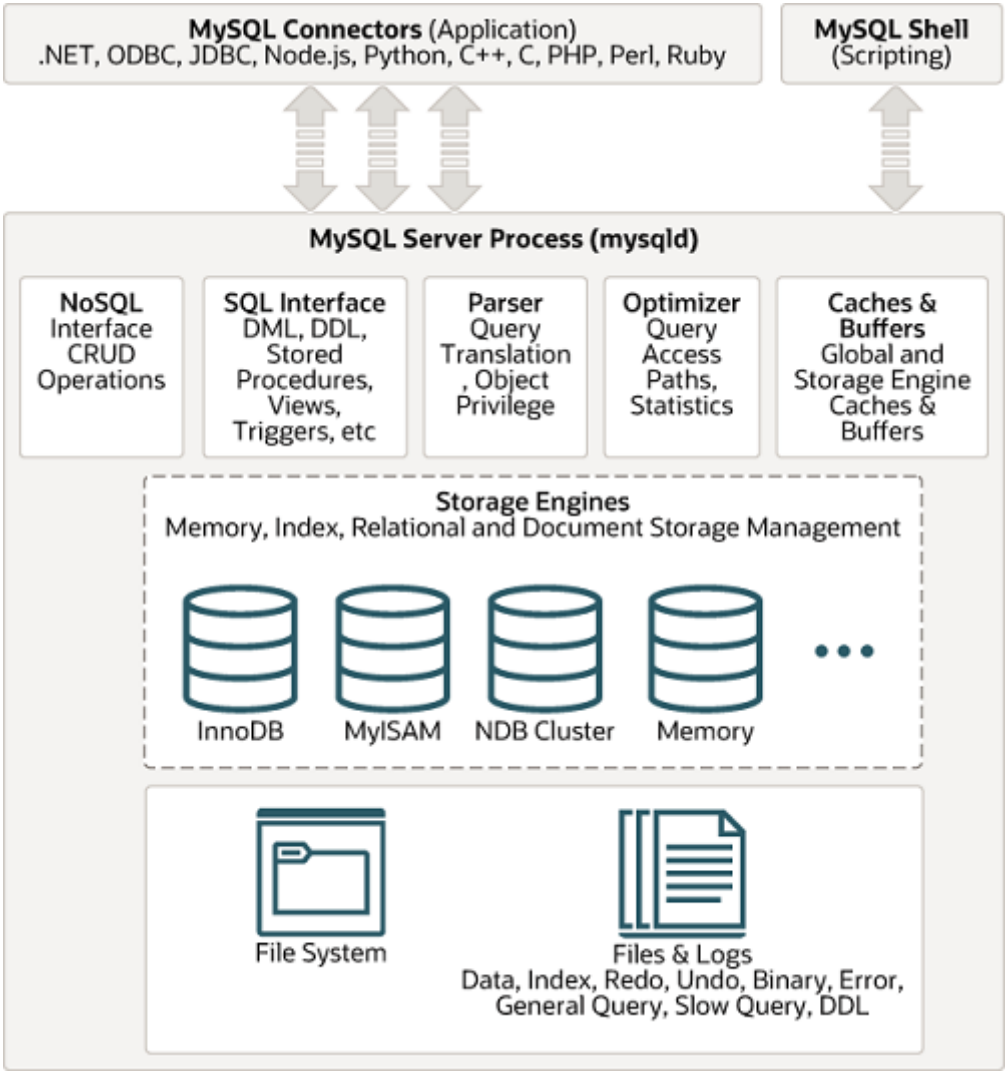
第二节 了解Mysql关系型数据库的整体设计

了解Mysql的前世今生

Msyql的发展历史简述如下：

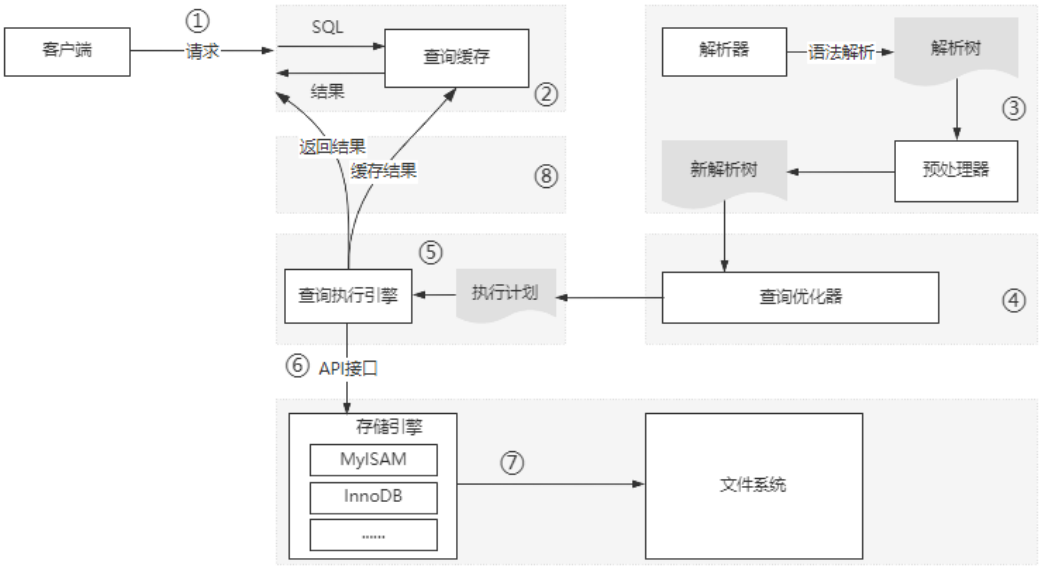
- 1995年5月23日，第一个内部版本的Mysql发布
- 然后在第二年，也就是1996年，MySQL官方正式发行版(3.11.1)对外公布。
- 2001年推出MySQL 4.1 版本，这个版本集成了InnoDB引擎，支持事务、子查询等能力
- 2008年2月，Sun花了10亿美金首购了Mysql，MySQL和Sun合并之后，推出了MySQL 5.1GA版和MySQL 5.4 Beta版。
- 2009年，数据库老大Oracle收购了Sun和Mysql
- 2019年，Mysql发布了8.0版本（目前最新GA版本应该是8.0.28），官方表示 MySQL 8 要比 MySQL 5.7 快 2 倍，还带来了大量的改进和更快的性能。

