课程目标

1、了解 MySQL 语句的执行流程

2、理解 MySQL 的架构与内部模块

3、掌握 InnoDB 存储引擎的磁盘与内存结构

内容定位

适合有 MySQL 使用经验，对数据库基本概念和 SQL 语法已经有所了解的同学。

MySQL 的发展历史和版本分支：

时间	里程碑
1996 年	MySQL 1.0 发布。它的历史可以追溯到 1979 年，作者 Monty 用 BASIC 设计的一个报表工具。
1996 年 10 月	3.11.1 发布。MySQL 没有 2.x 版本。
2000 年	ISAM 升级成 MyISAM 引擎。MySQL 开源。
2003 年	MySQL 4.0 发布，集成 InnoDB 存储引擎。
2005 年	MySQL 5.0 版本发布，提供了视图、存储过程等功能。
2008 年	MySQL AB 公司被 Sun 公司收购，进入 Sun MySQL 时代。
2009 年	Oracle 收购 Sun 公司，进入 Oracle MySQL 时代。
2010 年	MySQL 5.5 发布，InnoDB 成为默认的存储引擎。
2016 年	MySQL 发布 8.0.0 版本。为什么没有 6、7？5.6 可以当成 6.x，5.7 可以当成 7.x。

因为 MySQL 是开源的（也有收费版本），所以在 MySQL 稳定版本的基础上也发展出来了很多的分支，就像 Linux 一样，有 Ubuntu、RedHat、CentOS、Fedora [fɪ'dɔrə]、Debian[*Deb' -ee-en*]等等。

大家最熟悉的应该是 MariaDB，因为 CentOS 7 里面自带了一个 MariaDB。它是怎

么来的呢？Oracle 收购 MySQL 之后，MySQL 创始人之一 Monty 担心 MySQL 数据库发展的未来（开发缓慢，封闭，可能会被闭源），就创建了一个分支 MariaDB，默认使用全新的 Maria 存储引擎，它是原 MyISAM 存储引擎的升级版本。

其他流行分支：

Percona Server 是 MySQL 重要的分支之一，它基于 InnoDB 存储引擎的基础上，提升了性能和易管理性，最后形成了增强版的 XtraDB 引擎，可以用来更好地发挥服务器硬件上的性能。

国内也有一些 MySQL 的分支或者自研的存储引擎，比如网易的 InnoSQL，极数云舟的 ArkDB。

MySQL 应该怎么读？ 官网对于这个问题有解释。
https://dev.mysql.com/doc/refman/5.7/en/what-is-mysql.html

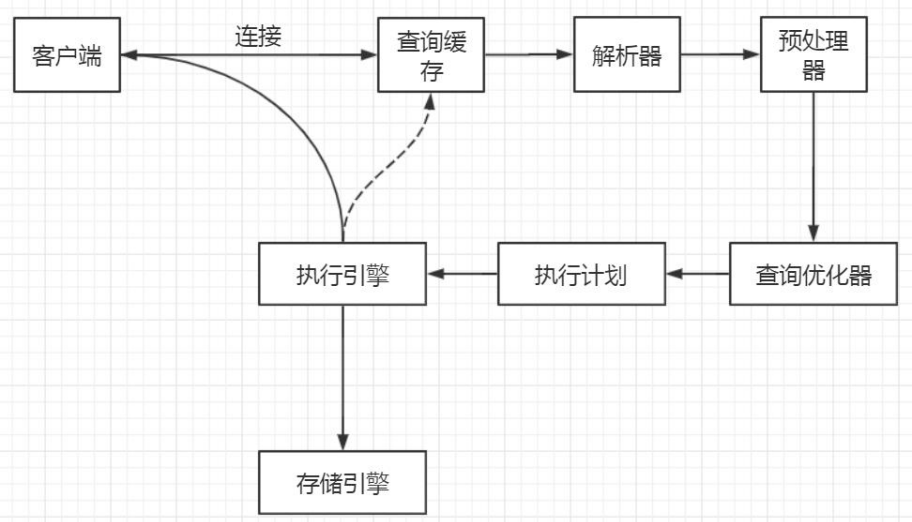
我们操作数据库有各种各样的方式，比如 Linux 系统中的命令行，比如数据库工具 Navicat，比如程序，例如 Java 语言的 JDBC API 或者 ORM 框架。

大家有没有思考过，当我们的工具或者程序连接到数据库之后，实际上发生了什么事情？它的内部是怎么工作的？

就像我们到餐厅去吃饭，点了菜以后，过一会儿菜端上来了，后厨里面有哪些人？他们分别做了什么事情？这个就涉及到 MySQL 的整体架构和工作流程了。

以一条查询语句为例，我们来看下 MySQL 的工作流程是什么样的。

1. 一条查询 SQL 语句是如何执行的？



我们的程序或者工具要操作数据库，第一步要做什么事情?

跟数据库建立连接。

1.1. 通信协议

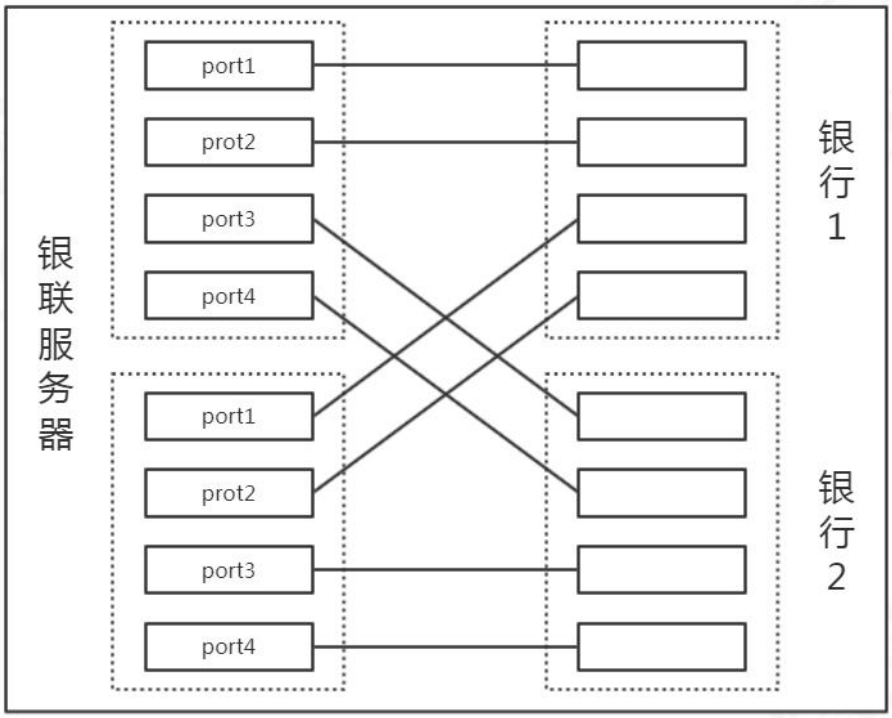
首先，MySQL 必须要运行一个服务，监听默认的 3306 端口。

在我们开发系统跟第三方对接的时候，必须要弄清楚的有两件事。

第一个就是通信协议，比如我们是用 HTTP 还是 WebService 还是 TCP?

第二个是消息格式，比如我们用 XML 格式，还是 JSON 格式，还是定长格式？报文头长度多少，包含什么内容，每个字段的详细含义。

比如我们之前跟银联对接，银联的银行卡联网规范，约定了一种比较复杂的通讯协议叫做：四进四出单工异步长连接（为了保证稳定性和性能）。



1.1.1. 通信协议

MySQL 是支持多种通信协议的，可以使用同步/异步的方式，支持长连接/短连接。

这里我们拆分来看。第一个是通信类型。

通信类型：同步或者异步

同步通信的特点：

1、同步通信依赖于被调用方，受限于被调用方的性能。也就是说，应用操作数据库，线程会阻塞，等待数据库的返回。

2、一般只能做到一对一，很难做到一对多的通信。

异步跟同步相反：

1、异步可以避免应用阻塞等待，但是不能节省 SQL 执行的时间。

2、如果异步存在并发，每一个 SQL 的执行都要单独建立一个连接，避免数据混乱。但是这样会给服务端带来巨大的压力（一个连接就会创建一个线程，线程间切换会占用大量 CPU 资源）。另外异步通信还带来了编码的复杂度，所以一般不建议使用。如果要异步，必须使用连接池，排队从连接池获取连接而不是创建新连接。

一般来说我们连接数据库都是同步连接。

连接方式：长连接或者短连接

MySQL 既支持短连接，也支持长连接。短连接就是操作完毕以后，马上 close 掉。长连接可以保持打开，减少服务端创建和释放连接的消耗，后面的程序访问的时候还可以使用这个连接。一般我们会在连接池中使用长连接。

保持长连接会消耗内存。长时间不活动的连接，MySQL 服务器会断开。

```
show global variables like 'wait_timeout';  -- 非交互式超时时间，如 JDBC 程序
show global variables like 'interactive_timeout';  -- 交互式超时时间，如数据库工具
```

默认都是 28800 秒，8 小时。

我们怎么查看 MySQL 当前有多少个连接?

可以用 show status 命令：

```
show global status like 'Thread%';
```

Threads_cached：缓存中的线程连接数。

Threads_connected：当前打开的连接数。

Threads_created：为处理连接创建的线程数。

Threads_running：非睡眠状态的连接数，通常指并发连接数。

每产生一个连接或者一个会话，在服务端就会创建一个线程来处理。反过来，如果要杀死会话，就是 Kill 线程。

有了连接数，怎么知道当前连接的状态?

