合理的，根本不存在于缓存中，也不存在于数据库中。这就导致这些请求直接到了数据库上，根本没有经过缓存这一层，对数据库造成了巨大的压力，可能直接就被这么多请求弄宕机了。

## 解决

- 做好传参校验防止黑客恶意攻击 (uid 存在、请求 id 不能<0、格式校验等)
- 缓存无效 key：查完数据库 set 缓存，就算数据库没查到数据也要 set 缓存（空缓存），过期时间正常设置。主要防止受到攻击时但是 key 变化不频繁可以有效挡住穿透到数据库【这个做法不能从根本解决穿透问题，如果制造大量不同恶意无效 key，请求还是会落到数据库】
- 布隆过滤器 - [《不了解布隆过滤器？一文给你整的明明白白！》](#)

做法：把所有可能存在的请求的值都存放在布隆过滤器中，当用户请求过来，先判断用户发来的请求的值是否存在于布隆过滤器中。不存在的话，直接返回请求参数错误信息给客户端，存在的话才会走下面的流程。(底层是位存储，所以适用海量数据，空间占比小，缺点是具有一定的错误识别率和删除难度。并且，理论情况下，添加到集合中的元素越多，误报的可能性就越大【哈希冲突】)

### 缓存击穿

解释：缓存击穿中，请求的 key 对应的是 热点数据，该数据 存在于数据库中，但不存在于缓存中（通常是因为缓存中的那份数据已经过期）。这就可能会导致瞬时大量的请求直接打到了数据库上，对数据库造成了巨大的压力，可能直接就被这么多请求弄宕机了。

场景：类似秒杀场景，该商品缓存过期了，在查不到缓存去查数据库但是还没来得及重新 set 缓存期间，一瞬间大量请求打进来都落到了数据库。

## 解决

- 设置热点数据永不过期或者过期时间比较长。
- 数据提前预热
- 加锁，保证一次只有一个请求落到数据库上

### 缓存雪崩

解释：同一时间出现大量缓存失效，导致请求都打到数据库来了。

## 解决

- 缓存过期时间设置随机
- 预热（用定时任务、MQ、set 数据更新最新缓存等都可以）

## 缓存穿透和缓存击穿区别

- **发生场景**。缓存穿透是访问一个不存在的key，缓存不起作用，请求会穿透到数据库；缓存击穿是访问一个存在的key，在缓存过期的一刻，同时有大量的请求，这些请求都会击穿到数据库。
- **影响**。缓存穿透因为访问的数据不存在，所以缓存中也没有数据，导致这些请求都会直接打到数据库上，流量大时数据库会挂掉；缓存击穿是因为缓存过期，正好又有很多并发请求过来，这些请求都会直接打到数据库上，造成瞬时数据库请求量大、压力骤增。

## 如何保证缓存和数据库数据的一致性？

博客：[缓存和数据库一致性问题，看这篇就够了 - 水滴与银弹](#)

### 1. 提高缓存利用率

写：更新数据库

查：先查缓存，查不到则查数据库然后塞缓存 + 设置失效时间

【这样能保证不经常访问的缓存逐渐过期淘汰掉，使得缓存利用率最大化】

### 2. 数据一致性

- 方案 1：先更新缓存，后更新数据库

如果缓存更新成功而数据库更新失败了，导致缓存过期后会从数据库读到错误的数据。

- 方案 2：先更新数据库，后更新缓存

如果更新数据库成功而更新缓存失败了，导致缓存没过期这段时间读取到错误的数据。

**【所以无论是方案 1 还是方案 2，只要后者的后者执行发生异常，都会造成业务影响】**

并且在并发情况下，如果线程 T1 先于 T2 执行【更新数据库，后更新缓存】，理论上 T1 更新缓存的操作也是优先于 T2，但是由于 CPU 调度可能发生执行顺序错乱【导致先触发 T2 更新缓存，那么当 T1 抢到 CPU 资源的时候执行更新缓存会把 T2 的缓存数据覆盖了，理论上缓存数据是后执行的 T2 触发的】，导致缓存数据和数据库数据不一致。

**【由于 CPU 调度顺序执行问题，所以当更新数据库后需要删除缓存，而不是更新缓存，不然会出现数据不一致】**

**结语：本来思考更新数据库还是更新缓存哪个优先，但是存在并发下 CPU 调度顺序问题，进而思考更新数据库和删除缓存哪个优先，从而摒弃了更新缓存的方案用删除缓存方案代替。**

### 3. 数据一致性之删除缓存【并发场景】

- 先删除缓存，后更新数据库【读写并发场景，T1 写，T2 读】

T1 先删除缓存，还没来得及更新数据库时，T2 进来发现缓存不存在则读取数据库然后将老数据塞到缓存，然后 T1 把最新值更新到数据库。导致缓存和数据库不一致。

- 先更新数据库，后删除缓存【读写并发场景，T1 读，T2 写】

一开始缓存不存在，T1 读取数据库【老数据】，还没来得及塞缓存，T2 就写数据库 + 删除缓存，然后 T1 才塞缓存。这样导致了缓存和数据库不一致。

**但是【先更新数据库，后删除缓存】场景发生数据不一致的概率很低，原因如下**

- 1、缓存刚好失效，这样 T1 才会去读数据
- 2、同一时间节点出现读写并发
- 3、T1 读数据库和塞缓存空隙之间，T2 完成了写数据库 + 删缓存，一般来说不太可能出现写数据库比读数据库更快。所以 T1 空隙之间 T2 无法完成写数据库 + 删缓存两个动作。

**结语：所以并发场景下，先更新数据库再删除缓存是能保证数据性一致**

## **4. 数据一致性之删除缓存失败**

先更新数据库，后删除缓存这种方案，一般情况下更新数据库会有事务回滚机制，所以可以保证数据库层面的操作如果失败就不操作数据库 + 不操作缓存。那么就应该考虑删除缓存失败了怎么办，一旦出现失败也会导致缓存和数据库不一致。

**如何保证删除缓存的动作成功？**

**同步重试删除缓存【摒弃】**

- 重试要考虑重试次数
- 一直重试还会占用线程资源
- 重试会导致接口后续流程的阻塞
- 重试过程中服务重启也会导致重试丢失造成数据不一致

**异步重试删除缓存【采用】**

**如下两种异步方案**

- 写数据库操作结束后，发 MQ 去处理删除缓存的事情，MQ 有重试机制，也不用在意服务器重启的问题。
- 监听 MySql 的 binlog，当数据发生变动后去删除缓存。

**总结：数据一致性可以通过【先更新数据库，后删除缓存】+【消息队列保证删除缓存动作成功】**

## **5. 【先更新数据库，后删除缓存】在数据库主从同步出现数据不一致问题**

**问题场景**

线程 T1 写数据库 + 删缓存，但是数据库从库的数据还没同步到最新且缓存已经被删除，此时 T2 去从查缓存没查到，就去数据库的从库查数据查到了老数据，再将老数据塞回缓存，随后从库才完成了最新数据的同步。这样就出现了数据不一致了。

**解决方案：通过 MQ 做延迟删除**

1、在【先删缓存，后更新数据库】方案下

采用延迟双删，先删一次缓存，再更新数据库，然而由于主从同步的延迟导致别的查询查到了从库的旧数据，导致缓存更新为数据库更新之前的数据，那么此刻我们再来一次延迟 X 秒删一次缓存，这就是延迟双删【现在都不太采用先删缓存，后更新数据库的策略，所以这个双删策略基本也没人使用】。

2、在【先更新数据库，后删除缓存】方案下

采用延迟删缓存，保证主从同步完后再删除缓存，这样避免了由于主从同步延迟导致的查询查到了从库的旧数据【不过在延迟的时间内，容易造成当下查到旧数据，无法保证数据强一致性】。

## **6.一定要数据保持强一致性？**

也可以做，通过分布式锁保证更新数据库 + 删除缓存之前，不能有任何其他请求进来，但是这样一来性能又会下降。所以需要根据不同场景对一致性的容忍程度去选择最优方案。

## **7.总结**

先更新数据库再删除缓存，更新数据库动作的成功由事务的回滚机制来保证，删除缓存动作的成功由 MQ 去解决，并且还能保证重试机制 + 无需关注服务器重启问题。如果考虑到数据主从同步问题，还可以考虑延迟删缓存的方案，但是会存在一定延时时间内查到旧数据，实在要保持数据强一致性，那么就考虑分布式锁，更新数据库 + 删除缓存之前不允许别的请求进来，但是性能又下降了。

### **得物目前前后端做法【C端只负责查不负责更新数据库】**

基于性能等综合考虑采用【更新数据库 + 删除缓存】方案，也能接受一定的数据不一致的容忍度。因为我们的缓存主要用于展示，并且缓存也有失效时间。如果查询的数据涉及到更新数据库等操作，我们的规范是指定去数据库而非查缓存。这样可以对整体服务的性能有所提升。

---

## **Redis 分布式锁**

### **简介**

分布式锁是控制不同系统之间访问共享资源的一种锁实现，如果不同的系统或同一个系统的不同主机之间共享了某个资源时，往往需要互斥来防止彼此干扰来保证一致性。

### **秒杀扣减库存场景**

在单机情况下可以通过 synchronized 同步代码块保证多线程有序进行扣减库存但是在集群情况下，加 synchronized 同步代码块就没用了，因为 synchronized 只会在一个 JVM 中生效。

因此在集群部署的时候，可以通过 Redis 的 setnx (只在键 key 不存在的情况下，将键 key 的值设置为 value。若键 key 已经存在，则 SETNX 命令不做任何动作。) 的指令来解决的。

### **setNx 代码优化**

- 通过 try-catch-finally delete key 保证在业务代码抛异常不出现死锁
- 通过 setnx + 过期时间解决服务器宕机重启导致 key 没有被删除而出现死锁情况
- 如果 redis 过期时间设置为 10s，但是出现了网络波动或者慢 sql 等导致程序执行时间需要 15s，那么在程序执行到 11s 的时候 key 已经过期被删除了，如果这个时候有请求进来则会认为目前没有业务正在执行，那么这个请求也能触发业务执行。**【解决方案：续命锁】**

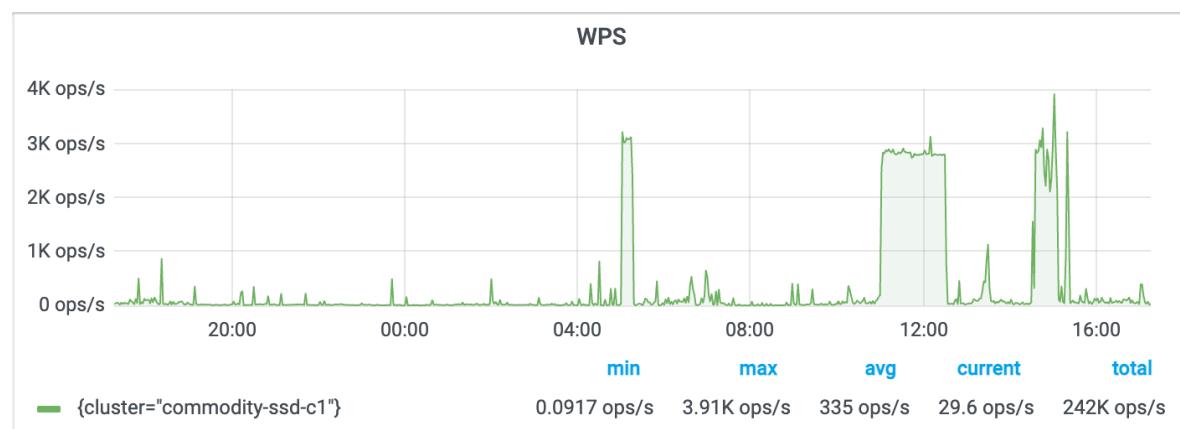
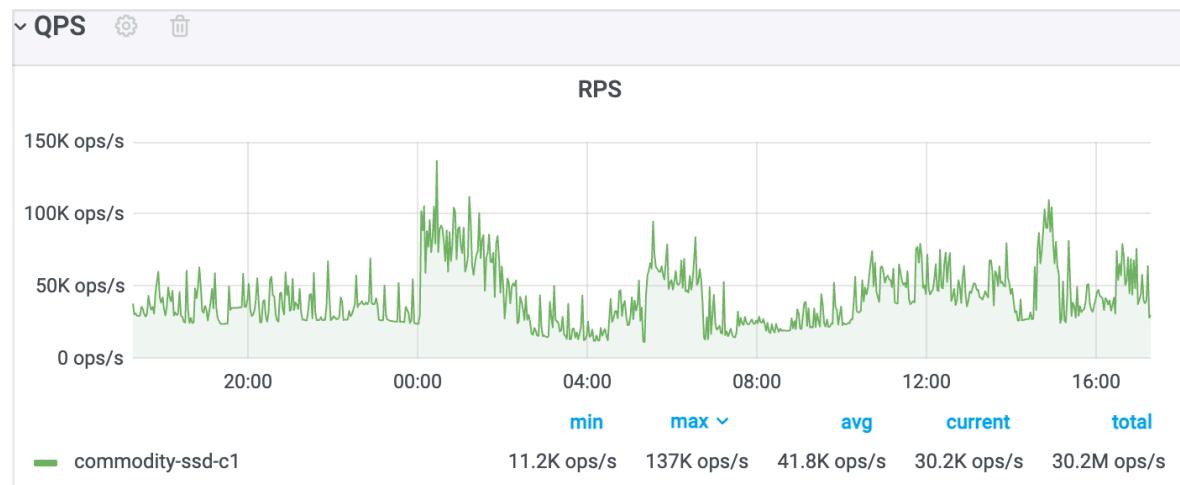
### **续命锁**

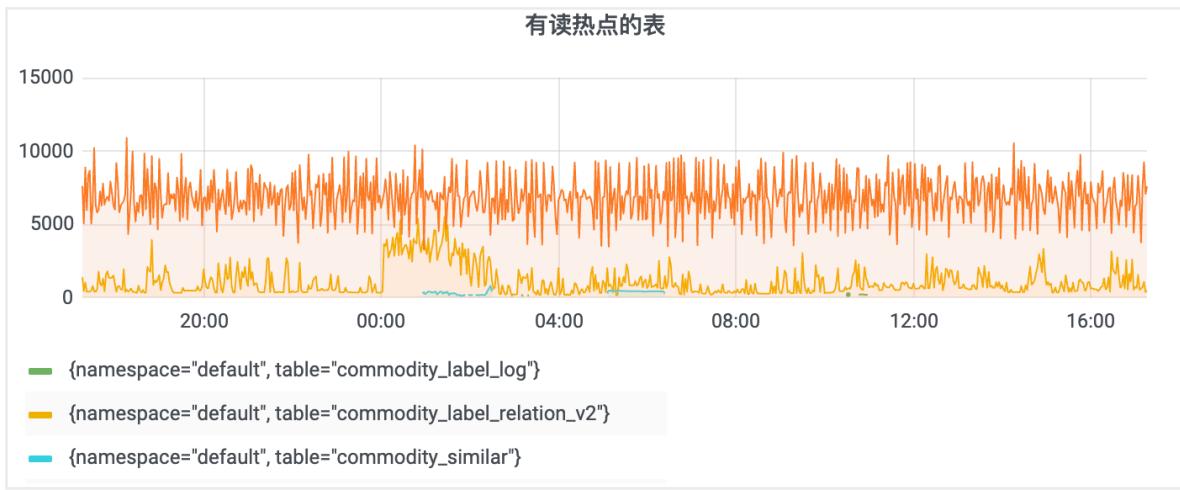
Redisson 框架实现的分布式 redis 锁，通过大量 lua 脚本加锁【lua 脚本可以保证加锁操作的各种原子性，将 Redis 当做一整个脚本执行】，并且在执行过程中设置了一个监听器在监听当前正在执行的任务，如果锁的过期时间 1/3 过去了还没释放锁则会重新 refresh 锁的时间，保证了由于慢 sql 等程序执行时间过长而出现 redis 的 key 自动过期删除。

当然续命锁有一个延长时间的次数限制，如果超过设定的次数还是失败，就会自动释放锁，并可能回滚业务。这样设计是为了防止因某些原因（如代码 bug、网络不稳定等）导致锁一直不能释放，而造成资源的长期占用或其他潜在问题。

## HBase

111



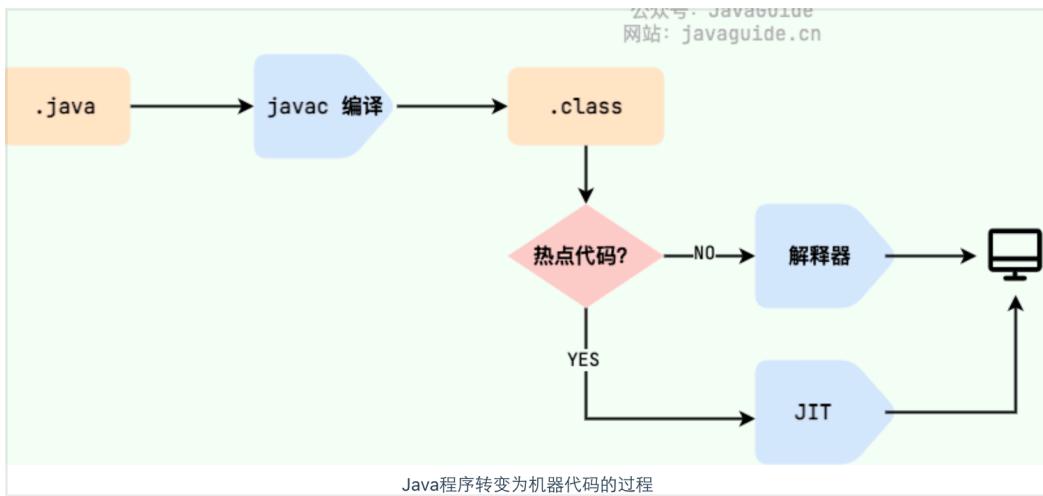


## JAVA 基础

### 为什么说 Java 语言“编译与解释并存”？

先编译转成字节码 (.class 文件)，再通过解释器解释字节码，如果是热点代码则由 JIT 运行时编译器将字节码的机器码保存到机器本地上。所以说 JAVA 是编译与解释共存的语言【运行效率来说机器码速度远快于解释器】

-- 无论是解释器还是编译器，都是为了把代码翻译成机器可理解的代码



## 基本类型和包装类型的区别

- 用途：除了定义一些常量和局部变量之外，我们在其他地方比如方法参数、对象属性中很少会使用基本类型来定义变量。并且，包装类型可用于泛型，而基本类型不可以。
- 存储方式：基本数据类型的局部变量存放在 Java 虚拟机栈中的局部变量表中，基本数据类型的成员变量（未被 `static` 修饰）存放在 Java 虚拟机的堆中。包装类型属于对象类型，我们知道几乎所有对象实例都存在于堆中。
- 占用空间：相比于包装类型（对象类型），基本数据类型占用的空间往往非常小。
- 默认值：成员变量包装类型不赋值就是 `null`，而基本类型有默认值且不是 `null`。
- 比较方式：对于基本数据类型来说，`==` 比较的是值。对于包装数据类型来说，`==` 比较的是对象的内存地址。所有整型包装类对象之间值的比较，全部使用 `equals()` 方法。

## 基本类型和包装类的使用异同

- 在 Java 中，一切皆对象，但八大基本类型却不是对象。
- 声明方式的不同，基本类型无需通过 `new` 关键字来创建，而封装类型需 `new` 关键字。
- 存储方式及位置的不同，基本类型是直接存储变量的值保存在堆栈中能高效的存取，封装类型需要通过引用指向实例，具体的实例保存在堆中。
- 初始值的不同，封装类型的初始值为 `null`，基本类型的初始值视具体的类型而定，比如 `int` 类型的初始值为 0，`boolean` 类型为 `false`；
- 使用方式的不同：如与集合类合作使用时只能使用包装类型。

**为什么说是几乎所有对象实例都存在于堆中呢？**这是因为 HotSpot 虚拟机引入了 JIT 优化之后，会对对象进行逃逸分析，如果发现某一个对象并没有逃逸到方法外部，那么就可能通过标量替换来实现栈上分配，而避免堆上分配内存

## 包装类型的缓存机制

Byte, Short, Integer, Long 这 4 种包装类默认创建了数值 [-128, 127] 的相应类型的缓存数据, Character 创建了数值在 [0,127] 范围的缓存数据, Boolean 直接返回 True or False。

如果超出对应范围仍然会去创建新的对象, 缓存的范围区间的大小只是在性能和资源之间的权衡。

两种浮点数类型的包装类 **Float, Double** 并没有实现缓存机制。

PS: == 比较地址值, equals 比较对象里面的值。

```
1 Integer i1 = 33;
2 Integer i2 = 33;
3 System.out.println(i1 == i2); // 输出 true
4
5 Float i11 = 333f;
6 Float i22 = 333f;
7 System.out.println(i11 == i22); // 输出 false
8
9 Double i3 = 1.2;
10 Double i4 = 1.2;
11 System.out.println(i3 == i4); // 输出 false
```

java

【INTEGER: [-128, 127] 不在这个范围的都是 new 出来的对象, == 是比较地址值, 所以超过这个范围的 == 比较结果为 false】

【float&double 的 == 比较永远为 false, 因为没有缓存机制, 都是 new 出来的对象】

【本质上, 包装类型的缓存机制是通过自己 new 了一个长度为  $127 - (-128) + 1$  的数组, 然后把这个范围的数字都塞到数组里 (数组存在堆上)。在这个范围 get 到的数字都属于基本数据类型, 因此上图的  $33 == 33$  为 true, 因为基本数据类型的 == 比较的是值。如果超过 127 则会 new 一个对象出来, == 比较的是地址值, 自然比较结果为 false】

## 自动装箱与拆箱

- **装箱:** 将基本类型用它们对应的引用类型包装起来【调用了 包装类的 valueOf() 方法】
- **拆箱:** 将包装类型转换为基本数据类型【调用了 xxxValue() 方法】

因此

```
Integer i = 10 等价于 Integer i = Integer.valueOf(10)
int n = i 等价于 int n = i.intValue();
```

---

## BigDecimal 【金钱为什么用 BigDecimal 类型表示?】

计算机是二进制的, 在表示一个数字时, 宽度是有限的, 无限循环的小数存储在计算机时, 只能被截断, 所以就会导致小数精度发生损失的情况。这也就是解释了为什么浮点数没有办法用二进制精确表示【就比如说十进制下的 0.2 就没办法精确转换成二进制小数】。

推荐 **BigDecimal(String val)** 构造方法, 而用 **double** 构造方法还是会有精度丢失

问题。

我们在使用 `BigDecimal` 时，为了防止精度丢失，推荐使用它的 `BigDecimal(String val)` 构造方法或者 `BigDecimal.valueOf(double val)` 静态方法来创建对象。

《阿里巴巴 Java 开发手册》对这部分内容也有提到，如下图所示。

## 12. 【强制】禁止使用构造方法 `BigDecimal(double)` 的方式把 `double` 值转化为 `BigDecimal` 对象。

**说明：**`BigDecimal(double)` 存在精度损失风险，在精确计算或值比较的场景中可能会导致业务逻辑异常。

如：`BigDecimal g = new BigDecimal(0.1F);` 实际的存储值为：0.10000000149

**正例：**优先推荐入参为 `String` 的构造方法，或使用 `BigDecimal` 的 `valueOf` 方法，此方法内部其实执行了 `Double` 的 `toString`，而 `Double` 的 `toString` 按 `double` 的实际能表达的精度对尾数进行了截断。

```
BigDecimal recommend1 = new BigDecimal("0.1");
BigDecimal recommend2 = BigDecimal.valueOf(0.1);
```

The screenshot shows an IDE interface with Java code. The code defines a main method that creates two `BigDecimal` objects, `a` and `b`, from the string "2.55". It then prints them to the console and calculates the difference between them.

```
121 public static void main(String[] args) {
122     BigDecimal a = new BigDecimal(val: 2.55);
123     BigDecimal b = new BigDecimal(val: "2.55");
124     System.out.println(a);
125     System.out.println(b);
126     System.out.println(0.2f - 0.19f);
127 }
```

The output window shows the results:

```
Run: BatchSaveLabelHandler
/Libary/Java/JavaVirtualMachines/jdk1.8.0_241.jdk/Contents/Home/bin/java ...
2.5499999999999982236431605997495353221893310546875
2.55
0.010000005
```

**底层原理**【带小数的整数放大 N 倍转成整数进行计算，再按照设置保留几位小数进行截取】

- 十进制整数在转化为二进制数时不会有精度问题，所以将十进制小数扩大 N 倍让它在整数维度上进行计算 (`BigInteger` 类型)，并记录小数点位置即可；
- `BigDecimal` 进行运算时分解为两部分，`BigInteger` 间的计算，以及小数点位置 `scale` 的更新；例如加法运算时，先按整数计算，然后更新小数点的位置为两者小数位小的那个值

## BigDecimal 等值比较问题

**【强制】**`BigDecimal` 的等值比较应使用 `compareTo()` 方法，而不是 `equals()` 方法。

**说明：**`equals()` 方法会比较值和精度 (1.0 与 1.00 返回结果为 `false`)，而 `compareTo()` 则会忽略精度。

## 变量

### 成员变量与局部变量的区别

- 语义形式：从语义形式上看，成员变量是属于类的，而局部变量是在代码块或方

法中定义的变量或是方法的参数；成员变量可以被 public,private,static 等修饰符所修饰，而局部变量不能被访问控制修饰符及 static 所修饰；但是，成员变量和局部变量都能被 final 所修饰。

- **存储方式**：从变量在内存中的存储方式来看，如果成员变量是使用 static 修饰的，那么这个成员变量是属于类的，如果没有使用 static 修饰，这个成员变量是属于实例的。而对象存在于堆内存，局部变量则存在于栈内存。
- **生存时间**：从变量在内存中的生存时间上看，成员变量是对象的一部分，它随着对象的创建而存在，而局部变量随着方法的调用而自动生成，随着方法的调用结束而消亡。
- **默认值**：从变量是否有默认值来看，成员变量如果没有被赋初始值，则会自动以类型的默认值而赋值（一种情况例外：被 final 修饰的成员变量也必须显式地赋值），而局部变量则不会自动赋值【局部变量没有给默认值时编译会报错】。

### 为什么局部变量需要显示赋值而成员变量不需要

- 成员变量其赋值和取值的顺序具有不确定性，可以在方法调用前赋值，也可以在方法调用后赋值，其具体顺序是在运行时发生的，编译器确定不了；
  - 而局部变量，赋值、取值顺序确定。所以局部变量赋值使用，是一种设计约束，可以减少犯错的可能性。同时局部变量存放到栈帧的局部变量表中，可控的赋值和销毁有利于局部变量表空间复用；
- 
- 

### 静态变量

静态变量也就是被 static 关键字修饰的变量。它可以被类的所有实例共享，无论一个类创建了多少个对象，它们都共享同一份静态变量。也就是说，静态变量只会被分配一次内存，即使创建多个对象，这样可以节省内存【静态变量是通过类名来访问的】。

---

---

### 静态方法为什么不能调用非静态成员？

这个需要结合 JVM 的相关知识，主要原因如下：

- 静态方法是属于类的，在类加载的时候就会分配内存，可以通过类名直接访问。而非静态成员属于实例对象，只有在对象实例化之后才存在，需要通过类的实例对象去访问。
  - 在类的非静态成员不存在的时候静态方法就已经存在了，此时调用在内存中还不存在的非静态成员，属于非法操作。
- 
- 

### 静态方法和实例方法有何不同？

#### 调用方式

在外部调用静态方法时，可以使用 类名.方法名 的方式，也可以使用 对象.方法名 的

方式，而实例方法只有后面这种方式。也就是说，调用静态方法可以无需创建对象。

### 访问类成员是否存在限制

静态方法在访问本类的成员时，只允许访问静态成员（即静态成员变量和静态方法），不允许访问实例成员（即实例成员变量和实例方法），而实例方法不存在这个限制【毕竟此时实例成员还未创建】。

---

---

### 重载和重写

- 重载：发生在同一个类中（或者父类和子类之间），方法名必须相同，参数类型不同、个数不同、顺序不同，方法返回值和访问修饰符可以不同。
  - 重写：重写发生在运行期，是子类对父类的允许访问的方法的实现过程进行重新编写。
- 
- 

## 面向对象三大特征

### 封装

封装是指把一个对象的状态信息（也就是属性）隐藏在对象内部，不允许外部对象直接访问对象的内部信息。但是可以提供一些可以被外界访问的方法来操作属性【例如 Student 类提供了 getName 方法】。

### 继承

不同类型的对象，相互之间经常有一定数量的共同点。

- 子类拥有父类对象所有的属性和方法（包括私有属性和私有方法），但是父类中的私有属性和方法子类是无法访问，只是拥有。
- 子类可以拥有自己属性和方法，即子类可以对父类进行扩展。
- 子类可以用自己的方式实现父类的方法。

### 多态

多态，顾名思义，表示一个对象具有多种的状态，具体表现为父类的引用指向子类的实例。

- 对象类型和引用类型之间具有继承（类）/实现（接口）的关系；
  - 引用类型变量发出的方法调用的到底是哪个类中的方法，必须在程序运行期间才能确定；
  - 多态不能调用“只在子类存在但在父类不存在”的方法；
  - 如果子类重写了父类的方法，真正执行的是子类覆盖的方法，如果子类没有覆盖父类的方法，执行的是父类的方法。
- 
-

## 深拷贝和浅拷贝区别了解吗？什么是引用拷贝？

### 浅拷贝【通过实现 Colonebale】

浅拷贝是按位拷贝对象，它会创建一个新对象，这个对象有着原始对象属性值的一份精确拷贝。如果属性是基本类型，拷贝的就是基本类型的值；如果属性是内存地址（引用类型），拷贝的就是内存地址，因此如果其中一个对象改变了这个地址，就会影响到另一个对象。【修改基本类型的值，不会相互影响；修改内存地址里的属性则相互影响，因为是引用类型，并不是像深拷贝一样把内存地址对应的对象也拷贝了一份，只是拷贝了地址值而已】

#### 适用场景

- 当源对象的属性全为基本数据类型或者不可变对象，并且不需要复制引用类型的属性时，可以使用浅拷贝。
- 当希望修改副本对象的属性同时影响原始对象时，可以使用浅拷贝。

#### 工作原理

- 浅拷贝只复制对象及其引用，而不复制引用指向的实际对象，新旧对象将共享同一个引用对象。修改副本对象会影响原始对象。

#### 实现方式

- 通常使用对象的 `clone()` 方法来进行浅拷贝。

#### 示例场景

- 快速创建对象副本，以便在某些操作中对其进行修改，同时保留原始对象。
- 在某些情况下，共享一部分数据以节省内存和提高性能。

### 深拷贝【通过序列化】

- 深拷贝指在克隆操作中，除了复制对象本身以及对象内部的基本数据类型的属性外，还要递归地复制对象内部的引用类型的属性。即深度克隆了所有引用类型的属性。
- 深拷贝创建了一个完全独立的新对象，该对象与原始对象没有任何关联，对新对象和原始对象的修改互不影响。
- 在深拷贝中，新对象和原始对象分别对应不同的内存区域，它们之间不存在引用关系，因此修改其中一个对象不会影响到另一个对象。

#### 适用场景：

- 当源对象包含引用类型的属性时，如果需要复制对象及其子对象的所有属性，而不仅仅只是复制引用，就需要使用深拷贝。
- 当希望修改副本对象的属性不影响原始对象时，需要使用深拷贝。

#### 工作原理

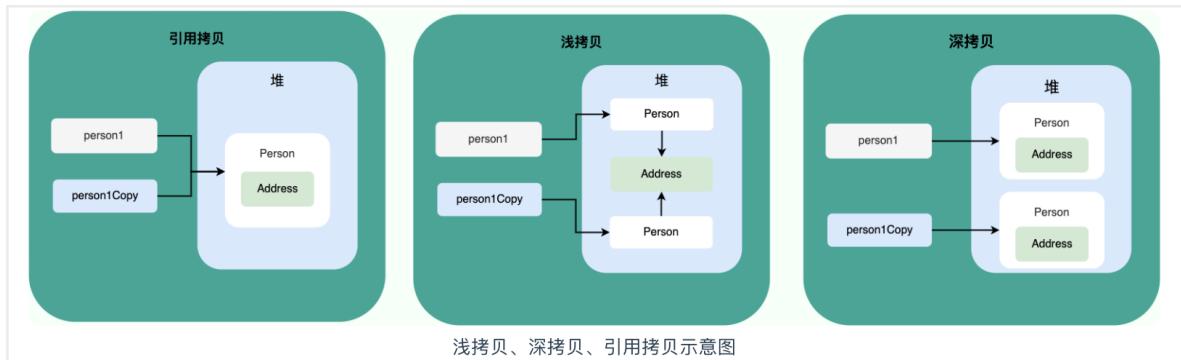
- 深拷贝将源对象及其关联的全部对象进行递归复制，每个对象都拥有独立的内存空间，修改副本对象不会影响原始对象。

#### 实现方式

- 使用递归或者拷贝构造函数来复制对象及其子对象的属性。

## 示例场景

- 复制复杂对象的副本，使其成为独立的个体，例如：拷贝一个包含集合、嵌套对象等的数据结构。
- 对象图的克隆，当原对象包含子对象，并且对子对象的修改不应该影响原对象时。



## 总结

**浅拷贝：**浅拷贝会在堆上创建一个新的对象（区别于引用拷贝的一点），不过，如果原对象内部的属性是引用类型的话，浅拷贝会直接复制内部对象的引用地址，也就是说**拷贝对象和原对象共用同一个内部对象**。

**深拷贝：**深拷贝会完全复制整个对象，包括这个对象所包含的内部对象。

## String 为什么是不可变的？

1. 保存字符串的数组被 final 修饰且为私有的，并且 String 类没有提供 / 暴露修改这个字符串的方法。
2. String 类被 final 修饰导致其不能被继承，进而避免了子类破坏 String 不可变。

String 不可变的本质是因为里面用 private 和 final 去修饰，且不提供 set / get 方法，这样也无法通过继承等方式防止被修改数据【因为里面还是用数组存储 byte 类型的数据，只是 String 把修改数组的渠道给封死了。所以 String 不可变关键在于底层实现，而不是 final 决定的】。

### String 不可变的好处主要有以下几点：

- 安全性：由于字符串在很多情况下用作敏感信息的存储（如用户名、密码等），将其设计为不可变可以确保数据的安全性。这样一来，其他部分的代码无法修改已经创建的 String 对象，从而避免了敏感信息被篡改的风险。
- 线程安全：由于 String 对象是不可变的，多个线程在访问同一个 String 对象时，不会出现数据不一致的问题。这使得 String 在多线程环境中能够安全地共享，无需额外的同步措施。
- 散列值缓存：字符串常用作散列表（如 HashMap）的键。将 String 设计为不可变的，使得可以缓存其散列值（hash code）。当创建 String 对象时，会计算其散列值，并在后续操作中重用这个值，从而提高散列表的查询性能。如果

String 对象是可变的，那么在每次修改后都需要重新计算散列值，导致性能降低。

- 减少内存占用：不可变字符串可以被安全地共享，这可以减少内存占用。Java 虚拟机 (JVM) 维护了一个字符串常量池 (String Constant Pool)，其中存储了所有字符串字面量。当创建一个新的 String 对象时，JVM 会首先检查字符串常量池中是否存在相同的字符串。如果存在，就返回该字符串的引用；否则，创建一个新的 String 对象。这种共享机制可以减少内存占用，提高性能。
- 
- 

## 字符串常量池的作用

字符串常量池 是 JVM 为了提升性能和减少内存消耗针对字符串 (String 类) 专门开辟的一块区域，主要目的是为了避免字符串的重复创建。

```
1 // 在堆中创建字符串对象"ab"
2 // 将字符串对象"ab"的引用保存在字符串常量池中
3 String aa = "ab";
4 // 直接返回字符串常量池中字符串对象"ab"的引用
5 String bb = "ab";
6 System.out.println(aa==bb); // true
```

java

## String s1 = new String("abc");这句话创建了几个字符串对象？

会创建 1 或 2 个字符串对象。

1. 如果字符串常量池没有，则会在常量池创建对象
  2. 由于 new String("abc") 肯定会在堆创建一个对象【所以至少创建1个】，如果字符串常量池存在“abc”，那么会指向常量池的引用
- 
- 

## Java 反射机制

优点：可以让我们的代码更加灵活、为各种框架提供开箱即用的功能提供了便利。

缺点：增加了安全问题，比如可以无视泛型参数的安全检查（泛型参数的安全检查发生在编译时）。另外，反射的性能也要稍差点，不过，对于框架来说实际是影响不大的。

应用场景：JDK 动态代理、注解【@component、@value】

<https://javaguide.cn/java/basis/reflection.html>

---

---

# SPI

## 概念

Java SPI (Service Provider Interface) 是 Java 官方提供的一种服务发现机制，它允许在运行时动态地加载实现特定接口的类，而不需要在代码中显式地指定该类，从而实现解耦和灵活性。

## 实现原理

Java SPI 的实现原理基于 Java 类加载机制和反射机制。

当使用 `ServiceLoader.load(Class<T> service)` 方法加载服务时，会检查 `META-INF/services` 目录下是否存在以接口全限定名命名的文件。如果存在，则读取文件内容，获取实现该接口的类的全限定名，并通过 `Class.forName()` 方法加载对应的类。

在加载类之后，`ServiceLoader` 会通过反射机制创建对应类的实例，并将其缓存起来。

总的来说，Java SPI 的实现原理比较简单，利用了 Java 类加载和反射机制，提供了一种轻量级的插件化机制，可以很方便地扩展功能。

## 优缺点

### 优点

- 松耦合性：SPI 具有很好的松耦合性，应用程序可以在运行时动态加载实现类，而无需在编译时将实现类硬编码到代码中。
- 扩展性：通过 SPI，应用程序可以为同一个接口定义多个实现类。这使得应用程序更容易扩展和适应变化。
- 易于使用：使用 SPI，应用程序只需要定义接口并指定实现类的类名，即可轻松地使用新的服务提供者。

### 缺点

- 配置较麻烦：SPI 需要在 `META-INF/services` 目录下创建配置文件，并将实现类的类名写入其中。这使得配置相对较为繁琐。
- 安全性不足：SPI 提供者必须将其实现类名称写入到配置文件中，因此如果未正确配置，则可能存在安全风险。
- 性能损失：每次查找服务提供者都需要重新读取配置文件，这可能会增加启动时间和内存开销。

## 应用场景

Java SPI 机制是一种服务提供者发现的机制，适用于需要在多个实现中选择一个进行使用的场景。

应用名称	具体应用场景
数据库驱动程序加载	JDBC 为了实现 可插拔 的数据库驱动，在Java.sql.Driver接口中定义了一组标准的API规范，而具体的数据库厂商则需要实现这个接口，以提供自己的数据库驱动程序。在Java中，JDBC驱动程序的加载就是通过SPI机制实现的。
日志框架的实现	流行的 开源日志框架，如 Log4j、SLF4J和Logback 等，都采用了SPI机制。用户可以根据自己的需求选择合适的日志实现，而不需要修改代码。
Spring框架	Spring框架 中的Bean加载机制就使用了SPI思想，通过读取classpath下的META-INF/spring.factories文件来 加载各种自定义的 Bean。
Dubbo框架	Dubbo框架 也使用了SPI思想，通过接口 注解@SPI 声明扩展点接口，并在classpath下的META-INF/dubbo目录中提供实现类的配置文件，来实现扩展点的动态加载。
MyBatis框架	MyBatis框架 中的插件机制也使用了SPI思想，通过在classpath下的META-INF/services目录中存放插件接口的实现类路径，来实现 插件的加载和执行。
Netty框架	Netty框架 也使用了SPI机制，让用户可以根据自己的需求选择合适的 网络协议实现方式。
Hadoop框架	Hadoop框架 中的输入输出格式也使用了SPI思想，通过在classpath下的META-INF/services目录中存放输入输出格式接口的实现类路径，来实现 输入输出格式的灵活配置和切换。

## Spring SPI VS Java原生 SPI

Spring的SPI机制相对于Java原生的SPI机制进行了改造和扩展，主要体现在以下几个方面：

- 支持多个实现类：Spring的SPI机制允许为同一个接口定义多个实现类，而Java原生的SPI机制只支持单个实现类。这使得在应用程序中使用Spring的SPI机制更加灵活和可扩展。
- 支持自动装配：Spring的SPI机制支持自动装配，可以通过将实现类标记为Spring组件（例如@Component），从而实现自动装配和依赖注入。这在一定程度上简化了应用程序中服务提供者的配置和管理。
- 支持动态替换：Spring的SPI机制支持动态替换服务提供者，可以通过修改配置文件或者其他方式来切换服务提供者。而Java原生的SPI机制只能在启动时加载一次服务提供者，并且无法在运行时动态替换。
- 提供了更多扩展点：Spring的SPI机制提供了很多扩展点，例如BeanPostProcessor、BeanFactoryPostProcessor等，可以在服务提供者初始化和创建过程中进行自定义操作。

## SPI和API区别

API：实现方提供接口和封装实现，调用方只和实现方提供的接口打交道。

SPI：调用方确定接口规则【所有的规则都存储在 META-INF/services 目录下，有点像多态父类的引用指向子类的实例】，不同的厂商去根据这个规则对这个接口进行实现，从而提供服务。

## Java 序列化详解

- 序列化：将数据结构或对象转换成二进制字节流的过程
- 反序列化：将在序列化过程中所生成的二进制字节流转换成数据结构或者对象的过程

### 为什么要序列化和作用：

序列化最终的目的是为了对象可以跨平台存储，和进行网络传输。而我们进行跨平

台存储和网络传输的方式就是 IO，而我们的 IO 支持的数据格式就是字节数组。因为我们单方面的只把对象转成字节数组还不行，因为没有规则的字节数组我们是没办法把对象的本来面目还原回来的，所以我们必须在把对象转成字节数组的时候就制定一种规则（序列化），那么我们从 IO 流里面读出数据的时候再以这种规则把对象还原回来（反序列化）。

如果我们要把一栋房子从一个地方运输到另一个地方去，序列化就是我把房子拆成一个个的砖块放到车子里，然后留下一张房子原来结构的图纸，反序列化就是我们把房子运输到了目的地以后，根据图纸把一块块砖头还原成房子原来面目的过程

---

## JAVA IO

### 简介

IO 即 Input/Output，输入和输出。数据输入到计算机内存的过程即输入，反之输出到外部存储（比如数据库，文件，远程主机）的过程即输出。数据传输过程类似于水流，因此称为 IO 流。IO 流在 Java 中分为输入流和输出流，而根据数据的处理方式又分为字节流和字符流。

Java IO 流的 40 多个类都是从如下 4 个抽象类基类中派生出来的。

- `InputStream/Reader`: 所有的输入流的基类，前者是字节输入流，后者是字符输入流。
- `OutputStream/Writer`: 所有输出流的基类，前者是字节输出流，后者是字符输出流。

### 作用

Java IO 用在 Java 程序和外部进行数据交互，Java 程序运行在内存中，要与外部（如：磁盘、网络、数据库等）地方交互数据则需要使用 Java IO。

比如：在本地磁盘的某个 txt 文件上读写运行日志、读写 MySQL 数据库的内容等。

---

## JAVA IO 模型详解

<https://javaguide.cn/java/io/io-model.html#aio-asynchronous-i-o>

PS: NIO 用的也是 IO 多路复用

---

## NIO

### 简介

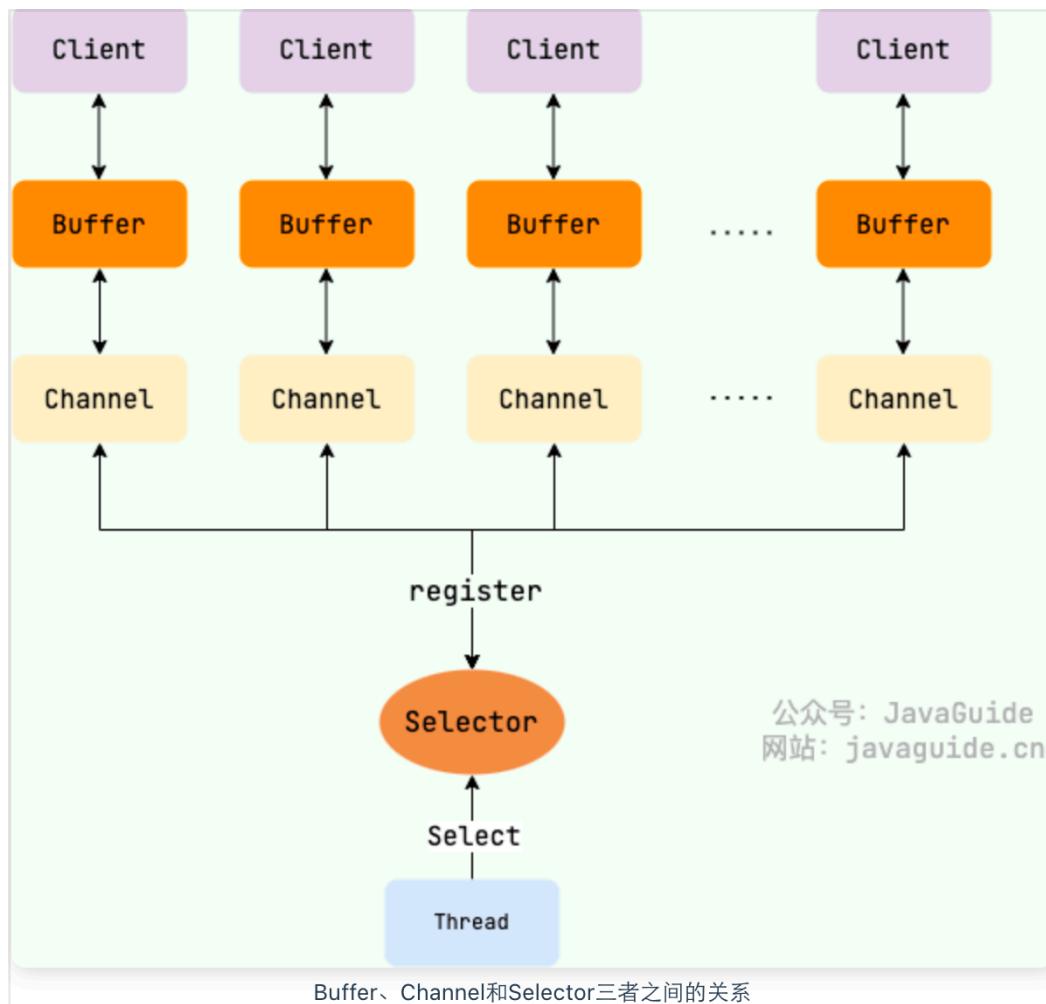
在传统的 Java I/O 模型（BIO）中，I/O 操作是以阻塞的方式进行的。也就是说，当一个线程执行一个 I/O 操作时，它会被阻塞直到操作完成。这种阻塞模型在处理多个并发连接时可能会导致性能瓶颈，因为需要为每个连接创建一个线程，而线程的创建和切换都是有开销的。

为了解决这个问题，在 Java 1.4 版本引入了一种新的 I/O 模型 — **NIO**（New IO，也称为 Non-blocking IO）。NIO 弥补了同步阻塞 I/O 的不足，它在标准 Java 代码中

提供了非阻塞、面向缓冲、基于通道的 I/O，可以使用少量的线程来处理多个连接，大大提高了 I/O 效率和并发。

## 核心组件

- **Buffer (缓冲区)**: NIO 读写数据都是通过缓冲区进行操作的。读操作的时候将 Channel 中的数据填充到 Buffer 中，而写操作时将 Buffer 中的数据写入到 Channel 中。
- **Channel (通道)**: Channel 是一个双向的、可读可写的 data 传输通道，NIO 通过 Channel 来实现数据的输入输出。通道是一个抽象的概念，它可以代表文件、套接字或者其他数据源之间的连接。
- **Selector (选择器)**: 允许一个线程处理多个 Channel，基于事件驱动的 I/O 多路复用模型。所有的 Channel 都可以注册到 Selector 上，由 Selector 来分配线程来处理事件。



## 应用场景

NIO 的应用场景非常广泛，特别适用于需要处理大量并发连接的网络编程和高性能服务器开发。以下是一些常见的 NIO 应用场景：

- **网络编程**: NIO 提供了非阻塞的网络 I/O 操作，使得可以在一个线程中同时处理多个连接。这对于需要处理大量并发连接的服务器应用非常有用，例如聊天服务器、即时通讯服务器、游戏服务器等。
- **高并发服务器**: NIO 的非阻塞 I/O 模型使得服务器能够高效地处理大量并发请求。通过使用选择器 (Selector) 和多路复用，可以在一个线程中同时处理多个

通道的 I/O 操作，提高了服务器的并发性能。

- 文件 I/O：NIO 提供了对文件的高效读写操作。通过使用通道 (Channel) 和缓冲区 (Buffer)，可以实现快速的文件读写，适用于需要处理大型文件的应用，例如日志处理、文件传输等。
- 数据库操作：NIO 可以与数据库进行高效的交互。通过使用 NIO 的通道和缓冲区，可以实现快速的数据库读写操作，提高了数据库操作的效率。
- 多媒体处理：NIO 可以用于处理多媒体数据，例如音频、视频等。通过使用通道和缓冲区，可以高效地读取和写入多媒体数据，适用于音视频处理、流媒体传输等应用。

NIO 适用于需要处理大量并发连接、高性能服务器、文件 I/O、数据库操作和多媒体处理等场景。它提供了高效的非阻塞 I/O 操作方式，能够提升系统的并发性能和吞吐量。

如果我们需要使用 NIO 构建网络程序的话，不建议直接使用原生 NIO，编程复杂且功能性太弱，推荐使用一些成熟的基于 NIO 的网络编程框架比如 **Netty**。Netty 在 NIO 的基础上进行了一些优化和扩展比如支持多种协议、支持 SSL/TLS 等等。

---

## Java 语法糖

### 简介

语法糖 (**Syntactic Sugar**) 也称糖衣语法，是英国计算机学家 Peter.J.Landin 发明的一个术语，指在计算机语言中添加的某种语法，这种语法对语言的功能并没有影响，但是更方便程序员使用。简而言之，语法糖让程序更加简洁，有更高的可读性。

语法糖的存在主要是方便开发人员使用。但其实，**Java 虚拟机并不支持这些语法糖**。这些语法糖在编译阶段就会被还原成简单的基础语法结构，这个过程就是解语法糖【编译器解糖为 JVM 能读懂的字节码，所以语法糖主要在编译层面去做处理】。

说到编译，大家肯定都知道，Java 语言中，javac 命令可以将后缀名为.java 的源文件编译为后缀名为.class 的可以运行于 Java 虚拟机的字节码。如果你去看 com.sun.tools.javac.main.JavaCompiler 的源码，你会发现在 compile() 中有一个步骤就是调用 desugar()，这个方法就是负责解语法糖的实现的。

### 常见语法糖

- switch 支持 String 与枚举
- 泛型
- 自动装箱和拆箱
- 可变长参数
- 枚举
- 内部类
- 条件编译
- 断言
- ...

<https://javaguide.cn/java/basis/syntactic-sugar.html>

---

## Java 值传递

Java 中将实参传递给方法 (或函数) 的方式是 **值传递**:

- 如果参数是基本类型的话，传递的就是基本类型的字面量值的拷贝，会创建副本【修改形参会不影响实参，因为只是操作形参的拷贝副本】。
- 如果参数是引用类型，传递的就是实参所引用的对象在堆中地址值的拷贝，同样也会创建副本【修改形参等于修改地址对应在堆里的数据，所以会影响实参的结果】。

<https://javaguide.cn/java/basis/why-there-only-value-passing-in-java.html>

---

---

## Java 代理模式

代理模式的主要作用是扩展目标对象的功能，比如说在目标对象的某个方法执行前后你可以增加一些自定义的操作。

### 静态代理和动态代理的对比

1. **灵活性**: 动态代理更加灵活，不需要必须实现接口，可以直接代理实现类，并且可以不需要针对每个目标类都创建一个代理类。另外，静态代理中，接口一旦新增加方法，目标对象和代理对象都要进行修改，这是非常麻烦的！
2. **JVM 层面**: 静态代理在编译时就将接口、实现类、代理类这些都变成了一个个实际的 class 文件。而动态代理是在运行时动态生成类字节码，并加载到 JVM 中的。

### JDK 动态代理和 CGLIB 动态代理对比

1. **JDK 动态代理只能代理实现了接口的类或者直接代理接口，而 CGLIB 可以代理未实现任何接口的类**。另外，CGLIB 动态代理是通过生成一个被代理类的子类来拦截被代理类的方法调用，因此不能代理声明为 final 类型的类和方法。
  2. 就二者的效率来说，大部分情况都是 JDK 动态代理更优秀，随着 JDK 版本的升级，这个优势更加明显
- 

---

## JAVA 集合

---

# ArrayList 源码分析

<https://javaguide.cn/java/collection/arraylist-source-code.html>

## ArrayList 与 LinkedList 区别?

1. 是否保证线程安全: 都线程不安全;
2. 底层数据结构: ArrayList 底层使用的是 **Object** 数组; LinkedList 底层使用的是 双向链表 数据结构
3. 插入和删除是否受元素位置的影响:
  - ArrayList 采用数组存储, 所以插入和删除元素的时间复杂度受元素位置的影响。比如: 执行 add(E e) 方法的时候, ArrayList 会默认在将指定的元素追加到此列表的末尾, 这种情况时间复杂度就是 O(1)。但是如果要在指定位置 i 插入和删除元素的话 (add(int index, E element)), 时间复杂度就为 O(n)。因为在进行上述操作的时候集合中第 i 和第 i 个元素之后的 (n-i) 个元素都要执行向后位/向前移一位的操作。
  - LinkedList 采用链表存储, 所以在头尾插入或者删除元素不受元素位置的影响 (add(E e)、addFirst(E e)、addLast(E e)、removeFirst()、removeLast()), 时间复杂度为 O(1), 如果是要在指定位置 i 插入和删除元素的话 (add(int index, E element), remove(Object o), remove(int index)), 时间复杂度为 O(n), 因为需要先移动到指定位置再插入和删除。
4. 是否支持快速随机访问: LinkedList 不支持, ArrayList 支持
5. 内存空间占用: ArrayList 的空间浪费主要体现在在 list 列表的结尾会预留一定的容量空间, 而 LinkedList 的空间花费则体现在它的每一个元素都需要消耗比 ArrayList 更多的空间 (每个元素都有前后指针)。

## 简介

数组初始大小为 10 (初始化时候有给定值就按照给定的值为数组大小) 实际上说现在初始值是 0, 只有调用了 add 方法才会初始化成 10, 没有 volatile 修饰, 非线程安全, modCount 统计当前数组被修改的版本次数。

三种初始化: 无参数初始化、指定大小初始化、指定初始数据初始化

无参数初始化: 默认大小是空数组, 不是大家常说的 10, 10 是在第一次 add 的时候扩容的初始值。

指定初始数据初始化: 用 elementData 容器保存传入的初始数据 (要转为数组), 并且要保证数据是 object 类型, 如果不是会通过 arrays.copyOf 来转成 object 类型。

## 新增方法

流程: 1、调用 add 方法, 进入 ensureCapacityInternal 方法来确保数组大小是否能容纳当前传入的数据。2、确定数组长度: 判断数组里是否有给定初始值, 没有给定初始值就代表数组长度为 0, 那就初始化大小为 10, 如果给了初始值, 就按照初始值的 size () 方法为数组初始大小。3、判断期望的最小容量是否大于目前数组长度, 大于就进行扩容。4、数组结构变动时版本号 +1, 删除方法也会 +1。5、当前数组不够容纳传入的数组大小时就扩容: 根据当前数组长度创建当前数组 1.5 倍长度的新数组 (按照初始化的数组长度和传入的元素大小比对是否进行扩容, 最大长度不超过 Integer 最大值)。6、通过 Arrays.copyOf 方法将数据复制到新数组上 (数组原数据拷贝到新数组)。7、将新增元素放入到新数组。

## 删除方法

不管是批量、根据值还是索引删除等，最主要的关注的方法就是 fastRemove (int index)，与其说是删除，不如说是拷贝数据到数组上，过程是：拷贝当前删除索引的后一个索引开始到最后的长度，然后拷贝到当前索引的位置，最后一个索引的数据设置为 null。所以说每次数组涉及到删除的时候就会移动整个数组，消耗很大。

## for 循环删除数据方式选择

1. for 循环，比如集合里面有两个 3，当你删除第一个的时候，第二个 3 就会移到上一个 3 位置的索引，所以遍历下一个元素的时候会找不到 3，因为 3 已经位置移到前面一个位置去了。导致无法删除干净。

2. foreach：调用个 remove 或者 add 方法会报错  
ConcurrentModificationException。

原理：

- 1、增强 for 底层实际依赖的是 iterator() 实现的，而 expectedModCount 是 ArrayList 中的一个内部类 Itr 中的成员变量【即 Iterator 类的成员变量，而非 ArrayList 的成员变量】，只有通过迭代器对集合进行操作，该值才会改变。
- 2、当在 foreach 中调用 add/remove 实际是直接使用集合类自己的方法【即 List/ArrayList 的 remove、add 方法】，modCount 会 +1，但是却无法同步 expectedModCount。
- 3、foreach 的遍历取数是采用 Iterator.next 方法，这个方法内部会调用 Iterator.checkForComodification 方法，所以会触发校验导致报错。
- 4、但是如果用迭代器 iterator.remove，那么是由迭代器做 remove 不会报错，因为迭代器在修改 modCount+1 的同时会同步 expectedModCount。

总结：foreach 底层是 iterator 作为遍历器的，所以 foreach 里针对元素调 remove 方法实际就是等同于在 iterator 里用集合本身调 remove 方法【即 list.remove】，只会使得 modCount+1，不会同步 expectedModCount。由于

expectedModCount 是 ArrayList 内部类 Iterator 类的成员变量，所以当你采用 Iterator 迭代器的 remove 方法就会同步 modCount 和 expectedModCount，因此不会报错。至于报错触发点在于 iterator.next 方法，这个方法内部会校验两者值是否相等。

所以本质上，无论用 foreach 还是迭代器，大家都是用迭代器遍历的，只不过 foreach 的 remove 方法不会同步 expectedModCount。

百度为您找到以下结果

搜索工具

### AI智能回答



不会

使用迭代器遍历元素时不会出现漏删的情况。迭代器内部会维护一个 `modCount` 变量，记录集合被修改的次数。当使用迭代器的 `remove` 方法删除元素时，会更新 `modCount` 的值，从而确保遍历的完整性和安全性，避免 `ConcurrentModificationException` 异常的发生 1。

### 迭代器的工作原理

迭代器在删除元素时，会更新内部的一个 `cursor` 变量，确保在删除元素后继续遍历时能够从正确的位置开始。这种机制避免了因删除操作导致的索引变化问题，从而确保了遍历的完整性 2 3。

### 对比普通for循环

使用普通的for循环遍历并删除元素时，由于删除操作会导致数组或列表的元素前移，后续的遍历可能会跳过某些元素，导致漏删。例如，在遍历一个ArrayList时删除元素，由于 ArrayList是动态数组，删除操作会导致后面的元素前移，从而在遍历过程中跳过某些元素 2 4。

综上所述，使用迭代器遍历并删除元素是安全的，不会出现漏删的情况，而普通for循环则可能出现漏删问题。

3.要用 `Iterator.remove`，因为在删除的时候 `modCount` 会同步 `expectedModCount` 数值。

【`list.removeIf` 底层也是用迭代器，并且保证了 `modCount` 和 `expectedModCount` 的同步】

### PS: `modCount` 作用

`modCount` 用于记录结构性改变的次数。所谓结构性改变，指的是那些会修改列表大小的操作，如添加或删除元素等。这个变量主要用于快速失败 (fail-fast) 机制，即当多个线程并发修改同一个列表时，能够尽快地抛出异常，防止数据不一致的问题。

## LinkedList 源码分析

<https://javaguide.cn/java/collection/linkedlist-source-code.html>

## 简介

LinkedList 是一个基于双向链表实现的集合类，不过，我们在项目中一般是不会使用到 LinkedList 的，需要用到 LinkedList 的场景几乎都可以使用 ArrayList 来代替。另外，不要下意识地认为 LinkedList 作为链表就最适合元素增删的场景。我在上面也说了，LinkedList 仅仅在头尾插入或者删除元素的时候时间复杂度近似  $O(1)$ ，其他情况增删元素的平均时间复杂度都是  $O(n)$ 。

## 添加

- 不指定 index 则默认添加到链表尾部
- 指定 index，遍历查找到 index 对应位置的 Node 节点，然后将传入的新数据放入到 Node 节点之后

## 删除

- 不指定 index 默认删除链表第一个节点
  - 指定 index，需要遍历找到数据再移除 Node 节点
- 
- 

# HashMap

## 简介

HashMap 底层是数组 + 链表 + 红黑树的数据结构，数组的主要作用是方便查找，时间复杂度是  $O(1)$ ，默认大小是 16，扩容的大小必须是 2 的幂次方（16、32、64...，这样做减少了哈希碰撞的几率，提高了查询效率），数组的下标索引是通过 key 的 hash() 方法计算出来的，数组元素叫做 Node，当多个 key 的哈希值一致，但 key 值不同时，单个 Node 就会转化为链表，链表的查询时间复杂度  $O(n)$ ，当链表的长度大于等于 8 并且数组的大小超过 64 时，链表就会转为红黑树（即使链表长度达到了 8 如果数组长度没有到 64，不会转成红黑树而是先去扩容 resize() 方法），同样当红黑树的长度小于等于 6 时，红黑树会转化为链表（6 和 8 之间有一个 7，防止频繁链表红黑树转变导致降低效率）。红黑树的查询时间复杂度是  $O(\log(n))$ ，简单来说最坏的查询次数相当于红黑树的最大深度。

## 了解与、或、异或、左移、右移、无符号右移知识

与【&】、或【|】、异或【⊕/^】、左移【<<】、右移【>>】、无符号右移【>>>】运行规则：<https://javaforall.cn/154688.html>

- &： $1 \& 1 = 1$ , 别的都为 0
- |：同数为同数结果，不同数为 1【 $1 | 1 = 1$ ,  $1 | 0 = 1$ ,  $0 | 1 = 1$ ,  $0 | 0 = 0$ 】
- ^：同数为 0，不同数为 1【 $0 \oplus 0 = 0$ ,  $0 \oplus 1 = 1$ ,  $1 \oplus 0 = 1$ ,  $1 \oplus 1 = 0$ 】
- <<：例如  $3 << 2$ , 3 的二进制为 0000 0000 0000 0000 0000 0000 0011【int 位 32 位，高位补 0】，左移两位 0000 0000 0000 0000 0000 0000 0000 0000 1100【整体往左移动两位，低位补 0】，继而结果转成十进制就是 12。简单理解就是 2 代表要乘 2 次 2【 $3 * 2 * 2$ 】

- `>>`: 只是变成了除 2【例如  $16 >> 1 = 8$ 、 $16 >> 2 = 4$ 】，正数高位补 0，负数高位补 1
- `>>>`: 无符号右移，和右移逻辑一样，也是除 2，但是无论是正数还是负数，高位都补 0

## 如何计算 key 的哈希值？

```
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

`h = key.hashCode() ^ (h >>> 16)` 解读以及右移 16 位原因

解读

1. 计算出 key 的哈希值为 h
2. h 对应二进制右移 16 位
3. 【h 的二进制】同【h 的二进制右移 16 位】进行异或操作，将异或结果的 32 位二进制转成十进制就是最终的哈希值

## 为什么右移 16 位？

1. 右移 16 位是为了让高 16 位也参与运算，可以更好的均匀散列，减少碰撞，进一步降低 hash 冲突的几率【key 的 hashCode 方法返回类型是 int 即最大长度为 32，并且根据异或操作中同数为 0，不同数为 1 的特性。当原数据（key 的 hash 转成的二进制）无符号右移后则为 16 个 0+ 原数据前 16 位 异或 原数据，由于原数据前 16 位是同 16 个 0 进行异或操作则保持原样，原数据后 16 位则和位移后的原数据前 16 位进行异或操作，这样就可以让高 16 位也参与了计算，均匀散列减少碰撞】例子如下图

`h = key.hashCode() ^ (h >>> 16)`

0000 0000 0011 0001 1010 1010 0010 0010 ( `h = key.hashCode()` )

`^ (异或)`

0000 0000 0000 0000 0000 0000 0011 0001 (`h >>> 16`)

结果为：

0000 0000 0011 0001 1010 1010 0001 0011

2. 异或运算是为了更好保留两组 32 位二进制数中各自的特征

【& 操作容易抬高 hash 冲突几率，| 操作稍微会向 1 靠近，都会抬高冲突几率】

参考来源：[https://blog.csdn.net/weixin\\_46195957/article/details/125274802](https://blog.csdn.net/weixin_46195957/article/details/125274802)

如何计算出key在map里对应的下标?

```
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, next: null);
    else {
        Node<K,V> e; K k;
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
```

p = tab[i = (n - 1) & hash】  
【hash是key计算出来的, n为数组长度】

例如map长度为16, 即下标i=15&hash【15的二进制同hash的二进制做与&操作, 再转回十进制即为i下标值】

【由于是与&操作, 只有1&1=1, 别的都是0, 所以i最大的值是不会超过15, 与&操作结果只会比参与计算的最小数值还要小】

开始思考: 如果构建map时给定的初始大小非2的n次方, 那么map最终大小怎么计算?

```
Returns a power of two size for the given target capacity.

4 usages
static final int tableSizeFor(int cap) {
    int n = cap - 1;
    n |= n >>> 1;
    n |= n >>> 2;
    n |= n >>> 4;
    n |= n >>> 8;
    n |= n >>> 16;
    return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
}
```

注释: Returns a power of two size for the given target capacity 【返回大于等于入参的最小2的N次幂的数】.

**n |= n >>> 1解释**

n=(n|n>>>1), n和n>>>1进行与操作后赋值给n【先执行>>>再执行|, >>>的操作优先级较高】

剖析流程【例如给的数字为10】

- int n = cap - 1;【10-1=9, 则n为9】

- $n |= n >>> 1$ ;【9的二进制即  $n=1001$ , 右移一位为  $0100$ , 再  $0100|1001=1101$ , 则  $n=1101$ 】
- $n |= n >>> 2$ ;【现在  $n=1101$ , 右移两位为  $0011$ ,  $1101|0011=1111$ , 则  $n=1111$ 】
- $n |= n >>> 4$ ;【现在  $n=1111$ , 右移四位为  $0000$ ,  $1111|0000=1111$ , 则  $n=1111$ 】
- $n |= n >>> 8$ ;【现在  $n=1111$ , 右移八位为  $0000\ 0000$ ,  $0000\ 1111|0000\ 0000=0000\ 1111$ , 则  $n=1111$ 】
- $n |= n >>> 16$ ;【同理得出最后  $n=1111$ 】
- 最后 return 的结果为  $n+1$ 【 $1111+1=10000$ , 转成十进制即为 2 的 4 次方 = 16, 因为前面计算的结果都为  $111\dots111$ , +1 保证了最后数据最小 2 的 N 次幂】

无论初始值给多少, 通过前面 5 轮的右移和与操作结果都会为  $111\dots111$ , 然后再对结果 +1 就会变成初始值的最小 2 的 N 次幂【 $1+2+4+8+16=31$  包含了 int 的最大次方, 也就是对数值从 0~31 次方都进行了洗礼】

参考来源: <https://blog.csdn.net/u014540814/article/details/88354793>

## 为什么 cap-1?

如果传进来的数为正好是 2 的 N 次幂, 这个时候没减 1, 就会得到一个更大的数。例如传入的是 8, 二进制是  $0000\ 1000$ , 右移后【经过上述 5 个步骤的位移】位得到  $0000\ 1111$ , 再加上 1 就是  $0001\ 0000$ , 就是十进制的 16。

## 知道了数组大小一定是 2 的 n 次方, 那么为什么呢?

### 1、降低 hash 碰撞几率

下标计算公式为  $i = (n - 1) \& hash$

- 当  $n=2$  的  $n$  次方:  $n-1$  的二进制为  $111\dots111$ , 和 hash 做  $\&$  操作【因为只有  $1\&1=1$ , 任何数值  $\&1=原值$ 】, 最终结果会按照 hash 二进制为结果呈现。
- 当  $n!=2$  的  $n$  次方:  $n-1$  的二进制不为  $111\dots111$ 【 $1$  和  $0$  交替出现】, 和 hash 做  $\&$  操作更容易出现 0【只有  $1\&1=1$ 】, 容易得到相同的结果, 毕竟  $\&$  操作还有三种情况结果都是 0, 所以不同数值  $\&$  非  $111\dots111$  会有更高概率等到相同的结果, 提高了碰撞概率, 之前做了 key 的 hash 均匀散列优化计算, 经过这种非 2 的  $n$  次方  $\&$  操作优化没了, 但是通过 2 的  $n$  次方可以保留之前 key 的 hash 优化计算【如下图】

16的二进制为:	0001 0000	0000 1111	0000 1111	0000 1111	0000 1111
减一得:	0000 1111	$\& 0000\ 1001$	$\& 0000\ 1101$	$\& 0000\ 1011$	$\& 0000\ 1110$
		0000 1001	0000 1101	0000 1011	0000 1110
如果是11,二进制为0000 1011	0000 1010	0000 1010	0000 1010	0000 1010	0000 1010
减一得:	0000 1010	$\& 0000\ 1001$	$\& 0000\ 1101$	$\& 0000\ 1011$	$\& 0000\ 1110$
		0000 1000	0000 1000	0000 1010	0000 1010

由图可见, 非2的n次幂的  $(n - 1) \& hash$  算法, hash碰撞几率变大

参考来源: [https://blog.csdn.net/cj\\_eryue/article/details/108081173](https://blog.csdn.net/cj_eryue/article/details/108081173)

### 2、提高数组利用率

如果长度为非固定的 2 的  $n$  次方, 例如是长度为 13, 那么长度减一后就变成偶数了, 偶数的二进制个位数肯定为 0, 再经过和 hash 进行  $\&$  操作后, 0  $\&$  任何数都为

0, 那么最终通过&计算出的二进制肯定为偶数【即i必然为偶数】，导致奇数下标永远不会被分配到数据，降低了一半的数组空间利用率，同时也提高了碰撞几率。

### 3、可以直接进行位运算

当根据一个hash计算如何平均放入到长度为n的数组中时，实际是通过取模操作平均的将数据放入到数组里。但是如果非2的n次方，  
hash%length==hash&(length-1)则不成立，例如长度8和7的数组，通过取模和&计算得出的结果是不一样的。但是数组长度是2的n次方话，可以保证取模和&操作的结果都是一样的，所以可以直接用&的位运算取代取模操作【位运算快于取模运算】，不然就要使用hash%length，因为取模操作才能保证数据均匀散列。

## Put方法

- 判断数组是否为空，为空则用resize方法初始化16长度的数组

```
final Node<K,V>[] resize() {
    Node<K,V>[] oldTab = table;
    int oldCap = (oldTab == null) ? 0 : oldTab.length;
    int oldThr = threshold;
    int newCap, newThr = 0;
    if (oldCap > 0) {
        if (oldCap >= MAXIMUM_CAPACITY) {
            threshold = Integer.MAX_VALUE;
            return oldTab;
        }
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                  oldCap >= DEFAULT_INITIAL_CAPACITY)
            newThr = oldThr << 1; // double threshold
    }
    else if (oldThr > 0) // initial capacity was placed in threshold
        newCap = oldThr;
    else { // zero initial threshold signifies using defaults
        newCap = DEFAULT_INITIAL_CAPACITY;
        newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
    }
}
```

- 判断当前索引位置是否有Node节点，如果无的话在当前索引位置构建Node节点并且放入数据

```
if ((p = tab[i = (n - 1) & hash]) == null)
    tab[i] = newNode(hash, key, value, next: null);
```

- 当前索引位置存在数据【即hash碰撞】

p：代表当前索引下的Node节点【默认链表第一个元素】

e：临时变量e，用来保存同key、需要覆盖value的节点

1. p.hash == hash &&((k = p.key) == key || (key != null &&  
key.equals(k)))【判断p节点的第一个元素是否和传入的相等，相等则将p  
的地址指向e】
2. p instanceof TreeNode【如果p节点为红黑树，将数据放入红黑树，返回

值为 null, 所以指针指向 e 也没事, 不会做任何处理】

3. 非 p 的第一个节点, p 也非红黑树结构【说明当前节点链表不止一个数据, 通过 p.next 指针对链表做遍历且将当前遍历的元素地址指向 e。1、next 指针的节点为空->构建新节点 + 判断是否需要转成红黑树, 停止循环, 此时 e 为 null 没有存在同 key 节点。2、next 指针的节点不为空->如果传入的 key 和存在的 key 相等则跳出循环 (for 循环一开始的时候已完成 e=p.next), 否则通过 p=e 继续循环。「p=e, 用于下次循环找到 p 的 next 节点」】

```
else {
    Node<K,V> e; K k;
    if (p.hash == hash &&
        ((k = p.key) == key || (key != null && key.equals(k))))
        e = p;
    else if (p instanceof TreeNode)
        e = ((TreeNode<K,V>)p).putTreeVal( map: this, tab, hash, key, value);
    else {
        for (int binCount = 0; ; ++binCount) {
            if ((e = p.next) == null) {
                p.next = newNode(hash, key, value, next: null);
                if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                    treeifyBin(tab, hash);
                break;
            }
            if (e.hash == hash &&
                ((k = e.key) == key || (key != null && key.equals(k))))
                break;
            p = e;
        }
    }
}
```

- 如果 e 不为空表示在 hash 碰撞下存在同 key 节点, 直接newValue 覆盖旧值。

```
if (e != null) { // existing mapping for key
    V oldValue = e.value;
    if (!onlyIfAbsent || oldValue == null)
        e.value = value;
    afterNodeAccess(e);
    return oldValue;
}
```

- 当 hashmap 实际元素个数大于阈值则进行扩容

```
++modCount;  
if (++size > threshold)  
    resize();  
afterNodeInsertion(evict);  
return null;
```

## Get方法

根据key去找对应下标的数据，查找途径无非根据首节点、链表、红黑树，返回对应的value数据，找不到返回null。

## resize() 扩容方法

### putVal可能会调两次

- 1、数组未初始化，会调用resize() 初始化 map 大小以及阈值【大小\*负载因子】，此时oldTab无数据，不进行任何数据迁移动作，只负责数组和阈值大小初始化。
- 2、**putVal**完后发现长度超过阈值则会进行扩容，将数组长度和阈值>>1翻倍，此时oldTab不为空，遍历oldTab进行数据迁移到newTab，会出现三种情况：
  - 1、当前下标只有一个元素【同新数组长度 (n-1)&hash计算新下标位置放入数据到新数组】
  - 2、当前下标是红黑树【对树进行拆分，重新计算每个元素的下标放入到新数组中去】
  - 3、当前下标不止包含一个元素的链表【由于扩容后链表下的数据有可能属于当前下标，也有可能属于别的位置。例如 hash 为 11011, 同 1111 和 11111 数组长度做&操作的结果不一样，如果 hash 为 101 则还是保持原来下标位置。所以创建两条包含头尾节点的链表（一个代表原来位置的链表，一个代表新位置的链表），通过判断来决定是放入新还是老的链表里面，最后将两个链表的数据分别放入到新老下标去】，所以在扩容后，原来链表的数据无非在老下标或者新下标，如果不用户两个链表就需要对链表的数据一个一个转移到新数组，而用了两个链表，则可以以链表单位直接迁移到新数组中去，提高了迁移的效率。
- 参考博客：[https://blog.csdn.net/weixin\\_44051038/article/details/116496412](https://blog.csdn.net/weixin_44051038/article/details/116496412)

## HashMap 在多线程并发下出现死锁

JDK7时，扩容【trasfer()方法】时通过遍历老数组将数据放入到新数组，如果遍历到的节点是一个链表【即存在哈希冲突，并且例如扩容后冲突的数据还是在同一个节点下】，通过头插法将链表的数据放入到新数组。举例如下：老数组下标3的存在c, b, a三个数据

- T1线程进来：遍历到老数组下标3的节点时，发现是个链表，准备将c, b, a转移到新数组，不过由于cpu时间片用完，暂时被挂起【还没来得及转移数据】。
- T2线程进来：由于头插法，放入到新数组下标为5的节点，此时为a, b, c
- T1线程再次抢到资源开始遍历老数组进行数据转移到新数组，此时线程T1遍历到老数组下标3的位置时，也准备将cba转移到新数组下标5的位置下，此时T1不知道下标5的节点已经存了数据abc，于是开始将老数组下标3的c元素放入到新数组下标5的头节点位置。本来是a->b->c，现在T1线程把c根据头插法变成了c->a->b->c，造成了环形链表，即取值时出现死循环，造成了cpu飙升已经内存溢出。

JDK8时优化了扩容方法，将头插法改成尾插法，保证了老数组的数据顺序和转移到新数组的顺序一致，无非是第二个线程重复第一个线程一模一样的操作，避免了死循环。

## JDK8下的HashMap在多线程并发下会出现什么问题？

- putVal容易丢失数据，当T1线程准备赋值数据A的时候突然CPU时间片用完则挂起，线程T2进来将数据赋值为B，然后T1抢到了CPU资源赋值数据为A，那么最后的数据为A造成B数据的丢失。
- 数组长度在put完后会进行size++，但是这个字段只是用了transient修饰【不参与序列化】并非volatile【线程的工作内存可以实时看到主内存的值变化并且同步回来】，那么多线程情况下，可能使得size结果不准确【T1进来准备将10->11但是突然被挂起，T2进来将10->11，T1抢到CPU资源时还是10->11，理论用size现在为12了】。
- 并发的读写操作可能导致数据不一致：当有线程在进行put或remove操作时，其他线程在进行get操作可能会读取到不一致的数据。这是因为HashMap不提供任何同步机制来保证多线程的可见性和顺序性，导致读取操作可能在并发修改时读取到中间状态的数据。

## 时间片

时间片即CPU分配给各个程序的时间，每个线程被分配一个时间段，称作它的时间片，即该进程允许运行的时间，使各个程序从表面上看是同时进行的。如果在时间片结束时进程还在运行，则CPU将被剥夺并分配给另一个进程。如果进程在时间片结束前阻塞或结束，则CPU当即进行切换。而不会造成CPU资源浪费。在宏观上：我们可以同时打开多个应用程序，每个程序并行不悖，同时运行。但在微观上：由于只有一个CPU，一次只能处理程序要求的一部分，如何处理公平，一种方法就是引入时间片，每个程序轮流执行。

## HashMap面试题

<https://blog.csdn.net/ymb615ymb/article/details/123449195>

---

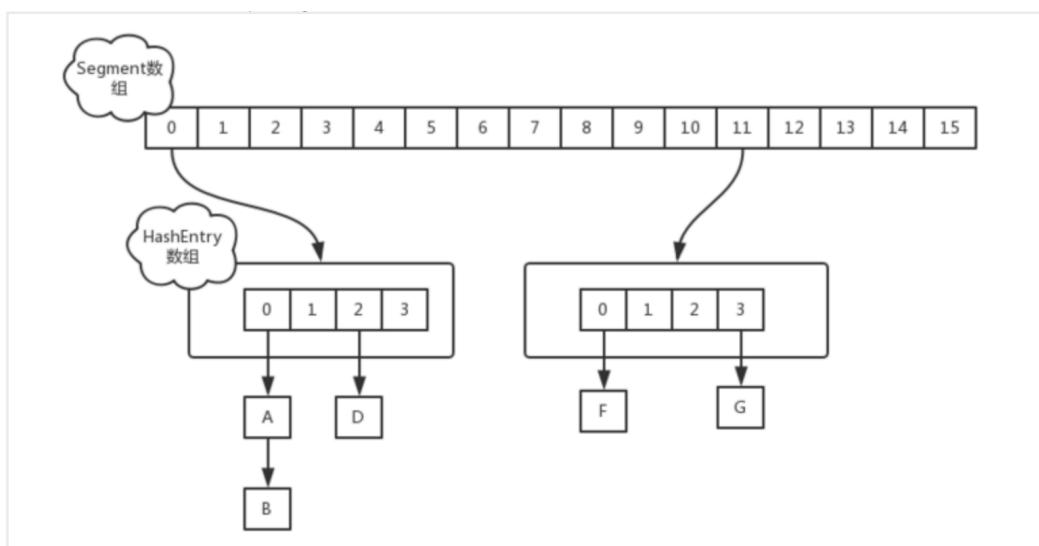
## ConcurrentHashMap

## JDK7

ConcurrentHashMap 类中包含两个静态内部类 HashEntry 和 Segment，其中 HashEntry 用来封装具体的 K/V 对，是个典型的四元组；Segment 用来充当锁的角色，每个 Segment 对象守护整个 ConcurrentHashMap 的若干个桶（可以把 Segment 看作是一个小型的哈希表），其中每个桶是由若干个 HashEntry 对象链接起来的链表。总的来说，一个 ConcurrentHashMap 实例中包含由若干个 Segment 实例组成的数组，而一个 Segment 实例又包含由若干个桶，每个桶中都包含一条由若干个 HashEntry 对象链接起来的链表。特别地，ConcurrentHashMap 在默认并发级别下会创建 16 个 Segment 对象的数组，如果键能均匀散列，每个 Segment 大约守护整个散列表中桶总数的 1/16。

个人总结：ConcurrentHashMap 最大支持 16 个 segment 即为最高支持 16 的并发，一个 segment 下由若干个 hashEntry 组成【每个 hashEntry 是一个链表且含有 next 指针】，如果发生冲突数据会放入当前 hashEntry 下【由于 next 被设置为 final，因此链表采用的是头插法，因为不能修改 next 引用】。并且 segment 继承于 ReentrantLock，所以当有线程执行某个 segment 时可以调用继承过来的 lock() 方法，后面线程进来也想执行这个 segment 时发现已经被别的线程占用则会进入等待直到释放锁才能操作，所以 JDK7 下的锁颗粒度是 segment 维度。【JDK7 下只有链表，没有引入红黑树】

参考博客：[https://blog.csdn.net/qq\\_24903931/article/details/82850170](https://blog.csdn.net/qq_24903931/article/details/82850170)



### 如何根据一个 key 定位具体位置【两段定位】

- 计算出 key 的 hash 值
- 哈希值同 segments 确定在哪一个 segment 下
- 再拿哈希值同当下 segment 的所有 hashEntry 长度做&操作确定数据是放在哪个下标下的 hashEntry

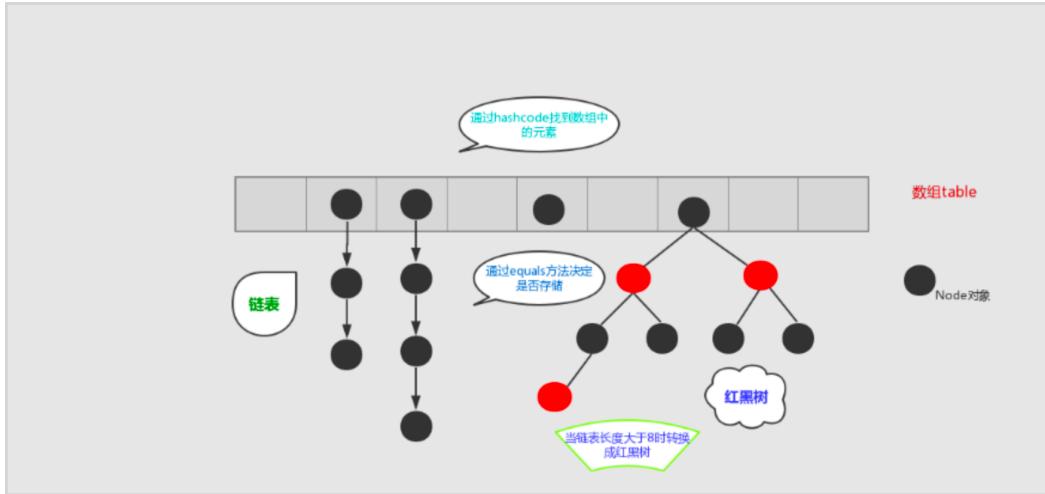
### 如何在并发下保证读写数据正确性

分段锁【继承 ReentrantLock】+ 节点用 volatile 修饰【保证所有线程可见，读时无需加锁】

## JDK8

如下图，取消了 segment 的分段思想，改用 **CAS + synchronized** 控制并发操作。

结构同 JDK8 hashMap一样采用的是数组 + 链表 + 红黑树进行数据存储。通过 CAS 去 put 数据 / 初始化数组 / 扩容，通过 synchronized 加锁解决哈希冲突【相较于 JDK7，这里的冲突采用链表 + 红黑树方式处理】，以及通过 volatile 修饰全局变量保证并发下线程可见性。



## 整体流程

整体流程：

- 首先对于每一个放入的值，首先利用spread方法对key的hashCode进行一次hash计算，由此来确定这个值在 table中的位置；
- 如果当前table数组还未初始化，先将table数组进行初始化操作；
- 如果这个位置是null的，那么使用CAS操作直接放入；
- 如果这个位置存在结点，说明发生了hash碰撞，首先判断这个节点的类型。如果该节点fh==MOVED(代表forwardingNode,数组正在进行扩容的话，说明正在进行扩容；
- 如果是链表节点 (fh>0) ,则得到的结点就是hash值相同的节点组成的链表的头节点。需要依次向后遍历确定这个新加入的值所在位置。如果遇到key相同的节点，则只需要覆盖该结点的value值即可。否则依次向后遍历，直到链表尾插入这个结点；
- 如果这个节点的类型是TreeBin的话，直接调用红黑树的插入方法进行插入新的节点；
- 插入完节点之后再次检查链表长度，如果长度大于8，把这个链表转换成红黑树；
- 对当前容量大小进行检查，如果超过了临界值（实际大小\*加载因子）就需要扩容。

参考博客：[https://blog.csdn.net/qq\\_21383435/article/details/131022859](https://blog.csdn.net/qq_21383435/article/details/131022859)

## JDK7和8的区别

JDK6,7中的ConcurrentHashMap主要使用Segment来实现减小锁粒度，分割成若干个Segment，在put的时候需要锁住Segment，get时候不加锁，使用volatile来保证可见性，当要统计全局时（比如size），首先会尝试多次计算modCount来确定，这几次尝试中，是否有其他线程进行了修改操作，如果没有，则直接返回size。如果有，则需要依次锁住所有的Segment来计算。

1.8之前put定位节点时要先定位到具体的segment，然后再在segment中定位到具体的桶。而在1.8的时候摒弃了segment臃肿的设计，直接针对的是Node[] table数组中的每一个桶，进一步减小了锁粒度。并且防止拉链过长导致性能下降，当链表长度大于8的时候采用红黑树的设计。

主要设计上的变化有以下几点：

1. 不采用segment而采用node，锁住node来实现减小锁粒度。
2. 设计了MOVED状态 当resize的中过程中 线程2还在put数据，线程2会帮助resize。
3. 使用3个CAS操作来确保node的一些操作的原子性，这种方式代替了锁。
4. sizeCtl的不同值来代表不同含义，起到了控制的作用。
5. 采用synchronized而不是ReentrantLock

## 为什么 JDK8 采用了 CAS + synchronized 而摒弃了 segments？

- JDK1.6后，synchronized进行了锁升级优化，引入了偏向锁，轻量级锁，重量级锁，在并发竞争低的情况下会优先选择轻量级锁。
- JDK1.8的ConcurrentHashMap采用的是数组+链表+红黑树，颗粒度低的情况下发生并发冲突写入的概率低，所以使用synchronized基本处于轻量级锁【CAS】。
- ReentrantLock底层是AQS，AQS的获取和释放锁的过程涉及到CAS操作、CLH队列的管理等复杂操作，以及ReentrantLock支持的高级功能比较多，例如轮询、超时、中断、公平锁和非公平锁等高级功能，这些功能使得ReentrantLock在处理复杂场景时更加灵活，但也增加了额外的开销。
- ReentrantLock是JDK层面的，synchronized是JVM层面的。相对而言synchronized的性能优化空间更大，这就使得synchronized能够随着JDK版本的升级而不改动代码的前提下获得性能上的提升。

---

## CopyOnWriteArrayList

### 原理

CopyOnWriteArrayList容器允许并发读，读操作是无锁的，性能较高。至于写操作，比如向容器中添加一个元素，则首先将当前容器复制一份，然后在新副本上执行写操作，结束之后再将原容器的引用指向新容器

- 线程安全的，多线程环境下可以直接使用，无需加锁；
- 通过锁 + 数组拷贝 + volatile关键字保证了线程安全；
- 每次数组操作，都会把数组拷贝一份出来，在新数组上进行操作，操作成功之后再赋值回去【写操作性能差，还占内存】。

### add() 方法