Mysql整体架构

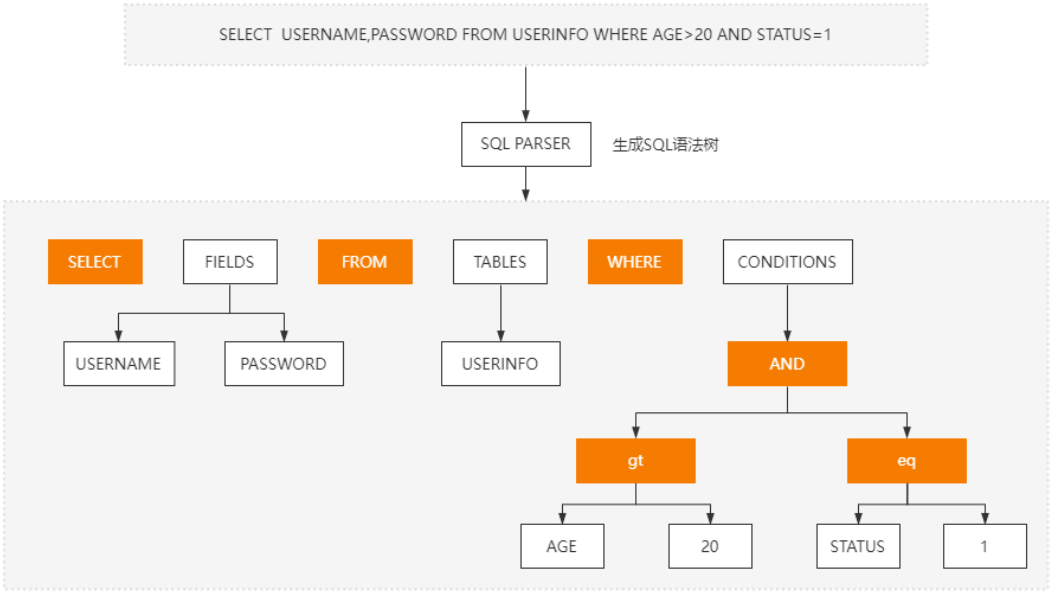


Parser解析器

> 注意， mysql8.0没有查询缓存了。



> Mysql利用了Bison来构建解析树，它是一种通用解析器生成器



如果写了一个语法完全正确的树，但是表或者字段不存在，还是在解析的时候报错，因为解析器处理之后，还有一个预处理器，它用来判断解析树的语义是否正确，也就是表名和字段名是否存在，预处理后生成一个新的解析树。

Optimizer优化器

Optimizer是Mysql中的查询优化器。

一条SQL语句有很多中执行方式，最终以那种方式来执行是查询优化器来决定的。

举两个简单的例子：

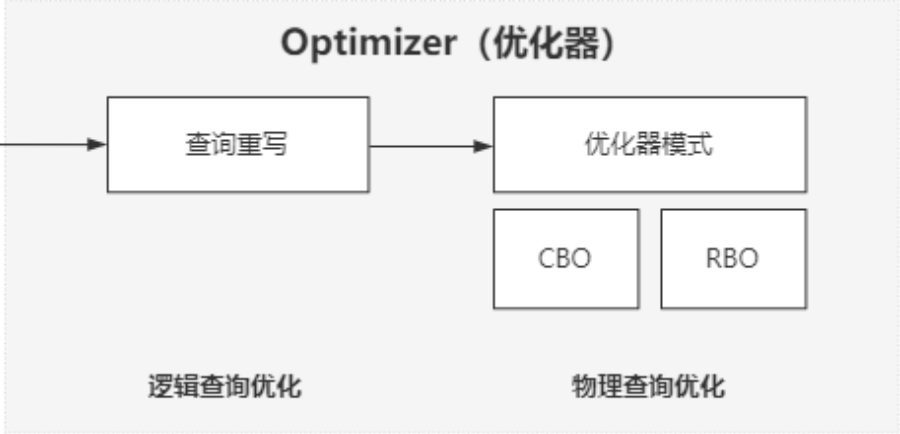
1. 当我们对多张表进行关联查询的时候，以哪个表的数据作为基准表。
2. 有多个索引可以使用的时候，选择哪个索引。

实际上，对于每一种数据库来说，优化器的模块都是必不可少的，他们通过复杂的算法实现尽可能优化查询效率的目标。

查询优化器的工作原理是：根据解析树生成的不同的执行计划（Execution Plan），然后选择一种最优的执行计划，mysql 里面使用的是基于开销（cost）的优化器，最终采用开销最小的作为最终执行的方案。

优化器最终会把解析树变成一个查询执行计划，查询执行计划一个数据结构。这个执行计划不一定是最优的结果，因为 mysql 也有可能覆盖不到所有的执行计划。

SQL优化器包括两项工作分别是逻辑优化、物理优化，运行流程如下。



这两项工作都要对语法分析树的形态进行修改，把语法分析树变为查询树。

逻辑优化

这个阶段主要是使用关系代数对SQL语句做一些等价，使得SQL执行效率最高。

对条件表达式进行等价谓词重写、条件简化，对视图进行重写，对子查询进行优化，对连接语义进行了外连接消除、嵌套连接消除等。

为了让大家更清晰的理解这个阶段做的事情，我们通过几个案例来说明一下。

子查询优化

子查询是SQL语句中出现频率很高的类型，而且也是较为耗时的操作。
而子查询可能会出现在SQL中的目标列、from子句、where子句、JOIN/ON子句、GROUPBY子句、Having子句等。
子查询出现在不同的位置，优化的方式和影响也不同.

子查询合并

在某些条件下，多个子查询能合并成一个子查询，这样就可以把多次表扫描，多次连接操作减少为单次表扫描和单次连接。

```
SELECT * FROM t1 WHERE a1<10 AND (
   EXISTS(SELECT a2 FROM t2 WHERE t2.a2<5 AND t2.b2=1) OR
   EXISTS(SELECT a2 FROM t2 WHERE t2.a2<5 AND t2.b2=2)
);
```

可优化成

```
SELECT * FROM t1 WHERE a1<10 AND (
EXISTS(SELECT a2 FROM t2 WHERE t2.a2<5 AND (t2.b2=1 OR
t2.b2=2)
);
```

等价谓词重写

数据库执行引擎中，对一些谓词的处理效率要高于其他谓词，因此，把一些逻辑表达式重写成等价的并且更高效的形式，能够提升查询效率。

1. like 规则

   ```
   SELECT * FROM USERINFO WHERE name LIKE 'Abc%'
   ```

   优化后：

   ```
   SELECT * FROM USERINFO WHERE name >='Abc' AND name
   <'Abd'
   ```

前者针对LIKE这个谓词只能进行全表扫描，而在后者改造中如果**name**列有索引，转化后就变成索引范围内扫描。

2. IN转化OR规则

   IN 转化 OR规则就是IN为此的OR等价重写，即改写IN谓词为等价的OR谓词，从而更好的利用索引进行优化，比如：

```
select * from t1 where age in (10,14,18)
```

