# 背景

Innodb**为了解决磁盘上磁盘速度和CPU速度不一致的问题**，在操作磁盘上的数据时，先将数据加载至内存中，在内存中对数据页进行操作。

MySQL在启动的时候，会向内存申请一块**连续**的空间，这块空间名为Bufffer Pool，也就是缓冲池，**默认情况下Buffer Pool只有128M**。

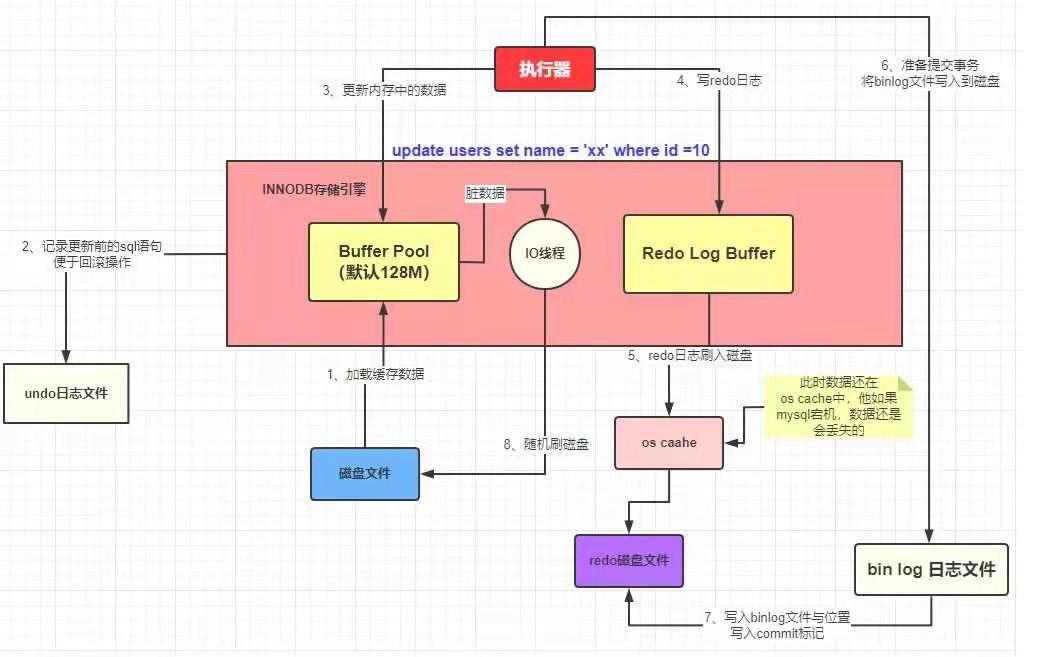
# 概述

## 简介

Buffer Pool是什么？从字面上看是缓存池的意思，没错，它其实也就是缓存池的意思。它是MySQL当中至关重要的一个组件，可以这么说，**MySQL的所有的增删改的操作都是在Buffer Pool中执行的**。

但是数据不是在磁盘中的吗？怎么会和缓存池又有什么关系呢？那是因为如果MySQL的操作都在磁盘中进行，那很显然效率是很低的，效率为什么低？因为数据库要从磁盘中拿数据，那肯定就需要IO，并且数据库并不知道它将要查找的数据是磁盘的哪个位置，所以这就需要进行随机IO，那样性能会很差。所以MySQL对数据的操作都是在内存中进行的，也就是在Buffer Pool这个内存组件中。

实际上他就好比是Redis，因为Redis是一个内存是数据库，他的操作就都是在内存中进行的，并且会有一定的策略将其持久化到磁盘中。那Buffer Pool的内存结构具体是什么样子的，那么多的增删改操作难道数据要一直在内存中吗？既然说类似redis缓存，那是不是也像 redis一样也有一定的淘汰策略呢？



注：从磁盘加载数据到内存中之后，不是先更新内存中的数据，而是WAL（日志先行），先写undo日志（回滚时候使用）。

参考：

<https://mp.weixin.qq.com/s/u6J0xv1YmBjs8KywKW_3wg>

注：与之不同的是，kafka效率高反而是因为写磁盘，主要是因为kafka是顺序追加的方式写磁盘，不存在随机读写，这种直接写磁盘反而效率高。

## 内存分配器

在MySQL中，buffer pool的内存，是通过mmap()方式直接向操作系统申请分配；除此之外，大多数的内存管理，都需要经过内存分配器。为了实现更高效的内存管理，避免频繁的内存分配与回收，内存分配器会长时间占用大量内存，以供内部重复使用。关于内存分配器的选择，推荐使用jemalloc，可以有效解决内存碎片与提升整体性能。  
 因此，MySQL占用内存高的原因可能包括：innodb\_buffer\_pool\_size设置过大、连接数/并发数过高、大量排序操作、内存分配器占用、以及MySQL Bug等等。一般来说，在MySQL整个运行周期内，刚启动时内存上涨会比较快，运行一段时间后会逐渐趋于平稳，这种情况是不需要过多关注的；如果在稳定运行后，出现内存突增、内存持续增长不释放的情况，那就需要我们进一步分析是什么原因造成的。

## MEM\_ROOT

以MySQL的架构来划分内存管理比较合理。即Server层与InnoDB层（Engine 层），而这两块内存是由不同的方式进行管理的。

其中Server层是由mem\_root来进行内存管理，包括Sharing与Thead memory；而InnoDB层则主要由Free List、LRU List、FLU List等多个链表来统一管理 Innodb\_buffer\_pool。

