InnoDB 引擎底层存储和缓存原理

到目前为止，MySQL 对于我们来说还是一个黑盒，我们只负责使用客户端发送请求并等待服务器返回结果，表中的数据到底存到了哪里？以什么格式存放的？MySQL 是以什么方式来访问的这些数据？这些问题我们统统不知道。要搞明白查询优化背后的原理，就必须深入 MySQL 的底层去一探究竟，而且事务、锁等的原理也要求我们必须深入底层。

InnoDB 记录存储结构和索引页结构

InnoDB 是一个将表中的数据存储到磁盘上的存储引擎，所以即使关机后重启我们的数据还是存在的。而真正处理数据的过程是发生在内存中的，所以需要把磁盘中的数据加载到内存中，如果是处理写入或修改请求的话，还需要把内存中的内容刷新到磁盘上。而我们知道读写磁盘的速度非常慢，和内存读写差了几个数量级，所以当我们想从表中获取某些记录时，InnoDB 存储引擎需要一条一条的把记录从磁盘上读出来么？

InnoDB 采取的方式是：将数据划分为若干个页，以页作为磁盘和内存之间交互的基本单位，InnoDB 中页的大小一般为 16 KB。也就是在一般情况下，一次最少从磁盘中读取 16KB 的内容到内存中，一次最少把内存中的 16KB 内容刷新到磁盘中。

我们平时是以记录为单位来向表中插入数据的，这些记录在磁盘上的存放方式也被称为行格式或者记录格式。InnoDB 存储引擎设计了 4 种不同类型的行格式，分别是 Compact、Redundant、Dynamic 和 Compressed 行格式。

行格式

我们可以在创建或修改表的语句中指定行格式：

CREATE TABLE 表名 (列的信息) ROW_FORMAT=行格式名称

COMPACT

我们知道 MySQL 支持一些变长的数据类型，比如 VARCHAR(M)、VARBINARY(M)、各种 TEXT 类型，各种 BLOB 类型，我们也可以把拥有这些数据类型的列称为变长字段，变长字段中存储多少字节的数据是不固定的，所以我们在存储真实数据的时候需要顺便把这些数据占用的字节数也存起来。如果该可变字段允许存储的最大字节数超过 255 字节并且真实存储的字节数超过 127 字节，则使用 2 个字节，否则使用 1 个字节。

表中的某些列可能存储 NULL 值，如果把这些 NULL 值都放到记录的真实数据中存储会很占地方，所以 Compact 行格式把这些值为 NULL 的列统一管理起来，

存储到 NULL 值列表。每个允许存储 NULL 的列对应一个二进制位，二进制位的值为1时，代表该列的值为NULL。二进制位的值为0时，代表该列的值不为NULL。

还有一个用于描述记录的记录头信息，它是由固定的 5 个字节组成。5 个字节也就是 40 个二进制位，不同的位代表不同的意思。

预留位 1 1　没有使用

预留位 2 1　没有使用

delete_mask　1　标记该记录是否被删除

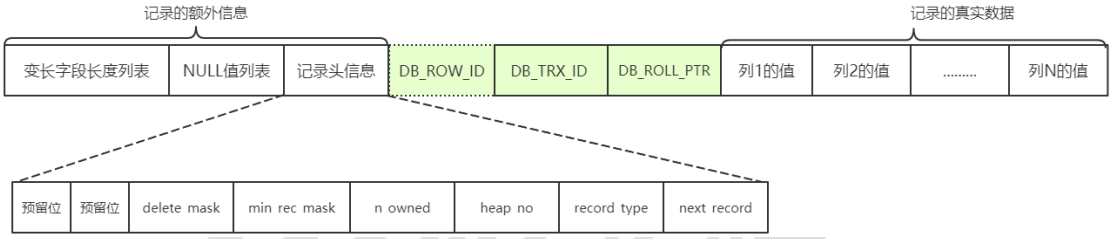
min_rec_mask　1　B+树的每层非叶子节点中的最小记录都会添加该标记

n_owned 4　表示当前记录拥有的记录数

heap_no　13　表示当前记录在页的位置信息

record_type　3　表示当前记录的类型，0 表示普通记录，1 表示 B+树非叶子节点记录，2 表示最小记录，3 表示最大记录

next_record　16　表示下一条记录的相对位置



记录的真实数据除了我们自己定义的列的数据以外，MySQL 会为每个记录默认的添加一些列（也称为隐藏列），包括：

DB_ROW_ID(row_id)：非必须，6 字节，表示行 ID，唯一标识一条记录

DB_TRX_ID：必须，6 字节，表示事务 ID

DB_ROLL_PTR：必须，7 字节，表示回滚指针

InnoDB 表对主键的生成策略是：优先使用用户自定义主键作为主键，如果用户没有定义主键，则选取一个 Unique 键作为主键，如果表中连 Unique 键都没有定义的话，则 InnoDB 会为表默认添加一个名为 row_id 的隐藏列作为主键。DB_TRX_ID（也可以称为 trx_id） 和 DB_ROLL_PTR（也可以称为 roll_ptr） 这两个列是必有的，但是 row_id 是可选的（在没有自定义主键以及 Unique 键的情况下才会添加该列）。

其他的行格式和 Compact 行格式差别不大。

Redundant 行格式

Redundant 行格式是 MySQL5.0 之前用的一种行格式，不予深究。

Dynamic 和 Compressed 行格式

MySQL5.7 的默认行格式就是 Dynamic，Dynamic 和 Compressed 行格式和 Compact 行格式挺像，只不过在处理行溢出数据时有所不同。Compressed 行格

式和 Dynamic 不同的一点是，Compressed 行格式会采用压缩算法对页面进行压缩，以节省空间。

数据溢出

如果我们定义一个表，表中只有一个 VARCHAR 字段，如下：

```
CREATE TABLE test_varchar( c VARCHAR(60000) )
```

然后往这个字段插入 60000 个字符，会发生什么？

前边说过，MySQL 中磁盘和内存交互的基本单位是页，也就是说 MySQL 是以页为基本单位来管理存储空间的，我们的记录都会被分配到某个页中存储。而一个页的大小一般是 16KB，也就是 16384 字节，而一个 VARCHAR(M)类型的列就最多可以存储 65532 个字节，这样就可能造成一个页存放不了一条记录的情况。

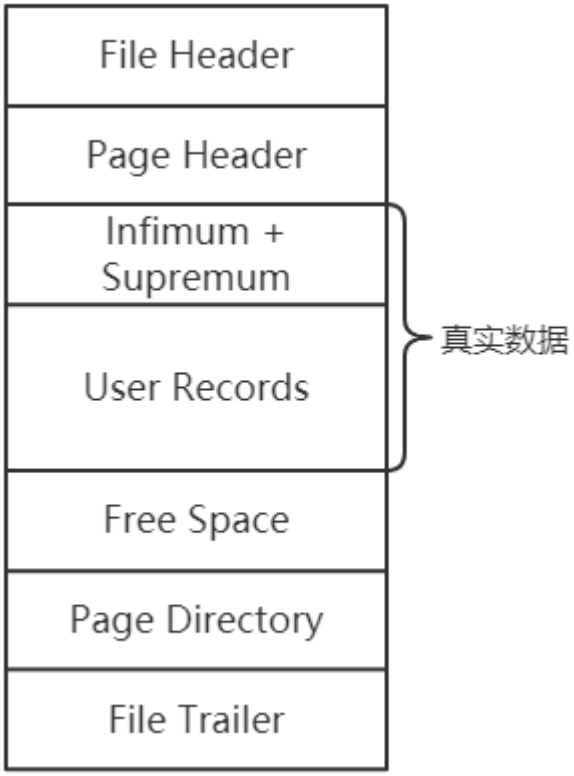
在 Compact 和 Redundant 行格式中，对于占用存储空间非常大的列，在记录的真实数据处只会存储该列的该列的前 768 个字节的数据，然后把剩余的数据分散存储在几个其他的页中，记录的真实数据处用 20 个字节存储指向这些页的地址。这个过程也叫做行溢出，存储超出 768 字节的那些页面也被称为溢出页。

Dynamic 和 Compressed 行格式，不会在记录的真实数据处存储字段真实数据的前 768 个字节，而是把所有的字节都存储到其他页面中，只在记录的真实数据处存储其他页面的地址。

索引页格式

前边我们简单提了一下页的概念，它是 InnoDB 管理存储空间的基本单位，一个页的大小一般是 16KB。

InnoDB 为了不同的目的而设计了许多种不同类型的页，存放我们表中记录的那种类型的页自然也是其中的一员，官方称这种存放记录的页为索引（INDEX）页，不过要理解成数据页也没问题，毕竟存在着聚簇索引这种索引和数据混合的东西。



一个 InnoDB 数据页的存储空间大致被划分成了 7 个部分：

File Header　文件头部　38 字节 页的一些通用信息

Page Header　页面头部　56 字节 数据页专有的一些信息

Infimum + Supremum　最小记录和最大记录　26 字节 两个虚拟的行记录

User Records　用户记录　大小不确定 实际存储的行记录内容

Free Space　空闲空间　大小不确定 页中尚未使用的空间

Page Directory　页面目录　大小不确定 页中的某些记录的相对位置

File Trailer　文件尾部　8 字节　校验页是否完整

User Records

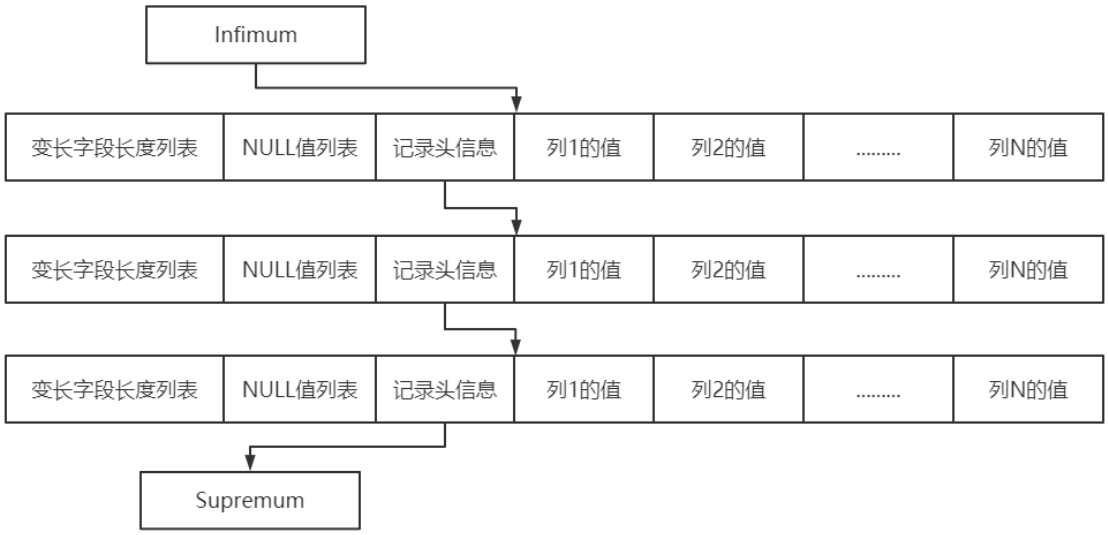
我们自己存储的记录会按照我们指定的行格式存储到 User Records 部分。但是在一开始生成页的时候，其实并没有 User Records 这个部分，每当我们插入一条记录，都会从 Free Space 部分，也就是尚未使用的存储空间中申请一个记录大小的空间划分到 User Records 部分，当 Free Space 部分的空间全部被 User Records 部分替代掉之后，也就意味着这个页使用完了，如果还有新的记录插入的话，就需要去申请新的页了。