```

public boolean add(E e) {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        Object[] elements = getArray();
        int len = elements.length;
        Object[] newElements = Arrays.copyOf(elements, newLength: len + 1);
        newElements[len] = e;
        setArray(newElements);
        return true;
    } finally {
        lock.unlock();
    }
}

```

- 1、加锁
- 2、getArray()【获得原数组的引用】
- 3、Arrays.copyOf(elements, len + 1)【从原数组中拷贝到新数组，且新数组长度比原数组多1用于添加元素】
- 4、新增元素e放在新数组末尾
- 5、新数组引用指向老数组进行老数组数据覆盖
- 6、解锁

## remove() 方法

和add()原理差不多，通过将要删除的元素之外的数据拷贝到新数组，然后将新数组的引用指向老数组进行数据覆盖

## 优点

读操作（不加锁）性能很高，因为无需任何同步措施，比较适用于读多写少的并发场景。Java的list在遍历时，若中途有别的线程对list容器进行修改，则会抛 ConcurrentModificationException异常。而CopyOnWriteArrayList由于其“读写分离”的思想，遍历和修改操作分别作用在不同的list容器，所以在使用迭代器进行遍历时候，也就不会抛出ConcurrentModificationException异常了。

## 缺点

1. 内存占用高：每次对CopyOnWriteArrayList进行写操作时都会复制一份新的数组【因此没有扩容概念，并且在写的过程会出现两份数组在内存里】，这会导致内存占用较高。尤其在写操作频繁、数据量大的情况下，可能会消耗大量内存以及引起频繁GC。
2. 写操作性能低：由于每次写操作都需要复制一份新的数组，写操作的性能较低。在写操作频繁的情况下，可能会影响整体性能。
3. 数据一致性问题：由于写操作是在复制的新数组上进行的，因此在写操作进行期间，读操作可能会看到老的数据。这可能导致数据一致性问题，特别是在需要实时更新的场景下。
4. 不适合实时性要求高的场景：由于每次写操作都需要创建新的副本，这会增加一定的延迟。这意味着在某些情况下，如实时数据处理【查的时候还有别的线程

正在添加数据，那时候正在操作创建新数组进行数据拷贝但是还未将新数组引用指向老数组，导致查的时候可能查到数组的老数据】，CopyOnWriteArrayList 可能不是最佳选择。

---

---

## LRU 链表以及优化

- 1、被访问到的数据就往链表的头部移动
  - 2、不过这样的做法，会导致一些冷门数据被访问到就会往头部移动，且冷门数据就只访问这一次，却把热点数据往后挪动，实际场景不合适。
  - 3、将链表做冷热分离，分成冷热两个部分去存储数据，因为热点数据被淘汰的几率比较小。
  - 4、例如 1/4 存储热数据，3/4 存储冷数据，查询到的数据如果属于冷数据，则不挪动，当然如果访问次数例如超过 10 (用一个计数器或者 key-value 的 value 当做计数器)，就可以将冷数据往热数据挪动。这样避免了冷门数据偶尔访问一次将热门数据挤压出去，并且也可以减少挪动次数。
- 
- 

## JAVA 并发编程

### 进程&线程

一个 QQ 音乐是一个进程【即程序】，一个进程里面可以有多条线程  
【一个 Java 程序的运行是 main 线程和多个其他线程同时运行。】

---

---

### 用户线程&内核线程

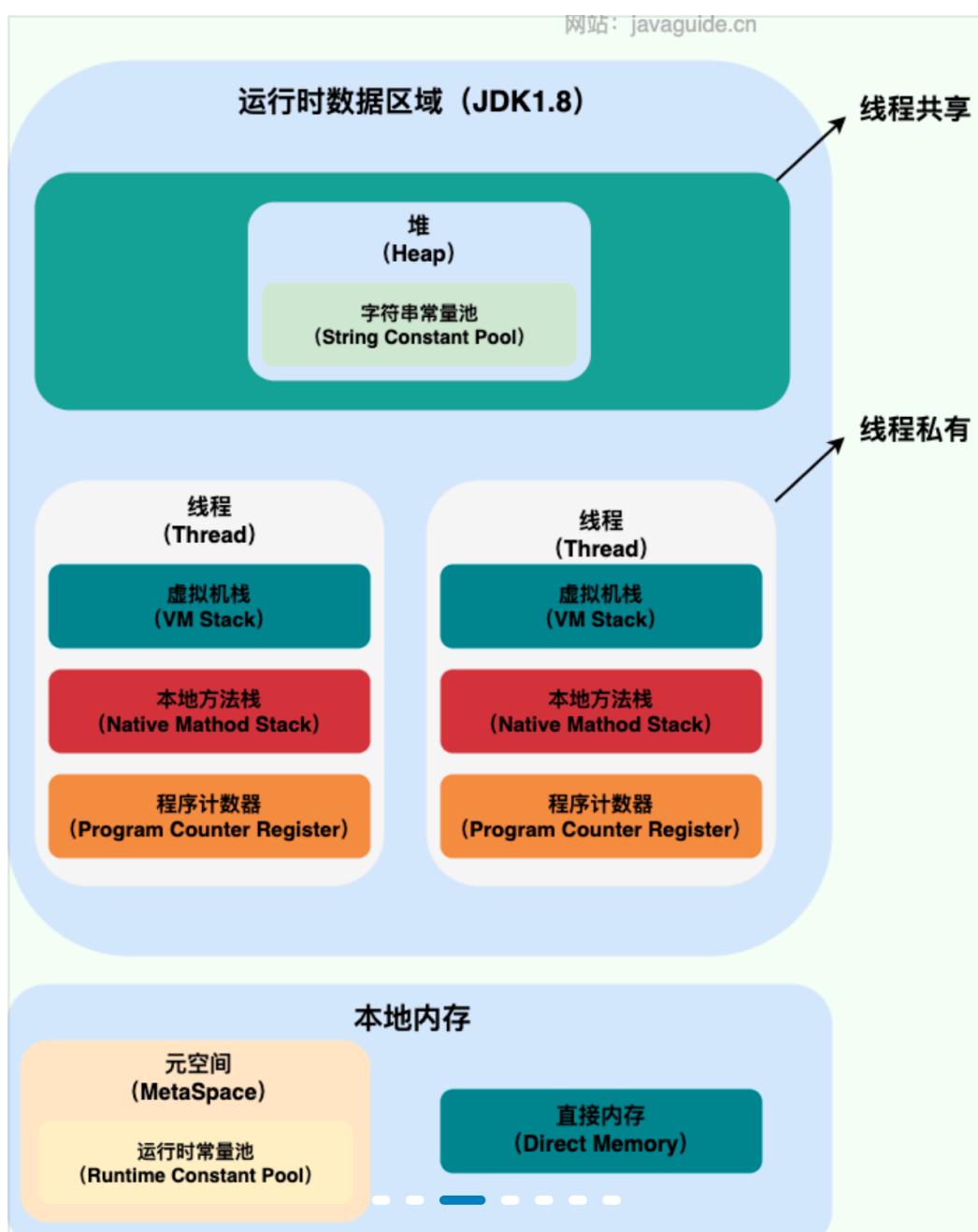
- 用户线程：由用户空间程序管理和调度的线程，运行在用户空间（专门给应用程序使用）。
- 内核线程：由操作系统内核管理和调度的线程，运行在内核空间（只有内核程序可以访问）。

简单总结一下用户线程和内核线程的区别和特点：

- 用户线程创建和切换成本低，但不可以利用多核。
- 内核态线程，创建和切换成本高，可以利用多核。

一句话概括 Java 线程和操作系统线程的关系：现在的 Java 线程的本质其实就是操作系统的线程【即内核线程】。

## JVM 内存区域下进程和线程的关系



从上图可以看出：一个进程中可以有多个线程，多个线程共享进程的堆和方法区 (JDK1.8 之后的元空间) 资源【即元空间=方法区，方法区主要用于存放已被加载的类信息、常量、静态变量、即时编译器编译后的代码等数据】，但是每个线程有自己的程序计数器、虚拟机栈 和 本地方法栈。

## 程序计数器为什么是私有的？

程序计数器主要有下面两个作用：

1. 字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制，如：顺序执行、选择、循环、异常处理。
2. 在多线程的情况下，程序计数器用于记录当前线程执行的位置，从而当线程被切换回来的时候能够知道该线程上次运行到哪儿了【例如 CPU 时间片用完被挂起后又重新抢到了 CPU 调度资源】。

需要注意的是，如果执行的是 native 方法，那么程序计数器记录的是 undefined 地址，只有执行的是 Java 代码时程序计数器记录的才是下一条指令的地址。

所以，程序计数器私有主要是为了线程切换后能恢复到正确的执行位置。

## 虚拟机栈和本地方法栈为什么是私有的？

- **虚拟机栈：** 每个 Java 方法在执行之前会创建一个栈帧用于存储局部变量表、操作数栈、常量池引用等信息。从方法调用直至执行完成的过程，就对应着一个栈帧在 Java 虚拟机栈中入栈和出栈的过程。
- **本地方法栈：** 和虚拟机栈所发挥的作用非常相似，区别是：虚拟机栈为虚拟机执行 Java 方法（也就是字节码）服务，而本地方法栈则为虚拟机使用到的 Native 方法服务。在 HotSpot 虚拟机中和 Java 虚拟机栈合二为一。

所以，为了保证线程中的局部变量不被别的线程访问到，虚拟机栈和本地方法栈是线程私有的。

## 一句话简单了解堆和方法区

堆和方法区是所有线程共享的资源，其中堆是进程中最大的一块内存，主要用于存放新创建的对象（几乎所有对象都在这里分配内存），方法区主要用于存放已被加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。

---

---

## Java 有几种创建线程的方式

Java 创建线程有很多种方式，像实现 Runnable、Callable 接口、继承 Thread 类、创建线程池等等，不过这些方式并没有真正创建出线程，严格来说，Java 就只有一种方式可以创建线程，那就是通过 new Thread().start() 创建。

而所谓的 Runnable、Callable……对象，这仅仅只是线程体，也就是提供给线程执行的任务，并不属于真正的 Java 线程，它们的执行，最终还是需要依赖于 new Thread()……

参考博客：<https://mp.weixin.qq.com/s/NspUsyhEmKnJ-4OprRFp9g>

---

---

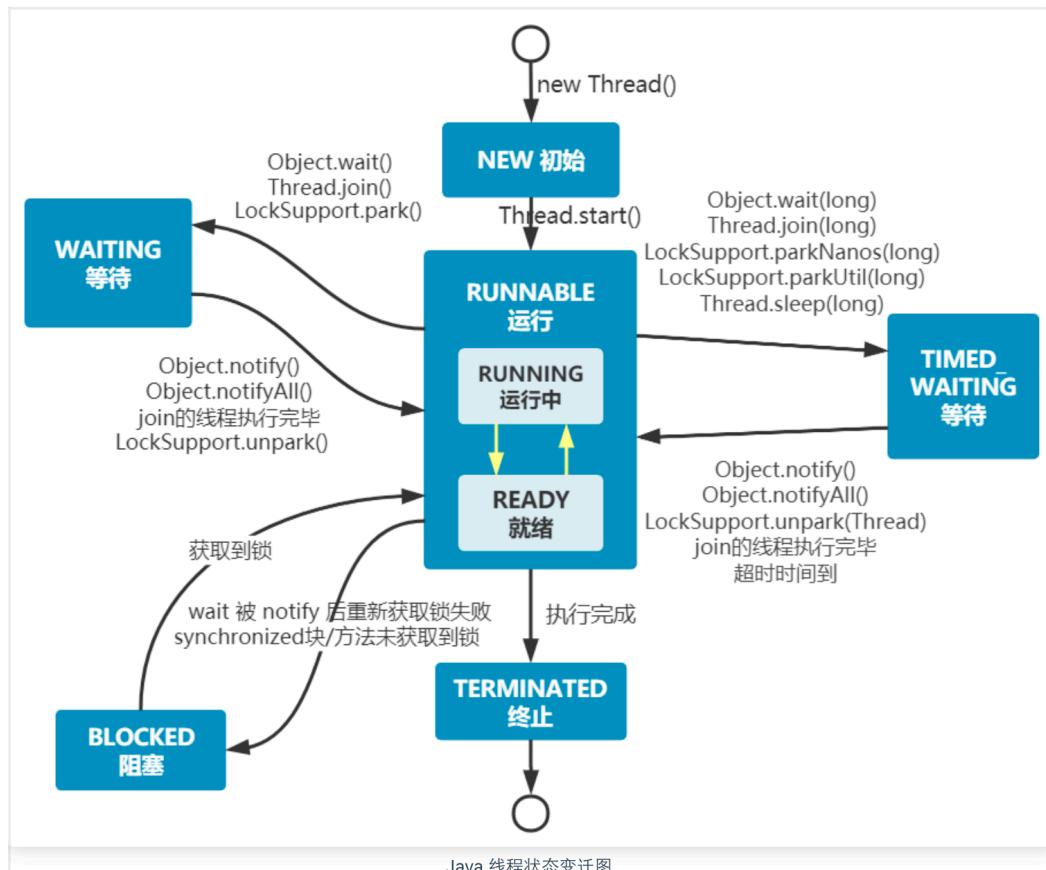
## 线程的生命周期和状态

Java 线程在运行的生命周期中的指定时刻只可能处于下面 6 种不同状态的其中一个

状态：

- NEW: 初始状态，线程被创建出来但没有被调用 start()。
- RUNNABLE: 运行状态，线程被调用了 start() 等待运行的状态。
- BLOCKED: 阻塞状态，需要等待锁释放。
- WAITING: 等待状态，表示该线程需要等待其他线程做出一些特定动作（通知或中断）。
- TIME\_WAITING: 超时等待状态，可以在指定的时间后自行返回而不是像 WAITING 那样一直等待。
- TERMINATED: 终止状态，表示该线程已经运行完毕。

线程在生命周期中并不是固定处于某一个状态而是随着代码的执行在不同状态之间切换。



由上图可以看出：线程创建之后它将处于 **NEW (新建)** 状态，调用 `start()` 方法后开始运行，线程这时候处于 **READY (可运行)** 状态。可运行状态的线程获得了 CPU 时间片 (timeslice) 后就处于 **RUNNING (运行)** 状态。

在操作系统层面，线程有 **READY** 和 **RUNNING** 状态；而在 JVM 层面，只能看到 **RUNNABLE** 状态，所以 Java 系统一般将这两个状态统称为 **RUNNABLE (运行中)** 状态。

为什么 JVM 没有区分这两种状态呢？（现在的时分 (time-sharing) 多任务 (multi-task) 操作系统架构通常都是用所谓的“时间分片 (time quantum or time slice)”方式进行抢占式 (preemptive) 轮转调度 (round-robin 式)。这个时间分片通常是很小的，一个线程一次最多只能在 CPU 上运行比如 10-20ms 的时间 (此时处于 running 状态)，也即大概只有 0.01 秒这一量级，时间片用后就要被切换下来放入调度队列的末尾等待再次调度。(也即回到 ready 状态)。线程切换的如此之快，区分这两种状态就没什么意义了。

- 1、当线程执行 `wait()` 方法之后，线程进入 **WAITING** (等待) 状态。进入等待状态的线程需要依靠其他线程的通知才能够返回到运行状态。
  - 2、**TIMED\_WAITING(超时等待)** 状态相当于在等待状态的基础上增加了超时限制，比如通过 `sleep (long millis)` 方法或 `wait (long millis)` 方法可以将线程置于 **TIMED\_WAITING** 状态。当超时时间结束后，线程将会返回到 **RUNNABLE** 状态。
  - 3、当线程进入 `synchronized` 方法/块或者调用 `wait` 后 (被 `notify`) 重新进入 `synchronized` 方法/块，但是锁被其它线程占有，这个时候线程就会进入 **BLOCKED (阻塞)** 状态。
  - 4、线程在执行完了 `run()` 方法之后将会进入到 **TERMINATED (终止)** 状态。
- 
- 

## 什么是线程上下文切换？

线程在执行过程中会有自己的运行条件和状态 (也称上下文)，比如上文所说到过的程序计数器，栈信息等。当出现如下情况的时候，线程会从占用 CPU 状态中退出。

- 主动让出 CPU，比如调用了 `sleep()`, `wait()` 等。
- 时间片用完，因为操作系统要防止一个线程或者进程长时间占用 CPU 导致其他线程或者进程饿死。
- 调用了阻塞类型的系统中断，比如请求 IO，线程被阻塞。
- 被终止或结束运行

这其中前三种都会发生线程切换，线程切换意味着需要保存当前线程的上下文，留待线程下次占用 CPU 的时候恢复现场。并加载下一个将要占用 CPU 的线程上下文。这就是所谓的 **上下文切换**。

上下文切换是现代操作系统的基本功能，因其每次需要保存信息恢复信息，这将会占用 CPU，内存等系统资源进行处理，也就意味着效率会有一定损耗，**如果频繁切换就会造成整体效率低下**。

---

---

## `sleep()` 方法和 `wait()` 方法对比

**共同点：**两者都可以暂停线程的执行。

**区别：**

- **sleep()** 方法没有释放锁，而 **wait()** 方法释放了锁。
  - `wait()` 通常被用于线程间交互/通信，`sleep()` 通常被用于暂停执行。
  - `wait()` 方法被调用后，线程不会自动苏醒，需要别的线程调用同一个对象上的 `notify()` 或者 `notifyAll()` 方法。`sleep()` 方法执行完成后，线程会自动苏醒，或者也可以使用 `wait(long timeout)` 超时后线程会自动苏醒。
  - `sleep()` 是 `Thread` 类的静态本地方法，`wait()` 则是 `Object` 类的本地方法。
-

## 为什么 `wait()` 方法不定义在 `Thread` 中？

`wait()` 是让获得对象锁的线程实现等待，会自动释放当前线程占有的对象锁。每个对象（Object）都拥有对象锁，既然要释放当前线程占有的对象锁并让其进入 WAITING 状态，自然是要操作对应的对象（Object）而非当前的线程（Thread）。

类似的问题：为什么 `sleep()` 方法定义在 `Thread` 中？

因为 `sleep()` 是让当前线程暂停执行，不涉及到对象类，也不需要获得对象锁。

## 可以直接调用 `Thread` 类的 `run` 方法吗？

`new` 一个 `Thread`，线程进入了新建状态。调用 `start()` 方法，会启动一个线程并使线程进入了就绪状态，当分配到时间片后就可以开始运行了。`start()` 会执行线程的相应准备工作，然后自动执行 `run()` 方法的内容，这是真正的多线程工作。但是，直接执行 `run()` 方法，会把 `run()` 方法当成一个 `main` 线程下的普通方法去执行，并不会在某个线程中执行它，所以这并不是多线程工作。

总结：调用 `start()` 方法方可启动线程并使线程进入就绪状态，直接执行 `run()` 方法的话不会以多线程的方式执行。

## JMM (Java 内存模型) 详解

### CPU 缓存模型

- 作用：CPU Cache 缓存的是内存数据用于解决 CPU 处理速度和内存不匹配的问题，内存缓存的是硬盘数据用于解决硬盘访问速度过慢的问题【CPU 缓存比内存更快】。
- 工作方式：先从内存将变量数据赋值一份到 CPU 缓存中，之后涉及计算的话直接从 CPU 缓存获取，计算完后将数据同步回缓存中去【并发下容易出现内存缓存数据不一致问题】。

### 指令重排序

#### 什么是指令重排序

简单来说就是系统在执行代码的时候并不一定是按照你写的代码的顺序依次执行。

常见的指令重排序有下面 3 种情况：

- 编译器优化重排：编译器（包括 JVM、JIT 编译器等）在不改变单线程程序语义的前提下，重新安排语句的执行顺序。
- 指令并行重排：现代处理器采用了指令级并行技术（Instruction-Level Parallelism, ILP）来将多条指令重叠执行。如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。

- 内存重排序：主存和本地内存的内容可能不一致，进而导致程序在多线程下执行可能出现问题。

Java 源代码会经历 编译器优化重排 → 指令并行重排 → 内存系统重排 的过程，最终才变成操作系统可执行的指令序列。

指令重排序可以保证串行语义一致，但是没有义务保证多线程间的语义也一致，所以在多线程下，指令重排序可能会导致一些问题【通过内存屏障来禁止指令重排，在并发情况下避免由于代码执行顺序重排导致的数据不一致】。

## JMM 登场【由于 CPU 缓存和指令重排序的问题而出现的解决方案】

JMM(Java 内存模型)，它本身只是一个抽象的概念，并不真实存在，它描述的是一种规则或规范，是和多线程相关的一组规范。通过这组规范，定义了程序中对各个变量（包括实例字段，静态字段和构成数组对象的元素）的访问方式。需要每个 JVM 的实现都要遵守这样的规范，有了 JMM 规范的保障，并发程序运行在不同的虚拟机上时，得到的程序结果才是安全可靠可信赖的。如果没有 JMM 内存模型来规范，就可能会出现，经过不同 JVM 翻译之后，运行的结果不相同也不正确的情况。

计算机在执行程序时，每条指令都是在CPU中执行的。而执行指令的过程中，势必涉及到数据的读取和写入。由于程序运行过程中的临时数据是存放在主存（物理内存）当中的，这时就存在一个问题，由于CPU执行速度很快，而从内存读取数据和向内存写入数据的过程，跟CPU执行指令的速度比起来要慢的多（硬盘 < 内存 < 缓存 cache < CPU）。因此如果任何时候对数据的操作都要通过和内存的交互来进行，会大大降低指令执行的速度。因此在CPU里面就有了高速缓存。也就是当程序在运行过程中，会将运算需要的数据从主存复制一份到CPU的高速缓存当中，那么CPU进行计算时，就可以直接从它的高速缓存中读取数据或向其写入数据了。当运算结束之后，再将高速缓存中的数据刷新到主存当中。

## JMM 是如何抽象线程和主内存之间的关系？

Java 内存模型 (JMM) 抽象了线程和主内存之间的关系，就比如说线程之间的共享变量必须存储在主内存中。

### 什么是主内存？什么是本地内存？

- **主内存：**所有线程创建的实例对象都存放在主内存中，不管该实例对象是成员变量，还是局部变量，类信息、常量、静态变量都是放在主内存中。为了获取更好的运行速度，虚拟机及硬件系统可能会让工作内存优先存储于寄存器和高速缓存中。
- **本地内存：**每个线程都有一个私有的本地内存，本地内存存储了该线程以读 / 写共享变量的副本。每个线程只能操作自己本地内存中的变量，无法直接访问其他线程的本地内存。如果线程间需要通信，必须通过主内存来进行。本地内存是 JMM 抽象出来的一个概念，并不真实存在，它涵盖了缓存、写缓冲区、寄存器以及其他硬件和编译器优化。

在JVM中，栈负责运行（主要是方法），堆负责存储（比如new的对象）。由于JVM运行程序的实体是线程，而每个线程在创建时，JVM都会为其创建一个工作内存（有些地方称为栈空间），工作内存是每个线程的私有数据区域。而JAVA内存模型中规定，所有变量都存储在主内存中，主内存是共享内存区域，所有线程都可以访问。

但线程对变量的操作（读取赋值等）必须在自己的工作内存中进行。首先要将变量从主内存拷贝到自己的工作内存空间，然后对变量进行操作，操作完成后，再将变量写回到主内存。由于不能直接操作主内存中的变量，各个线程的工作内存中存储着主内存中的变量副本，因此，不同的线程之间无法直接访问对方的工作内存，线程

间的通信（传值）必须通过主内存来完成。

## Java 内存区域和 JMM 有何区别？

- JVM 内存结构和 Java 虚拟机的运行时区域相关，定义了 JVM 在运行时如何分区存储程序数据，就比如说堆主要用于存放对象实例。
- Java 内存模型和 Java 的并发编程相关，抽象了线程和主内存之间的关系就比如说线程之间的共享变量必须存储在主内存中，规定了从 Java 源代码到 CPU 可执行指令的这个转化过程要遵守哪些和并发相关的原则和规范，其主要目的是为了简化多线程编程，增强程序可移植性的。

## happens-before 原则是什么【描述两个操作之间的内存可见性】？

- 为了对编译器和处理器的约束尽可能少，只要不改变程序的执行结果（单线程程序和正确执行的多线程程序），编译器和处理器怎么进行重排序优化都行。
- 对于会改变程序执行结果的重排序，JMM 要求编译器和处理器必须禁止这种重排序【如果不改变执行结果 JMM 也允许重排序】。

## 并发编程三个重要特性

原子性、可见性、有序性

个人总结：JMM 是一组抽象的概念，用于描述 JVM 和线程之间交互的规则【例如从主内存读到工作内存，操作完后工作内存的数据同步回主内存】。例如 happens-before 原则用于解决指令重排，以及解决由于线程工作内存同主内存的交互方式导致在并发场景下出现的数据不一致问题。

---

---

## volatile

可以保证变量的可见性，以及防止 JVM 的指令重排序。如果我们将变量声明为 volatile，在对这个变量进行读写操作的时候，会通过插入特定的 内存屏障 的方式来禁止指令重排序。

例如单例模式下，正常 new 一个对象的流程为：

1. 为 uniqueInstance 分配内存空间
2. 初始化 uniqueInstance
3. 将 uniqueInstance 指向分配的内存地址

但是由于 JVM 具有指令重排的特性，执行顺序有可能变成 1->3->2。指令重排在单线程环境下不会出现问题，但是在多线程环境下会导致一个线程获得还没有初始化的实例。例如，线程 T1 执行了 1 和 3，此时 T2 调用 getUniqueInstance() 后发现 uniqueInstance 不为空，因此返回 uniqueInstance，但此时 uniqueInstance 还未被初始化。所以可以通过 volatile 修饰字段/对象防止指令重排。

PS： volatile 只能保证变量的可见性但是不保证原子性【利用 synchronized、Lock 或者 AtomicInteger 可以保证原子性】。

---

## 悲观锁&乐观锁

### 悲观锁

悲观锁总是假设最坏的情况，认为共享资源每次被访问的时候就会出现问题(比如共享数据被修改)，所以每次在获取资源操作的时候都会上锁，这样其他线程想拿到这个资源就会阻塞直到锁被上一个持有者释放。也就是说，**共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程。**

高并发的场景下，激烈的锁竞争会造成线程阻塞，大量阻塞线程会导致系统的上下文切换，增加系统的性能开销。并且，悲观锁还可能会存在死锁问题，影响代码的正常运行。

### 乐观锁

乐观锁总是假设最好的情况，认为共享资源每次被访问的时候不会出现问题，线程可以不停地执行，无需加锁也无需等待，只是在提交修改的时候去验证对应的资源(也就是数据)是否被其它线程修改了(**具体方法可以使用版本号机制或 CAS 算法**)。高并发的场景下，乐观锁相比悲观锁来说，不存在锁竞争造成线程阻塞，也不会有死锁的问题，在性能上往往更胜一筹。但是，如果冲突频繁发生(写占比非常多的情况)，会频繁失败和重试，这样同样会非常影响性能，导致 CPU 飙升【还有 ABA 问题，可以通过加版本号解决】。

### 两种锁实际场景选择

- 悲观锁通常多用于写比较多的情况(多写场景，竞争激烈)，这样可以避免频繁失败和重试影响性能，悲观锁的开销是固定的。不过，如果乐观锁解决了频繁失败和重试这个问题的话(比如 LongAdder)，也是可以考虑使用乐观锁的，要视实际情况而定。
  - 乐观锁通常多用于写比较少的情况(多读场景，竞争较少)，这样可以避免频繁加锁影响性能。不过，乐观锁主要针对的对象是单个共享变量(参考 `java.util.concurrent.atomic` 包下面的原子变量类)。
- 

## synchronized 关键字

### 用法总结

- `synchronized` 关键字加到 `static` 静态方法和 `synchronized(class)` 代码块上都是给 Class 类上锁；
- `synchronized` 关键字加到实例方法上是给对象实例上锁；
- 尽量不要使用 `synchronized(String a)` 因为 JVM 中，字符串常量池具有缓存功能。

## synchronized 底层原理【优化之锁升级】

### 锁信息存放在哪？

对象头里的 Markword：存储对象的 hashCode、垃圾回收对象的年龄以及锁信息等。

HotSpot64位虚拟机对象的Markword头信息						
锁信息/gc	56位		1位	4位	1位（是否偏向锁）	2位（锁标志位）
无锁	unused(25位)	对象的hashCode (31位, 有调用的情况下才有值)	unused (1位)	分代年龄 (最大15岁)	0	01
偏向锁	偏向线程ID (54位)		Epoch (2位, 偏向锁的时间戳)	unused (1位)	分代年龄 (最大15岁)	1
轻量级锁	指向线程栈中锁记录的指针 (62位)					00
重量级锁	指向重量级锁的指针 (62位)					10
GC标记	CMS过程中用到的标记信息 (56位)					11

解释如下：

- 对象的hashCode占31位，重写类的hashCode方法返回int类型，只有在无锁情况下，是在有调用的情况下会计算该值并写到对象头中，其他情况该值是空的。
- 分代年龄占4位，最大值也就是15，在GC中，当survivor区中对象复制一次，年龄加1，默认是到15之后会移动到老年代。
- 是否偏向锁占1位，无锁和偏向锁的最后两位都是01，使用这一位来标识区分是无锁还是偏向锁。
- 锁标志位占2位，锁状态标记位，同是否偏向锁标志位标识对象处于什么锁状态。
- 偏向线程ID占54位，只有偏向锁状态才有，这个ID是操作系统层面的线程唯一id，跟java中的线程id是不一致的。

### ● 无锁

对于共享资源，不涉及多线程的竞争访问。

### ● 偏向锁

共享资源首次被访问时，JVM会对该共享资源对象做一些设置，比如将对象头中是否偏向锁标志位置为1，对象头中的线程ID设置为当前线程ID（注意：这里是操作系统的线程ID），当发生线程竞争时，会去比较当前线程ID和对象头设置的线程ID是否一致，如果一致则表明当前线程已经获取到锁了，这个点也可以说明

synchronized具有可重入锁功能。如果当前线程ID和对象头设置的线程ID不一样，表示发生了线程竞争，可能会触发偏向锁撤销或者升级成轻量级锁的行为。

1、假如当前线程被检测到已经死亡则会触发偏向锁撤销。因为偏向锁不会主动释放，只会在发生竞争的时候可能发生偏向锁撤销或者锁升级，因为很可能在一段时间内一直都是这个线程在调度。所以这个也能解释为什么会有偏向锁撤销的行为，因为当前线程执行完任务后被销毁了，下一个线程进来时候需要把对象头的偏向锁ID替换成自己来取代上一个线程ID。

2、假如当前线程还存活会升级成轻量级锁或重量级锁，不过当前线程仍然持有当前锁，只不过锁类型变了。如果升级成轻量级锁，那么其他等待线程会通过CAS方式获取，如果升级成重量级锁，其他等待线程会进入阻塞等待操作系统的调用。

### ● 轻量级锁

当多个线程同时申请共享资源锁的访问时，这就产生了竞争，JVM会先尝试使用轻量级锁，以CAS方式来获取锁（一般就是自旋加锁，不阻塞线程采用循环等待的方式），成功则获取到锁，状态为轻量级锁，失败（达到一定的自旋次数还未成功）则锁升级到重量级锁。

**PS:**

## 无锁和偏向锁的区别：

偏向锁的核心思想是，在大多数情况下，锁不仅不存在多线程竞争，而且总是由同一个线程多次获得（因此偏向锁设计初衷是假定只有一个线程在访问操作）。因此，为了降低这种场景下获取锁的开销，引入了偏向锁。它的目标是在没有竞争的情况下，消除同步操作，甚至连CAS（Compare-And-Swap）操作都省去，从而提高程序性能。

## 偏向锁实现原理：

- 1、当一个线程第一次获取锁时，JVM会将对象头中的“是否为偏向锁”标志位设为“1”，并将线程ID记录在对象头的Mark Word中。此时，锁就“偏向”于这个线程。
- 2、当这个线程再次尝试获取该锁时，它只需要检查对象头中记录的线程ID是否是自己的ID即可。如果是，它就无需再进行任何加锁操作，直接可以执行同步代码，大大降低了获取锁的成本。
- 3、线程退出同步代码块时，并不会主动释放偏向锁。

- 重量级锁

如果共享资源锁已经被某个线程持有，此时是偏向锁状态，未释放锁前，再有其他线程来竞争时，则会升级到重量级锁，另外轻量级锁状态多线程竞争锁时，也会升级到重量级锁，重量级锁由操作系统来实现，所以性能消耗相对较高。

这4种级别的锁，在获取时性能消耗：重量级锁 > 轻量级锁 > 偏向锁 > 无锁。

## 对象头里的类型指针的优化【题外话】

JAVA指针压缩：指针压缩是一种内存管理技术，旨在减少对象引用在内存中所占用的空间，这样可以减少堆内存中对象引用的空间占用，从而降低整体内存占用，并提升程序的性能【引用指针由8字节变为4字节，降低了一半的对象引用内存占用】。

## 锁升级流程

偏向锁【JDK1.8后默认开启】->轻量级锁【CAS】->重量级锁【操作系统控制内核态和用户态的切换，性能较差】。

如果JVM参数设置为不开启偏向锁，那么默认为无锁，无锁一旦碰到线程竞争直接升级为重量级锁【没有无锁升级成偏向锁这个过程，二者通过JVM设置只能配置一种】。

## 偏向锁->轻量级锁【满足条件】

一旦发生线程竞争就会从偏向锁升级为轻量级锁【第二条线程进来开始计算】

## 轻量级锁->重量级锁【满足条件】

若当前只有一个等待线程，则可通过自旋继续尝试，当自旋超过一定的次数，或者一个线程在持有锁，一个线程在自旋，又有第三个线程来访问时，轻量级锁就会膨胀为重量级锁。

## 重量级锁底层处理方式

当轻量级锁获取锁失败时，说明有竞争存在，轻量级锁会升级为重量级锁，此时，JVM会将线程阻塞，直到获取到锁后才能进入临界区域，底层是通过操作系统的mutex lock来实现的，每个对象指向一个monitor对象，这个monitor对象在堆中与锁是关联的，通过monitorenter指令插入到同步代码块在编译后的开始位置，monitorexit指令插入到同步代码块的结束处和异常处，这两个指令配对出现。JVM的线程和操作系统的线程是对应的，重量级锁的Markword里存储的指针是这个monitor对象的地址，操作系统来控制内核态中的线程的阻塞和恢复，从而达到

JVM 线程的阻塞和恢复，涉及内核态和用户态的切换，影响性能，所以叫重量级锁。

## AQS

AQS 的全称为 AbstractQueuedSynchronizer，翻译过来的意思就是抽象队列同步器。这个类在 `java.util.concurrent.locks` 包下面。

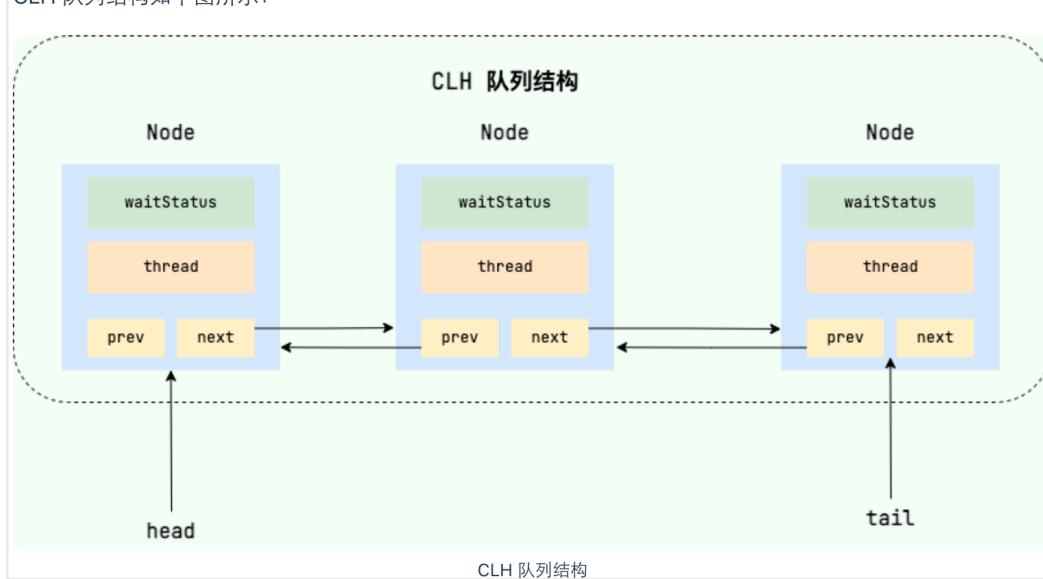
### AQS 核心思想

如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并且将共享资源设置为锁定状态。如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制，这个机制 AQS 是基于 **CLH 锁** (Craig, Landin, and Hagersten locks) 实现的。

CLH 锁是对自旋锁的一种改进，是一个虚拟的双向队列（虚拟的双向队列即不存在队列实例，仅存在结点之间的关联关系），暂时获取不到锁的线程将被加入到该队列中。AQS 将每条请求共享资源的线程封装成一个 CLH 队列锁的一个结点 (Node) 来实现锁的分配。在 CLH 队列锁中，一个节点表示一个线程，它保存着线程的引用 (thread)、当前节点在队列中的状态 (waitStatus)、前驱节点 (prev)、后继节点 (next)。

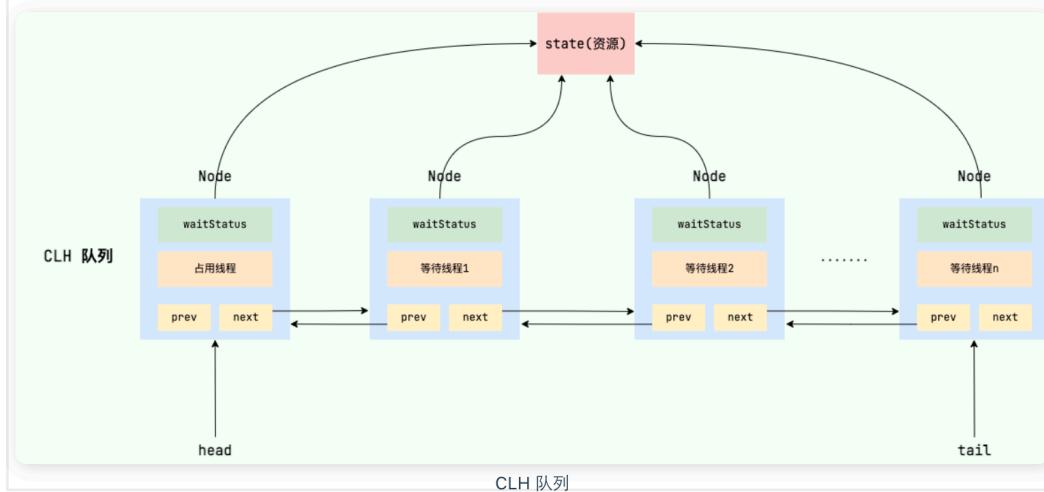
**【所以 CLH 锁可以理解为双向链表的数据结构，在获取不到锁资源的时候将这个线程放入到链表里】**

CLH 队列结构如下图所示：



以可重入的互斥锁 `ReentrantLock` 为例，它的内部维护了一个 `state` 变量，用来表示锁的占用状态。`state` 的初始值为 0，表示锁处于未锁定状态。当线程 A 调用 `lock()` 方法时，会尝试通过 `tryAcquire()` 方法独占该锁，并让 `state` 的值加 1。如果成功了，那么线程 A 就获取到了锁。如果失败了，那么线程 A 就会被加入到一个等待队列 (CLH 队列) 中，直到其他线程释放该锁。假设线程 A 获取锁成功了，释放锁之前，A 线程自己是可以重复获取此锁的 (`state` 会累加)。这就是可重入性的体现：一个线程可以多次获取同一个锁而不会被阻塞。但是，这也意味着，一个线程必须释放与获取的次数相同的锁，才能让 `state` 的值回到 0，也就是让锁恢复到未锁定状态。只有这样，其他等待的线程才能有机会获取该锁。

AQS( AbstractQueuedSynchronizer )的核心原理图：



## State 获取方式和作用

AbstractQueuedSynchronizer (AQS) 中的 state 是通过 CAS (Compare and Swap) 操作来获取和更新的。

在 AQS 中, state 是用来表示同步状态的, 通过对 state 的操作实现线程的阻塞和唤醒。CAS 是一种乐观锁定的机制, 用于实现原子操作, 其操作步骤为“比较-更新-写入”。当多个线程尝试更新同一个变量时, CAS 可以确保只有一个线程成功更新, 其他线程需要重试。

在 AQS 中, 通过 CAS 操作来获取和更新 state, 确保对 state 的操作是原子的, 避免了线程之间的竞争条件。例如, 在 ReentrantLock 和 CountDownLatch 等同步器中, state 的更新都是通过 CAS 操作来实现的, 保证了线程安全性和同步的正确性。因此, CAS 是实现 AQS 中 state 操作的关键机制之一, 确保了多线程环境下对同步状态的安全操作。

## AQS 的 state 是不是由 CLH 队列通过自旋方式不断轮询 state 状态的?

AbstractQueuedSynchronizer (AQS) 中的 state 是由 CLH (Craig, Landin, and Hagersten) 队列通过自旋方式不断轮询 state 状态的。

在 AQS 中, CLH 队列是一种基于链表的队列, 用于管理等待线程。当一个线程尝试获取锁或者资源时, 如果锁或资源已被其他线程占用, 该线程会被加入到 CLH 队列中并进行自旋等待。在自旋过程中, 线程不断轮询 state 状态, 以判断是否可以获取锁或资源。

通过自旋方式不断轮询 state 状态, 可以减少线程由用户态到内核态的切换次数, 提高线程的性能和效率。当 state 的状态发生变化时 (比如锁被释放), 等待线程可以及时感知并尝试再次获取锁或资源。

CLH 队列结合了自旋等待和链表的管理方式, 使得多线程之间的竞争和等待得以有效地管理和调度, 确保了线程安全和同步的正确性。

因此, 在 AQS 中, CLH 队列通过自旋方式不断轮询 state 状态, 是实现线程等待和竞争管理的重要机制之一。

## 总结

AQS 底层设计是一个虚拟的 CLH 锁【双向队列】，如果方法没被线程占用则 state=0, 如果被线程占用则 state=1【那么这个线程则会加入到队列】，并且 state 可以不断自增所以用了 AQS 底层的锁是支持可重入的【指的是线程在没有释放锁的

情况下可以再次获取自己的内部锁】，并且由于CLH锁是一种自旋锁，所以会不断监听state状态，一旦监听到state=0时则表示可以获取锁或资源。

PS：ReentrantLock默认是非公平锁，所以采用的是CAS去自旋获取state的资源，但是如果设定为公平锁，就不会以CAS的方式去获取state资源，而是按照CLH队列一个一个有序的去获取资源。

---

## ReentrantLock 源码

### ReentrantLock 是什么？

ReentrantLock 实现了 Lock 接口，是一个可重入且独占式的锁，和 synchronized 关键字类似。不过，ReentrantLock 更灵活、更强大，增加了轮询、超时、中断、公平锁和非公平锁等高级功能。

### ReentrantLock VS Synchronized

其实 ReentrantLock 和 Synchronized 最核心的区别就在于 Synchronized 适合于并发竞争低的情况，因为 Synchronized 的锁升级如果最终升级为重量级锁在使用的过程中是没有办法消除的，意味着每次都要和cpu去请求锁资源，而 ReentrantLock 主要是提供了阻塞的能力，通过在高并发下线程的 park 挂起，CAS 抢锁等，提高并发能力。并且功能支持比 synchronized 更加丰富。

所以低竞争情况下两者差不多，高并发下 ReentrantLock 效率更高。

### 什么是 AQS

- AQS 里面的内部类 Node 的几个属性

```
1 | 
2 | volatile int waitStatus; // Node里，记录状态用的
3 | 
4 | volatile Thread thread; // Node里，标识哪个线程
5 | 
6 | volatile Node prev; // 前驱节点（这个Node的上一个是谁）
7 | 
8 | volatile Node next; // 后继节点（这个Node的下一个是谁）
```

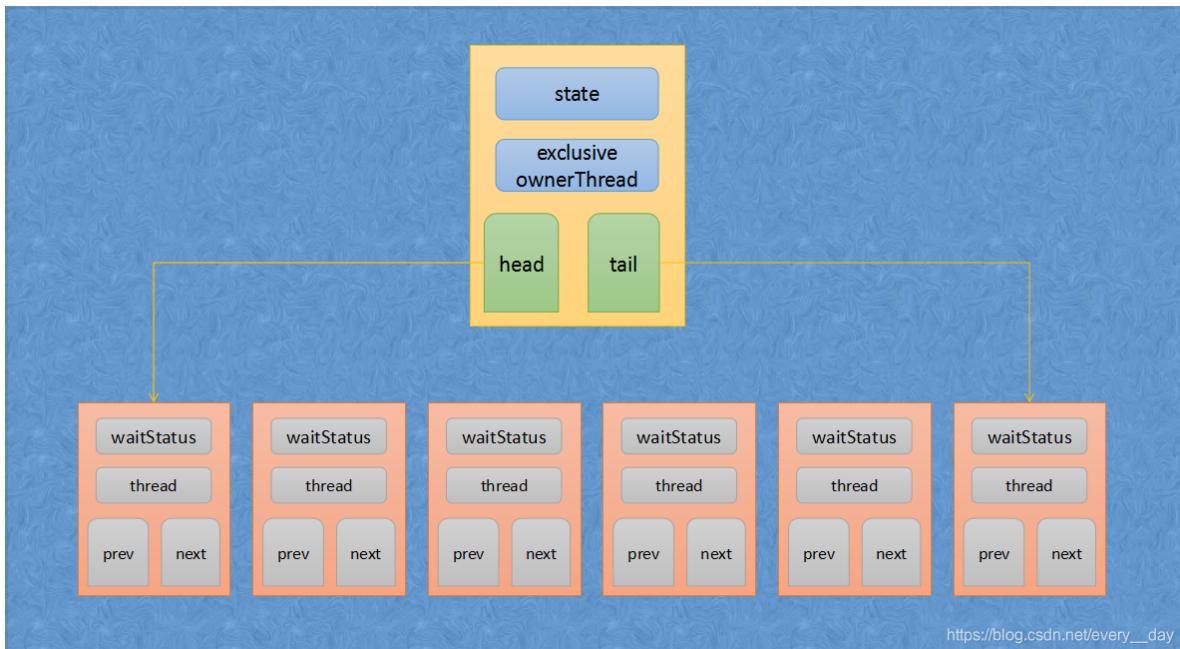
- AQS 本身的属性

```

1 private transient Thread exclusiveOwnerThread; // 标识拿到锁的是哪个线程
2
3 private transient volatile Node head; // 标识头节点
4
5 private transient volatile Node tail; // 标识尾节点
6
7
8 private volatile int state; // 同步状态, 为0时, 说明可以抢锁

```

- AQS 流程图



AQS 实现原理依赖内部 state(同步状态) 和 CLH 队列 (FIFO 双向队列), 如果当前线程获取 state 同步状态失败 AQS 会将该线程以及状态等信息构造一个 Node 节点, 并将这个 Node 节点添加到队尾, 同时阻塞当前线程, 当同步状态释放时, 唤醒队列头节点。

而 ReentrantLock 这个类虽然并没有直接继承 AQS 类, 而是有一个 sync 内部类, 它继承了 AQS, 例如非公平锁 NonFairSync.Class 和公平锁 FairSync.Class, 他们是继承了 Sync 对象。而 ReentrantLock 实现的加锁、解锁都是依赖于 AQS 独占模式下获取资源和释放资源所提供的方法, 如下:

### 获取资源

- (1) tryAcquire(arg) 尝试获取资源, 如果获取成功返回 true 则 acquire() 直接返回。如果返回 false, 则进入 (2);
- (2) addWaiter(Node.EXCLUSIVE), arg 将该线程加入 CLH 等待队列的尾部, 并标记为独占模式, 完成后进入 (3);
- (3) acquireQueued() 以独占模式不间断获取队列中已存在的线程直到获取元素。获取元素成功后若线程未中断过则返回 false 然后 acquire() 直接返回, 如果等待过程中被中断过则返回 true, 然后进入 (4);
- (4) selfInterrupt() 这个方法翻译过来就是自我在中断, 注意这个中断方法必须是在获取元素成功之后才会执行的, 就是说获取资源成功了才会执行的, 不是立即响应中断的。

## 释放资源

- (1) 使用 tryRelease(arg) 尝试释放资源
- (2) 释放成功则使用 unPark() 唤醒等待队列里面的下一个线程

## ReentrantLock 加锁逻辑【非公平锁举例】

非公平锁和公平锁区别：

- 1、非公平锁会先通过 CAS 尝试抢锁，抢不到走 acquire(1) 方法，公平锁直接走 acquire(1) 方法。
- 2、在 tryAcquire(arg) 方法中，公平锁多了一个 hasQueuedPredecessors() 方法的判断。

```
1 |     final void lock() {
2 |         if (compareAndSetState(0, 1)) // 假设A执行此代码成功
3 |             setExclusiveOwnerThread(Thread.currentThread()); // 标识哪个线程持有该锁
4 |         else
5 |             acquire(1);
6 |     }
7 |
8 | // 这里compareAndSetState(0, 1),就是进行CAS操作
9 |     protected final boolean compareAndSetState(int expect, int update) {
10 |         // unsafe类调用的是native方法
11 |         return unsafe.compareAndSwapInt(this, stateOffset, expect, update);
12 |     }
13 |
14 | // 线程A拿到了锁就直接返回了,那线程B C D 就进入 acquire(1) 方法
15 |     public final void acquire(int arg) {
16 |         if (!tryAcquire(arg) && acquireQueued(addWaiter(Node.EXCLUSIVE), arg)){
17 |             selfInterrupt();
18 |         }
19 |     }
```

假设有ABCD四个线程，同时执行到加锁这行代码。前文说过，state是0，代表可抢锁。接下来主要分析 acquire(1) 方法里面的细节【走到 acquire(1) 方法说明锁已经被占用了】。

### 1、tryAcquire(arg) 即为抢锁

```
1 // tryAcquire 这个方法，非公平锁调用这个方法，传入参数是1，这个与可重入相关，待会再细说。
2 protected final boolean tryAcquire(int acquires) {
3     return nonfairTryAcquire(acquires);
4 }
5
6 final boolean nonfairTryAcquire(int acquires) {
7     final Thread current = Thread.currentThread();
8     int c = getState(); // 获取 state状态,
9     if (c == 0) { // state是0，继续抢锁。刚进入lock 没抢到，现在可能锁已释放，有可能这次就抢成功了。
10        if (compareAndSetState(0, acquires)) { // B C D 线程再次竞争拿锁
11            setExclusiveOwnerThread(current);
12            return true; // 拿到了锁，lock 方法就结束了。
13        }
14    }
15    // 来抢锁的线程，本身就执有锁，说明是再次加锁，state 再加 1
16    else if (current == getExclusiveOwnerThread()) {
17        int nextc = c + acquires;
18        if (nextc < 0) // overflow 内在溢出了，抛出异常
19            throw new Error("Maximum lock count exceeded");
20        setState(nextc);
21        return true;
22    }
23    return false; // 抢锁失败
24 }
25 }
```

先判断state是否为0，如果为0尝试通过CAS抢锁【如果是公平锁，抢锁之前会先调hasQueuedPredecessors()方法，判断CLH队列为空或者头结点的下一个节点是自己才会去通过CAS抢锁，否则乖乖加入线程队列排队】，如果state不为0则去判断当前持锁的线程是不是现在请求的线程，如果是的话state+1，这就是reentrantlock的可重入锁的代码设计。

## 2、addWaiter(Node.EXCLUSIVE)，入队，自旋能保证，一定入队成功

```

1 | addWaiter(Node.EXCLUSIVE) 这个方法就是用自旋的方式，保证线程入队
2 |
3 |     private Node addWaiter(Node mode) {
4 |         Node node = new Node(Thread.currentThread(), mode); // 线程与Node绑定
5 |         Node pred = tail;
6 |         if (pred != null) { // 队列已经初始化，直接将new 出来的node 放到队尾
7 |             node.prev = pred;
8 |             if (compareAndSetTail(pred, node)) { // CAS 设置队尾
9 |                 pred.next = node;
10 |                 return node;
11 |             }
12 |         }
13 |         // 走到这里，说明队列未初始化，或者上面并发入队，入队失败了。
14 |         enq(node);
15 |         return node;
16 |     }
17 |
18 |     // 自旋方式入队
19 |     private Node enq(final Node node) {
20 |         for (;;) { // 死循环，保证入队一定成功
21 |             Node t = tail;
22 |             if (t == null) { // 队尾是null，说明得初始化队列
23 |                 if (compareAndSetHead(new Node()))
24 |                     tail = head;
25 |             } else {
26 |                 node.prev = t;
27 |                 if (compareAndSetTail(t, node)) {
28 |                     t.next = node;
29 |                     return t;
30 |                 }
31 |             }
32 |         }
33 |     }

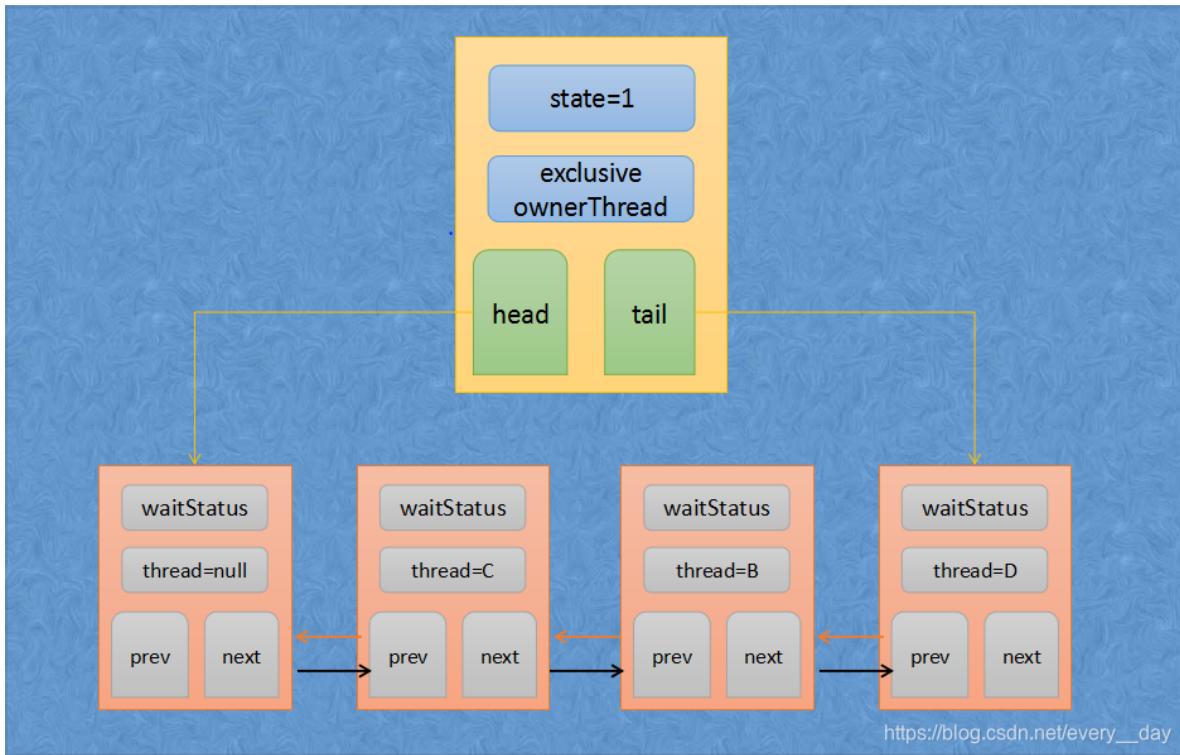
```

当发生竞争的线程 tryAcquire 抢锁失败，则进入 CLH 队列。

如果队列已经初始化，通过 CAS 方式加入到 CLH 队列的队尾。

如果队列未初始化，先初始化队列，通过自旋方式，若发生并发，第一个加入到队列头部，后续加入到队列尾部。

CLH 队列图示：



### 3、acquireQueued()

```

final boolean acquireQueued(final Node node, int arg) {
    //失败标记
    boolean failed = true;
    try {
        //中断标记
        boolean interrupted = false;
        for (;;) {
            // 获取当前节点的前一个节点，如果前一个节点为head，说明是head节点释放锁唤醒后续节点的操作，那么进行抢锁，
            // 这里有可能获取不到，因为后续进来的也会直接尝试加锁
            final Node p = node.predecessor();
            // 如果获取到了锁
            if (p == head && tryAcquire(arg)) {
                // 将head设置为自己
                setHead(node);
                // 将自己的next设置为null，方便GC
                p.next = null; // help GC
                // 设置为false代表获取成功了
                failed = false;
                return interrupted;
            }
            // 这里没有获取到锁，阻塞自己，然后等待下一次被唤醒，然后还会记录在park期间有没有收到中断信号。
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                interrupted = true;
        }
    } finally {
        //如果失败了
        if (failed)
            //取消获取锁
            cancelAcquire(node);
    }
}

```

CSDN @bulukezz

上面已经调用了 addWaiter() 方法把新节点加入队列了，这个方法是尝试让加入队列的新节点尝试去获取锁。

**Attention:** 注意 addWaiter(Node.EXCLUSIVE) 方法里的 CLH 队列图，可知当 CLH 队列初始化后，Head 节点里的 thread 必定是 null，Head 节点之后的节点里面的 thread 必定有值，所以 Head 之后的第一个节点是有资格去抢锁的。

因此，这段代码会先看当前线程节点的上一个节点是不是头节点，如果是头节点表示当前线程节点是排在第一个是有资格去抢锁的，因此尝试去抢锁。

当然，如果上个节点是 Head 节点且当前线程抢锁成功，将自己设置为 Head 节点，并且把 Head 节点的 next 设置为 null 方便 GC 回收【next 设置为 null 表示之前的 Head 结点和链表已经断开连接，则 GC 可以对之前的 Head 节点进行回收】若当前线程的上个节点不是 Head 节点，则没有资格去抢锁，需要调用 shouldParkAfterFailedAcquire() 和 parkAndCheckInterrupt() 来决定是否调 selfInterrupt() 方法【给线程设置一个中断标志，线程仍继续运行】。

**Attention：如果 shouldParkAfterFailedAcquire() 和 parkAndCheckInterrupt() 这两个方法都返回 true，其实也表示当前线程调用了 park() 方法，表示将当前线程置于“等待状态”，当线程被 park 时，它不会消耗 CPU 资源。**

#### 4、shouldParkAfterFailedAcquire()

```
1 static final class Node {  
2     // 线程被取消  
3     static final int CANCELLED = 1;  
4     // 等待队列中存在待被唤醒的挂起线程  
5     static final int SIGNAL = -1;  
6     // 当前线程在Condition队列中，未在AQS对列中  
7     static final int CONDITION = -2;  
8     // 解决JDK1.5的BUG。共享锁在释放资源后，若头节点为0，无法确定真的没有后继节点  
9     // 如果头节点为0，需要将头节点的状态改为 -3，当最新拿到锁资源的线程查看  
10    // 是否有后继节点并且为当前锁为共享锁，需唤醒排队的线程。  
11    static final int PROPAGATE = -3;  
12 }  
13  
14 // 获取锁资源失败，挂起线程  
15 private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {  
16     // 获取当前节点的上一个节点的状态  
17     int ws = pred.waitStatus;  
18     // 上一节点被挂起  
19     if (ws == Node.SIGNAL)  
20         // 返回true，挂起当前线程  
21         return true;  
22     if (ws > 0) {  
23         // 上一节点被取消，获取最近的线程挂起节点，  
24         // 并将当前节点的上一节点指向最近的线程挂起节点  
25         do {  
26             node.prev = pred = pred.prev;  
27         } while (pred.waitStatus > 0);  
28         // 最近线程挂起节点的下一节点指向当前节点  
29         pred.next = node;  
30     } else {  
31         // 上一节点状态小于等于0，存在线程处于等待状态，但未被挂起的场景  
32         // 通过CAS将处于等待的线程挂起，避免在挂起前节点获取到锁资源  
33         compareAndSetWaitStatus(pred, ws, Node.SIGNAL);  
34     }  
35     // 返回true，不挂起当前线程  
36     return false;  
37 }
```

如果当前线程节点的上一个节点不是 Head 节点，则没有资格去抢锁。

1、如果上一个节点的状态为 signal 表示被挂起状态，返回 true，表示去执行 parkAndCheckInterrupt() 方法将调用 park 方法将线程挂起。

2、如果上一个节点状态不为 signal 挂起状态，通过 do-while 继续往前节点循环遍历直至找到最深的非被取消的状态节点，将这个节点的上一个节点的 next 引用指向当前节点。如果上一个节点状态 <= 0，则表示为非取消状态，通过 CAS 将当前线程 waitStatus 状态设置为 SIGNAL 即挂起状态。最终返回 false，说明没必要调 selfInterrupt() 方法【给线程设置一个中断标志，线程仍继续运行】。

其实这个方法主要用于当前线程抢锁失败后，是否需要调用 park 方法将当前线程挂起，当然也只有当前线程节点的上一个节点为 SIGNAL 状态才会去调用 park 方法。反之去将上一个节点设置为 SIGNAL 状态。

## 5、parkAndCheckInterrupt() 阻塞线程

```
1 |     private final boolean parkAndCheckInterrupt() {  
2 |         LockSupport.park(this); // 阻塞线程  
3 |         return Thread.interrupted(); // 清除线程中断标记  
4 |     }
```

1 | 题外话  
2 | park() 阻塞了线程，有两种途径可以唤醒该线程：1) 被 unpark()；2) 被 interrupt()。  
3 |  
4 | Thread.interrupted() 当且仅当 线程被阻断时返回 true，它还会清除当前线程的中断标记位，

## 6、selfInterrupt()

先了解如下方法用法

### 1、interrupt () 方法

其作用是中断此线程（此线程不一定是当前线程，而是指调用该方法的 Thread 实例所代表的线程），但实际上只是给线程设置一个中断标志，线程仍会继续运行

### 2、interrupted () 方法

作用是测试当前线程是否被中断（检查中断标志），返回一个 boolean 并清除中断状态，第二次再调用时中断状态已经被清除，将返回一个 false。

### 3、isInterrupted () 方法

作用是只测试此线程是否被中断，不清除中断状态。

```
static void selfInterrupt() {  
    Thread.currentThread().interrupt();  
}
```

如果发生竞争的线程抢锁失败，且加入到 CLH 队列后的新节点的上一个节点为 Head 节点，则尝试抢锁，并不会调 selfInterrupt() 方法来设置中断标志，如果新节点的上一个节点并非 Head 节点，需要调 shouldParkAfterFailedAcquire() 和 parkAndCheckInterrupt() 方法都为 true 才会调 selfInterrupt()，给当前线程设置中断标志，并且在设置中断标志之前就已通过 park() 将线程挂起。

## ReentrantLock 解锁逻辑

```
1 |     public void unlock() {
2 |         sync.release(1);
3 |     }
4 |
5 |     public final boolean release(int arg) {
6 |         if (tryRelease(arg)) {
7 |             Node h = head;
8 |             if (h != null && h.waitStatus != 0)
9 |                 unparkSuccessor(h);
10 |            return true;
11 |        }
12 |        return false;
13 |    }
```

```
1 |     protected final boolean tryRelease(int releases) {
2 |         int c = getState() - releases;
3 |         if (Thread.currentThread() != getExclusiveOwnerThread())
4 |             throw new IllegalMonitorStateException();
5 |         boolean free = false;
6 |         if (c == 0) {
7 |             free = true;
8 |             setExclusiveOwnerThread(null);
9 |         }
10 |         setState(c);
11 |         return free;
12 |     }
```

```

private void unparkSuccessor(Node node) {

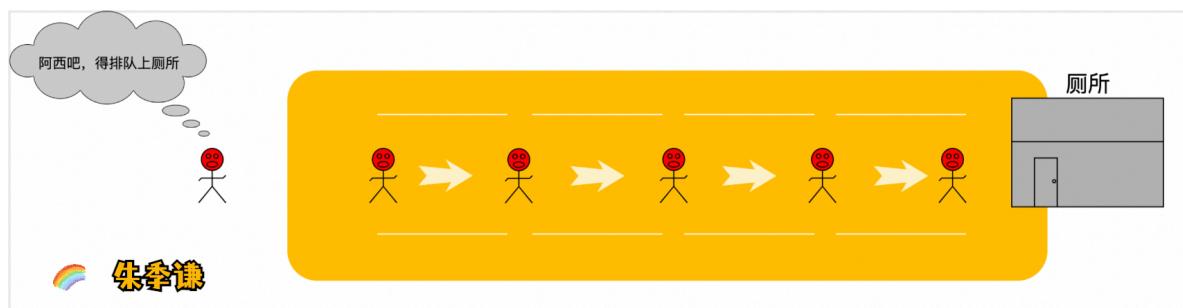
    int ws = node.waitStatus;
    if (ws < 0)
        compareAndSetWaitStatus(node, ws, 0); // 负值设置为0
    Node s = node.next; // 找到有效的后继节点
    if (s == null || s.waitStatus > 0) {
        s = null;
        for (Node t = tail; t != null && t != node; t = t.prev)
            if (t.waitStatus <= 0)
                s = t;
    }
    if (s != null)
        LockSupport.unpark(s.thread); // 将后继节点唤醒。
}

```

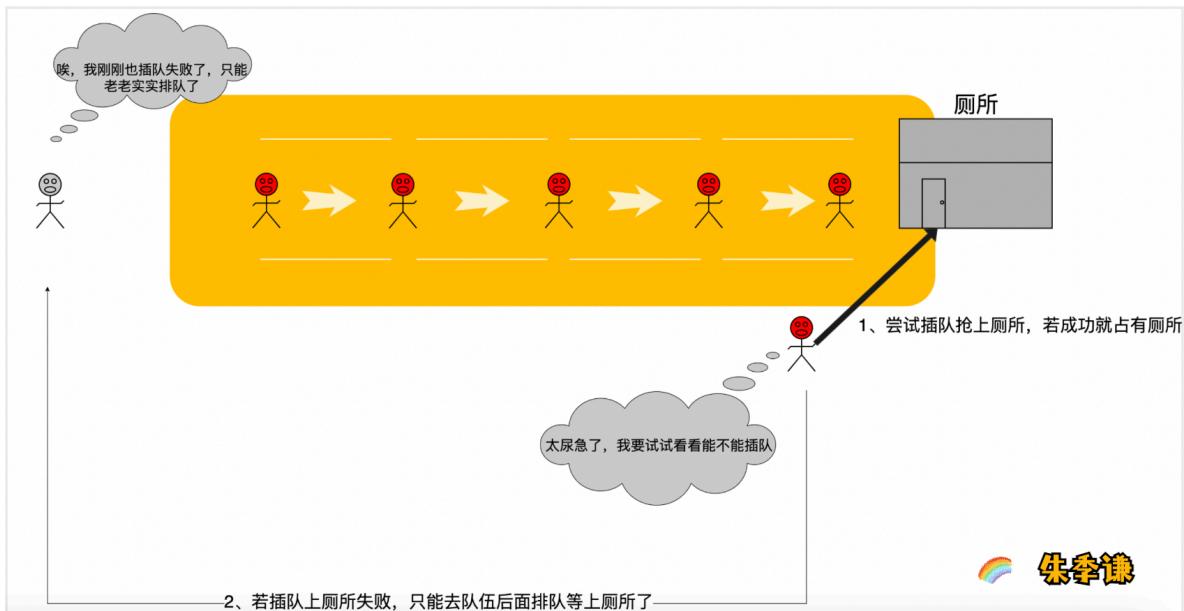
释放锁的逻辑比较简单，主要通过 tryRelease() 方法修改 state 值，最后通过 unparkSuccessor() 方法里的 unpark() 方法将后继节点唤醒【加锁的时候会对后继节点调用 park() 方法】。

## Reentrantlock 的公平锁和非公平锁实现

公平锁：N个线程去申请锁时，会按照先后顺序进入一个队列当中去排队，依次按照先后顺序获取锁。就像下图描述的上厕所的场景一样，先来的先占用厕所，后来的只能老老实实排队。



非公平锁：N个线程去申请锁，会直接去竞争锁，若能获取锁就直接占有，获取不到锁，再进入队列排队顺序等待获取锁。同样以排队上厕所打比分，这时候，后来的线程会先尝试插队看看能否抢占到厕所，若能插队抢占成功，就能使用厕所，若失败就得老老实实去队伍后面排队。



朱李谦

针对这两个概念，我们通过 ReentrantLock 底层源码来分析下：公平锁和非公平锁在 ReentrantLock 类当中锁怎样实现的。

ReentrantLock 内部实现的公平锁类是 FairSync，非公平锁类是 NonfairSync。

当 ReentrantLock 以无参构造器创建对象时，默认生成的是非公平锁对象

NonfairSync，只有带参且参数为 true 的情况下 FairSync，才会生成公平锁，若传参为 false 时，生成的依然是非公平锁，两者构造器源码结果如下



在实际开发当中，关于 ReentrantLock 的使用案例，一般是这个格式

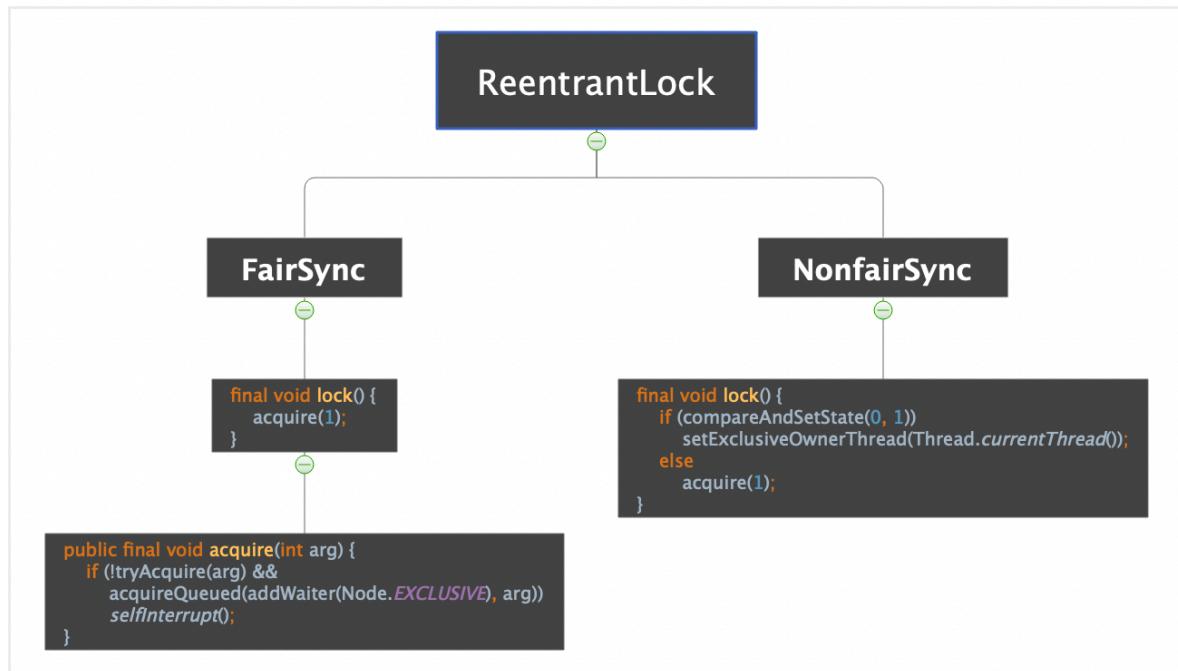
```

class X {
    private final ReentrantLock lock = new ReentrantLock();
    // ...
    public void m() {
        lock.lock();
        // block until condition holds
        try {
            // ... method body
        } finally {
            lock.unlock()
        }
    }
}

```

这时的lock指向的其实是NonfairSync对象，即非公平锁。

当使用lock.lock()对临界区进行占锁操作时，最终会调用到NonfairSync对象的lock()方法。根据上图可知，NonfairSync和FairSync两者的lock方法实现逻辑是不一样的，而体现其锁是否符合公平与否的地方，就是在两者的lock方法里。



可以看到，在非公平锁NonfairSync的上锁lock方法当中，若if(compareAndSetState(0,1))判断不满足，就会执行acquire(1)方法，该方法跟公平锁FairSync的lock方法里调用的acquire(1)其实是同一个，但方法里的tryAcquire具体实现又存在略微不同【不同在于公平锁的acquire (1) 方法多了hasQueuedPredecessors () 方法判断，下面再细聊】。

这里就呼应前文提到的非公平锁的概念——当N个线程去申请非公平锁，它们会直接去竞争锁，若能获取锁就直接占有，获取不到锁，再进入队列排队顺序等待获取锁。这里的“获取不到锁，再进入队列排队顺序等待获取锁”可以理解成——当线程过来直接竞争锁失败后，就会变成公平锁的形式，进入到一个队列当中，按照先后顺序排队去获取锁。

而if(compareAndSetState(0,1))语句块的逻辑，恰好就体现了“当N个线程去申请非公平锁，它们会直接去竞争锁，若能获取锁就直接占有”这句话的意思。

## 公平锁和非公平锁的实现区别

公平锁类和非公平锁类在lock方法里都会调用acquire(1)方法，区别在于两者的acquire(1)方法中，公平锁类的tryAcquire(1)多了一个hasQueuedPredecessors()方法判断，如下图

```
protected final boolean tryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (!hasQueuedPredecessors() &&
            compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    } else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0)
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}

final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    } else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0)
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

### ● 非公平锁的tryAcquire(1)方法

```
// tryAcquire 这个方法，非公平锁调用这个方法，传入参数是1，这个与可重入相关，待会再细说。
protected final boolean tryAcquire(int acquires) {
    return nonfairTryAcquire(acquires);
}

final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState(); // 获取 state状态
    if (c == 0) { // state是0，继续抢锁。刚进入lock 没抢到，现在可能锁已释放，有可能这次就抢成功了。
        if (compareAndSetState(0, acquires)) { // B C D 线程再次竞争拿锁
            setExclusiveOwnerThread(current);
            return true; // 拿到了锁，lock 方法就结束了。
        }
    }
    // 来抢锁的线程，本身就执有锁，说明是再次加锁，state 再加 1
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0) // overflow 内存溢出了，抛出异常
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false; // 抢锁失败
}
```

先分析非公平锁的!tryAcquire(arg)底层源码实现，该方法的整体逻辑是，通过getState()获取state状态值，判断是否已为0。若state等于0了，说明此时锁资源处于无锁状态，那么，当前线程就可以直接再执行一遍CAS原子抢锁操作，若CAS成功，说明已成功抢占锁。若state不为0，再判断当前线程是否与占有资源的锁为同一个线程，若同一个线程，那么就进行重入锁操作，即ReentrantLock支持同一个线程对资源的重复加锁，每次加锁，就对state值加1，解锁时，就对state解锁，直至减到0最后释放锁。

最后，若在该方法里，通过CAS抢占锁成功或者重入锁成功，那么就会返回true，若失败，就会返回false。

- 公平锁的tryAcquire (1) 方法

```
if (!hasQueuedPredecessors() &&
    compareAndSetState(0, acquires)) {
    setExclusiveOwnerThread(current);
    return true;
}
```

上面提到了多了一个hasQueuedPredecessors () 方法，看看下图这个方法做了什么逻辑：

```
public final boolean hasQueuedPredecessors() {
    // The correctness of this depends on head being initialized
    // before tail and on head.next being accurate if the current
    // thread is first in queue.
    Node t = tail; // Read fields in reverse initialization order
    Node h = head;
    Node s;
    return h != t &&
        ((s = h.next) == null || s.thread != Thread.currentThread());
}
```

hasQueuedPredecessors() 表示：

- 若等待队列为空，即未初始化，返回 false；
- 若等待队列已初始化，哨兵结点没有后继结点，返回 false；
- 若哨兵结点有后继结点，后继结点的线程是当前线程，返回 false；
- 其它情况返回 true

意思总结为：**要么 CLH 队列为空或者头结点的下一个节点是自己才会去通过 CAS 抢锁**，反之表示tryAcquire (1) 失败，直接走

acquireQueued(addWaiter(Node.EXCLUSIVE), arg) 方法尝试去加入到线程队列里。

因此公平锁下，除非上面红字情况，否则就乖乖的去加入到线程队列里。

## 总结

- 非公平锁：在调用 lock() 方法时候，一开始就会尝试通过 CAS 去抢锁，抢不到会进入到acquire (1) 方法，然后在acquire (1) 方法的tryAcquire(1) 方法看看state是否为0，如果为0就再次去通过CAS抢锁，抢不到的话就乖乖去加入到线程队列中去排队抢锁。
- 公平锁：在调用 lock() 方法时候，不会一开始就会想通过 CAS 去抢锁而是直接调acquire(1)方法，在acquire(1)方法的tryAcquire(1)方法里看看state是不是为0，如果为0时，**且 CLH 队列为空或者头结点的下一个节点是自己**，这样的话才会去尝试用 CAS 抢锁，反之都要乖乖去加入到线程队列中排队抢锁，别妄图不入队列就尝试抢锁。

所以非公平锁和公平锁的最大区别在于公平锁在 tryAcquire(1) 比非公平锁多了一个 hasQueuedPredecessors() 方法的判断。

## ThreadLocal

### 作用

ThreadLocal类主要解决的就是让每个线程绑定自己的值，可以将ThreadLocal类形象的比喻成存放数据的盒子，盒子中可以存储每个线程的私有数据。如果你创建了一个ThreadLocal变量，那么访问这个变量的每个线程都会有这个变量的本地副本，这也是ThreadLocal变量名的由来。他们可以使用get()和set()方法来获取默认值或将其值更改为当前线程所存的副本的值，从而避免了线程安全问题。

### 原理

```
public void set(T value) {  
    Thread t = Thread.currentThread();  
    ThreadLocalMap map = getMap(t);  
    if (map != null)  
        map.set(this, value);  
    else  
        createMap(t, value);  
}
```

```
ThreadLocalMap getMap(Thread t) {  
    return t.threadLocals;  
}
```

申明的变量是放在了当前线程的ThreadLocalMap中，并不是存在ThreadLocal上，ThreadLocal可以理解为只是ThreadLocalMap的封装，传递了变量值。

ThreadLocal类中可以通过Thread.currentThread()获取到当前线程对象后，直接通过getMap(Thread t)可以访问到该线程的ThreadLocalMap对象。

每个Thread中都具备一个ThreadLocalMap，而ThreadLocalMap可以存储以ThreadLocal为key，Object对象为value的键值对。

【Thread类含有ThreadLocalMap变量，当线程start()操作ThreadLocal修饰的变量时，实际上是从Thread类获取ThreadLocalMap变量(不存在则new一个出来)，然后把当前ThreadLocal修饰的变量的引用当做key存放到ThreadLocalMap，

value则为实际存储的数值，所以每个ThreadLocal地址引用不一样，达到了线程隔离的作用，即 ThreadLocal其实是与线程绑定的一个变量】

- ( 1 ) 每个Thread线程内部都有一个Map (ThreadLocalMap)
- ( 2 ) Map里面存储ThreadLocal对象 ( key ) 和线程的变量副本 ( value )
- ( 3 ) Thread内部的Map是由ThreadLocal维护的，由ThreadLocal负责向map获取和设置线程的变量值。
- ( 4 ) 对于不同的线程，每次获取副本值时，别的线程并不能获取到当前线程的副本值，形成了副本的隔离，互不干扰。

## ThreadLocal 内存泄露问题是怎么导致的？

ThreadLocalMap 中使用的 key 为 ThreadLocal 的弱引用，而 value 是强引用。所以，如果 ThreadLocal 没有被外部强引用的情况下，在垃圾回收的时候，key 会被清理掉，而 value 不会被清理掉。

这样一来，ThreadLocalMap 中就会出现 key 为 null 的 Entry。假如我们不做任何措施的话，value 永远无法被 GC 回收，这个时候就可能会产生内存泄露。

ThreadLocalMap 实现中已经考虑了这种情况，在调用 set()、get()、remove() 方法的时候，会清理掉 key 为 null 的记录。使用完 ThreadLocal 方法后最好手动调用 remove() 方法

## 线程之间的局部变量是隔离的，那么为什么还需要用 threadlocal？

简单来说，答案是：局部变量的隔离是“方法级别”的，而 ThreadLocal 提供的是“线程级别”的隔离。

它们的生命周期和作用域完全不同，解决了不同的问题。让我们来详细分解一下。

### 1. 局部变量 (Local Variables)

- 隔离性来源：Java 虚拟机规范规定，每个线程都有自己独立的虚拟机栈 (**VM Stack**)。
- 生命周期：当一个方法被调用时，Java 虚拟机会为该方法在当前线程的栈中创建一个栈帧 (**Stack Frame**)。这个方法内部声明的所有局部变量（包括基本类型和对象的引用）都存放在这个栈帧中。
- 作用域：局部变量的作用域仅限于该方法内部。一旦方法执行完毕，对应的栈帧就会被弹出并销毁，里面的所有局部变量也随之消失。
- 核心痛点：正因为其生命周期和方法绑定，我们无法在一个方法的执行过程中，去访问另一个方法里的局部变量。如果想跨方法传递数据，最常规的方式就是通过方法参数。

例子：

Generated java

```
public class MyService {
    public void methodA() {
        String user = "Alice"; // user是methodA的局部变量
        methodB();
    }

    public void methodB() {
        // 在这里，能访问到methodA里的user变量吗？
        // 答案是：不能！因为它和methodB毫无关系。
        // 如果methodB需要user信息，必须通过参数传进来，如 methodB("Alice")
    }
}
```

Use code with caution. Java

## 2. ThreadLocal

ThreadLocal的出现正是为了解决局部变量的这个“痛点”。它提供了一种机制，让你可以在一个线程的**任何地方**（任何方法调用深度）都能存取一个值，而这个值对其他线程是完全隔离的。

- **隔离性来源：**每个 Thread 对象内部都有一个 ThreadLocalMap 类型的成员变量。当你通过一个 ThreadLocal 变量存取数据时，实际上是把这个 ThreadLocal 实例作为 Key，把你存的数据作为 Value，存入了**当前线程自己的 ThreadLocalMap 中**。
- **生命周期：**只要线程本身是存活的，并且 ThreadLocal 实例是可达的，那么它存储在线程 ThreadLocalMap 里的值就不会被回收。它的生命周期和**线程绑定**，而不是和某个方法绑定。
- **作用域：**它的作用域是整个线程的执行过程。你可以在线程执行的入口处（如一个 Web 请求的 Filter 或 Interceptor）set 一个值，然后在业务逻辑的深层方法中 get 这个值，无需通过方法参数层层传递。

例子 (解决上面的痛点):

Generated java

```

public class UserContextHolder {
    // 创建一个ThreadLocal实例，通常是static final的
    public static final ThreadLocal<String> holder = new ThreadLocal<>();
}

public class MyService {
    public void methodA(String user) {
        // 在调用链路的开始处，将用户信息放入ThreadLocal
        UserContextHolder.holder.set(user);
        System.out.println("Method A: 当前用户是 " + UserContextHolder.holder.get());
        methodB();
        // 在调用链路的最后，务必remove，防止内存泄漏
        UserContextHolder.holder.remove();
    }

    public void methodB() {
        // 在任何地方，都可以轻松获取到当前线程的用户信息
        String user = UserContextHolder.holder.get();
        System.out.println("Method B: 成功获取到用户 " + user);
    }
}

// 模拟调用
new Thread(() -> new MyService().methodA("Alice")).start();
new Thread(() -> new MyService().methodA("Bob")).start();

□ ⌂ Use code with caution. Java

```

输出：

Generated code

```

Method A: 当前用户是 Alice
Method B: 成功获取到用户 Alice
Method A: 当前用户是 Bob
Method B: 成功获取到用户 Bob

□ ⌂ Use code with caution.

