

# Mysql

## 安装过程

### 1. 官网下载Mysql最新版

地址: <https://dev.mysql.com/downloads/mysql/>

### 2. 解压到要安装的路径

- 在初始目录新建my.ini文件

```
1 [mysqld]
2 # 设置mysql安装目录
3 basedir=D:\mysql-8.0.21-winx64
4 # 设置mysql数据存放目录
5 datadir=D:\mysql-8.0.21-winx64\data
6 # 设置端口号
7 port=3306
```

- 再新建一个data文件 (结果长这样)

安装 (D:) > mysql-8.0.21-winx64				
搜索"mysql-8.0.21-winx64"				
名称	修改日期	类型	大小	
bin	2020/9/9 10:19	文件夹		
data	2020/9/9 10:23	文件夹		
docs	2020/9/9 10:19	文件夹		
include	2020/9/9 10:19	文件夹		
lib	2020/9/9 10:19	文件夹		
share	2020/9/9 10:19	文件夹		
LICENSE	2020/6/17 0:31	文件	396 KB	
my.ini	2020/9/9 10:24	配置设置	1 KB	
README	2020/6/17 0:31	文件	1 KB	

### 3. 以管理员身份运行cmd (否则会出现问题: Install/Remove of the Service Denied)

进入刚刚配置完的bin目录

- 输入 `mysql --initialize --console` 初始化数据库, 其中**标红部分**(root@localhost:后面)密码需要先记下来, 一会登录使用, 如果密码忘记就把data文件删了, 重新初始化数据库

```
D:\mysql-8.0.21-winx64\bin>mysql --initialize --console
2020-09-09T02:37:31.607529Z 0 [System] [MY-013169] [Server] D:\mysql-8.0.21-winx64\bin\mysqld.exe (mysqld 8.0.21) initializing of server in progress as process 364
2020-09-09T02:37:31.616838Z 1 [System] [MY-013576] [InnoDB] InnoDB initialization has started.
2020-09-09T02:37:37.811421Z 1 [System] [MY-013577] [InnoDB] InnoDB initialization has ended.
2020-09-09T02:37:48.250591Z 6 [Note] [MY-010454] [Server] A temporary password is generated for root@localhost: /JyHbli3#jal
D:\mysql-8.0.21-winx64\bin>
```

- 再然后输入 `mysql --install` 安装mysql 服务, 输入 `net start mysql` 启动服务

备注: `mysqld --remove` 是卸载MySQL服务, `net stop mysql` 是停止服务(注意使用顺序)

```
D:\mysql-8.0.21-winx64\bin>mysqld --install
Service successfully installed.

D:\mysql-8.0.21-winx64\bin>net start mysql
MySQL 服务正在启动 . . .
MySQL 服务已经启动成功。

D:\mysql-8.0.21-winx64\bin>
```

#### 4. 进入数据库配置

- 输入 `mysql -u root -p` 后会让你输入密码, 密码为前面让你记住的密码, 输入正确后就会出现如下界面, 表示进入了MySQL命令模式。

```
D:\mysql-8.0.21-winx64\bin>mysql -u root -p
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 9
Server version: 8.0.21

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

- 输入 `ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql_native_password BY '新密码';` 出现如下界面表示更改成功。

```
mysql> ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql_native_password BY 'root';
Query OK, 0 rows affected (0.06 sec)

mysql>
```

#### 5. 把Mysql的bin目录放入环境变量的path中

#### 6. 重启mysql即可正常使用

```
1 #退出mysql管理界面
2 exit
3 #停止mysql服务
4 net stop mysql
5 #开始mysql服务
6 net start mysql
```

## 数据库操作

### 登录数据库

```
1 # 登录数据库 (-root是密码)
2 mysql -uroot -root
3 # 上边方法我没登陆进去, 改用初始方法了
4 mysql -u root -p # 然后输入密码 root
5 # 退出数据库
6 exit;
```

## 基本语法（关键字不分大小写）

```
1  -- 显示所有数据库
2  show databases;
3
4  -- 创建数据库
5  CREATE DATABASE test;
6
7  -- 切换数据库
8  use test;
9
10 -- 显示数据库中的所有表
11 show tables;
12
13 -- 创建数据表
14 CREATE TABLE pet (
15     name VARCHAR(20),
16     owner VARCHAR(20),
17     species VARCHAR(20),
18     sex CHAR(1),
19     birth DATE,
20     death DATE
21 );
22
23 -- 查看数据表结构
24 describe pet;
25 -- 或者
26 desc pet;
27
28 -- 查询表
29 SELECT * from pet;
30
31 -- 插入数据
32 INSERT INTO pet VALUES ('puffball', 'Diane', 'hamster', 'f', '1990-03-30',
33     NULL);
34
35 -- 插入单个数据
36 INSERT INTO pet (name) VALUES ('puffball');
37
38 -- 修改数据
39 UPDATE pet SET name = 'squirrel' where owner = 'Diane';
40
41 -- 删除数据
42 DELETE FROM pet where name = 'squirrel';
43
44 -- 删除表
45 DROP TABLE myorder;
```

# 支持的数据类型

- 数值

类型	大小	范围 (有符号)	范围 (无符号)	用途
TINYINT	1 byte	(-128, 127)	(0, 255)	小整数值
SMALLINT	2 bytes	(-32 768, 32 767)	(0, 65 535)	大整数值
MEDIUMINT	3 bytes	(-8.3E+6, 8.3E+6)	(0, 1.6E+6)	大整数值
INT或 INTEGER	4 bytes	(-2.1E+9, 2.1E+9)	(0, E+9)	大整数值
BIGINT	8 bytes	(-9.2E+18, 9.2E+18)	(0, 1.8E+19)	极大整数值
FLOAT	4 bytes	(-3.4E+38, 3.4E+38)	(0, 3.4E+38)小数点9位	单精度浮点数值
DOUBLE	8 bytes	(-1.7E+308, 1.7E+308)	(0, 1.7E+308)小数点16位	双精度浮点数值
DECIMAL	对DECIMAL(M,D), 如果M>D, 为M+2否则为D+2	依赖于M和D的值	依赖于M和D的值	小数值

- 日期/时间

类型	大小 (bytes)	范围	格式	用途
DATE	3	1000-01-01/9999-12-31	YYYY-MM-DD	日期值
TIME	3	'-838:59:59'/'838:59:59'	HH:MM:SS	时间值或持续时间
YEAR	1	1901/2155	YYYY	年份值
DATETIME	8	1000-01-01 00:00:00/9999-12-31 23:59:59	YYYY-MM-DD HH:MM:SS	混合日期和时间值
TIMESTAMP	4	1970-01-01 00:00:00/2038结束时间是第 <b>2147483647</b> 秒，北京时间 <b>2038-1-19 11:14:07</b> ，格林尼治时间2038年1月19日 凌晨 03:14:07	YYYYMMDD HHMMSS	混合日期和时间值，时间戳

- 字符串

类型	大小	用途
CHAR	0-255 bytes	定长字符串
VARCHAR	0-65535 bytes	变长字符串
TINYBLOB	0-255 bytes	不超过 255 个字符的二进制字符串
TINYTEXT	0-255 bytes	短文本字符串
BLOB	0-65 535 bytes	二进制形式的长文本数据
TEXT	0-65 535 bytes	长文本数据
MEDIUMBLOB	0-16 777 215 bytes	二进制形式的中等长度文本数据
MEDIUMTEXT	0-16 777 215 bytes	中等长度文本数据
LOBLOB	0-4 294 967 295 bytes	二进制形式的极大文本数据
LONGTEXT	0-4 294 967 295 bytes	极大文本数据

# 建表约束（用describe table能看到约束）

## 主键约束 KEY(PRI)

创建时候设置主键（唯一标识），主键的内容**不可重复，不能为空**，否则报错，这样我们可以唯一确认一条记录

```
1  -- 主键约束
2  -- 使某个字段不重复且不得为空，确保表内所有数据的唯一性。
3  CREATE TABLE user (
4      id INT PRIMARY KEY,
5      name VARCHAR(20)
6  );
7
8  -- 联合主键
9  -- 联合主键中的每个字段都不能为空，并且加起来不能和已设置的联合主键重复。
10 CREATE TABLE user (
11     id INT,
12     name VARCHAR(20),
13     password VARCHAR(20),
14     PRIMARY KEY(id, name)
15 );
16
17 -- 自增约束
18 -- 自增约束的主键由系统自动递增分配（学号啥的）
19 CREATE TABLE user (
20     id INT PRIMARY KEY AUTO_INCREMENT,
21     name VARCHAR(20)
22 );
23
24 -- 添加主键约束
25 -- 如果忘记设置主键，还可以通过SQL语句设置（两种方式）：
26 ALTER TABLE user ADD PRIMARY KEY(id);
27 ALTER TABLE user MODIFY id INT PRIMARY KEY;
28
29 -- 删除主键
30 ALTER TABLE user drop PRIMARY KEY;
```

## 唯一约束 UNI

被唯一约束的字段不可以重复

```
1  -- 建表时创建唯一主键
2  CREATE TABLE user (
3      id INT,
4      name VARCHAR(20),
5      UNIQUE(name)
6  );
7
8  -- 添加唯一主键
```

```

9  -- 如果建表时没有设置唯一建，还可以通过SQL语句设置（两种方式）：
10 ALTER TABLE user ADD UNIQUE(name);
11 ALTER TABLE user MODIFY name VARCHAR(20) UNIQUE;
12
13 -- 删除唯一主键
14 ALTER TABLE user DROP INDEX name;

```

## 非空约束 NULL(NO)

被约束的字段不能为空

