1. 逻辑架构剖析

1.1服务器处理客户端请求

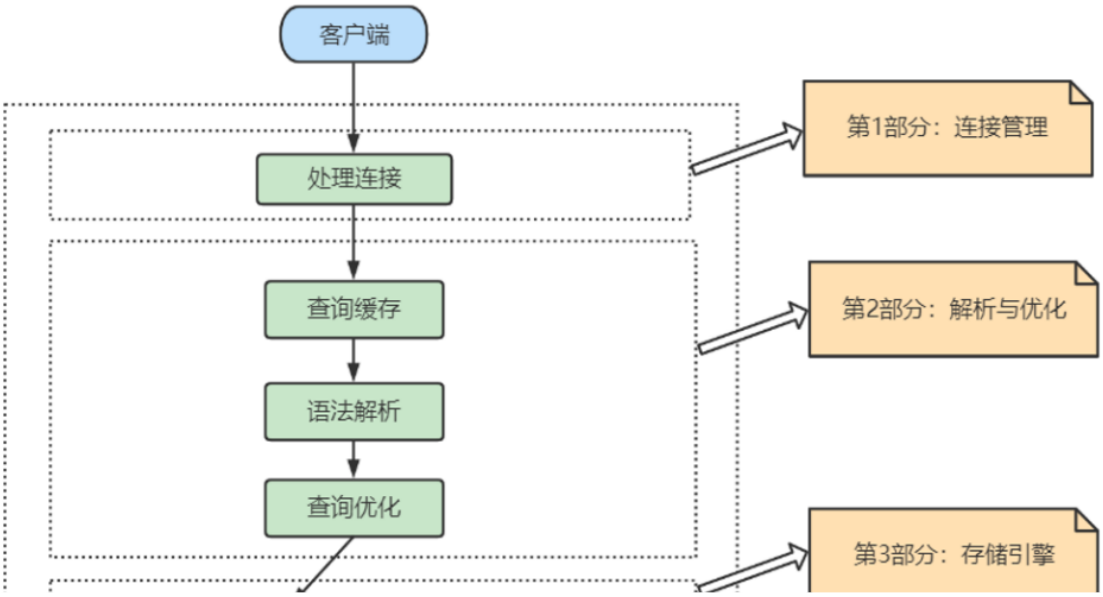
首先MySQL是典型的C/s架构, 即 `Client/Server` 架构,服务器端程序使用的`mysqld`

不论客户端进程和服务器进程是采用哪种方式进行通信,最后实现的效果都是:

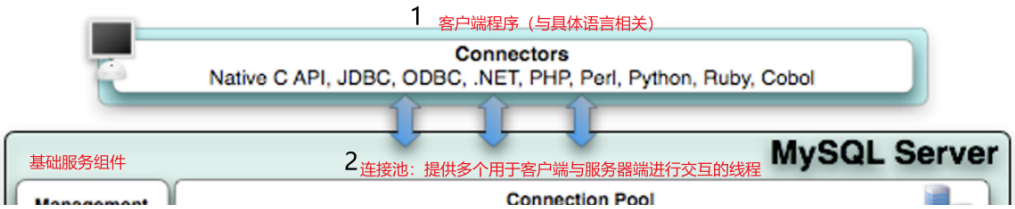
客户端进程向服务器进程发送一段文本(SQL语句) , 服务器进程处理后再向客户端进程发送一段文本(处理结果)

那服务器进程对客户端进程发送的请求做了什么处理,才能产生最后的处理结果呢?

这里以查询请求为例展示:



下面具体展开看一下：**MySQL5.7**



1.2 Connectors

Connectors, 指的是不同语言中与SQL的交互 （不同客户端与MySQL服务器进行交互）

MySQL首先是一个网络程序,在TCP之上定义了自己的应用层协议

所以要使用MySQL,我们可以编写代码,跟MySQL Server建立TCP连接,之后按照其定义好的协议进行交互

或者比较方便的办法是调用SDK,比如Native C API, JDBC, PHP等各语言MySQL Connector,或者通过ODBC

但通过SDK来访问MySQL,本质上还是在TCP连接上通过MySQL协议跟MySQL进行交互

接下来的MySQL Server结构可以分为如下的三层:

1.3 第1层：连接层

系统（客户端）访问 MySQL 服务器前，做的第一件事就是建立 TCP 连接

经过三次握手建立连接成功后， MySQL 服务器对 TCP 传输过来的账号密码做身份认证、权限获取

用户名或密码不对，会收到一个Access denied for user错误，客户端程序结束执行

用户名密码认证通过，会从权限表查出账号拥有的权限与连接关联，之后的权限判断逻辑，都将依 赖于此时读到的权限

接着我们来思考一个问题:

一个系统只会和MySQL服务器建立一个连接吗?只能有一个系统和MySQL服务器建立连接吗?

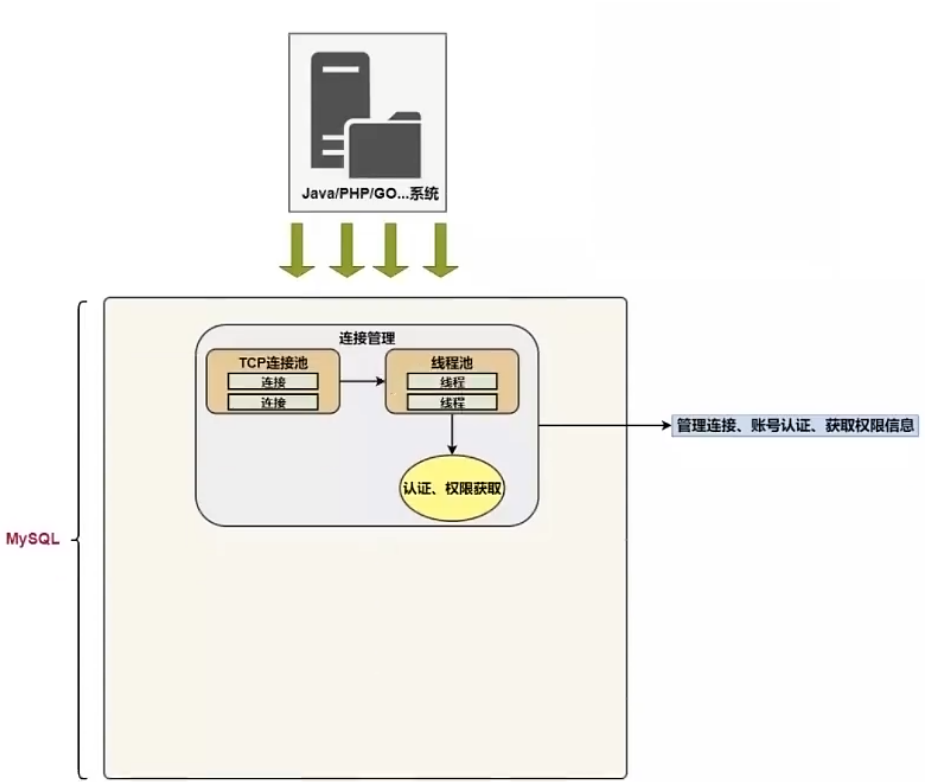
当然不是,多个系统都可以和MySQL服务器建立连接,每个系统建立的连接肯定不止一个