当前记录被删除时，则会修改记录头信息中的 delete_mask 为 1，也就是说被删除的记录还在页中，还在真实的磁盘上。这些被删除的记录之所以不立即从磁盘上移除，是因为移除它们之后把其他的记录在磁盘上重新排列需要性能消耗。

所以只是打一个删除标记而已，所有被删除掉的记录都会组成一个所谓的垃圾链表，在这个链表中的记录占用的空间称之为所谓的可重用空间，之后如果有新记录插入到表中的话，可能把这些被删除的记录占用的存储空间覆盖掉。

同时我们插入的记录在会记录自己在本页中的位置，写入了记录头信息中 heap_no 部分。heap_no 值为 0 和 1 的记录是 InnoDB 自动给每个页增加的两个记录，称为伪记录或者虚拟记录。这两个伪记录一个代表最小记录，一个代表最大记录，这两条存放在页的 User Records 部分，他们被单独放在一个称为 Infimum + Supremum 的部分。

记录头信息中 next_record 记录了从当前记录的真实数据到下一条记录的真实数据的地址偏移量。这其实是个链表，可以通过一条记录找到它的下一条记录。但是需要注意注意再注意的一点是，下一条记录指得并不是按照我们插入顺序的下一条记录，而是按照主键值由小到大的顺序的下一条记录。而且规定 Infimum 记录（也就是最小记录） 的下一条记录就是本页中主键值最小的用户记录，而本页中主键值最大的用户记录的下一条记录就是 Supremum 记录（也就是最大记录）



我们的记录按照主键从小到大的顺序形成了一个单链表，记录被删除，则从这个链表上摘除。

Page Directory

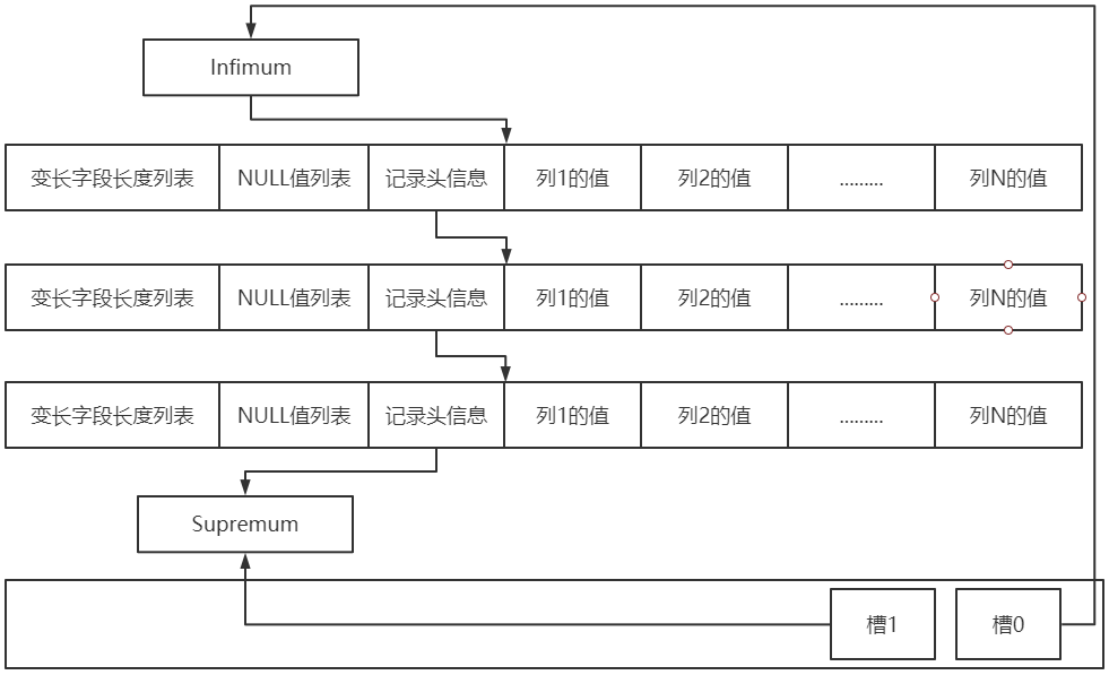
Page Directory 主要是解决记录链表的查找问题。如果我们想根据主键值查找页中的某条记录该咋办？按链表查找的办法：从 Infimum 记录（最小记录）开始，沿着链表一直往后找，总会找到或者找不到。但是时间复杂度不低。

InnoDB 的改进是，为页中的记录再制作了一个目录，他们的制作过程是这样的：

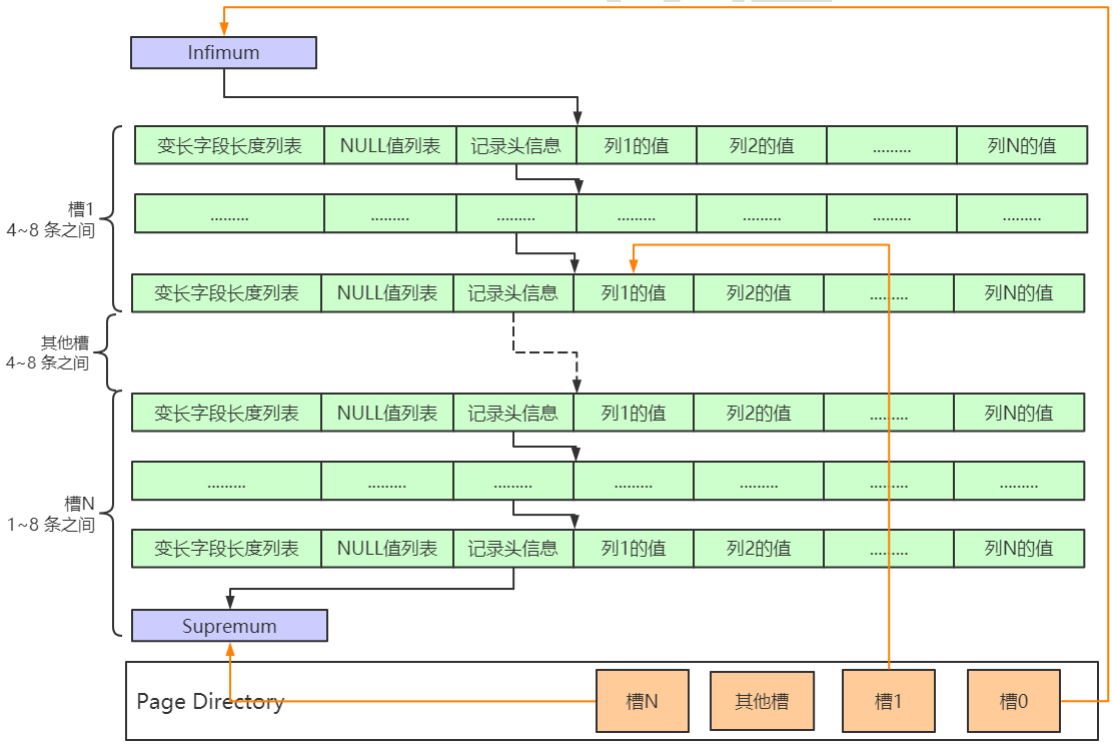
1、将所有正常的记录（包括最大和最小记录，不包括标记为已删除的记录）划分为几个组。

2、每个组的最后一条记录（也就是组内最大的那条记录）的头信息中的 n_owned 属性表示该记录拥有多少条记录，也就是该组内共有几条记录。

3、将每个组的最后一条记录的地址偏移量单独提取出来按顺序存储到靠近页的尾部的地方，这个地方就是所谓的 Page Directory，也就是页目录页面目录中的这些地址偏移量被称为槽（英文名：Slot），所以这个页面目录就是由槽组成的。



4、每个分组中的记录条数是有规定的：对于最小记录所在的分组只能有 1 条记录，最大记录所在的分组拥有的记录条数只能在 1~8 条之间，剩下的分组中记录的条数范围只能在是 4~8 条之间。如下图：



这样，一个数据页中查找指定主键值的记录的过程分为两步：

通过二分法确定该记录所在的槽，并找到该槽所在分组中主键值最小的那条记录。

通过记录的 next_record 属性遍历该槽所在的组中的各个记录。

Page Header

InnoDB 为了能得到一个数据页中存储的记录的状态信息，比如本页中已经存储了多少条记录，第一条记录的地址是什么，页目录中存储了多少个槽等等，特意在页中定义了一个叫 Page Header 的部分，它是页结构的第二部分，这个部分占用固定的 56 个字节，专门存储各种状态信息。

File Header

File Header 针对各种类型的页都通用，也就是说不同类型的页都会以 File Header 作为第一个组成部分，它描述了一些针对各种页都通用的一些信息，比方说页的类型，这个页的编号是多少，它的上一个页、下一个页是谁，页的校验和等等，这个部分占用固定的 38 个字节。

页的类型，包括 Undo 日志页、段信息节点、Insert Buffer 空闲列表、Insert Buffer 位图、系统页、事务系统数据、表空间头部信息、扩展描述页、溢出页、索引页，有些页会在后面的课程看到。

同时通过上一个页、下一个页建立一个双向链表把许许多多的页就串联起来，而无需这些页在物理上真正连着。但是并不是所有类型的页都有上一个和下一个页的属性，数据页是有这两个属性的，所以所有的数据页其实是一个双向链表。

File Trailer

我们知道 InnoDB 存储引擎会把数据存储到磁盘上，但是磁盘速度太慢，需要以页为单位把数据加载到内存中处理，如果该页中的数据在内存中被修改了，那么在修改后的某个时间需要把数据同步到磁盘中。但是在同步了一半的时候中断电了咋办？

为了检测一个页是否完整（也就是在同步的时候有没有发生只同步一半的尴尬情况），InnoDB 每个页的尾部都加了一个 File Trailer 部分，这个部分由 8 个字节组成，可以分成 2 个小部分：

前 4 个字节代表页的校验和

这个部分是和 File Header 中的校验和相对应的。每当一个页面在内存中修改了，在同步之前就要把它的校验和算出来，因为 File Header 在页面的前边，所以校验和会被首先同步到磁盘，当完全写完时，校验和也会被写到页的尾部，如果完全同步成功，则页的首部和尾部的校验和应该是一致的。如果写了一半儿断电了，那么在 File Header 中的校验和就代表着已经修改过的页，而在 File Trailer 中的校验和代表着原先的页，二者不同则意味着同步中间出了错。