```

1  -- 建表时添加非空约束
2  -- 约束某个字段不能为空
3  CREATE TABLE user (
4      id INT,
5      name VARCHAR(20) NOT NULL
6  );
7
8  -- 移除非空约束
9  ALTER TABLE user MODIFY name VARCHAR(20);

```

## 默认约束 Default(数值)

被约束的字段有默认值

```

1  -- 建表时添加默认约束
2  -- 约束某个字段的默认值
3  CREATE TABLE user2 (
4      id INT,
5      name VARCHAR(20),
6      age INT DEFAULT 10
7  );
8
9  -- 移除非空约束
10 ALTER TABLE user MODIFY age INT;

```

## 外键约束 KEY(MUL)

涉及到两个表：父表，子表

主表：班级，副表：学生

```

1  -- 班级
2  CREATE TABLE classes (
3      id INT PRIMARY KEY,
4      name VARCHAR(20)
5  );
6
7  -- 学生表
8  CREATE TABLE students (
9      id INT PRIMARY KEY,

```

```

10     name VARCHAR(20),
11     -- 这里的 class_id 要和 classes 中的 id 字段相关联
12     class_id INT,
13     -- 表示 class_id 的值必须来自于 classes 中的 id 字段值
14     FOREIGN KEY(class_id) REFERENCES classes(id)
15 );
16
17 -- 1. 主表（父表）classes 中没有的数据值，在副表（子表）students 中，是不可以使用的；
18 -- 2. 主表中的记录被副表引用时，主表不可以被删除。

```

## 数据库的三大设计范式

### 1NF

只要字段值还可以继续拆分，就不满足第一范式。

- 地址：辽宁省沈阳市 一个字段（可拆）  
拆分成：辽宁省 + 沈阳市 两个字段（拆到不可拆）

范式设计得越详细，对某些实际操作可能会更好，但并非都有好处，需要对项目的实际情况进行设定。

### 2NF

在满足第一范式的前提下，其他列都必须完全依赖于主键列。如果出现不完全依赖，只可能发生在联合主键的情况下：

```

1  -- 订单表
2  CREATE TABLE myorder (
3      product_id INT,
4      customer_id INT,
5      product_name VARCHAR(20),
6      customer_name VARCHAR(20),
7      PRIMARY KEY (product_id, customer_id)
8  );

```

实际上，在这张订单表中，`product_name` 只依赖于 `product_id`，`customer_name` 只依赖于 `customer_id`。也就是说，`product_name` 和 `customer_id` 是没用关系的，`customer_name` 和 `product_id` 也是没有关系的。

这就不满足第二范式：其他列都必须完全依赖于主键列！

```

1  CREATE TABLE myorder (
2      order_id INT PRIMARY KEY,
3      product_id INT,
4      customer_id INT
5  );
6
7  CREATE TABLE product (

```



```

8      id INT PRIMARY KEY,
9      name VARCHAR(20)
10 );
11
12 CREATE TABLE customer (
13     id INT PRIMARY KEY,
14     name VARCHAR(20)
15 );

```

拆分之后，`myorder` 表中的 `product_id` 和 `customer_id` 完全依赖于 `order_id` 主键，而 `product` 和 `customer` 表中的其他字段又完全依赖于主键。满足了第二范式的设计！

## 3NF

在满足第二范式的前提下，除了主键列之外，其他列之间不能有传递依赖关系。

```

1 CREATE TABLE myorder (
2     order_id INT PRIMARY KEY,
3     product_id INT,
4     customer_id INT,
5     customer_phone VARCHAR(15)
6 );

```

表中的 `customer_phone` 有可能依赖于 `order_id`、`customer_id` 两列，也就不满足了第三范式的设计：其他列之间不能有传递依赖关系。

```

1 CREATE TABLE myorder (
2     order_id INT PRIMARY KEY,
3     product_id INT,
4     customer_id INT
5 );
6
7 CREATE TABLE customer (
8     id INT PRIMARY KEY,
9     name VARCHAR(20),
10    phone VARCHAR(15)
11 );

```

修改后就不存在其他列之间的传递依赖关系，其他列都只依赖于主键列，满足了第三范式的设计！

## 查询练习

---

### 准备数据

```

1 -- 创建数据库
2 CREATE DATABASE select_test;
3 -- 切换数据库
4 USE select_test;

```

```

5
6  -- 创建学生表
7  CREATE TABLE student (
8      no VARCHAR(20) PRIMARY KEY,
9      name VARCHAR(20) NOT NULL,
10     sex VARCHAR(10) NOT NULL,
11     birthday DATE, -- 生日
12     class VARCHAR(20) -- 所在班级
13 );
14
15  -- 创建教师表
16  CREATE TABLE teacher (
17      no VARCHAR(20) PRIMARY KEY,
18      name VARCHAR(20) NOT NULL,
19      sex VARCHAR(10) NOT NULL,
20      birthday DATE,
21      profession VARCHAR(20) NOT NULL, -- 职称
22      department VARCHAR(20) NOT NULL -- 部门
23 );
24
25  -- 创建课程表
26  CREATE TABLE course (
27      no VARCHAR(20) PRIMARY KEY,
28      name VARCHAR(20) NOT NULL,
29      t_no VARCHAR(20) NOT NULL, -- 教师编号
30      -- 表示该 tno 来自于 teacher 表中的 no 字段值
31      FOREIGN KEY(t_no) REFERENCES teacher(no)
32 );
33
34  -- 成绩表
35  CREATE TABLE score (
36      s_no VARCHAR(20) NOT NULL, -- 学生编号
37      c_no VARCHAR(20) NOT NULL, -- 课程号
38      degree DECIMAL, -- 成绩
39      -- 表示该 s_no, c_no 分别来自于 student, course 表中的 no 字段值
40      FOREIGN KEY(s_no) REFERENCES student(no),
41      FOREIGN KEY(c_no) REFERENCES course(no),
42      -- 设置 s_no, c_no 为联合主键
43      PRIMARY KEY(s_no, c_no)
44 );
45
46  -- 查看所有表
47  SHOW TABLES;
48
49  -- 添加学生表数据
50  INSERT INTO student VALUES('101', '曾华', '男', '1977-09-01', '95033');
51  INSERT INTO student VALUES('102', '匡明', '男', '1975-10-02', '95031');
52  INSERT INTO student VALUES('103', '王丽', '女', '1976-01-23', '95033');
53  INSERT INTO student VALUES('104', '李军', '男', '1976-02-20', '95033');
54  INSERT INTO student VALUES('105', '王芳', '女', '1975-02-10', '95031');
55  INSERT INTO student VALUES('106', '陆军', '男', '1974-06-03', '95031');
56  INSERT INTO student VALUES('107', '王尼玛', '男', '1976-02-20', '95033');

```

```

57 INSERT INTO student VALUES('108', '张全蛋', '男', '1975-02-10', '95031');
58 INSERT INTO student VALUES('109', '赵铁柱', '男', '1974-06-03', '95031');
59
60 -- 添加教师表数据
61 INSERT INTO teacher VALUES('804', '李诚', '男', '1958-12-02', '副教授', '计算机系');
62 INSERT INTO teacher VALUES('856', '张旭', '男', '1969-03-12', '讲师', '电子工程系');
63 INSERT INTO teacher VALUES('825', '王萍', '女', '1972-05-05', '助教', '计算机系');
64 INSERT INTO teacher VALUES('831', '刘冰', '女', '1977-08-14', '助教', '电子工程系');
65
66 -- 添加课程表数据
67 INSERT INTO course VALUES('3-105', '计算机导论', '825');
68 INSERT INTO course VALUES('3-245', '操作系统', '804');
69 INSERT INTO course VALUES('6-166', '数字电路', '856');
70 INSERT INTO course VALUES('9-888', '高等数学', '831');
71
72 -- 添加添加成绩表数据
73 INSERT INTO score VALUES('103', '3-105', '92');
74 INSERT INTO score VALUES('103', '3-245', '86');
75 INSERT INTO score VALUES('103', '6-166', '85');
76 INSERT INTO score VALUES('105', '3-105', '88');
77 INSERT INTO score VALUES('105', '3-245', '75');
78 INSERT INTO score VALUES('105', '6-166', '79');
79 INSERT INTO score VALUES('109', '3-105', '76');
80 INSERT INTO score VALUES('109', '3-245', '68');
81 INSERT INTO score VALUES('109', '6-166', '81');
82
83 -- 查看表结构
84 SELECT * FROM course;
85 SELECT * FROM score;
86 SELECT * FROM student;
87 SELECT * FROM teacher;

```

## 1到10

```

1 -- 查询 student 表的所有行
2 SELECT * FROM student;
3
4 -- 查询 student 表中的 name、sex 和 class 字段的所有行
5 SELECT name, sex, class FROM student;
6
7 -- 查询 teacher 表中不重复的 department 列
8 -- department: 去重查询
9 SELECT DISTINCT department FROM teacher;
10
11 -- 查询 score 表中成绩在60-80之间的所有行（区间查询和运算符查询）
12 -- BETWEEN xx AND xx: 查询区间，AND 表示 "并且"
13 SELECT * FROM score WHERE degree BETWEEN 60 AND 80;

```