**Buffer pool**

InnoDB存储引擎的内存管理方式。

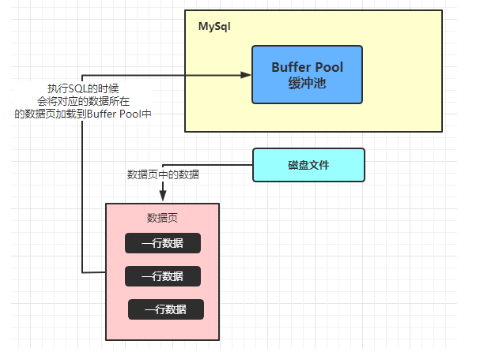
**Tcmalloc**

Linux系统的内存优化器。

# 原理

## 数据页

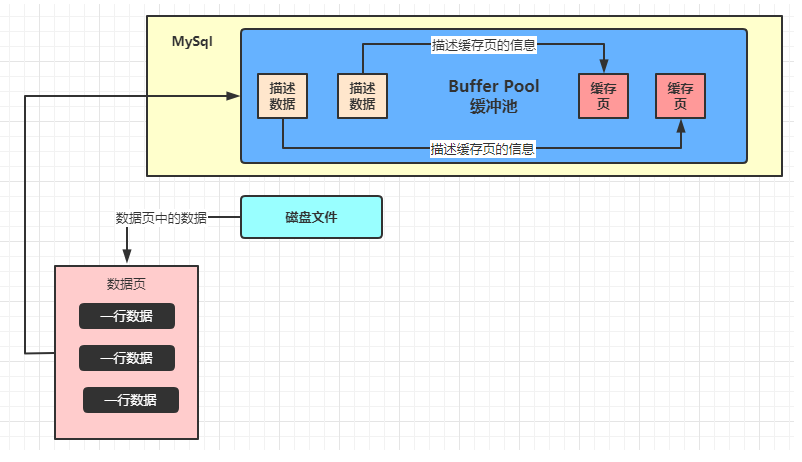
 MySQL在执行增删改的时候数据是会被加载到Buffer Pool中的，既然这样数据是怎么被加载进来的，是一条一条还是说是以其他的形式呢。我们操作的数据都是以表+行的方式，而表+行仅仅是逻辑上的概念，MySQL并不会像我们一样去操作行数据，而是抽象出来一个一个的**数据页**概念，**每个数据页的大小默认是16KB**，这些参数都是可以调整的。但是**建议使用默认的就好**，毕竟MySQL能做到极致的都已经做了。每个数据页存放着多条的数据，MySQL在执行增删改首先会定位到这条数据所在数据页，然后会将数据所在的数据页加载到Buffer Pool中。



## 缓存页

当数据页被加载到缓冲池中后，Buffer Pool中也有叫缓存页的概念与其一一对应，大小同样是16KB，但是 MySQL还为每个缓存也开辟额外的一些空间，用来描述对应的缓存页的一些信息，例如：数据页所属的表空间，数据页号，这些描述数据块的大小大概是缓存页的15%左右（约800B）。

**# 缓存页是什么时候被创建的？** 当MySql启动的时候，就会初始化Buffer Pool，这个时候MySQL会根据系统中设置的innodb\_buffer\_pool\_size大小去内存中申请一块连续的内存空间，实际上在这个内存区域比配置的值稍微大一些，因为【描述数据】也是占用一定的内存空间的，当在内存区域申请完毕之后，MySql会根据默认的缓存页的大小（16KB）和对应`缓存页\*15%`大小(800B左右)的数据描述的大小，将内存区域划分为一个个的缓存页和对应的描述数据。

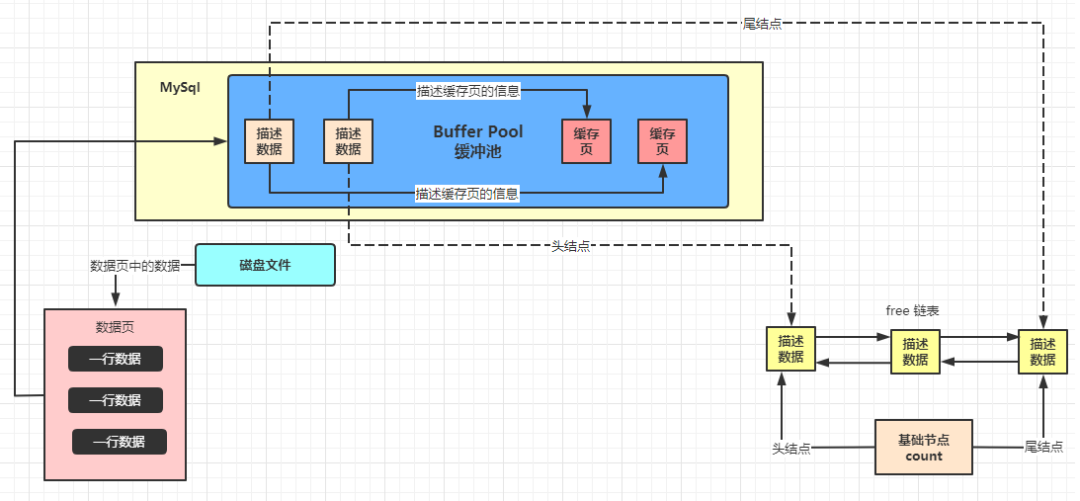


## Free链表

上面是说了每个数据页会被加载到一个缓存页中，但是加载的时候 MySQL是如何知道那个缓存页有数据，那个缓存页没有数据呢？换句话说， MySQL是怎么区分哪些缓存页是空闲的状态，是可以用来存放数据页的。

