MySQL 笔记

Pionpill [1]

本文档为作者学习 MySQL 时的笔记。

2022 年 12 月 13 日

[1]笔名：北岸，电子邮件：673486387@qq.com，Github：https://github.com/Pionpill

前言：

笔者为软件工程系在校本科生，有计算机学科理论基础 (操作系统，数据结构，计算机网络，编译原理等)，本人在撰写此笔记时已经学过基础的数据库概念，类似于表，行，主键等概念不再赘述。

在指令中出现的 <> 表示必要参数，[] 表示可选参数。一般的，做前端开发或者小厂开发看完基础部分即可。进阶部分适合中大厂的开发人员以及高级开发人员。

本文分为以下几个部分:

- MySQL 基础: 主要参考《MySQL 必知必会》[1]一书，快速过一遍 MySQL 基础语法，一些样例数据下载方式如下:
  - `https://forta.com/wp-content/uploads/books/0672327120/mysql_scripts.zip`
- MySQL 进阶: 主要参考《高性能 MySQL》[2]。讲解了内部原理及优化方法。
  - 这是一本面向 DBA 的专业书籍，部分和后端无关的内容被我省略了。

本人的编写及开发环境如下：

- OS: Windows11
- MySQL: 8.0.3

本文默认使用 MySQL5 及以上版本，远古版本不再提及。

2022 年 12 月 13 日

[1]《MySQL Crash Course》: [英] Ben Forta 2009 年第一版。

[2]High Performance MySQL(3rd): Baron Schwartz, Peter Zaitsev，Vadim Tkachenko. 2013 年第三版

目录

第一部分

MySQL 基础

I 介绍与基本操作

1 MySQL 简介

1.1 安装

MySQL 有两种安装方式，一种是线上安装，一种是安装包安装。线上安装比较适合对 MySQL 熟悉的用户，可以自定义安装一些应用功能，但如果一次安装错了，后果很严重 (亲身体会)。这里建议新手使用安装包安装的方式。MySQL8 社区版的安装可以参考这篇文章: `https://blog.csdn.net/weixin_43605266/article/details/110477391`。

在 Window 上安装好 MySQL8 后，我们通过终端 `mysql` 命令进入数据库，可能会报如下错误:

```
ERROR 1045 (28000): Access denied for user 'ODBC'@'localhost' (using password: NO)
```

这是因为 Windows 默认进入 SQL 的用户为 ODBC，但我们没有创建 ODBC 用户，使用如下指令进入 MySQL 即可:

```
mysql -u <userName> -p
```

注意安装好后，本地服务器名: localhost (基本所有数据库系统的本地数据库服务器都是这个名字)，端口 3306 (MySQL 默认端口号)。

1.2 常用命令

为了更好的操作 MySQL，这里提前给出一些常用的命令。首先是在终端操作 MySQL 的一些命令:

- mysql 服务器的启用与关闭 (Windows 默认开启)。

```
net stop mysql # 停止 mysql 服务
net start mysql # 启用 mysql 服务
```

- 登录 mysql 服务器。如果是远程链接，需要参数 -h。

```
mysql [-h <hostIP>] -u <userName> -p [<password>]
```

其次是 MySQL 内部的几个常用命令 (注意 mysql 所有命令都需要; 结尾，sql 语句不区分大小写，但关键字一般使用大写)。

- 增加新用户。

```
CREATE USER '<userName>'@'LOCALHOST' IDENTIFIED BY '<password>';
```

- 修改用户密码。

```
ALTER USER '<userName>'@'LOCALHOST' IDENTIFIED WITH MYSQL_NATIVE_PASSWORD BY
    '<newPassword>';
```

- 显示所有数据库。

```
show databases;
```

- 显示用户信息 (注意 mysql 会对用户密码进行加密，所以无法获取密码，只能获取 user,host 等信息)。

```
SELECT <infos> FROM mysql.user;
```

1.3 GUI 应用

MySQL 本体的 GUI 工具不是很好用，推荐两个好用的数据库 GUI 软件:

- Navicat: 主流的 GUI 工具，公司一般都会用这个，费用昂贵。
  - 官网: `https://navicat.com.cn/`
- Dbeaver: 社区版免费，功能齐全，但有点卡。
  - 官网: `https://dbeaver.io/`
- VSCode-MySQL 插件: 全宇宙最好用的文本编辑器和最好的生态。

不使用这些辅助工具也完全没问题，只是在终端中写命令不是很方便。

2 数据库的基本操作

2.1 数据库的创建与删除

SHOW 语句

进入数据库后，我们需要选择我们要操作的数据库，在这之前我们需要查看已有的数据库:

```
SHOW databases;
```

类似的，还有如下常用的 SHOW 语句:

```
SHOW TABLES;   -- 显示数据库下所有的表
SHOW COLUMNS FROM <tableName>; -- 显示表的所有列信息
```

创建与删除数据库

创建数据库指令

```
CREATE DATABASE <databaseName>;
```

删除数据库指令

```
DROP DATABASE <databaseName>;
```

HELP 语句

与 Linux 操作系统类似，MySQL 提供了 HELP 语句帮助我们查询一些命令的使用方式。

```
HELP CREATE;   -- 查询 CREATE 相关的命令
HELP CREATE DATABASE; -- 查询该语句的使用方式
```

如果你使用的是命令行，可以通过如下指令查看当前正在使用的数据库:

```
select database();
```

II 基础数据操作语言

3 DML 数据操作语言 (增删改查)

数据库操作语言，主要针对表中数据操作。是最常用的数据库操作语言，它包括以下几个主要语句关键字:

- SELECT: 查找/检索记录。
- INSERT: 插入记录。
- UPDATE: 修改记录。
- DELETE: 删除记录。

其中 SELECT 即查数据，高级一点叫检索数据。毋庸置疑 SELECT 语句是 SQL 语句中最复杂，最常用的。本章的 1-11 节都在介绍 SELECT 语句。

3.1 SELECT 语句

3.1.1 SELECT 检索数据

单表查询非常简单，这里只给出一些例子，不做解释:

```
SELECT * FROM mysql.user;     -- 检索全部列
SELECT user FROM mysql.user; -- 检索单个指定列
SELECT user,host FROM mysql.user; -- 检索多个指定列
```

3.1.2 DISTINCT 返回不同的值

如果我们执行下面语句:

```
USE mysql;
SELECT host FROM user;
```

会返回多个值相同的视图，如果我们只想让重复的值出现一次，可以加上关键字: DISTINCT

```
SELECT DISTINCT host from user;
```

注意，DISTINCT 关键字会作用于所有的列，可以尝试下列语句观察结果。

```
SELECT DISTINCT host,user from user;
```

3.1.3 LIMIT 限制显示数量

默认的 SELECT 语句会返回所有行，有时候数据太多会很卡，LIMIT 关键字将限制返回的行数量:

```
SELECT * FROM information_schema.plugins LIMIT 10;
```

在终端中这只会显示前 10 行，如果需要返回指定的行数 (11-20)，可以这样写:

```
SELECT * FROM information_schema.plugins LIMIT 10,10;
```

> **注意 3.1.** 这里 *LIMIT 10,10* 表示的是从第 *10* 行开始的 *10* 行，也即 *11-20* 行。*MySQL* 还支持另一种写法 *4 OFFSET 3*，表示从第 *3* 行开始的 *4* 行。和 *LIMIT 3,4* 作用一样。

完全限定名

下面语句的作用是一样的:

```
-- 语句 1
USE mysql;
SELECT host FROM user;
-- 语句 2
SELECT user.host FROM mysql.user;
```

完全限定名更显示地指出了具体数据库，数据表，字段的信息。

3.2 数据排序

> **注意 3.2.** 子句: *SQL* 语句由子句构成，有些子句是必须的，有些的可选的。比如 *SELECT* 语句有一个必须的子句 *FROM*。

3.2.1 ORDER BY 排序

ORDER BY 子句取一个或多个列的名字，并据此进行排序。

```
SELECT * FROM mysql.help_keyword ORDER BY help_keyword_id LIMIT 10;
```

如果有必须要的话，也可以按多个列排序。

> **注意 3.3.** 通常，*ORDER BY* 子句中使用的列将是为显示所选择的列。但是，实际上并不一定要这样，用非检索的列排序数据是完全合法的。

默认的排序方式为升序排序 (A-Z), 同样可以指定降序排序。两种排序的关键字分别为:

- ASC: 升序排序。
- DESC: 降序排序。

关键字跟在列名后面，且仅对改列有效:

```
SELECT prod_id, prod_price, prod_name
    FROM products
    ORDER BY prod_price DESC, prod_name;
```

上述例子表示从 products 表中获取数据，按 prod_price 降序排序，若相同，再按 prod_name 升序排序。

3.3 数据过滤

3.3.1 WHERE 过滤

WHERE 子句可以对搜索条件进行过滤:

```
SELECT user,host FROM mysql.user WHERE user='root';
```

基本的 WHERE 子句操作符有:

表 2.1 WHERE 子句操作符

操作符	说明	操作符	说明	操作符	说明
=	等于	<>/!=	不等于	<	小于
<=	小于等于	>	大于	>=	大于等于

如果是对数值进行过滤，直接填写相应的值即可:

```
SELECT * FROM product WHERE price=10;
```

如果是对字符串进行过滤，则需要加上单引号，以表示这是字符串:

```
SELECT * FROM product WHERE produce_name='rice';
```

此外还有一个常用的范围比较操作符: BETWEEN, 用法如下:

```
SELECT prod_name, prod_price
    FROM products
    WHERE prod_price BETWEEN 5 AND 10;
```

数据库中有一个特殊的值: NULL，对他进行过滤的时候，需要特殊的子句 IS NULL。

```
SELECT produce_name FROM produces WHERE prod_price IS NULL;
```

3.3.2 AND/OR 组合操作符

上述 WHERE 语句比较简单，只能对单一条件进行限定，为了更强的过滤控制，MYSQL 允许给出多个 WHERE 子句: 通过 AND / OR 操作符。

这两个操作符意思非常简单，就和 JAVA 中的 && 和 || 类似。

```sql
-- AND 操作符
SELECT prod_id, prod_price, prod_name
    FROM products
    WHERE vend_id = 1003 AND prod_price <= 10;
-- OR 操作符
SELECT prod_name, prod_price
    FROM products
    WHERE vend_id = 1002 OR vend_id = 1003;
```

类似 JAVA，这两个操作符可以组合使用，组合过程中一定要使用 () 以表示清晰的组合逻辑。

```sql
SELECT prod_name, prod_price
    FROM products
    WHERE (vend_id = 1002 OR vend_id = 1003) AND prod_price >= 10;
```

3.3.3 IN/NOT 操作符

IN 与圆括号的组合，可以更方便地指定离散条件:

```sql
SELECT prod_name, prod_price
    FROM products
    WHERE vend_id IN (1002,1003)
    ORDER BY prod_name;
```

可以发现，IN 和 OR 操作符的作用是相同的，但 IN 有以下几个优点:

- IN 操作符更直观，计算次序更容易管理。
- IN 操作符一般比 OR 操作符执行速度更快。
- IN 可以包含其他 SELECT 语句。

NOT 操作符只有一个功能，否定后面跟的任何条件，类似于 Java 中的!。

```sql
SELECT prod_name, prod_price
    FROM products
    WHERE vend_id NOT IN (1002,1003)
    ORDER BY prod_name;
```

3.3.4 LIKE 操作符

通配符: 用来匹配值的一部分的特殊字符。

前面介绍的所有操作符都是针对已知值进行过滤的。LIKE 通配符类似于简单的正则表达式，可以进行模糊匹配。

LIKE 操作符主要有两个常用的通配符:

- %: 匹配任何字符出现任意次数 (0 个也算)。
- _: 匹配单个字符。

% 通配符的使用:

```
-- 产品名以 jet 开头的产品数据
SELECT prod_id, prod_name
    FROM products
    WHERE prod_name LIKE 'jet%';
-- 产品名中包含 anvil 字符串的产品数据
SELECT prod_id, prod_name
    FROM products
    WHERE prod_name LIKE '%anvil%';
```

虽然% 通配符可以匹配任何东西，但有一个例外，即 NULL。

_ 通配符的使用:

```
SELECT prod_id, prod_name
    FROM products
    WHERE prod_name LIKE '_ ton anvil';
```

尽管通配符非常好用，但它的搜索速度远不及之前提到的几种过滤方式。而且将通配符放在最前面的搜索速度是极慢的。

3.3.5 正则表达式

上文说过，LIKE 类似于简单的正则表达式，这里介绍正则表达式在 MySQL 中的使用方式。正则表达式的语法比较复杂，MySQL 也只实现了一部分的正则匹配，完整的正则表达式语法请参考其他文章。

在 WHERE 子句中使用正则表达式的关键字为 REGEXP(Regular Expression)。注意 LIKE 和 REGEX 的区别，LIKE 匹配的是整个文本，REGEX 匹配的是文本中的值。

```
-- 文本包含 1000 的所有行
SELECT prod_name
    FROM products
    WHERE prod_name REGEXP '1000'
    ORDER BY prod_name;
-- 文本为 1000 的所有行
SELECT prod_name
    FROM products
    WHERE prod_name LIKE '1000'
    ORDER BY prod_name;
```

当然，使用 $ 定位符，REGEX 可以实现 LIKE 的效果。

下面再给出几个例子:

```
-- OR 匹配: 文本中出现 1000 或 2000 的数据
SELECT prod_name
    FROM products
    WHERE prod_name REGEXP '1000|2000'
    ORDER BY prod_name;
-- [] 出现中括号内的字符
```

```
SELECT prod_name
    FROM products
    WHERE prod_name REGEXP '[123] Ton'
    ORDER BY prod_name;
```

类似的用法还有很多，需要对正则表达式有一定了解。

此外，正则表达式会使用一些特定字符，比如?。如果我们要查询这些字符，使用两个反斜杠进行转义即可:

```
SELECT vend_name
    FROM vendors
    WHERE vend_name REGEXP '\\?'
    ORDER BY vend_name;
```

3.4 计算字段

我们取出的数据往往不是数据库中单独的某些数据段，例如我们需要取出一个姓名 + 国家的数据字段，而数据库中只有这两个单独的数据段。一种方式是将数据字段发送给程序，让程序进行处理。但 MySQL 也为我们提供了字段拼接服务。

3.4.1 Concat 拼接字段

Concat() 函数可以将多列拼接起来，它的用法和 Java 函数十分相似，参数为列名或字符串。

```
SELECT Concat(user, ' (', host, ')') FROM mysql.user;
```

观察上述语句的返回情况，会发现列名就是我们的 Concat 函数，我们可以用别名来指定我们所需要的名字。

```
SELECT Concat(user, ' (', host, ')') AS title
    FROM mysql.user;
```

3.4.2 算数运算

MySQL 支持最基本的加减乘除运算 (+,-,*,/), 下面是一个例子:

```
SELECT prod_id,
    quantity,
    item_price,
    quantity*item_price AS expanded_price
FROM orderitems
WHERE order_num = 20005;
```

3.5 数据处理函数

函数可以帮我我们对数据进行处理，本章不深入讨论函数，仅给出一些常用的数据处理函数及例子。

SQL 中的函数按作用可以分为如下几类:

- 处理文本串的文本函数，即处理字符串的。
- 处理数值数据，进行算数操作。
- 处理日期和时间并从这些值中提取特定部分。
- 返回 DBMS 正使用的特殊信息。

函数的使用非常简单，像 Java 一样传参即可。

3.5.1 文本处理函数

举一个简单的例子，Upper() 函数将字符串转换为大写:

```
1 SELECT host, Upper(host) FROM mysql.user;
```

常见文本处理函数:

表 2.2 常见文本处理函数

函数	说明	函数	说明
Left()	返回串左边的字符	Right()	返回串右边的字符
Length()	返回串的长度	Locate()	找出串的一个字串
Lower()	转换为小写	Upper()	转换为大写
LTrim()	去掉串左边的空格	RTrim()	去掉串右边的空格
Soundex()	返回 Soundex 值	SubString()	返回字串的字符