所以,为了解决TCP无限创建与TCP频繁创建销毁带来的资源耗尽、性能下降问题

MySQL服务器里有专门的`TCP连接池限制连接数`,采用`长连接模式复用TCP连接`,来解决上述问题

TCP 连接收到请求后，必须要分配给一个线程专门与这个客户端的交互，去走后 面的流程

每一个连接从线程池中获取线程，省去了创建和销毁线程的开销



TCP 连接收到请求后,必须要分配给一个线程专门与这个客户端的交互

所以还会有个线程池,去走后面的流程。每一个连接从线程池中获取线程,省去了创建和销毁线程的开销

这些内容我们都归纳到MySQL的连接管理组件中

所以`连接管理的职责`是**负责认证**、管理连接、**获取权限信息**

1.4 第2层：服务层

- SQL Interface: SQL接口

  接收用户的SQL命令，并且返回用户需要查询的结果。比如SELECT ... FROM就是调用SQL Interface

  MySQL支持DML（数据操作语言）、DDL（数据定义语言）、存储过程、视图、触发器、自定 义函数等多种SQL语言接口

- Parser: 解析器

  在解析器中对 SQL 语句进行语法分析、语义分析

  将SQL语句分解成数据结构，并将这个结构 传递到后续步骤，以后SQL语句的传递和处理就是基于这个结构的

  如果在分解构成中遇到错 误，那么就说明这个SQL语句是不合理的

  在SQL命令传递到解析器的时候会被解析器验证和解析，并为其创建 语法树

  并根据数据字典`丰富查询语法树`，会`验证该客户端是否具有执行该查询的权限`

  创建好语法树后，MySQL还 会对SQl查询进行语法上的优化，进行查询重写

- Optimizer: 查询优化器

  SQL语句在语法解析之后、查询之前会使用查询优化器确定 SQL 语句的执行路径，生成一个`执行计划`

  这个执行计划表明应该`使用哪些索引`进行查询（全表检索还是使用索引检索），`表之间的连 接顺序`如何，最后会按照执行计划中的步骤调用存储引擎提供的方法来真正的执行查询，并将 查询结果返回给用户

  它使用`选取-投影-连接`"策略进行查询

  例如：

```
SELECT id,name FROM student WHERE gender = '女';
```

这个SELECT查询先根据WHERE语句进行 选取 ，而不是将表全部查询出来以后再进行gender过 滤

+这个SELECT查询先根据id和name进行属性 投影 ，而不是将属性全部取出以后再进行过 滤，将这两个查询条件 连接 起来生成最终查询结果

- Caches & Buffers： 查询缓存组件

  MySQL内部维持着一些Cache和Buffer，比如Query Cache用来缓存一条SELECT语句的执行结果，如果能够在其中找到对应的查询结果，那么就不必再进行查询解析、优化和执行的整个过 程了，直接将结果反馈给客户端

  这个缓存机制是由一系列小缓存组成的。比如表缓存，记录缓存，key缓存，权限缓存等 。 这个查询缓存可以在 不同客户端之间共享

  从MySQL 5.7.20开始，不推荐使用查询缓存，**并在 MySQL 8.0中删除**

1.5 第3层：引擎层

和其它数据库相比, MySQL有点与众不同,它的架构**可以在多种不同场景中应用并发挥良好作用**，`主要体现在存储引擎的架构上`

插件式的存储引擎架构将查询处理和其它的系统任务以及数据的存储提取相分离

这种架构可以**根据业务的需求和实际需要 选择合适的存储引擎**

同时开源的MySQL还 **允许开发人员设置自己的存储引擎**

这种高效的模块化架构为那些希望专门针对特定应用程序需求(例如数据仓库、事务处理或高可用性情况)的人提供了巨大的好处,同时享受使用一组独立于任何接口和服务的优势存储引擎

插件式存储引警层(Storage Engines),真正的负责了MySQL中数据的存储和提取,对物理服务器级别维护的底层数据执行操作,服务器通过API与存储引擎进行通信

不同的存储引擎具有的功能不同,这样我们可以根据自己的实际需要进行选取

MySQL 8.0.25默认支持的存储引擎如下:

```
SHOW ENGINES;
```

```
+--------------------+---------+----------------------------------------------------------------+--------------+------+------------+
| Engine             | Support | Comment                                                        | Transactions | XA   | Savepoints |
+--------------------+---------+----------------------------------------------------------------+--------------+------+------------+
| FEDERATED          | NO      | Federated MySQL storage engine                                 | NULL         | NULL | NULL       |
| MEMORY             | YES     | Hash based, stored in memory, useful for temporary tables      | NO           | NO   | NO         |
| InnoDB             | YES     | Supports transactions, row-level locking, and foreign keys     | YES          | YES  | YES        |
| PERFORMANCE_SCHEMA | YES     | Performance Schema                                             | NO           | NO   | NO         |
| MyISAM             | DEFAULT | MyISAM storage engine                                          | NO           | NO   | NO         |
| MRG_MYISAM         | YES     | Collection of identical MyISAM tables                          | NO           | NO   | NO         |
| BLACKHOLE          | YES     | /dev/null storage engine (anything you write to it disappears) | NO           | NO   | NO         |
| CSV                | YES     | CSV storage engine                                             | NO           | NO   | NO         |
| ARCHIVE            | YES     | Archive storage engine                                         | NO           | NO   | NO         |
+--------------------+---------+----------------------------------------------------------------+--------------+------+------------+
9 rows in set (0.01 sec)
```