优化后：

```
select * from t1 where age =10 or age=14 or age=18
```

IN转化OR规则后，执行效率是否能够提高，需要看数据库中对IN谓词是否只支持全表扫描，如果数据库对IN谓词只支持全表扫描并且OR谓词中的表的age列上存在索引，那么转化后查询效率会得到提高。

条件简化

WHERE、HAVING、ON条件可能是由很多的表达式组成的，而这些表达式在某些情况下存在一定的联系， 数据库可以利用等式和不等式的性质，把WHERE、HAVING、ON条件简化。

常见的优化方式有：

1. 把HAVING条件并入到WHERE条件， 方便统一、集中化解子条件，节约多次化解的时间。
2. 去除表达式中的冗余括号，减少语法分析时产生的AND和OR 树的层次，比如((a AND b) AND (c AND d))简化为a AND b AND c AND d。
3. 常量传递，比如col_1=col_2 AND col_2=3 可以简化成col_1=3 AND col_2=3。
4. 表达式计算， 对可以求解的表达式进行计算，比如WHERE col_1=1+2 转化为 WHERE col_1=3。

....

物理优化

在生成逻辑查询计划后，查询优化器会进一步对查询树进行物理查询优化，物理优化主要解决几个问题：

1. 在单表扫描方式中，选择什么样的单表扫描方式是最优的
2. 对于存在两个表连接时，那种连接方式最优

3. 对于多个表连接，连接顺序有多种组合，那种连接顺序是最优的

物理查询优化一般分为两种

- 基于规则的优化（RBO，Rule-Based Optimizer），这种方式主要是基于一些预置的规则对查询进行优化。
- 基于代价的优化（CBO，Cost-Based Optimizer），这种方式会根据模型计算出各个可能的执行计划的代价，然后选择代价最少的那个。它会利用数据库里面的统计信息来做判断，因此是动态的。

> Mysql默认采用的是CBO的优化方式

基于代价估算是基于CPU代价和IO代价这两个纬度来实现的。

```
总代价=IO代价+CPU代价
```

单表扫描代价

单表扫描代价，单表扫描需要从表上获取元组，直接关联到物理IO的读取，因此不同的单表扫描方式，会有不同的代价

> 元组是关系数据库中的基本概念,关系是一张表,表中的每行(即数据库中的每条记录)就是一个元组,在二维表里,元组也称为记录。

单表数据的获取方式，通常有两种。

- 全表扫描表数据，获取全部的元组，读取表对应的全部数据页
- 局部扫描表数据，获取表的部分元组，读取指定位置对应的数据页

而单表扫描涉及到的算法有很多，比如

1. 顺序扫描，从物理存储上按照存储顺序直接读取表的数据
2. 索引扫描，根据索引键获取索引找到物理元组的位置，再根据该位置从存储中读取数据页面。
3. 并行表扫描， 对同一个表，并行地通过索引的方式获取表的数据，然后将结果合并在一起。
4. ....

对于局部扫描，Optimizer优化器会根据数据量以及元组获取的条件，可能采用顺序读取或者随机读取的方式，这个时候，优化器会使用选择率来决定最终的优化方案。

> 选择率在代价估算模型中占非常重要的地位，它的计算精确程度直接影响最优计划的选择，通常有无参数方法、参数法、抽样法、曲线拟合法等方式。

如果选择率的值很大，意味着采取顺序扫描的方式可能比局部扫描的随机读方式效率更高，因为顺序IO会减少磁盘头移动的等待时间，如果数据库文件在磁盘上没有碎片，那么这种方式对性能的改善更加明显

对于局部扫描，通常会采用索引实现少量数据的读取优化，这是一种随机读取数据的方式。虽然顺序扫描可能比读取许多行的索引扫描花费的时间少，但是如果顺序扫描被执行多次（比如嵌套循环连接的表）

它的整体代价更大，而索引扫描访问的数据页比较少，而且这些也可能会被保存在数据缓冲区中，所以访问速度会更快。

至于最终采用哪种扫描方式，查询优化器会采用代价估算比较代价大小后再决定。

总的来说，对于单表扫描的代价，由于单表扫描需要把数据从存储系统中加载到内存，所以单表扫描的代价需要考虑IO的开销。

1. 顺序扫描主要是IO的开销加上元组懂数据也中解析的开销，开销公式如下：

   ```
   N_page * a_page_IO_time + N_tuple * a_tuple_CPU_time
   ```

2. 索引扫描和其他扫描方式，由于扫描的数据不是所有的元组，所以需要考虑选择率的问题。

   ```
   C_index + N_page_index * a_page_IO_time
   ```

扫描方式	代价估算公式
顺序扫描	N_page * a_page_IO_time + N_tuple * a_tuple_CPU_time
索引扫描	C_index + N_page_index * a_page_IO_time

上述参数说明如下：

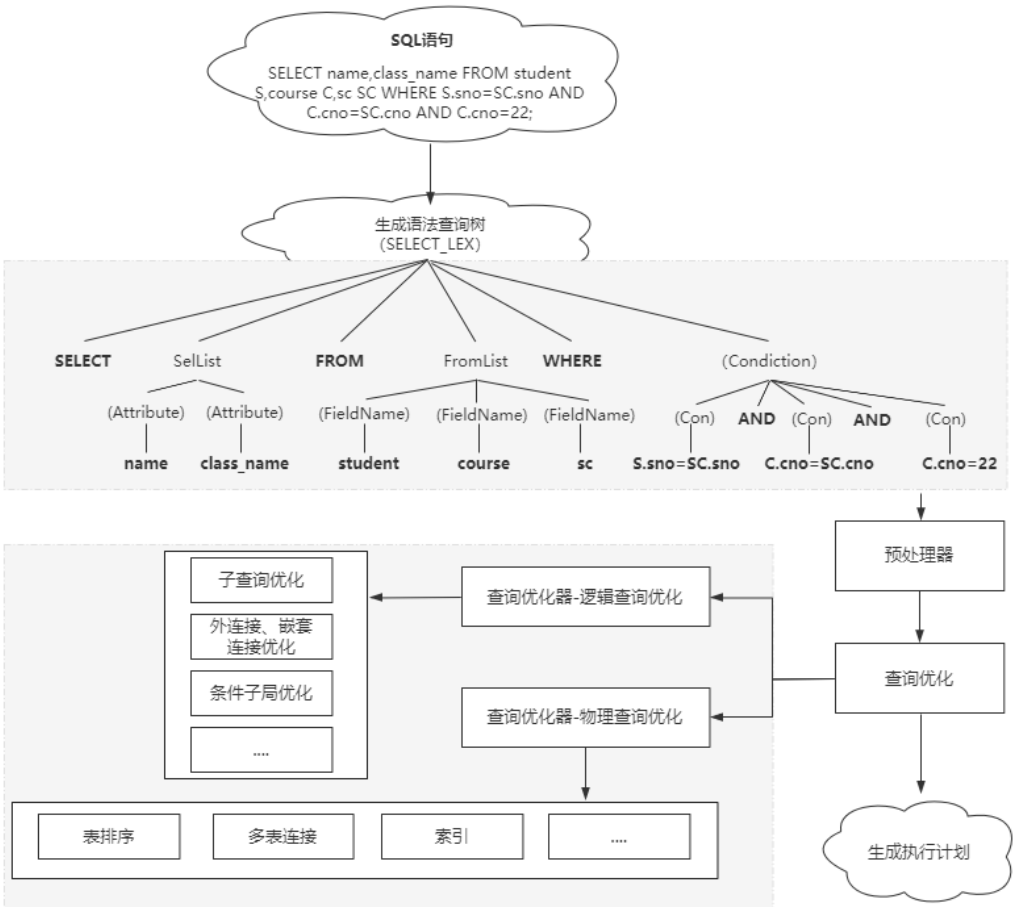
- a_page_IO_time， 一个数据页加载的IO耗时
- N_page，数据页数量
- N_tuple，元组数
- a_tuple_CPU_time，一个元组从数据页中解析的CPU耗时
- C_index，索引的IO耗时
- N_page_index，索引页数量