这些函数的具体用法，可以使用 help 语句查询。

3.5.2 日期和时间处理函数

日期和时间采用相应的数据类型和特殊的格式存储，以便能快速和有效地排序或过滤，并且节省物理存储空间。

```
1 SELECT Date(last_update) from mysql.server_cost;
```

常见日期和时间处理函数:

表 2.3 常见日期和时间处理函数

函数	说明	函数	说明
AddDate()	增加一天	AddTime()	增加一个时间
CurDate()	返回当前日期	CurTime()	返回当前时间
Date()	返回日期时间的日期部分	DateDiff()	计算两个日期之差
Date_Add()	高度灵活的日期运算函数	DayofWeek()	返回日期对应的星期
Date_Format()	格式化的日期或时间串	Now()	返回当前日期和时间
Day(), Hour(), Minute(), Month(), Second(), Time(), Year()			返回对应的日期或时间部分

3.5.3 数值处理函数

这个没什么好说的，和 Java 中的 Math 类似。

常见数值处理函数:

表 2.4 常见数值处理函数

函数	说明	函数	说明
Abs()	返回一个数的绝对值	Cos()	返回余弦值
Exp()	返回一个数的指数值	Mod()	返回取余操作
Pi()	返回圆周率	Rand()	返回一个随机数
Sin()	返回正弦值	Sqrt()	返回数的平方根
Tan()	返回正切值		

3.5.4 聚集函数

前面介绍了几个检索与处理函数，聚集函数用于汇总数据，便于分析和报表生成。聚集函数有以下几种:

- 确定表中的行数。
- 获得表中行组的和。
- 找出表列的最大值，最小值，平均值。

聚集函数运行在行组上，计算和返回单个值的函数。常见聚集函数:

表 2.5 常见聚集函数

函数	说明	函数	说明
AVG()	返回某列平均值	COUNT()	返回某列行数
MAX()	返回某列最大值	MIN()	返回某列最小值
SUM()	返回某列之和		

这些函数和 EXCEL 的函数类似，没什么好说的，部分函数可以加上 DISTINCT 等关键字，详情使用 help 语句查看。下面给出一个例子:

```
SELECT COUNT(*) AS num_items,
    MIN(prod_price) AS price_min,
    AVG(DISTINCT prod_price) AS price_avg
FROM produces;
```

3.6 数据分组

分组允许把数据分为多个逻辑组，以便能对每个组进行聚集计算。

3.6.1 GROUP BY 创建分组

GROUP BY 的使用方式:

```
SELECT object_type , Count(*)
    FROM performance_schema.setup_objects
    GROUP BY object_type;
```

上述例子中，计算结果会按 object_type 进行分组，也可以理解为计算各个 object_type 包含的数据行数。

GROUP BY 子句有一些规定:

- GROUP BY 子句可以包含任意数目的列。这使得能对分组进行嵌套，为数据分组提供更细致的控制。
- 如果在 GROUP BY 子句中嵌套了分组，数据将在最后规定的分组上进行汇总。
- GROUP BY 子句中列出的每个列都必须是检索列或有效的表达式（但不能是聚集函数）。如果在 SELECT 中使用表达式，则必须在 GROUP BY 子句中指定相同的表达式。不能使用别名。
- 除聚集计算语句外，SELECT 语句中的每个列都必须在 GROUP BY 子句中给出。
- 如果分组列中具有 NULL 值，则 NULL 将作为一个分组返回。如果列中有多行 NULL 值，它们将分为一组。
- GROUP BY 子句必须出现在 WHERE 子句之后，ORDER BY 子句之前。

使用 WITH ROLLUP 关键字，可以得到每个分组以及每个分组汇总级别（针对每个分组）的值。

3.6.2 HAVING 过滤分组

我们前面已经使用过 WHERE 进行过滤，但是 WHERE 子句只能对行进行过滤，事实上，WHERE 子句没有分组的概念。

HAVING 子句为我们提供了组过滤的作用，HAVING 非常类似于 WHERE，事实上 HAV-

ING 完全可以替代 WHERE 子句。唯一的区别是 WHERE 用于过滤行，HAVING 用来过滤组。

> **注意 3.4.** *HAVING 支持所有 WHERE 操作符，它们的句法是相同的，只是关键字有差别。WHERE 在数据分组前进行过滤，HAVING 在分组后进行过滤。*

执行下面语句:

```
SELECT plugin_version as version,
    Count(*) as count
FROM information_schema.plugins
GROUP BY plugin_version
HAVING Count(*) > 3;
```

我们使用 HAVING Count(*) > 3 子句仅返回了数量大于 3 的分组。

3.7 阶段小结

目前位置，SELECT 语句的基本语法就结束了，下面总结一下各个子句的顺序以及用处。

表 2.6 SELECT 子句及其顺序

子句	说明	是否必须
SELECT	要返回的列或表达式	是
FROM	从中检索数据的表	仅在从表选择数据时使用
WHERE	行级过滤器	否
GROUP BY	分组说明	仅在按组计算聚集时使用
HAVING	组级过滤	否
ORDER BY	输出排序顺序	否
LIMIT	要检索的行数	否

3.8 子查询

子查询，即嵌套在其他查询中的查询。将查询结果作为上一级查询的目标。在多表查询中经常使用。

3.8.1 利用子查询进行过滤

考虑下面一种情况: 需要列出订购物品 TNT2 的所有客户，按照之前的方法，需要有以下步骤:

- 检索包含物品 TNT2 的所有订单的编号;
- 检索具有前一步骤列出的订单编号的所有客户的 ID;
- 检索前一步骤返回的所有客户 ID 的客户信息。

具体实现脚本为:

```
-- 查编号，然后记下
SELECT order_num FROM orderitems WHERE prod_id = 'TNT2';
-- 查编号对应的 id
SELECT cust_id FROM orders WHERE order_num IN (20005,20007);
```

现在我们使用子查询，合并上面的步骤:

```
SELECT cust_id FROM orders
    WHERE order_num IN (SELECT order_num
                        FROM orderitems
                        WHERE prod_id = 'TNT2');
```

子查询总是从内向外处理。在处理上面的 SELECT 语句时，MySQL 实际上执行了两个操作: 先子查询，再查询。

MySQL 并没有限制子查询的嵌套数量，不过为了性能考虑，不建议使用过深的嵌套。

3.8.2 作为计算字段使用子查询

使用子查询的另一方法是创建计算字段。假如需要显示 customers 表中每个客户的订单总数。订单与相应的客户 ID 存储在 orders 表中。

为了执行这个操作，需要执行以下步骤:

- 从 customers 表中检索客户列表。
- 对于检索出的每个客户，统计其在 orders 表中的订单数目。

```
SELECT cust_name, cust_state,
    (SELECT COUNT(*) FROM orders WHERE orders.cust_id = customers.cust_id) AS orders
    FROM customers
    ORDER BY cust_name;
```

这条语句中 orders 是一个计算字段，它是由圆括号中的子查询建立的。该子查询对检索出的每个客户执行一次。在此例子中，该子查询执行了 5 次，因为检索出了 5 个客户。

> **注意 3.5. 相关子查询**: 涉及外部查询的子查询。任何时候只要列名可能有多义性，就必须使用这种语法（表名和列名由一个句点分隔）。

3.9 联结表

自本节开始，将讨论多表查询/检索的语法。

3.9.1 理论

在数据库设计中，为了避免数据重复，我们往往将不同类型的数据区分在不同的表中。关系表的设计就是要保证把信息分解成多个表，一类数据一个表。各表通过某些常用的值（即

关系设计中的关系（relational））互相关联。

外键（foreignkey）为某个表中的一列，它包含另一个表的主键值，定义了两个表之间的关系。

总之，关系数据可以有效地存储和方便地处理。因此，关系数据库的可伸缩性远比非关系数据库要好。

如果数据存储在多个表中，怎样用单条 SELECT 语句检索出数据？

答案是使用联结。简单地说，联结是一种机制，用来在一条 SELECT 语句中关联表，因此称之为联结。使用特殊的语法，可以联结多个表返回一组输出，联结在运行时关联表中正确的行。

3.9.2 创建联结

联结的创建非常简单，规定要联结的所有表以及它们如何关联即可。

```
SELECT vend_name, prod_name, prod_price
    FROM vendors, products
    WHERE vendors.vend_id = products.vend_id
    ORDER BY vend_name, prod_name;
```

上述代码中,所指定的两个列（prod_name 和 prod_price ）在一个表中,而另一个列（vend_name ）在另一个表中。在 FROM 语句中出现了两个表，分别是 vendors 和 products 。它们就是这条 SELECT 语句联结的两个表的名字。这两个表用 WHERE 子句正确联结，WHERE 子句指示 MySQL 匹配 vendors 表中的 vend_id 和 products 表中的 vend_id 。

在一条 SELECT 语句中联结几个表时，相应的关系是在运行中构造的。在联结两个表时，你实际上做的是将第一个表中的每一行与第二个表中的每一行配对。WHERE 子句作为过滤条件，它只包含那些匹配给定条件（这里是联结条件）的行。没有 WHERE 子句，第一个表中的每个行将与第二个表中的每个行配对，而不管它们逻辑上是否可以配在一起。

如果没有 WHERE 子句，将返回笛卡尔积: 检索出的行的数目将是第一个表中的行数乘以第二个表中的行数。可以尝试下面语句:

```
SELECT vend_name, prod_name, prod_price
    FROM vendors, products
    ORDER BY vend_name, prod_name;
```

应该保证所有联结都有 WHERE 子句，否则 MySQL 将返回比想要的数据多得多的数据。同理，应该保证 WHERE 子句的正确性。不正确的过滤条件将导致 MySQL 返回不正确的数据。

3.9.3 INNER JOIN 等值/内部联结

目前为止所用的联结称为等值联结（equijoin），它基于两个表之间的相等测试 (可以理解为两个集合的交集)。这种联结也称为内部联结。其实，对于这种联结可以使用稍微不同的语

法来明确指定联结的类型。下面的 SELECT 语句返回与前面例子完全相同的数据：

```
SELECT vend_name, prod_name, prod_price
    FROM vendors INNER JOIN products
    ON vendors.vend_id = products.vend_id;
```

此语句使用 INNER JOIN 指定内部联结，ON 指定联结条件。本质上和 WHERE 子句效果相同。

与自联结相对应的是交叉联结 CROSS JOIN, 他将返回笛卡尔积。

> **注意 3.6.** 虽然两种写法达到的目的一致，但 *ANSI SQL* 规范首选 *INNER JOIN* 语法。此外，尽管使用 *WHERE* 子句定义联结的确比较简单，但是使用明确的联结语法能够确保不会忘记联结条件，有时候这样做也能影响性能。

3.9.4 NATURAL JOIN 自然联结

自然连接是一种特殊的等值连接，它要求两个关系中进行比较的分量必须是相同的属性组，并且在结果中把重复的属性列去掉。可以理解为等值联结 + 删除重复列。

3.9.5 联结多个表

SQL 对一条 SELECT 语句中可以联结的表的数目没有限制。创建联结的基本规则也相同。首先列出所有表，然后定义表之间的关系。例如：

```
SELECT prod_name, vend_name, prod_price, quantity
    FROM orderitems, products, vendors
    WHERE products.vend_id = vendors.vend_id
        AND orderitems.prod_id = products.prod_id
        AND order_num = 20005;
```

> **注意 3.7.** *MySQL* 在运行时关联指定的每个表以处理联结。这种处理可能是非常耗费资源的，因此应该仔细，不要联结不必要的表。联结的表越多，性能下降越厉害。

再看之前的子查询，下面两个语句作用相同，而多表联结显然可读性更高:

```
-- 子查询
SELECT cust_name, cust_contact
FROM customers
WHERE cust_id IN (SELECT cust_id
                  FROM orders
                  WHERE order_num IN (SELECT order_num
                                      FROM orderitems
                                      WHERE prod_id = 'TNT2'));
-- 多表联结
SELECT cust_name, cust_contact
FROM customers, orders, orderitems
WHERE customers.cust_id = orders.cust_id
    AND orderitems.order_num = orders.order_num
```

```
    AND prod_id = 'TNT2';
```

表别名

为了缩短 SQL 语句，可以对表使用表别名，句法和列一样:

```
SELECT cust_name, cust_contact
    FROM customers AS c, orders AS o, orderitems AS oi
    WHERE c.cust_id = o.cust_id;
```

注意表别名不会返回到客户机上。

3.9.6 自联结

使用表别名的主要原因之一是能在单条 SELECT 语句中不止一次引用相同的表。因此，我们可以对同一张表引用两次，看作不同的表。

现在考虑这样一种情况，我们要找到生产 ID 为 DTNTR 的物品的供应商，然后找出这个供应商生产的其他物品。下面两种写法等价:

```
-- 子查询
SELECT prod_id, prod_name
FROM products
WHERE vend_id = (SELECT vend_id
                 FROM products
                 WHERE prod_id = 'DTNTR');
-- 自联结
SELECT p1.prod_id, p1.prod_name
FROM products AS p1, products AS p2
WHERE p1.vend_id = p2.vend_id
  AND p2.prod_id = 'DTNTR';
```

此查询中需要的两个表实际上是相同的表，因此 products 表在 FROM 子句中出现了两次。虽然这是完全合法的，但对 products 的引用具有二义性，因为 MySQL 不知道你引用的是 products 表中的哪个实例。

为解决此问题，使用了表别名。products 的第一次出现为别名 p1 ，第二次出现为别名 p2 。现在可以将这些别名用作表名。例如，SELECT 语句使用 p1 前缀明确地给出所需列的全名。如果不这样，MySQL 将返回错误，因为分别存在两个名为 prod_id 、prod_name 的列。MySQL 不知道想要的是哪一个列（即使它们事实上是同一个列）。WHERE（通过匹配 p1 中的 vend_id 和 p2 中的 vend_id ）首先联结两个表，然后按第二个表中的 prod_id 过滤数据，返回所需的数据。

在自联结中，将 p1, p2 看作不同的表，会更好理解。

3.9.7 OUTER JOIN 外部联结

外部联结的关键词是 OUTER JOIN, 使用外联结必须指定左或右外联结。左联结和右联结本质上是一样的，颠倒顺序而已。下面以左联结为例:

```
SELECT * FROM A LEFT JOIN B ON A.aID = B.bID;
```

左外联结是以左表为基础的，即 left join 是以左表为基础的，在这个例子中，左表（表 A）的记录全部会显示出来，而表 B 只显示符合过滤条件的那部分行。

3.9.8 小结

MySQL 必知必会讲联结有点乱，这里整理一下。我们使用 HELP JOIN; 语句可以查找出和表联结相关的语法:

```
HELP JOIN;
-- joined_table: {
--     table_reference {[INNER | CROSS] JOIN | STRAIGHT_JOIN} table_factor
    [join_specification]
--   | table_reference {LEFT|RIGHT} [OUTER] JOIN table_reference join_specification
--   | table_reference NATURAL [INNER | {LEFT|RIGHT} [OUTER]] JOIN table_factor
-- }
```

其中 STRAIGHT_JOIN 属于进阶语法，大概作用为使用左表驱动右表，提高检索性能，仅适用于内联结。

总的来说，表联结有以下几个注意点:

- NATURAL 关键字用于删除重复列。
- INNER JOIN / CROSS JOIN: 一般无需显示说明 CROSS JOIN，如果没有联结条件，默认返回笛卡尔积，否则返回等值连接。
- INNER / OUTER JOIN: 等值联结取的是并集，外联结取的是本身 + 并集。