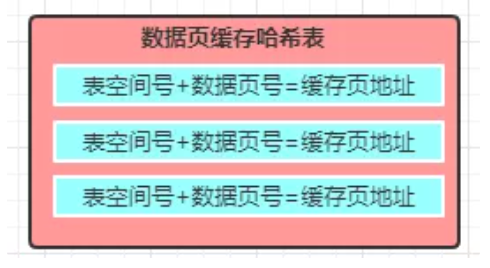
为了解决这个问题，MySQL为Buffer Pool设计了一个双向链表—free链表，这个free链表的作用就是用来保存空闲缓存页的描述块（这句话这么说其实不严谨，换句话：每个空闲缓存页的描述数据组成一个双向链表，这个链表就是free链表）。之所以说free链表的作用就是用来保存空闲缓存页的描述数据是为了先让大家明白free链表的作用，另外free链表还会有一个基础节点，他会引用该链表的头结点和尾结点，还会记录节点的个数（也就是可用的空闲的缓存页的个数）。

这个时候，他可以用下面的图片来描述：



当加载数据页到缓存池中的时候，MySQL会从free链表中获取一个描述数据的信息，根据描述节点的信息拿到其对应的缓存页，然后将数据页信息放到该缓存页中，同时将链表中的该描述数据的节点移除。这就是数据页被读取Buffer Pool中的缓存页的过程。

但MySQL是怎么知道哪些数据页已经被缓存了，哪些没有被缓存呢。实际上数据库中还有后一个哈希表结构，他的作用是用来存储表空间号+数据页号作为数据页的key，缓存页对应的地址作为其value，这样数据在加载的时候就会通过哈希表中的key来确定数据页是否被缓存了。



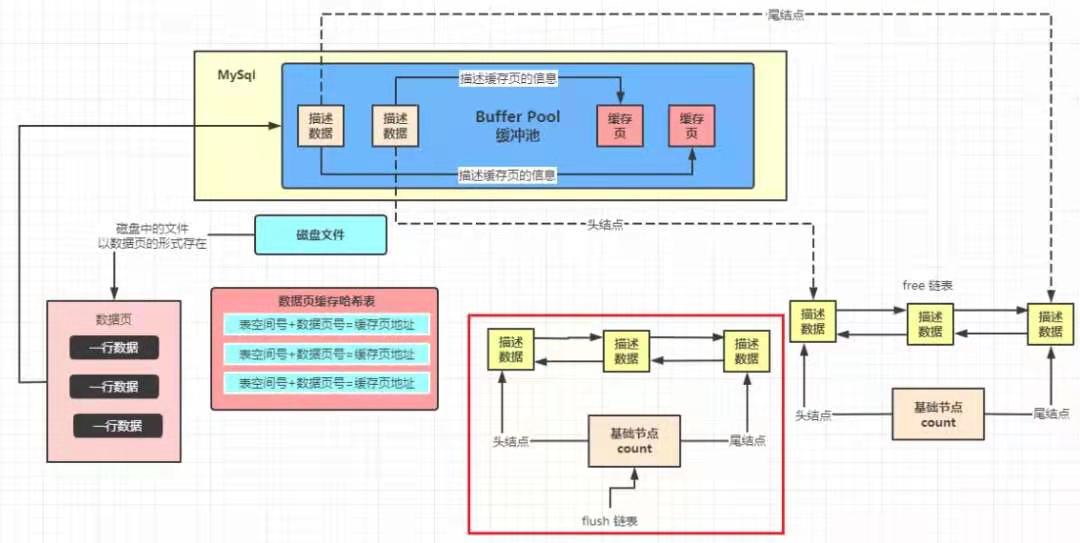
## Flush链表

MySql在执行增删改的时候会一直将数据以数据页的形式加载到Buffer Pool的缓存页中，增删改的操作都是在内存中执行的，然后会有一个后台的线程数将脏数据刷新到磁盘中，但是后台的线程肯定是需要知道应该刷新哪些。

针对这个问题，MySQL设计出了Flush链表，其作用就是记录被修改过的脏数据所在的缓存页对应的描述数据。如果内存中的数据和数据库和数据库中的数据不一样，那这些数据我们就称之为脏数据，脏数据之所以叫脏数据，本质上就是被缓存到缓存池中的数据被修改了，但是还没有刷新到磁盘中。

同样的这些已经被修改了的数据所在的缓存页的描述数据会被维护到Flush中（其实结构和free链表是一样的），所以Flush中维护的是一些脏数据数据描述（准确地说是脏数据的所在的缓存页的数据描述）。