```

14 SELECT * FROM score WHERE degree > 60 AND degree < 80;
15
16 -- 查询 score 表中成绩为 85, 86 或 88 的行
17 -- IN: 查询规定中的多个值 (单个值可以用 = )
18 SELECT * FROM score WHERE degree IN (85, 86, 88);
19
20 -- 查询 student 表中 '95031' 班或性别为 '女' 的所有行
21 -- or: 表示或者关系
22 SELECT * FROM student WHERE class = '95031' or sex = '女';
23
24 -- 以 class 降序的方式查询 student 表的所有行
25 -- DESC: 降序, 从高到低
26 -- ASC (默认): 升序, 从低到高
27 SELECT * FROM student ORDER BY class DESC;
28 SELECT * FROM student ORDER BY class ASC;
29
30 -- 以 c_no 升序、degree 降序查询 score 表的所有行
31 SELECT * FROM score ORDER BY c_no ASC, degree DESC;
32
33 -- 查询 "95031" 班的学生人数
34 -- COUNT: 统计
35 SELECT COUNT(*) FROM student WHERE class = '95031';
36
37 -- 查询 score 表中的最高分的学生学号和课程编号 (子查询或排序查询) 。
38 -- (SELECT MAX(degree) FROM score): 子查询, 算出最高分
39 SELECT s_no, c_no FROM score WHERE degree = (SELECT MAX(degree) FROM
score);
40
41 -- 排序查询
42 -- LIMIT r, n: 表示从第r行开始, 查询n条数据
43 SELECT s_no, c_no, degree FROM score ORDER BY degree DESC LIMIT 0, 1;

```

## 分组计算平均成绩

查询每门课的平均成绩

```

1 -- AVG: 平均值
2 SELECT AVG(degree) FROM score WHERE c_no = '3-105';
3 SELECT AVG(degree) FROM score WHERE c_no = '3-245';
4 SELECT AVG(degree) FROM score WHERE c_no = '6-166';
5
6 -- GROUP BY: 分组查询
7 SELECT c_no, AVG(degree) FROM score GROUP BY c_no;

```

## 分组条件与模糊查询

查询score表中至少有2名学生选修, 并以3开头的课程的平均分数

```

1 SELECT * FROM score;
2 -- c_no 课程编号
3 +-----+-----+-----+

```

	s_no	c_no	degree
4			
5			
6	103	3-105	92
7	103	3-245	86
8	103	6-166	85
9	105	3-105	88
10	105	3-245	75
11	105	6-166	79
12	109	3-105	76
13	109	3-245	68
14	109	6-166	81
15			

分析表发现，至少有 2 名学生选修的课程是 3-105、3-245、6-166，以 3 开头的课程是 3-105、3-245。也就是说，我们要查询所有 3-105 和 3-245 的 degree 平均分。

```

1  -- 首先把 c_no, AVG(degree) 通过分组查询出来
2  SELECT c_no, AVG(degree) FROM score GROUP BY c_no
3
4  +-----+-----+-----+
5  | c_no | AVG(degree) |
6  +-----+-----+-----+
7  | 3-105 | 85.3333 |
8  | 3-245 | 76.3333 |
9  | 6-166 | 81.6667 |
10 +-----+-----+-----+
11
12 -- 再查询出至少有 2 名学生选修的课程
13 -- HAVING: 表示持有
14 HAVING COUNT(c_no) >= 2
15
16 -- 并且是以 3 开头的课程
17 -- LIKE 表示模糊查询, "%" 是一个通配符, 匹配 "3" 后面的任意字符。
18 AND c_no LIKE '3%';
19
20 -- 把前面的SQL语句拼接起来,
21 -- 后面加上一个 COUNT(*), 表示将每个分组的个数也查询出来。
22 SELECT c_no, AVG(degree), COUNT(*) FROM score GROUP BY c_no
23 HAVING COUNT(c_no) >= 2 AND c_no LIKE '3%';
24
25 +-----+-----+-----+
26 | c_no | AVG(degree) | COUNT(*) |
27 +-----+-----+-----+
28 | 3-105 | 85.3333 | 3 |
29 | 3-245 | 76.3333 | 3 |
30 +-----+-----+-----+

```

## 多表查询 - 1

查询所有学生的name，以及该学生在score表中对应的c\_no和degree

```

1  SELECT no, name FROM student;
2  +-----+-----+

```

```

3 | no | name |
4 +-----+
5 | 101 | 曾华 |
6 | 102 | 匡明 |
7 | 103 | 王丽 |
8 | 104 | 李军 |
9 | 105 | 王芳 |
10 | 106 | 陆军 |
11 | 107 | 王尼玛 |
12 | 108 | 张全蛋 |
13 | 109 | 赵铁柱 |
14 +-----+
15
16 SELECT s_no, c_no, degree FROM score;
17 +-----+-----+-----+
18 | s_no | c_no | degree |
19 +-----+-----+-----+
20 | 103 | 3-105 | 92 |
21 | 103 | 3-245 | 86 |
22 | 103 | 6-166 | 85 |
23 | 105 | 3-105 | 88 |
24 | 105 | 3-245 | 75 |
25 | 105 | 6-166 | 79 |
26 | 109 | 3-105 | 76 |
27 | 109 | 3-245 | 68 |
28 | 109 | 6-166 | 81 |
29 +-----+-----+-----+

```

通过分析可以发现，只要把 score 表中的 s\_no 字段值替换成 student 表中对应的 name 字段值就可以了，如何做呢？

```

1 -- FROM...: 表示从 student, score 表中查询
2 -- WHERE 的条件表示为，只有在 student.no 和 score.s_no 相等时才显示出来。
3 SELECT name, c_no, degree FROM student, score
4 WHERE student.no = score.s_no;
5 +-----+-----+-----+
6 | name | c_no | degree |
7 +-----+-----+-----+
8 | 王丽 | 3-105 | 92 |
9 | 王丽 | 3-245 | 86 |
10 | 王丽 | 6-166 | 85 |
11 | 王芳 | 3-105 | 88 |
12 | 王芳 | 3-245 | 75 |
13 | 王芳 | 6-166 | 79 |
14 | 赵铁柱 | 3-105 | 76 |
15 | 赵铁柱 | 3-245 | 68 |
16 | 赵铁柱 | 6-166 | 81 |
17 +-----+-----+-----+

```

## 多表查询 - 2

查询所有学生的no，课程名称（course表中的name）和成绩（score表中的degree）列

只有 score 关联学生的 no，因此只要查询 score 表，就能找出所有和学生相关的 no 和 degree：

```
1 SELECT s_no, c_no, degree FROM score;
2 +-----+-----+-----+
3 | s_no | c_no | degree |
4 +-----+-----+-----+
5 | 103 | 3-105 | 92 |
6 | 103 | 3-245 | 86 |
7 | 103 | 6-166 | 85 |
8 | 105 | 3-105 | 88 |
9 | 105 | 3-245 | 75 |
10 | 105 | 6-166 | 79 |
11 | 109 | 3-105 | 76 |
12 | 109 | 3-245 | 68 |
13 | 109 | 6-166 | 81 |
14 +-----+-----+-----+
```

然后查询 course 表：

```
1 +-----+-----+
2 | no | name |
3 +-----+-----+
4 | 3-105 | 计算机导论 |
5 | 3-245 | 操作系统 |
6 | 6-166 | 数字电路 |
7 | 9-888 | 高等数学 |
8 +-----+-----+
```

只要把 score 表中的 c\_no 替换成 course 表中对应的 name 字段值就可以了。

```
1 -- 增加一个查询字段 name，分别从 score、course 这两个表中查询。
2 -- as 表示取一个该字段的别名。
3 SELECT s_no, name as c_name, degree FROM score, course
4 WHERE score.c_no = course.no;
5 +-----+-----+-----+
6 | s_no | c_name | degree |
7 +-----+-----+-----+
8 | 103 | 计算机导论 | 92 |
9 | 105 | 计算机导论 | 88 |
10 | 109 | 计算机导论 | 76 |
11 | 103 | 操作系统 | 86 |
12 | 105 | 操作系统 | 75 |
13 | 109 | 操作系统 | 68 |
14 | 103 | 数字电路 | 85 |
15 | 105 | 数字电路 | 79 |
16 | 109 | 数字电路 | 81 |
17 +-----+-----+-----+
```

## 三表关联查询

查询所有学生的name、课程名（course表中的name）和degree。

只有 `score` 表中关联学生的学号和课堂号，我们只要围绕着 `score` 这张表查询就好了。

```
1 SELECT * FROM score;
2 +-----+-----+-----+
3 | s_no | c_no | degree |
4 +-----+-----+-----+
5 | 103 | 3-105 | 92 |
6 | 103 | 3-245 | 86 |
7 | 103 | 6-166 | 85 |
8 | 105 | 3-105 | 88 |
9 | 105 | 3-245 | 75 |
10 | 105 | 6-166 | 79 |
11 | 109 | 3-105 | 76 |
12 | 109 | 3-245 | 68 |
13 | 109 | 6-166 | 81 |
14 +-----+-----+-----+
```

只要把 `s_no` 和 `c_no` 替换成 `student` 和 `srouse` 表中对应的 `name` 字段值就好了。

首先把 `s_no` 替换成 `student` 表中的 `name` 字段：

```
1 SELECT name, c_no, degree FROM student, score WHERE student.no =
   score.s_no;
2 +-----+-----+-----+
3 | name      | c_no | degree |
4 +-----+-----+-----+
5 | 王丽      | 3-105 | 92 |
6 | 王丽      | 3-245 | 86 |
7 | 王丽      | 6-166 | 85 |
8 | 王芳      | 3-105 | 88 |
9 | 王芳      | 3-245 | 75 |
10 | 王芳      | 6-166 | 79 |
11 | 赵铁柱    | 3-105 | 76 |
12 | 赵铁柱    | 3-245 | 68 |
13 | 赵铁柱    | 6-166 | 81 |
14 +-----+-----+-----+
```

再把 `c_no` 替换成 `course` 表中的 `name` 字段：

```
1 -- 课程表
2 SELECT no, name FROM course;
3 +-----+-----+
4 | no      | name      |
5 +-----+-----+
6 | 3-105 | 计算机导论 |
7 | 3-245 | 操作系统   |
8 | 6-166 | 数字电路   |
9 | 9-888 | 高等数学   |
```