1.6 存储层

所有的数据，数据库、表的定义，表的每一行的内容，索引，都是`存在 文件系统 上`，以`文件 的方式存 在的`，并完成与存储引擎的交互

当然有些存储引擎比如InnoDB，也支持不使用文件系统直接管理裸设 备，但现代文件系统的实现使得这样做没有必要了

在文件系统之下，可以使用本地磁盘，可以使用 DAS、NAS、SAN等各种存储系统

> /var/lib/ 目录下

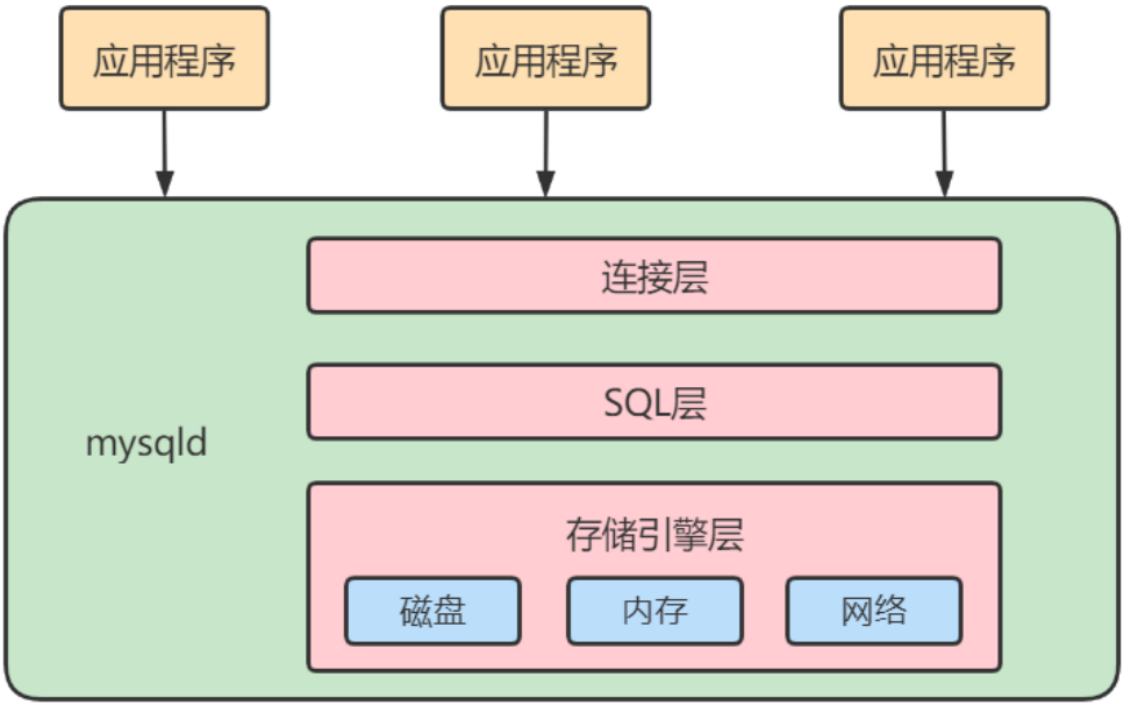
```
SHOW VARIABLES LIKE '%datadir%';

+---------------+-----------------+
| variable_name | value           |
+---------------+-----------------+
| datadir       | /var/lib/mysql/ |
+---------------+-----------------+
1 row in set (0.00 sec)
```

1.7 小结

MySQL架构图本节开篇所示

下面为了熟悉SQL执行流程方便，我们可以简化如下：

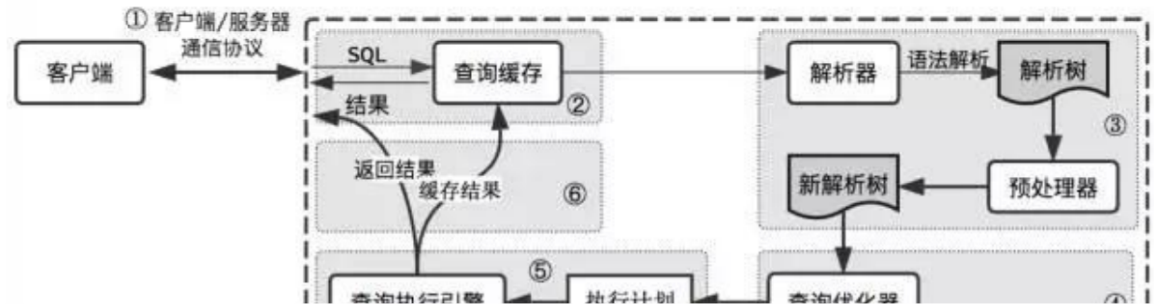


简化为三层结构：

1. 连接层：**客户端和服务器端建立连接**，客户端发送 SQL 至服务器端；
2. SQL 层（服务层）：**对 SQL 语句进行查询处理**；与数据库文件的存储方式无关；
3. 存储引擎层：与数据库文件打交道，**负责数据的存储和读取**

2. SQL执行流程

2.1 MySQL 中的 SQL执行流程



MySQL拿到一个查询请求后,会先到查询缓存看看,之前是不是执行过这条语句

之前执行过的语句及其结果可能会以key-value 对的形式,被直接缓存在内存中

key是查询的语句, value是查询的结果。如果你的查询能够直接在这个缓存中找到key,那么这个value就会被直接返回给客户端

如果语句不在查询缓存中,就会继续后面的执行阶段

执行完成后,执行结果会被存入查询缓存中

所以,如果查询命中缓存, MySQL不需要执行后面的复杂操作,就可以直接返回结果,这个效率会很高

MySQL的查询流程：

1、查询缓存：

Server 如果在查询缓存中发现了这条 SQL 语句，就会直接将结果返回给客户端；

如果没有，就进入到解析器阶段。需要说明的是，因为查询缓存往往效率不高，所以在 MySQL8.0 之后就抛弃 了这个功能。

大多数情况查询缓存就是个鸡肋，为什么呢?

举例：

```
SELECT employee_id,last_name FROM employees WHERE employee_id = 101;
```

查询缓存是提前把查询结果缓存起来，这样下次不需要执行就可以直接拿到结果

需要说明的是，在 MySQL 中的查询缓存，不是缓存查询计划，而是查询对应的结果

这就意味着查询匹配的`鲁棒性大大降 低`，只有 **相同的查询操作** `（两个相同的字符串）`才会命中查询缓存

1、两个查询请求在任何字符上的不同（例如：**空格**、注释、 大小写），都会导致缓存不会命中。因此 MySQL 的 查询缓存命中率不高

2、如果查询请求中包含**某些系统函数**、用户自定义变量和函数、一些系统表

如 mysql 、 information_schema、 performance_schema 数据库中的表，那这个请求就不会被缓存

以某些系统函数 举例，可能同样的函数的两次调用会产生不一样的结果，比如函数 NOW ，每次调用都会产生最新的当前 时间，如果在一个查询请求中调用 了这个函数，那即使查询请求的文本信息都一样，那不同时间的两次 查询也应该得到不同的结果，如果在第一次查询时就缓存了，那第二次查询的时候直接 使用第一次查询 的结果就是错误的!