也可以使用 SHOW PROCESSLIST; （root 用户）查看 SQL 的执行状态。

https://dev.mysql.com/doc/refman/5.7/en/show-processlist.html

Id	User	Host	db	Command	Time	State	Info
55	root	192.168.	gupao	Query	0	starting	SHOW
56	root	192.168.	gupao	Sleep	1892		(Null)
58	root	192.168.	gupao	Sleep	1861		(Null)
64	root	192.168.	gupao	Sleep	1861		(Null)
67	root	192.168.	gupao	Sleep	1855		(Null)
69	root	192.168.	gupao	Sleep	1851		(Null)
71	root	192.168.	gupao	Sleep	1849		(Null)

一些常见的状态：

https://dev.mysql.com/doc/refman/5.7/en/thread-commands.html

状态	含义
Sleep	线程正在等待客户端，以向它发送一个新语句
Query	线程正在执行查询或往客户端发送数据
Locked	该查询被其它查询锁定
Copying to tmp table on disk	临时结果集合大于 tmp_table_size。线程把临时表从存储器内部格式改变为磁盘模式，以节约存储器
Sending data	线程正在为 SELECT 语句处理行，同时正在向客户端发送数据
Sorting for group	线程正在进行分类，以满足 GROUP BY 要求
Sorting for order	线程正在进行分类，以满足 ORDER BY 要求

MySQL 服务允许的最大连接数是多少呢?

在 5.7 版本中默认是 151 个，最大可以设置成 16384（2^14）。

```
show variables like 'max_connections';
```

Variable_name	Value
max_connections	151

show 的参数说明：

1、级别：会话 session 级别（默认）；全局 global 级别

2、动态修改：set，重启后失效；永久生效，修改配置文件/etc/my.cnf

```
set global max_connections = 1000;
```

通信协议

MySQL 支持哪些通信协议呢?

第一种是 Unix Socket。

比如我们在 Linux 服务器上，如果没有指定-h 参数，它就用 socket 方式登录（省略了-S /var/lib/mysql/mysql.sock）。

```
[root@localhost tmp]# mysql
ERROR 1045 (28000): Access denied for user 'root'
```

它不用通过网络协议，也可以连接到 MySQL 的服务器，它需要用到服务器上的一个物理文件（/var/lib/mysql/mysql.sock）。

```
select @@socket;
```

如果指定-h 参数，就会用第二种方式，TCP/IP 协议。

```
mysql -h192.168.8.211 -uroot -p123456
```

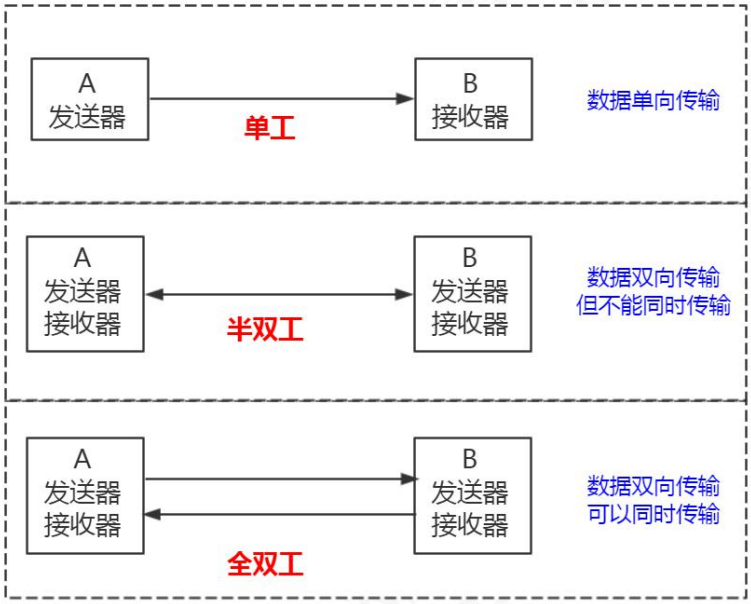
我们的编程语言的连接模块都是用 TCP 协议连接到 MySQL 服务器的，比如 mysql-connector-java-x.x.xx.jar。

```
[root@localhost ~]# netstat -an|grep 3306
tcp6       0      0 :::3306                 :::*                    LISTEN
tcp6       0      0 192.168.8.211:3306      192.168.8.1:10629       ESTABLISHED
tcp6       0      0 192.168.8.211:3306      192.168.8.1:10648       ESTABLISHED
```

另外还有命名管道（Named Pipes）和内存共享（Share Memory）的方式，这两种通信方式只能在 Windows 上面使用，一般用得比较少。

1.1.2. 通信方式

第二个是通信方式。



单工：

在两台计算机通信的时候，数据的传输是单向的。生活中的类比：遥控器。

半双工：

在两台计算机之间，数据传输是双向的，你可以给我发送，我也可以给你发送，但是在这个通讯连接里面，同一时间只能有一台服务器在发送数据，也就是你要给我发的话，也必须等我发给你完了之后才能给我发。生活中的类比：对讲机。

全双工：

数据的传输是双向的，并且可以同时传输。生活中的类比：打电话。

MySQL 使用了半双工的通信方式?

要么是客户端向服务端发送数据，要么是服务端向客户端发送数据，这两个动作不能同时发生。所以客户端发送 SQL 语句给服务端的时候，（在一次连接里面）数据是不能分成小块发送的，不管你的 SQL 语句有多大，都是一次性发送。

比如我们用 MyBatis 动态 SQL 生成了一个批量插入的语句，插入 10 万条数据，values 后面跟了一长串的内容，或者 where 条件 in 里面的值太多，会出现问题。

这个时候我们必须要调整 MySQL 服务器配置 max_allowed_packet 参数的值（默认是 4M），把它调大，否则就会报错。

Variable_name	Value
max_allowed_packet	4194304

另一方面，对于服务端来说，也是一次性发送所有的数据，不能因为你已经取到了想要的数据就中断操作，这个时候会对网络和内存产生大量消耗。

所以，我们一定要在程序里面避免不带 limit 的这种操作，比如一次把所有满足条件的数据全部查出来，一定要先 count 一下。如果数据量的话，可以分批查询。

执行一条查询语句，客户端跟服务端建立连接之后呢？下一步要做什么?

1.2. 查询缓存

MySQL 内部自带了一个缓存模块。

缓存的作用我们应该很清楚了，把数据以 KV 的形式放到内存里面，可以加快数据的

读取速度，也可以减少服务器处理的时间。但是 MySQL 的缓存我们好像比较陌生，从来没有去配置过，也不知道它什么时候生效？

比如 user_innodb 有 500 万行数据，没有索引。我们在没有索引的字段上执行同样的查询，大家觉得第二次会快吗？

```
select * from user_innodb where name='青山';
```

缓存没有生效，为什么？MySQL 的缓存默认是关闭的。

```
show variables like 'query_cache%';
```

默认关闭的意思就是不推荐使用，为什么 MySQL 不推荐使用它自带的缓存呢？

主要是因为 MySQL 自带的缓存的应用场景有限，第一个是它要求 SQL 语句必须一模一样，中间多一个空格，字母大小写不同都被认为是不同的的 SQL。

第二个是表里面任何一条数据发生变化的时候，这张表所有缓存都会失效，所以对于有大量数据更新的应用，也不适合。

所以缓存这一块，我们还是交给 ORM 框架（比如 MyBatis 默认开启了一级缓存），或者独立的缓存服务，比如 Redis 来处理更合适。

在 MySQL 8.0 中，查询缓存已经被移除了。

1.3. 语法解析和预处理(Parser & Preprocessor)

我们没有使用缓存的话，就会跳过缓存的模块，下一步我们要做什么呢？

OK，这里我会有一个疑问，为什么我的一条 SQL 语句能够被识别呢？假如我随便执行一个字符串 penyuyan，服务器报了一个 1064 的错：

```
[Err] 1064 - You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'penyuyan' at line 1
```