另外，当某个脏缓存页被刷新到磁盘后，其空间就腾出来了，然后又会跑到Free链表中了。



## LRU链表

如果系统一直在进行数据库的增删改操作，数据库内部的基本流程就是：

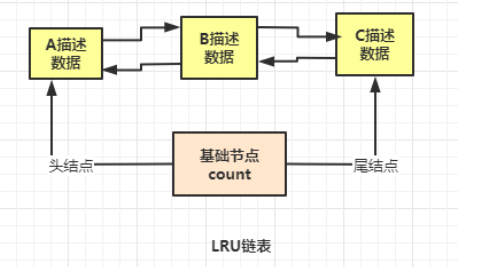
我们还拿redis类做类比，以便更好的帮助大家明白其原理。Flush的作用其实类似redis的key设置的过期时间，所以一般情况下，redis内存不会不够使用，但是总有特殊的情况，问题往往就是在这种极端和边边角角的情况下产生的。

如果redis的内存不够使用了，是不是自己还有一定的淘汰策略？最基本的准则就是淘汰掉不经常使用到的key。Buffer Pool也类似，它也会有内存不够使用的情况，它是通过LRU链表来维护的。LRU即Least Recently Uesd（最近最少使用）。

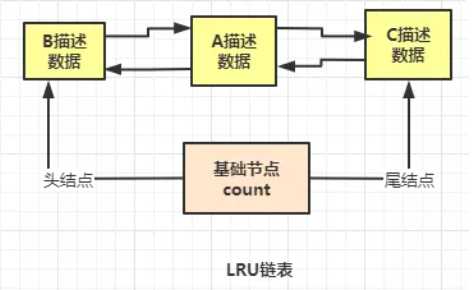
MySql会把最近使用最少的缓存页数据刷入到磁盘去，那MySql如何判断出LRU数据的呢？为此MySql专门设计了LUR链表，还引入了另一个概念：缓存命中率

**# 缓存命中率** 可以理解为缓存被使用到的频率，举个例子来说：现在有两个缓存页，在100次请求中A缓存页被命中了20次，B缓存页被命中了2次，很显然A缓存页的命中率更高，这也就意味着A在未来还会被使用到的可能性比较大，而B就会被MySQL认为基本不会被使用到；

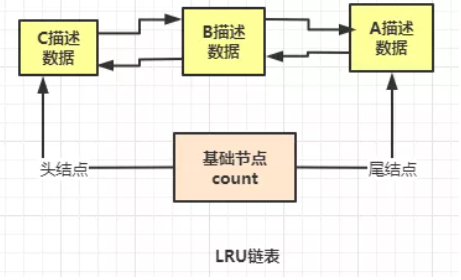
说到这里，那LRU究竟是怎么工作的。假设MySQL在将数据加载到缓存池的时候，他会将被加载进来的缓存页按照被加载进来的顺序插入到LRU链表的头部（就是链表的头插法），假设MySQL现在先后分别加载A、B、C数据页到缓存页A、B、C中，然后LRU的链表大致是这样子的。



现在又来了一个请求，假设查询到的数据是已经被缓存在缓存页B中，这时候 MySQL就会将B缓存页对应的描述信息插入到LRU链表的头部，如下图：

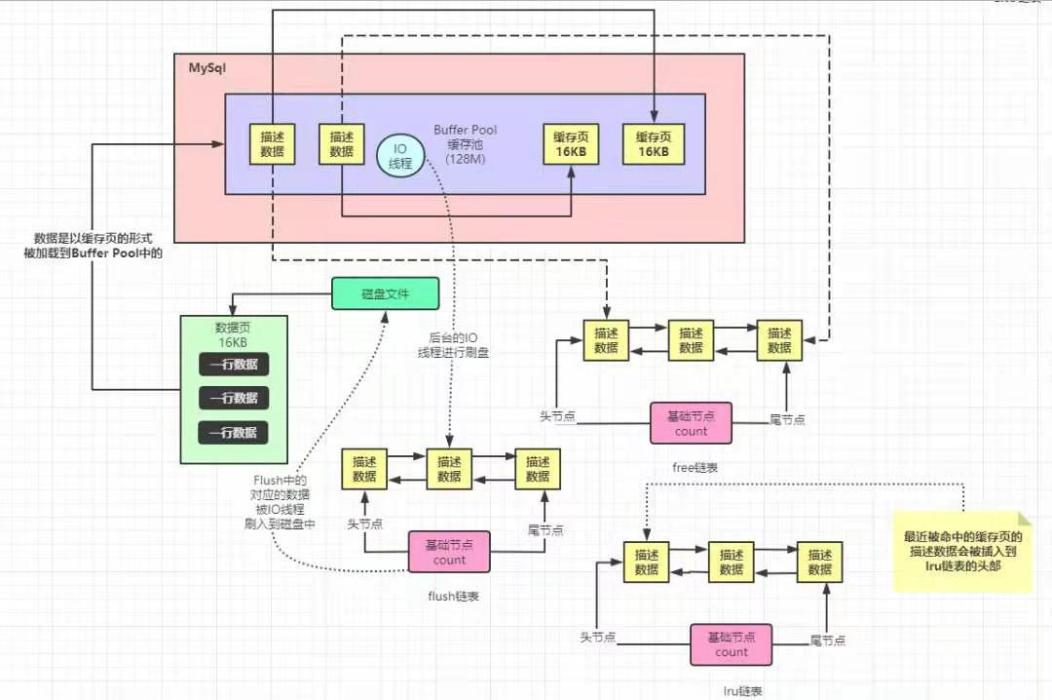


然后又来了一个请求，数据是已经被缓存在了缓存页C中，然后LRU会变成这样子：



说到底，每次查询数据的时候如果数据已经在缓存页中，那么就会将该缓存页对应的描述信息放到LRU链表的头部，如果不在缓存页中，就去磁盘中查找，如果查找到了，就将其加载到缓存中，并将该数据对应的缓存页的描述信息插入到LRU链表的头部。也就是说最近使用的缓存页都会排在前面，而排在后面的说明是不经常被使用到的。

