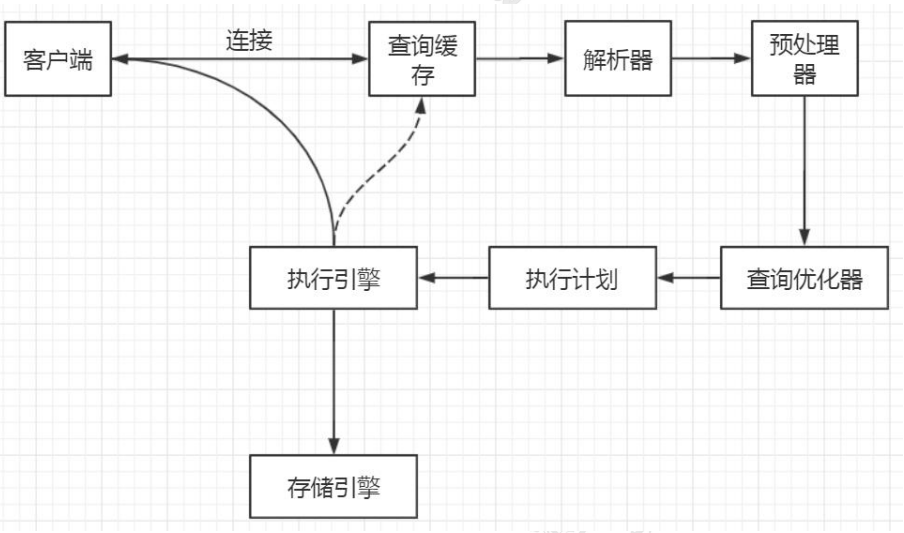
《MySQL 架构与内部模块》——青山

1. 一条查询 SQL 语句是如何执行的？



首先，数据是存储在 MySQL 服务端的。应用程序或者工具都是客户端。客户端要读写数据库，第一步要跟数据库建立连接。

1.1. 通信协议

1.1.1. 通信协议

通信类型：同步或者异步

一般来说客户端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。一般来说都是长连接，而且会把这个连接放到客户端的连接池。

可以用 show status 命令查看 MySQL 当前有多少个连接：

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

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。

字段	含义
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。

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。

MySQL 服务允许的最大连接数默认是 151 个，最大可以设置成 100000。

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

Variable_name	Value
max_connections	151

参数级别说明：

MySQL 中的参数分为 session 和 global 级别，分别是在当前会话中生效和全局生效，但是并不是每个参数都有两个级别，比如 max_connections 就只有全局级别。

当没有带参数的时候，默认是 session 级别，包括查询和修改。

比如修改了一个参数以后，在本窗口查询已经生效，但是其他窗口不生效：

```
show VARIABLES like 'autocommit';
set autocommit = on;
```

通信协议

在 Linux 服务器上，如果没有指定-h 参数，它就用 socket 方式登录。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;
```

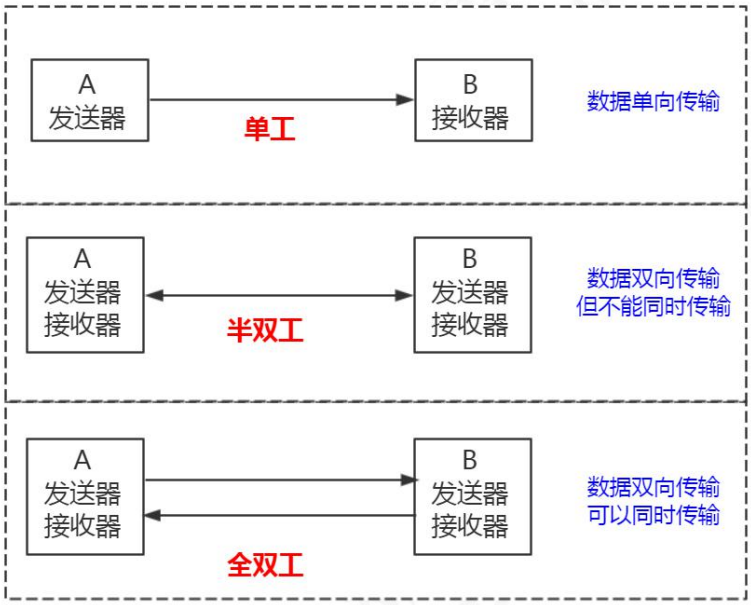
如果指定-h 参数，就会用第二种方式，TCP/IP 协议。

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

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
```