3.10 组合查询

注意 3.8. 组合查询与使用 *WHERE* 条件的效果往往是一样的，使用那种技术取决于脚本可读性，性能等。

3.10.1 UNION 组合结果

使用 UNION 可以将多个 SELECT 语句的结果合并，取交集并自动删除重复的行。下面条语句效果是一样的:

```
-- WHERE
SELECT vend_id, prod_id, prod_price
    FROM products
    WHERE prod_price <= 5
```

```
      OR vend_id IN (1001,1002);
-- UNION
SELECT vend_id, prod_id, prod_price
    FROM products
    WHERE prod_price <= 5
UNION
SELECT vend_id, prod_id, prod_price
    FROM products
    WHERE vend_id IN (1001,1002);
```

如果我们不想删除重复的行，可以使用 UNION ALL 关键字。

UNION 关键字的使用规则如下:

- UNION 必须由两条或以上 SELECT 语句组成，语句之间用关键字 UNION 分隔。
- UNION 中每个查询必须包括相同的列，表达式或聚集函数。不然没法合并。
- 列数据必须兼容，不然没法转换比较。

使用了 UNION 关键字之后，只能在最后使用一次 ORDER BY 进行排序。提示一下，没啥好解释的。

3.11 全文本搜索

> **注意 3.9. 并非所有引擎都支持全文本搜索**: *MySQL 的两个最常用引擎: MyISAM 和 InnoDB。后者不支持全文本搜索。但一般都使用 InnoDB 引擎。*

3.11.1 启用全文本搜索

一般在创建表时启用全文本搜索，必须指定 FULLTEXT 子句和引擎:

```
CREATE TABLE productnotes (
  note_id   int          NOT NULL AUTO_INCREMENT,
  prod_id   char(10)     NOT NULL,
  note_date datetime     NOT NULL,
  note_text text         NULL ,
  PRIMARY KEY(note_id),
  FULLTEXT(note_text)
) ENGINE=MyISAM;
```

现在，只需知道这条 CREATE TABLE 语句定义表 productnotes 并列出它所包含的列即可。这些列中有一个名为 note_text 的列，为了进行全文本搜索，MySQL 根据子句 FULLTEXT(note_text) 的指示对它进行索引。这里的 FULLTEXT 索引单个列，如果需要也可以指定多个列。

3.11.2 进行全文本搜索

在索引之后，使用两个函数 Match() 和 Against() 执行全文本搜索:

• Match() 指定被搜索的列, 传递给 Match() 的值必须与 FULLTEXT() 定义中的相同。如果指定多个列，则必须列出它们（而且次序正确）。

• Against() 指定要使用的搜索表达式

下面两个语句效果类似:

```
-- LIKE 语句
SELECT note_text
    FROM productnotes
    WHERE note_text LIKE '%rabbit%';
-- 全文本搜索
SELECT note_text
    FROM productnotes
    WHERE Match(note_text) Against('rabbit');
```

此 SELECT 语句检索单个列 note_text 。由于 WHERE 子句，一个全文本搜索被执行。Match(note_text) 指示 MySQL 针对指定的列进行搜索，Against('rabbit') 指定词 rabbit 作为搜索文本。

上面两个语句返回结果内容相同，但顺序不一致，全文本搜索的一个重要部分就是对结果排序。具有较高等级的行先返回。且由于全文本搜索是索引的，速度相当快。

全文本搜索还有许多内容，但由于 InnoDB 引擎不支持，这里不做过多说明，详细参考这篇文章: `https://zhuanlan.zhihu.com/p/146361883`

3.12 INSERT 语句

3.12.1 插入完整行

INSERT 的常用语法结构如下:

```
INSERT INTO <tableName[(columnName)]>
VALUES <(value...)>, <(value...)>;
```

最简单的 INSERT 向表中插入完整的一行:

```
INSERT INTO Customers
    VALUES(NULL, 'Pep E. LaPew', '100 Main Street', 'Los Angeles', 'CA', '90046', 'USA',
        NULL, NULL);
```

虽然这种语法很简单，但并不安全，应该尽量避免使用。上面的 SQL 语句高度依赖于表中列的定义次序，并且还依赖于其次序容易获得的信息。

编写 INSERT 语句的更安全（不过更烦琐）的方法如下：

```
INSERT INTO customers(cust_name, cust_address, cust_city, cust_state, cust_zip, cust_country,
    cust_contact, cust_email)
VALUES('Pep E. LaPew', '100 Main Street', 'Los Angeles', 'CA', '90046', 'USA', NULL, NULL);
```

因为提供了列名，VALUES 必须以其指定的次序匹配指定的列名，不一定按各个列出现

在实际表中的次序。

使用这种语法，还可以省略列。这表示可以只给某些列提供值，给其他列不提供值。省略的列必须满足以下某个条件。

- 该列定义为允许 NULL 值（无值或空值）。
- 在表定义中给出默认值。这表示如果不给出值，将使用默认值。

如果对表中不允许 NULL 值且没有默认值的列不给出值，则 MySQL 将产生一条错误消息，并且相应的行插入不成功。

3.12.2 插入多行

插入多行可以用多个上述语句，或者在 VALUES 中使用, 隔开不同行的数据:

```
INSERT INTO customers(cust_name, cust_address, cust_city, cust_state, cust_zip, cust_country)
VALUES ( 'Pep E. LaPew', '100 Main Street', 'Los Angeles', 'CA', '90046', 'USA'),
       ( 'M. Martian', '42 Galaxy Way', 'New York', 'NY', '11213', 'USA');
```

3.12.3 插入检索出的数据

INSERT 还存在另一种形式，可以利用它将一条 SELECT 语句的结果插入表中。这就是所谓的 INSERT SELECT。

```
INSERT INTO customers(cust_id, cust_contact, cust_email, cust_name, cust_address, cust_city,
    cust_state, cust_zip, cust_country)
SELECT cust_id, cust_contact, cust_email, cust_name, cust_address, cust_city, cust_state,
    cust_zip, cust_country
FROM custnew;
```

3.13 UPDATE 语句

3.13.1 更新数据

UPDATE 语句非常简单，由三部分组成:

- 需要更新的表；
- 列名和它的新值；
- WHERE 过滤条件。

UPDATE 语句的常用语法如下:

```
UPDATE <tableName>
    SET <column = newValue>
    WHERE <condition>;
```

一个简单的例子如下:

```sql
UPDATE customers
    SET cust_name = 'The Fudds', cust_email = 'elmer@fudd.com'
    WHERE cust_id = 10005;
```

注意 WHERE 子句十分重要，如果缺失了 WHERE 语句，将对对应列的所有值进行修改。

3.14 DELETE 语句

3.14.1 删除数据

和更新数据类似，删除数据可以删除所有行或特定行，取决于 WHERE 子句。删除一行的例子如下:

```sql
DELETE FROM customers
    WHERE cust_id = 10006;
```

注意，DELETE 是删除一整个行，因此不需要列名，如果需要删除某列的数据，应使用 UPDATE 语句。

4 DDL 数据定义语言

数据定义语言，主要是针对数据库，数据表，视图的创建，修改。它包括以下主要语句关键字:

- CREATE: 创建数据库，数据表，视图。
- DROP: 删除表。
- ALTER: 修改表。
- RENAME: 对表进行重命名。
- TRUNCATE: 清空表内容。

> **注 4.1.** 非常奇怪，《*MySQL* 必知必会》一书没有介绍数据库操作，数据类型。本人自行添加了这部分内容。

4.1 数据库操作

数据库操作非常简单，查看；创建；删除以及很少用到的修改。

4.1.1 创建数据库

在创建数据库之前，我们可以使用以下语句查看当前拥有的数据库:

```
SHOW DATABASES;
```

更详细的，我们可以使用如下指令查看数据库信息:

```
SHOW CREATE DATABASE <databaseName>;
```

创建数据库只需要确定数据库名称即可:

```
CREATE DATABASE <databaseName>;
```

创建之后，我们可以使用数据库，如果不使用，则需要完全限定名来指定数据库。

```
USE <databaseName>;
```

4.1.2 删除数据库

就这:

```
DROP DATABASE <databaseName>;
```

4.2 数据类型

在进入到数据表操作之前，先说明一下基本的 MySQL 数据类型。

如前文所述，MySQL 数据类型大致可分为四种: 串数据类型，数值数据类型，日期和时间数据类型，二进制数据类型。

4.2.1 串数据类型

串数据类型类比高级语言中的字符串。MySQL 的串数据类型大致分为两种:

- 定长串: 串长度是确定的，占用的空间也是确定的，处理速度快，空间换时间。
- 变长串: 串长度及其分配的空间不确定，由具体值确定，处理数据满，时间换空间。

不管使用何种形式的串数据类型，串值都必须括在引号内（通常单引号更好）。

表 2.7 串数据类型

数据类型	类别	说明	示例
CHAR	定长	1-255 个字符。长度必须在创建时指定，默认 1	CHAR(10)
VARCHAR	变长	0-255 个字符。最大长度可在创建时指定	VARCHAR(255)
TINYTEXT	变长	与 TEXT 相同，最大长度为 255 字节	TINYTEXT
MEDIUMTEXT	变长	与 TEXT 相同，最大长度为 16K	MEDIUMTEXT
TEXT	变长	最大 64K, 创建时无需确定长度	TEXT
LONGTEXT	变长	与 TEXT 相同，最大长度为 4G	LONGTEXT
ENUM	变长	接受最多 64K 个串组成的集合的某个串	ENUM(‘A’,‘B’)
SET	变长	接受最多 64 个串组成的集合的零个或多个串	SET(‘A’,‘B’)

关于 ENUM 和 SET 类型可以看下面两篇文章有个初步了解:

- ENUM: `https://blog.csdn.net/qq_41684621/article/details/123111915`
- SET: `https://blog.csdn.net/qq_32253969/article/details/121746458`

4.2.2 数值数据类型

所有数值数据类型（除 BIT 和 BOOLEAN 外）都可以有符号或无符号。默认情况为有符号，可以使用 UNSIGNED 关键字，这样做将允许你存储两倍大小的值。

表 2.8 数值数据类型

数据类型	类别	说明	示例
BIT	二进制	位字段，1-64 位	b’01010001’, 0x0001
BOOLEAN / BOOL	布尔	本质上是 TINYINT(1)	true,0
TINYINT	整数	8 位，默认最大支持 127	
SMALLINT	整数	16 位，默认最大支持 3.2×10^4	
MEDIUMINT	整数	24 位，默认最大支持 8.3×10^6	
INT / INTEGER	整数	32 位，默认最大支持 2.1×10^9	
BIGINT	整数	64 位，默认最大支持 9.2×10^18	
REAL	浮点	4 字节的浮点值	
FLOAT	浮点	单精度浮点值 (4 位)	
DOUBLE	浮点	双精度浮点值 (8 位)	
DECIMAL / DEC	浮点	精度可变浮点值	decimal(5,2)

关于几种数值数据类型的基础用法如下:

- BIT: `https://blog.csdn.net/dreamyuzhou/article/details/125535450`
- DECIMAL: `https://blog.csdn.net/u013214212/article/details/103028775`

4.2.3 日期和时间数据类型

表 2.9 日期和时间数据类型

数据类型	说明	格式
DATA	表示 1000-01-01～9999-12-31 的日期	YYYY-MM-DD
TIME	时间	HH:MM:SS
DATETIME	DATE 和 TIME 的组合	
TIMESTAMP	功能和 DATETIME 相同（但范围较小）	
YEAR	2 位数字: 范围是 1970～2069，4 位数字: 范围是 1901～2155	

TIMESTAMP 相关内容: `https://blog.csdn.net/hyfsx/article/details/114101630`

4.2.4 二进制数据类型

二进制数据类型可以存储图像，多媒体等等文件。

表 2.10 二进制数据类型

数据类型	最大长度	数据类型	最大长度
TINYBLOB	255B	BLOB	64K
MEDIUMBLOB	16M	LONGBLOB	4G

4.3 创建数据表

4.3.1 CREATE TABLE 创建表

创建表必须给出两个主要信息: 表名和表列的名字及定义。

```
CREATE TABLE customers (
    cust_id       int        NOT NULL AUTO_INCREMENT,
    cust_name     char(50)   NOT NULL ,
    cust_address  char(50)   NULL ,
    cust_city     char(50)   NULL ,
    cust_state    char(5)    NULL ,
    cust_zip      char(10)   NULL ,
    cust_country  char(50)   NULL ,
    cust_contact  char(50)   NULL ,
    cust_email    char(255)  NULL ,
    PRIMARY KEY (cust_id)
) ENGINE=InnoDB;
```

除了表名，列名，列类型。还包含三个重要的其他信息: 列规定关键字，主键约束，引擎类型。

> **注意 4.1.** 如果你仅想在一个表不存在时创建它，应该在表名后给出 *IF NOT EXISTS* 。这样做不检查已有表的模式是否与你打算创建的表模式相匹配。它只是查看表名是否存在，并且仅在表名不存在时创建它。

4.3.2 NOT NULL 强制数据

默认情况下，列都是 NULL 列，即可以空缺。如果要指定某列不能空缺，可以使用 NOT NULL 进行规定。

```
CREATE TABLE orders (
     order_num int      NOT NULL AUTO_INCREMENT,
     order_date datetime NOT NULL ,
     cust_id   int      NOT NULL ,
     PRIMARY KEY (order_num)
    ) ENGINE=InnoDB;
```

4.3.3 AUTO_INCREMENT 自增

AUTO_INCREMENT 很好理解，自增。有两种方式指定初始值，一种是在创建表时用 AUTO_INCREMENT=n 指定，一种通过 ALTER 语句指定:

```
ALTER TABLE table_name AUTO_INCREMENT=n。
```

AUTO_INCREMENT 具体的数值遵循以下算法:

- 如果插入的值与已有的编号重复，报错。
- 如果插入的值大于已编号的值，则从这个新值开始递增。

AUTO_INCREMENT 有以下注意点:

- 值必须唯一, 且必须有唯一索引，以避免序号重复 (是主键，或是主键的一部分)。
- 值必须具有 NOT NULL 属性。
- 序号的最大值受该列的数据类型约束，一旦达到上限，AUTO_INCREMENT 就会失效。

如果我们要获取上一个 AUTO_INCREMENT 值，可以通过下列语法:

```
SELECT last_insert_id();
```

4.3.4 DEFAULT 指定默认值

使用 DEFAULT 可以指定某列的默认值，许多数据库开发人员使用默认值而不是 NULL 列。

```
CREATE TABLE orderitems (
  order_num int         NOT NULL ,
  order_item int        NOT NULL ,
  prod_id   char(10)    NOT NULL ,
  quantity  int         NOT NULL DEFAULT 1,
  item_price decimal(8,2) NOT NULL ,
  PRIMARY KEY (order_num, order_item)
) ENGINE=InnoDB;
```

4.3.5 UNIQUE 唯一

顾名思义，这列的值必须是唯一的，有两种方法规定:

```
create table department(
    id int,
    name char(10) unique
)
create table department(
    id int,
    name char(10),
    unique(name)
)
```

4.3.6 PRIMARY KEY 主键

主键有的主要作用是确定该数据的唯一性，因此主键必须 UNIQUE 且 NOT NULL。

主键有两种方式进行规定:

```
-- 方式 1
CREATE TABLE orderitems
(
  order_num int         NOT NULL,
```

```
  order_item int         NOT NULL,
  prod_id   char(10)     NOT NULL,
  quantity  int          NOT NULL,
  item_price decimal(8,2) NOT NULL,
  PRIMARY KEY (order_num, order_item)
) ENGINE=InnoDB;
-- 方式 2
CREATE TABLE orderitems
(
  order_num int          NOT NULL PRIMARY KEY,
  order_item int         NOT NULL PRIMARY KEY,
  prod_id   char(10)     NOT NULL,
  quantity  int          NOT NULL,
  item_price decimal(8,2) NOT NULL
) ENGINE=InnoDB;
```

4.3.7 FOREIGN KEY 外键

> **注意 4.2.** 阿里开发规范: 禁用外键。除非不得已或者老板要求，不要用外键。

外键的定义:

- 外键是某个表中的一列，它包含在另一个表的主键中。
- 外键也是索引的一种，是通过一张表中的一列指向另一张表中的主键，来对两张表进行关联。
- 一张表可以有一个外键，也可以存在多个外键，与多张表进行关联。

外键的主要作用是保证数据的一致性和完整性，并且减少数据冗余。简言之，外键是为了关联其他数据表，例如在学生表中加入课程 ID 外键，以此来关联对应的课程数据。

与主键不同，外键可以是 NULL 或重复 (不要求 UNIQUE)。外键有如下要求:

- 从表插入/修改/删除新行，其外键值不是主表的主键值便阻止插入/修改/删除。
- 主表删除行，其主键值在从表里存在便阻止删除 (要想删除，必须先删除从表的相关行)。
- 主表修改主键值，旧值在从表里存在便阻止修改 (要想修改，必须先删除从表的相关行)。

级联执行:

- 主表删除行，连带从表的相关行一起删除。
- 主表修改主键值，连带从表相关行的外键值一起修改。

创建外键的子句如下:

```
FOREIGN KEY (<columnName>) REFERENCE <tableName> (<columnName>)
```

两种创建外键的方式

```
-- 创建表时创建
CREATE TABLE Products (
```

```
  prod_id    INT           NOT NULL PRIMARY KEY AUTO_INCREMENT,
  vend_id    INT           NOT NULL COMMENT '供应商ID',
  prod_name  VARCHAR(30)   NOT NULL COMMENT '产品名',
  prod_price DOUBLE        NOT NULL COMMENT '产品价格',
  prod_desc  VARCHAR(100)  COMMENT '产品描述',
  FOREIGN KEY (vend_id) REFERENCES Vendors (vend_id)CO
);
-- 已创建的表添加外键
ALTER TABLE Products
    ADD FOREIGN KEY products_vendors_fk_1 (vend_id) REFERENCES Vendors (vend_id);
```

总而言之，外键虽然能带来一定的好处，但操作外键却比较复杂，且影响数据库性能。如果可以，应尽量在应用层完成相关操作。下面有两个深入了解外键的链接:

- 外键: https://blog.csdn.net/Jerry_Chang31/article/details/105093881
- 创建外键: https://blog.csdn.net/Jack_PengPeng/article/details/87690101

4.3.8 引擎类型

MySQL5 及后续版本默认使用 InnoDB 引擎，下面说一下几个常用引擎的区别:

- InnoDB: MySQL5 及之后的默认引擎，具有事务处理能力，不支持全文本搜索，是最主流的引擎。
- MyISAM: MySQL5 之前的默认引擎，不具备事务处理能力，支持全文本搜索，逐渐被 InnoDB 取代。
- MEMORY: 功能等同于 MyISAM, 数据存储在内存中，不是磁盘，速度快，适用于临时表。

主要，只有 InnoDB 支持外键，其他引擎不支持外键。

4.3.9 字符集与校对

这部分了解即可。

MySQL 支持众多的字符集。为查看所支持的字符集完整列表，使用以下语句：

```
SHOW CHARACTER SET;
```

校对是为规定字符如何比较的指令，如是否区分大小写。为了查看所支持校对的完整列表，使用以下语句：

```
SHOW COLLATION;
```

实际上，字符集很少是服务器范围（甚至数据库范围）的设置。不同的表，甚至不同的列都可能需要不同的字符集，而且两者都可以在创建表时指定。

```
CREATE TABLE mytable (
   columnn1  INT,
   columnn2  VARCHAR(10)
```

```
) DEFAULT CHARACTER SET hebrew
  COLLATE hebrew_general_ci;
```

除了能指定字符集和校对的表范围外，MySQL 还允许对每个列设置它们，如下所示：

```
CREATE TABLE mytable (
   columnn1  INT,
   columnn2  VARCHAR(10),
   column3   VARCHAR(10) CHARACTER SET latin1 COLLATE latin1_general_ci
) DEFAULT CHARACTER SET hebrew
  COLLATE hebrew_general_ci;
```

校对在对用 ORDER BY 子句检索出来的数据排序时起重要的作用。可以显示指出 ORDER BY 所用的校对类型:

```
SELECT * FROM customers
    ORDER BY lastname, firstname COLLATE latin1_general_cs;
```

4.4 其他表操作

4.4.1 ALTER TABLE 更新表

理想状态下，当表中存储数据以后，该表就不应该再被更新。更新表往往需要大量的时间处理。

ALTER TABLE 本身的语法十分简单:

```
ALTER TABLE vendors
    ADD vend_phone CHAR(20);  -- 增加列
    DROP old_vend_phone;      -- 删除列
```

默认情况下，ADD/DROP 都是对列进行操作。也可以显示指定 COLUMN 关键字。

更常用的，ALTER TABLE 用来指定外键约束:

```
ALTER TABLE Products
    ADD FOREIGN KEY products_vendors_fk_1 (vend_id) REFERENCES Vendors (vend_id)
```

> **注意 4.3.** 使用 *ALTER TABLE* 需要及其小心，因为 *MySQL* 没有回滚，不能撤销。修改表往往会对一整列进行修改。

4.4.2 DROP TABLE 删除表

删除表是删除整个表，不同于行操作中的 DELETE FROM 删除所有行。

```
DROP TABLE customers2;
```

除此之外，还可以用 TRUNCATE 删除表中所有数据:

```
TRUNCATE [TABLE] <tableName>;
```

虽然 TRUNCATE 删除的是所有数据而不是表本省，但他属于 DDL 语言，速度更快。

4.4.3 RENAME TABLE 重命名表

可以重命名一个或多个表:

```
-- 重命名一个
RENAME TABLE customers2 TO customers;
-- 重命名多个
RENAME TABLE old_student TO student,
             old_major TO major,
             old_teacher TO teacher;
```

4.5 视图

4.5.1 视图的作用

视图是虚拟的表，表面上它和表的显示形式一样，都是表结构。实际上是对表的数据显示。

视图从 MySQL5 开始引入，为什么要引入视图? 举个很简单的例子，学生选课，在学生表中只有课程的 ID，这就够了，因为可以根据 ID 再次查询到选课的相关信息，这样设计数据库避免了数据冗余。但是我要看数据的时候，选课 ID 对于人来说并不能提供任何信息，我需要将学生表和课程表组合起来。

再举个例子，我要从几个表中提取相关的数据信息，可以这样做:

```
SELECT cust_name, cust_contact
    FROM customers, orders, orderitems
    WHERE customers.cust_id = orders.cust_id
      AND orderitems.order_num = orders.order_num
      AND prod_id = 'TNT2';
```

也不是不行，但似乎有点麻烦，假如可以把整个查询包装成一个名为 productcustomers 的虚拟表，则可以如下轻松地检索出相同的数据：

```
SELECT cust_name, cust_contact
    FROM productcustomers
    WHERE prod_id = 'TNT2';
```

显而易见，这样更方便。可以看出，设计数据表的时候我们更关注的是性能，后续维护。但在使用数据表的时候，我们却希望更加方便，视图就是为了解决这个问题而出现的。

使用视图有以下好处:

- 简化复杂的 SQL 操作。
- 使用表的组成部分而不是整个表。
- 保护数据。可以给用户授予表的特定部分的访问权限而不是整个表的访问权限。

- 更改数据格式和表示。视图可返回与底层表的表示和格式不同的数据。

在视图创建之后，可以用与表基本相同的方式利用它们。可以对视图执行 SELECT 操作，过滤和排序数据，将视图联结到其他视图或表，甚至能添加和更新数据。

重要的是知道视图仅仅是用来查看存储在别处的数据的一种设施。视图本身不包含数据，因此它们返回的数据是从其他表中检索出来的。在添加或更改这些表中的数据时，视图将返回改变过的数据。

> **注意 4.4.** 试图本身并不包含数据，如果你用多个联结和过滤创建了复杂的视图或者嵌套了视图，可能会发现性能下降得很厉害。

视图有如下规则和限制:

- 与表一样，视图必须唯一命名。
- 对于可以创建的视图数目没有限制。
- 为了创建视图，必须具有足够的访问权限。这些限制通常由数据库管理人员授予。
- 视图可以嵌套，即可以利用从其他视图中检索数据的查询来构造一个视图。
- ORDER BY 可以用在视图中，但如果从该视图检索数据 SELECT 中也含有 ORDER BY ，那么该视图中的 ORDER BY 将被覆盖。
- 视图不能索引，也不能有关联的触发器或默认值。
- 视图可以和表一起使用。例如，编写一条联结表和视图的 SELECT 语句。

4.5.2 使用视图

操作视图和操作表类似，常见语句如下:

- 创建视图: CREATE VIEW ... AS ...
- 显示视图: SHOW CREATE VIEW viewName
- 删除视图: DROP VIEW viewName
- 更新视图: 先删除后创建，或者一起: CREATE OR REPLACE VIEW

视图的作用有很多，下面举几个例子:

```
-- 利用视图简化复杂的联结
CREATE VIEW productcustomers AS
    SELECT cust_name, cust_contact, prod_id
    FROM customers, orders, orderitems
    WHERE customers.cust_id = orders.cust_id
        AND orderitems.order_num = orders.order_num;
-- 用视图重新格式化检索出的数据
SELECT Concat(RTrim(vend_name), ' (', RTrim(vend_country), ')')
    AS vend_title
    FROM vendors
    ORDER BY vend_name;
-- 用视图过滤不想要的数据
CREATE VIEW customeremaillist AS
    SELECT cust_id, cust_name, cust_email
    FROM customers
```

```
    WHERE cust_email IS NOT NULL;
-- 使用视图与计算字段
CREATE VIEW orderitemsexpanded AS
    SELECT prod_id, quantity, item_price, quantity*item_price AS expanded_price
    FROM orderitems;
```

一般,应该将视图用于检索 (SELECT 语句) 而不用于更新 (INSERT 、UPDATE 和 DELETE)。前面提到过，视图本身并没有数据，只是对表数据的显示。如果视图定义中有如下操作，则不能对视图进行更新:

- 分组: GROUP BY, HAVING
- 联结
- 子查询
- 并
- 聚集函数
- DISTINCT
- 导出/计算列

由此可以看出，视图的限制还很很多的，没事别更新视图，看看就行了。

5 DCL 数据控制语言

数据控制语言一般由数据库管理员使用，数据库使用者了解即可，《MySQL 必知必会》原书对这部分内容讲解很少，本人添加了一些基本指令的用法。

5.1 用户

顺便讲一下用户名 @ 地址的作用:

表 2.11 地址类型

示例	说明
'user'@'localhost'	本机登录
'user'@'%'	任意地址登录
'user'@'192.168.0.100'	特定地址登录
'user'@'192.168.*.*'	只允许 ip 为 192.168 网段的地址登录

注意，从不同 IP 地址登陆的用户所拥有的权限是不同的。

5.1.1 创建用户

一般的，创建用户需要三个主要参数: 用户名，IP 地址，密码。

```
CREATE USER 'userName'@'IPAdress' IDENTIFIED WITH mysql_native_password BY 'password';
-- 简化
CREATE USER 'userName'@'IPAdress' IDENTIFIED BY 'password';
```

对用户信息进行修改后，一般要刷新一下权限，之后的操作也类似:

```
FLUSH PRIVILEGES;
```

对应的，可以修改删除用户:

```
-- 修改用户
RENAME USER '旧用户名' TO '新用户名';
-- 删除用户
DROP USER 'userName'@'IPAdress';
```

5.1.2 修改密码

修改密码的方式有很多，常用的三种如下:

第一种，修改用户表，5.7.6 以下版本密码列为 password，后续版本将 mysql.user 表中的 password 列代替成了 authentication_string，并对密码进行了加锁，无法直接读取密码。

```
USE mysql;
```

```
-- 5.7.6 之前
UPDATE user SET password=password('新密码')
    WHERE user='用户名' AND host='IP地址';
-- 5.7.6 之后
UPDATE user SET authentication_string = PASSWORD('新密码')
    WHERE user='用户名' AND host='IP地址';
```

第二种 SET PASSWORD 语句，可以直接修改所用用户的密码，如果需要修改其他人的密码，则需要有 UPDATE 权限。同样的 5.7.6 版本进行了小调整。

```
-- 修改自己的密码，建议格式
SET PASSWORD = '新密码';
-- 修改别人的密码，5.7.6 之前
SET PASSWORD FOR '用户名'@'IP地址' = PASSWORD('新密码');
-- 修改别人的密码，5.7.6 之后建议使用
SET PASSWORD FOR '用户名'@'IP地址' = '新密码';
```

第三种，使用 ALTER USER 语句:

```
ALTER USER '用户名'@'IP地址' IDENTIFIED BY '新密码';
```

5.2 权限管理

权限管理有两个关键字，GRANT 和 REVOKE，分别代表授予权限和剥夺权限，两者的语法类似。

5.2.1 GRANT 授予权限

GRANT 的基本语法如下:

```
GRANT 权限1, 权限2… ,权限n ON 数据库名.表名 TO '用户名'@'地址';
```

注意这里的数据库名和数据表名，可以用通配符 * 表示所有数据库/表。

```
-- 给所有 IP 地址登录的 user 账户所有表的所有权限
GRANT ALL PRIVILEGES ON *.* TO 'user'@'%';
-- 给所有 IP 地址登录的 user 账户 mysql.user 表的增删改查权限
GRANT SELECT, INSERT, UPDATE, DELETE ON mysql.user to 'user'@'%';
```

查看权限可以使用如下语句:

```
SHOW grants FOR 'user'@'%';
```

具体的权限可以通过 SHOW PRIVILEGES; 语句查询。

5.2.2 REVOKE 剥夺权限

REVOKE 语法和 GRANT 一样，例子如下:

```
REVOKE SELECT ON crashcourse.* FROM beforta;
```

5.3 事务处理

> **注意 5.1.** 注意，*InnoDB* 支持事务处理，*MyISAM* 则不支持。

5.3.1 事务处理的概念

事务处理（transactionprocessing）可以用来维护数据库的完整性，它保证成批的 MySQL 操作要么完全执行，要么完全不执行。

举个很简单的数据库不完整的例子: 学生表添加课程，需要先检查是否存在课程，在进入学生表添加对应的数据，可能还会创建一个新的学期课程表添加数据。可能会因为某种数据库故障（如超出磁盘空间、安全限制、表锁等）阻止了这个过程的完成。如果是检查课程出现了故障还比较容易恢复，继续插入数据即可。但如果是插入数据过程出现了故障，这一列数据就会报废，甚至整个表出现问题。

如何解决这种问题？这里就需要使用事务处理了。事务处理是一种机制，用来管理必须成批执行的 MySQL 操作，以保证数据库不包含不完整的操作结果。利用事务处理，可以保证一组操作不会中途停止，它们或者作为整体执行，或者完全不执行（除非明确指示）。如果没有错误发生，整组语句提交给（写到）数据库表。如果发生错误，则进行回退（撤销）以恢复数据库到某个已知且安全的状态。

在使用事务和事务处理时，有几个关键词汇反复出现。下面是关于事务处理需要知道的几个术语：

- 事务 (transaction) 指一组 SQL 语句;
- 回退 (rollback) 指撤销指定 SQL 语句的过程;
- 提交 (commit) 指将未存储的 SQL 语句结果写入数据库表;
- 保留点 (savepoint) 指事务处理中设置的临时占位符 (place-holder)，你可以对它发布回退 (与回退整个事务处理不同)。

5.3.2 ROLLBACK 回滚

管理事务处理的关键在于将 SQL 语句组分解为逻辑块，并明确规定数据何时应该回退，何时不应该回退。

MySQL 使用下面的语句来标识事务的开始:

```
START TRANSACTION;
```

MySQL 的 ROLLBACK 命令用来回退（撤销）MySQL 语句，ROLLBACK 只能在一个事务处理内使用（在执行一条 START TRANSACTION 命令之后）。请看下面的语句：

```
SELECT * FROM ordertotals;
```

```
START TRANSACTION;
DELETE FROM ordertotals;
SELECT * FROM ordertotals;
ROLLBACK;
SELECT * FROM ordertotals;
```