```

你看，methodB 不需要任何参数，就能获取到 methodA 设置的值，并且两个线程之间的数据 ("Alice" 和 "Bob") 是完全隔离的。

## 总结与对比

特性	局部变量	ThreadLocal
数据隔离级别	方法级别	线程级别
作用域	单个方法内部	整个线程的生命周期
生命周期	随方法调用开始，随方法结束而销毁	随线程开始，随线程结束或手动remove而销毁
解决的问题	方法内部的临时数据存储	跨方法、跨类的线程内数据共享，避免参数传来传去（也称为“参数污染”）
典型应用场景	循环变量、方法内临时计算结果	1. 存储请求上下文信息（如用户信息、事务ID） 2. 管理非线程安全的工具类（如 SimpleDateFormat） 3. 数据库连接、Session管理

结论：  
 “线程之间的局部变量是隔离的”这句话是完全正确的，但这仅仅是方法栈层面的隔离。ThreadLocal 解决的是一个更高维度的问题：如何在同一个线程的复杂调用链中方便、安全地传递数据，同时又保持这个数据在多线程间的隔离性。

**总结：通过把想要的前置数据放入到 threadLocal，可以使得线程在运行过程随时随时随地获取（例如每个请求的网省 Code，什么时候获取都可以，只要执行的线程还存活）。**

## ThreadLocal为什么key是弱引用而value是强引用?

这是一个非常经典且深入的ThreadLocal设计问题。ThreadLocal的ThreadLocalMap中将Key(即ThreadLocal实例)设计为弱引用，而Value设计为强引用，主要是为了在一定程度上避免内存泄漏。

我们来一步步拆解这个设计背后的逻辑和原因。

### 背景：ThreadLocal是如何工作的？

首先要回顾一下ThreadLocal的内部结构：

1. 每个Thread对象都有一个成员变量threadLocals，它的类型是ThreadLocal.ThreadLocalMap。
2. ThreadLocalMap是ThreadLocal的一个静态内部类，它的内部有一个Entry数组，用于存储数据。
3. 这个Entry继承了WeakReference<ThreadLocal<?>>，它的结构是(Key, Value)，其中：
  - **Key**就是ThreadLocal对象本身，并且它被包装成一个弱引用(**Weak Reference**)。
  - **Value**就是我们通过threadLocal.set(value)方法设置的值，它是一个强引用(**Strong Reference**)。

The screenshot shows a Java code editor window with the title 'Generated java'. The code is a simplified version of the ThreadLocalMap.Entry class:

```
// ThreadLocalMap.Entry 的简化源码
static class Entry extends WeakReference<ThreadLocal<?>> {
    /** The value associated with this ThreadLocal. */
    Object value;

    Entry(ThreadLocal<?> k, Object v) {
        super(k); // Key是弱引用
        value = v; // Value是强引用
    }
}
```

Below the code, there is a note: 'Use code with caution.' and a 'Java' tab.

### 为什么Key要用弱引用？

这主要是为了解决一个问题：当外部代码不再持有ThreadLocal对象的强引用时，这个ThreadLocal实例能够被垃圾回收器(GC)正常回收，从而让它所对应的Entry有机会被清理。

我们来考虑两种情况的对比：

#### ○ 情况一：如果Key是强引用（实际没有这样设计）

1. 我们创建了一个ThreadLocal实例：ThreadLocal<MyObject>  
myThreadLocal = new ThreadLocal<>();
2. 在一个线程中设置了值：myThreadLocal.set(new MyObject());
3. 这时，在当前线程的ThreadLocalMap中，形成了一条强引用链：  
当前线程(Thread) -> ThreadLocalMap -> Entry -> Key (myThreadLocal实例)
4. 现在，我们在代码中释放了对myThreadLocal的引用，比如myThreadLocal = null;。
5. 问题来了：尽管我们已经不再需要myThreadLocal这个对象了，但是因为线程的ThreadLocalMap中的Entry还强引用着它，所以GC无法回收myThreadLocal实例。
6. 只要这个线程还在运行(比如线程池中的核心线程)，这条引用链就一直存在，

myThreadLocal 实例和它对应的 Value (那个 new MyObject()) 都无法被回收, 从而导致了内存泄漏。

## ○ 情况二: Key 是弱引用 (JDK 的实际设计)

1. 和上面一样, 我们执行了 myThreadLocal = null;, 断开了外部对 ThreadLocal 实例的强引用。
2. 现在, myThreadLocal 实例只被 Entry 的 Key 这个弱引用指向。
3. 当下一次垃圾回收 (GC) 发生时, GC 会忽略弱引用。由于没有其他强引用指向 myThreadLocal 实例, 这个实例就会被成功回收。
4. 当 myThreadLocal 实例被回收后, ThreadLocalMap 中那个 Entry 的 Key 就变成了 null。
5. 这样, ThreadLocalMap 在后续操作 (如 get(), set(), remove()) 时, 会检查到这些 Key 为 null 的 Entry, 并将它们从 Map 中清除 (这个过程被称为启发式清理), 从而释放掉其对应的强引用的 Value 【如果 key 为 null, 后续调用了 get() 之类方法, 会把强引用的 value 也删除掉】。

**小结:** 将 Key 设置为弱引用, 打破了 ThreadLocalMap 对 ThreadLocal 实例的强引用关系, 使得 ThreadLocal 实例在不再被外部使用时能够被顺利回收, 这是防止内存泄漏的第一道防线。

## 为什么 Value 不能是弱引用?

既然弱引用这么好, 为什么 Value 不也用弱引用呢?

**答案是: 没有意义, 而且会出问题。**

设想一下, 如果 Value 也是弱引用:

- 1、我们创建一个对象, 并把它设置到 ThreadLocal 中:

```
Generated java

MyObject myObject = new MyObject(); // myObject是一个强引用
myThreadLocal.set(myObject);

□ ↓ Use code with caution. ^ Java
```

- 2、ThreadLocalMap 中的 Value 弱引用地指向堆中的 MyObject 实例。

- 3、接着, 我们将外部的强引用释放掉:

```
Generated java

myObject = null;

□ ↓ Use code with caution. ^ Java
```

- 4、现在, 堆中的 MyObject 实例只被 ThreadLocalMap 中的 Value 这个弱引用指向。

- 5、**问题来了:** 当下一次 GC 发生时, 这个 MyObject 实例就会被回收。

- 6、那么, 当我们稍后尝试通过 myThreadLocal.get() 去获取值时, 只会得到一个 null。

这就完全违背了我们使用 ThreadLocal 的初衷。我们就是希望把一个对象寄存到 ThreadLocal 里, 只要线程还活着, 我就能在任何时候把它取出来用。如果存进去的对象, 因为没有其他外部强引用就莫名其妙地被 GC 回收了, 那 ThreadLocal 就失去了它作为“线程局部变量”的意义 (\*\*结果可能造成 get 的时候返回了 null)。  
所以, Value 必须是强引用, 以保证在线程的生命周期内, 只要我们没有显式地 remove() 它, 它就一直存在。

## 最终的结论与最佳实践

- **Key 使用弱引用：**是为了在 ThreadLocal 实例本身被开发者丢弃后，GC 能够回收它，从而让 ThreadLocalMap 有机会清理掉对应的整个 Entry，这是为了防止因 ThreadLocal 对象本身导致的内存泄漏。
- **Value 使用强引用：**是为了保证存入 ThreadLocal 的值的生命周期和线程绑定，不会因为没有外部强引用而被 GC 意外回收，这是为了保证 ThreadLocal 功能的正常实现。

### 重要的最佳实践：

这种“Key 弱 Value 强”的设计虽然能在一定程度上防止内存泄漏，但它并非万无一失。如果一个线程执行完任务后长时间不消亡（比如在线程池中），而我们忘记调用 threadLocal.remove()，那么那些 Key 已经被回收（变为 null）但 Value 依然被强引用的“脏 Entry”还是会存在，导致 Value 对象无法被回收，这依然是内存泄漏。因此，最保险、最规范的做法是：

在使用了 ThreadLocal 之后，必须在 finally 代码块中调用 remove() 方法，手动清理数据。

```
Generated java
try {
    myThreadLocal.set(someValue);
    // ... 业务逻辑 ...
} finally {
    myThreadLocal.remove(); // 这是最重要的好习惯!
}
```

Use code with caution.

总结：key 为弱引用，是因为当不再引用的时候，threadLocal 能被回收。而 value 为强引用是因为我需要在线程存活的时候，能随时去拿值（既然使用 threadLocal，就是为了在线程生命周期内，能随时获取 value，key 被回收了大不了再重新建立引用）。如果 value 为弱引用，那我可能因为 GC 而拿到 value 为 null，这显然不是我所期望的效果。当然如果线程是线程池里的核心线程，一直都不会被销毁，那么一大堆强引用的 value（一个线程可以放到多个 new 出来的 threadLocalMap）也可能会导致内存泄漏，最好还是用 remove 保证不会内存泄漏。

---

## 线程池

### 什么是线程池？

顾名思义，线程池就是管理一系列线程的资源池。当有任务要处理时，直接从线程池中获取线程来处理，处理完之后线程并不会立即被销毁，而是等待下一个任务。

### 为什么要用线程池？

池化技术想必大家已经屡见不鲜了，线程池、数据库连接池、HTTP 连接池等等都是对这个思想的应用。池化技术的思想主要是为了减少每次获取资源的消耗，提高对资源的利用率。

- 降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
- 提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。

- 提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

## 如何创建线程池？

方式一：通过 **ThreadPoolExecutor** 构造函数来创建（推荐）。

方式二：通过 **Executor** 框架的工具类 **Executors** 来创建。

- **FixedThreadPool**: 该方法返回一个固定线程数量的线程池。该线程池中的线程数量始终不变。当有一个新的任务提交时，线程池中若有空闲线程，则立即执行。若没有，则新的任务会被暂存在一个任务队列中，待有线程空闲时，便处理在任务队列中的任务。
- **SingleThreadExecutor**: 该方法返回一个只有一个线程的线程池。若多余一个任务被提交到该线程池，任务会被保存在一个任务队列中，待线程空闲，按先入先出的顺序执行队列中的任务。
- **CachedThreadPool**: 该方法返回一个可根据实际情况调整线程数量的线程池。初始大小为 0。当有新任务提交时，如果当前线程池中没有线程可用，它会创建一个新的线程来处理该任务。如果在一段时间内（默认为 60 秒）没有新任务提交，核心线程会超时并被销毁，从而缩小线程池的大小。
- **ScheduledThreadPool**: 该方法返回一个用来在给定的延迟后运行任务或者定期执行任务的线程池。

《阿里巴巴 Java 开发手册》中强制线程池不允许使用 **Executors** 去创建，而是通过 **ThreadPoolExecutor** 构造函数的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险

**Executors** 返回线程池对象的弊端如下：

- **FixedThreadPool 和 SingleThreadExecutor**: 使用的是无界的 **LinkedBlockingQueue**，任务队列最大长度为 **Integer.MAX\_VALUE**，可能堆积大量的请求，从而导致 OOM。
- **CachedThreadPool**: 使用的是同步队列 **SynchronousQueue**，允许创建的线程数量为 **Integer.MAX\_VALUE**，如果任务数量过多且执行速度较慢，可能会创建大量的线程，从而导致 OOM。
- **ScheduledThreadPool 和 SingleThreadScheduledExecutor**: 使用的无界的延迟阻塞队列 **DelayedWorkQueue**，任务队列最大长度为 **Integer.MAX\_VALUE**，可能堆积大量的请求，从而导致 OOM。

## 线程池常见参数有哪些？如何解释？

**ThreadPoolExecutor** 3 个最重要的参数：

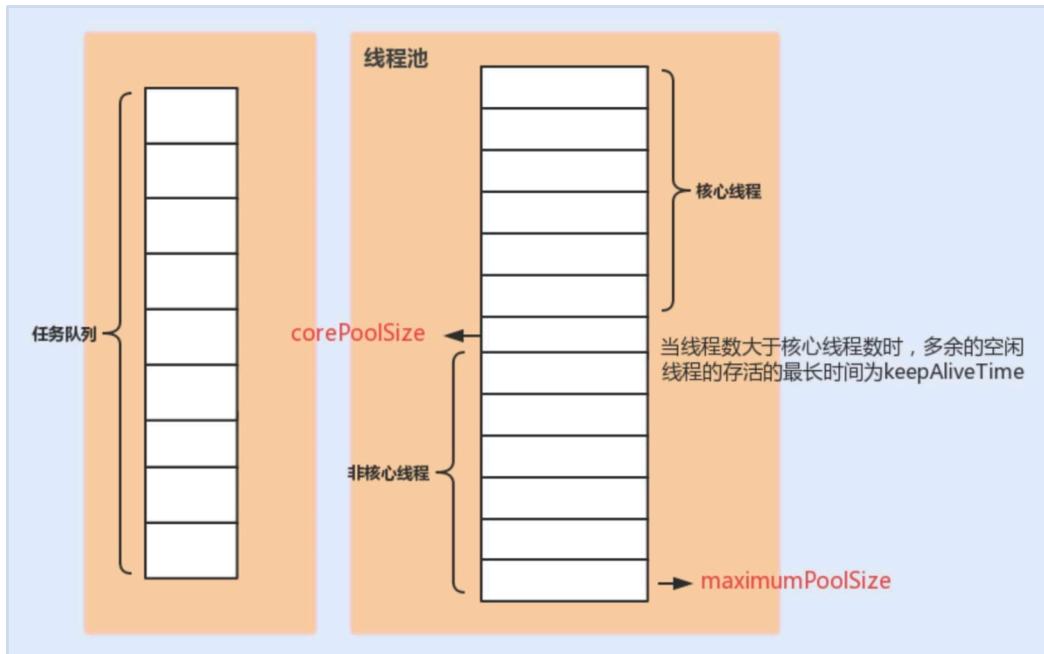
- **corePoolSize** : 任务队列未达到队列容量时，最大可以同时运行的线程数量。
- **maximumPoolSize** : 任务队列中存放的任务达到队列容量的时候，当前可以同时运行的线程数量变为最大线程数【core用完了，且队列任务塞满了，就会启动 maximum 数量去创建线程，如果 maximum 数量也达标了，就会丢弃任务】。
- **workQueue**: 新任务来的时候会先判断当前运行的线程数量是否达到核心线程数，如果达到的话，新任务就会被存放在队列中。

**ThreadPoolExecutor** 其他常见参数：

- **keepAliveTime**: 线程池中的线程数量大于 **corePoolSize** 的时候，如果这时没有新的任务提交，多余的空闲线程不会立即销毁，而是会等待，直到等待的时间

超过了 `keepAliveTime` 才会被回收销毁，线程池回收线程时，会对核心线程和非核心线程一视同仁，直到线程池中线程的数量等于 `corePoolSize`，回收过程才会停止。

- **unit** : `keepAliveTime` 参数的时间单位。
- **threadFactory** : `executor` 创建新线程的时候会用到。
- **handler** : 饱和策略。关于饱和策略下面单独介绍一下。

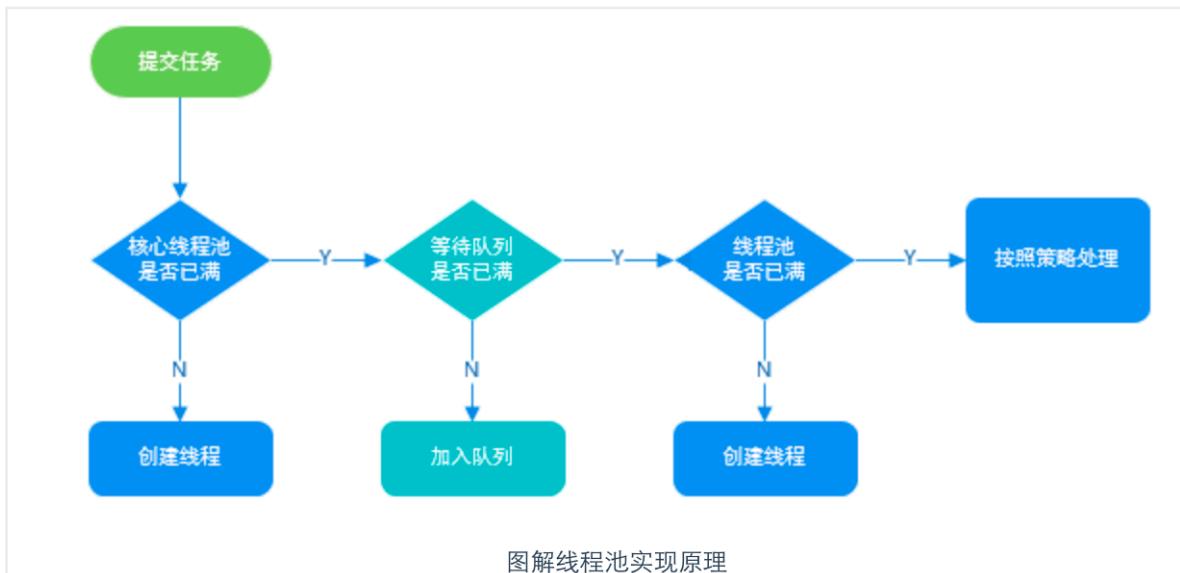


## 线程池常用的阻塞队列有哪些？

不同的线程池会选用不同的阻塞队列，我们可以结合内置线程池来分析。

- 容量为 `Integer.MAX_VALUE` 的 `LinkedBlockingQueue` (无界队列): `FixedThreadPool` 和 `SingleThreadExecutor`。`FixedThreadPool` 最多只能创建核心线程数的线程 (核心线程数和最大线程数相等)，`SingleThreadExecutor` 只能创建一个线程 (核心线程数和最大线程数都是 1)，二者的任务队列永远不会被放满。
- `SynchronousQueue` (同步队列): `CachedThreadPool`。  
`SynchronousQueue` 没有容量，不存储元素，目的是保证对于提交的任务，如果有空闲线程，则使用空闲线程来处理；否则新建一个线程来处理任务。也就是说，`CachedThreadPool` 的最大线程数是 `Integer.MAX_VALUE`，可以理解为线程数是可以无限扩展的，可能会创建大量线程，从而导致 OOM。
- `DelayedWorkQueue` (延迟阻塞队列): `ScheduledThreadPool` 和 `SingleThreadScheduledExecutor`。`DelayedWorkQueue` 的内部元素并不是按照放入的时间排序，而是会按照延迟的时间长短对任务进行排序，内部采用的是“堆”的数据结构，可以保证每次出队的任务都是当前队列中执行时间最靠前的。`DelayedWorkQueue` 添加元素满了之后会自动扩容原来容量的 1/2，即永远不会阻塞，最大扩容可达 `Integer.MAX_VALUE`，所以最多只能创建核心线程数的线程。

## 线程池处理任务的流程了解吗？



1. 如果当前运行的线程数小于核心线程数，那么就会新建一个线程来执行任务。
2. 如果当前运行的线程数等于或大于核心线程数，但是小于最大线程数，那么就将该任务放入到任务队列里等待执行。
3. 如果向任务队列投放任务失败（任务队列已经满了），但是当前运行的线程数是小于最大线程数的，就新建一个线程来执行任务。
4. 如果当前运行的线程数已经等同于最大线程数了，新建线程将会使当前运行的线程超出最大线程数，那么当前任务会被拒绝，饱和策略会调用 `RejectedExecutionHandler.rejectedExecution()` 方法。

## 线程池的拒绝策略

线程池的拒绝策略主要有四种，它们在 Java 的 `ThreadPoolExecutor` 中定义，并在任务队列已满且线程数达到最大限制时，用于处理新提交的任务。以下是这四种拒绝策略的详细介绍：

1. **AbortPolicy**：这是默认的拒绝策略。当新任务被提交但无法执行时，它会直接抛出 `RejectedExecutionException` 异常。这种策略适用于对任务丢失敏感的场景，因为它会立即通知调用者任务被拒绝。
2. **DiscardPolicy**：当新任务被提交但无法执行时，它会直接丢弃这个任务，并且不会抛出任何异常。这种策略适用于对任务丢失不敏感的场景，但可能会造成数据丢失。
3. **DiscardOldestPolicy**：当新任务被提交但无法执行时，它会丢弃队列中等待时间最长的任务（即最老的任务），然后尝试执行新提交的任务。这种策略适用于对新任务优先级较高的场景。
4. **CallerRunsPolicy**：当新任务被提交但无法执行时，该策略会直接在提交任务的线程中运行被拒绝的任务。如果线程池已经关闭，则任务将被丢弃。这种策略适用于希望调用者自己处理被拒绝的任务的场景。

## 如何设定线程池的大小？

- 如果我们设置的线程池数量太小的话，如果同一时间有大量任务/请求需要处理，可能会导致大量的请求/任务在任务队列中排队等待执行，甚至会出现任务队列满了之后任务/请求无法处理的情况，或者大量任务堆积在任务队列导致

OOM。这样很明显是有问题的，CPU 根本没有得到充分利用。

- 如果我们设置线程数量太大，大量线程可能会同时在争取 CPU 资源，这样会导致大量的上下文切换，从而增加线程的执行时间，影响了整体执行效率。

## 下上下文切换

多线程编程中一般线程的个数都大于 CPU 核心的个数，而一个 CPU 核心在任意时刻只能被一个线程使用，为了让这些线程都能得到有效执行，CPU 采取的策略是为每个线程分配时间片并轮转的形式。当一个线程的时间片用完的时候就会重新处于就绪状态让给其他线程使用，这个过程就属于一次上下文切换。概括来说就是：当前任务在执行完 CPU 时间片切换到另一个任务之前会先保存自己的状态，以便下次再切换回这个任务时，可以再加载这个任务的状态。**任务从保存到再加载的过程就是一次上下文切换。**

上下文切换通常是计算密集型的。也就是说，它需要相当可观的处理器时间，在每秒几十上百次的切换中，每次切换都需要纳秒量级的时间。所以，上下文切换对系统来说意味着消耗大量的 CPU 时间，事实上，可能是操作系统中时间消耗最大的操作。

### 如何判断是 CPU 密集任务还是 IO 密集任务？

CPU 密集型简单理解就是利用 CPU 计算能力的任务比如你在内存中对大量数据进行排序。但凡涉及到网络读取，文件读取这类都是 IO 密集型，这类任务的特点是 CPU 计算耗费时间相比于等待 IO 操作完成的时间来说很少，大部分时间都花在了等待 IO 操作完成上。

所以如果是 CPU 密集型的任务，线程数设置为 CPU 核数 +1【线程越多抢占 CPU 资源情况越频繁，这样设置可以减少线程抢占 CPU 资源而出现的频繁上下文切换，+1 当个兜底使用】，IO 密集型的任务设置线程数为 CPU 核数 \*2【瓶颈在于 IO 处理，所以线程数可以多设置点】

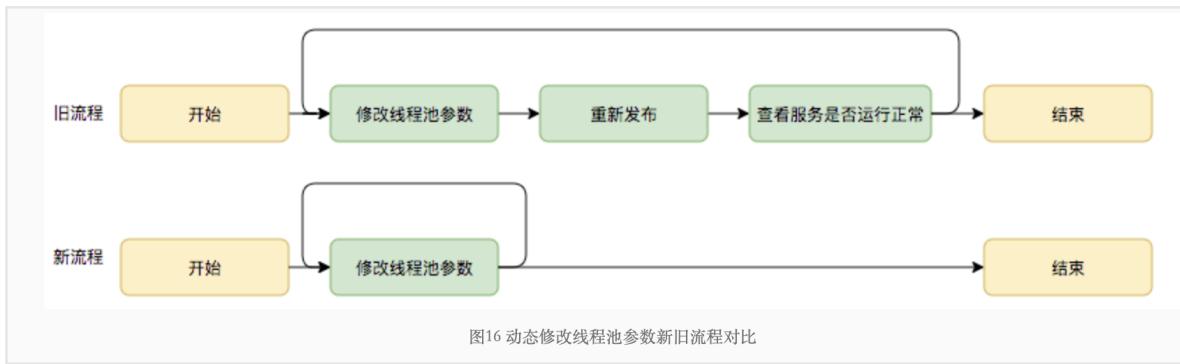
Java 线程池实现原理及其在美团业务中的实践：<https://tech.meituan.com/2020/04/02/java-pooling-practice-in-meituan.html>

前半幅篇章讲解的是线程池使用姿势以及原理，后半部分讲解的是美团如何做到动态设置线程池参数 + 告警 + 监控【毕竟如何设置线程池参数业界目前也没有给出一个确定的规则】

## 动态线程池（参考美团博客总结）

### 出现原因：

在配置线程池参数（例如 corePoolSize、maximumPoolSize、workQueue）时，我们一般基于是分析任务的性质是 IO 密集型还是 CPU 密集型，再决定该如何合理配置参数，但是应用中很难准确的评估实际运行情况（如果没配置好，可能会出现 corePoolSize 设置过少导致在线上高并发情况下页面响应变慢，或者 workQueue 设置过长导致无法应用上 maximumPoolSize，进而任务导致任务堆积在队列中。），因此要考虑在发生故障时，如何可以快速调整从而缩短故障恢复的时间。基于这个思考背景，美团团队考虑将线程池的参数从代码中迁移到分布式配置中心上，实现线程池参数可动态配置和即时生效，线程池参数动态化前后的参数修改流程对比如下：



## 动态线程池设计方向：

1. 简化线程池配置：线程池构造参数有 8 个，但是最核心的是 3 个：`corePoolSize`、`maximumPoolSize`、`workQueue`，它们最大程度地决定了线程池的任务分配和线程分配策略。考虑到在实际应用中我们获取并发性的场景主要是两种：(1) 并行执行子任务，提高响应速度。这种情况下，应该使用同步队列，没有什么任务应该被缓存下来，而是应该立即执行。(2) 并行执行大批量任务，提升吞吐量。这种情况下，应该使用有界队列，使用队列去缓冲大批量的任务，队列容量必须声明，防止任务无限制堆积。所以线程池只需要提供这三个关键参数的配置，并且提供两种队列的选择，就可以满足绝大多数的业务需求，*Less is More*。
2. 参数可动态修改：为了解决参数不好配，修改参数成本高等问题。在 Java 线程池留有高扩展性的基础上，封装线程池，允许线程池监听同步外部的消息，根据消息进行修改配置。将线程池的配置放置在平台侧，允许开发同学简单的查看、修改线程池配置。
3. 增加线程池监控：对某事物缺乏状态的观测，就对其改进无从下手。在线程池执行任务的生命周期添加监控能力，帮助开发同学了解线程池状态。

## 动态线程池提供功能如下：

**动态调参：**支持线程池参数动态调整、界面化操作；包括修改线程池核心大小、最大核心大小、队列长度等；参数修改后及时生效。

JDK原生线程池ThreadPoolExecutor提供了如下几个public的setter方法，如下图所示：

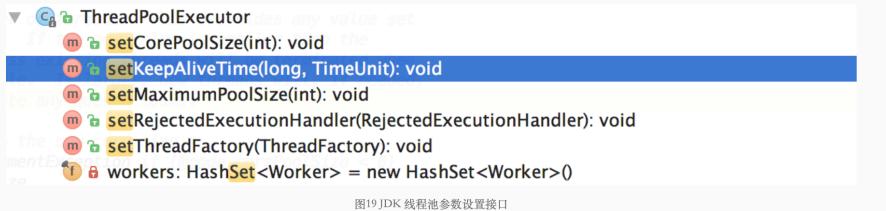


图19 JDK 线程池参数设置接口

**任务监控：**支持应用粒度、线程池粒度、任务粒度的 Transaction 监控；可以看到线程池的任务执行情况、最大任务执行时间、平均任务执行时间、95/99 线等。

**负载告警：**线程池队列任务积压到一定值的时候会通过大象（美团内部通讯工具）告知应用开发负责人；当线程池负载数达到一定阈值的时候会通过大象告知应用开发负责人。

**操作监控：**创建/修改和删除线程池都会通知到应用的开发负责人。

**操作日志：**可以查看线程池参数的修改记录，谁在什么时候修改了线程池参数、修改前的参数值是什么。

**权限校验：**只有应用开发负责人才能够修改应用的线程池参数。

---

## CompletableFuture

<https://tech.meituan.com/2022/05/12/principles-and-practices-of-completablefuture.html>

<https://www.jianshu.com/p/abfa29c01e1d>

---

## CAS

### 什么是 CAS? 【compare and swap】

CAS 操作包含三个操作数 — 内存位置 (V)、预期原值 (A) 和新值 (B)。如果内存位置的值与预期原值相匹配，那么处理器会自动将该位置值更新为新值。否则，处理器不做任何操作。

我们假设内存中原数据 V，旧的预期值 A，需要修改的新值 B。

1. 比较 A 与 V 是否相等 (比较)
2. 如果比较相等，将 B 写入 V。 (交换)
3. 返回操作是否成功。

### 解决了什么问题？

- 用于解决多线程环境下的并发访问问题，保证在多线程环境中对共享数据进行原子性的读写操作，避免了多线程并发访问时可能引发的数据不一致问题。
- 相比于传统的锁机制，CAS 操作不需要阻塞线程或切换上下文，因为它是一种乐观锁机制。这使得 CAS 在高并发场景下具有更好的性能和可伸缩性，尤其适用于细粒度的并发控制。

### 举例

CAS 机制当中使用了 3 个基本操作数：内存地址 V，旧的预期值 A (V 的一个 copy)，计算后要修改的新值 B。

- 1. 在内存地址 V 当中，存储着值为 10 的变量。
- 2. 此时线程 1 想要把变量的值增加 1。对线程 1 来说，旧的预期值 A=10，要修改的新值 B=11
- 3. 在线程 1 要提交更新之前，另一个线程 2 抢先一步，把内存地址 V 中的变量值率先更新成了 11。
- 4. 线程 1 开始提交更新，首先进行 A 和地址 V 的实际值比较 (Compare)，发现 A 不等于 V 的实际值【即  $10 \neq 11$ 】，提交失败。
- 5. 线程 1 重新获取内存地址 V 的当前值，并重新计算想要修改的新值。此时对线程 1 来说，A=11，B=12。这个重新尝试的过程被称为自旋【将主内存的值同步到线程 1 的工作内存，即工作内存也存了同主内容相同的变量 11】。
- 6. 这一次比较幸运，没有其他线程改变地址 V 的值。线程 1 进行 Compare，发

现 A 和地址 V 的实际值是相等的【通过自旋重新获取了地址 V 的当前值为 11，然后发现 V 地址的值和当前 A 是相等的，则可进行提交尝试修改数据】。

- 7. 线程 1 进行 SWAP，把地址 V 的值替换为 B，也就是 12

参考博客：<https://www.cnblogs.com/myopensource/p/8177074.html>

## CAS 必须借助 volatile 才能读取到共享变量的最新值来实现【比较并交换】的效果

在 CAS 时，变量需要用 volatile 修饰，他可以避免线程从自己的工作缓存中查找变量的值，必须到主存中获取它的值，线程操作 volatile 变量都是直接操作主存。即一个线程对 volatile 变量的修改，对另一个线程可见。但是 volatile 不保证原子性。

### 缺点

CAS 的缺点主要有 3 个

- 循环时间长，开销很大。可能线程 1 每次要执行 CAS 操作时，发现值都修改了，然后一直在循环里。
- 只能保证一个共享变量的原子操作。如果有多个共享变量的操作的话，最终还是要用 synchronized。
- ABA 问题。

比如该对象初始值是 5

- 1) 线程 1 准备将值要改为 6
- 2) 此时线程 2 抢占到了 CPU，将值改为了 8
- 3) 接着线程 3 抢占到了 CPU，又将值改为了 6

此时线程 1 终于抢占到了 CPU，然后使用 CAS 发现原始值和当前值都是 6，就可以执行操作。但是实际是在这个过程中，其他线程都进行了操作，可能存在逻辑上的其他异常，这就是 ABA 问题。

## CAS 为什么一定要借助 volatile 修饰的变量？

### ○ 1、volatile 变量：工作线程如何感知到主内存的变化？

首先，我们需要理解 JMM 的基本设定：

- **主内存 (Main Memory)**：所有线程共享的区域，存储了所有的实例字段、静态字段和数组元素。
- **工作内存 (Working Memory)**：每个线程私有的区域。线程对变量的所有操作（读取、赋值等）都必须在工作内存中进行，不能直接读写主内存。它大致对应于 CPU 的高速缓存 (Cache)。

### 正常变量的流程：

1. 线程 A 读取一个变量，会先把主内存的值拷贝一份到自己的工作内存中。
2. 线程 A 修改这个变量，是修改自己工作内存中的副本。
3. 在某个不确定的时间点，线程 A 才会把修改后的值写回 (Flush) 主内存。
4. 在此之前，线程 B 如果去读取该变量，它读到的可能还是主内存中未被修改的旧值。这就是可见性问题。

### volatile 修饰的变量如何解决这个问题？

volatile 关键字就像一个“主内存直通车”和“全局广播系统”，它通过\*\*内存屏障 (Memory Barrier)\*\* 这一底层机制来保证可见性：

1. 当一个线程写 volatile 变量时：

- 它会生成一个 Store 内存屏障。
- 这个屏障会强制该线程立即将工作内存中的最新值刷新到主内存中。
- 同时，这个写操作会向总线发出一个信号，导致其他 CPU 核心里缓存了该变量地址的缓存行失效。这就像一个全局广播：“嘿，各位！地址为 0x1234 的数据更新了，你们手里的都是旧的，不能用了！”

## 2. 当一个线程读 volatile 变量时：

- 它会生成一个 Load 内存屏障。
- 这个屏障会强制该线程忽略自己工作内存中的值（因为可能已经失效）。
- 线程会直接从主内存中重新读取最新的值到自己的工作内存。

**总结：**工作线程正是通过“本地缓存失效，必须从主内存重新加载”这一机制，来感知到 volatile 变量被其他线程修改了。

## ○ 2、如果变量没有 volatile 修饰会怎么样？

CAS（比较并交换）是一个乐观锁技术，它的操作包含三个核心值：

- **V**: 要更新的内存地址（变量）。
- **E**: 预期该变量的旧值（Expected Value）。
- **N**: 要更新成的新值（New Value）。

操作逻辑是：只有当内存地址 V 处的当前值等于预期旧值 E 时，才将该地址的值更新为新值 N。整个“比较并更新”的过程是一个原子操作，由 CPU 指令（如 x86 的 CMPXCHG）保证。

没有 volatile，线程 A 拿到的预期值 E 就是从它自己的工作内存（CPU 缓存）里来的。如果主内存的值已经被其他线程改变，而线程 A 对此一无所知，它的 E 就是一个 **stale data**（陈旧数据）。

基于陈旧的数据进行 CAS，会直接导致以下灾难性后果：

### 后果一：致命的效率问题：无限循环（活锁 Livelock）

这可以说是最直接、最致命的问题。我们知道，像

AtomicInteger.getAndIncrement() 这样的操作，其内部实现通常是一个 while(true) 循环。

```

public final int getAndIncrement() {
    for (;;) { // 等价于 while(true)
        int current = get(); // 步骤1: 获取当前值
        int next = current + 1;
        if (compareAndSet(current, next)) // 步骤2: CAS操作
            return current;
    }
}
```
这里的 `get()` 方法内部就是直接读取 `volatile` 修饰的 `value` 字段。
**现在，我们假设 `value` 没有被 `volatile` 修饰，看看会发生什么：**

1. **初始状态**: 主内存 `value = 10`。
2. **线程A执行**:
   * 执行 `get()`，从主内存读取 `value=10` 到自己的工作内存。此时，它认定的 `current` 是 10。
3. **线程B抢先执行**:
   * 线程B也读取了 `value=10`，并成功执行了CAS，将主内存的 `value` 更新为 11。
   * 线程B可能继续执行，将主内存的 `value` 更新为 12, 13, 14...
4. **轮到线程A执行CAS**:
   * 线程A开始执行 `compareAndSet(10, 11)`。
   * CPU执行CAS指令，比较主内存的值（现在是14）和它的预期值（10）。
   * 结果：不相等，CAS失败。循环继续。
5. **灾难发生：线程A进入下一次循环**:
   * 它再次调用 `get()`。**因为 `value` 没有 `volatile` 修饰，JMM不保证它会去主内存重新读取！**
   * 线程A极有可能直接从自己的工作内存（缓存）中读取 `value`，它读到的 `current` **仍然是 10**！
   * 它再次尝试 `compareAndSet(10, 11)`，再次失败。
   * ...
   * **线程A将永远在 `while(true)` 循环中挣扎，用它那个早已过时的 `current=10` 去和主内存中不断变化的值作比较，永不成功，永不退出。**

```

这就是\*\*活锁 (Livelock)\*\*，线程在不停地忙碌，消耗大量CPU资源，但程序逻辑却没有任何推进。

#### \*\*`volatile` 如何解决？\*\*

当 `value` 被 `volatile` 修饰后，每次循环中调用 `get()`，都会强制线程 A \*\*抛弃自己的工作内存副本，直接从主内存重新加载最新的值\*\*。

- \* 在上面的第5步，线程A会从主内存读到最新的 `value=14`。
- \* 它的 `current` 就变成了 14。
- \* 它会尝试 `compareAndSet(14, 15)`，这次就很大概率成功。即使失败，下一次循环它也会再次获取最新值，保证程序总能向前推进。

#### #### 2. 破坏了CAS的“乐观”前提，使其逻辑失效

CAS是一种\*\*乐观锁\*\*。它的哲学是：“我乐观地认为，在我准备数据的这段时间里，没有别人修改过它。现在我来确认一下，如果真的没变，我就更新它。”

- \* \*\*“我准备数据”\*\* -> 读取变量，得到预期值 `E`。
- \* \*\*“确认一下”\*\* -> 执行CAS，比较内存值和 `E`。

没有 `volatile`，这个哲学就崩溃了。线程获取的 `E` 值本身就是陈旧的，它乐观的那个“快照”是错误的，后续的一切比较和尝试都建立在了一个错误的事实 上。

`volatile` 强制每次 `get()` 都从主内存读取，确保了这个“快照”是\*\*当前最新\*\*的，从而保证了乐观锁的逻辑基础是正确的。

#### ### 总结：为什么 `volatile` 对 CAS 是不可或缺的

|                    |                                     |                             |
|--------------------|-------------------------------------|-----------------------------|
| 对比项                | **CAS (无 volatile)**                | **CAS (有 volatile)**        |
| :---   :---   :--- |                                     |                             |
| **数据来源**           | 从线程本地的**工作内存（缓存）**读取，可能是陈旧数据。       | 强制从**主内存**读取，保证是最新数据。       |
| **循环行为**           | 极易因无法获取新值而陷入**无限循环（活锁）**，白白消耗CPU。   | 每次循环都能获取最新值，保证逻辑可以向前推进。     |
| **逻辑基础**           | 破坏了乐观锁的“基于最新快照进行比较”的前提，逻辑上存在根本缺陷。   | 维护了乐观锁的正确逻辑前提。              |
| **指令重排**           | 无法禁止 CAS 操作前后指令的重排序，可能在复杂场景下导致程序错误。 | **通过内存屏障禁止指令重排**，保证了程序的有序性。 |

\*\*最终结论：\*\*

CAS 和 `volatile` 是不可分割的整体。`volatile` 为 CAS 提供了\*\*正确\*\* 和 \*\*高效\*\* 运行所必需的内存可见性和有序性保障。

- \* \*\*可见性\*\* 确保 CAS 不会在陈旧数据上做无用功，避免了\*\*活锁\*\*。
- \* \*\*有序性\*\* 确保 CAS 相关的复杂逻辑不会因指令重排而出错。

没有 `volatile`，CAS 就会变成一个在陈旧、不可靠的数据上空转的、甚至会引发严重逻辑错误的“残次品”。因此，\*\*CAS 必须借助 `volatile` 修饰的变量才能正确执行。\*\*

## 后果二：致命的逻辑错误——指令重排导致的“半成品”问题

这是比活锁更阴险的问题，因为它不会让程序卡住，而是会让程序在某些情况下悄无声息地出错，产生错误的结果，极难排查。

volatile关键字除了保证可见性，还有一个核心作用是禁止指令重排。CAS操作本身虽然是原子的，但它无法管理自己前后其他普通指令的顺序。

例如获取单例模式下，可能造成拿到一个未初始化的对象。让我们来看一个非常经典的场景：**延迟初始化（Lazy Initialization）**。假设我们要实现一个线程安全的单例，但又不想在程序启动时就创建它。

```
public class UnsafeSingleton {
    private static UnsafeSingleton instance = null; // 没有 volatile
    private int someData; // 单例中的一些数据

    private UnsafeSingleton() {
        // 构造函数可能很复杂，需要很多步骤
        this.someData = 42; // 步骤A: 初始化数据
        // ... 其他可能更复杂的初始化 ...
    }

    public static UnsafeSingleton getInstance() {
        if (instance == null) { // 第一次检查
            synchronized (UnsafeSingleton.class) {
                if (instance == null) { // 第二次检查
                    instance = new UnsafeSingleton(); // 问题出在这里!
                }
            }
        }
        return instance;
    }
}

``````new UnsafeSingleton()`` 这个操作在JVM中大致分为三步：
1. **分配内存**：为 `UnsafeSingleton` 对象分配一块内存空间。
2. **初始化对象**：调用 `UnsafeSingleton` 的构造函数，填充字段（如 `someData = 42`）。
3. **建立引用**：将 `instance` 变量指向刚刚分配的内存地址。
```

\*\*没有`volatile`，指令重排的风险就来了！\*\*

由于性能优化，JVM和CPU可能会将步骤2和步骤3重排。执行顺序可能变成 **1 -> 3 -> 2**。

\*\*现在，我们设想一下两个线程并发执行`getInstance()`的场景：\*\*

1. \*\*线程A\*\* 进入同步块，执行 `instance = new UnsafeSingleton();`。
2. 由于指令重排，它先执行了**步骤1（分配内存）**和**步骤3（建立引用）**。此时，`instance`变量已经**不再是`null`**了，它指向了一块刚分配的内存。
3. \*\*但是，步骤2（初始化对象）还没有执行！\*\* 这意味着这块内存里的`someData`字段还是默认值0，而不是42。这个`instance`是一个**半成品**。
4. 就在这一瞬间，**线程B**到达了第一个`if (instance == null)`判断。
5. 线程B发现`instance`已经不是`null`了（因为线程A执行了步骤3），于是它**跳过了整个`synchronized`代码块**，直接返回了当前的`instance`。
6. \*\*灾难发生\*\*：线程B拿到了一个它认为已经初始化好的单例对象，但实际上这个对象是个“半成品”。当线程B去使用这个对象的`someData`字段时，它得到的是错误的值（0而不是42），从而引发了严重的程序逻辑错误。

\*\*`volatile`如何解决这个问题？\*\*

如果`instance`变量被`volatile`修饰：

```
```java
private static volatile UnsafeSingleton instance = null;
```

防止了指令重排！

### ○ 3、最终总结

所以，“CAS为什么一定要借助 volatile？”

- 为了避免活锁：volatile的可见性保证了CAS在循环尝试时，总能拿到最新的值，让逻辑得以推进。

- 为了保证正确性：volatile的有序性（禁止指令重排）保证了CAS操作的上下文是正确的，避免了读到“半成品”数据或状态，从而防止了毁灭性的、难以调试的逻辑错误。

仅仅“CAS失败”是远远不够的，因为失败只能阻止当前一次的错误赋值。它无法阻止线程因为看不到新值而陷入死循环，更无法阻止指令重排导致的其他线程拿到一个已被错误“发布”的半成品对象。volatile正是填补了这些致命漏洞的唯一手段。

---

---

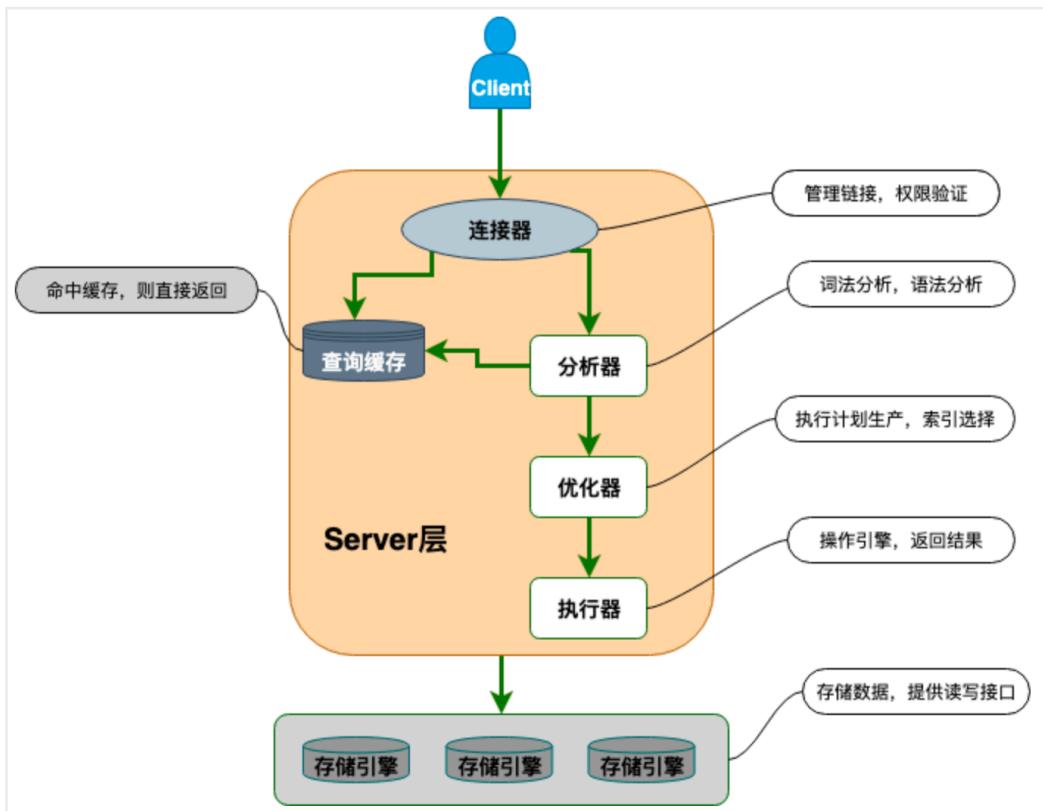
## MySQL

---

---

### MySQL 基础架构

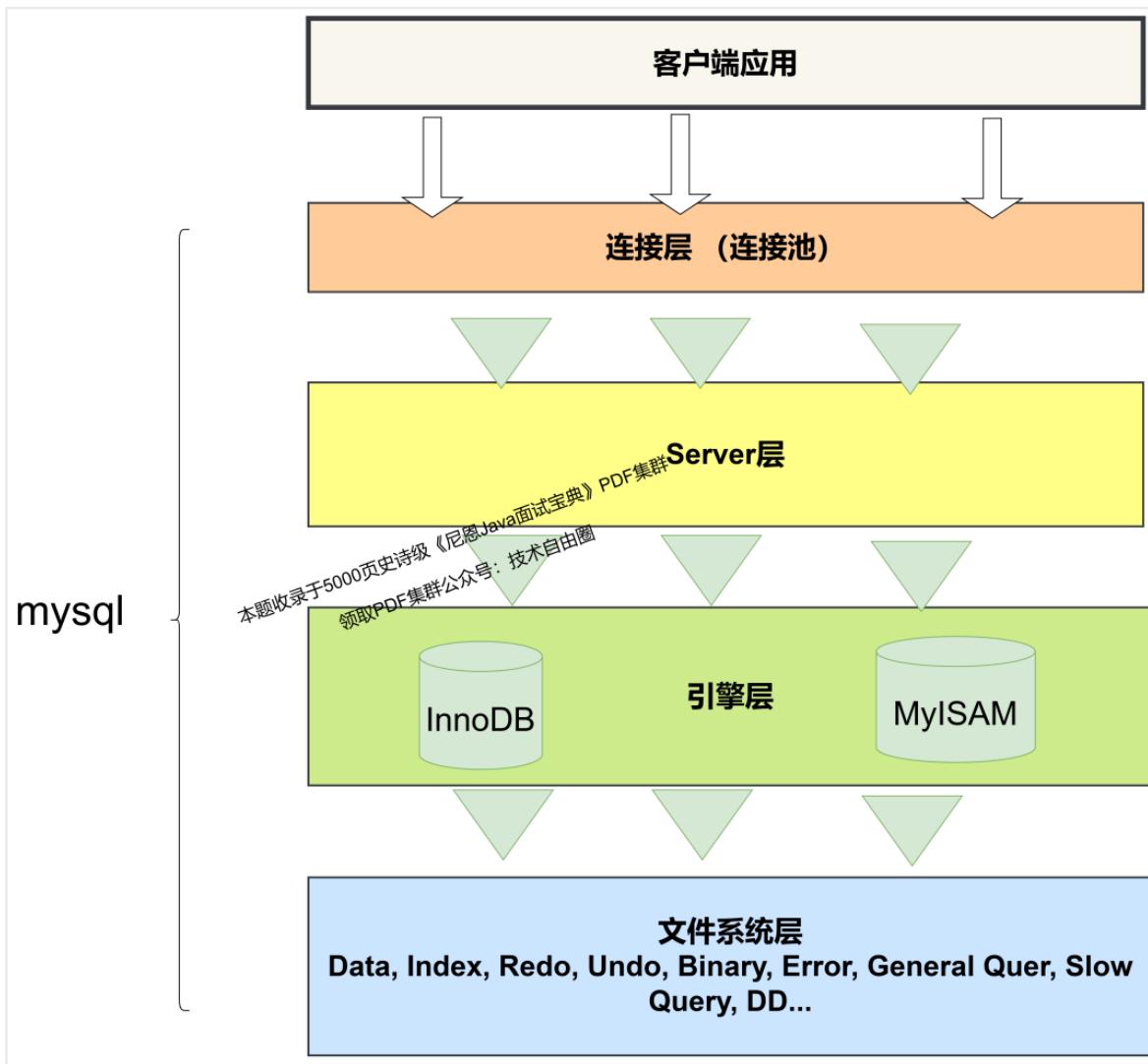
- **连接器**：身份认证和权限相关（登录 MySQL 的时候）。
- **查询缓存**：执行查询语句的时候，会先查询缓存（MySQL 8.0 版本后移除，因为这个功能不太实用）。
- **分析器**：没有命中缓存的话，SQL 语句就会经过分析器，分析器说白了就是要先看你的 SQL 语句要干嘛，再检查你的 SQL 语句语法是否正确【词法&语义分析】。
- **优化器**：按照 MySQL 认为最优的方案去执行。
- **执行器**：执行语句，然后从存储引擎返回数据。执行语句之前会先判断是否有权限，如果没有权限的话，就会报错。
- **插件式存储引擎**：主要负责数据的存储和读取，采用的是插件式架构，支持 InnoDB、MyISAM、Memory 等多种存储引擎。



## MySQL 如何选择最优执行计划?

### Mysql的架构

Mysql的架构，整体是分为服务层、引擎层和文件系统层，其架构图如下所示：



MySQL Server 服务层 (Service Layer) 解析 SQL 语句、优化查询以及执行操作的，分别有三个关键组件完成：

- **解析器 (Parser)**
- **优化器 (Optimizer)**
- **执行器 (Executor)**。

每个组件在查询执行的过程中扮演不同的角色，下面分别介绍这三者的作用：

## 1. 解析器 (Parser)

解析器是 SQL 查询执行的第一步，它的职责是将用户发送的 SQL 语句解析为数据库能够理解的内部结构。

- **SQL 词法分析**：解析器首先对 SQL 语句进行词法分析，将 SQL 语句分割成多个“单词”或“标记”，如表名、列名、关键字等。
- **语法分析**：接着，解析器会根据 SQL 语法规则生成对应的解析树 (Parse Tree)，用来描述 SQL 语句的逻辑结构。这个过程检查 SQL 语句的语法是否正确。
- **语义分析**：确认 SQL 语句中涉及的数据库对象是否存在（比如表名、字段名是否有效），并且检查权限。

解析完成后，生成一个中间表示结构，交由下一步进行处理。

## 2. 优化器 (Optimizer)

优化器负责选择最优的执行计划，使查询能够以最高效的方式运行。

- **逻辑优化**: 优化器会对 SQL 语句进行逻辑优化，比如 SQL 语句重写、消除冗余操作、合并重复条件、重新排列 WHERE 子句中的条件等。
- **物理优化**: 在物理优化阶段，优化器会选择最优的访问路径和执行顺序。例如，它会决定使用哪种索引（如果有多个索引可选），是否做全表扫描，如何连接多张表（选择嵌套循环、哈希连接或排序合并连接等）。
- **成本估算**: 优化器会基于数据库的统计信息（例如表的大小、索引的选择性等）来估算不同执行计划的成本，选择代价最低的执行方案。

经过优化后，优化器会生成一个查询执行计划，并交给执行器处理。

### 3. 执行器 (Executor)

执行器的任务是按照优化器生成的执行计划，逐步执行查询，访问数据并返回结果。

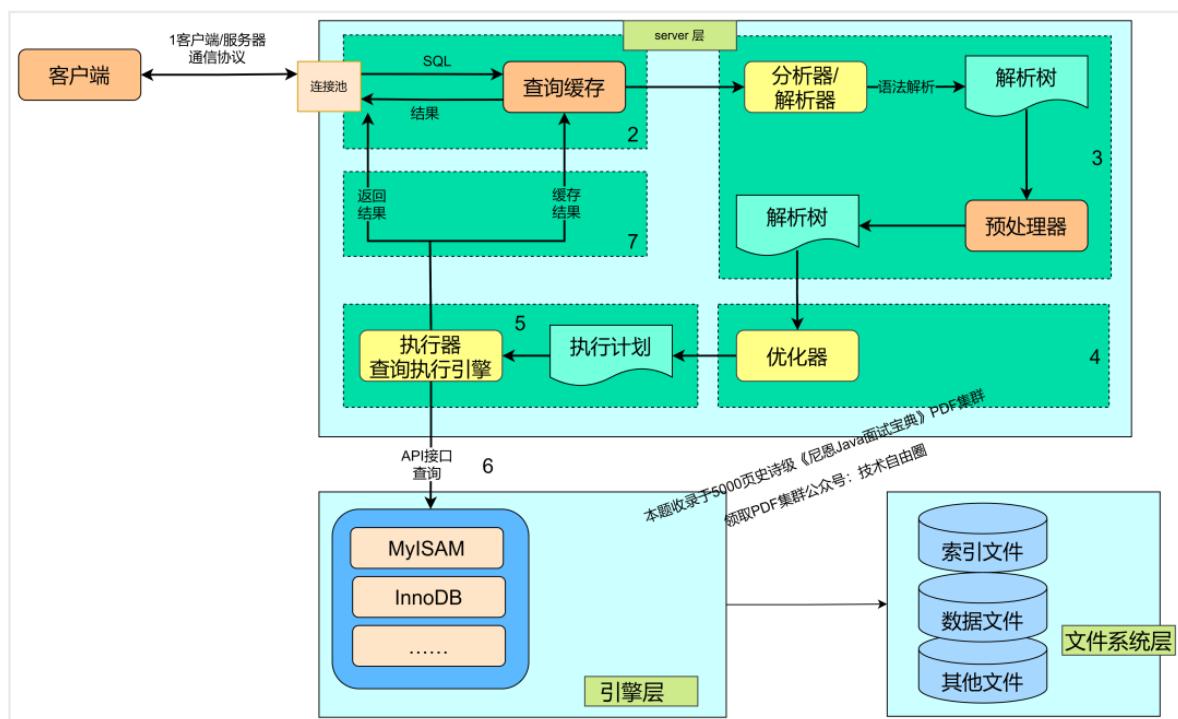
- **权限检查**: 在执行之前，执行器会首先检查用户是否有权限执行相应的操作。如果没有权限，则返回错误信息。
- **执行执行计划**: 执行器根据生成的执行计划，依次调用存储引擎的接口来执行具体的操作。例如，如果是查询操作，执行器会调用存储引擎来读取相应的数据；如果是插入操作，执行器则会调用存储引擎来插入数据。
- **结果返回**: 执行器根据查询的结果，将数据以合适的格式返回给客户端。如果涉及多个步骤（如 JOIN 操作），执行器会协调各个步骤的执行，并组合最终的结果集。

#### 三个核心组件之间的交互流程

1. 解析器: SQL 语句转换为解析树。
2. 优化器: 生成最优的执行计划。
3. 执行器: 根据计划调用存储引擎执行操作并返回结果。

这三个组件相互协作，完成从接收到 SQL 查询到返回结果的整个过程。

## 一条完整的 SQL 查询语句执行流程



- 根据sql的结构生成不同的执行计划，并选择一个最优的计划是 MySQL 优化器的主要任务。

- 执行器选择一个最优的计划，这是 MySQL 执行器的主要任务。  
执行器最终选择最优或者执行效率最高的执行计划，执行 SQL 并返回数据。

## 影响执行计划选择的因素

那么这个最优的执行计划是如何选择的呢？

优化器会考虑多种因素，如每个执行计划的成本、索引的可用性、表的大小、数据分布情况、连接条件等。

- **因素 1：每个执行计划的成本**

优化器会估算每个执行计划的成本。成本主要包括 I/O 成本（从磁盘读取数据的开销）和 CPU 成本（对数据进行处理的开销）。

例如，使用索引进行查询可能会减少 I/O 成本，但如果索引需要大量的 CPU 资源来维护或者遍历，那么优化器会综合考虑这两种成本来决定是否使用该索引。

- **因素 2：索引是否存在？**

当查询语句中的列有合适的索引时，优化器会考虑使用索引来加速查询。例如，对于一个 SELECT \* FROM users WHERE user\_id = 123 的查询，如果 user\_id 列有索引，优化器可能会选择使用索引来定位满足条件的行，而不是全表扫描。

- **因素 3：复合索引的是否能最左匹配？**

对于复合索引（包含多个列的索引），列的顺序也很重要。

例如，有一个复合索引 (col1, col2, col3)，如果查询条件是 col1 = 'value1' AND col2 = 'value2'，那么这个复合索引可以被有效地利用。

但如果查询条件是 col2 = 'value2' AND col3 = 'value3'，索引的使用效率可能会降低，优化器可能需要重新评估是否使用该索引。

- **因素 4：连接类型和顺序？**

不同的连接类型（如内连接、外连接、交叉连接等）会影响执行计划。例如，内连接是比较常用的连接方式，它只返回满足连接条件的行。优化器会根据连接条件和表的大小来确定连接的顺序。一般来说，会先选择较小的表作为驱动表，这样可以减少连接操作的成本。

在多表连接的情况下，连接顺序的不同组合会产生多种执行计划。例如，对于三个表 A、B、C 的连接，可能的连接顺序有 ((A JOIN B) JOIN C)、((A JOIN C) JOIN B) 和 ((B JOIN C) JOIN A) 等，优化器会通过计算成本来选择最优的连接顺序。

- **因素 5：连接条件的选择性？**

连接条件的选择性是指连接条件能够过滤掉多少行。如果连接条件的选择性很高，例如 A.col1 = B.col1，并且 col1 列的值在两个表中都比较唯一，那么优化器可能会更倾向于使用这种连接条件来减少连接操作后的结果集大小。

- **因素 6：数据分布情况？**

如果列的数据分布不均匀，例如一个列大部分值都相同，只有少数几个不同的值，那么索引在这种情况下的作用可能会受到限制。优化器会考虑这种数据分布情况来决定是否使用索引以及如何使用索引。

- **其他的考察因素：如表的数据量和增长趋势**

表的数据量大小直接影响查询的性能。对于大数据量的表，优化器会更倾向于寻找能够减少数据读取量的执行计划。

同时，表的增长趋势也很重要。如果一个表的数据量在不断增加，那么优化器可能需要重新评估现有的执行计划是否仍然是最优的。

## 通过 EXPLAIN 查看每个执行计划的成本

理解 EXPLAIN 结果中的关键列

- **id 列：**表示查询中每个 SELECT 子句的标识符。如果是简单的单表查询，id 通

常为 1。在复杂的子查询或连接查询中，id 可以帮助区分不同的子查询或连接部分。

- **select\_type** 列：描述了查询的类型，如 SIMPLE (简单查询)、PRIMARY (主查询)、SUBQUERY (子查询) 等。不同的查询类型可能会有不同的执行计划。
- **table** 列：显示查询涉及的表的名称。
- **type** 列：表示表的访问类型，这是评估执行计划效率的一个重要指标。常见的访问类型有 ALL (全表扫描)、index (索引扫描)、range (范围扫描)、ref (非唯一索引扫描)、eq\_ref (唯一索引扫描) 等。一般来说，eq\_ref 和 ref 的效率较高，而 ALL 效率较低。
- **possible\_keys** 列：显示查询可能使用的索引。这只是一个参考，优化器可能会根据实际情况选择不使用其中的某些索引。
- **key** 列：实际使用的索引。如果这个列显示为 NULL，表示没有使用索引，可能是因为优化器认为使用索引的成本更高或者没有合适的索引。
- **key\_len** 列：表示使用的索引的长度。这个长度可以帮助判断索引是否被充分利用。例如，对于一个复合索引，如果 key\_len 只覆盖了索引的一部分列，可能表示索引没有被完全利用。
- **ref** 列：显示索引的引用情况。如果是通过索引进行连接操作，这个列会显示连接所引用的列。
- **rows** 列：估算的需要扫描的行数。这个数字越小，通常表示执行计划越高效。
- **Extra** 列：包含了一些额外的信息，如是否使用了临时表、是否使用了文件排序等。这些信息可以帮助发现潜在的性能问题。

在 MySQL 中，EXPLAIN 命令用于获取查询的执行计划信息，FORMAT 参数可以指定输出的格式。

以下是一些常见的格式及说明：

#### 1. TRADITIONAL (默认格式):

以传统的表格形式展示执行计划信息，包括 id、select\_type、table、type、possible\_keys、key、key\_len、ref、rows、Extra 等列。每一列都代表了执行计划的不同方面，例如：

#### 2. JSON:

以 JSON 格式输出执行计划信息，更易于解析和处理，尤其是在需要进行自动化分析或与其他工具集成时。

JSON 格式的输出包含了更多详细的信息，如成本估算、访问路径等。

## SQL 执行结果分析示例

执行sql如下所示：

```
EXPLAIN FORMAT=JSON SELECT * from `test_user` where height = 120;
```

sql的执行结果如下所示：

```
{  
  "query_block": {  
    "select_id": 1,  
    "cost_info": {  
      "query_cost": "7252.80"  
    },  
    "table": {  
      "table_name": "test_user",  
      "access_type": "ref",  
      "possible_keys": [  
        "idx_height"  
      ],  
      "key": "idx_height",  
      "used_key_parts": [  
        "height"  
      ],  
      "key_length": "5",  
      "ref": [  
        "const"  
      ],  
      "rows_examined_per_scan": 6044,  
      "rows_produced_per_join": 6044,  
      "filtered": "100.00",  
      "cost_info": {  
        "read_cost": "6044.00",  
        "eval_cost": "1208.80",  
        "prefix_cost": "7252.80",  
        "data_read_per_join": "1M"  
      },  
      "used_columns": [  
        "id",  
        "id_card",  
        "age",  
        "user_name",  
        "height",  
        "address"  
      ]  
    }  
  }  
}
```

以下是对给定 JSON 格式的执行计划结果的详细解释：

1. "query\_block":
  - 代表查询块信息，描述了整个查询的执行计划相关内容。
2. "select\_id": 1:
  - 查询的唯一标识符为 1。
3. "cost\_info":
  - "query\_cost": "7252.80": 查询的预估总成本为 7252.80。这个成本是 MySQL 优化器根据多种因素估算出来的，包括读取数据的成本、评估条件的成本等。一般来说，成本越低，查询的执行效率可能越高。
4. "table":

- "read\_cost": "6044.00": 读取数据的成本为 6044.00。
  - "eval\_cost": "1208.80": 评估条件的成本为 1208.80。
  - "prefix\_cost": "7252.80": 前缀成本为 7252.80，通常是指查询的总成本减去某些特定操作的成本。
  - "table\_name": "test\_user": 查询涉及的表名为 test\_user。
  - "access\_type": "ref": 访问类型为 ref，表示通过非唯一索引进行查找。在这个例子中，说明查询使用了索引进行查找，并且不是唯一索引。
  - "possible\_keys": ["idx\_height"]：可能使用的索引列表为 idx\_height，这表明优化器考虑了这个索引，并最终选择了它。
  - "key": "idx\_height": 实际使用的索引是 idx\_height，说明优化器认为使用这个索引可以提高查询性能。
  - "used\_key\_parts": ["height"]：使用的索引部分是 height 列，表明索引 idx\_height 是基于 height 列创建的，并且在查询中只使用了这个索引的 height 部分。
  - "key\_length": "5": 索引的长度为 5，可能与存储 height 列的值所需的空间有关。
  - "ref": ["const"]：索引引用的值是常量，这里可能是查询条件中的 height = 120 中的 120。
  - "rows\_examined\_per\_scan": 6044: 每次扫描检查的行数为 6044。这表示在执行查询过程中，需要检查的行数估计为 6044 行。
  - "rows\_produced\_per\_join": 6044: 每次连接产生的行数为 6044。如果查询涉及多个表的连接，这个值表示从这个表中产生的行数，用于连接操作。
  - "filtered": "100.00": 过滤后的行数比例为 100%。这意味着经过查询条件 height = 120 过滤后，所有满足条件的行数占总检查行数的比例为 100%。
  - "cost\_info":
  - "data\_read\_per\_join": "1M": 每次连接读取的数据量为 1M (可能是 1 兆字节)。
  - "used\_columns": ["id", "id\_card", "age", "user\_name", "height", "address"]：查询使用的列包括 id、id\_card、age、user\_name、height 和 address。这意味着查询将从表中读取这些列的数据。
- 
- 

## MyISAM 和 InnoDB 有什么区别？

- InnoDB 支持行级别的锁粒度，MyISAM 不支持，只支持表级别的锁粒度。
- MyISAM 不提供事务支持。InnoDB 提供事务支持，实现了 SQL 标准定义了四个隔离级别。
- MyISAM 不支持外键，而 InnoDB 支持。
- MyISAM 不支持 MVCC，而 InnoDB 支持。
- 虽然 MyISAM 引擎和 InnoDB 引擎都是使用 B+Tree 作为索引结构，但是两者

的实现方式不太一样。

- MyISAM 不支持数据库异常崩溃后的安全恢复，而 InnoDB 支持。
- InnoDB 的性能比 MyISAM 更强大。

所以，一般情况下我们选择 InnoDB 都是没有问题的，但是某些情况下你并不在乎可扩展能力和并发能力，也不需要事务支持，也不在乎崩溃后的安全恢复问题的话，选择 MyISAM 也是一个不错的选择。但是一般情况下，我们都是需要考虑到这些问题的。

因此，对于咱们日常开发的业务系统来说，你几乎找不到什么理由再使用 MyISAM 作为自己的 MySQL 数据库的存储引擎。

---

---

## MySQL 事务

### 事务 ACID 特性

- **原子性 (Atomicity)**: 事务是最小的执行单位，不允许分割。事务的原子性确保动作要么全部完成，要么完全不起作用；
- **一致性 (Consistency)**: 执行事务前后，数据保持一致，例如转账业务中，无论事务是否成功，转账者和收款人的总额应该是不变的；
- **隔离性 (Isolation)**: 并发访问数据库时，一个用户的事务不被其他事务所干扰，各并发事务之间数据库是独立的；
- **持久性 (Durability)**: 一个事务被提交之后。它对数据库中数据的改变是持久的，即使数据库发生故障也不应该对其有任何影响。

推荐一本书《数据密集型应用系统设计》：<https://github.com/Vonng/ddia?tab=readme-ov-file>

---

---

## MySQL 的事务隔离级别

### 读未提交

效率最高安全性最差，无锁，容易拿到脏数据，可能在 B 获取到数据之后被 A 回滚了，但是 B 还是拿着原来的数据去进行别的操作，这个就是脏数据。

### 读提交

**【解释】** 读提交就是一个事务只能读到其他事务已经提交过的数据，也就是其他事务调用 commit 命令之后的数据，可解决脏读

**【例子】** 在 A 没有提交事务的时候，B 读到的还是老数据而不是更新后的数据，只有 A 提交了事务才能读到最新数据。

**【问题】** 但是也同样造成了一个问题就是事务 A 的提交影响了事务 B 的查询结果，这就造成了不可重复读，即 A 提交前 B 读到的是老数据，A 提交后 B 读到的是新数据，所以在 B 的同一事务不同时刻读到的数据值可能不一致。

## 可重复读

【解释】事务不会读到其他事务对已有事务的修改，即使其他事务已提交，事务开始读到的已有数据是什么，在这个事务提交之前的任意时刻，这些数据的值都是一样的。

【例子】无论 A 处于事务提交前还是提交后，在 B 的事务开始到结束之间，读到 A 的数据都是一致的。

【问题】但是，对于其他事务新插入的数据是可以读到的，这也就引发了幻读的问题。

## 幻读（非隔离级别，而是隔离级别产生的问题）

【解释】多个事务同时访问相同的数据集时。当一个事务在查询数据时，另一个事务插入了满足查询条件的新数据，导致事务再次查询时发现数据条数发生了变化，但这种变化实际上并不存在，因此产生了幻读现象

【例子】事务 A 在开始的时候查到了一条数据，事务 B 在事务 A 提交事务之前又 insert 了一条数据，事务 A 在提交事务之前又去查了一次，导致查出了两条数据。

## 串行化

【解释】解决了脏读、可重复读、幻读的问题，但是效果最差，它将事务的执行变为顺序执行，与其他三个隔离级别相比，它就相当于单线程，后一个事务的执行必须等待前一个事务结束。

---

---

## MVCC

### 多版本并发控制 (Multi-Version Concurrency Control)

MVCC 是一种并发控制机制，用于在多个并发事务同时读写数据库时保持数据的一致性和隔离性。它是通过在每个数据行上维护多个版本的数据来实现的。当一个事务要对数据库中的数据进行修改时，MVCC 会为该事务创建一个数据快照，而不是直接修改实际的数据行。

MVCC 通过创建数据的多个版本和使用快照读取来实现并发控制。读操作使用旧版本数据的快照，写操作创建新版本，并确保原始版本仍然可用。这样，不同的事务可以在一定程度上并发执行，而不会相互干扰，从而提高了数据库的并发性能和数据一致性。

## InnoDB 存储引擎对 MVCC 的实现

### 一致性非锁定读

对于一致性非锁定读 (Consistent Nonlocking Reads) open in new window 的实现，通常做法是加一个版本号或者时间戳字段，在更新数据的同时版本号 + 1 或者更新时间戳。查询时，将当前可见的版本号与对应记录的版本号进行比对，如果记录的版本小于可见版本，则表示该记录可见。

在 InnoDB 存储引擎中，多版本控制 (multi versioning)open in new window 就是对非锁定读的实现。如果读取的行正在执行 DELETE 或 UPDATE 操作，这时读取操作不会去等待行上锁的释放。相反地，InnoDB 存储引擎会去读取行的一个快照数

据，对于这种读取历史数据的方式，我们叫它快照读 (snapshot read)。

## InnoDB对MVCC的实现之隐藏字段、ReadView、undolog

MVCC的实现依赖于：**隐藏字段**、**Read View**、**undo log**。在内部实现中，InnoDB通过数据行的DB\_TRX\_ID和Read View来判断数据的可见性，如不可见，则通过数据行的DB\_ROLL\_PTR找到undo log中的历史版本。每个事务读到的数据版本可能是不一样的，在同一个事务中，用户只能看到该事务创建Read View之前已经提交的修改和该事务本身做的修改

### 隐藏字段

在内部，InnoDB存储引擎为每行数据添加了三个隐藏字段：

- DB\_TRX\_ID (6字节)：表示最后一次插入或更新该行的事务id。此外，delete操作在内部被视为更新，只不过会在记录头Record header中的deleted\_flag字段将其标记为已删除
- DB\_ROLL\_PTR (7字节) 回滚指针，指向该行的 undo log 。如果该行未被更新，则为空
- DB\_ROW\_ID (6字节)：如果没有设置主键且该表没有唯一非空索引时，InnoDB会使用该id来生成聚簇索引

### ReadView

```
1 class ReadView {  
2     /* ... */  
3     private:  
4         trx_id_t m_low_limit_id;      /* 大于等于这个 ID 的事务均不可见 */  
5  
6         trx_id_t m_up_limit_id;      /* 小于这个 ID 的事务均可见 */  
7  
8         trx_id_t m_creator_trx_id;    /* 创建该 Read View 的事务ID */  
9  
10        trx_id_t m_low_limit_no;    /* 事务 Number, 小于该 Number 的 Undo Logs 均可以被 Purge */  
11  
12        ids_t m_ids;                /* 创建 Read View 时的活跃事务列表 */  
13  
14        m_closed;                  /* 标记 Read View 是否 close */  
15 }
```

Read View 主要是用来做可见性判断，里面保存了“当前对本事务不可见的其他活跃事务”

主要有以下字段：

- m\_low\_limit\_id：目前出现过的最大的事务ID+1，即下一个将被分配的事务ID。大于等于这个ID的数据版本均不可见
- m\_up\_limit\_id：活跃事务列表m\_ids中最小的事务ID，如果m\_ids为空，则m\_up\_limit\_id为m\_low\_limit\_id。小于这个ID的数据版本均可见
- m\_ids：Read View创建时其他未提交的活跃事务ID列表。创建Read View时，将当前未提交事务ID记录下来，后续即使它们修改了记录行的值，对于当前事务也是不可见的。m\_ids不包括当前事务自己和已提交的事务（正在内存中）
- m\_creator\_trx\_id：创建该Read View的事务ID

【m\_up\_limit\_id 可以理解为当 m\_ids 不为空时，等于 m\_ids 列表的最左边。如果 m\_ids 列表为空时，m\_up\_limit\_id=m\_low\_limit\_id】

### undo-log

undo-log 主要有两个作用：当事务回滚时用于将数据恢复到修改前的样子，另一个作用是 MVCC，当读取记录时，若该记录被其他事务占用或当前版本对该事务不可见，则可以通过 undo log 读取之前的版本数据，以此实现非锁定读在 InnoDB 存储引擎中 undo log 分为两种：insert undo log 和 update undo log：

1. **insert undo log**: 指在 insert 操作中产生的 undo log。因为 insert 操作的记录只对事务本身可见，对其他事务不可见，故该 undo log 可以在事务提交后直接删除。不需要进行 purge 操作

insert 时的数据初始状态：

事务1: insert into user(id, name) values (1, '菜花');

DB_TRX_ID	DB_ROLL_PTR	ID	NAME
1		1	菜花

2. **update undo log**: update 或 delete 操作中产生的 undo log。该 undo log 可能需要提供 MVCC 机制，因此不能在事务提交时就进行删除。提交时放入 undo log 链表，等待 purge 线程进行最后的删除

数据第一次被修改时：

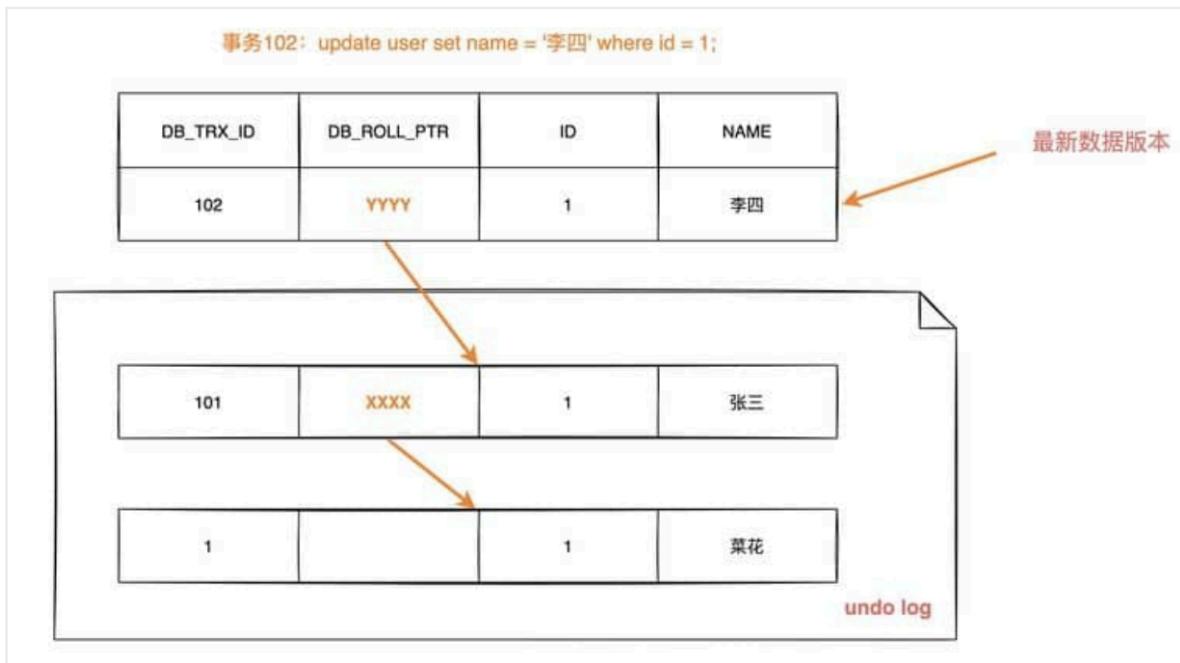
事务101: update user set name = '张三' where id = 1;

DB_TRX_ID	DB_ROLL_PTR	ID	NAME
101	XXXX	1	张三

最新数据版本

undo log

数据第二次被修改时：



不同事务或者相同事务的对同一记录行的修改，会使该记录行的 undo log 成为一条链表，链首就是最新的记录，链尾就是最早的旧记录。

### 可见性算法之如何获取最近一次已提交事务的数据

在 InnoDB 存储引擎中，创建一个新事务后，执行每个 select 语句前，都会创建一个快照 (Read View)，快照中保存了当前数据库系统中正处于活跃 (没有 commit) 的事务的 ID 号。其实简单的说保存的是系统中当前不应该被本事务看到的其他事务 ID 列表 (即 m\_ids)。当用户在这个事务中要读取某个记录行的时候，InnoDB 会将该记录行的 DB\_TRX\_ID 与 Read View 中的一些变量及当前事务 ID 进行比较，判断是否满足可见性条件