后 4 个字节代表页面被最后修改时对应的日志序列位置（LSN），这个也和校验页的完整性有关。

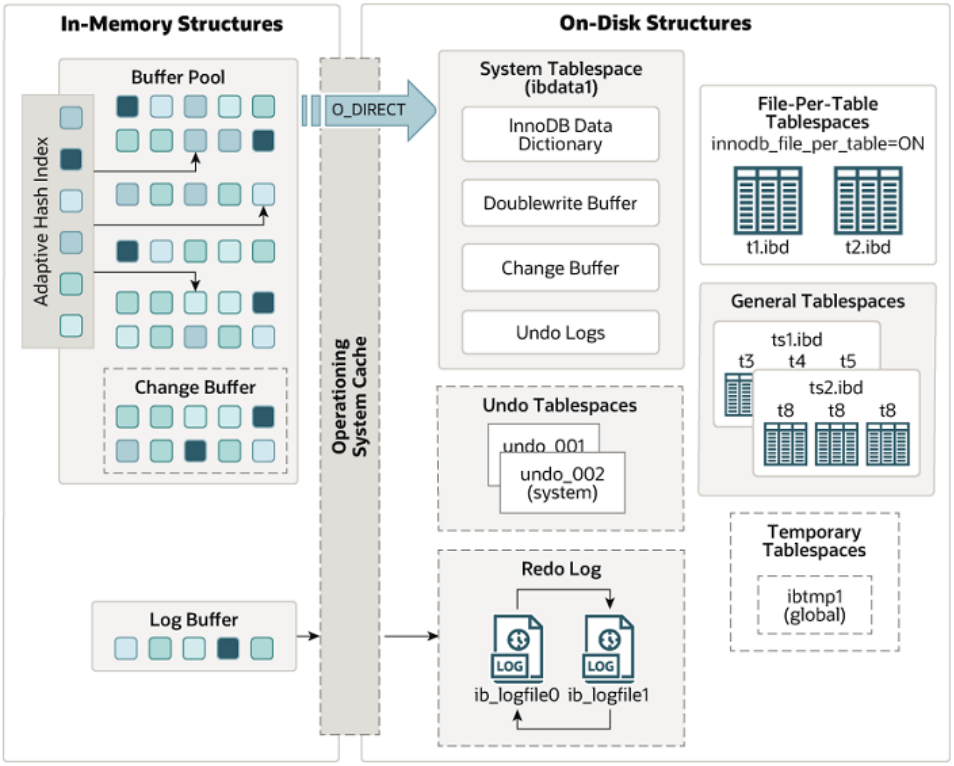
这个 File Trailer 与 File Header 类似，都是所有类型的页通用的。

InnoDB 的体系结构

前面，我们站在微观的角度了解了数据记录和页面的存储格式，现在我们需要站在宏观的角度看看 InnoDB 的内存结构和磁盘存储结构。

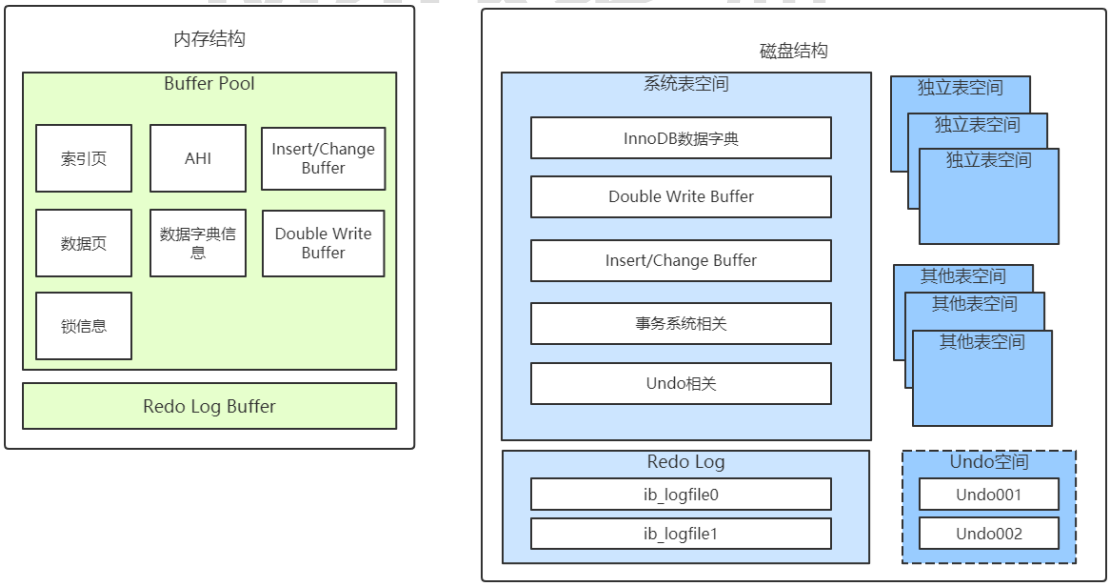
可以参考 MySQL 官方文档

https://dev.mysql.com/doc/refman/5.7/en/innodb-architecture.html



可以看见，比较关键的是其中的各种 Buffer 和 Tabelspace（表空间），这些也是我们接下来要学习的重点。

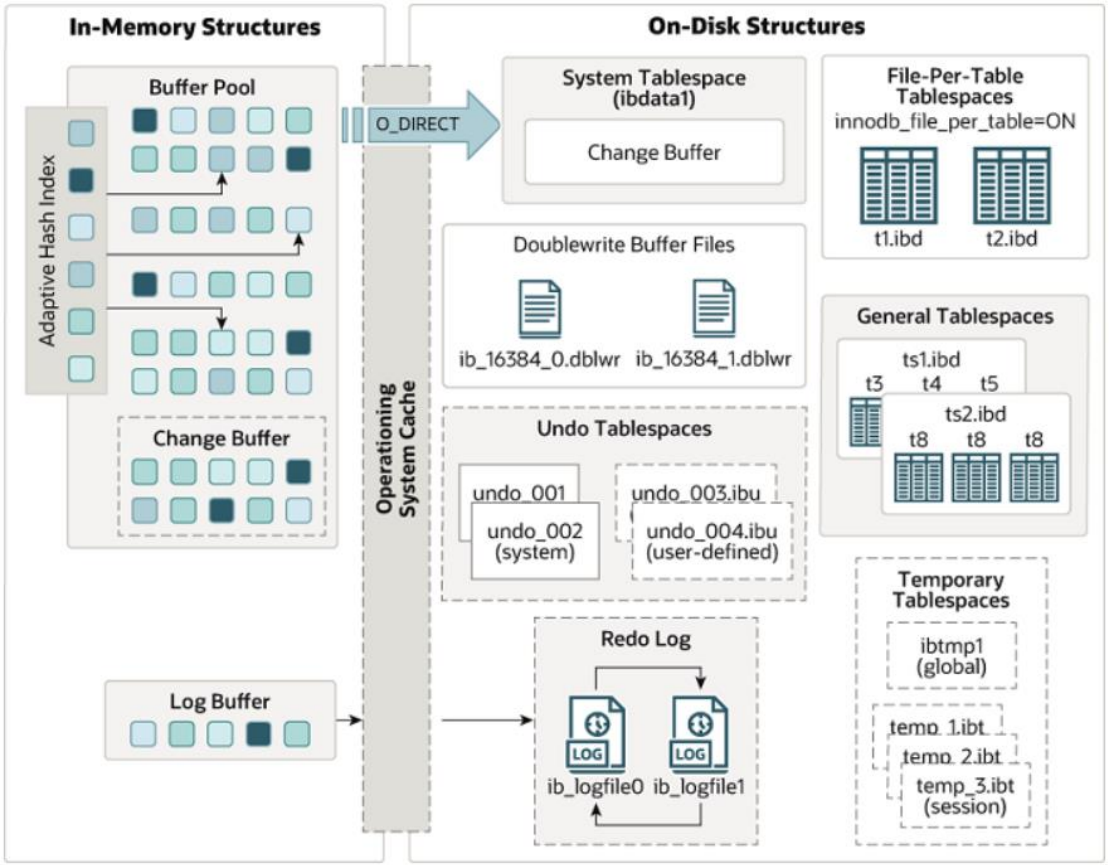
太长不看版：



不过 InnoDB 的内存结构和磁盘存储结构在 MySQL8.0 有所变化：

https://dev.mysql.com/doc/refman/8.0/en/innodb-architecture.html

但是不影响我们后面对 InnoDB 内部原理的学习。



InnoDB 的表空间

表空间是一个抽象的概念，对于系统表空间来说，对应着文件系统中一个或多个实际文件，一般是（ibdata1）；对于每个独立表空间（也就是上图的 File-Per-Table Tablespaces）来说，对应着文件系统中一个名为表名.ibd 的实际文件。

```
[mysql@iZwz9j203ithc4gu1uvb2wZ mysqladv]$ ls
db.opt                    goods_order_detail_2.frm  order_exp_cut.ibd  student.frm
e1.frm                    goods_order_detail_2.ibd  order_exp.frm      student.ibd
e1.ibd                    goods_order_detail.frm    order_exp.ibd      tbl_chain_voucher_lock_v2.frm
e2.frm                    goods_order_detail.ibd    s1.frm             tbl_chain_voucher_lock_v2.ibd
e2.ibd                    goods_order.frm           s1.ibd             teacher.frm
goods.frm                 goods_order.ibd           s2.frm             teacher.ibd
goods.ibd                 order_exp2.frm            s2.ibd             teacher_mvcc.frm
goods_order_detail_1.frm  order_exp2.ibd            score.frm          teacher_mvcc.ibd
goods_order_detail_1.ibd  order_exp_cut.frm         score.ibd          test_abc_copy1.frm
```

```
mysql> show tables;
+---------------------------+
| Tables_in_mysqladv        |
+---------------------------+
| e1                        |
| e2                        |
| goods                     |
| goods_order               |
| goods_order_detail        |
| goods_order_detail_1      |
| goods_order_detail_2      |
| order_exp                 |
| order_exp2                |
| order_exp_cut             |
| s1                        |
| s2                        |
| score                     |
| student                   |
| t                         |
| tbl_chain_voucher_lock_v2 |
| teacher                   |
| teacher_mvcc              |
| test                      |
```

大家可以把表空间想象成被切分为许许多多个页的池子，当我们想为某个表插入一条记录的时候，就从池子中捞出一个对应的页来把数据写进去。

再回忆一次，InnoDB 是以页为单位管理存储空间的，我们的聚簇索引（也就是完整的表数据）和其他的二级索引都是以 B+树的形式保存到表空间的，而 B+树的节点就是数据页。

任何类型的页都有专门的地方保存页属于哪个表空间，同时表空间中的每一个页都对应着一个页号，这个页号由 4 个字节组成，也就是 32 个比特位，所以一个表空间最多可以拥有 2^{32} 个页，如果按照页的默认大小 16KB 来算，一个表空间最多支持 64TB 的数据。

独立表空间结构

区（extent）

表空间中的页可以达到 2^{32} 个页，实在是太多了，为了更好的管理这些页面，InnoDB 中还有一个区（英文名：extent）的概念。对于 16KB 的页来说，连续的 64 个页就是一个区，也就是说一个区默认占用 1MB 空间大小。