3、此外，既然是缓存，那就有它 缓存失效的时候

MySQL的缓存系统会监测涉及到的每张表，只要该表的 结构或者数据被修改，如**对该表使用了 INSERT 、 UPDATE 、 DELETE 、 TRUNCATE TABLE 、 ALTER TABLE 、 DROP TABLE 或 DROP DATABASE 语句**，那使用该表的所有高速缓存查询都将变为无效并从高 速缓存中删除!

对于 更新压力大的数据库 来说，查询缓存的`命中率会非常低`

总之,因为查询缓存往往弊大于利,查询缓存的失效非常频繁

一般建议大家在静态表里使用查询缓存

什么叫静态表呢?就是一般我们极少更新的表

比如,一个系统配置表、字典表,这张表上的查询才适合使用查询缓存

好在MySQL也提供了这种“按需使用”的方式:

你可以将 `my.cnf` 参数 `query_cache_type` 设置成 `DEMAND`, 代表当sql语句中有**SQL_CACHE关键词时才缓存**

比如:

```
#query_cache_type有3个值 0 代表关闭查询缓存 OFF, 1代表开启ON 2 (DEMAND)

query_cache_type=2
```

这样对于默认的SQL语句都不使用查询缓存

而对于你确定要使用查询缓存的语句,可以用 `SQL_CACHE` 显式指定,像下面这个语句一样:

```
-- 先过查询缓存
select SQL_CACHE * from test where ID=5;

-- 不走查询缓存
select SQL_NO_CACHE * from test where ID=5;
```

查看当前mysql实例是否开启缓存机制:

```
-- MySQL5.7
show variables like '%query_cache_type%';

+------------------+-------+
| Variable_name    | Value |
+------------------+-------+
| query_cache_type | OFF   |
+------------------+-------+
1 row in set (0.01 sec)



-- MySQL8.0
show variables like '%query_cache_type%';

Empty set (0.00 sec)
```

2、解析器:

在解析器中对 SQL 语句进行语法分析、语义分析

如果没有命中查询缓存,就要开始真正执行语句了

首先,MySQL需要知道你要做什么,因此需要**对SQL语句做解析**

SQL语句的分析分为**词法分析**与**语法分析**

1、分析器先做“词法分析”

你输入的是由多个字符串和空格组成的一条SQL语句, MySQL需要识别出里面的 字符串 分别是 什么代表什么

MySQL 从你输入的"select"这个关键字识别出来,这是一个查询语句

它也要把字符串"T”识别成“表名T",把字符"串"ID"识别成“列ID”

2、做“语法分析”

根据词法分析的结果,语法分析器(比如:Bison)会根据语法规则,判断你输入的这个SQL语句是否满足 MySQL 语法

如果你的语句不对,就会收到 `You have an error in your SQL syntax` 的错误提醒

比如下面这个语句 from 写成了"rom"

```
#语句:
select * fro test where id=1;

#错误;
ERROR 1064 (42000): 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
'fro test where id=1' at line 1
```

如果SQL语句正确，则会生成一个这样的语法树：

```
SELECT username,ismale FROM userinfo
WHERE age>20 AND level>5 AND 1=1;
```

下图是SQL词法分析的过程步骤:

3、优化器：

在优化器中会确定 SQL 语句的执行路径，比如是根据 全表检索 ， 还是根据 索引检索 等

经过了解析器,MySQL就知道你要做什么了

在开始执行之前,还要先经过优化器的处理

一条查询可以有很多种执行方式,最后都返回相同的结果

优化器的作用就是**找到这其中最好的执行计划**

优化器优化后会生成一个执行计划

举例：

如下语句是执行两个表的 join：

```
select * from test1 join test2
using(ID) -- 使用id进行等值内连接
where test1.name='zhangwei' and test2.name='mysql高级课程';
```

方案1：

可以先从表 test1 里面取出 `name='zhangwei'` 的记录的 ID 值，再根据 ID 值关联到表 test2，再判 断 test2 里面 name的值是否等于 `'mysql高级课程'`

方案2：

可以先从表 test2 里面取出 `name='mysql高级课程'` 的记录的 ID 值，再根据 ID 值关联到 test1， 再判断 test1 里面 `name的值是否等于 zhangwei` 。

这两种执行方法的**逻辑结果是一样的**，但是执行的效率会有不同，而**优化器的作用就是决定选择使用哪一个方案**

优化 器阶段完成后，这个语句的执行方案就确定下来了，然后进入执行器阶段

如果你还有一些疑问，比如优化器是怎么选择索引的，有没有可能选择错等。后面讲到索引我们再谈

在查询优化器中，可以分为 `逻辑查询` 优化阶段和 `物理查询` 优化阶段

1、**逻辑查询优化**就是通过 `改变SQL语句的内容` 来使得SQL查询更高效,同时为 物理查询优化 提供 更多的 候选执行计划

通常采用的方式是对 **SQL语句进行等价变换** ,对查询进行重写,而查询重写的数学基础就是 **关系代数**

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。

2、**物理查询优化** 是基于 **关系代数** 进行的 查询重写 ,而关系代数的每一步都对应着 **物理计算** ,这些物理计算往往存在多种算法,因此需要计算 **各种物理路径** 的代价,从中选择 **代价最小的作为执行计划** 在这个阶段里,对于单表和多表连接的操作,需要 **高效地使用索引** ， 提升查询效率。

