

Table of Contents

Introduction	1.1
第一章 MySQL 基础	1.2
1.1 MySQL安装与配置	1.2.1
1.2 MySQLtee命令	1.2.2
1.3 MySQL查看报错	1.2.3
1.4 MySQL字符集与校对集	1.2.4
第二章 数据库基本操作	1.3
2.1 数据库操作	1.3.1
2.2 数据表操作	1.3.2
2.3 数据表数据操作	1.3.3
第三章 数据库查询	1.4
3.1 基本查询	1.4.1
3.2 查询函数	1.4.2
3.3 分组与排序	1.4.3
3.4 复杂查询	1.4.4
3.5 合并查询	1.4.5
3.6 视图	1.4.6
第四章 数据库进阶操作	1.5
4.1 索引	1.5.1
4.2 数据库完整性	1.5.2
4.3 触发器	1.5.3
4.4 存储过程	1.5.4
第五章 数据库管理	1.6
5.1 用户管理基础	1.6.1
5.2 用户管理进阶	1.6.2
5.3 权限管理	1.6.3
5.4 维护表	1.6.4
5.5 备份与恢复	1.6.5

Introduction

这是一本由 gitbook 制作的数据库实验笔记

实验环境

MySQL 8.0、DataGrip

书籍简介

本书分为四个板块，旨在由浅入深，帮助读者快速掌握数据库的基础知识和编程技术，相信读者在耐心读完这本书后，能够像搭积木一样轻松地使用 SQL 语句完成数据库操作

制作时间

2024.5.1 ~ 2024.5.20

关于本书

My GitHub: [Cosmo Clara](#)

前言

MySQL 是最流行的关系型数据库管理系统，在 WEB 应用方面 MySQL 是最好的 RDBMS (Relational Database Management System：关系数据库管理系统) 应用软件之一。

什么是数据库？

数据库 Database 是按照数据结构来组织、存储和管理数据的仓库。

每个数据库都有一个或多个不同的 API 用于创建，访问，管理，搜索和复制所保存的数据。

我们也可以将数据存储在文件中，但是在文件中读写数据速度相对较慢。

所以，现在我们使用关系型数据库管理系统 RDBMS 来存储和管理大数据量。所谓的关系型数据库，是建立在关系模型基础上的数据库，借助于集合代数等数学概念和方法来处理数据库中的数据。

RDBMS 的特点

RDBMS 即关系数据库管理系统 (Relational Database Management System) 的特点：

- 数据以表格的形式出现
- 每行为各种记录名称
- 每列为记录名称所对应的数据域
- 许多的行和列组成一张表单
- 若干的表单组成 database

RDBMS 术语

在我们开始学习 MySQL 数据库前，让我们先了解下 RDBMS 的一些术语：

- **数据库**: 数据库是一些关联表的集合
- **数据表**: 表是数据的矩阵。在一个数据库中的表看起来像一个简单的电子表格
- **列 (字段)** :一列 (数据元素) 包含了相同类型的数据, 例如邮政编码的数据
- **行**: 一行 (元组, 或记录) 是一组相关的数据, 例如一条用户订阅的数据
- **冗余**: 存储两倍数据, 冗余降低了性能, 但提高了数据的安全性
- **主键**: 主键是唯一的。一个数据表中只能包含一个主键。你可以使用主键来查询数据

- 外键：外键用于关联两个表
- 复合键：复合键（组合键）将多个列作为一个索引键，一般用于复合索引
- 索引：使用索引可快速访问数据库表中的特定信息。索引是对数据库表中一列或多列的值进行排序的一种结构。类似于书籍的目录
- 参照完整性：参照的完整性要求关系中不允许引用不存在的实体。与实体完整性是关系模型必须满足的完整性约束条件，目的是保证数据的一致性

MySQL 安装与配置

检查电脑名称中是否含有中文字符

在安装前，你需要检查电脑(设备)名称中是否含有中文字符，如果含有中文字符，后续安装可能会失败。（修改电脑名称即可）

安装 MySQL (8.0 版本)

具体安装教程请参看我的博客：[MySQL 安装教程](#)

1. 通过 MySQL 8.0 官网下载安装包：

如图所示：

The screenshot shows the MySQL Installer 8.0.36 download page. At the top, there are tabs for 'General Availability (GA) Releases' (which is selected), 'Archives', and a help icon. Below the tabs, it says 'MySQL Installer 8.0.36'. A note states: 'Note: MySQL 8.0 is the final series with MySQL Installer. As of MySQL 8.1, use a MySQL product's MSI or Zip archive for installation. MySQL Server 8.1 and higher also bundle MySQL Configurator, a tool that helps configure MySQL Server.' Under 'Select Version:', the dropdown is set to '8.0.36'. Under 'Select Operating System:', the dropdown is set to 'Microsoft Windows'. There are two download options listed:

Download Type	Version	File Size	Action
Windows (x86, 32-bit), MSI Installer	8.0.36	2.1M	Download
Windows (x86, 32-bit), MSI Installer	8.0.36	285.3M	Download

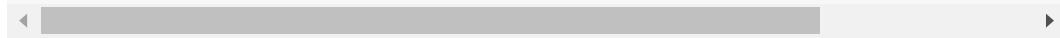
Below the download links, a note says: 'We suggest that you use the MD5 checksums and GnuPG signatures to verify the integrity of the packages you download.'

有两个下载选项，选择第二个完整的离线安装包（无 web 字样），点击 Download 进行下载。

下载后打开进入安装即可

2. 进行自定义安装：

如果想要自定义安装的路径（如D盘），那么请选择Custom，选择Custom后，根据自己需求手动选择好组件之后，需要一个一个点击右边的组件，并点击下方的 Advanced Options（高级选项）选择完安装路径后，点击 Next；
点击Execute检查运行环境；
点击Next; Yes；
点击Execute安装MySQL；
Next到"Type and Networking"这一页，开始进入配置阶段；
"Type and Networking"默认，点击Next；
选择密码类型：默认第一个即可；
Next；输入root权限的密码：第一行输入，第二行确认；
Next; Next；
点击Execute写入刚才的配置；
稍等片刻；待所有对勾打完后，点击Finish完成对MySQL服务器的配置；
Next; Finish; Next；
在Password一栏中输入刚才设置的密码，点击Check验证；
密码正确后，点击Next; Execute; Finish; Next; Finish；安装成功。



由于安装时间过于久远，此处没有图片指引，按照文字进行操作即可。

MySQL 系统环境变量配置

找到系统环境变量，在 path 中添加 MySQL 下 bin 目录的路径即可

验证 MySQL 安装是否成功

在命令行输入

```
mysql -u root -p
```

输入设定的密码，成功登入即可验证安装成功

可能遇到的问题

• 重启电脑后 MySQL 无法打开：

在计算机管理中找到服务与应用程序—服务，下拉找到 MySQL（有可能有后缀，比如我电脑上是 MySQL80），右键启动即可。

- **ERROR 1045 (28000): Access denied for user 'root'@'localhost' (using password: YES)**

密码错误

- **无法启动 MySQL:**

使用管理员命令行面板（开始搜索栏搜索 cmd，以管理员身份运行），输入

```
net start mysql;
```

查找 MySQL 全局配置文件 (my.ini)

1. 通过命令行登录 MySQL

```
mysql -u root -p  
# 输入安装时设定的密码
```

2. 查询 MySQL 数据存放目录(datadir)

```
# 若按照上述教程安装，此处应输出 D:\MySQL\MySQL Server 8.0\Data  
SELECT @@datadir;
```

修改 MySQL 配置文件内容

目的：提前把字符集设置为 `utf-8`，省去后续调整

此为 MySQL 8.0 版本，如自定义了路径(如上述安装教程)，需要在 basedir、datadir 后输入 MySQL 本体和数据存储的路径

1. 找到 my.ini 所在目录，用记事本打开，将其替换为以下内容：

```
[mysqld]
port=3306
basedir=C:\Program Files\MySQL\MySQL Server 8.0
datadir=C:\ProgramData\MySQL\MySQL Server 8.0\Data
max_connections=200
max_connect_errors=10
character-set-server=utf8
default-storage-engine=INNODB
default_authentication_plugin=mysql_native_password
skip_ssl
default-time_zone='+8:00'

[mysql]
default-character-set=utf8

[client]
port=3306
default-character-set=utf8
```

如未找到 my.ini 或其在 data 内，在桌面创建 txt 文档，修改名字与后缀为 my.ini，键入配置文件内容，放入 MySQL 本体目录下即可。

2. 若 MySQL 已经启动(一般是自动启动)，修改完配置文件需要重启 MySQL：以管理员权限打开 cmd 窗口，输入

注意：MySQL 的名字可能不同，如我的 mysql 在系统名字中为 mysql80，需要将上述为 mysql 的地方改为 mysql80

```
net stop mysql
net start mysql
```

命令行登入测试即可

MySQL tee(\T)命令

日志部分在第五章中介绍，涉及到 MySQL 的备份等操作。在此之前，如有必要，我们使用 MySQL 的 tee 命令来记录操作及运行结果。

Linux tee 命令

Linux tee 命令用于读取标准输入的数据，并将其内容输出成文件。tee 指令会从标准输入设备读取数据，将其内容输出到标准输出设备，同时保存成文件，其语法如下所示。

```
tee [-ai][--help][--version][文件...]
```

参数详解：

- -a 或 --append：附加到既有文件的后面，而非覆盖它。
- -i 或 --ignore-interrupts：忽略中断信号。
- --help：在线帮助。
- --version：显示版本信息。

MySQL tee 命令

MySQL 扩展了 tee 命令，采用 tee 命令将远程执行结果保存到本地，具体可以采用以下两种形式。

1. 在 MySQL 启动时添加参数：--tee = [your file]

```
mysql --tee = D/record.txt -u root -p
```

2. 在 MySQL 启动后，使用 tee 命令：tee [your file]; 开启记录功能，使用 note 命令：note; 关闭记录功能

```
# 也可以使用\T D/record.txt
tee D/record.txt
# 中间的操作命令和执行结果都会保存到指定的文件record.txt中
# 也可使用 \t
note;
```

尝试使用 tee 命令

在登入后使用 tee 命令记录操作记录

1. 登入 MySQL

```
mysql -u root -p
```

2. 使用 `tee` 命令指定输出文件路径

```
\T D:/I-X/Extel/record.txt
```

3. 查看已有数据库

```
SHOW DATABASES;
```

4. 使用 `notee` 关闭输出

```
\t  
# 退出MySQL  
\q
```

5. 查看 [记录文件](#)

MySQL查看报错

显示执行语句所产生的错误信息：

```
SHOW ERRORS;  
# 查看错误条数  
SHOW COUNT(*) ERRORS;
```

查看上一步操作产生的警告信息：

在 MySQL 5 后大部分以前的 `Warning` 都直接显示为 `error`，所以一般用 `SHOW ERRORS` 即可

```
SHOW WARNINGS;
```

MySQL 字符集与校对集

本章为知识点，无实验操作

字符集：用来定义 MySQL 存储字符串的方式

- 查看字符集信息

```
SHOW CHARSET;
```

- 常用字符集

字符集	最大长度	支持的语言字符
latin1	1字节	西欧字符
gbk	2字节	简体和繁体中文、日文、韩文等
utf8	3字节	世界上绝大多数国家的文字

校对集：用来定义比较字符串的方式，比如是否区分大小写等

- 查看校对集信息：

```
SHOW COLLATION;
```

- 校对集组成：字符集名称_国家名 / general_ci / cs / bin

- ci : 不区分大小写
- cs : 区分大小写
- bin : 以二进制方式比较

字符集和校对集的设置

- 查看字符集相关变量：

```
SHOW VARIABLES LIKE 'character%';
```

Variable_name变量名	说明
character_set_client	客户端字符集
character_set_connection	客户端和服务器连接用的字符集
character_set_database	默认数据库使用的字符集
character_set_filesystem	文件系统字符集
character_set_results	将查询结果返回给客户端使用的字符集
character_set_server	服务器默认字符集
character_set_system	服务器用来存储标识符的字符集
character_set_dir	安装字符集的目录

• 设置字符集:

可在创建数据库、数据表时指明数据库、数据表、字段的字符集和校对集，通过 `SET` 变量名 = 值; 来更改，**修改只对当前会话有效**

```
# 将服务器默认字符集设置为utf8;
SET character_server = utf8;
```

同时更改客户机、链接和返回结果字符串：

```
SET NAMES 字符集名;
```

SQL语言

Structure Query Language (结构化查询语言)：用于访问和处理关系数据库的标准的计算机语言

本章为知识点，无实验操作

分类

1. DDL：数据库、数据表操作

Data Definition Language——数据定义语言, 用来定义数据库对象：数据库、表、列等。

关键字： CREATE , DROP , ALTER 等

2. DML：增删改动表中的数据

Data Manipulation Language——数据操纵语言, 用来对数据库中表的数据进行增删改动。

关键字： INSERT , DELETE , UPDATE 等

3. DQL：查询表中的数据

Data Query Language——数据查询语言, 用来查询数据库中表的记录（数据）。

关键字： SELECT

4. DCL, TCL：管理用户，授权及事务

Data Control Language——数据控制语言 / Transaction Control Language——事务控制语言, 用来定义数据库的访问权限和安全级别，及创建用户。

关键字： GRANT , REVOKE , ROLLBACK , COMMIT 等

数据类型

• 查看信息：

- HELP 类型名 (或者用 \? 类型名)

• 数值类型

- 整数

TINYINT , SMALLINT , MEDIUMINT , INT , BIGINT

- 补零：数据类型后加 `unsigned zerofill`

◦ 小数

- 浮点数：`FLOAT`、`DOUBLE`

 | `FLOAT (M,D)` 与 `DOUBLE (M,D)`：M 表示数字总位数，D 表示小数点后的位数

- 定点数：`DECIMAL`

 | `DECIMAL (M,D)`：M 表示数字总位数，D 表示小数点后的位数

◦ 时间和日期类型

- `YEAR`：年份，`YYYY`
- `DATE`：年月日，`YYYY-MM-DD`
- `TIME`：时分秒，`hh:mm:ss`
- `DATETIME`：年月日时分秒，`YYYY-MM-DD hh:mm:ss`
- `TIMESTAMP`：时间戳类型，格式与 `DATETIME` 相同

 | 若不给该字段赋值或赋值为 `null`，则默认使用当前的系统时间对其进行自动赋值

◦ 字符串类型

- `CHAR`：定长字符串，需指明长度 `CHAR(M)`
- `VARCHAR`：变长字符串，需指明最大长度 `VARCHAR(M)`
- `TEXT`：大文本数据（也叫富文本）
- `ENUM`：枚举对象
- `SET`：字符串对象

语法

1. SQL 语句可以单行或多行书写，以分隔符（常用分号；）结尾。

2. **SQL 语句不区分大小写，但强烈建议遵循规范：**

 | 关键字和函数名称全部大写，数据库名称、表名称、字段名称等全部小写

注释

1. 单行注释：`--`（注意有一个空格）注释内容 或 `#`（这里不需要空格）注释内容（MySQL 特有）

2. 多行注释： /*注释*/

数据库的操作

实验 **数据库 test3 源文件** (由 DataGrip 导出, 可导入数据库)

查看数据库 SHOW

• 查看已有数据库

```
SHOW DATABASES;
```

```
SHOW DATABASES;
```

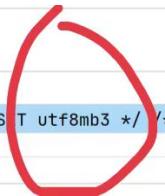
	Database
1	information_schema
2	mysql
3	performance_schema
4	sys
5	test1
6	test2
7	test3

• 查看数据库创建信息

```
# test3是已存在的数据库  
SHOW CREATE DATABASE test3;
```

```
# test3是已存在的数据库  
SHOW CREATE DATABASE test3;
```

```
Database : Create Database  
1 test3   CREATE DATABASE `test3` /*!40100 DEFAULT CHARACTER SET utf8mb3 */ /*!80016 DEFAL
```



创建数据库 CREATE

- 创建数据库

```
# 使用默认字符集(已经通过修改配置文件修改默认字符集为utf8了)
```

```
CREATE DATABASE test4;
```

- 判断数据库不存在再创建，并指定字符集和校对集：

```
CREATE DATABASE IF NOT EXISTS test5
```

```
CHARACTER SET utf8
```

```
COLLATE utf8_general_ci;
```

- 查看已有数据库来验证操作是否成功

```
SHOW DATABASES;
```

165

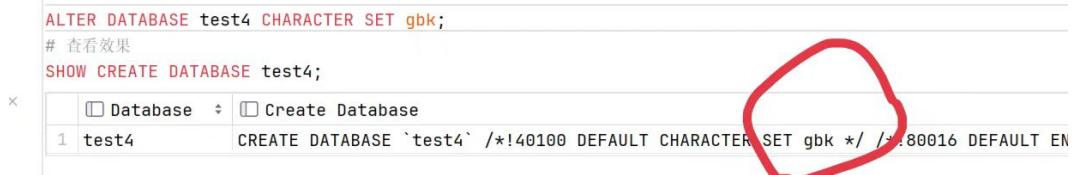
SHOW DATABASES;

	Database
1	information_schema
2	mysql
3	performance_schema
4	sys
5	test1
6	test2
7	test3
8	test4
9	test5

修改数据库 ALTER

- 修改数据库的字符集：

```
ALTER DATABASE test4 CHARACTER SET gbk;  
# 查看效果  
SHOW CREATE DATABASE test4;
```



```
ALTER DATABASE test4 CHARACTER SET gbk;  
# 查看效果  
SHOW CREATE DATABASE test4;  
+-----+-----+  
| Database | Create Database |  
+-----+-----+  
| 1 test4 | CREATE DATABASE `test4` /*!40100 DEFAULT CHARACTER SET gbk */ /*!80016 DEFAULT EN
```

选择数据库 USE

- 选择使用的数据库

```
USE test4;
```

- 查看当前正在使用的数据库

```
SELECT DATABASE();
```



```
USE test4;  
  
SELECT DATABASE();  
+-----+  
| `DATABASE()` |  
+-----+  
| test4 |
```

删除数据库 DROP

- **删除数据库如果存在**

```
DROP DATABASE IF EXISTS test4;  
# 查看效果  
SHOW DATABASES;
```

175 **DROP DATABASE IF EXISTS test4;**
176
177 ✓ **SHOW DATABASES;**

	Database
1	information_schema
2	mysql
3	performance_schema
4	sys
5	test1
6	test2
7	test3
8	test5

178

数据表操作

查看数据表 SHOW / DESC

- 查看当前数据库中已有的数据表:

```
# 先选择使用的数据库( test3 中已建好表)
USE test3;
SHOW TABLES;
```

```
USE test3;
SHOW TABLES;
```

Tables_in_test3	
1	courses
2	records
3	students
4	teachers

- 查看指定表结构:

```
# DESC 是 DESCRIBE 的简写, 也可完整写出进行查看
DESC students;
# 效果相同
SHOW COLUMNS FROM students;
# 显示完整表结构
SHOW FULL COLUMNS FROM students;
```

```

58 # DESC是DESCRIBE的简写，也可完整写出进行查看
59 DESC students;
60
61 # 效果相同
62 SHOW COLUMNS FROM students;
63
64 # 显示完整表结构
65 SHOW FULL COLUMNS FROM students;

```

Field	Type	Null	Key	Default	Extra
stu_id	int	NO	PRI	<null>	
sname	varchar(8)	YES		<null>	
sgender	char(1)	YES		<null>	
sage	tinyint	YES		<null>	
sclass	tinyint	YES		<null>	
smajor	varchar(16)	YES		<null>	

Field	Type	Collation	Null	Key	Default	Extra	Privileges	Comment
stu_id	int	<null>	NO	PRI	<null>		select,insert,update,references	
sname	varchar(8)	utf8mb3_general_ci	YES		<null>		select,insert,update,references	C
sgender	char(1)	utf8mb3_general_ci	YES		<null>		select,insert,update,references	C
sage	tinyint	<null>	YES		<null>		select,insert,update,references	C
sclass	tinyint	<null>	YES		<null>		select,insert,update,references	C
smajor	varchar(16)	utf8mb3_general_ci	YES		<null>		select,insert,update,references	C

168:1 CRLF UTF-8 4个空格

• 查看数据表创建语句

```
SHOW CREATE TABLE students;
```

```
SHOW CREATE TABLE students;
```

Table	Create Table
1 students	CREATE TABLE `students` (`stu_id` int NOT NULL, `sname` var

创建数据表 CREATE

创建数据表语法：

```

CREATE TABLE 表名称(
    列名1 数据类型1,
    列名2 数据类型2,
    ...
    列名n 数据类型n
    # 注意：最后一列，不需要加逗号
);

```

创建数据表：

- 判断不存在创建数据表

```
CREATE TABLE IF NOT EXISTS test1(
    id INT,
    name VARCHAR(20)
);
# 验证是否成功创建
SHOW TABLES;
DESC test1;
```

```
CREATE TABLE IF NOT EXISTS test1(
    id INT,
    name VARCHAR(20)
);
# 验证是否成功创建
SHOW TABLES;
+-----+
| Tables_in_test3 |
+-----+
| 1 courses      |
| 2 records       |
| 3 students      |
| 4 teachers      |
| 5 test1         |
+-----+
DESC test1;
+-----+
| Field   | Type    | Null | Key | Default | Extra |
+-----+
| 1 id     | int     | YES  |      | <null>  |        |
| 2 name   | varchar | YES  |      | <null>  |        |
+-----+
```

- 在创建数据表时指定字符集

由于修改了配置文件，默认为 `utf8`，为展示效果指定为 `gbk`

```
CREATE TABLE IF NOT EXISTS test2(
    id INT,
    name VARCHAR(20)
) CHARSET gbk;
SHOW FULL COLUMNS FROM test2;
```

```
CREATE TABLE IF NOT EXISTS test2(
    id INT,
    name VARCHAR(20)
) CHARSET gbk;
SHOW FULL COLUMNS FROM test2;
+-----+
| Field | Type    | Collation | Null | Key | Default | Extra |
+-----+
| 1 id   | int     | <null>    | YES  |      | <null>  |        |
| 2 name | varchar | gbk_chinese_ci | YES  |      | <null>  |        |
+-----+
```

- 在创建数据表时为列添加备注信息

```
CREATE TABLE IF NOT EXISTS test3(
    id INT COMMENT '学号',
    name VARCHAR(20) COMMENT '姓名'
);
SHOW FULL COLUMNS FROM test3;
```

```
CREATE TABLE IF NOT EXISTS test3(
    id INT COMMENT '学号',
    name VARCHAR(20) COMMENT '姓名'
);
SHOW FULL COLUMNS FROM test3;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| Field | Type | Collation | Null | Key | Default | Extra | Privileges | Comment |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id   | int  | <null>    | YES  |   | <null>  |       | select,insert,update,references | 学号   |
| name | varchar(20) | utf8mb3_general_ci | YES  |   | <null>  |       | select,insert,update,references | 姓名   |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

158:13 CRLF UTF-8

修改数据表 ALTER

字段名：数据表的列名

- 修改字符集 CHARSET

```
# CHARSET 是 CHARACTER SET的缩写
ALTER TABLE test2 CHARSET utf8;
SHOW CREATE TABLE test2;
```

```
ALTER TABLE test2 CHARSET utf8;
SHOW CREATE TABLE test2;
```

	1 JE=InnoDB DEFAULT CHARSET=utf8mb3
--	---------------------------------------

- 修改表名称 RENAME

```
ALTER TABLE test2 RENAME AS test4;
SHOW TABLES;
# 等效
RENAME TABLE test4 to test0;
SHOW TABLES;
```

```
ALTER TABLE test2 RENAME AS test4;  
SHOW TABLES;
```

	☐ Tables_in_test3
1	courses
2	records
3	students
4	teachers
5	test1
6	test3
7	test4

```
RENAME TABLE test4 TO test0;  
SHOW TABLES;
```

	☐ Tables_in_test3
1	courses
2	records
3	students
4	teachers
5	test0
6	test1
7	test3

- 新增字段 ADD

```
# 在最后添加一个字段:  
ALTER TABLE test2 ADD gender VARCHAR(4);  
DESC test2;  
# 在最前添加一个字段  
ALTER TABLE test2 ADD major VARCHAR(8) FIRST;  
DESC test2;  
# 一次添加多个字段:  
ALTER TABLE test2 ADD(  
    class INT,  
    country VARCHAR(16)  
)  
DESC test2;  
# 指定字段后添加字段:  
ALTER TABLE test2 ADD phone_number INT AFTER class;  
DESC test2;
```

```

DESC test2;



|   | Field | Type        | Null |
|---|-------|-------------|------|
| 1 | id    | int         | YES  |
| 2 | name  | varchar(20) | YES  |



ALTER TABLE test2 ADD gender VARCHAR(4);

DESC test2;



|   | Field  | Type        | Null |
|---|--------|-------------|------|
| 1 | id     | int         | YES  |
| 2 | name   | varchar(20) | YES  |
| 3 | gender | varchar(4)  | YES  |



ALTER TABLE test2 ADD major VARCHAR(8) FIRST;

DESC test2;



|   | Field  | Type        | Null |
|---|--------|-------------|------|
| 1 | major  | varchar(8)  | YES  |
| 2 | id     | int         | YES  |
| 3 | name   | varchar(20) | YES  |
| 4 | gender | varchar(4)  | YES  |



ALTER TABLE test2 ADD(
    class INT,
    country VARCHAR(16)
);

DESC test2;



|   | Field   | Type        | Null |
|---|---------|-------------|------|
| 1 | major   | varchar(8)  | YES  |
| 2 | id      | int         | YES  |
| 3 | name    | varchar(20) | YES  |
| 4 | gender  | varchar(4)  | YES  |
| 5 | class   | int         | YES  |
| 6 | country | varchar(16) | YES  |



ALTER TABLE test2 ADD phone_number INT AFTER class;

DESC test2;



|   | Field        | Type        | Null |
|---|--------------|-------------|------|
| 1 | major        | varchar(8)  | YES  |
| 2 | id           | int         | YES  |
| 3 | name         | varchar(20) | YES  |
| 4 | gender       | varchar(4)  | YES  |
| 5 | class        | int         | YES  |
| 6 | phone_number | int         | YES  |
| 7 | country      | varchar(16) | YES  |


```

- **修改字段 CHANGE / MODIFY**

- 修改字段名:

```
ALTER TABLE test2 CHANGE major email VARCHAR(16);  
DESC test2;
```

```
ALTER TABLE test2 CHANGE major email VARCHAR(16);  
DESC test2;|
```

	Field	Type	Null	Key
1	email	varchar(16)	YES	
2	id	int	YES	
3	name	varchar(20)	YES	
4	gender	varchar(4)	YES	
5	class	int	YES	
6	phone_number	int	YES	
7	country	varchar(16)	YES	

- 修改字段类型:

```
ALTER TABLE test2 MODIFY name VARCHAR(16);  
DESC test2;
```

```
ALTER TABLE test2 MODIFY name VARCHAR(16);  
DESC test2;|
```

	Field	Type	Null	Key
1	email	varchar(16)	YES	
2	id	int	YES	
3	name	varchar(16)	YES	
4	gender	varchar(4)	YES	
5	class	int	YES	
6	phone_number	int	YES	
7	country	varchar(16)	YES	

- 修改字段位置:

```
ALTER TABLE test2 MODIFY id INT FIRST;
DESC test2;
# 指定字段后添加字段:
ALTER TABLE test2 MODIFY email VARCHAR(16) AFTER phone_number;
DESC test2;
```

	Field	Type	Null	Key	
1	id	int	YES		<
2	email	varchar(16)	YES		<
3	name	varchar(16)	YES		<
4	gender	varchar(4)	YES		<
5	class	int	YES		<
6	phone_number	int	YES		<
7	country	varchar(16)	YES		<

	Field	Type	Null	Key	
1	id	int	YES		<
2	name	varchar(16)	YES		<
3	gender	varchar(4)	YES		<
4	class	int	YES		<
5	phone_number	int	YES		<
6	email	varchar(16)	YES		<
7	country	varchar(16)	YES		<

- 修改字段字符集:

```
SHOW FULL COLUMNS FROM test2;
ALTER TABLE test2 MODIFY name VARCHAR(16) CHARSET gbk;
SHOW FULL COLUMNS FROM test2;
```

```
SHOW FULL COLUMNS FROM test2;
```

	Field	Type	Collation	Null	
1	id	int	<null>	YES	
2	name	varchar(16)	utf8mb3_general_ci	YES	
3	gender	varchar(4)	utf8mb3_general_ci	YES	
4	class	int	<null>	YES	
5	phone_number	int	<null>	YES	
6	email	varchar(16)	utf8mb3_general_ci	YES	
7	country	varchar(16)	utf8mb3_general_ci	YES	

```
ALTER TABLE test2 MODIFY name VARCHAR(16) CHARSET gbk;  
SHOW FULL COLUMNS FROM test2;
```

	Field	Type	Collation	Null	
1	id	int	<null>	YES	
2	name	varchar(16)	gbk_chinese_ci	YES	
3	gender	varchar(4)	utf8mb3_general_ci	YES	
4	class	int	<null>	YES	
5	phone_number	int	<null>	YES	
6	email	varchar(16)	utf8mb3_general_ci	YES	
7	country	varchar(16)	utf8mb3_general_ci	YES	

- **删除字段** `DROP`

```
# `COLUMN` 可省略  
ALTER TABLE test2 DROP COLUMN country;  
  
DESC test2;  
  
# 删除多个字段  
ALTER TABLE test2  
    DROP gender,  
    DROP class,  
    DROP email,  
    DROP phone_number;  
  
DESC test2;
```

```

ALTER TABLE test2 DROP COLUMN country;
DESC test2;



|   | Field        | Type        | Null | Key | Default |
|---|--------------|-------------|------|-----|---------|
| 1 | id           | int         | YES  |     | <null>  |
| 2 | name         | varchar(16) | YES  |     | <null>  |
| 3 | gender       | varchar(4)  | YES  |     | <null>  |
| 4 | class        | int         | YES  |     | <null>  |
| 5 | phone_number | int         | YES  |     | <null>  |
| 6 | email        | varchar(16) | YES  |     | <null>  |


```

```

✓ ALTER TABLE test2
  DROP gender,
  DROP class,
  DROP email,
  DROP phone_number;
✓ DESC test2;

```

```


|   | Field | Type        | Null | Key | Default |
|---|-------|-------------|------|-----|---------|
| 1 | id    | int         | YES  |     | <null>  |
| 2 | name  | varchar(16) | YES  |     | <null>  |


```

删除数据表 DROP

```

DROP TABLE IF EXISTS test2;
SHOW TABLES;
# 一次性删除多个数据表
DROP TABLE test1, test3;
SHOW TABLES;

```

```
DROP TABLE IF EXISTS test2;  
SHOW TABLES;
```

	☐ Tables_in_test3
1	courses
2	records
3	students
4	teachers
5	test1
6	test3

```
# 一次性删除多个数据表  
DROP TABLE test1, test3;  
SHOW TABLES;
```

	☐ Tables_in_test3
1	courses
2	records
3	students
4	teachers

注意，数据表可以一次删除多个，但数据库只能一个一个删除

数据表数据操作

由于在这一部分中涉及到数据的添加，这受到数据库完整性（主键，外键）约束，比如外键字段对主表字段的依赖性，因此不使用存在约束的数据库 `test3`，使用在 2.1 节中创建的空数据库 `test5`，在 `test5` 中创建数据表 `teachers`

```
CREATE TABLE teachers
(
    tch_id  INT ,
    tname   VARCHAR(8),
    tgender CHAR,
    tage    TINYINT
);
```

当然啦，要看到这一节的可视化效果，我们需要知道一条查看数据表数据的 sql 查询语句：

```
SELECT * FROM teachers;
```

本章就只使用这条查询语句来展示对数据表数据的操作

```

USE test5;

CREATE TABLE teachers
(
    tch_id  INT ,
    tname   VARCHAR(8),
    tgender CHAR,
    tage    TINYINT
);

SHOW TABLES;
+-----+
| Tables_in_test5 |
+-----+
| 1 teachers      |
+-----+

DESC teachers;
+-----+
| Field     | Type   | Null | Key | Default |
+-----+
| 1 tch_id   | int    | YES  |     | <null>  |
| 2 tname    | varchar(8) | YES  |     | <null>  |
| 3 tgender  | char(1) | YES  |     | <null>  |
| 4 tage    | tinyint | YES  |     | <null>  |
+-----+

SELECT * FROM teachers;
+-----+
| tch_id | tname | tgender | tage |
+-----+

```

插入数据 INSERT INTO + VALUES / SET

- 添加一条记录

除了数字类型，其他类型的值需要使用引号(单双都可以)引起来

```

# 可以省略表名后的字段名，图中就是省略字段名的情况
INSERT INTO teachers (tch_id, tname, tgender, tage)
VALUES
(101, '小李', '女', 31);
SELECT * FROM teachers;

```

```

5  INSERT INTO teachers
6      VALUES
7          (101, '小李', '女', 31);
3 ✓  SELECT * FROM teachers;
+-----+
| tch_id | tname | tgender | tage |
+-----+
| 1      | 101   | 小李    | 女    | 31   |
+-----+

```

• 一次添加多条记录

```
INSERT INTO teachers (tch_id, tname, tgender, tage)
VALUES
    (102, '小宋', '男', 46),
    (103, '小黑', '女', 33),
    (104, '小唐', '男', 42),
    (105, '小江', '男', 51),
    (106, '小白', '女', 27);
SELECT * FROM teachers;
```

```
INSERT INTO teachers (tch_id, tname, tgender, tage)
VALUES
    (102, '小宋', '男', 46),
    (103, '小黑', '女', 33),
    (104, '小唐', '男', 42),
    (105, '小江', '男', 51),
    (106, '小白', '女', 27);
SELECT * FROM teachers;
```

	tch_id	tname	tgender	tage
1	101	小李	女	31
2	102	小宋	男	46
3	103	小黑	女	33
4	104	小唐	男	42
5	105	小江	男	51
6	106	小白	女	27

• 通过 SET 形式插入记录 (不推荐)

```
INSERT INTO teachers
SET
    tch_id = 107,
    tname = '小红',
    tgender = '女',
    tage = 33;
SELECT * FROM teachers;
```

```
INSERT INTO teachers
SET
    tch_id = 107,
    tname = '小红',
    tgender = '女',
    tage = 33;
SELECT * FROM teachers;
```

	tch_id	tname	tgender	tage
1	101	小李	女	31
2	102	小宋	男	46
3	103	小黑	女	33
4	104	小唐	男	42
5	105	小江	男	51
6	106	小白	女	27
7	107	小红	女	33

修改数据 UPDATE + SET

- 使用 WHERE 条件表达式修改

```
UPDATE 表名 SET
    字段名1=值1,
    字段名2=值2,
    ...,
    WHERE 条件表达式;
```

如果不加 WHERE 条件，将修改全部数据

- 不使用 WHERE 条件表达式修改全表数据

```
UPDATE 表名 SET
    字段名1=值1,
    字段名2=值2,
    ...;
```

```

UPDATE teachers
SET
    tage = 36
WHERE tch_id = 107;
SELECT * FROM teachers;

```

	tch_id	tname	tgender	tage
1	101	小李	女	31
2	102	小宋	男	46
3	103	小黑	女	33
4	104	小唐	男	42
5	105	小江	男	51
6	106	小白	女	27
7	107	小红	女	36

```

UPDATE teachers
SET
    tage = 36;
SELECT * FROM teachers;

```

	tch_id	tname	tgender	tage
1	101	小李	女	36
2	102	小宋	男	36
3	103	小黑	女	36
4	104	小唐	男	36
5	105	小江	男	36
6	106	小白	女	36
7	107	小红	女	36

删除数据 DELETE / TRUNCATE

为了比较 `DELETE` 和 `TRUNCATE` 的区别，我们创建一个新的数据表 `teacher`，和 `teachers` 表删去 107 后一模一样

```

CREATE TABLE teacher
(
    tch_id  INT ,
    tname   VARCHAR(8),
    tgender CHAR,
    tage    TINYINT
);
INSERT INTO teacher (tch_id, tname, tgender, tage)
VALUES
    (101, '小李', '女', 31),
    (102, '小宋', '男', 46),
    (103, '小黑', '女', 33),
    (104, '小唐', '男', 42),
    (105, '小江', '男', 51),
    (106, '小白', '女', 27);

```

• 删除数据 —— 相当于一列一列清除

```

DELETE FROM teachers WHERE tch_id= 107 ;
SELECT * FROM teachers;
# 不加WHERE 清除所有数据
DELETE FROM teachers;
SELECT * FROM teachers;

```

x	<input type="checkbox"/> tch_id	<input type="checkbox"/> tname	<input type="checkbox"/> tgender	<input type="checkbox"/> tage
1	101	小李	女	36
2	102	小宋	男	36
3	103	小黑	女	36
4	104	小唐	男	36
5	105	小江	男	36
6	106	小白	女	36

x	<input type="checkbox"/> tch_id	<input type="checkbox"/> tname	<input type="checkbox"/> tgender	<input type="checkbox"/> tage
---	---------------------------------	--------------------------------	----------------------------------	-------------------------------

• 清空数据表 —— 相当于删除表，重建新的同名空表

```
TRUNCATE TABLE teacher;  
SELECT * FROM teacher;
```

```
✓ TRUNCATE TABLE teacher;  
✓ SELECT * FROM teacher;  
< ┌─────────┬─────────┬─────────┐  
   └── tch_id ──────────┴── tname ──────────┴── tgender ──────────┴── tage ──────────┘
```

• DELETE 和 TRUNCATE 差异

DELETE 和 TRUNCATE 的区别是：

- DELETE 是逐行删除数据，而 TRUNCATE 是清空数据表，重建新的同名空表
- DELETE 支持回滚， TRUNCATE 不支持回滚
- DELETE 会触发触发器， TRUNCATE 不会触发触发器

从下图中可以看出， DELETE 删除表的速度比 TRUNCATE 快很多，这是因为此时数据量小，重建表的时间大于逐行删除的时间，当数据量大时，使用 TRUNCATE 会明显提升速度，但是 TRUNCATE 不支持回滚，因此要小心地使用 TRUNCATE 。

```
test5> DELETE FROM teachers  
[2024-04-08 17:11:01] 8 ms 中有 6 行受到影响  
test5> TRUNCATE TABLE teacher  
[2024-04-08 17:11:10] 在 22 ms 内完成
```

查询准备

设计数据库

我们以暗区突围这款游戏为例，数据库设计如下：

- `players` 表：玩家 id, 姓名, 性别, 注册时间, 仓库 id, 等级
- `items` 表：物品 id, 物品名字, 重量, 空间, 价值, 物品信息
- `warehouse` 表：仓库 id, 物品 id, 物品数量, 物品占用空间大小
- `records` 表：记录 id, 卖家 id, 物品 id, 物品数量(`TINYINT` , 利用数据库本身进行单次上架数目的限制), 物品价格, 上架时间, 下架时间, 买家 id
- `market` 表：市场物品 id, 物品价格

需要注意的是，物品的价格不等同于其价值，`market` 表为动态表，其物品价格依赖于 `record` 表中的物品价格字段，取其最小值，当 `records` 表中发生变化时（物品被购买，我们规定玩家只会购买市场中最低价的物品，实际上在这个实验设计中，玩家也只能看到市场中最低价的物品），我们利用参照完整性对 `market` 表进行同步修改（重新查询 `records` 表，获取最新的最低物品价格））

```

CREATE DATABASE game;

CREATE TABLE players
(
    p_id          INT,
    p_name        VARCHAR(32),
    p_gender      CHAR,
    registration_time DATE,
    warehouse_id  INT,
    p_grade       INT
);

CREATE TABLE items
(
    i_id          INT,
    i_name        VARCHAR(16),
    i_weight      INT,
    i_space       INT,
    i_value       INT,
    i_information TEXT,
    i_class       VARCHAR(16)
);

CREATE TABLE records
(
    r_id          INT,
    seller_id    INT,
    ri_id         INT,
    ri_num        TINYINT,
    ri_price      INT,
    shelf_time   DATE,
    closing_time DATE,
    buyer_id     INT
);

CREATE TABLE market
(
    mi_id        INT,
    mi_price     INT
);

INSERT INTO players (p_id, p_name, p_gender, registration_time, warehouse_id,

```

```

VALUES
(10001, '医疗兵_001', '女', '2022-10-10', 10001, 24),
(10002, '突击兵_106', '男', '2023-7-2', 10002, 23),
(10003, '弗雷德', '男', '2023-5-2', 10003, 29),
(10004, '雷诺伊尔', '男', '2023-3-7', 10004, 29),
(10005, '多斯', '男', '2022-12-20', 10005, 30),
(10006, '杰克逊', '男', '2023-1-7', 10006, 23),
(10007, '寄术大师', '男', '2022-4-7', 10007, 30),
(10008, '无敌CS大王', '女', '2023-10-21', 10008, 27),
(10009, '沃伦', '男', '2022-12-10', 10009, 30),
(10010, '尤文', '男', '2022-1-1', 10010, 30),
(10011, '阿贾克斯', '男', '2022-1-1', 10011, 30),
(10012, '德文潘', '男', '2022-1-2', 10012, 30),
(10013, '兰德尔', '男', '2023-9-24', 10013, 25),
(10014, '罗尔夫', '男', '2023-11-5', 10014, 21),
(10015, '卡尔', '男', '2023-7-3', 10015, 25),
(10016, '思密达', '男', '2023-12-5', 10016, 22),
(10017, '科特', '男', '2023-11-5', 10017, 23),
(10018, '萌新', '女', '2024-1-13', 10018, 17);

```

```

INSERT INTO items (i_id, i_name, i_weight, i_space, i_value, i_class)
VALUES
(1001, '花瓶', 5, 4, 40, '收藏品'),
(1002, '保温水壶', 1, 2, 1, '生活用品'),
(1003, '茶壶', 5, 4, 40, '收藏品'),
(1004, '金豹雕像', 4, 2, 2, '收藏品'),
(1005, '金杯', 4, 2, 4, '收藏品'),
(1006, '金魔方', 4, 1, 5, '收藏品'),
(1007, '机密文件', 1, 2, 600, '收藏品'),
(1008, '金狮雕像', 9, 6, 42, '收藏品'),
(1009, '金块', 4, 2, 5, '收藏品'),
(1010, '金手镯', 3, 1, 2, '收藏品'),
(1011, '金手表', 3, 1, 2, '收藏品'),
(1012, '银徽章', 2, 1, 1, '纪念品'),
(1013, '银片手链', 2, 1, 1, '收藏品'),
(1014, '钻戒', 1, 1, 2, '收藏品'),
(1015, '红色火药', 1, 2, 1, '易燃物'),
(1016, '绿色火药', 1, 2, 1, '易燃物'),
(1017, '蓝色火药', 1, 2, 1, '易燃物'),
(1018, '主客房钥匙', 1, 1, 27, '钥匙'),
(1019, '201钥匙', 1, 1, 6, '钥匙'),

```

```
(1020, '别墅钥匙', 1, 1, 8, '钥匙'),  
(1021, '储藏室钥匙', 1, 1, 4, '钥匙'),  
(1022, '民宅钥匙', 1, 4, 1, '钥匙'),  
(1023, '北村住宅钥匙', 1, 1, 1, '钥匙'),  
(1024, '马厩钥匙', 1, 1, 2, '钥匙'),  
(1025, '海滨别墅钥匙', 1, 1, 8, '钥匙'),  
(1026, '车库钥匙', 1, 1, 3, '钥匙'),  
(1027, '墓地钥匙', 1, 1, 8, '钥匙'),  
(1028, '一楼休息室钥匙', 1, 1, 10, '钥匙'),  
(1029, '武器贮藏室钥匙', 1, 1, 16, '钥匙'),  
(1030, '水闸房钥匙', 1, 1, 11, '钥匙');
```

```
INSERT INTO market (mi_id, mi_price)  
VALUES  
(1001, 44),  
(1002, 1),  
(1003, 47),  
(1004, 2),  
(1005, 5),  
(1006, 6),  
(1008, 49),  
(1009, 6),  
(1010, 2),  
(1011, 3),  
(1012, 1),  
(1013, 1),  
(1014, 2),  
(1015, 1),  
(1016, 1),  
(1017, 1),  
(1018, 32),  
(1019, 6),  
(1020, 10),  
(1021, 4),  
(1022, 1),  
(1023, 1),  
(1024, 2),  
(1025, 8),  
(1026, 3),  
(1028, 1),  
(1029, 2),  
(1030, 13);
```

```

INSERT INTO records (r_id, seller_id, ri_id, ri_num, ri_price, shelf_time, clo
VALUES
(100001, 10001, 1001, 3, 44, '2024-4-10', '2024-4-11', 10002),
(100002, 10001, 1022, 1, 1, '2024-4-1', null, null),
(100003, 10001, 1030, 2, 13, '2024-4-10', '2024-4-11', 10005),
(100004, 10001, 1004, 1, 2, '2024-4-7', '2024-4-8', 10006),
(100005, 10001, 1023, 1, 1, '2024-4-7', '2024-4-8', 10005),
(100006, 10001, 1008, 4, 49, '2024-4-7', '2024-4-8', 10009),
(100007, 10001, 1011, 1, 13, '2024-4-2', '2024-4-8', 10013),
(100008, 10001, 1017, 1, 1, '2024-4-2', '2024-4-8', 10018),
(100009, 10001, 1014, 5, 2, '2024-4-1', '2024-4-8', 10017),
(100010, 10002, 1006, 1, 6, '2024-4-1', '2024-4-8', 10012),
(100011, 10002, 1015, 1, 1, '2024-4-5', '2024-4-8', 10013),
(100012, 10002, 1017, 5, 1, '2024-4-5', '2024-4-8', 10001),
(100013, 10002, 1028, 1, 1, '2024-4-5', '2024-4-8', 10017),
(100014, 10002, 1005, 5, 5, '2024-4-5', '2024-4-8', 10003),
(100015, 10002, 1008, 2, 50, '2024-4-5', '2024-4-8', 10006),
(100016, 10002, 1018, 5, 32, '2024-4-7', '2024-4-8', 10009),
(100017, 10003, 1001, 1, 46, '2024-4-7', '2024-4-8', 10005),
(100018, 10003, 1021, 2, 4, '2024-4-5', '2024-4-8', 10004),
(100019, 10003, 1020, 1, 10, '2024-4-5', '2024-4-8', 10010),
(100020, 10003, 1029, 2, 2, '2024-4-8', '2024-4-10', 10011),
(100021, 10004, 1017, 1, 1, '2024-4-8', '2024-4-10', 10003),
(100022, 10004, 1013, 1, 1, '2024-4-8', '2024-4-11', 10001),
(100023, 10005, 1012, 1, 1, '2024-4-8', null, null),
(100024, 10005, 1015, 3, 1, '2024-4-8', '2024-4-9', 10006),
(100025, 10005, 1016, 1, 1, '2024-4-8', '2024-4-9', 10012),
(100026, 10005, 1019, 1, 6, '2024-4-1', '2024-4-9', 10014),
(100027, 10007, 1023, 4, 1, '2024-4-1', '2024-4-9', 10016),
(100028, 10007, 1024, 1, 2, '2024-4-1', '2024-4-9', 10017),
(100029, 10007, 1025, 3, 8, '2024-4-3', '2024-4-9', 10013),
(100030, 10007, 1009, 1, 6, '2024-4-3', '2024-4-9', 10001),
(100031, 10007, 1026, 1, 3, '2024-4-3', '2024-4-8', 10002),
(100032, 10007, 1028, 2, 1, '2024-4-7', '2024-4-8', 10003),
(100033, 10007, 1030, 1, 13, '2024-4-7', '2024-4-8', 10004),
(100034, 10008, 1022, 1, 1, '2024-4-4', '2024-4-8', 10005),
(100035, 10008, 1029, 1, 2, '2024-4-4', null, null),
(100036, 10009, 1029, 3, 2, '2024-4-4', null, null),
(100037, 10009, 1010, 1, 2, '2024-4-10', '2024-4-11', 10008),
(100038, 10012, 1003, 1, 47, '2024-4-10', '2024-4-11', 10009),
(100039, 10013, 1001, 4, 46, '2024-4-3', '2024-4-5', 10010),

```

```
(100040, 10013, 1002, 1, 1 , '2024-4-3', null, null),
(100041, 10014, 1013, 2, 1 , '2024-4-3', '2024-4-4', 10012),
(100042, 10015, 1016, 1, 1 , '2024-4-3', '2024-4-4', 10013),
(100043, 10016, 1015, 1, 1 , '2024-4-3', '2024-4-5', 10014),
(100044, 10016, 1023, 3, 1 , '2024-4-3', '2024-4-7', 10015),
(100045, 10016, 1017, 1, 1 , '2024-4-1', '2024-4-4', 10018),
(100046, 10017, 1017, 1, 1 , '2024-4-1', null, null),
(100047, 10017, 1018, 1, 32, '2024-4-1', '2024-4-1', 10004),
(100048, 10017, 1015, 5, 1 , '2024-4-1', '2024-4-2', 10006),
(100049, 10017, 1001, 1, 48, '2024-4-1', '2024-4-3', 10007),
(100050, 10017, 1016, 1, 1 , '2024-4-1', '2024-4-3', 10011);
```

3.1 基本查询

查询语句语法：([] 内为可省略内容)

```
SELECT [DISTINCT]
    字段列表
FROM
    表名列表
[
WHERE
    条件列表
GROUP BY
    分组字段
HAVING
    分组之后的条件
ORDER BY
    排序
LIMIT
    分页限定
]
```

简单查询

基础查询

- 查询表中全部数据

```
SELECT * FROM players;
```

- 查询指定列

```
SELECT p_id, p_name FROM players;
```

- 查询时去除重复值

```
SELECT ri_id FROM records;
SELECT DISTINCT ri_id FROM records;
```

可以看到，不去重共有50条数据，去重后只有28条数据

```
SELECT * FROM players;
```

	p_id	p_name	p_gender	registration_time	warehouse_id	p_grade
1	10001	医疗兵_001	女	2022-10-10		10001
2	10002	突击兵_106	男	2023-07-02		10002
3	10003	弗雷德	男	2023-05-02		10003
4	10004	雷诺伊尔	男	2023-03-07		10004
5	10005	多斯	男	2022-12-20		10005


```
SELECT p_id, p_name FROM players;
```

	p_id	p_name
1	10001	医疗兵_001
2	10002	突击兵_106
3	10003	弗雷德
4	10004	雷诺伊尔
5	10005	多斯


```
SELECT ri_id FROM records;
```

	ri_id
42	1010
43	1015
44	1023
45	1017
46	1017
47	1018
48	1015
49	1001
50	1016


```
SELECT DISTINCT ri_id FROM records;
```

	ri_id
20	1010
21	1019
22	1024
23	1025
24	1009
25	1026
26	1010
27	1003
28	1002

限制查询 LIMIT

- **查询前 n 条数据**

```
SELECT * FROM players LIMIT 5;
```

- **查询指定位置开始的前 n 条数据 OFFSET**

```
# 查询玩家表中从第5行起的四个玩家的名字
```

```
SELECT p_id, p_name FROM players LIMIT 4 OFFSET 4;
```

- **实现分页查询, SELECT 字段名 FROM 表名 LIMIT 开始的索引, 每页显示的条数;**

```
# 查询玩家表中从第5条数据开始的, 每页显示三条
```

```
SELECT p_id, p_name, p_grade FROM players LIMIT 5, 3;
```

SELECT * FROM players LIMIT 5;						
	p_id	p_name	p_gender	registration_time	warehouse_id	p_grade
1	10001	医疗兵_001	女	2022-10-10		10001
2	10002	突击兵_106	男	2023-07-02		10002
3	10003	弗雷德	男	2023-05-02		10003
4	10004	雷诺伊尔	男	2023-03-07		10004
5	10005	多斯	男	2022-12-20		10005

查询玩家表中从第5行起的四个玩家的名字
SELECT p_id, p_name FROM players LIMIT 4 OFFSET 4;

SELECT p_id, p_name FROM players LIMIT 4 OFFSET 4;		
	p_id	p_name
1	10005	多斯
2	10006	杰克逊
3	10007	寄术大师
4	10008	无敌CS大王

查询玩家表中从第5条数据开始的，每页显示三条
SELECT p_id, p_name, p_grade FROM players LIMIT 5, 3;

SELECT p_id, p_name, p_grade FROM players LIMIT 5, 3;			
	p_id	p_name	p_grade
1	10006	杰克逊	23
2	10007	寄术大师	30
3	10008	无敌CS大王	27

条件查询

MySQL 默认不区分大小写，要区分的话需要使用 `BINARY` 关键字区分大小写

过滤查询 WHERE

- `WHERE` 字句操作符

- 逻辑比较: `=`、`>`、`<`、`>=`、`<=`、`<>` (`!=`)
- 范围检查: `BETWEEN AND` / `IN`