不论是系统表空间还是独立表空间，都可以看成是由若干个区组成的，每 256 个区又被划分成一个**组**。

第一个组最开始的 3 个页面的类型是固定的：用来登记整个表空间的一些整体属性以及本组所有的区被称为 FSP_HDR，也就是 extent 0 ~ extent 255 这 256 个区，整个表空间只有一个 FSP_HDR。

其余各组最开始的 2 个页面的类型是固定的，一个 XDES 类型，用来登记本组 256 个区的属性，FSP_HDR 类型的页面其实和 XDES 类型的页面的作用类似，只不过 FSP_HDR 类型的页面还会额外存储一些表空间的属性。

引入区的主要目的是什么？我们每向表中插入一条记录，本质上就是向该表的聚簇索引以及所有二级索引代表的 B+树的节点中插入数据。而 B+树的每一层中的页都会形成一个双向链表，如果是以页为单位来分配存储空间的话，双向链表相邻的两个页之间的物理位置可能离得非常远。

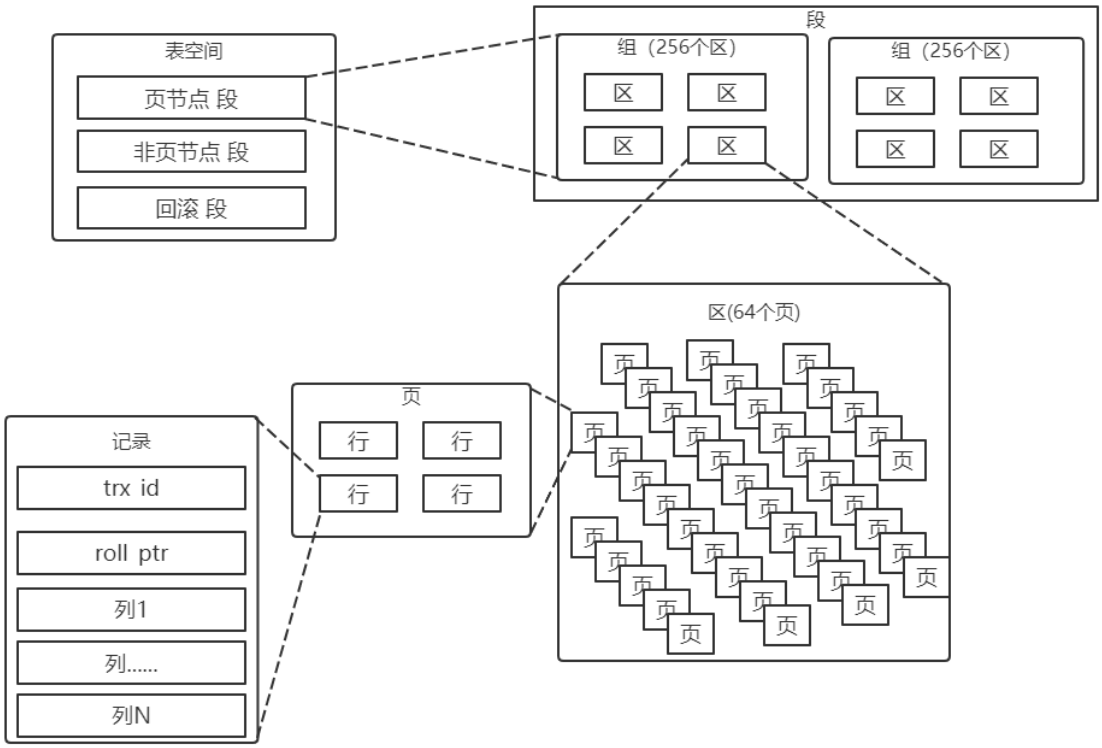
我们介绍 B+树索引的适用场景的时候特别提到范围查询只需要定位到最左边的记录和最右边的记录，然后沿着双向链表一直扫描就可以了，而如果链表中相邻的两个页物理位置离得非常远，就是所谓的随机 I/O。再一次强调，磁盘的速度和内存的速度差了好几个数量级，随机 I/O 是非常慢的，所以我们应该尽量让链表中相邻的页的物理位置也相邻，这样进行范围查询的时候才可以使用所谓的顺序 I/O。

一个区就是在物理位置上连续的 64 个页。在表中数据量大的时候，为某个索引分配空间的时候就不再按照页为单位分配了，而是按照区为单位分配，甚至在表中的数据十分非常特别多的时候，可以一次性分配多个连续的区，从性能角度看，可以消除很多的随机 I/O。

段（segment）

我们提到的范围查询，其实是对 B+树叶子节点中的记录进行顺序扫描，而如果不区分叶子节点和非叶子节点，统统把节点代表的页面放到申请到的区中的话，进行范围扫描的效果就大打折扣了。所以 InnoDB 对 B+树的叶子节点和非叶子节点进行了区别对待，也就是说叶子节点有自己独有的区，非叶子节点也有自己独有的区。存放叶子节点的区的集合就算是一个段（segment），存放非叶子节点的区的集合也算是一个段。也就是说一个索引会生成 2 个段，一个叶子节点段，一个非叶子节点段。

段其实不对应表空间中某一个连续的物理区域，而是一个逻辑上的概念。



系统表空间

整体结构

系统表空间的结构和独立表空间基本类似，只不过由于整个 MySQL 进程只有一个系统表空间，在系统表空间中会额外记录一些有关整个系统信息的页面，所以会比独立表空间多出一些记录这些信息的页面，相当于是表空间之首，所以它的表空间 ID（Space ID）是 0。

系统表空间有 extent 1 和 extent 两个区，也就是页号从 64~191 这 128 个页面被称为 Doublewrite buffer，也就是双写缓冲区。

双写缓冲区/双写机制

双写缓冲区/双写机制是 InnoDB 的三大特性之一，还有两个是 Buffer Pool、自适应 Hash 索引。

它是一种特殊文件 flush 技术，带给 InnoDB 存储引擎的是数据页的可靠性。它的作用是，在把页写到数据文件之前，InnoDB 先把它们写到一个叫 doublewrite buffer（双写缓冲区）的连续区域内，在写 doublewrite buffer 完成后，InnoDB 才会把页写到数据文件的适当的位置。如果在写页的过程中发生意外崩溃，InnoDB 在稍后的恢复过程中在 doublewrite buffer 中找到完好的 page 副本用于恢复。

所以，虽然叫双写缓冲区，但是这个缓冲区不仅在内存中有，更多的是属于 MySQL 的系统表空间，属于磁盘文件的一部分。那为什么要引入一个双写机制呢？

InnoDB 的页大小一般是 16KB，其数据校验也是针对这 16KB 来计算的，将数据写入到磁盘是以页为单位进行操作的。而操作系统写文件是以 4KB 作为单位的，那么每写一个 InnoDB 的页到磁盘上，操作系统需要写 4 个块。

而计算机硬件和操作系统，在极端情况下（比如断电）往往并不能保证这一操作的原子性，16K 的数据，写入 4K 时，发生了系统断电或系统崩溃，只有一部分写是成功的，这种情况下会产生 partial page write（部分页写入）问题。这时页数据出现不一样的情形，从而形成一个"断裂"的页，使数据产生混乱。在 InnoDB 存储引擎未使用 doublewrite 技术前，曾经出现过因为部分写失效而导致数据丢失的情况。

doublewrite buffer 是 InnoDB 在表空间上的 128 个页（2 个区，extend1 和 extend2），大小是 2MB。为了解决部分页写入问题，当 MySQL 将脏数据 flush 到数据文件的时候，先使用 memcopy 将脏数据复制到内存中的一个区域（也是 2M），之后通过这个内存区域再分 2 次，每次写入 1MB 到系统表空间，然后马上调用 fsync 函数，同步到磁盘上。在这个过程中是顺序写，开销并不大，在完成 doublewrite 写入后，再将数据写入各数据文件文件，这时是离散写入。

所以在正常的情况下，MySQL 写数据页时，会写两遍到磁盘上，第一遍是写到 doublewrite buffer，第二遍是写到真正的数据文件中。如果发生了极端情况（断电），InnoDB 再次启动后，发现了一个页数据已经损坏，那么此时就可以从 doublewrite buffer 中进行数据恢复了。

前面说过，位于系统表空间上的 doublewrite buffer 实际上也是一个文件，写系统表空间会导致系统有更多的 fsync 操作，而硬盘的 fsync 性能因素会降低 MySQL 的整体性能。不过在存储上，doublewrite 是在一个连续的存储空间，所以硬盘在写数据的时候是顺序写，而不是随机写，这样性能影响不大，相比不双写，降低了大概 5-10%左右。

所以，在一些情况下可以关闭 doublewrite 以获取更高的性能。比如在 slave 上可以关闭，因为即使出现了 partial page write 问题，数据还是可以从中继日志中恢复。比如某些文件系统 ZFS 本身有些文件系统本身就提供了部分写失效的防范机制，也可以关闭。

在数据库异常关闭的情况下启动时，都会做数据库恢复（redo）操作，恢复的过程中，数据库都会检查页面是不是合法（校验等等），如果发现一个页面校验结果不一致，则此时会用到双写这个功能。

有经验的同学也许会想到，如果发生写失效，可以通过重做日志(Redo Log)进行恢复啊！但是要注意，重做日志中记录的是对页的物理操作，如偏移量 800，写' aaaa'记录，而不是页面的全量记录，而如果发生 partial page write（部分页写入）问题时，出现问题的是未修改过的数据，此时重做日志(Redo Log)无能为力。写 doublewrite buffer 成功了，这个问题就不用担心了。

如果是写 doublewrite buffer 本身失败，那么这些数据不会被写到磁盘，InnoDB 此时会从磁盘载入原始的数据，然后通过 InnoDB 的事务日志来计算出正确的数据，重新写入到 doublewrite buffer，这个速度就比较慢了。如果 doublewrite buffer 写成功的话,但是写数据文件失败，innodb 就不用通过事务日志来计算了,而是直接用 doublewrite buffer 的数据再写一遍，速度上会快很多。

总体来说，doublewrite buffer 的作用有两个：提高 innodb 把缓存的数据写到硬盘这个过程的安全性；间接的好处就是，innodb 的事务日志不需要包含所有数据的前后映像,而是二进制变化量，这可以节省大量的 IO。

InnoDB 数据字典(Data Dictionary Header)

我们平时使用 INSERT 语句向表中插入的那些记录称之为用户数据，MySQL 只是作为一个软件来为我们来保管这些数据，提供方便的增删改查接口而已。但是每当我们向一个表中插入一条记录的时候，MySQL 先要校验一下插入语句对应的表存不存在，插入的列和表中的列是否符合，如果语法没有问题的话，还需要知道该表的聚簇索引和所有二级索引对应的根页面是哪个表空间的哪个页面，然后把记录插入对应索引的 B+树中。所以说，MySQL 除了保存着我们插入的用户数据之外，还需要保存许多额外的信息，比方说：

某个表属于哪个表空间，表里边有多少列，表对应的每一个列的类型是什么，该表有多少索引，每个索引对应哪几个字段，该索引对应的根页面在哪个表空间的哪个页面，该表有哪些外键，外键对应哪个表的哪些列，某个表空间对应文件系统上文件路径是什么。