索引扫描代价

索引是建立在表上的，索引建立的目的是便于快速定位表中的物理元组，加快数据的获取效率，它也是属于物理查询优化阶段做的事情。

对于索引的具体使用和原理，在后续的内容中详细分析。

查询优化器整体结构



执行计划

当一条SQL语句经过Optimizer优化器优化之后，会针对这条SQL产生一个执行计划，从这个执行计划中可以得知Mysql会如何执行这条SQL。

在Mysql中，提供了EXPLAIN关键字来查看指定SQL的执行计划，通过该计划，可以得到以下信息。

- 表的读取顺序
- 数据读取操作的操作类型
- 哪些索引可以使用
- 哪些索引被实际使用
- 表之间的引用
- 每张表有多少行被优化器查

Storage Engines

一条SQL语句经过解析和优化之后，最终还是需要从某一个地方获取数据，那么：

1. 从逻辑的角度来说，我们的数据是放在哪里的，或者说放在一个什么结构里面？
2. 执行计划在哪里执行？是谁去执行？

这里就涉及到存储引擎(Storage Engines)的概念了。

存储引擎是Mysql中才有的概念，它表示数据如何存储、如何提取、如何更新等具体的实现，不同存储引擎的底层实现方式不同，因此会呈现不同存储引擎独特的功能和特点。

因为在关系型数据库中数据的存储形式是二维表，所以存储引擎也可以称为表类型。

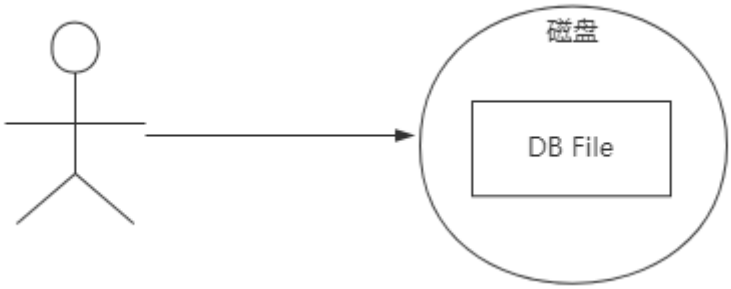
在Mysql中支持多种存储引擎，最常用的引擎是MyISAM和InnoDB，我们可以根据自己的业务场景来选择使用不同的存储引擎。

一条更新语句的执行流程

如果是事务性的操作指令，比如UPDATE？Mysql的整个工作流程是怎么样的呢？

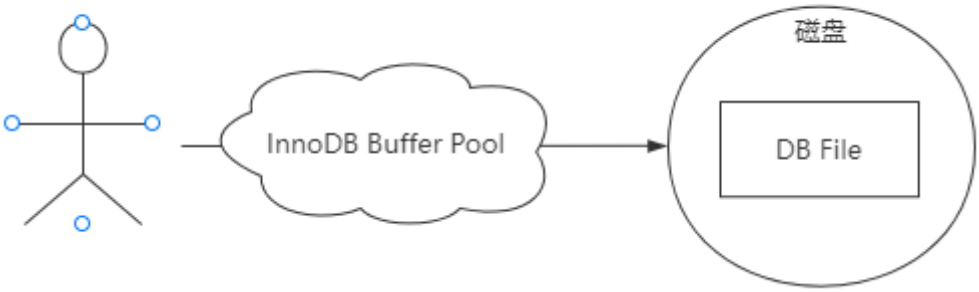
缓冲池（Buffer Pool）

在Mysql种，不管采用什么引擎（除了内存以外），数据最终都会保存到磁盘上。当我们需要对数据进行操作时，必须先要把磁盘中的数据加载到内存中，操作方式如下。



对数据进行操作的时候，每次都需要从磁盘中读取数据到内存，在内存中计算完成后再写回到磁盘，这个过程的效率比较低。

于是，在InnoDB中引入了缓存的设计理念（Buffer Pool）。



1. 读取数据的时候，先判断数据是否存在内存的缓冲区种，如果存在，则直接从缓冲区读取，否则，从磁盘读取数据，再写入到内存缓冲区。
2. 修改数据时，数据先写入到Buffer Pool，InnoDB专门有一个后台线程把Buffer Pool中的数据写入到磁盘。