这个例子从显示 ordertotals 表（此表在第 24 章中填充）的内容开始。首先执行一条 SELECT 以显示该表不为空。然后开始一个事务处理，用一条 DELETE 语句删除 ordertotals 中的所有行。另一条 SELECT 语句验证 ordertotals 确实为空。这时用一条 ROLLBACK 语句回退 START TRANSACTION 之后的所有语句，最后一条 SELECT 语句显示该表不为空。

有两种语句无法回退，一是 SELECT 语句，因为回不回退没有区别; 二是 CREATE/DROP 操作。事务处理块中可以使用这两条语句，但如果你执行回退，它们不会被撤销。

5.3.3 COMMIT 提交

一般的 MySQL 语句都是直接针对数据库表执行和编写的。这就是所谓的隐含提交（implicitcommit），即提交（写或保存）操作是自动进行的。

但是，在事务处理块中，提交不会隐含地进行。为进行明确的提交，使用 COMMIT 语句，如下所示：

```
START TRANSACTION;
DELETE FROM orderitems WHERE order_num = 20010;
DELETE FROM orders WHERE order_num = 20010;
COMMIT;
```

在这个例子中，从系统中完全删除订单 20010 。因为涉及更新两个数据库表 orders 和 orderItems ，所以使用事务处理块来保证订单不被部分删除。最后的 COMMIT 语句仅在不出错时写出更改。如果第一条 DELETE 起作用，但第二条失败，则 DELETE 不会提交 (实际上，它是被自动撤销的)。

5.3.4 SAVEPOINT 保留点

简单的 ROLLBACK 和 COMMIT 语句就可以写入或撤销整个事务处理。但是，只是对简单的事务处理才能这样做，更复杂的事务处理可能需要部分提交或回退。

为了支持回退部分事务处理，必须能在事务处理块中合适的位置放置占位符。这样，如果需要回退，可以回退到某个占位符。这些占位符称为保留点。为了创建占位符，可如下使用 SAVEPOINT 语句：

```
SAVEPOINT delete1;
```

每个保留点都取标识它的唯一名字，以便在回退时，MySQL 知道要回退到何处。为了回退到本例给出的保留点，可如下进行：

```
ROLLBACK TO delete1;
```

保留点在事务处理完成（执行一条 ROLLBACK 或 COMMIT ）后自动释放。自 MySQL 5 以来，也可以用 RELEASE SAVEPOINT 明确地释放保留点。

详细案例: `https://morty.blog.csdn.net/article/details/103434169`

5.3.5 AUTOCOMMIT

默认的 MySQL 行为是自动提交所有更改。为指示 MySQL 不自动提交更改，需要使用以下语句：

```
SET AUTOCOMMIT=0;
```

autocommit 标志决定是否自动提交更改，不管有没有 COMMIT 语句。设置 autocommit 为 0 （假）指示 MySQL 不自动提交更改（直到 autocommit 被设置为真为止）。

5.4 存储过程

5.4.1 存储过程基础

MySQL 称存储过程的执行为调用，因此 MySQL 执行存储过程的语句为 CALL 。CALL 接受存储过程的名字以及需要传递给它的任意参数。请看以下例子：

```
CALL productpricing(@pricelow, @pricehigh, @priceaverage);
```

其中，执行名为 productpricing 的存储过程，它计算并返回产品的最低、最高和平均价格。

创建一个返回产品平均价格的存储过程。以下是其代码：

```
CREATE PROCEDURE productpricing()
BEGIN
   SELECT Avg(prod_price) AS priceaverage
   FROM products;
END;
```

如果存储过程接受参数，它们将在 () 中列举出来。

为显示用来创建一个存储过程的 CREATE 语句，使用 SHOW CREATE PROCEDURE 语句：

```
SHOW CREATE PROCEDURE ordertotal;
```

这有个问题，在命令行中; 标志着 MySQL 语句的结尾，所以上述语句在定义过程中会出问题 (遇到第一个; 就结束了)。所以要用下面这种语法:

```
DELIMITER //
CREATE PROCEDURE productpricing()
BEGIN
   SELECT Avg(prod_price) AS priceaverage
   FROM products;
```

```
END //
DELIMITER ;
```

其中 DELIMITER// 告诉命令行实用程序使用// 作为新的语句结束分隔符，可以看到标志存储过程结束的 END 定义为 END// 而不是 END;

存储过程在创建之后，被保存在服务器上以供使用，直至被删除。为删除刚创建的存储过程，可使用以下语句：

```
DROP PROCEDURE productpricing;
```

注意这里不需要给出 () 和参数。

> **注意 5.2.** *如果指定的过程不存在，则 DROP PROCEDURE 将产生一个错误。当过程存在想删除它时（如果过程不存在也不产生错误）可使用 DROP PROCEDURE IF EXISTS。其他句子也类似。*

5.4.2 使用参数

一般，存储过程并不显示结果，而是把结果返回给你指定的变量。

以下是 productpricing 的修改版本 (如果不先删除此存储过程，则不能再次创建它)：

```
CREATE PROCEDURE productpricing (
    OUT pl DECIMAL(8,2),
    OUT ph DECIMAL(8,2),
    OUT pa DECIMAL(8,2)
)
BEGIN
    SELECT Min(prod_price) INTO pl
        FROM products;
    SELECT Max(prod_price) INTO ph
        FROM products;
    SELECT Avg(prod_price) INTO pa
        FROM products;
END;
```

此存储过程接受 3 个参数：pl 存储产品最低价格，ph 存储产品最高价格，pa 存储产品平均价格。每个参数必须具有指定的类型。

MySQL 有三种参数类型:

- IN(默认): 表明该参数是一个输入参数，无需输出
- OUT: 表明该参数是一个输出参数，执行完存储过程之后会返回该值
- INOUT: 既是输入也是输出

为调用此修改过的存储过程，必须指定 3 个变量名:

```
CALL productpricing(@pricelow, @pricehigh, @priceaverage);
```

> **注意 5.3.** *所有 MySQL 变量都必须以 @ 开始。*

在调用时，这条语句并不显示任何数据。它返回以后可以显示（或在其他处理中使用）的变量。为了显示检索出的产品平均价格，可如下进行：

```
SELECT @priceaverage;
```

下面是另外一个例子，这次使用 IN 和 OUT 参数。ordertotal 接受订单号并返回该订单的合计：

```
CREATE PROCEDURE ordertotal(
    IN onumber INT,
    OUT ototal DECIMAL(8,2)
    )
BEGIN
    SELECT Sum(item_price*quantity)
        FROM orderitems
        WHERE order_num = onumber
        INTO ototal;
END;

CALL ordertotal(20005, @total);
SELECT @total;
```

5.4.3 智能存储过程

迄今为止使用的所有存储过程基本上都是封装 MySQL 简单的 SELECT 语句。虽然它们全都是有效的存储过程例子，但它们所能完成的工作你直接用这些被封装的语句就能完成（如果说它们还能带来更多的东西，那就是使事情更复杂）。只有在存储过程内包含业务规则和智能处理时，它们的威力才真正显现出来。

考虑这个场景。你需要获得与以前一样的订单合计，但需要对合计增加营业税，不过只针对某些顾客（或许是你所在州中那些顾客）。那么，你需要做下面几件事情：

- 获得合计；
- 把营业税有条件地添加到合计；
- 返回合计。

存储过程如下:

```
-- Name: ordertotal
-- Parameters: onumber = order number
--             taxable = 0 if not taxable, 1 if taxable
--             ototal = order total variable

CREATE PROCEDURE ordertotal(
    IN onumber INT,
    IN taxable BOOLEAN,
    OUT ototal DECIMAL(8,2)
) COMMENT 'Obtain order total, optionally adding tax'

BEGIN
```

```
    -- Declare variable for total
    DECLARE total DECIMAL(8,2);
    -- Declare tax percentage
    DECLARE taxrate INT DEFAULT 6;

    -- Get the order total
    SELECT Sum(item_price*quantity)
        FROM orderitems
        WHERE order_num = onumber
        INTO total;

    -- Is this taxable?
    IF taxable THEN
        -- Yes, so add taxrate to the total
        SELECT total+(total/100*taxrate) INTO total;
    END IF;

    -- And finally, save to out variable
    SELECT total INTO ototal;
END;
```

这显然是一个更高级，功能更强的存储过程。为试验它，请用以下两条语句：

```
-- 无税
CALL ordertotal(20005, 0, @total);
SELECT @total;
-- 有税
CALL ordertotal(20005, 1, @total);
SELECT @total;
```

5.5 游标

MySQL 检索操作返回一组称为结果集的行。这组返回的行都是与 SQL 语句相匹配的行（零行或多行）。使用简单的 SELECT 语句，例如，没有办法得到第一行、下一行或前 10 行，也不存在每次一行地处理所有行的简单方法（相对于成批地处理它们）。

有时，需要在检索出来的行中前进或后退一行或多行。这就是使用游标的原因。游标（cursor）是一个存储在 MySQL 服务器上的数据库查询，它不是一条 SELECT 语句，而是被该语句检索出来的结果集。在存储了游标之后，应用程序可以根据需要滚动或浏览其中的数据。

游标主要用于交互式应用，其中用户需要滚动屏幕上的数据，并对数据进行浏览或做出更改。

> **注意 5.4.** *不像多数 DBMS，MySQL 游标只能用于存储过程（和函数）。*

5.5.1 使用游标

使用游标涉及几个明确的步骤:

- 在能够使用游标前，必须声明（定义）它。这个过程实际上没有检索数据，它只是定义要使用的 SELECT 语句。
- 一旦声明后，必须打开游标以供使用。这个过程用前面定义的 SELECT 语句把数据实际检索出来。
- 对于填有数据的游标，根据需要取出（检索）各行。
- 在结束游标使用时，必须关闭游标。

在声明游标后，可根据需要频繁地打开和关闭游标。在游标打开后，可根据需要频繁地执行取操作。

游标用 DECLARE 语句创建。DECLARE 命名游标，并定义相应的 SELECT 语句，根据需要带 WHERE 和其他子句。

```
CREATE PROCEDURE processorders()
BEGIN
   DECLARE ordernumbers CURSOR
   FOR
   SELECT ordernum FROM orders;
END;
```

DECLARE 语句用来定义和命名游标，这里为 ordernumbers 。存储过程处理完成后，游标就消失（因为它局限于存储过程）。

在定义游标后，可以使用 OPEN CURSOR 语句来打开它:

```
OPEN ordernumbers;
```

在处理 OPEN 语句时执行查询，存储检索出的数据以供浏览和滚动。游标处理完成后，应当使用如下语句关闭游标：

```
CLOSE ordernumbers;
```

CLOSE 释放游标使用的所有内部内存和资源，因此在每个游标不再需要时都应该关闭。

在一个游标关闭后，如果没有重新打开，则不能使用它。但是，使用声明过的游标不需要再次声明，用 OPEN 语句打开它就可以了。

> **注意 5.5.** 隐含关闭: 如果你不明确关闭游标，*MySQL* 将会在到达 *END* 语句时自动关闭它。

下面是前面例子得修改版本:

```
CREATE PROCEDURE processorders()
BEGIN
   -- Declare the cursor
   DECLARE ordernumbers CURSOR
   FOR
   SELECT order_num FROM orders;
   -- Open the cursor
   OPEN ordernumbers;
   -- Close the cursor
   CLOSE ordernumbers;
```

```
END;
```

这个存储过程声明、打开和关闭一个游标。但对检索出的数据什么也没做。

5.5.2 使用游标数据

在一个游标被打开后，可以使用 FETCH 语句分别访问它的每一行。FETCH 指定检索什么数据（所需的列），检索出来的数据存储在什么地方。它还向前移动游标中的内部行指针，使下一条 FETCH 语句检索下一行（不重复读取同一行）。

第一个例子从游标中检索单个行（第一行）：

```
CREATE PROCEDURE processorders()
BEGIN
   -- Declare local variables
   DECLARE o INT;
   -- Declare the cursor
   DECLARE ordernumbers CURSOR
   FOR
   SELECT order_num FROM orders;
   -- Open the cursor
   OPEN ordernumbers;
   -- Get order number
   FETCH ordernumbers INTO o;
   -- Close the cursor
   CLOSE ordernumbers;
END;
```

其中 FETCH 用来检索当前行的 order_num 列（将自动从第一行开始）到一个名为 o 的局部声明的变量中。对检索出的数据不做任何处理。

在下一个例子中，循环检索数据，从第一行到最后一行：

```
CREATE PROCEDURE processorders()
BEGIN
   -- Declare local variables
   DECLARE done BOOLEAN DEFAULT 0;
   DECLARE o INT;
   -- Declare the cursor
   DECLARE ordernumbers CURSOR
   FOR
   SELECT order_num FROM orders;
   -- Declare continue handler
   DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done=1;
   -- Open the cursor
   OPEN ordernumbers;
   -- Loop through all rows
   REPEAT
      -- Get order number
      FETCH ordernumbers INTO o;
   -- End of loop
   UNTIL done END REPEAT;
```

```
   -- Close the cursor
   CLOSE ordernumbers;
END;
```

与前一个例子一样，这个例子使用 FETCH 检索当前 order_num 到声明的名为 o 的变量中。但与前一个例子不一样的是，这个例子中的 FETCH 是在 REPEAT 内，因此它反复执行直到 done 为真（由 UNTIL done END REPEAT; 规定）。为使它起作用，用一个 DEFAULT 0（假，不结束）定义变量 done 。那么，done 怎样才能在结束时被设置为真呢？答案是用以下语句：

```
DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done=1;
```

这条语句定义了一个 CONTINUE HANDLER ，它是在条件出现时被执行的代码。这里，它指出当 SQLSTATE '02000' 出现时，SET done=1 。SQLSTATE '02000' 是一个未找到条件，当 REPEAT 由于没有更多的行供循环而不能继续时，出现这个条件。

如果调用这个存储过程，它将定义几个变量和一个 CONTINUE HANDLER ，定义并打开一个游标，重复读取所有行，然后关闭游标。

如果一切正常，你可以在循环内放入任意需要的处理（在 FETCH 语句之后，循环结束之前）。

为了把这些内容组织起来，下面给出我们的游标存储过程样例的更进一步修改的版本，这次对取出的数据进行某种实际的处理：

```
CREATE PROCEDURE processorders()
BEGIN
   -- Declare local variables
   DECLARE done BOOLEAN DEFAULT 0;
   DECLARE o INT;
   DECLARE t DECIMAL(8,2);
   -- Declare the cursor
   DECLARE ordernumbers CURSOR
   FOR
   SELECT order_num FROM orders;
   -- Declare continue handler
   DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done=1;
   -- Create a table to store the results
   CREATE TABLE IF NOT EXISTS ordertotals
      (order_num INT, total DECIMAL(8,2));
   -- Open the cursor
   OPEN ordernumbers;
   -- Loop through all rows
   REPEAT
      -- Get order number
      FETCH ordernumbers INTO o;
      -- Get the total for this order
      CALL ordertotal(o, 1, t);
      -- Insert order and total into ordertotals
      INSERT INTO ordertotals(order_num, total)
      VALUES(o, t);
   -- End of loop
```

```
  UNTIL done END REPEAT;
  -- Close the cursor
  CLOSE ordernumbers;
END;
```

5.6 触发器

5.6.1 触发器基础

MySQL 语句在需要时被执行，存储过程也是如此。但是，如果你想要某条语句（或某些语句）在事件发生时自动执行，怎么办呢？触发器类似回调函数。例如插入电话号码时，检查格式是否正确; 订购一个产品时，都从库存数量中减去订购的数量。