上述这些数据并不是我们使用 INSERT 语句插入的用户数据，实际上是为了更好的管理我们这些用户数据而不得已引入的一些额外数据，这些数据也称为元数据。InnoDB 存储引擎特意定义了一些列的内部系统表（internal system table）来记录这些这些元数据：

表名 描述

SYS_TABLES 整个 InnoDB 存储引擎中所有的表的信息

SYS_COLUMNS 整个 InnoDB 存储引擎中所有的列的信息

SYS_INDEXES 整个 InnoDB 存储引擎中所有的索引的信息

SYS_FIELDS 整个 InnoDB 存储引擎中所有的索引对应的列的信息

SYS_FOREIGN 整个 InnoDB 存储引擎中所有的外键的信息

SYS_FOREIGN_COLS 整个 InnoDB 存储引擎中所有的外键对应列的信息

SYS_TABLESPACES 整个 InnoDB 存储引擎中所有的表空间信息

SYS_DATAFILES 整个 InnoDB 存储引擎中所有的表空间对应文件系统的文件路径信息

SYS_VIRTUAL 整个 InnoDB 存储引擎中所有的虚拟生成列的信息

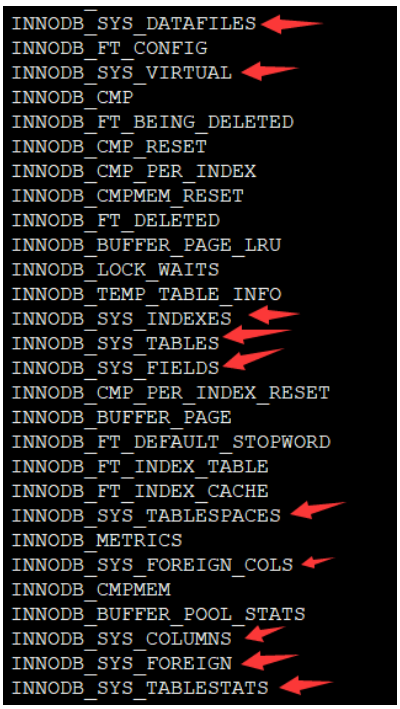
这些系统表也被称为数据字典，它们都是以 B+树的形式保存在系统表空间的某些页面中，其中 SYS_TABLES、SYS_COLUMNS、SYS_INDEXES、SYS_FIELDS 这四个表尤其重要，称之为基本系统表。

这 4 个表是表中之表，那这 4 个表的元数据去哪里获取呢？只能把这 4 个表的元数据，就是它们有哪些列、哪些索引等信息硬编码到代码中，然后 InnoDB 的又拿出一个固定的页面来记录这 4 个表的聚簇索引和二级索引对应的 B+树位置，这个页面就是页号为 7 的页面 Data Dictionary Header，类型为 SYS，记录了数据字典的头部信息。除了这 4 个表的 5 个索引的根页面信息外，这个页号为 7 的页面还记录了整个 InnoDB 存储引擎的一些全局属性，比如 Row ID。

数据字典头部信息中有个 Max Row ID 字段，我们说过如果我们不显式的为表定义主键，而且表中也没有 UNIQUE 索引，那么 InnoDB 存储引擎会默认为我们生成一个名为 row_id 的列作为主键。因为它是主键，所以每条记录的 row_id 列的值不能重复。

原则上只要一个表中的 row_id 列不重复就可以了，也就是说表 a 和表 b 拥有一样的 row_id 列也没啥关系，不过 InnoDB 只提供了这个 Max Row ID 字段，不论哪个拥有 row_id 列的表插入一条记录时，该记录的 row_id 列的值就是 Max Row ID 对应的值，然后再把 Max Row ID 对应的值加 1，也就是说这个 Max Row ID 是全局共享的。

用户是不能直接访问 InnoDB 的这些内部系统表的，除非你直接去解析系统表空间对应文件系统上的文件。不过 InnoDB 考虑到查看这些表的内容可能有助于大家分析问题，所以在系统数据库 information_schema 中提供了一些以 innodb_sys 开头的表：



在 information_schema 数据库中的这些以 INNODB_SYS 开头的表并不是真正的内部系统表（内部系统表就是我们上边说过的以 SYS 开头的那些表），而是在存储引擎启动时读取这些以 SYS 开头的系统表，然后填充到这些以 INNODB_SYS 开头的表中。

InnoDB 的 Buffer Pool

缓存的重要性

我们知道，对于使用 InnoDB 作为存储引擎的表来说，不管是用于存储用户数据的索引（包括聚簇索引和二级索引），还是各种系统数据，都是以页的形式存放在表空间中的，而所谓的表空间只不过是 InnoDB 对文件系统上一个或几个实际文件的抽象，也就是说我们的数据说到底还是存储在磁盘上的。

但是磁盘的速度慢，所以 InnoDB 存储引擎在处理客户端的请求时，当需要访问某个页的数据时，就会把完整的页的数据全部加载到内存中，也就是说即使我们只需要访问一个页的一条记录，那也需要先把整个页的数据加载到内存中。将整个页加载到内存中后就可以进行读写访问了，在进行完读写访问之后并不着急把该页对应的内存空间释放掉，而是将其缓存起来，这样将来有请求再次访问该页面时，就可以省去磁盘 IO 的开销了。

Buffer Pool

InnoDB 为了缓存磁盘中的页，在 MySQL 服务器启动的时候就向操作系统申请了一片连续的内存，他们给这片内存起了个名，叫做 Buffer Pool（中文名是缓冲池）。那它有多大呢？这个其实看我们机器的配置，默认情况下 Buffer Pool 只有 128M 大小。

show variables like 'innodb_buffer_pool_size';

```
mysql> show variables like 'innodb_buffer_pool_size';
+-------------------------+-----------+
| Variable_name           | Value     |
+-------------------------+-----------+
| innodb_buffer_pool_size | 134217728 |
+-------------------------+-----------+
1 row in set (0.01 sec)
```

可以在启动服务器的时候配置 innodb_buffer_pool_size 参数的值，它表示 Buffer Pool 的大小，就像这样：

[server]

innodb_buffer_pool_size = 268435456

其中，268435456 的单位是字节，也就是指定 Buffer Pool 的大小为 256M。需要注意的是，Buffer Pool 也不能太小，最小值为 5M(当小于该值时会自动设置成 5M)。

Buffer Pool 的缺省值其实是偏小的，一个比较合理的设置方法是按比例设置，一般的网上惯例是给 buffer pool 设置的机器内存的 60%左右，当然这个值偏保守，因为按照 MySQL 官方的说法（5.7 版本和 8.0 版本都是）：

https://dev.mysql.com/doc/refman/8.0/en/innodb-parameters.html

A larger buffer pool requires less disk I/O to access the same table data more than once. On a dedicated database server, you might set the buffer pool size to 80% of the machine's physical memory size. Be aware of the following potential issues when configuring buffer pool size, and be prepared to scale back the size of the buffer pool if necessary.

- Competition for physical memory can cause paging in the operating system.
- `InnoDB` reserves additional memory for buffers and control structures, so that the total allocated space is approximately 10% greater than the specified buffer pool size.

上文的意思是：更大的缓冲池只需更少的磁盘 I/O 来多次访问相同的表数据。在**专用**数据库服务器上，您可以将缓冲池大小设置为机器物理内存大小的 80%。配置缓冲池大小时请注意以下潜在问题，并准备在必要时缩减缓冲池的大小。

……。

InnoDB 为缓冲区和控制结构保留了额外的内存，因此分配的总空间比指定的缓冲池大小大约大 10%。

也就是说其实按照官方的分配最终 Buffer Pool 占据的空间可能达到机器物理内存的 90%，这个内存占用还是有点冒险的，因为即使是**专用**数据库服务器，还需要考虑：

* 每个查询至少需要几 K 的内存(有时候是几 M)

* 有各种其它内部的 MySQL 结构和缓存

* InnoDB 有一些结构是不用缓冲池的内存的(字典缓存，文件系统等)

* 也有一些 MySQL 文件是在 OS 缓存里的(binary 日志，relay 日志,innodb 事务日志等)

* 此外也必须为操作系统留出些内存

所以比较权衡的值是 70%~75%之间，但是需要监控好服务器的内存使用情况。当然最好的情况是在 DBA 的监控下根据业务的繁忙情况按照 Buffer Pool 的命中率来设置：

show engine innodb status\G

```
----------------------
BUFFER POOL AND MEMORY
----------------------
Total large memory allocated 137428992
Dictionary memory allocated 416116
Buffer pool size   8192
Free buffers       5867
Database pages     2324
Old database pages 877
Modified db pages  0
Pending reads      0
Pending writes: LRU 0, flush list 0, single page 0
Pages made young 0, not young 0
0.00 youngs/s, 0.00 non-youngs/s
Pages read 2248, created 76, written 601
0.00 reads/s, 0.00 creates/s, 0.00 writes/s
Buffer pool hit rate 683 / 1000,  young-making rate 0 / 1
Pages read ahead 0.00/s, evicted without access 0.00/s,
LRU len: 2324, unzip_LRU len: 0
I/O sum[0]:cur[0], unzip sum[0]:cur[0]
--------------
```

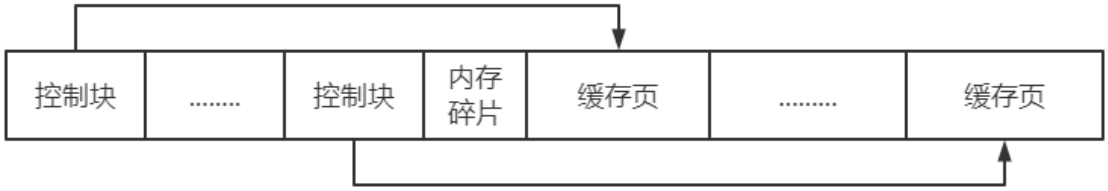
对于读取多的情况,如果没达到 98%以上，都说明 buffer 不够，可以扩，如果给命中都能达到 98%~100%了，而且还有大量的 free page 那说明够用了。当然如果业务不繁忙或者是写多读少的情况下命中率参考意义就不大了。

总的来说，没有专人管理和实时监控的情况下，可以设置为 60%较为稳妥，有专人管理和实时监控的情况下，可以设置为 75%，并根据业务情况适度增大或者缩小。

Buffer Pool 内部组成

Buffer Pool 中默认的缓存页大小和在磁盘上默认的页大小是一样的，都是 16KB。为了更好的管理这些在 Buffer Pool 中的缓存页，InnoDB 为每一个缓存页都创建了一些所谓的控制信息，这些控制信息包括该页所属的表空间编号、页号、缓存页在 Buffer Pool 中的地址、链表节点信息、一些锁信息以及 LSN 信息，当然还有一些别的控制信息。

每个缓存页对应的控制信息占用的内存大小是相同的，我们称为控制块。控制块和缓存页是一一对应的，它们都被存放到 Buffer Pool 中，其中控制块被存放到 Buffer Pool 的前边，缓存页被存放到 Buffer Pool 后边，所以整个 Buffer Pool 对应的内存空间看起来就是这样的：