1.1.2. 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。

在一个连接中，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，在程序里面要避免不带 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。

1.2. 查询缓存

MySQL 内部自带了一个缓存模块。

在没有索引的字段上执行两次同样的查询：

```
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%';
```

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，例如中间多一个空格、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。

缓存这一块，还是交给 ORM 框架，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)

假如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 SQL 语句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：

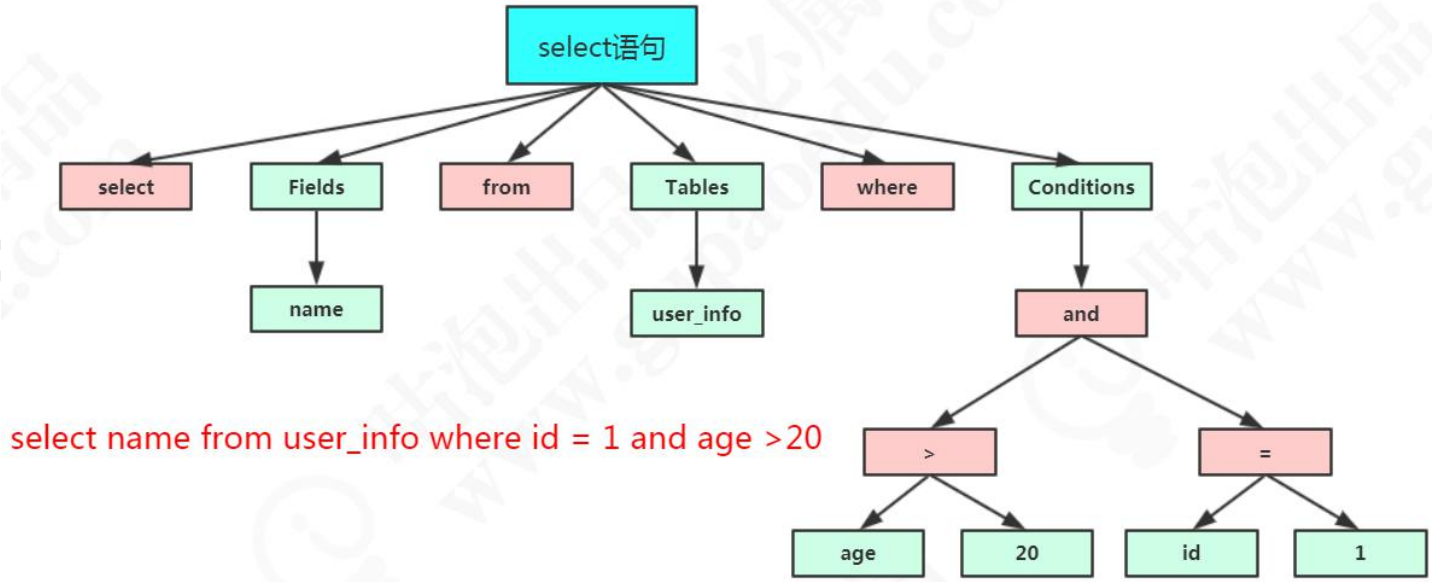
```
select name from user where id = 1;
```