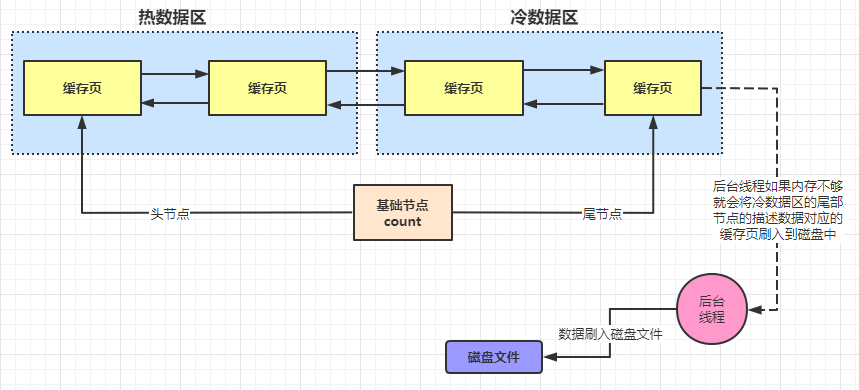
最后，如果Buffer Pool不够使用了，那么 MySQL就会将LRU链表中的尾节点刷入到磁盘中，用来给Buffer Pool腾出内存空间。来个整体的流程图给大家看下



## 总结

# free链表  
 用来存放空闲的缓存页的描述数据，如果某个缓存页被使用了，那么该缓存页对应的描述数据就会被从free链表中移除  
 # flush链表

被修改的脏数据都记录在Flush中，同时会有一个后台线程会不定时的将Flush中记录的描述数据对应的缓存页刷新到磁盘中，如果某个缓存页被刷新到磁盘中了，那么该缓存页对应的描述数据会从Flush中移除，同时也会从LRU链表中移除（因为该数据已经不在Buffer Pool 中了，已经被刷入到磁盘，所以就也没必要记录在LRU链表中了），同时还会将该缓存页的描述数据添加到free链表中，因为该缓存页变得空闲了。  
 # LRU链表  
 数据页被加载到Buffer Pool中的对应的缓存页后，同时会将缓存页对应的描述数据放到LRU链表的冷数据的头部，当在一定时间过后，冷数据区的数据被再次访问了，就会将其转移到热数据区链表的头部，如果被访问的数据就在热数据区，那么如果是在前25%就不会移动，如果在后75%仍然会将其转移到热数据区链表的头部



# 并发性能

我们平时的系统绝对不可能每次只有一个请求来访问的，说白了就是如果多个请求同时来执行增删改，那他们会并行的去操作Buffer Pool中的各种链表吗？如果是并行的会不会有什么问题。

实际上MySQL在处理这个问题的时候考虑的非常简单，就是：**Buffer Pool一次只能允许一个线程来操作**，一次只有一个线程来执行这一系列的操作，因为**MySQL为了保证数据的一致性，操作的时候必须缓存池加锁，一次只能有一个线程获取到锁**。

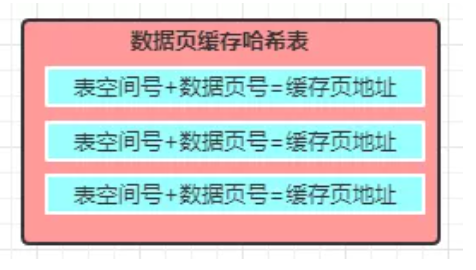
串行那还谈什么效率？大家别忘记了，这一系列的操作都是在内存中操作的，实际上这是一个瞬时的过程，在内存中的操作基本是几毫秒的甚至微妙级别的事情。

但是话又说回来，串行执行再怎么快也是串行，虽然不是性能瓶颈，这还有更好的优化办法吗？那肯定的MySQL早就设计好了这些规则。那就是Buffer Pool是可以有多个的，可以通过MySQL的配置文件来配置，参数分别是：

# Buffer Pool 的总大小  
 innodb\_buffer\_pool\_size=8589934592  
 # Buffer Pool 的实例数（个数）  
 innodb\_buffer\_pool\_instance=4

一般在生产环境中，在硬件不紧张的情况下，建议使用此策略。这个时候大家是不是又会有一个疑问，大家应该有这样的疑问：

# 问：多个Buffer Pool所带来的问题思考  
 在多个线程访问不同的Buffer Pool那不同的线程加载的数据必然是在不同的Buffer Pool中，假设A线程加载数据页A到Buffer Pool A中，B线程加载数据页B到Buffer Pool B中，然后两个都执行完了，这个时候C线程来了，他到达的是Buffer Pool B中，但是C要访问的数据是在 Buffer Pool A中的数据页上了，这个时候C还会去加载数据页A吗？，这种情况会发生吗？在不同的 Buffer Pool 缓存中会去缓存相同的数据页吗？  
 # 答：多个Buffer Pool所带来的问题解答  
 这种情况很显然不会发生，既然不会发生，那MySql是如何解决这种问题的？其实前面已经提到过了，那就是数据页缓存哈希表，里面存放的是表空间号+数据页号=缓存页地址，所以MySQL在加载数据所在的数据页的时候根据这一系列的映射关系判断数据页是否被加载，被加载到了那个缓存页中，所以MySQL能够精确的确定某个数据页是否被加载，被加载的到了哪个缓存页，绝不可能出现重复加载的情况。