```
SELECT * FROM items WHERE i_value BETWEEN 3 AND 7;
# 等价于
SELECT * FROM items WHERE i_value IN (3, 4, 5, 6, 7);
```

SELECT * FROM items WHERE i_value BETWEEN 3 AND 7;							
	i_id	i_name	i_weight	i_space	i_value	i_information	i_type
1	1005	金杯	4	2	4 <null>	收藏品	
2	1006	金魔方	4	1	5 <null>	收藏品	
3	1009	金块	4	2	5 <null>	收藏品	
4	1019	201钥匙	1	1	6 <null>	钥匙	
5	1021	储藏室钥匙	1	1	4 <null>	钥匙	
6	1026	车库钥匙	1	1	3 <null>	钥匙	

等价于
SELECT * FROM items WHERE i_value IN (3, 4, 5, 6, 7);

SELECT * FROM items WHERE i_value IN (3, 4, 5, 6, 7);							
	i_id	i_name	i_weight	i_space	i_value	i_information	i_type
1	1005	金杯	4	2	4 <null>	收藏品	
2	1006	金魔方	4	1	5 <null>	收藏品	
3	1009	金块	4	2	5 <null>	收藏品	
4	1019	201钥匙	1	1	6 <null>	钥匙	
5	1021	储藏室钥匙	1	1	4 <null>	钥匙	
6	1026	车库钥匙	1	1	3 <null>	钥匙	

- 空值检查: `IS NULL` / `IS NOT NULL`

不可使用 `=` 和 `!=` 检查

- 使用 AND / OR 连接多个过滤查询

```
SELECT * FROM items WHERE i_value > 20 OR i_value < 2;
```

=

	i_id	i_name	i_weight	i_space	i_value	
1	1001	花瓶		5	4	40 <
2	1002	保温水壶		1	2	1 <
3	1003	茶壶		5	4	40 <
4	1007	机密文件		1	2	600 <
5	1008	金狮雕像		9	6	42 <
6	1012	银徽章		2	1	1 <
7	1013	银片手链		2	1	1 <
8	1015	红色火药		1	2	1 <
9	1016	绿色火药		1	2	1 <
10	1017	蓝色火药		1	2	1 <
11	1018	主客房钥匙		1	1	27 <
12	1022	民宅钥匙		1	4	1 <
13	1023	北村住宅钥匙		1	1	1 <

AND 优先级比 OR 高，优先执行。若需表达复杂逻辑关系，可使用括号，或者使用 IN() 操作符，如：

```
WHERE id IN(3,4) AND grade=9;
```

模糊查询 LIKE

- '%' 代表任意字符出现任意次数

```
# 前缀匹配
SELECT * FROM items WHERE i_name LIKE '金%';
# 后缀匹配
SELECT * FROM items WHERE i_name LIKE '%壶';
# 包含匹配
SELECT * FROM items WHERE i_name LIKE '%火%';
```

```
# 前缀匹配
SELECT * FROM items WHERE i_name LIKE '金%';

```

	i_id	i_name	i_weight	i_space	i_value
1	1004	金豹雕像		4	2
2	1005	金杯		4	2
3	1006	金魔方		4	1
4	1008	金狮雕像		9	6
5	1009	金块		4	2
6	1010	金手镯		3	1
7	1011	金手表		3	1


```
# 后缀匹配
SELECT * FROM items WHERE i_name LIKE '%壺';

```

	i_id	i_name	i_weight	i_space	i_value
1	1002	保温水壺		1	2
2	1003	茶壺		5	40


```
# 包含匹配
SELECT * FROM items WHERE i_name LIKE '%火%';

```

	i_id	i_name	i_weight	i_space	i_value
1	1015	红色火药		1	2
2	1016	绿色火药		1	2
3	1017	蓝色火药		1	2

- _ 代表一个字符

前缀匹配，且两个字符

```
SELECT * FROM items WHERE i_name LIKE '金_';
```

前缀匹配，且三个字符

```
SELECT * FROM items WHERE i_name LIKE '金__';
```

前缀匹配，且两个字符

```
SELECT * FROM items WHERE i_name LIKE '金_';
```

	i_id	i_name	i_weight	i_space	i_value
1	1005	金杯		4	2
2	1009	金块		4	2

前缀匹配，且三个字符

```
SELECT * FROM items WHERE i_name LIKE '金__';
```

	i_id	i_name	i_weight	i_space	i_value
1	1006	金魔方		4	1
2	1010	金手镯		3	1
3	1011	金手表		3	1

- \ 转义字符

| 在 MySQL 中路径用 / 隔开

若要查询含下划线 _ 的，需要在下划线前加上 \，如：

```
# 名称中含有下划线的玩家的 id
SELECT p_id FROM players WHERE p_name LIKE '%\_%';
```

```
# 名称中含有下划线的玩家的 id 与名字
SELECT p_id, p_name FROM players WHERE p_name LIKE '%\_%';
```

	<input type="checkbox"/> p_id ↴	<input type="checkbox"/> p_name ↴
1	10001	医疗兵_001
	10002	突击兵_106

正则查询 REGEXP

- 正则表达式 (regular expression)

- 查询包含 '金' 或 '钥匙' 的物品的 id 和名称

```
SELECT i_id, i_name FROM items WHERE i_name REGEXP '金|钥匙';
```

```
# 查询包含 '金' 或 '钥匙' 的物品的 id
SELECT i_id, i_name FROM items WHERE i_name REGEXP '金|钥匙';
```

	<input type="checkbox"/> i_id ↴	<input type="checkbox"/> i_name ↴
1	1004	金豹雕像
2	1005	金杯
3	1006	金魔方
4	1008	金狮雕像
5	1009	金块
6	1010	金手镯
7	1011	金手表
8	1018	主客房钥匙
9	1019	201钥匙
10	1020	别墅钥匙
11	1021	储藏室钥匙
12	1022	民宅钥匙
13	1023	北村住宅钥匙
14	1024	马厩钥匙
15	1025	海滨别墅钥匙
16	1026	车库钥匙
17	1027	墓地钥匙
18	1028	一楼休息室钥匙
19	1029	武器贮藏室钥匙
20	1030	水闸房钥匙

- 查询姓名中包含英文字母的玩家的 id 和姓名

```
SELECT p_id, p_name FROM players WHERE p_name REGEXP '[A-Za-z]';
```

查询姓名中包含英文字母的玩家的 id
SELECT p_id, p_name FROM players WHERE p_name REGEXP '[A-Za-z]';

	□ p_id ▼	□ p_name ▼
1	10008	无敌CS大王

- 查询姓名中包含3个及以上连续数字的玩家的 id 和姓名

```
SELECT p_id, p_name FROM players WHERE p_name REGEXP '[0-9]{3,}';
```

查询姓名中包含3个以上连续数字的玩家的 id
SELECT p_id, p_name FROM players WHERE p_name REGEXP '[0-9]{3,}';

	□ p_id ▼	□ p_name ▼
1	10001	医疗兵_001
2	10002	突击兵_106

- 查询物品名称以'金'或 '银'开头的物品的 id 和名称

相当于 '^金|^银'

```
SELECT i_id, i_name FROM items WHERE i_name REGEXP '^金|银';
```

```
SELECT i_id, i_name FROM items WHERE i_name REGEXP '^金|^银';
```

```
# 查询物品名称以'金'或'银'开头的物品的 id
# 相当于 '^金|^银'
SELECT i_id, i_name FROM items WHERE i_name REGEXP '^金|银';
```

	i_id	i_name
1	1004	金豹雕像
2	1005	金杯
3	1006	金魔方
4	1008	金狮雕像
5	1009	金块
6	1010	金手镯
7	1011	金手表
8	1012	银徽章
9	1013	银片手链

```
SELECT i_id, i_name FROM items WHERE i_name REGEXP '^金|^银';
```

	i_id	i_name
1	1004	金豹雕像
2	1005	金杯
3	1006	金魔方
4	1008	金狮雕像
5	1009	金块
6	1010	金手镯
7	1011	金手表
8	1012	银徽章
9	1013	银片手链

- 查询物品名称以'壺'结尾的物品的 id 和名称

```
SELECT i_id, i_name FROM items WHERE i_name REGEXP '壺$';
```

```
# 查询物品名称以'壺'结尾的物品的 id
SELECT i_id, i_name FROM items WHERE i_name REGEXP '壺$';
```

	i_id	i_name
1	1002	保温水壺
2	1003	茶壺

• REGEXP 和 LIKE 的区别

如果不使用通配符，`LIKE` 对整个字段进行比对，而 `REGEXP` 只对正则表达式内容进行比对。

```
SELECT * FROM items WHERE i_name LIKE '壺';
SELECT * FROM items WHERE i_name REGEXP '壺';
```

也就是说，第一个查询查找的是'壺'这个字符串，而第二个查找的是包含'壺'的字符串

SELECT * FROM items WHERE i_name LIKE '壺';
□ i_id ▽ □ i_name ▽ □ i_weight ▽ □ i_space ▽ □ i_value ▽ □
SELECT * FROM items WHERE i_name REGEXP '壺';
□ i_id ▽ □ i_name ▽ □ i_weight ▽ □ i_space ▽ □ i_value ▽ □
1 1002 保温水壺 1 2 1
2 1003 茶壺 5 4 40

3.2 查询函数

计算字段

当存储在数据库表中的数据不能直接满足我们的需求时，就需要我们利用已有的字段创造出所需内容，这就是计算字段

• 拼接 CONCAT()

- 字段与字段拼接

```
SELECT CONCAT(p_name, p_id) FROM players;
```

- 字符与字段拼接

```
SELECT CONCAT('昂贵的', i_name) FROM items WHERE i_name REGEXP '金';
```

	□ `CONCAT(p_name, p_id)` ▽
12	德文潘10012
13	兰德尔10013
14	罗尔夫10014
15	科特10017
16	萌新10018

	□ `CONCAT('昂贵的', i_name)` ▽
1	昂贵的金豹雕像
2	昂贵的金杯
3	昂贵的金魔方
4	昂贵的金狮雕像
5	昂贵的金块
6	昂贵的金手镯
7	昂贵的金手表

• 使用字段别名 AS

拼接后字段名消失，MySQL 不方便识别，因此使用字段别名

可以省略 AS，但是为了规范我们不省略 AS

```
SELECT CONCAT(p_id, p_name) AS 玩家姓名id FROM players;
```

```
SELECT CONCAT(p_id, p_name) AS 玩家姓名id FROM players;
```

	□ 玩家姓名id ▾
1	10001医疗兵_001
2	10002突击兵_106
3	10003弗雷德
4	10004雷诺伊尔
5	10005多斯
6	10006杰克逊
7	10007寄术大师
8	10008无敌CS大王
9	10009王八

• 算术运算: + - * /

```
SELECT p_id, p_name, YEAR(NOW()) - YEAR(registration_time) AS 注册时长 FROM pla
```

```
◀ ▶
```

	□ p_id ▾	□ p_name ▾	□ 注册时长 ▾
1		10001 医疗兵_001	2
2		10002 突击兵_106	1
3		10003 弗雷德	1
4		10004 雷诺伊尔	1
5		10005 多斯	2
6		10006 杰克逊	1
7		10007 寄术大师	2
8		10008 无敌CS大王	1
9		10009 王八	2

使用函数

这一部分只做介绍，实际查询可以参考上述**计算字段**的使用，只需要把计算字段换成相应的函数即可

文本处理函数

函数名	函数作用
UPPER(~)	将字符串转换为大写
LOWER(~)	将字符串转换为小写
LENGTH(~)	返回字符串的长度
LEFT(~,n)	返回字符串左边第n个字符
RIGHT(~,n)	返回字符串右边第n个字符
SUBSTRING(~,m,n)	返回字符串从m开始的n个字符
LTRIM(~)	去除字符串左边的空格
RTRIM(~)	去除字符串右边的空格
TRIM(~)	去除字符串两边的空格
REPLACE(~,a,b)	将字符串中的a替换为b
CONCAT(~,~,~,...)	将多个字符串连接起来

数值处理函数

函数名	函数作用
ABS(~)	取绝对值
CEIL(~)	向上取整
FLOOR(~)	向下取整
ROUND(~,n)	四舍五入
SQRT(~)	开平方
POWER(~,n)	幂函数

日期和时间处理函数

函数名	函数作用
CURDATE()	返回当前日期
CURTIME()	返回当前时间
NOW()	返回当前时间
YEAR(~)	返回日期的年
MONTH(~)	返回日期的月
DAY(~)	返回日期的日
DATEDIFF(~,~)	返回两个日期之间的天数

聚集函数

对一组数据(一列数据)进行统计操作，如求和、平均值、最大值、最小值等，返回单个值

常见的聚集函数：

SUM()、AVG() 仅对数值类型计算，MAX()，MIN() 可对任意类型的列

- COUNT() 计数

null 值字段不参与运算，因此选取计算的字段时一般选择非空字段，如主键或主属性(id)

COUNT() 函数还可以使用 DISTINCT 关键字来忽略重复项，统计数目，比如

```
# 统计交易记录中有多少个不同的物品  
SELECT COUNT(DISTINCT ri_id) AS 交易记录物品类别数 FROM records;
```

✓	SELECT COUNT(DISTINCT ri_id) AS 交易记录物品类别数 FROM records;
×	□ 交易记录物品类别数 ▽ 1 28

- SUM() 总计
- AVG() 平均值
- MAX() / MIN() 最大值 / 最小值

WHERE 与聚集函数

在 WHERE 子句中不能使用聚集函数

聚集函数也叫列函数，基于整列数据进行计算，而 WHERE 子句则是对数据进行过滤。在 WHERE 字句执行结束前，聚集函数无法获得列数据（列数据需要等待 WHERE 子句通过行条件进行过滤后才能获得），因此， WHERE 子句中不能使用聚集函数。

3.3 分组与排序

结果分组 GROUP BY

为什么需要分组？

在上一章中我们提到了聚集函数的概念，它能对一组数据(一列数据)进行统计操作，如求和、平均值、最大值、最小值等，并返回单个值(结果)。

然而，在实际需求中，我们常需要将结果分类展示，最经典的：在表中男女各有多少人？我们很轻松的就能想到使用聚合函数计数，如：

```
# 查两次 ?
SELECT COUNT(p_id) AS 男性数目 FROM players WHERE p_gender = '男';
SELECT COUNT(p_id) AS 女性数目 FROM players WHERE p_gender = '女';
# 当然你也可以这样写
SELECT
    SUM(IF(p_gender = '男', 1, 0)) AS 男性数目,
    SUM(IF(p_gender = '女', 1, 0)) AS 女性数目
FROM players;
```

但是这太麻烦了不是吗？这就需要先对数据进行分组，再对分组结果进行聚合 (GROUP BY 子句将结果集划分为值分组，聚合函数为每个分组返回单个值)

```
SELECT p_gender, COUNT(p_gender) AS 人数 FROM players GROUP BY p_gender;
```

	□ p_gender ▼	□ 人数 ▼
1	女	3
2	男	15

同时获取分组结果和汇总结果 WITH ROLLUP

如果既想要分组结果，又想获得汇总结果，可以在 GROUP BY 后面加上 WITH ROLLUP 关键字，对分组后的数据进行汇总计算，在查询得到的结果中添加一行汇总的结果，相当于整合了一个 COUNT(*) FROM 结果集的查询结果，如：

```
SELECT p_gender, COUNT(*) AS 人数 FROM players GROUP BY p_gender WITH ROLLUP;
```

```
SELECT p_gender, COUNT(*) AS 人数 FROM players GROUP BY p_gender WITH ROLLUP;
```

	□ p_gender ↴	□ 人数 ↴
1	女	3
2	男	15
3	<null>	18

需要汇总/分组结果进行筛选 HAVING

在[上一章](#)的最后我们提到， WHERE 语句不能和聚集函数一起使用，因此，当需要汇总/结果分组结果进行筛选时，我们需要使用 HAVING 语句，它的用法与 WHERE 基本相同。

```
SELECT seller_id, COUNT(seller_id) AS 售卖记录大于4 FROM records GROUP BY seller_id HAVING COUNT(seller_id) > 4;
```

	□ seller_id ↴	□ 售卖记录大于4 ↴
1	10001	9
2	10002	7
3	10007	7
4	10017	5

结果排序 ORDER BY

注意： ORDER BY 语句必须出现在整个 SELECT 语句的最后

• 排序方式

排序方式关键字	排序方式
ASC	升序， 默认排序方式，可不写出
DESC	降序，如果使用必须写出

• 示例

```
# 默认
SELECT i_value FROM items ORDER BY i_value;
# 升序
SELECT i_value FROM items ORDER BY i_value ASC;
# 降序
SELECT i_value FROM items ORDER BY i_value DESC;
# 多字段排序
SELECT i_value, i_space FROM items ORDER BY i_value DESC, i_space ASC;
# utf8 中文排序方式
SELECT i_name FROM items ORDER BY CONVERT(i_name USING gbk);
```

默认

```
SELECT i_value FROM items ORDER BY i_value;
```

	i_value
1	1
2	1
3	1
4	1

升序

```
SELECT i_value FROM items ORDER BY i_value ASC;
```

	i_value
1	1
2	1
3	1
4	1

降序

```
SELECT i_value FROM items ORDER BY i_value DESC;
```

	i_value
1	600
2	42
3	40
4	40

多字段排序

```
SELECT i_value, i_space FROM items ORDER BY i_value DESC, i_space ASC;
```

	i_value	i_space
1	600	2
2	42	6
3	40	4
4	40	4

utf8 中文排序方式

```
SELECT i_name FROM items ORDER BY CONVERT(i_name USING gbk);
```

	i_name
1	201钥匙
2	保温水壶
3	北村住宅钥匙
4	别墅钥匙

3.4 复杂查询

子查询，把一个查询嵌套在另一个查询中（单表查询）

将两条相关的语句合并成一个语句，建立父子继承关系，将父亲运算的返回值作为参数传递给儿子的运算或者筛选条件

• 不相关子查询

父查询作为查询条件，把父亲的返回值用于子查询 WHERE 语句中

```
SELECT p_id, p_name FROM players WHERE p_grade = (SELECT MAX(p_grade) FROM pla
```

```
SELECT p_id, p_name FROM players WHERE p_grade = (SELECT MAX(p_grade) FROM players);
```

	□ p_id ▼	□ p_name ▼
1	10005	多斯
2	10007	寄术大师
3	10009	沃伦
4	10010	尤文
5	10011	阿贾克斯
6	10012	德文潘

• 相关子查询

父查询作为计算字段，根据父亲的返回值计算子查询结果

```
SELECT i_id, i_name, (SELECT COUNT(*) FROM records WHERE records.ri_id = item
```

```
SELECT i_id, i_name, (SELECT COUNT(*) FROM records WHERE records.ri_id = items.i_id) 上架记录数 FROM items;
```

	□ i_id ▼	□ i_name ▼	□ 上架记录数 ▼
1	1001	花瓶	4
2	1002	保温水壶	1
3	1003	茶壺	1
4	1004	金豹雕像	1
5	1005	金杯	1
6	1006	金魔方	1
7	1007	机密文件	0
8	1008	金狮雕像	2
9	1009	金块	1
10	1010	金手镯	1
11	1011	金手表	1
12	1012	银徽章	1
13	1013	银片手链	2
14	1014	钻戒	1

多表查询

在本章中不考虑对性能的优化，索引、主键、优化策略会在后续章节中介绍

- 笛卡尔积现象：两张表没有任何条件限制进行连接查询，最终查询结果条数等于两表条数乘积

$A \times B$

- 无匹配条件和筛选条件，最终查询结果条数等于两表条数乘积

```
# 在这里我们可以看到会出现 1500 条查询记录( 30 * 50 )
# 这也叫做隐式交叉连接 (笛卡尔积)

SELECT
    items.i_name, records.ri_price
FROM
    items, records;
/*
# 等价于
SELECT
    items.i_name, records.ri_price
FROM
    items
CROSS JOIN
    records
*/
```

```
# 笛卡尔积，全连接
SELECT
    items.i_name, records.ri_price
FROM
    items, records;
```

	i_name	ri_price
1492	金块	1
1493	金狮雕像	1
1494	机密文件	1
1495	金魔方	1
1496	金杯	1
1497	金豹雕像	1
1498	茶壶	1
1499	保温水壶	1
1500	花瓶	1

|< < 1,001-1,500 > /1,500 >| ↻ ⏴

- 添加匹配条件（只改变显示结果，实际效率未提高，）

```
SELECT
    items.i_name, records.ri_price
FROM
    items, records
WHERE
    items.i_id = records.ri_id;
```

```
# 匹配条件，连接次数不变
SELECT
    items.i_name, records.ri_price
FROM
    items, records
WHERE
    items.i_id = records.ri_id;
```

	□ i_name ▼	□ ri_price ▼
40	保温水壶	1
41	银片手链	1
42	绿色火药	1
43	红色火药	1
44	北村住宅钥匙	1
45	蓝色火药	1
46	蓝色火药	1
47	主客房钥匙	32
48	红色火药	1
49	花瓶	48
50	绿色火药	1

- 在查询时添加寻找限制（提高查询效率）---减少连接次数

```
SELECT
    items.i_name, records.ri_price
FROM
    items
JOIN
    records
ON
    items.i_id < 1010
```

```
SELECT
    items.i_name, records.ri_price
FROM
    items
JOIN
    records
ON
    items.i_id < 1010
```

	i_name	ri_price
446	立屏	1
447	金豹雕像	1
448	茶壶	1
449	保温水壶	1
450	花瓶	1

- 年代分类：

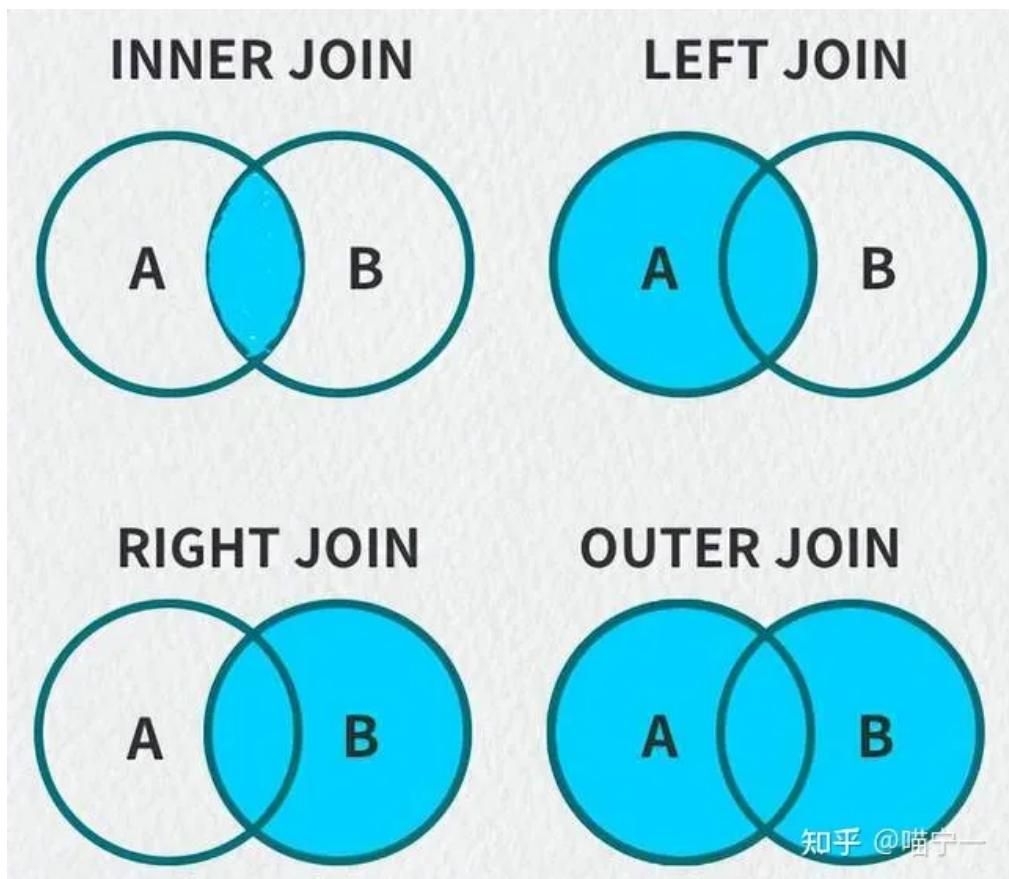
- SQL92：

- 结构不清晰，表连接条件和筛选条件都在 WHERE 后

- SQL99：

- 表连接的条件 ON 和筛选条件 WHERE 分离

- 连接方式分类:



- 内连接 (INNER)

$A \bowtie B$

- 等值连接 --- 条件为等量关系

```
SELECT
    items.i_name, records.ri_price
FROM
    items
INNER JOIN
    records
ON
    items.i_id = records.ri_id;
```

```

SELECT
    items.i_name, records.ri_price
FROM
    items
INNER JOIN
    records
ON
    items.i_id = records.ri_id;

```

	i_name	ri_price
42	冰巴火药	1
43	红色火药	1
44	北村住宅钥匙	1
45	蓝色火药	1
46	蓝色火药	1
47	主客房钥匙	32
48	红色火药	1
49	花瓶	48
50	绿色火药	1

- 非等值连接 ---条件不是一个等量关系

```

SELECT
    i.i_name, r.ri_price
FROM
    items i
INNER JOIN
    records r
ON
    i.i_value < r.ri_price;

```

```

SELECT
    i.i_name, r.ri_price
FROM
    items i
INNER JOIN
    records r
ON
    i.i_value < r.ri_price;

```

	i_name	ri_price
472	木屋小业	48
493	水闸房钥匙	48
494	武器贮藏室钥匙	48
495	一楼休息室钥匙	48
496	墓地钥匙	48
497	车库钥匙	48
498	海滨别墅钥匙	48
499	马厩钥匙	48
500	北村住宅钥匙	48

- 自连接 ---一张表看成两张表，通过别名连接

```

# 查询哪几天既有上架也有售出
SELECT DISTINCT
    a.shelf_time
FROM
    records a
JOIN
    records b
ON
    a.shelf_time=b.closing_time;

```