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）。



词法语法分析是一个非常基础的功能，编译器、搜索引擎如果要识别语句，必须也要有词法语法分析功能。

1.3.3. 预处理器

如果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？是解析的时候报错还是执行的时候报错？

比如：

```
select * from penyuyan;
```

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？

解析树是一个可以被执行器认识的数据结构。

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（先访问哪张表）。

2、有多个索引可以使用的时候，选择哪个索引。

3、对于查询条件的优化，比如移除 1=1 之类的恒等式，移除不必要的括号，表达式的计算，子查询和连接查询的优化。

……

经过优化器处理之后，得到一个什么东西呢？

1.4.3. 优化器得到的结果

优化器最终会把解析树变成一个执行计划（execution_plans），执行计划也是一个数据结构。

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;
```

如果要得到详细的信息，还可以用 FORMAT=JSON。

1.5. 存储引擎

1.5.1. 存储引擎基本介绍

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。

1.5.2. 查看存储引擎

查看数据库里面已经存在的表的存储引擎：

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

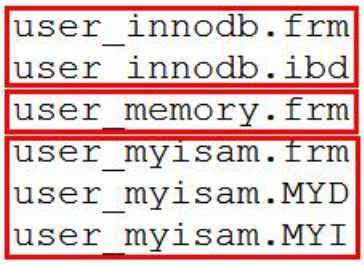
Name	Engine	Version	Row_format	Rows	Avg_row_length	Data_length
course	InnoDB	10	Dynamic	4	4096	16384
employees	InnoDB	10	Dynamic	10	1638	16384
student	InnoDB	10	Dynamic	2	8192	16384
teacher	InnoDB	10	Dynamic	3	5461	16384
teacher_contact	InnoDB	10	Dynamic	3	5461	16384
user_archive	ARCHIVE	10	Compressed	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

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（所以叫表类型）。而且，创建表之后还可以修改存储引擎。

```
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。



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。

1.5.3. 存储引擎比较

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)

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

MyISAM（3 个文件）

MySQL 自带的存储引擎，由 ISAM 升级而来。

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.

最开始是第三方公司为 MySQL 开发的。

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。

默认使用哈希索引。

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。

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

按照这个开发规范，实现相应的接口，给执行器操作。

也就是说，为什么能支持这么多存储引擎，还能自定义存储引擎，表的存储引擎改了对 Server 访问没有任何影响，就是因为大家都遵循了一定了规范，提供了相同的操作接口。