所有这些例子的共同之处是它们都需要在某个表发生更改时自动处理。这确切地说就是触发器。触发器是 MySQL 响应以下任意语句而自动执行的一条 MySQL 语句（或位于 BEGIN 和 END 语句之间的一组语句）：

- DELETE; INSERT; UPDATE。

在创建触发器时，需要给出 4 条信息：

- 唯一的触发器名;
- 触发器关联的表;
- 触发器应该响应的活动 (DELETE, INSERT, UPDATE);
- 触发器合适执行 (处理前，处理后)。

触发器用 CREATE TRIGGER 语句创建。下面是一个简单的例子：

```
CREATE TRIGGER newproduct AFTER INSERT ON products
    FOR EACH ROW SELECT 'Product added' INTO @asd;
```

在这个例子中，文本 Product added 将对每个插入的行显示一次。

> **版本 5.1.** *原书上这段语句最后没有 INTO @asd 因为 MYSQL5 以后，不允许触发器返回任何结果。因此只能将结果传入变量，再用 SELECT 调用。*

为了测试这个触发器，使用 INSERT 语句添加一行或多行到 products 中，你将看到对每个成功的插入，显示 Product added 消息。

触发器按每个表每个事件每次地定义，每个表每个事件每次只允许一个触发器。因此，每个表最多支持 6 个触发器（每条 INSERT 、UPDATE 和 DELETE 的之前和之后）。单一触发器不能与多个事件或多个表关联，所以，如果你需要一个对 INSERT 和 UPDATE 操作执行的触发器，则应该定义两个触发器。

如果 BEFORE 触发器失败，则 MySQL 将不执行请求的操作。此外，如果 BEFORE 触发器或语句本身失败，MySQL 将不执行 AFTER 触发器（如果有的话）。

> **注意 5.6.** *仅有表支持触发器，视图，临时表都不支持。*

删除触发器使用 DROP TRIGGER 语句:

```
DROP TRIGGER newproduct;
```

触发器不能更新或覆盖。为了修改一个触发器，必须先删除它，然后再重新创建。

5.6.2 INSERT 触发器

INSERT 触发器需要知道以下几点：

- 在 INSERT 触发器代码内，可引用一个名为 NEW 的虚拟表，访问被插入的行；
- 在 BEFORE INSERT 触发器中，NEW 中的值也可以被更新（允许更改被插入的值）；
- 对于 AUTO_INCREMENT 列，NEW 在 INSERT 执行之前包含 0，在 INSERT 执行之后包含新的自动生成值。

举个例子:

```
CREATE TRIGGER neworder AFTER INSERT ON orders
    FOR EACH ROW SELECT NEW.order_num INTO @temp;
```

此代码创建一个名为 neworder 的触发器，它按照 AFTER INSERT ON orders 执行。在插入一个新订单到 orders 表时，MySQL 生成一个新订单号并保存到 order_num 中。触发器从 NEW.order_num 取得这个值并返回它。此触发器必须按照 AFTER INSERT 执行，因为在 BEFORE INSERT 语句执行之前，新 order_num 还没有生成。对于 orders 的每次插入使用这个触发器将总是返回新的订单号。

测试触发器:

```
INSERT INTO orders(order_date, cust_id)
    VALUES(Now(), 10001);
SELECT @temp;
```

5.6.3 DELETE 触发器

DELETE 触发器需要知道以下两点：

- 在 DELETE 触发器代码内，你可以引用一个名为 OLD 的虚拟表，访问被删除的行；
- OLD 中的值全都是只读的，不能更新。

下面的例子演示使用 OLD 保存将要被删除的行到一个存档表中：

```
CREATE TRIGGER deleteorder BEFORE DELETE ON orders
FOR EACH ROW
BEGIN
   INSERT INTO archive_orders(order_num, order_date, cust_id)
   VALUES(OLD.order_num, OLD.order_date, OLD.cust_id);
END;
```

在任意订单被删除前将执行此触发器。它使用一条 INSERT 语句将 OLD 中的值（要被删除的订单）保存到一个名为 archive_orders 的存档表中（为实际使用这个例子，你需要用与

orders 相同的列创建一个名为 archive_orders 的表）。

5.6.4 UPDATE 触发器

UPDATE 触发器需要知道以下几点：

- 在 UPDATE 触发器代码中，你可以引用一个名为 OLD 的虚拟表访问以前（UPDATE 语句前）的值，引用一个名为 NEW 的虚拟表访问新更新的值；
- 在 BEFORE UPDATE 触发器中，NEW 中的值可能也被更新（允许更改将要用于 UPDATE 语句中的值）；
- OLD 中的值全都是只读的，不能更新。

下面的例子保证州名缩写总是大写（不管 UPDATE 语句中给出的是大写还是小写）：

```
CREATE TRIGGER updatevendor BEFORE UPDATE ON vendors
    FOR EACH ROW SET NEW.vend_state = Upper(NEW.vend_state);
```

5.7 数据库维护

5.7.1 备份数据

由于 MySQL 数据库是基于磁盘的文件，普通的备份系统和例程就能备份 MySQL 的数据。但是，由于这些文件总是处于打开和使用状态，普通的文件副本备份不一定总是有效。

下面列出这个问题的可能解决方案。

- 使用命令行实用程序 mysqldump 转储所有数据库内容到某个外部文件。在进行常规备份前这个实用程序应该正常运行，以便能正确地备份转储文件。
- 可用命令行实用程序 mysqlhotcopy 从一个数据库复制所有数据（并非所有数据库引擎都支持这个实用程序）。
- 可以使用 MySQL 的 BACKUP TABLE 或 SELECT INTO OUTFILE 转储所有数据到某个外部文件。这两条语句都接受将要创建的系统文件名，此系统文件必须不存在，否则会出错。数据可以用 RESTORE TABLE 来复原。

5.7.2 进行数据库维护

MySQL 提供了一系列的语句，可以（应该）用来保证数据库正确和正常运行。

ANALYZE TABLE ，用来检查表键是否正确。

```
ANALYZE TABLE orders;
```

CHECK TABLE 用来针对许多问题对表进行检查。

```
CHECK TABLE orders, orderitems;
```

5.7.3 日志文件

MySQL 维护管理员依赖的一系列日志文件。主要的日志文件有以下几种:

• 错误日志。它包含启动和关闭问题以及任意关键错误的细节。此日志通常名为 hostname.err ，位于 data 目录中。此日志名可用–log-error 命令行选项更改。

• 查询日志。它记录所有 MySQL 活动，在诊断问题时非常有用。此日志文件可能会很快地变得非常大，因此不应该长期使用它。此日志通常名为 hostname.log ，位于 data 目录中。此名字可以用–log 命令行选项更改。

• 二进制日志。它记录更新过数据（或者可能更新过数据）的所有语句。此日志通常名为 hostname-bin ，位于 data 目录内。此名字可以用–log-bin 命令行选项更改。注意，这个日志文件是 MySQL 5 中添加的，以前的 MySQL 版本中使用的是更新日志。

• 缓慢查询日志。顾名思义，此日志记录执行缓慢的任何查询。这个日志在确定数据库何处需要优化很有用。此日志通常名为 hostname-slow.log ，位于 data 目录中。此名字可以用–log-slow-queries 命令行选项更改。

第二部分

MySQL 进阶

III 深入了解 MySQL

6 MySQL 架构

6.1 MySQL 逻辑架构

MySQL 的基本逻辑架构如下，其中最上层的服务是大多数基于网络的客户端/服务器公有的。第二层包含了 MySQL 的核心服务功能。第三层则包含了存储引擎。:

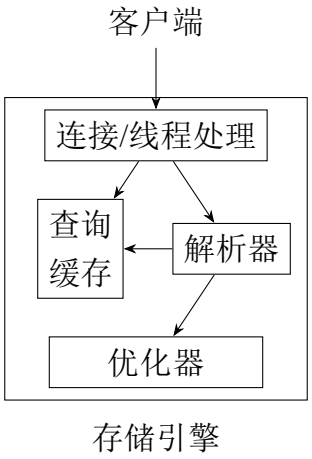


图 6.1 MySQL 服务器逻辑架构图

连接管理与安全性

每个客户端连接都会在服务器进程中拥有一个线程，这个连接的查询只会在这个单独的线程中执行，该线程只能轮流在某个 CPU 核心或者 CPU 中运行。服务器会负责缓存线程。

当客户端（应用）连接到 MySQL 服务器时，服务器需要对其进行认证。认证基于用户名、原始主机信息和密码。如果使用了安全套接字（SSL）的方式连接，还可以使用 X.509 证书认证。一旦客户端连接成功，服务器会继续验证该客户端是否具有执行某个特定查询的权限。

优化与执行

MySQL 会解析查询，并创建内部数据结构（解析树），然后对其进行各种优化，包括重写查询、决定表的读取顺序，以及选择合适的索引等。

优化器并不关心表使用的是什么存储引擎，但存储引擎对于优化查询是有影响的。优化器会请求存储引擎提供容量或某个具体操作的开销信息，以及表数据的统计信息等。

对于 SELECT 语句，在解析查询之前，服务器会先检查查询缓存（Query Cache），如果

能够在其中找到对应的查询，服务器就不必再执行查询解析、优化和执行的整个过程，而是直接返回查询缓存中的结果集。

6.2 并发控制

但如果两个进程在同一时刻对同一个邮箱投递邮件，会发生什么情况？显然，邮箱的数据会被破坏，两封邮件的内容会交叉地附加在邮箱文件的末尾。设计良好的邮箱投递系统会通过锁（lock）来防止数据损坏。如果客户试图投递邮件，而邮箱已经被其他客户锁住，那就必须等待，直到锁释放才能进行投递。

这种锁的方案在实际应用环境中虽然工作良好，但并不支持并发处理。因为在任意一个时刻，只有一个进程可以修改邮箱的数据，这在大容量的邮箱系统中是个问题。

读写锁

在处理并发读或者写时，可以通过实现一个由两种类型的锁组成的锁系统来解决数据安全问题。这两种类型的锁通常被称为共享锁（shared lock）和排他锁（exclusive lock），也叫读锁（read lock）和写锁（write lock）。

读锁是共享的，或者说是相互不阻塞的。多个客户在同一时刻可以同时读取同一个资源，而互不干扰。写锁则是排他的，也就是说一个写锁会阻塞其他的写锁和读锁，这是出于安全策略的考虑，只有这样，才能确保在给定的时间里，只有一个用户能执行写入，并防止其他用户读取正在写入的同一资源。

粒度锁

一种提高共享资源并发性的方式就是让锁定对象更有选择性。尽量只锁定需要修改的部分数据，而不是所有的资源。更理想的方式是，只对会修改的数据片进行精确的锁定。任何时候，在给定的资源上，锁定的数据量越少，则系统的并发程度越高，只要相互之间不发生冲突即可。

所谓的锁策略，就是在锁的开销和数据的安全性之间寻求平衡，这种平衡当然也会影响到性能。大多数商业数据库系统没有提供更多的选择，一般都是在表上施加行级锁（row-level lock），并以各种复杂的方式来实现，以便在锁比较多的情况下尽可能地提供更好的性能。

而 MySQL 则提供了多种选择。每种 MySQL 存储引擎都可以实现自己的锁策略和锁粒度。MySQL 有两种重要的锁策略：表锁，行级锁。

- 表锁：表锁是 MySQL 中最基本的锁策略，并且是开销最小的策略。写锁也比读锁有更高的优先级，因此一个写锁请求可能会被插入到读锁队列的前面
- 行级锁：行级锁只在存储引擎层实现，而 MySQL 服务器层没有实现。

6.3 事务

事务就是一组原子性的 SQL 查询，或者说一个独立的工作单元。事务内的语句，要么全部执行成功，要么全部执行失败。

一个运行良好的事务处理系统，必须具备这些标准特征 (ACIS)。

- 原子性 (atomicity): 一个事务必须被视为一个不可分割的最小工作单元，整个事务中的所有操作要么全部提交成功，要么全部失败回滚，对于一个事务来说，不可能只执行其中的一部分操作。
- 一致性 (consistency): 数据库总是从一个一致性的状态转换到另外一个一致性的状态。例如某一部执行失败，只要事务没有提交，就没有改变对应的状态。
- 隔离性 (isolation): 一个事务所做的修改在最终提交以前，对其他事务是不可见的。
- 持久性 (durability): 一旦事务提交，则其所做的修改就会永久保存到数据库中。此时即使系统崩溃，修改的数据也不会丢失。

隔离级别

在 SQL 标准中定义了四种隔离级别，每一种级别都规定了一个事务中所做的修改，哪些在事务内和事务间是可见的，哪些是不可见的。较低级别的隔离通常可以执行更高的并发，系统的开销也更低。

- READ UNCOMMITTED(未提交): 事务中的修改，即使没有提交，对其他事务也都是可见的。事务可以读取未提交的数据，这也被称为脏读。这个级别会导致很多问题，从性能上来说，READ UNCOMMITTED 不会比其他的级别好太多，但却缺乏其他级别的很多好处，除非真的有非常必要的理由，在实际应用中一般很少使用。
- READ COMMITTED(提交读): 大多数数据库系统的默认隔离级别都是 READ COMMITTED（但 MySQL 不是）。READ COMMITTED 满足前面提到的隔离性的简单定义：一个事务开始时，只能“看见”已经提交的事务所做的修改。换句话说，一个事务从开始直到提交之前，所做的任何修改对其他事务都是不可见的。
- REPEATABLE READ(可重复读): 解决了脏读的问题。该级别保证了在同一个事务中多次读取同样记录的结果是一致的。但是理论上，可重复读隔离级别还是无法解决另外一个幻读问题。所谓幻读，指的是当某个事务在读取某个范围内的记录时，另外一个事务又在该范围内插入了新的记录，当之前的事务再次读取该范围的记录时，会产生幻行。InnoDB 和 XtraDB 存储引擎通过多版本并发控制解决了幻读问题。
- SERIALIZABLE(可串行化): SERIALIZABLE 是最高的隔离级别。它通过强制事务串行执行，避免了前面说的幻读的问题。简单来说，SERIALIZABLE 会在读取的每一行数据上都加锁，所以可能导致大量的超时和锁争用的问题。实际应用中也很少用到这个隔离级别，只有在非常需要确保数据的一致性而且可以接受没有并发的情况下，才考虑采用该级别。

死锁

死锁是指两个或者多个事务在同一资源上相互占用，并请求锁定对方占用的资源，从而导致恶性循环的现象。当多个事务试图以不同的顺序锁定资源时，就可能会产生死锁。多个事务同时锁定同一个资源时，也会产生死锁。

```
START TRANSACTION;
UPDATE StockPrice SET close = 45.50 WHERE stock_id = 4 AND date = '2022-05-01';
UPDATE StockPrice SET close = 19.80 WHERE stock_id = 3 AND date = '2022-05-02';
COMMIT;

START TRANSACTION;
UPDATE StockPrice SET high = 20.12 WHERE stock_id = 3 AND date = '2022-05-02';
UPDATE StockPrice SET high = 47.20 WHERE stock_id = 4 AND date = '2022-05-01';
COMMIT;
```

如果这两个事务都执行了第一条语句，那么就至少会锁定对应的行数据，接着执行第二条语句发现对应的资源被锁定，双方都无法获取资源，就陷入了死锁循环。

为了解决这种问题，数据库系统实现了各种死锁检测和死锁超时机制。越复杂的系统，比如 InnoDB 存储引擎，越能检测到死锁的循环依赖，并立即返回一个错误。这种解决方式很有效，否则死锁会导致出现非常慢的查询。还有一种解决方式，就是当查询的时间达到锁等待超时的设定后放弃锁请求，这种方式通常来说不太好。InnoDB 目前处理死锁的方法是，将持有最少行级排他锁的事务进行回滚。

死锁发生以后，只有部分或者完全回滚其中一个事务，才能打破死锁。对于事务型的系统，这是无法避免的，所以应用程序在设计时必须考虑如何处理死锁。大多数情况下只需要重新执行因死锁回滚的事务即可。