```

SELECT DISTINCT
    a.shelf_time
FROM
    records a
JOIN
    records b
ON
    a.shelf_time=b.closing_time;

```

	shelf_time
1	2024-04-08
2	2024-04-10
3	2024-04-05
4	2024-04-04
5	2024-04-07
6	2024-04-01
7	2024-04-02
8	2024-04-03

- 外连接 (OUTER) ---两张表连接产生主次关系，保留主表中不符合连接条件的行信息，对应从表属性为 0

- 左连接 ---将 JOIN 左边当作主表，主要是为了将这张表的数据全部查询出来

```

# 可以看到明显的以左表为主表，依照左表的 i_id 进行连接(顺序)
SELECT
*
FROM
    items i
LEFT JOIN
    records r
ON
    i.i_id = r.ri_id;

```

A U A \bowtie B

```

SELECT
  *
FROM
  items i
LEFT JOIN
  records r
ON
  i.i_id = r.ri_id;

```

	value	i_information	i_class	r_id	seller_id	ri_id	ri_num
1	40 <null>	收藏品		100049	10017	1001	
2	40 <null>	收藏品		100039	10013	1001	
3	40 <null>	收藏品		100017	10003	1001	
4	40 <null>	收藏品		100001	10001	1001	
5	1 <null>	生活用品		100040	10013	1002	
6	40 <null>	收藏品		100038	10012	1003	
7	2 <null>	收藏品		100004	10001	1004	
8	4 <null>	收藏品		100014	10002	1005	

- 右连接 --- 将 JOIN 右边当作主表，主要是为了将这张表的数据全部查询出来

```
# 可以看到明显的以右表为主表，依照左表的 ri_id 进行连接(顺序)
```

```

SELECT
  *
FROM
  items i
RIGHT JOIN
  records r
ON
  i.i_id = r.ri_id;

```

A \bowtie B \cup B

```

SELECT
  *
FROM
  items i
RIGHT JOIN
  records r
ON
  i.i_id = r.ri_id;

```

	i_id	i_name	i_weight	i_space	i_value	i_information	
1	1001	花瓶	5	4	40 <null>	收藏	
2	1022	民宅钥匙	1	4	1 <null>	钥匙	
3	1030	水闸房钥匙	1	1	11 <null>	钥匙	
4	1004	金豹雕像	4	2	2 <null>	收藏	
5	1023	北村住宅钥匙	1	1	1 <null>	钥匙	
6	1008	金狮雕像	9	6	42 <null>	收藏	
7	1011	金手表	3	1	2 <null>	收藏	
8	1017	蓝色火药	1	2	1 <null>	易燃	

- 全连接：a 和 b 都是主表

MySQL 目前还不支持全连接 FULL JOIN，不过我们可以通过下一章中的 UNION 关键字来实现全连接

3.5 合并查询

相比较于上一章中提到的复杂查询中的连接操作，使用 UNION 可以减少匹配次数，相当于乘法转化为加法，提高效率

UNION 关键字

UNION 操作符用于合并两个或多个 SELECT 语句的结果集。

A \cup B

在 MySQL 中，使用 UNION 关键字需要遵从以下规则：

1. UNION 必须由两条或两条以上的 SELECT 语句组成
2. UNION 的每个 SELECT 语句必须拥有相同数量的字段，次序可不一致
3. UNION 的每个 SELECT 语句合并的列字段类型必须相同（可隐式转化也行）

例子：

```
# 查询在物品表和交易记录表中价格都高于10的物品id
SELECT ri_id FROM records WHERE ri_price > 10
UNION
SELECT i_value FROM items WHERE i_value > 10;
```

查询在物品表和交易记录表中价格都高于10的物品id

```
SELECT ri_id FROM records WHERE ri_price > 10
UNION
SELECT i_value FROM items WHERE i_value > 10;
```

x	ri_id
1	1011
5	1018
6	1003
7	40
8	600
9	42
10	27
11	16
12	11

UNION ALL

UNION ALL 与 UNION 的区别在于，UNION ALL 将不会对合并结果进行去重，

```
SELECT ri_id FROM records WHERE ri_price > 10
UNION ALL
SELECT i_value FROM items WHERE i_value > 10;
```

✓ SELECT ri_id FROM records WHERE ri_price > 10
UNION ALL
SELECT i_value FROM items WHERE i_value > 10;

×

ri_id	ri_price
11	1010
12	1001
13	40
14	40
15	600
16	42
17	27
18	16
19	11

视图

实际上是对查询的一种抽象实例，个人感觉视图像是 C++ 中的引用，访问视图就是通过其定义的引用，访问对应列。可以把视图当成一个虚拟表使用。

创建视图

语法：

```
CREATE [OR REPLACE] [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
VIEW 视图名称 [(字段列表)]
AS 查询语句 [WITH [CASCADED| LOCAL] CHECK OPTION]
```

- OR REPLACE：如果视图已经存在，则先删除再创建
- ALGORITHM：视图的算法，默认为 UNDEFINED，可选值为 MERGE 和 TEMPTABLE
- WITH [CASCADED| LOCAL] CHECK OPTION：视图的 CHECK OPTION，默认为 CASCADED，可选值为 CASCADED 和 LOCAL

该选项的作用是：通过视图插入、删除或修改元组时检查元组是否满足视图定义中的条件，如果不满足将拒绝执行这些操作。

- 字段列表：视图的字段列表，如果不指定，则默认为查询语句中的字段列表

如果视图定义语句中含有列名表选项，则该选项中的列名个数和次序都要与 SELECT 后的列名表一致。

- 查询语句：视图的查询语句，必须为 SELECT 语句

在视图定义的 SELECT 语句中，不能包含 ORDER BY 关键字

常见示例

创建视图

- 单表视图

```
CREATE VIEW 玩家基本信息 AS
SELECT
    p_id, p_name, p_gender, p_grade
FROM players;
SELECT * FROM 玩家基本信息;
```

<pre>CREATE VIEW 玩家基本信息 AS SELECT p.p_id, p_name, p_gender, p_grade FROM players p;</pre>																																													
<pre>SELECT * FROM 玩家基本信息;</pre>																																													
<table border="1"> <thead> <tr> <th></th> <th>□ p_id ↴</th> <th>□ p_name ↴</th> <th>□ p_gender ↴</th> <th>□ p_grade ↴</th> </tr> </thead> <tbody> <tr><td>1</td><td>10001</td><td>医疗兵_001</td><td>女</td><td>24</td></tr> <tr><td>2</td><td>10002</td><td>突击兵_106</td><td>男</td><td>23</td></tr> <tr><td>3</td><td>10003</td><td>弗雷德</td><td>男</td><td>29</td></tr> <tr><td>4</td><td>10004</td><td>雷诺伊尔</td><td>男</td><td>29</td></tr> <tr><td>5</td><td>10005</td><td>多斯</td><td>男</td><td>30</td></tr> <tr><td>6</td><td>10006</td><td>杰克逊</td><td>男</td><td>23</td></tr> <tr><td>7</td><td>10007</td><td>寄术大师</td><td>男</td><td>30</td></tr> <tr><td>8</td><td>10008</td><td>无敌CS大王</td><td>女</td><td>27</td></tr> </tbody> </table>		□ p_id ↴	□ p_name ↴	□ p_gender ↴	□ p_grade ↴	1	10001	医疗兵_001	女	24	2	10002	突击兵_106	男	23	3	10003	弗雷德	男	29	4	10004	雷诺伊尔	男	29	5	10005	多斯	男	30	6	10006	杰克逊	男	23	7	10007	寄术大师	男	30	8	10008	无敌CS大王	女	27
	□ p_id ↴	□ p_name ↴	□ p_gender ↴	□ p_grade ↴																																									
1	10001	医疗兵_001	女	24																																									
2	10002	突击兵_106	男	23																																									
3	10003	弗雷德	男	29																																									
4	10004	雷诺伊尔	男	29																																									
5	10005	多斯	男	30																																									
6	10006	杰克逊	男	23																																									
7	10007	寄术大师	男	30																																									
8	10008	无敌CS大王	女	27																																									

- 在视图中使用别名

除去在 SELECT 语句中直接使用别名外，还可以使用视图中的别名使用法。

```
# 删去旧视图
DROP VIEW 玩家基本信息;

# 指定视图中别名
CREATE VIEW 玩家基本信息 (id, name, gender, grade) AS
    SELECT
        p.p_id, p_name, p_gender, p_grade
    FROM players p;

SELECT * FROM 玩家基本信息;
```

<pre>DROP VIEW 玩家基本信息;</pre>																																													
<pre># 指定视图中别名 CREATE VIEW 玩家基本信息 (id, name, gender, grade) AS SELECT p.p_id, p_name, p_gender, p_grade FROM players p;</pre>																																													
<pre>SELECT * FROM 玩家基本信息;</pre>																																													
<table border="1"> <thead> <tr> <th></th> <th>□ id ↴</th> <th>□ name ↴</th> <th>□ gender ↴</th> <th>□ grade ↴</th> </tr> </thead> <tbody> <tr><td>1</td><td>10001</td><td>医疗兵_001</td><td>女</td><td>24</td></tr> <tr><td>2</td><td>10002</td><td>突击兵_106</td><td>男</td><td>23</td></tr> <tr><td>3</td><td>10003</td><td>弗雷德</td><td>男</td><td>29</td></tr> <tr><td>4</td><td>10004</td><td>雷诺伊尔</td><td>男</td><td>29</td></tr> <tr><td>5</td><td>10005</td><td>多斯</td><td>男</td><td>30</td></tr> <tr><td>6</td><td>10006</td><td>杰克逊</td><td>男</td><td>23</td></tr> <tr><td>7</td><td>10007</td><td>寄术大师</td><td>男</td><td>30</td></tr> <tr><td>8</td><td>10008</td><td>无敌CS大王</td><td>女</td><td>27</td></tr> </tbody> </table>		□ id ↴	□ name ↴	□ gender ↴	□ grade ↴	1	10001	医疗兵_001	女	24	2	10002	突击兵_106	男	23	3	10003	弗雷德	男	29	4	10004	雷诺伊尔	男	29	5	10005	多斯	男	30	6	10006	杰克逊	男	23	7	10007	寄术大师	男	30	8	10008	无敌CS大王	女	27
	□ id ↴	□ name ↴	□ gender ↴	□ grade ↴																																									
1	10001	医疗兵_001	女	24																																									
2	10002	突击兵_106	男	23																																									
3	10003	弗雷德	男	29																																									
4	10004	雷诺伊尔	男	29																																									
5	10005	多斯	男	30																																									
6	10006	杰克逊	男	23																																									
7	10007	寄术大师	男	30																																									
8	10008	无敌CS大王	女	27																																									

- 多表视图

```

CREATE VIEW 物品价格 ( item_name, value, price ) AS
SELECT
    i.i_name, i.i_value, m.mi_price
FROM
    items i
JOIN
    market m
ON
    i.i_id = m.mi_id;
SELECT * FROM 物品价格;

```

```

CREATE VIEW 物品价格 ( item_name, value, price ) AS
SELECT
    i.i_name, i.i_value, m.mi_price
FROM
    items i
JOIN
    market m
ON
    i.i_id = m.mi_id;
SELECT * FROM 物品价格;

```

	□ item_name ▾	□ value ▾	□ price ▾
1	花瓶	40	44
2	保温水壶	1	1
3	茶壶	40	47
4	金豹雕像	2	2
5	金杯	4	5
6	金魔方	5	6
7	金狮雕像	42	49
8	金块	5	6

- 基于视图的视图

```

CREATE VIEW 贵重物品价格 AS
SELECT
    *
FROM
    物品价格
WHERE
    value > 10;
SELECT * FROM 贵重物品价格;

```

```

CREATE VIEW 贵重物品价格 AS
SELECT
    *
FROM
    物品价格
WHERE
    value > 10;

```

```
SELECT * FROM 贵重物品价格;
```

	□ item_name ▾	□ value ▾	□ price ▾
1	花瓶	40	44
2	茶壶	40	47
3	金狮雕像	42	49
4	主客房钥匙	27	32
5	武器贮藏室钥匙	16	2
6	水闸房钥匙	11	13

查看视图

正如前面所看到的，视图其实可以当成一张虚拟表进行查看，因此查询方式和正常表一样，参见本章前几节。

查看视图的属性信息，会发现除了 `Create_time` 和 `Comment` 外，其他属性都为空，`Comment` 为 `view` 表示该表为视图，是虚表。

```
SHOW TABLE STATUS LIKE '物品价格';
```

```
SHOW TABLE STATUS LIKE '物品价格';
```

createTime ▾	□ Check_time ▾	□ Collation ▾	□ Checksum ▾	□ Create_options ▾	□ Comment ▾
1	<null>	<null>	<null>	<null>	VIEW

修改视图

类似于 C++ 中的引用，对视图的修改其实是对视图基表的修改，或者说，因为视图只是虚表，想要修改视图只能对其基表进行修改，因此对视图的修改会作用到其基表上，当然对视图的修改是有限制的，在一些情况下（要使视图可更新，视图中的行和底层基本表中的行之间必须存在一对一的关系），视图不允许修改。

```
# 插入数据和删除等操作也类似于基本表，不过多展示
```

```
UPDATE 玩家基本信息 SET name = '医疗兵_102' WHERE id = 10001;
```

```
SELECT * FROM 玩家基本信息;
```

<code>UPDATE 玩家基本信息 SET name = '医疗兵_102' WHERE id = 10001;</code>																																													
<code>SELECT * FROM 玩家基本信息;</code>																																													
<table border="1"> <thead> <tr> <th></th><th><input type="checkbox"/> id</th><th><input type="checkbox"/> name</th><th><input type="checkbox"/> gender</th><th><input type="checkbox"/> grade</th></tr> </thead> <tbody> <tr><td>1</td><td>10001</td><td>医疗兵_102</td><td>女</td><td>24</td></tr> <tr><td>2</td><td>10002</td><td>突击兵_106</td><td>男</td><td>23</td></tr> <tr><td>3</td><td>10003</td><td>弗雷德</td><td>男</td><td>29</td></tr> <tr><td>4</td><td>10004</td><td>雷诺伊尔</td><td>男</td><td>29</td></tr> <tr><td>5</td><td>10005</td><td>多斯</td><td>男</td><td>30</td></tr> <tr><td>6</td><td>10006</td><td>杰克逊</td><td>男</td><td>23</td></tr> <tr><td>7</td><td>10007</td><td>寄术大师</td><td>男</td><td>30</td></tr> <tr><td>8</td><td>10008</td><td>无敌CS大王</td><td>女</td><td>27</td></tr> </tbody> </table>		<input type="checkbox"/> id	<input type="checkbox"/> name	<input type="checkbox"/> gender	<input type="checkbox"/> grade	1	10001	医疗兵_102	女	24	2	10002	突击兵_106	男	23	3	10003	弗雷德	男	29	4	10004	雷诺伊尔	男	29	5	10005	多斯	男	30	6	10006	杰克逊	男	23	7	10007	寄术大师	男	30	8	10008	无敌CS大王	女	27
	<input type="checkbox"/> id	<input type="checkbox"/> name	<input type="checkbox"/> gender	<input type="checkbox"/> grade																																									
1	10001	医疗兵_102	女	24																																									
2	10002	突击兵_106	男	23																																									
3	10003	弗雷德	男	29																																									
4	10004	雷诺伊尔	男	29																																									
5	10005	多斯	男	30																																									
6	10006	杰克逊	男	23																																									
7	10007	寄术大师	男	30																																									
8	10008	无敌CS大王	女	27																																									

不支持修改

当视图定义出现如下情况时，视图不支持修改操作：

- 在定义视图的时候指定了 `ALGORITHM = TEMPTABLE`，视图将不支持 `INSERT` 和 `DELETE` 操作；
- 视图中不包含基表中所有被定义为非空又未指定默认值的列，视图将不支持 `INSERT` 操作；
- 在定义视图的 `SELECT` 语句中使用了 `JOIN` 联合查询，视图将不支持 `INSERT` 和 `DELETE` 操作；
- 在定义视图的 `SELECT` 语句后的字段列表中使用了数学表达式或子查询，视图将不支持 `INSERT`，也不支持 `UPDATE` 使用了数学表达式、子查询的字段值；
- 在定义视图的 `SELECT` 语句后的字段列表中使用 `DISTINCT`、聚合函数、`GROUP BY`、`HAVING`、`UNION` 等，视图将不支持 `INSERT`、`UPDATE`、`DELETE`；
- 在定义视图的 `SELECT` 语句中包含了子查询，而子查询中引用了 `FROM` 后面的表，视图将不支持 `INSERT`、`UPDATE`、`DELETE`；
- 视图定义基于一个不可更新视图；
- 常量视图。

变更视图

好吧，其实就是更新视图（替换视图名字对应的查询语句）

- 使用 `CREATE OR REPLACE VIEW` 语句

```
# 使玩家注册时间在视图中可见
CREATE OR REPLACE VIEW 玩家基本信息 AS
SELECT
    p_id, p_name, p_gender, p_grade, registration_time
FROM players;
SELECT * FROM 玩家基本信息;
```

使玩家注册时间在视图中可见

	p_id	p_name	p_gender	p_grade	registration_time
1	10001	医疗兵_102	女	24	2022-10-10
2	10002	突击兵_106	男	23	2023-07-02
3	10003	弗雷德	男	29	2023-05-02
4	10004	雷诺伊尔	男	29	2023-03-07
5	10005	多斯	男	30	2022-12-20
6	10006	杰克逊	男	23	2023-01-07
7	10007	寄术大师	男	30	2022-04-07
8	10008	无敌CS大王	女	27	2023-10-21

- 使用 ALTER VIEW 语句 (两个方法其实一样)

```
ALTER VIEW 玩家基本信息 AS
SELECT
    p_id, p_name, p_gender, p_grade
FROM players;
SELECT * FROM 玩家基本信息;
```

ALTER VIEW 玩家基本信息 AS

	p_id	p_name	p_gender	p_grade
1	10001	医疗兵_102	女	24
2	10002	突击兵_106	男	23
3	10003	弗雷德	男	29
4	10004	雷诺伊尔	男	29
5	10005	多斯	男	30
6	10006	杰克逊	男	23
7	10007	寄术大师	男	30
8	10008	无敌CS大王	女	27

删除视图

只是删除视图的定义，不影响基表

```

DROP VIEW 玩家基本信息;
DROP VIEW IF EXISTS 物品价格;
DROP VIEW 贵重物品价格;

```

左侧树状目录显示了数据库结构，包括 @localhost、架构、game 表、test1、test2、test3 和服务器对象。

	p_id	p_name
2	10002	突击兵_10
3	10003	弗雷德
4	10004	雷诺伊尔
5	10005	多斯
6	10006	杰克逊
7	10007	寄术大师
8	10008	无敌CS大王

```

43
44 ALTER VIEW 玩家基本信息 AS
45     SELECT
46         p_id, p_name, p_gender
47     FROM players;
48     SELECT * FROM 玩家基本信息;

```

	p_id	p_name
0	10008	无敌CS大王
9	10009	沃伦
10	10010	尤文
11	10011	阿贾克斯
12	10012	德文潘
13	10013	兰德尔
14	10014	罗尔夫
15	10017	科特
...	10018	...

```

49
50     DROP VIEW 玩家基本信息;
51     DROP VIEW IF EXISTS 物品价格;
52     DROP VIEW 贵重物品价格;
53

```

服务

左侧展示了事务 (Tx) 列表，右侧展示了操作日志。

```

[2024-05-12 21:56:32] 在 12 ms 内完成
game> SELECT * FROM 玩家基本信息
[2024-05-12 21:56:32] 在 39 ms (execution: 4 ms)
game> DROP VIEW 玩家基本信息
[2024-05-12 21:59:42] 在 8 ms 内完成
game> DROP VIEW IF EXISTS 物品价格
[2024-05-12 21:59:43] 在 5 ms 内完成
game> DROP VIEW 贵重物品价格
[2024-05-12 21:59:43] 在 5 ms 内完成

```

可以看到之前的视图已经消失了

(视图未被删除时)

DatabaseMySQL 版本控制

数据库资源管理器

+ ⚙️ | ⌂ | ⌂ | ⌂ | ⌂ | DDL ⌂ | ⌂ | ⌂

▼ @localhost 4/8

- 架构 4
 - game
 - 表 4
 - items
 - market
 - players
 - records
 - 视图 3
 - 物品价格
 - 玩家基本信息
 - 列 4
 - p_id int
 - p_name varchar(32)
 - p_gender char
 - p_grade int
 - 贵重物品价格
 - test1
 - test2
 - 服务器对象

SQL 进阶技术

本章，我们将进一步探索 MySQL 数据库的进阶编程技术。在前几章中，我们已经学习了 MySQL 的基础知识和常见的编程技巧，现在是时候深入了解一些更高级的概念和技术了。

本章将重点介绍 MySQL 中的一些关键概念和功能，包括索引、主键和外键、触发器以及存储过程的使用。这些技术能让我们能够更好地组织和管理数据库，并提供更高效的数据访问和处理方法。

首先，我们将学习索引的作用和优势。索引是一种数据结构，可以加快数据库的查询速度，并提高数据的检索效率。

接下来，我们将探讨主键和外键的概念。主键是用于唯一标识数据库表中每一行数据的列，而外键用于建立表与表之间的关联。我们将学习如何定义主键和外键，并了解它们在数据库设计和数据完整性方面的重要性。

随后，我们将研究触发器的使用。触发器是一种数据库对象，可以在特定的数据库操作（如插入、更新或删除）发生时自动触发相关的动作或事件。我们将学习如何创建和管理触发器，并了解它们在数据库业务逻辑和数据一致性方面的应用。

最后，我们将介绍存储过程的使用。存储过程是一组预编译的 SQL 语句集合，可以通过简单的调用来执行复杂的数据库操作。我们将学习如何创建和调用存储过程，并探讨它们在提高数据库性能和简化开发过程方面的优势。

索引

MySQL 索引是一种数据结构，用于加快数据库查询的速度和性能。

MySQL 索引类似于书籍的目录，通过存储指向数据行的指针，可以快速定位和访问表中的特定数据。

形象地讲，不使用索引就好像查字典只能一页一页的阅读内容查找数据，而索引就好像目录一样，可以通过偏旁目录来直接定位到相应的页，从而减少查询时间。

索引分单列索引和组合索引：

- 单列索引，即一个索引只包含单个列，一个表可以有多个单列索引。
- 组合索引，即一个索引包含多个列。

索引虽然能够提高查询性能，但也需要注意以下几点：

- 索引需要占用额外的存储空间
 - 空间换时间，相当于书籍里面的目录页
- 对表进行插入、更新和删除操作时，索引需要维护，可能会影响性能
- 过多或不合理的索引可能会导致性能下降，因此需要谨慎选择和规划索引

显示索引信息

```
SHOW INDEX FROM table_name;
```

执行上述命令后，将会显示指定表中所有索引的详细信息，包括索引名称 (Key_name)、索引列 (Column_name)、是否是唯一索引 (Non_unique)、排序方式 (Collation)、索引的基数 (Cardinality) 等。示例将在后续内容中结合展示。

普通索引

普通索引是最常见的索引类型，用于加速对表中数据的查询。

创建索引

语法：

```
CREATE INDEX index_name ON table_name (column1 [ASC|DESC], column2 [ASC|DESC],
```

- 
- CREATE INDEX: 用于创建普通索引的关键字。
 - index_name: 指定要创建的索引的名称。**索引名称在表中必须是唯一的。**

- `table_name`: 指定要在哪个表上创建索引。
- `(column1, column2, ...)`: 指定要索引的表列名，可以指定一个或多个列作为索引的组合。
- `ASC` 和 `DESC` (可选) : 用于指定索引的排序顺序。默认情况下，索引以升序 (ASC) 排序。

示例

```

CREATE INDEX items_name ON items (i_name);
CREATE INDEX items_details ON items (i_weight, i_value DESC, i_space);
# 显示索引信息
SHOW INDEX FROM items;

```

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation
1 items	1	items_name	1	i_name	A
2 items	1	items_details	1	i_weight	A
3 items	1	items_details	2	i_value	D
4 items	1	items_details	3	i_space	A

我们可以看到左侧索引下已经创建了两个索引。

添加索引

```

# 已有表，使用 ALTER TABLE 添加索引
ALTER TABLE items ADD INDEX items_(i_name, i_value);
##### 创建表时指定
CREATE TABLE test (
    id INT,
    name CHAR(16),
    INDEX student_id (id)
);

```

删除索引

语法

```
DROP INDEX index_name ON table_name;  
# 或者使用 ALTER TABLE .. DROP INDEX  
ALTER TABLE table_name DROP INDEX index_name;
```

示例

```
DROP INDEX items_ ON items;  
ALTER TABLE items DROP INDEX items_name, DROP INDEX items_details;  
SHOW INDEX FROM items;
```

DROP INDEX items_ ON items;
ALTER TABLE items DROP INDEX items_name, DROP INDEX items_details;
SHOW INDEX FROM items;

唯一索引

唯一索引 (UNIQUE INDEX) 用于确保表中每行数据的某个字段的值是唯一的，不允许存在重复值。

唯一索引时一种约束，因此它和后续章节中完整性约束中所使用的关键字是一样的

创建唯一索引

语法

```
CREATE UNIQUE INDEX index_name  
ON table_name (column1 [ASC|DESC], column2 [ASC|DESC], ...);
```

示例

```
CREATE UNIQUE INDEX players_id ON players (p_id);  
SHOW INDEX FROM players;
```

CREATE UNIQUE INDEX players_id ON players (p_id);
SHOW INDEX FROM players;

添加索引

注意：使用的是 CONSTRAINT 关键字

```
# 已有表，使用 ALTER TABLE 添加索引
ALTER TABLE items ADD CONSTRAINT items_id UNIQUE (i_id);
SHOW INDEX FROM items;

# 创建表时指定
CREATE TABLE test (
    id INT,
    name CHAR(16),
    CONSTRAINT student_id UNIQUE (id)
);
SHOW INDEX FROM test;
```

```
# 已有表，使用 ALTER TABLE 添加索引
ALTER TABLE items ADD CONSTRAINT items_id UNIQUE (i_id);
SHOW INDEX FROM items;


| Table   | Non_unique | Key_name | Seq_in_index | Column_name | Collation |
|---------|------------|----------|--------------|-------------|-----------|
| 1 items | 0          | items_id | 1            | i_id        | A         |