它是怎么知道我输入的内容是错误的？

这个就是 MySQL 的 Parser 解析器和 Preprocessor 预处理模块。

这一步主要做的事情是对语句基于 SQL 语法进行词法和语法分析和语义的解析。

1.3.1. 词法解析

词法分析就是把一个完整的 SQL 语句打碎成一个个的单词。

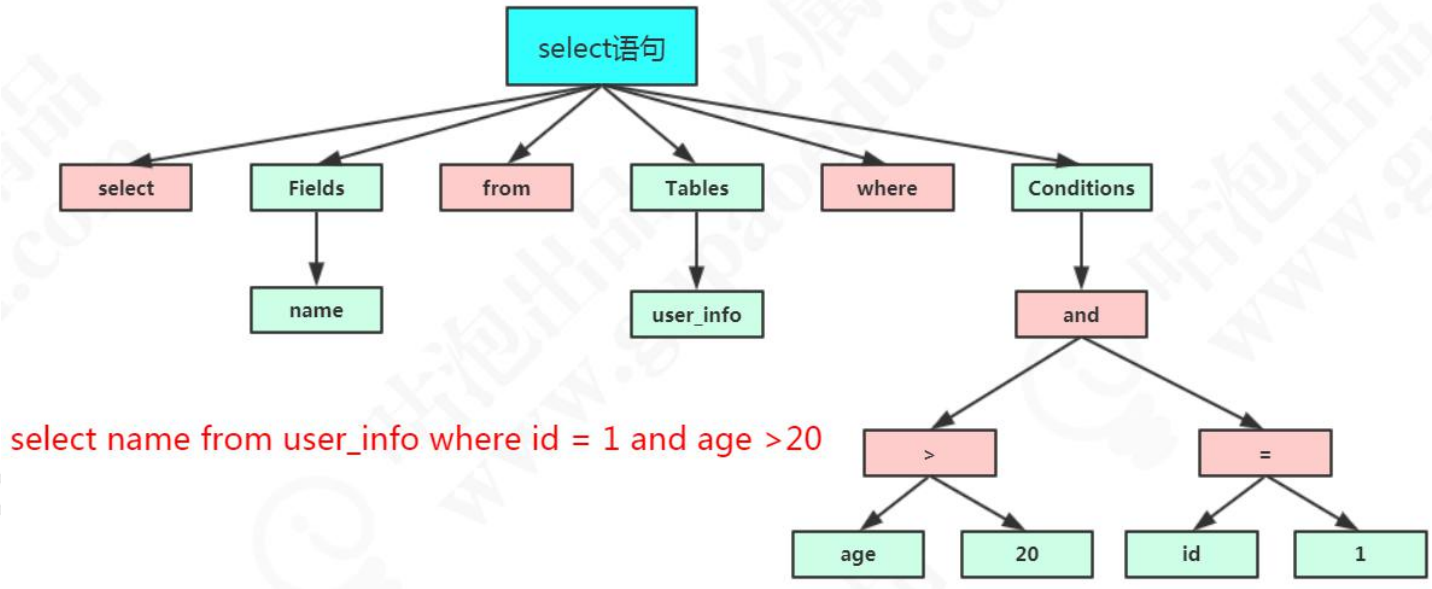
比如一个简单的 SQL 语句：

```
select name from user where id = 1;
```

它会打碎成 8 个符号，每个符号是什么类型，从哪里开始到哪里结束。

1.3.2. 语法解析

第二步就是语法分析，语法分析会对 SQL 做一些语法检查，比如单引号有没有闭合，然后根据 MySQL 定义的语法规则，根据 SQL 语句生成一个数据结构。这个数据结构我们把它叫做解析树（select_lex）。



任何数据库的中间件，比如 Mycat，Sharding-JDBC（用到了 Druid Parser），都必须要有词法和语法分析功能，在市面上也有很多的开源的词法解析的工具（比如 LEX，

Yacc）。

1.3.3. 预处理器

问题：如果我写了一个词法和语法都正确的 SQL，但是表名或者字段不存在，会在哪里报错？是在数据库的执行层还是解析器？比如：

```
select * from penyuyan;
```

解析器可以分析语法，但是它怎么知道数据库里面有什么表，表里面有什么字段呢？

实际上还是在解析的时候报错，解析 SQL 的环节里面有个预处理器。

它会检查生成的解析树，解决解析器无法解析的语义。比如，它会检查表和列名是否存在，检查名字和别名，保证没有歧义。

预处理之后得到一个新的解析树。

1.4. 查询优化（Query Optimizer）与查询执行计划

1.4.1. 什么是优化器？

得到解析树之后，是不是执行 SQL 语句了呢？

这里我们有一个问题，一条 SQL 语句是不是只有一种执行方式？或者说数据库最终执行的 SQL 是不是就是我们发送的 SQL？

这个答案是否定的。一条 SQL 语句是可以有很多种执行方式的，最终返回相同的结果，他们是等价的。但是如果有这么多种执行方式，这些执行方式怎么得到的？最终选择哪一种去执行？根据什么判断标准去选择？

这个就是 MySQL 的查询优化器的模块（Optimizer）。

查询优化器的目的就是根据解析树生成不同的执行计划（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，那种执行计划开销最小，就用哪种。

可以使用这个命令查看查询的开销：

```
show status like 'Last_query_cost';
```

https://dev.mysql.com/doc/refman/5.7/en/server-status-variables.html#statvar_Last_query_cost

1.4.2. 优化器可以做什么？

MySQL 的优化器能处理哪些优化类型呢?

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。

如果对于优化器的细节感兴趣,可以看看《数据库查询优化器的艺术-原理解析与SQL性能优化》。

但是优化器也不是万能的，并不是再垃圾的 SQL 语句都能自动优化，也不是每次都能选择到最优的执行计划，大家在编写 SQL 语句的时候还是要注意。

如果我们想知道优化器是怎么工作的，它生成了几种执行计划，每种执行计划的 cost 是多少，应该怎么做？

1.4.3. 优化器是怎么得到执行计划的？

https://dev.mysql.com/doc/internals/en/optimizer-tracing.html

首先我们要启用优化器的追踪（默认是关闭的）：

```
SHOW VARIABLES LIKE 'optimizer_trace';
set optimizer_trace='enabled=on';
```

注意开启这开关是会消耗性能的，因为它要把优化分析的结果写到表里面，所以不要轻易开启，或者查看完之后关闭它（改成 off）。

注意：参数分为 session 和 global 级别。

接着我们执行一个 SQL 语句，优化器会生成执行计划：

```
select t.tcid from teacher t,teacher_contact tc where t.tcid = tc.tcid;
```

这个时候优化器分析的过程已经记录到系统表里面了，我们可以查询：

```
select * from information_schema.optimizer_trace\G
```

它是一个 JSON 类型的数据，主要分成三部分，准备阶段、优化阶段和执行阶段。

expanded_query 是优化后的 SQL 语句。

considered_execution_plans 里面列出了所有的执行计划。

分析完记得关掉它：

```
set optimizer_trace="enabled=off";
SHOW VARIABLES LIKE 'optimizer_trace';
```

1.4.4. 优化器得到的结果

优化完之后，得到一个什么东西呢？

优化器最终会把解析树变成一个查询执行计划，查询执行计划是一个数据结构。

当然，这个执行计划是不是一定是最优的执行计划呢？不一定，因为 MySQL 也有可能覆盖不到所有的执行计划。

我们怎么查看 MySQL 的执行计划呢？比如多张表关联查询，先查询哪张表？在执行查询的时候可能用到哪些索引，实际上用到了什么索引？

MySQL 提供了一个执行计划的工具。我们在 SQL 语句前面加上 EXPLAIN，就可以看到执行计划的信息。

```
EXPLAIN select name from user where id=1;
```

*注意 Explain 的结果也不一定最终执行的方式。

1.5. 存储引擎

得到执行计划以后，SQL 语句是不是终于可以执行了？

问题又来了：

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？

1.5.1. 存储引擎基本介绍

我们先回答第一个问题：在关系型数据库里面，数据是放在什么结构里面的？

（放在表 Table 里面的）

我们可以把这个表理解成 Excel 电子表格的形式。所以我们的表在存储数据的同时，还要组织数据的存储结构，这个存储结构就是由我们的存储引擎决定的，所以我们也可以把存储引擎叫做表类型。

在 MySQL 里面，支持多种存储引擎，他们是可以替换的，所以叫做插件式的存储引擎。为什么要搞这么多存储引擎呢？一种还不够用吗？

这个问题先留着。

1.5.2. 查看存储引擎

比如我们数据库里面已经存在的表，我们怎么查看它们的存储引擎呢？

```
show table status from `gupao`;
```

Name	Engine	Version	Row_format	Rows	Avg_row_length	Data_length
course	InnoDB	10	Dynamic	4	4096	16384
employees	InnoDB	10	Dynamic	10	1638	16384
student	InnoDB	10	Dynamic	2	8192	16384
teacher	InnoDB	10	Dynamic	3	5461	16384
teacher_contact	InnoDB	10	Dynamic	3	5461	16384
user_archive	ARCHIVE	10	Compressed	0	1073	8740
user_csv	CSV	10	Dynamic	2	0	0
user_innodb	InnoDB	10	Dynamic	996770	48	48840704
user_memory	MEMORY	10	Fixed	0	1073	0
user_myisam	MyISAM	10	Dynamic	0	0	0

或者通过 DDL 建表语句来查看。

在 MySQL 里面，我们创建的每一张表都可以指定它的存储引擎，而不是一个数据库只能使用一个存储引擎。存储引擎的使用是以表为单位的。而且，创建表之后还可以修改存储引擎。

我们说一张表使用的存储引擎决定我们存储数据的结构，那在服务器上它们是怎么存储的呢？我们先要找到数据库存放数据的路径：

```
show variables like 'datadir';
```

默认情况下，每个数据库有一个自己文件夹，以 gupao 数据库为例。

任何一个存储引擎都有一个 frm 文件，这个是表结构定义文件。

```
user_innodb.frm
user_innodb.ibd
user_memory.frm
user_myisam.frm
user_myisam.MYD
user_myisam.MYI
```

不同的存储引擎存放数据的方式不一样，产生的文件也不一样，innodb 是 1 个，memory 没有，myisam 是两个。

这些存储引擎的差别在哪呢？

1.5.3. 存储引擎比较

常见存储引擎

MyISAM 和 InnoDB 是我们用得最多的两个存储引擎，在 MySQL 5.5 版本之前，默认的存储引擎是 MyISAM，它是 MySQL 自带的。我们创建表的时候不指定存储引擎，它就会使用 MyISAM 作为存储引擎。