# 参数设置

在MySQL中，内存占用主要包括以下几部分，全局共享的内存、线程独占的内存、内存分配器占用的内存，具体如下：

## 全局共享

### innodb\_buffer\_pool\_instance

参考：

[https://mp.weixin.qq.com/s?\_\_biz=MzU2NzgwMTg0MA==&mid=2247492455&idx=1&sn=8f2a266870fdffc5f33e5f69c0d94986&chksm=fc9501f8cbe288eedbc9bc023475054d08a1afb3090a69a2ef7e2611f9a7d673828e093b8226&mpshare=1&scene=24&srcid=0326ssV9gC1mdYtX2hoMLeMl&sharer\_sharetime=1616752871601&sharer\_shareid=33f795d236f19ac7c128b2e279563f84#rd](https://mp.weixin.qq.com/s?__biz=MzU2NzgwMTg0MA==&mid=2247492455&idx=1&sn=8f2a266870fdffc5f33e5f69c0d94986&chksm=fc9501f8cbe288eedbc9bc023475054d08a1afb3090a69a2ef7e2611f9a7d673828e093b8226&mpshare=1&scene=24&srcid=0326ssV9gC1mdYtX2hoMLeMl&sharer_sharetime=1616752871601&sharer_shareid=33f795d236f19ac7c128b2e279563f84" \l "rd)

随着MySQL使用的内存越来越大，我们建议使用多个buffer pool instance。

那么我们的问题是: 一张表有多少在buffer pool中，一张表只能在一个buffer pool instance中么？

为什么buffer pool需要使用多个POOL?

访问buffer pool时需要上锁，只是用一个POOL，锁冲突比较严重。使用多个POOL，可以分担锁的冲突压力。

一张表的各个页为什么交替出现在各个POOL中？

为了让各个POOL中的数据量相对平衡。

那为什么不是一页一轮换，而是64页一轮换？

我们访问数据，经常扫描连续的多个页。如果一页一轮换，那我们一次扫描就要涉及多个POOL，那么锁的冲突压力就不得分担，迷失了最初的目标。

### innodb\_buffer\_pool\_size

innodb\_buffer\_pool\_size：InnoDB缓冲池的大小

Buffer Pool是InnoDB中的一块内存区域，他一定是有自己的大小的，且大小默认是128M，不过这个容量似乎有点小了，大家的自己的生产环境可以根据实际的内存大小进行调整，参数为：innodb\_buffer\_pool\_size=2147483648 ，单位是字节。

# 查看和调整innodb\_buffer\_pool\_size  
 1、查看@@innodb\_buffer\_pool\_size大小，单位字节  
 SELECT @@innodb\_buffer\_pool\_size/1024/1024/1024; #字节转为G  
 2、在线调整InnoDB缓冲池大小，如果不设置，默认为128M  
 set global innodb\_buffer\_pool\_size = 4227858432; ##单位字节

### innodb\_additional\_mem\_pool\_size

innodb\_additional\_mem\_pool\_size：InnoDB存放数据字典和其他内部数据结构的内存大小，5.7已被移除。

### innodb\_log\_buffer\_size

innodb\_log\_buffer\_size：InnoDB日志缓冲的大小

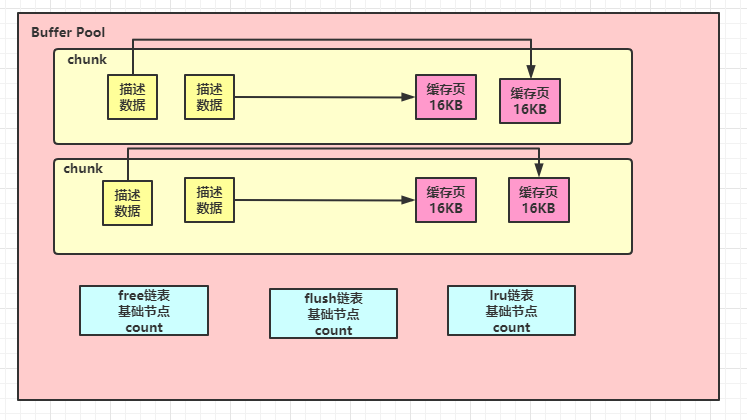
### innodb\_buffer\_pool\_chunk\_size

假设我们现在的Buffer Pool的大小是2GB大小，现在想将其扩大到4GB，现在说一下如果真的要这么做，我们的MySq需要做哪些事情。首先，MySQL需要向操作系统申请一块大小为4G的连续的地址连续的内存空间，然后将原来的Buffer Pool中的数据拷贝到新的Buffer Pool中。