```
changes_visible() 的返回结果 true 代表可见, false 代表不可见.
```

```
/* storage/innobase/include/read0types.h */

bool changes_visible(trx_id_t id, const table_name_t &name) const
    MY_ATTRIBUTE((warn_unused_result)) {
    ut_ad(id > 0);

    /* 假如 trx_id 小于 Read View 限制的最小活跃事务ID m_up_limit_id 或者等于正在创建的事务ID
     * m_creator_trx_id 即满足事务的可见性. */
    if (id < m_up_limit_id || id == m_creator_trx_id) {
        return (true);
    }

    /* 检查 trx_id 是否有效. */
    check_trx_id_sanity(id, name);

    if (id >= m_low_limit_id) {
        /* 假如 trx_id 大于最大活跃的事务ID m_low_limit_id, 即不可见. */
        return (false);
    } else if (m_ids.empty()) {
        /* 假如目前不存在活跃的事务, 即可见. */
        return (true);
    }

    const ids_t::value_type *p = m_ids.data();

    /* 利用二分查找搜索活跃事务列表, 当 trx_id 在 m_up_limit_id 和 m_low_limit_id 之间
     * 如果 id 在 m_ids 数组中, 表明 ReadView 创建时候, 事务处于活跃状态, 因此记录不可见. */
    return (!std::binary_search(p, p + m_ids.size(), id));
}
```

- 1、如果记录 DB\_TRX\_ID < m\_up\_limit\_id, 那么表明最新修改该行的事务 (DB\_TRX\_ID) 在当前事务创建快照之前就提交了, 所以该记录行的值对当前事务是可见的
- 2、如果 DB\_TRX\_ID >= m\_low\_limit\_id, 那么表明最新修改该行的事务 (DB\_TRX\_ID) 在当前事务创建快照之后才修改该行, 所以该记录行的值对当前事务不可见。跳到步骤 5
- 3、m\_ids 为空, 则表明在当前事务创建快照之前, 修改该行的事务就已经提交了, 所以该记录行的值对当前事务是可见的
- 4、如果 m\_up\_limit\_id <= DB\_TRX\_ID < m\_low\_limit\_id, 表明最新修改该行的事务 (DB\_TRX\_ID) 在当前事务创建快照的时候可能处于“活动状态”或者“已提交状态”; 所以就要对活跃事务列表 m\_ids 进行查找 (源码中是用的二分查找, 因为是有序的)
  - 如果在活跃事务列表 m\_ids 中能找到 DB\_TRX\_ID, 表明: ① 在当前事务创建快照前, 该记录行的值被事务 ID 为 DB\_TRX\_ID 的事务修改了, 但没有提交; 或者 ② 在当前事务创建快照后, 该记录行的值被事务 ID 为 DB\_TRX\_ID 的事务修改了。这些情况下, 这个记录行的值对当前事务都是不可见的。跳到步骤 5
  - 在活跃事务列表中找不到, 则表明“id 为 trx\_id 的事务”在修改“该记录行的值”后, 在“当前事务”创建快照前就已经提交了, 所以记录行对当前事务可见
- 5、在该记录行的 DB\_ROLL\_PTR 指针所指向的 undo log 取出快照记录, 用快照记录的 DB\_TRX\_ID 跳到步骤 1 重新开始判断, 直到找到满足的快照版本或返回空

## RC 和 RR 隔离级别下 MVCC 的差异

在事务隔离级别 RC 和 RR (InnoDB 存储引擎的默认事务隔离级别) 下, InnoDB 存储引擎使用 MVCC (非锁定一致性读), 但它们生成 Read View 的时机却不同:

在 RC 隔离级别下的每次 select 查询前都生成一个 Read View (m\_ids 列表)

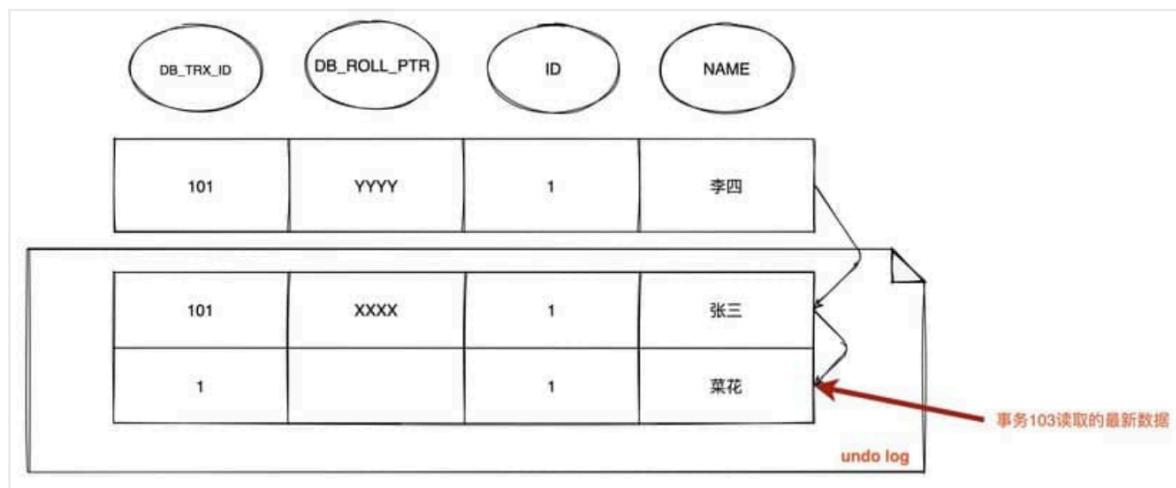
在 RR 隔离级别下只在事务开始后第一次 select 数据前生成一个 Read View (m\_ids

列表)

	事务101	事务102	事务103
T1	begin;		
T2		begin;	begin;
T3	update user set name = '张三' where id = 1;		
T4	update user set name = '李四' where id = 1;	...	select * from user where id = 1;
T5	commit;	update user set name = '王五' where id = 1;	
T6			select * from user where id = 1;
T7		update user set name = '赵六' where id = 1;	
T8		commit;	
T9			select * from user where id = 1;
T10			commit;

在 RC 下 ReadView 生成情况

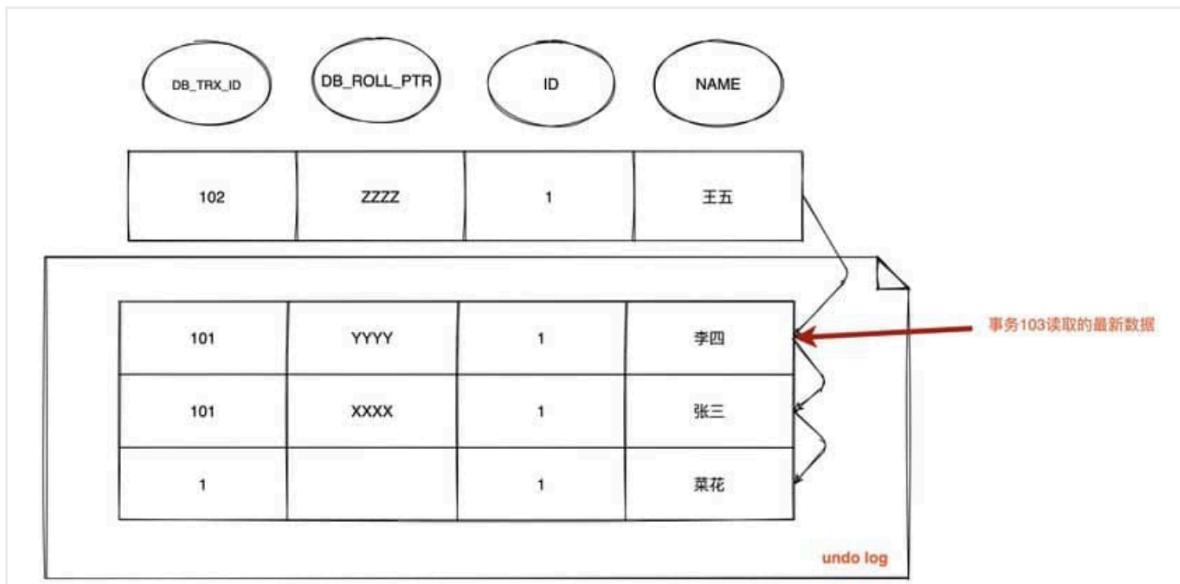
1. 假设时间线来到 T4 , 那么此时数据行 id = 1 的版本链为:



由于 RC 级别下每次查询都会生成 Read View , 并且事务 101、102 并未提交, 此时 103 事务生成的 Read View 中活跃的事务 m\_ids 为: [101,102] , m\_low\_limit\_id 为: 104, m\_up\_limit\_id 为: 101, m\_creator\_trx\_id 为: 103

- 此时最新记录的 DB\_TRX\_ID 为 101, m\_up\_limit\_id <= 101 < m\_low\_limit\_id, 所以要在 m\_ids 列表中查找, 发现 DB\_TRX\_ID 存在列表中, 那么这个记录不可见
- 根据 DB\_ROLL\_PTR 找到 undo log 中的上一版本记录, 上一条记录的 DB\_TRX\_ID 还是 101, 不可见
- 继续找上一条 DB\_TRX\_ID 为 1, 满足 1 < m\_up\_limit\_id, 可见, 所以事务 103 查询到数据为 name = 菜花

2. 时间线来到 T6 , 数据的版本链为:

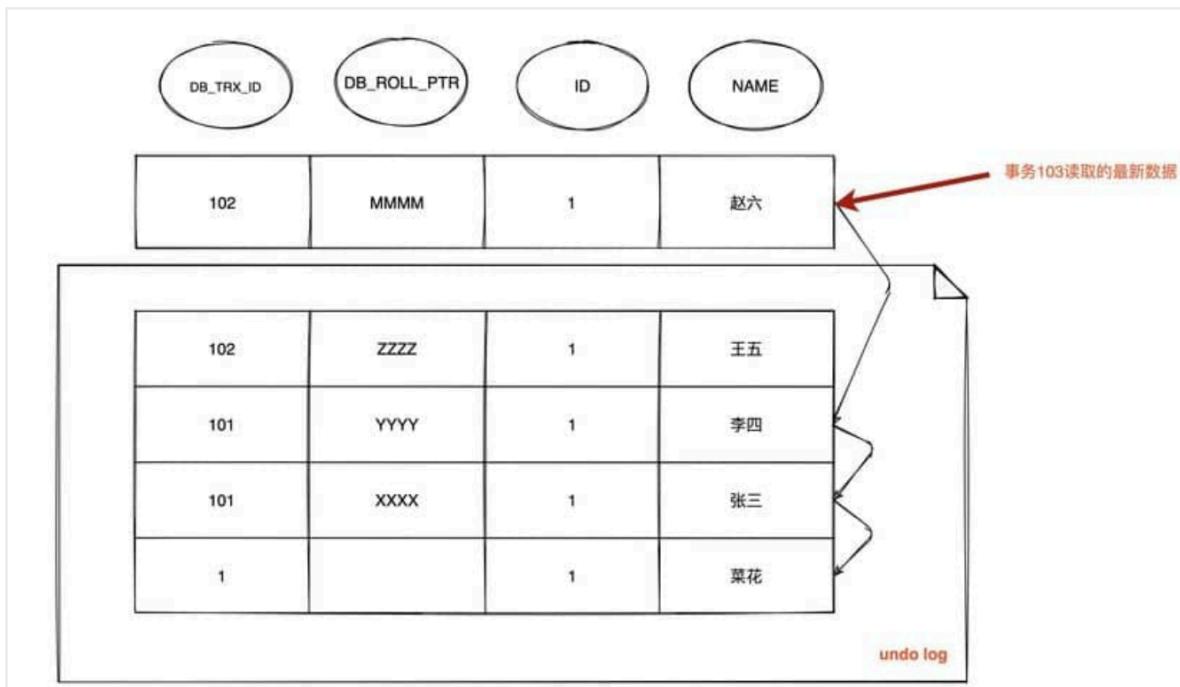


因为在 RC 级别下，重新生成 Read View，这时事务 101 已经提交，102 并未提交，所以此时 Read View 中活跃的事务 **m\_ids**: [102]，**m\_low\_limit\_id** 为：

**104**，**m\_up\_limit\_id** 为：102，**m\_creator\_trx\_id** 为：103

- 此时最新记录的 DB\_TRX\_ID 为 102，**m\_up\_limit\_id**  $\leq$  102  $<$  **m\_low\_limit\_id**，所以在 **m\_ids** 列表中查找，发现 DB\_TRX\_ID 存在列表中，那么这个记录不可见
- 根据 DB\_ROLL\_PTR 找到 undo log 中的上一版本记录，上一条记录的 DB\_TRX\_ID 为 101，满足  $101 < m_{up\_limit\_id}$ ，记录可见，所以在 T6 时间点查询到数据为 name = 李四，与时间 T4 查询到的结果不一致，不可重复读！

### 3. 时间线来到 T9，数据的版本链为：



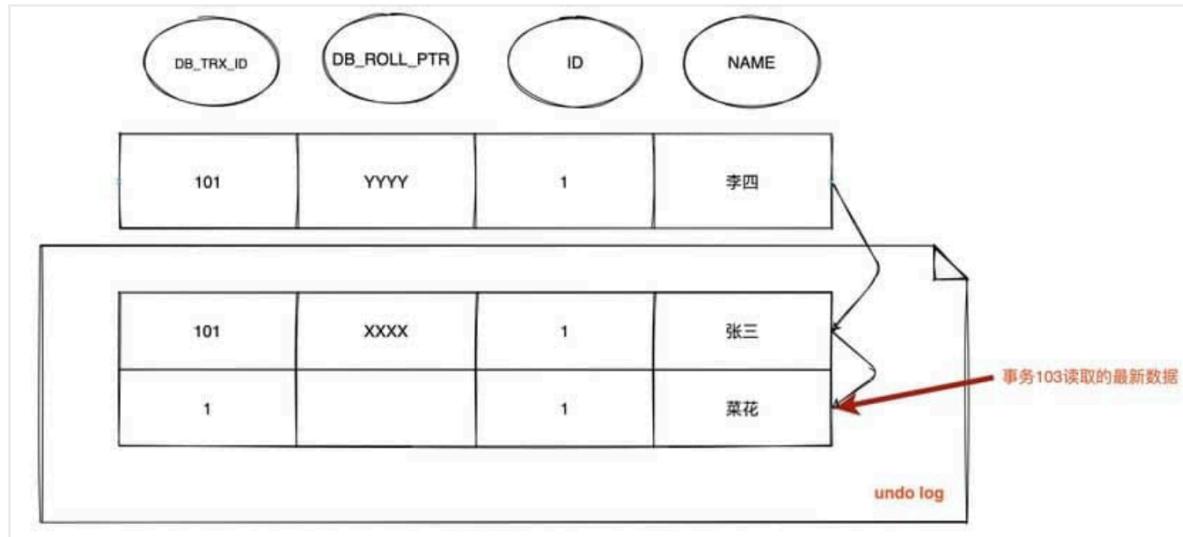
重新生成 Read View，这时事务 101 和 102 都已经提交，所以 **m\_ids** 为空，则 **m\_up\_limit\_id** = **m\_low\_limit\_id** = 104，最新版本事务 ID 为 102，满足  $102 < m_{low\_limit\_id}$ ，可见，查询结果为 name = 赵六

总结：在 RC 隔离级别下，事务在每次查询开始时都会生成并设置新的 Read View，所以导致不可重复读

## 在 RR 下 ReadView 生成情况

在可重复读级别下，只会在事务开始后第一次读取数据时生成一个 Read View (m\_ids 列表)

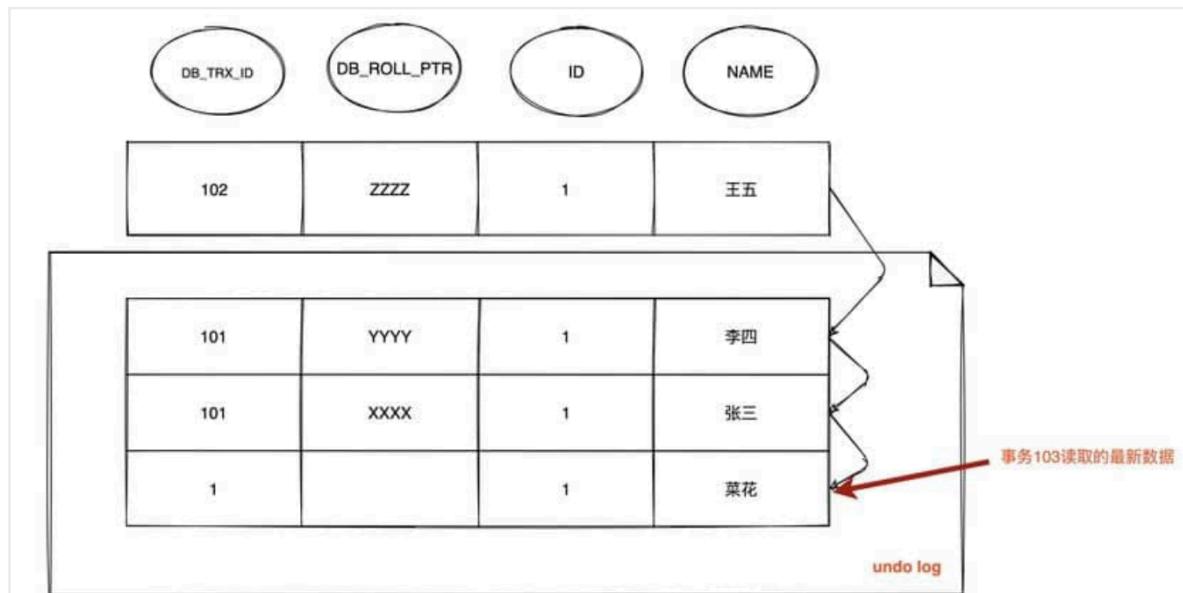
### 1. 在 T4 情况下的版本链为：



在当前执行 `select` 语句时生成一个 Read View, 此时 `m_ids: [101,102]`, `m_low_limit_id` 为: 104, `m_up_limit_id` 为: 101, `m_creator_trx_id` 为: 103  
此时和 RC 级别下一样:

- 最新记录的 DB\_TRX\_ID 为 101,  $m_{up\_limit\_id} <= 101 < m_{low\_limit\_id}$ , 所以要在 m\_ids 列表中查找, 发现 DB\_TRX\_ID 存在列表中, 那么这个记录不可见
- 根据 DB\_ROLL\_PTR 找到 undo log 中的上一版本记录, 上一条记录的 DB\_TRX\_ID 还是 101, 不可见
- 继续找上一条 DB\_TRX\_ID 为 1, 满足  $1 < m_{up\_limit\_id}$ , 可见, 所以事务 103 查询到数据为 name = 菜花

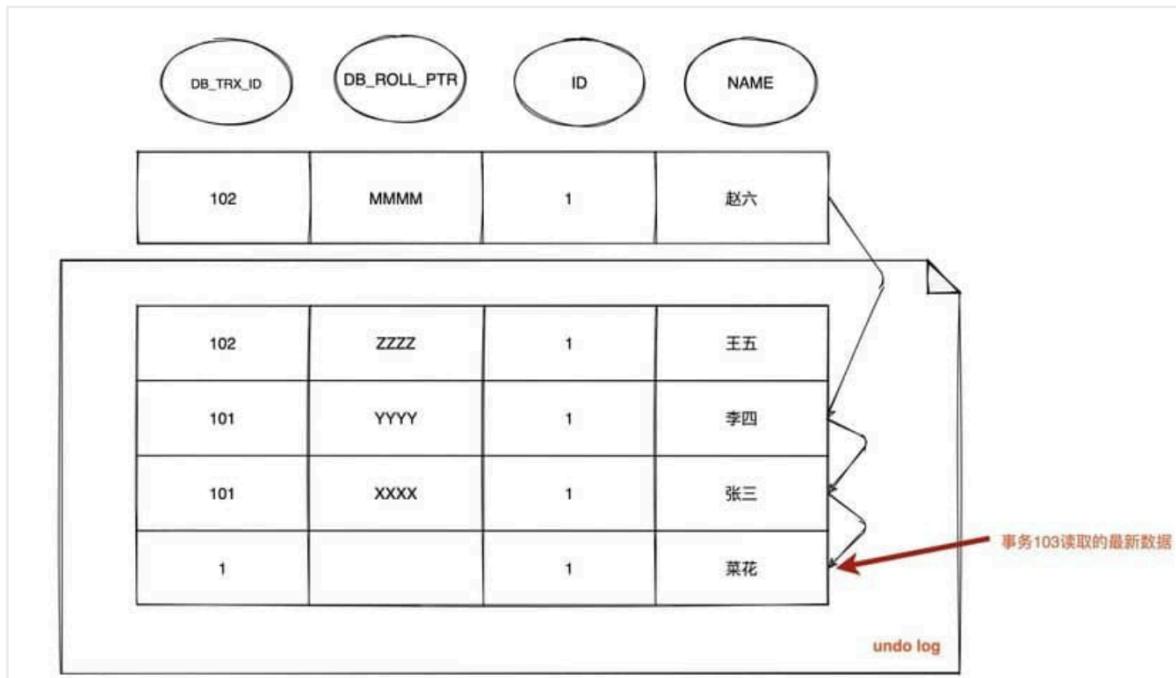
### 2. 时间点 T6 情况下：



在 RR 级别下只会生成一次 Read View, 所以此时依然沿用 `m_ids: [101,102]`, `m_low_limit_id` 为: 104, `m_up_limit_id` 为: 101, `m_creator_trx_id` 为: 103

- 最新记录的 DB\_TRX\_ID 为 102,  $m_{up\_limit\_id} \leq 102 < m_{low\_limit\_id}$ , 所以要在  $m\_ids$  列表中查找, 发现 DB\_TRX\_ID 存在列表中, 那么这个记录不可见
- 根据 DB\_ROLL\_PTR 找到 undo log 中的上一版本记录, 上一条记录的 DB\_TRX\_ID 为 101, 不可见
- 继续根据 DB\_ROLL\_PTR 找到 undo log 中的上一版本记录, 上一条记录的 DB\_TRX\_ID 还是 101, 不可见
- 继续找上一条 DB\_TRX\_ID 为 1, 满足  $1 < m_{up\_limit\_id}$ , 可见, 所以事务 103 查询到数据为 name = 菜花

### 3. 时间点 T9 情况下:



此时情况跟 T6 完全一样, 由于已经生成了 Read View, 此时依然沿用  $m\_ids: [101, 102]$ , 所以查询结果依然是 name = 菜花

### 总结: 什么是 MVCC?

MVCC, 多版本并发控制, 主要解决 rc 和 rr 下一致性非锁定读的问题, 通过获取 DB\_TRX\_ID (最后一次插入或更新该行的事务 id) 和某个时刻生成的 Read View (快照, 包含 up\_limit\_id、low\_limit\_id、m\_ids 列表) 进行可见性判断, 从而让读写操作可以并发执行。

但是 rc 和 rr 在的区别在于 Read View 快照的变化:

- rc 隔离级别下, 事务开始的时候, 每次的 select 都会生成最新的 Read View 快照 ( $up\_limit\_id$ 、 $low\_limit\_id$ 、 $m\_ids$  列表每次都可能会变), 因此它能读到其他已提交事务所做的最新修改, 从而实现了读已提交。
- rr 隔离级别下, 只有事务中的第一次 SELECT 会生成 Read View, 后续的所有查询都将复用这个快照, 从而保证了在一个事务内多次读取同一份数据的结果是一致的, 实现了可重复读。

PS: 关于幻读问题, 其实在 RR 隔离级别下已经解决了, 通过临键锁 (Next-Key Lock, 即记录锁 + 间隙锁的组合) 阻止其他事务在这个范围内插入新数据 (RC 只有记录锁), 并且通过 MVCC 机制复用同一个 Read View 快照的特性确保了事务期间不会读到新插入的数据。因此幻读问题是由 MVCC 和临键锁两者共同协作解决的。

---

## MySQL 中的行级锁

### 行级锁的类型主要有三类

- Record Lock, 记录锁, 也就是仅仅把一条记录锁上【记录锁使得该条记录无法 delete 或 update 操作】;
- Gap Lock, 间隙锁, 锁定一个范围, 但是不包含记录本身, 只存在于可重复读隔离级别, 目的是为了解决可重复读隔离级别下幻读的现象
- Next-Key Lock: Record Lock + Gap Lock 的组合, 锁定一个范围, 并且锁定记录本身。

### 间隙锁和 Next-Key Lock 区别

- 间隙锁: 锁定一个范围, 例如 (3,5), 右边区间为开, 表示没有锁定记录本身, 因此 id=5 的记录可以进行 update 或 delete 【间隙锁的目的是防止插入幻影记录而提出的】。
- Next-Key Lock: 锁定一个范围且锁定记录本身, 例如 (15, 20]【意味着其他事务即无法 update 或者 delete id = 20 的记录, 同时无法插入 id 值为 16、17、18、19 的这一些新记录】, 右边区间为闭, 从其由 Record Lock 组成也可以看出, 会有一个记录锁, 所以闭合区间是有 Record Lock 来执行的。作用于防止范围内的新增以及闭合位置的删除或更新操作。

### MySQL 加行级锁小简介

- 1、RC 模式只有记录锁 + 全表锁 (并非锁类型, 而是全表扫描, 并锁定它扫描过的每一行, 从效果上看就等同于锁定了全表), RR 模式有记录锁 + 间隙锁 + next-key 锁 + 全表锁 (没有间隙锁就没有 next-key 锁), 间隙锁和 next-key 锁主要用于在 RR 模式下解决幻读的问题, RC 本身语义是为了获取数据的一致性而非在读取过程防止其他事务对数据的修改造成幻读, 所以 RC 无需用到间隙锁和 next-key lock。
- 2、当 update 的 where 条件需要为索引才可触发行级锁, 如果条件不是索引则为全表锁
- 3、在 InnoDB 默认的隔离级别 REPEATABLE-READ 下, 行锁默认使用的是 **Next-Key Lock**。但是, 如果操作的索引是唯一索引或主键, InnoDB 会对 Next-Key Lock 进行优化, 将其降级为 Record Lock, 即仅锁住索引本身, 而不是范围。如果不是, 则会保持 Next-Key Lock 或者退化为间隙锁

### 为什么 mysql 在 RC 下不支持间隙锁和 next-key lock?

因为 RC 的语义只要求读到已提交的数据, 它允许“不可重复读”和“幻读”的发生, 所以没必要去锁住间隙来阻止插入。

### 加行级锁之 where 条件为主键 / 唯一索引【等值查询】

- 1、情况一: 当查询的记录是「存在」的, 在索引树上定位到这一条记录后, 将该记录的索引中的 next-key lock 会退化成「记录锁」。

例子：select \* from user where id = 1 for update;那么会为 id=1的这条数据加上 X 型记录锁

给 id 为 1 的这条记录加 X 型的记录锁

id 列：	1	5	10	15	20
name 列：	路飞	索隆	山治	乌索普	香克斯
age 列：	19	21	22	20	39

当条件为主键时只需通过加记录锁就可以解决幻读

- 1、由于主键具有唯一性，所以其他事务插入 id = 1 的时候，会因为主键冲突，导致无法插入 id = 1 的新记录。这样事务 A 在多次查询 id = 1 的记录的时候，不会出现前后两次查询的结果集不同，也就避免了幻读的问题。
- 2、由于对 id = 1 加了记录锁，其他事务无法删除该记录，这样事务 A 在多次查询 id = 1 的记录的时候，不会出现前后两次查询的结果集不同，也就避免了幻读的问题。

- 2、情况二：当查询的记录是「不存在」的，在索引树找到第一条大于该查询记录的记录后，将该记录的索引中的 next-key lock 会退化成「间隙锁」。

例子：select \* from user where id = 2 for update; 生成了(1,5)的间隙锁【此时无法插入 id 为 2、3、4 数据】

X 型间隙锁：(1, 5)

id 列：	1	5	10	15	20
name 列：	路飞	索隆	山治	乌索普	香克斯
age 列：	19	21	22	20	39

为什么唯一索引等值查询并且查询记录「不存在」的场景下，在索引树找到第一条大于该查询记录的记录后，要将该记录的索引中的 next-key lock 会退化成「间隙锁」以及为何通过「间隙锁」就可以解决幻读？

- 1、为什么 id = 5 记录上的主键索引的锁不可以是 next-key lock？如果是 next-key lock，就意味着其他事务无法删除 id = 5 这条记录，但是这次的案例是查询 id = 2 的记录，只要保证前后两次查询 id = 2 的结果集相同，就能避免幻读的问题了，所以即使 id = 5 被删除，也不会有什么影响，那就没必要加 next-key lock，因此只需

要在  $id = 5$  加间隙锁，避免其他事务插入  $id = 2$  的新记录就行了。

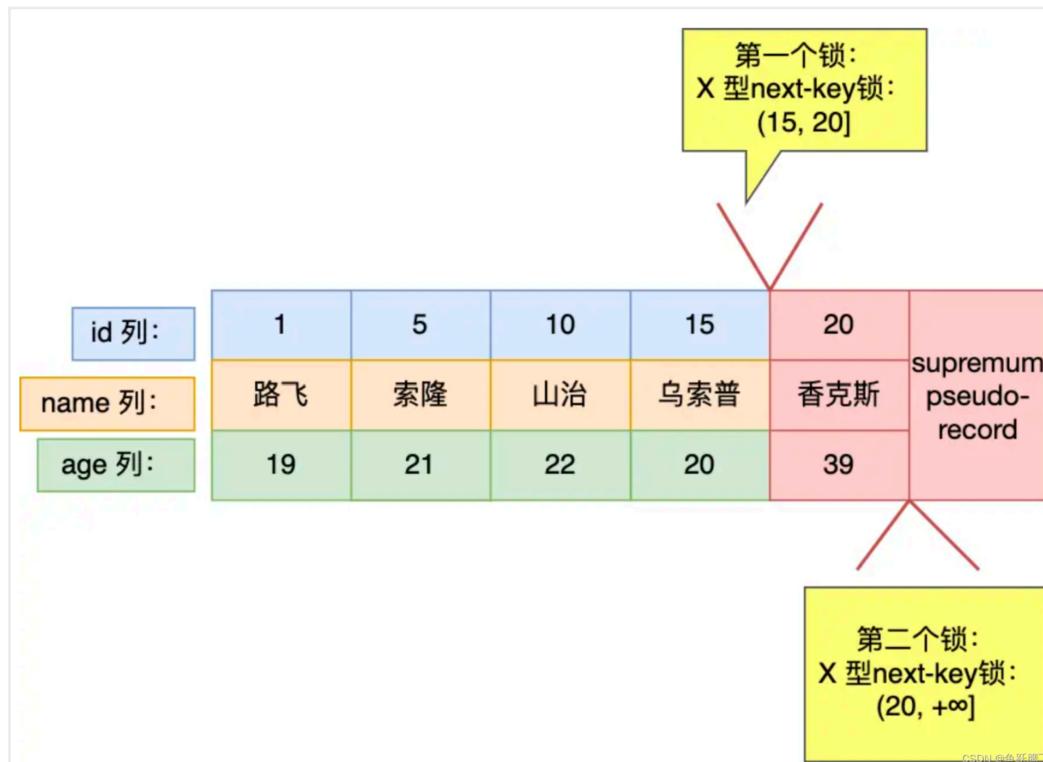
2、为什么不可以针对不存在的记录加记录锁？锁是加在索引上的，而这个场景下查询的记录是不存在的，自然就没办法锁住这条不存在的记录。

## 加行级锁之 where 条件为主键/唯一索引【范围查询】

- 1、情况一：针对「大于等于」的范围查询，因为存在等值查询的条件，那么如果等值查询的记录是存在于表中，那么该记录的索引中的 next-key 锁会退化成记录锁。

### 「大于」情况：

例子：select \* from user where id > 15 for update；【产生两把锁】

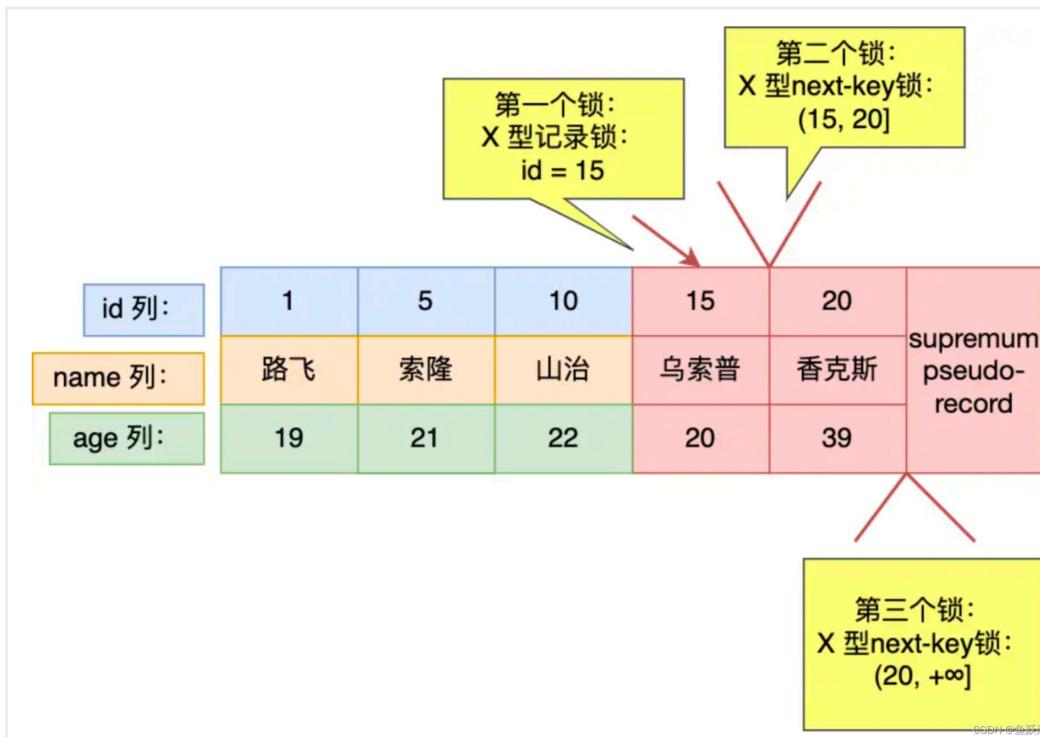


1、在  $id = 20$  这条记录的主键索引上，加了范围为  $(15, 20]$  的 next-key 锁，意味着其他事务即无法更新或者删除  $id = 20$  的记录，同时无法插入  $id$  值为 16、17、18、19 的这一些新记录。

2、在特殊记录（supremum pseudo-record）的主键索引上，加了范围为  $(20, +\infty]$  的 next-key 锁，意味着其他事务无法插入  $id$  值大于 20 的这一些新记录。

### 「大于等于」情况：

例子：select \* from user where id >= 15 for update；【产生三把锁】



- 最开始要找的第一行是  $id = 15$ , 由于查询该记录是一个等值查询(等于 15), 所以该主键索引的 next-key 锁会退化成记录锁, 也就是仅锁住  $id = 15$  这一行记录【使其这条记录不能更新也不能删除】。
- 由于是范围查找, 就会继续往后找存在的记录, 扫描到的第二行是  $id = 20$ , 于是对该主键索引加的是范围为  $(15, 20]$  的 next-key 锁;
- 接着扫描到第三行的时候, 扫描到了特殊记录(supremum pseudo-record), 于是对该主键索引加的是范围为  $(20, +\infty]$  的 next-key 锁。
- 停止扫描。

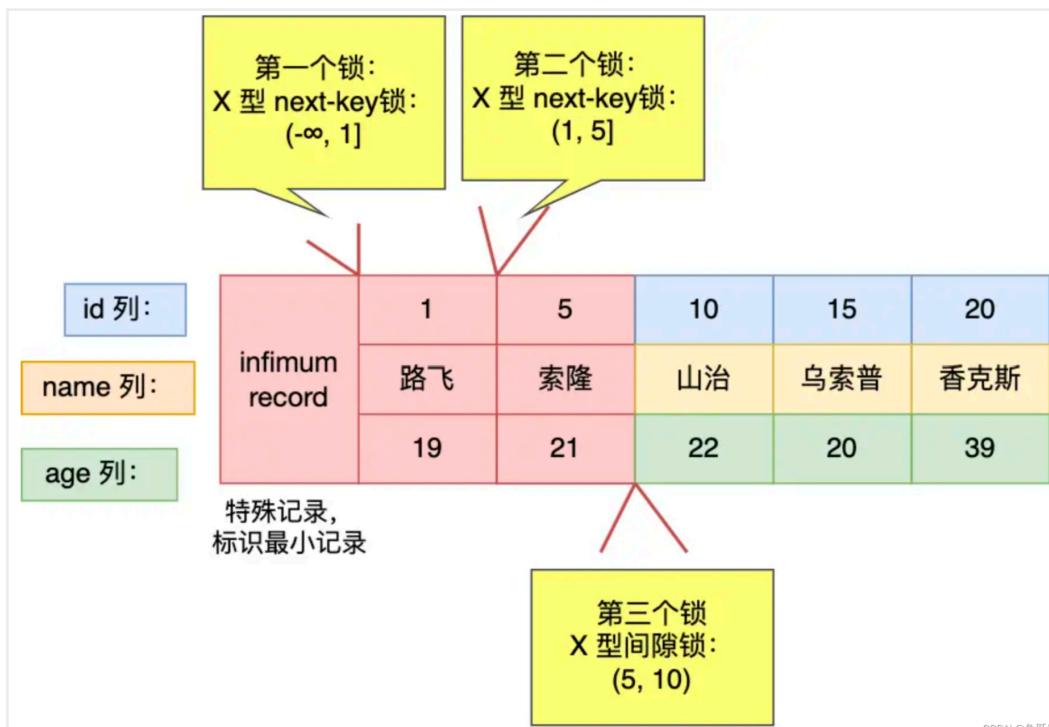
- 2、情况二：针对「小于或者小于等于」的范围查询，要看条件值的记录是否存在于表中：

(1) 当条件值的记录不在表中, 那么不管是「小于」还是「小于等于」条件的范围查询, 扫描到终止范围查询的记录时, 该记录的索引的 next-key 锁会退化成间隙锁, 其他扫描到的记录, 都是在这些记录的索引上加 next-key 锁【记录不存在索引位置会退化成间隙锁】。

(2) 当条件值的记录在表中, 如果是「小于」条件的范围查询, 扫描到终止范围查询的记录时, 该记录的索引的 next-key 锁会退化成间隙锁, 其他扫描到的记录, 都是在这些记录的索引上加 next-key 锁; 如果「小于等于」条件的范围查询, 扫描到终止范围查询的记录时, 该记录的索引 next-key 锁不会退化成间隙锁。其他扫描到的记录, 都是在这些记录的索引上加 next-key 锁。【记录存在, 小于条件索引位置变成间隙锁, 小于等于条件索引位置还是 next-key 锁】

「小于」的范围查询时：

例子：select \* from user where id < 6 for update; 【产生三把锁】



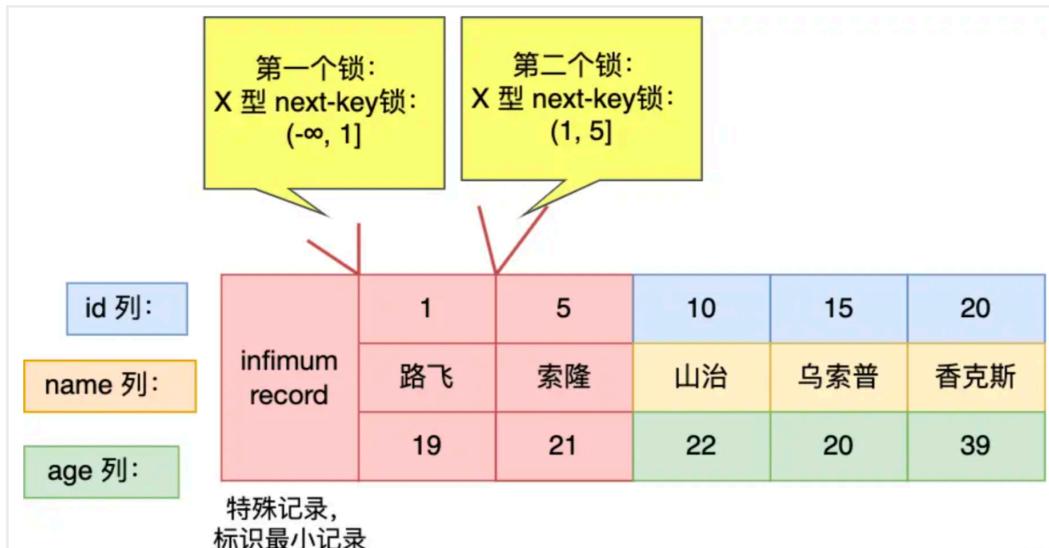
1、在  $\text{id} = 1$  这条记录的主键索引上，加了范围为  $(-\infty, 1]$  的 next-key 锁，意味着其他事务即无法更新或者删除  $\text{id} = 1$  的这一条记录，同时也无法插入  $\text{id}$  小于 1 的这一些新记录。

2、在  $\text{id} = 5$  这条记录的主键索引上，加了范围为  $(1, 5]$  的 next-key 锁，意味着其他事务即无法更新或者删除  $\text{id} = 5$  的这一条记录，同时也无法插入  $\text{id}$  值为 2、3、4 的这一些新记录。

3、在  $\text{id} = 10$  这条记录的主键索引上，加了范围为  $(5, 10)$  的间隙锁，意味着其他事务无法插入  $\text{id}$  值为 6、7、8、9 的这一些新记录。

#### 「小于等于」的范围查询时：

例子：select \* from user where  $\text{id} \leq 5$  for update; 【产生两把锁】



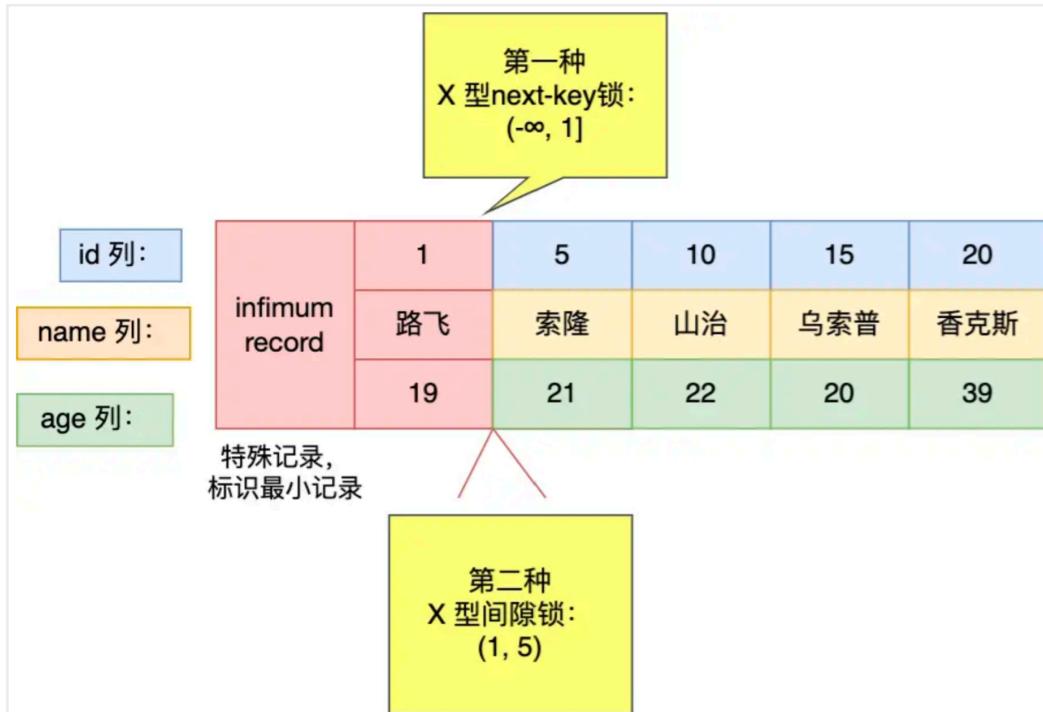
1、在  $\text{id} = 1$  这条记录的主键索引上，加了范围为  $(-\infty, 1]$  的 next-key 锁。意味着其他事务即无法更新或者删除  $\text{id} = 1$  的这一条记录，同时也无法插入  $\text{id}$  小于 1 的这一些新记录。

2、在  $\text{id} = 5$  这条记录的主键索引上，加了范围为  $(1, 5]$  的 next-key 锁。意味着其他事务即无法更新或者删除  $\text{id} = 5$  的这一条记录，同时也无法插入  $\text{id}$  值为 2、3、4 的这一些新记录。

的这一些新记录。

「小于」的范围查询时：

例子：select \* from user where id < 5 for update;【产生两把锁】



1、在  $id = 1$  这条记录的主键索引上，加了范围为  $(-\infty, 1]$  的 next-key 锁，意味着其他事务即无法更新或者删除  $id = 1$  的这一条记录，同时也无法插入  $id$  小于 1 的这一些新记录。

2、在  $id = 5$  这条记录的主键索引上，加了范围为  $(1, 5)$  的间隙锁，意味着其他事务无法插入  $id$  值为 2、3、4 的这一些新记录。

### 加行级锁之 where 条件为二级索引【等值查询】

- 1、情况一：当查询的记录「不存在」时，扫描到第一条不符合条件的二级索引记录，该二级索引的 next-key 锁会退化成间隙锁。因为不存在满足查询条件的记录，所以不会对主键索引加锁。

「针对非唯一索引等值查询时，查询的值不存在的情况」：

例子：select \* from user where age = 25 for update;【产生一把锁】



1、定位到第一条不符合查询条件的二级索引记录，即扫描到  $age = 39$ ，于是该二级索引的 next-key 锁会退化成间隙锁，范围是  $(22, 39)$ 。

2、停止查询

## 什么情况下可以在间隙锁下插入数据？

首先了解下二级索引在 B+Tree 下数据是怎么存放的：二级索引树是按照二级索引值（age 列）按顺序存放的，在相同的二级索引值情况下，再按主键 id 的顺序存放。知道了这个前提，我们才能知道执行插入语句的时候，插入的位置的下一条记录是谁。

基于前面的实验，事务 A 是在 age = 39 记录的二级索引上，加了 X 型的间隙锁，范围是 (22, 39)。

插入 age = 22 记录的成功和失败的情况分别如下：

- 当其他事务插入一条 age = 22, id = 3 的记录的时候，在二级索引树上定位到插入的位置，而该位置的下一条是 id = 10、age = 22 的记录，该记录的二级索引上没有间隙锁，所以这条插入语句可以执行成功。
- 当其他事务插入一条 age = 22, id = 12 的记录的时候，在二级索引树上定位到插入的位置，而该位置的下一条是 id = 20、age = 39 的记录，正好该记录的二级索引上有间隙锁，所以这条插入语句会被阻塞，无法插入成功。

插入 age = 39 记录的成功和失败的情况分别如下：

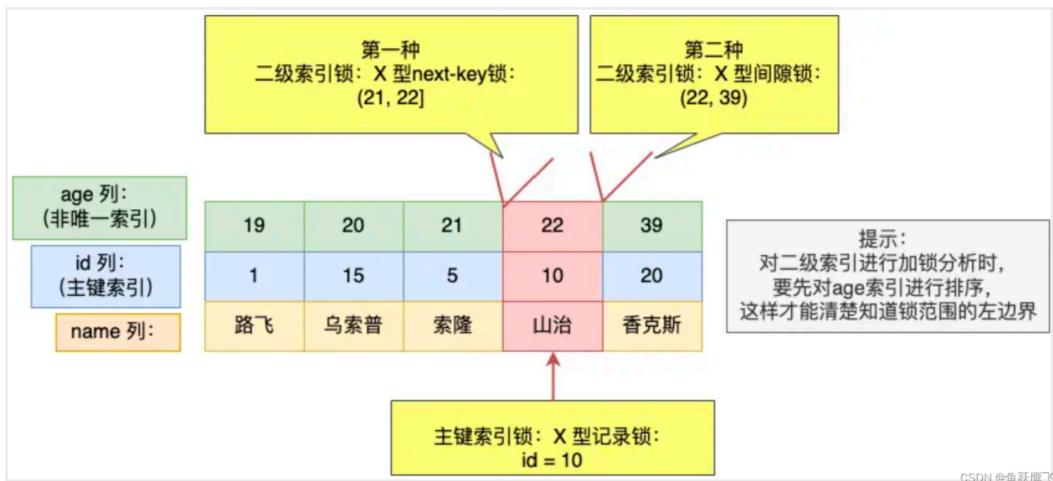
- 当其他事务插入一条 age = 39, id = 3 的记录的时候，在二级索引树上定位到插入的位置，而该位置的下一条是 id = 20、age = 39 的记录，正好该记录的二级索引上有间隙锁，所以这条插入语句会被阻塞，无法插入成功。
- 当其他事务插入一条 age = 39, id = 21 的记录的时候，在二级索引树上定位到插入的位置，而该位置的下一条记录不存在，也就没有间隙锁了，所以这条插入语句可以插入成功。

所以，当有一个事务持有二级索引的间隙锁 (22, 39) 时，插入 age = 22 或者 age = 39 记录的语句是否可以执行成功，关键还要考虑插入记录的主键值，因为「二级索引值 (age 列) + 主键值 (id 列)」才可以确定插入的位置，确定了插入位置后，就要看插入的位置的下一条记录是否有间隙锁，如果有间隙锁，就会发生阻塞，如果没有间隙锁，则可以插入成功。

- 2、情况二：当查询的记录「存在」时，由于不是唯一索引，所以肯定存在索引值相同的记录，于是非唯一索引等值查询的过程是一个扫描的过程，直到扫描到第一个不符合条件的二级索引记录就停止扫描，然后在扫描的过程中，对扫描到的二级索引记录加的是 next-key 锁，而对于第一个不符合条件的二级索引记录，该二级索引的 next-key 锁会退化成间隙锁。同时，在符合查询条件的记录的主键索引上加记录锁。

「针对非唯一索引等值查询时，查询的值存在的情况」：

例子：select \* from user where age = 22 for update;【产生三把锁】



主键索引：

- 在  $\text{id} = 10$  这条记录的主键索引上，加了记录锁，意味着其他事务无法更新或者删除  $\text{id} = 10$  的这一行记录。

二级索引 (非唯一索引)：

- 在  $\text{age} = 22$  这条记录的二级索引上，加了范围为  $(21, 22]$  的 next-key 锁，意味着其他事务无法更新或者删除  $\text{age} = 22$  的这一些新记录【不过可以插入  $\text{age}=21$  下  $\text{id}<5$  的数据】。
- 在  $\text{age} = 39$  这条记录的二级索引上，加了范围  $(22, 39)$  的间隙锁。意味着其他事务无法插入  $\text{age}$  值为  $23, 24, \dots, 38$  的这一些新记录【不过可以插入  $\text{age}=39$  下  $\text{id}>20$  的数据】。

### 什么情况下可以在间隙锁下插入数据？

在  $\text{age} = 22$  这条记录的二级索引上，加了范围为  $(21, 22]$  的 next-key 锁，意味着其他事务无法更新或者删除  $\text{age} = 22$  的这一些新记录，针对是否可以插入  $\text{age} = 21$  和  $\text{age} = 22$  的新记录，分析如下：

- 是否可以插入  $\text{age} = 21$  的新记录，还要看插入的新记录的  $\text{id}$  值，如果插入  $\text{age} = 21$  新记录的  $\text{id}$  值小于 5，那么就可以插入成功，因为此时插入的位置的下一条记录是  $\text{id} = 5$ ,  $\text{age} = 21$  的记录，该记录的二级索引上没有间隙锁。如果插入  $\text{age} = 21$  新记录的  $\text{id}$  值大于 5，那么就无法插入成功，因为此时插入的位置的下一条记录是  $\text{id} = 10$ ,  $\text{age} = 22$  的记录，该记录的二级索引上有间隙锁【所以在 21 和 22 之间加了 next-key 锁，只要是 21-5 到 22-10 之间都不允许数据插入，防止插入 22-9 类似的情况数据插入，保证了  $\text{age}=22$  左边这块不会出现幻读】。
- 是否可以插入  $\text{age} = 22$  的新记录，还要看插入的新记录的  $\text{id}$  值，从  $\text{LOCK\_DATA} : 22, 10$  可以得知，其他事务插入  $\text{age}$  值为  $22$  的新记录时，如果插入的新记录的  $\text{id}$  值小于 10，那么插入语句会发生阻塞；如果插入的新记录的  $\text{id}$  大于 10，还要看该新记录插入的位置的下一条记录是否有间隙锁，如果没有间隙锁则可以插入成功，如果有间隙锁，则无法插入成功。

在  $\text{age} = 39$  这条记录的二级索引上，加了范围  $(22, 39)$  的间隙锁。意味着其他事务无法插入  $\text{age}$  值为  $23, 24, \dots, 38$  的这一些新记录，针对是否可以插入  $\text{age} = 22$  和  $\text{age} = 39$  的新记录，分析如下：

- 是否可以插入  $\text{age} = 22$  的新记录，还要看插入的新记录的  $\text{id}$  值，如果插入  $\text{age} = 22$  新记录的  $\text{id}$  值小于 10，那么插入语句会被阻塞，无法插入，因为此时插入的位置的下一条记录是  $\text{id} = 10$ ,  $\text{age} = 22$  的记录，该记录的二级索引

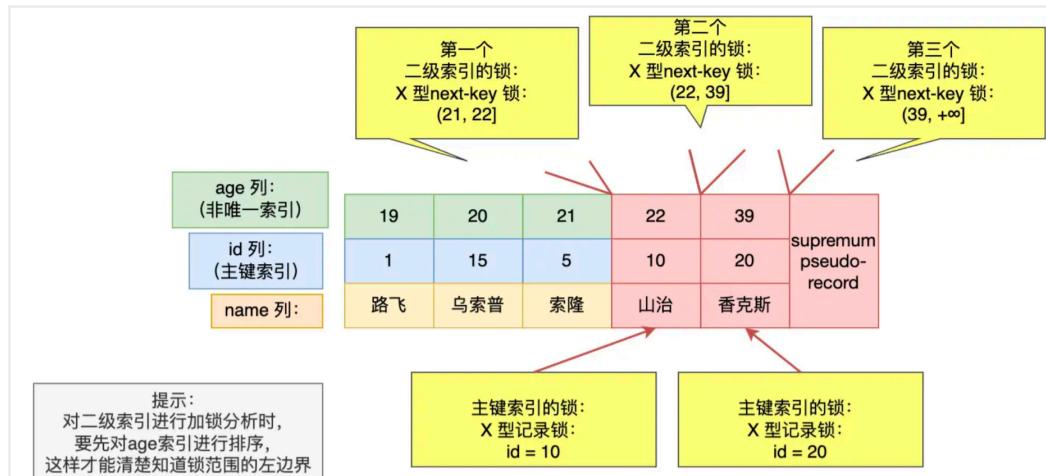
上有间隙锁 ( $age = 22$  这条记录的二级索引上有 next-key 锁)。如果插入  $age = 22$  新记录的 id 值大于 10，也无法插入，因为此时插入的位置的下一条记录是  $id = 20, age = 39$  的记录，该记录的二级索引上有间隙锁。

- 是否可以插入  $age = 39$  的新记录，还要看插入的新记录的 id 值，从  $LOCK\_DATA : 39, 20$  可以得知，其他事务插入  $age$  值为 39 的新记录时，如果插入的新记录的 id 值小于 20，那么插入语句会发生阻塞，如果插入的新记录的 id 大于 20，则可以插入成功。

## 加行级锁之 where 条件为二级索引【范围查询】

非唯一索引和主键索引的范围查询的加锁也有所不同，不同之处在于非唯一索引范围查询，索引的 **next-key lock** 不会有退化为间隙锁和记录锁的情况，也就是非唯一索引进行范围查询时，对二级索引记录加锁都是加 next-key 锁。

例子：select \* from user where age >= 22 for update;【产生五把锁】



1、最开始要找的第一行是  $age = 22$ ，虽然范围查询语句包含等值查询，但是这里不是唯一索引范围查询，所以是不会发生退化锁的现象，因此对该二级索引记录加 next-key 锁，范围是  $(21, 22]$ 。同时，对  $age = 22$  这条记录的主键索引加记录锁，即对  $id = 10$  这一行记录的主键索引加记录锁。

2、由于是范围查询，接着继续扫描已经存在的二级索引记录。扫描的第二行是  $age = 39$  的二级索引记录，于是对该二级索引记录加 next-key 锁，范围是  $(22, 39]$ ，同时，对  $age = 39$  这条记录的主键索引加记录锁，即对  $id = 20$  这一行记录的主键索引加记录锁。

3、虽然我们看见表中最后一条二级索引记录是  $age = 39$  的记录，但是实际在 InnoDB 存储引擎中，会用一个特殊的记录来标识最后一条记录，该特殊的记录的名字叫 supremum pseudo-record，所以扫描第二行的时候，也就扫描到了这个特殊记录的时候，会对该二级索引记录加的是范围为  $(39, +∞]$  的 next-key 锁。

4、停止查询

## 加行级锁之 where 条件为不为索引

如果锁定读查询语句，没有使用索引列作为查询条件，或者查询语句没有走索引查询，导致扫描是全表扫描。那么，每一条记录的索引上都会加 next-key 锁，这样就相当于锁住的全表，这时如果其他事务对该表进行增、删、改操作的时候，都会被阻塞。

不只是锁定读查询语句不加索引才会导致这种情况，update 和 delete 语句如果查

询条件不加索引，那么由于扫描的方式是全表扫描，于是就会对每一条记录的索引上都会加 next-key 锁，这样就相当于锁住的全表。

因此，在线上在执行 update、delete、select ... for update 等具有加锁性质的语句，一定要检查语句是否走了索引，如果是全表扫描的话，会对每一个索引加 next-key 锁，相当于把整个表锁住了，这是挺严重的问题。

## 总结

- 唯一索引等值查询：

1、当查询的记录是「存在」的，在索引树上定位到这一条记录后，将该记录的索引中的 next-key lock 会退化成「记录锁」。

2、当查询的记录是「不存在」的，在索引树找到第一条大于该查询记录的记录后，将该记录的索引中的 next-key lock 会退化成「间隙锁」。

- 非唯一索引等值查询：

1、当查询的记录「存在」时，由于不是唯一索引，所以肯定存在索引值相同的记录，于是非唯一索引等值查询的过程是一个扫描的过程，直到扫描到第一个不符合条件的二级索引记录就停止扫描，然后在扫描的过程中，对扫描到的二级索引记录加的是 next-key 锁，而对于第一个不符合条件的二级索引记录，该二级索引的 next-key 锁会退化成间隙锁。同时，在符合查询条件的记录的主键索引上加记录锁。

2、当查询的记录「不存在」时，扫描到第一条不符合条件的二级索引记录，该二级索引的 next-key 锁会退化成间隙锁。因为不存在满足查询条件的记录，所以不会对主键索引加锁。

- 非唯一索引和主键索引的范围查询的加锁规则不同之处在于：

唯一索引在满足一些条件的时候，索引的 next-key lock 退化为间隙锁或者记录锁。

非唯一索引范围查询，索引的 next-key lock 不会退化为间隙锁和记录锁。

## 个人总结

1、唯一索引等值查询可能存在记录锁+间隙锁，唯一索引范围查询可能存在 next-key+记录锁+间隙锁。

2、二级索引等值查询可能存在 next-key 锁 + 记录锁 + 间隙锁，二级索引范围查询可能存在 next-key 锁 + 记录锁。

- 唯一索引等值查询：记录存在【记录锁】，记录不存在【间隙锁】
- 唯一索引范围查询：大于【next-key 锁】，记录存在&大于等于【记录锁 + next-key 锁】，小于【next-key 锁 + 间隙锁】，小于等于【next-key 锁】
- 二级索引等值查询：记录存在【next-key 锁 + 记录锁 + 间隙锁】，记录不存在【间隙锁】
- 二级索引范围查询：记录存在【next-key 锁 + 记录锁】，记录不存在【next-key 锁】

博客参考：[https://blog.csdn.net/Chang\\_Yafei/article/details/131180391](https://blog.csdn.net/Chang_Yafei/article/details/131180391)

# MySQL 索引详解

## 索引的优缺点

### 优点

- 加快检索速度
- 创建唯一索引可以保证每行数据的唯一性

### 缺点

- 创建索引和维护索引需要耗费许多时间。当对表中的数据进行增删改的时候，如果数据有索引，那么索引也需要动态的修改，会降低 SQL 执行效率。
- 索引需要使用物理文件存储，也会耗费一定空间。

## 索引底层数据结构选型

- Hash 表

Hash 索引不支持顺序和范围查询。假如我们要对表中的数据进行排序或者进行范围查询，那 Hash 索引可就不行了。并且，每次 IO 只能取一个。例如查询  $id < 500$  的数据，需要针对这 500 条数据进行 500 次的哈希计算定位查询。

- 二叉树

非平衡二叉树，可能导致查询的时间复杂度变成  $O(n)$

- AVL 树

平衡二叉树的一种，但是 AVL 树在插入和删除操作时需要频繁地进行旋转操作来保持平衡【为了追求严格的平衡】，因此会有较大的计算开销进而降低了插入和删除性能。并且，在使用 AVL 树时，每个树节点仅存储一个数据，而每次进行磁盘 IO 时只能读取一个节点的数据，如果需要查询的数据分布在多个节点上，那么就需要进行多次磁盘 IO。磁盘 IO 是一项耗时的操作，在设计数据库索引时，我们需要优先考虑如何最大限度地减少磁盘 IO 操作的次数。

- 红黑树

和 AVL 树不同的是，红黑树并不追求严格的平衡，而是大致的平衡。正因如此，红黑树的查询效率稍有下降，因为红黑树的平衡性相对较弱，可能会导致树的高度较高，这可能会导致一些数据需要进行多次磁盘 IO 操作才能查询到，这也是 MySQL 没有选择红黑树的主要原因。也正因如此，红黑树的插入和删除操作效率大大提高了，因为红黑树在插入和删除节点时只需进行  $O(1)$  次数的旋转和变色操作，即可保持基本平衡状态，而不需要像 AVL 树一样进行  $O(\log n)$  次数的旋转操作。

## AVL 树 VS 红黑树

MySQL 没有选择 AVL 树和红黑树的原因主要是每次查询只能查一条数据，如果批量查询会导致磁盘的多次 IO 操作【两者树的高度都会比较高，树的高度决定磁盘 IO 读取次数，而 B+ 树一般最多也就三层，IO 次数很低】。

AVL 树优点在于查询比红黑树快【严格的平衡】，红黑树的优点在于插入删除操作比 AVL 树快【因为红黑树追求大致平衡，所以树的高度相对比 AVL 高，因此查询效率会比 AVL 树低】。

## B 树 VS B+ 树

MySQL 选择 B+ 树作为索引结构的主要原因有以下几点：

1. **范围查询效率高**: B+ 树的叶子节点形成一个有序链表，范围查询时可以通过遍历这个有序链表来获取范围内的数据，效率较高。而 B 树则需要在不同层级进行搜索，效率相对较低。
2. **磁盘 IO 友好**: B+ 树的内部节点只存储索引信息，不存储实际数据，而叶子节点形成一个有序链表，可以减少磁盘 IO 次数，提高查询效率。
3. **适合范围查询和排序操作**: 在实际数据库应用中，范围查询和排序操作是比较常见的，而 B+ 树对于这类操作具有较好的性能表现。
4. **支持稳定性范围查询**: B+ 树的叶子节点形成有序链表，支持稳定性范围查询，即在相同条件下查询结果的顺序是固定的，这对一些需要保持结果顺序的场景非常重要。

总的来说，MySQL 选择 B+ 树作为索引结构，主要是考虑到 B+ 树在范围查询、磁盘 IO 友好和稳定性范围查询等方面的优势。这使得 B+ 树成为了较为常用的数据库索引结构之一。

**【B+ 树只有叶子结点存储数据，不像 b 树非叶子节点也会存储数据，所以 b+ 树比 b 树更矮，IO 次数更少，并且 b+tree 支持范围查询和随机单个查询】**

## 选择 B+ 树作为 MySQL 数据结构原因

树的高度越高，MySQL 的 IO 次数越多的主要原因是由于数据库系统在进行查询操作时，需要通过访问磁盘上的数据块来获取索引信息和实际数据。而树的高度直接影响了需要访问的磁盘块数，从而影响了 IO 次数的多少。

在一个平衡树结构（比如 B+ 树）中，查询操作需要从根节点开始逐级向下查找，直到叶子节点才能获取到实际数据。如果树的高度较低，即使在多层索引结构中，查询操作也需要经过较少的层级，访问的磁盘块数相对较少，IO 次数也就相对较少。相反，如果树的高度较高，查询操作需要经过更多层级的索引节点才能到达叶子节点获取数据，这意味着需要访问更多的磁盘块，导致 IO 次数增多。

因此，树的高度越高，查询操作需要访问的磁盘块数就越多，进而导致 MySQL 的 IO 次数也就越多。为了提高查询效率，数据库系统通常会倾向于选择能够保持树结构较低高度的索引结构，以减少 IO 次数。

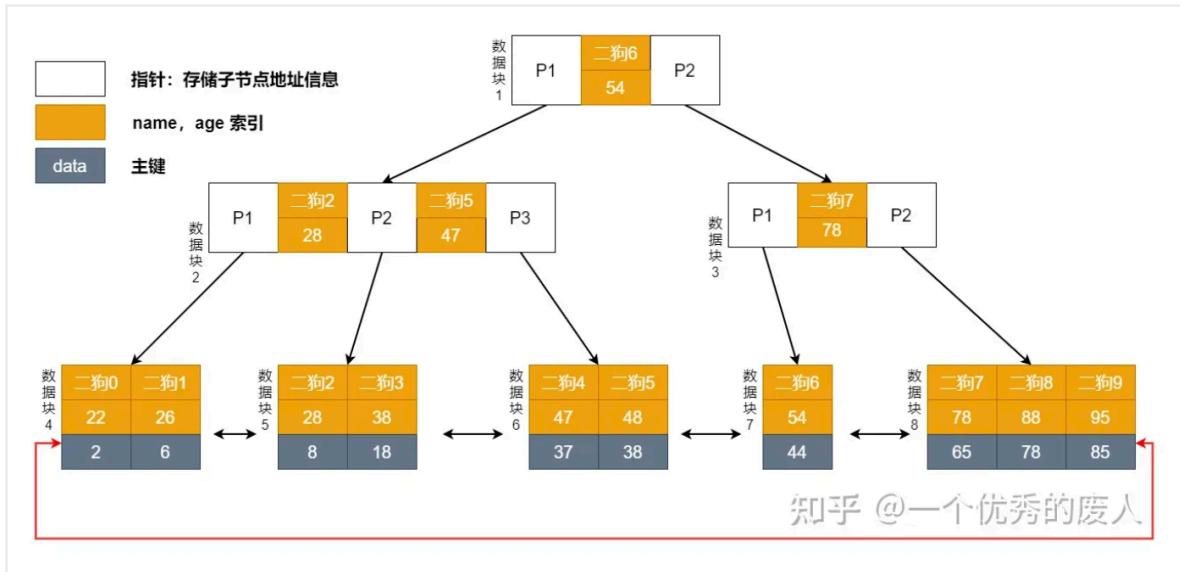
因此，**数据库设计中通常会倾向于选择能够保持树结构较低高度的索引结构，以减少 IO 次数，提高查询效率**。这也是为什么 B+ 树等平衡树结构被广泛应用于数据库索引的原因之一。

## B+Tree 特性带来的优点

- 1、设计索引时选择唯一值比例较高的为索引，提高检索速度
- 2、创建组合索引来提高多条件查询的速度，并且减少索引占用空间和维护成本
- 3、利用覆盖索引的特性减少回表操作
- 4、b+ 树子节点存储数据且有序性，可以用于范围查询
- 5、由于 b+tree 的子节点才存数据，非子节点存储的索引，那么一页 16KB 可以存储更多的索引数据，那么可以一页可以加载更多的索引数据到 buffer pool，减少磁盘 IO 操作**【buffer pool 既存储叶子结点也存储非叶子结点】**。
- 6、B+tree 的高度较矮，因此磁盘 IO 次数也少

## 【矮范缓组覆】

### MySQL的组合索引为什么不能最右匹配？



如上图是一个组合索引，优先按照 name 排序然后再按照 age 排序，相同的 name 的话会按照 age 再进行排序。既然知道了这种排序方式，就该知道 age 的有序只存在于相同 name 下，如果要进行最右匹配会导致全表扫描，因为 age 在全表来看就是无序的。

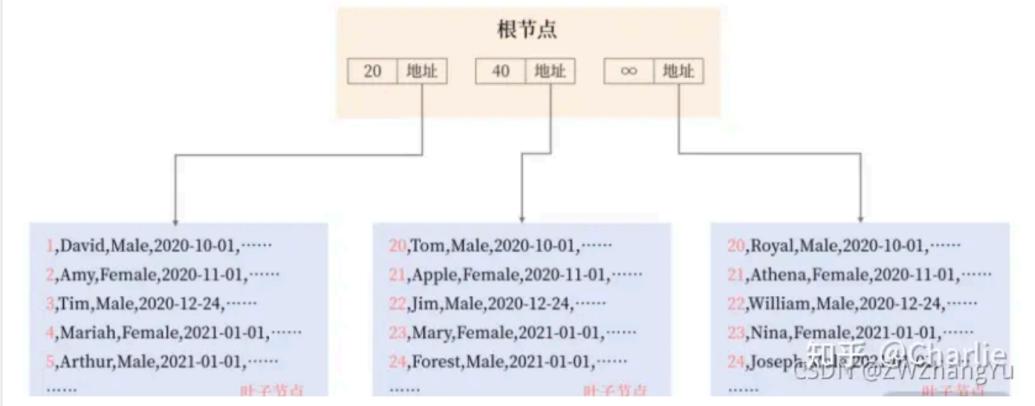
## B+ 树数据结构特点

MySQL 支持多种类型的索引，其中最常用的是 B+ 树索引。B+ 树是一种平衡的多路搜索树，它有以下几个特点：

- B+ 树的每个节点可以存储多个关键字，关键字之间按照一定的顺序排列。
- B+ 树的非叶子节点只存储关键字和指向子节点的指针，**不存储实际的数据**。
- **B+ 树的所有叶子节点都存储实际的数据**，并且通过指针相互连接，形成一个有序的链表。
- B+ 树的高度相对较低，因为每个节点可以存储多个关键字，所以可以减少树的层数。

## MySQL 单表数据量大小的计算方式

索引键就是排序的列，而指针是指向下一层的地址，在 MySQL 的 InnoDB 存储引擎中占用 6 个字节。下图显示了 B+ 树高度为 2 时，B+ 树索引的样子：



### 以 bigint 类型 (8 字节) 作为索引计算

#### 根节点计算

根节点为 16KB,  $16KB / (8\text{字节} + 6\text{字节的索引指针}) = 1170$ , 因此根节点最多存储 1170 条数据。

#### 第二层计算

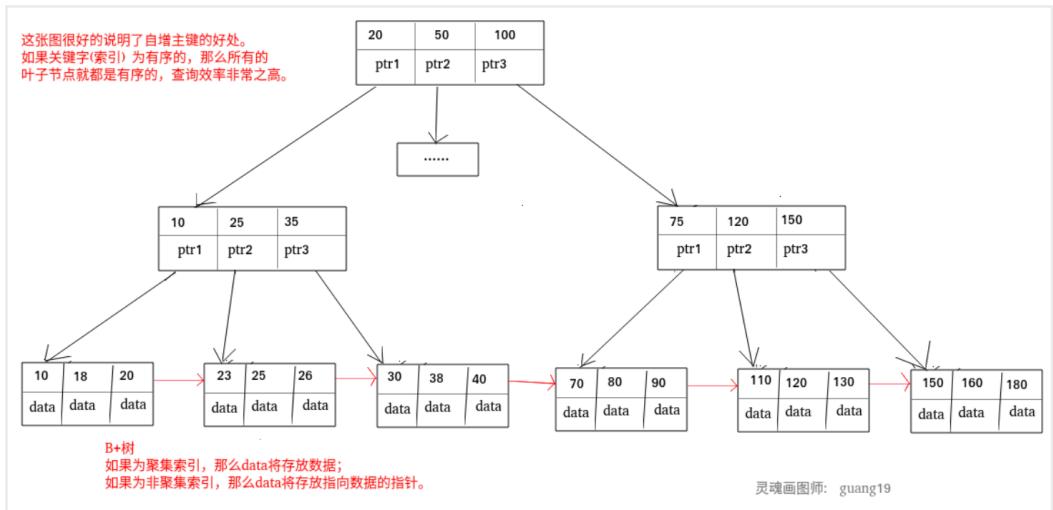
因为每个节点数据结构和根节点一样, 而且在跟节点每个元素都会延伸出来一个节点, 所以第二层的数据量是  $1170 * 1170 = 1368900$  个页。

#### 第三层计算

因为 innodb 的叶子节点, 是直接包含整条 mysql 数据的, 如果字段非常多的话数据所占空间是不小的, 我们这里以 1kb 计算, 所以在第三层, 每个节点为 16kb, 那么每个节点是可以放 16 个数据的, 所以最终 mysql 可以存储的总数据为:  $1170 * 1170 * 16 = 21902400$  (千万级条)

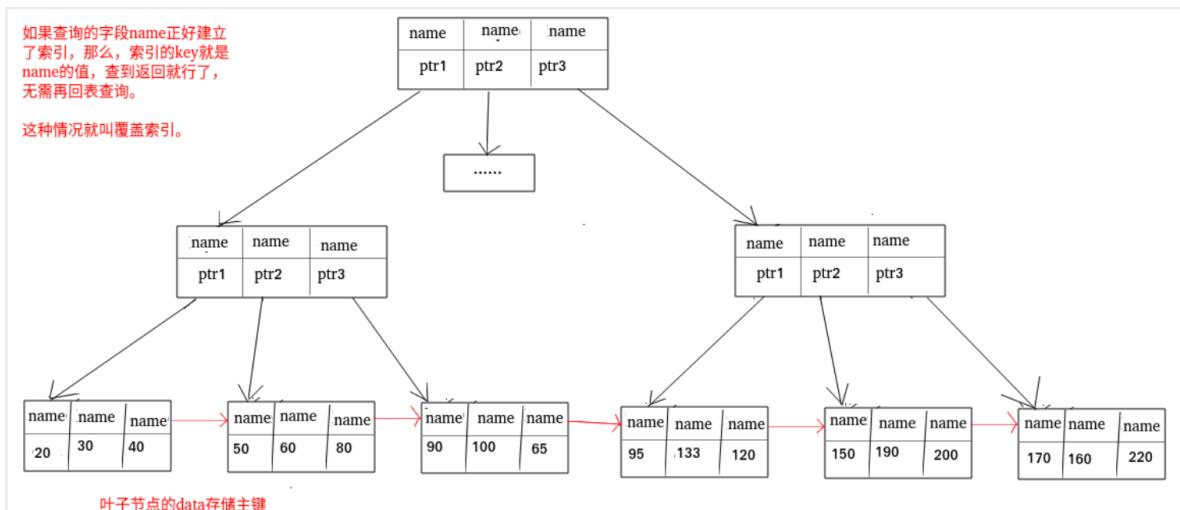
## 聚簇索引&非聚簇索引

- 聚簇索引：即索引结构和数据一起存放的索引，并不是一种单独的索引类型。对于 InnoDB 引擎表来说，该表的索引(B+树)的每个非叶子节点存储索引，叶子节点存储索引和索引对应的数据。【主键索引属于聚簇索引，相比于非聚簇索引，聚簇索引少了一次读取数据的 IO 操作】
- 非聚簇索引：即索引结构和数据分开存放的索引，并不是一种单独的索引类型。二级索引(辅助索引)就属于非聚簇索引【二级索引的叶子节点就存放的是主键，根据主键再回表查数据，可能会需要两次查询，速度较慢于聚簇索引】。
- 非聚簇索引未必一定两次查询【覆盖索引】：例如 SELECT id, name FROM table WHERE name='guang19', 为 name 字段建立了索引，无需回表查询【非聚簇索引的叶子节点存放的是主键 + 本身的列值】。



## 覆盖索引&联合索引

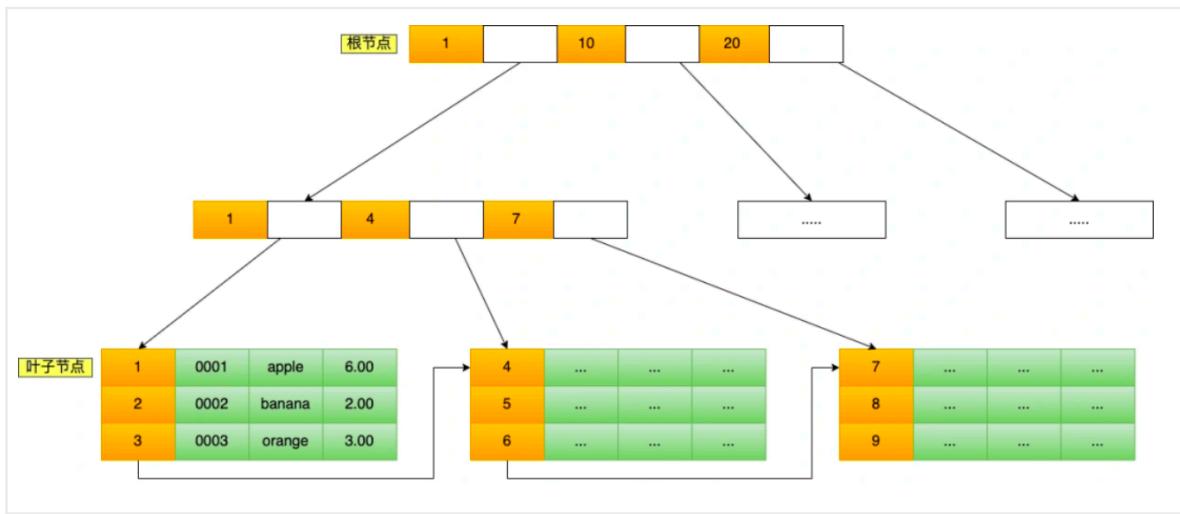
- 覆盖索引：如果一个索引包含（或者说覆盖）所有需要查询的字段的值，我们就称之为 **覆盖索引 (Covering Index)**。我们知道在 InnoDB 存储引擎中，如果不是主键索引，叶子节点存储的是主键 + 列值。最终还是要“回表”，也就是要通过主键再查找一次，这样就会比较慢。而覆盖索引就是把要查询出的列和索引是对应的，不做回表操作【覆盖索引即需要查询的字段正好是索引的字段，那么直接根据该索引，就可以查到数据了，而无需回表查询】。



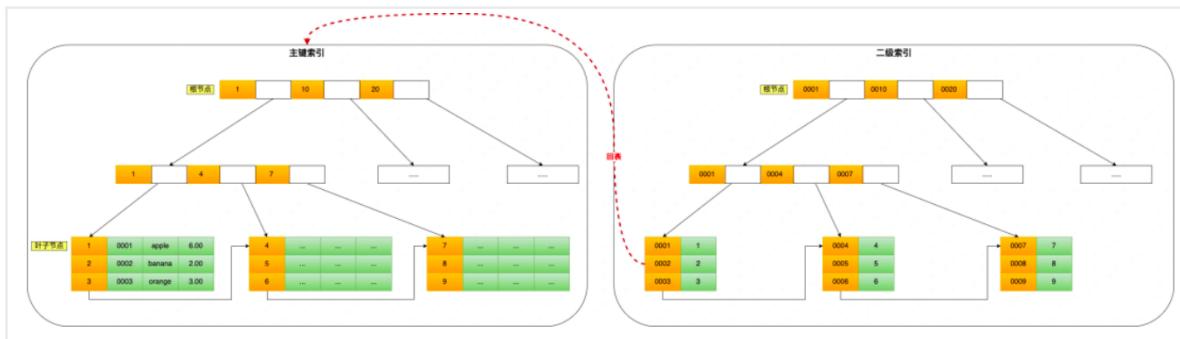
- 联合索引：多个字段组成的索引，要符合最左前缀匹配原则。联合索引的最左匹配原则。但是遇到范围查询时，范围查询的列本身可以使用到索引，以缩小扫描范围。但是，在该范围查询列之后的所有列，都无法再利用索引的有序性进行快速匹配，索引会在此处“停止”工作。常见的范围查询操作包括：>, <, >=, <=, BETWEEN, LIKE '前缀 %'。

## 聚簇索引&二级索引的 B+Tree 图

- 聚簇索引 B+Tree 图



- 二级索引 B+Tree 图



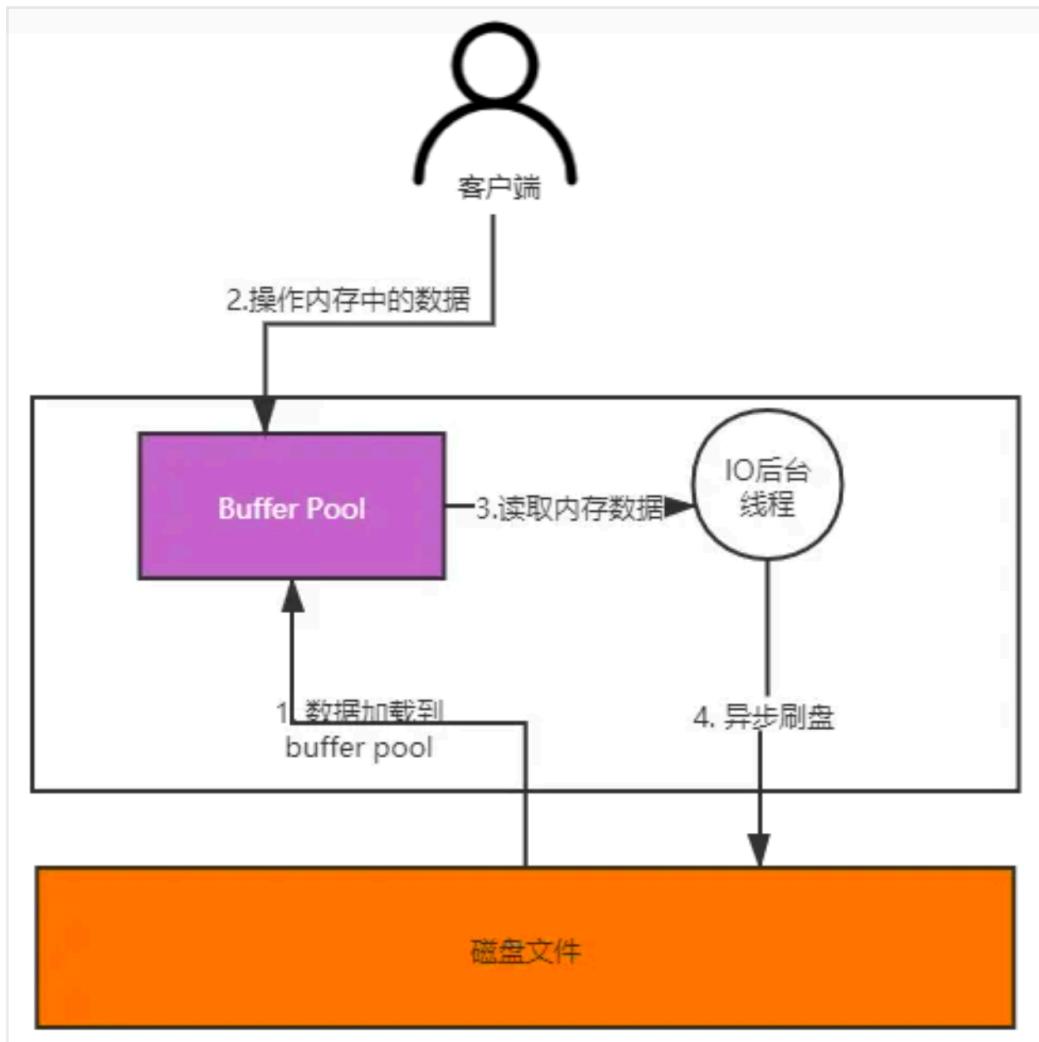
## 正确使用索引的一些建议

- 选择合适的字段创建索引：频繁查询和排序的字段要创建索引，对于可能存在 null 的字段在创建索引的时候数据库难以优化，所以避免字段有 null 的存在，可以用 0 或者 1 替代 null 值。
- 被频繁更新的字段应该慎重建立索引：频繁更新的字段如果创建了索引，容易提高索引维护成本。
- 限制每张表的索引数量：单张表建议不超过 5 个，索引太多会影响插入和更新的效率。并且如果索引过多，在 mysql 生成执行计划的时候发现很多个已创建索引都可以使用，就会增加 MySQL 优化器生成执行计划的时间，同样会降低查询性能。
- 尽可能的考虑建立联合索引而不是单列索引：减少磁盘占用空间
- 注意避免冗余索引：不要同时出现 (a,b) 和 (a)
- 字符串类型的字段使用前缀索引代替普通索引：前缀索引仅限于字符串类型，较普通索引会占用更小的空间，所以可以考虑使用前缀索引代替普通索引。
- 避免索引失效：例如 select \* 容易导致无法使用覆盖索引，以及网络传输等性能问题；或者创建了组合索引但未遵守最左前缀匹配；% 开头的 LIKE 查询比如 LIKE '%abc'；where 条件使用 OR；in 的取值范围过大。等等都会导致索引失效

# Buffer Pool

## Buffer Pool 简介

Buffer Pool【默认 128M】是 MySQL 数据库中的一个重要的内存组件，介于外部系统和存储引擎之间的一个缓存区，数据库的增删改查这些操作都是针对这个内存数据结构中的缓存数据执行的，在操作数据之前，都会将数据从磁盘加载到 Buffer Pool 中，操作完成之后异步刷盘、写 undo log、binlog、redolog 等一些列操作，避免每次访问都进行磁盘 IO 影响性能。



## 为什么 MySQL 会有 Buffer Pool? 作用是什么?

- 减少磁盘 I/O 操作**: Buffer Pool 通过在内存中缓存数据和索引页，使得当数据被访问时，可以直接从内存中读取，而不需要从磁盘中读取。这大大减少了磁盘 I/O 操作的次数，于是内存操作也提高了查询性能。
- 提高数据访问速度**: 由于内存的访问速度远快于磁盘，因此通过 Buffer Pool 缓存数据页，可以显著提高数据的访问速度，从而提升整个数据库系统的性能。
- 支持并发访问**: Buffer Pool 通过缓存数据页，可以减少对磁盘的并发访问冲突，从而支持更多的并发查询操作。
- 优化数据访问模式**: InnoDB 的 Buffer Pool 使用了一种称为“最近最少使用”(LRU) 的缓存淘汰策略，以确保经常访问的数据保留在内存中，从而进一步提高性能。

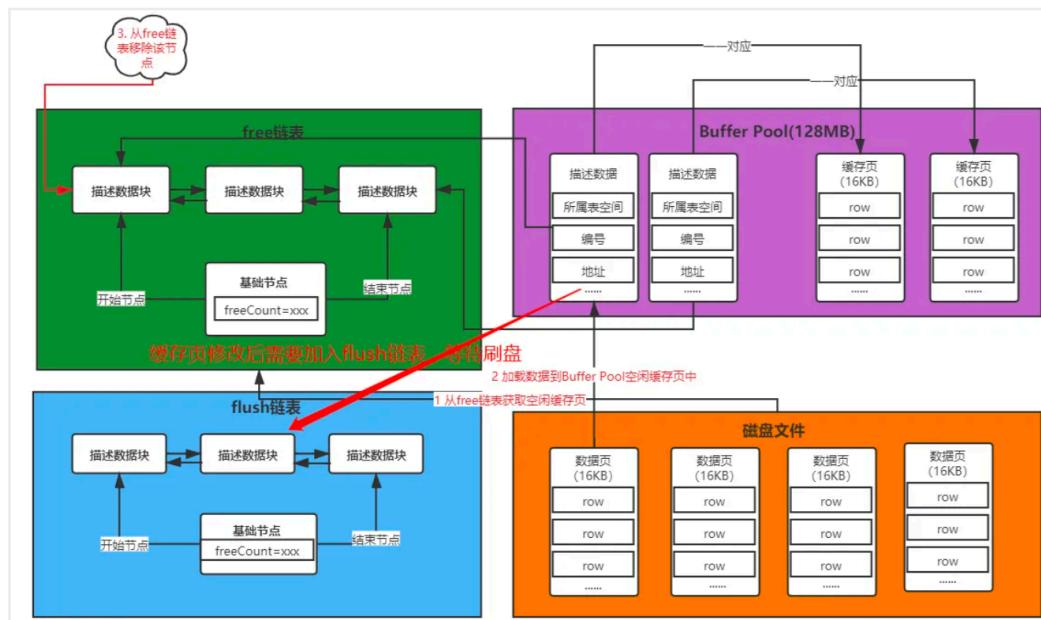
(LRU) 的算法来管理缓存中的数据页。这种算法可以确保最近访问的数据页始终保留在缓存中，从而提高数据的访问效率。

5. 支持数据修改：当数据在 Buffer Pool 中被修改时，InnoDB 会先将修改后的数据页标记为“脏页”，并在后台异步地将这些脏页写回磁盘。这种延迟写回的策略可以减少写操作对性能的影响。

总之，MySQL 的 Buffer Pool 通过减少磁盘 I/O 操作、提高数据访问速度、支持并发访问、优化数据访问模式以及支持数据修改等方式，显著提高了数据库系统的性能。

## Buffer Pool 的数据结构

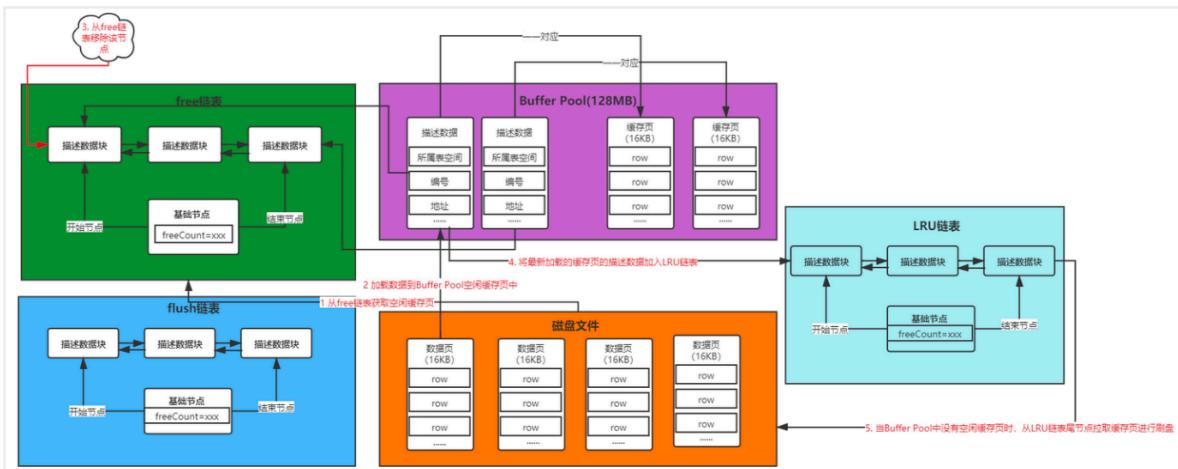
- 缓存页：存储对应从磁盘读取的数据，大小同磁盘页一样都是 16KB
- 描述数据：分别对应一个数据页【即数据库 16KB 数据页】，记录的是缓存页的元数据信息，包括数据页所属表空间、数据页编号、缓存页在 Buffer Pool 中的地址等等。
- free 双向链表：加载磁盘的数据页时用于确定哪些缓存页是空闲的，链表的每个节点就是一个空闲的缓存页的描述数据地址。
- 哈希表数据结构：用表空间号 + 数据页号作为 key，缓存页的地址作为 value。用于 select/update 等操作时，确认数据是否存在 buffer pool 里，不存在则去磁盘的数据页拉取数据。
- flush 双向链表：update 操作时是直接操作 buffer pool 而非操作磁盘，当一条数据修改了后，当前缓存页的数据势必和磁盘的数据页数据不一致，所以当前缓存页也称之为脏页。脏页是需要刷盘来保持缓存页和数据页数据一致。那么我们怎么知道是哪个缓存页是脏页呢，总不能把所有的缓存页都进行一次刷盘吧。所以会通过 flush 链表存储记录描述数据来表示哪些缓存页是脏页，那么刷盘的时候后台线程只需要处理这个链表上面记录的数据页即可，刷盘结束后，将节点从链表上抹去。



## 基于 LRU 算法淘汰 Buffer Pool 内部缓存页

由于不停的加载数据页到 Buffer Pool 中，Buffer Pool 内存空间有限，迟早会出现 Buffer Pool 中没有空闲缓存页的情况，此时如果还需要加载数据进来，就必须有一个数据淘汰策略来淘汰一些缓存页。

那么通过区分经常访问和很少访问的缓存页，引入 LRU 链表来进行记录和区分。



## 一、标准 LRU 算法的缺陷 (为什么需要优化?)

在数据库场景中，标准的 LRU (Least Recently Used, 最近最少使用) 算法存在两个主要缺陷：

### 1. 预读 (Read-Ahead) 失效：

- InnoDB为了提升性能，会进行“预读”。当它认为你可能很快会访问某些数据页时，会提前把它们加载到 Buffer Pool 中。
- 在标准 LRU 中，这些预读进来的页会立刻被放到 LRU 链表的头部（最近使用的位置）。
- **问题在于：**如果这些预读的页，实际上并没有被访问，它们就会占据头部位置，反而导致真正被频繁访问的“热数据”页被更快地挤到链表尾部并被淘汰。这污染了缓存。

### 2. 全表扫描 (Full Table Scan) 的冲击：

- 当执行一个 `SELECT * FROM large_table` 这样的全表扫描时，大量的数据页会被依次读入 Buffer Pool。
- 在标准 LRU 中，这些仅仅被访问一次的数据页，会像洪水一样冲进 LRU 链表的头部。
- **问题在于：**这会导致 Buffer Pool 中所有原有的、可能被频繁访问的“热数据”页，被迅速地推向链表尾部并被淘汰。一次偶然的全表扫描，就可能“清洗”掉整个 Buffer Pool 的有效缓存，导致数据库性能急剧下降。

为了解决这两个问题，InnoDB引入了带有“中点插入策略 (Midpoint Insertion Strategy)”的、分区的 LRU 链表，也就是我们常说的冷热数据分离。

## 二、InnoDB的 LRU 链表：冷热数据分离结构

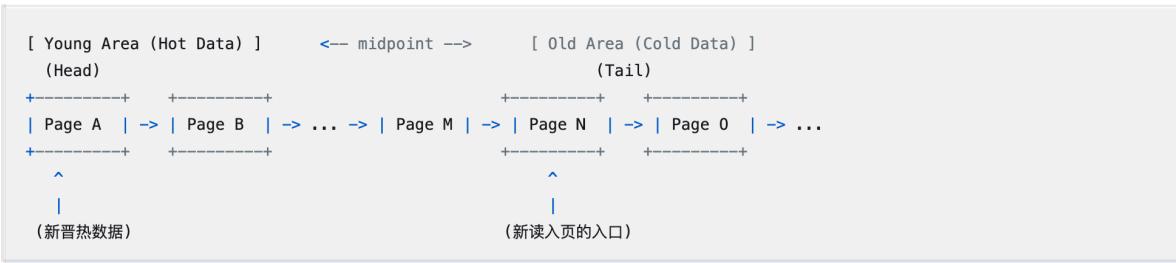
InnoDB不是使用一个简单的 LRU 链表，而是将它逻辑上划分为两个区域：

- **热数据区 (Young Area):** 位于链表的头部。这里存放的是被频繁访问的、真正的热点数据页。
- **冷数据区 (Old Area):** 位于链表的尾部。这里是新读入数据页的“观察区”或“缓冲区”。

这两个区域的比例由参数 `innodb_old_blocks_pct` 控制，默认为 **37**，表示冷数据区约占整个 LRU 链表的 **37%**。

一个指针 midpoint 标志着冷热数据的分界点。

**LRU 链表的逻辑结构示意图：**



### 三、具体的工作逻辑：页的移动与晋升

现在，我们来看数据页在这个特殊的LRU链表中是如何移动的。

#### 1. 新数据页的加载

当需要从磁盘读取一个数据页到Buffer Pool时（无论是普通查询还是预读），这个新页不会直接放到链表头部。

- 规则：新加的数据页会被插入到冷数据区的头部，也就是 midpoint 指向的位置。
- 目的：这就是“中点插入策略”。让新来的页先进入“冷数据观察区”。这样，即便是全表扫描或无效的预读，这些“一次性”的页也只是污染了冷数据区，不会立刻冲击到真正的热数据。它们会相对较快地在冷数据区被淘汰，对热数据区的影响降到了最低。

#### 2. 页从冷数据区移动到热数据区（冷变热）

这是您问题的核心：“什么时候从冷数据变成热数据？”

- 触发条件：一个处于冷数据区的页，在它被加载进Buffer Pool后，被再次访问。
- 时间窗口限制：为了防止全表扫描时，短时间内对同一个页的多次访问（比如在一个大的SQL操作内部）导致其“伪晋升”，InnoDB引入了一个时间限制参数 `innodb_old_blocks_time`，默认为 **1000毫秒（1秒）**。
- 晋升规则（移动依据）：
  1. 一个页位于冷数据区（Old Area）。
  2. 在它被加载进来后，经过了 `innodb_old_blocks_time` 所指定的时间（默认1秒）。
  3. 在这个时间之后，这个页再一次被访问（被查询或修改）。
  4. 当这三个条件同时满足时，这个页就会被移动到热数据区的头部（Head），完成“晋升”，正式成为一个热数据页。
- 目的：这个1秒的“冷却时间”设计得非常巧妙。
  - 对于**全表扫描**，虽然一个页可能在一个大的查询中被多次逻辑访问，但这些访问通常发生在1秒之内。因此，它不满足时间条件，**不会被晋升到热数据区**，最终会在冷数据区被淘汰。
  - 对于**真正的热点数据**，比如一个被不同用户、不同请求反复查询的商品信息页，它被加载进来后，很可能会在1秒之后被再次访问。此时，它就能成功晋升为热数据，从而在Buffer Pool中存活更长时间。

#### 3. 热数据区的内部移动

- 规则：当一个已经处于热数据区的页被访问时，它会被移动到热数据区的头部。
- 优化：为了避免在高并发下频繁地移动热数据区的页（这需要加锁，有性能开销），InnoDB做了一个优化：只有当一个热数据区的页位于**热数据区后 1/4 部分**时，访问它才会触发移动。如果它本来就在热数据区的前 3/4 部分，访问它并不会改变它在链表中的位置。
- 目的：减少不必要的链表操作，降低内部锁竞争，提高高并发下的性能。

## 四、总结

事件	页的初始位置	移动规则	移动目标	设计目的
新页加载 (查询/预读)	(从磁盘加载)	无条件	冷数据区头部 (midpoint)	防止预读和全表扫描污染热数据区。
冷数据区页被访问	冷数据区	第一次加载后 > 1秒，且被再次访问	热数据区头部	筛选出真正的热点数据，防止全表扫描时的“伪热点”。
热数据区页被访问	热数据区	(优化) 只有当该页位于热区后 1/4时	热数据区头部	减少高并发下的锁竞争和链表操作。
淘汰页	冷数据区尾部	当Buffer Pool满时	(从内存中移除)	淘汰最久未被访问的、且未被证明是热点的冷数据。

通过这套精密的“中点插入 + 延迟晋升”的冷热数据分离机制，InnoDB 的 Buffer Pool 成功地抵御了预读失效和全表扫描带来的缓存冲击，极大地提高了缓存的效率和稳定性。

**五：额外话题->那什么时候热数据会变成冷数据？是不是如果有新的数据晋升为热数据，热数据区里的后 1/4 会被挤到冷数据中去？**

您的猜测——“是不是如果有新的数据晋升为热数据，热数据区里的后 1/4 会被挤到冷数据中去？”——是不正确的。

热数据区的页，一旦进入，就不会主动“降级”回到冷数据区。它只会在热数据区内部移动，或者最终被淘汰。

让我们来详细解释一下热数据是如何“变冷”并最终离开 Buffer Pool 的。

### 一、热数据不会“降级”回冷数据区

首先，我们要明确一个核心机制：InnoDB 的 LRU 链表是一个单向流动的系统。页的“晋升”通道是从冷数据区头部移动到热数据区头部，这是一个单行道。不存在一个反向的、从热数据区“降级”到冷数据区的机制。

- 热数据区的页：一旦一个页通过了“1秒冷却期 + 再次访问”的考验，晋升到了热数据区的头部，它就被打上了“热数据”的标签。
- 后续访问：之后对这个页的每一次访问，只会让它在热数据区内部“保鲜”——要么因为访问而重新移动到热数据区头部，要么因为长期不被访问而逐渐向 midpoint 方向漂移。
- 不会跨越分界点：它永远不会因为不被访问而主动跨过 midpoint 回到冷数据区。

### 二、那么，热数据是如何“变冷”并被淘汰的？

热数据的“变冷”是一个被动的、相对的过程。它不是自己降级，而是被其他更热的数据“挤”下去的。

这个过程可以分解为以下几个步骤：

#### 1. 逐渐“冷却”：向 midpoint 漂移

- 一个曾经的热数据页（比如 Page B），在被访问后，被移动到了热数据区的头部。
- 随着时间的推移，大量的新的、更热的页被访问（或者从冷数据区晋升上来），它们不断地被插入到热数据区的头部。
- 这个过程就像一个拥挤的队列，新来的人不断插到队首。我们那个曾经热门的 Page B，因为长时间没有被再次访问，它在队列中的位置就会被动地、一步步

地向后移动，越来越靠近 midpoint。

- 这个“向后移动”的过程，就是它在\*\*相对意义上“变冷”\*\*的过程。

## 2. 到达淘汰的“临界点”：被挤到 midpoint 之后

- 真正的淘汰压力来自冷数据区。当 Buffer Pool 已满，并且有新的数据页需要从磁盘加载时，必须淘汰一个页来腾出空间。
- 淘汰的目标是冷数据区的尾部 (**Tail**)。
- 当一个页被从冷数据区尾部淘汰时，整个 LRU 链表的长度减一。为了维持冷热数据区的比例（由 innodb\_old\_blocks\_pct 决定），midpoint 指针会向链表头部方向移动。
- **关键点：**随着 midpoint 不断地向头部移动，我们那个在热数据区末尾、长期未被访问的 Page B，最终会被 midpoint “越过”。

## 3. 最终的身份转变与淘汰

- 一旦 midpoint 越过了 Page B，Page B 的身份就发生了根本性的转变：它现在逻辑上属于冷数据区的头部了。
- 虽然它曾经是“热数据”，但现在它和那些刚刚从磁盘加载进来的新页一样，都位于冷数据区。
- 接下来，它的命运就和所有冷数据页一样了：如果它在 1 秒的冷却期之后被再次访问，它还有机会重新晋升回热数据区头部。
- 但如果它继续不被访问，它就会随着新页的不断加入，在冷数据区中一路向尾部移动，最终被淘汰出 Buffer Pool。

## 总结

问题	答案
热数据会变成冷数据吗？	不会主动降级。它是在相对意义上“变冷”，即被更热的数据挤到了热数据区的末尾。
热数据如何被淘汰？	1. 因长期不被访问，被动地移动到热数据区的末尾。 2. 由于冷数据区的页被淘汰，导致 midpoint 分界点向头部移动，最终越过了这个页。 3. 该页的身份从“热数据”变为“冷数据”。 4. 在冷数据区中继续不被访问，最终从链表尾部被淘汰。
移动的依据是什么？	* 向头部移动：因为被访问（满足条件时）。 * 向尾部移动：因为不被访问，被其他更热的页“挤”下去。

其实后 1/4 的热数据会变冷，只不过不是主动降级过程，而是随着新的数据页加载，midpoint 会往头部移动，让尾部的热数据变成冷数据，并且如果一直没被访问，会随着时间推移直到被淘汰出 Buffer Pool。

## 多线程并发访问出现并发修改数据怎么办？

当多个事务并发地尝试修改位于 Buffer Pool 中同一个数据页上的数据时，MySQL 的 InnoDB 引擎通过一套精密且环环相扣的机制来处理，主要包括：**Latch**、锁（**Lock**）、**MVCC** 和 **Redo Log**。

简单来说，处理过程是这样的：先用轻量级的 **Latch** 保护内存结构，再用重量级的 **Lock** 保护数据逻辑，同时利用 MVCC 让读写分离，最后通过 **Redo Log** 保证持久性。

下面我们来详细拆解这个过程。

### 第一道防线：Latch（闩锁）—— 保护内存，防止踩踏

- **是什么？**: Latch 是一种轻量级的、短暂的锁，它保护的是内存中的数据结构，而不是数据本身。可以把它理解为多线程编程中的 Mutex（互斥锁）或 RWLock（读写锁）。
- **保护对象**: Buffer Pool 本身、LRU 链表、Free 链表、哈希表等这些内存数据结构。
- **工作场景**:
  1. 当一个事务需要访问 Buffer Pool 中的某个数据页时，它必须先获取这个数据页对应的 block control body（控制块）上的 Latch。
  2. 如果事务只是想读取这个页（不修改），它会尝试获取一个共享的 **S-Latch**。多个事务可以同时持有 S-Latch。
  3. 如果事务想要修改这个页的内容（比如更新一条记录），它必须获取一个排他的 **X-Latch**。
- **作用**: Latch 的作用是保证在极短的时间内，只有一个线程在操作某个内存块或链表。比如，防止两个线程同时把一个数据页从 LRU 链表中摘下或挂上，或者同时修改页的内部指针，从而导致内存结构损坏。
- **生命周期**: Latch 的持有时间非常短，通常只在一次内存操作的几个或几十个 CPU 指令期间。一旦内存操作完成，Latch 会立即释放。它与事务的生命周期无关。

### 第二道防线：Lock（锁）—— 保护数据，实现隔离

在获取了 Latch，安全地访问到数据页之后，并发修改的真正核心——数据逻辑的并发控制——才刚刚开始。这部分由 **Lock** 来负责。

- **是什么？**: Lock 是我们通常谈论的事务锁，如行锁（Record Lock）、间隙锁（Gap Lock）、表锁等。它保护的是数据的逻辑内容，遵循数据库的隔离级别。
- **保护对象**: 表、行、或者行与行之间的间隙。
- **工作场景**:
  1. 事务 A 想要执行 UPDATE users SET age = 30 WHERE id = 101;。
  2. 它首先通过 Latch 安全地访问到包含 id=101 这条记录的数据页。
  3. 接着，它会尝试获取 id=101 这一行的排他锁（X-Lock）。
  4. 如果成功获取了 X-Lock，它就可以在内存中（Buffer Pool 的数据页上）修改这行数据了。
  5. 此时，事务 B 也想执行 UPDATE users SET age = 31 WHERE id = 101;。
  6. 事务 B 同样能通过 Latch 访问到这个数据页，但当它尝试获取 id=101 这一行的 X-Lock 时，会发现这个锁已经被事务 A 持有。
  7. **结果**: 事务 B 会进入等待状态，直到事务 A 提交或回滚，释放了行锁 X-Lock，事务 B 才能获取锁并继续执行。
- **生命周期**: Lock 的持有时间与事务绑定。在默认的 REPEATABLE READ 隔离级别下，一个事务获取的锁会一直持有到事务结束（COMMIT 或 ROLLBACK）。

### 第三道防线：MVCC（多版本并发控制）—— 实现读写并行

当并发修改发生时，不仅有“写-写”冲突，还有“读-写”冲突。MVCC 正是为了解决这个问题，实现“非锁定读”。

- 工作场景：

1. 事务 A 持有 id=101 的行锁，并将其在 Buffer Pool 中的数据从 age=29 改为了 age=30（此时事务 A 还未提交）。
2. 事务 C 此时执行一个普通的 SELECT age FROM users WHERE id = 101;。
3. 如果没有 MVCC，事务 C 会被事务 A 的行锁阻塞，直到事务 A 提交。
4. 有了 MVCC：
  - ◆ 事务 C 是一个“快照读”，它不需要获取行锁。
  - ◆ 它会根据自己的 Read View（在事务开始或首次查询时创建的快照），去读取 id=101 的行。
  - ◆ 它看到这行的最新版本是被一个活跃的（未提交的）事务 A 修改的，因此这个版本对它不可见。
  - ◆ 于是，事务 C 会顺着这行记录的回滚指针，去 Undo Log 中查找上一个版本。
  - ◆ 它找到了那个 age=29 的、由已提交事务所创建的旧版本，这个版本对它是可见的。
  - ◆ 结果：事务 C 无需等待，立即读取到了 age=29 这个一致性的数据。

**MVCC 的作用：**通过创建数据的多版本，让读操作去读旧版本，写操作去操作新版本，从而使得“读”和“写”可以完全并行，互不阻塞，极大地提高了数据库的并发性能。

### 第四道防线：Redo Log —— 保证修改不丢失

在 Buffer Pool 中对数据页的所有修改都是在内存中进行的。如果此时数据库崩溃，这些修改就会丢失。为了保证持久性（Durability），InnoDB 引入了 Redo Log。

- 工作场景：

1. 当事务 A 在 Buffer Pool 中将 age 从 29 修改为 30 时，它不仅仅是修改了数据页。
  2. 它会同时生成一条“重做日志”（Redo Log），内容大致是：“在某个数据页的某个偏移量位置，将值从 29 改为 30”。
  3. 这条 Redo Log 会先被写入 Buffer Pool 中的 Log Buffer。
  4. 在事务\*\*提交（COMMIT）时，Log Buffer 中的 Redo Log 会通过一种称为 WAL（Write-Ahead Logging，预写日志）\*\*的策略，被刷新到磁盘上的 Redo Log 文件中。这个写日志的过程是顺序 I/O，非常快。
- 作用：只要 Redo Log 成功写入磁盘，这次提交就被认为是成功的。即使 Buffer Pool 中被修改的“脏页”还未来得及刷回磁盘数据库就宕机了，在重启时，InnoDB 也可以通过重放 Redo Log，将所有已提交事务的修改恢复到数据页上，保证了数据不丢失。

### 总结

当并发修改 Buffer Pool 中的数据时，InnoDB 的协同工作流程如下：

1. 获取 Latch：线程安全地访问 Buffer Pool 中的数据页，操作完内存结构后立即释放。

2. **获取 Lock**: 获取要修改的数据行的事务锁 (如行锁), 阻止其他事务进行“写”操作。此锁会持有到事务结束。
  3. **利用 MVCC**: 在持有行锁进行修改的同时, 允许其他事务通过 MVCC 读取数据的旧版本, 实现读写不阻塞。
  4. **记录 Redo Log**: 在修改数据页的同时, 将修改操作记录到 Redo Log 中。
  5. **提交事务**: 将 Redo Log 刷入磁盘, 确保修改的持久性, 然后释放 Lock。
- 这套组合拳, 既保证了内存操作的线程安全 (Latch), 又实现了 SQL 层面的事务隔离 (Lock + MVCC), 还确保了数据的持久性 (Redo Log), 是 InnoDB 能够支持高并发事务处理的基石。
- 
- 

## MySQL 日志之 Redo Log

### 解释

由于 mysql 的操作不是直接同磁盘而是内存的 buffer pool 打交道的, 并且当你修改一条数据也是在 buffer pool 里操作的, 然后由异步 IO 进行刷到磁盘。但是在如果在还未刷盘之前数据库发生宕机, 那么这部分数据可能由于还未来得及刷盘导致丢失, 这时候 redo log 就应运而生, 保证数据的一致性和持久化。

redo log (重做日志) 是 InnoDB 存储引擎独有的, 它让 MySQL 拥有了崩溃恢复能力。

比如 MySQL 实例挂了或宕机了, 重启时, InnoDB 存储引擎会使用 redo log 恢复数据, 保证数据的持久性与完整性。

### 为什么需要 redo log?

mysql 为了提升性能不会把每次的修改都实时同步到磁盘, 而是会先存到 Buffer Pool(缓冲池) 里头, 把这个当作缓存来用。然后使用后台线程去做缓冲池和磁盘之间的同步。

那么问题来了, 如果还没来的急同步的时候宕机或断电了怎么办? 这样会导致丢失部分已提交事务的修改信息!

所以引入了 redo log 来记录已成功提交事务的修改信息, 并且会把 redo log 持久化到磁盘, 系统重启之后在读取 redo log 恢复最新数据。

总结:

redo log 是用来恢复数据的, 用于保障已提交事务的持久化特性。

### 作用【保证事务持久性, 即使系统崩溃在恢复后数据不能丢失】

InnoDB 引擎的事务采用了 WAL 技术 (Write-Ahead Logging), 这种技术的思想就是先写日志, 再写磁盘, 只有日志写入成功, 才算事务提交成功, 这里的日志就是 Redo Log。当发生宕机且数据未刷到磁盘的时候, 可以通过 Redo Log 来恢复, 保证 ACID 中的 D, 这就是 Redo Log 的作用。

MySQL 的 redo log 是用来记录数据库中发生的所有更改操作的日志文件, 它的作用主要有以下几点:

1. 数据持久性: MySQL 的 redo log 可以确保在数据库发生异常情况下 (如崩溃、断电等) 可以通过 redo log 中的记录来实现数据的恢复, 保证数据的持久性和一

致性。

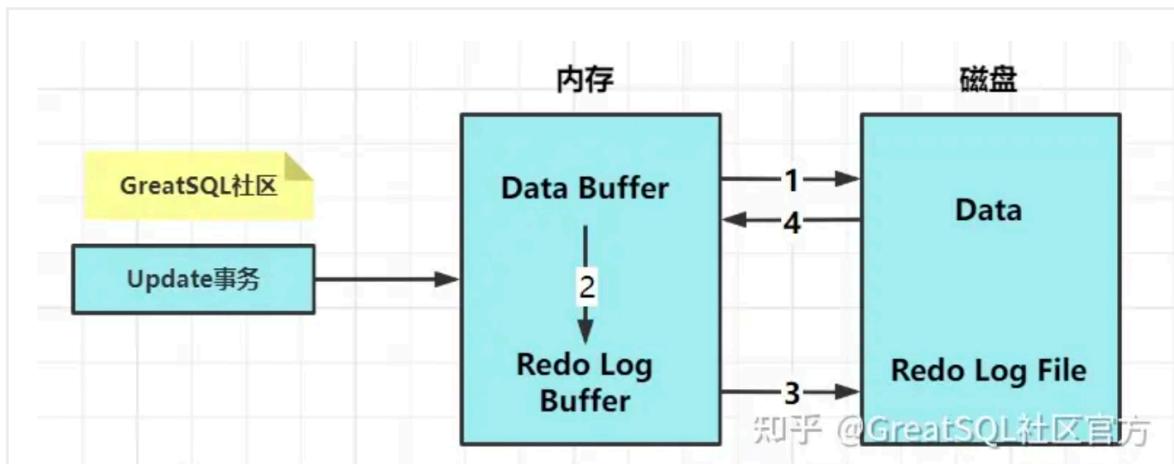
2. 数据恢复：当数据库发生异常情况导致数据丢失或损坏时，可以通过 redo log 进行数据恢复，将数据库恢复到之前的状态。
3. 提高性能：redo log 的存在可以减少数据库的实际操作次数，将修改操作批量写入 redo log 文件中，然后由后台线程进行异步刷盘到磁盘，减少数据库的 I/O 操作，从而提高数据库的性能。

总之，redo log 是 MySQL 中非常重要的一个组成部分，可以确保数据库的数据安全和一致性，并提高数据库的性能。

博客参考：<https://zhuanlan.zhihu.com/p/587553430>

## Redo的整体流程

以一个更新事务为例，Redo Log 流转过程，如下图所示：



流程说明：

- 第1步：先将原始数据从磁盘中读入内存中来，修改数据的内存拷贝
- 第2步：生成一条重做日志并写入 Redo Log Buffer，记录的是数据被修改后的值
- 第3步：当事务 commit 时，将 Redo Log Buffer 中的内容刷新到 Redo Log File，对 Redo Log File 采用追加写的方式
- 第4步：定期将内存中修改的数据刷新到磁盘中【即 redo.file】。

## insert/update/delete SQL 语句下会触发 Redo Log 写入

MySQL 中数据是以页为单位，你查询一条记录，会从硬盘把一页的数据加载出来，加载出来的数据叫数据页，会放入到 Buffer Pool 中。

后续的查询都是先从 Buffer Pool 中找，没有命中再去硬盘加载，减少硬盘 IO 开销，提升性能。

更新表数据的时候，也是如此，发现 Buffer Pool 里存在要更新的数据，就直接在 Buffer Pool 里更新。

然后会把“在某个数据页上做了什么修改”记录到重做日志缓存 (redo log buffer) 里，接着刷盘到 redo log 文件里。

【redo log 存储的是物理数据的变更，如果我们内存的数据已经刷到磁盘 redo.file，那 redo log 里面的数据就无效了。所以 redo log 不会存储着历史所有数据的变更，文件的内容会被覆盖的】

## redo log buffer 刷盘到 redo.file 时机

- 1、事务提交：当事务提交时，log buffer 里的 redo log 会被刷新到磁盘
- 2、log buffer 空间不足时：log buffer 中缓存的 redo log 已经占满了 log buffer 总容量的大约一半左右，就需要把这些日志刷新到磁盘上。
- 3、事务日志缓冲区满：InnoDB 使用一个事务日志缓冲区 (transaction log buffer) 来暂时存储事务的重做日志条目。当缓冲区满时，会触发日志的刷新，将日志写入磁盘。
- 4、Checkpoint (检查点)：InnoDB 定期会执行检查点操作，将内存中的脏数据 (已修改但尚未写入磁盘的数据) 刷新到磁盘，并且会将相应的重做日志一同刷新，以确保数据的一致性。
- 5、后台刷新线程：InnoDB 启动了一个后台线程，负责周期性 (每隔 1 秒) 地将脏页 (已修改但尚未写入磁盘的数据页) 刷新到磁盘，并将相关的重做日志一同刷新。
- 6、正常关闭服务器：MySQL 关闭的时候，redo log 都会刷入到磁盘里去。

## 事务提交时 redo log buffer 刷盘到 redo.file 策略

我们要注意设置正确的刷盘策略 `innodb_flush_log_at_trx_commit`。根据 MySQL 配置的刷盘策略的不同，MySQL 宕机之后可能会存在轻微的数据丢失问题。

`innodb_flush_log_at_trx_commit` 的值有 3 种，也就是共有 3 种刷盘策略：

- 0：设置为 0 的时候，表示每次事务提交时不进行刷盘操作。这种方式性能最高，但是也最不安全，因为如果 MySQL 挂了或宕机了，可能会丢失最近 1 秒内的事务。
- 1：设置为 1 的时候【默认策略】，表示每次事务提交时都将进行刷盘操作。这种方式性能最低，但是也最安全，因为只要事务提交成功，redo log 记录就一定在磁盘里，不会有任何数据丢失。
- 2：设置为 2 的时候，表示每次事务提交时都只把 log buffer 里的 redo log 内容写入 page cache (文件系统缓存)。page cache 是专门用来缓存文件的，这里被缓存的文件就是 redo log 文件。这种方式的性能和安全性都介于前两者中间。

【另外，InnoDB 存储引擎有一个后台线程，每隔 1 秒，就会把 redo log buffer 中的内容写到文件系统缓存 (page cache)，然后调用 fsync 刷盘。也就是说，一个没有提交事务的 redo log 记录，也可能会刷盘。】

**如果 mysql 事务回滚了，但是数据已经写入 redo log 磁盘里，此时发生了宕机重启去用 redo log 用于恢复数据，会不会把之前回滚之前的数据也加载进去？**

在 MySQL 的 InnoDB 存储引擎中，当事务回滚时，已经写入到 redo log 中的数据并不会被撤销或删除。redo log 的主要目的是在系统崩溃时提供恢复机制，确保在数据库重启后能够重新应用那些尚未应用到数据文件上的事务操作。

当事务回滚时，InnoDB 会执行相应的操作来撤销对数据的修改，但这不会影响到 redo log 中的记录。换句话说，redo log 中的记录是持久的，并且不会因为事务的回滚而被更改或删除。如果数据库在系统崩溃后重启，InnoDB 会使用 redo log 来恢复数据，包括那些已经被回滚的事务的操作。

然而，InnoDB 不仅仅依赖 redo log 来恢复数据。它还有一个叫做 undo log 的机制，用于存储旧版本的数据，以便在事务回滚或一致性读时能够访问这些版本。当事务回滚时，InnoDB 会使用 undo log 来撤销对数据的修改。因此，即使 redo log 中包含

了已经被回滚的事务的操作，这些操作也不会在数据库重启后的恢复过程中被重新应用，因为 InnoDB 会使用 undo log 来撤销这些操作。

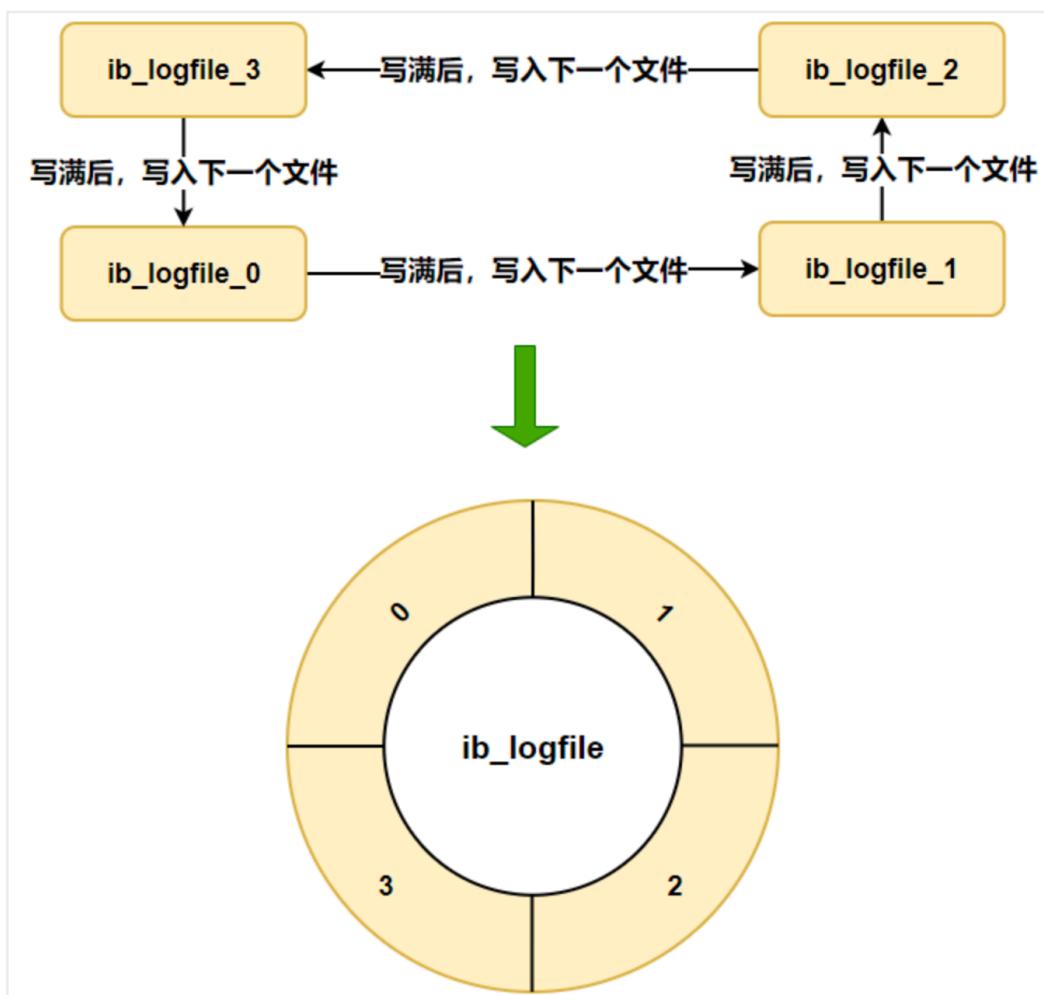
总结来说，即使事务已经回滚，但 redo log 中仍然保留了这些事务的操作记录。在数据库重启后的恢复过程中，InnoDB 会使用 redo log 和 undo log 一起来确保数据的完整性和一致性。redo log 用于重新应用那些尚未应用到数据文件上的事务操作，而 undo log 用于撤销那些已经被回滚的事务的操作。

## 日志文件组

硬盘上存储的 redo log 日志文件不只一个，而是以一个日志文件组的形式出现的，每个的 redo 日志文件大小都是一样的。

比如可以配置为一组 4 个文件，每个文件的大小是 1GB，整个 redo log 日志文件组可以记录 4G 的内容。

它采用的是环形数组形式，从头开始写，写到末尾又回到头循环写，如下图所示。



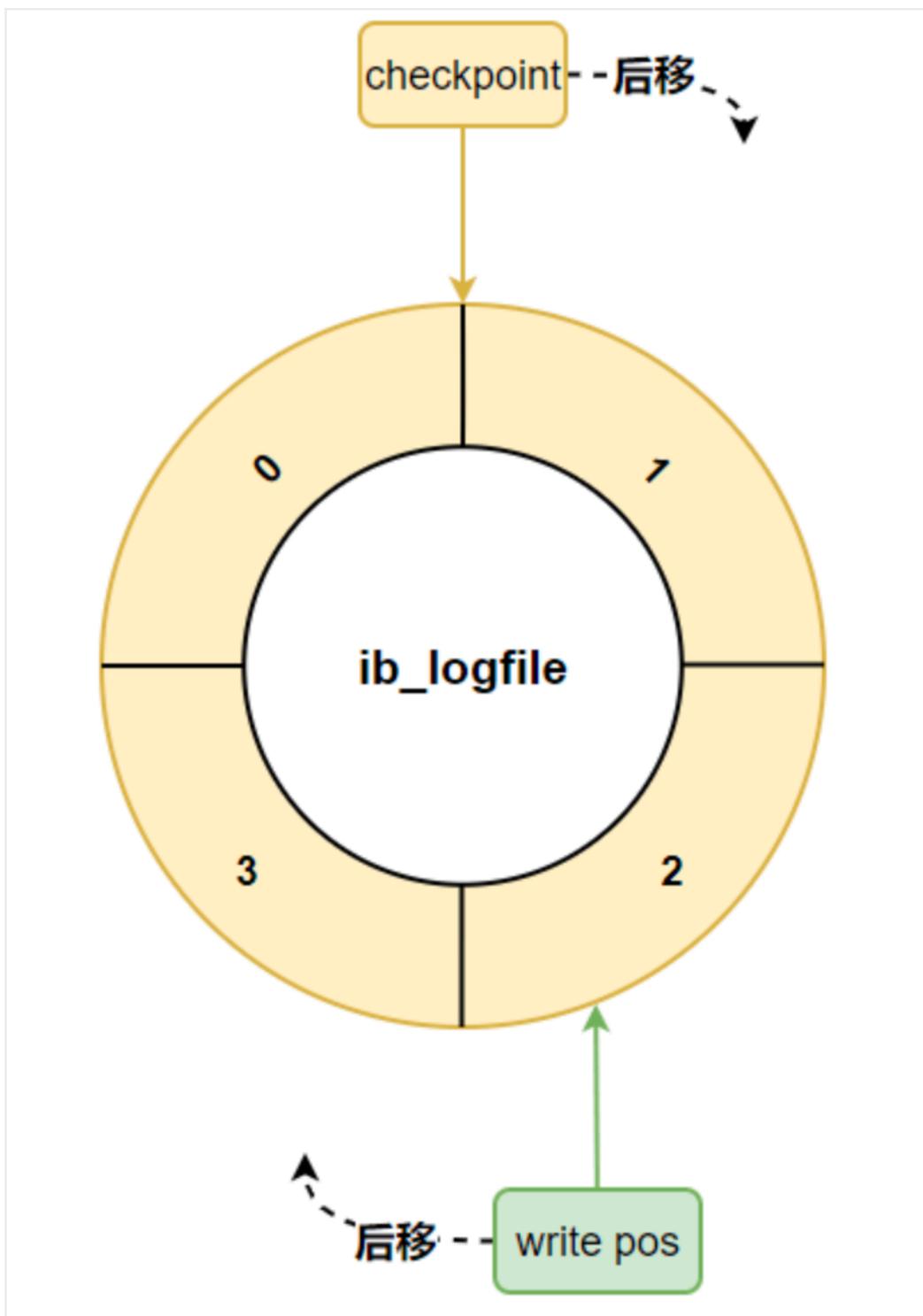
在个日志文件组中还有两个重要的属性，分别是 write pos、checkpoint

- **write pos** 是当前记录的位置，一边写一边后移
- **checkpoint** 是当前要擦除的位置，也是往后推移

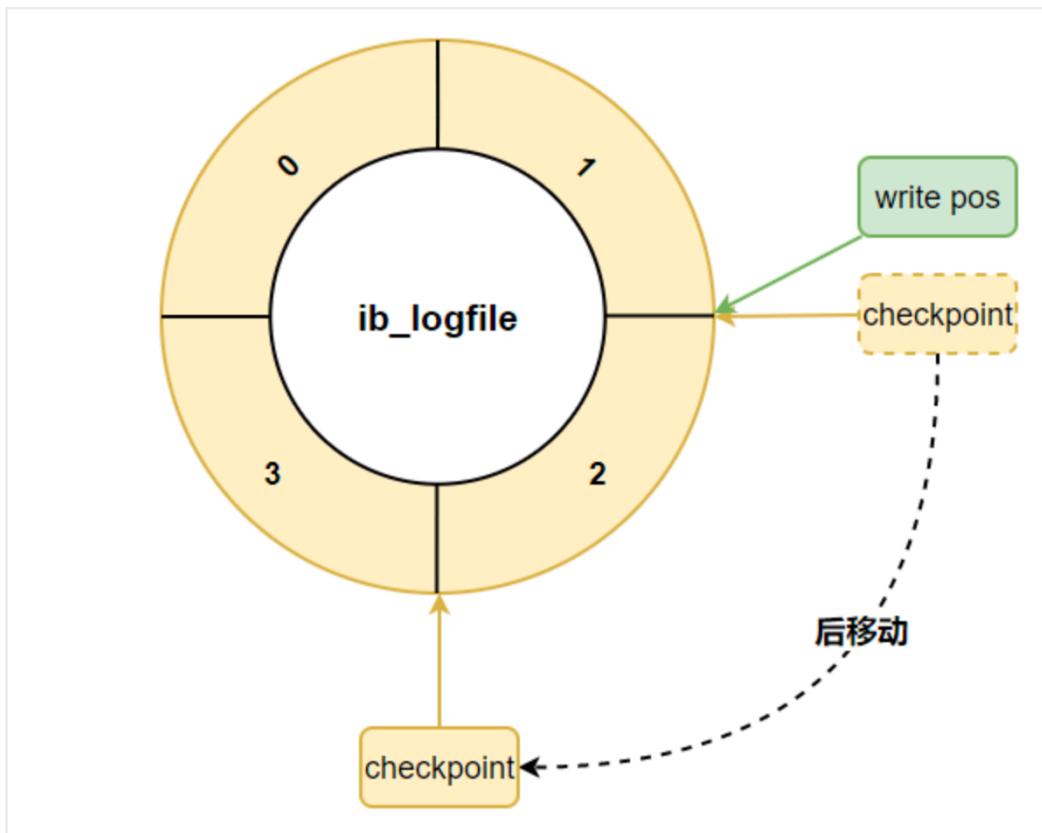
每次刷盘 redo log 记录到日志文件组中，write pos 位置就会后移更新。

每次 MySQL 加载日志文件组恢复数据时，会清空加载过的 redo log 记录，并把 checkpoint 后移更新。

write pos 和 checkpoint 之间的还空着的部分可以用来写入新的 redo log 记录。



如果 `write pos` 追上 `checkpoint`，表示日志文件组满了，这时候不能再写入新的 redo log 记录，MySQL 得停下来，清空一些记录，把 `checkpoint` 推进一下。



## 每次修改后的数据页为什么不直接刷盘？

答案很简单：如果每次修改后都立刻直接刷盘，数据库的性能将会差到完全无法使用的地步。

为了彻底解决这个问题，InnoDB 引入了 Buffer Pool 和 Redo Log 这套复杂的机制。这套机制的本质，就是为了避免每次修改都直接刷盘，从而换取极高的性能。

下面我们来详细分析为什么不能直接刷盘，以及 InnoDB 的解决方案是如何巧妙地规避这个问题的。

### 一、为什么不能直接刷盘？——“随机 I/O”是性能的头号杀手

假设我们没有 Buffer Pool，没有 Redo Log，每次 UPDATE 一条记录，都直接去修改磁盘上的数据文件。这会带来两个致命的性能问题：

#### 1. 性能极差：随机 I/O 的巨大成本

- 数据在磁盘上是分散的：数据库中的数据行，是存储在不同的数据页（Page）里的。而这些数据页在物理磁盘上通常是分散的、不连续的。
- 一次 UPDATE 可能涉及多次随机 I/O：
  - 一个简单的 UPDATE users SET name = 'B' WHERE id = 101;，可能只修改了一行数据。
  - 但是，一个复杂的事务，比如一次银行转账，可能需要修改用户的账户表、交易流水表、积分表等等。这会涉及到多个不同表空间、多个完全不相邻的数据页。
- 随机 I/O 的本质：对于机械硬盘来说，每次随机 I/O 都意味着磁头需要进行\*\*寻道 (Seek) 和旋转延迟 (Rotational Latency) 才能找到正确的物理位置。这个过程非常耗时，通常在毫秒 (ms) \*\*级别。
- 灾难性的后果：如果一个事务修改了 10 个不同的数据页，就意味着需要进行 10 次缓慢的随机 I/O。数据库的 TPS (每秒事务数) 可能会降到几十甚至个位数，

这种性能在任何现代应用中都是完全不可接受的。

## 2. 效率低下：I/O 单位与修改单位不匹配

- 操作系统 I/O 的最小单位是“页”：当你需要修改磁盘上的一个字节时，操作系统并不能只写那一个字节。它必须将整个数据页（在 InnoDB 中通常是 **16KB**）读入内存，在内存中修改，然后再将整个 **16KB** 的页写回磁盘。
- 巨大的浪费：假设你只是 UPDATE 一行数据，可能只改变了其中的几十个字节。但为了这几个字节的修改，你却不得不进行一次读取 16KB、一次写入 16KB 的完整 I/O 操作。这个开销是极其巨大的，绝大部分 I/O 带宽都被浪费了。

## 二、InnoDB 的解决方案：用“快的”代替“慢的”

为了解决上述问题，InnoDB 设计了一套精妙的机制，其核心思想就是：**用成本极低的内存操作和顺序 I/O，来替代成本极高的随机 I/O。**

### 1. 引入 Buffer Pool：化磁盘为内存

- 所有操作在内存完成：InnoDB 将所有的数据读写操作，都放在内存中的 Buffer Pool 里进行。内存的读写速度是纳秒级别的，比磁盘快几个数量级。
- 合并写入：一个数据页在 Buffer Pool 中可能被连续修改多次。但无论修改多少次，最终都只需要在某个合适的时机，将这个“脏页”一次性地刷回磁盘即可。这极大地减少了 I/O 次数。

### 2. 引入 Redo Log：化随机 I/O 为顺序 I/O (这是关键！)

现在，问题变成了：数据在 Buffer Pool 里改了，但还没刷回磁盘，如果这时宕机了怎么办？Redo Log 就是为了解决这个问题。

- WAL (Write-Ahead Logging) -> 预写式日志 (日志先行)：在修改 Buffer Pool 中数据页的同时，InnoDB 会生成一条对应的 Redo Log。这条日志记录的是对数据页的物理修改（“在哪个页的哪个位置，把什么改成了什么”）。
- 顺序 I/O 的魔力：
  - Redo Log 的写入是顺序追加的，就像在笔记本上从上往下写字。
  - 顺序 I/O 几乎没有寻道和旋转的开销，速度非常快，接近内存的速度。
- 核心转换：InnoDB 成功地将“对数据文件的、多次的、缓慢的、随机的写操作”，转换成了“对日志文件的、一次的、快速的、顺序的写操作”。
- 保证持久性：只要 Redo Log 成功写入磁盘（在事务提交时），这次修改就被认为是持久化的了。即使数据库宕机，也可以通过重放 Redo Log 来恢复数据。至于那个缓慢的、将脏页从 Buffer Pool 刷回磁盘的操作，可以留给后台线程慢慢去做，完全不影响前台的事务响应。

## 总结

操作方式	每次修改直接刷盘（假设）	InnoDB 的实际方式
修改对象	磁盘上的数据文件	内存中的 Buffer Pool
I/O 类型	随机 I/O	顺序 I/O (写 Redo Log)
性能	极差 (毫秒级响应)	极高 (微秒 / 纳秒级响应)
I/O 效率	低下 (为修改几十字节，读写 16KB)	高 (日志紧凑，批量写入)
宕机恢复	不需要恢复 (因为已同步)	通过 Redo Log 进行恢复

复杂度	简单，但无法实用	复杂，但保证了高性能和高可靠性
-----	----------	-----------------

### 最终结论：

MySQL之所以不选择“每次修改后的数据页直接刷盘”这种简单粗暴的方式，完全出于对性能的极致追求。直接刷盘带来的随机I/O开销是现代数据库完全无法承受的。InnoDB通过Buffer Pool + Redo Log (WAL)这套组合拳，巧妙地将高成本的随机I/O问题，转换成了低成本的顺序I/O问题，从而在保证数据持久性和一致性的前提下，实现了极高的并发处理能力。

## MySQL Redo Log 的触发与工作流程

MySQL InnoDB存储引擎的Redo Log (重做日志)是其实现事务ACID特性，特别是\*\*持久性(Durability)和崩溃恢复(Crash Recovery)\*\*能力的基石。其触发和工作流程与事务的生命周期紧密相连。

### 第一阶段：日志的产生(在内存中)

- 触发时机：当事务执行任何数据修改操作(**INSERT, UPDATE, DELETE**)时，Redo Log即被触发。
- 动作：InnoDB首先在内存中的**Buffer Pool**里修改对应的数据页。与此同时，它会生成一条或多条对应的Redo Log记录。这些记录是**物理日志**，精确地描述了“在哪个数据页的哪个偏移量位置，做了什么物理修改”。
- 存放位置：这些新生成的Redo Log记录，会被立即写入内存中的另一块区域——**Redo Log Buffer**。这个过程完全在内存中进行，速度极快。

### 第二阶段：日志的持久化(从内存到磁盘)

Redo Log Buffer中的日志数据，需要被持久化到磁盘上的redo log文件中，这个过程称为“刷盘”(Flush)。刷盘的触发时机是保证数据安全的关键，主要有以下几种：

#### 1. 事务提交时(最关键的触发点)：

- 当一个事务执行COMMIT操作时，为了保证其持久性，InnoDB必须确保该事务所产生的所有Redo Log都已写入磁盘。
- 这个行为由参数innodb\_flush\_log\_at\_trx\_commit精确控制，其不同设置值代表了不同的安全与性能权衡：
  - ◆ = 1(默认，最高安全)：每次事务提交时，都必须将**Redo Log Buffer**中的日志同步刷入磁盘文件，并确保完成。
  - ◆ = 2(次高安全)：每次事务提交时，只将日志写入操作系统的**文件系统缓存(Page Cache)**，然后由操作系统决定何时刷盘。性能较高，但操作系统宕机可能丢失数据。
  - ◆ = 0(性能最高)：事务提交时不主动刷盘，而是依赖InnoDB的后台主线程每秒进行一次刷新。MySQL宕机可能丢失最后一秒的事务。

#### 2. Redo Log Buffer空间不足时：

- Redo Log Buffer的大小是有限的(由innodb\_log\_buffer\_size定义)。当Buffer中写入的日志量即将达到其容量的一半时，会触发后台线程主动将日志刷盘。

#### 3. 后台线程定时刷新：

- InnoDB有一个后台主线程，会以大约每秒一次的频率，将Redo Log Buffer中的日志刷入磁盘文件。这是即使没有事务提交，日志也能被持久化的一种保障机制。

#### 4. 数据库正常关闭时：

- 在数据库关闭服务前，会确保将所有Redo Log Buffer中的日志以及

Buffer Pool 中的脏页全部刷新到磁盘。

### 第三阶段：日志的使用（崩溃恢复）

- 触发时机：当 MySQL 实例异常宕机后重启。
  - 动作：InnoDB 会从上一个检查点（Checkpoint）开始，扫描并重放（Replay）redo log 文件中的所有日志记录，将这些物理修改重新应用到数据页上。这个过程会恢复所有已提交事务的修改，以及未提交事务在宕机前的修改，从而将数据库恢复到宕机瞬间的物理一致性状态。之后，再通过 Undo Log 来回滚那些未完成的事务，最终达到数据的逻辑一致性。
- 

## MySQL 日志之 Binlog

### 简介

二进制日志（binary log）以事件形式记录了对 MySQL 数据库执行更改的所有操作。binlog 是记录所有数据库表结构变更（例如 DDL：CREATE、ALTER TABLE...）以及表数据修改（DML：INSERT、UPDATE、DELETE...）的二进制日志。不会记录 SELECT 和 SHOW 这类操作，因为这类操作对数据本身并没有修改，但可以通过查询通用日志来查看 MySQL 执行过的所有语句。

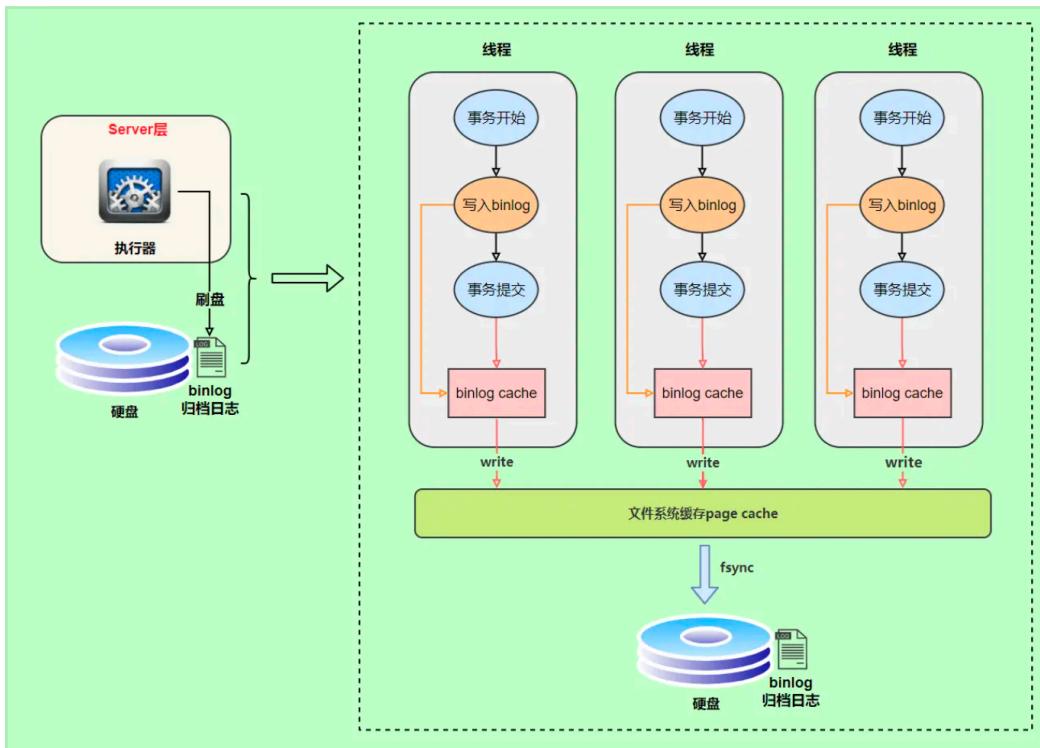
需要注意的一点是，即便 update 操作没有造成数据变化，也是会记入 binlog。

### 主要作用

- 数据恢复：MySQL 可以通过 bin log 恢复某一时刻的误操作的数据，是 DBA 常打交道的日志。
- 数据复制：MySQL 的数据备份、集群高可用、读写分离、主从数据同步都是基于 bin log 的重放实现的。

### bin log 写入机制

事务执行过程中，会先把日志写到 binlog cache 中去，事务提交时，才把 binlog cache 写到 binlog 文件中去（刷盘）。**binlog cache** 是为了保证一个事务的所有操作能够一次性写入 bin log 不被拆开而设置的缓存，binlog cache 大小受 binlog\_cache\_size 参数控制。



上图 binlog cache 写到 bin log 日志文件的过程包含了 write 和 fsync 两步操作：

- write 是把日志写到文件系统缓存中，这一步是系统为了提高文件 IO 效率；
- fsync 是把数据持久化到日志文件中去。

### write 和 fsync 策略设置

- sync\_binlog=0：表示每次提交事务都只 write，由操作系统自行判断什么时候执行 fsync。  
【优点：性能提升；缺点：但是如果机器宕机，page cache 里的 bin log 会丢失】
- sync\_binlog=N：表示每次提交事务都只 write，积攒 N 个事务后才执行 fsync。  
【优点：机器宕机只丢失 N 个事务的 bin log；缺点：如果 N 设置的很小可能会出现 IO 瓶颈，需要适当调整 N 的大小。】
- sync\_binlog=1：表示每次提交事务都会执行 fsync，就如同 redo log 日志刷盘流程一样。  
【优点：能保证数据不丢失；缺点：刷盘频繁，效率低】

### MySQL 的 binlog 和 redo log 区别：

1. 作用不同：binlog 用于恢复数据或进行主从数据同步复制；redo log 作为异常宕机或者介质故障后的数据恢复使用
2. 记录内容不同：binlog 是逻辑日志，记录所有数据的改变信息；redo log 是物理日志，记录所有 InnoDB 表数据的变化
3. 记录内容时间不同：binlog 记录 commit 完毕之后的 DML 和 DDL SQL 语句；redo log 记录事务发起之后的 DML 和 DDL SQL 语句。
4. 恢复方式不同：binlog 可以通过 mysqlbinlog 命令将 binlog 文件解析为 SQL 语句，然后执行来恢复数据；redo log 通过循环写入的方式来实现数据的恢复。
5. 文件使用方式不同：binlog 不是循环使用，在写满或者实例重启之后，会生产新的 binlog 文件【需要定期处理】；redo log 是循环使用，最后一个文件写满之

后，会重新写第一个文件

## 如何保证 binlog 和 redo log 日志都写入成功？【两阶段提交】

redo log (重做日志) 让 InnoDB 存储引擎拥有了崩溃恢复能力。

binlog (归档日志) 保证了 MySQL 集群架构的数据一致性。

虽然它们都属于持久化的保证，但是侧重点不同。

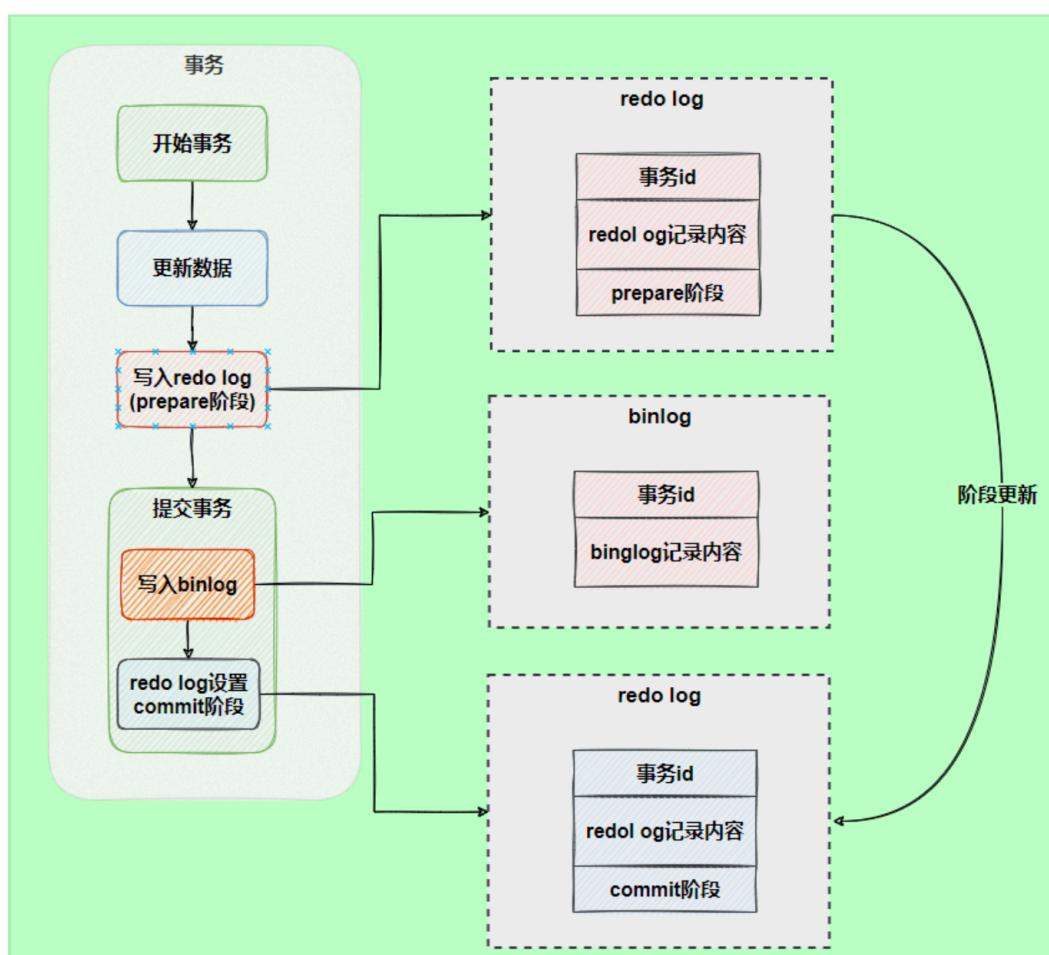
在执行更新语句过程，会记录 redo log 与 binlog 两块日志，以基本的事务为单位，redo log 在事务执行过程中可以不断写入，而 binlog 只有在提交事务时才写入，所以 redo log 与 binlog 的写入时机不一样。

如果 binlog 和 redo log 在写入过程时，在事务提交后写入 binlog 失败了会导致什么问题？

可能会导致主从数据不一致，redo log 可以保证宕机后主库数据不会丢失数据，但是由于 binlog 写入失败，那么从库就无法根据 binlog 日志同步主库最新的数据。

为了解决两份日志之间的逻辑一致问题，InnoDB 存储引擎使用两阶段提交方案。

两阶段提交：将 redo log 的写入操作拆成了两个步骤 prepare 和 commit 进行，在事务执行期间，写入的 redo log 标记为 prepare 阶段，待事务提交且 bin log 写入成功时，才将 redo log 标记为 commit 阶段。



按照这样两阶段提交的解决方案，看看如下的问题：

- **写入 bin log 时发生异常：** 使用两阶段提交后，写入 bin log 时发生异常也不会有影响，因为 MySQL 在使用 redo log 恢复时，发现 redo log 还处于 prepare 阶段，而且此时没有 bin log，认为该事务操作尚未完成提交，会回滚此操作。

- **redo log 在 commit 阶段发生异常：**虽然 MySQL 重启后发现 redo log 是处于 prepare 阶段，但是能通过事务 id 找到了对应的 bin log 记录，所以 MySQL 认为此事务执行是完整的，就会提交事务恢复数据。
- 
- 

## MySQL 日志之 undo log

### 简介

我们知道如果想要保证事务的原子性，就需要在异常发生时，对已经执行的操作进行回滚，在 MySQL 中，恢复机制是通过 **回滚日志（undo log）** 实现的，所有事务进行的修改都会先记录到这个回滚日志中，然后再执行相关操作。如果执行过程中遇到异常的话，我们直接利用回滚日志中的信息将数据回滚到修改之前的样子即可！并且，回滚日志会先于数据持久化到磁盘上。这样就保证了即使遇到数据库突然宕机等情况，当用户再次启动数据库的时候，数据库还能够通过查询回滚日志来回滚将之前未完成的事务。

另外，MVCC 的实现依赖于：**隐藏字段、Read View、undo log**。在内部实现中，InnoDB 通过数据行的 DB\_TRX\_ID 和 Read View 来判断数据的可见性，如不可见，则通过数据行的 DB\_ROLL\_PTR 找到 undo log 中的历史版本。每个事务读到的数据版本可能是不一样的，在同一个事务中，用户只能看到该事务创建 Read View 之前已经提交的修改和该事务本身做的修改。

**Undo Log 是事务原子性的保证。**在事务中更新数据的前置操作其实是要先写入一个 Undo Log，当需要回滚的时候就可以去 undo log 获取针对之前的操作进行反向操作，达到数据恢复的目的。

### Undo Log 如何保证事务回滚

Insert 操作时，InnoDB 存储引擎会完成一个 Delete 操作，用于之后回滚；

Delete 操作时，InnoDB 存储引擎会完成一个 Insert 操作，用于之后回滚；

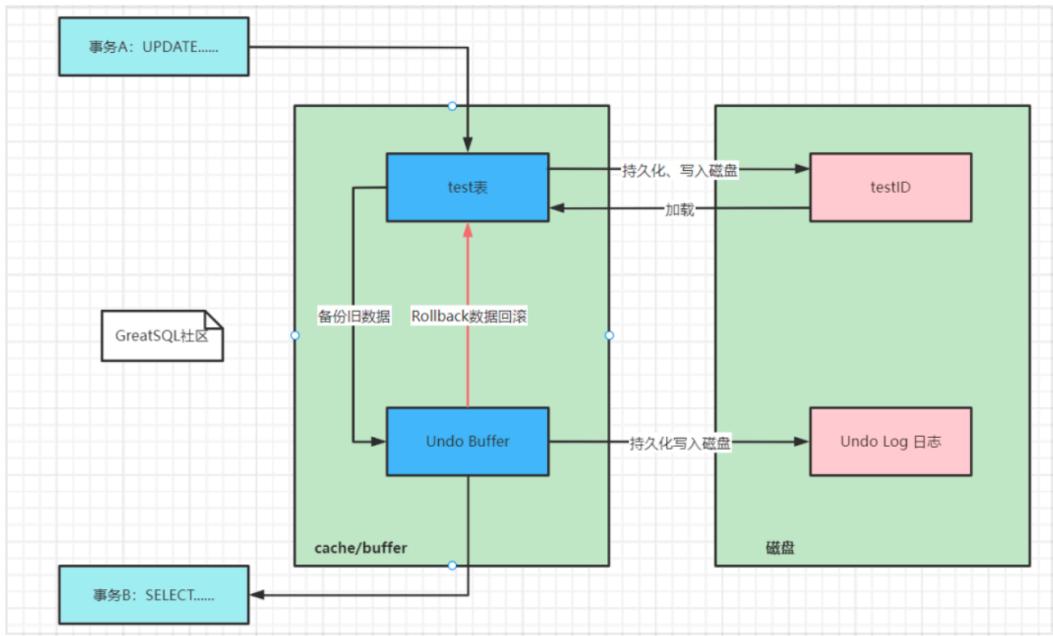
Update 操作时，InnoDB 存储引擎会完成一个相反的 update 操作，将修改前的行放回去，用于之后回滚；

### 作用于？

- **提供数据回滚-原子性：**当事务回滚时或者数据库崩溃时，可以利用 Undo Log 来进行数据回滚。
- **多版本并发控制（MVCC）-隔离性：**即在 InnoDB 存储引擎中 MVCC 的实现是通过 Undo Log 来完成。当用户读取一行记录时，若该记录已经被其他事务占用，当前事务可以通过 Undo Log 读取之前的行版本信息，以此实现非锁定读取。

### Undo Log 的工作原理

在更新数据之前，MySQL 会提前生成 Undo Log 日志，当事务提交的时候，并不会立即删除 Undo Log，因为后面可能需要进行回滚操作，要执行回滚（ROLLBACK）操作时，从缓存中读取数据。Undo Log 日志的删除是通过后台 purge 线程进行回收处理的。



- 1、事务 A 执行 UPDATE 操作，此时事务还没提交，会将数据进行备份到对应的 Undo Buffer，然后由 Undo Buffer 持久化到磁盘中的 Undo Log 文件中，此时 Undo Log 保存了未提交之前的操作日志，接着将操作的数据，也就是 test 表的数据持久保存到 InnoDB 的数据文件 IBD。
- 2、此时事务 B 进行查询操作，直接从 Undo Buffer 缓存中进行读取，这时事务 A 还没提交事务，如果要回滚 (ROLLBACK) 事务，是不读磁盘的，先直接从 Undo Buffer 缓存读取【即事务 A 未提交的话，事务 B 进来会读 undo buffer 而非磁盘，保证事务 B 不会读到未提交事务的数据】。

参考博客：[https://mp.weixin.qq.com/s?\\_\\_biz=MzkzMTIzMdgwMg==&mid=2247496981&idx=1&sn=ec496da6e52e19ee505483a15fb54f6b&chksm=c26c9028f51b193e8d14a5e17f32d696a2cda0eb61f2fa7a3f2b3f179c9bfbd1cba09d0d83fa&scene=21#wechat\\_redirect](https://mp.weixin.qq.com/s?__biz=MzkzMTIzMdgwMg==&mid=2247496981&idx=1&sn=ec496da6e52e19ee505483a15fb54f6b&chksm=c26c9028f51b193e8d14a5e17f32d696a2cda0eb61f2fa7a3f2b3f179c9bfbd1cba09d0d83fa&scene=21#wechat_redirect)

## 三大日志区别

- Redo log：由于 mysql 所有操作不是直接同磁盘而是和 buffer pool 打交道，buffer pool 会异步的每隔一段时间将 mysql 的操作刷到磁盘。但是万一服务器宕机，就可能导致操作过的数据不能从 buffer pool 异步刷到磁盘保证磁盘数据正确性，那么就需要 redo log 保证宕机恢复后能将之前的操作同步回磁盘。并且 redo log 在事务 commit 之前产生，多少次的操作就会产生多少条 redo log。
- Binlog：事务提交后才会写入到磁盘，常用于数据恢复和主从同步
- Undo log：发生异常时用于数据回滚，或者用于 MVCC。

## MySQL 读写分离和分库分表

# 读写分离

## 概念

在一主多从的部署下，主库负责写，从库负责读。

## 主从复制原理【binlog】

- 1、主库将数据库中数据的变化写入到 binlog
- 2、从库连接主库
- 3、从库会创建一个 I/O 线程向主库请求更新的 binlog
- 4、主库会创建一个 binlog dump 线程来发送 binlog，从库中的 I/O 线程负责接收
- 5、从库的 I/O 线程将接收的 binlog 写入到 relay log 中。
- 6、从库的 SQL 线程读取 relay log 同步数据本地（也就是再执行一遍 SQL）。

## 如何避免主从延迟

- 1、强制将读请求路由到主库处理【针对特定业务且不能接受主从延迟状况发生的数据】

比如 Sharding-JDBC 就是采用的这种方案。通过使用 Sharding-JDBC 的 HintManager 分片键值管理器，我们可以强制使用主库。