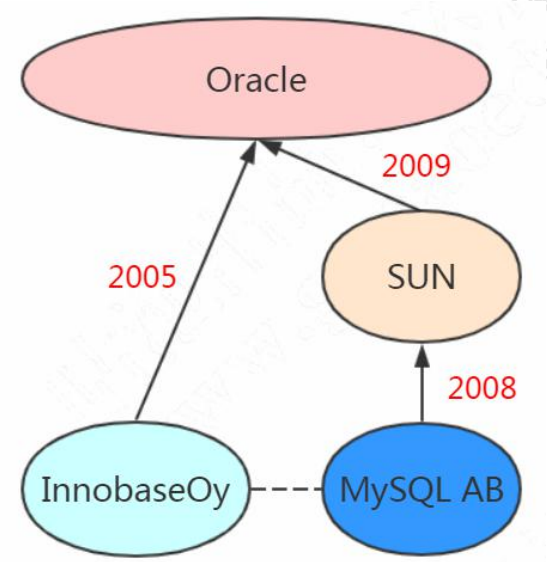
MyISAM 的前身是 ISAM（Indexed Sequential Access Method：利用索引，顺序存取数据的方法）。

5.5 版本之后默认的存储引擎改成了 InnoDB，它是第三方公司为 MySQL 开发的。为什么要改呢？最主要的原因还是 InnoDB 支持事务，支持行级别的锁，对于业务一致性要求高的场景来说更适合。

这个里面又有 Oracle 和 MySQL 公司的一段恩怨情仇。

InnoDB 本来是 InnobaseOy 公司开发的，它和 MySQL AB 公司合作开源了 InnoDB 的代码。但是没想到 MySQL 的竞争对手 Oracle 把 InnobaseOy 收购了。

后来 08 年 Sun 公司（开发 Java 语言的 Sun）收购了 MySQL AB，09 年 Sun 公司又被 Oracle 收购了，所以 MySQL，InnoDB 又是一家了。有人觉得 MySQL 越来越像 Oracle，其实也是这个原因。



那么除了这两个我们最熟悉的存储引擎，数据库还支持其他哪些常用的存储引擎呢?

数据库支持的存储引擎

我们可以用这个命令查看数据库对存储引擎的支持情况:

```
show engines ;
```

其中有存储引擎的描述和对事务、XA 协议和 Savepoints 的支持。

XA 协议用来实现分布式事务（分为本地资源管理器，事务管理器）。

Savepoints 用来实现子事务（嵌套事务）。创建了一个 Savepoints 之后，事务就可以回滚到这个点，不会影响到创建 Savepoints 之前的操作。

Engine	Support	Comment	Transactions	XA	Savepoints
InnoDB	DEFAULT	Supports transactions, row-level locking, and foreign keys	YES	YES	YES
MRG_MYISAM	YES	Collection of identical MyISAM tables	NO	NO	NO
MEMORY	YES	Hash based, stored in memory, useful for temporary tables	NO	NO	NO
BLACKHOLE	YES	/dev/null storage engine (anything you write to it disappears)	NO	NO	NO
MyISAM	YES	MyISAM storage engine	NO	NO	NO
CSV	YES	CSV storage engine	NO	NO	NO
ARCHIVE	YES	Archive storage engine	NO	NO	NO
PERFORMANCE_SCHEMA	YES	Performance Schema	NO	NO	NO
FEDERATED	NO	Federated MySQL storage engine	(Null)	(Null	(Null)

这些数据库支持的存储引擎，分别有什么特性呢?

https://dev.mysql.com/doc/refman/5.7/en/storage-engines.html

MyISAM（3 个文件）

These tables have a small footprint. Table-level locking limits the performance in read/write workloads, so it is often used in read-only or read-mostly workloads in Web and data warehousing configurations.

应用范围比较小。表级锁定限制了读/写的性能，因此在 Web 和数据仓库配置中，它通常用于只读或以读为主的工作。

特点:

支持表级别的锁（插入和更新会锁表）。不支持事务。

拥有较高的插入（insert）和查询（select）速度。

存储了表的行数（count 速度更快）。

（怎么快速向数据库插入 100 万条数据？我们有一种先用 MyISAM 插入数据，然后修改存储引擎为 InnoDB 的操作。）

适合：只读之类的数据分析的项目。

InnoDB（2 个文件）

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

The default storage engine in MySQL 5.7. InnoDB is a transaction-safe (ACID compliant) storage engine for MySQL that has commit, rollback, and crash-recovery capabilities to protect user data. InnoDB row-level locking (without escalation to coarser granularity locks) and Oracle-style consistent nonlocking reads increase multi-user concurrency and performance. InnoDB stores user data in clustered indexes to reduce I/O for common queries based on primary keys. To maintain data integrity, InnoDB also supports FOREIGN KEY referential-integrity constraints.

mysql 5.7 中的默认存储引擎。InnoDB 是一个事务安全（与 ACID 兼容）的 MySQL 存储引擎，它具有提交、回滚和崩溃恢复功能来保护用户数据。InnoDB 行级锁（不升级为更粗粒度的锁）和 Oracle 风格的一致非锁读提高了多用户并发性和性能。InnoDB 将用户数据存储在聚集索引中，以减少基于主键的常见查询的 I/O。为了保持数据完整性，

InnoDB 还支持外键引用完整性约束。

特点：

支持事务，支持外键，因此数据的完整性、一致性更高。

支持行级别的锁和表级别的锁。

支持读写并发，写不阻塞读（MVCC）。

特殊的索引存放方式，可以减少 IO，提升查询效率。

适合：经常更新的表，存在并发读写或者有事务处理的业务系统。

Memory（1 个文件）

Stores all data in RAM, for fast access in environments that require quick lookups of non-critical data. This engine was formerly known as the HEAP engine. Its use cases are decreasing; InnoDB with its buffer pool memory area provides a general-purpose and durable way to keep most or all data in memory, and NDBCLUSTER provides fast key-value lookups for huge distributed data sets.

将所有数据存储在 RAM 中，以便在需要快速查找非关键数据的环境中快速访问。这个引擎以前被称为堆引擎。其使用案例正在减少；InnoDB 及其缓冲池内存区域提供了一种通用、持久的方法来将大部分或所有数据保存在内存中，而 ndbcluster 为大型分布式数据集提供了快速的键值查找。

特点：

把数据放在内存里面，读写的速度很快，但是数据库重启或者崩溃，数据会全部消失。只适合做临时表。

将表中的数据存储到内存中。

CSV（3 个文件）

Its tables are really text files with comma-separated values. CSV tables let you import or dump data in CSV format, to exchange data with scripts and applications that read and write that same format. Because CSV tables are not indexed, you typically keep the data in InnoDB tables during normal operation, and only use CSV tables during the import or export stage.

它的表实际上是带有逗号分隔值的文本文件。csv 表允许以 csv 格式导入或转储数据，以便与读写相同格式的脚本和应用程序交换数据。因为 csv 表没有索引，所以通常在正常操作期间将数据保存在 innodb 表中，并且只在导入或导出阶段使用 csv 表。

特点：不允许空行，不支持索引。格式通用，可以直接编辑，适合在不同数据库之间导入导出。

Archive（2 个文件）

These compact, unindexed tables are intended for storing and retrieving large amounts of seldom-referenced historical, archived, or security audit information.

这些紧凑的未索引的表用于存储和检索大量很少引用的历史、存档或安全审计信息。

特点：不支持索引，不支持 update delete。

这是 MySQL 里面常见的一些存储引擎，我们看到了，不同的存储引擎提供的特性都不一样，它们有不同的存储机制、索引方式、锁定水平等功能。

我们在不同的业务场景中对数据操作的要求不同，就可以选择不同的存储引擎来满足我们的需求，这个就是 MySQL 支持这么多存储引擎的原因。

1.5.4. 如何选择存储引擎？

如果对数据一致性要求比较高，需要事务支持，可以选择 InnoDB。

如果数据查询多更新少，对查询性能要求比较高，可以选择 MyISAM。

如果需要一个用于查询的临时表，可以选择 Memory。

如果所有的存储引擎都不能满足你的需求，并且技术能力足够，可以根据官网内部手册用 C 语言开发一个存储引擎：

https://dev.mysql.com/doc/internals/en/custom-engine.html

1.6. 执行引擎（Query Execution Engine），返回结果

OK，存储引擎分析完了，它是我们存储数据的形式，继续第二个问题，是谁使用执行计划去操作存储引擎呢?