```

10 | +-----+-----+
11 |
12 | -- 由于字段名存在重复, 使用 "表名.字段名 as 别名" 代替。
13 | SELECT student.name as s_name, course.name as c_name, degree
14 | FROM student, score, course
15 | WHERE student.NO = score.s_no
16 | AND score.c_no = course.no;

```

## 子查询加分组求平均分

### 查询95031班学生每门课的平均成绩

在 `score` 表中根据 `student` 表的学生编号筛选出学生的课堂号和成绩:

```

1 | -- IN (...): 将筛选出的学生号当做 s_no 的条件查询
2 | SELECT s_no, c_no, degree FROM score
3 | WHERE s_no IN (SELECT no FROM student WHERE class = '95031');
4 | +-----+-----+-----+
5 | | s_no | c_no | degree |
6 | +-----+-----+-----+
7 | | 105 | 3-105 | 88 |
8 | | 105 | 3-245 | 75 |
9 | | 105 | 6-166 | 79 |
10 | | 109 | 3-105 | 76 |
11 | | 109 | 3-245 | 68 |
12 | | 109 | 6-166 | 81 |
13 | +-----+-----+-----+

```

这时只要将 `c_no` 分组一下就能得出 95031 班学生每门课的平均成绩:

```

1 | SELECT c_no, AVG(degree) FROM score
2 | WHERE s_no IN (SELECT no FROM student WHERE class = '95031')
3 | GROUP BY c_no;
4 | +-----+-----+
5 | | c_no | AVG(degree) |
6 | +-----+-----+
7 | | 3-105 | 82.0000 |
8 | | 3-245 | 71.5000 |
9 | | 6-166 | 80.0000 |
10 | +-----+-----+

```

### 高能部分

```

1 | -- !!!如果想把c_no替换成course表中的name
2 | -- course表:
3 | mysql> select * from course;
4 | +-----+-----+-----+
5 | | no | name | t_no |
6 | +-----+-----+-----+
7 | | 3-105 | 计算机导论 | 825 |
8 | | 3-245 | 操作系统 | 804 |

```

```

9 | 6-166 | 数字电路 | 856 |
10 | 9-888 | 高等数学 | 831 |
11 +-----+-----+-----+
12
13 -- 语句:
14 select name,AVG(degree)
15 from course,score
16 WHERE s_no IN (SELECT no FROM student WHERE class = '95031') and c_no=no
17 GROUP BY c_no; -- name也行
18
19 -- c_no = no 是精髓

```

## 子查询 - 1

查询在3-105课程中，所有成绩高于109号同学的记录

首先筛选出课堂号为 3-105，在找出所有成绩高于 109 号同学的的行。

```

1 | select s_no,c_no,degree from score
2 | where c_no='3-105'
3 | and degree>(select degree from score
4 | where c_no='3-105' and s_no='109');

```

## 子查询 - 2

查询所有成绩高于 109 号同学的 3-105 课程成绩记录。

```

1 | -- 不限制课程号，只要成绩大于109号同学的3-105课程成绩就可以。
2 | SELECT * FROM score
3 | WHERE degree > (SELECT degree FROM score WHERE s_no = '109' AND c_no = '3-105');

```

## YEAR函数与带IN关键字查询

查询所有和 101、108 号学生同年出生的 no、name、birthday 列。

```

1 | -- YEAR(..): 取出日期中的年份
2 | SELECT no, name, birthday FROM student
3 | WHERE YEAR(birthday) IN (SELECT YEAR(birthday) FROM student WHERE no IN (101, 108));

```

## 多层嵌套子查询

查询 '张旭' 教师任课的学生成绩表。

首先找到教师编号：

```

1 | SELECT NO FROM teacher WHERE NAME = '张旭'

```

通过 source 表找到该教师课程号：

```
1 SELECT NO FROM course WHERE t_no = ( SELECT NO FROM teacher WHERE NAME = '张旭' );
```

通过筛选出的课程号查询成绩表：

```
1 SELECT * FROM score WHERE c_no = (  
2     SELECT no FROM course WHERE t_no = (  
3         SELECT no FROM teacher WHERE NAME = '张旭'  
4     )  
5 );
```

## 多表查询

查询某选修课程多于5个同学的教师姓名。

首先在 teacher 表中，根据 no 字段来判断该教师的同一门课程是否有至少5名学员选修：

```
1  -- 查询 teacher 表  
2  SELECT no, name FROM teacher;  
3  
4  +-----+-----+  
5  | no   | name   |  
6  +-----+-----+  
7  | 804  | 李诚   |  
8  | 825  | 王萍   |  
9  | 831  | 刘冰   |  
10 | 856  | 张旭   |  
11 +-----+-----+  
12  
13 SELECT name FROM teacher WHERE no IN (  
14     -- 在这里找到对应的条件  
15 );
```

查看和教师编号有有关的表的信息：

```
1 SELECT * FROM course;  
2  -- t_no: 教师编号  
3  
4  +-----+-----+-----+  
5  | no   | name           | t_no |  
6  +-----+-----+-----+  
7  | 3-105 | 计算机导论     | 825  |  
8  | 3-245 | 操作系统       | 804  |  
9  | 6-166 | 数字电路       | 856  |  
10 | 9-888 | 高等数学       | 831  |  
11 +-----+-----+-----+
```

我们已经找到和教师编号有关的字段就在 `course` 表中，但是还无法知道哪门课程至少有5名学生选修，所以还需要根据 `score` 表来查询：

```
1  -- 在此之前向 score 插入一些数据，以便丰富查询条件。
2  INSERT INTO score VALUES ('101', '3-105', '90');
3  INSERT INTO score VALUES ('102', '3-105', '91');
4  INSERT INTO score VALUES ('104', '3-105', '89');
5
6  -- 查询 score 表
7  SELECT * FROM score;
8
9  +-----+-----+-----+
10 | s_no | c_no | degree |
11 +-----+-----+-----+
12 | 101 | 3-105 | 90 |
13 | 102 | 3-105 | 91 |
14 | 103 | 3-105 | 92 |
15 | 103 | 3-245 | 86 |
16 | 103 | 6-166 | 85 |
17 | 104 | 3-105 | 89 |
18 | 105 | 3-105 | 88 |
19 | 105 | 3-245 | 75 |
20 | 105 | 6-166 | 79 |
21 | 109 | 3-105 | 76 |
22 | 109 | 3-245 | 68 |
23 | 109 | 6-166 | 81 |
24 +-----+-----+-----+
25
26 -- 在 score 表中将 c_no 作为分组，并且限制 c_no 持有至少 5 条数据。
27 SELECT c_no FROM score GROUP BY c_no HAVING COUNT(*) > 5;
28
29 +-----+
30 | c_no |
31 +-----+
32 | 3-105 |
```

根据筛选出来的课程号，找出在某课程中，拥有至少5名学员的教师编号：

```
1  SELECT t_no FROM course WHERE no IN (
2      SELECT c_no FROM score GROUP BY c_no HAVING COUNT(*) > 5
3  );
4
5  +-----+
6  | t_no |
7  +-----+
8  | 825 |
9  +-----+
```

在 `teacher` 表中，根据筛选出来的教师编号找到教师姓名：

```

1 SELECT name FROM teacher WHERE no IN (
2     -- 最终条件
3     SELECT t_no FROM course WHERE no IN (
4         SELECT c_no FROM score GROUP BY c_no HAVING COUNT(*) > 5
5     )
6 );

```

## 子查询 - 3

查询“计算机系”课程的成绩表。

思路是，先找出 `course` 表中所有 计算机系 课程的编号，然后根据这个编号查询 `score` 表。

```

1  -- 通过 teacher 表查询所有 `计算机系` 的教师编号
2  SELECT no, name, department FROM teacher WHERE department = '计算机系'
3  +-----+-----+-----+
4  | no   | name  | department |
5  +-----+-----+-----+
6  | 804  | 李诚  | 计算机系   |
7  | 825  | 王萍  | 计算机系   |
8  +-----+-----+-----+
9
10 -- 通过 course 表查询该教师的课程编号
11 SELECT no FROM course WHERE t_no IN (
12     SELECT no FROM teacher WHERE department = '计算机系'
13 );
14 +-----+
15 | no   |
16 +-----+
17 | 3-245 |
18 | 3-105 |
19 +-----+
20
21 -- 根据筛选出来的课程号查询成绩表
22 SELECT * FROM score WHERE c_no IN (
23     SELECT no FROM course WHERE t_no IN (
24         SELECT no FROM teacher WHERE department = '计算机系'
25     )
26 );
27 +-----+-----+-----+
28 | s_no | c_no  | degree |
29 +-----+-----+-----+
30 | 103  | 3-245 | 86     |
31 | 105  | 3-245 | 75     |
32 | 109  | 3-245 | 68     |
33 | 101  | 3-105 | 90     |
34 | 102  | 3-105 | 91     |
35 | 103  | 3-105 | 92     |
36 | 104  | 3-105 | 89     |
37 | 105  | 3-105 | 88     |
38 | 109  | 3-105 | 76     |

```

## UNION 和 NOTIN 的使用

查询 计算机系 与 电子工程系 中的不同职称的教师。

```

1  -- NOT: 代表逻辑非
2  SELECT * FROM teacher WHERE department = '计算机系' AND profession NOT IN (
3      SELECT profession FROM teacher WHERE department = '电子工程系'
4  )
5  -- 合并两个集
6  UNION
7  SELECT * FROM teacher WHERE department = '电子工程系' AND profession NOT IN (
8      SELECT profession FROM teacher WHERE department = '计算机系'
9  );

```

## ANY 表示至少一个 - DESC ( 降序 )

查询课程 3-105 且成绩 至少 高于 3-245 的 score 表。