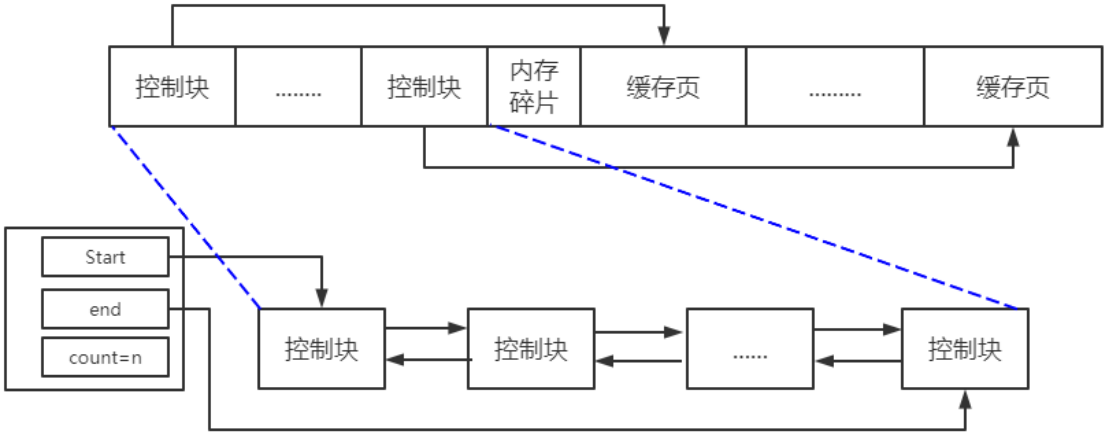


每个控制块大约占用缓存页大小的 5%，而我们设置的 innodb_buffer_pool_size 并不包含这部分控制块占用的内存空间大小，也就是说 InnoDB 在为 Buffer Pool 向操作系统申请连续的内存空间时，这片连续的内存空间一般会比 innodb_buffer_pool_size 的值大 5%左右。

free 链表的管理

最初启动 MySQL 服务器的时候，需要完成对 Buffer Pool 的初始化过程，就是先向操作系统申请 Buffer Pool 的内存空间，然后把它划分成若干对控制块和缓存页。但是此时并没有真实的磁盘页被缓存到 Buffer Pool 中（因为还没有用到），之后随着程序的运行，会不断的有磁盘上的页被缓存到 Buffer Pool 中。

那么问题来了，从磁盘上读取一个页到 Buffer Pool 中的时候该放到哪个缓存页的位置呢？或者说怎么区分 Buffer Pool 中哪些缓存页是空闲的，哪些已经被使用了呢？最好在某个地方记录一下 Buffer Pool 中哪些缓存页是可用的，这个时候缓存页对应的控制块就派上大用场了，我们可以把所有空闲的缓存页对应的控制块作为一个节点放到一个链表中，这个链表也可以被称作 free 链表（或者说空闲链表）。刚刚完成初始化的 Buffer Pool 中所有的缓存页都是空闲的，所以每一个缓存页对应的控制块都会被加入到 free 链表中，假设该 Buffer Pool 中可容纳的缓存页数量为 n，那增加了 free 链表的效果图就是这样的：



有了这个 free 链表之后，每当需要从磁盘中加载一个页到 Buffer Pool 中时，就从 free 链表中取一个空闲的缓存页，并且把该缓存页对应的控制块的信息填上（就是该页所在的表空间、页号之类的信息），然后把该缓存页对应的 free 链表节点从链表中移除，表示该缓存页已经被使用了。

缓存页的哈希处理

我们前边说过，当我们需要访问某个页中的数据时，就会把该页从磁盘加载到 Buffer Pool 中，如果该页已经在 Buffer Pool 中的话直接使用就可以了。那么问题也就来了，我们怎么知道该页在不在 Buffer Pool 中呢？难不成需要依次遍历 Buffer Pool 中各个缓存页么？

我们其实是根据表空间号 + 页号来定位一个页的，也就相当于表空间号 + 页号是一个 key，缓存页就是对应的 value，怎么通过一个 key 来快速找着一个 value 呢？

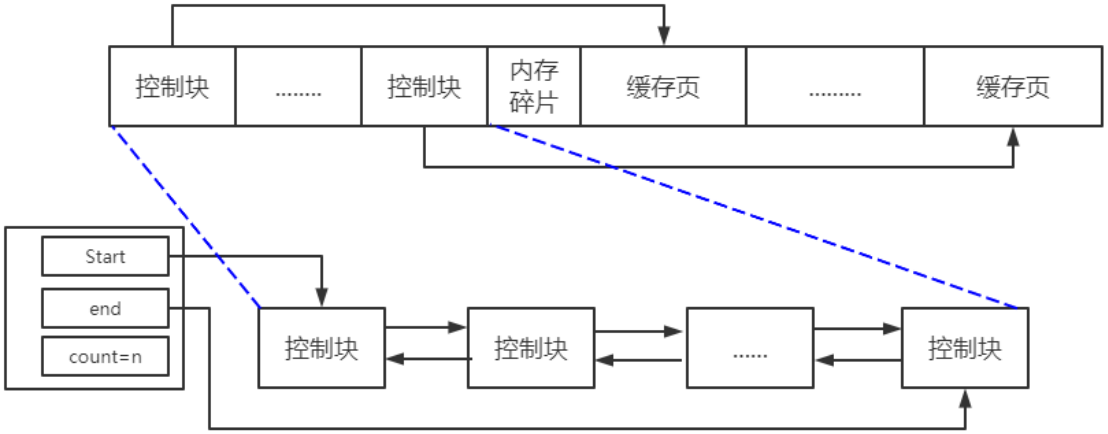
所以我们可以用表空间号 + 页号作为 key，缓存页作为 value 创建一个哈希表，在需要访问某个页的数据时，先从哈希表中根据表空间号 + 页号看看有没有对应的缓存页，如果有，直接使用该缓存页就好，如果没有，那就从 free 链表中选一个空闲的缓存页，然后把磁盘中对应的页加载到该缓存页的位置。

flush 链表的管理

如果我们修改了 Buffer Pool 中某个缓存页的数据，那它就和磁盘上的页不一致了，这样的缓存页也被称为脏页（英文名：dirty page）。当然，最简单的做法就是每发生一次修改就立即同步到磁盘上对应的页上，但是频繁的往磁盘中写数据会严重的影响程序的性能。所以每次修改缓存页后，我们并不着急立即把修改同步到磁盘上，而是在未来的某个时间点进行同步。

但是如果不立即同步到磁盘的话，那之后再同步的时候我们怎么知道 Buffer Pool 中哪些页是脏页，哪些页从来没被修改过呢？总不能把所有的缓存页都同步到磁盘上吧，假如 Buffer Pool 被设置的很大，比方说 300G，那一次性同步会非常慢。

所以，需要再创建一个存储脏页的链表，凡是修改过的缓存页对应的控制块都会作为一个节点加入到一个链表中，因为这个链表节点对应的缓存页都是需要被刷新到磁盘上的，所以也叫 flush 链表。链表的构造和 free 链表差不多。



LRU 链表的管理

缓存不够的窘境

Buffer Pool 对应的内存大小毕竟是有限的，如果需要缓存的页占用的内存大小超过了 Buffer Pool 大小，也就是 free 链表中已经没有多余的空闲缓存页的时候该咋办？当然是把某些旧的缓存页从 Buffer Pool 中移除，然后再把新的页放进来，那么问题来了，移除哪些缓存页呢？

为了回答这个问题，我们还需要回到我们设立 Buffer Pool 的初衷，我们就是想减少和磁盘的 IO 交互，最好每次在访问某个页的时候它都已经被缓存到 Buffer Pool 中了。假设我们一共访问了 n 次页，那么被访问的页已经在缓存中的次数除以 n 就是所谓的缓存命中率，我们的期望就是让缓存命中率越高越好。

从这个角度出发，回想一下我们的微信聊天列表，排在前边的都是最近很频繁使用的，排在后边的自然就是最近很少使用的，假如列表能容纳下的联系人有限，你是会把最近很频繁使用的留下还是最近很少使用的留下呢？当然是留下最近很频繁使用的了。

简单的 LRU 链表

管理 Buffer Pool 的缓存页其实也是这个道理，当 Buffer Pool 中不再有空闲的缓存页时，就需要淘汰掉部分最近很少使用的缓存页。不过，我们怎么知道哪些缓存页最近频繁使用，哪些最近很少使用呢？

再创建一个链表，由于这个链表是为了按照最近最少使用的原则去淘汰缓存页的，所以这个链表可以被称为 LRU 链表（LRU 的英文全称：Least Recently Used）。当我们需要访问某个页时，可以这样处理 LRU 链表：

如果该页不在 Buffer Pool 中，在把该页从磁盘加载到 Buffer Pool 中的缓存页时，就把该缓存页对应的控制块作为节点塞到 LRU 链表的头部。

如果该页已经缓存在 Buffer Pool 中，则直接把该页对应的控制块移动到 LRU 链表的头部。

也就是说：只要我们使用到某个缓存页，就把该缓存页调整到 LRU 链表的头部，这样 LRU 链表尾部就是最近最少使用的缓存页。所以当 Buffer Pool 中的空闲缓存页使用完时，到 LRU 链表的尾部找些缓存页淘汰就行了。

缓存机制一直是各种系统提高性能的首选手段，使用了缓存就必然有缓存淘汰的需求，LRU 就是常见的算法，再比如 Redis 中的缓存淘汰算法也一直是面试中的高频考点，其中就有 volatile-lru、allkeys-lru 等淘汰策略。

所以关于 LRU 缓存的实现在算法笔试中几乎所有的大厂都有考察，具体的实现，大家可以到《大厂高频笔试题 Top20、精选 LeetCode 热题 100 详解-第三期》中查看，具体地址：

https://vip.tulingxueyuan.cn/detail/p_61a344ffe4b09240f0e4b59e/8。

划分区域的 LRU 链表

但是这种实现存在两种比较尴尬的情况：

情况一：InnoDB 提供了预读（英文名：read ahead）。所谓预读，就是 InnoDB 认为执行当前的请求可能之后会读取某些页面，就预先把它们加载到 Buffer Pool 中。根据触发方式的不同，预读又可以细分为下边两种：

线性预读

InnoDB 提供了一个系统变量 innodb_read_ahead_threshold，如果顺序访问了某个区（extent）的页面超过这个系统变量的值，就会触发一次异步读取下一个区中全部的页面到 Buffer Pool 的请求。

这个 innodb_read_ahead_threshold 系统变量的值默认是 56，我们可以在服务器启动时通过启动参数或者服务器运行过程中直接调整该系统变量的值，取值范围是 0~64。

随机预读

如果 Buffer Pool 中已经缓存了某个区的 13 个连续的页面，不论这些页面是不是顺序读取的，都会触发一次异步读取本区中所有其他的页面到 Buffer Pool 的请求。InnoDB 同时提供了 innodb_random_read_ahead 系统变量，它的默认值为 OFF。

show variables like '%_read_ahead%';

```
+-----------------------------+-------+
| Variable_name               | Value |
+-----------------------------+-------+
| innodb_random_read_ahead    | OFF   |
| innodb_read_ahead_threshold | 56    |
+-----------------------------+-------+
```