那这里我们思考一下，对于数据的一次读操作，一次从磁盘加载多少数据到内存呢？假设我要读取6个字节，那么它是一次加载6个字节的数据吗？

很显然不是，磁盘I/O相对内存来说是非常慢的，特别是磁盘的随机读操作，产生的I/O次数更多。

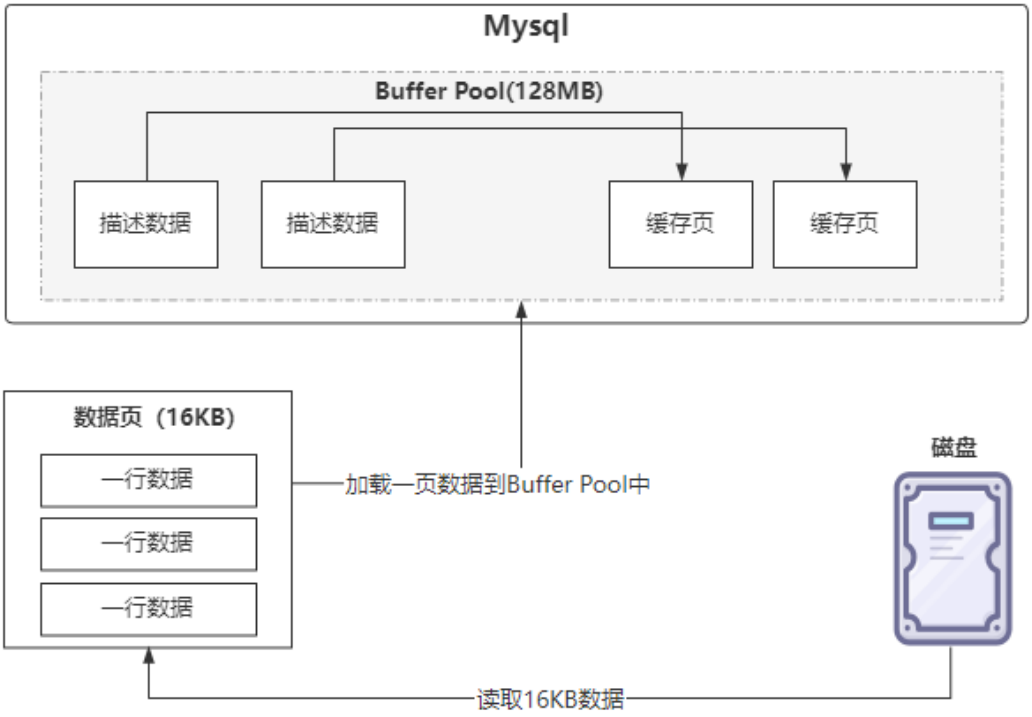
所以这里也用到了预读取的概念，也就是说，当磁盘上的某块数据被读取时，根据局部性原理，很有可能它附近位置的数据马上也会被用到，于是，一次性多读取一些数据保存到Buffer Pool中，通过空间换时间的设计思想，提升数据的IO效率。

缓存池的预读机制

InnoDB设定了一个存储引擎从磁盘读取数据到内存的最小单位叫页（操作系统也有页的概念，它的大小一般是4k）。在InnoDB种，一页数据默认的最小单位是16KB，也就是说一次数据读取操作，会从磁盘上加载16KB的数据保存到Buffer Pool中。

每个缓存页会对应一个描述数据，这个描述数据本身也是一块数据，它包含数据页所属的表空间、数据页编号、数据页在Buffer Pool种的地址等信息。

在Buffer Pool中，每个缓存页的描述数据放在最前面，然后各个缓存页放在后面。所以此时我们看下面的图，Buffer Pool实际看起来大概长这个样子 。

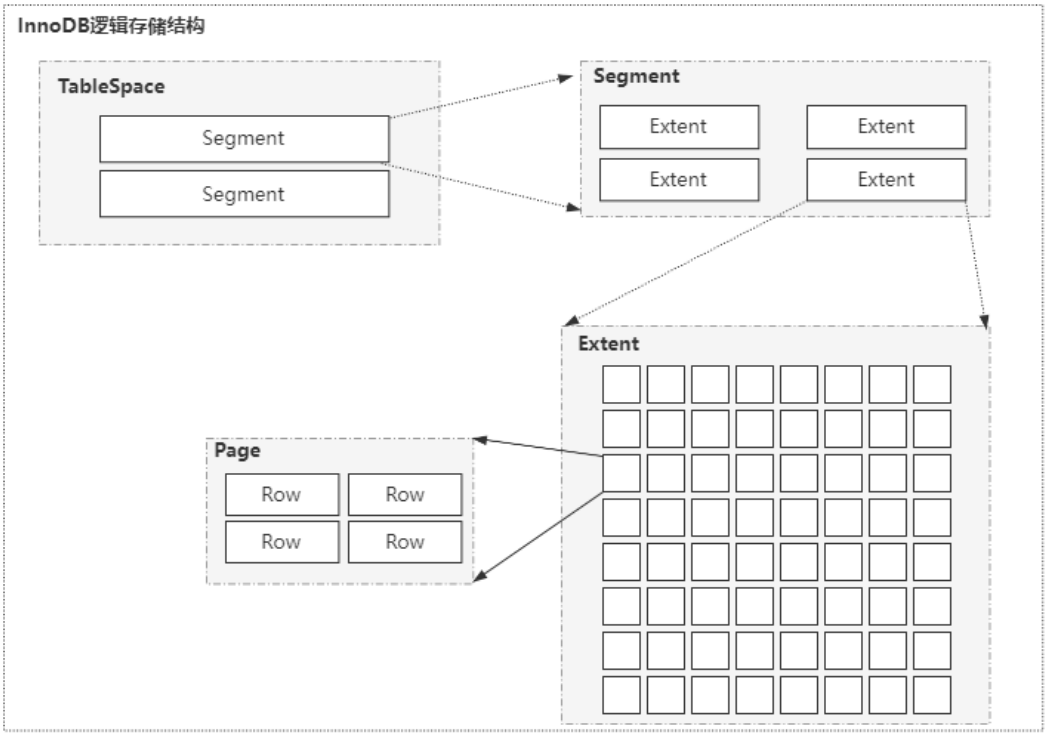


InnoDB使用两种预读算法来提高I/O性能： 线性预读（linear read-ahead）和随机预读（randomread-ahead）

线性预读以extent（64个相邻的Page称为一个extent[区]）为单位，随机预读以extent种的Page为单位。

为了更好的偶理解extent和page，我们可以看下面这个InnoDB逻辑存储机构的图，InnoDB 逻辑存储结构，分为表空间（Tablespace）、段(Segment)、区 (Extent)、页 Page) 以及行 (row)。