```

1  SELECT * FROM score WHERE c_no = '3-105';
2  +-----+-----+-----+
3  | s_no | c_no | degree |
4  +-----+-----+-----+
5  | 101 | 3-105 | 90 |
6  | 102 | 3-105 | 91 |
7  | 103 | 3-105 | 92 |
8  | 104 | 3-105 | 89 |
9  | 105 | 3-105 | 88 |
10 | 109 | 3-105 | 76 |
11 +-----+-----+-----+
12
13 SELECT * FROM score WHERE c_no = '3-245';
14 +-----+-----+-----+
15 | s_no | c_no | degree |
16 +-----+-----+-----+
17 | 103 | 3-245 | 86 |
18 | 105 | 3-245 | 75 |
19 | 109 | 3-245 | 68 |
20 +-----+-----+-----+
21
22 -- ANY: 符合SQL语句中的任意条件。
23 -- 也就是说, 在 3-105 成绩中, 只要有一个大于从 3-245 筛选出来的任意行就符合条件,
24 -- 最后根据降序查询结果。
25 SELECT * FROM score WHERE c_no = '3-105' AND degree > ANY(
26     SELECT degree FROM score WHERE c_no = '3-245'
27 ) ORDER BY degree DESC;
28 +-----+-----+-----+
29 | s_no | c_no | degree |
30 +-----+-----+-----+
31 | 103 | 3-105 | 92 |

```

32		102		3-105		91	
33		101		3-105		90	
34		104		3-105		89	
35		105		3-105		88	
36		109		3-105		76	
37		+-----+-----+-----+					

## 表示所有的 ALL

查询课程 3-105 且成绩高于 3-245 的 score 表。

```

1  -- 只需对上一道题稍作修改。
2  -- ALL：符合SQL语句中的所有条件。
3  -- 也就是说，在 3-105 每一行成绩中，都要大于从 3-245 筛选出来全部行才算符合条件。
4  SELECT * FROM score WHERE c_no = '3-105' AND degree > ALL(
5      SELECT degree FROM score WHERE c_no = '3-245'
6  );
7  +-----+-----+-----+
8  | s_no | c_no | degree |
9  +-----+-----+-----+
10 | 101 | 3-105 | 90 |
11 | 102 | 3-105 | 91 |
12 | 103 | 3-105 | 92 |
13 | 104 | 3-105 | 89 |
14 | 105 | 3-105 | 88 |
15 +-----+-----+-----+

```

## 复制表的数据作为条件查询

查询某课程成绩比该课程平均成绩低的 score 表。

```

1  -- 查询平均分
2  SELECT c_no, AVG(degree) FROM score GROUP BY c_no;
3  +-----+-----+
4  | c_no | AVG(degree) |
5  +-----+-----+
6  | 3-105 | 87.6667 |
7  | 3-245 | 76.3333 |
8  | 6-166 | 81.6667 |
9  +-----+-----+
10
11 -- 查询 score 表
12 SELECT degree FROM score;
13 +-----+
14 | degree |
15 +-----+
16 | 90 |
17 | 91 |
18 | 92 |
19 | 86 |
20 | 85 |

```

```

21 |      89 |
22 |      88 |
23 |      75 |
24 |      79 |
25 |      76 |
26 |      68 |
27 |      81 |
28 +-----+
29
30 -- 将表 b 作用于表 a 中查询数据
31 -- score a (b): 将表声明为 a (b),
32 -- 如此就能用 a.c_no = b.c_no 作为条件执行查询了。
33 SELECT * FROM score a WHERE degree < (
34     (SELECT AVG(degree) FROM score b WHERE a.c_no = b.c_no)
35 );
36 +-----+-----+-----+
37 | s_no | c_no | degree |
38 +-----+-----+-----+
39 | 105 | 3-245 | 75 |
40 | 105 | 6-166 | 79 |
41 | 109 | 3-105 | 76 |
42 | 109 | 3-245 | 68 |
43 | 109 | 6-166 | 81 |
44 +-----+-----+-----+

```

## 子查询 - 4

查询所有任课 ( 在 `course` 表里有课程 ) 教师的 `name` 和 `department` 。

```

1 SELECT name, department FROM teacher WHERE no IN (SELECT t_no FROM course);
2 +-----+-----+
3 | name | department |
4 +-----+-----+
5 | 李诚 | 计算机系 |
6 | 王萍 | 计算机系 |
7 | 刘冰 | 电子工程系 |
8 | 张旭 | 电子工程系 |
9 +-----+-----+

```

## 条件加组筛选

查询 `student` 表中至少有 2 名男生的 `class` 。

```

1 -- 查看学生表信息
2 SELECT * FROM student;
3 +-----+-----+-----+-----+-----+
4 | no | name | sex | birthday | class |
5 +-----+-----+-----+-----+-----+
6 | 101 | 曾华 | 男 | 1977-09-01 | 95033 |
7 | 102 | 匡明 | 男 | 1975-10-02 | 95031 |
8 | 103 | 王丽 | 女 | 1976-01-23 | 95033 |

```



```

9 | 104 | 李军 | 男 | 1976-02-20 | 95033 |
10 | 105 | 王芳 | 女 | 1975-02-10 | 95031 |
11 | 106 | 陆军 | 男 | 1974-06-03 | 95031 |
12 | 107 | 王尼玛 | 男 | 1976-02-20 | 95033 |
13 | 108 | 张全蛋 | 男 | 1975-02-10 | 95031 |
14 | 109 | 赵铁柱 | 男 | 1974-06-03 | 95031 |
15 | 110 | 张飞 | 男 | 1974-06-03 | 95038 |
16 +-----+-----+-----+-----+
17
18 -- 只查询性别为男, 然后按 class 分组, 并限制 class 行大于 1。
19 SELECT class FROM student WHERE sex = '男' GROUP BY class HAVING COUNT(*) >
20 1;
21 +-----+
22 | class |
23 +-----+
24 | 95033 |
25 | 95031 |

```

## NOTLIKE 模糊查询取反

查询 `student` 表中不姓 "王" 的同学记录。

```

1 -- NOT: 取反
2 -- LIKE: 模糊查询
3 mysql> SELECT * FROM student WHERE name NOT LIKE '王%';
4 +-----+-----+-----+-----+
5 | no | name | sex | birthday | class |
6 +-----+-----+-----+-----+
7 | 101 | 曾华 | 男 | 1977-09-01 | 95033 |
8 | 102 | 匡明 | 男 | 1975-10-02 | 95031 |
9 | 104 | 李军 | 男 | 1976-02-20 | 95033 |
10 | 106 | 陆军 | 男 | 1974-06-03 | 95031 |
11 | 108 | 张全蛋 | 男 | 1975-02-10 | 95031 |
12 | 109 | 赵铁柱 | 男 | 1974-06-03 | 95031 |
13 | 110 | 张飞 | 男 | 1974-06-03 | 95038 |
14 +-----+-----+-----+-----+

```

## YEAR 与 NOW 函数

查询 `student` 表中每个学生的姓名和年龄。

```

1 -- 使用函数 YEAR(NOW()) 计算出当前年份, 减去出生年份后得出年龄。
2 SELECT name, YEAR(NOW()) - YEAR(birthday) as age FROM student;
3 +-----+-----+
4 | name | age |
5 +-----+-----+
6 | 曾华 | 42 |
7 | 匡明 | 44 |
8 | 王丽 | 43 |
9 | 李军 | 43 |

```

10		王芳		44	
11		陆军		45	
12		王尼玛		43	
13		张全蛋		44	
14		赵铁柱		45	
15		张飞		45	
16		+-----+			

MAX 与 MIN 函数

查询 student 表中最大和最小的 birthday 值。

1		SELECT MAX(birthday), MIN(birthday) FROM student;			
2		+-----+			
3		MAX(birthday)		MIN(birthday)	
4		+-----+			
5		1977-09-01		1974-06-03	
6		+-----+			

多段排序

以 class 和 birthday 从大到小的顺序查询 student 表。

1		SELECT * FROM student ORDER BY class DESC, birthday;				
2		+-----+				
3		no		name		sex   birthday   class
4		+-----+				
5		110		张飞		男   1974-06-03   95038
6		103		王丽		女   1976-01-23   95033
7		104		李军		男   1976-02-20   95033
8		107		王尼玛		男   1976-02-20   95033
9		101		曾华		男   1977-09-01   95033
10		106		陆军		男   1974-06-03   95031
11		109		赵铁柱		男   1974-06-03   95031
12		105		王芳		女   1975-02-10   95031
13		108		张全蛋		男   1975-02-10   95031
14		102		匡明		男   1975-10-02   95031
15		+-----+				

子查询 - 5

查询 "男" 教师及其所上的课程。

```

1 SELECT * FROM course WHERE t_no in (SELECT no FROM teacher WHERE sex =
  '男');
2 +-----+-----+-----+
3 | no      | name          | t_no |
4 +-----+-----+-----+
5 | 3-245   | 操作系统      | 804   |
6 | 6-166   | 数字电路      | 856   |
7 +-----+-----+-----+

```

## MAX 函数与子查询

查询最高分同学的 `score` 表。

```

1  -- 找出最高成绩（该查询只能有一个结果）
2  SELECT MAX(degree) FROM score;
3
4  -- 根据上面的条件筛选出所有最高成绩表，
5  -- 该查询可能有多个结果，假设 degree 值多次符合条件。
6  SELECT * FROM score WHERE degree = (SELECT MAX(degree) FROM score);
7
8  +-----+-----+-----+
9  | s_no | c_no | degree |
10 +-----+-----+-----+
11 | 103  | 3-105 | 92     |
12 +-----+-----+-----+

```

## 子查询 - 6

查询和 "李军" 同性别的所有同学 `name` 。