这就是我们的执行引擎，它利用存储引擎提供的相应的 API 来完成操作。

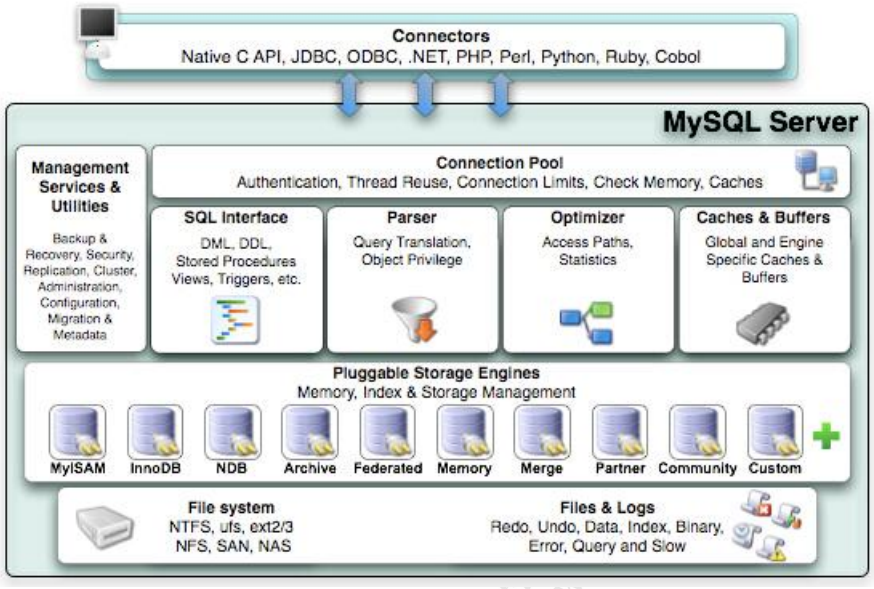
为什么我们修改了表的存储引擎，操作方式不需要做任何改变？因为不同功能的存储引擎实现的 API 是相同的。

最后把数据返回给客户端，即使没有结果也要返回。

2. MySQL 体系结构总结

基于上面分析的流程，我们一起来梳理一下 MySQL 的内部模块。

2.1. 模块详解



1、 Connector：用来支持各种语言和 SQL 的交互，比如 PHP，Python，Java 的 JDBC；

2、 Management Serveices & Utilities：系统管理和控制工具，包括备份恢复、

MySQL 复制、集群等等；

3、 Connection Pool：连接池，管理需要缓冲的资源，包括用户密码权限线程等等；

4、 SQL Interface：用来接收用户的 SQL 命令，返回用户需要的查询结果

5、 Parser：用来解析 SQL 语句；

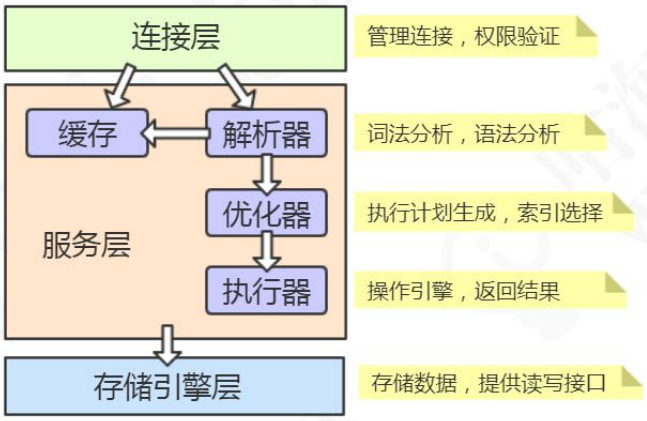
6、 Optimizer：查询优化器；

7、 Cache and Buffer：查询缓存，除了行记录的缓存之外，还有表缓存，Key 缓存，权限缓存等等；

8、 Pluggable Storage Engines：插件式存储引擎，它提供 API 给服务层使用，跟具体的文件打交道。

2.2. 架构分层

总体上，我们可以把 MySQL 分成三层，跟客户端对接的连接层，真正执行操作的服务层，和跟硬件打交道的存储引擎层（参考 MyBatis：接口、核心、基础）。



2.1.1. 连接层

我们的客户端要连接到 MySQL 服务器 3306 端口，必须要跟服务端建立连接，那么管理所有的连接，验证客户端的身份和权限，这些功能就在连接层完成。

2.1.2. 服务层

连接层会把 SQL 语句交给服务层，这里面又包含一系列的流程：

比如查询缓存的判断、根据 SQL 调用相应的接口，对我们的 SQL 语句进行词法和语法的解析（比如关键字怎么识别，别名怎么识别，语法有没有错误等等）。

然后就是优化器，MySQL 底层会根据一定的规则对我们的 SQL 语句进行优化，最后再交给执行器去执行。

2.1.3. 存储引擎

存储引擎就是我们的数据真正存放的地方，在 MySQL 里面支持不同的存储引擎。

再往下就是内存或者磁盘。

3. 一条更新 SQL 是如何执行的？

讲完了查询流程，我们是不是再讲讲更新流程、插入流程和删除流程？

在数据库里面，我们说的 update 操作其实包括了更新、插入和删除。如果大家有看过 MyBatis 的源码，应该知道 Executor 里面也只有 doQuery()和 doUpdate()的方法，没有 doDelete()和 doInsert()。

更新流程和查询流程有什么不同呢？

基本流程也是一致的，也就是说，它也要经过解析器、优化器的处理，最后交给执行器。

区别就在于拿到符合条件的数据之后的操作。

3.1. 缓冲池 Buffer Pool

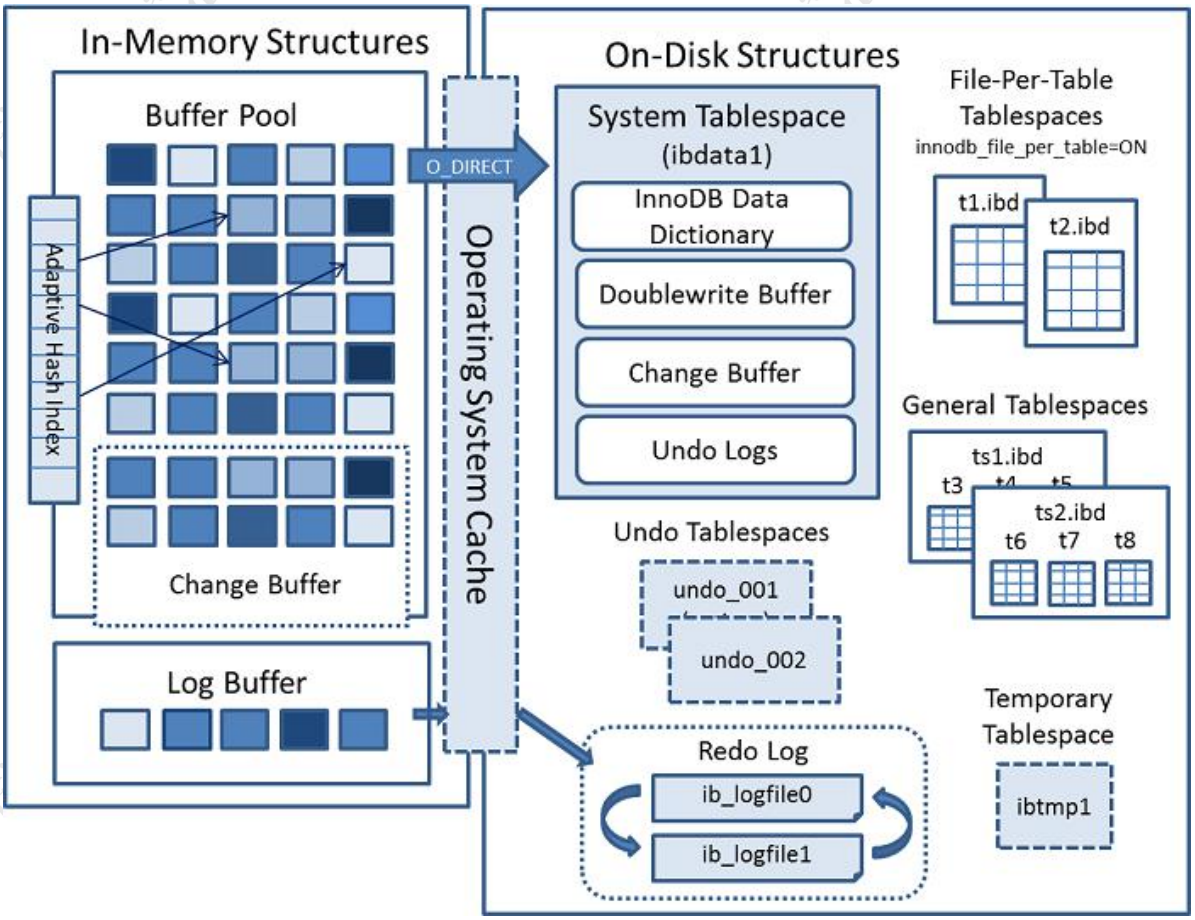
首先，InnnoDB 的数据都是放在磁盘上的，InnoDB 操作数据有一个最小的逻辑单位，叫做页（索引页和数据页）。我们对于数据的操作，不是每次都直接操作磁盘，因为磁盘的速度太慢了。InnoDB 使用了一种缓冲池的技术，也就是把磁盘读到的页放到一块内存区域里面。这个内存区域就叫 Buffer Pool。

下一次读取相同的页，先判断是不是在缓冲池里面，如果是，就直接读取，不用再次访问磁盘。

修改数据的时候，先修改缓冲池里面的页。内存的数据页和磁盘数据不一致的时候，我们把它叫做脏页。InnoDB 里面有专门的后台线程把 Buffer Pool 的数据写入到磁盘，每隔一段时间就一次性地把多个修改写入磁盘，这个动作就叫做刷脏。

Buffer Pool 是 InnoDB 里面非常重要的一个结构，它的内部又分成几块区域。这里我们趁机到官网来认识一下 InnoDB 的内存结构和磁盘结构。

3.2. InnoDB 内存结构和磁盘结构



3.3.1. 内存结构

Buffer Pool 主要分为 3 个部分： Buffer Pool、Change Buffer、Adaptive Hash Index，另外还有一个（redo）log buffer。

1、Buffer Pool

Buffer Pool 缓存的是页面信息，包括数据页、索引页。

查看服务器状态，里面有很多跟 Buffer Pool 相关的信息：

```
SHOW STATUS LIKE '%innodb_buffer_pool%';
```