区是表空间的单元结构，每个区的大小为 1MB。**而页是组成区的最小单元，**页也是 InnoDB 存储引擎磁盘管理的最小单元，每个页的大小默认为 16KB。



线性预读

线性预读的作用是将下一个extent提前读取到buffer pool中

随机预读

随机预读的作用是将当前extent中的剩余的page提前读取到buffer pool中。

缓存池的空间管理（LRU）

缓存池的空间默认是128M，当然我们可以根据服务器的配置来调整缓存池的大小，官方建议是，实际情况中可以配置机器内存的50%~75%左右。

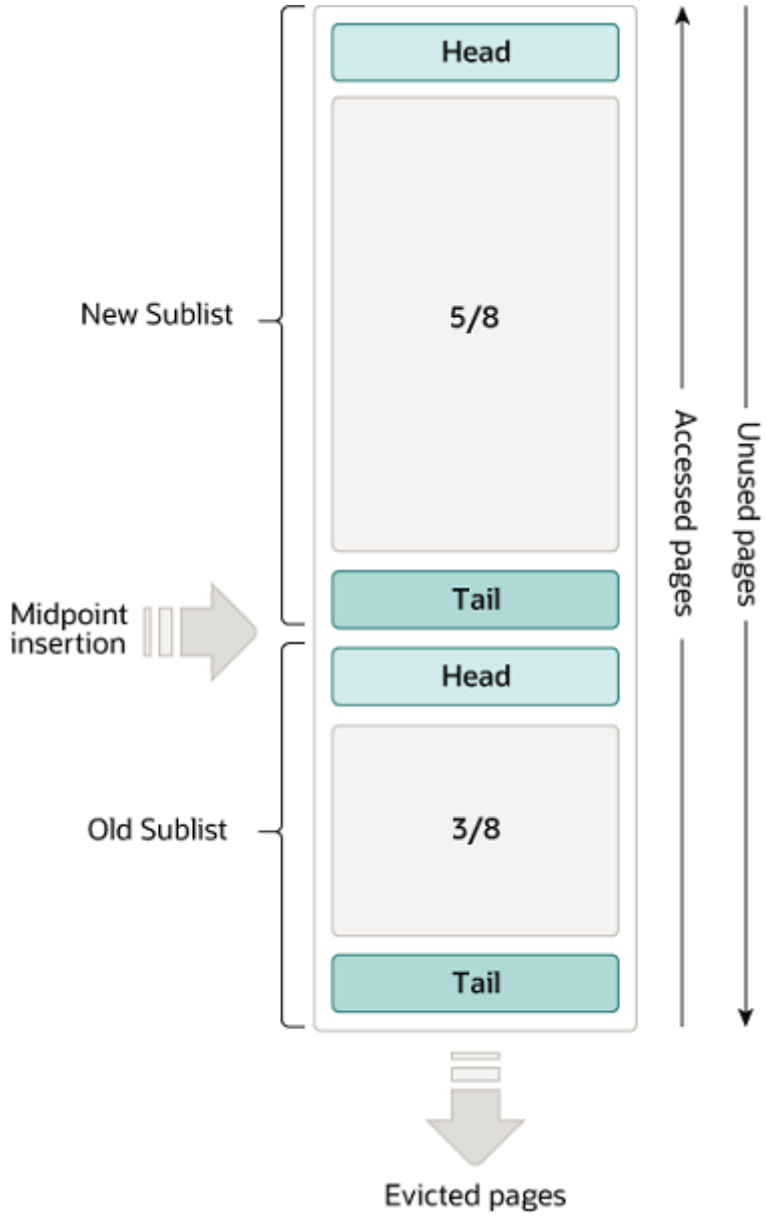
> https://dev.mysql.com/doc/refman/8.0/en/optimizing-innodb-diskio.html

即便是这样，缓存池空间大小总是有限制的，如果缓存页中加载了非常多的数据导致缓存池耗尽了怎么处理呢?

我们来看下面这个图，表示缓冲池种的LRU链表，LRU链表会被拆分成为两部分，

1. 一部分为热数据，叫New Sublist
2. 另一部分为冷数据，叫Old Sublist。

中间的分割线叫做midpoint，通过这样一个分割线来实现对缓冲池的冷热数据分离。



所有新的数据页加入到Buffer Pool时，一律先放到冷数据区的head位置，不管是预读数据还是普通的读操作。

另外，从上图右侧可以看到两个箭头，如果是Old Sublist的数据被访问，就会移动到New Sublist中，没有被访问的数据页会移动到Old Sublist，而在实现数据淘汰时，会直接从Old Sublist中进行淘汰（淘汰tail部分的数据）

如上图所示，标注了冷数据区和热数据区的大小，默认情况下，热数据区占5/8，冷数据区占3/8，这个值是由下面这个属性控制的，它表示old区的空间大小，默认是37%。

```
innodb_old_blocks_pct
```

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

这个值可以修改，修改区间为5%~95%，值越小，就使得Old区没有被访问的数据的淘汰速度更快。

> 一般生产的机器，内存比较大。我们会把innodb_old_blocks_pct值调低，防止热数据被刷出内存。

LRU的实现到这里还没完，因为还有一个很重要的问题，假设我们执行一条这样的语句

```
select * from USERINFO;
```

由于没有使用索引，所以会进行全表扫描，这种查询属于短时间内访问一次，但是后面基本上都不会用到了。

假设如果被访问了一次导致它从冷数据区移动到热数据区，使得热数据区的热点数据被移动到冷数据区从而被淘汰，这种情况下，导致BufferPool中全是低频的数据页，使得缓冲命中率大大降低，那这种情况改怎么处理呢？

于是InnoDB指定了一个冷数据区移动到热数据区的规则：**如果这个数据页在LRU链表中冷数据区存在的时间超过了1秒，就把它移动到热数据区**

这个存在时间由innodb_old_blocks_time控制，默认值是1秒。

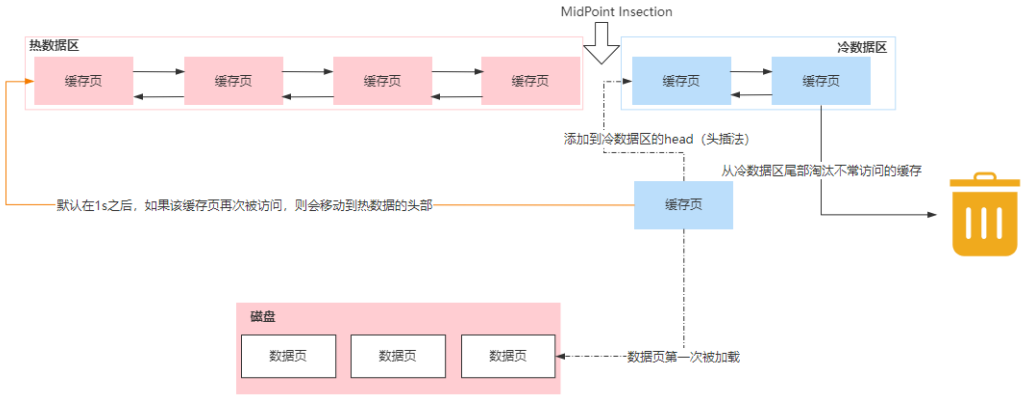
```
SHOW VARIABLES LIKE 'innodb_old_blocks_time';
```

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

> 为什么是1秒钟呢？

因为通过预读机制和全表扫描加载进来的数据页通常是1秒内就加载了很多然后对他们访问一下，这些都是1秒内完成，他们会存放在冷数据区域等待刷盘清空，基本上不太会有机会放入到热数据区域，除非在1秒后还有人访问，说明后续可能还会有人访问，才会放入热数据区域的头部。