```

1  -- 首先将李军的性别作为条件取出来
2  SELECT sex FROM student WHERE name = '李军';
3
4  +-----+
5  | sex |
6  +-----+
7  | 男  |
8  +-----+
9
10 -- 根据性别查询 name 和 sex
11 SELECT name, sex FROM student WHERE sex = (
12     SELECT sex FROM student WHERE name = '李军'
13 );
14
15 +-----+-----+
16 | name      | sex |
17 +-----+-----+
18 | 曾华      | 男  |
19 | 匡明      | 男  |
20 | 李军      | 男  |
21 | 陆军      | 男  |
22 | 王尼玛    | 男  |
23 | 张全蛋    | 男  |
24 | 赵铁柱    | 男  |

```

23	张飞	男	
24	+-----+-----+		

## 子查询 - 7

查询和 "李军" 同性别且同班的同学 `name` 。

```

1  SELECT name, sex, class FROM student WHERE sex = (
2      SELECT sex FROM student WHERE name = '李军'
3  ) AND class = (
4      SELECT class FROM student WHERE name = '李军'
5  );
6  +-----+-----+-----+
7  | name      | sex | class |
8  +-----+-----+-----+
9  | 曾华      | 男  | 95033 |
10 | 李军      | 男  | 95033 |
11 | 王尼玛    | 男  | 95033 |
12 +-----+-----+-----+

```

## 子查询 - 8

查询所有选修 "计算机导论" 课程的 "男" 同学成绩表。

需要的 "计算机导论" 和性别为 "男" 的编号可以在 `course` 和 `student` 表中找到。

```

1  SELECT * FROM score WHERE c_no = (
2      SELECT no FROM course WHERE name = '计算机导论'
3  ) AND s_no IN (
4      SELECT no FROM student WHERE sex = '男'
5  );
6  +-----+-----+-----+
7  | s_no | c_no | degree |
8  +-----+-----+-----+
9  | 101  | 3-105 | 90 |
10 | 102  | 3-105 | 91 |
11 | 104  | 3-105 | 89 |
12 | 109  | 3-105 | 76 |
13 +-----+-----+-----+

```

## 按等级查询

建立一个 `grade` 表代表学生的成绩等级，并插入数据：

```

1  CREATE TABLE grade (
2      low INT(3),
3      upp INT(3),
4      grade CHAR(1)
5  );
6
7  INSERT INTO grade VALUES (90, 100, 'A');

```

```

8  INSERT INTO grade VALUES (80, 89, 'B');
9  INSERT INTO grade VALUES (70, 79, 'C');
10 INSERT INTO grade VALUES (60, 69, 'D');
11 INSERT INTO grade VALUES (0, 59, 'E');
12
13 SELECT * FROM grade;
14 +-----+-----+-----+
15 | low  | upp  | grade |
16 +-----+-----+-----+
17 | 90   | 100  | A     |
18 | 80   | 89   | B     |
19 | 70   | 79   | C     |
20 | 60   | 69   | D     |
21 | 0    | 59   | E     |
22 +-----+-----+-----+

```

查询所有学生的 `s_no`、`c_no` 和 `grade` 列。

思路是，使用区间 ( `BETWEEN` ) 查询，判断学生的成绩 ( `degree` ) 在 `grade` 表的 `low` 和 `upp` 之间。

```

1  SELECT s_no, c_no, grade FROM score, grade
2  WHERE degree BETWEEN low AND upp;
3  +-----+-----+-----+
4  | s_no | c_no  | grade |
5  +-----+-----+-----+
6  | 101  | 3-105 | A     |
7  | 102  | 3-105 | A     |
8  | 103  | 3-105 | A     |
9  | 103  | 3-245 | B     |
10 | 103  | 6-166 | B     |
11 | 104  | 3-105 | B     |
12 | 105  | 3-105 | B     |
13 | 105  | 3-245 | C     |
14 | 105  | 6-166 | C     |
15 | 109  | 3-105 | C     |
16 | 109  | 3-245 | D     |
17 | 109  | 6-166 | B     |
18 +-----+-----+-----+

```

## 连接查询

准备用于测试连接查询的数据：

```

1  CREATE DATABASE testJoin;
2
3  CREATE TABLE person (
4      id INT,
5      name VARCHAR(20),
6      cardId INT
7  );
8
9  CREATE TABLE card (

```

```

10     id INT,
11     name VARCHAR(20)
12 );
13
14 INSERT INTO card VALUES (1, '饭卡'), (2, '建行卡'), (3, '农行卡'), (4, '工商
卡'), (5, '邮政卡');
15 SELECT * FROM card;
16 +-----+-----+
17 | id  | name      |
18 +-----+-----+
19 |  1  | 饭卡      |
20 |  2  | 建行卡    |
21 |  3  | 农行卡    |
22 |  4  | 工商卡    |
23 |  5  | 邮政卡    |
24 +-----+-----+
25
26 INSERT INTO person VALUES (1, '张三', 1), (2, '李四', 3), (3, '王五', 6);
27 SELECT * FROM person;
28 +-----+-----+-----+
29 | id  | name  | cardId |
30 +-----+-----+-----+
31 |  1  | 张三  |    1   |
32 |  2  | 李四  |    3   |
33 |  3  | 王五  |    6   |
34 +-----+-----+-----+

```

分析两张表发现，`person` 表并没有为 `cardId` 字段设置一个在 `card` 表中对应的 `id` 外键。如果设置的话，`person` 中 `cardId` 字段值为 6 的行就插不进去，因为该 `cardId` 值在 `card` 表中并没有。

## 内连接

要查询这两张表中有关系的数据，可以使用 `INNER JOIN`（内连接）将它们连接在一起。

```

1  -- INNER JOIN：表示为内连接，将两张表拼接在一起。
2  -- on：表示要执行某个条件。
3  SELECT * FROM person INNER JOIN card on person.cardId = card.id;
4  +-----+-----+-----+-----+-----+
5  | id  | name  | cardId | id  | name      |
6  +-----+-----+-----+-----+-----+
7  |  1  | 张三  |    1   |  1  | 饭卡      |
8  |  2  | 李四  |    3   |  3  | 农行卡    |
9  +-----+-----+-----+-----+-----+
10
11 -- 将 INNER 关键字省略掉，结果也是一样的。
12 -- SELECT * FROM person JOIN card on person.cardId = card.id;

```

注意：`card` 的整张表被连接到了右边。

## 左外连接

完整显示左边的表 ( `person` )，右边的表如果符合条件就显示，不符合则补 `NULL`。

```
1  -- LEFT JOIN 也叫做 LEFT OUTER JOIN，用这两种方式的查询结果是一样的。
2  SELECT * FROM person LEFT JOIN card on person.cardId = card.id;
3  +-----+-----+-----+-----+-----+
4  | id   | name  | cardId | id   | name  |
5  +-----+-----+-----+-----+-----+
6  | 1    | 张三  | 1      | 1    | 饭卡  |
7  | 2    | 李四  | 3      | 3    | 农行卡 |
8  | 3    | 王五  | 6      | NULL | NULL  |
9  +-----+-----+-----+-----+-----+
```

## 右外链接

完整显示右边的表 ( `card` )，左边的表如果符合条件就显示，不符合则补 `NULL`。

```
1  SELECT * FROM person RIGHT JOIN card on person.cardId = card.id;
2  +-----+-----+-----+-----+-----+
3  | id   | name  | cardId | id   | name  |
4  +-----+-----+-----+-----+-----+
5  | 1    | 张三  | 1      | 1    | 饭卡  |
6  | 2    | 李四  | 3      | 3    | 农行卡 |
7  | NULL | NULL  | NULL   | 2    | 建行卡 |
8  | NULL | NULL  | NULL   | 4    | 工商卡 |
9  | NULL | NULL  | NULL   | 5    | 邮政卡 |
10 +-----+-----+-----+-----+-----+
```

## 全外链接

完整显示两张表的全部数据。

```
1  -- MySQL 不支持这种语法的全外连接
2  -- SELECT * FROM person FULL JOIN card on person.cardId = card.id;
3  -- 出现错误：
4  -- ERROR 1054 (42S22): Unknown column 'person.cardId' in 'on clause'
5
6  -- MySQL全连接语法，使用 UNION 将两张表合并在一起。
7  SELECT * FROM person LEFT JOIN card on person.cardId = card.id
8  UNION
9  SELECT * FROM person RIGHT JOIN card on person.cardId = card.id;
10 +-----+-----+-----+-----+-----+
11 | id   | name  | cardId | id   | name  |
12 +-----+-----+-----+-----+-----+
13 | 1    | 张三  | 1      | 1    | 饭卡  |
14 | 2    | 李四  | 3      | 3    | 农行卡 |
15 | 3    | 王五  | 6      | NULL | NULL  |
16 | NULL | NULL  | NULL   | 2    | 建行卡 |
17 | NULL | NULL  | NULL   | 4    | 工商卡 |
18 | NULL | NULL  | NULL   | 5    | 邮政卡 |
19 +-----+-----+-----+-----+-----+
```

# 事务

在 MySQL 中，事务其实是一个最小的不可分割的工作单元。事务能够**保证一个业务的完整性**。

比如我们的银行转账：

```
1  -- a -> -100
2  UPDATE user set money = money - 100 WHERE name = 'a';
3
4  -- b -> +100
5  UPDATE user set money = money + 100 WHERE name = 'b';
```

在实际项目中，假设只有一条 SQL 语句执行成功，而另外一条执行失败了，就会出现数据前后不一致。

因此，在执行多条有关联 SQL 语句时，**事务**可能会要求这些 SQL 语句要么同时执行成功，要么就都执行失败。

## 如何控制事务 COMMIT/ROLLBACK

### 1. 自动提交 (1开0关)

在 MySQL 中，事务的**自动提交**状态默认是开启的。（不能回滚）