```
1 HintManager hintManager = HintManager.getInstance();
2 hintManager.setMasterRouteOnly();
3 // 继续JDBC操作
```

java

对于这种方案，你可以将那些必须获取最新数据的读请求都交给主库处理。

- 2、延迟读取数据，在主同步完从后再读取数据，例如读取数据在 0.5s 之后，不过这个看业务场景需要的 RT 是否要求严格等。

## 什么情况下会出现主从延迟？如何尽量避免？

- 1、从库机器性能比主库差，导致从库接收到 binlog 并写入 relay log 以及执行 SQL 语句速度慢。
- 2、从库处理的读请求过多，但是占用了大量 CPU、内存、网络等资源，进而影响 binlog 消费速度。可以通过缓存来减少从库的请求。
- 3、避免大事务，从库如果执行过长的事务也会导致 binlog 消费速度。
- 4、从库太多，导致主库需要同步压力变大，可能致使主库创建的 binlog dump 线程发送 binlog 的执行效率降低无法及时发送，影响从库接收 binlog 消息的时间。
- 5、网络延迟
- 6、MySQL 复制模式，MySQL 默认的复制是异步的，必然会存在延迟问题。全同步复制不存在延迟问题，但性能太差了。

## mysql怎么做主从复制 mysql主从复制三种模式

### 一、MySQL主从复制的三种同步模式

#### 1.异步复制（Asynchronous replication）

MySQL默认的复制即是异步的，主库在执行完客户端提交的事务后会立即将结果返给客户端，并不关心从库是否已经接收并处理，这样就会有一个问题：主如果crash掉了，此时主上已经提交的事务可能并没有传到从上，如果此时，强行将从提升为主，可能导致新主上的数据不完整。

#### 2.全同步复制（Fully synchronous replication）

因为需要等待所有从库执行完该事务才能返回，所以全同步复制的性能必然会受到严重的影响。

#### 3.半同步复制（Semisynchronous replication）

半同步复制提高了数据的安全性，同时它也造成了一定程度的延迟，这个延迟最少是一个TCP/IP往返的时间。所以，半同步复制最好在低延时的网络中使用。

## 分库分表

### 概念

**【分库】：**就是将数据库中的数据分散到不同的数据库上，可以垂直分库，也可以水平分库。

- 垂直分库：例如用户库、订单库、商品库。
- 水平分库：例如订单表太大，那么进行拆分表，将拆分后的多张订单表放在不同库上。

**【分表】：**分表也可以分为水平和垂直，水平分表就是将1000W数据分成10张100W的表，垂直分表就是把100个字段的表分成各50个字段的两张表。

### 什么情况下需要分库分表？

- 单表的数据达到千万级别以上，数据库读写速度比较缓慢。
- 数据库中的数据占用的空间越来越大，备份时间越来越长。
- 应用的并发量太大。

### 分库分表带来的问题

- Join操作：数据分布在不同的库上导致无法join操作，需要手动封装数据进而内存里做代码逻辑处理
- 事务问题：分库分表的话就无法用事务保证数据的ACID特性，需要引入分布式事务
- 跨库聚合查询：不像单张表可以直接用SQL语法，尤其在group by、order by等语法上需要更复杂的业务封装操作。

### 分库分表在电商的实际场景应用

在实际的电商场景中，MySQL分库的设计通常是为了解决数据量过大和并发访问过高的问题。以订单系统为例，假设我们有一个电商平台的订单表，随着业务的发展，订单数据量不断增长，导致查询和更新操作变得缓慢，同时并发访问量也在不

断增加，单库已经无法满足需求。

这时，我们可以考虑进行分库分表的设计。首先，根据业务特点，我们可以选择按照订单 ID 进行分表，将订单数据均匀分散到多个表中，每个表存储一部分订单数据。然后，为了支持高并发访问，我们可以将这些表分散到多个数据库实例中，每个数据库实例承载一部分并发请求。

具体设计如下：

1. 分表策略：我们可以采用哈希分表的方式，将订单 ID 进行哈希计算，然后根据哈希值对表数量进行取模，得到的结果决定将数据插入到哪个表中。这样，订单数据就被均匀分散到了多个表中。
  2. 分库策略：为了支持高并发访问，我们可以将分表后的表分散到多个数据库实例中。每个数据库实例可以部署在不同的服务器上，通过负载均衡技术将请求分发到各个实例上。这样，并发请求就被分散到了多个实例中，减轻了单库的压力。需要注意的是，分库分表虽然可以解决数据量和并发问题，但也带来了一些挑战。例如，跨库表的联合查询、事务的一致性保证、数据迁移和维护等。因此，在设计分库分表方案时，需要综合考虑业务需求、系统架构、开发成本等因素，做出合理的决策。
- 
- 

## 分布式事务

### 分布式事务之 2PC (两阶段提交)

场景：下单后扣减库存，保持这两个服务动作的事务性

在两阶段提交中，主要涉及到两个角色，分别是协调者和参与者。

#### 第一阶段

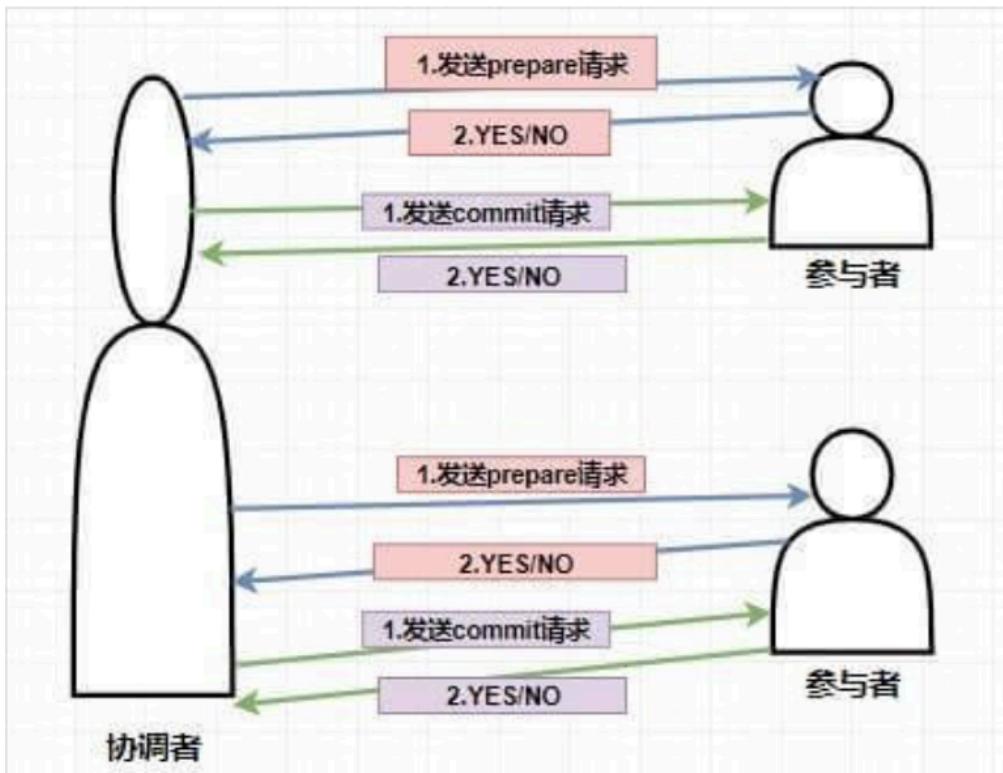
当要执行一个分布式事务的时候，事务发起者首先向协调者发起事务请求，然后协调者会给所有参与者发送 prepare 请求（其中包括事务内容）告诉参与者你们需要执行事务了，如果能执行我发的事务内容那么就先执行但不提交，执行后请给我回复。然后参与者收到 prepare 消息后，他们会开始执行事务（但不提交），并将 Undo 和 Redo 信息记入事务日志中，之后参与者就向协调者反馈是否准备好了。

#### 第二阶段

第二阶段主要是协调者根据参与者反馈的情况来决定接下来是否可以进行事务的提交操作，即提交事务或者回滚事务。

比如这个时候 **所有的参与者** 都返回了准备好了的消息，这个时候就进行事务的提交，协调者此时会给所有的参与者发送 **Commit 请求**，当参与者收到 **Commit 请求** 的时候会执行前面执行的事务的 **提交操作**，提交完毕之后将给协调者发送提交成功的响应。

而如果在第一阶段并不是所有参与者都返回了准备好了的消息，那么此时协调者将会给所有参与者发送 **回滚事务的 rollback 请求**，参与者收到之后将会 **回滚它在第一阶段所做的事务处理**，然后再将处理情况返回给协调者，最终协调者收到响应后便给事务发起者返回处理失败的结果。



### 两阶段提交的缺点

- 1、单点故障问题，如果协调者挂了那么整个系统都处于不可用的状态了。
- 2、阻塞问题，即当协调者发送 prepare 请求，参与者收到之后如果能处理那么它将会进行事务的处理但并不提交，这个时候会一直占用着资源不释放，如果此时协调者挂了，那么这些资源都不会再释放了，这会极大影响性能。
- 3、数据不一致问题，比如当第二阶段，协调者只发送了一部分的 commit 请求就挂了，那么也就意味着，收到消息的参与者会进行事务的提交，而后面没收到的则不会进行事务提交，那么这时候就会产生数据不一致性问题。

## 分布式事务之 3PC (三阶段提交)

### CanCommit 阶段

协调者向所有参与者发送 CanCommit 请求，参与者收到请求后会根据自身情况查看是否能执行事务，如果可以则返回 YES 响应并进入预备状态，否则返回 NO。

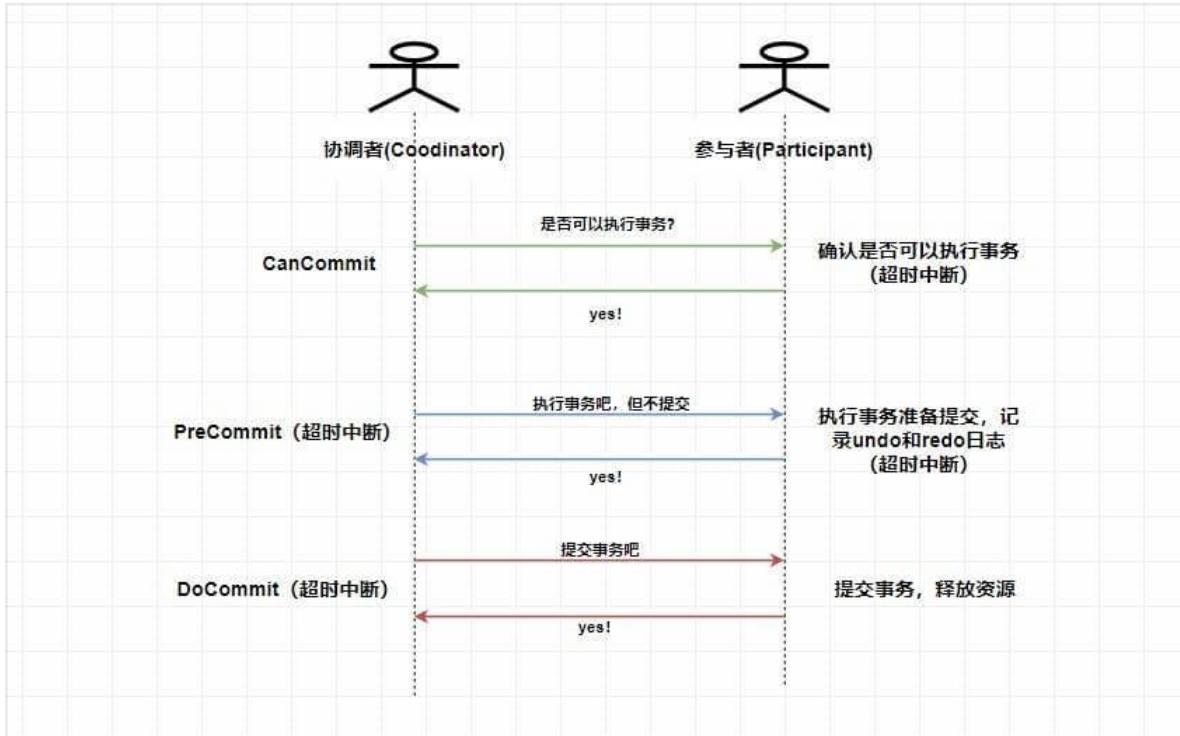
### PreCommit 阶段

协调者根据参与者返回的响应来决定是否可以进行下面的 PreCommit 操作。如果上面参与者返回的都是 YES，那么协调者将向所有参与者发送 PreCommit 预提交请求，参与者收到预提交请求后，会进行事务的执行操作，并将 Undo 和 Redo 信息写入事务日志中，最后如果参与者顺利执行了事务则给协调者返回成功的响应。如果在第一阶段协调者收到了任何一个 NO 的信息，或者在一定时间内并没有收到全部的参与者的响应，那么就会中断事务，它会向所有参与者发送中断请求 (abort)，参与者收到中断请求之后会立即中断事务，或者在一定时间内没有收到协调者的请求，它也会中断事务。

### DoCommit 阶段

这个阶段其实和 2PC 的第二阶段差不多，如果协调者收到了所有参与者在 PreCommit 阶段的 YES 响应，那么协调者将会给所有参与者发送 DoCommit 请求，参与者收到 DoCommit 请求后则会进行事务的提交工作，完成后则会给协调者返回响应，协调者收到所有参与者返回的事务提交成功的响应之后则完成事务。若

协调者在 PreCommit 阶段 收到了任何一个 NO 或者在一定时间内没有收到所有参与者的响应，那么就会进行中断请求的发送，参与者收到中断请求后则会通过上面记录的回滚日志来进行事务的回滚操作，并向协调者反馈回滚状况，协调者收到参与者返回的消息后，中断事务。



这里是 **3PC** 在成功的环境下的流程图，你可以看到 **3PC** 在很多地方进行了超时中断的处理，比如协调者在指定时间内未收到全部的确认消息则进行事务中断的处理，这样能减少同步阻塞的时间。还有需要注意的是，**3PC** 在 **DoCommit** 阶段参与者如未收到协调者发送的提交事务的请求，它会在一定时间内进行事务的提交。为什么这么做呢？是因为这个时候我们肯定保证了在第一阶段所有的协调者全部返回了可以执行事务的响应，这个时候我们有理由相信其他系统都能进行事务的执行和提交，所以不管协调者有没有发消息给参与者，进入第三阶段参与者都会进行事务的提交操作。

## 总结

总之，**3PC** 通过一系列的超时机制很好的缓解了阻塞问题，但是最重要的一致性并没有得到根本的解决，比如在 **DoCommit** 阶段，当一个参与者收到了请求之后其他参与者和协调者挂了或者出现了网络分区，这个时候收到消息的参与者都会进行事务提交，这就会出现数据不一致性问题。

## 什么是网络分区以及 3PC 在网络分区下为什么会出现数据不一致情况？

### 一、什么是网络分区？

网络分区，在分布式计算领域，不是指硬盘分区，而是指一种网络故障状态。

它描述的是：一个网络中的节点，因为网络连接的中断，被分割成了两个或多个无法相互通信的、独立的“子网络”或“孤岛”。

### 一个生动的比喻：

想象一个团队正在开一个重要的电话会议。这个团队有1个项目经理（协调者）和多个工程师（参与者）。

- 正常情况：所有人都在一个大的电话会议室里，互相能听到对方说话。

- **发生网络分区**: 突然之间, 电话线路出现了故障。
  - 项目经理和工程师 A、B 被分到了一个“**小组 A**”里, 他们之间可以正常通话。
  - 工程师 C、D 被分到了另一个“**小组 B**”里, 他们之间也可以正常通话。
  - 但是, **小组 A 和小组 B 之间完全失联了!** 小组 A 听不到小组 B 的任何声音, 反之亦然。

这个“**小组 A**”和“**小组 B**”就是两个网络分区。每个分区内部的节点通信是正常的, 但分区与分区之间的通信完全中断。

## 二、网络分区在 3PC 场景下的致命影响

**场景设定:**

- 系统中有 1 个协调者 (C) 和 3 个参与者 (P1, P2, P3)。
- 事务已经成功通过了 CanCommit 阶段, 所有参与者都已返回 VOTE\_COMMIT。
- 协调者 C 已经进入了 PreCommit 阶段, 并开始向 P1, P2, P3 发送 PRECOMMIT 消息。

**故障过程:**

1. **发送 PRECOMMIT**: 协调者 C 成功地向 **P1** 和 **P2** 发送了 PRECOMMIT 消息。
2. **网络分区发生**: 就在此时, 网络发生了分区。
  - **分区一**: 协调者 C 和参与者 P3 被隔离在一个分区。C 发往 P3 的 PRECOMMIT 消息丢失了, 并且 C 与 P1、P2 失联。
  - **分区二**: 参与者 P1 和 P2 被隔离在另一个分区。它们与 C 和 P3 失联。

现在, 我们来分析不同分区的行为, 看看不一致是如何产生的:

- **在分区二 (P1, P2 的世界里):**
  - P1 和 P2 都成功收到了 PRECOMMIT 指令。
  - 它们会执行本地的预提交操作 (写日志、锁资源), 并进入“准备提交”的状态。
  - 然后, 它们尝试向协调者 C 发送 ACK, 但因为网络分区, ACK 无法送达。
  - P1 和 P2 开始等待协调者 C 的最终 DO\_COMMIT 指令。
  - **等待超时**: 根据 3PC 协议, 参与者如果在 PreCommit 阶段后等待 DoCommit 超时, 为了避免永久阻塞, 它会自动执行事务提交。
  - **最终结果: P1 和 P2 提交了事务。**
- **在分区一 (C, P3 的世界里):**
  - 协调者 C 因为它没有收到 P1 和 P2 的 ACK (因为失联), 并且它也无法成功向 P3 发送 PRECOMMIT 消息, 它会等待超时。
  - **等待超时**: 根据 3PC 协议, 协调者如果在 PreCommit 阶段没有收到所有参与者的 ACK, 它就会认为预提交失败, 必须决定中断事务。
  - 协调者 C 会向它能联系到的所有参与者 (这里只有 P3) 发送 GLOBAL\_ABORT 消息。
  - 参与者 P3, 因为它从来没有收到过 PRECOMMIT 消息 (它还在等待这个消息), 在等待 PRECOMMIT 超时或者直接收到了协调者的 GLOBAL\_ABORT 后, 它会中断并回滚事务。
  - **最终结果: P3 回滚了事务。**

**数据不一致性产生!**

- 在系统恢复后, 我们发现:
  - 参与者 P1 和 P2 的数据已经被成功提交。

- 参与者 P3 的数据已经被成功回滚。
- 整个分布式系统的原子性被彻底破坏，出现了严重的数据不一致。

### 总结

- **网络分区是什么：**它是一种网络故障，将分布式系统中的节点分割成多个无法互相通信的孤岛。
- **它如何破坏 3PC 的一致性：**在 3PC 的最后阶段 (DoCommit)，如果发生网络分区，可能会导致：
  - 一部分参与者收到了 DoCommit 指令并执行提交。
  - 另一部分参与者因为收不到指令而进入超时逻辑。3PC 的超时逻辑（通常是在 PreCommit 成功后选择提交）是为了解决阻塞问题而设计的，但它并不能保证 100% 猜对协调者的最终决策。
- **最终结果：**当超时逻辑的“猜测”与协调者的实际决策（或者其他分区的参与者收到的决策）不一致时，整个系统的数据就会出现不一致的状态，有的节点提交了，有的节点可能回滚了（如果在更早阶段就失联并超时），从而破坏了分布式事务的原子性。

这正是为什么像 Paxos、Raft 这样的共识算法诞生的原因，它们旨在解决在可能出现网络分区和节点故障的不可靠网络中，如何让所有节点对一个值达成一致的问题。

**PS：当一个参与者成功收到并完成了 PreCommit 操作之后，如果因为网络分区导致等待最终指令 (DoCommit 或 Abort) 超时，它会默认执行“事务提交”操作**

---

---

## Spring

---

---

## Spring IOC

### 概要

**IoC (Inversion of Control:控制反转)** 是一种设计思想，而不是一个具体的技术实现。IoC 的思想就是将原本在程序中手动创建对象的控制权，交由 Spring 框架来管理。不过，IoC 并非 Spring 特有，在其他语言中也有应用。

### 为什么叫控制反转？

- **控制：**指的是对象创建（实例化、管理）的权力
- **反转：**控制权交给外部环境（Spring 框架、IoC 容器）

### IOC 大致流程：

#### 粗糙版

首先类要生产成一个 bean，分为两个阶段，第一个阶段就是容器启动阶段，包含了 beanDefinition 的生成；第二个阶段是 Bean 的实例化阶段，包括 Bean 的实例化、填充字段、初始化等步骤。

#### 详细版

Spring IoC 容器的启动和 Bean 的创建，可以看作是一个从“蓝图”到“成品”的“工厂生产线”。整个流程大致可以分为两大阶段：容器启动阶段（BeanDefinition 的加载与处理）和 Bean 实例化阶段（getBean 触发）。

## 第一阶段：容器启动——“蓝图”的准备与加工

这个阶段的核心任务是解析配置，找到所有需要被 Spring 管理的 Bean，并将它们转换成 Spring 内部的统一数据结构——BeanDefinition。BeanDefinition 就像是创建 Bean 的“详细施工图纸”。

### 1. 启动与定位配置 (new)

#### AnnotationConfigApplicationContext(AppConfig.class))

- 入口：一切始于创建一个 ApplicationContext 的实例，例如 AnnotationConfigApplicationContext。
- 动作：容器启动后，首先需要定位到配置信息。对于注解驱动的 Spring，就是找到你传入的配置类（如 AppConfig.com）。

### 2. 扫描与解析 (@ComponentScan, @Component, etc.)

- 动作：Spring 会解析配置类上的注解。最重要的注解之一是 @ComponentScan，它告诉 Spring 要去哪些包（package）路径下进行扫描。
- 扫描：Spring 会像一个“雷达”一样，扫描指定包及其子包下的所有 .class 文件，寻找被特定注解（如 @Component, @Service, @Repository, @Controller, @Configuration）标记的类。

### 3. 生成 BeanDefinition (BeanDefinition)

- 动作：对于每一个扫描到的被标记的类，Spring 并不是立刻创建它的实例。相反，它会解析这个类的信息（如类名、作用域 Scope、是否懒加载 Lazy、构造函数参数、属性等），然后将这些元数据封装成一个 BeanDefinition 对象。
- 目的：BeanDefinition 是 Spring 管理 Bean 的“蓝图”。它包含了创建这个 Bean 所需的一切信息。

### 4. 注册 BeanDefinition

- 动作：所有生成的 BeanDefinition 都会被注册到一个内部的、 ConcurrentHashMap 类型的 \*\*beanDefinitionMap\*\* 中。这个 Map 的 Key 是 Bean 的名称（beanName），Value 就是对应的 BeanDefinition。此时，工厂里已经有了所有产品的“施工图纸”。

### 5. 后处理器介入：修改“蓝图”（这是 Spring 强大扩展性的体现）

- 关键时刻：在所有常规的 BeanDefinition 都已注册完毕，但任何 Bean 实例都还未创建的这个时间点，Spring 会调用一个非常重要的后处理器—— BeanFactoryPostProcessor。
- 动作：
  - ◆ BeanFactoryPostProcessor（及其子接口 BeanDefinitionRegistryPostProcessor）可以对 beanDefinitionMap 中的“蓝图”进行最后的修改。
  - ◆ 典型应用：
    - 我们最常用的@Configuration 类之所以能处理 @Bean、@Import 等注解，就是通过一个名为 ConfigurationClassPostProcessor 的

BeanDefinitionRegistryPostProcessor 来实现的。它会进一步解析配置类，并注册更多的 BeanDefinition。

- 占位符替换 (如 \${...}) 也是在这个阶段完成的。

到此为止，容器的启动阶段基本完成。工厂里所有的“图纸”都已备齐并最终定稿，但生产线上还没有任何一个“实体产品”。

## 第二阶段：Bean 实例化 —— 按需生产“成品”

这个阶段通常由第一次调用 getBean() 方法来触发 (对于非懒加载的单例 Bean，会在容器启动阶段结束后立刻触发)。

### 1. getBean(beanName) 调用

- 入口: 当代码执行 applicationContext.getBean("userService") 时，或者当一个 Bean 需要注入另一个 Bean 时，实例化流程开始。

### 2. 实例化 (Instantiation)

- 动作: Spring 根据 userService 对应的 BeanDefinition，通过反射 (或工厂方法) 来创建这个 Bean 的一个空白实例。此时，对象被创建出来了，但里面的属性都还是 null (或默认值)。

### 3. 填充属性 (Populate Properties)

- 动作: Spring 会检查 BeanDefinition 中定义的属性依赖。如果 UserService 需要注入一个 UserRepository，Spring 会去调用 getBean("userRepository") 来获取 (或创建) UserRepository 的实例，然后通过反射将它设置到 UserService 实例的对应字段上。
- 解决循环依赖: Spring AOP 和三级缓存解决循环依赖的关键就在于这个阶段。

### 4. 初始化 (Initialization)

- 这是 Bean “活过来”的最后一步，提供了多个扩展点，让 Bean 在准备好被使用前，可以执行一些自定义的初始化逻辑。
- 动作 (按顺序):
  - ◆ 执行各种 Aware 接口的回调 (如 BeanNameAware, BeanFactoryAware)。
  - ◆ 执行 BeanPostProcessor 的 postProcessBeforeInitialization 方法。这是一个非常重要的扩展点，AOP 的代理对象通常就是在这里创建的。
  - ◆ 执行 @PostConstruct 注解标记的方法，或 InitializingBean 接口的 afterPropertiesSet 方法。
  - ◆ 执行 BeanDefinition 中指定的 init-method。
  - ◆ 执行 BeanPostProcessor 的 postProcessAfterInitialization 方法。AOP 也可能在这里完成代理包装。

### 5. Bean 就绪

- 经过初始化后，一个完全配置好、可随时使用的 Bean 实例就诞生了。它会被放入单例缓存池 (singletonObjects) 中，以便下一次 getBean() 调用时能直接返回。

## IOC 详细流程之 12 个方法解析

- 1. prepareRefresh();

记录下容器的启动时间、标记“已启动”/“关闭”状态、初始化属性源、验证必须的

配置属性等。

- 2、ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

**创建 BeanFactory 对象【DefaultListableBeanFactory】，并解析 XML 封装成 BeanDefinition 对象注册到 BeanFactory 中**，主要目的是为了获取一个配置好的、但未初始化的 BeanFactory 实例，以便进行进一步的 bean 定义加载、注册和管理。

- 3、prepareBeanFactory(beanFactory);

配置 beanFactory 的标准参数：例如设置 BeanFactory 属性【例如是否支持循环引用、能否覆盖已定义的 bean】、添加 XXXAware 接口、添加 XXXPostProcessor 接口等，主要为了确保在实际的 bean 创建之前，BeanFactory 所有相关的基础设施和配置都已经正确设置。

- 4、postProcessBeanFactory(beanFactory);

此方法在所有的 bean 定义被加载到 BeanFactory 之后，但在 bean 的初始化之前被调用。这意味着你可以在这个方法中对 BeanFactory 中的 bean 定义进行额外的处理，例如添加新的 bean 定义、修改现有 bean 的属性或作用域等。

- 5、invokeBeanFactoryPostProcessors(beanFactory);

Spring 框架确保了在 bean 定义加载完成后、但在 bean 初始化之前，所有的 BeanFactoryPostProcessor 都被正确调用，从而对 BeanFactory 进行必要的修改和增强。

- 6、registerBeanPostProcessors(beanFactory);

**主要作用是将所有的 BeanPostProcessor 实例注册到 BeanFactory 中，以便在后续的 bean 初始化过程中对 bean 进行额外的处理。**

BeanPostProcessor 接口定义了两个方法：

**postProcessBeforeInitialization** 和 **postProcessAfterInitialization**，它们分别在 bean 的属性设置（依赖注入）之后但在 bean 自定义初始化方法之前，以及 bean 自定义初始化方法之后被调用。这使得 BeanPostProcessor 能够在 bean 的生命周期中的特定点插入自定义逻辑【即为前置处理和后置处理，AOP 代理增强使用】。

注册 BeanPostProcessor 之后，当 Spring 容器创建和初始化一个 bean 时，这些后处理器将被自动调用，允许你在 bean 的生命周期中插入自定义逻辑。例如，你可以使用 BeanPostProcessor 来自动装配额外的依赖、修改 bean 的属性、执行 AOP 增强等。

- 7、initMessageSource();

初始化国际化【不重要】

- 8、initApplicationEventMulticaster();

初始化应用事件广播器：广播器的主要作用就是存放 Listener。这里看 beanFactory 是否包含 applicationEventMulticaster，包含则 getBean 获取，不包含则通过 new SimpleApplicationEventMulticaster (ApplicationEventMulticaster 子类) 注册到 Spring 容器中，所以此方法只是初始化用的。

- 9、onRefresh();

这是个模板方法，实现是空的，具体实现交由子类处理。

主要看看 ServletWebServerApplicationContext 里面的 createWebServer() 方法【其中一个实现的子类】，其中里面有一个 factory.getWebServer 方法，主要就是 TomcatWebServer 的 initialize()，会有一个 this.tomcat.start()。所以这个方法就是用来实例化一些特殊的 bean，就像启动 tomcat。

- 10、registerListeners();

注册事件监听器：此方法里面是从 Spring 容器中，拿到实现了 ApplicationListener 接口的所有 BeanName，然后添加到广播器中的 this.defaultRetriever.applicationListenerBeans 中。

- 11、finishBeanFactoryInitialization(beanFactory);

初始化所有的 singleton beans (lazy-init 的除外)，内部比较重要的方法如下：

1、getBean【获取指定名称和类型的 bean 实例，也会去处理循环依赖、bean 的作用域、单例缓存等问题】

2、getSingleton【从 BeanFactory 的单例缓存中获取 bean 实例】、

3、addSingletonFactory【用于向 BeanFactory 的单例缓存中添加一个新的 bean 实例工厂，确保了即使在 bean 完全初始化之前，也可以通过 getSingleton 方法获取到 bean 的实例】

- 12、finishRefresh();

最后，广播事件，ApplicationContext 初始化完成

### PS：简单小总结【不完全版】

1、创建 beanFactory，并解析 bean 为 beanDefinition 注册到 beanFactory

2、设置 beanFactory 属性

3、设置 beanDefinition 属性

4、将前置后置处理器注册到 beanFactory，便于后续的代理使用

5、注册监听器

6、初始化所有的 bean，包含实例化 bean、解决循环依赖等

## IOC 和 DI 关系

IOC 是一种设计思想或者说是某种模式。这个设计思想就是 将原本在程序中手动创建对象的控制权交给第三方比如 IoC 容器。对于 Spring 来说，DI 是 IOC 一种实现方式，通过容器自动将所需的依赖注入到需要它们的对象中，从而实现对象之间的解耦。

---

## BeanFactory 和 FactoryBean 区别

## BeanFactory 和 FactoryBean 的详细区别如下

1. 作用不同：BeanFactory 是 Spring 框架的基础设施，它是用于管理 bean 的工厂容器。它只提供了基础的功能，如获取 bean、检测 bean、管理 bean 的生命周期等。而 FactoryBean 本身也是一个 BeanFactory，用于创建其他 Bean，它提供了一种灵活的方式来创建和配置复杂的 Bean 对象。
2. 返回值不同：BeanFactory 获取的是 bean 的实例对象，而 FactoryBean 获取的是 FactoryBean 本身的实例对象，也就是调用 FactoryBean 的 getObject() 方法后返回的对象。
3. 配置方式不同：BeanFactory 通常通过 XML 配置文件或 Java 注解进行配置，定义和管理 Bean 对象。而 FactoryBean 则没有特定的配置方式，它根据自身的逻辑来创建和配置 Bean 对象。
4. 创建时机不同：BeanFactory 在获取 Bean 时动态创建 Bean 对象，即按需创建。而 FactoryBean 则在调用其 getObject() 方法时创建 Bean 对象。