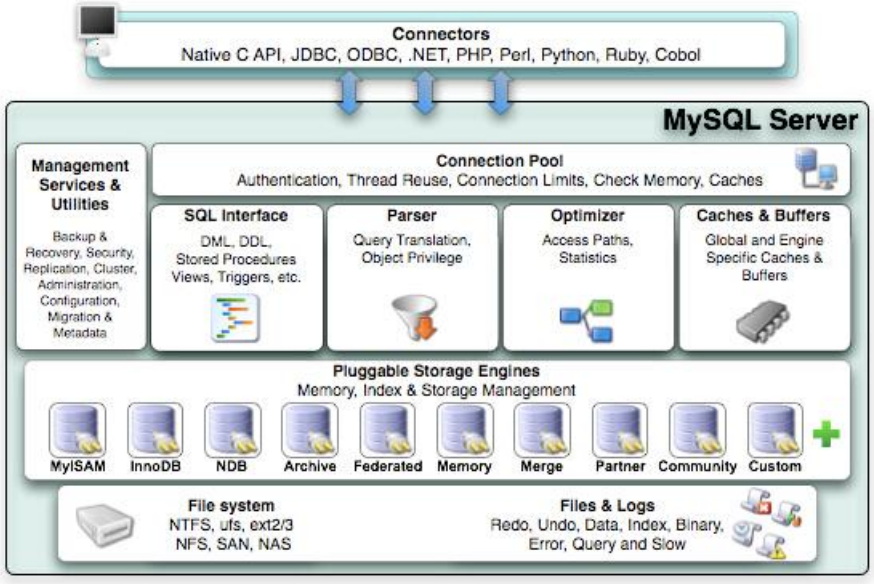
1.6. 执行引擎（Execution Engine），返回结果

执行器，或者叫执行引擎，它利用存储引擎提供的相应的 API 来完成操作。

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

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；

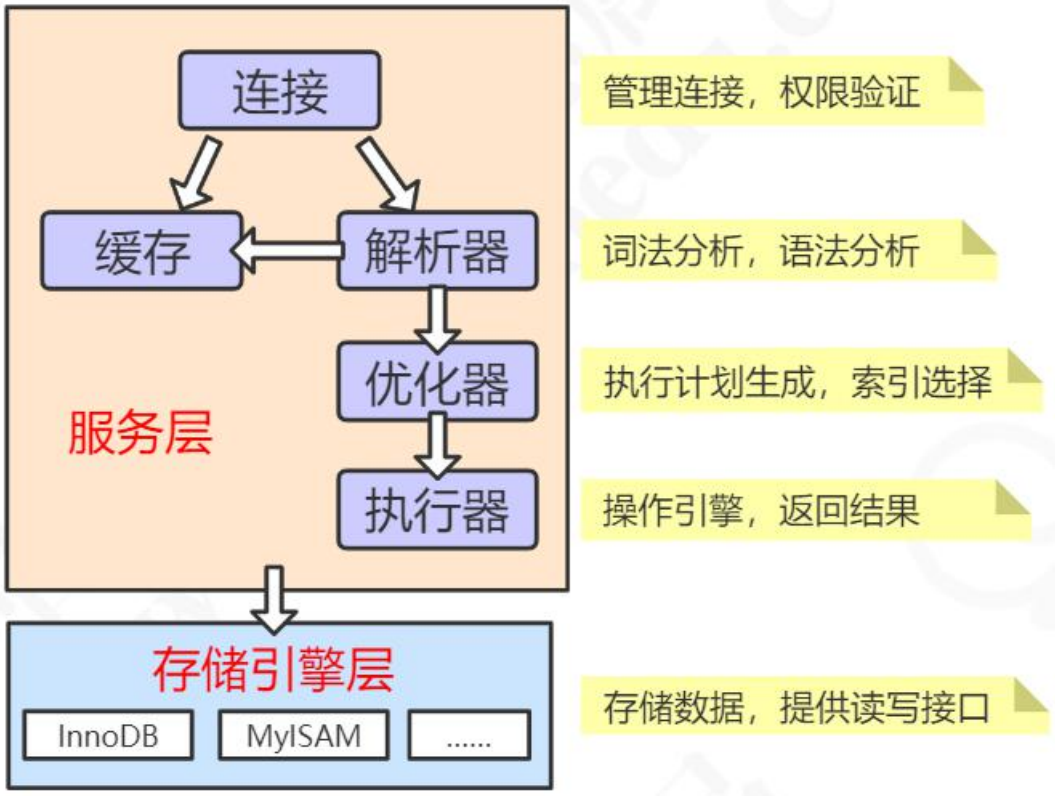
6、 Optimizer：查询优化器；

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 分成两层，执行操作的服务层，和存储管理数据的存储引擎层（参考 MyBatis：接口、核心、基础）。



2.1.1. 服务层

包括客户端跟服务端的连接，查询缓存的判断、对 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。

2.1.2. 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。

再往下就是文件管理系统，内存或者磁盘。

3. 一条更新 SQL 是如何执行的？

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，对于 InnoDB 存储引擎来说，数据都是放在磁盘上的，存储引擎要操作数据，必须先把磁盘里面的数据加载到内存里面才可以操作。

这里就有个问题，是不是我们需要的数据多大，我们就一次从磁盘加载多少数据到内存呢？磁盘 I/O 的读写相对于内存的操作来说是很慢的。如果我们需要的数据分散在磁盘的不同的地方，那就意味着会产生很多次的 I/O 操作。

所以，无论是操作系统也好，还是存储引擎也好，都有一个预读取的概念。也就是说，当磁盘上的一块数据被读取的时候，很有可能它附近的位置也会马上被读取到，这个就叫做局部性原理。那么这样，我们干脆每次多读取一点，而不是用多少读多少。

InnoDB 设定了一个存储引擎从磁盘读取数据到内存的最小的单位，叫做页。操作系统也有页的概念。操作系统的页大小一般是 4K，而在 InnoDB 里面，这个最小的单位默认是 16KB 大小，它是一个逻辑单位。如果要修改这个值的大小，必须修改源码重新编译安装。