```
2. 1  -- 查询事务的自动提交状态
   2  SELECT @@AUTOCOMMIT;
   3  +-----+
   4  | @@AUTOCOMMIT |
   5  +-----+
   6  |              1 |
   7  +-----+
```

**自动提交的作用：**当我们执行一条 SQL 语句的时候，其产生的效果就会立即体现出来，且不能**回滚**。

### 3. 回滚 ROLLBACK

什么是回滚？举个例子：

```
1  CREATE DATABASE bank;
2
3  USE bank;
4
5  CREATE TABLE user (
6      id INT PRIMARY KEY,
7      name VARCHAR(20),
8      money INT
9  );
10
11 INSERT INTO user VALUES (1, 'a', 1000);
12
13 SELECT * FROM user;
```



```

14  +----+-----+-----+
15  | id | name | money |
16  +----+-----+-----+
17  |  1 | a    |  1000 |
18  +----+-----+-----+

```

可以看到，在执行插入语句后数据立刻生效，原因是 MySQL 中的事务自动将它**提交**到了数据库中。那么所谓**回滚**的意思就是，撤销执行过的所有 SQL 语句，使其回滚到**最后一次提交**数据时的状态。

在 MySQL 中使用 `ROLLBACK` 执行回滚：（就是撤销）

```

1  -- 回滚到最后一次提交
2  ROLLBACK;
3
4  SELECT * FROM user;
5  +----+-----+-----+
6  | id | name | money |
7  +----+-----+-----+
8  |  1 | a    |  1000 |
9  +----+-----+-----+

```

由于所有执行过的 SQL 语句都已经被提交过了，所以数据并没有发生回滚。那如何让数据可以发生回滚？

```

1  -- 关闭自动提交
2  SET AUTOCOMMIT = 0;
3
4  -- 查询自动提交状态
5  SELECT @@AUTOCOMMIT;
6  +-----+
7  | @@AUTOCOMMIT |
8  +-----+
9  |              0 |
10 +-----+

```

将自动提交关闭后，测试数据回滚：

```

1  INSERT INTO user VALUES (2, 'b', 1000);
2
3  -- 关闭 AUTOCOMMIT 后，数据的变化是在一张虚拟的临时数据表中展示，
4  -- 发生变化的数据并没有真正插入到数据表中。
5  SELECT * FROM user;
6  +----+-----+-----+
7  | id | name | money |
8  +----+-----+-----+
9  |  1 | a    |  1000 |
10 |  2 | b    |  1000 |
11 +----+-----+-----+
12
13 -- 数据表中的真实数据其实还是：
14 +----+-----+-----+

```

```

15 | id | name | money |
16 +----+-----+-----+
17 | 1 | a | 1000 |
18 +----+-----+-----+
19
20 -- 由于数据还没有真正提交，可以使用回滚
21 ROLLBACK;
22
23 -- 再次查询
24 SELECT * FROM user;
25 +----+-----+-----+
26 | id | name | money |
27 +----+-----+-----+
28 | 1 | a | 1000 |
29 +----+-----+-----+

```

#### 4. COMMIT

那如何将虚拟的数据真正提交到数据库中？使用 `COMMIT`：

```

1  INSERT INTO user VALUES (2, 'b', 1000);
2  -- 手动提交数据（持久性），
3  -- 将数据真正提交到数据库中，执行后不能再回滚提交过的数据。
4  COMMIT;
5
6  -- 提交后测试回滚
7  ROLLBACK;
8
9  -- 再次查询（回滚无效了）
10 SELECT * FROM user;
11 +----+-----+-----+
12 | id | name | money |
13 +----+-----+-----+
14 | 1 | a | 1000 |
15 | 2 | b | 1000 |
16 +----+-----+-----+

```

#### 5. 总结

##### 1. 自动提交

- 查看自动提交状态： `SELECT @@AUTOCOMMIT` ；
- 设置自动提交状态： `SET AUTOCOMMIT = 0` 。

##### 2. 手动提交

`@@AUTOCOMMIT = 0` 时，使用 `COMMIT` 命令提交事务。

##### 3. 事务回滚

`@@AUTOCOMMIT = 0` 时，使用 `ROLLBACK` 命令回滚事务。

#### 6. 实际应用

让我们再回到银行转账项目：

```

1  -- 转账
2  UPDATE user set money = money - 100 WHERE name = 'a';
3
4  -- 到账
5  UPDATE user set money = money + 100 WHERE name = 'b';
6
7  SELECT * FROM user;
8  +----+-----+-----+
9  | id | name | money |
10 +----+-----+-----+
11 |  1 | a    |  900 |
12 |  2 | b    | 1100 |
13 +----+-----+-----+

```

这时假设在转账时发生了意外，就可以使用 `ROLLBACK` 回滚到最后一次提交的状态：

```

1  -- 假设转账发生了意外，需要回滚。
2  ROLLBACK;
3
4  SELECT * FROM user;
5  +----+-----+-----+
6  | id | name | money |
7  +----+-----+-----+
8  |  1 | a    | 1000 |
9  |  2 | b    | 1000 |
10 +----+-----+-----+

```

这时我们又回到了发生意外之前的状态，也就是说，事务给我们提供了一个可以反悔的机会。假设数据没有发生意外，这时可以手动将数据真正提交到数据表中：`COMMIT`。

## 手动开启事务 BEGIN / START TRANSACTION

事务的默认提交被开启 ( `@@AUTOCOMMIT = 1` ) 后，此时就不能使用事务回滚了。但是我们还可以手动开启一个事务处理事件，使其可以发生回滚。

```

1  -- 使用 BEGIN 或者 START TRANSACTION 手动开启一个事务
2  -- 开启了之后，就相当于接下来的操作中AUTOCOMMIT都是关闭的，可以自己手动提交/回滚
3  -- START TRANSACTION;
4  BEGIN;
5  UPDATE user set money = money - 100 WHERE name = 'a';
6  UPDATE user set money = money + 100 WHERE name = 'b';
7
8  -- 由于手动开启的事务没有开启自动提交，
9  -- 此时发生变化的数据仍然是被保存在一张临时表中。
10 SELECT * FROM user;
11 +----+-----+-----+
12 | id | name | money |
13 +----+-----+-----+
14 |  1 | a    |  900 |
15 |  2 | b    | 1100 |
16 +----+-----+-----+

```

```

17
18 -- 测试回滚
19 ROLLBACK;
20
21 SELECT * FROM user;
22 +----+-----+-----+
23 | id | name | money |
24 +----+-----+-----+
25 |  1 | a    |  1000 |
26 |  2 | b    |  1000 |
27 +----+-----+-----+

```

仍然使用 `COMMIT` 提交数据，提交后无法再发生本次事务的回滚。

```

1 BEGIN;
2 UPDATE user set money = money - 100 WHERE name = 'a';
3 UPDATE user set money = money + 100 WHERE name = 'b';
4
5 SELECT * FROM user;
6 +----+-----+-----+
7 | id | name | money |
8 +----+-----+-----+
9 |  1 | a    |   900 |
10 |  2 | b    |  1100 |
11 +----+-----+-----+
12
13 -- 提交数据
14 COMMIT;
15
16 -- 测试回滚（无效，因为表的数据已经被提交）
17 ROLLBACK;

```

## 事务的 ACID 特征与使用

事务的四大特征：

- **A 原子性**：事务是最小的单位，不可以再分割；
- **C 一致性**：要求同一事务中的 SQL 语句，必须保证同时成功或者失败；
- **I 隔离性**：事务1 和 事务2 之间是具有隔离性的；
- **D 持久性**：事务一旦结束（`COMMIT`），就不可以再返回了（`ROLLBACK`）。

### 事务的隔离性

事务的隔离性可分为四种（性能从低到高）：

#### 1. READ UNCOMMITTED（读取未提交）

如果有多个事务，那么任意事务都可以看见其他事务的**未提交数据**。

#### 2. READ COMMITTED（读取已提交）

只能读取到其他事务**已经提交的数据**。

#### 3. REPEATABLE READ（可被重复读）

如果有多个连接都开启了事务，那么事务之间不能共享数据记录，否则只能共享已提交的记录。

#### 4. SERIALIZABLE ( 串行化 )

所有的事务都会按照**固定顺序执行**，执行完一个事务后再继续执行下一个事务的**写入操作**。

查看当前数据库的默认隔离级别：

```
1  -- MySQL 8.x, GLOBAL 表示系统级别，不加表示会话级别。
2  SELECT @@GLOBAL.TRANSACTION_ISOLATION;
3  SELECT @@TRANSACTION_ISOLATION;
4  +-----+
5  | @@GLOBAL.TRANSACTION_ISOLATION |
6  +-----+
7  | REPEATABLE-READ                | -- MySQL的默认隔离级别，可以重复读。
8  +-----+
9
10 -- MySQL 8.0:
11 SELECT @@GLOBAL.TRANSACTION_ISOLATION;
12 -- GLOBAL.不加也行
13 SELECT @@TRANSACTION_ISOLATION;
14
15 -- MySQL 5.x:
16 SELECT @@GLOBAL.TX_ISOLATION;
17 SELECT @@TX_ISOLATION;
```

修改隔离级别：

```
1  -- 设置系统隔离级别，LEVEL 后面表示要设置的隔离级别 (READ UNCOMMITTED)。
2  SET GLOBAL TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
3
4  -- 查询系统隔离级别，发现已经被修改。
5  SELECT @@GLOBAL.TRANSACTION_ISOLATION;
6  +-----+
7  | @@GLOBAL.TRANSACTION_ISOLATION |
8  +-----+
9  | READ-UNCOMMITTED                |
10 +-----+
```

#### 脏读 READ UNCOMMITTED (读取未提交)

测试 READ UNCOMMITTED ( 读取未提交 ) 的隔离性：