这些状态都可以在官网查到详细的含义，用搜索功能。

https://dev.mysql.com/doc/refman/5.7/en/server-status-variables.html

Variable_name	Value
Innodb_buffer_pool_dump_status	Dumping of buffer
Innodb_buffer_pool_load_status	Buffer pool(s) load
Innodb_buffer_pool_resize_status	
Innodb_buffer_pool_pages_data	1218
Innodb_buffer_pool_bytes_data	19955712
Innodb_buffer_pool_pages_dirty	0
Innodb_buffer_pool_bytes_dirty	0
Innodb_buffer_pool_pages_flushed	336
Innodb_buffer_pool_pages_free	6972
Innodb_buffer_pool_pages_misc	1
Innodb_buffer_pool_pages_total	8191
Innodb_buffer_pool_read_ahead_rnd	0
Innodb_buffer_pool_read_ahead	0
Innodb_buffer_pool_read_ahead_evicted	0
Innodb_buffer_pool_read_requests	7189
Innodb_buffer_pool_reads	1162
Innodb_buffer_pool_wait_free	0
Innodb_buffer_pool_write_requests	2820

Buffer Pool 默认大小是 128M（134217728 字节），可以调整。

查看参数（系统变量）：

```
SHOW VARIABLES like '%innodb_buffer_pool%';
```

这些参数都可以在官网查到详细的含义，用搜索功能。

https://dev.mysql.com/doc/refman/5.7/en/server-system-variables.html

Variable_name	Value
innodb_buffer_pool_chunk_size	134217728
innodb_buffer_pool_dump_at_shutdown	ON
innodb_buffer_pool_dump_now	OFF
innodb_buffer_pool_dump_pct	25
innodb_buffer_pool_filename	ib_buffer_pool
innodb_buffer_pool_instances	1
innodb_buffer_pool_load_abort	OFF
innodb_buffer_pool_load_at_startup	ON
innodb_buffer_pool_load_now	OFF
innodb_buffer_pool_size	134217728

文件、实例、预热

内存的缓冲池写满了怎么办？（Redis 设置的内存满了怎么办？）InnoDB 用 LRU 算法来管理缓冲池（链表实现，不是传统的 LRU，分成了 young 和 old），经过淘汰的数据就是热点数据。

内存缓冲区对于提升读写性能有很大的作用。思考一个问题：

当需要更新一个数据页时，如果数据页在 Buffer Pool 中存在，那么就直接更新好了。

否则的话就需要从磁盘加载到内存，再对内存的数据页进行操作。也就是说，如果没有命中缓冲池，至少要产生一次磁盘 IO，有没有优化的方式呢？

2、Change Buffer 写缓冲

如果这个数据页不是唯一索引，不存在数据重复的情况，也就不需要从磁盘加载索引页判断数据是不是重复（唯一性检查）。这种情况下可以先把修改记录在内存的缓冲池中，从而提升更新语句（Insert、Delete、Update）的执行速度。

这一块区域就是 Change Buffer。5.5 之前叫 Insert Buffer 插入缓冲，现在也能支持 delete 和 update。

最后把 Change Buffer 记录到数据页的操作叫做 merge。什么时候发生 merge？有几种情况：在访问这个数据页的时候，或者通过后台线程、或者数据库 shut down、redo log 写满时触发。

如果数据库大部分索引都是非唯一索引，并且业务是写多读少，不会在写数据后立刻读取，就可以使用 Change Buffer（写缓冲）。写多读少的业务，调大这个值：

```
SHOW VARIABLES LIKE 'innodb_change_buffer_max_size';
```

代表 Change Buffer 占 Buffer Pool 的比例，默认 25%。

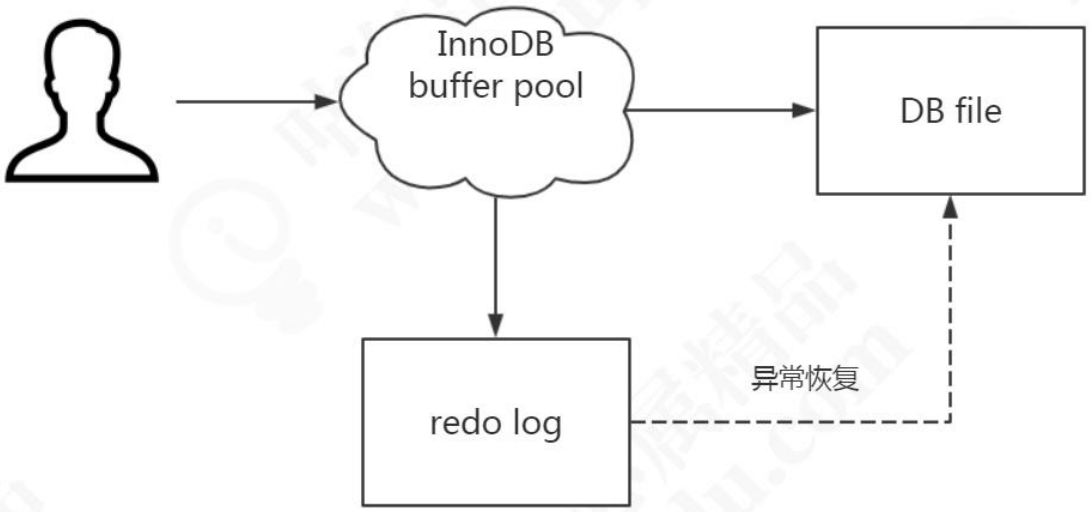
3、Adaptive Hash Index

索引应该是放在磁盘的，为什么要专门把一种哈希的索引放到内存？下次课再说。

4、（redo）Log Buffer

思考一个问题：如果 Buffer Pool 里面的脏页还没有刷入磁盘时，数据库宕机或者重启，这些数据丢失。如果写操作写到一半，甚至可能会破坏数据文件导致数据库不可用。

为了避免这个问题，InnoDB 把所有对页面的修改操作专门写入一个日志文件，并且在数据库启动时从这个文件进行恢复操作（实现 crash-safe）——用它来实现事务的持久性。



这个文件就是磁盘的 redo log（叫做重做日志），对应于/var/lib/mysql/目录下的 ib_logfile0 和 ib_logfile1，每个 48M。

这种日志和磁盘配合的整个过程，其实就是 MySQL 里的 WAL 技术（Write-Ahead Logging），它的关键点就是先写日志，再写磁盘。

```
show variables like 'innodb_log%';
```

值	含义
innodb_log_file_size	指定每个文件的大小，默认 48M
innodb_log_files_in_group	指定文件的数量，默认为 2
innodb_log_group_home_dir	指定文件所在路径，相对或绝对。如果不指定，则为 datadir 路径。

问题：

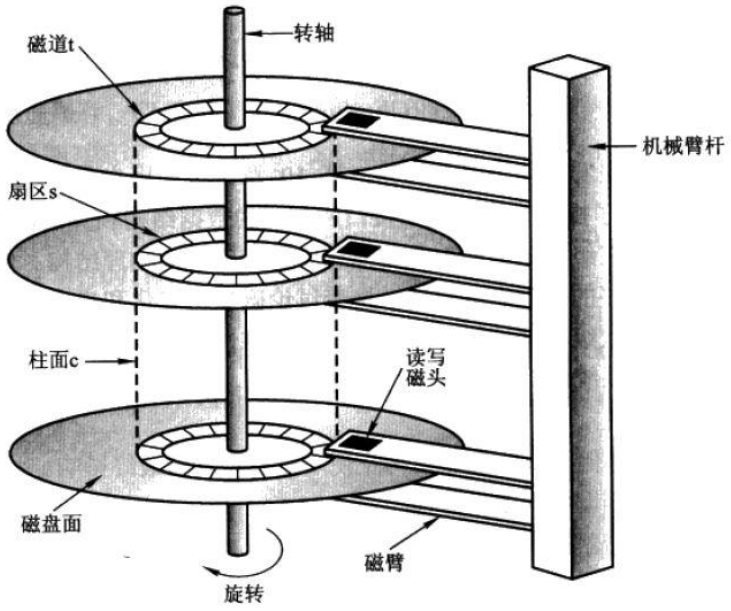
同样是写磁盘，为什么不直接写到 db file 里面去？为什么先写日志再写磁盘？

我们先来了解一下随机 I/O 和顺序 I/O 的概念。

磁盘的最小组成单元是扇区，通常是 512 个字节。

操作系统和内存打交道，最小的单位是页 Page。

操作系统和磁盘打交道，读写磁盘，最小的单位是块 Block。



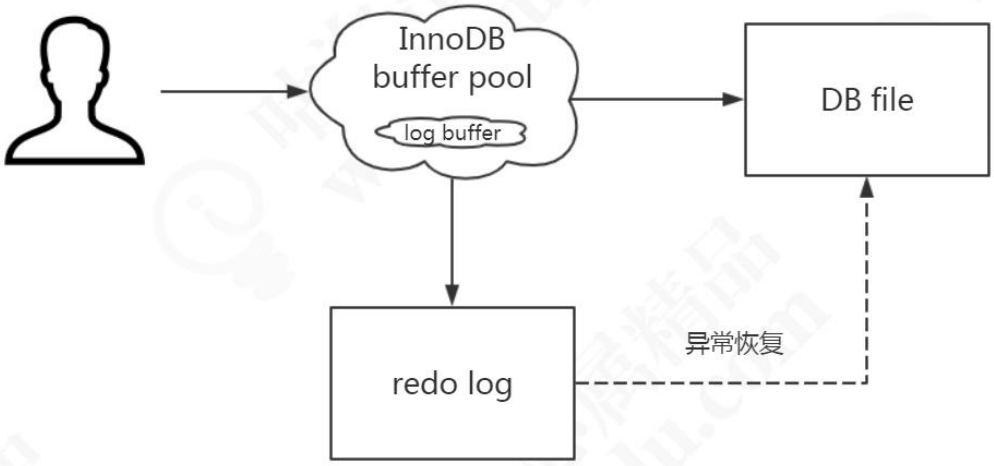
如果我们所需要的数据是随机分散在不同页的不同扇区中，那么找到相应的数据需要等到磁臂旋转到指定的页，然后盘片寻找到对应的扇区，才能找到我们所需要的一块数据，一次进行此过程直到找完所有数据，这个就是随机 IO，读取数据速度较慢。