我们要操作的数据就在这样的页里面，数据所在的页叫数据页。

设想一下：如果对于数据页的操作，每次都直接操作磁盘，从磁盘加载到内存，这

样会不会很慢？能不能把这些页缓存起来呢？

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。下一次读取相同的页，先判断是不是在这个内存区域里面，如果是，就直接读取，然后操作，不用再次从磁盘加载。

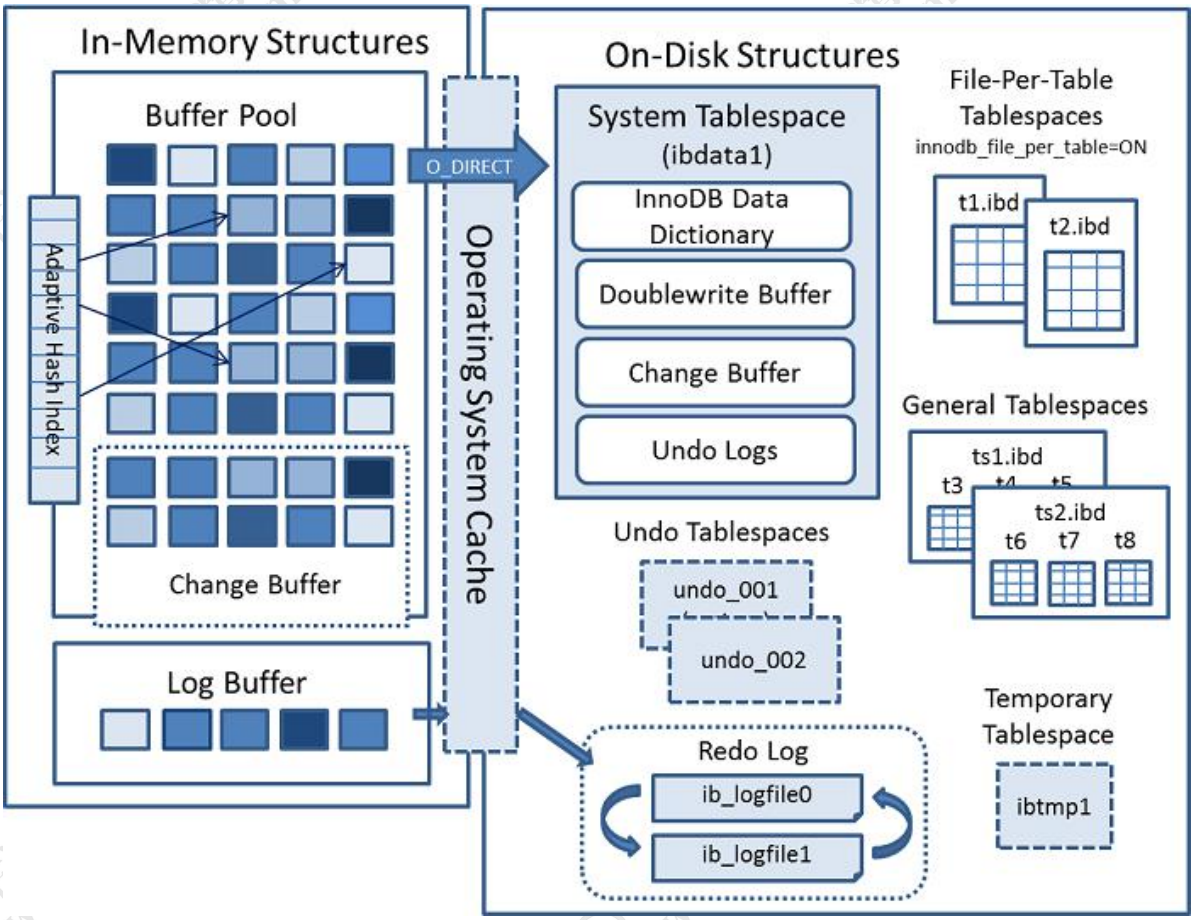
这个内存区域就叫 Buffer Pool。

修改数据的时候，先修改内存缓冲池里面的页。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。那脏页什么时候同步到磁盘呢？