事务日志

事务日志可以帮助提高事务的效率。使用事务日志，存储引擎在修改表的数据时只需要修改其内存拷贝，再把该修改行为记录到持久在硬盘上的事务日志中，而不用每次都将修改的数据本身持久到磁盘。

MySQL 中的事务

MySQL 提供了两种事务型的存储引擎：InnoDB 和 NDB Cluster。另外还有一些第三方存储引擎也支持事务，比较知名的包括 XtraDB 和 PBXT。

MySQL 默认采用自动提交（AUTOCOMMIT）模式。也就是说，如果不是显式地开始一个事务，则每个查询都被当作一个事务执行提交操作。在当前连接中，可以通过设置 AUTOCOMMIT 变量来启用或者禁用自动提交模式

```
SET AUTOCOMMIT = 1;
```

当 AUTOCOMMIT=0 时，所有的查询都是在一个事务中，直到显式地执行 COMMIT 提

交或者 ROLLBACK 回滚，该事务结束，同时又开始了另一个新事务。

另外还有一些命令，在执行之前会强制执行 COMMIT 提交当前的活动事务。典型的例子，在数据定义语言（DDL）中，如果是会导致大量数据改变的操作，比如 ALTER TABLE，就是如此。另外还有 LOCK TABLES 等其他语句也会导致同样的结果。

MySQL 可以通过执行 SET TRANSACTION ISOLATION LEVEL 命令来设置隔离级别。

6.4 多版本并发控制

MySQL 的大多数事务型存储引擎实现的都不是简单的行级锁。基于提升并发性能的考虑，它们一般都同时实现了多版本并发控制（MVCC）。

可以认为 MVCC 是行级锁的一个变种，但是它在很多情况下避免了加锁操作，因此开销更低。虽然实现机制有所不同，但大都实现了非阻塞的读操作，写操作也只锁定必要的行。

MVCC 的实现，是通过保存数据在某个时间点的快照来实现的。也就是说，不管需要执行多长时间，每个事务看到的数据都是一致的。MySQL 通过保存 Undo Redo 日志的方式，使得多个事务均可以获取正确的数据。

前面说到不同存储引擎的 MVCC 实现是不同的，典型的有乐观（optimistic）并发控制和悲观（pessimistic）并发控制。

InnoDB 的 MVCC，是通过在每行记录后面保存两个隐藏的列来实现的。这两个列，一个保存了行的创建时间，一个保存行的过期时间（或删除时间）。当然存储的并不是实际的时间值，而是系统版本号（system version number）。每开始一个新的事务，系统版本号都会自动递增。事务开始时刻的系统版本号会作为事务的版本号，用来和查询到的每行记录的版本号进行比较。下面看一下在 REPEATABLE READ 隔离级别下，MVCC 具体是如何操作的。

- **SELECT** InnoDB 会根据以下两个条件检查每行记录：
  - InnoDB 只查找版本早于当前事务版本的数据行，这样可以确保事务读取的行，要么是在事务开始前已经存在的，要么是事务自身插入或者修改过的。
  - 行的删除版本要么未定义，要么大于当前事务版本号。这可以确保事务读取到的行，在事务开始之前未被删除。
- **INSERT** InnoDB 为新插入的每一行保存当前系统版本号作为行版本号。
- **DELETE** InnoDB 为删除的每一行保存当前系统版本号作为行删除标识。
- **UPDATE** InnoDB 为插入一行新记录，保存当前系统版本号作为行版本号，同时保存当前系统版本号到原来的行作为行删除标识。

MVCC 只在 REPEATABLE READ 和 READ COMMITTED 两个隔离级别下工作。

7 性能库 performance_schema

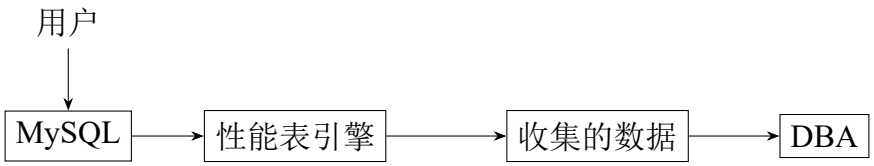
MySQL 自带四个默认数据库如下:

- mysql: 保存 MySQL 的权限、参数、对象和状态信息。
- sys: 留给专业人员排查数据库使用的表。
- information_schema: 保存数据库信息，表信息...
- performance_schema: 收集性能参数的表。

7.1 介绍

Performance Schema 提供了有关 MySQL 服务器内部运行的操作上的底层指标。理解 Performance Schema 之前有两个概念选哟了解:

- **程序插桩**: 程序插桩在 MySQL 代码中插入探测代码，以获取我们想了解的信息。
- **消费表**: 存储关于程序插桩代码信息的表。如果我们为查询模块添加插桩，相应的消费者表将记录诸如执行总数、未使用索引的次数、花费的时间等信息。



在 performance_schema 中，setup_instruments 表包含所有支持的插桩的列表。所有插桩的名称都由用斜杠分隔的部件组成。

- statement/sql/select
- wait/synch/mutex/innodb/autoinc_mutex

存放事件的表包含如下结尾:

- ***_current**: 当前服务器上进行中的事件。
- ***_history**: 每个线程最近完成的 10 个事件。
- ***_history_long**: 每个线程最近完成的 10,000 个事件。
- **event_waits**: 底层服务器等待，例如获取互斥对象。
- **events_statements**: SQL 查询语句。
- **events_stages**: 配置文件信息，例如创建临时表或发送数据。
- **events_stransactions**: 事务。

类似的，还有其他几种类型的表:

- **汇总表和摘要**: 保存有关该表所建议的内容的聚合信息。例如，memory_summary_by_thread_by_event_name 表保存了用户连接或任何后台线程的每个 MySQL 线程的聚合内存使用情况。

  摘要是一种通过删除查询中的变量来聚合查询的方法。例如以下查询：

```sql
SELECT user, birthday FROM users WHERE user_id = 19;
SELECT user, birthday FROM users WHERE user_id = 13;
SELECT user, birthday FROM users WHERE user_id = 27;
```

其摘要为:

```sql
SELECT user, birthday FROM users WHERE user_id = ?;
```

这允许 Performance Schema 跟踪摘要的延迟等指标，而无须单独保留查询的每个变体。

- **实例表 (Instance)**: 实例是指对象实例，用于 MySQL 安装程序。例如，file_instances 表包含文件名和访问这些文件的线程数。

MySQL 服务端是多线程软件。它的每个组件都使用线程。每个线程至少有两个唯一标识符：一个是操作系统线程 ID，另一个是 MySQL 内部线程 ID。

7.2 配置

Performance Schema 的部分设置只能在服务器启动时更改：比如启用或禁用 Performance Schema 本身以及与内存使用和数据收集的限制相关的变量。Performance Schema 插桩和消费者表则可以被动态启用或禁用。

启动或者禁用: 要启用或禁用 Performance Schema，可以将变量 performance_schema 设置为 ON 或 OFF。这是一个只读变量，要么在配置文件中更改，要么在 MySQL 服务器启动时通过命令行参数更改。

启用或禁用插桩: 插桩也可以被启用或禁用。可以通过 setup_instruments 表查看插桩的状态：

```
*************************** 1. row ***************************
    NAME: wait/synch/mutex/pfs/LOCK_pfs_share_list
 ENABLED: NO
   TIMED: NO
PROPERTIES: singleton
VOLATILITY: 1
DOCUMENTATION: Components can provide their own performance_schema tables. This lock protects
    the list of such tables definitions.
```

有三个方法可用于启用或禁用 performance_schema 插桩：

- 使用 setup_instruments 表。
- 调用 sys schema 中的 ps_setup_enable_instrument 存储过程。
- 使用 performance-schema-instrument 启动参数。

```sql
-- 直接修改表
UPDATE performance_schema.setup_instruments SET ENABLED = 'YES' WHERE
    NAME='statement/sql/select';
-- 使用预定义的存储过程
CALL sys.ps_setup_enable_instruments('statement/sql/select');
```

消费者表的启用，禁用方式也类似，不再赘述。

7.3 使用

检查 SQL 语句

Schema 提供了一组丰富的插桩来检查 SQL 语句的性能。你可以找到用于标准预处理语句和存储例程的插桩。使用 performance_schema，很容易找到引起性能问题的查询以及原因。

要启用语句检测，需要启用 statement 类型的插桩:

表 3.1 statement 类型的插桩

插桩类	描述
statement/sql	SQL 语句，如 SELECT
statement/sp	存储过程控制
statement/schedular	事件调度器
statement/com	命令，如 kill,quit
statement/abstract	包含四种命令: clone, query, new_packet, relay_log

这部分内容比较晦涩，不写了。

IV 优化 MySQL

这部分内容仅针对 InnoDB 引擎。

8 schema 设计与管理

首先，schema(模式) 是啥，不同的数据库有不同的定义，MySQL 在物理上将 schema 与数据库等价。从概念上讲，模式是一组相互关联的数据库对象，如表，表列，列的数据类型，索引，外键等等。

8.1 选择优化的数据类型

不管存储哪种类型的数据，下面几个简单的原则都有助于你做出更好的选择。

- **更小的通常更好**: 尽量使用能够正确存储和表示数据的最小数据类型。更小的数据类型通常更快，消耗更少的空间与时间。
- **简单为好**: 简单数据类型的操作通常需要更少的 CPU 周期。例如，整型数据比字符型数据的比较操作代价更低。
- **尽量避免存储 NULL**: 如果查询中包含可为 NULL 的列，对 MySQL 来说更难优化，因为可为 NULL 的列使得索引、索引统计和值比较都更复杂。

8.1.1 数值类型

有两种基本类型的数字：整数和实数。MySQL 为了兼容性支持很多别名，例如 INTEGER 映射到 INT，NUMERIC 映射到 DECIMAL，需要注意，并不是什么新的类型。

在使用整数类型时，有可选的 UNSIGNED 属性，可以提高正数上限的一倍，且没有性能影响。

MySQL 可以为整数类型指定宽度，例如，INT（11），这对大多数应用毫无意义：它不会限制值的合法范围，只是规定了 MySQL 的一些交互工具（例如，MySQL 命令行客户端）用来显示字符的个数。对于存储和计算来说，INT（1）和 INT（20）是相同的。

浮点类型通常比 DECIMAL 使用更少的空间来存储相同范围的值。MySQL 会使用 DOUBLE 进行浮点类型的内部计算。

由于额外的空间需求和计算成本，应该尽量只在对小数进行精确计算时才使用 DECIMAL。假设要存储财务数据并精确到万分之一分，则可以把所有金额乘以一百万，然后将结果存储在 BIGINT 里，这样可以同时避免浮点存储计算不精确和 DECIMAL 精确计算代价高的问题。

8.1.2 字符串类型

存储引擎在内存中存储 CHAR 或 VARCHAR 值的方式可能与在磁盘上存储该值的方式不同，并且服务器在从存储引擎检索该值时可能会将其转换为另一种存储格式。

- **VARCHAR** 变长字符串
  - 需要额外使用 1 或 2 字节记录字符串的长度，假设采用 latin1 字符集，一个 VARCHAR（10）的列需要 11 字节的存储空间。VARCHAR（1000）的列则需要 1002 个字节，因为需要 2 字节存储长度信息。
  - varchar 长度是以实际保存的字符串长度为准的。由于行是可变长度的，在更新时可能会增长，这会导致额外的工作。如果行的增长使得原位置无法容纳更多内容，InnoDB 可能需要分割页面来容纳行。
- **CHAR** 定长字符串
  - MySQL 总是为定义的字符串长度分配足够的空间。当存储 CHAR 值时，MySQL 删除所有尾随空格。如果需要进行比较，值会用空格填充。
  - CHAR 适合存储非常短的字符串，或者适用于所有值的长度都几乎相同的情况。
  - 对于经常修改的数据，CHAR 也比 VARCHAR 更好，因为固定长度的行不容易出现碎片。对于非常短的列，CHAR 也比 VARCHAR 更高效。

CHAR 的尾部截断问题：

```
CREATE TABLE char_test (
    char_col CHAR(10) NOT NULL,
    varchar_col VARCHAR(10) NOT NULL
);

INSERT INTO char_test(char_col, varchar_col) VALUES
    ('string1','string1'),
    (' string2',' string2'),
    ('string3 ','string3 ');

SELECT CONCAT("'",char_col,"'"),CONCAT("'",varchar_col,"'") FROM char_test;
```

执行上述代码，观察返回结果:

```
+--------------------------+-----------------------------+
| CONCAT("'",char_col,"'") | CONCAT("'",varchar_col,"'") |
+--------------------------+-----------------------------+
| 'string1'                | 'string1'                   |
| ' string2'               | ' string2'                  |
| 'string3'                | 'string3 '                  |
+--------------------------+-----------------------------+
```

发现 CHAR 类型字符串的尾部空格被截断了，而 VARCHAR 却没有。

与 CHAR 和 VARCHAR 类似的类型还有 BINARY 和 VARBINARY，它们存储的是二进制字符串。二进制字符串与常规字符串非常相似，但它们存储的是字节而不是字符。填充也不同：MySQL 填充 BINANRY 用的是零字节而不是空格，并且在检索时不会去除填充值。

当需要存储二进制数据，并且希望 MySQL 将值作为字节而不是字符进行比较时，这些类型非常有用。字节比较的优势不仅仅是大小写不敏感。MySQL 比较 BINARY 字符串时，每次按一个字节，并且根据该字节的数值进行比较。因此，二进制比较比字符比较简单得多，因此速度更快。

使用 VARCHAR（5）和 VARCHAR（200）存储'hello' 的空间开销是一样的。但较大的列会使用更多的内存，因为 MySQL 通常会在内部分配固定大小的内存块来保存值。这对于使用内存临时表的排序或操作来说尤其糟糕。

BLOB 与 TEXT 类型 BLOB 和 TEXT 都是为存储很大的数据而设计的字符串数据类型，分别采用二进制和字符方式存储。

与其他数据类型不同，MySQL 把每个 BLOB 和 TEXT 值当作一个具有自己标识的对象来处理。存储引擎通常会专门存储它们。当 BLOB 和 TEXT 值太大时，InnoDB 会使用独立的“外部”存储区域，此时每个值在行内需要 1～4 字节的存储空间，然后在外部存储区域需要足够的空间来存储实际的值。

MySQL 对 BLOB 和 TEXT 列的排序与其他类型不同：它只对这些列的最前 max_sort_length 字节而不是整个字符串做排序。MySQL 不能将 BLOB 和 TEXT 数据类型的完整字符串放入索引，也不能使用索引进行排序。

> **建议 8.1.** *在过去，某些应用程序接受上传的图像并将其作为 BLOB 数据存储在 MySQL 数据库中，这种方法便于将应用程序的数据保存在一起；但是，随着数据大小的增长，修改 schema 等操作会由于 BLOB 数据的大小而变得越来越慢。如果可以避免的话，不要在数据库中存储像图像这样的数据。相反，应该将它们写入单独的对象数据存储，并使用该表来跟踪图像的位置或文件名。*

使用枚举代替字符串类型 ENUM 列可以存储一组预定义的不同字符串值。MySQL 在存储枚举时非常紧凑，会根据列表值的数量压缩到 1 或者 2 字节中。在内部会将每个值在列表中的位置保存为整数。

ENUM 字段是根据内部整数值排序的，而不是根据字符串本身。

MySQL 将每个枚举值存储为整数，并且必须进行查找以将其转换为字符串表示，因此 ENUM 列有一些开销。这些开销通常可以被 ENUM 列的小尺寸所抵消，但并不总是如此。特别是，将 CHAR/VARCHAR 列联接到 ENUM 列可能比联接到另一个 CHAR/VARCHAR 列更慢。