# 创建表时指定
DROP TABLE test;
CREATE TABLE test (
    id INT,
    name CHAR(16),
    CONSTRAINT student_id UNIQUE (id)
);
SHOW INDEX FROM test;


| Table  | Non_unique | Key_name   | Seq_in_index | Column_name | Collation |
|--------|------------|------------|--------------|-------------|-----------|
| 1 test | 0          | student_id | 1            | id          | A         |


```

删除索引

语法与删除普通索引的语法一样

全文索引

全文索引的弊端个人认为较大，当前的数据量也无法展示全文索引的优点，因此此处放置一篇[参考文章的链接](#)

数据库完整性

数据的正确性(设计要求)和相容性(一致性); 本章的最后会展示使用数据库完整性约束重构的数据库 game, 原始数据库建立参见[第三章](#)

关于键/码的相关术语

候选码、主码简称码

- 超码/超键 (super key): 在关系中某一属性能唯一的标识一个元组(表中的一条记录)的称该属性为超键
- 候选码/候选键 (candidate key): 当超键中不含有多余属性时称为候选键

候选键中的属性称为主属性, 不包含在任何候选键中的属性称为非主属性

- 主码/主键 (primary key): 用户根据需要选定多个候选键中的一个作为主键
- 外码/外键 (foreign key): 如果某属性不是关系 R1 的码, 而在另一个关系 R2 中是主键, 则该属性是关系模式 R1 的外键, 其参照于 R2 的主键

实体完整性

主键约束: PRIMARY KEY

限定字段非空且唯一

添加主键:

语法:

```
CREATE TABLE 表名 (字段名1 字段类型1 PRIMARY KEY, 字段名2 字段类型2, ... 字段名n 字段类型n, PRI
```

示例:

- 在创建表时, 添加主键约束

```

# 针对列的声明
DROP TABLE IF EXISTS test;
CREATE TABLE test (id INT PRIMARY KEY);
DESC test;

# 针对表的声明。创建联合主键: PRIMARY KEY (字段名1, 字段名2);
DROP TABLE test;
CREATE TABLE test (id INT, name VARCHAR(20), registration_data DATE, PRIMARY KEY(id, registration_data));
DESC test;

```

```

# 针对列的声明
DROP TABLE IF EXISTS test;
CREATE TABLE test (id INT PRIMARY KEY);
DESC test;



| Field | Type | Null | Key | Default | Extra |
|-------|------|------|-----|---------|-------|
| id    | int  | NO   | PRI | <null>  |       |



# 针对表的声明。创建联合主键: PRIMARY KEY (字段名1, 字段名2);
DROP TABLE test;
CREATE TABLE test (id INT, name VARCHAR(20), registration_data DATE, PRIMARY KEY(id, registration_data));
DESC test;



| Field             | Type        | Null | Key | Default | Extra |
|-------------------|-------------|------|-----|---------|-------|
| id                | int         | NO   | PRI | <null>  |       |
| name              | varchar(20) | YES  |     | <null>  |       |
| registration_data | date        | NO   | PRI | <null>  |       |


```

- 创建完表后，添加主键

需要注意的是，一个表只能有一个主键，因此只能在没有主键的表中添加主键。

```

# 先删除原有主键
ALTER TABLE test DROP PRIMARY KEY;
DESC test;

# 针对列的声明
ALTER TABLE test MODIFY id INT PRIMARY KEY;
DESC test;

# 先删除原有主键
ALTER TABLE test DROP PRIMARY KEY;
DESC test;

# 针对表的声明
ALTER TABLE test ADD PRIMARY KEY(id, registration_data);

```

```

# 先删除原有主键
ALTER TABLE test DROP PRIMARY KEY;
DESC test;



|   | Field             | Type        | Null | Key | Default |
|---|-------------------|-------------|------|-----|---------|
| 1 | id                | int         | NO   |     | <null>  |
| 2 | name              | varchar(20) | YES  |     | <null>  |
| 3 | registration_data | date        | NO   |     | <null>  |



# 针对列的声明
ALTER TABLE test MODIFY id INT PRIMARY KEY;
DESC test;



|   | Field             | Type        | Null | Key | Default |
|---|-------------------|-------------|------|-----|---------|
| 1 | id                | int         | NO   | PRI | <null>  |
| 2 | name              | varchar(20) | YES  |     | <null>  |
| 3 | registration_data | date        | NO   |     | <null>  |



ALTER TABLE test DROP PRIMARY KEY;
# 针对表的声明
ALTER TABLE test ADD PRIMARY KEY(id, registration_data);
DESC test;



|   | Field             | Type        | Null | Key | Default |
|---|-------------------|-------------|------|-----|---------|
| 1 | id                | int         | NO   | PRI | <null>  |
| 2 | name              | varchar(20) | YES  |     | <null>  |
| 3 | registration_data | date        | NO   | PRI | <null>  |


```

删除主键

```

# 和索引不一样的是不需要知道主键名字
ALTER TABLE test DROP PRIMARY KEY;
DESC test;

```

```

# 和索引不一样的是不需要知道主键名字
ALTER TABLE test DROP PRIMARY KEY;
DESC test;

```

	Field	Type	Null	Key	Default
1	id	int	NO		
2	name	varchar(20)	YES		
3	registration_data	date	NO		

参照完整性

外键约束 CONSTRAINT + FOREIGN KEY + REFERENCES

在创建表时需要注意创建顺序，保证不引用不存在的数据（也就是说对于该属性而言的主表必须先有这个属性的数据，该属性的从表中才能有与之相同的数据）

此外，外键约束中，外键列不能是主键，主表列必须是主键或唯一键值

添加外键

语法：

```
CREATE TABLE IF NOT EXIST 表名(所有字段名声明及主键声明, CONSTRAINT [外键名] FOREIGN KEY (外键列名称) REFERENCES 主表(主键列名称))
```

示例：

- 在创建表时添加外键

```
DROP TABLE IF EXISTS test;
DROP TABLE IF EXISTS record;
CREATE TABLE test(
    id INT,
    name VARCHAR(20),
    registration_date DATE,
    PRIMARY KEY (id)
);
CREATE TABLE record(
    r_id INT,
    r_name VARCHAR(20),
    CONSTRAINT FOREIGN KEY (r_id) REFERENCES test(id)
);
# 系统会自动给外键一个名称
SELECT * FROM information_schema.KEY_COLUMN_USAGE WHERE REFERENCED_TABLE_NAME='test';
```

```
DROP TABLE IF EXISTS test;
DROP TABLE IF EXISTS record;
CREATE TABLE test(
    id INT,
    name VARCHAR(20),
    registration_date DATE,
    PRIMARY KEY (id)
);
CREATE TABLE record(
    r_id INT,
    r_name VARCHAR(20),
    CONSTRAINT FOREIGN KEY (r_id) REFERENCES test(id)
);
# 系统会自动给外键一个名称
SELECT * FROM information_schema.KEY_COLUMN_USAGE WHERE REFERENCED_TABLE_NAME='test';


|   | CONSTRAINT_CATALOG | CONSTRAINT_SCHEMA | CONSTRAINT_NAME | TABLE_CATALOG | TABLE |
|---|--------------------|-------------------|-----------------|---------------|-------|
| 1 | def                | game              | record_ibfk_1   | def           | game  |


```

- 创建表之后添加外键

```
ALTER TABLE test ADD UNIQUE(name);
ALTER TABLE record ADD CONSTRAINT 名字依赖 FOREIGN KEY (r_name) REFERENCES tes
SELECT * FROM information_schema.KEY_COLUMN_USAGE WHERE REFERENCED_TABLE_NAME=
```

CONSTRAINT_CATALOG	CONSTRAINT_SCHEMA	CONSTRAINT_NAME	TABLE_CATALOG	TABLE_SCHEMA
1 f	game	record_ibfk_1	def	game
2 f	game	名字依赖	def	game

删除外键

```
# 也可以通过查看数据库创建语句查看外键名
SHOW CREATE TABLE record;
ALTER TABLE record DROP FOREIGN KEY record_ibfk_1;
ALTER TABLE record DROP FOREIGN KEY 名字依赖;
SELECT * FROM information_schema.KEY_COLUMN_USAGE WHERE REFERENCED_TABLE_NAME=
```

Table	Create Table
	<pre>KEY `名字依赖`(`r_name`), KEY `r_id`(`r_id`), CONSTRAINT `record_ibfk_1` FOREIGN KEY (`r_id`) REFERENCES `test`(`id`), CONSTRAINT `名字依赖` FOREIGN KEY (`r_name`) REFERENCES `test`(`name`)) ENGINE=InnoDB DEFAULT CHARSET=utf8mb3</pre>

```
ALTER TABLE record DROP FOREIGN KEY record_ibfk_1;
ALTER TABLE record DROP FOREIGN KEY 名字依赖;
SELECT * FROM information_schema.KEY_COLUMN_USAGE WHERE REFERENCED_TABLE_NAME='test';
```

但是由于 MySQL 在创建外键后还会自动创建一个索引，因此还需要删除索引

```
ALTER TABLE record DROP KEY r_id;
ALTER TABLE record DROP KEY 名字依赖;
```

外键约束的参照操作

父表中更新、删除数据时外键的应对

语法：

```
ALTER TABLE 表名 ADD CONSTRAINT 外键名 FOREIGN KEY (外键列名称) REFERENCES 主表名
    ON
        UPDATE CASCADE
    ON
        DELETE CASCADE;
```

- `CASCADE` : 从父表删除或更新且自动删除或更新子表中匹配的行。
- `SET NULL` : 从父表删除或更新行，并设置子表中的外键列为 `NULL`。--- 如果使用该选项，必须保证子表列没有指定 `NOT NULL`。
- `RESTRICT` : 拒绝对父表的删除或更新操作，有对应的外键则报错。。---默认
- `NO ACTION` : 标准 SQL 的关键字，在 MySQL 中与 `RESTRICT` 相同。

示例：

```
ALTER TABLE record ADD CONSTRAINT record_id FOREIGN KEY (r_id) REFERENCES test
    ON
        UPDATE CASCADE
    ON
        DELETE RESTRICT;

INSERT INTO test (id, name)
VALUES
    (1, '小白'),
    (2, '小黑'),
    (3, '小红'),
    (4, '小蓝');

INSERT INTO record (r_id, r_name)
VALUES
    (1, '小明'),
    (1, '小李'),
    (2, '小张'),
    (2, '小漆');
SELECT * FROM record;

UPDATE test SET id = 5 WHERE name = '小黑';
SELECT * FROM record;
DELETE FROM test WHERE name = '小白';
SELECT * FROM record;
```

```

37 ALTER TABLE record ADD CONSTRAINT record_id FOREIGN KEY (r_id) REFERENCES test(id)
38   ON UPDATE CASCADE
39   ON DELETE RESTRICT;
40
41
42
43 INSERT INTO test (id, name)
44 VALUES
45   (1, '小白'),
46   (2, '小黑'),
47   (3, '小红'),
48   (4, '小蓝');
49
50 INSERT INTO record (r_id, r_name)
51 VALUES
52   (1, '小明'),
53   (1, '小李'),
54   (2, '小红'),
55   (2, '小蓝');
56 SELECT * FROM record;
57
58 UPDATE test SET id = 5 WHERE name = '小黑';
59 SELECT * FROM record;
60 ① DELETE FROM test WHERE name = '小白';
61
62
63
64

```

[23000][1451] Cannot delete or update a parent row: a foreign key constraint fails ('game'.'record', CONSTRAINT 'record_id' FOREIGN KEY ('r_id') REFERENCES 'test' ('id') ON DELETE RESTRICT ON UPDATE CASCADE)

可以看到 UPDATE 能够正常运行——当主表数据更新，从表也修改对应行；而当对外键所依赖的主表列进行 DELETE 操作时，外键约束会阻止删除操作，因为我们使用了 RESTRICT 关键字来拒绝该操作。

用户自定义完整性

可在创建和修改表时添加

非空约束 NOT NULL

限制字段内容不能为 null 值

MySQL 中支持在含有 null 值的列上使用索引，但是 Oracle 不支持。这就是我们平时所说的如果列上含有 null 那么将会使索引失效。严格来说，这句话对与 MySQL 来说是不准确的。

语法：

```

CREATE TABLE IF NOT EXISTS 表名 (字段名 字段类型 NOT NULL);
ALTER TABLE 表名 MODIFY 字段名 字段类型 NOT NULL;

```

示例：

```
DROP TABLE IF EXISTS record;
DROP TABLE IF EXISTS test;
CREATE TABLE IF NOT EXISTS test (
    f_id INT NOT NULL,
    s_id INT,
    t_id INT
);
ALTER TABLE test MODIFY s_id INT NOT NULL;
DESC test;
```

```
7  DROP TABLE IF EXISTS record;
8  DROP TABLE IF EXISTS test;
9  CREATE TABLE IF NOT EXISTS test (
0    f_id INT NOT NULL,
1    s_id INT,
2    t_id INT
3 );
4 ALTER TABLE test MODIFY s_id INT NOT NULL;|
5 ✓ DESC test;

```

x	<input type="checkbox"/> Field ↴	<input type="checkbox"/> Type ↴	<input type="checkbox"/> Null ↴	<input type="checkbox"/> Key ↴	<input type="checkbox"/> Default ↴
1	f_id	int	NO		<null>
2	s_id	int	NO		<null>
3	t_id	int	YES		<null>

唯一约束 UNIQUE

限定字段不能有重复 (但允许为 `NULL`)

语法：

```
ALTER TABLE 表名 MODIFY 字段名 字段类型 UNIQUE;
ALTER TABLE test ADD CONSTRAINT 约束别名 UNIQUE(字段名);
# 如果没有设置约束别名，直接使用字段名进行删除
ALTER TABLE 表名 DROP INDEX 约束别名/字段名;
```

示例：

```
DROP TABLE IF EXISTS test;
CREATE TABLE IF NOT EXISTS test (
    f_id INT,
    s_id INT,
    t_id INT
);
ALTER TABLE test MODIFY f_id INT UNIQUE;
DESC test;

ALTER TABLE test ADD CONSTRAINT 约束别名 UNIQUE(s_id) ;
DESC test;

ALTER TABLE test DROP INDEX f_id;
DESC test;

ALTER TABLE test DROP INDEX 约束别名;
DESC test;
```

DROP TABLE IF EXISTS test;
CREATE TABLE IF NOT EXISTS test (
 f_id INT,
 s_id INT,
 t_id INT
);
ALTER TABLE test MODIFY f_id INT UNIQUE;
DESC test;

	Field	Type	Null	Key	Default	Extra
1	f_id	int	YES	UNI	<null>	
2	s_id	int	YES		<null>	
3	t_id	int	YES		<null>	

ALTER TABLE test ADD CONSTRAINT 约束别名 UNIQUE(s_id) ;
DESC test;

	Field	Type	Null	Key	Default	Extra
1	f_id	int	YES	UNI	<null>	
2	s_id	int	YES	UNI	<null>	
3	t_id	int	YES		<null>	

ALTER TABLE test DROP INDEX f_id;
DESC test;

	Field	Type	Null	Key	Default	Extra
1	f_id	int	YES		<null>	
2	s_id	int	YES	UNI	<null>	
3	t_id	int	YES		<null>	

✓ ALTER TABLE test DROP INDEX 约束别名;
✓ DESC test;

	Field	Type	Null	Key	Default	Extra
1	f_id	int	YES		<null>	
2	s_id	int	YES		<null>	
3	t_id	int	YES		<null>	

主键要求唯一 `UNIQUE` 和非空 `NOT NULL`，似乎同时具有这两个属性会直接被查询为主键

默认值约束 DEFAULT

给字段设定一个默认值

语法:

```
ALTER TABLE 表名 MODIFY 字段名 字段类型 DEFAULT 值;
```

示例:

```
ALTER TABLE test MODIFY f_id INT DEFAULT 0;  
DESC test;
```

```
ALTER TABLE test MODIFY f_id INT DEFAULT 0;  
DESC test;
```

	Field	Type	Null	Key	Default	Extra
1	f_id	int	YES		0	
2	s_id	int	YES		<null>	
3	t_id	int	YES		<null>	

自动增长 AUTO_INCREMENT

设置自动增长的列必须是 key, 也就是需要是主键或者索引

实现数值的自动增长

语法:

```
ALTER TABLE 表名 MODIFY 字段名 字段类型 AUTO_INCREMENT;
```

示例:

```
ALTER TABLE test ADD INDEX id(s_id);  
ALTER TABLE test MODIFY s_id INT AUTO_INCREMENT;  
DESC test;
```

```
ALTER TABLE test ADD INDEX id(s_id);  
ALTER TABLE test MODIFY s_id INT AUTO_INCREMENT;  
DESC test;
```

	Field	Type	Null	Key	Default	Extra
1	f_id	int	YES		0	
2	s_id	int	NO	MUL	<null>	auto_increment
3	t_id	int	YES		<null>	

检查约束 CHECK

用于验证数据的正确性,

直到 MySQL 8.0.16, MySQL 才真正的支持 `CHECK` 约束。在更早的版本中, 只能通过触发器或者带有 `WITH CHECK OPTION` 的视图来模拟 `CHECK` 约束。使用 `WITH CHECK OPTION` 的教程参见[此处](#)

```
DROP TABLE IF EXISTS test;
# 在行级声明 CHECK 约束
CREATE TABLE test (
    id INT,
    age INT NOT NULL CHECK(age > 0)
);
# 非法插入, 将报错
INSERT INTO test VALUE (1, 0);
```

```
```
36 DROP TABLE IF EXISTS test;
37
38 CREATE TABLE test (
39 id INT,
40 age INT NOT NULL CHECK(age > 0)
41);
42
43 ! INSERT INTO test VALUE (1, 0);
44
45
46
47
48
49
````
```

[HY000][3819] Check constraint 'test_chk_1' is violated.

当然你也可以在表级别声明 `CHECK` 约束, 或者实验 `ALTER TABLE` 语句来添加 `CHECK` 约束。示例如下:

```
CREATE TABLE user (
    id INT,
    age INT NOT NULL,
    CONSTRAINT CHECK(age > 0)
);
ALTER TABLE user ADD CONSTRAINT CHECK(age > 0);
```

重构数据库 game

game 的原始代码可以参见[第三章](#), 现在我们来为它加上完整性约束(为下一节演示触发器, 并未使用侧键的变化应对策略), 来建立一些基本的关系。

下述代码源文件参见本章[源码](#)

重建表:

```

DROP TABLE IF EXISTS items, records, market, players;

CREATE TABLE players
(
    p_id          INT AUTO_INCREMENT,
    p_name        VARCHAR(32) NOT NULL,
    p_gender      CHAR DEFAULT '男',
    registration_time DATE      NOT NULL,
    warehouse_id   INT      NOT NULL UNIQUE,
    p_grade       INT  DEFAULT 1,
    PRIMARY KEY (p_id)
) AUTO_INCREMENT = 10001;

CREATE TABLE items
(
    i_id          INT AUTO_INCREMENT,
    i_name        VARCHAR(16) NOT NULL,
    i_weight      INT      NOT NULL DEFAULT 1,
    i_space       INT      NOT NULL DEFAULT 1,
    i_value       INT      NOT NULL,
    i_information TEXT,
    i_class       VARCHAR(16) NOT NULL,
    PRIMARY KEY (i_id)
) AUTO_INCREMENT = 1001;

CREATE TABLE records
(
    r_id          INT AUTO_INCREMENT,
    seller_id     INT      NOT NULL,
    ri_id         INT      NOT NULL,
    ri_num        TINYINT NOT NULL DEFAULT 1,
    ri_price      INT      NOT NULL DEFAULT 1,
    shelf_time    DATE     NOT NULL,
    closing_time  DATE,
    buyer_id      INT      NULL,
    PRIMARY KEY (r_id),
    CONSTRAINT sellers_id FOREIGN KEY sellers_id (seller_id) REFERENCES player
    CONSTRAINT buyers_id FOREIGN KEY buyers_id (buyer_id) REFERENCES players (
        CONSTRAINT items_id FOREIGN KEY items_id (ri_id) REFERENCES items (i_id) O
        CONSTRAINT CHECK(seller_id != buyer_id)
) AUTO_INCREMENT = 100001;

```

```
CREATE TABLE market
(
    mi_id      INT AUTO_INCREMENT,
    mi_price   INT NOT NULL,
    PRIMARY KEY (mi_id),
    CONSTRAINT market_items_id FOREIGN KEY market_items_id (mi_id) REFERENCES
) AUTO_INCREMENT = 1001;
```

插入数据:

```

INSERT INTO players (p_name, p_gender, registration_time, warehouse_id, p_grade)
VALUES
    ('医疗兵_001', '女', '2022-10-10', 10001, 24),
    ('突击兵_106', '男', '2023-7-2', 10002, 23),
    ('弗雷德', '男', '2023-5-2', 10003, 29),
    ('雷诺伊尔', '男', '2023-3-7', 10004, 29),
    ('多斯', '男', '2022-12-20', 10005, 30),
    ('杰克逊', '男', '2023-1-7', 10006, 23),
    ('寄术大师', '男', '2022-4-7', 10007, 30),
    ('无敌CS大王', '女', '2023-10-21', 10008, 27),
    ('沃伦', '男', '2022-12-10', 10009, 30),
    ('尤文', '男', '2022-1-1', 10010, 30),
    ('阿贾克斯', '男', '2022-1-1', 10011, 30),
    ('德文潘', '男', '2022-1-2', 10012, 30),
    ('兰德尔', '男', '2023-9-24', 10013, 25),
    ('罗尔夫', '男', '2023-11-5', 10014, 21),
    ('卡尔', '男', '2023-7-3', 10015, 25),
    ('思密达', '男', '2023-12-5', 10016, 22),
    ('科特', '男', '2023-11-5', 10017, 23),
    ('萌新', '女', '2024-1-13', 10018, 17);

INSERT INTO items (i_name, i_weight, i_space, i_value, i_class)
VALUES
    ('花瓶', 5, 4, 40, '收藏品'),
    ('保温水壶', 1, 2, 1, '生活用品'),
    ('茶壶', 5, 4, 40, '收藏品'),
    ('金豹雕像', 4, 2, 2, '收藏品'),
    ('金杯', 4, 2, 4, '收藏品'),
    ('金魔方', 4, 1, 5, '收藏品'),
    ('机密文件', 1, 2, 600, '收藏品'),
    ('金狮雕像', 9, 6, 42, '收藏品'),
    ('金块', 4, 2, 5, '收藏品'),
    ('金手镯', 3, 1, 2, '收藏品'),
    ('金手表', 3, 1, 2, '收藏品'),
    ('银徽章', 2, 1, 1, '纪念品'),
    ('银片手链', 2, 1, 1, '收藏品'),
    ('钻戒', 1, 1, 2, '收藏品'),
    ('红色火药', 1, 2, 1, '易燃物'),
    ('绿色火药', 1, 2, 1, '易燃物'),
    ('蓝色火药', 1, 2, 1, '易燃物'),
    ('主客房钥匙', 1, 1, 27, '钥匙'),
    ('201钥匙', 1, 1, 6, '钥匙'),

```

```
('别墅钥匙', 1, 1, 8, '钥匙'),  
('储藏室钥匙', 1, 1, 4, '钥匙'),  
('民宅钥匙', 1, 4, 1, '钥匙'),  
('北村住宅钥匙', 1, 1, 1, '钥匙'),  
('马厩钥匙', 1, 1, 2, '钥匙'),  
('海滨别墅钥匙', 1, 1, 8, '钥匙'),  
('车库钥匙', 1, 1, 3, '钥匙'),  
('墓地钥匙', 1, 1, 8, '钥匙'),  
('一楼休息室钥匙', 1, 1, 10, '钥匙'),  
('武器贮藏室钥匙', 1, 1, 16, '钥匙'),  
('水闸房钥匙', 1, 1, 11, '钥匙');
```

```
INSERT INTO market (mi_price)
```

```
VALUES
```

```
(44),  
(1),  
(47),  
(2),  
(5),  
(6),  
(49),  
(6),  
(2),  
(3),  
(1),  
(1),  
(2),  
(1),  
(1),  
(1),  
(32),  
(6),  
(10),  
(4),  
(1),  
(1),  
(2),  
(8),  
(3),  
(1),  
(2),  
(13);
```

```

INSERT INTO records (seller_id, ri_id, ri_num, ri_price, shelf_time, closing_t
VALUES
(10001, 1001, 3, 44, '2024-4-10', '2024-4-11', 10002),
(10001, 1022, 1, 1, '2024-4-1', null, null),
(10001, 1030, 2, 13, '2024-4-10', '2024-4-11', 10005),
(10001, 1004, 1, 2, '2024-4-7', '2024-4-8', 10006),
(10001, 1023, 1, 1, '2024-4-7', '2024-4-8', 10005),
(10001, 1008, 4, 49, '2024-4-7', '2024-4-8', 10009),
(10001, 1011, 1, 13, '2024-4-2', '2024-4-8', 10013),
(10001, 1017, 1, 1, '2024-4-2', '2024-4-8', 10018),
(10001, 1014, 5, 2, '2024-4-1', '2024-4-8', 10017),
(10002, 1006, 1, 6, '2024-4-1', '2024-4-8', 10012),
(10002, 1015, 1, 1, '2024-4-5', '2024-4-8', 10013),
(10002, 1017, 5, 1, '2024-4-5', '2024-4-8', 10001),
(10002, 1028, 1, 1, '2024-4-5', '2024-4-8', 10017),
(10002, 1005, 5, 5, '2024-4-5', '2024-4-8', 10003),
(10002, 1008, 2, 50, '2024-4-5', '2024-4-8', 10006),
(10002, 1018, 5, 32, '2024-4-7', '2024-4-8', 10009),
(10003, 1001, 1, 46, '2024-4-7', '2024-4-8', 10005),
(10003, 1021, 2, 4, '2024-4-5', '2024-4-8', 10004),
(10003, 1020, 1, 10, '2024-4-5', '2024-4-8', 10010),
(10003, 1029, 2, 2, '2024-4-8', '2024-4-10', 10011),
(10004, 1017, 1, 1, '2024-4-8', '2024-4-10', 10003),
(10004, 1013, 1, 1, '2024-4-8', '2024-4-11', 10001),
(10005, 1012, 1, 1, '2024-4-8', null, null),
(10005, 1015, 3, 1, '2024-4-8', '2024-4-9', 10006),
(10005, 1016, 1, 1, '2024-4-8', '2024-4-9', 10012),
(10005, 1019, 1, 6, '2024-4-1', '2024-4-9', 10014),
(10007, 1023, 4, 1, '2024-4-1', '2024-4-9', 10016),
(10007, 1024, 1, 2, '2024-4-1', '2024-4-9', 10017),
(10007, 1025, 3, 8, '2024-4-3', '2024-4-9', 10013),
(10007, 1009, 1, 6, '2024-4-3', '2024-4-9', 10001),
(10007, 1026, 1, 3, '2024-4-3', '2024-4-8', 10002),
(10007, 1028, 2, 1, '2024-4-7', '2024-4-8', 10003),
(10007, 1030, 1, 13, '2024-4-7', '2024-4-8', 10004),
(10008, 1022, 1, 1, '2024-4-4', '2024-4-8', 10005),
(10008, 1029, 1, 2, '2024-4-4', null, null),
(10009, 1029, 3, 2, '2024-4-4', null, null),
(10009, 1010, 1, 2, '2024-4-10', '2024-4-11', 10008),
(10012, 1003, 1, 47, '2024-4-10', '2024-4-11', 10009),
(10013, 1001, 4, 46, '2024-4-3', '2024-4-5', 10010),

```