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。

3.2. InnoDB 内存结构和磁盘结构

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



3.3.1. 内存结构

内存结构里面主要是 Buffer Pool、Change Buffer、Log Buffer、AHI。

1、Buffer Pool

https://dev.mysql.com/doc/refman/5.7/en/innodb-performance-buffer-pool.html

Buffer Pool 缓存的是 page 页面信息。

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。

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%';
```

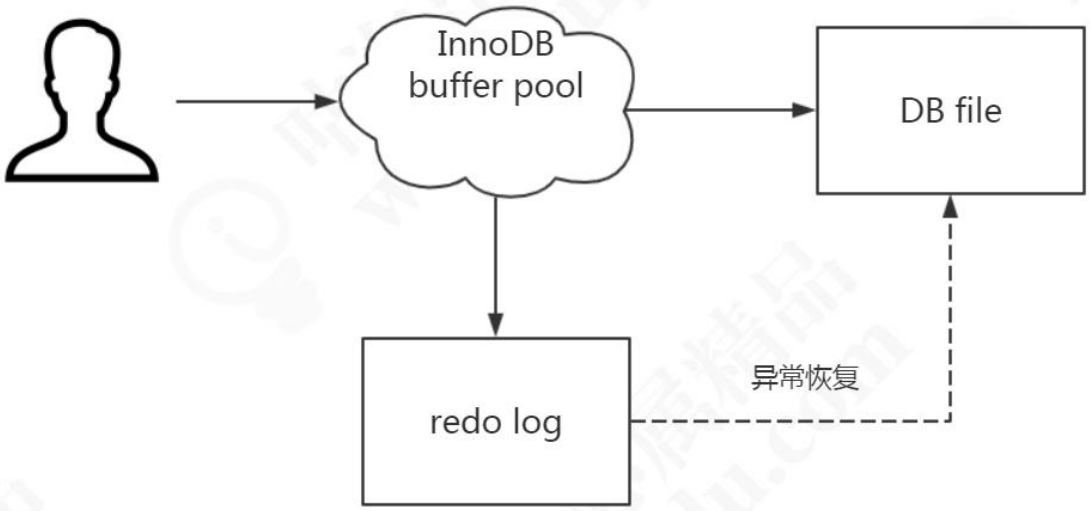
2、（redo）Log Buffer

因为刷脏不是实时的，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，这些数据就会丢失。

内存的数据必须要有一个持久化的措施。

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。

如果有未同步到磁盘的数据，数据库在启动的时候，会从这个日志文件进行恢复操作（实现 crash-safe）。我们说的事务的 ACID 里面 D（持久性），就是用它来实现的。



这个日志文件就是磁盘的 redo log（叫做重做日志），对应于/var/lib/mysql/目录下的 ib_logfile0 和 ib_logfile1，默认 2 个文件，每个 48M。

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

参数	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。

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？

写日志文件和和写到数据文件有什么区别？

如果我们所需要的数据是随机分散在磁盘上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（连续写的），顺序 I/O 效率更高。因此先把修改写入日志文件，在保证了内存数据的安全性的情况下，可以延迟刷盘时机，进而提升系统吞吐。

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。支持崩溃恢复是 InnoDB 的一个特性。

2、redo log 是物理日志，记录的是“在某个数据页上做了什么修改”。

3、redo log 的大小是固定的，前面的内容会被覆盖，一旦写满，就会触发 buffer pool 到磁盘的同步，以便腾出空间记录后面的修改。

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（只有 redo log 写满了，不能再记录更多内存的数据了，才把 buffer pool 刷盘，然后覆盖 redo log）。

除了 redo log 之外，还有一个跟修改有关的日志，叫做 undo log。redo log 和 undo log 与事务密切相关，统称为事务日志。

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。

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%';
```