4、执行器：

截止到现在，还没有真正去读写真实的表，仅仅只是产出了一个执行计划。于是就进入了 执行器阶段

在执行之前需要判断该用户是否 具备权限

如果没有，就会返回权限错误

如果具备权限，就执行 SQL 查询并返回结果

在 MySQL8.0 以下的版本，如果设置了查询缓存，这时会将查询结果进行缓存

```
select * from test where id=1;
```

如果有权限,就打开表继续执行

打开表的时候,执行器就会根据表的引擎定义,调用存储引擎API对表进行的读写

存储引擎API只是抽象接口,下面还有个存储引擎层,具体实现还是要看表选择的存储引擎

比如：表 test 中，ID 字段没有索引，那么执行器的执行流程是这样的：

调用 InnoDB 引擎接口取这个表的第一行，判断 ID 值是不是1，如果不是则跳过，如果是则将这行存在结果集中；

调用引擎接口取“下一行”，重复相同的判断逻辑，直到取到这个表的最后一行

执行器将上述遍历过程中所有满足条件的行组成的记录集作为结果集返回给客户端

至此，这个语句就执行完成了

对于有索引的表，执行的逻辑也差不多

SQL 语句在 MySQL 中的流程是：`SQL语句→查询缓存→解析器→优化器→执行器`

2.2 MySQL8中SQL执行原理

前面的结构图很复杂,我们需要抓取最核心的部分: SQL的执行原理

不同的DBMS的 SQL的执行原理是相通的 ,只是在不同的软件中,各有各的实现路径

既然一条SQL语句会经历不同的模块,那我们就来看下,在不同的模块中,SQL执行所使用的资源(时间)是怎样的

如何在MySQL中对一条SQL语句的执行时间进行分析

1. 确认profiling 是否开启

```
-- 当前MySQL8.0
select version();
```

```
+-----------+
| version() |
+-----------+
| 8.0.25    |
+-----------+
1 row in set (0.00 sec)


-- 确认profiling 是否开启 (方式一)
select @@profiling; -- 系统变量 默认是SESSION级别

+-------------+
| @@profiling |
+-------------+
|           0 |
+-------------+
1 row in set, 1 warning (0.00 sec)



-- 确认profiling 是否开启 (方式二)
show variables like 'profiling';

+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| profiling     | OFF   |
+---------------+-------+
1 row in set (0.01 sec)
```

`开启profile`**（让系统记录一些指标信息，默认不记录）**

profiling=0 代表关闭，我们需要把 profiling 打开，即设置为 1：

```
set profiling=1;

Query OK, 0 rows affected, 1 warning (0.00 sec)
```

2. 多次执行相同SQL查询

然后我们执行一个 SQL 查询（你可以执行任何一个 SQL 查询）：

```
select * from employees; -- 两次

107 rows in set (0.00 sec)
```

3. 查看profiles 查看 **当前会话** 所产生的所有 profiles：

```
show profiles; # 显示最近的几次查询

+----------+------------+---------------------------+
| Query_ID | Duration   | Query                     |
+----------+------------+---------------------------+
|        1 | 0.00284525 | select * from employees   |
|        2 | 0.00760025 | 107 rows in set (0.00 sec) |
|        3 | 0.00115325 | select * from employees   |
+----------+------------+---------------------------+
3 rows in set, 1 warning (0.00 sec)
```

4. 查看profile

显示执行计划，查看程序的执行步骤：

```
 show profile;

 +--------------------------------+----------+
| Status                         | Duration |
+--------------------------------+----------+
| starting                       | 0.000062 |
| Executing hook on transaction  | 0.000003 |
| starting                       | 0.000009 |
| checking permissions           | 0.000006 | -- 权限检查
| Opening tables                 | 0.000035 | -- 打开表
| init                           | 0.000005 | -- 初始化
| System lock                    | 0.000008 | -- 所系统
| optimizing                     | 0.000004 | -- 优化查询
| statistics                     | 0.000016 | -- 统计
| preparing                      | 0.000015 | -- 准备
| executing                      | 0.000129 | -- 执行
| end                            | 0.000003 |
| query end                      | 0.000002 |
| waiting for handler commit     | 0.000006 |
| closing tables                 | 0.000006 |
| freeing items                  | 0.000290 |
| cleaning up                    | 0.000555 |
+--------------------------------+----------+
17 rows in set, 1 warning (0.00 sec)
```

当然你也可以查询指定的 Query ID **（查看某条语句的执行计划）**

比如：

```
+----------+------------+---------------------------+
| Query_ID | Duration   | Query                     |
+----------+------------+---------------------------+
|        1 | 0.00284525 | select * from employees   |
```

```
|        2 | 0.00760025 | 107 rows in set (0.00 sec) |
|        3 | 0.00115325 | select * from employees    |
+----------+------------+----------------------------+



show profile for query 3;

+--------------------------------+----------+
| Status                         | Duration |
+--------------------------------+----------+
| starting                       | 0.000062 |
| Executing hook on transaction  | 0.000003 |
| starting                       | 0.000009 |
| checking permissions           | 0.000006 |
| Opening tables                 | 0.000035 |
| init                           | 0.000005 |
| System lock                    | 0.000008 |
| optimizing                     | 0.000004 |
| statistics                     | 0.000016 |
| preparing                      | 0.000015 |
| executing                      | 0.000129 |
| end                            | 0.000003 |
| query end                      | 0.000002 |
| waiting for handler commit     | 0.000006 |
| closing tables                 | 0.000006 |
| freeing items                  | 0.000290 |
| cleaning up                    | 0.000555 |
+--------------------------------+----------+
17 rows in set, 1 warning (0.00 sec)
```

查询 SQL 的执行时间结果和上面（show profile）是一样的

此外，还可以查询更丰富的内容：

```
show profile cpu,block io for query 5;

+--------------------------------+----------+----------+------------+---------
-----+---------------+
| Status                         | Duration | CPU_user | CPU_system |
Block_ops_in | Block_ops_out |
+--------------------------------+----------+----------+------------+---------
-----+---------------+
| starting                       | 0.000104 | 0.000030 |   0.000068 |
   0 |             0 |
| Executing hook on transaction  | 0.000005 | 0.000001 |   0.000002 |
   0 |             0 |
| starting                       | 0.000008 | 0.000003 |   0.000006 |
   0 |             0 |
| checking permissions           | 0.000005 | 0.000001 |   0.000004 |
   0 |             0 |
| Opening tables                 | 0.000034 | 0.000011 |   0.000023 |
   0 |             0 |
```

```
| init                           | 0.000004 | 0.000001 |   0.000003 |
   0 |             0 |
| System lock                    | 0.000006 | 0.000002 |   0.000004 |
   0 |             0 |
| optimizing                     | 0.000034 | 0.000034 |   0.000000 |
   0 |             0 |
| statistics                     | 0.000046 | 0.000046 |   0.000000 |
   0 |             0 |
| preparing                      | 0.000009 | 0.000008 |   0.000000 |
   0 |             0 |
| executing                      | 0.000010 | 0.000010 |   0.000000 |
   0 |             0 |
| end                            | 0.000002 | 0.000002 |   0.000000 |
   0 |             0 |
| query end                      | 0.000002 | 0.000002 |   0.000000 |
   0 |             0 |
| waiting for handler commit     | 0.000007 | 0.000007 |   0.000000 |
   0 |             0 |
| closing tables                 | 0.000006 | 0.000006 |   0.000000 |
   0 |             0 |
| freeing items                  | 0.000086 | 0.000087 |   0.000000 |
   0 |             0 |
| cleaning up                    | 0.000009 | 0.000008 |   0.000000 |
   0 |             0 |
+--------------------------------+----------+----------+------------+---------
-----+--------------+
17 rows in set, 1 warning (0.00 sec)
```

除了查看`cpu`,`io`阻塞等参数情况,还可以查询下列参数的利用情况

```
SHOW PROFILE [type [,type] ...)
[FOR QUERY n]
[LIMIT row_count [OFFSET offset]]
```

type:

`ALL`：显示所有参数的开销信息

`BLOCK IO`： 显示IO的相关开销

`CONTEXT SWITCHES`： 上下文切换相关开销

`CPU`： 显示CPU相关开销信息

`IPC`： 显示发送和接收相关开销信息

`MEMORY`： 显示内存相关开销信息

`PAGE FAULTS`：显示页面错误相关开销信息

`SOURCE`： 显示和Source_function, Source_file, Source_line 相关的开销信息

`SWAPS`：显示交换次数相关的开销信息

2.3 MySQL5.7中SQL执行原理

> 下列操作在MySQL5.7中测试

这里我们需要 显式开启查询缓存模式 。在MySQL5.7中如下设置：

> MySQL5.7配置文件中开启查询缓存 查看执行计划

修改配置文件

> vim /etc/my.cnf

添加：

```
query_cache_type=1
```

重启服务：

```
systemctl restart mysqld
```

查看 查询缓存是否开启：

```
show variables like 'query_cache_type';

+------------------+-------+
| Variable_name    | Value |
+------------------+-------+
| query_cache_type | ON    |
+------------------+-------+
1 row in set (0.00 sec)
```

开启 查询执行计划

```
set profiling=1;

Query OK, 0 rows affected, 1 warning (0.03 sec)
```

走缓存的情况：

```
-- 显示最近的几次查询
show profiles;

+----------+------------+--------------------------+
| Query_ID | Duration   | Query                    |
+----------+------------+--------------------------+
|        1 | 0.00985375 | select database()        |
|        2 | 0.00695000 | show databases           |
|        3 | 0.00014225 | SELECT DATABASE()        |
|        4 | 0.00017675 | show databases           |
|        5 | 0.00008550 | show tables              |
|        6 | 0.00021600 | show tables              |
|        7 | 0.00333350 | select * from departments |
|        8 | 0.00006775 | select * from departments |
+----------+------------+--------------------------+
8 rows in set, 1 warning (0.00 sec)




-- 查看第七条查询语句的执行计划
 show profile for query 7;
+--------------------------------+----------+
| Status                         | Duration |
+--------------------------------+----------+
| starting                       | 0.000031 |
| Waiting for query cache lock   | 0.000003 |
| starting                       | 0.000001 |
| checking query cache for query | 0.000196 |
| checking permissions           | 0.000024 |
| Opening tables                 | 0.000022 |
| init                           | 0.000041 |
| System lock                    | 0.000019 |
| Waiting for query cache lock   | 0.000003 |
| System lock                    | 0.000027 |
| optimizing                     | 0.000003 |
| statistics                     | 0.000014 |
| preparing                      | 0.000010 |
| executing                      | 0.000002 |
| Sending data                   | 0.002876 |
| end                            | 0.000004 |
| query end                      | 0.000006 |
| closing tables                 | 0.000005 |
| freeing items                  | 0.000004 |
| Waiting for query cache lock   | 0.000001 |
| freeing items                  | 0.000007 |
| Waiting for query cache lock   | 0.000001 |
| freeing items                  | 0.000001 |
| storing result in query cache  | 0.000001 | -- 把结果缓存起来
| cleaning up                    | 0.000035 |
+--------------------------------+----------+
25 rows in set, 1 warning (0.00 sec)
```

```
-- 查看第8条查询语句的执行计划
SHOW PROFILE FOR QUERY 8;

+------------------------------+----------+
| Status                       | Duration |
+------------------------------+----------+
| starting                     | 0.000029 |
| Waiting for query cache lock | 0.000002 |
| starting                     | 0.000001 |
| checking query cache for query | 0.000007 |
| checking privileges on cached | 0.000003 |
| checking permissions         | 0.000012 |
| sending cached result to clien | 0.000010 |
| cleaning up                  | 0.000004 |
+------------------------------+----------+
8 rows in set, 1 warning (0.00 sec)
```

验证查询语句不一致，导致命中失败

```
-- 执行语句两次：（意思一样，但字符串不一样）

select * from departments where department_id=10;
+---------------+-----------------+------------+-------------+
| department_id | department_name | manager_id | location_id |
+---------------+-----------------+------------+-------------+
|            10 | Administration  |        200 |        1700 |
+---------------+-----------------+------------+-------------+
1 row in set (0.01 sec)



select * from departments where department_id=    10;
+---------------+-----------------+------------+-------------+
| department_id | department_name | manager_id | location_id |
+---------------+-----------------+------------+-------------+
|            10 | Administration  |        200 |        1700 |
+---------------+-----------------+------------+-------------+
1 row in set (0.00 sec)



-- 查询最近的查询语句
```

```
SHOW PROFILES;

 SHOW PROFILES;
+----------+------------+-----------------------------------------------------
-+
| Query_ID | Duration   | Query
 |
+----------+------------+-----------------------------------------------------
-+
|        1 | 0.00284525 | select * from employees
 |
|        2 | 0.00760025 | 107 rows in set (0.00 sec)
|
|        3 | 0.00115325 | select * from employees
 |
|        4 | 0.01454600 | select * from departments where department_id=10
|
|        5 | 0.00037550 | select * from departments where department_id=    10
|
|        6 | 0.00030775 | x show profile cpu,block io for query 6
 |
+----------+------------+-----------------------------------------------------
-+
6 rows in set, 1 warning (0.00 sec)




-- 查看第五条语句
show profile for query 5;

+--------------------------------+----------+
| Status                         | Duration |
+--------------------------------+----------+
| starting                       | 0.000104 |
| Executing hook on transaction  | 0.000005 |
| starting                       | 0.000008 |
| checking permissions           | 0.000005 |
| Opening tables                 | 0.000034 |
| init                           | 0.000004 |
| System lock                    | 0.000006 |
| optimizing                     | 0.000034 |
| statistics                     | 0.000046 |
| preparing                      | 0.000009 |
| executing                      | 0.000010 |
| end                            | 0.000002 |
| query end                      | 0.000002 |
| waiting for handler commit     | 0.000007 |
| closing tables                 | 0.000006 |
| freeing items                  | 0.000086 |
| cleaning up                    | 0.000009 |
+--------------------------------+----------+
```

```
17 rows in set, 1 warning (0.00 sec)



-- 查看第4条语句
show profile for query 4;

+--------------------------------+----------+
| Status                         | Duration |
+--------------------------------+----------+
| starting                       | 0.004945 |
| Executing hook on transaction  | 0.000027 |
| starting                       | 0.000039 |
| checking permissions           | 0.000072 |
| Opening tables                 | 0.004444 |
| init                           | 0.000015 |
| System lock                    | 0.000046 |
| optimizing                     | 0.000244 |
| statistics                     | 0.004020 |
| preparing                      | 0.000103 |
| executing                      | 0.000076 |
| end                            | 0.000003 |
| query end                      | 0.000003 |
| waiting for handler commit     | 0.000064 |
| closing tables                 | 0.000015 |
| freeing items                  | 0.000127 |
| cleaning up                    | 0.000307 |
+--------------------------------+----------+
17 rows in set, 1 warning (0.00 sec)
```