```
(10013, 1002, 1, 1 , '2024-4-3', null, null),
(10014, 1013, 2, 1 , '2024-4-3', '2024-4-4', 10012),
(10015, 1016, 1, 1 , '2024-4-3', '2024-4-4', 10013),
(10016, 1015, 1, 1 , '2024-4-3', '2024-4-5', 10014),
(10016, 1023, 3, 1 , '2024-4-3', '2024-4-7', 10015),
(10016, 1017, 1, 1 , '2024-4-1', '2024-4-4', 10018),
(10017, 1017, 1, 1 , '2024-4-1', null, null),
(10017, 1018, 1, 32, '2024-4-1', '2024-4-1', 10004),
(10017, 1015, 5, 1 , '2024-4-1', '2024-4-2', 10006),
(10017, 1001, 1, 48, '2024-4-1', '2024-4-3', 10007),
(10017, 1016, 1, 1 , '2024-4-1', '2024-4-3', 10011);
```

触发器

在上一章中我们介绍了实体完整性，参照完整性以及用户自定义完整性的相关知识，这三者一般称为数据库的完整性约束机制（CONSTRAINT），是数据库实现业务规则和数据完整性的一种方法。但是完整性约束机制只能在用户违反约束机制后做出简单的响应，如果要对用户操作进行更复杂的响应，就需要用到触发器。

触发器是用编程的方法，灵活地实现业务规则并保障数据完整性。触发器是一种特殊的存储过程，存储过程需要人为手动调用，而触发器则不需要，它可以在执行某项数据操作后自动触发。

百度百科：触发器与存储过程的唯一区别是触发器不能执行 `EXECUTE` 语句调用，而是在用户执行 `Transact-SQL` 语句时自动触发执行

触发器的类型

按照触发时间和业务类型，MySQL 支持三种类型的触发器：

- BEFORE 触发器：在执行实际的数据操作之前触发，例如在插入、更新或删除数据之前执行某些操作，例如数据验证、数据转换等。
- AFTER 触发器：在执行实际的数据操作之后触发，例如在插入、更新或删除数据之后执行某些操作，例如生成审计日志、更新相关数据等。
- INSTEAD OF 触发器：在执行实际的数据操作之前触发，并可以在触发器中替代原始的数据操作，例如在插入、更新或删除数据之前执行自定义的数据操作，而不是执行原始的数据操作。

按照操作的类型，MySQL 支持 `INSERT`、`UPDATE` 和 `DELETE` 三种触发器。

创建触发器

基于一个表创建，对多个表进行操作。

语法：

```
CREATE TRIGGER trigger_name
[ BEFORE / AFTER ] [ INSERT / UPDATE / DELETE ] ON table_name
FOR EACH ROW
trigger_body
```

其中，

- `trigger_name`: 触发器的名称，用户自定义的唯一名称。

- BEFORE / AFTER: 触发器的触发时机，可以是 BEFORE (在事件之前触发) 或 AFTER (在事件之后触发)。
- [INSERT / UPDATE / DELETE]: 触发器所监听的事件，可以是 INSERT (插入操作)、UPDATE (更新操作) 或 DELETE (删除操作)。
- table_name: 触发器所关联的数据库表名。
- FOR EACH ROW: 表示触发器的作用范围是每一行，即每当表中的一行数据发生指定的事件时，触发器都会被触发。
- trigger_body: 触发器的执行体，包含一系列的 SQL 语句，用于定义触发器在触发时所执行的操作。

示例：

在先前提到的 game 数据库中，records 表还可以添加 status 列，用来记录每一条记录的状态，默认为 0，表示还没有成交的状态。我们可以用 INSERT 触发器和 UPDATE 触发器来维护 status 列的值()

需要注意的是，同表更新不能使用 UPDATE 和 INSERT 来进行触发操作，避免递归触发

需要注意的是，在 AFTER 触发器中，NEW 的赋值已经结束了，只能读取内容，因此只能在 BEFORE 触发器中修改 NEW 的值。

```
# 当然直接将 status 列的值默认设置为 0 是一个更好的选择
DROP TABLE records;
# CREATE TABLE records(..);( 省略，参见上一章节末尾的[代码](./code/game.sql))
ALTER TABLE records ADD status TINYINT(1);
CREATE TRIGGER records_after_insert
BEFORE INSERT ON records
FOR EACH ROW
BEGIN
    SET NEW.status = FALSE;
END;
# INSERT...( 省略，参见上一章节末尾的[代码](./code/game.sql))
SELECT * FROM records;
```

```

# 使用 DELETE 会导致计数器不重置, 使用 TRUNCATE 会导致计数器重置到 1, 改变自定义自增初值, 因此此处重建表
DROP TABLE records;
> CREATE TABLE records...;
ALTER TABLE records ADD status TINYINT(1);
CREATE TRIGGER records_after_insert
BEFORE INSERT ON records
FOR EACH ROW
BEGIN
    SET NEW.status = FALSE;
END;
> INSERT INTO records ...;
SELECT * FROM records;

```

| ri_num | ri_price | shelf_time | closing_time | buyer_id | status |
|--------|----------|---------------|--------------|----------|--------|
| 1 | 3 | 44 2024-04-10 | 2024-04-11 | 10002 | 0 |
| 2 | 1 | 1 2024-04-01 | <null> | <null> | 0 |
| 3 | 2 | 13 2024-04-10 | 2024-04-11 | 10005 | 0 |
| 4 | 1 | 2 2024-04-07 | 2024-04-08 | 10006 | 0 |
| 5 | 1 | 1 2024-04-07 | 2024-04-08 | 10005 | 0 |
| 6 | 4 | 49 2024-04-07 | 2024-04-08 | 10009 | 0 |
| 7 | 1 | 13 2024-04-02 | 2024-04-08 | 10013 | 0 |
| 8 | 1 | 1 2024-04-02 | 2024-04-08 | 10010 | 0 |

可以看到，在插入数据时，触发器自动将 status 列的值设置为 0。当然，此处并不算是业务逻辑，而是为了演示触发器的使用，业务所面向的插入应该是没有 buyer_id 和 time 等列值的，而是通过触发器或者业务代码来补充这些列的值。

下面展示UPDATE 触发器：

```

# 更新数据, 这里我们默认更新的是 buyer_id, 也就是默认此处业务时交易完成
CREATE TRIGGER records_before_update
    BEFORE UPDATE ON records
    FOR EACH ROW
    BEGIN
        SET
            NEW.status = TRUE,
            NEW.closing_time = CURRENT_DATE;
    END;

UPDATE records SET buyer_id = 10003 WHERE r_id = 100002;
SELECT * FROM records;

```

```

CREATE TRIGGER records_before_update
    BEFORE UPDATE ON records
    FOR EACH ROW
    BEGIN
        SET
            NEW.status = TRUE,
            NEW.closing_time = CURRENT_DATE;
    END;

UPDATE records SET buyer_id = 10003 WHERE r_id = 100002;
SELECT * FROM records;

```

| ri_num | ri_price | shelf_time | closing_time | buyer_id | status |
|--------|----------|---------------|--------------|----------|--------|
| 1 | 3 | 44 2024-04-10 | 2024-04-11 | 10002 | 0 |
| 2 | 1 | 1 2024-04-01 | 2024-05-14 | 10003 | 1 |
| 3 | 2 | 13 2024-04-10 | 2024-04-11 | 10005 | 0 |
| 4 | 1 | 2 2024-04-07 | 2024-04-08 | 10006 | 0 |
| 5 | 1 | 1 2024-04-07 | 2024-04-08 | 10005 | 0 |
| 6 | 4 | 49 2024-04-07 | 2024-04-08 | 10009 | 0 |
| 7 | 1 | 13 2024-04-02 | 2024-04-08 | 10013 | 0 |
| 8 | 1 | 1 2024-04-02 | 2024-04-08 | 10010 | 0 |

可以看到，在更新数据后，触发器自动将 status 列的值设置为 1，同时将 closing_time 列的值设置为当前日期。

上面展示的两个触发器都是对自身表进行操作，下面将在 players 表中定义 DELETE 触发器来演示对 records 表的操作：

```
# 由于表中存在外键关系，所以需要先删除外键关系，再进行操作
ALTER TABLE records DROP FOREIGN KEY buyers_id;
ALTER TABLE records DROP FOREIGN KEY sellers_id;
ALTER TABLE records DROP KEY buyers_id;
ALTER TABLE records DROP KEY sellers_id;
CREATE TRIGGER players_before_delete
    BEFORE DELETE ON players
    FOR EACH ROW
    BEGIN
        DELETE FROM records WHERE seller_id = OLD.p_id || buyer_id = OLD.p_id;
    END;
DELETE FROM players WHERE p_id = 10001;
SELECT * FROM records;
```

已经更正为 player_before_delete

```
ALTER TABLE records DROP FOREIGN KEY buyers_id;
ALTER TABLE records DROP FOREIGN KEY sellers_id;
ALTER TABLE records DROP KEY buyers_id;
ALTER TABLE records DROP KEY sellers_id;
CREATE TRIGGER records_before_delete
    BEFORE DELETE ON players
    FOR EACH ROW
    BEGIN
        DELETE FROM records WHERE seller_id = OLD.p_id || buyer_id = OLD.p_id;
    END;
DELETE FROM players WHERE p_id = 10001;
SELECT * FROM records;
```

| r_id | seller_id | ri_id | ri_num | ri_price | shelf_time | clo |
|------|-----------|-------|--------|----------|---------------|--------|
| 1 | 100010 | 10002 | 1006 | 1 | 6 2024-04-01 | 2024-0 |
| 2 | 100011 | 10002 | 1015 | 1 | 1 2024-04-05 | 2024-0 |
| 3 | 100013 | 10002 | 1028 | 1 | 1 2024-04-05 | 2024-0 |
| 4 | 100014 | 10002 | 1005 | 5 | 5 2024-04-05 | 2024-0 |
| 5 | 100015 | 10002 | 1008 | 2 | 50 2024-04-05 | 2024-0 |
| 6 | 100016 | 10002 | 1018 | 5 | 32 2024-04-07 | 2024-0 |
| 7 | 100017 | 10003 | 1001 | 1 | 46 2024-04-07 | 2024-0 |
| 8 | 100018 | 10002 | 1021 | 2 | 4 2024-04-05 | 2024-0 |

可以看到，删除了 id 为 10001 的玩家后，records 表中的数据也相应被删除了。

查看触发器

语法

```
SHOW TRIGGERS [LIKE 'pattern' | WHERE expr];
```

- `LIKE 'pattern'` 是可选的，表示根据模式匹配来筛选要查看的触发器。
 - `pattern` 是一个字符串模式，可以包含通配符 % 表示任意字符序列，可以帮助用户根据触发器名称进行模糊匹配。
- `WHERE expr` 也是可选的，表示根据表达式来筛选要查看的触发器。
 - `expr` 是一个布尔表达式，用于筛选符合条件的触发器。

示例

在上面的示例中，我们创建了三个触发器，分别是

`records_after_insert`、`records_before_update` 和 `players_before_delete`。我们可以使用 `SHOW TRIGGERS` 命令来查看这些触发器的定义：

```
SHOW TRIGGERS;
# 使用 LIKE 模糊查询
SHOW TRIGGERS LIKE 'records%';
```

| Trigger | Event | Table | Statement | Timing | Created | sql_mode | Definer | character |
|----------------------|--------|---------|-----------|-------------------|----------------------|----------------------|----------------|-----------|
| players_before_de... | DELETE | players | BEGIN | DEL... BEFORE | 2024-05-18 16:46:... | ONLY_FULL_GROUP_B... | root@localhost | utf8mb4 |
| records_after_ins... | INSERT | records | BEGIN | SET NEW... BEFORE | 2024-05-14 20:04:... | ONLY_FULL_GROUP_B... | root@localhost | utf8mb4 |
| records_before_up... | UPDATE | records | BEGIN | SET... BEFORE | 2024-05-14 20:04:... | ONLY_FULL_GROUP_B... | root@localhost | utf8mb4 |

| Trigger | Event | Table | Statement | Timing | Created | sql_mode | Definer | character |
|----------------------|--------|---------|-----------|-------------------|----------------------|----------------------|----------------|-----------|
| records_after_ins... | INSERT | records | BEGIN | SET NEW... BEFORE | 2024-05-14 20:04:... | ONLY_FULL_GROUP_B... | root@localhost | utf8mb4 |
| records_before_up... | UPDATE | records | BEGIN | SET... BEFORE | 2024-05-14 20:04:... | ONLY_FULL_GROUP_B... | root@localhost | utf8mb4 |

删除触发器

语法

```
DROP TRIGGER [IF EXISTS] [schema_name.]trigger_name;
```

其中：

- `[IF EXISTS]` 可选，表示在删除触发器时是否检查触发器是否存在，如果存在则删除，如果不存在则忽略。
- `schema_name` 是可选的，表示触发器所属的数据库模式（Schema）名称，如果省略则表示当前数据库模式。
- `trigger_name` 是触发器的名称，表示要删除的触发器的名称。

示例

删除之前创建的触发器

```
DROP TRIGGER IF EXISTS records_after_insert;
DROP TRIGGER IF EXISTS records_before_update;
DROP TRIGGER IF EXISTS players_before_delete;
SHOW TRIGGERS;
```

A screenshot of the MySQL Workbench interface. The main window shows the following SQL code:

```
DROP TRIGGER IF EXISTS records_after_insert;
DROP TRIGGER IF EXISTS records_before_update;
DROP TRIGGER IF EXISTS players_before_delete;
SHOW TRIGGERS;
```

Below the code, there is a search bar with the following dropdown menu options: Trigger, Event, Table, Statement, Timing, Created, sql_mode, Definer, character_set, and collation.

存储过程

MySQL 5.0 版本开始支持存储过程。

存储过程（Stored Procedure）是一种在数据库中存储复杂程序，以便外部程序调用的一种数据库对象。

存储过程思想上很简单，就是数据库 SQL 语言层面的代码封装与重用，类似于 C/C++ 的函数调用。

存储过程的优点：

- 通过把处理封装在容易使用的单元中，简化复杂的操作；
- 简化对变动的管理。如果表名、列名或业务逻辑有变化，只需要更改存储过程的代码，使用它的人员不会改自己的代码；
- 通常存储过程有助于提高应用程序的性能。当创建的存储过程被编译之后，就存储在数据库中。但是，MySQL 实现的存储过程略有不同。MySQL 存储过程按需编译。在编译存储过程之后，MySQL 将其放入缓存中。MySQL 为每个连接维护自己的存储过程高速缓存。如果应用程序在单个连接中多次使用存储过程，则使用编译版本，否则存储过程的工作方式类似于查询；
- 存储过程有助于减少应用程序和数据库服务器之间的流量，因为应用程序不必发送多个冗长的 SQL 语句，而只用发送存储过程的名称和参数；
- 存储过程对任何应用程序都是可重用的和透明的。存储过程将数据库接口暴露给所有应用程序，以便开发人员不必开发存储过程中已支持的功能；
- 存储过程是安全的。数据库管理员可以向访问数据库中存储过程的应用程序授予适当的权限，而不向基础数据库表提供任何权限。

存储过程的缺点

- 如果使用大量存储过程，那么使用这些存储过程的每个连接的内存使用量将会大大增加。此外，如果您在存储过程中过度使用大量逻辑操作，则 CPU 使用率也会增加，因为 MySQL 数据库最初的设计侧重于高效的查询，不利于逻辑运算；
- 存储过程的构造使得开发具有复杂业务逻辑的存储过程变得更加困难；
- 很难调试存储过程。只有少数数据库管理系统允许您调试存储过程。不幸的是，MySQL 不提供调试存储过程的功能；
- 开发和维护存储过程不容易。开发和维护存储过程通常需要一个不是所有应用程序开发人员拥有的专业技能。这可能会导致应用程序开发和维护阶段的问题

存储过程的使用流程

创建存储过程 —— 调用存储过程 —— 显示流程结果

语法

```
# 创建存储过程
CREATE
[DEFINER = 'user_name'@'host_name']
[SQL SECURITY { DEFINER | INVOKER }]
PROCEDURE 存储过程名( 参数类型 变量名 变量类型, ... )
BEGIN
    [DECLARE 变量名 类型 [DEFAULT 值];]
    存储过程的语句块;
END;
# 调用存储过程, @result 为存储过程返回值也就是 OUT 参数返回的值
CALL 存储过程名(参数列表, @result);
# 显示流程结果
SELECT @result;
```

DEFINER = 用于指定存储过程定义者， 默认为当前用户。

SQL SECURITY { DEFINER | INVOKER }: 选择调用时的使用权限， DEFINER 表示使用存储过程定义者的权限， INVOKER 表示使用存储过程调用者的权限。

存储过程中的参数分别有 in, out, inout 三种类型:

- in 代表输入参数（默认情况下为 in 参数）， 表示该参数的值必须由调用程序指定。
- out 代表输出参数， 表示该参数的值经存储过程计算后， 将 out 参数的计算结果返回给调用程序。
- inout 代表即时输入参数， 又是输出参数， 表示该参数的值即可有调用程序制定， 又可以将 inout 参数的计算结果返回给调用程序。

DECLARE 中用来声明变量， 变量默认赋值使用的 DEFAULT， 语句块中改变变量值， 使用 SET 变量 = 值；

示例

```
CREATE PROCEDURE try_select(IN t1 INT,OUT t2 INT)
BEGIN
    DECLARE i INT DEFAULT 1;
    SET t2 = 0;
    WHILE i <= t1
        DO
            SET t2 = t2 + i;
            SET i = i + 1;
        END WHILE;
    END;
# 调用存储过程 try_select(IN t1 INT,OUT t2 INT);
CALL try_select(5 ,@result);
SELECT @result;

CREATE PROCEDURE try_select(IN t1 INT,OUT t2 INT)
BEGIN
    DECLARE i INT DEFAULT 1;
    SET t2 = 0;
    WHILE i<=t1
        DO
            SET t2=t2+i;
            SET i=i+1;
        END WHILE;
    END;
# 调用存储过程 try_select(IN t1 INT,OUT t2 INT);
CALL try_select( t1 5 , t2 @result);
SELECT @result;
```

| | | |
|---|----------------------------------|----|
| | <input type="checkbox"/> @result | ▼ |
| 1 | | 15 |

存储过程的管理

- 查看存储过程

```

SHOW PROCEDURE STATUS;
# 查看特定数据库的存储过程
SHOW PROCEDURE STATUS WHERE DB = 'game';
# 查看特定名存储过程
SHOW PROCEDURE STATUS WHERE name LIKE 'ps%';

```

| SHOW PROCEDURE STATUS; | | | | | | | |
|------------------------|------------------------------------|-----------|---------------------|---------------------|---------------------|---------------------|--|
| Db | Name | Type | Definer | Modified | Created | Cr | |
| sys | ps_setup_delegate_consumer | PROCEDURE | mysql.sys@localhost | 2020-07-24 11:02:38 | 2020-07-24 11:02:38 | 2020-07-24 11:02:38 | |
| sys | ps_setup_disable_instrument | PROCEDURE | mysql.sys@localhost | 2023-09-24 11:02:38 | 2023-09-24 11:02:38 | 2023-09-24 11:02:38 | |
| sys | ps_setup_disable_thread | PROCEDURE | mysql.sys@localhost | 2023-09-24 11:02:38 | 2023-09-24 11:02:38 | 2023-09-24 11:02:38 | |
| sys | ps_setup_enable_background_threads | PROCEDURE | mysql.sys@localhost | 2023-09-24 11:02:38 | 2023-09-24 11:02:38 | 2023-09-24 11:02:38 | |
| sys | ps_setup_enable_consumer | PROCEDURE | mysql.sys@localhost | 2023-09-24 11:02:38 | 2023-09-24 11:02:38 | 2023-09-24 11:02:38 | |
| sys | ps_setup_enable_instrument | PROCEDURE | mysql.sys@localhost | 2023-09-24 11:02:38 | 2023-09-24 11:02:38 | 2023-09-24 11:02:38 | |
| sys | ps_setup_enable_thread | PROCEDURE | mysql.sys@localhost | 2023-09-24 11:02:38 | 2023-09-24 11:02:38 | 2023-09-24 11:02:38 | |
| sys | ps_setup_reload_saved | PROCEDURE | mysql.sys@localhost | 2023-09-24 11:02:38 | 2023-09-24 11:02:38 | 2023-09-24 11:02:38 | |

| # 查看特定数据库的存储过程 | | | | | | | |
|--|------------|-----------|----------------|---------------------|---------------------|---------|--|
| SHOW PROCEDURE STATUS WHERE DB = 'game'; | | | | | | | |
| Db | Name | Type | Definer | Modified | Created | Cr | |
| game | try_select | PROCEDURE | root@localhost | 2024-05-19 00:52:28 | 2024-05-19 00:52:28 | DEFINER | |

| # 查看特定名存储过程 | | | | | | | |
|--|-------------------------------------|-----------|---------------------|---------------------|---------------------|---------------------|--|
| SHOW PROCEDURE STATUS WHERE name LIKE 'ps%'; | | | | | | | |
| Db | Name | Type | Definer | Modified | Created | Cr | |
| 1 sys | ps_setup_disable_background_threads | PROCEDURE | mysql.sys@localhost | 2023-09-24 11:02:38 | 2023-09-24 11:02:38 | 2023-09-24 11:02:38 | |
| 2 sys | ps_setup_disable_consumer | PROCEDURE | mysql.sys@localhost | 2023-09-24 11:02:38 | 2023-09-24 11:02:38 | 2023-09-24 11:02:38 | |
| 3 sys | ps_setup_disable_instrument | PROCEDURE | mysql.sys@localhost | 2023-09-24 11:02:38 | 2023-09-24 11:02:38 | 2023-09-24 11:02:38 | |
| 4 sys | ps_setup_disable_thread | PROCEDURE | mysql.sys@localhost | 2023-09-24 11:02:38 | 2023-09-24 11:02:38 | 2023-09-24 11:02:38 | |
| 5 sys | ps_setup_enable_background_threads | PROCEDURE | mysql.sys@localhost | 2023-09-24 11:02:38 | 2023-09-24 11:02:38 | 2023-09-24 11:02:38 | |
| 6 sys | ps_setup_enable_consumer | PROCEDURE | mysql.sys@localhost | 2023-09-24 11:02:38 | 2023-09-24 11:02:38 | 2023-09-24 11:02:38 | |
| 7 sys | ps_setup_enable_instrument | PROCEDURE | mysql.sys@localhost | 2023-09-24 11:02:38 | 2023-09-24 11:02:38 | 2023-09-24 11:02:38 | |
| 8 sys | ps_setup_enable_thread | PROCEDURE | mysql.sys@localhost | 2023-09-24 11:02:38 | 2023-09-24 11:02:38 | 2023-09-24 11:02:38 | |

- 显示存储过程创建代码

```
SHOW CREATE PROCEDURE try_select;
```

| SHOW CREATE PROCEDURE try_select; | | Create Procedure | char |
|-----------------------------------|--|---|---------|
| Procedure | sql_mode | | |
| try_select | ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,NO_ENGINE_SUBSTITUTION | CREATE DEFINER='root'@'localhost' PROCEDURE `try_select`() BEGIN DECLARE i INT DEFAULT 1; SET t2 = 0; WHILE i<=t1 DO SET t2=t2+i; SET i=i+1; END WHILE; END | utf8mb4 |

- 删除存储过程

```

DROP PROCEDURE try_select;
SHOW PROCEDURE STATUS WHERE DB = 'game';

```

```
DROP PROCEDURE try_select;
SHOW PROCEDURE STATUS WHERE DB = 'game';
```

| Db | Name | Type | Definer | Modified | Created | Security_type |
|----|------|------|---------|----------|---------|---------------|
|----|------|------|---------|----------|---------|---------------|

数据库管理

本章节将介绍如何使用数据库管理功能，包括创建与管理用户，管理用户权限、更改用户密码、数据库备份与恢复等。

关于 MySQL 的权限简单的理解就是 MySQL 允许你做你权利以内的事情，不可以越界。MySQL 服务器通过权限表来控制用户对数据库的访问，权限表存放在 mysql 数据库中，由 mysql_install_db 脚本初始化。

存储账户权限信息表主要有：user、db、tables_priv、columns_priv、procs_priv、proxies_priv这六张表（5.6之前还有host表，现在已经把host内容整合进user表）

| 权限 | 描述 |
|-------------------------|--|
| All/All Privileges | 全局或者全数据库对象级别的所有权限 |
| Alter | 修改表结构的权限，但必须要求有create和insert权限配合 |
| Alter routine | 修改或者删除存储过程、函数的权限 |
| Create | 创建新的数据库和表的权限 |
| Create routine | 允许创建存储过程、函数的权限 |
| Create tablespace | 允许创建、修改、删除表空间和日志组的权限 |
| Create temporary tables | 创建临时表权限 |
| Create user | 创建、修改、删除、重命名user |
| Create view | 创建视图 |
| Delete | 删除行数据 |
| drop | 删除数据库、表、视图的权限，包括truncate table命令 |
| Event | 查询，创建，修改，删除MySQL事件 |
| Execute | 执行存储过程和函数 |
| File | 在MySQL可以访问的目录进行读写磁盘文件操作，可使用的命令包括load data infile,select ,into outfile,load file()函数等 |
| Grant option | 授权或者收回给其他用户你给予的权限 |
| Index | 创建和删除索引 |
| Insert | 在表里插入数据 |
| Lock | 对拥有select权限的表进行锁定，以防止其他链接对此表的读或写 |
| Process | 允许查看MySQL中的进程信息，比如执行show processlist, mysqladmin processlist, show engine等命令 |
| Reference | 允许创建外键 |
| Reload | 执行flush命令，指明重新加载权限表到系统内存中 |

| 权限 | 描述 |
|--------------------|--|
| Replication client | 执行show master status,show slave status,show binary logs命令 |
| Replication slave | 允许slave主机通过此用户连接master以便建立主从复制关系 |
| Select | 从表中查看数据 |
| Show databases | 通过执行show databases命令查看所有的数据库名 |
| Show view | 通过执行show create view命令查看视图创建的语句 |
| Shutdown | 关闭数据库实例，执行语句包括mysqladmin shutdown |
| Super | 允许执行一系列数据库管理命令，包括kill强制关闭某个连接命令，change master to创建复制关系命令，以及create/alter/drop server等命令 |
| Trigger | 允许创建，删除，执行，显示触发器的权限 |
| Update | 修改表中数据的权限 |
| Usage | 创建一个用户之后的默认权限，本身代表无权限 |

用户管理基础

用户是 MySQL 认证的基本元素。您只能通过正确的用户名和密码登录进入 MySQL 数据库，然后授予用户不同的权限，以便让不同的用户可以进行不同的操作。

创建用户

创建用户是精确控制权限的第一步。在 MySQL 中，可以使用 CREATE USER 语句在数据库服务器中创建一个新用户。

语法