到目前为止，缓存池的LRU算法整体原理如下：



Buffer Pool中的脏数据何时刷盘？

有了Buffer Pool之后，当我们进行数据修改时，会先修改Buffer Pool中的数据。那这里大家会有一个疑问，Buffer Pool中的修改的数据什么时候刷新到磁盘呢？有下面四种情况。（**脏数据就是在内存已被修改，但是仍未写入磁盘的数据。**）

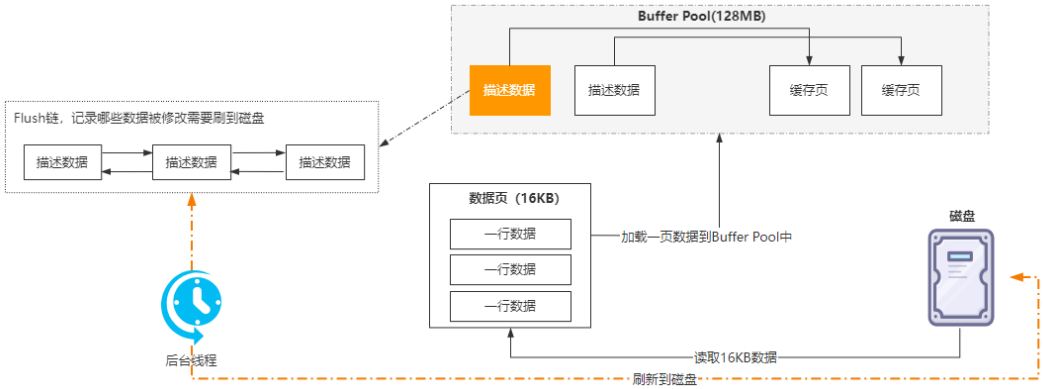
1. 后台线程定时刷新
2. Buffer Pool内存不足
3. Redo Log写满
4. 数据库正常关闭

但是这里又有一个问题，就是最终要把哪些数据刷新到磁盘呢？

不可能所有的缓存页都刷回磁盘的，因为有的缓存页可能是因为查询的时候因为预读取机制加载到buffer pool中的，可能根本没修改过，如果也同步一次，那显然也不合理。

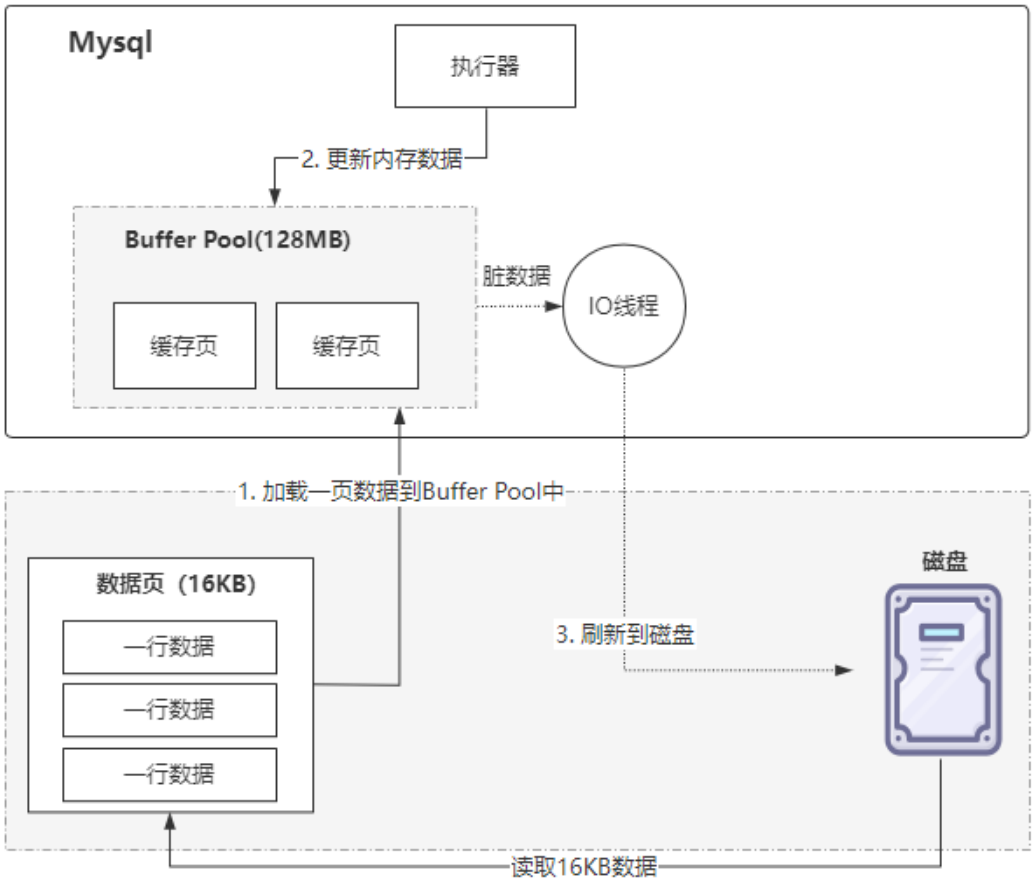
因此，我们需要做的是：把修改过的数据刷新到磁盘。

所以，在这里引入了 一个Flush链表，这个Flush链表本质上也是通过缓存页的描述数据块中的两个指针，让被修改过的缓存页的描述数据块，组成一个双向链表，如下图所示。



Redo/Undo Log

有了缓冲池之后，数据的执行过程如下。



在这个过程中，我们发现一个问题，就是我们对数据库中的内存完成了一系列的增删改操作，虽然内存数据更新了，单是内存到磁盘的刷新是异步的，如果在异步刷新之前，数据库挂了，由于内存数据没有持久化导致这些数据丢失，这个问题怎么解决呢？

为了避免这种问题的出现，InnoDB对所有数据页的修改操作，都记录到了一个日志文件中（这个日志文件叫Redo Log）。

当数据库崩溃时，一旦存在未同步到磁盘的数据，在数据库启动时，就会从redo log中读取之前做过的数据库操作，然后把这些操作重新在内存中执行一遍，从而实现数据的恢复，这就是事务ACID特性中D(持久性)的保障机制。

Log Buffer

需要注意，在上图中我们发现写Redo Log时，是先写到Redo Log Buffer种，然后再刷新到Redo Log文件?