假设我们已经找到了第一块数据，并且其他所需的数据就在这一块数据后边，那么

就不需要重新寻址，可以依次拿到我们所需的数据，这个就叫顺序 IO。

刷盘是随机 I/O，而记录日志是顺序 I/O，顺序 I/O 效率更高。因此先把修改写入日志，可以延迟刷盘时机，进而提升系统吞吐。

当然 redo log 也不是每一次都直接写入磁盘，在 Buffer Pool 里面有一块内存区域（Log Buffer）专门用来保存即将要写入日志文件的数据，默认 16M，它一样可以节省磁盘 IO。




```
SHOW VARIABLES LIKE 'innodb_log_buffer_size';
```

需要注意：redo log 的内容主要是用于崩溃恢复。磁盘的数据文件，数据来自 buffer pool。redo log 写入磁盘，不是写入数据文件。

那么，Log Buffer 什么时候写入 log file?

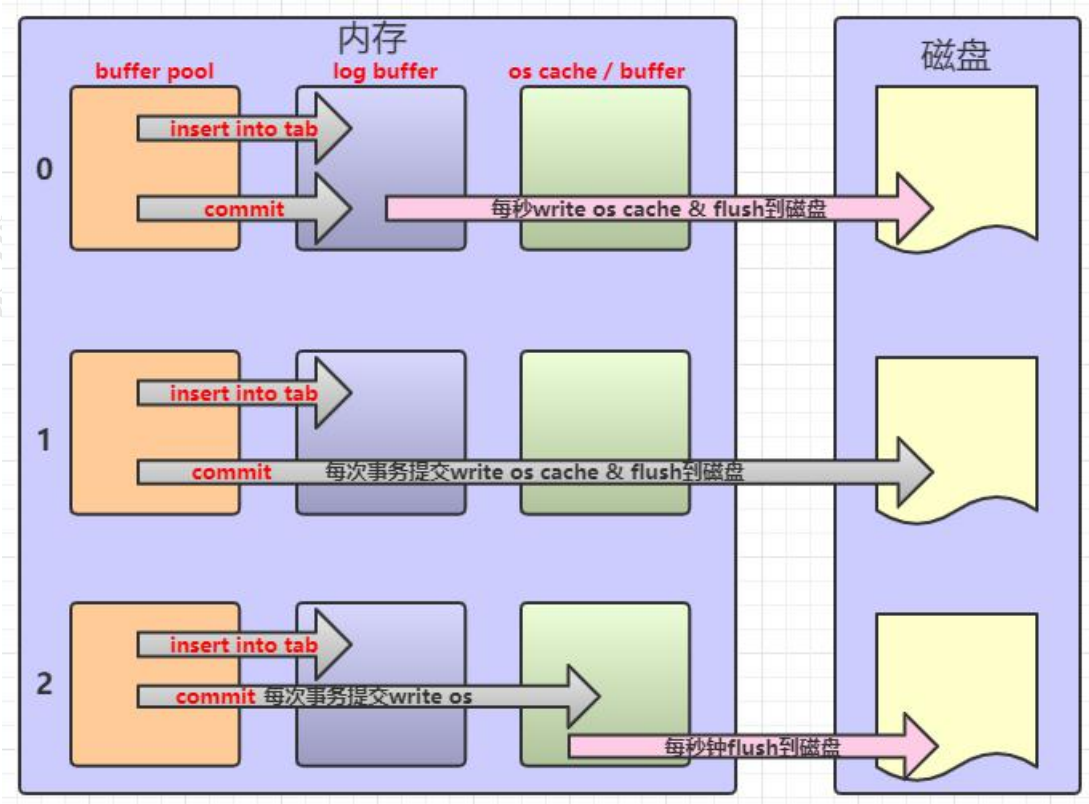
在我们写入数据到磁盘的时候，操作系统本身是有缓存的。flush 就是把操作系统缓冲区写入到磁盘。

log buffer 写入磁盘的时机，由一个参数控制，默认是 1。

```
SHOW VARIABLES LIKE 'innodb_flush_log_at_trx_commit';
```

https://dev.mysql.com/doc/refman/5.7/en/innodb-parameters.html#sysvar_innodb_flush_log_at_trx_commit

值	含义
0（延迟写）	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。
1（默认，实时写，实时刷）	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。
2（实时写，延迟刷）	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。

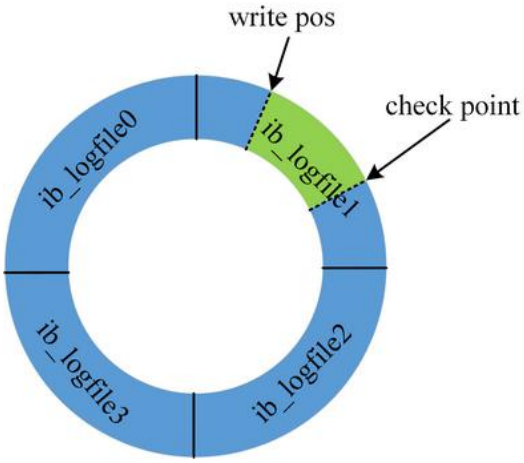


这是内存结构的第 4 块内容，redo log，它又分成内存和磁盘两部分。redo log 有什么特点?

1、redo log 是 InnoDB 存储引擎实现的，并不是所有存储引擎都有。

2、不是记录数据页更新之后的状态，而是记录这个页做了什么改动，属于物理日志。

3、redo log 的大小是固定的，前面的内容会被覆盖。



check point 是当前要覆盖的位置。如果 write pos 跟 check point 重叠，说明 redo log 已经写满，这时候需要同步 redo log 到磁盘中。

这是 MySQL 的内存结构，总结一下，分为：

Buffer pool、change buffer、Adaptive Hash Index、 log buffer。

磁盘结构里面主要是各种各样的表空间，叫做 Table space。

3.3.2. 磁盘结构

表空间可以看做是 InnoDB 存储引擎逻辑结构的最高层，所有的数据都存放在表空间中。InnoDB 的表空间分为 5 大类。

系统表空间 system tablespace

在默认情况下 InnoDB 存储引擎有一个共享表空间（对应文件/var/lib/mysql/ibdata1），也叫系统表空间。

InnoDB 系统表空间包含 InnoDB 数据字典和双写缓冲区，Change Buffer 和 Undo

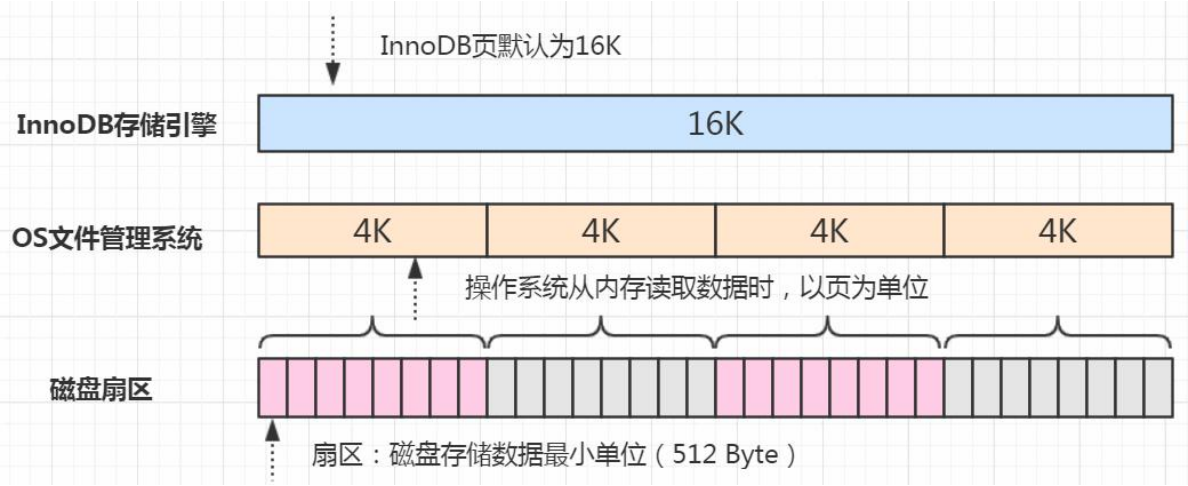
Logs），如果没有指定 file-per-table，也包含用户创建的表和索引数据。

1、undo 在后面介绍，因为有独立的表空间。

2、数据字典：由内部系统表组成，存储表和索引的元数据（定义信息）。

3、双写缓冲（InnoDB 的一大特性）：

InnoDB 的页和操作系统的页大小不一致，InnoDB 页大小一般为 16K，操作系统页大小为 4K，InnoDB 的页写入到磁盘时，一个页需要分 4 次写。



如果存储引擎正在写入页的数据到磁盘时发生了宕机，可能出现页只写了一部分的情况，比如只写了 4K，就宕机了，这种情况叫做部分写失效（partial page write），可能会导致数据丢失。

```
show variables like 'innodb_doublewrite';
```