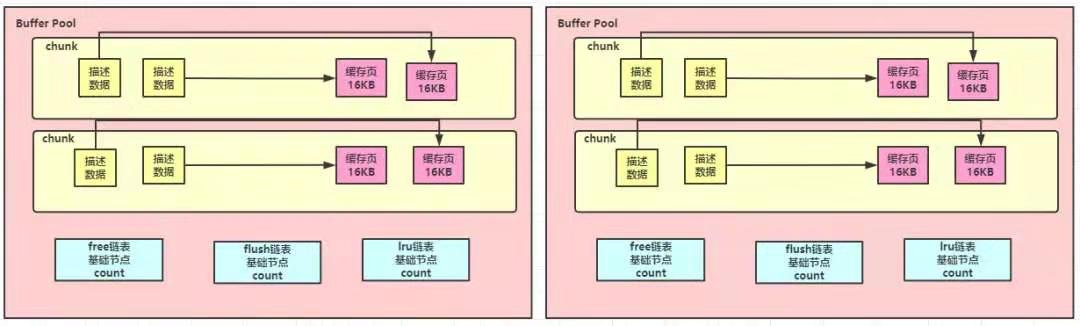
这样可能吗？如果原来的是8G，扩大到16G，那这个将原来的数据复制到新的Buffer Pool中是不是极为耗时的，所以这样的操作MySQL必然是不支持的。但实际上这样的需求是客观存在的，那MySQL是如何解决的呢？

**# 什么是chunk机制**  
 chunk是MySQL 设计的一种机制，这种机制的原理是将Buffer Pool拆分一个一个大小相等的chunk块，每个chunk默认大小为128M（可以通过参数innodb\_buffer\_pool\_chunk\_size来调整大小），也就是说Buffer Pool是由一个个的chunk组成的。  
 假设 Buffer Pool大小是2GB，而一个chunk大小默认是128M，也就是说一个2GB大小的Buffer Pool里面由16个chunk 组成，每个chunk中有自己的缓存页和描述数据，而free链表、flush链表和lru链表是共享的。

为了处理这种情况，MySQL设计出chunk（http协议中也有使用到这个思想，所以我们会发现很多技术的优秀思想都是在相互借鉴）机制来解决的。



如果说有多个Buffer Pool ，那就是这样



说到这里好像还是没有说到 MySQL到底是如何通过chunk机制来调整大小的。实际上是这样的，假设现在Buffer Pool有2GB，里面有16个chunk，现在想要扩大到 4GB，那么这个时候只需要新申请一个个的chunk就可以了。

这样不但不需要申请一块很大的连续的空间，更不需要将复制数据。这样就能达到动态调整大小了。

### key\_buffer\_size

key\_buffer\_size：MyISAM缓存索引块的内存大小

### query\_cache\_size

query\_cache\_size：查询缓冲的大小，8.0已被移除。

## 线程独占

### thread\_stack

thread\_stack：每个线程分配的堆栈大小

### sort\_buffer\_size

sort\_buffer\_size：排序缓冲的大小

### join\_buffer\_size

join\_buffer\_size：连接缓冲的大小

### read\_buffer\_size

read\_buffer\_size：MyISAM顺序读缓冲的大小

### read\_rnd\_buffer\_size

read\_rnd\_buffer\_size：MyISAM随机读缓冲的大小、MRR缓冲的大小

### tmp\_table\_size/max\_heap\_table\_size

tmp\_table\_size/max\_heap\_table\_size：内存临时表的大小

### binlog\_cache\_size

binlog\_cache\_size：二进制日志缓冲的大小

## 查看设置

Buffer Pool是不是越大越好，理论上是的。那如果一个机器内存是16GB那分配给Buffer Pool 15GB，这样很显然是不行的，因为操作系统要占内存，你的机器上总会运行其他的进行的吧？那肯定也是需要占用内存的。根据很多实际生产经验得出的比较合理的大小是机器内存大小的（50%~60%）。

最后一起来看看你的INNODB的相关参数，命令是show engine innodb status

show engine innodb status;  
 ----------------------  
 Buffer Pool AND MEMORY  
 ----------------------  
 -- Buffer Pool 的最终大小  
 Total memory allocated  
 -- Buffer Pool 一共有多少个缓存页  
 Buffer Pool size  
 -- free 链表中一共有多少个缓存也是可以使用的  
 Free buffers   
 -- lru链表中一共有多少个缓存页  
 Database pages   
 -- lru链表链表中的冷数据区一共有多少个缓存页  
 Old database pages   
 -- flush链表中的缓存页的数量  
 Modified db pages   
 -- 等待从磁盘上加载进来的缓存页的数量  
 Pending reads   
 -- 即将从lru链表中刷入磁盘的数量，flush链表中即将刷入磁盘的缓存页的数量  
 Pending writes: LRU 0, flush list 0, single page 0  
 -- lru链表的冷数据区的缓存页被访问之后转移到热数据区的缓存页的数量，以及冷数据区里1s之内被访问但是没有进入到热数据区的缓存页的数量  
 Pages made young 260368814, not young 0  
 -- 每秒从冷数据转移到热数据区的缓存页的数量，以及每秒在冷数据区被访问但是没有进入热数据区的缓存页的数量  
 332.69 youngs/s, 0.00 non-youngs/s  
 -- 已经读取创建和写入的缓存页的数量，以及每秒读取、创建和写入的缓存页的数量  
 Pages read 249280313, created 1075315, written 32924991 359.96 reads/s, 0.02 creates/s, 0.23 writes/s  
 -- 表示1000次访问中，有多少次是命中了BufferPool缓存中的缓存页，以及每1000次访问有多少数据从冷数据区转移到热数据区，以及没有转移的缓存页的数量  
 Buffer Pool hit rate 867 / 1000, young-making rate 123 / 1000 not 0 / 1000  
 -- lru链表中缓存页的数量  
 LRU len: 8190  
 -- 最近50s读取磁盘页的总数，cur[0]表示现在正在读取的磁盘页的总数  
I/O sum[5198]:cur[0],