```
CREATE USER [IF NOT EXISTS] user_name[@host_name]
IDENTIFIED BY 'password';
```

其中：

- `user_name@host_name`：用户名和从该用户连接到 MySQL 服务器的主机名

如果 `username` 和 `hostname` 中包含空格或 - 等特殊字符，则需要将用户名和主机名分别按如下方式引用：

```
# 除了单引号 ('), 还可以使用反引号 (`) 或双引号 (").
'username'@'host_name'
```

`@host_name` 是可选的，如果省略，则表示用户可以从任何主机连接，也等效于使用 `@%`

- `IDENTIFIED BY 'password'`：指定用户密码

需要注意的是，`CREATE USER` 语句创建的用户默认是没有权限的，要授予权限请参看[下一章](#)，使用 `GRANT` 语句来授予权限。

示例

使用 `root` 用户登录到 MySQL 服务器，然后创建一个用户：

由于这里使用的是 DataGrip 工具，登录操作可以不需要，我们直接创建用户

```
# 通过 MySQL 系统表查看当前全部的用户
SELECT user, host FROM mysql.user;
# 创建名字为 user1 的用户
CREATE USER 'user1'@'%' IDENTIFIED BY '111111';
SELECT user, host FROM mysql.user;
```

```
# 通过 MySQL 系统表查看当前全部的用户
SELECT user, host FROM mysql.user;
```

| | user | host |
|---|------------------|-----------|
| 1 | mysql.infoschema | localhost |
| 2 | mysql.session | localhost |
| 3 | mysql.sys | localhost |
| 4 | root | localhost |

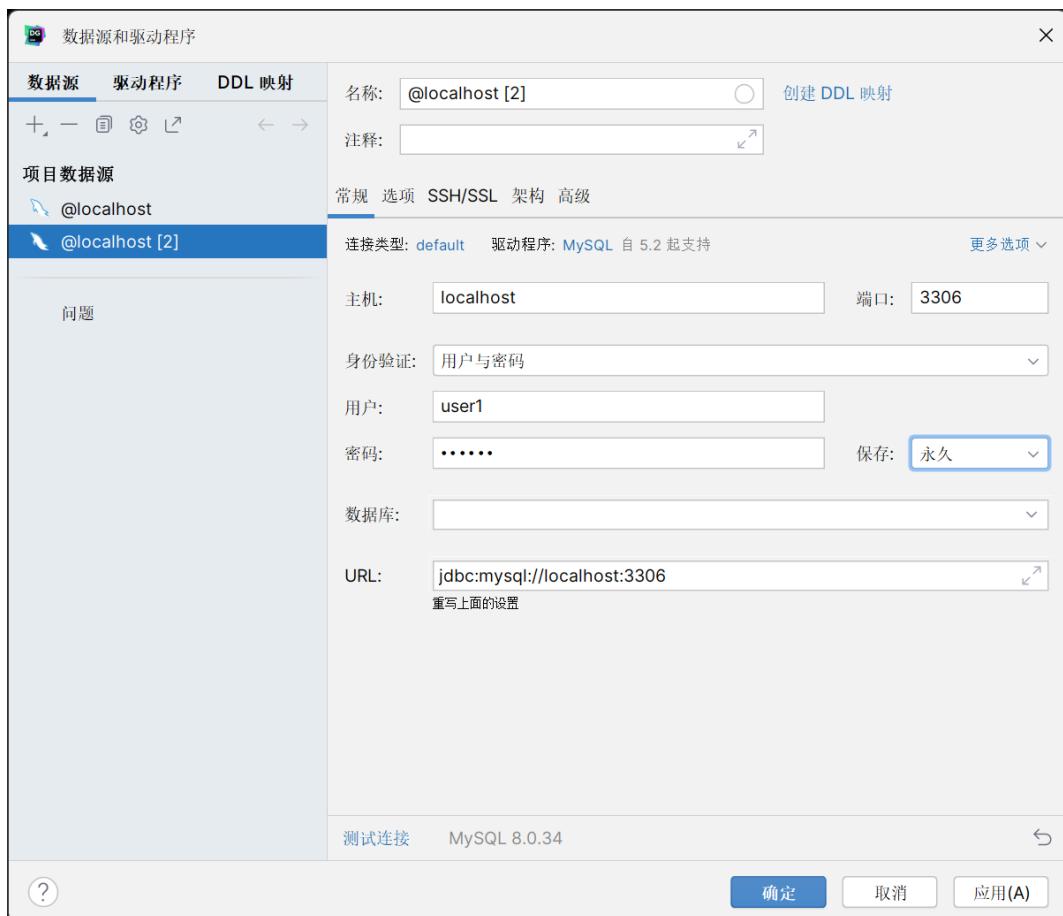
```
# 创建名字为 user1 的用户
CREATE USER 'user1'@'%' IDENTIFIED BY '111111';
SELECT user, host FROM mysql.user;
```

| | user | host |
|---|------------------|-----------|
| 1 | user1 | % |
| 2 | root | localhost |
| 3 | mysql.sys | localhost |
| 4 | mysql.session | localhost |
| 5 | mysql.infoschema | localhost |

可以看到，创建用户后，系统表 mysql.user 中多了一个 user1 的用户。

现在我们使用 user1 用户登录到 MySQL 服务器，查看其可访问的数据库列表

在 DataGrip 中，在左侧的数据库资源管理器中点击 + 号，选中数据源，找到 MySQL 按照下图设置并应用。



在 user1 登录的数据源中打开 console 窗口，输入 `SHOW DATABASES;` 命令，查看 user1 可访问的数据库列表。

```
# 查看 user1 可访问的数据库列表
SHOW DATABASES;
```

查看 user1 可以访问的数据库

`SHOW DATABASES;`

| | Database |
|---|--------------------|
| 1 | information_schema |
| ? | performance_schema |

删除用户

某些特定的场景下，需要删除已有的用户，比如：

- 此用户账户已经不被使用
- 此用户账户已经被泄露

要从 MySQL 服务器中删除用户帐户，需要使用 DROP USER 语句

语法

```
DROP USER [IF EXISTS] account_name [,account_name2]...;
```

可以一次删除多个账户

DROP USER 语句不但删除用户帐户，而且从所有授权表中删除用户的权限。

示例

我们继续使用 root 用户登录到 MySQL 服务器，删除我们在上一节创建的 user1 用户

```
DROP USER 'user1'@'%';
SELECT user, host FROM mysql.user;
```

`SELECT user, host FROM mysql.user;`

| | user | host |
|---|------------------|-----------|
| 1 | user1 | % |
| 2 | root | localhost |
| 3 | mysql.sys | localhost |
| 4 | mysql.session | localhost |
| 5 | mysql.infoschema | localhost |

`DROP USER 'user1'@'%';
SELECT user, host FROM mysql.user;`

| | user | host |
|---|------------------|-----------|
| 1 | mysql.infoschema | localhost |
| 2 | mysql.session | localhost |
| 3 | mysql.sys | localhost |
| 4 | root | localhost |

可以看到，user1 用户已经被删除。

杀掉已删除用户的会话

如果删除的用户在删除之前已经登录了一个会话，删除后并不会影响此会话，直到会话结束。这也会带来危险。

我们可以使用 `SHOW PROCESSLIST;` 语句来查看当前正在运行的会话列表。

```
SHOW PROCESSLIST;
```

| | Id | User | Host | db | Command | Time | State |
|---|----|-----------------|-----------------|--------------------|---------|--------|------------------------|
| 1 | 5 | event_scheduler | localhost | <null> | Daemon | 266099 | Waiting on empty queue |
| 2 | 48 | root | localhost:8867 | game | Query | 0 | init |
| 3 | 55 | user1 | localhost:10209 | information_schema | Sleep | 639 | |

如您所见，用户帐户 `user1@localhost` 会话仍然存在，其 ID 为 55。我们可以使用 `KILL` 语句终止会话 55：

```
KILL 55;
SHOW PROCESSLIST;
```

| | Id | User | Host | db | Command | Time | State |
|---|----|-----------------|-----------------|--------------------|---------|--------|------------------------|
| 1 | 5 | event_scheduler | localhost | <null> | Daemon | 266099 | Waiting on empty queue |
| 2 | 48 | root | localhost:8867 | game | Query | 0 | init |
| 3 | 55 | user1 | localhost:10209 | information_schema | Sleep | 639 | |

KILL 55;
SHOW PROCESSLIST;

| | Id | User | Host | db | Command | Time | State |
|---|----|-----------------|----------------|--------|---------|--------|------------------------|
| 1 | 5 | event_scheduler | localhost | <null> | Daemon | 266274 | Waiting on empty queue |
| 2 | 48 | root | localhost:8867 | game | Query | 0 | init |

用户重命名

某些特定的场景下，需要重命名已有的用户，比如：

- 此用户账户已经被泄露
- 将用户名改为一个更有意义的用户名

要从 MySQL 服务器中重命名一个或者多个已有的用户账户，需要使用 `RENAME USER` 语句。

语法

```
RENAME USER
  user_account TO new_user_account
[, user_account2 TO new_user_account2]
[, ...];
```

其中：

- 在 TO 关键字之前指定要重命名的现有用户账户。

- 在 TO 关键字之后指定新用户账户。

不能使用一个已有的用户账户名作为新的用户账户名

注意事项

`RENAME USER` 将旧用户的所有权限转移给新用户。但是，它不会删除或使依赖于旧用户的数据库对象无效。

例如，假设您有一个存储过程，其 `DEFINER` 属性指定了旧用户。这个存储过程在定义者安全上下文中执行。如果重命名旧用户，则执行存储过程时会出现错误。

示例

```
# 创建三个用户
CREATE USER 'user1'@'%' IDENTIFIED BY '111111';
CREATE USER 'user2'@'%' IDENTIFIED BY '222222';
CREATE USER 'user3'@'%' IDENTIFIED BY '333333';
# 查看用户列表
SELECT user, host FROM mysql.user;
```

```

# 创建三个用户
CREATE USER 'user1'@'%' IDENTIFIED BY '111111';
CREATE USER 'user2'@'%' IDENTIFIED BY '222222';
CREATE USER 'user3'@'%' IDENTIFIED BY '333333';
# 查看用户列表
SELECT user, host FROM mysql.user;

```

| | User | Host |
|---|------------------|-----------|
| 1 | user1 | % |
| 2 | user2 | % |
| 3 | user3 | % |
| 4 | mysql.infoschema | localhost |
| 5 | mysql.session | localhost |
| 6 | mysql.sys | localhost |
| 7 | root | localhost |

使用 RENAME USER 语句将 user1 @ % 重命名为 user4 @ localhost

```

RENAME USER 'user1'@'%' TO 'user4'@'localhost';
SELECT user, host FROM mysql.user;

```

```

RENAME USER 'user1'@'%' TO 'user4'@'localhost';
SELECT user, host FROM mysql.user;

```

| | User | Host |
|---|------------------|-----------|
| 1 | user2 | % |
| 2 | user3 | % |
| 3 | mysql.infoschema | localhost |
| 4 | mysql.session | localhost |
| 5 | mysql.sys | localhost |
| 6 | root | localhost |
| 7 | user4 | localhost |

修改用户密码

在 MySQL 中，可以使用 UPDATE, SET PASSWORD 和 ALTER USER 语句来更改用户的密码。

请注意，从 MySQL 5.7.6 开始，用户表使用列 authentication_string 来存储密码并且删除了 password 列，并且在 MySQL 5.7.9 以后废弃了 PASSWORD() 函数，在 MySQL 8.0 以后版本中，我们使用 ALTER USER 语句来修改密码。

语法

```
ALTER USER 'user_name'@'host_name' IDENTIFIED BY 'new_password';
```

示例

```
SELECT user, host, authentication_string FROM mysql.user WHERE user = 'user4';
ALTER USER 'user4'@'localhost' IDENTIFIED BY '444444';
FLUSH PRIVILEGES;
SELECT user, host, authentication_string FROM mysql.user WHERE user = 'user4';
```

```
SELECT user, host, authentication_string FROM mysql.user WHERE user = 'user4';
+-----+-----+-----+
| user | host | authentication_string |
+-----+-----+-----+
| user4 | localhost | $A$005$|BELCANUDC4$SUB9I_FM/8;ENQ=wPp6vEAgwuX3paI7MJ2c3RqDoEWKqFdRYK
ALTER USER 'user4'@'localhost' IDENTIFIED BY '444444';
FLUSH PRIVILEGES;
SELECT user, host, authentication_string FROM mysql.user WHERE user = 'user4';
+-----+-----+-----+
| user | host | authentication_string |
+-----+-----+-----+
| user4 | localhost | $A$005$y[ L7K$US5DELW2DELONAK-soTCZLmqvx77fvo.LZXo/ba6fTMgsPQeJWC
```

虽然不是明文，但是还是可以大概看出来密码变了

如果您想重置 MySQL root 帐户的密码，则需要在不使用授权表验证的情况下强制 MySQL 数据库服务器停止并重新启动。

查看用户

和查看数据库和数据表不同，查看用户不是使用 `SHOW USERS` 语句，而是通过对表 `user` 查询

在 `mysql.user` 表中有很多列，分别保存了用户的各种各样的信息，比如密码，密码有效期，是否锁定 和 各种权限等。

```
SELECT user, host FROM mysql.user;
# 查看 mysql.user 表的所有列
DESC mysql.user;
```

| SELECT user, host FROM mysql.user; | |
|------------------------------------|-----------|
| user | host |
| 1 user2 | % |
| 2 user3 | % |
| 3 mysql.infoschema | localhost |
| 4 mysql.session | localhost |
| 5 mysql.sys | localhost |
| 6 root | localhost |
| 7 user4 | localhost |

| # 查看 mysql.user 表的所有列 | | | | | | |
|-----------------------|---------------|------|-----|---------|--|--|
| DESC mysql.user; | | | | | | |
| Field | Type | Null | Key | Default | | |
| 1 Host | char(255) | NO | PRI | | | |
| 2 User | char(32) | NO | PRI | | | |
| 3 Select_priv | enum('N','Y') | NO | | N | | |
| 4 Insert_priv | enum('N','Y') | NO | | N | | |
| 5 Update_priv | enum('N','Y') | NO | | N | | |
| 6 Delete_priv | enum('N','Y') | NO | | N | | |
| 7 Create_priv | enum('N','Y') | NO | | N | | |
| 8 Drop_priv | enum('N','Y') | NO | | N | | |

显示当前使用的用户

```
SELECT USER();
SELECT CURRENT_USER();
```

列出当前登录 MySQL 数据库服务器的所有用户

```
SHOW PROCESSLIST;
SELECT user, host, db, command
FROM information_schema.processlist;
```

| ✗ | SELECT USER(); | | | | |
|---|--|--|----------------|---|----------------|
| | <table border="1"> <thead> <tr> <th></th> <th>USER()</th> </tr> </thead> <tbody> <tr> <td>1</td><td>root@localhost</td></tr> </tbody> </table> | | USER() | 1 | root@localhost |
| | USER() | | | | |
| 1 | root@localhost | | | | |
| ✓ | SELECT CURRENT_USER(); | | | | |
| ✗ | <table border="1"> <thead> <tr> <th></th> <th>CURRENT_USER()</th> </tr> </thead> <tbody> <tr> <td>1</td><td>root@localhost</td></tr> </tbody> </table> | | CURRENT_USER() | 1 | root@localhost |
| | CURRENT_USER() | | | | |
| 1 | root@localhost | | | | |

用户管理进阶

锁定用户账户

某些特定的场景下，需要锁定一个用户账户，比如：

- 创建一个锁定的用户，等授权完成后再解锁
- 此用户账户已经不被使用
- 此用户账户已经被泄露
- 此用户只是临时使用，使用完成后将用户锁定

要锁定一个已经存在的用户，使用 `ALTER USER .. ACCOUNT LOCK` 语句。

要直接创建一个锁定的用户，使用 `CREATE USER .. ACCOUNT LOCK` 语句。

查询用户的锁定状态

可以在 mysql 数据库中的 user 表中查看用户的锁定状态。mysql.user 表中的 account_locked 列中保存了帐户是否被锁定的状态： Y 指示了此用户被锁定， N 指示了此用户未锁定

```
SELECT user, host, account_locked FROM mysql.user;
```

```
SELECT user, host, account_locked FROM mysql.user;
```

| | user | host | account_locked |
|---|------------------|-----------|----------------|
| 1 | user2 | % | N |
| 2 | user3 | % | N |
| 3 | mysql.infoschema | localhost | Y |
| 4 | mysql.session | localhost | Y |
| 5 | mysql.sys | localhost | Y |
| 6 | root | localhost | N |
| 7 | user4 | localhost | N |

锁定现有的用户

```
ALTER USER 'user4'@'localhost' ACCOUNT LOCK;  
# 查看效果  
SELECT user, host, account_locked FROM mysql.user WHERE user = 'user4';
```

```

ALTER USER 'user4'@'localhost' ACCOUNT LOCK;
# 查看效果
SELECT user, host, account_locked FROM mysql.user WHERE user = 'user4';

```

| | user | host | account_locked |
|---|-------|-----------|----------------|
| 1 | user4 | localhost | Y |

可以看到用户 user4 已经被锁定，现在使用 user4 登录会被 MySQL 拒绝访问。

创建锁定的用户

```

CREATE USER 'user1'@'%' IDENTIFIED BY '111111' ACCOUNT LOCK;
SELECT user, host, account_locked FROM mysql.user WHERE user = 'user1';

```

```

CREATE USER 'user1'@'%' IDENTIFIED BY '111111' ACCOUNT LOCK;
SELECT user, host, account_locked FROM mysql.user WHERE user = 'user1';

```

| | user | host | account_locked |
|---|-------|------|----------------|
| 1 | user1 | % | Y |

可以看到我们直接创建了一个锁定的用户，它无法进行登录

查看锁定用户的连接次数

MySQL 维护了一个变量 Locked_connects，它用来保存锁定的用户尝试连接到服务器的次数。当锁定的帐户尝试登录时，Locked_connects 变量的值将加 1。

```

SHOW GLOBAL STATUS LIKE 'Locked_connects';
# 我们使用 user1 尝试登录一次
SHOW GLOBAL STATUS LIKE 'Locked_connects';

```

在 DataGrip 中添加数据源时选择用户就相当于自动执行登录操作，所以我们使用 user1 尝试登录一次，此时 Locked_connects 变量的值会加 1，我们的登陆操作也被拒绝

 @localhost [2]
[HY000][3118] Access denied for user
'user1'@'localhost'. Account is locked.

| | | |
|---|--|-------|
| | SHOW GLOBAL STATUS LIKE 'Locked_connects'; | |
| x | Variable_name | Value |
| 1 | Locked_connects | 0 |
| ✓ | SHOW GLOBAL STATUS LIKE 'Locked_connects'; | |
| x | Variable_name | Value |
| 1 | Locked_connects | 1 |

解锁用户账户

要解锁一个或多个锁定的用户，使用 ALTER USER .. ACCOUNT UNLOCK 语句。

语法

```
ALTER USER [IF EXISTS] user@host [, user@host, ...] ACCOUNT UNLOCK;
```

其实和先前锁定用户类似。

示例

解锁 user1 账户

| ALTER USER 'user1'@'%' ACCOUNT UNLOCK; | | | | | | |
|--|------|----------------|----------------|-------|---|---|
| SELECT user, host, account_locked FROM mysql.user WHERE user = 'user1'; | | | | | | |
| SELECT user, host, account_locked FROM mysql.user WHERE user = 'user1'; | | | | | | |
| <table border="1"> <thead> <tr> <th>user</th><th>host</th><th>account_locked</th></tr> </thead> <tbody> <tr> <td>user1</td><td>%</td><td>Y</td></tr> </tbody> </table> | user | host | account_locked | user1 | % | Y |
| user | host | account_locked | | | | |
| user1 | % | Y | | | | |
| ALTER USER 'user1'@'%' ACCOUNT UNLOCK; | | | | | | |
| SELECT user, host, account_locked FROM mysql.user WHERE user = 'user1'; | | | | | | |
| <table border="1"> <thead> <tr> <th>user</th><th>host</th><th>account_locked</th></tr> </thead> <tbody> <tr> <td>user1</td><td>%</td><td>N</td></tr> </tbody> </table> | user | host | account_locked | user1 | % | N |
| user | host | account_locked | | | | |
| user1 | % | N | | | | |

权限管理

作为一个数据库管理员或者维护人员，为了数据库的安全性，我们需要更精确的权限控制——授予不同的用户不同的权限，并在合适的时机收回权限（撤销授权）。

用户授权

当我们创建了一个新用户（没有锁定）之后，这个新的用户可以登录 MySQL 数据库服务器，但是他没有任何权限（只有一些基本的权限）。只有在赋予他数据库和相关表的权限之后，他才可以进行选择数据库和查询等操作。

在 MySQL 中，我们使用 `GRANT` 语句用于给用户赋予权限。

语法

```
GRANT privilege_type [, privilege_type, ...] ON privilege_object  
TO 'user'@'host' [IDENTIFIED BY 'password']  
[WITH {GRANT OPTION | resource_option}];
```

其中：

- `privilege_type`: 表示权限类型，可以是一个或多个。
常用的如：ALL、SELECT、INSERT、UPDATE、DELETE、DROP、ALTER 等。完整的权限列表参见 [MySQL 8.0 官网说明](#)
- `privilege_object`: 表示权限对象，可以是所有对象，也可以是某个数据库中的对象，表等。

```
# 分配全局权限，全部数据库全部对象的权限  
GRANT ALL ON *.* TO 'user'@'host';  
  
# 分配数据库全部对象权限  
GRANT ALL ON db_name.* TO 'user'@'host';  
  
# 分配表权限 SELECT  
GRANT SELECT ON db_name.tb_name TO 'user'@'host';
```

- `IDENTIFIED BY 'password'`: 可以用于重新设置密码。
- `WITH {GRANT OPTION | resource_option}`: 表示权限的附加选项。WITH GRANT OPTION 子句可以把自身权限授予其他用户或。此外，您可以使用 WITH resource_option 子句分配 MySQL 数据库服务器的资源。例如，设置用户每小时可以使用的连接数或语句数。

需要使用 `REVOKE ALL PRIVILEGES, GRANT OPTION FROM 'user'@'host'` 回收 WITH GRANT OPTION 权限

示例

我们为 user1, user2, user3 分别赋予不同的权限。并使用 SHOW GRANTS 查看权限。

```
GRANT ALL ON *.* TO 'user1'@'%';
GRANT ALL ON game.* TO 'user2'@'%';
GRANT SELECT ON game.items TO 'user3'@'%';

SHOW GRANTS FOR 'user1'@'%';
SHOW GRANTS FOR 'user2'@'%';
SHOW GRANTS FOR 'user3'@'%';
```

```
GRANT ALL ON *.* TO 'user1'@'%';
GRANT ALL ON game.* TO 'user2'@'%';
GRANT SELECT ON game.items TO 'user3'@'%';

SHOW GRANTS FOR 'user1'@'%';
    □ Grants for user1@%
1 GRANT APPLICATION_PASSWORD_ADMIN,AUDIT_ABORT_EXEMPT,AUDIT_ADMIN,AUTHENTICATION_POLICY_ADMIN,BACKUP_ADMIN,BINL...
2 GRANT SELECT, INSERT, UPDATE, DELETE, CREATE, DROP, RELOAD, SHUTDOWN, PROCESS, FILE, REFERENCES, INDEX, ALTER...

