数据库 MySQL

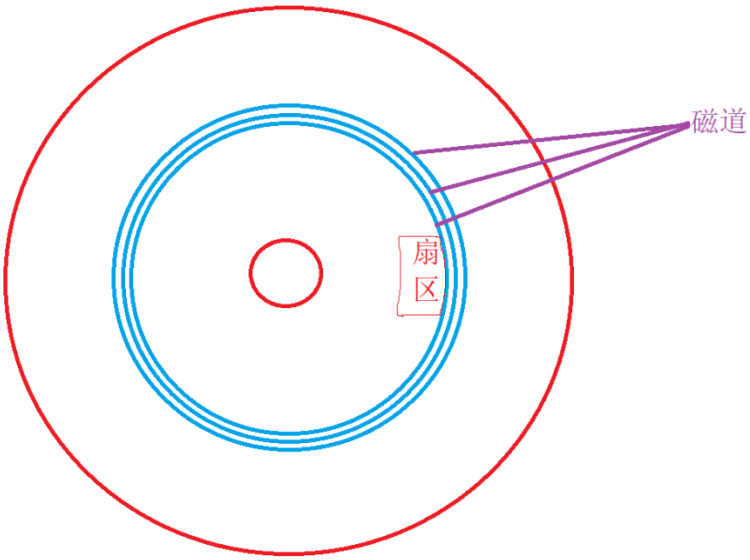
第一章：数据库简介

一、数据库基础

（1）数据库

1. 数据库的概念

数据库，英文名称为 Database，简称 DB。从字面意思来看就是存储数据的仓库；从专业角度解释为存储在计算机磁盘上的有组织、可共享的大量数据的集合。



2. 数据库的类型

数据库分为关系型数据库和非关系型数据库两大类。常见的关系型数据库有 MySQL 、Oracle 、SQL Server 、SQLite 、DB2 等。常见的非关系型数据库有 Redis 、MongoDB 等。

3. 数据库管理系统

数据库管理系统，英文名称为Database Management System，简称 DBMS。主要用于科学组织和存储数据、高效的获取和维护数据。

（2）MySQL 简介

MySQL 是目前最流行的开源的、免费的关系型数据库，适用于中小型甚至大型互联网应用，能够在 windows 和 linux 平台上部署。

MySQL Oracle 属于 Oracle 公司。

二、结构化查询语言

结构化查询语句，英文名称为 Structured Query Language，简称 SQL 。

（1）SQL 分类

结构化查询语句分为数据定义语言、数据操作语言、数据查询语言和数据控制语言四大类。

名称	描述	命令
数据定义语言（DDL）	数据库、数据表的创建、修改和删除	CREATE、ALTER、DROP
数据操作语言（DML）	数据的增加、修改和删除	INSERT、UPDATE、DELETE
数据查询语言（DQL）	数据的查询	SELECT
数据控制语言（DCL）	用户授权、事务的提交和回滚	GRANT、COMMIT、ROLLBACK

（2）数据库操作

创建数据库：

语法：

```
CREATE DATABASE [IF NOT EXISTS] 数据库名称 DEFAULT CHARACTER SET 字符集 COLLATE 排序规则;
```

当加上 IF NOT EXISTS 时，如果该名称的数据库不存在才会创建，否则不创建；如果不加IF NOT EXISTS ，若数据库存在还创建就会报错。

修改数据库：

语法：

```
ALTER DATABASE 数据库名称 CHARACTER SET 字符集 COLLATE 排序规则;
```

删除数据库：

语法：

```
DROP DATABASE [IF EXISTS] 数据库名称;
```

当加上 `IF EXISTS` 时，如果该名称的数据库存在才会删除，否则就不删除；如果不加 `IF EXISTS`，若数据库不存在再删除就会报错。

查看数据库：

语法：

```
SHOW DATABASES;
```

使用数据库：

语法：

```
USE 数据库名称;
```

示例：

```
--创建数据库
CREATE DATABASE IF NOT EXISTS lesson DEFAULT CHARACTER SET GBK COLLATE GBK_CHINESE_CI;
--修改数据库
ALTER DATABASE lesson CHARACTER SET UTF8 COLLATE UTF8_GENERAL_CI;
--删除数据库
DROP DATABASE IF EXISTS lesson;
--查看数据库
SHOW DATABASES;
--使用数据库，也可以理解为选择数据库
USE lesson;
```

（3）列类型

在 `MySQL` 中，常用列类型主要分为数值类型、日期时间类型、字符串类型。

1. 数值类型

类型	说明	取值范围	存储需求

类型	说明	取值范围	存储需求
`tinyint`	非常小的数据	有符号值：$-2^7 \sim 2^7-1$ 无符号值：$0 \sim 2^8-1$	1字节
`smallint`	较小的数据	有符号值：$-2^{15} \sim 2^{15}-1$ 无符号值：$0 \sim 2^{16}-1$	2字节
`mediumint`	中等大小的数据	有符号值：$-2^{23} \sim 2^{23}-1$ 无符号值：$0 \sim 2^{24}-1$	3字节
`int`	标准整数	有符号值：$-2^{31} \sim 2^{31}-1$ 无符号值：$0 \sim 2^{32}-1$	4字节
`bigint`	较大的整数	有符号值：$-2^{63} \sim 2^{63}-1$无符号值：$0 \sim 2^{64}-1$	8字节
`float`	单精度浮点数	无符号值：$1.1754351 * 10^{-38} \sim 3.402823466 * 10^{38}$	4字节
`double`	双精度浮点数	无符号值：$2.22507385 * 10^{-308} \sim 1.79769313 * 10^{308}$	8字节
`decimal`	字符串形式的浮点数	decimal(m, d) 整数部分最多m位，小数部分最多d位	m字节