8.1.3 日期和时间类型

当插入一行记录时没有指定第一个 TIMESTAMP 列的值，MySQL 会将该列的值设置为当前时间。当更新一行记录时没有指定第一个 TIMESTAMP 列的值，MySQL 默认也会将该列的值更新为当前时间。可以为任何 TIMESTAMP 列配置插入和更新行为。最后，TIMESTAMP 列在默认情况下为 NOT NULL。

8.1.4 位压缩数据类型

MySQL 有几种使用值中的单个位来紧凑地存储数据的类型。所有这些位压缩类型，不管底层存储和处理方式如何，从技术上来说都是字符串类型。

BIT 可以使用 BIT 列存储一个或多个 true/false 值。BIT（1）定义一个包含 1 位的字段，BIT（2）存储 2 位的字段，依此类推；BIT 列的最大长度为 64 位。InnoDB 将每一列存储为足够容纳这些位的最小整数类型，所以使用 BIT 列不会节省任何存储空间。

MySQL 在处理时会将 BIT 视为字符串类型，而不是数字类型。当检索 BIT（1）的值时，结果是一个包含二进制值 0 或 1 的字符串，而不是 ASCII 码的“0”或“1”。

但是，如果在数字上下文中检索该值，则会将 BIT 字符串转换为数字。

8.2 设计陷阱

太多的列

MySQL 的存储引擎 API 通过在服务器和存储引擎之间以行缓冲区格式复制行来工作；然后，服务器将缓冲区解码为列。将行缓冲区转换为具有解码列的行数据结构的操作代价是非常高的。InnoDB 的行格式总是需要转换的。这种转换的成本取决于列数。

当调查一个具有非常宽的表（数百列）的客户的高 CPU 消耗问题时，我们发现这种转换代价可能会变得非常昂贵，尽管实际上只使用了几列。如果计划使用数百列，请注意服务器的性能特征会有所不同。

太多的联接

MySQL 限制每个联接有 61 个表,即使联接数远小于 61,规划和优化查询的成本对 MySQL 来说也会成为问题。一个粗略的经验法则是，如果需要以高并发性快速执行查询，那么每个查询最好少于十几个的表。

9 高性能索引

索引，在 MySQL 中也叫作键（key），是存储引擎用于快速找到记录的一种数据结构。索引优化应该是对查询性能优化最有效的手段了。索引能够轻易将查询性能提高几个数量级，“最优”的索引有时比一个“好的”索引性能要好两个数量级。

9.1 索引基础

在 MySQL 中，先在索引结构中找到对应值，这样就可以找到包含这个值的记录。

```
1 SELECT first_name FROM sakila.actor WHERE actor_id=5;
```

如果在 actor_id 列上建了索引，则 MySQL 将使用该索引找到 actor_id 为 5 的记录。也就是说，MySQL 先在索引上按值进行查找，然后返回所有包含该值的记录。

索引可以包含一列或多列的值。如果索引包含多列，那么列的顺序也十分重要，因为 MySQL 只能有效地使用索引的最左前缀列。创建一个包含两列的索引，和创建两个只包含一列的索引是大不相同的。

9.1.1 索引的类型

MySQL 中，索引是在存储引擎层而不是服务器层实现的。所以，并没有统一的索引标准：不同存储引擎的索引的工作方式并不一样，也不是所有的存储引擎都支持所有类型的索引。

B-tree 索引 如果没有特别指明类型，那么多半说的是 B-tree 索引，它使用 B-tree 数据结构来存储数据。InnoDB 使用的是 B+tree[1]。

B-tree 索引能够加快数据访问的速度，这是因为有了索引，在查询某些条件的数据时，存储引擎不再需要进行全表扫描。而是从索引的根节点开始进行搜索，根节点的槽中存放了指向子节点的指针，存储引擎根据这些指针向下层查找。通过比较节点页的值和要查找的值可以找到合适的指针进入下层子节点，这些指针实际上定义了子节点页中值的上限和下限。最终存储引擎要么找到对应的值，要么该记录不存在。

叶子节点比较特殊，它们的指针指向的是被索引的数据，而不是其他的节点页。

B-tree 是按照索引列中的数据大小顺序存储的，所以很适合按照范围来查询。例如，在一个基于文本列的索引树上遍历，按字母顺序传递连续的值进行范围查找是非常合适的。

索引对多个值进行排序的依据是 CREATE TABLE 语句中定义索引时列的顺序。

自适应哈希索引。InnoDB 存储引擎有一个被称为自适应哈希索引的特性。当 InnoDB 发现某些索引值被非常频繁地被访问时，它会在原有的 B-tree 索引之上，在内存中再构建一个哈希索引。这个过程是完全自动化的，用户无法进行控制或者配置。不过，可以通过参数彻底关闭自适应哈希索引这个特性。

[1] InnoDB B+ 树参考文章: https://blog.csdn.net/b_x_p/article/details/86434387

可以使用 B-tree 索引的查询类型。B-tree 索引适用于全键值、键值范围或键前缀查找。其中键前缀查找只适用于根据最左前缀的查找。

全文索引 FULLTEXT 是一种特殊类型的索引，它查找的是文本中的关键词，而不是直接比较索引中的值。在相同的列上同时创建全文索引和基于值的 B-tree 索引并不会有冲突，全文索引适用于 MATCH AGAINST 操作，而不是普通的 WHERE 条件操作。

使用索引的优点 索引最主要的优点：可以让服务器快速地定位到表的指定位置。

最常见的 B-tree 索引，按照顺序存储数据，所以 MySQL 可以用来做 ORDER BY 和 GROUP BY 操作。因为数据是有序的，所以 B-tree 也就会将相关的列值都存储在一起。最后，因为索引中存储了实际的列值，所以某些查询只使用索引就能够完成全部查询。据此特性，总结下来索引有如下三个优点：

- 索引大大减少了服务器需要扫描的数据量。
- 索引可以帮助服务器避免排序和临时表。
- 索引可以将随机 I/O 变为顺序 I/O。

9.2 高性能索引策略

9.2.1 前缀索引和索引的选择性

有时候为了提升索引的性能，同时也节省索引空间，可以只对字段的前一部分字符进行索引，这样做的缺点是，会降低索引的选择性。

$$\text{选择性} = \frac{\text{不重复的索引值 (基数)}}{\text{数据表的记录总数}}$$

索引的选择性越高则查询效率越高，因为选择性高的索引可以让 MySQL 在查找时过滤掉更多的行。唯一索引的选择性是 1，这是最好的索引选择性，性能也是最好的。

一般情况下，列前缀的选择性也是足够高的，足以满足查询性能。对于 BLOB、TEXT 或者很长的 VARCHAR 类型的列，必须使用前缀索引，因为 MySQL 并不支持对这些列的完整内容进行索引。

具体选择多少个前缀创建索引要视情况而定，常用的办法是比较不使用前缀索引和使用前缀索引情况下的选择性，越接近越好，同时也需要考虑空间。

创建一个前缀索引的语句如下:

```
ALTER TABLE table ADD KEY (city(7));
```

前缀索引是一种能使索引更小、更快的有效办法，但它也有缺点：MySQL 无法使用前缀索引做 ORDER BY 和 GROUP BY 操作，也无法使用前缀索引做覆盖扫描。

9.2.2 多列索引

首先，多列索引不是为每一个列创建索引，其次多列索引的次序十分重要。

在多列上独立地创建多个单列索引，在大部分情况下并不能提高 MySQL 的查询性能。MySQL 引入了一种叫“索引合并”（index merge）的策略，它在一定程度上可以使用表中的多个单列索引来定位指定的行。在这种情况下，查询能够同时使用两个单列索引进行扫描，并将结果进行合并。这种算法有三个变种：OR 条件的联合（union），AND 条件的相交（intersection），组合前两种情况的联合及相交。

索引合并策略有时候效果非常不错，但更多的时候，它说明了表中的索引建得很糟糕。如果在 EXPLAIN 中看到有索引合并，那么就应该好好检查一下查询语句的写法和表的结构，看是不是已经是最优的。

9.2.3 选择合适的索引列顺序

在一个多列 B-tree 索引中，索引列的顺序意味着索引首先按照最左列进行排序，其次是第二列，等等。所以索引可以按照升序或者降序进行扫描，以满足精确符合列顺序的 ORDER BY、GROUP BY 和 DISTINCT 等子句的查询需求。

当不需要考虑排序和分组时，将选择性最高的列放在前面通常是很好的。这时索引的作用只是优化查询语句中的 WHERE 条件。

然而，性能不只依赖于所有索引列的选择性（整体基数），也和查询条件的具体值有关，也就是和值的分布有关。这和前面介绍的选择前缀的长度需要考虑的因素一样。可能需要根据那些运行频率最高的查询来调整索引列的顺序，让这种情况下索引的选择性最高。

举个例子:

```
1 SELECT * FROM payment WHERE staff_id = 2 AND customer_id = 584;
```

是应该创建一个（staff_id、customer_id）索引还是应该颠倒一下顺序？这时，可以通过运行某些查询来确定在这个表中值的分布情况，并确定哪列的选择性更高。

9.2.4 聚簇索引

聚簇索引并不是一种单独的索引类型，而是一种数据存储方式。当表有聚簇索引时，它的数据行实际上存放在索引的叶子页（leaf page）中。InnoDB 根据主键聚簇数据。主键的逻辑顺序就是数据存储的物理顺序。

如果你没有定义主键，InnoDB 会选择一个唯一的非空索引代替。如果没有这样的索引，InnoDB 会隐式定义一个主键来作为聚簇索引。这样做的缺点在于，所有需要使用这种隐藏主键的表都依赖一个单点的“自增值”，这可能会导致非常高的锁竞争，从而出现性能问题。

聚簇索引的每一个叶子节点都包含了主键值、事务 ID、用于事务和 MVCC 的回滚指针，以及所有的剩余列。如果主键是一个列前缀索引，InnoDB 也会包含完整的主键列和剩下的其他列。

InnoDB 的二级索引的叶子节点中存储的是主键值，并以此作为指向行的“指针”。

最好避免随机的（不连续且值的分布范围非常大）聚簇索引，特别是对于 I/O 密集型的应用。例如，从性能的角度考虑，使用 UUID 作为聚簇索引会很糟糕：它使得聚簇索引的插入变得完全随机，这就是最糟糕的情况，数据本身没有任何聚集特性。

主键顺序插入会带来高聚集的好处，但在高并发的工作负载下，在 InnoDB 中按主键顺序插入可能会造成明显的写入竞争。主键的上界会成为“热点”。因为所有的插入都发生在这里，所以并发插入可能导致间隙锁竞争。另一个热点可能是 AUTO_INCREMENT 锁机制。

9.2.5 覆盖索引

如果一个索引包含（或者说覆盖）所有需要查询的字段的值，我们就称之为覆盖索引。需要注意的是，只有 B-tree 索引可以用于覆盖索引。

覆盖索引是非常有用的工具，能够极大地提高性能。如果查询只需要扫描索引而无须回表，会带来很大的性能提升。

当执行一个被索引覆盖的查询（也叫作索引覆盖查询）时，在 EXPLAIN 的 Extra 列可以看到“Using index”的信息。

在大多数存储引擎中，索引只能覆盖那些只访问索引中部分列的查询，不过，可以更进一步优化 InnoDB。回想一下，InnoDB 的二级索引的叶子节点都包含了主键的值，这意味着 InnoDB 的二级索引可以有效地利用这些“额外”的主键列来覆盖查询。

9.2.6 使用索引扫描来做排序

MySQL 有两种方式可以生成有序的结果：通过排序操作，或者按索引顺序扫描。如果在 EXPLAIN 的输出结果中，type 列的值为“index”，则说明 MySQL 使用了索引扫描来做排序。

扫描索引本身是很快的，因为只需要从一条索引记录移动到紧接着的下一条记录。但如果索引不能覆盖查询所需的全部列，那么就不得不每扫描一条索引记录都回表查询一次对应的记录。这基本上都是随机 I/O，因此按索引顺序读取数据的速度通常要比顺序地全表扫描慢，尤其是在 I/O 密集型的应用负载上。

MySQL 可以使用同一个索引既满足排序，又用于查找行。因此，如果可能，设计索引时应该尽可能地同时满足这两项任务，这样是最好的。

只有当索引的顺序和 ORDER BY 子句的顺序完全一致，并且所有列的排序方向（倒序或正序）都一样时，MySQL 才能使用索引来对结果做排序。如果查询需要联接多张表，则只有当 ORDER BY 子句引用的字段全部在第一个表中时，才能使用索引做排序。ORDER BY 子句和查找型查询的限制是一样的：需要满足索引的最左前缀的要求，否则，MySQL 需要执行排序操作，而无法利用索引排序。

9.2.7 冗余和重复索引

MySQL 允许在相同列上创建多个相同的索引。虽然 MySQL 会抛出一个警告，但是并不会阻止你这么做。MySQL 需要单独维护重复的索引，优化器在优化查询的时候也需要逐个地进行评估，这会影响性能，同时也浪费磁盘空间。

重复索引是指在相同的列上按照相同顺序创建的相同类型的索引。有时，还是会在不经意间创建重复索引，例如下面的代码：

```
CREATE TABLE test(
    ID INT NOT NULL PRIMARY KEY,
    A INT NOT NULL,
    B INT NOT NULL,
    UNIQUE(ID),
    INDEX(ID)
);
```

先加上唯一限制，然后再加上索引以供查询使用。事实上，MySQL 的唯一限制和主键限制都是通过索引实现的，因此，上面的写法实际上在相同的列上创建了三个重复的索引。

冗余索引和重复索引有一些不同。如果创建了索引（A，B），再创建索引（A）就是冗余索引，因为这只是前一个索引的前缀索引，因此，索引（A，B）也可以当作索引（A）来使用（这种冗余只是对 B-tree 索引来说的）。但是如果再创建索引（B，A），则不是冗余索引，索引（B）也不是，因为 B 不是索引（A，B）的最左前缀列。

冗余索引通常发生在为表添加新索引的时候。例如，有人可能会增加一个新的索引（A，B）而不是扩展已有的索引（A）。还有一种情况是，将一个索引扩展为（A，ID），其中 ID 是主键，因为主键列已经包含在二级索引中了，所以这也是冗余的。

有时候出于性能方面的考虑也需要冗余索引，因为扩展已有的索引会导致其变得太大，从而影响其他使用该索引的查询的性能。

一般来说，增加索引会提高 SELECT 操作速度，但会导致 INSERT，UPDATE，DELETE 等操作的速度变慢。

9.2.8 未使用的索引

除了冗余索引和重复索引，可能还会有一些服务器永远不用的索引。这样的索引完全是累赘，建议删除。

找到未使用索引的最好办法就是使用系统数据库 performance_schema 和 sys。在 sys 数据库中，在 table_io_waits_summary_by_index_usage 视图中可以非常简单地知道哪些索引从来没有被使用过。

9.3 维护索引和表

维护表有三个主要目的：找到并修复损坏的表，维护准确的索引统计信息，减少碎片。

9.3.1 找到并修复损坏的表

如果表出现了莫须有的错误，可以尝试使用 CHECK TABLE 来检查是否发生了表损坏(有些存储引擎不支持)。如果确定表存在问题，可以使用 REPAIR TABLE 命令来修复损坏的表。如果引擎也不支持这个语句，建立换个引擎。

```
ALTER TABLE <table> ENGINE = InnoDB;
```

如果上述的方法无法恢复数据表，可以从备份中回复，或者尝试从损坏的数据文件中恢复数据。

果是 InnoDB 存储引擎的表发生了损坏，那么一定是发生了严重的错误，需要立刻调查一下原因。InnoDB 一般不会出现损坏，它的设计保证了它并不容易被损坏。如果发生了，一般要么是数据库的硬件问题，要么是由于数据库管理员的误操作，抑或是 InnoDB 本身的缺陷（不太可能）。不存在什么查询能够让 InnoDB 表损坏，也不用担心暗处有“陷阱”。如果某条查询导致 InnoDB 数据的损坏，那么一定是遇到了 bug，而不是查询的问题。