```
@Component
public class MyBeanFactory implements FactoryBean<MyBean> {

    @Override
    public MyBean getObject() throws Exception {
        // 在这里执行创建和配置 MyBean 对象的逻辑
        MyBean myBean = new MyBean();
        myBean.setName("Example");
        myBean.setValue(123);
        return myBean;
    }

    @Override
    public Class<?> getObjectType() {
        return MyBean.class;
    }

    @Override
    public boolean isSingleton() {
        return true;
    }
}
```

## 总结

BeanFactory 是 Spring 框架生产 Bean 的基础设施，包含了获取 bean、检测 bean、管理 bean 的生命周期等，而 FactoryBean 是一个修饰对象且生产出复杂的 Bean，本身就是一个 Bean（通过 getObject() 返回对象本身来获取 Bean），只不过可以创建复杂的 Bean，所以他就是一个 Bean。

## BeanFactory 和 ApplicationContext 区别

### 共同点

都有生产 bean 的能力，因为 context (无论是 AnnotationConfigApplicationContext 或 ClassPathXmlApplicationContext 都实现了 BeanFactory)

### 区别

Context 的功能更加完善 【context 也是实现了 BeanFactory，只不过添加了更多的功能】

context 可以指定配置类或者扫描配置类，以及包括了 Spring 上下文的生命周期、事件、扫描器、读取器、AOP 等，还有比如上面的 BeanDefinition 读取、扫描、注册的功能都会被包含在里面，工厂却无法有这些功能；而 BeanFactory 只是一个没有感情的生产 Bean 的机器，不会承担更多的功能，且没有扫描包的能力，没法给工厂指定一个包，只能注册一个 bean 定义（类似图纸）

### 加载方式

BeanFactory 是延时加载，在容器启动时不会注入 bean，只有在需要使用 bean 的时候，才会对该 bean 进行加载实例化；ApplicationContext 是在容器启动的时候，一次性创建所有的 bean，所以运行的时候速度相对 BeanFactory 比较快。

---

---

## IOC 中 doGetBean 里的三级缓存如何解决 Spring 的循环依赖？

### 三级缓存介绍

- 一级缓存 (**singletonObjects**): 成品仓。存放已经完全初始化好的单例 Bean。
- 二级缓存 (**earlySingletonObjects**): 半成品仓。存放提早暴露的 Bean 实例（可能是原始对象，也可能是被提前创建的代理对象）。它里面的 Bean 属性可能还未填充完毕。
- 三级缓存 (**singletonFactories**): 原材料 / 蓝图仓。不直接存放 Bean，而是存放能创建出 Bean 最终形态的工厂 (**ObjectFactory**)。

### 一二级缓存即可解决循环依赖

A 的 Bean 在创建过程中，在进行依赖注入之前，先把 A 的原始 Bean 放入缓存（提早暴露，只要放到缓存了，其他 Bean 需要的时候就可以从缓存中拿出来用【即放入到 **earlySingletonObjects**，其他 Bean 需要引用的时候只需指向引用，毕竟内存地址不会变，等到这个 Bean 填充完属性等初始化流程结束自然也等于依赖到了完整流程的 Bean】），放入缓存后，再进行依赖注入，此时 A 的 Bean 依赖了 B 的 Bean，如果 B 的 Bean 不存在，就需要创建 B 的 Bean，而创建 B 的 Bean 的过程和 A 一样，也是先创建一个 B 的原始对象，然后把 B 的原始对象提早暴露出来放入缓存中，然后再对 B 的原始对象进行依赖注入 A，此时能从缓存中拿到 A 的原始对象（虽然是 A 的原始对象，还不是最终的 Bean），B 的原始对象依赖注入完之后，B 的生命周期结束，那么 A 的生命周期也

能结束。

### 三级缓存存在的意义是什么？

如果 A 的原始对象注入给 B 属性之后, A 原始对象进行了 AOP 产生了一个代理对象, 此时就会出现, 对于 A 而言, 它的 Bean 对象起始应该是 AOP 之后的代理对象, 而 B 的 A 属性对应的并不是 AOP 之后的代理对象, 这就产生冲突了。

B 依赖的 A 和最终的 A 不是同一个对象【A 对象如果被 AOP 代理了, 原本实例和代理之后的实例将不会是同一个对象, 那么 B 依赖的 A 和实际代理后的 A 就不是一个对象, 造成冲突】

## Spring 三级缓存解决循环依赖与 AOP 代理问题的核心工作流程

情况一：A、B 循环依赖，且 A、B 均无 AOP 代理的流程

### 1. getBean("a") -> 启动 A 的创建流程

- Spring 容器接到指令, 需要获取 Bean "a"。
- 检查缓存:
  - \* 一级缓存 singletonObjects 中没有 "a"。
  - \* 二级缓存 earlySingletonObjects 中没有 "a"。
  - \* 三级缓存 singletonFactories 中没有 "a"。
- 标记为“正在创建”: Spring 将 "a" 标记为正在创建中, 防止重复创建。

### 2. 实例化 A

- Spring 通过反射调用 A 的无参构造函数, new A(), 创建了一个原始的 A 对象实例。此时, 这个 A 对象是一个“空壳”, 其字段 b 还是 null。

### 3. 将 A 的“工厂”放入三级缓存 (关键步骤)

- 虽然 A 不需要 AOP 代理, 但 Spring 的流程是统一的。因为它是一个可能产生循环依赖的单例, Spring 依然会为其创建一个 ObjectFactory。
- 这个工厂的内部逻辑是: () -> getEarlyBeanReference("a", a(beanDefinition, originalA))。
- singletonFactories.put("a", a\_factory)。
- 注意: 此时, 一、二级缓存仍然是空的。

### 4. 填充 A 的属性 -> 触发 getBean("b")

- Spring 开始为 A 的原始对象填充属性, 发现它 @Autowired 了一个 B。
- Spring 暂停 A 的创建, 转而去调用 getBean("b")。

### 5. 启动 B 的创建流程

- getBean("b") 的流程与 A 完全一样:
  - \* 检查三级缓存, 找不到 "b"。
  - \* 将 "b" 标记为正在创建中。
  - \* 实例化 B, new B(), 得到一个原始的 B 对象实例 (a 字段为 null)。
  - \* 为 B 创建一个 ObjectFactory, 并放入三级缓存 singletonFactories。  
singletonFactories.put("b", b\_factory)。

### 6. 填充 B 的属性 -> 触发 getBean("a")

- Spring 填充 B 的属性, 发现它 @Autowired 了一个 A。
- 再次调用 getBean("a")。

### 7. 解决循环依赖的关键一步 (与 AOP 场景的不同之处)

- getBean("a") 再次被调用, Spring 开始查找:
  - \* 查一级缓存: 没有。

- \* 查二级缓存：没有。
- \* 查三级缓存：找到了 A 的 ObjectFactory！
- 执行工厂：Spring 会执行这个工厂，即调用 a\_factory.getObject()。
- 工厂内部逻辑：
  - \* 工厂会调用 getEarlyBeanReference() 方法，询问所有 BeanPostProcessor：“是否需要为 A 提前创建代理？”
  - \* 因为 A 是普通 POJO，没有 AOP 切面，所有后处理器都回答“不需要”。
  - \* 因此，getEarlyBeanReference() 方法直接返回了 A 的原始对象实例。
- 缓存升级（重要！）：
  - \* 工厂的执行结果——A 的原始对象——被返回。
  - \* 这个返回的原始对象会被放入二级缓存 earlySingletonObjects 中。  
`earlySingletonObjects.put("a", originalA);`
  - \* 同时，三级缓存 singletonFactories 中 "a" 对应的工厂会被删除。
  - \* 这一步的意义：将三级缓存中的“可能性”（工厂）具体化为一个“确定的半成品”（原始对象），并放入二级缓存。这样，如果还有其他 Bean 也依赖 A，可以直接从二级缓存获取，无需重复执行工厂。
- 最终，A 的原始对象被注入到 B 的 a 字段中。

## 8. 完成 B 的创建

- B 的属性填充完毕，进入初始化阶段。由于 B 是普通 POJO，初始化很快完成。
- 最终的、完整的 B 对象被放入 \*\*一级缓存 singletonObjects\*\* 中。
- 同时，B 会从二级缓存和正在创建的标记中被移除。

## 9. 回到 A 的创建流程

- B 的实例被成功返回，并注入到 A 的 b 字段中。
- A 的属性填充完毕，进入初始化阶段。
- A 的初始化完成。

## 10. 完成 A 的创建

- \* 最终的、完整的 A 对象（就是那个原始对象）被放入一级缓存 singletonObjects。
- \* 同时，A 会从二级缓存（之前放入的那个）和正在创建的标记中被移除。

## 情况二：A、B 循环依赖，且 A 需要 AOP 代理的流程

- 1. getBean("a") -> 创建 A 实例**
  - Spring 尝试创建 Bean A。在一级、二级缓存中都找不到。
  - **实例化（Instantiation）：**通过反射 new A()，创建一个原始的 A 对象实例。
  - 放入三级缓存：为了解决潜在的循环依赖，Spring 会立刻为这个原始 A 对象创建一个 ObjectFactory（可以理解为一个 lambda 表达式：() -> getEarlyBeanReference("a", a\_beandefinition, originalA)），并将其存入三级缓存 singletonFactories。这个工厂封装了判断并创建 AOP 代理的逻辑。
- 2. 填充 A 的属性 -> 触发 getBean("b")**
  - Spring 开始为 A 的原始对象填充属性，发现它依赖于 B。
  - Spring 暂停 A 的创建，转而去调用 getBean("b")。
- 3. 创建 B 实例**
  - B 的创建流程与 A 类似：实例化 B 的原始对象，然后也为 B 创建一个工厂放入三级缓存。
- 4. 填充 B 的属性 -> 触发 getBean("a")**

- Spring 填充 B 的属性，发现它依赖于 A。
- 再次调用 getBean("a")。

## 5. 解决循环依赖的关键一步

- getBean("a") 再次被调用，Spring 开始查找：
  - ◆ 查一级缓存：没有（A 还未创建完）。
  - ◆ 查二级缓存：没有（A 的实例还未被提前具体化）。
  - ◆ 查三级缓存：找到了 A 的 ObjectFactory！
- 执行工厂，提前 AOP：Spring 会执行这个工厂。工厂内部的 getEarlyBeanReference 方法会检查所有 BeanPostProcessor，询问“这个 A 对象是否需要提前创建代理？”。处理 AOP 的后处理器会回答“是！”，于是 A 的代理对象在此刻被提前创建出来。
- 缓存升级：这个新鲜出炉的 A 的代理对象，会被放入二级缓存 **earlySingletonObjects** 中，同时三级缓存中的工厂会被删除。这是为了保证对 A 的所有后续引用都指向这个唯一的代理对象。
- B 的 a 字段成功注入了 A 的代理对象。

## 6. 完成 B 的创建

- B 的属性填充和初始化完成。
- 最终的 B 对象被放入一级缓存 **singletonObjects**。

## 7. 回到 A 的创建流程

- B 的实例被成功返回，并注入到 A 的原始对象中。
- A 的生命周期继续，进入初始化阶段。当 AOP 的后处理器再次运行时，它会发现“a”这个 Bean 的代理已经存在于二级缓存中了，于是直接返回该代理，不再重复创建。
- A 的初始化全部完成后，最终的 A 的代理对象被放入一级缓存 **singletonObjects**。

至此，A 和 B 都成功创建，并且 B 中引用的 A，和最终容器中的 A，是同一个代理对象，完美解决了所有问题。

PS：无论一个 Bean 最终是否需要被 AOP 代理，只要它满足了可以被“提早暴露”的条件，它就会被放入三级缓存。所以，即使在没有 AOP 的场景下，Spring 依然会完整地走完三级缓存的流程，只不过区别在于创建 Bean 对象的时候会检查其 BeanPostProcessor 来判断是否要创建代理对象，是的话->三级缓存 ObjectFactory 创建代理对象，否的话->返回原始对象。

## 精简总结

理论上，一二级缓存可以解决循环依赖，但是如果涉及到创建的对象被 AOP 代理，导致原先依赖的对象和被代理后的对象不是同一个对象，为了解决这个问题，需要引入三级缓存来解决问题。

---

## AOP 面向切面编程

## 概要

面向切面编程，AOP 的目的是将横切关注点（如日志记录、事务管理、权限控制、接口限流、接口幂等等）从核心业务逻辑中分离出来，通过动态代理、字节码操作等技术，实现代码的复用和解耦，提高代码的可维护性和可扩展性。OOP 的目的是将业务逻辑按照对象的属性和行为进行封装，通过类、对象、继承、多态等概念，实现代码的模块化和层次化（也能实现代码的复用），提高代码的可读性和可维护性。所以，AOP 是 OOP（面向对象编程）的一种延续，二者互补，并不对立。

## AOP 解决了什么问题

公共行为的代码重复编写导致代码冗余、复杂和难以维护，例如日志记录，可能在多个场景或者接口都需要记录到数据库。那么通过 AOP 将日志记录的逻辑封装成一个切面，然后通过切入点和通知来指定在哪些方法需要执行日志记录的操作。这样的话，我们一行注解即可实现日志记录：

## AOP 实现方式

AOP 的常见实现方式有动态代理、字节码操作等方式。

Spring AOP 就是基于动态代理的，如果要代理的对象，实现了某个接口，那么 Spring AOP 会使用 **JDK Proxy**，去创建代理对象，而对于没有实现接口的对象，就无法使用 JDK Proxy 去进行代理了，这时候 Spring AOP 会使用 **Cglib** 生成一个被代理对象的子类来作为代理。

## JDK 代理和 CGLIB 代理的区别如下：

- 实现方式**：JDK 代理基于接口来创建被代理对象的代理实例，而 CGLIB 代理基于继承的方式对被代理类生成子类。
- 性能差异**：JDK 代理因为它基于反射机制，所以在调用代理方法时性能上不如 CGLIB。而 CGLIB 代理通过直接操作字节码生成新的类，避免了使用反射的开销，所以通常认为其性能要比 JDK 代理更好。
- 使用场景差异**：JDK 代理适用于接口驱动的代理场景，在不涉及具体类，只关心接口定义时非常适用。而 CGLIB 代理在需要代理没有实现接口的类，或者需要通过继承来提供增强功能的场景更适用。
- 限制**：JDK 代理要求被代理类必须实现至少一个接口，而 CGLIB 代理要求被代理类不能是 final 类，且方法也不能是 final 或 static 的。

## AOP 代理中，为什么反射的性能比字节码差？

首先，反射是一种在运行时动态地获取和操作类、接口、字段和方法信息的机制。它允许程序在运行时改变其行为，提供了极大的灵活性。然而，这种灵活性是以性能为代价的。反射操作通常涉及到对类型信息的动态解析，这导致了额外的运行时开销。Java 虚拟机 (JVM) 在执行反射操作时，无法进行某些优化，因为这些操作是在运行时动态确定的，而不是在编译时静态确定的。因此，反射操作的性能通常比直接方法调用或字节码操作要差。

相比之下，字节码操作通常是通过修改类的字节码来实现的。这种方式可以在编译时或类加载时完成，因此不涉及到运行时的动态解析。一旦字节码被修改，JVM 就可以像处理普通代码一样对其进行优化。这意味着基于字节码的代理通常具有更好的性能。

在 AOP 代理中，使用反射还是字节码操作取决于具体的需求和场景。基于接口的代理通常使用 Java 动态代理，它基于反射机制实现。这种方式适用于那些已经实现了

接口的类，并且要求目标对象实现一个接口。而基于字节码的代理则适用于那些没有实现接口的类，通过使用字节码操作库（如CGLIB、ASM等）来生成代理类。由于不需要目标对象实现接口，基于字节码的代理具有更广泛的适用性。

综上所述，反射在AOP代理中性能较差的原因主要在于其动态性和运行时解析的特性，而CGLIB在方法调用时不需要通过反射，而是直接调用目标方法，这通常比JDK动态代理通过反射调用目标方法具有更高的性能。

**PS：早期Spring：**默认策略是，如果目标类实现了接口，就用JDK动态代理；如果没实现接口，就用CGLIB。**Spring Boot 2.x及以后：**由于CGLIB在处理没有接口的类时更具通用性，并且现代JDK下性能差异不再是决定性因素，Spring Boot为了统一代理行为、解决一些因代理方式不同而导致的怪异问题（比如一个Bean注入给自己时，类型转换失败），做出了一个重要的改变：默认情况下，无论目标类是否实现接口，都统一使用CGLIB来创建代理。

不过可以通过配置`spring.aop.proxy-target-class=false`或如下图的注解，来切换回旧的默认行为->`false`表示统一走CGLIB



```
> import ...  
  
    @EnableAsync  张恺  
    @SpringBootApplication  
    @ComponentScan(basePackages = {"com.bangdao"})  
    @EnableScheduling  
    @EnableAspectJAutoProxy(exposeProxy = true) // This line is highlighted with a red rectangle.  
    @MapperScan({"com.bangdao.**.mapper"})  
    public class EnergysalesApplication {  
  
        public static void main(String[] args) {  张恺  
            new SpringApplicationBuilder(EnergysalesApplication.class).bannerMode(Banner.Mode.LOG).run(args);  
        }  
    }  
}
```

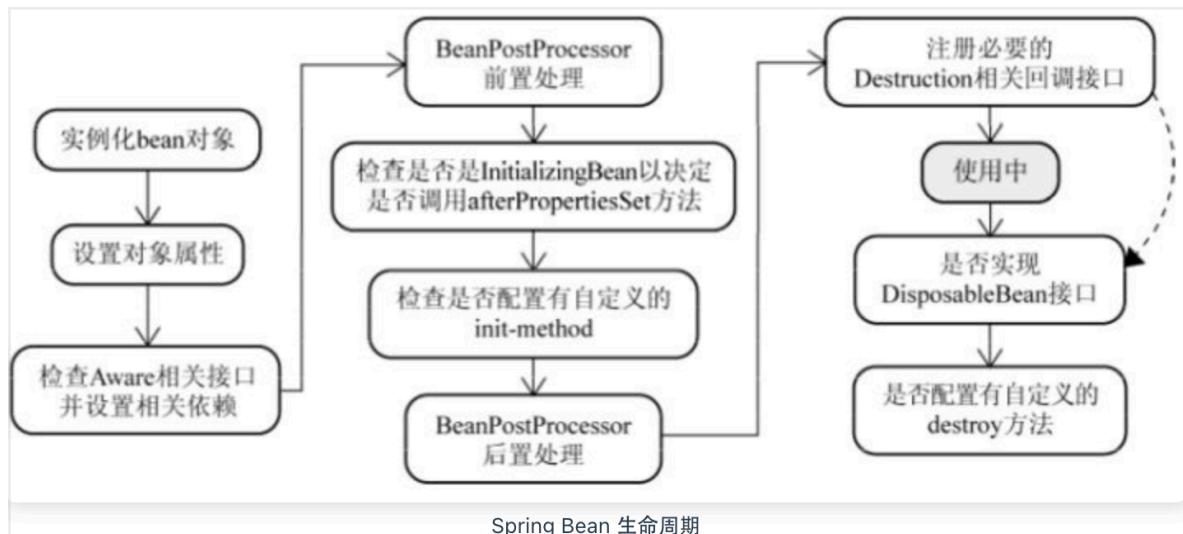
## AOP获取代理对象失败了怎么办？

在AOP代理中，如果获取代理的对象失败了，首先要做的是检查导致失败的具体原因。通常，失败的原因可能涉及以下几个方面：

- 配置问题：**检查AOP的配置是否正确。这包括切点表达式、通知类型以及代理模式的配置。错误的配置可能导致代理对象无法正确创建或通知无法正确应用。
- 目标对象问题：**确保被代理的目标对象是正确的，并且它已经被Spring容器管理。Spring AOP依赖于Spring的IoC容器来创建和管理代理对象，如果被代理的类没有被Spring容器管理，代理将不会生效。
- 方法调用问题：**如果在同一个类中的方法调用另一个方法，AOP通知可能不会触发，因为AOP通常是通过代理对象拦截外部方法调用的。解决方法可以是注入本类对象进行调用，或者设置暴露当前代理对象到本地线程。
- 静态和私有方法：**AOP通常无法拦截静态方法和私有方法的调用，因为这些方法不是通过对象调用的。如果需要对这些方法应用切面逻辑，可能需要考虑其他技术或设计模式来实现。
- 代理对象创建问题：**检查代理对象的创建过程是否有误。这包括代理类的生成、增强方法的扫描以及代理对象的缓存等步骤。通过调试底层源码可以跟踪代理对象的创建过程，确保代理对象正确创建并且AOP通知能够拦截代理对象的方法调用。

## Bean 的生命周期

- Bean 容器找到配置文件中 Spring Bean 的定义。
- Bean 容器利用 Java Reflection API 创建一个 Bean 的实例。
- 如果涉及到一些属性值 利用 set() 方法设置一些属性值。
- 如果 Bean 实现了 BeanNameAware 接口，调用 setBeanName() 方法，传入 Bean 的名字。
- 如果 Bean 实现了 BeanClassLoaderAware 接口，调用 setBeanClassLoader() 方法，传入 ClassLoader 对象的实例。
- 如果 Bean 实现了 BeanFactoryAware 接口，调用 setBeanFactory() 方法，传入 BeanFactory 对象的实例。
- 与上面的类似，如果实现了其他 \*.Aware 接口，就调用相应的方法。
- 如果有和加载这个 Bean 的 Spring 容器相关的 BeanPostProcessor 对象，执行 postProcessBeforeInitialization() 方法
- 如果 Bean 实现了 InitializingBean 接口，执行 afterPropertiesSet() 方法。
- 如果 Bean 在配置文件中的定义包含 init-method 属性，执行指定的方法。
- 如果有和加载这个 Bean 的 Spring 容器相关的 BeanPostProcessor 对象，执行 postProcessAfterInitialization() 方法
- 当要销毁 Bean 的时候，如果 Bean 实现了 DisposableBean 接口，执行 destroy() 方法。
- 当要销毁 Bean 的时候，如果 Bean 在配置文件中的定义包含 destroy-method 属性，执行指定的方法。



## Spring 中的设计模式

### 1、工厂模式：

Spring 使用工厂模式可以通过 BeanFactory 或 ApplicationContext 创建 bean 对象。

2、单例模式：Spring 中的 Bean 默认都是单例的【当然也可以设置 Bean 作用域例如 prototype 就是多例模式】。

3、代理模式：Spring AOP 功能的实现【JDK 或 cglib 代理模式】。

4、模板模式：Spring 中 jdbcTemplate、hibernateTemplate 等以 Template 结尾的对数据库操作的类，它们就使用到了模板模式。

5、观察者模式：Spring 的事件驱动【ApplicationEvent 事件、ApplicationListener 事件监听者、ApplicationEventPublisher 事件发布者】

事件驱动流程：

- 定义一个事件：实现一个继承自 ApplicationEvent，并且写相应的构造函数；
- 定义一个事件监听者：实现 ApplicationListener 接口，重写 onApplicationEvent() 方法；
- 使用事件发布者发布消息：可以通过 ApplicationEventPublisher 的 publishEvent() 方法发布消息。

6、适配器模式：Spring AOP 的增强或通知 (Advice) 使用到了适配器模式、spring MVC 中也是用到了适配器模式适配 Controller。

## →最终完善的总结列表

1. 工厂模式: BeanFactory, ApplicationContext, FactoryBean
  2. 单例模式: Bean 的默认 Scope, 由单例注册表管理。
  3. 代理模式: Spring AOP (JDK 动态代理, CGLIB)。
  4. 模板方法模式: JdbcTemplate, RedisTemplate。
  5. 观察者模式: Spring 的事件驱动模型 (ApplicationEvent, ApplicationListener)。
  6. 适配器模式: Spring MVC (HandlerAdapter), Spring AOP (Advice 适配)。
  7. 策略模式: ResourceLoader, InstantiationStrategy。
  8. 装饰器模式: DataSource 包装, I/O 流包装。
  9. 委派模式: DispatcherServlet 的工作机制。
  10. 责任链模式: AOP 拦截器链, HandlerInterceptor 链。
- 
- 

## Spring 事务详解

### 题外话

- innodb 支持事务，myisam 不支持事务
- 数据密集型应用系统设计：<https://github.com/Vonng/ddia>

### MySQL 怎么保证原子性的？

我们知道如果想要保证事务的原子性，就需要在异常发生时，对已经执行的操作进行回滚，在 MySQL 中，恢复机制是通过 回滚日志（undo log）实现的，所有事务进行的修改都会先记录到这个回滚日志中，然后再执行相关的操作。如果执行过程

中遇到异常的话，我们直接利用回滚日志中的信息将数据回滚到修改之前的样子即可！并且，回滚日志会先于数据持久化到磁盘上。这样就保证了即使遇到数据库突然宕机等情况，当用户再次启动数据库的时候，数据库还能够通过查询回滚日志来回滚之前未完成的事务。

## Spring 支持两种方式的事务管理

- 编程式事务管理：通过 TransactionTemplate 或者 TransactionManager 手动管理事务，实际应用中很少使用。
- 声明式事务：通过 AOP 实现（基于 @Transactional 的全注解方式使用最多）

## Spring 事务管理接口介绍

事务管理相关最重要的3个接口如下

- **PlatformTransactionManager**: (平台) 事务管理器，Spring 事务策略的核心。
- **TransactionDefinition**: 事务定义信息(事务隔离级别、传播行为、超时、只读、回滚规则)。
- **TransactionStatus**: 事务运行状态。

我们可以把 PlatformTransactionManager 接口可以被看作是事务上层的管理者，而 TransactionDefinition 和 TransactionStatus 这两个接口可以看作是事务的描述。

PlatformTransactionManager 会根据 TransactionDefinition 的定义比如事务超时时间、隔离级别、传播行为等来进行事务管理，而 TransactionStatus 接口则提供了一些方法来获取事务相应状态比如是否新事务、是否可以回滚等等。

### 1、Spring 事务管理器：PlatformTransactionManager

PlatformTransactionManager 接口中定义了三个方法：

```
1 package org.springframework.transaction;
2
3 import org.springframework.lang.Nullable;
4
5 public interface PlatformTransactionManager {
6     //获得事务
7     TransactionStatus getTransaction(@Nullable TransactionDefinition var1) throws TransactionException;
8     //提交事务
9     void commit(TransactionStatus var1) throws TransactionException;
10    //回滚事务
11    void rollback(TransactionStatus var1) throws TransactionException;
12 }
13
```

### 2、事务属性：TransactionDefinition

事务管理器接口 PlatformTransactionManager 通过

getTransaction(TransactionDefinition definition) 方法来得到一个事务，这个方法里面的参数是 TransactionDefinition 类，这个类就定义了一些基本的事务属性。

**什么是事务属性呢？** 事务属性可以理解成事务的一些基本配置，描述了事务策略如何应用到方法上。

事务属性包含了 5 个方面：

- 隔离级别
- 传播行为
- 回滚规则
- 是否只读
- 事务超时

`TransactionDefinition` 接口中定义了 5 个方法以及一些表示事务属性的常量比如隔离级别、传播行为等。

```
1 package org.springframework.transaction;
2
3 import org.springframework.lang.Nullable;
4
5 public interface TransactionDefinition {
6     int PROPAGATION_REQUIRED = 0;
7     int PROPAGATION_SUPPORTS = 1;
8     int PROPAGATION_MANDATORY = 2;
9     int PROPAGATION_REQUIRE_NEW = 3;
10    int PROPAGATION_NOT_SUPPORTED = 4;
11    int PROPAGATION_NEVER = 5;
12    int PROPAGATION_NESTED = 6;
13    int ISOLATION_DEFAULT = -1;
14    int ISOLATION_READ_UNCOMMITTED = 1;
15    int ISOLATION_READ_COMMITTED = 2;
16    int ISOLATION_REPEATABLE_READ = 4;
17    int ISOLATION_SERIALIZABLE = 8;
18    int TIMEOUT_DEFAULT = -1;
19    // 返回事务的传播行为， 默认值为 REQUIRED。
20    int getPropagationBehavior();
21    // 返回事务的隔离级别， 默认值是 DEFAULT
22    int getIsolationLevel();
23    // 返回事务的超时时间， 默认值为 -1。如果超过该时间限制但事务还没有完成，则自动回滚事务。
24    int getTimeout();
25    // 返回是否为只读事务， 默认值为 false
26    boolean isReadOnly();
27
28    @Nullable
29    String getName();
30 }
```

### 3、事务状态：`TransactionStatus`

`TransactionStatus` 接口用来记录事务的状态 该接口定义了一组方法,用来获取或判断事务的相应状态信息。

`PlatformTransactionManager.getTransaction(...)` 方法返回一个 `TransactionStatus` 对象。

`TransactionStatus` 接口内容如下：

```
1 public interface TransactionStatus{
2     boolean isNewTransaction(); // 是否是新的事务
3     boolean hasSavepoint(); // 是否有恢复点
4     void setRollbackOnly(); // 设置为只回滚
5     boolean isRollbackOnly(); // 是否为只回滚
6     boolean isCompleted(); // 是否已完成
7 }
```

## 事务传播行为

```
1  @Service
2  Class A {
3      @Autowired
4      B b;
5      @Transactional(propagation = Propagation.xxx)
6      public void aMethod {
7          //do something
8          b.bMethod();
9      }
10 }
11
12 @Service
13 Class B {
14     @Transactional(propagation = Propagation.xxx)
15     public void bMethod {
16         //do something
17     }
18 }
```

### 1. TransactionDefinition.PROPAGATION\_REQUIRED

使用的最多的一个事务传播行为，我们平时经常使用的@**Transactional**注解默认使用就是这个事务传播行为。如果当前存在事务，则加入该事务；如果当前没有事务，则创建一个新的事务。也就是说：

- 如果外部方法没有开启事务的话，Propagation.REQUIRED修饰的内部方法会新开启自己的事务，且开启的事务相互独立，互不干扰。
- 如果外部方法开启事务并且被Propagation.REQUIRED的话，所有Propagation.REQUIRED修饰的内部方法和外部方法均属于同一事务，只要一个方法回滚，整个事务均回滚。

举个例子：如果我们上面的**aMethod()** 和 **bMethod()** 使用的都是**PROPAGATION\_REQUIRED**传播行为的话，两者使用的就是同一个事务，只要其中一个方法回滚，整个事务均回滚。

### 2. TransactionDefinition.PROPAGATIONQUIRES\_NEW

创建一个新的事务，如果当前存在事务，则把当前事务挂起。也就是说不管外部方法是否开启事务，Propagation.REQUIRES\_NEW修饰的内部方法会新开启自己的事务，且开启的事务相互独立，互不干扰。

举个例子：如果我们上面的**bMethod()** 使用 PROPAGATION\_REQUIRES\_NEW 事务传播行为修饰，**aMethod**还是用 PROPAGATION\_REQUIRED 修饰的话。如果 **aMethod()**发生异常回滚，**bMethod()**不会跟着回滚，因为 **bMethod()** 开启了独立的事务。但是，如果 **bMethod()** 抛出了未被捕获的异常并且这个异常满足事务回滚规则的话，**aMethod()** 同样也会回滚，因为这个异常被 **aMethod()** 的事务管理机制检测到了。

### 3. TransactionDefinition.PROPAGATION\_NESTED:

如果当前存在事务，就在嵌套事务内执行；如果当前没有事务，就执行与 TransactionDefinition.PROPAGATION\_REQUIRED 类似的操作。也就是说：

- 在外部方法开启事务的情况下，在内部开启一个新的事务，作为嵌套事务存在。
- 如果外部方法无事务，则单独开启一个事务，与 PROPAGATION\_REQUIRED

类似。

这里还是简单举个例子：如果 bMethod() 回滚的话，aMethod() 不会回滚。如果 aMethod() 回滚的话，bMethod() 会回滚。

#### 4. TransactionDefinition.PROPAGATION\_MANDATORY

如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常。  
(mandatory：强制性)

这个使用的很少，就不举例子来说了。

5.若是错误的配置以下3种事务传播行为，事务将不会发生回滚，这里不对照案例讲解了，使用的很少。

- **TransactionDefinition.PROPAGATION\_SUPPORTS**: 如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行。
- **TransactionDefinition.PROPAGATION\_NOT\_SUPPORTED**: 以非事务方式运行，如果当前存在事务，则把当前事务挂起。
- **TransactionDefinition.PROPAGATION\_NEVER**: 以非事务方式运行，如果当前存在事务，则抛出异常。

## 事务只读性

```
1 package org.springframework.transaction;
2
3 import org.springframework.lang.Nullable;
4
5 public interface TransactionDefinition {
6     .....
7     // 返回是否为只读事务，默认值为 false
8     boolean isReadOnly();
9
10 }
```

java

对于只有读取数据查询的事务，可以指定事务类型为 `readonly`，即只读事务。只读事务不涉及数据的修改，数据库会提供一些优化手段，适合用在有多条数据库查询操作的方法中。

MySQL 默认对每一个新建立的连接都启用了 `autocommit` 模式。在该模式下，每一个发送到 MySQL 服务器的 sql 语句都会在一个单独的事务中进行处理，执行结束后会自动提交事务，并开启一个新的事务。

但是，如果你给方法加上了 **Transactional** 注解的话，这个方法执行的所有 `sql` 会被放在一个事务中。如果声明了只读事务的话，数据库就会去优化它的执行，并不会带来其他的什么收益。

如果不加 **Transactional**，每条 `sql` 会开启一个单独的事务，中间被其它事务改了数据，都会实时读取到最新值。

分享一下关于事务只读属性，其他人的解答：

- 如果你一次执行单条查询语句，则没有必要启用事务支持，数据库默认支持 SQL 执行期间的读一致性；
- 如果你一次执行多条查询语句，例如统计查询，报表查询，在这种场景下，多条查询 SQL 必须保证整体的读一致性，否则，在前条 SQL 查询之后，后条 SQL 查询之前，数据被其他用户改变，则该次整体的统计查询将会出现读数据不一致

的状态，此时，应该启用事务支持

## @Transactional 注解使用详解

### @Transactional 的作用范围

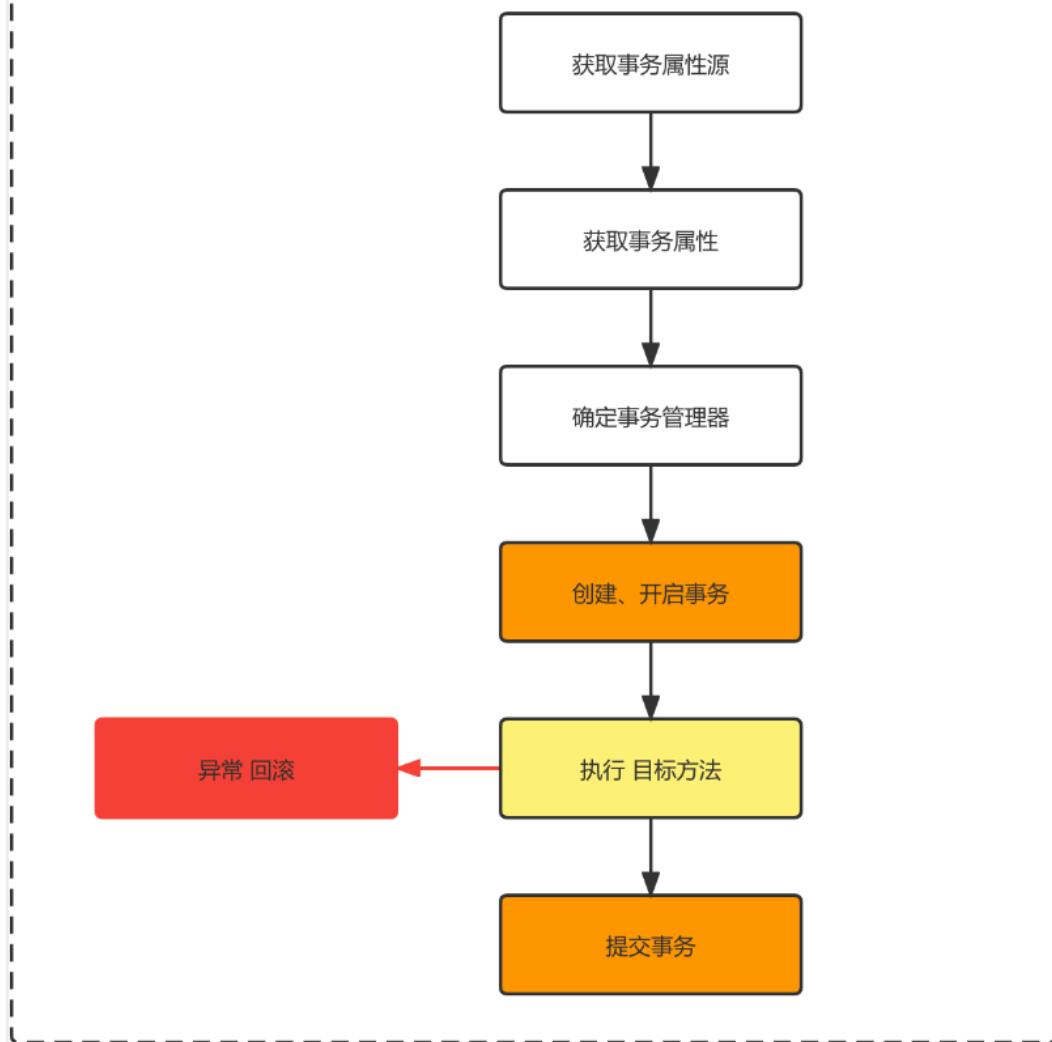
1. 方法：推荐将注解使用于方法上，不过需要注意的是：**该注解只能应用到 public 方法上，否则不生效。**
2. 类：如果这个注解使用在类上的话，表明该注解对该类中所有的 public 方法都生效。
3. 接口：不推荐在接口上使用。

## @Transactional 事务注解原理

如果一个类或者一个类中的 public 方法上被标注 @Transactional 注解的话，Spring 容器就会在启动的时候为其创建一个代理类 **【createAopProxy() 方法决定了是使用 JDK 还是 Cglib 来做动态代理】**，在调用被 @Transactional 注解的 public 方法的时候，实际调用的是，TransactionInterceptor 类中的 invoke() 方法。这个方法的作用就是在目标方法之前开启事务，方法执行过程中如果遇到异常的时候回滚事务，方法调用完成之后提交事务。

TransactionInterceptor 类中的 invoke() 方法内部实际调用的是 TransactionAspectSupport 类的 invokeWithinTransaction() 方法。

### TransactionAspectSupport#invokeWithinTransaction 事务执行流程



## Spring AOP 自调用问题

当一个方法被标记了`@Transactional`注解的时候，Spring 事务管理器只会在被其他类方法调用的时候生效，而不会在一个类中方法调用生效。

这是因为 Spring AOP 工作原理决定的。因为 Spring AOP 使用动态代理来实现事务的管理，它会在运行的时候为带有`@Transactional`注解的方法生成代理对象，并在方法调用的前后应用事物逻辑。如果该方法被其他类调用我们的代理对象就会拦截方法调用并处理事务。但是在一个类中的其他方法内部调用的时候，我们代理对象就无法拦截到这个内部调用，因此事务也就失效了。

## Spring 扩展点使用

### 1、获取 Spring 容器对象

实现`ApplicationContextAware`或`BeanFactoryAware`，获取bean对象。

```

    */
    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
        loginTypeConfig.getTypes().forEach((k, y) -> {
            granterPool.put(k, (UserGranter) applicationContext.getBean(y));
        });
    }

    /**
     * 对外提供获取具体策略
     *
     * @param grantType 用户的登录方式, 需要跟配置文件中匹配
     * @return 具体策略
     */
    public UserGranter getGranter(String grantType) {
        UserGranter tokenGranter = granterPool.get(grantType);
        return tokenGranter;
    }
}

```

场景应用：C端执行策略模式的时候，在项目启动时用 ApplicationContextAware 获取到所有的 handler 对象放入到 map 里，之后执行策略要获取具体的 handler 时到 map 去获取 bean。

## 2、发布-订阅模式

通过自定义事件类继承 ApplicationEvent，自定义监听器类实现 ApplicationListener，然后用 ApplicationContext 发布事件，监听器类就能接收到消息。

原理：通过 SimpleApplicationEventMulticaster 监听器获取所有实现了 ApplicationListener 自定义监听器，当发布事件的时候，Spring 会在 doInvokeListener 方法调用自定义监听器重写的 onApplicationEvent 方法实现事件的发布【这一切都在 refresh() 方法实现了】。

场景应用：当创建标签成功时会通过监听器的事件发布，监听到事件的发布后会创建飞书审批，达到解耦提高系统运行性能【毕竟创建飞书审批的一堆逻辑也需要占用执行性能，通过监听器可以解耦提高创标接口执行效率】

## 3、通过 AOP 代理做接口调用时的用户权限校验

---



---

# Spring&SpringBoot 常用注解总结

## @SpringBootApplication

@Configuration、@EnableAutoConfiguration、@ComponentScan 注解的集合。  
三个注解分别作用

- @EnableAutoConfiguration：SpringBoot 的自动装配机制  
【@EnableAutoConfiguration 注解告诉 Spring Boot 根据你添加的 jar 依赖自动配置你的 Spring 应用程序。例如，如果你的 pom.xml 文件中添加了 Spring Data JPA 的依赖，Spring Boot 会自动配置与数据库相关的所有设置，如数据源、事务管理器、实体管理器工厂等，而无需你手动配置】
- @ComponentScan：扫描被 @Component

(@Repository,@Service,@Controller)注解的 bean，注解默认会扫描该类所在的包下所有的类。

- @Configuration：一般用来声明配置类，可以使用 @Component注解替代，不过使用 @Configuration注解声明配置类更加语义化。本质上也是将对象交由 Spring管理，只不过更专注于配置类，并且通过@Bean去创建和管理 Bean

## @Bean详解

**@Bean**是一个注解，用于告诉Spring框架将标注的方法返回的对象注册为一个Bean（组件）。

具体来说，@Bean注解可以用于方法上，方法返回的对象将被Spring容器管理，并且提供给其他程序组件使用。此外，@Bean通常与@Configuration一起使用，@Configuration用于标记一个Java类为Spring配置类，可以包含@Bean注解的方法，这些方法返回的对象将被Spring容器管理。

@Bean注解的作用如下：

- Bean注册。使用@Bean注解可以将方法返回的对象注册为一个Bean，并且该Bean会被Spring容器管理。
- 依赖注入。当其他组件需要使用这个Bean时，Spring框架会自动将该Bean注入到相应的位置，实现依赖注入。
- 自定义组件配置。通过@Bean注解可以对Bean进行自定义配置，例如设置属性值、初始化方法、销毁方法等。
- 替代XML配置。在过去，需要通过XML配置文件来定义和配置Bean，而现在通过@Bean注解可以在Java代码中实现同样的功能，避免了繁琐的XML配置。

## @Value(常用)

使用 @Value("\${property}") 读取比较简单的配置信息

## @ConfigurationProperties(常用)

它允许你将外部配置文件（如 application.properties 或 application.yml）中的属性绑定到一个Java对象上。这个注解通常与@Component或@Configuration一起使用，以便让Spring容器能够管理这个对象，并将其作为Bean注入到其他组件中。

要使用@ConfigurationProperties，你需要按照以下步骤操作：

- 1、创建一个Java类，用于保存配置属性。这个类通常使用@Component或@Configuration注解标记为Spring组件。
- 2、在这个类上添加@ConfigurationProperties注解，并指定需要绑定的属性文件的前缀。例如，如果你的属性文件中有以my.custom为前缀的属性，你可以这样指定：

```
@Component  
@ConfigurationProperties(prefix = "my.custom")  
public class MyCustomProperties {  
    private String property1;  
    private int property2;  
    // getters and setters  
}
```

java

3、在属性文件中添加以指定前缀开头的属性。例如：

```
my.custom.property1=value1  
my.custom.property2=123
```

4、在需要使用这些属性的地方，通过@Autowired或@Inject将MyCustomProperties注入进来。

```
@Service  
public class MyService {  
    private final MyCustomProperties myCustomProperties;  
  
    @Autowired  
    public MyService(MyCustomProperties myCustomProperties) {  
        this.myCustomProperties = myCustomProperties;  
    }  
  
    public void doSomething() {  
        String value1 = myCustomProperties.getProperty1();  
        int value2 = myCustomProperties.getProperty2();  
        // ...  
    }  
}
```

java

5、在启动类上添加@EnableConfigurationProperties注解，以便让Spring Boot知道需要扫描这个类。

```
@SpringBootApplication  
@EnableConfigurationProperties(MyCustomProperties.class)  
public class MyApplication {  
    public static void main(String[] args) {  
        SpringApplication.run(MyApplication.class, args);  
    }  
}
```

java

通过以上步骤，Spring Boot会自动将配置文件中的属性绑定到MyCustomProperties类的对应字段上。当应用程序启动时，MyCustomProperties对象会被创建并注入到需要它的地方。

需要注意的是，@ConfigurationProperties还支持嵌套属性、集合属性等复杂类型的绑定，可以灵活处理各种配置场景。此外，它还提供了验证和回调等功能，可以在属性绑定后进行额外的处理或验证。

# SpringBoot 自动装配原理详解

## 什么是自动装配？

通过注解或者一些简单的配置就能在 Spring Boot 的帮助下实现某块功能。

## SpringBoot 是如何实现自动装配的？

即 @EnableAutoConfiguration, @EnableAutoConfiguration 注解告诉 Spring Boot 根据你添加的 jar 依赖自动配置你的 Spring 应用程序。例如，如果你的 pom.xml 文件中添加了 Spring Data JPA 的依赖，Spring Boot 会自动配置与数据库相关的所有设置，如数据源、事务管理器、实体管理器工厂等，而无需你手动配置

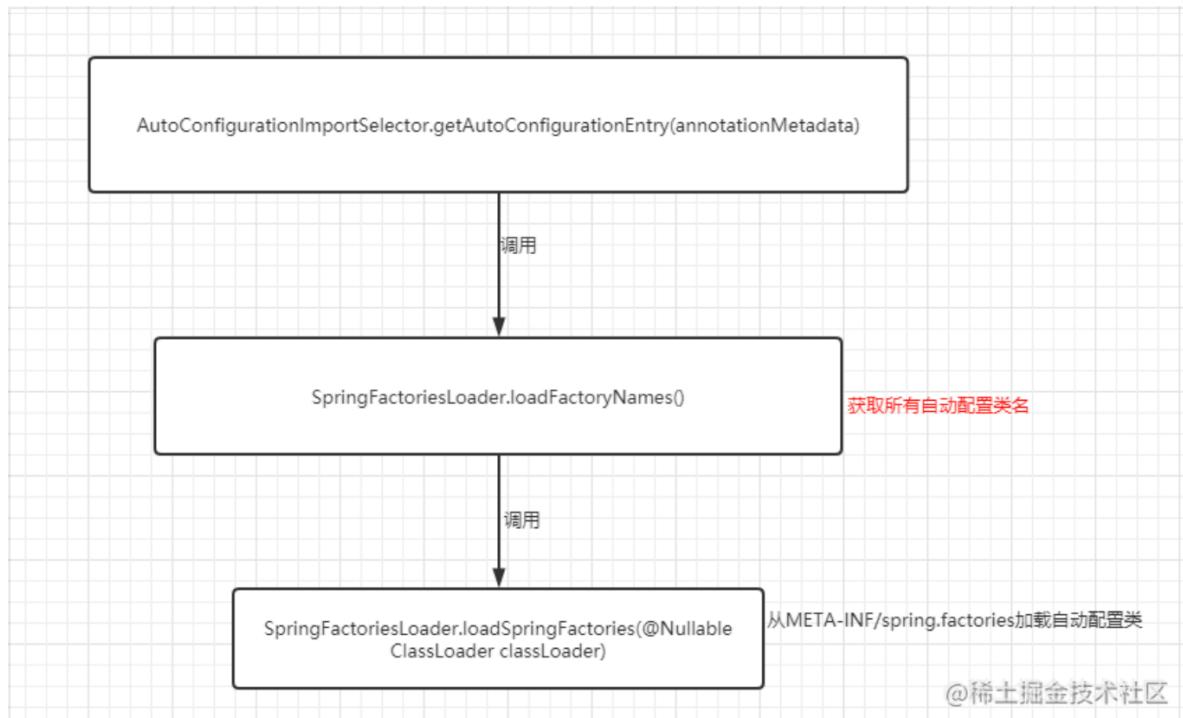
**EnableAutoConfiguration** 只是一个简单地注解，自动装配核心功能的实现实际是通过 **AutoConfigurationImportSelector** 类。

## AutoConfigurationImportSelector: 加载自动装配类

AutoConfigurationImportSelector 类实现了 ImportSelector 接口，也就实现了这个接口中的 selectImports 方法，该方法主要用于获取所有符合条件的类的全限定类名，这些类需要被加载到 IoC 容器中。

主要关注下 selectImports 方法中 getAutoConfigurationEntry() 方法，这个方法主要负责加载自动配置类的。

调用链路如下：



现在我们结合 `getAutoConfigurationEntry()` 的源码来详细分析一下：

```
1 private static final AutoConfigurationEntry EMPTY_ENTRY = new AutoConfigurationEntry();
2
3 AutoConfigurationEntry getAutoConfigurationEntry(AutoConfigurationMetadata autoConfigur
4     //<1>
5     if (!this.isEnabled(annotationMetadata)) {
6         return EMPTY_ENTRY;
7     } else {
8         //<2>
9         AnnotationAttributes attributes = this.getAttributes(annotationMetadata);
10        //<3>
11        List<String> configurations = this.getCandidateConfigurations(annotationMet
12        //<4>
13        configurations = this.removeDuplicates(configurations);
14        Set<String> exclusions = this.getExclusions(annotationMetadata, attributes);
15        this.checkExcludedClasses(configurations, exclusions);
16        configurations.removeAll(exclusions);
17        configurations = this.filter(configurations, autoConfigurationMetadata);
18        this.fireAutoConfigurationImportEvents(configurations, exclusions);
19        return new AutoConfigurationImportSelector.AutoConfigurationEntry(configurat
20    }
21 }
```

- 1、判断自动装配开关是否打开。默认  
spring.boot.enableautoconfiguration=true，可在 application.properties 或 application.yml 中设置
- 2、用于获取EnableAutoConfiguration注解中的 exclude【通过指定的 Class 排除某个需要自动装配的类】和 excludeName【通过指定的完全限定的类名排除某个需要自动装配的类】。
- 3、获取需要自动装配的所有配置类，读取 META-INF/spring.factories，所有 Spring Boot Starter 下的 META-INF/spring.factories 都会被读取到。
- 4、对 META-INF/spring.factories 下的配置文件内容按需加载，通过  
@ConditionalOnXXX 中的所有条件都满足，该类才会被加载进来。  
【@ConditionalOnXXX 结尾的注解，是用于条件化地创建beans的。这意味着，根据某些条件，Spring Boot 可以决定是否要创建和注册某个bean到 Spring 容器中。】例子如下

## 1. 使用@ConditionalOnClass

假设我们有一个自定义的starter，我们希望只有在类路径中存在某个特定的类（例如MyCustomClass）时，才自动配置某个bean。

```
@Configuration  
public class MyAutoConfiguration {  
  
    @Bean  
    @ConditionalOnClass(MyCustomClass.class)  
    public MyBean myBean() {  
        return new MyBean();  
    }  
}
```

在这个例子中，如果MyCustomClass在类路径中不存在，那么myBean()方法将不会被调用，因此MyBean的bean也不会被创建。

## 2. 使用@ConditionalOnProperty

我们可能希望根据配置文件中的某个属性来决定是否创建bean。

```
@Configuration  
public class MyAutoConfiguration {  
  
    @Bean  
    @ConditionalOnProperty(name = "my.feature.enabled", havingValue = "true")  
    public MyFeatureBean myFeatureBean() {  
        return new MyFeatureBean();  
    }  
}
```

在application.properties文件中：

```
my.feature.enabled=true
```

如果my.feature.enabled的值是true，那么myFeatureBean()方法会被调用，并且MyFeatureBean的bean会被创建。如果值是false或者属性不存在，bean将不会被创建。

### 3. 使用@ConditionalOnMissingBean

我们可以使用@ConditionalOnMissingBean来确保只有当Spring容器中不存在某个类型的bean时，才创建我们的bean。