那Redo Log Buffer是什么呢？ Redo Log Buffer是Redo Log 的缓冲池，它里面保存了要写入到磁盘文件的内容。

为什么不直接写磁盘，而是又要增加一个缓冲区呢？下面是官网的描述。

> https://dev.mysql.com/doc/refman/8.0/en/innodb-redo-log-buffer.html

对于大的Log Buffer空间 可以容纳大的事务无需将数据写入到Redo 日志中，从而减少与磁盘的交互。所以，如果事务中如果有大量的DML操作，可以考虑增大Log Buffer的值，减少磁盘IO从而提升效率。

注意，日志缓冲区也是有大小的，默认大小是16MB，可以通过innodb_log_buffer_size变量来修改。

Log Buffer刷盘机制

Mysql给使用者一些参数，使用者根据场景来设置不同参数来解决不同场景的问题。

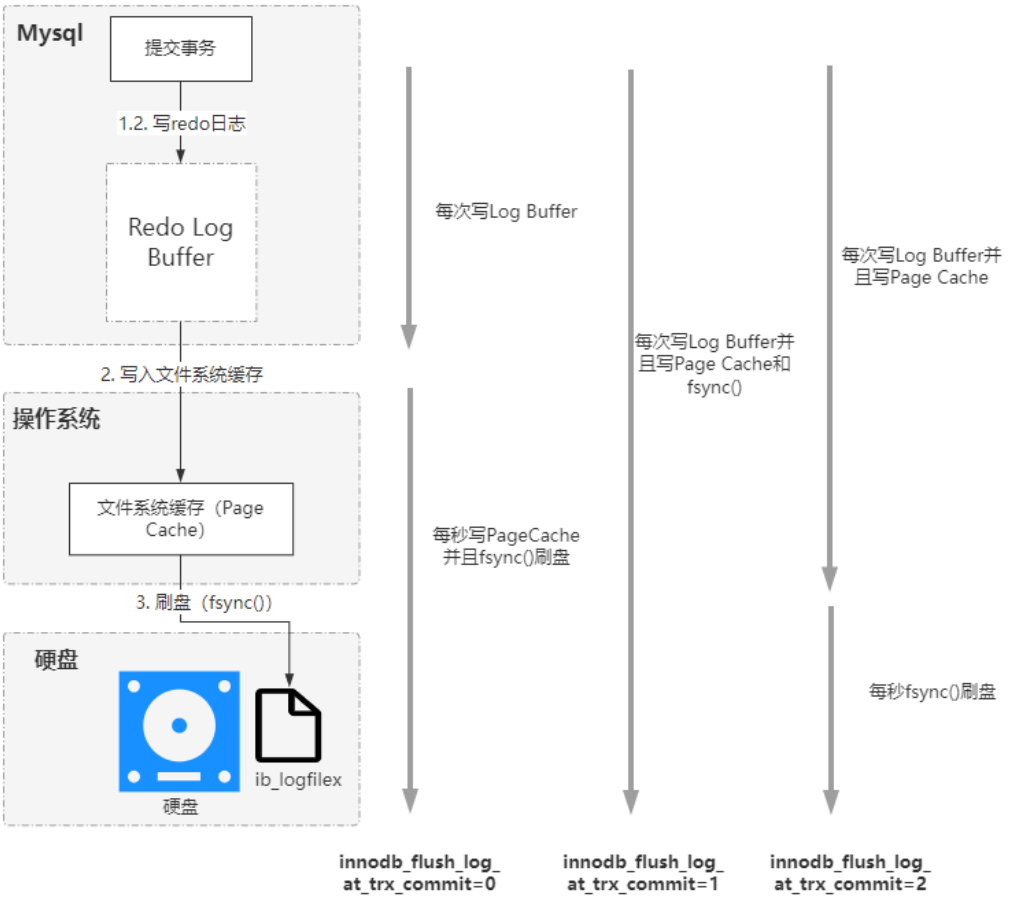
```
innodb_flush_log_at_trx_commit
```

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

这个变量是用来空值Log Buffer缓冲区种的内容如何写入和刷新到磁盘的，它有三个值分别是：

- 0， log buffer将每秒一次地写入log file中，并且log file的flush操作同时进行。该模式下，在事务提交的时候，不会主动触发写入磁盘的操作。**这个策略的性能是最佳的，但是会存在1s的数据丢失.**
- 1，每次事务提交时MySQL都会把log buffer的数据写入log file，并且刷到磁盘中去。**这个策略能保证强一致性，也是InnoDB默认的配置，为的是保证事务的ACID特性**
- 2，每次事务提交时MySQL都会把log buffer的数据写入log file。但是flush操作并不会同时进行。该模式下，MySQL会每秒执行一次 flush操作。**这种策略，如果操作系统出现崩溃，也可能会存在1s的数据丢失**

为了更加清晰的说明这三种策略，我们可以看下面这个图。



随堂问题

---

redolog恢复的时候怎么知道哪些是应该提交的，哪些是不应该提交的

---

redolog恢复的时候怎么知道哪些是应该提交的，哪些是不应该提交的
如果将修改放到change buffer，然后还没来得及写入磁盘，这个时候又要读数据，岂不是直接从磁盘读到脏数据了
能说说唯一索引非唯一索引数据在bufferpool 存在和不错在情况下更新的流程么，还有双写缓存中dwr 存的是刷新前系统页的数据备份么
innodb页断裂是因为写完4k后将原来页的数据即时删除造成的吗?
双写为啥是两个文件呢？正常情况下，两个文件内容完全一致么?
dblwr为啥是两个文件
事务正常提交后，磁盘正常刷盘了，RedoLog要清理么?
Insert操作和bufferpool的关系
如果一条数据在很短的时间内做了多次更新，page cache里面的操作记录会合并么
binlog 文件在哪怎么读取，一些同步是不是读取的这个文件?
如果一条数据在很短的时间内做了多次更新，pagecache
redolog和binlog两阶段提交帮忙复习一下
要更新的缓存页中的数据，是否有可能已经被lru算法淘汰?
redo为什么要分成2个
change buffer在更新时的作用是啥
如果在写redo log时，数据库挂了，那这部分数据怎么恢复呢?
预读机制在什么时候发生
一条SQL语句在用ABC三个字段查询的时候，C查询条件一样，A B字段查询的数据量多的时候C字段如果用exists查询很慢，用in查询很快，AB字段查询数据量不多的时候用exists很快，用in很慢，这种SQL怎么优化，或者有什么好方法解决呢

redolog恢复的时候怎么知道哪些是应该提交的，哪些是不应该提交的
tablespace里面的segment数据是指buffer pool里面，还是单独的一块内存区域
Extent是在buffer pool里面吗？没听明白
顺序预读和随机预读都是取 下个区的数据
mysql 跟oracle对比 有啥优势
Buffer Pool运行时实际物理位置一定是内存么？还是由DBMS或者OS管理？
间隙锁、临键锁和记录锁的原理讲下？在可重复读隔离级别下怎么解决幻读的？