值	含义
innodb_undo_directory	undo 文件的路径
innodb_undo_log_truncate	设置为 1，即开启在线回收（收缩）undo log 日志文件
innodb_max_undo_log_size	如果 innodb_undo_log_truncate 设置为 1，超过这个大小的时候会触发 truncate 回收（收缩）动作，如果 page 大小是 16KB，truncate 后空间缩小到 10M。默认 1073741824 字节=1G。
innodb_undo_logs	回滚段的数量， 默认 128，这个参数已经过时。
innodb_undo_tablespaces	表空间格式，最大 95，这个参数已经过时。

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，内存（buffer pool）或磁盘(data file)取到这条数据，返回给 Server 的执行器；

2、Server 的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。

3.3.2. 后台线程

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。

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

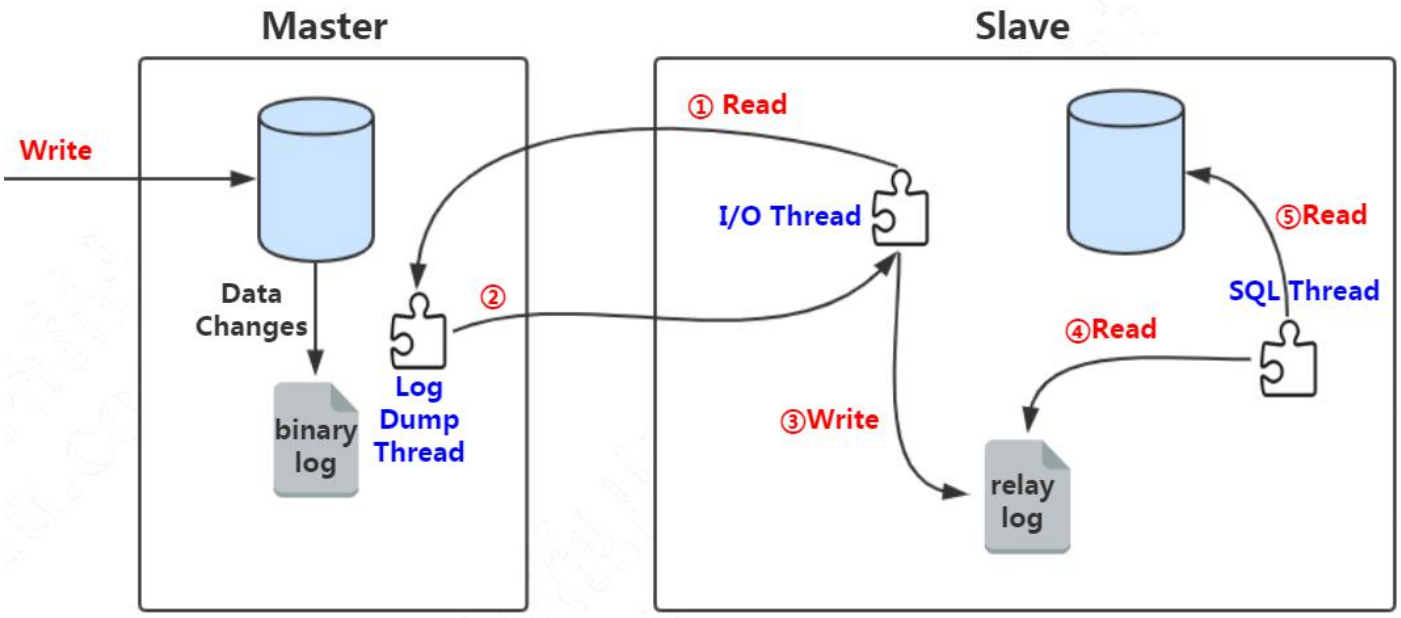
https://dev.mysql.com/doc/refman/5.7/en/binary-log.html

binlog 以事件的形式记录了所有的 DDL 和 DML 语句，比如“给 ID=1 这一行的 count 字段加 1 ”,因为它记录的是操作而不是数据值，属于逻辑日志）。binlog 可以用来做主从复制和数据恢复。