# 监控

在绝大多数情况下，我们是不需要花费过多精力，去关注MySQL内存使用情况的；但是，也不能排除确实存在内存占用异常的情况，这个时候我们应该如何去进行深入排查呢？其实，MySQL官方就提供了强大的实时监控工具——performance\_schema库下的监控内存表，通过这个工具，我们可以很清晰地观察到MySQL内存到底是被谁占用了、分别占用了多少。

## 开启内存监控

我们可以选择，在实例启动时，开启内存监控采集器，具体方法如下：

vi my.cnf  
 performance-schema-instrument='memory/%=ON'

禁用方法如下：

vi my.cnf  
 performance-schema-instrument='memory/%=OFF'

实例运行时开启

我们也可以选择，在实例运行时，动态开启内存监控采集器，具体方法如下：

mysql> UPDATE performance\_schema.setup\_instruments SET ENABLED = 'YES' WHERE NAME LIKE 'memory/%';

禁用方法如下：

mysql> UPDATE performance\_schema.setup\_instruments SET ENABLED = 'NO' WHERE NAME LIKE 'memory/%';

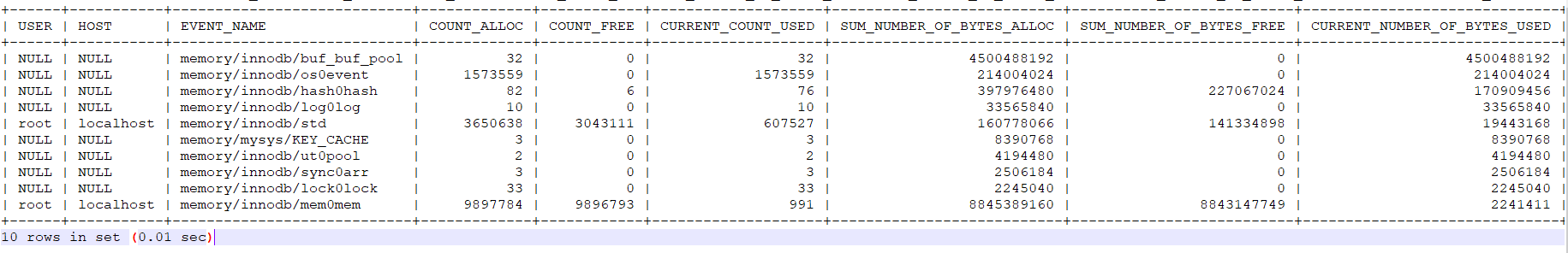
因为采集器的实现原理，是在内存进行分配/回收时，更新相对应内存监控表的数据；换句话说，就是采集器只能监控到开启之后的内存使用情况；而MySQL很大一部分内存都是在实例启动时就预先分配的，因此要想准确监控实例的内存使用率，需要在实例启动时就开启内存采集器。

## 内存监控表

在performance\_schema库下，提供多个维度的内存监控表，具体如下：  
 memory\_summary\_by\_account\_by\_event\_name：账号纬度的内存监控表  
 memory\_summary\_by\_host\_by\_event\_name：主机纬度的内存监控表  
 memory\_summary\_by\_thread\_by\_event\_name：线程维度的内存监控表  
 memory\_summary\_by\_user\_by\_event\_name：用户纬度的内存监控表  
 memory\_summary\_global\_by\_event\_name：全局纬度的内存监控表

内存监控表均包括以下关键字段：  
 COUNT\_ALLOC：内存分配次数  
 COUNT\_FREE：内存回收次数  
 SUM\_NUMBER\_OF\_BYTES\_ALLOC：内存分配大小  
 SUM\_NUMBER\_OF\_BYTES\_FREE：内存回收大小  
 CURRENT\_COUNT\_USED：当前分配的内存，通过COUNT\_ALLOC-COUNT\_FREE计算得到  
 CURRENT\_NUMBER\_OF\_BYTES\_USED：当前分配的内存大小，通过SUM\_NUMBER\_OF\_BYTES\_ALLOC-SUM\_NUMBER\_OF\_BYTES\_FREE计算得到  
 LOW\_COUNT\_USED：CURRENT\_COUNT\_USED的最小值  
 HIGH\_COUNT\_USED：CURRENT\_COUNT\_USED的最大值  
 LOW\_NUMBER\_OF\_BYTES\_USED：CURRENT\_NUMBER\_OF\_BYTES\_USED的最小值  
 HIGH\_NUMBER\_OF\_BYTES\_USED：CURRENT\_NUMBER\_OF\_BYTES\_USED的最大值  
 接下来，让我们看一个正常运行实例的内存使用情况，具体如下：

mysql> select USER,HOST,EVENT\_NAME,COUNT\_ALLOC,COUNT\_FREE,CURRENT\_COUNT\_USED,SUM\_NUMBER\_OF\_BYTES\_ALLOC,SUM\_NUMBER\_OF\_BYTES\_FREE,CURRENT\_NUMBER\_OF\_BYTES\_USED from performance\_schema.memory\_summary\_by\_account\_by\_event\_name order by CURRENT\_NUMBER\_OF\_BYTES\_USED desc limit 10;



# 总结