```
@Configuration  
public class MyAutoConfiguration {  
  
    @Bean  
    @ConditionalOnMissingBean  
    public DefaultService defaultService() {  
        return new DefaultService();  
    }  
}
```

如果开发者已经在其他地方定义了一个DefaultService类型的bean，那么defaultService()方法将不会被调用，因为Spring会优先使用已经存在的bean。

## 自动装配总结

### @EnableAutoConfiguration 自动装配机制总结

Spring Boot 的 @EnableAutoConfiguration 是其“约定优于配置”理念的核心，它能够根据项目类路径 (Classpath) 中的依赖，自动地、有条件地配置 Spring 应用程序。整个过程可以概括为以下几个步骤：

#### 1. 启动与入口

- 我们的启动类上通常标注有 @SpringBootApplication 注解，这是一个复合注解，其中就包含了 @EnableAutoConfiguration。这个注解是自动装配的总开关。
- @EnableAutoConfiguration 注解内部通过 @Import(AutoConfigurationImportSelector.class) 引入了一个关键的选择器。

#### 2. 扫描与加载配置清单

- 当 Spring 容器启动时，AutoConfigurationImportSelector 会被触发。
- 它的核心任务是去扫描所有引入的 JAR 包的类路径，查找一个固定的文件：META-INF/spring.factories。（在 Spring Boot 2.7 及以后，会优先查找新的 META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports 文件）。
- 这个文件是一个标准的 Properties 文件，其中 org.springframework.boot.autoconfigure.EnableAutoConfiguration 这个键 (Key) 下列出了所有官方和第三方提供的自动配置类 (Auto-Configuration Class) 的全限定名列表。

#### 3. 过滤与条件化装配 (最精髓的部分)

- AutoConfigurationImportSelector 将从清单文件中读取到的所有配置类名，加载到内存中。
- 但它并不会立即实例化所有这些配置类。相反，Spring Boot 会对这些配置类进行一轮精细的过滤和筛选。
- 每个自动配置类 (例如

DataSourceAutoConfiguration, JpaRepositoriesAutoConfiguration) 的头部, 都使用了大量的 @ConditionalOn...注解, 例如:

- **@ConditionalOnClass**: 只有当类路径中存在某个特定的类时 (比如 javax.persistence.EntityManager), 这个配置才可能生效。这解释了为什么我们添加了 spring-boot-starter-data-jpa 依赖后, JPA 的自动配置才会被触发。
- **@ConditionalOnBean**: 当容器中已存在某个 Bean 时, 或不存在某个 Bean 时, 配置才生效。
- **@ConditionalOnProperty**: 允许我们通过在 application.properties 中设置属性 (如 spring.jpa.hibernate.ddl-auto) 来启用、禁用或改变自动配置的行为。
- **@ConditionalOnMissingBean**: 这是最常见的一个, 表示“如果用户自己没有定义同类型的 Bean, 那么我才提供一个默认的”。这给予了开发者极大的灵活性, 可以随时覆盖掉 Spring Boot 的默认配置。

#### 4. Bean的创建与注入

- 只有通过了所有 @Conditional 条件检查的自动配置类, 才会被 Spring 容器正式处理。
- 接着, Spring 会执行这些配置类中被 @Bean 注解标记的方法。
- 这些方法会创建出预先配置好的实例 (如 DataSource, EntityManagerFactory, JdbcTemplate 等), 并将它们注册到 IOC 容器中。

#### 总结

@EnableAutoConfiguration 的本质是一个智能的、按需加载的配置机制。它通过扫描依赖 (决定可能性)、读取清单 (找到配置类)、\*\*条件化判断 (决定是否生效) 和实例化 Bean (完成配置)\*\*这一整套流程, 极大地简化了 Spring 应用的搭建, 让开发者可以“开箱即用”, 同时保留了完全的自定义和覆盖能力。

## spring.factories 和 pom.xml 关系

### ○ 第一阶段: 构建时 (Build Time) - pom.xml 的舞台

这个阶段发生在您执行 mvn package、mvn install 或者在 IDE 中点击“构建/运行”按钮的时候。

1. 主角: Maven (或 Gradle 等构建工具)。

2. 核心任务:

- 解析 pom.xml: Maven 会读取您项目以及所有父项目的 pom.xml 文件。
- 依赖解析: 根据 pom.xml 中的<dependencies>和<dependencyManagement>, Maven 会计算出一个最终的、确定的依赖树。
- 下载依赖: Maven 会从远程仓库 (如 Maven Central) 或本地仓库下载所有需要的 JAR 包。
- 构建类路径 (Classpath): 这是最关键的一步。Maven 会将所有下载好的依赖 JAR 包, 以及您自己项目的编译结果 (target/classes 目录), 共同组成一个最终的类路径。这个类路径会传递给 Java 虚拟机 (JVM) 去执行。

简单来说, pom.xml 的作用是决定“哪些 JAR 包最终会出现在运行时的类路径上”。它的使命在程序真正运行之前就已经完成了。

## ○ 第二阶段：运行时 (Runtime) - `spring.factories` 的舞台

这个阶段发生在您的 Spring Boot 应用程序的 main 方法被 JVM 执行之后。

1. 主角: **Spring Boot 框架本身** (特别是 AutoConfigurationImportSelector)。
2. 核心任务:
  - 扫描类路径: 此时, Spring Boot 已经拿到了由 Maven 在第一阶段准备好的那个完整的类路径。
  - 查找特定文件: AutoConfigurationImportSelector 的任务非常专一, 它就是要在这个类路径下的所有位置 (包括您自己项目的 target/classes 和所有依赖的 JAR 包内部), 去寻找一个固定路径和名称的文件: META-INF/spring.factories (以及新的.imports 文件)。
  - 读取和处理: 它会打开所有找到的 spring.factories 文件, 读取其中的内容, 并加载 EnableAutoConfiguration 键下指定的自动配置类。

**我们可以用一个比喻来理解:**

- **pom.xml**: 就像是去图书馆借书的借书单。你写好单子, 图书管理员 (Maven) 就会根据单子把所有你需要的书 (JAR 包) 从书架上拿下来, 堆在你面前。
- **spring.factories**: 就像是每一本书 (JAR 包) 的目录页。你 (Spring Boot) 拿到一堆书之后, 不会去关心借书单了, 而是会翻开每一本书的目录 (spring.factories), 看看里面有哪些“有趣的章节” (自动配置类) 值得一读。

## ○ 总结: 流程的正确顺序

1. 构建时: **Maven** 读取 `pom.xml` -> 计算依赖 -> 构建 类路径。
2. 运行时: **Spring Boot** 启动 -> AutoConfigurationImportSelector 扫描 类路径 -> 查找并读取所有 JAR 包中的 **META-INF/spring.factories** -> 加载自动配置类。

所以, 您的提问非常棒, 它帮助我们澄清了:

- **pom.xml** 是给构建工具看的, 决定了“有什么”。
- **spring.factories** 是给 **Spring Boot** 框架看的, 决定了在已有的东西里“做什么”。

Spring Boot 的自动装配之所以如此智能, 正是因为它巧妙地利用了 Maven (或 Gradle) 构建好的类路径环境。它通过检查类路径中\*\*“是否存在某个类” (这是由 **pom.xml** 决定的), 来有条件地触发\*\*`spring.factories` 中定义的自动配置。两者是分工协作、前后衔接的关系。

---

---

# SpringBoot 的启动流程

我们可以将整个启动流程更清晰地划分为三个主要阶段: 初始化准备阶段、Spring 容器创建与刷新阶段、应用运行阶段。

## 第一阶段: 初始化准备 (执行 `SpringApplication.run()` 的前期工作)

这是 `new SpringApplication(...)` 和 `run()` 方法被调用后, 但在创建 Spring 容器之前发生的事情。

### 1. 创建 `SpringApplication` 实例

- 当你执行 `SpringApplication.run(MyApplication.class, args)` 时, 首先会创建一个 `SpringApplication` 对象。

- 在这个构造函数中，Spring Boot 会做一些重要的准备工作：
  - ◆ 推断应用类型（是 SERVLET Web 应用、REACTIVE Web 应用，还是 NONE 非 Web 应用）。
  - ◆ 加载 `spring.factories` 文件中配置的 `ApplicationContextInitializer` 和 `ApplicationListener`，这是 Spring Boot 早期扩展机制的关键。

## 2. 启动 RunListeners 并准备环境

- `run()` 方法开始执行，它会发布一个 `ApplicationStartingEvent` 事件。
- 接着，它会创建并准备 **Environment** 对象。这个 **Environment** 对象是所有配置信息的汇集地。
- **加载配置文件**：它会加载所有 `application.properties` 或 `application.yml` 文件（包括 `application-{profile}.yml` 等），以及命令行参数、系统属性、环境变量等，并将它们统一放入 `Environment` 中。

## 第二阶段：创建并刷新 Spring 容器（核心流程）

这是整个启动流程的心脏地带，真正创建 IoC 容器并装配 Bean。

### 1. 创建 ApplicationContext (Spring 容器)

- 根据第一阶段推断出的应用类型，Spring Boot 会创建一个具体的 `ApplicationContext` 实例。
- 如果是 Web 应用，通常会创建 `AnnotationConfigServletWebServerApplicationContext`。

### 2. 准备并刷新容器（最复杂的一步）

- Spring Boot 会调用 `context.refresh()` 方法，这会触发 Spring 框架本身的标准 IoC 容器启动流程。这个庞大的 `refresh()` 方法内部，包含了您提到的“注解扫描”和“加载自动配置”。
- **a) 调用 BeanFactoryPostProcessor**: 这是 Spring 的扩展点。Spring Boot 的\*\*自动配置 (Auto-Configuration)\*\*机制正是在这里被触发的。
- **b) 加载自动配置 (@EnableAutoConfiguration)**:
  - ◆ 一个名为 `ConfigurationClassPostProcessor` 的后处理器会被执行。
  - ◆ 它会处理启动类上的 `@EnableAutoConfiguration` 注解。
  - ◆ 该注解通过 `@Import(AutoConfigurationImportSelector.class)`，会去读取所有 starter 依赖包中 `META-INF/spring.factories` 或新版 `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports` 文件里定义的自动配置类 (`XXXAutoConfiguration`)。
  - ◆ Spring Boot 会根据当前 classpath 中存在的类、`Environment` 中的配置、以及 `@ConditionalOn...` 等条件注解，来决定哪些自动配置类需要被加载。
- **c) 注解扫描 (@ComponentScan)**:
  - ◆ `ConfigurationClassPostProcessor` 同样会处理启动类上的 `@ComponentScan` 注解。
  - ◆ 它会扫描您自己编写的、被 `@Component`, `@Service`, `@Controller` 等注解标记的类。
  - ◆ 对于扫描到的自动配置类和您自己的业务类，Spring 都会将它们解析并注册为\*\*BeanDefinition\*\* (Bean 的“蓝图”) 到容器中。
- **d) 实例化非懒加载的单例 Bean**:
  - ◆ 在 `refresh()` 方法的最后，容器会遍历所有 `BeanDefinition`，并实例化

所有非懒加载的单例 Bean。这个过程包括了依赖注入、初始化等步骤。

### 第三阶段：应用运行

#### 1. 调用 ApplicationRunner 和 CommandLineRunner

- 在容器刷新完毕后，Spring Boot 会查找容器中所有实现了 ApplicationRunner 或 CommandLineRunner 接口的 Bean，并按顺序调用它们的 run() 方法。这为你提供了一个在应用启动后、开始接受请求前，执行自定义初始化代码的绝佳时机。

#### 2. 启动 Web 服务器并接受请求

- 至此，一切准备就绪。
- 如果是 Web 应用，内嵌的 Web 服务器（如 Tomcat）会被完全启动，并开始监听端口，准备接受外部请求。
- 应用正式进入运行状态。

## ○ 总结与完善后的流程图

### 1. SpringApplication.run() 启动

- 创建 SpringApplication 实例，推断应用类型，加载初始化器和监听器。

### 2. 准备 Environment

- 加载所有配置文件（application.properties 等）到 Environment 中。

### 3. 创建 ApplicationContext

- 根据应用类型创建对应的 Spring 容器实例。

### 4. context.refresh() 容器刷新（核心）

- 触发自动配置：@EnableAutoConfiguration 加载所有满足条件的 XXXAutoConfiguration 类。
- 执行组件扫描：@ComponentScan 扫描你自己的 @Component 等业务类。
- 注册 BeanDefinition：将上述两步找到的所有类，解析成 Bean 的“蓝图”并注册。
- 实例化 Bean：创建所有非懒加载的单例 Bean 实例（包括依赖注入、初始化）。

### 5. 执行 Runners

- 调用所有 ApplicationRunner 和 CommandLineRunner。

### 6. 应用就绪

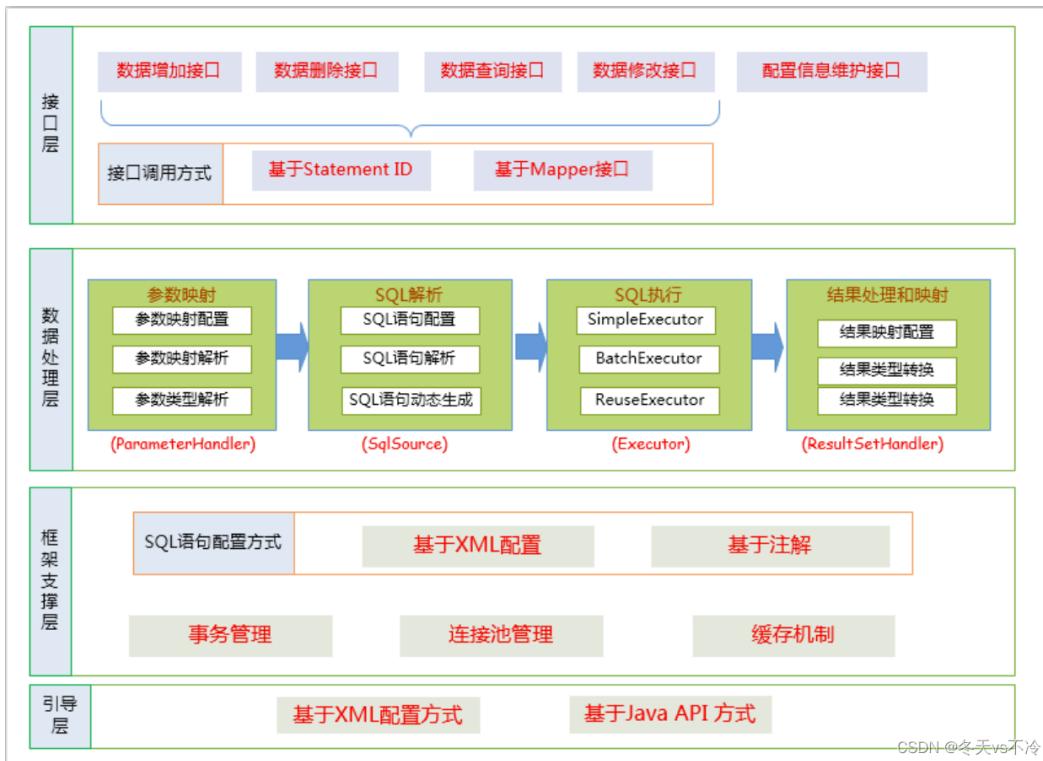
- （如果是 Web 应用）启动内嵌 Web 服务器，开始接受请求。

---

## MyBatis

Mybatis 源码博客系列：[https://blog.csdn.net/qq\\_35512802/article/details/127592740](https://blog.csdn.net/qq_35512802/article/details/127592740)

## 架构设计



Mybatis 架构四层作用：

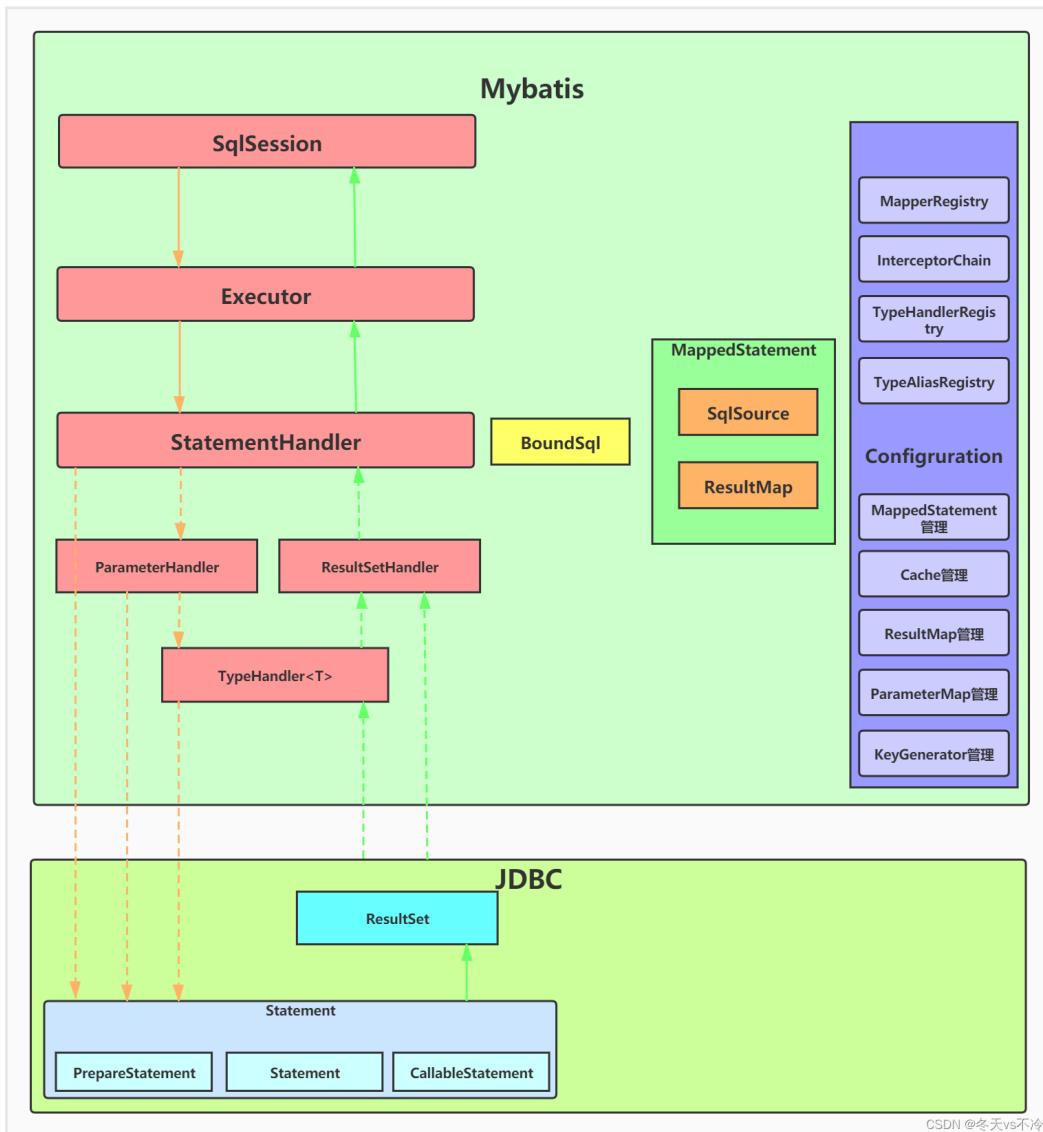
Api接口层：提供 API 增加、删除、修改、查询等接口，通过 API 接口对数据库进行操作。

数据处理层：主要负责 sql 的查询、解析、执行以及结果映射的处理，主要作用解析 sql 根据调用请求完成一次数据库操作

框架支撑层：负责通用基础服务支撑，包含事务管理、连接池管理、缓存管理等公用组件的封装，为上层提供基础服务支撑

引导层：引导层是配置和启动 MyBatis 配置信息的方法

## MyBatis 主要组件及其相互关系



CSDN @冬天vs不冷

## 组件介绍

类名定义	角色定位	分工协作
Resources	资源辅助类	负责读取配置文件转化为输入流
Configuration	数据库资源类	负责存储数据库连接信息
MappedStatement	SQL与结果集资源类	负责存储SQL映射定义、存储结果集映射定义
SqlSessionFactoryBuilder	会话工厂构建者	负责解析配置文件，创建会话工厂 SqlSessionFactory
SqlSessionFactory	会话工厂	负责创建会话 SqlSession
SqlSession	会话	指派执行器 Executor
Executor	执行器	负责执行SQL (配合指定资源 Mapped Statement)

**SqlSession:** 是 Mybatis 对外暴露的核心 API，提供的对数据库的 CRUD 接口操作  
**Executor:** 执行器，由 SqlSession 调用，负责数据库操作以及 Mybatis 两级缓存的维护

**StatementHandler:** 封装了 JDBC Statement 操作，负责对 Statement 的操作，如 PreparedStatement 参数的设置以及结果集的处理

**ParameterHandler:** 是 StatementHandler 内部一个组件，主要负责对 ParameterStatement 参数的设置

**ResultSetHandler:** 是 StatementHandler 内部一个组件，主要负责对 ResultSet 结

果集的处理，封装成目标对象返回

TypeHandler：用于Java类型与JDBC类型之间的数据转换，ParameterHandler和ResultSetHandler会分别使用到它的类型转换功能

MappedStatement：是对Mapper配置文件或Mapper接口方法上通过注解申明sql的封装

Configuration：Mybatis所有配置都统一由Configuration进行管理，内部由具体对象分别管理各自的小功能模块

## MyBatis 源码解析

```
/*  
 * 测试方法：传统方式  
 */  
  
@Test  
public void test1() throws IOException {  
    // 1. 通过类加载器对配置文件进行加载，加载成了字节输入流，存到内存中 注意：配置文件并没有被解析  
    InputStream resourceAsStream = Resources.getResourceAsStream("sqlMapConfig.xml");  
  
    // 2. (1)解析了配置文件，封装configuration对象 (2) 创建了DefaultSqlSessionFactory工厂对象  
    SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(resourceAsStream);  
  
    // 3. (1)创建事务对象 (2) 创建了执行器对象cachingExecutor (3)创建了DefaultSqlSession对象  
    SqlSession sqlSession = sqlSessionFactory.openSession();  
  
    // 4. 委派给Executor来执行，Executor执行时又会调用很多其他组件（参数设置、解析sql的获取，sql的执行、结果集的封装）  
    User user = sqlSession.selectOne(statement: "user.findUserById", parameter: 100);  
  
    System.out.println(user);  
    System.out.println("MyBatis源码环境搭建成功....");  
  
    sqlSession.close();  
}
```

CSDN @冬天vs不冷

### 1、全局配置文件的解析

加载和解析mybatis-config.xml配置文件，根据文件内容解析封装Configuration对象，再构建工厂会话对象SqlSessionFactory

- Configuration作用：Configuration是一个非常重要的类，它代表了MyBatis的全局配置信息，比如：环境配置、属性设置、缓存配置等等
- SqlSessionFactory作用：用于创建SqlSession实例，表示和数据库的一次会话，提供了执行SQL语句、获取映射器(Mapper)实例等方法【每次需要执行数据库查询或更新时都会创建一个新的SqlSession实例，且SqlSession实例线程不安全】。
- sqlSession关闭时机：如果应用程序使用Spring框架，那么SqlSession的管理通常由Spring负责。在Spring中，SqlSession通常作为方法参数通过@Autowired或@Resource注解注入，并在方法执行完毕后由Spring容器自动管理其生命周期，包括关闭SqlSession。

博客：[https://blog.csdn.net/qq\\_35512802/article/details/127594570](https://blog.csdn.net/qq_35512802/article/details/127594570)

### 2、映射配置文件的解析

AOP动态代理通过解析<package>标签生成mapper对应的代理对象，当sqlSession获取mapper的某个接口时，实际是从mapper的代理对象获取到接口。当遇到mapper里面的<select><insert><update><delete>标签时，实际也是封装

成 MappedStatement 对象去执行 sql 语句的【一个标签生成一个 MappedStatement 对象】。

博客：[https://blog.csdn.net/qq\\_35512802/article/details/127658682](https://blog.csdn.net/qq_35512802/article/details/127658682)

### 3、sql 语句及#{ }、\$ 的解析

在执行 SQL 语句之前，SqlSession 先创建 BoundSql 对象用于①封装解析 sql 语句、②替换 sql 语句的占位符为?、③缓存和重用【BoundSql 可以被缓存起来，然后遇到相同的 sql 执行可以直接拿来使用，避免重复解析 sql 语句】等来生成最终可执行的 sql 语句，然后传递给 StatementHandler 来执行，StatementHandler 会使用 BoundSql 对象中的信息来创建 JDBC Statement 或 PreparedStatement，并设置相应的参数，然后执行 SQL 语句。

博客：[https://blog.csdn.net/qq\\_35512802/article/details/127594155](https://blog.csdn.net/qq_35512802/article/details/127594155)

### 4、sqlSession 会话的创建

通过 SqlSessionFactory.openSession() 创建 sqlSession 对象

博客：[https://blog.csdn.net/qq\\_35512802/article/details/127836774](https://blog.csdn.net/qq_35512802/article/details/127836774)

### 5、缓存执行器操作流程

① sqlSession.selectOne() 里的入参 statement【命名空间 namespace 例如 mapper 包路径 +sql 语句的 ID 即唯一标识符例如 queryById】获取对应的 MappedStatement 对象。

② 从 MappedStatement 对象中获取 BoundSql 对象

MappedStatement 创建&填充数据过程

MappedStatement 对象是在 MyBatis 初始化阶段由配置解析器填充数据的。它封装了映射语句（即 SQL 语句）的全部信息，包括 SQL 语句本身、参数类型、返回类型、结果映射等信息。

关于 MappedStatement 对象的创建和填充数据的过程大致如下

#### 1. 加载配置文件：

当 MyBatis 启动时，它会加载其配置文件（通常是 mybatis-config.xml）以及映射器（Mapper）的 XML 配置文件。这些配置文件定义了数据库操作所需的 SQL 语句、结果映射等信息。

#### 2. 解析配置文件：

MyBatis 使用 XML 解析器来解析这些配置文件，并将解析得到的信息转换为内部数据结构。在这个过程中，MyBatis 会为每个 , , , 和  等元素分别创建一个 MappedStatement 对象。

#### 3. 填充 MappedStatement 对象【一个 SQL 就会创建一个 MappedStatement】：

对于每个 , , , 和  元素，MyBatis 会创建一个 MappedStatement 对象，并设置该对象的属性，如 id（语句的唯一标识符）、sql（SQL 语句本身）、parameterType（参数类型）、resultType（返回类型）、resultMap（结果映射）等。这些属性都是根据 XML 配置文件中的元素属性和子元素来设置的。

#### 4. 构建缓存：

一旦 MappedStatement 对象被创建并填充了数据，它们就会被存储在 MyBatis 的内部缓存中，以便在运行时能够快速查找和重用【只是缓存 sql 信

息，而不是缓存 sql 结果】。

#### 5. 执行阶段：

当应用程序需要执行某个数据库操作时（例如调用 `SqlSession.selectOne()` 或 `SqlSession.insert()` 等方法），MyBatis 会根据操作类型和提供的参数信息，从缓存中查找相应的 `MappedStatement` 对象，并使用这个对象来执行 SQL 语句。

总之，`MappedStatement` 对象在 MyBatis 的初始化阶段通过解析配置文件来填充数据，并在执行阶段被用于指导 SQL 语句的执行。这个过程确保了 MyBatis 能够正确地执行配置文件中定义的数据库操作。

### BoundSql 创建&填充数据过程

`BoundSql` 对象的填充数据是在准备执行 SQL 语句之前进行的。具体来说，当 `SqlSession` 的 `selectList`、`selectOne`、`insert`、`update` 或 `delete` 等方法被调用时，MyBatis 会开始准备执行相应的 SQL 语句，并在这个过程中创建和填充 `BoundSql` 对象。

下面是 `BoundSql` 对象填充数据的大致流程：

#### 1. 解析映射语句：

当 `SqlSession` 的方法被调用时，MyBatis 首先会根据传入的参数和映射语句的 ID（通常是一个命名空间加上 SQL 语句的 ID）来查找对应的 `MappedStatement`。`MappedStatement` 是 MyBatis 中对 SQL 语句的封装，包含了 SQL 语句本身、参数类型、返回类型等信息。

#### 2. 动态 SQL 解析：

如果映射语句包含了动态 SQL 元素（如 `<if>`、`<choose>`、`<when>`、`<otherwise>` 等），MyBatis 会根据传入的参数值来解析这些动态元素，生成最终的 SQL 语句。

#### 3. 参数处理：

MyBatis 会根据 `MappedStatement` 中的参数类型信息来处理传入的参数。这包括将 Java 对象转换为 JDBC 类型的值，以及处理可能存在的参数占位符（如 `#{} 或 ${}`）。

#### 4. 创建 `BoundSql` 对象：

在确定了最终的 SQL 语句和参数值之后，MyBatis 会创建一个 `BoundSql` 对象，并将 SQL 语句和参数信息设置到这个对象中。这包括将参数值替换到 SQL 语句中的占位符，以及将参数值存储在 `BoundSql` 的 `parameterObject` 属性中。

#### 5. 执行 SQL 语句：

最后，`BoundSql` 对象会被传递给 `StatementHandler`，`StatementHandler` 会使用 `BoundSql` 中的信息来创建 JDBC 的 `PreparedStatement` 或 `Statement`，并设置参数值，然后执行 SQL 语句。需要注意的是，`BoundSql` 对象的创建和填充数据是在每次执行 SQL 语句时都会发生的，而不是在 MyBatis 初始化阶段。这是因为 SQL 语句的参数值通常是在执行时才知道的，所以需要在执行前对 `BoundSql` 进行填充。

### `MappedStatement` 和 `BoundSql` 关系

MyBatis 初始化时候会创建 `MappedStatement` 对象【此对象包含了 SQL 语句本身、参数类型、返回类型、结果映射等信息，并且还会将 sql 语句塞入缓存以便运行时快速查找和重用】，然后在执行 sql 语句例如 `selectOne()` 时根据 `nameSpace`&唯一标

识符去获取对应的 MappedStatement 对象，再通过 MappedStatement 对象获取 BoundSql 对象拿到 sql 语句以及参数信息。

### ③ 获取一级或二级缓存 Map 的 key, value 是查询的结果

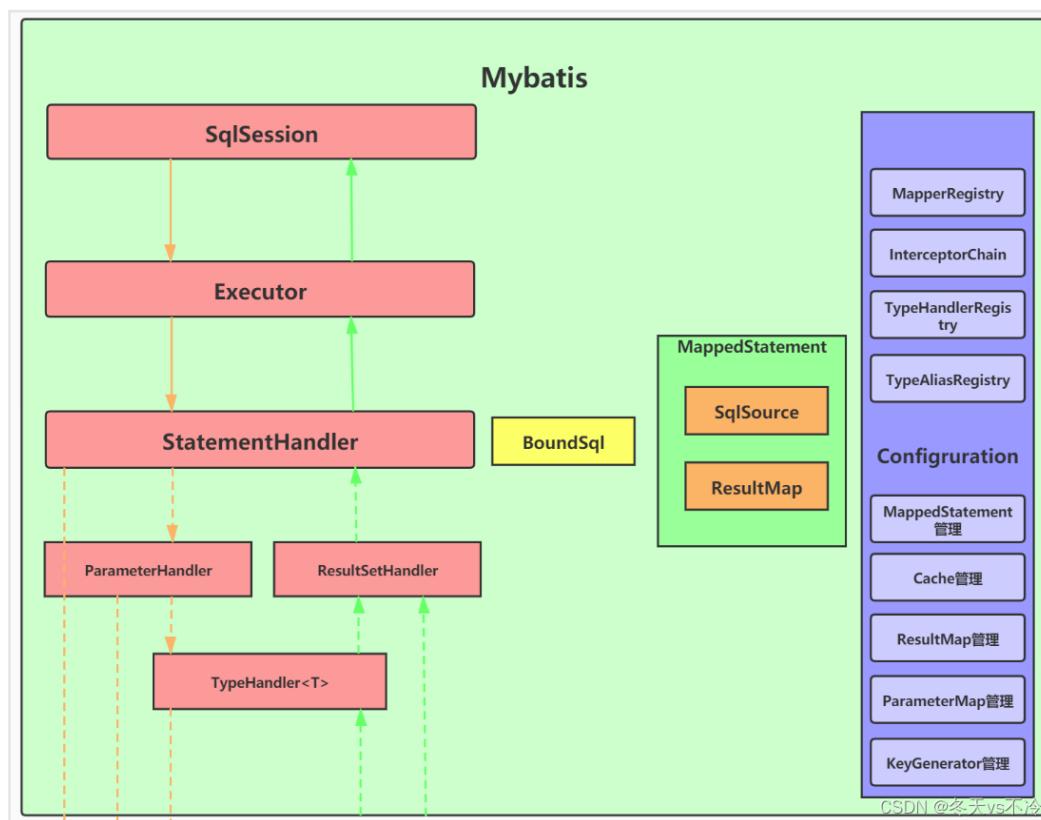
- 触发时机：在 executor 执行 query 操作才会塞入缓存
- 一级缓存的生命周期与 SqlSession 的生命周期相同。当 SqlSession 关闭或执行了任何更新操作（如 insert、update 或 delete）时，一级缓存中的数据会被清空。这是因为一级缓存的设计目的是为了减少相同 SqlSession 中重复查询的开销，而不是为了跨多个 SqlSession 共享数据。
- 二级缓存是跨 SqlSession 的，这意味着不同 SqlSession 之间可以共享缓存的数据。然而，由于二级缓存的数据是在事务提交后才被写入的，因此在高并发场景下，可能会存在多个 SqlSession 同时执行相同查询但无法命中缓存的情况，因为查询结果可能还没有被写入缓存。
- 如果开启了二级缓存，会先从二级缓存获取，没有则从一级缓存获取，还没有则查询数据库（查询结果后先添加到一级缓存，再添加到二级缓存）

博客：[https://blog.csdn.net/qq\\_35512802/article/details/127872970](https://blog.csdn.net/qq_35512802/article/details/127872970)

## 6、查询数据库主流程

### 处理流程

- 1、sqlSession 调用方法，查询数据库操作会交给不同类型的执行器 Executor
- 2、执行器会将任务交给不同类型的语句处理器 StatementHandler(JDBC statement 进行了封装)
- 3、入参和返回结果分别由 ParameterHandler 和 ResultSetHandler 处理器，而真正执行操作的是类型处理器 TypeHandler



### 查询数据库解析

- 默认使用简单执行器，所以这里是 SimpleExecutor 的 doQuery 方法

- newStatementHandler: 创建语句处理器 StatementHandler
- prepareStatement: 获取数据库连接 + 创建 Statement 对象 + 参数化设置 (赋值?)
- handler.query: 向数据库发出 sql 执行, 返回结果集
- resultSetHandler: 封装结果集, 将结果集转换为 pojo

```

1 | @Override
2 | public <E> List<E> doQuery(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler result
3 |   Statement stmt = null;
4 |   try {
5 |     // 1. 获取配置实例
6 |     Configuration configuration = ms.getConfiguration();
7 |     // 2. new一个StatementHandler实例
8 |     StatementHandler handler = configuration.newStatementHandler(wrapper, ms, parameter, rowBounds, result
9 |     // 3. 准备处理器, 主要包括创建statement以及动态参数的设置
10 |     stmt = prepareStatement(handler, ms.getStatementLog());
11 |     // 4. 执行真正的数据库操作调用
12 |     return handler.query(stmt, resultHandler);
13 |   } finally {
14 |     // 5. 关闭statement
15 |     closeStatement(stmt);
16 |   }
17 |

```

博客: [https://blog.csdn.net/qq\\_35512802/article/details/127954732](https://blog.csdn.net/qq_35512802/article/details/127954732)

## 7、Mapper 代理原理

触发时机: SqlSession【SqlSession.getMapper(MapperInterface.class)】

- <package> 标签配置【位于 mybatis-config.xml 文件】的包名下的 Mapper 接口文件都会被加载成对应的代理类工厂

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
  <!-- 其他配置 -->

  <mappers>
    <!-- 指定 Mapper 接口和映射文件所在的包路径 -->
    <package name="com.example.mapper"/>
  </mappers>
</configuration>

```

- 通过 Mapper 接口获取同包同名的 xml 文件, 并解析
- Mapper 接口通过 jdk 代理创建代理类, 接口方法匹配 xml 中标签的 id 值, 执行增删改查

博客: [https://blog.csdn.net/qq\\_35512802/article/details/128086834](https://blog.csdn.net/qq_35512802/article/details/128086834)

## 8、插件机制

允许拦截的方法如下:

- Executor 【SQL 执行器】【update,query,commit,rollback】
- StatementHandler 【Sql 语法构建器对象】

【prepare,parameterize,batch,update,query 等】

- ParameterHandler 【参数处理器】【getParameterObject, setParameters 等】

- ResultSetHandler 【结果集处理器】

【handleResultSets, handleOutputParameters 等】

**Mybatis 插件就是为上述四大对象创建代理类**

拦截四大对象中的方法，**invoke** 添加增强的内容，之后再执行四大对象的原始方法

博客：[https://blog.csdn.net/qq\\_35512802/article/details/128163699](https://blog.csdn.net/qq_35512802/article/details/128163699)

## 9、一级缓存和二级缓存

一级缓存是 SqlSession 级别的缓存。在操作数据库时需要构造 sqlSession 对象，在对象中有一个数据结构 (HashMap) 用于存储缓存数据。不同的 sqlSession 之间的缓存数据区域 (HashMap) 是互相不影响的

二级缓存是 mapper 级别的缓存。多个 SqlSession 去操作同一个 Mapper 的 sql 语句，多个 SqlSession 可以共用二级缓存，二级缓存是跨 SqlSession 的作用范围

- 一级缓存放在执行器里，每次创建 sqlSession 都会创建执行器，所以一级缓存的作用范围在 sqlSession 里
- 二级缓存放在 MappedStatement 里，解析映射配置文件创建而成，不论创建多少 sqlSession 都只用这一个 MappedStatement 对象，所以二级缓存的作用范围在多个 sqlSession(也就是 namespace)

一级缓存

- 默认开启，就是简单得存取值
- update 操作无 commit 会使缓存失效

二级缓存【commit 后才生效】

- 手动开启，缓存结果通过序列化传递，所以地址不同，内容相同
- 查询流程缓存是数据只是放入临时 map 集合，commit 以后才将数据转移到二级缓存，故查询操作后 commit 以后二级缓存才生效
- update 操作 commit 以后才会使缓存失效

博客：[https://blog.csdn.net/qq\\_35512802/article/details/128194303](https://blog.csdn.net/qq_35512802/article/details/128194303)

## mybatis 执行流程

第一阶段：初始化阶段 (应用启动时)

### 1. 加载配置，构建 Configuration

- MyBatis 启动时，会解析 mybatis-config.xml 全局配置文件和所有的 Mapper.xml 映射文件。
- 所有解析出的信息，包括 SQL 语句、参数映射、结果映射等，都被封装成一个个 MappedStatement 对象。
- 最终，所有的配置信息汇集于一个核心的 Configuration 对象中。

### 2. 创建 SqlSessionFactory

- 基于这个 Configuration 对象，MyBatis 会创建一个 SqlSessionFactory 实例。
- SqlSessionFactory 是一个重量级的、线程安全的对象，它的作用是作为生产 SqlSession 的工厂，在应用的整个生命周期中通常只有一个。

第二阶段：SQL 执行阶段 (代码调用时)

## 1. 获取 SqlSession

- 当需要执行数据库操作时，业务代码会从 SqlSessionFactory 中获取一个 SqlSession 实例。
- SqlSession 是 MyBatis 的核心 API，它代表了一次与数据库的会话，是非线程安全的，因此必须在每次请求的开始获取，在结束时关闭。

## 2. 获取 Mapper 代理对象

- 业务代码通过调用 sqlSession.getMapper(UserMapper.class) 来获取 Mapper 接口的动态代理实例。
- 这个代理对象封装了对 SqlSession 的调用，使得开发者可以像调用普通 Java 方法一样，来执行 SQL。

## 3. 代理方法执行，委派给 Executor

- 当执行 userMapper.queryById(1) 时，Mapper 的代理逻辑被触发。
- 代理对象会根据“接口全名 + 方法名”(如 com.example.UserMapper.queryById) 作为 Key，从 Configuration 中找到对应的 MappedStatement。
- 然后，它将执行请求委派给 SqlSession 内部持有的\*\*Executor (执行器)\*\*。

## 4. Executor 处理（缓存与 SQL 准备）

- Executor 是 MyBatis 的调度核心。它首先会处理缓存：
  - ◆ 查询一级缓存 (SqlSession 级别的缓存)，如果命中，直接返回结果。
  - ◆ 如果一级缓存未命中，再查询二级缓存 (Mapper 级别的、跨 SqlSession 的缓存)，如果命中，也直接返回结果。
- 如果缓存都未命中，Executor 会根据 MappedStatement，结合传入的参数，生成最终要执行的 SQL 语句，并封装成 BoundSql 对象。

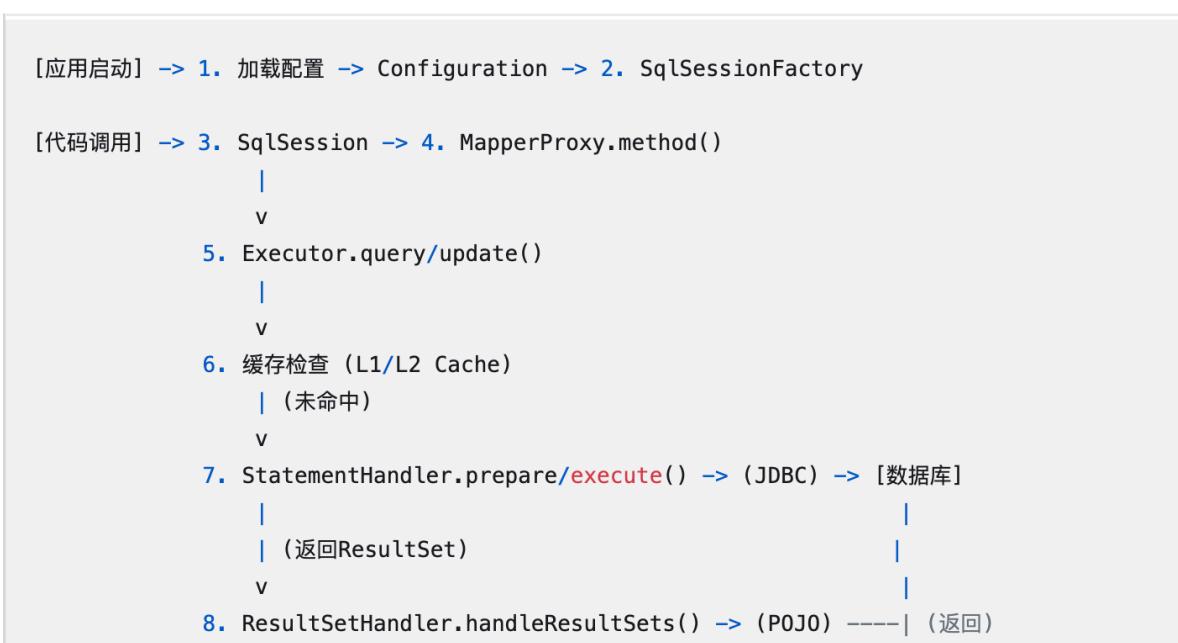
## 5. StatementHandler 执行 SQL

- Executor 会将 SQL 执行的具体工作，委托给\*\*StatementHandler\*\*。
- StatementHandler 会创建一个 JDBC 的 PreparedStatement 对象。
- 它利用 ParameterHandler，将参数设置到 PreparedStatement 的占位符 (?) 上。
- 最后，调用 preparedStatement.execute()，通过 JDBC 与数据库进行真正的交互。

## 6. ResultSetHandler 处理结果

- StatementHandler 执行 SQL 后，如果产生了结果集 (ResultSet)，会将其交给\*\*ResultSetHandler\*\* 来处理。
- ResultSetHandler 会根据 Mapper.xml 中定义的结果映射规则 (<resultMap> 或 resultType)，将 ResultSet 中的数据，一行行地转换成 Java 对象 (POJO)。
- 最终，将转换好的对象 (或对象列表) 返回给调用者，并 (如果配置了) 将查询结果放入缓存。

流程图总结



## 为什么不推荐使用 MyBatis 二级缓存？

### 一级缓存

- 默认开启，会话级别，执行了 update/insert/delete 会使缓存失效即使此 sql 语句事务还未提交。
- SpringBoot 下一级缓存会失效：如果结合 Springboot 话会失效，因为 SpringBoot 集成的 MyBatis， 默认每次执行 SQL 语句时，都会创建一个新的 SqlSession！所以一级缓存会失效。因为 MyBatis 在查询时，会先从事务管理器中尝试获取 SqlSession，取不到才会去创建新的 SqlSession，所以当你的代码加了 @Transactional 事务注解，就能保证一级缓存生效。
- 事务管理器获取 sqlSession 原理：如果当前线程存在事务，并且存在相关会话，就从 ThreadLocal 中取出。如果没有事务，就重新创建一个 SqlSession 并存储到 ThreadLocal 当中，共下次查询使用。
- 一级缓存查询在哪：是在 BaseExecutor 中的 queryFromDatabase 方法中。执行 doQuery 从数据库中查询数据后，会立马缓存到 localCache(PerpetualCache 类型) 中。

### 二级缓存

- 默认关闭，通过在 yaml 中配置 cache-enabled 为 true 来开启二级缓存
- 二级缓存数据填充时机：只有在提交事务之后才会塞缓存，因为二级缓存是允许多个 sqlSession 会话共同可见，所以需要事务提交后才生效避免脏读。
- 开启二级缓存的话，会优先查询二级缓存再一级缓存最后数据库。
- 跨 sqlSession，Mapper 级别，不同 SqlSession 之间可以共享缓存的数据。但不同 Mapper 的二级缓存是独立的。
- 二级缓存可能造成的问题

### 数据不一致

本质是因为二级缓存是跨 sqlSession 的，是 mapper 级别的缓存，所以二级缓存是可以共享于多个会话的，所以可以理解为二级缓存是一个共享资源，多个会话的访问就是对共享资源的竞争，既然对共享资源的读写竞争就会容易造成数据不一致的

情况。

缓存命中率降低

由于二级缓存的数据是在事务提交后才被写入的，因此在高并发场景下，可能会存在多个 SqlSession 同时执行相同查询但无法命中缓存的情况，因为查询结果可能还没有被写入缓存。

博客：[https://mp.weixin.qq.com/s/ff\\_zCZZo0umCNol-D6YjxA](https://mp.weixin.qq.com/s/ff_zCZZo0umCNol-D6YjxA)

---

---

## 类加载过程&&时机

### 1、加载

1. 通过全类名获取定义此类的二进制字节流。
2. 将字节流所代表的静态存储结构转换为方法区的运行时数据结构。
3. 在内存中生成一个代表该类的 Class 对象，作为方法区这些数据的访问入口。

### 2、验证

验证是连接阶段的第一步，这一阶段的目的是确保 Class 文件的字节流中包含的信息符合《Java 虚拟机规范》的全部约束要求，保证这些信息被当作代码运行后不会危害虚拟机自身的安全。

验证阶段主要由四个检验阶段组成：

1. 文件格式验证 (Class 文件格式检查)
2. 元数据验证 (字节码语义检查)
3. 字节码验证 (程序语义检查)
4. 符号引用验证 (类的正确性检查)

【如果代码已经被验证过没问题，可以通过-Xverify:none 参数来关闭大部分的类验证措施，以缩短虚拟机类加载的时间】

### 3、准备

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些内存都将在方法区中分配【仅包括类变量例如被 static 修饰的，并且只会赋值初始的默认值，例如 final int 的默认值是 0】。

### 4、解析

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程。解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用限定符 7 类符号引用进行。

举个例子：在程序执行方法时，系统需要明确知道这个方法所在的位置。Java 虚拟机为每个类都准备了一张方法表来存放类中所有的方法。当需要调用一个类的方法的时候，只要知道这个方法在方法表中的偏移量就可以直接调用该方法了。通过解析操作符号引用就可以直接转变为方法在类中方法表的位置，从而使得方法可以被调用。

综上，解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程，也就是得到类或者字段、方法在内存中的指针或者偏移量。

## 5、初始化

定义：这一步骤才算真正开始执行类中定义的 Java 程序代码（或者说字节码）。在准备阶段，变量已经赋过一次系统要求的初始值，而在初始化阶段，则根据程序员通过程序制定的主观计划去初始化类变量和其他资源（例如类变量加上 final 就会在准备阶段赋值），或者可以从另外一个角度来表达：初始化阶段是执行类构造器 `<clinit>()` 方法的过程。

`<clinit>()` 方法：是由编译器自动收集类中的所有类变量的赋值动作和静态语句块（`static{}` 块）中的语句合并产生的。这个主要针对一个类中是否有静态语句块、静态变量的初始化，如果没有类变量或者静态语句块，就不会生成`<clinit>()` 方法。

## 类加载作用：

### 1. 将“.class”文件转化为 JVM 内部的数据结构

这是类加载最根本、最直接的作用。它是一个从物理文件到内存结构的转换过程。

- 输入：磁盘上的一个.class 文件（或者网络上的一段字节流）。
- 输出：在 JVM 的方法区中，创建出一套完整描述这个类的数据结构（包括类型信息、字段信息、方法信息、运行时常量池等）。
- 核心意义：使得 JVM 能够理解和识别这个 Java 类，为后续的所有操作（如创建对象、调用方法）提供了元数据基础。

### 2. 为类的使用提供“模板”和“蓝图”

类加载完成后，方法区中的类信息就像一个不可变的模板或蓝图。

- 创建对象的依据：当程序执行 `new User()` 时，JVM 会根据方法区中 User 类的“蓝图”，在\*\*堆（Heap）\*\*上分配内存，并按照蓝图中的实例变量定义来布局内存，然后执行构造方法`<init>()` 来初始化这个实例。
- 方法调用的依据：当执行 `user.getName()` 时，JVM 会通过 user 对象的引用找到它在方法区中的类信息，然后定位到 `getName()` 方法的字节码并执行。
- 核心意义：实现了\*\*“类”与“对象”的分离\*\*。一份类信息（模板）可以被用来创建无数个对象实例，大大节约了内存。

### 3. 执行类的静态初始化

类加载的最后一个阶段是初始化。在这个阶段，JVM 会执行类的“构造器”方法 `<clinit>()`。

- 动作：
  1. 为\*\*静态变量（Static Variables）\*\*赋予其在代码中指定的初始值（例如 `private static int count = 10;`）。
  2. 执行\*\*静态初始化块（static {}）\*\*中的所有代码。
- 核心意义：确保在任何对象实例被创建或任何静态方法被调用之前，这个类的静态部分已经被正确地初始化完毕。这是程序逻辑正确性的重要保障。

### 4. 实现 Java 语言的动态性

类加载机制本身，特别是其\*\*“懒加载（Lazy Loading）”\*\*的特性（即只在第一次需要时才加载类），是 Java 语言动态性的基础。

- 动态加载：允许程序在运行时根据需要，动态地加载、链接和初始化类。
- 核心意义：这是许多高级功能和框架的基石，例如：
  - 反射（Reflection）：`Class.forName("...")` 就是通过类加载器在运行时加载一个类。
  - 热部署（Hot Swap）：允许在不停止应用的情况下，更新类的定义。
  - 模块化系统（OSGi 等）和插件化架构：都可以基于自定义的类加载器来实

现模块的隔离和动态加载。

### 精简版一句话总结：

类加载的核心作用，就是将.class文件的字节码加载到内存中，并将其转换为JVM可以识别和使用的、描述该类结构的“元数据模板”（存放在方法区），同时完成该类的静态初始化，为后续创建对象和调用方法提供基础，并支撑起Java的动态性。

### 类加载时机：

#### 1. 创建类的实例 (new关键字)

- 场景: User user = new User();
- 解释: 这是您提到的情况，也是最直观的一种。当JVM遇到new指令时，如果User类还没有被初始化，就必须立即触发它的加载和初始化过程，否则无法创建实例。

#### 2. 访问类的静态变量

- 场景: System.out.println(MyConfig.DEFAULT\_TIMEOUT);
- 解释: 当代码读取或写入一个类的静态字段时，如果这个类还没有被初始化，就必须先触发其初始化。
- 例外: 如果这个静态字段是编译期常量（即被static final修饰，并且在声明时就已赋值的原始类型或String），那么对它的访问不会触发类的初始化。因为这个常量的值在编译阶段就已经被存入了调用方的常量池中，运行时无需再访问这个类。  
\* public static final int TIMEOUT = 100; -> 访问TIMEOUT不会触发类初始化。  
\* public static final int TIMEOUT = new Random().nextInt(); -> 访问TIMEOUT会触发类初始化，因为它不是编译期常量。

#### 3. 调用类的静态方法

- 场景: MyUtils.doSomething();
- 解释: 当调用一个类的静态方法时，如果该类未被初始化，必须先触发其初始化。

#### 4. 反射调用

- 场景: Class.forName("com.example.MyClass");
- 解释: 使用反射API（如Class.forName()）来加载一个类时，如果该类未被初始化，就会触发其初始化。
- 注意: ClassLoader.loadClass("...")方法，默认只进行加载、链接，而不进行初始化，除非你特别指定。这是Class.forName()和ClassLoader.loadClass()的一个重要区别。

#### 5. 初始化子类

- 场景: 当初始化一个子类时。
- 解释: 如果一个类的父类还没有被初始化，那么JVM会保证在初始化子类之前，先初始化其父类。这保证了继承关系的正确性。

#### 6. 动态语言支持 (invokedynamic)

- 场景: 当一个java.lang.invoke.MethodHandle实例最后的解析结果，REF\_getStatic, REF\_putStatic, REF\_invokeStatic的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，则需要先触发其初始化。
- 解释: 这是为支持在JVM上运行的动态语言（如JRuby, Groovy）而设计的。在处理特定的方法句柄时，也需要确保相关类被初始化。这个场景在常规Java开发中不那么常见。

---

## new一个对象的过程

1) 类加载检查：具体来说，当 Java 虚拟机遇到一条字节码 new 指令时，它会首先检查根据 class 文件中的常量池表 (Constant Pool Table) 能否找到这个类对应的符号引用，然后去方法区中的运行时常量池中查找该符号引用所指向的类是否已被 JVM 加载、解析和初始化过

- 如果没有，那就先执行相应的类加载过程
- 如果有，那么进入下一步，为新生对象分配内存

2) 分配内存：就是在堆中给划分一块内存空间分配给这个新生对象用。具体的分配方式根据堆内存是否规整有两种方式：

- 堆内存规整的话采用的分配方式就是指针碰撞：所有被使用过的内存都被放在一边，空闲的内存被放在另一边，中间放着一个指针作为分界点的指示器，分配内存就是把这个指针向空闲空间方向挪动一段与对象大小相等的距离
- 堆内存不规整的话采用的分配方式就是空闲列表：所谓内存不规整就是已被使用的内存和空闲的内存相互交错在一起，那就没有办法简单地进行指针碰撞了，JVM 就必须维护一个列表，记录哪些内存块是可用的，在分配的时候从列表中找到一块足够大的连续空间划分给这个对象，并更新列表上的记录，这就是空闲列表的方式

3) 初始化零值：对象在内存中的布局可以分为 3 块区域：对象头、实例数据和对齐填充，对齐填充仅仅起占位作用，没啥特殊意义，初始化零值这个操作就是初始化实例数据这个部分，比如 boolean 字段初始化为 false 之类的

4) 设置对象头：这个步骤就是设置对象头中的一些信息

5) 执行 init 方法：最后就是执行构造函数，构造函数即 Class 文件中的 <init>() 方法，一般来说，new 指令之后会接着执行 <init>() 方法，按照构造函数的意图对这个对象进行初始化，这样一个真正可用的对象才算完全地被构造出来了

---

## 类加载器详解

### 类加载器作用

- 1、类加载器是一个负责加载类的对象，用于实现类加载过程中的加载这一步。
- 2、每个 Java 类都有一个引用指向加载它的 ClassLoader。
- 3、数组类不是通过 ClassLoader 创建的 (数组类没有对应的二进制字节流)，是由 JVM 直接生成的

简单来说，类加载器的主要作用就是加载 Java 类的字节码 (.class 文件) 到 JVM 中 (在内存中生成一个代表该类的 Class 对象)

### 类加载器加载规则

JVM 启动的时候，并不会一次性加载所有的类，而是根据需要去动态加载。也就是

说，大部分类在具体用到的时候才会去加载，这样对内存更加友好。

对于已经加载的类会被放在 ClassLoader 中。在类加载的时候，系统会首先判断当前类是否被加载过。已经被加载的类会直接返回，否则才会尝试加载。也就是说，对于一个类加载器来说，相同二进制名称的类只会被加载一次。

## 类加载器总结

JVM 中内置了三个重要的 ClassLoader：

1. **BootstrapClassLoader(启动类加载器)**：最顶层的加载类，由 C++ 实现，通常表示为 null，并且没有父级，主要用来加载 JDK 内部的核心类库（%JAVA\_HOME%/lib 目录下的 rt.jar、resources.jar、charsets.jar 等 jar 包和类）以及被 -Xbootclasspath 参数指定的路径下的所有类。
2. **ExtensionClassLoader(扩展类加载器)**：主要负责加载 %JRE\_HOME%/lib/ext 目录下的 jar 包和类以及被 java.ext.dirs 系统变量所指定的路径下的所有类。
3. **AppClassLoader(应用程序类加载器)**：面向我们用户的加载器，负责加载当前应用 classpath 下的所有 jar 包和类。

## 双亲委派模型

1、双亲委派模型的执行流程：

- 在类加载的时候，系统会首先判断当前类是否被加载过。已经被加载的类会直接返回，否则才会尝试加载（每个父类加载器都会走一遍这个流程）。
- 类加载器在进行类加载的时候，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成（调用父加载器 loadClass() 方法来加载类）。这样的话，所有的请求最终都会传送到顶层的启动类加载器 BootstrapClassLoader 中。
- 只有当父加载器反馈自己无法完成这个加载请求（它的搜索范围中没有找到所需的类）时，子加载器才会尝试自己去加载（调用自己的 findClass() 方法来加载类）。
- 如果子类加载器也无法加载这个类，那么它会抛出一个 ClassNotFoundException 异常。

## 拓展一下

**JVM 判定两个 Java 类是否相同的具体规则：** JVM 不仅要看类的全名是否相同，还要看加载此类的类加载器是否一样。只有两者都相同的情况，才认为两个类是相同的。即使两个类来源于同一个 Class 文件，被同一个虚拟机加载，只要加载它们的类加载器不同，那这两个类就必定不相同。

**【每次加载类的时候会传达到最顶层的 BootstrapClassLoader 再一步一步的往下加载，每个加载器会在自己所管辖的扫描范围搜索是否加载过，例如 Object 类由 BootstrapClassLoader 所管辖的加载范围，那么就直接返回 Object 类，而自己写 Object 类不会得到加载只会拿到 BootstrapClassLoader 返回的 Object 类，所以防止了某些类重复加载的可能性】**

## 2、双亲委派模型的好处

双亲委派模型保证了 Java 程序的稳定运行，可以避免类的重复加载（JVM 区分不同

类的方式不仅仅根据类名，相同的类文件被不同的类加载器加载产生的是两个不同的类)，也保证了 Java 的核心 API 不被篡改。

如果没有使用双亲委派模型，而是每个类加载器加载自己的话就会出现一些问题，比如我们编写一个称为 `java.lang.Object` 类的话，那么程序运行的时候，系统就会出现两个不同的 `Object` 类。双亲委派模型可以保证加载的是 JRE 里的那个 `Object` 类，而不是你写的 `Object` 类。这是因为 `AppClassLoader` 在加载你的 `Object` 类时，会委托给 `ExtClassLoader` 去加载，而 `ExtClassLoader` 又会委托给 `BootstrapClassLoader`，`BootstrapClassLoader` 发现自己已经加载过了 `Object` 类，会直接返回，不会去加载你写的 `Object` 类。

## 如何打破双亲委派模型机制

自定义加载器的话，需要继承 `ClassLoader`。如果我们不想打破双亲委派模型，就重写 `ClassLoader` 类中的 `findClass()` 方法即可，无法被父类加载器加载的类最终会通过这个方法被加载。但是，如果想打破双亲委派模型则需要重写 `loadClass()` 方法。

为什么是重写 `loadClass()` 方法打破双亲委派模型呢？双亲委派模型的执行流程已经解释了：

类加载器在进行类加载的时候，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成（调用父加载器 `loadClass()` 方法来加载类）。

重写 `loadClass()` 方法之后，我们就可以改变传统双亲委派模型的执行流程。例如，子类加载器可以在委派给父类加载器之前，先自己尝试加载这个类，或者在父类加载器返回之后，再尝试从其他地方加载这个类。具体的规则由我们自己实现，根据项目需求定制化。

我们比较熟悉的 Tomcat 服务器为了能够优先加载 Web 应用目录下的类，然后再加载其他目录下的类，就自定义了类加载器 `WebAppClassLoader` 来打破双亲委托机制。这也是 Tomcat 下 Web 应用之间的类实现隔离的具体原理。

---

## 垃圾收集器与内存分配策略

为何只针对 JAVA 堆和方法区进行 GC 回收：程序计数器、虚拟机栈、本地方法栈这三个区域随线程而生，随线程而灭；栈中的栈帧随着方法的进入和退出而有条不紊地执行着出栈和入栈操作，每一个栈帧中分配多少内存基本上是在类结构确定下来时就已知的，因此这几个区域的内存分配和回收都具备确定性，不需过多考虑回收问题，因为方法结束或者线程结束时。内存自然就跟着回收了。而 JAVA 堆和方法区则不一样，一个接口的多少个实现类需要的内存可能不一样，一个方法的多个分支需要的内存也可能不一样，我们只有在程序处于运行期间才能知道创建哪些对象，这部分的分配和回收都是动态的，垃圾收集器所关注的就是这部分内存。

判断对象是否存活的算法：

引用计数法：给对象中添加一个引用计数器，每当有一个地方引用它时，计数器 +1；当引用失效时，计数器 -1；任何时刻计数器为 0 的对象就是不可能再使用的。但是一个致命的问题是难以解决对象之间相互循环引用的问题，导致计数器都不为 0，GC 就无法回收。

可达性分析：这个算法的基本思路就是通过一系列的称为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链（Reference Chain），当一个对象到 GC Roots 没有任何引用链相连时，证明此对象是不可用。JVM 就是采用这个算法的。在 JAVA 语言中，可作为 GC Roots 的对象包括下面几种：1、虚拟机栈（栈帧中的本地变量表）中的引用对象。2、方法区中类静态属性引用的对象。3、方法区中常量引用的对象。4、本地方法栈中 JNI（即一般说的 Native 方法）引用的对象。

生存还是死亡：即使在可达性分析算法中不可达的对象，也并非是“非死不可”的，这时候它们暂时处于“缓刑”阶段，要真正宣告一个对象死亡，至少要经历两次标记过程：如果对象在进行可达性分析后发现没有与 GC Roots 相连接的引用链，那它将会被第一次标记并且进行一次筛选，筛选的条件是此对象是否有必要执行 finalize() 方法。当对象没有覆盖 finalize() 方法，或者 finalize() 方法已经被虚拟机调用过，虚拟机将这两种情况都视为“没有必要执行”。

如果这个对象被判定为有必要执行 finalize() 方法，那么这个对象将会放置在一个叫做 F-Queue 的队列之中，并在稍后由一个由虚拟机自动建立的、低优先级的 Finalizer 线程去执行它。这里所谓的“执行”是指虚拟机会触发这个方法，但并不承诺会等待它运行结束，这样做的原因是，如果一个对象在 finalize() 方法中执行缓慢，或者发生了死循环（更极端的情况），将很可能导致 F-Queue 队列中其他对象永久处于等待，甚至导致整个内存回收系统崩溃。finalize() 方法是对对象逃脱死亡命运的最后一次机会，稍后 GC 将对 F-Queue 中的对象进行第二次小规模的标记，如果对象要在 finalize() 中成功拯救自己——只要重新与引用链上的任何一个对象建立关联即可，譬如把自己（this 关键词）赋值给某个类变量或者其他成员变量，那在第二次标记时它将被移除出“即将回收”的集合；如果对象这时候还没有逃脱，那基本上它就真正的被回收了。（个人觉得只要没有和 GC Roots 相连接，就会被标记筛选判定为是否有必要执行 finalize() 方法，有必要执行的话就会放入 F-Queue 队列中，等到 GC 对 F-Queue 中的对象进行第二次小规模标记，看对象是否成功拯救了自己与 GC Roots 连接上，否则就要被回收；如果被筛选为没必要执行 finalize() 方法，并且内存空间还足够时，则保留在内存中，如果内存空间紧张，就会被筛选为有必要执行 finalize() 方法）

回收方法区：永久代的垃圾收集效率非常低，永久代的垃圾收集主要回收两部分内容：废弃常量和无用的类

回收废弃常量与回收 Java 堆中的对象非常相似，假如一个字符串“abc”已经进了常量池中，但是当前系统没有任何 String 对象引用常量池中的“abc”，也没有其他地方引用了这个字面量，这个“abc”常量池就会被系统清理出常量池。常量池中的其他类（接口）、方法、字段的符号引用也与此类似。

判断一个类是否是“无用的类”的条件比较苛刻，要同时满足以下三个条件：1、该类所有的实例都已经被回收，也就是 Java 堆中不存在该类的任何实例。2、加载该类的 ClassLoader 已经被回收。3、该类对应的 java.lang.Class 对象没有在任何地方被引用，无法在任地方通过反射访问该类的方法。

垃圾收集算法：

标记—清除算法：此算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。不足方面有两个：1、效率问题，标记和清除两个过程的效率都不高。2、空间问题，标记清除之后会产生大量不

连续的内存碎片，空间碎片太多可能会导致以后在程序运行过程中需要分配较大的对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。GC触发STW，会影响用户使用体验，也会产生性能消耗，因为JVM会派出线程进行GC动作。

复制算法：为了解决效率问题，一种称为“复制”的收集算法出现了。它将可用内存按容量划分为大小相等的两块，每次只是用其中的一块。当这一块的内存用完了，就将还存活的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。这样使得每次都是对整个半区进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。只是这种算法的代价是将内存缩小为了原来的一般，未免太高了一点。但是根据IBM公司研究表明，新生代中的对象98%是“朝生夕死”的，所以不用按照1:1的比例划分内存空间，而是将内存分为一块较大的Eden空间和两块较小的Survivor空间，每次使用Eden和其中一块SurvivorFrom，最后清理Eden和SurvivorFrom，将存活对象转移到SurvivorTo（Eden和Survivor默认大小比例是8:1），这样的做法使得只有10%的内存被“浪费”。但是我们没办法保证每次回收都只有不多余10%的对象存活，当发生GC时候发现SurvivorTo空间不够存放现有存活对象时，无法容纳的存活对象包括SurvivorTo里的对象将会通过分配担保机制被转移到老年代中去。（当Eden+SurvivorFrom+SurvivorTo总空间无法容纳新的对象实例时就会触发Minor GC）。

### 为什么有两个 survivor 空间？

Java的复制算法GC回收机制需要两个Survivor区的主要原因是为了防止内存碎片化，并提高GC的效率。

当只有一个Survivor区时，每次进行Minor GC（新生代垃圾回收）后，存活的对象会被复制到Survivor区。然而，由于Survivor区本身也会经历GC，这可能导致部分空间无法存放一个完整的对象，从而产生内存碎片化。碎片化会降低内存的使用效率，并可能导致在需要分配大对象时无法找到足够的连续空间。

通过引入两个Survivor区（通常称为From区和To区），可以在每次GC时，将From区中的存活对象复制到To区，然后清空From区。由于To区在GC前是空的，可以确保复制的对象在内存中是连续的，从而避免了碎片化问题。当From区和To区的角色互换时，会再次清空之前的To区（现在成为新的From区），并重复上述过程。此外，两个Survivor区的设计还允许在GC时保留部分存活对象，以便在下一次GC时能够更快地判断哪些对象是存活的，哪些对象是垃圾。这是因为部分对象可能在多次GC后仍然存活，这些对象会被逐渐晋升到老年代（Old Generation）中。

标记—整理算法：复制收集算法在对象存活率较高时就要进行较多的复制操作，效率将会低。更关键的是，如果不想浪费50%的空间，就需要有额外的空间进行分配担保，以应对被使用的内存中所有对象都100%存活的极端情况，所以老年代一般不能直接选用这种算法。根据老年代的特点，有人提出了“标记—整理”算法，标记过程仍然与“标记—清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后清理掉端边界以外的内存。

分代收集（Generational Collection）算法：这种算法没有什么新的思想，只是根据对象存活周期的不同将内存划分为几块。在新生代中，每次垃圾收集时都发现大批量对象死去，只有少量存活，就使用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用“标记—清理”或者“标记—整理”算法来进行回收。

## HotSpot的算法实现

枚举根节点：这里的 GC Roots 的节点主要在全局性的引用（例如常量或类静态属性）与执行上下文（例如栈帧中的本地变量表）中，如果没有在 GC Roots 链里面的对象将被回收。但是在侦查 GC Roots 链的过程中要保持“一致性”，不可以出现分析过程中对象引用关系在不断变化，这样会导致结果不准确。所以在 GC 时需要停止所有线程即为“stop the world”。目前主流 java 虚拟机使用的都是准确式 GC，通过一组 OopMap 数据结构在类加载完成后来保存当类加载时对象内什么偏移量是什么类型的数据、也会保存 JIT 编译过程中在特定位置记录栈和寄存器中哪些位置是引用。从而虚拟机就可以得知哪些地方存放着对象的引用即为 GC Roots 链。

安全点：然而虚拟机不会为每一条 java 指令都生成对应 OopMap，那将会浪费大量额外空间，提高 GC 的空间成本。实际上只会在安全点生成 OopMap 然后停顿下来 GC，安全点的数量太少会让 GC 执行时间变长，太多会增大运行时的负荷。所以安全点选择的标准为“是否具有让程序长时间执行的特征”，比如方法调用、循环异常跳转等指令的地方才会产生 Safepoint。对于 safepoint 这里有两种方案来让当 GC 发生时让所有线程都跑到最近的安全点停顿下来：“抢先式中断”和“主动式中断”。“抢先式中断”不需要线程的执行代码主动去配合，在 GC 发生时，首先把所有线程全部中断，如果发现有线程中断的地方不在安全点上，就恢复线程，让它“跑”到安全点上。但是现在几乎没有虚拟机采用抢先式中断来暂停线程从而响应 GC 事件。“主动式中断”的思想是当 GC 需要中断线程的时候，不直接对线程操作，仅仅简单地设置一个标志，各个线程执行时主动去轮询这个标志，发现中断标志为真时就自己中断挂起，轮询标志的地方和安全点是重合的，另外再加上创建对象需要分配内存的地方。

安全区域：如果程序不执行即为没有分配 CPU 时间，典型的例子就是线程处于 sleep 或者 blocked 状态，这时候线程无法响应 JVM 的中断请求，对于这种情况就需要安全区域来解决。“安全区域”指的是在一段代码片段之中，引用关系不会发生变化。在这个区域任何地方 GC 都是安全的。

## 垃圾收集器：

Serial：新生代、单线程、简单高效，一般用于用户的桌面场景中，分配给虚拟机管理的内存一般来说不会很大，收集几十兆甚至一两百兆的新生代，停顿时间完全可以控制在几十毫秒最多一百多毫秒以内。

ParNew：新生代、多线程版的 Serial，目前只有它能与 CMS 收集器配合工作。

Parallel Scavenge：新生代、多线程收集器，其目的是达到一个可控制的吞吐量，也称为“吞吐量优先”收集器，可用通过-XX:MaxGCPauseMillis 控制停顿时间和-XX:GCTimeRatio 控制吞吐量大小，但是不要以为停顿时间设置的越小越好，GC 停顿时间缩短是以牺牲吞吐量和新生代空间来换取的。停顿时间设置的越小，所能收集的新生代内存空间就越小，并且收集发生得会更加频繁些，吞吐量也会变小。所以要合理设置。里面还有一项自适应调节策略也是和 ParNew 的重要区别。

Serial Old：Serial 收集器的老年代版本、单线程，是 CMS 收集器失败时候的后备预案。

Parallel Old：Parallel Scavenge 收集器的老年代版本、多线程，配合 Parallel Scavenge 使用。

CMS：老年代，是一种以获取最短回收停顿时间为目地的收集器，实现了让垃圾收

集线程与用户线程（基本上）同时工作。

G1：目标替换掉 CMS 收集器，也是并发与并行收集器，但是目前还有些不成熟，性能待议。

内存分配与回收策略：

对象优先在 Eden 分配、大对象直接进入老年期、长期存活的对象将进入老年期（默认 15 岁）、动态对象年龄判定（如果 Survivor 空间中相同年龄所有对象大小的总和大于 Survivor 空间的一半，年龄大于或者等于该年龄的对象就可以直接进入老年期，无须等到 MaxTenuringThreshold 中要求的年龄）、空间分配担保（当 SurvivorTo 无法容纳存活对象时，会判断老年期的连续空间是否大于新生代对象总大小或者历次晋升的平均大小就会进行 Minor GC，然后将通过空间分配担保直接进入老年期。否则进行 Full GC）

---

---

## 并发场景设计&解决方案

---

---

### 并发场景下单防止库存超卖解决方案

#### 【问题】

- 1、通过 select for update 可以保证在并发更新库存的时候一个一个执行，属于悲观锁，但是并发很高的时候效率很低，会导致行级锁的大量阻塞甚至死锁。
- 2、MySQL 默认隔离级别是可重复读，在并发场景时，例如 T1 先读取到库存数是 100 并且准备扣 100，但是由于可重复读，在 T1 事务里还没执行扣减库存动作之前，可能存在 T2 线程操作了库存扣减了 10，但是在 T1 线程还是按照 100 的库存去减 100 的库存需求，但是实际库存数由于 T2 线程已经变成了 90，最后导致库存超卖。

#### 【解决】

- 1、如果是日常商品下单且伴随一定的并发，可以用版本号【CAS 乐观锁思想】方式去完成库存扣减。

**版本号【CAS 乐观锁思想】做法：**每次扣减库存的时候看一下 MySQL 返回的版本号和当前线程携带的版本号是否一致，一致就可以进行扣减库存，不一致说明这条记录已经被别的线程更改过了进行自旋。

```
select version from goods WHERE id= 1001

update goods set num = num - 1, version = version + 1 WHERE id= 1001 AND num > 0 AND version = @version(上面查到的version);
```

这种方式采用了版本号的方式，其实也就是 CAS 的原理。

假设此时 version = 100, num = 1; 100 个线程进入到了这里，同时他们 select 出来版本号都是 version = 100。

然后直接 update 的时候，只有其中一个先 update 了，同时更新了版本号。

那么其他 99 个在更新的时候，会发觉 version 并不等于上次 select 的 version，就说明 version 被其他线程修改过了。那么我就放弃这次 update

### 【MySQL 返回的版本号和当前线程携带的版本号是否一致】解释

每个线程执行扣减库存之前都要查一下还剩多少库存，然后顺便查出当前数据的版本号，在 update 的时候要把查到的 version 和数据库的 version 做 where 相等条件执行，如果执行失败了，所以版本号被人动过了，那么进行自旋。

- 2、如果针对某一个商品做秒杀【在一个时段内可能出现大量请求】，通过 redis 做库存预扣减【redis 支持 decr 等原子操作指令，所以可以在 redis 里面做库存扣减】，降低 MySQL 压力，当 redis 的库存扣减完时或者秒杀时间结束，同步回数据库。
- 3、如果要确保同一时刻只有一个请求做处理【并发量会被限制】，那么利用 redis 的分布式锁去解决。
- 4、如果要保证抢购的有序性，就将请求放入 MQ 去消费处理。

---

## 高并发场景下解决重复下单问题

### 【问题描述】

在确认订单页多次点击下单按钮，导致服务端接收到了 2 次请求，进而可能执行两次下单动作包含支付动作。所以解决的思想是在执行下单接口实际逻辑之前，通过生成一个 requestId 用于卡点。这样无论连续点击多少次下单按钮，下单逻辑的触发是根据 requestId 是否存在来决定是否触发的。

总而言之，每次下单时候都要携带新生成的 requestId 即身份令牌，逻辑执行完就会销毁令牌。

### 【问题解决】

1、当用户进入提交订单页时，客户端调用后端请求获取唯一 ID【例如根据 uid+spuld+ 时间戳之类的规则计算出来的 requestId】，**服务端先将生成的唯一 ID 放入 Redis，并返回给客户端，客户端将唯一 ID 值埋点在页面里面。**

**PS：如果没有 requestId，直接在确认订单页快速点击多次，可能导致服务端接收到多次确认订单命令进而导致接口多次执行，从而导致重复下单造成多次支付扣款，所以需要在提交订单页的时候生成 requestId 用于卡点检测。并且确认订单页接口执行之前，只需关注 requestId 是否存在，你就算多次点击也无所谓了。**

2、当用户点击确认订单页时，客户端需要将唯一 ID 传给后端，后端检查这个唯一 ID 是否存在，如果不存在则报错。如果存在，继续下面流程。

3、使用 redis 分布式锁，用唯一 ID 在限定时间内加锁，如果加锁成功继续下面流程，否则提示“服务正在处理，请勿重复提交”。

4、如果加锁成功，则进行生成订单 + 处理完付款逻辑，然后将锁手动释放。

---

---

## Kafka

---

---

### kafka是什么？有哪些概念？

Kafka是一种高吞吐量、分布式、高可用、基于发布/订阅的消息系统。

#### • 3、Kafka的核心概念

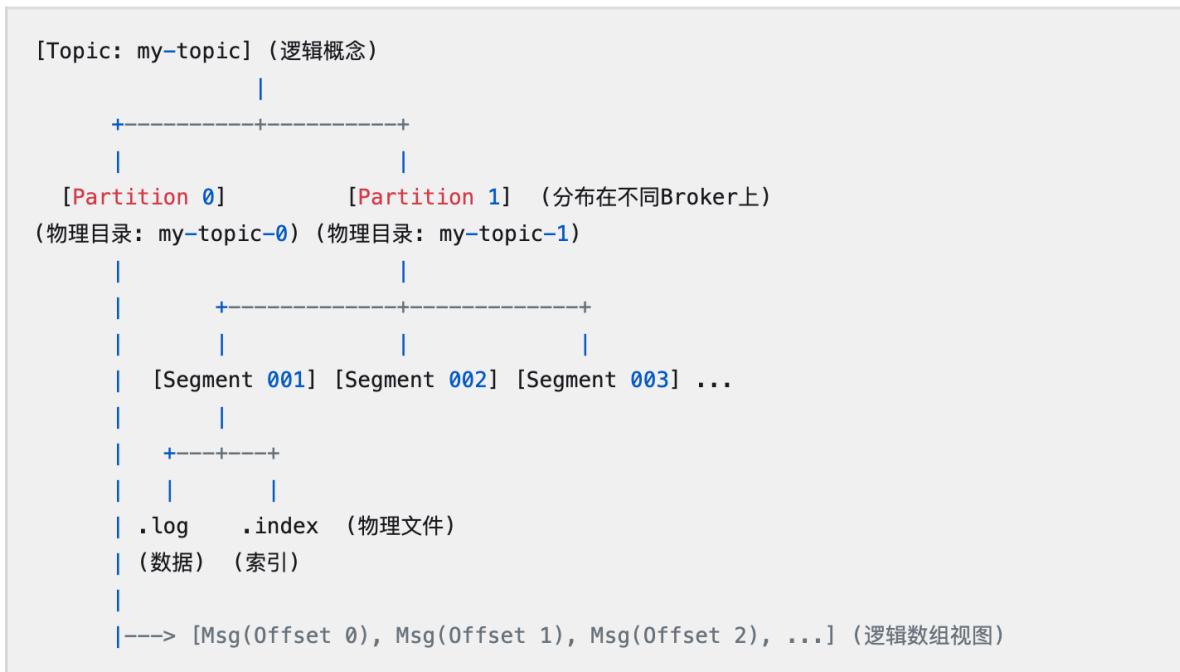
名词	解释
Producer	消息的生成者
Consumer	消息的消费者
ConsumerGroup	消费者组，可以并行消费Topic中的partition的消息
Broker	缓存代理，Kafka集群中的一台或多台服务器统称broker.
Topic	Kafka处理资源的消息源(feeds of messages)的不同分类
Partition	Topic物理上的分组，一个topic可以分为多个partition,每个partition是一个有序的队列。partition中每条消息都会被分配一个有序的id(offset)
Message	消息，是通信的基本单位，每个producer可以向一个topic（主题）发布一些消息
Producers	消息和数据生成者，向Kafka的一个topic发布消息的过程叫做producers
Consumers	消息和数据的消费者，订阅topic并处理其发布的消费过程叫做consumers

---

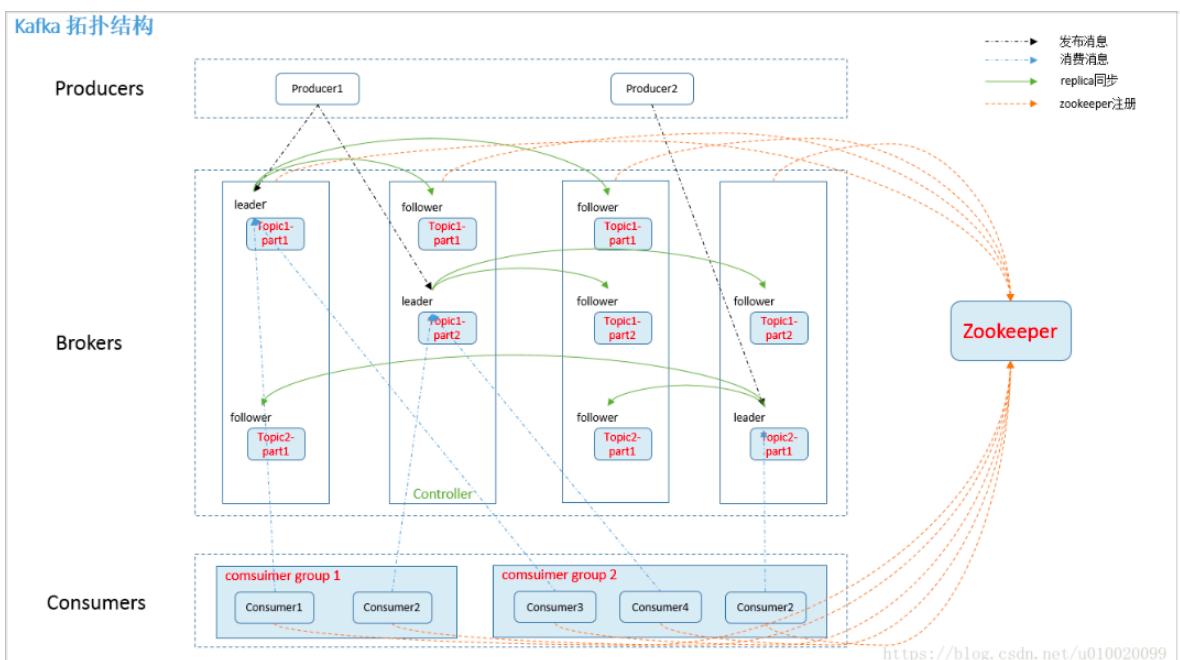
---

### kafka的集群模式结构

Kafka是一个分布式消息订阅——发布系统，无论是发布还是订阅，都须指定 Topic。Topic只是一个逻辑的概念。每个Topic都包含一个或多个 Partition，不同 Partition 可位于不同节点。同时 Partition 在物理上对应一个本地文件夹，每个 Partition 包含一个或多个 Segment，每个 Segment 包含一个数据文件和一个与之对应的索引文件。在逻辑上，可以把一个 Partition 当作一个非常长的数组，可通过这个“数组”的索引 (offset) 去访问其数据。



**特点：**高并发【分布式架构多 broker 和 partition 设计】、高可用【partition 副本机制】、消息持久化【数据存储在磁盘】、高性能【零拷贝实现了高吞吐量和低延迟的消息处理】、数据一致性【ISR 机制】等等



## topic

一个 topic 可以设置成多个 partition 均衡的分布在多个 broker 服务器上。

## 生产者&消费者

通过 broker，生产者发送消息到 partition leader，消费者到 partition leader 读取消息。都是和 partition leader 打交道的。

注意：partition follower 不支持用于消费者的读取，只是用作数据备份，当 partition leader 崩溃时用于故障转移，实现高可用。**分区读写都是由 partition leader 处理【leader 和 follower 是主备关系，非主从关系】。**

## partition

在物理结构上，每个 partition 对应一个物理的目录（文件夹），文件夹命名是

[topicname]\_[partition]\_[序号]。每个 partition 在存储层面是 append log 文件。任何发布到此 partition 的消息都会被直接追加到 log 文件的尾部，每条消息在文件中的位置称为 offset (偏移量)，offset 是一个消息的唯一标记，所以 consumer 也是根据 offset 来准确获取 message。当 broker 收到来自生产者的 message 时，会把 message 以 IO 的方式写入磁盘，并且根据不同的 ack 设置返回确认消息。

### Segment

Kafka 中的 segment 是用于存储消息的物理文件单位，它是 Kafka 存储消息的基本单元。每个主题分区 (partition) 都由多个 segment 组成。每个 segment 是一个独立的物理文件，用于持久化存储消息，并且可以通过追加写入的方式快速地向其中添加新的消息。

### broker

解释：负责接收、存储和分发消息，充当了生产者和消费者的中间件，以及负责消息的持久化和传输。

**【消息存储】** 负责接收来自生产者的消息，并将这些消息持久化存储在磁盘上即 partition 目录文件。这种持久化存储机制确保了 Kafka 能够处理大量的数据并保证数据的可靠性。

**【消息分发】** Broker 不仅存储消息，还负责将消息分发给订阅了相应主题的消费者。它维护了称为分区 (partition) 的消息副本，并按照一定的规则将消息分发到相应的分区中。

**【副本管理】** 当生产者发送消息到 Kafka 集群时，消息首先被写入到 leader 副本中。随后，leader 副本负责将数据同步到所有的 follower 副本，以确保数据的可靠性和一致性。而数据的同步实际上是通过 broker 之间的通信来实现的。并且当 leader 副本宕机后，也负责副本 leader 的选举。

**【消费者协调】** Broker 还负责协调消费者组中的消费者。它维护了消费者组的信息，包括消费者的偏移量 (offset) 和消费状态。通过跟踪消费者的偏移量，Broker 确保每个消费者可以从正确的位置开始消费消息。

**【集群协同】** 在 Kafka 集群中，Broker 与其他 Broker 协同工作，以提供高可用性和伸缩性。例如，当某个 Broker 宕机时，Kafka 集群能够自动进行 Leader 选举，确保服务的连续性。

### 消费者组

消费者是消费者群组的一部分，也就是说，会有一个或多个消费者共同读取一个主题。群组保证每个分区只能被一个消费者使用（但是一个消费者可以读取多个分区）。通过这种方式，消费者可以消费包含大量消息的主题。而且如果一个消费者失效，群组里的其他消费者可以接管失效消费者的工作。

### ZooKeeper/KRaft (集群元数据管理)

Kafka 集群的元数据（如 Broker、Topic、分区信息等）需要一个高可用的组件来管理，以确保集群的协调和一致性。根据 Kafka 版本的不同，这个角色由 ZooKeeper 或内置的 KRaft 协议来承担。

#### 1. 在传统架构中 (依赖 ZooKeeper，常见于 Kafka 3.x 之前的版本)

ZooKeeper 扮演着至关重要的中心化协调角色：

- **Broker 与 Controller 管理：** ZK 通过心跳检测维护 Broker 的存活状态，并负责选举唯一的 Controller 节点来管理集群。
- **Topic 与分区信息维护：** ZK 存储了所有 Topic 的配置、分区列表、副本分配以及每个分区的 Leader 和 ISR (In-Sync Replicas) 集合。
- **(早期) 消费者管理：** 在非常早期的版本中，ZK 也用于存储消费者的 Offset，但此功能后已迁移至 Kafka 内部的 \_\_consumer\_offsets 主题。

#### 2. 在现代架构中 (KRaft 模式，Kafka 3.x 开始推荐并逐渐成为主流)

Kafka 通过内置的 **KRaft** 协议实现了自我管理，完全移除了对 ZooKeeper 的依赖，带来了更简单、更高效的架构。

- **Controller Quorum**: 集群中选举出若干个 Controller 节点，组成一个 Raft 共识小组，共同管理所有元数据。
  - **元数据日志**: 所有元数据变更都作为记录写入一个内部的、高可用的日志主题中。
  - **职责内化**: 所有之前由 ZK 承担的职责，如 Leader 选举、成员管理、配置存储等，全部由 Controller Quorum 通过 Raft 协议在 Kafka 内部完成。
- 

## 为什么说 kafka 是高吞吐量？

1. **分布式架构与并发读写**: Kafka 采用分布式架构，将消息分散存储在多个 Broker 上。这种设计支持多节点并发读写，从而大大提高了吞吐量【每个 partition leader 在不同的 broker 上，自然提升了读写速度】。
  2. **批量读写**: Kafka 支持批量读写消息，这减少了网络传输开销和磁盘 I/O 开销，进一步提高了吞吐量。生产者可以配置批量发送消息的大小，通过在一次请求中发送多个消息来降低网络 I/O 的延迟和负载。
  3. **Zero-copy 技术**: Kafka 采用 Zero-copy 技术，避免了数据从内核空间到用户空间的拷贝，减少了内存和 CPU 的开销，提高了数据传输效率和吞吐量。
  4. **磁盘顺序写**: Kafka 的消息存储采用磁盘顺序写的方式，充分利用了操作系统的预读机制，避免了磁盘随机读写的低效率，提高了数据的写入速度和吞吐量。
- 

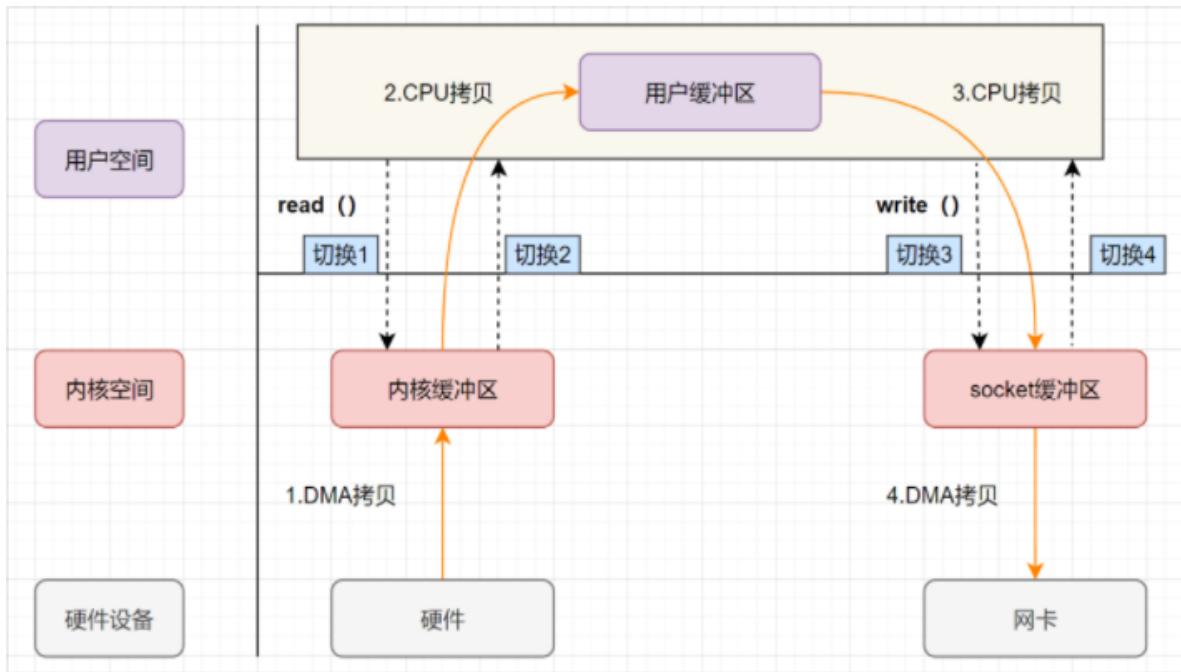
## kafka 的零拷贝是什么？

参考两篇博客：

[https://blog.csdn.net/m0\\_50546235/article/details/136892707](https://blog.csdn.net/m0_50546235/article/details/136892707)

<https://baijiahao.baidu.com/s?id=1769186849807925293&wfr=spider&for=pc>

传统的文件拷贝【磁盘中的某个文件内容发送到远程服务器上】



1. read 之后, 也即向操作系统发出IO 调用, 用户态切换到内核态

2. **DMA** 拷贝数据从硬盘到内核缓冲区

3. **CPU** 拷贝内核缓冲区数据到用户缓冲区, 内核态切换到用户态

4. write 之后, 也即发起IO 调用, 用户态切换到内核态

5. **CPU** 拷贝用户缓冲区数据到 socket 缓冲区

6. **DMA** 拷贝 socket 缓冲区到网卡设备, 内核态切换到用户态

总结: 上述过程可以看出有4次上下文切换, 4次拷贝. 其实这个地方可以优化, 我们把数据拷贝到用户缓冲区再从用户缓冲区拿出数据到socket纯属多此一举, 如果有一种操作直接可以把数据从内核缓冲区到socket缓冲区的话, 就能减少拷贝操作了

## 零拷贝 mmap() 优化

mmap() 优化是基于虚拟内存实现的。

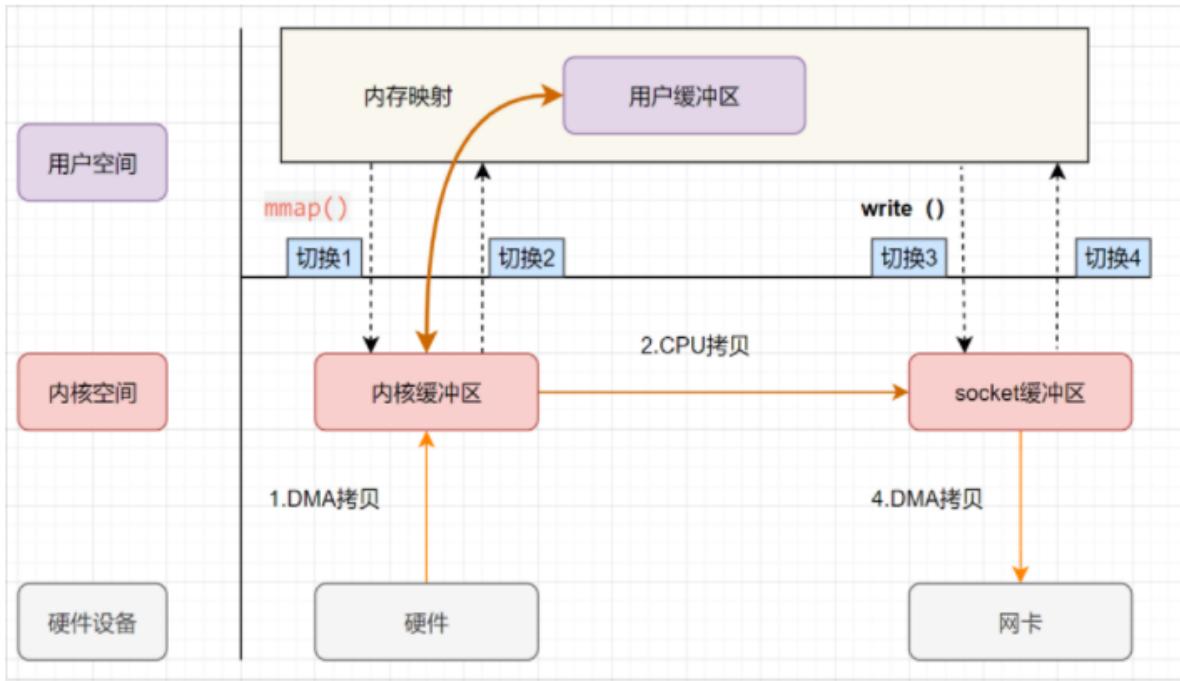
### 虚拟内存是什么?