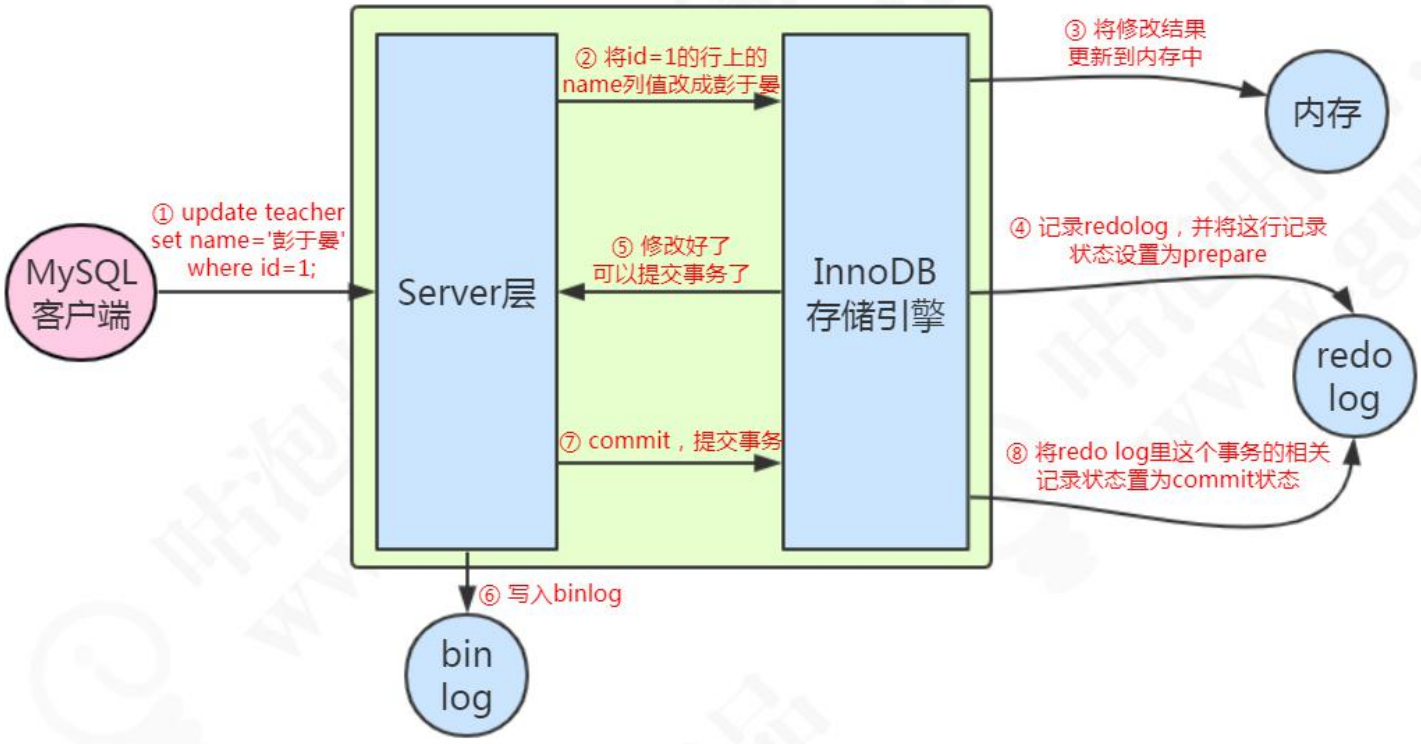
跟 redo log 不一样，它的文件内容是可以追加的，没有固定大小限制。

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（归档）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。



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（这里省略了undo）：



整体流程

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

1、先从内存或者磁盘拿到这条数据。

2、把 name 改成盆鱼宴，然后调用存储引擎的 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

4、然后调用存储引擎接口，设置 redo log 为 commit 状态。更新完成。

总结一下这张图片的重点：

1、先记录到内存（buffer pool），再写日志文件。

2、记录 redo log 分为两个阶段（prepare 和 commit）。

3、存储引擎和 server 分别记录不同的日志。

3、先记录 redo，再记录 binlog。