发现前后两次相同的sql语句，执行的查询过程仍然是相同的。说明没有缓存之说

在`8.0版本`之后,MySQL `不再支持缓存的查询`

因为一旦数据表有更新,缓存都将清空； 因此只有数据表是静态的时候；或者数据表很少发生变化时,使用缓存查询才有价值,否则如果数据表经常更新,反而增加了SQL的查询时间。

结论：字符串不一致，就不走缓存

2.4 SQL语法顺序

需求：查询每个部门年龄高于20岁的人数且高于20岁人数不能少于2人，显示人数最多的第一名部门信息

```
SELECT d.department_id,COUNT(employee_id)
    FROM departments d LEFT JOIN employees e
    ON d.department_id = e.department_id
    WHERE e.salary>3000
GROUP BY department_id
HAVING COUNT(employee_id)>2
    ORDER BY COUNT(employee_id) DESC
    LIMIT 1;
```

2.5 Oracle中的SQL执行流程(了解)

Oracle 中采用了 共享池 来判断 SQL 语句是否存在缓存和执行计划，通过这一步骤我们可以知道应该采用 硬解析还是软解析。 我们先来看下 SQL 在 Oracle 中的执行过程：

从上面这张图中可以看出，SQL 语句在 Oracle 中经历了以下的几个步骤

1.语法检查：检查 SQL 拼写是否正确，如果不正确，Oracle 会报语法错误。

2.语义检查：检查 SQL 中的访问对象是否存在。比如我们在写 SELECT 语句的时候，列名写错了，系统就会提示错误。语法检查和语义检查的作用是保证 SQL 语句没有错误。

3.权限检查：看用户是否具备访问该数据的权限。

4.共享池检查：共享池（Shared Pool）是一块内存池，最主要的作用是缓存 SQL 语句和该语句的执行计 划。Oracle 通过检查共享池是否存在 SQL 语句的执行计划，来判断进行软解析，还是硬解析

那软解析 和硬解析又该怎么理解呢?

在共享池中，Oracle 首先对 SQL 语句进行 Hash 运算，然后根据 Hash 值在库缓存（Library Cache）中 查找，如果 存在 SQL 语句的执行计划，就直接拿来执行，直接进入“执行器”的环节，这就是`软解析`

如果没有找到 SQL 语句和执行计划，Oracle 就需要`创建解析树`进行解析，生成执行计划，进入“优化器” 这个步骤，这就是 硬解析

5 .优化器：优化器中就是要进行硬解析，也就是`决定怎么做`，比如创建解析树，生成执行计划。

6.执行器：当有了解析树和执行计划之后，就知道了 SQL 该怎么被执行，这样就可以在执行器中执 行语句了

`共享池`是`Oracle`中的术语，包括了`库缓存`，`数据字典缓冲区`等。

我们上面已经讲到了`库缓存区`，它主要`缓存 SQL 语句`和`执行计划`

而`数据字典缓冲区`存储的是 Oracle 中的对象定义，比如`表、视图、索引等对象`

当对 SQL 语句进行解析的时候，如果需要相关的数据，会从数据字典缓冲区中提取

库缓存 这一个步骤，决定了 SQL 语句是否需要进行硬解析。为了提升 SQL 的执行效率，我们应该尽量避免硬解析，因为在 SQL 的执行过程中，创建解析树，生成执行计划是很消耗资源的

你可能会问，如何避免硬解析，尽量使用软解析呢?

在 Oracle 中， `绑定变量` 是它的一大特色

举个例子，我们可以使用下面的查询语句:

```
select * from player where player_id = 10001;


-- 等价替换
select * from player where player_id = :player_id;
```

这两个查询语句的效率在 Oracle 中是完全不同的

如果你在查询 player_id = 10001 之后，还会查询 10002、10003 之类的数据，那么每一次查询都会创建一个新的查询解析

而第二种方式使用了绑定变 量，那么在第一次查询之后，在 `共享池` 中就会存在这类查询的 `执行计划` ，也就是 `软解析`

因此，我们可以通过使用绑定变量来减少硬解析，减少 Oracle 的解析工作量。但是这种方式也有缺点，使用动态 SQL 的方式，因为参数不同，会导致 SQL 的执行效率不同不足之处在于可能会导致生成的执行计划不够优化;，同时 SQL 优化也会比较困难

因此是否需要绑定变量还需要视情况 而定

Oracle的架构图:

简图:

小结:

Oracle 和 MySQL 在进行 SQL 的查询上面有软件实现层面的差异。Oracle 提出了共享池的概念，通过共享 池来判断是进行软解析，还是硬解析

3. 数据库缓冲池(buffer pool)

InnoDB 存储引擎 是以 页为单位 来 管理存储空间 的，我们进行的 增删改查操作 其实本质上都是在访问 页 面 （包括读页面、写页面、创建新页面等操作）

而 磁盘 I/O 需要消耗的时间很多，而在内存中进行操 作，效率则会高很多，为了能让数据表或者索引中的数据随时被我们所用

DBMS 会申请 占用内存 来作为 数据缓冲池 ，在真正访问页面之前，需要把在磁盘上的页缓存到 内存中的 Buffer Pool 之后才可以访 问

这样做的好处是可以让 磁盘活动最小化 ，从而 减少与磁盘直接进行 I/O 的时间

要知道，这种策略对提 升 SQL 语句的查询性能来说至关重要。如果索引的数据在缓冲池里，那么访问的成本就会降低很多

3.1 缓冲池 vs 查询缓存

1 . 缓冲池（Buffer Pool） 首先我们需要了解在 InnoDB 存储引擎 中，缓冲池都包括了哪些？

在 InnoDB 存储引擎 中有 一部分数据会放到内存 中， 缓冲池 则占了这部分内存的 大部分 ，它用来存储各种 数据的缓存

如下图所示：

从图中，你能看到 InnoDB 缓冲池 包括了 数据页、索引页、插入缓冲、锁信息、自适应 Hash 和数据字典信息 等

缓存池的重要性：

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

但是各位也都知道,磁盘的速度慢的跟乌龟一样,怎么能配得上“快如风,疾如电”的CPU呢?