虚拟内存是由于主存不够大而出现的辅存(理论上来说, 主存想多大就多大, 实际上来说怎么可能, 主存越大价格越高, 追求性价比的情况下才出现的虚拟内存)

### 虚拟内存用处?

虚拟内存可以把内核空间和用户空间的虚拟地址映射到同一个地方, 这样用户对这个映射地址的操作, 内核空间也可以感知到, 那么内核和用户之间就可以减少拷贝了。

**【这样用户态和内核态的交互只需要和映射地址打交道就可以影响到内核空间的数据, 不像之前内核态数据拷贝到用户态, 用户态再将数据拷贝的内核态】**



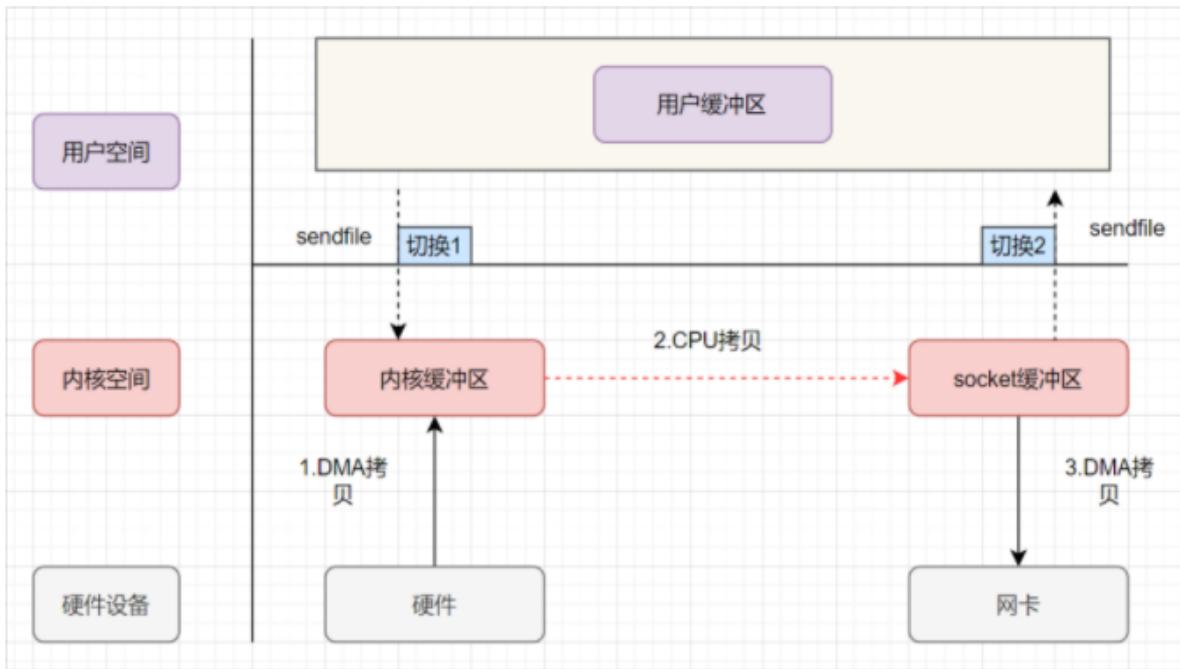
### 当引入 mmap() 机制后的IO操作

1. `mmap()` 调用向操作系统发起IO, 用户态切换到内核态
2. **DMA** 拷贝数据从硬盘到内核缓冲区
3. 内核态切换到用户态
4. `write` 之后, 也即发起IO调用, 用户态切换到内核态
5. **CPU** 拷贝内核缓冲区数据到 socket 缓冲区
6. **DMA** 拷贝 socket 缓冲区到网卡设备, 内核态切换到用户态

总结：上述操作可以看出，我们进行了4次上下文切换，3次拷贝，好像还是不够优化，我们虽然优化了拷贝次数，但是上下文切换也很耗费时间的，4次上下文切换能否可以优化呢？对于系统调用来说(`read`, `write`, `mmap`这类函数)上下文切换是不可避免的，想要优化就必然减少系统调用次数，上述我们不可避免使用到了`write`函数，如果我们将`read`和`write`合并成一次系统调用，在内核中实现磁盘和网课数据传输，就能够减少上下文切换了，也就是`sendfile`优化。

通过`mmap`，进程像读写硬盘一样读写内存（当然是虚拟机内存），也不必关心内存的大小有虚拟内存为我们兜底。

### sendfile 优化



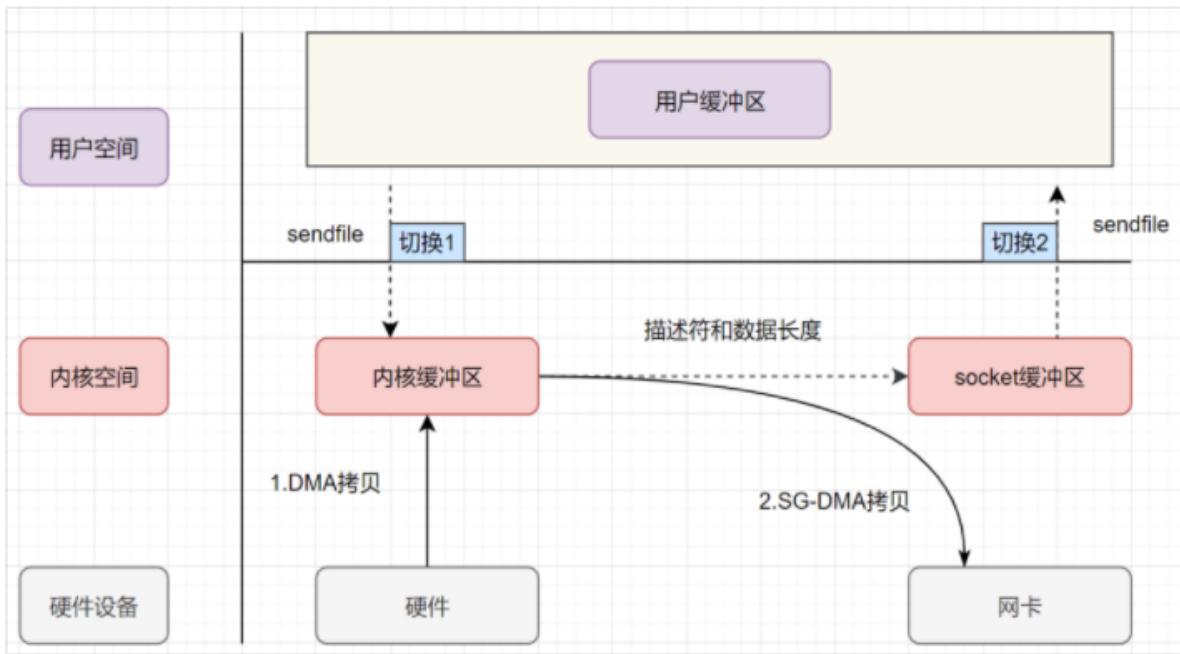
当引入 sendfile 优化后的IO操作

1. 用户进程发起 sendfile 进行 IO, 用户态切换至内核态
2. **DMA** 拷贝数据从硬盘到内核缓冲区
3. **CPU** 拷贝内核缓冲区到 socket 缓冲区
4. **DMA** 拷贝数据从 socket 缓冲区到网卡
5. 内核态切换到用户态, sendfile 函数返回

总结：上述操作发现, 我们只进行了2次上下文切换, 这下上下文切换好像是优化到极致了,但是依旧是3次拷贝【2次DMA和一次CPU】，如何继续优化拷贝次数？

Linux 2.4 版本之后, 对 sendfile 做了升级优化, 引入了 SG-DMA 技术, 其实就是对 DMA 拷贝加入了 scatter/gather 操作, 它可以直接从内核空间缓冲区中将数据读取到网卡, 无需将内核空间缓冲区的数据再复制一份到 socket 缓冲区, 从而省去了一次 CPU 拷贝。

## sendfile +DMA scatter/gather 优化



### 当使用优化后的IO操作

1. 用户进程发起 `sendfile` 进行 IO，用户态切换至内核态
2. **DMA** 拷贝数据从硬盘到内核缓冲区
3. CPU 直接将文件描述符等信息(内核缓冲区的地址 + 偏移量)复制到 socket 缓冲区中
4. DMA 根据文件描述符拷贝内核区数据到网卡
5. 内核态切换到用户态，`sendfile` 函数返回

总结：至此，我们通过两次上下文切换+两次拷贝完成零拷贝终极优化，这里的两次拷贝不是cpu拷贝，而是DMA拷贝，零拷贝的意义也是在减少cpu拷贝，使用mmap和sendfile实现的也叫做零拷贝，只是不够那么零。

## ○ 零拷贝技术的演进之路总结

### 阶段一：传统的I/O (4次拷贝, 4次上下文切换)

- 场景: `File.read() -> Socket.write()`
- 流程:
  1. [拷贝 1: DMA] 硬盘 -> 内核缓冲区
  2. [拷贝 2: CPU] 内核缓冲区 -> 用户缓冲区
  3. [拷贝 3: CPU] 用户缓冲区 -> 内核 Socket 缓冲区
  4. [拷贝 4: DMA] 内核 Socket 缓冲区 -> 网卡
- 痛点: 存在两次完全多余的CPU拷贝(第2和第3次)，数据在用户空间绕了一圈，毫无意义。

### 阶段二：mmap内存映射 (3次拷贝, 4次上下文切换)

- 场景: `mmap() -> Socket.write()`
- 原理: `mmap()` 系统调用通过共享内核缓冲区与用户缓冲区，省去了一次从内核到用户的CPU拷贝。应用程序看到的“用户缓冲区”，实际上就是内核缓冲区的“映射”或“指针”。
- 流程:
  1. [拷贝 1: DMA] 硬盘 -> 内核缓冲区 (此时用户空间已可访问)
  2. (省略了从内核到用户的拷贝)

3. [拷贝 2: CPU] 内核缓冲区 -> 内核 Socket 缓冲区

4. [拷贝 3: DMA] 内核 Socket 缓冲区 -> 网卡

- **进步:** 减少了 1 次 CPU 拷贝, 但上下文切换次数没变。

#### 阶段三: `sendfile` 系统调用 (2 次拷贝, 2 次上下文切换)

- **场景:** `sendfile(socket_fd, file_fd, ...)`
- **原理:** `sendfile()` 是一个更彻底的优化。它告诉内核：“请把这个文件的内容, 直接从你的文件缓冲区, 拷贝到你的 Socket 缓冲区, 然后发出去。整个过程**不要经过用户空间**。”
- **流程:**
  1. [拷贝 1: DMA] 硬盘 -> 内核缓冲区
  2. [拷贝 2: CPU] 内核缓冲区 -> 内核 Socket 缓冲区
  - (省略了从内核到用户和从用户到内核的两次拷贝)
  4. [拷贝 3: DMA] 内核 Socket 缓冲区 -> 网卡
- **进步:** 减少了 1 次 CPU 拷贝 (因为数据没出内核态), 同时**减少了 2 次上下文切换**。这是 **Kafka** 消费者端零拷贝的核心。

#### 阶段四: 带 DMA 收集的 `sendfile` (理论上的终极形态)

- **原理:** 如果网卡驱动支持“DMA 收集”(Gather DMA) 功能, `sendfile` 可以做到更极致。它不再需要将数据从文件缓冲区拷贝到 Socket 缓冲区, 而是直接将文件缓冲区的内存地址和长度等描述信息传递给网卡。网卡 DMA 控制器会根据这些信息, 直接去内存中读取数据并发送。
- **流程:**
  1. [拷贝 1: DMA] 硬盘 -> 内核缓冲区
  - (省略了所有 CPU 拷贝)
  3. [拷贝 2: DMA] 内核缓冲区 -> 网卡
- **进步:** 完全消除了 CPU 数据拷贝。这才是最纯粹意义上的“零拷贝”。

### ○ Kafka 如何应用零拷贝技术总结

现在, 我们从 Producer 和 Consumer 的角度, 看看这些技术是如何在 Kafka 中落地生根的。

#### 1. Consumer (消费者): 零拷贝的“最大受益者”

消费者的核心任务是: 从 Broker 的磁盘读取数据, 然后通过网络发送给消费者客户端。这个场景与 `sendfile` 的需求完美契合。

- **工作流程:**

1. 一个消费者客户端发起 Fetch 请求, 希望从某个 Topic 的某个 Partition 的某个 Offset 开始拉取数据。
2. Kafka Broker (服务器) 接收到请求。
3. Broker 通过 Partition 的索引文件 (.index) 快速定位到数据文件 (.log) 中的物理位置。
4. **关键时刻:** Broker 不会在自己的 JVM 堆内存中为这些数据分配任何缓冲区。相反, 它直接调用操作系统的\*\*`sendfile`\*\*系统调用。
5. `sendfile` 指令内核: “请将 x.log 文件从偏移量 Y 开始的 Z 个字节, 直接发送到这个网络连接 (Socket) 上。”
6. 操作系统内核接管一切, 高效地将数据从磁盘文件缓冲区 (**Page Cache**) 直接拷贝到网络 **Socket** 缓冲区, 然后通过 DMA 发送到网卡。

- **带来的好处:**

- 极高的吞吐量: 数据完全在内核态进行流转, 避免了用户态和内核态之间的昂贵拷贝, 这是 Kafka 能够实现单机每秒处理几十万甚至上百万条消息的关键原因之一。
- **Broker JVM 内存占用低:** Broker 的 JVM 堆内存主要用于元数据管理和少量请求/响应对象, 而不需要为海量的消息数据分配内存。这使得 Broker 非常稳定, 不容易发生 GC 暂停。
- 充分利用 **Page Cache**: sendfile 天然地利用了操作系统的 Page Cache。如果消费者请求的数据恰好还在 Page Cache 中(被其他消费者刚读过), 那么连第一步的磁盘 DMA 读取都省了, 可以直接从内存发送, 速度极快。

## 2. Producer (生产者): 零拷贝的“间接受益者”

生产者的核心任务是: 从客户端接收数据, 然后写入 Broker 的磁盘。这个场景不能直接使用 **sendfile**, 因为数据源在网络, 目的地是磁盘。但是, 生产者依然从零拷贝的思想中间接受益。

- 工作流程:

1. 生产者客户端将一批消息通过网络发送给 Broker。
2. Broker 的网络线程接收到数据, 这些数据首先被放入内核的 **Socket 缓冲区**。
3. Broker 的应用层代码从 Socket 缓冲区读取数据到 **JVM 堆内存中**。(这里存在一次从内核到用户的拷贝, 无法避免)
4. Broker 对消息进行校验、解压等处理。
5. **关键时刻:** Broker 将消息\*\*追加写入 (Append)\*\*到 Partition 对应的.log 文件中。
6. 这个“写入”操作, 实际上是写入到了操作系统的 **Page Cache** 中。操作系统会将 Page Cache 中的“脏页”异步地、批量地刷写到物理磁盘上。

- 如何体现“零拷贝思想”?

- **写入 Page Cache 而非直接写盘:** Broker 的写入操作, 实际上是写内存 (Page Cache), 速度非常快。它把“何时以及如何将数据安全地刷到磁盘”这个复杂的任务, 委托给了操作系统。
- **顺序写:** Kafka 对日志文件的写入是纯粹的顺序追加。这最大化地利用了磁盘的顺序 I/O 性能, 避免了随机写的寻道开销。
- **批量刷盘:** 操作系统会积累一定量的脏页后, 进行一次批量的、高效的刷盘操作。

**总结:** 虽然生产者端无法像消费者端那样利用 sendfile 实现端到端的零拷贝, 但它通过依赖 **Page Cache** 和强制顺序写, 最大限度地减少了直接的、昂贵的磁盘 I/O 操作, 实现了极高的写入性能。这可以看作是“零拷贝”设计哲学在写入场景的一种应用和延伸。

### ○ 零拷贝精简总结

零拷贝 (Zero-Copy) 是一种 I/O 优化技术, 其核心目标是减少甚至消除在数据传输过程中, CPU 在内核空间和用户空间之间进行的不必要的数据拷贝, 以降低 CPU 开销、减少内存带宽占用和减少上下文切换次数, 从而极大地提升数据传输效率。

核心实现技术:

- **DMA (Direct Memory Access):** 硬件层面的基础, 允许外设(硬盘、网卡)与主内存直接交换数据, 无需 CPU 介入。
- **mmap (Memory Mapping):** 通过内存映射, 将内核的文件缓冲区与用户空间的内存共享, 避免了一次从内核到用户的 CPU 拷贝。

- **sendfile**: 一个系统调用，将数据传输完全封装在内核态，直接在内核的文件缓冲区和 Socket 缓冲区之间进行拷贝，避免了数据进出用户空间，显著减少了 CPU 拷贝和上下文切换。

#### Kafka 中的应用：

- **消费者端 (最大受益者)**: Broker 向 Consumer 发送数据时，直接使用 **sendfile** 系统调用。这使得数据可以从操作系统的 **Page Cache** 直接发送到网卡，几乎不经过 Broker 的 JVM 堆内存。这是 Kafka 实现超高读取吞吐量的关键。
- **生产者端 (间接受益者)**: Producer 向 Broker 写入数据时，虽然数据需要从网络进入 Broker 的 JVM，无法完全零拷贝，但其设计体现了零拷贝的思想：
  1. **写入 Page Cache**: 生产者写入的数据，实际上是快速写入操作系统的 **Page Cache** (内存)，而非直接写入物理磁盘。
  2. **顺序追加**: 写入操作是纯粹的顺序写，最大化了磁盘性能。
  3. **委托刷盘**: 将数据从 Page Cache 持久化到磁盘的任务，委托给操作系统异步、批量地完成。

通过这些机制，Kafka 将数据流动的瓶颈从 CPU 和内存拷贝，转移到了纯粹的硬件 I/O (网络带宽和磁盘速度) 上，这是其成为顶级高性能消息系统的基石。

---

---

## kafka 的网络通信模型是什么？

基于 Reactor 模式的 IO 多路复用，不过 Kafka 的 Selector 是对 Java NIO Selector 的二次封装，负责监听多个 socket，以及当某个 Socket 可读或可写时，多路复用器会通知相应的处理器 (Handler) 进行读写操作。

---

---

## kafka 图解【可看内容 很多概念汇总】

<https://zhuanlan.zhihu.com/p/568719368>

---

---

## kafka 如何保证顺序消费

Kafka 只能保证同一分区内的消息顺序，不同分区之间的消息顺序是无法保证的，所以要保证消费者针对一个分区去消费数据是肯定能顺序消费的。

---

---

## kafka ack 三种机制

request.required.acks 有三个值 0 1 -1(all)，具体如下：

0: 生产者不会等待 broker 的 ack，这个延迟最低但是存储的保证最弱当 server 挂

掉的时候就会丢失。

- 1: 服务端会等待 ack 值 leader 副本确认接收到消息后发送 ack, 但是如果 leader 挂掉后, 他不确保是否复制完成新 leader 也会导致数据丢失。
  - 1(all): 服务端会等所有的 follower 的副本收到数据后才会受到 leader 发出的 ack, 这样数据不会丢失。
- 

## kafka 如何保证数据不重复消费?

可能出现重复消费的场景

1. **消费者组再均衡**: 当 Kafka 认为消费者已经离线或者挂掉时, 会触发 Rebalance, 将消息重新分配给其他消费者。如果消费者的消费速度过慢, 可能会导致持续触发 Rebalance, 从而使消息被重复分配给不同的消费者, 导致重复消费。
2. **服务宕机**: 当消费者服务宕机后恢复时, 它可能会从上次保存的偏移量开始重新消费, 如果上次保存的偏移量不正确或者处理逻辑出现问题, 就可能导致部分消息被重复消费。
3. **手动提交偏移量时的逻辑错误**: 当消费者使用手动提交偏移量的方式时, 如果在消息处理完成之前错误地提交了偏移量, 或者在消息处理失败时未正确处理(如未回滚偏移量), 也可能导致消息被重复消费。
4. **生产者重试**: 当生产者发送消息失败时, 可能会进行重试。如果重试成功, 而第一次发送的消息也被 Kafka 成功接收并存储, 那么消费者将会消费到两条相同的消息, 造成重复消费。
5. **消费者并发处理不当**: 如果消费者使用多线程并发处理消息, 并且没有正确地处理线程间的同步和数据一致性, 也可能导致消息的重复消费。

解决重复消费方案

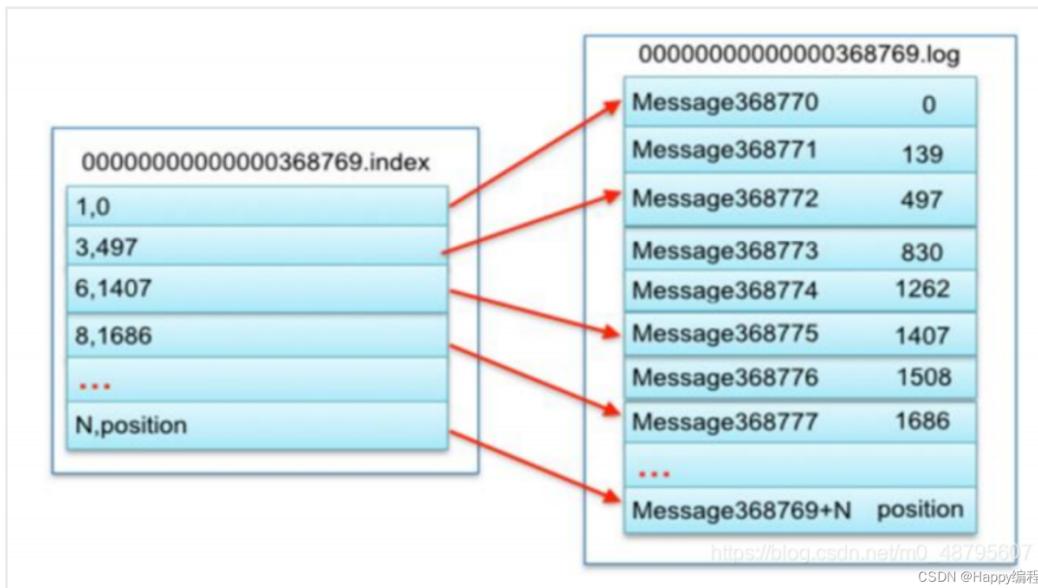
这个问题换种问法, 就是 kafka 如何保证消息的幂等性。对于消息队列来说, 出现重复消息的概率还是挺大的, 不能完全依赖消息队列, 而是应该在业务层进行数据的一致性幂等校验。

比如你处理的数据要写库(mysql, redis 等), 你先根据主键查一下, 如果这数据都有了, 你就别插入了, 进行一些消息登记或者 update 等其他操作。另外, 数据库层面也可以设置唯一键, 确保数据不要重复插入等。一般这里要求生产者在发送消息的时候, 携带全局的唯一 id。

---

## kafka 如何实现数据的高效读取? (顺序读写、分段命令、二分查找)

Kafka 为每个分段后的数据文件建立了索引文件, 文件名与数据文件的名字是一样的, 只是文件扩展名为 index。index 文件中并没有为数据文件中的每条 Message 建立索引, 而是采用了稀疏存储的方式, 每隔一定字节的数据建立一条索引。这样避免了索引文件占用过多的空间, 从而可以将索引文件保留在内存中。



## Kafka 消费者组 Rebalance 再均衡操作什么时候发生？

- 主题分区发生变化：**当 Kafka 主题的分区数增加或减少时，消费者组需要重新分配分区给消费者，以确保每个消费者都能均衡地处理消息。这通常是由于管理员主动进行分区调整或 Kafka 集群的自动扩展 / 缩减导致的。
- 消费者组成员数量发生变化：**当消费者组中新增或删除消费者时，消费者组需要重新进行分区分配。新增消费者可以帮助分摊负载，而删除消费者则可能导致其他消费者需要处理更多的分区。
- 消费者订阅的主题发生变化：**如果消费者订阅的主题发生变化，例如通过正则表达式订阅的主题增加了新的匹配项，或者订阅的特定主题被创建或删除，那么消费者组需要重新调整分区分配，以包括或排除新的主题分区。
- 消费者崩溃或网络问题：**如果消费者由于某种原因（如崩溃或网络中断）与 Kafka 集群失去联系，Kafka 会认为该消费者已经离线，并触发再均衡以将原本由该消费者处理的分区分配给其他消费者。

**再均衡的分配分区过程：**当消费者要加入群组时，它会向群组协调器发送一个 JoinGroup 请求。第一个加入群组的消费者将会成为“群主”。群主从协调器那边获得群组的成员列表（列表包含了发送心跳的活跃消费者），并负责给每一个消费者分配分区。它使用一个实现了 PartitionAssignor 接口的类来决定哪些分区应该被分配给那个消费者。

**流程：**1、群主为每个消费者分配分区。2、群主把分配情况列表发给群组协调器。3、协调器把这些分配信息发送给所有消费者，这样每个消费者就能知道自己被分配到哪个分区去消费了。【每个消费者只能看到自己的分配信息，只有群主知道群组里所有消费者的分配信息。这个过程会在每次的再均衡时重复发生】。再均衡由协调器发起。

### 再均衡发生后产生的问题

再均衡非常重要，为消费者群组带来了高可用性和伸缩性（我们可以放心的添加删除消费者），不过正常情况下，应该尽量避免这种情况。**在再均衡期间，消费者无法读取消息，造成整个群组一小段时间的不可用。**另外，当分区被重新分配给另外一个消费者时，消费者当前的读取状态会丢失，它有可能还需要去刷新缓存，在它重新恢复状态之前会拖慢应用程序。

---

## kafka的leader选举机制策略

Kafka的Partition Leader选举策略主要包括两种：首领选举 (Leader Election) 和 ISR优先选举 (ISR Preferred Election)。

首领选举是Kafka默认的选举策略。当首领副本发生故障或者失去活跃状态时，Kafka会自动进行首领选举，选择一个新的副本作为首领。选举的过程中，Kafka会考虑副本的数据同步进度、健康状态等因素，选择最适合的副本作为新的首领。

ISR优先选举是一种优化选举策略，旨在提高ISR (In-Sync Replicas, 同步副本集合) 中的副本被选举为首领的概率，从而降低首领选举的成本和延迟。当需要进行首领选举时，Kafka会首先检查ISR列表，确定其中的副本是否足够健康和可靠。如果ISR列表中有足够数量的副本，Kafka会优先选择ISR列表中的副本作为新的首领。只有当ISR列表中的副本不足或者不可用时，Kafka才会考虑其他副本作为新的首领。

需要注意的是，Kafka的Partition Leader选举是由Kafka集群中的一个Controller来管理和负责的。Controller的选举基于ZooKeeper的节点注册顺序，谁先注册到ZooKeeper的/controller节点，谁就是Controller。Controller负责整个集群中所有分区和副本的状态管理，包括Partition Leader的选举和分配。

---

## 请简述下你在哪些场景下会选择 Kafka?

标签商品关系变动比较频繁，数据量大，通过kafka发送多个分区，那么多个下游可以执行partition去订阅数据，提高下游各自的消费能力。

---

## kafka数据一致性原理

先来了解下**ISR、OSR、AR、HW、LEO**概念

**ISR(InSyncReplica)、OSR(OutSyncReplica)、AR(AllReplica)**概念

1. **ISR (InSyncReplicas)**: ISR代表一组与领导者 (Leader) 副本保持同步状态的追随者 (Follower) 副本集合。只有当 Follower 副本与 Leader 副本的数据保持同步时，它才会被包含在 ISR 中。这意味着 ISR 中的副本已经接收并确认了所有已提交的消息。只有当消息被 ISR 中的所有副本接收并确认时，它才会被视为

“已同步”状态。此外，只有处于ISR集合中的副本才有资格被选举为新的领导者。ISR机制是Kafka确保数据可靠性和一致性的重要手段。

2. **OSR (OutSyncReplicas)**: OSR代表那些被领导者(Leader)剔除出ISR的副本集合。当追随者(Follower)副本在一定时间内未能与领导者保持同步时，它将被移出ISR并放入OSR中。OSR中的副本可能由于网络延迟、性能瓶颈或其他原因而无法及时同步数据。
3. **AR (AllReplicas)**: AR代表主题分区的所有副本，包括领导者(Leader)副本和所有追随者(Follower)副本，无论它们是否与领导者保持同步。AR是副本集合的全集，包括了ISR和OSR中的所有副本。

#### HW、LEO概念

在Apache Kafka中，HW (High Watermark) 和 LEO (Log End Offset) 是与分区的复制和消息传递相关的两个关键概念。

**HW (High Watermark)** 是一个分区的消息复制进度的指示器。它表示了已经成功复制到所有副本的消息的位置。换句话说，HW之前的所有消息都被认为是已提交的消息，这意味着消费者可以安全地消费这些消息。HW是消费者组维护的偏移量的参考点，也是消费者能够看到的此partition的位置。

**LEO (Log End Offset)** 表示一个分区中消息日志的最后一个位置，即下一条消息要写入的位置。LEO是动态变化的，因为消息不断被追加到分区。它表示了分区中的最新消息位置。

理解HW和LEO的关系以及它们在Kafka中的作用非常重要。消费者通常会从分区的HW位置开始消费消息，因为在HW之前的消息是已经被所有ISR副本复制的，可以认为是已提交的消息，不会丢失。生产者将消息写入分区，然后经过复制流程，最终达到ISR中的所有副本，最终更新HW。这确保了消息的可靠性和持久性。

#### 数据一致性原理

一致性就是说不论是老的Leader还是新选举的Leader，Consumer都能读到一样的数据。

**结论：Kafka通过ISR机制来维护副本之间的数据同步，并通过HW值来控制消息的可见性，从而确保数据的一致性和可靠性**

---

## hbase在批量插入不连续的rowKey数据性能是否高效？

### 一、回顾HBase的写入流程：与RowKey的“物理位置”无关

我们再来看看下一个Put请求（无论是单条还是批量中的一条）在HBase中的完整旅程：

1. **客户端路由**：
  - 客户端的HBase API会首先查询.META.表（这是一个缓存在客户端的元数据表），根据你要插入的RowKey，找到负责管理该RowKey范围的**RegionServer**的地址。
  - **关键点**：即使你批量插入1000个完全随机、不连续的RowKey，客户端API也会自动地将这些Put请求按其目标**RegionServer**进行分组。例如，RowKey A..., B...可能发往RS1，RowKey X..., Y...可能发往RS2。这个分组过程是在客户端完成的。
2. **数据到达RegionServer**：
  - 一个RegionServer收到了属于它管辖范围的一批Put请求。

### 3. 写入 WAL (预写日志):

- RegionServer 会将这批 Put 请求涉及的所有数据变更，以顺序追加 (**Append-Only**) 的方式，一次性地写入到 \*\*WAL (Write-Ahead Log)\*\* 文件中。
- 核心原理：这是一个纯粹的顺序磁盘写。无论你传来的 RowKey 是 A 还是 Z，它们在 WAL 日志文件中的物理位置都是紧挨着的、连续的。WAL 只关心记录操作，不关心 RowKey 的顺序。这个操作极快。

### 4. 写入 MemStore:

- 在写完 WAL 后，RegionServer 会将这批 Put 的数据，写入到对应 Region 的 **MemStore** 中。
- MemStore 是一个内存中的、有序的数据结构（通常是跳表 /SkipList）。当数据写入时，它会被插入到 MemStore 中正确的位置，以维持内存中的 **RowKey 有序性**。
- 核心原理：这是一个纯内存操作。在内存中对一个有序结构（如跳表）进行插入，即使 Key 是不连续的，其操作也是非常高效的（通常是  $O(\log n)$  的复杂度），完全没有磁盘 I/O 的瓶颈。

### 5. 向客户端返回成功：

- 一旦数据成功写入 WAL 和 MemStore，RegionServer 就可以 \*\*立即\*\* 向客户端返回“写入成功”的响应了。

看到了吗？整个客户端感知的、同步的写入流程，只涉及“一次顺序磁盘写 (WAL)”和“一次内存写 (MemStore)”。这两个操作的性能，与你插入的 RowKey 是否连续，几乎没有任何关系。

## 二、不连续的 RowKey 在哪里产生影响？

不连续的 RowKey，其影响主要体现在后台的、异步的操作以及读操作上，而不是在写操作的延迟上。

### 1. 对后台 Flush 的影响：

- 当 MemStore 满，需要 Flush 到磁盘形成 HFile 时，因为 MemStore 中的数据已经是按 RowKey 排好序的，所以无论原始插入的顺序如何，刷写到磁盘的过程依然是一个高效的、将有序数据块顺序写入新文件的过程。
- 因此，对 Flush 的性能影响也很小。

### 2. 对后台 Compaction 的影响：

- Compaction 是将多个小的、有序的 HFile 合并成一个大的、有序的 HFile 的过程。这个过程本质上是一个“多路归并排序”。
- 无论 HFile 内部的 RowKey 跨度有多大，只要它们是有序的，合并的算法和效率都是一样的。
- 因此，对 Compaction 的性能影响也很小。

### 3. 对读性能的巨大影响 (这才是关键！)：

- **Scan 操作**：如果你需要扫描一个 RowKey 范围（例如，Scan from 'user\_1000' to 'user\_2000'），RowKey 的连续性就至关重要。如果你的 RowKey 设计得很好，这些连续的数据很可能都存储在同一个 Region，甚至同一个 HFile 中，使得扫描是一个高效的顺序读。
- **不连续 RowKey 的 Scan**：如果你想查询一批不连续的 RowKey，你就不能使用 Scan，而必须使用 Multi-Get (批量 Get)。Multi-Get 会被客户端拆分成多个针对不同 Region 的 Get 请求，这实际上是一系列的随机读，性能

远低于 Scan。

### 最终结论：

HBase 的 LSM-Tree 架构，通过将\*\*“写入时排序”的成本，从客户端的同步路径中剥离出来，转移到内存 (region 的 memStore，跳表数据结构，因此插入数据也很快) 和后台异步合并中，从而实现了无论批量插入的 RowKey 是否连续，都能保持极高写入性能\*\*的强大特性。

所以，在设计 HBase 的 RowKey 时，我们通常更关心的是如何通过散列、加盐等方式，让写入的 RowKey 变得“不连续”，以打散写入压力，避免“写热点”。而对 RowKey 连续性的要求，主要是为了优化“读”(特别是 Scan) 的性能。这是一个非常经典的设计权衡。

---

---

## Hbase VS ES

### 写入

**\*\*结论：HBase 性能远高于 Elasticsearch。**

#### HBase 的写入机制：极致的“追加写”艺术

HBase 是一个基于 **LSM-Tree (Log-Structured Merge-Tree)** 架构的分布式、列式存储数据库。它的写入 (Put) 和删除 (Delete) 操作，为了追求极致的写入性能，被设计为纯粹的、内存中的、顺序追加的操作。

**HBase 的新增 (Put) 流程：**

##### 1. 写入 WAL (Write-Ahead Log):

- 当一个 Put 请求到达 RegionServer 时，数据首先会被顺序追加到一个名为 \*\*WAL (HLog)\*\* 的文件中。
- 这是一个纯粹的顺序写磁盘操作，速度极快。它的唯一目的是为了在 RegionServer 宕机时，能够恢复数据，保证持久性。

##### 2. 写入 MemStore:

- 在写入 WAL 之后，数据会被写入到该行键 (RowKey) 所属的 Region 的内存缓存——**MemStore** 中。
- MemStore 是一个内存中的有序数据结构 (通常是跳表)。数据写入 MemStore，是一个纯内存操作，速度同样极快。
- 写入 MemStore 后，就可以立即向客户端返回成功了。

##### 3. 异步刷写 (Flush):

- MemStore 中的数据会不断累积。当它达到一定大小时 (hbase.hregion.memstore.flush.size)，或者定期刷写时，整个 MemStore 的内容会被异步地、批量地、作为一个整体刷写到 HDFS 上，形成一个新的、不可变的 **HFile** (StoreFile)。
- 这个刷写过程也是一个高效的顺序写过程。

#### HBase 写入性能高的原因总结：

- 无“就地更新”：HBase 从不去磁盘上查找并修改旧的数据。所有写入都是“追加”。
- 内存操作为主：客户端的写入请求，实际上只需要完成一次顺序写 WAL 和一次写内存 (MemStore) 即可返回，这两个操作都非常快。

- 批量顺序刷盘: 真正昂贵的磁盘 I/O 被转换成了后台的、异步的、批量的顺序写操作, 与客户端的请求路径完全解耦。

## Elasticsearch 的写入机制: 复杂的“倒排索引”世界

Elasticsearch 是一个基于 **Apache Lucene** 的分布式搜索和分析引擎。它的核心数据结构是倒排索引 (**Inverted Index**), 这个结构天生就是为“读”而优化的, 对于“写”则相对不友好。

### Elasticsearch 的新增 (Index) 流程:

1. 协调节点路由: 请求到达集群中的某个节点 (协调节点), 根据文档 ID 或路由规则, 确定该文档应该被写入哪个主分片 (Primary Shard)。请求被转发到该主分片所在的节点。
2. 写入 Lucene 索引 (这是一个复杂的过程):
  - 写入内存缓冲区 (**in-memory buffer**): 文档首先被写入 Lucene 的内存缓冲区。
  - 创建新的 Segment: 当内存缓冲区满, 或者定期 (`index.refresh_interval`, 默认 1 秒) 刷新时, 缓冲区中的所有文档会被处理并写入到一个新的、不可变的磁盘文件——\*\*段 (Segment)\*\* 中。
    - ◆ 这个写入过程非常复杂, 它不仅仅是写原始文档, 更重要的是要对文档进行分析 (**Analyze**)——分词、去除停用词、词干提取等, 然后构建倒排索引、正排索引 (Doc Values)、存储字段等多种数据结构。这是一个计算密集型和 I/O 密集型的操作。
  - Commit 到磁盘: 为了保证数据持久性, 变更会被写入 **Translog** (事务日志), 类似于 HBase 的 WAL。然后, 新的 Segment 信息会被写入一个 commit point, 持久化到磁盘。
3. 同步到副本: 主分片完成写入后, 会将写入请求并发地发送给所有的副本分片 (Replica Shards)。只有当指定数量的副本 (quorum) 也成功写入后, 才会向客户端返回成功。

### Elasticsearch 写入性能相对较低的原因:

- 构建索引的巨大开销: 每次写入, 都需要进行分词、构建倒排索引等复杂的计算。这是其与 HBase 最根本的区别。
- Segment 的不可变性: Lucene 的 Segment 是不可变的。这意味着\*\*“更新”操作的代价非常高\*\* (见下文)。

## 删除

\*\*结论: HBase 性能远高于 Elasticsearch。

**HBase 为什么快? —— “假删除”即“追加写”**

- 原因剖析: HBase 的删除操作同样遵循 LSM-Tree 的哲学。
  1. 写入墓碑标记 (**Tombstone**): 执行 Delete 操作时, HBase 并不会去磁盘上查找并物理删除数据。它只是写入一条新的、特殊的“墓碑”记录。
  2. 性能等同于新增: 这个写入“墓碑”的过程, 和一次 Put 操作几乎完全一样, 都是一次快速的“写 WAL + 写 MemStore”。
  3. 后台清理: 真正的物理删除, 被推迟到后台的\*\*Compaction (合并)\*\*过程中异步完成。

**Elasticsearch 为什么相对慢? —— 同样是“假删除”, 但链路更长**

- 原因剖析: 由于 Segment 不可变, ES 的删除也是“假删除”。

1. 写入删除标记: ES会在一个特殊的.del文件中，记录下被删除文档的ID。
2. 链路开销: 虽然也是一次标记写入，但它依然需要经过ES的路由、主副本同步等一系列写入流程，这个链路比HBase的“写内存即返回”要长。
3. 查询性能影响: 这种“假删除”会给查询带来额外开销，因为每次查询都需要去检查.del文件来过滤掉已删除的文档。

## 更新

**\*\*结论: HBase 性能远高于 Elasticsearch。**

**HBase 为什么快? —— “更新”就是一次新的“新增”**

- 原因剖析: HBase 没有真正的“更新”操作。
  1. 带时间戳的新版本: 当你“更新”一行数据时，HBase 只是简单地执行了一次 Put 操作，插入了一条带有更新时间戳的新版本数据。
  2. 性能等同于新增: 因此，更新操作的性能和新增操作一样高。旧版本的数据会被自然地保留下来(可以通过查询历史版本来获取)，并在后台 Compaction 时被清理。
- 举例: 更新用户的手机号码。HBase 只是插入了一条新的 phone 列数据，其时间戳比旧数据更新。

**Elasticsearch 为什么极慢? —— 昂贵的“读取-删除-新增”三部曲**

- 原因剖析: Segment 的不可变性，使得ES的更新操作成为其性能上的最大痛点。
  1. 读取 (Read): 首先，ES 必须从旧的 Segment 中找到并读取出完整的原始文档。
  2. 删除 (Delete): 然后，在.del文件中将这个旧文档标记为删除。
  3. 新增 (Index): 最后，将更新后的文档，作为一个全新的文档，执行一次完整的\*\*索引 (新增) \*\*流程。
- 开销巨大: 一次 Update 操作，包含了至少一次读 I/O 和一次完整的写 I/O 及索引构建，其成本是新增操作的好几倍。
- 举例: 修改一篇博客文章的标题。ES 需要经历读出全文、标记旧文、重新索引新文的整个过程。

## 查询

**\*\*结论: 高度依赖场景，两者各有专长，无法简单分出高下。**

**HBase 的读取机制: 基于 RowKey 的“地图导航”**

HBase 是一个分布式、有序的、稀疏的巨型 Map。它的所有数据都严格按照行键 (RowKey) 的字典序进行物理存储。它的读取性能完全取决于你如何利用这个有序的 RowKey。

**场景 1: 基于 RowKey 的精确 Get (单点查询)**

- 查询示例: get 'my\_table', 'user\_12345'
- 内部流程:
  1. 客户端首先访问 ZooKeeper (或.META表) 来定位包含 user\_12345 这个 RowKey 范围的 RegionServer。
  2. 请求到达该 RegionServer。
  3. RegionServer 首先在内存中的 MemStore 里查找。
  4. 如果没找到，它会查询 BlockCache (读缓存)。
  5. 如果缓存还没命中，它会利用 HFile 的索引 (多级 B+ 树索引) 来快速定位到

磁盘上包含该 RowKey 的数据块 (Data Block)。

6. 将数据块从磁盘读入内存，找到 user\_12345 这一行，返回给客户端。

- **性能表现: 极高且稳定 (毫秒级)。**

- 整个过程就像查字典一样，路径非常清晰。通过 RowKey，可以一次性、精准地定位到数据所在的物理位置，I/O 次数非常少。
- 这是 HBase 最擅长、性能最高的查询方式。

### 场景 2: 基于 RowKey 的范围 Scan (范围扫描)

- **查询示例:** scan 'my\_table', {STARTROW => 'user\_1000', ENDROW => 'user\_2000'}
- **内部流程:** 与 Get 类似，先定位到 STARTROW (user\_1000) 的位置。然后，利用 HBase 数据按 RowKey 物理有序的特性，在磁盘上进行高效的顺序扫描，直到 ENDROW 为止。
- **性能表现: 非常高。** 顺序读磁盘的性能远高于随机读。只要范围不是过大，性能都非常好。

### 场景 3: 不基于 RowKey 的条件查询 (HBase 的“噩梦”)

- **查询示例:** scan 'my\_table', {FILTER => "SingleColumnValueFilter('cf', 'city', =, 'beijing')"} (查询所有城市为“北京”的用户)
- **内部流程:**
  1. 因为 city 这个列没有索引，HBase 别无选择，只能执行全表扫描 (**Full Table Scan**)。
  2. 它会从表的第一行开始，逐行地读取数据，然后检查 city 列的值是否等于“北京”。
  3. 这个过程会遍历表的所有 Region、所有 HFile。
- **性能表现: 极其低下，甚至是灾难性的。**
  - 对于一个 TB 甚至 PB 级别的大表，全表扫描可能需要数小时甚至数天。
  - 为了解决这个问题，通常需要引入 HBase 的二级索引方案，如 Phoenix 或集成 Elasticsearch，但这已经超出了 HBase 自身的能力范畴。

### Elasticsearch 的读取机制: 基于倒排索引的“精准搜索”

ES 的核心是倒排索引，它天生就是为了解决“从值反查文档”这个问题的。

#### 场景 1: 全文检索

- **查询示例:** 搜索所有包含“高性能”和“分布式”这两个词的文档。
- **内部流程:**
  1. **查询词项 (Term):** ES 会去倒排索引中，分别查找“高性能”和“分布式”这两个词项。
  2. **获取文档列表 (Posting List):** 瞬间就能得到两个包含这些文档 ID 的列表。例如：高性能 -> {doc1, doc5, doc10}，分布式 -> {doc5, doc12, doc20}。
  3. **计算交集:** ES 会对这两个文档列表进行高速的交集运算，得到结果 {doc5}。
  4. **获取文档:** 最后，根据文档 ID doc5，去获取完整的文档内容。
- **性能表现: 极高。**
  - 整个过程主要是对索引的查找和集合运算，速度飞快。这是 ES 的看家本领，HBase 完全无法做到。

#### 场景 2: 复杂的结构化查询 (组合条件过滤)

- **查询示例:** GET /products/\_search { "query": { "bool": { "filter": [ { "term": { "brand": "apple" } }, { "range": { "price": { "gte":

1000 } } } ] } } } (查找品牌为“苹果”且价格大于等于1000的商品)

- 内部流程:

1. ES会利用倒排索引(对于brand字段)和专门为数值、日期等优化的索引结构(如BKD树,对于price字段),分别快速地找到满足每个条件的文档集合。
  2. 然后,对这些文档集合进行\*\*位运算(Bitset)\*\*等高效的交集计算。
  3. 最终得到满足所有条件的文档。
- 性能表现: 非常高。ES能够高效地处理任意字段的组合查询,而HBase只能处理基于RowKey的查询。

### 场景3: 聚合分析

- 查询示例: 按品牌统计商品数量,并计算每个品牌的平均价格。
- 内部流程: ES会利用其\*\*列存(Doc Values)\*\*特性。Doc Values将每个字段的值按列式存储,非常适合进行聚合计算。ES可以在不加载整个文档的情况下,对brand和price这两列进行快速的扫描和聚合。
- 性能表现: 非常高。聚合分析是ES的另一个核心优势,而HBase需要通过MapReduce或Spark等外部计算框架才能完成复杂的聚合。

### 场景4: 基于文档ID的精确Get

- 查询示例: GET /products/\_doc/12345
- 性能表现: 也非常高。ES对文档ID也有内部的快速索引。但通常来说,由于其写入链路更复杂,数据可能分布在多个Segment中,其单点Get的延迟稳定性可能略逊于HBase那种简单直接的路径。

#### 总结与对比

查询场景	HBase	Elasticsearch	性能更高者
基于唯一ID的精确Get	极高(通过RowKey精准定位)	非常高(通过文档ID索引)	HBase(通常延迟更低、更稳定)
基于ID前缀的范围Scan	非常高(利用RowKey物理有序性)	较差(需要特殊查询)	HBase
全文检索	不支持(需要全表扫描)	极高(核心能力)	Elasticsearch(碾压)
任意字段的组合条件查询	极差(需要全表扫描或二级索引)	非常高(倒排索引的优势)	Elasticsearch(碾压)
聚合分析(Grouping, Aggregations)	不支持(需要外部计算框架)	非常高(利用列存Doc Values)	Elasticsearch(碾压)

#### 一句话总结:

- 把HBase想象成一本巨大的、严格按“拼音(RowKey)”排序的字典。如果你按拼音查字,或者查一个拼音区间的字,它快得惊人。但如果你想找所有“11划”的字,它只能一页一页地翻。
- 把Elasticsearch想象成一本带有极其详尽“偏旁部首索引”、“笔画索引”、“主题索引”的百科全书。无论你从哪个角度、用什么复杂的条件组合去查找信息,它都能通过这些丰富的索引,快速地为你定位到内容。

因此,技术选型时,应该根据你的核心查询模式来决定:

- 如果你的业务是海量的、简单的KV式读写,比如用户画像、订单快照、监控时序数据,HBase是更好的选择。
- 如果你的业务是复杂的、多维度的搜索和分析,比如商品搜索、日志分析、地理位置查询,Elasticsearch是唯一的选择。

**总结: Hbase优势在于新增、删除、更新能力远超ES,但是在查询方面,除了rowkey或范围scan性能很高,别的查询方面都不如ES,ES强大的复杂检索能力是Hbase无法比拟的。**

## Hbase VS CK

核心结论速览		
操作维度	性能更高者	核心设计原理差异
新增 (Create)	HBase (通常更高)	LSM-Tree (内存追加写) vs. MergeTree (批量合并写), HBase对单条/小批量写入更友好。
删除 (Delete)	HBase (通常更高)	“写墓碑” (LSM-Tree) vs. “写标记+后台合并” (MergeTree), 两者类似, 但HBase的写入链路更短。
更新 (Update)	HBase (远高于CK)	“版本追加写” (LSM-Tree) vs. 不支持/变相重写 (MergeTree), CK天生不支持高效更新。
查询 (Read)	ClickHouse (在分析场景下碾压)	OLAP列存引擎 (MergeTree) vs. KV式列存 (LSM-Tree), CK为海量数据扫描和聚合分析做了极致优化。

## 一、新增数据 (Create / Insert)

**结论：对于实时、高并发的单条或小批量写入，HBase 性能通常更高。对于海量数据的批量导入，两者性能都很高，ClickHouse 可能更有优势。**

**HBase 为什么快？—— LSM-Tree 的极致写优化**

- **设计原理:** HBase 基于 **LSM-Tree (Log-Structured Merge-Tree)** 架构。其核心思想是将所有随机写操作，全部转换成内存中的高速操作和磁盘上的顺序写操作。
  1. **内存写入:** 新增数据 (Put) 首先写入 **WAL** (预写日志，一次顺序磁盘写)，然后直接写入 **MemStore** (纯内存中的有序结构)。完成这两步即可向客户端返回成功。
  2. **异步刷盘:** 只有当 MemStore 累积到一定大小时，才会异步地、批量地将数据刷写到 HDFS，形成一个新的、不可变的 **HFile**。
- **性能优势:** 客户端的写入延迟极低，因为它面对的是一个**内存数据库**。真正耗时的磁盘 I/O 被放到了后台，并且是以高效的**顺序写**方式进行。这使得 HBase 能够轻松应对高并发、低延迟的写入场景。

**ClickHouse 为什么相对“慢”？—— MergeTree 的批量合并哲学**

- **设计原理:** ClickHouse 的核心引擎是 **MergeTree** 家族。其设计哲学是\*\*“数据分批写入，后台不断合并”\*\*，旨在为后续的分析查询 (OLAP) 做极致优化。
  1. **分 Part 写入:** 当一批数据 (即使是一行) 被 INSERT 时，它会形成一个小的数据部分 (**Data Part**)。每个 Part 都是一组独立的、按主键排序的列式文件，存储在磁盘上。
  2. **后台合并:** ClickHouse 的后台线程会不断地将这些小的、零散的 Data Parts\*\*合并 (Merge)\*\*成更大、更有秩序的 Part。
- **性能特点:**
  - **不适合高频单条写入:** 如果你以极高的频率 (如每秒几万次) 执行单行 INSERT，会导致磁盘上瞬间产生大量的小文件 (**Parts**)。这不仅会给后台合并线程带来巨大压力，还会因为文件句柄过多、元数据管理复杂而导致写入性能急剧下降，最终甚至可能报错 Too many parts。
  - **批量写入性能极高:** ClickHouse 强烈推荐以大批量 (例如，每次几千到几十万行) 的方式进行 INSERT。当数据成批写入时，它能形成结构良好、规模适中的 Part，后台合并压力小，整体吞吐量非常高。

**总结：**HBase 像一个“接线员”，能快速响应每一个打进来的电话 (单条写入)。

ClickHouse 像一个“邮递员”，它希望你把信攒成一大包再给它，它一次性处理一大包的效率极高，但如果你一封一封地给，它会“不堪其扰”。

## 二、删除 & 更新数据 (Delete & Update)

**结论：HBase 在删除和更新操作上，性能远高于 ClickHouse。**

## HBase为什么快？——“假删除”与“版本追加”

- **设计原理:** 在 LSM-Tree 架构中，没有真正的“就地”删除和更新。
  1. **删除 (Delete):** 只是写入一条\*\*“墓碑标记” (Tombstone)\*\*。这次操作和 Put一样，是一次廉价的内存追加写。真正的物理删除发生在后台 Compaction 时。
  2. **更新 (Update):** 就是一次新的 Put。HBase 会插入一个带有更新时间戳的新版本数据，旧版本依然保留。
- **性能优势:** 将昂贵的“查找并修改 / 删除”的随机 I/O 操作，转换成了与新增一样高效的顺序写操作。

## ClickHouse为什么极慢/不支持？——为分析而生的不可变性

- **设计原理:** MergeTree 的 Data Parts 在设计上是不可变的 (**Immutable**)。一旦一个 Part 被写入磁盘，就不会再被修改。这是其能够提供稳定、高速查询性能的基础。
  1. **删除 (Delete):** ClickHouse 通过一种\*\*变相 (mutation)\*\* 的方式支持删除，语法是 ALTER TABLE ... DELETE WHERE ...。
    - ◆ **内部机制:** 这不是一个即时操作。它会在后台异步地重写所有包含符合条件数据的 Data Parts。它会读取旧 Part，过滤掉要删除的行，然后写一个新的 Part 来替代旧的。这是一个非常重的 I/O 密集型操作。
  2. **更新 (Update):** 与删除类似，ALTER TABLE ... UPDATE ... WHERE ... 也是通过异步重写 Part 来实现的。
- **性能劣势:** ClickHouse 的删除和更新操作，其成本与需要被重写的数据量成正比，而不是与你删除 / 更新的行数成正比。对于大表来说，这是一个非常缓慢且昂贵的操作，完全不适用于高并发的 OLTP 场景。

总结：HBase 的删改是“轻量级标记”，ClickHouse 的删改是“重量级重写”。

## 三、查询数据 (Read)

结论：对于 OLAP (在线分析处理) 场景，如复杂的过滤、分组、聚合，  
ClickHouse 的性能碾压 HBase。对于简单的、基于主键的 KV 式查询，HBase 性能更高。

## ClickHouse为什么快？——极致的OLAP优化

- **设计原理:** MergeTree 引擎是为 OLAP 而生的。
  1. **列式存储:** 数据按列存储。当查询只涉及少数几列时 (OLAP 查询的典型特征)，它只需读取这几列的数据文件，避免了读取无关数据，I/O 开销极小。
  2. **数据压缩:** 列式存储使得同一列的数据类型相同，具有很高的相似性，因此可以实现极高的压缩比，进一步减少 I/O。
  3. **稀疏主键索引:** 它允许在巨大的数据块 (Granule，默认 8192 行) 之间进行快速跳转，迅速定位到查询范围的起点。
  4. **向量化执行引擎:** 利用 CPU 的 SIMD 指令，对一批数据 (一个向量 / 列块) 进行并行计算，而不是一行一行地处理，计算效率极高。
- **举例:** 在上亿条的订单数据中，计算“过去一个季度，每个省份的销售总额 TOP 10 的商品品类”。这种查询需要扫描、过滤、聚合海量数据，正是 ClickHouse 的用武之地，可能在数秒内返回结果。HBase 自身无法完成，需要借助 Spark、Flink 等外部计算引擎，耗时可能是分钟甚至小时级别。

## HBase为什么在OLAP场景下慢？——为OLTP而生的KV模型

- **设计原理:** HBase 的核心是基于 RowKey 的有序 Key-Value 存储。

- 单点/范围查询:** 当你通过 RowKey 进行精确 Get 或小范围 Scan 时，性能极高。
  - 非 RowKey 查询:** 如果你的查询条件不涉及 RowKey，HBase 只能进行全表扫描，性能极其低下。
  - 无内置聚合能力:** HBase 本身只是一个存储系统，不具备复杂的计算能力。任何聚合分析都必须在客户端或通过外部计算框架完成，这意味着需要将海量数据从 HBase 中读出，通过网络传输到计算节点，效率很低。
- 举例:** 同样是上面的聚合查询，HBase 需要一个 Spark 作业来读取全表数据，进行 shuffle 和计算，整个流程非常重。但是，如果查询是“获取订单号为 order\_123 的详细信息”（订单号是 RowKey），HBase 可以在几毫秒内完成。

最终总结	
HBase (基于LSM-Tree)	ClickHouse (基于MergeTree)
设计定位 OLTP/实时读写 NoSQL数据库	OLAP/实时分析 数据库
新增性能 极高 (尤其适合高频、单条/小批量)	高 (只适合大批量写入)
删除/更新性能 极高 (轻量级的标记写入)	极低 (重量级的异步数据重写)
查询性能 KV式查询极高 (基于RowKey)   分析查询极低 (需全表扫描)	KV式查询一般   分析/聚合查询极高 (列存+向量化)
一句话比喻 一个**“记账神速的档案柜”**，按编号 (RowKey) 存取文件飞快，但要统计所有文件的内容就得全翻一遍。	一个**“超级计算机驱动的分析引擎”**，存文件需要整批存，但一旦存进去，就能瞬间对所有文件的任意内容进行复杂的统计分析。

## CK VS ES

核心结论速览		
操作维度	性能更高者	核心设计原理差异
新增 (Create)	ClickHouse (在大批量场景下通常更高)	MergeTree (批量列存构建) vs. Lucene (实时倒排索引构建)，ES的索引构建更复杂。
删除 (Delete)	两者机制类似，性能相当	都是**“写标记+后台合并”**的“假删除”模式。
更新 (Update)	两者性能都很差，不相上下	CK: 异步重写数据部分(Part)。 ES: “读-删-写”组合。两者都非原生支持，代价高昂。
查询 (Read)	高度依赖场景，各有千秋	原生列存 + 向量化执行引擎，为海量数据聚合、计算而生。
	ClickHouse (OLAP聚合分析)	倒排索引，为文本搜索和复杂条件过滤做了极致优化。
	Elasticsearch (全文检索/复杂搜索)	

### 一、新增数据 (Create / Insert / Index)

**结论：**对于海量数据的批量写入，ClickHouse 的吞吐量通常更高。对于低延迟、实时性要求高的单条写入，两者性能接近，ES 可能略有优势。

ClickHouse 为什么在高吞吐写入上更快？—— MergeTree 的批量优化

- 设计原理:** CK 的核心引擎 MergeTree 是为批量写入和后续的快速分析而设计的。
  - 数据分 Part 写入:** INSERT 的数据会被组织成一个按主键排序的数据部分 (Data Part)。在这个过程中，数据会被直接转换成高度压缩的列式格式。
  - 延迟合并:** CK 不要求每次写入都让数据立即可见并与其他数据合并。它允许磁盘上存在大量小的、临时的 Parts，然后通过一个后台线程异步地、高效

地将它们合并 (Merge) 成更大的 Part。

- 3. 索引结构简单: MergeTree 的主键索引是稀疏索引, 它只记录每个数据块 (Granule, 默认 8192 行) 的起始位置。相比 ES 复杂的倒排索引, 这个索引的构建和维护成本要低得多。

- **性能优势:** 整个写入过程非常“粗线条”, 主要是数据转换、压缩和顺序写。它将大量的合并和索引维护工作放到了后台, 从而最大化了写入的吞吐能力。尤其适合一次性写入几十万甚至上百万行的大数据块。

#### Elasticsearch 为什么在写入时开销更大? —— 倒排索引的实时性代价

- **设计原理:** ES 基于 **Lucene**, 其首要任务是让新写入的数据尽快变得可被搜索。
  1. **复杂的索引构建:** 每一篇文档写入时, 都需要经过分析 (**Analysis**) 过程 (分词、词干提取等), 然后为其中的每个词项 (**Term**) 更新倒排索引。这是一个 **CPU 密集型** 的操作。
  2. **多种数据结构:** 除了倒排索引, ES 还需要为排序、聚合等功能维护\*\*列存 (Doc Values) 和存储字段 (Stored Fields)\*\*等多种数据结构。
  3. **Refresh 的准实时性:** 为了让数据尽快可被搜索, ES 会定期 (`index.refresh_interval`, 默认 1 秒) 执行 `refresh` 操作, 将内存缓冲区的数据生成一个新的\*\*段 (Segment)\*\* 并开放给搜索。这个过程是有性能开销的。
- **性能特点:** ES 的写入链路涉及更多的计算和更复杂的数据结构维护, 因此单次写入的开销比 CK 更高。但它的优势在于**准实时性**, 数据写入后通常在 1 秒内即可被搜到。

**总结:** CK 的写入像“批量卸货”, 一次拉一大车, 效率最高。ES 的写入像“精加工入库”, 每件货物都要贴好各种标签 (索引), 方便以后查找, 所以单件处理时间更长。

## 二、删除 & 更新数据 (Delete & Update)

**结论:** 两者在这方面性能都很差, 因为它们的核心数据结构都是“不可变的” (**Immutable**), 都不支持真正的“就地”更新。它们是半斤八两, 都不适合频繁删改的场景。

#### ClickHouse 的删改: 重量级的后台重写

- **设计原理:** `ALTER TABLE ... DELETE/UPDATE` 操作是一种 **Mutation**。
  - **异步执行:** 它不会立即修改数据。相反, 它会在后台重写所有包含需要修改数据的 Data Parts。
  - **过程:** 读取一个旧 Part -> 在内存中应用修改 / 删除 -> 写一个新的 Part -> 最终用新 Part 替换旧 Part。
- **性能劣势:** 这是一个非常重的 I/O 密集型操作, 其开销与被修改 Part 的大小成正比, 而不是与被修改的行数成正比。修改 1 行和修改 100 万行, 如果它们在同一个大 Part 里, 开销可能是一样的。

#### Elasticsearch 的删改: 昂贵的“读-删-写”组合

- **设计原理:** Lucene 的 Segment 也是不可变的。
  - **删除 (Delete):** 只是在一个.del 文件中记录一个“墓碑标记”。查询时会过滤掉这些被标记的文档。
  - **更新 (Update):** 是一个开销极大的组合操作:
    1. 读取出旧文档。
    2. 将旧文档标记为删除。
    3. 将新文档作为一个全新的文档执行一次完整的新增流程。
- **性能劣势:** 更新操作的代价包含了读、写和索引构建的全部开销, 非常昂贵。大

量的删除标记也会拖慢查询性能，需要等待后台的 Segment Merging 来清理。  
**总结：**对于删改，CK 和 ES 都是“惹不起”的。如果你有频繁更新的需求，应该选择 MySQL、PostgreSQL 或 HBase 这样的 OLTP 数据库。

### 三、查询数据 (Read)

**结论：**两者都是顶级的查询引擎，但术业有专攻。ClickHouse 是 OLAP 聚合分析之王，Elasticsearch 是全文检索和复杂搜索之王。

**ClickHouse 的优势场景：OLAP 聚合分析**

- **设计原理：**
  1. **原生列式存储：**这是其最大优势。当进行聚合查询（如 SUM, AVG, GROUP BY）时，只需读取相关的几列，I/O 效率极高。
  2. **向量化执行引擎：**利用 CPU 的 SIMD 指令，以“列批次”为单位进行计算，而不是一行一行地执行，将 CPU 的计算能力发挥到极致。
  3. **数据压缩：**高效的列压缩算法进一步减少了 I/O。
- **性能优势：**对于需要扫描海量数据、进行数值计算和分组聚合的典型 OLAP 查询，CK 的速度通常会显著快于 ES。
- **举例：**“计算过去一年，每个商品类目下，所有男性用户的平均客单价和总销售额”。这种查询对 CK 来说是家常便饭。

**Elasticsearch 的优势场景：全文检索与复杂搜索**

- **设计原理：**
  1. **倒排索引：**这是 ES 的灵魂。它提供了从“词”到“文档”的超快速映射，使得任何文本搜索、term 或 match 查询都能在毫秒级完成。
  2. **丰富的查询 DSL：**支持极其灵活的布尔组合查询（bool query）、相关性算分、地理位置搜索、模糊查询等，查询能力远比 CK 的 SQL 强大和灵活。
- **性能优势：**对于文本相关的查询，或者需要在大量字段上进行复杂条件组合过滤的场景，ES 的性能和能力是 CK 无法比拟的。
- **举例：**“在一个电商网站搜索所有标题包含‘笔记本电脑’，品牌是‘苹果’或‘联想’，价格在 5000-8000 之间，并且按用户好评度排序的商品”。这种查询是 ES 的典型应用。

最终总结	
设计定位	ClickHouse (基于MergeTree) OLAP分析引擎 / 数据仓库
新增性能	批量吞吐量极高 极差 (重量级后台重写)
删除/更新性能	文本搜索能力弱 极差 (昂贵的读-删-写)
查询性能	OLAP聚合分析极强 (列存+向量化)   文本搜索能力弱 全文检索/复杂搜索极强 (倒排索引)   聚合分析能力强
一句话比喻	一个***“数据统计学家”***，不善于修改旧账本，但给他一大堆账本，他能以最快速度算出各种财务报表。 一个***“图书管理员”***，整理新书（写入）要花点时间，但你告诉他任何关键词，他都能瞬间帮你找到所有相关的书籍。

## 高可用设计方向

1. 集群配置，避免单点故障（当然也能提高处理能力）。
2. 限流：流量控制（flow control），其原理是监控应用流量的 QPS 或并发线程数等

指标，当达到指定的阈值时对流量进行控制，以避免被瞬时的流量高峰冲垮，从而保障应用的高可用性。

### 3、超时和重试机制

超时机制：可以防止资源长时间被占用。

重试：可以解决由于网络不稳定、服务节点故障等原因导致的请求失败，通过重试可以提高调用成功的概率，提高系统的可靠性。

### 4、熔断机制：

熔断机制说的是系统自动收集所依赖服务的资源使用情况和性能指标，当所依赖的服务恶化或者调用失败次数达到某个阈值的时候就迅速失败，让当前系统立即切换依赖其他备用服务。例如得物接口上的 BlockException 注解配置，当超过一定的限流阈值，就会走到异常提示。

### 5、异步

异步调用的话我们不需要关心最后的结果，这样我们就可以用户请求完成之后就立即返回结果，具体处理我们可以后续再做，秒杀场景用这个还是蛮多的。但是，使用异步之后我们可能需要适当修改业务流程进行配合，比如用户在提交订单之后，不能立即返回用户订单提交成功，需要在消息队列的订单消费者进程真正处理完该订单之后，甚至出库后，再通过电子邮件或短信通知用户订单成功。除了可以在程序中实现异步之外，我们常常还使用消息队列，消息队列可以通过异步处理提高系统性能（削峰、减少响应所需时间）并且可以降低系统耦合性。

### 6、缓存

并发请求比较高的情况下做缓存设计，避免大量请求打到数据库，对服务的响应能力造成影响。

### 7、监控系统

能实时监测系统运行状态，保证发生问题之前能提前被通知并且即使进行止血操作。

### 8、蓝绿发布

保证服务丝滑发布，避免影响线上服务使用。

## 高可用设计之冗余设计

高可用集群 (High Availability Cluster，简称 HA Cluster)、同城灾备、异地灾备、同城多活和异地多活是冗余思想在高可用系统设计中最典型的应用。

- **高可用集群**：同一份服务部署两份或者多份，当正在使用的服务突然挂掉的话，可以切换到另外一台服务，从而保证服务的高可用。
- **同城灾备**：一整个集群可以部署在同一个机房，而同城灾备中相同服务部署在同一个城市的不同机房中。并且，备用服务不处理请求。这样可以避免机房出现意外情况比如停电、火灾。
- **异地灾备**：类似于同城灾备，不同的是，相同服务部署在异地（通常距离较远，甚至是在不同的城市或者国家）的不同机房中
- **同城多活**：类似于同城灾备，但备用服务可以处理请求，这样可以充分利用系统资源，提高系统的并发。
- **异地多活**：将服务部署在异地的不同机房中，并且，它们可以同时对外提供服务。

**【灾备和多活区别：灾备的备用服务不支持处理请求，而多活的备用服务支持处理请求】**

光做好冗余还不够，必须要配合上 故障转移 才可以！ 所谓故障转移，简单来说就是实现不可用服务快速且自动地切换到可用服务，整个过程不需要人为干涉。

异地多活博客：

- 1、<https://mp.weixin.qq.com/s/T6mMDdtTfBuliEowCpqu6Q>
- 2、[https://mp.weixin.qq.com/s/hMD-IS\\_\\_4JE5\\_nQhYPYSTg](https://mp.weixin.qq.com/s/hMD-IS__4JE5_nQhYPYSTg)

## 高可用设计之服务限流

### 1. 固定窗口计数器 (Fixed Window Counter)

- **核心原理:** 将时间划分为固定的、不重叠的窗口（例如，每秒一个窗口）。在每个窗口内，维护一个计数器，当请求进入时计数器加一。如果计数超过预设阈值，则拒绝该窗口内的后续请求。窗口结束时，计数器重置。
- **优点:** 实现简单，易于理解。
- **缺点:** 存在\*\*“临界问题”\*\*。在两个窗口的边界处，可能出现瞬时流量远超限制的情况（例如，第1秒的最后100ms和第2秒的最初100ms都涌入了大量请求），导致限流不够平滑，无法有效保护系统。
- **适用场景:** 对限流精度要求不高的简单场景。

### 2. 滑动窗口计数器 (Sliding Window Counter)

- **核心原理:** 这是固定窗口的改进版。它将一个大的时间窗口（例如1分钟）细分为多个更小的格子（例如6个10秒的格子）。随着时间的推移，窗口会“滑动”地抛弃最旧的格子，并纳入新的格子。限流决策依据是当前窗口内所有格子的计数总和。
- **优点:** 通过平滑地移动统计区间，完美解决了固定窗口的“临界问题”，使得限流控制更加平滑和精确。
- **缺点:** 实现相对复杂，且需要占用更多的内存来存储多个小格子的计数。
- **适用场景:** 需要精确、平滑流控的场景，如API网关、微服务间的流量控制。

### 3. 漏桶算法 (Leaky Bucket)

- **核心原理:** 将所有进入的请求视为水流，先放入一个容量固定的“漏桶”中进行排队。系统以一个绝对恒定的速率从桶底处理（漏出）请求。如果请求到达时桶已满，则直接丢弃或拒绝。
- **优点:** 能够实现强制的流量整形 (Traffic Shaping)，确保了处理请求的速率绝对平滑和固定，能有效保护处理能力有限的下游系统。
- **缺点:** 无法应对任何突发流量（所有请求必须排队等待匀速处理），导致请求处理有延迟，对系统资源的利用率不高。
- **适用场景:** 主要用于保护下游服务，例如，需要严格控制调用频率的第三方API接口、短信网关等。

### 4. 令牌桶算法 (Token Bucket)

- **核心原理:** 系统以一个恒定的速率，不断地向一个容量固定的“令牌桶”中放入令牌。每个进入的请求都必须先从桶中获取一个令牌，拿到令牌后才能被处理。如果桶中没有令牌，请求则需等待或被拒绝。
- **优点:** 既能限制长期的平均速率（由令牌生成速率决定），又能有效地应对短时间的突发流量（通过消耗桶中积攒的令牌）。这是其相比漏桶算法的最大优势，资源利用率更高。
- **缺点:** 实现相对漏桶更复杂一些，需要合理配置令牌生成速率和桶容量，否则可能导致突发流量超出系统承载能力。
- **适用场景:** 绝大多数Web应用、API网关和微服务限流场景的事实标准\*\*。  
\*\* 它在控制平均速率和允许合理突发之间取得了最佳平衡。

## 一句话速记

- 固定窗口: 简单粗暴, 但有致命缺陷。
  - 滑动窗口: 精准平滑, 是固定窗口的完美升级。
  - 漏桶: 削峰填谷, 保证下游绝对安逸 (速率恒定)。
  - 令牌桶: 能扛能打, 平时稳健 (限制均速), 战时能爆发 (应对突发)。
- 
- 

## 性能优化思路? 找到性能瓶颈【超哥提供的研究方向】

- 1.硬件层面 cpu 内存 硬盘 网络
- 2.软件层面 数据库 sql 中间件 (查询缓存 mq 解耦 nginx 负载均衡) 操作系统 代码 算法 (时间复杂度) 架构
- 3.意识层面 需求设计 测试
- 4.业务层面 实施方案

高可用思路?

- 1.构建冗余 主从同步 避免单点
- 2.负载均衡 多个机器同时提供服务
- 3.限流和降级
- 4.隔离 线程池 多个进程 集群化 多机房 读写分离
- 5.超时和重试机制
- 6.压测和预案

高并发思路

- 1.缓存
- 2.线程池
- 3.异步化 非阻塞
- 4.水平扩容 垂直扩容
- 5.最终一致性 (减少锁 分布式事务的使用) 削峰填谷

**SLA关键指标: 可用性、准确性、容量、延迟**

---

---

---

---

---

---

---

---

---

---

---