我们不是有 redo log 吗？但是有个问题，如果这个页本身已经损坏了，用它来做崩溃恢复是没有意义的。所以在对于应用 redo log 之前，需要一个页的副本。如果出现了写入失效，就用页的副本来还原这个页，然后再应用 redo log。这个页的副本就是 double write，InnoDB 的双写技术。通过它实现了数据页的可靠性。

跟 redo log 一样，double write 由两部分组成，一部分是内存的 double write，

一个部分是磁盘上的 double write。因为 double write 是顺序写入的，不会带来很大的开销。

在默认情况下，所有的表共享一个系统表空间，这个文件会越来越大，而且它的空间不会收缩。

独占表空间 file-per-table tablespaces

我们可以让每张表独占一个表空间。这个开关通过 innodb_file_per_table 设置，默认开启。

```
SHOW VARIABLES LIKE 'innodb_file_per_table';
```

开启后，则每张表会开辟一个表空间，这个文件就是数据目录下的 ibd 文件（例如 /var/lib/mysql/gupao/user_innodb.ibd），存放表的索引和数据。

但是其他类的数据，如回滚（undo）信息，插入缓冲索引页、系统事务信息，二次写缓冲（Double write buffer）等还是存放在原来的共享表空间内。

通用表空间 general tablespaces

通用表空间也是一种共享的表空间，跟 ibdata1 类似。

可以创建一个通用的表空间，用来存储不同数据库的表，数据路径和文件可以自定义。语法：

```
create tablespace ts2673 add datafile '/var/lib/mysql/ts2673.ibd' file_block_size=16K engine=innodb;
```

在创建表的时候可以指定表空间，用 ALTER 修改表空间可以转移表空间。

```
create table t2673(id integer) tablespace ts2673;
```

不同表空间的数据是可以移动的。

删除表空间需要先删除里面的所有表：

```
drop table t2673;
drop tablespace ts2673;
```

临时表空间 temporary tablespaces

存储临时表的数据，包括用户创建的临时表，和磁盘的内部临时表。对应数据目录下的 ibtmp1 文件。当数据服务器正常关闭时，该表空间被删除，下次重新产生。

Redo log

磁盘结构里面的 redo log，在前面已经介绍过了。

undo log tablespace

https://dev.mysql.com/doc/refman/5.7/en/innodb-undo-tablespaces.html
https://dev.mysql.com/doc/refman/5.7/en/innodb-undo-logs.html

undo log (撤销日志或回滚日志) 记录了事务发生之前的数据状态 (不包括 select)。如果修改数据时出现异常，可以用 undo log 来实现回滚操作（保持原子性）。

在执行 undo 的时候，仅仅是将数据从逻辑上恢复至事务之前的状态，而不是从物理页面上操作实现的，属于逻辑格式的日志。

redo Log 和 undo Log 与事务密切相关，统称为事务日志。

undo Log 的数据默认在系统表空间 ibdata1 文件中，因为共享表空间不会自动收缩，也可以单独创建一个 undo 表空间。

```
show global variables like '%undo%';
```

有了这些日志之后，我们来总结一下一个更新操作的流程，这是一个简化的过程。

name 原值是 qingshan。

```
update user set name = 'penyuyan' where id=1;
```

1、事务开始，从内存或磁盘取到这条数据，返回给 Server 的执行器；

2、执行器修改这一行数据的值为 penyuyan；

3、记录 name=qingshan 到 undo log；

4、记录 name=penyuyan 到 redo log；

5、调用存储引擎接口，在内存（Buffer Pool）中修改 name=penyuyan；

6、事务提交。

内存和磁盘之间，工作着很多后台线程。

3.3.3. 后台线程

（供了解）

后台线程的主要作用是负责刷新内存池中的数据和把修改的数据页刷新到磁盘。后台线程分为：master thread，IO thread，purge thread，page cleaner thread。

master thread 负责刷新缓存数据到磁盘并协调调度其它后台进程。

IO thread 分为 insert buffer、log、read、write 进程。分别用来处理 insert buffer、重做日志、读写请求的 IO 回调。

purge thread 用来回收 undo 页。

page cleaner thread 用来刷新脏页。

除了 InnoDB 架构中的日志文件，MySQL 的 Server 层也有一个日志文件，叫做 binlog，它可以被所有的存储引擎使用。

3.3. Binlog

https://dev.mysql.com/doc/refman/5.7/en/binary-log.html

binlog 以事件的形式记录了所有的 DDL 和 DML 语句（因为它记录的是操作而不是数据值，属于逻辑日志），可以用来做主从复制和数据恢复。

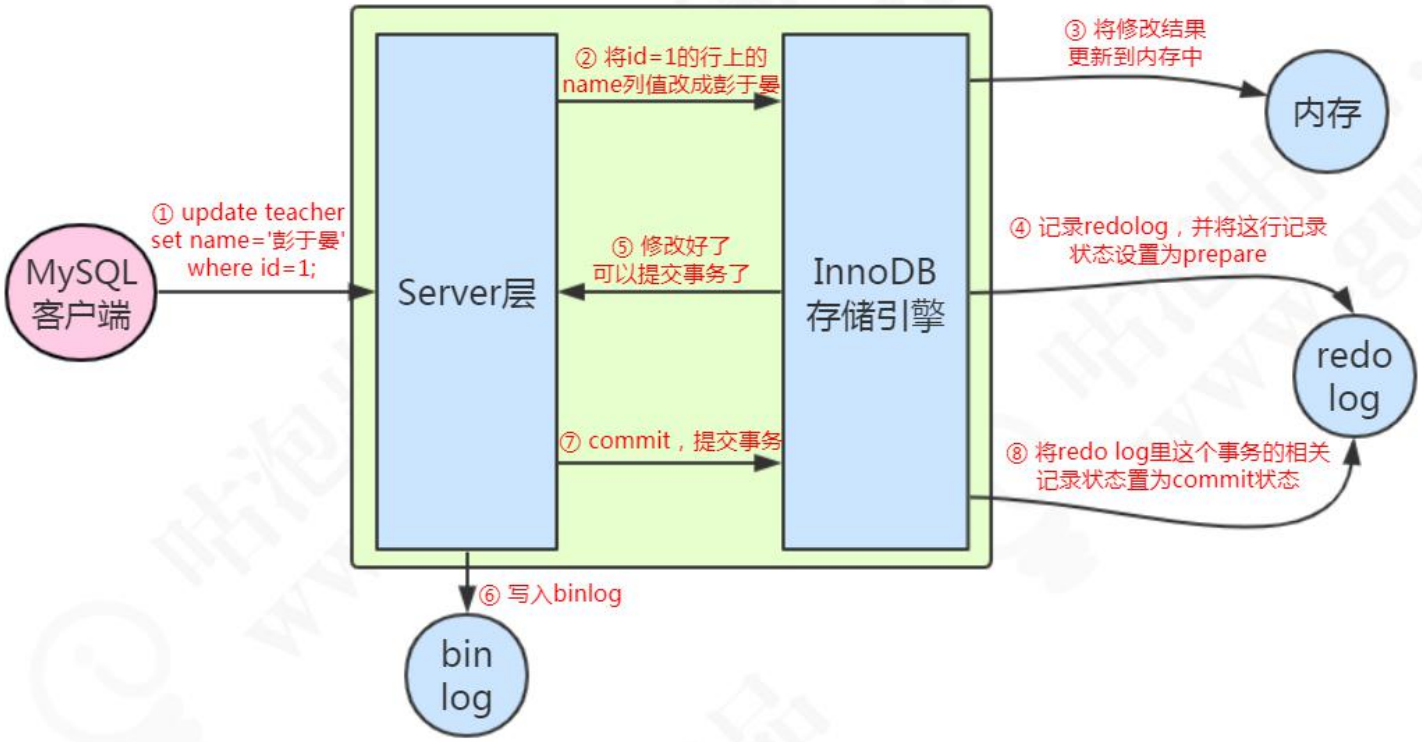
跟 redo log 不一样，它的文件内容是可以追加的，没有固定大小限制。

在开启了 binlog 功能的情况下，我们可以把 binlog 导出成 SQL 语句，把所有的操作重放一遍，来实现数据的恢复。

binlog 的另一个功能就是用来实现主从复制，它的原理就是从服务器读取主服务器的 binlog，然后执行一遍。

配置方式和主从复制的实现原理在 Mycat 第二节课中有讲述。

有了这两个日志之后，我们来看一下一条更新语句是怎么执行的：



整体流程

例如一条语句：update teacher set name='盆鱼宴' where id=1;

1、先查询到这条数据，如果有缓存，也会用到缓存。

2、把 name 改成盆鱼宴，然后调用引擎的 API 接口，写入这一行数据到内存，同时记录 redo log。这时 redo log 进入 prepare 状态，然后告诉执行器，执行完成了，可以随时提交。

3、执行器收到通知后记录 binlog，然后调用存储引擎接口，设置 redo log 为 commit 状态。

4、更新完成。

这张图片的重点（没必要背下来）：

1、先记录到内存，再写日志文件。

2、记录 redo log 分为两个阶段。

3、存储引擎和 Server 记录不同的日志。

3、先记录 redo，再记录 binlog。

作者：咕泡学院-青山

最后更新时间：2019 年 12 月 30 日 11:12:40