这里， 缓冲池 可以帮助我们消除 CPU 和 磁盘 之间的鸿沟

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

缓存原则:

`位置 * 频次` 这个原则，可以帮我们对 I/O 访问效率进行优化

首先，位置决定效率，提供缓冲池就是为了在内存中可以直接访问数据

其次，频次决定优先级顺序。因为缓冲池的大小是有限的，比如磁盘有 200G，但是内存只有 16G，缓冲 池大小只有 1G，就无法将所有数据都加载到缓冲池里，这时就涉及到优先级顺序，会 `优先对使用频次高 的热数据` 进行加载

缓冲池的预读特性:

了解了缓冲池的作用之后,我们还需要了解缓冲池的另一个特性: 预读

缓冲池的作用就是提升I/0效率,而我们进行读取数据的时候存在一个 `局部性原理` ,也就是说我们使用了一些数据,大概率还会使用它周围的一些数据,因此采用 `“预读”的机制提前加载,可以 减少 未来可能的磁盘1/0` 操作

2. 查询缓存

那么什么是查询缓存呢?

查询缓存是提前把 `查询结果缓存 起来` ，这样下次不需要执行就可以直接拿到结果

需要说明的是，在 `MySQL 中的查询缓存` ， `不是缓存查询计划` （Oracle缓存执行计划），而是 `缓存查询对应的结果`

因为命中条件苛刻（字符串不是百分比匹配就不命中），而 **且 只要数据表 发生变化**，查询缓存就会失效，因此命中率低

缓冲池和查询缓存是一个东西吗?

不是，一个是在内存中缓存一条查询语句的结果；一个是将一页一页的数据放到缓存中；

3.2 缓冲池如何读取数据

`缓冲池管理器` 会尽量将经常使用的数据保存起来，在 `数据库进行 页面 读` 操 作的时候，首先会判断该页面 是否在缓冲池中，如果存在就直接读取，如果不存在，就会通过内存或磁盘将页面存放到缓冲池中再进 行读取

缓存在数据库中的结构和作用如下图所示:

如果我们执行 SQL 语句的时候更新了缓存池中的数据，那么这些数据会马上同步到磁盘上吗?

实际上,当我们对数据库中的记录进行修改的时候, 首先 会 修改 缓冲池中 **页** 里面的 记录信息 ,然后**数据库**会以 **一定的频率** 刷新到磁盘上（刷盘）

注意并不是每次发生更新操作,都会立刻进行磁盘回写。缓冲池会采用一种叫做 checkpoint 的机制 将数据回写到磁盘上

这样做的好处就是 **提升** 了 数据库的整体性能

比如,当缓冲池不够用时,需要释放掉一些不常用的页,此时就可以强行采用checkpoint的方式,将不常用的 脏页 回 写到磁盘上 ,然后再从缓冲池中将这些 **页** 释放掉

这里 脏页 (dirty page)指的是 缓冲池中被修改过的页 ， 与 磁盘上的数据页 **不一致**

3.3 查看/设置缓冲池的大小

如果你使用的是MySQL MyISAM存储引擎 ,它只缓存索引,不缓存数据,对应的键缓存参数为 key_buffer_size ,你可以用它进行查看。

如果你使用的是 InnoDB 存储引擎，可以通过查看 innodb_buffer_pool_size 变量来查看缓冲池的大 小。命令如下：

```
show variables like 'innodb_buffer_pool_size';

+-------------------------+-----------+
| Variable_name           | Value     |
+-------------------------+-----------+
| innodb_buffer_pool_size | 134217728 | -- 128兆
+-------------------------+-----------+
1 row in set (0.03 sec)
```

你能看到此时 InnoDB 的缓冲池大小只有 134217728/1024/1024=128MB

我们可以修改缓冲池大小，比如 改为256MB，方法如下：

```
-- 全局范围修改缓冲持大小 （重启失效）
set global innodb_buffer_pool_size = 268435456;

Query OK, 0 rows affected (0.00 sec)
```

也可以修改配置文件

> vim /etc/my.cnf

```
[server]
innodb_buffer_pool_size = 268435456
```

查看缓冲池大小：

```
-- 已修改
show variables like 'innodb_buffer_pool_size';

+-------------------------+-----------+
| Variable_name           | Value     |
+-------------------------+-----------+
| innodb_buffer_pool_size | 268435456 |
+-------------------------+-----------+
1 row in set (0.00 sec)
```

3.4 多个Buffer Pool实例

`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中的数据` 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

比方说这样:

```
[server]
innodb_buffer_pool_instances = 2
```

这样就表明我们要创建2个 Buffer Pool 实例

我们看下如何查看缓冲池的个数，使用命令：

```
-- 默认情况下 缓冲池个数是1
show variables like 'innodb_buffer_pool_instances';

+------------------------------+-------+
| Variable_name                | Value |
+------------------------------+-------+
| innodb_buffer_pool_instances | 1     |
+------------------------------+-------+
1 row in set (0.00 sec)
```

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 `缓冲池总共的大小` 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,

值得注意的是：

nnoDB规定：当 `innodb_buffer_pool_size` 的值小于 `1G` 的时候 **设置多个实例是无效** 的,

nnoDB会默认把 `innodb_buffer_pool_instances` 的值修改为1

而我们鼓励在 Buffer Pool `大于` 或 `等于` 1G 的时候 `设置多个Buffer Pool实例`

3.5 引申问题

Buffer Pool是MySQL内存结构中十分核心的一个组成，你可以先把它想象成一个黑盒子

黑盒下的更新数据流程

当我们查询数据的时候,会 `先去Buffer Pool中查询`

如果 `Buffer Pool中不存在` ,存储引擎 `会先将数据从磁盘加载到Buffer Pool` 中,然后 `将数据返回给客户端` ；

同理,当我们 `更新某个数据` 的时候,如果这个 `数据不存在于BufferPoo` ,同样会 `先数据加载进来缓冲池（内存）` ,然后 `修改修改内存的数据`

被修改过的数据会在 `之后` `统一` `刷入磁盘`

这个过程看似没啥问题,实则是有问题的

假设我们 **修改Buffer Pool中的数据成功** ,但是 **还没来得及将数据刷入磁盘** **MySQL就挂** 了怎么办?

按照上图的逻辑,此时 **更新之后的数据只存在于Buffer Pool** 中,如果 此时MySQL启机了 ,这部分数据将会永久地丢失;

再者,我 **更新到一半突然发生错误** 了,想要 **回滚到更新之前的版本** , 该怎么办?

连 **数据持久化的保证** 、 **事务回滚** 都 **做不到** 还谈什么 **崩溃恢复** ?

答案: `Redo Log` & `Undo Log`