并且可以根据通过执行 show engine innodb status 命令显示的三个参数判断 read-ahead 算法的有效性：

read_ahead、read_ahead_evicted、read_ahead_rnd

```
Pages read ahead 0.00/s, evicted without access 0.00/s, Random read ahead 0.00/s
```

如果通过监控发现，这个预读功能长期有效性很低，可以考虑关闭这个预读功能。

如果预读到 Buffer Pool 中的页成功的被使用到，那就可以极大的提高语句执行的效率。可是如果用不到呢？这些预读的页都会放到 LRU 链表的头部，但是如果此时 Buffer Pool 的容量不太大而且很多预读的页面都没有用到的话，这就会导致处在 LRU 链表尾部的一些缓存页会很快的被淘汰掉，也就是所谓的劣币驱逐良币，会大大降低缓存命中率。

情况二：应用程序可能会写一些需要扫描全表的查询语句（比如没有建立合适的索引或者压根儿没有 WHERE 子句的查询）。

扫描全表意味着什么？意味着将访问到该表所在的所有页！假设这个表中记录非常多的话，那该表会占用特别多的页，当需要访问这些页时，会把它们统统都加载到 Buffer Pool 中，这也就意味着 Buffer Pool 中的所有页都被换了一次血，其他查询语句在执行时又得执行一次从磁盘加载到 Buffer Pool 的操作。而这种全表扫描的语句执行的频率也不高，每次执行都要把 Buffer Pool 中的缓存页换一次血，这严重的影响到其他查询对 Buffer Pool 的使用，从而大大降低了缓存命中率。

总结一下上边说的可能降低 Buffer Pool 的两种情况：

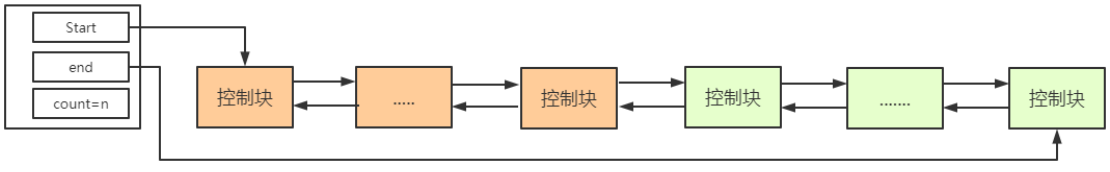
加载到 Buffer Pool 中的页不一定被用到。

如果非常多的使用频率偏低的页被同时加载到 Buffer Pool 时，可能会把那些使用频率非常高的页从 Buffer Pool 中淘汰掉。

因为有这两种情况的存在，所以 InnoDB 把这个 LRU 链表按照一定比例分成两截，分别是：

一部分存储使用频率非常高的缓存页，所以这一部分链表也叫做热数据，或者称 young 区域。

另一部分存储使用频率不是很高的缓存页，所以这一部分链表也叫做冷数据，或者称 old 区域。



我们是按照某个比例将 LRU 链表分成两半的，不是某些节点固定是 young 区域的，某些节点固定是 old 区域的，随着程序的运行，某个节点所属的区域也可能发生变化。那这个划分成两截的比例怎么确定呢？对于 InnoDB 存储引擎来说，我们可以通过查看系统变量 innodb_old_blocks_pct 的值来确定 old 区域在 LRU 链表中所占的比例，比方说这样：

SHOW VARIABLES LIKE 'innodb_old_blocks_pct';

```
mysql>  SHOW VARIABLES LIKE 'innodb_old_blocks_pct';
+-----------------------+-------+
| Variable_name         | Value |
+-----------------------+-------+
| innodb_old_blocks_pct | 37    |
+-----------------------+-------+
1 row in set (0.01 sec)
```

从结果可以看出来，默认情况下，old 区域在 LRU 链表中所占的比例是 37%，也就是说 old 区域大约占 LRU 链表的 3/8。这个比例我们是可以设置的，我们可以在启动时修改 innodb_old_blocks_pct 参数来控制 old 区域在 LRU 链表中所占的比例。在服务器运行期间，我们也可以修改这个系统变量的值，不过需要注意的是，这个系统变量属于全局变量。

有了这个被划分成 young 和 old 区域的 LRU 链表之后，InnoDB 就可以针对我们上边提到的两种可能降低缓存命中率的情况进行优化了：

针对预读的页面可能不进行后续访问情况的优化：

InnoDB 规定，当磁盘上的某个页面在初次加载到 Buffer Pool 中的某个缓存页时，该缓存页对应的控制块会被放到 old 区域的头部。这样针对预读到 Buffer Pool 却不进行后续访问的页面就会被逐渐从 old 区域逐出，而不会影响 young 区域中被使用比较频繁的缓存页。

针对全表扫描时，短时间内访问大量使用频率非常低的页面情况的优化：

在进行全表扫描时，虽然首次被加载到 Buffer Pool 的页被放到了 old 区域的头部，但是后续会被马上访问到，每次进行访问的时候又会把该页放到 young 区域的头部，这样仍然会把那些使用频率比较高的页面给顶下去。

有同学会想：可不可以在第一次访问该页面时不将其从 old 区域移动到 young 区域的头部，后续访问时再将其移动到 young 区域的头部。回答是：行不通！因为 InnoDB 规定每次去页面中读取一条记录时，都算是访问一次页面，而一个页面中可能会包含很多条记录，也就是说读取完某个页面的记录就相当于访问了这个页面好多次。

全表扫描有一个特点，那就是它的执行频率非常低，出现了全表扫描的语句也是我们应该尽快优化的对象。而且在执行全表扫描的过程中，即使某个页面中有很多条记录，也就是去多次访问这个页面所花费的时间也是非常少的。

所以在对某个处在 old 区域的缓存页进行第一次访问时就在它对应的控制块中记录下来这个访问时间，如果后续的访问时间与第一次访问的时间在某个时间间隔内，那么该页面就不会被从 old 区域移动到 young 区域的头部，否则将它移动到 young 区域的头部。上述的这个间隔时间是由系统变量 innodb_old_blocks_time 控制的：

SHOW VARIABLES LIKE 'innodb_old_blocks_time';

```
mysql> SHOW VARIABLES LIKE 'innodb_old_blocks_time';
+------------------------+-------+
| Variable_name          | Value |
+------------------------+-------+
| innodb_old_blocks_time | 1000  |
+------------------------+-------+
1 row in set (0.01 sec)
```

这个 innodb_old_blocks_time 的默认值是 1000，它的单位是毫秒，也就意味着对于从磁盘上被加载到 LRU 链表的 old 区域的某个页来说，如果第一次和最后一次访问该页面的时间间隔小于 1s（很明显在一次全表扫描的过程中，多次访问一个页面中的时间不会超过 1s），那么该页是不会被加入到 young 区域的，当然，像 innodb_old_blocks_pct 一样，我们也可以在服务器启动或运行时设置 innodb_old_blocks_time 的值，这里需要注意的是，如果我们把 innodb_old_blocks_time 的值设置为 0，那么每次我们访问一个页面时就会把该页面放到 young 区域的头部。

综上所述，正是因为将 LRU 链表划分为 young 和 old 区域这两个部分，又添加了 innodb_old_blocks_time 这个系统变量，才使得预读机制和全表扫描造成的缓存命中率降低的问题得到了遏制，因为用不到的预读页面以及全表扫描的页面都只会被放到 old 区域，而不影响 young 区域中的缓存页。

更进一步优化 LRU 链表

对于 young 区域的缓存页来说，我们每次访问一个缓存页就要把它移动到 LRU 链表的头部，这样开销是不是太大？

毕竟在 young 区域的缓存页都是热点数据，也就是可能被经常访问的，这样频繁的对 LRU 链表进行节点移动操作也会拖慢速度？为了解决这个问题，MySQL 中还有一些优化策略，比如只有被访问的缓存页位于 young 区域的 1/4 的后边，才会被移动到 LRU 链表头部，这样就可以降低调整 LRU 链表的频率，从而提升性能

还有没有什么别的针对 LRU 链表的优化措施呢？当然还有，我们这里不继续说了，更多的需要看 MySQL 的源码，但是不论怎么优化，出发点就是：尽量高效的提高 Buffer Pool 的缓存命中率。

其他的一些链表

为了更好的管理 Buffer Pool 中的缓存页，除了我们上边提到的一些措施，InnoDB 们还引进了其他的一些链表，比如 unzip LRU 链表用于管理解压页，zip clean 链表用于管理没有被解压的压缩页，zip free 数组中每一个元素都代表一个链表，它们组成所谓的伙伴系统来为压缩页提供内存空间等等。

刷新脏页到磁盘

后台有专门的线程每隔一段时间负责把脏页刷新到磁盘，这样可以不影响用户线程处理正常的请求。主要有两种刷新路径：

1、从 LRU 链表的冷数据中刷新一部分页面到磁盘。

后台线程会定时从 LRU 链表尾部开始扫描一些页面，扫描的页面数量可以通过系统变量 innodb_lru_scan_depth 来指定，如果从里边儿发现脏页，会把它们刷新到磁盘。这种刷新页面的方式被称之为 BUF_FLUSH_LRU。

2、从 flush 链表中刷新一部分页面到磁盘。

后台线程也会定时从 flush 链表中刷新一部分页面到磁盘，刷新的速率取决于当时系统是不是很繁忙。这种刷新页面的方式被称之为 BUF_FLUSH_LIST。

有时候后台线程刷新脏页的进度比较慢，导致用户线程在准备加载一个磁盘页到 Buffer Pool 时没有可用的缓存页，这时就会尝试看看 LRU 链表尾部有没有可以直接释放掉的未修改页面，如果没有的话会不得不将 LRU 链表尾部的一个脏页同步刷新到磁盘（和磁盘交互是很慢的，这会降低处理用户请求的速度）。这种刷新单个页面到磁盘中的刷新方式被称之为 BUF_FLUSH_SINGLE_PAGE。

当然，有时候系统特别繁忙时，也可能出现用户线程批量的从 flush 链表中刷新脏页的情况，很显然在处理用户请求过程中去刷新脏页是一种严重降低处理速度的行为，这属于一种迫不得已的情况。

多个 Buffer Pool 实例

我们上边说过，Buffer Pool 本质是 InnoDB 向操作系统申请的一块连续的内存空间，在多线程环境下，访问 Buffer Pool 中的各种链表都需要加锁处理，在 Buffer Pool 特别大而且多线程并发访问特别高的情况下，单一的 Buffer Pool 可能会影响请求的处理速度。所以在 Buffer Pool 特别大的时候，我们可以把它们拆分成若干个小的 Buffer Pool，每个 Buffer Pool 都称为一个实例，它们都是独立的，独立的去申请内存空间，独立的管理各种链表，所以在多线程并发访问时并不会相互影响，从而提高并发处理能力。