```
1  INSERT INTO user VALUES (3, '小明', 1000);
2  INSERT INTO user VALUES (4, '淘宝店', 1000);
3
4  SELECT * FROM user;
5  +---+-----+-----+
6  | id | name    | money |
7  +---+-----+-----+
8  | 1  | a       | 900   |
9  | 2  | b       | 1100  |
10 | 3  | 小明    | 1000  |
11 | 4  | 淘宝店  | 1000  |
```

```

12  +---+-----+-----+
13
14  -- 开启一个事务操作数据
15  -- 假设小明在淘宝店买了一双800块钱的鞋子：
16  START TRANSACTION;
17  UPDATE user SET money = money - 800 WHERE name = '小明';
18  UPDATE user SET money = money + 800 WHERE name = '淘宝店';
19
20  -- 然后淘宝店在另一方查询结果，发现钱已到账。
21  SELECT * FROM user;
22  +---+-----+-----+
23  | id | name      | money |
24  +---+-----+-----+
25  | 1  | a        | 900  |
26  | 2  | b        | 1100 |
27  | 3  | 小明     | 200  |
28  | 4  | 淘宝店   | 1800 |
29  +---+-----+-----+

```

由于小明的转账是在新开启的事务上进行操作，而该操作的结果是可以被其他事务（另一方的淘宝店）看见的，因此淘宝店的查询结果是正确的，淘宝店确认到账。但就在这时，如果小明在它所处的事务上又执行了 `ROLLBACK` 命令，会发生什么？

```

1  -- 小明所处的事务
2  ROLLBACK;
3
4  -- 此时无论对方是谁，如果再去查询结果就会发现：
5  SELECT * FROM user;
6  +---+-----+-----+
7  | id | name      | money |
8  +---+-----+-----+
9  | 1  | a        | 900  |
10 | 2  | b        | 1100 |
11 | 3  | 小明     | 1000 |
12 | 4  | 淘宝店   | 1000 |
13 +---+-----+-----+

```

这就是所谓的**脏读**，一个事务读取到另外一个事务还未提交的数据。这在实际开发中是不允许出现的。

## 读取已提交 READ COMMITTED

把隔离级别设置为 **READ COMMITTED**：

```

1  SET GLOBAL TRANSACTION ISOLATION LEVEL READ COMMITTED;
2  SELECT @@GLOBAL.TRANSACTION_ISOLATION;
3  +-----+
4  | @@GLOBAL.TRANSACTION_ISOLATION |
5  +-----+
6  | READ-COMMITTED                  |
7  +-----+

```

这样，再有新的事务连接进来时，它们就只能查询到已经提交过的事务数据了。但是对于当前事务来说，它们看到的还是未提交的数据，例如：

```
1  -- 正在操作数据事务（当前事务）
2  START TRANSACTION;
3  UPDATE user SET money = money - 800 WHERE name = '小明';
4  UPDATE user SET money = money + 800 WHERE name = '淘宝店';
5
6  -- 虽然隔离级别被设置为了 READ COMMITTED，但在当前事务中，
7  -- 它看到的仍然是数据表中临时改变数据，而不是真正提交过的数据。
8  SELECT * FROM user;
9  +---+-----+
10 | id | name      | money |
11 +---+-----+
12 | 1  | a         | 900   |
13 | 2  | b         | 1100  |
14 | 3  | 小明      | 200   |
15 | 4  | 淘宝店    | 1800  |
16 +---+-----+
17
18
19 -- 假设此时在远程开启了一个新事务，连接到数据库。
20 $ mysql -u root -p12345612
21
22 -- 此时远程连接查询到的数据只能是已经提交过的
23 SELECT * FROM user;
24 +---+-----+
25 | id | name      | money |
26 +---+-----+
27 | 1  | a         | 900   |
28 | 2  | b         | 1100  |
29 | 3  | 小明      | 1000  |
30 | 4  | 淘宝店    | 1000  |
31 +---+-----+
```

但是这样还有问题，那就是假设一个事务在操作数据时，其他事务干扰了这个事务的数据。例如：

```
1  -- 小张在查询数据的时候发现：
2  SELECT * FROM user;
3  +---+-----+
4  | id | name      | money |
5  +---+-----+
6  | 1  | a         | 900   |
7  | 2  | b         | 1100  |
8  | 3  | 小明      | 200   |
9  | 4  | 淘宝店    | 1800  |
10 +---+-----+
11
12 -- 在小张求表的 money 平均值之前，小王做了一个操作：
13 START TRANSACTION;
14 INSERT INTO user VALUES (5, 'c', 100);
```

```

15 COMMIT;
16
17 -- 此时表的真实数据是:
18 SELECT * FROM user;
19 +---+-----+-----+
20 | id | name      | money |
21 +---+-----+-----+
22 |  1 | a         |  900 |
23 |  2 | b         | 1100 |
24 |  3 | 小明      | 1000 |
25 |  4 | 淘宝店    | 1000 |
26 |  5 | c         |  100 |
27 +---+-----+-----+
28
29 -- 这时小张再求平均值的时候, 就会出现计算不符合的情况:
30 SELECT AVG(money) FROM user;
31 +-----+
32 | AVG(money) |
33 +-----+
34 | 820.0000    |
35 +-----+

```

虽然 **READ COMMITTED** 让我们只能读取到其他事务已经提交的数据, 但还是会出现问题, 就是在读取同一个表的数据时, 可能会发生前后不一致的情况。*\*这被称为\*不可重复读现象 ( READ COMMITTED )*。

## 幻读 REPEATABLE READ (可被重复读取)

将隔离级别设置为 **REPEATABLE READ ( 可被重复读取 )** :

```

1 SET GLOBAL TRANSACTION ISOLATION LEVEL REPEATABLE READ;
2 SELECT @@GLOBAL.TRANSACTION_ISOLATION;
3 +-----+
4 | @@GLOBAL.TRANSACTION_ISOLATION |
5 +-----+
6 | REPEATABLE-READ                |
7 +-----+

```

测试 **REPEATABLE READ** , 假设在两个不同的连接上分别执行 `START TRANSACTION` :

```

1 -- 小张 - 成都
2 START TRANSACTION;
3 INSERT INTO user VALUES (6, 'd', 1000);
4
5 -- 小王 - 北京
6 START TRANSACTION;
7
8 -- 小张 - 成都
9 COMMIT;

```

当前事务开启后, 没提交之前, 查询不到, 提交后可以被查询到。但是, 在提交之前其他事务被开启了, 那么在这条事务线上, 就不会查询到当前有操作事务的连接。相当于开辟出一条单独的线程。



无论小张是否执行过 `COMMIT`，在小王这边，都不会查询到小张的事务记录，而是只会查询到自己所处事务的记录：

```
1  SELECT * FROM user;
2  +-----+-----+-----+
3  | id | name      | money |
4  +-----+-----+-----+
5  |  1 | a         |  900 |
6  |  2 | b         | 1100 |
7  |  3 | 小明      | 1000 |
8  |  4 | 淘宝店    | 1000 |
9  |  5 | c         |  100 |
10 +-----+-----+-----+
```

这是因为小王在此之前开启了一个新的事务（`START TRANSACTION`），那么\*在他的这条新事务的线上，跟其他事务是没有联系的，也就是说，此时如果其他事务正在操作数据，它是不知道的。

然而事实是，在真实的数据表中，小张已经插入了一条数据。但是小王此时并不知道，也插入了同一条数据，会发生什么呢？

```
1  INSERT INTO user VALUES (6, 'd', 1000);
2  -- ERROR 1062 (23000): Duplicate entry '6' for key 'PRIMARY'
```

报错了，操作被告知已存在主键为 6 的字段。这种现象也被称为**幻读**，一个事务提交的数据，不能被其他事务读取到。

## 串行化

顾名思义，就是所有事务的**写入操作**全都是串行化的。什么意思？把隔离级别修改成 `SERIALIZABLE`：

```
1  SET GLOBAL TRANSACTION ISOLATION LEVEL SERIALIZABLE;
2  SELECT @@GLOBAL.TRANSACTION_ISOLATION;
3  +-----+
4  | @@GLOBAL.TRANSACTION_ISOLATION |
5  +-----+
6  | SERIALIZABLE                    |
7  +-----+
```

还是拿小张和小王来举例：

```
1  -- 小张 - 成都
2  START TRANSACTION;
3
4  -- 小王 - 北京
5  START TRANSACTION;
6
7  -- 开启事务之前先查询表，准备操作数据。
8  SELECT * FROM user;
9  +-----+-----+-----+
10 | id | name      | money |
11 +-----+-----+-----+
12 |  1 | a         |  900 |
```

```
13 | 2 | b | 1100 |
14 | 3 | 小明 | 1000 |
15 | 4 | 淘宝店 | 1000 |
16 | 5 | c | 100 |
17 | 6 | d | 1000 |
18 +-----+
19
20 -- 发现没有 7 号王小花，于是插入一条数据：
21 INSERT INTO user VALUES (7, '王小花', 1000);
```

此时会发生什么呢？由于现在的隔离级别是 **SERIALIZABLE (串行化)**，串行化的意思就是：假设把所有的事务都放在一个串行的队列中，那么所有的事务都会按照**固定顺序执行**，执行完一个事务后再继续执行下一个事务的**写入操作 (这意味着队列中同时只能执行一个事务的写入操作)**。

根据这个解释，小王在插入数据时，会出现等待状态，直到小张执行 `COMMIT` 结束它所处的事务，或者出现等待超时。