SHOW GRANTS FOR 'user2'@'%';
    □ Grants for user2@%
1 GRANT USAGE ON *.* TO `user2`@%
2 GRANT ALL PRIVILEGES ON `game`.* TO `user2`@%

SHOW GRANTS FOR 'user3'@'%';
    □ Grants for user3@%
1 GRANT USAGE ON *.* TO `user3`@%
2 GRANT SELECT ON `game`.`items` TO `user3`@%
```

现在，分别登入这三个账户，执行下列语句

```
SELECT CURRENT_USER;
SHOW DATABASES;
USE game;
SHOW TABLES;
SELECT * FROM items;
UPDATE items SET i_information = '一个历史悠久的古董，上面有着精美的花纹' WHERE i...
CREATE DATABASE IF NOT EXISTS test;
```

user1 具有全局权限，可以看到全部数据库对象，并具有全局下数据库权限。

```

SELECT CURRENT_USER;
+-----+
| user1@% |
+-----+
SHOW DATABASES;
+-----+
| Database |
| game    |
| information_schema |
| mysql   |
| performance_schema |
| sys     |
| test1  |
| test2  |
| test3  |
+-----+
USE game;
SHOW TABLES;
+-----+
| Tables_in_game |
| items          |
| market         |
| players        |
| records        |
| test           |
+-----+
SELECT * FROM items;
+----+----+----+----+----+----+----+----+
| i_id | i_name | i_weight | i_space | i_value | i_information | |
+----+----+----+----+----+----+----+
| 1001 | 花瓶    | 5          | 4        | 40       | 一个历史悠久的古董，上面有着精美的花纹! |
| 1002 | 保温水壶 | 1          | 2        | 1 <null> |                |
| 1003 | 茶壶    | 5          | 4        | 40       | <null>          |
| 1004 | 金豹雕像 | 4          | 2        | 2 <null> |                |
| 1005 | 金杯    | 4          | 2        | 4 <null> |                |
| 1006 | 金魔方  | 4          | 1        | 5 <null> |                |
| 1007 | 机密文件 | 1          | 2        | 600      | <null>          |
| 1008 | 未知物品 | 0          | 4        | 42       | <null>          |
+----+----+----+----+----+----+----+
UPDATE items SET i_information = '一个历史悠久的古董，上面有着精美的花纹!' WHERE i_id = 1001;
CREATE DATABASE IF NOT EXISTS test;

```

user2 具有数据库 game 权限，可以看到 game 数据库和默认的两个数据库，并具有 game 数据库下的全部权限，但是不具有全局下数据库权限，因此在创建数据库时报错。

```

! ✓ SELECT CURRENT_USER;
+-----+
| user2@% |
+-----+
! ✓ SHOW DATABASES;
+-----+
| Database |
| game    |
| information_schema |
| performance_schema |
+-----+
! ✓ USE game;
! ✓ SHOW TABLES;
+-----+
| Tables_in_game |
| items          |
| market         |
| players        |
| records        |
| test           |
+-----+
! ✓ SELECT * FROM items;
+----+----+----+----+----+----+----+
| i_id | i_name | i_weight | i_space | i_value | i_information | |
+----+----+----+----+----+----+----+
| 1005 | 金杯    | 4          | 2        | 4 <null> |                |
| 1006 | 金魔方  | 4          | 1        | 5 <null> |                |
| 1007 | 机密文件 | 1          | 2        | 600      | <null>          |
| 1008 | 金豹雕像 | 9          | 6        | 42       | <null>          |
| 1009 | 金块    | 4          | 2        | 5 <null> |                |
| 1010 | 金手镯  | 3          | 1        | 2 <null> |                |
| 1011 | 金手表  | 3          | 1        | 2 <null> |                |
+----+----+----+----+----+----+----+
! ✓ UPDATE items SET i_information = '一个历史悠久的古董，上面有着精美的花纹!' WHERE i_id = 1001;
! ✎ CREATE DATABASE IF NOT EXISTS test;

```

user3 具有数据库 game 下表 items 的 SELECT 权限，只能看到 game 数据库下的 items 表，只具有 items 表的 SELECT 权限，不具有其他权限，因此在更新数据和创建数据库时都报错。

```

1 ✓ SELECT CURRENT_USER;
x   □ `CURRENT_USER` □
1 user3@%
2 ✓ SHOW DATABASES;
x   □ Database □
1 game
2 information_schema
3 performance_schema
3 ✓ USE game;
4 ✓ SHOW TABLES;
x   □ Tables_in_game □
1 items
5 ✓ SELECT * FROM items;
x   □ i_id □ : □ i_name □ : □ i_weight □ : □ i_space □ : □ i_value □ : □ i_information □ : □
1 1001 花瓶      5     4     40 一个历史悠久的古董，上面有着精美的花纹 收藏
2 1002 保温水壶    1     2     1 <null> 收藏
3 1003 茶壶      5     4     40 <null> 收藏
4 1004 金豹雕像    4     2     2 <null> 收藏
5 1005 金杯      4     2     4 <null> 收藏
6 1006 金魔方    4     1     5 <null> 收藏
7 1007 机密文件    1     2     600 <null> 收藏
8 1008 金条      9     4     62 <null> 收藏
6 ⓘ UPDATE items SET i_information = '一个历史悠久的古董，上面有着精美的花纹' WHERE i_id = 1001;
7 CREATE DATABASE IF NOT EXISTS test;

```

撤销授权

如果面临以下的问题，我们需要撤销用户的权限：

- 授予了用户错误的权限
- 授权到期

MySQL 允许使用 REVOKE 语句撤销授予用户的权限。

语法

MySQL REVOKE 语句有几种形式。

撤销一项或多项权限

```

REVOKE
priv1 [, priv2 [, ...] ]
ON [object_type] privilege_level
FROM user1 [, user2 [, ...]];

```

撤销所有权限

主要用于收回 WITH GRANT OPTION 权限

```

REVOKE
ALL PRIVILEGES,
GRANT OPTION
FROM user1 [, user2 [, ...]];

```

示例

回收 user1, user2, user3 的权限。

```
REVOKE ALL PRIVILEGES, GRANT OPTION FROM 'user1'@'%';
REVOKE ALL ON game.* FROM 'user2'@'%';
REVOKE SELECT ON game.items FROM 'user3'@'%';
SHOW GRANTS FOR 'user1'@'%';
SHOW GRANTS FOR 'user2'@'%';
SHOW GRANTS FOR 'user3'@'%';
```

```
REVOKE ALL PRIVILEGES, GRANT OPTION FROM 'user1'@'%';
REVOKE ALL ON game.* FROM 'user2'@'%';
REVOKE SELECT ON game.items FROM 'user3'@'%';
SHOW GRANTS FOR 'user1'@'%';
```

| |
|---|
| <input type="checkbox"/> Grants for user1@% |
| 1 GRANT USAGE ON *.* TO `user1`@`%` |

```
SHOW GRANTS FOR 'user2'@'%';
```

| |
|---|
| <input type="checkbox"/> Grants for user2@% |
| 1 GRANT USAGE ON *.* TO `user2`@`%` |

```
SHOW GRANTS FOR 'user3'@'%';
```

| |
|---|
| <input type="checkbox"/> Grants for user3@% |
| 1 GRANT USAGE ON *.* TO `user3`@`%` |

撤销代理

要撤销代理用户，使用 REVOKE PROXY :

实际上在 MySQL 8.0 中几乎不使用代理用户，代理用户是 MySQL 中可以模拟另一个用户的有效用户，因此，代理用户拥有它模拟的用户的所有权限。

```
REVOKE PROXY ON proxied_user FROM proxy_user;
```

REVOKE 的生效时机

REVOKE 语句的生效时机取决于权限级别：

- 全局: 当用户帐户在后续会话中连接到 MySQL 服务器时, 更改生效。更改不会应用于所有当前连接的用户。
- 数据库级别: 更改在下 USE 一条语句后生效。
- 表和列级别: 更改对所有后续查询生效。

使用角色来简化授权

作为一个数据库管理员或者运维人员, 我们可能要对多个用户赋予相同的权限。这个过程很耗时, 也很容易带来错误。

MySQL 中的角色可以让你简化授权的过程。只需要为权限相同的用户创建一个角色, 并将角色赋予这些用户即可。

我们可以创建多个角色, 授权给不同的用户, 也可以为一个用户赋予多个不同的角色。

类似于 C++ 中类的组合与继承

语法

```
# 创建角色
CREATE ROLE role_name [, role_name [, ...]];

# 删除角色
DROP ROLE role_name [, role_name [, ...]];

# 角色赋权
GRANT privilege_type [, privilege_type [, ...]]
ON privilege_object
TO role_name;

# 撤销角色权限
REVOKE privilege_type [, privilege_type [, ...]]
ON privilege_object
FROM role_name;

# 分配角色
GRANT role_name [, role_name [, ...]] TO user_name [, user_name [, ...]];
```

角色名称类似于用户帐户, 由两部分组成: 名称和主机。如果省略主机部分, 则默认认为 %, 表示任何主机。

示例

我们为 game 表设计玩家角色 player, 管理者角色 manager, 开发者角色 developer。

```
CREATE ROLE player, manager, developer;
GRANT SELECT ON game.items TO player;
GRANT SELECT ON game.market TO player;
GRANT SELECT, INSERT, UPDATE ON game.items TO manager;
GRANT SELECT ON game.players TO manager;
GRANT ALL ON game.* TO developer;
# 为用户分配角色
GRANT player TO 'user1'@'%', 'user2'@'%';
GRANT manager TO 'user3'@'%';
GRANT developer TO 'user4'@'localhost';
# 查看结果
SHOW GRANTS FOR 'user1'@'%';
SHOW GRANTS FOR 'user2'@'%';
SHOW GRANTS FOR 'user3'@'%';
SHOW GRANTS FOR 'user4'@'localhost';
```

```

CREATE ROLE player, manager, developer;
GRANT SELECT ON game.items TO player;
GRANT SELECT ON game.market TO player;
GRANT INSERT, UPDATE ON game.records TO player;
GRANT SELECT, INSERT, UPDATE ON game.items TO manager;
GRANT SELECT ON game.players TO manager;
GRANT ALL ON game.* TO developer;
# 为用户分配角色
GRANT player TO 'user1'@'%', 'user2'@'%';
GRANT manager TO 'user3'@'%';
GRANT developer TO 'user4'@'localhost';
# 查看结果
SHOW GRANTS FOR 'user1'@'%';

```

| | <input type="checkbox"/> Grants for user1@% |
|---|---|
| 1 | GRANT USAGE ON *.* TO `user1`@`%` |
| 2 | GRANT `player`@`%` TO `user1`@`%` |

```
SHOW GRANTS FOR 'user2'@'%';
```

| | <input type="checkbox"/> Grants for user2@% |
|---|---|
| 1 | GRANT USAGE ON *.* TO `user2`@`%` |
| 2 | GRANT `player`@`%` TO `user2`@`%` |

```
SHOW GRANTS FOR 'user3'@'%';
```

| | <input type="checkbox"/> Grants for user3@% |
|---|---|
| 1 | GRANT USAGE ON *.* TO `user3`@`%` |
| 2 | GRANT `manager`@`%` TO `user3`@`%` |

```
SHOW GRANTS FOR 'user4'@'localhost';
```

| | <input type="checkbox"/> Grants for user4@localhost |
|---|---|
| 1 | GRANT USAGE ON *.* TO `user4`@`localhost` |
| 2 | GRANT `developer`@`%` TO `user4`@`localhost` |

SHOW GRANTS 只返回授予的角色。要显示角色代表的权限，需要使用带有授予角色名称的 USING 子句，我们将在下一小节具体介绍

设置默认角色

现在我们使用 user1 登录 MySQL 数据库，并访问 game 数据库。

```
USE game;
```

发出错误信息：

```
[2024-05-20 12:08:19] 已连接
> USE game
[2024-05-20 12:08:20] [42000][1044] Access denied for user 'user1'@'%' to database 'game'
```

这是因为当您向用户帐户授予角色时，并不会在用户帐户连接到数据库服务器时自动使角色变为活动状态。

我们调用 CURRENT_ROLE() 函数来查看当前活动的角色。

```
SELECT CURRENT_ROLE();
```

DataGrip 对 CURRENT_ROLE() 有着莫名其妙的报错，但是运行是正常的

| 1 | NONE |
|---|------|
| 1 | NONE |

返回 NONE，这意味着没有活动角色。要指定每次用户帐户连接到数据库服务器时应激活哪些角色，需要使用 SET DEFAULT ROLE 语句。

```
# 使用 root 用户执行
SET DEFAULT ROLE player TO 'user1'@'%';
# 重新登录 user1
SELECT CURRENT_ROLE();
USE game;
```

The screenshot shows the MySQL Workbench interface. At the top, there are several icons: a clock, a circular arrow, a gear, and a refresh symbol. To the right of these are dropdown menus for 'Tx: 自动' (Transaction: Auto) and 'Playground'. Below the toolbar, a query editor window is open. It contains the following SQL code:

```
SELECT CURRENT_ROLE();
```

The result set shows one row with the value 'player'@'%'. Below the result set, another query is visible:

```
USE game;
```

设置活跃角色

用户帐户可以通过指定哪个授予的角色处于活动状态来修改当前用户在当前会话中的有效权限。

```
# 将活动角色设置为 NONE, 表示没有活动角色。  
SET ROLE NONE;  
# 将活动角色设置为所有授予的角色  
SET ROLE ALL;  
# 将活动角色设置为 SET DEFAULT ROLE 语句设置的默认角色  
SET ROLE DEFAULT;
```

粘贴用户权限

MySQL 将用户帐户视为角色，因此，您可以将一个用户帐户授予另一个用户帐户，就像向该用户帐户授予角色一样。这允许您将权限从一个用户复制到另一个用户。

显示权限

MySQL 用 SHOW GRANTS 语句来显示分配给用户帐户或角色的权限。

语法

```
SHOW GRANTS  
[FOR {user | role}  
[USING role [, role] ...]];
```

其中：

- FOR 关键字后指定要显示先前授予用户帐户或角色的权限的用户帐户或角色的名称。如果跳过 FOR 子句，则 SHOW GRANTS 返回当前用户的权限。

- USING 子句检查与用户角色相关的权限。在 USING 子句中指定的角色必须事先授予用户。

除了可以显示当前用户的权限和角色，要执行 SHOW GRANTS 的语句，你需要有 mysql 系统数据库的 SELECT 权限。

示例

```
# 显示当前用户的权限
SHOW GRANTS;
# 显示 user1 的权限
SHOW GRANTS FOR 'user1'@'%';
# 显示 user1 的权限，包括 player 授予的权限
SHOW GRANTS FOR 'user1'@'%' USING 'player'@'%';
```

DataGrip 对 USING 有着莫名其妙的报错，但是运行是正常的，我猜测是因为我将 DataGrip 的 MySQL 驱动更新到了 8.2 导致的部分兼容问题，在 8.0 的文档中显示 USING 子句是可选合法的

```
# 显示当前用户的权限
SHOW GRANTS;
+ Grants for root@localhost +
1 GRANT SELECT, INSERT, UPDATE, DELETE, CREATE, DROP, RELOAD, SHUTDOWN, PROCESS, FILE, REFERENCES, INDEX, ALTER...
2 GRANT APPLICATION_PASSWORD_ADMIN,AUDIT_ABORT_EXEMPT,AUDIT_ADMIN,AUTHENTICATION_POLICY_ADMIN,BACKUP_ADMIN,BINL...
3 GRANT PROXY ON `''@` TO `root`@`localhost` WITH GRANT OPTION
+ Grants for user1@% +
1 GRANT USAGE ON *.* TO `user1`@`%`
2 GRANT `player`@`%` TO `user1`@`%`
+ Grants for user1@% USING `player`@`%` +
1 GRANT USAGE ON *.* TO `user1`@`%`
2 GRANT SELECT ON `game`.`items` TO `user1`@`%`
3 GRANT SELECT ON `game`.`market` TO `user1`@`%`
4 GRANT INSERT, UPDATE ON `game`.`records` TO `user1`@`%`
5 GRANT `player`@`%` TO `user1`@`%`
```

维护表

定期的维护数据表是一个很好的习惯。对提高数据库的性能很有帮助。

MySQL 提供了几个维护数据库表的语句：

| 语句 | 作用 |
|----------------|-----|
| ANALYZE TABLE | 分析表 |
| OPTIMIZE TABLE | 优化表 |
| CHECK TABLE | 检查表 |
| REPAIR TABLE | 修复表 |

分析表

MySQL 使用 ANALYZE TABLE 语句分析表，它用于分析和存储键的分布。分析表的结果将可以使得系统得到准确的统计信息，使得 SQL 能够生成正确的执行计划。

典型的，在一个表中进行了大量数据插入，更新或者删除操作后，键分布可能是不准确的。如果键分布不准确，查询优化器可能会选择错误的查询执行计划，这可能会导致严重的性能问题。

```
ANALYZE TABLE records;
ANALYZE TABLE items, players;
```

```
ANALYZE TABLE records;
```

| Table | Op | Msg_type | Msg_text |
|--------------|---------|----------|----------|
| game.records | analyze | status | OK |

```
ANALYZE TABLE items, players;
```

| Table | Op | Msg_type | Msg_text |
|--------------|---------|----------|----------|
| game.items | analyze | status | OK |
| game.players | analyze | status | OK |

优化表

MySQL 使用 OPTIMIZE TABLE 语句优化表，它的主要作用是消除更新或者删除造成空间浪费。

典型的，在一个表中进行了大量数据更新或者删除操作后，表的物理存储可能变得碎片化，导致数据库服务器的性能下降。

```
OPTIMIZE TABLE records;  
OPTIMIZE TABLE items, players, market;
```

不过按照文档的说法直接优化 InnoDB 表会报错，如图所示 Table does not support optimize，似乎需要用 mysql --skip-new 或者 mysql --safe-mode 命令来重启 MySQL，参见[回答](#)

| OPTIMIZE TABLE records; | | | |
|-------------------------|--------------|----------|---|
| | Table | Op | Msg_text |
| 1 | game.records | optimize | note
Table does not support optimize, doing r... |
| 2 | game.records | optimize | status
OK |

| OPTIMIZE TABLE items, players, market; | | | |
|--|--------------|----------|---|
| | Table | Op | Msg_text |
| 1 | game.items | optimize | note
Table does not support optimize, doing r... |
| 2 | game.items | optimize | status
OK |
| 3 | game.players | optimize | note
Table does not support optimize, doing r... |
| 4 | game.players | optimize | status
OK |
| 5 | game.market | optimize | note
Table does not support optimize, doing r... |
| 6 | game.market | optimize | status
OK |

检查表

数据库服务器可能发生一些错误，例如服务器意外关闭、向硬盘写入数据时出错等。这些情况可能导致数据库运行不正确，最坏的情况可能是崩溃。

MySQL 允许您使用 CHECK TABLE 语句检查数据库表的完整性。

```
CHECK TABLE records;  
CHECK TABLE items, players, market;
```

CHECK TABLE 语句仅检测数据库表中的问题，但不会修复它们。要修复表，请使用 REPAIR TABLE 语句。

| CHECK TABLE records; | | | |
|----------------------|--------------|-------|--------------|
| | Table | Op | Msg_text |
| 1 | game.records | check | status
OK |

| CHECK TABLE items, players, market; | | | |
|-------------------------------------|--------------|-------|--------------|
| | Table | Op | Msg_text |
| 1 | game.items | check | status
OK |
| 2 | game.players | check | status
OK |
| 3 | game.market | check | status
OK |

修复表

REPAIR TABLE 语句允许您修复数据库表中发生的一些错误。MySQL 不保证 REPAIR TABLE 语句可以修复表可能存在的所有错误。

REPAIR TABLE 只支持 MyISAM 引擎。否则会给出提示: The storage engine for the table doesn't support repair。

我们使用的是 InnoDB 引擎, 所以无法使用 REPAIR TABLE。

```
REPAIR TABLE records;
REPAIR TABLE items, players, market;
```

| REPAIR TABLE records;
REPAIR TABLE items, players, market; | | | |
|---|--------|----------|---|
| Result 35 Result 35-2 × | | | |
| Table | Op | Msg_type | Msg_text |
| 1 game.items | repair | note | The storage engine for the table doesn't support rep... |
| 2 game.players | repair | note | The storage engine for the table doesn't support rep... |
| 3 game.market | repair | note | The storage engine for the table doesn't support rep... |

备份与恢复

作为一个数据库管理员或者运维人员，定期备份线上的 MySQL 数据库是一个很有必要的工作。它可能帮你在数据库遭到损坏的时候保留数据或者恢复备份。

MySQL 提供了 mysqldump 工具用于从 MySQL 数据库服务器中导出数据库结构和数据。

可以实现数据的完全备份、差分备份、增量备份，不过由于本次实验数据量较小，且使用 DataGrip 作为实验环境，仅测试完全备份，以便完全在 DataGrip 中进行。

更多备份方法可以参看这个[回答](#)

mysqldump 介绍

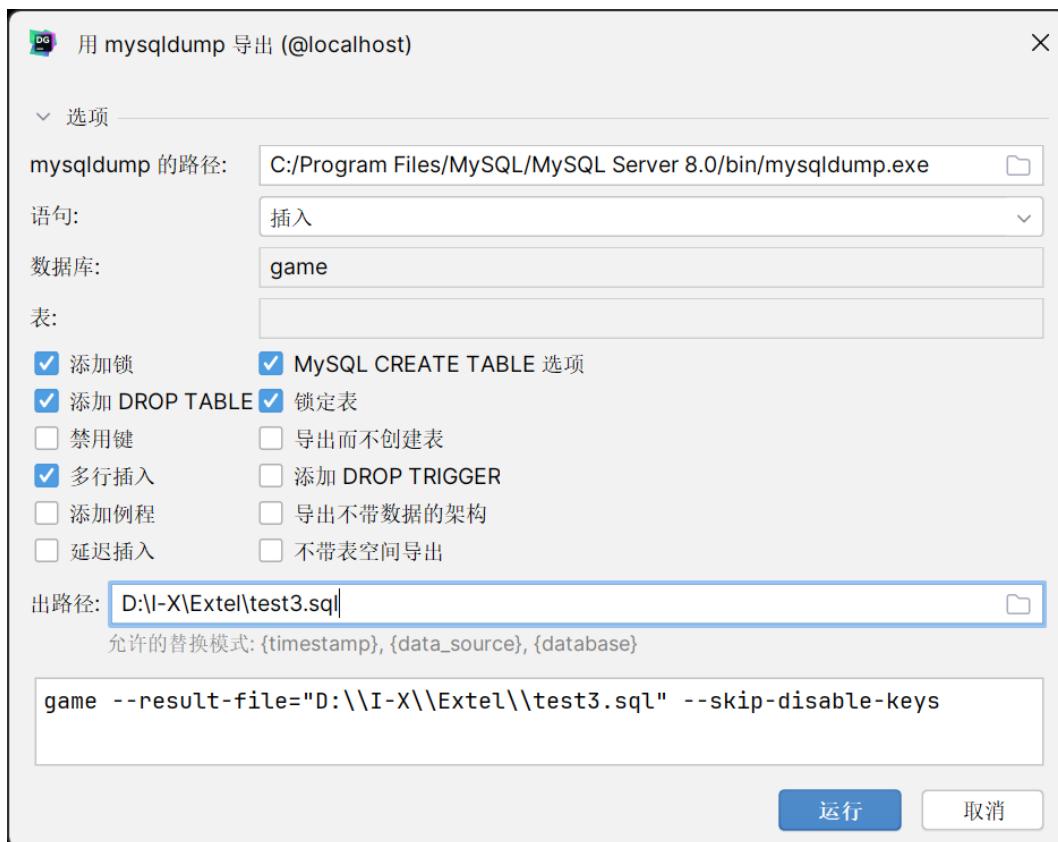
mysqldump 工具一般会随着安装 MySQL 数据库时自动安装。它能将一个或者多个数据库，或者数据库中的一个或者多个表导出为一个 SQL 文件，包括 DDL 语句和 DML 语句。

通常，安装完 MySQL 服务器后，可以直接使用 mysqldump 工具。

如果找不到 mysqldump 工具，请将 MySQL 安装目录下的 bin 目录配置到环境变量 PATH 中。或者导航到 MySQL 安装目录下的 bin 目录下再使用 mysqldump 工具。

在 DataGrip 中使用 mysqldump 完全备份数据库

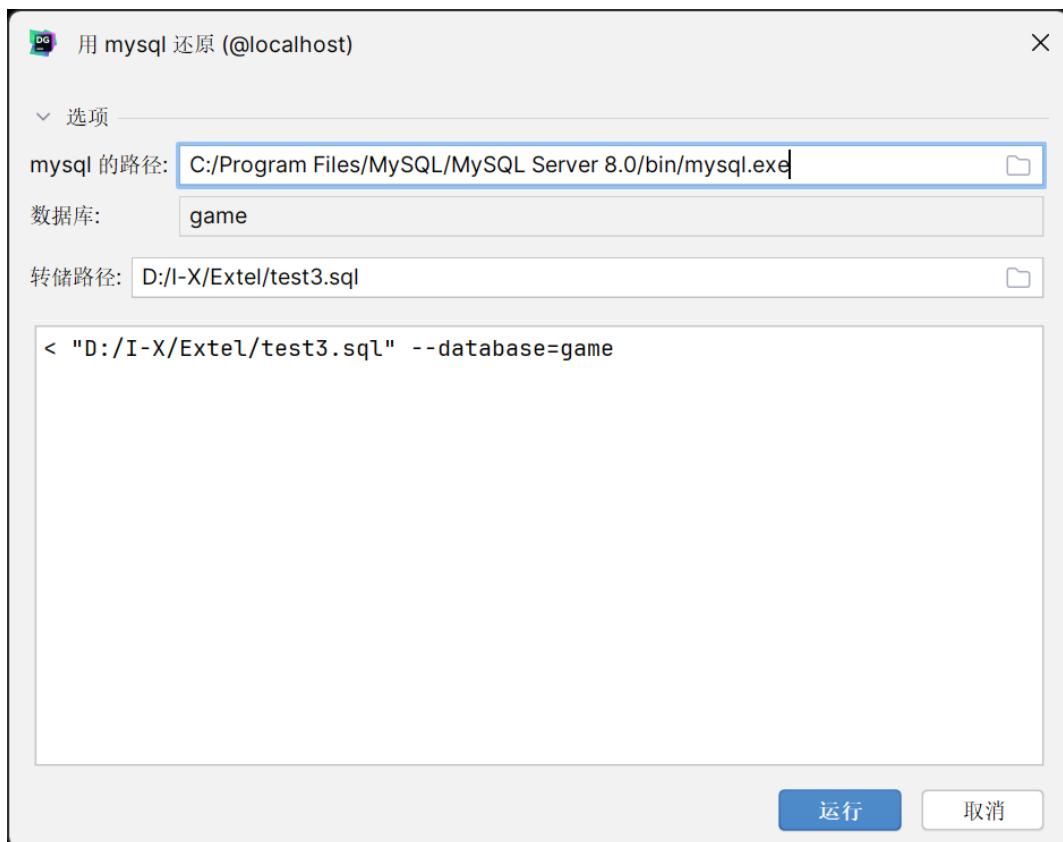
选中希望备份的数据库文件，右键选择 `导入/导出`，在弹出的选项中选择 `用'mysqldump'导出`，将会出现下述窗口



选择 mysqldump.exe 的路径和导出文件的路径，选择导出的规则，点击导出即可。

在 DataGrip 中通过备份文件恢复数据库

备份数据库后我们可以删除该数据库并进行还原，需要注意的是使用 DataGrip 进行还原时，如上图，我们选中 game 进行备份，实际上只备份了 game 内的表，因此如果删除 game 数据库，还原时需要先建 game 表，再选中 game，如上操作，选中使用 '使用'mysql'还原'，将出现下述窗口，选择 mysql.exe 路径，点击还原即可。



注意，还原后需要刷新数据库，否则无法看到还原效果，这是 DataGrip 的一个设计问题。