我们可以在服务器启动的时候通过设置 innodb_buffer_pool_instances 的值来修改 Buffer Pool 实例的个数

那每个 Buffer Pool 实例实际占多少内存空间呢？其实使用这个公式算出来的：innodb_buffer_pool_size/innodb_buffer_pool_instances

也就是总共的大小除以实例的个数，结果就是每个 Buffer Pool 实例占用的大小。

不过也不是说 Buffer Pool 实例创建的越多越好，分别管理各个 Buffer Pool 也是需要性能开销的，InnoDB 规定：innodb_buffer_pool_instances 能设置的最大值是 64，而且当 innodb_buffer_pool_size（默认 128M）的值小于 1G 的时候设置多个实例是无效的，InnoDB 会默认把 innodb_buffer_pool_instances 的值修改为 1。

按照官方的说明，最佳的 innodb_buffer_pool_instances 的数量是，innodb_buffer_pool_size 除以 innodb_buffer_pool_instances，可以让每个 Buffer Pool 实例达到 1 个 G，这个公式在 8.0 和 5.7 中都适用。

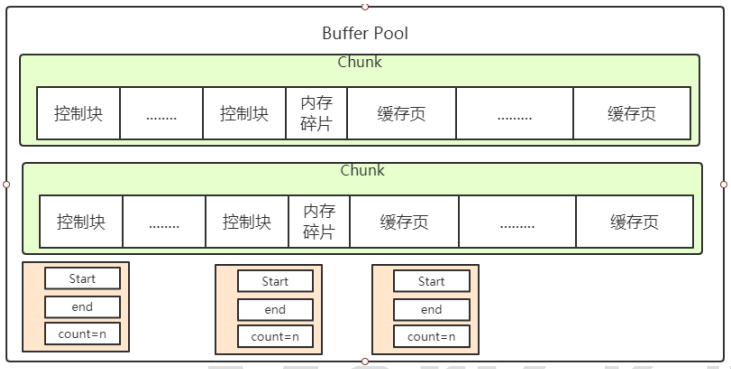
https://dev.mysql.com/doc/refman/8.0/en/innodb-multiple-buffer-pools.html

> To enable multiple buffer pool instances, set the `innodb_buffer_pool_instances` configuration option to a value greater than 1 (the default) up to 64 (the maximum). This option takes effect only when you set `innodb_buffer_pool_size` to a size of 1GB or more. The total size you specify is divided among all the buffer pools. For best efficiency, specify a combination of `innodb_buffer_pool_instances` and `innodb_buffer_pool_size` so that each buffer pool instance is at least 1GB.

innodb_buffer_pool_chunk_size

在 MySQL 5.7.5 之前，Buffer Pool 的大小只能在服务器启动时通过配置 innodb_buffer_pool_size 启动参数来调整大小，在服务器运行过程中是不允许调整该值的。不过 MySQL 在 5.7.5 以及之后的版本中支持了在服务器运行过程中调整 Buffer Pool 大小的功能，

但是有一个问题，就是每次当我们要重新调整 Buffer Pool 大小时，都需要重新向操作系统申请一块连续的内存空间，然后将旧的 Buffer Pool 中的内容复制到这一块新空间，这是极其耗时的。所以 MySQL 决定不再一次性为某个 Buffer Pool 实例向操作系统申请一大片连续的内存空间，而是以一个所谓的 chunk 为单位向操作系统申请空间。也就是说一个 Buffer Pool 实例其实是由若干个 chunk 组成的，一个 chunk 就代表一片连续的内存空间，里边儿包含了若干缓存页与其对应的控制块：



正是因为发明了这个 chunk 的概念，我们在服务器运行期间调整 Buffer Pool 的大小时就是以 chunk 为单位增加或者删除内存空间，而不需要重新向操作系统申请一片大的内存，然后进行缓存页的复制。这个所谓的 chunk 的大小是我们在启动操作 MySQL 服务器时通过 innodb_buffer_pool_chunk_size 启动参数指定的，它的默认值是 134217728，也就是 128M。不过需要注意的是，innodb_buffer_pool_chunk_size 的值只能在服务器启动时指定，在服务器运行过程中是不可以修改的。

Buffer Pool 的缓存页除了用来缓存磁盘上的页面以外，还可以存储锁信息、自适应哈希索引等信息。

查看 Buffer Pool 的状态信息

MySQL 给我们提供了 SHOW ENGINE INNODB STATUS 语句来查看关于 InnoDB 存储引擎运行过程中的一些状态信息，其中就包括 Buffer Pool 的一些信息，我们看一下（为了突出重点，我们只把输出中关于 Buffer Pool 的部分提取了出来）：

SHOW ENGINE INNODB STATUS\G

```
----------------------
BUFFER POOL AND MEMORY
----------------------
Total large memory allocated 137428992
Dictionary memory allocated 183405
Buffer pool size   8191
Free buffers       6958
Database pages     1196
Old database pages 421
Modified db pages  0
Pending reads      0
Pending writes: LRU 0, flush list 0, single page 0
Pages made young 0, not young 0
0.00 youngs/s, 0.00 non-youngs/s
Pages read 292, created 904, written 4224
0.00 reads/s, 0.00 creates/s, 0.00 writes/s
No buffer pool page gets since the last printout
Pages read ahead 0.00/s, evicted without access 0.00/s, Random read ahead 0.00/s
LRU len: 1196, unzip_LRU len: 0
I/O sum[0]:cur[0], unzip sum[0]:cur[0]
--------------
```

这里边的每个值都代表什么意思如下，知道即可：

Total memory allocated：代表 Buffer Pool 向操作系统申请的连续内存空间大小，包括全部控制块、缓存页、以及碎片的大小。

Dictionary memory allocated：为数据字典信息分配的内存空间大小，注意这个内存空间和 Buffer Pool 没啥关系，不包括在 Total memory allocated 中。

Buffer pool size：代表该 Buffer Pool 可以容纳多少缓存页，注意，单位是页！

Free buffers：代表当前 Buffer Pool 还有多少空闲缓存页，也就是 free 链表中还有多少个节点。

Database pages：代表 LRU 链表中的页的数量，包含 young 和 old 两个区域的节点数量。

Old database pages：代表 LRU 链表 old 区域的节点数量。

Modified db pages：代表脏页数量，也就是 flush 链表中节点的数量。

Pending reads：正在等待从磁盘上加载到 Buffer Pool 中的页面数量。

当准备从磁盘中加载某个页面时，会先为这个页面在 Buffer Pool 中分配一个缓存页以及它对应的控制块，然后把这个控制块添加到 LRU 的 old 区域的头部，但是这个时候真正的磁盘页并没有被加载进来，Pending reads 的值会跟着加 1。

Pending writes LRU：即将从 LRU 链表中刷新到磁盘中的页面数量。

Pending writes flush list：即将从 flush 链表中刷新到磁盘中的页面数量。

Pending writes single page：即将以单个页面的形式刷新到磁盘中的页面数量。

Pages made young：代表 LRU 链表中曾经从 old 区域移动到 young 区域头部的节点数量。

Page made not young：在将 innodb_old_blocks_time 设置的值大于 0 时，首次访问或者后续访问某个处在 old 区域的节点时由于不符合时间间隔的限制而不能将其移动到 young 区域头部时，Page made not young 的值会加 1。

youngs/s：代表每秒从 old 区域被移动到 young 区域头部的节点数量。

non-youngs/s：代表每秒由于不满足时间限制而不能从 old 区域移动到 young 区域头部的节点数量。

Pages read、created、written：代表读取，创建，写入了多少页。后边跟着读取、创建、写入的速率。

Buffer pool hit rate：表示在过去某段时间，平均访问 1000 次页面，有多少次该页面已经被缓存到 Buffer Pool 了。

young-making rate：表示在过去某段时间，平均访问 1000 次页面，有多少次访问使页面移动到 young 区域的头部了。

not (young-making rate)：表示在过去某段时间，平均访问 1000 次页面，有多少次访问没有使页面移动到 young 区域的头部。

LRU len：代表 LRU 链表中节点的数量。

unzip_LRU：代表 unzip_LRU 链表中节点的数量。

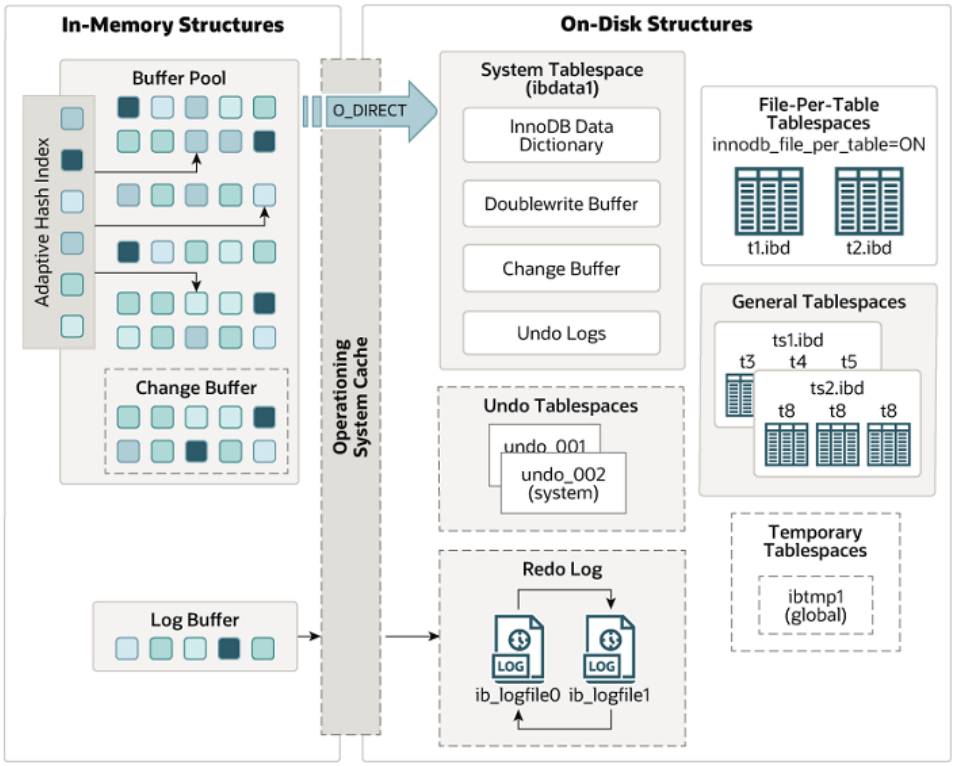
I/O sum：最近 50s 读取磁盘页的总数。

I/O cur：现在正在读取的磁盘页数量。

I/O unzip sum：最近 50s 解压的页面数量。

I/O unzip cur：正在解压的页面数量。

InnoDB 体系图补充说明



其中的 Insert/Change Buffer 主要是用于对二级索引的写入优化，Undo 空间则是 undo 日志一般放在系统表空间，但是通过参数配置后，也可以用独立表空间存放，所以用虚线表示。通用表空间和独立表空间不同，通用表空间是允许多个表存储数据的共享表空间。

在线文档：

http://note.youdao.com/noteshare?id=f5a6c3e4a7407bbe9f8651a287d4767f&sub=C739DBC1F2CE4170B5F1D11DAECF503B