2. 时间日期类型

类型	说明	取值范围
`DATE`	`YYYY-MM-dd`，日期格式	`1000-01-01` ~ `9999-12-31`
`TIME`	`HH:mm:ss`，时间格式	`-838:59:59.000000` ~ `838:59:59.000000`
	`YY-MM-dd`	`1000-01-01 00:00:00.000000`

`DATETIME`	`HH:mm:ss`，日期时间格式	取`9999-12-31 23:59:59.999999`
`TIMESTAMP`	`YYYY-MM-dd HH:mm:ss` 格式 表示的时间戳	`1970-01-01 00:00:01.000000` ~ `2038-01-19 03:14:07.999999`
`YEAR`	`YYYY` 格式的年份值	`1901` ~ `2155`

3. 字符串类型

类型	说明	最大长度
`char` [(M)]	固定长字符串，检索快但费空间， 0 <= M <= 255	M字符
`varchar` [(M)]	可变字符串，0 <= M <= 65535	变长度
`text`	文本串	2^{16}–1字节

4. 列类型修饰属性

属性名	说明	示例
UNSIGNED	无符号，只能修来修饰数值类型，表名该列数据不能出现负数	INT(4) UNSIGNED，表示只能为4位大于等于0的整数
`ZEROFILL`	不足的位数使用0来填充	INT(4) `ZEROFILL` ，如果给定的值为10，此时只有2位，而该列需要4位，不足的2位由0来填充，最终值为0010
NOT NULL	表示该列类型的值不能为空	`VARCHAR` (20) NOT NULL，表示该列数据不能为空值

属性名	说明	示例
DEFAULT	表示设置默认值	INT(4) DEFAULT 0，表示该列不赋值时默认为0
AUTO_INCREMENT	表示自增长，只能应用于数值列类型，该列类型必须为键，且不能为空	INT(11) AUTO_INCREMENT NOT NULL PRIMARY KEY。第一次为该列中插入值时为1，第二次为2

（4）数据表操作

1. 数据表类型

`MySQL` 中的数据表类型有许多，如 `MyISAM`、`InnoDB`、`HEAP`、`BOB`、`CSV` 等。其中最常用的就是 `MyISAM` 和 `InnoDB`。

`MyISAM` 与 `InnoDB` 的区别：

名称	`MyISAM`	`InnoDB`
事务处理	不支持	支持
数据行锁定	不支持	支持
外键约束	不支持	支持
全文索引	支持	不支持
表空间大小	较小	较大，约2倍

事务：涉及的所有操作是一个整体，要么都执行，要么都不执行。

数据行锁定：一行数据，当一个用户在修改该数据时，可以直接将该条数据锁定。

当涉及的业务操作以查询居多，修改和删除较少时，可以使用 `MyISAM`。
当涉及的业务操作经常会有修改和删除操作时，使用 `InnoDB`。

2. 创建数据表

语法：

```
CREATE TABLE [IF NOT EXISTS] 数据表名称(
字段名1 列类型(长度) [修饰属性] [键/索引] [注释],
字段名2 列类型(长度) [修饰属性] [键/索引] [注释],
字段名3 列类型(长度) [修饰属性] [键/索引] [注释],
......
字段名n 列类型(长度) [修饰属性] [键/索引] [注释]
) [ENGINE = 数据表类型][CHARSET=字符集编码] [COMMENT=注释];
```

3. 修改数据表

修改表明:

```
ALTER TABLE 表名 RENAME AS 新表名;
```

增加字段:

```
ALTER TABLE 表名 ADD 字段名 列类型(长度) [修饰属性] [键/索引] [注释];
```

查看表结构:

```
DESC 表名; -- 查看表结构
```

修改字段:

```
-- MODIFY 只能修改字段的修饰属性
ALTER TABLE 表名 MODIFY 字段名 列类型(长度) [修饰属性] [键/索引] [注释];

-- CHANGE 可以修改字段的名字以及修饰属性
ALTER TABLE 表名 CHANGE 字段名 新字段名 列类型(长度) [修饰属性] [键/索引] [注释];
```

删除字段:

```
ALTER TABLE 表名 DROP 字段名;
```

4. 删除数据表

语法:

```
DROP TABLE [IF EXISTS] 表名;
```

示例：

```
-- 创建学生表，表中有字段学号，学生，姓名，性别，年龄和成绩
CREATE TABLE IF NOT EXISTS student(
    `number` VARCHAR(30) NOT NULL PRIMARY KEY COMMENT '学号，主键',
    name VARCHAR(30) NOT NULL COMMENT '姓名',
    sex TINYINT(1) UNSIGNED DEFAULT 0 COMMENT '性别：0-男 1-女 2-其他',
    age TINYINT(3) UNSIGNED DEFAULT 0 COMMENT '年龄',
    score DOUBLE(5, 2) UNSIGNED COMMENT '成绩'
)ENGINE=InnoDB CHARSET=UTF8 COMMENT='学生表';

-- 将student表名修改为stu
ALTER TABLE student RENAME AS stu;

-- 在stu表中添加字段联系电话(phone)，类型为字符串，长度为11，非空
ALTER TABLE stu ADD phone VARCHAR(11) NOT NULL COMMENT '联系电话';

-- 将 stu 表中的 sex 字段的类型设置为 VARCHAR ，长度为2，默认值为'男',
-- 注释为 "性别，男，女，其他"
ALTER TABLE stu MODIFY sex VARCHAR(2) DEFAULT '男' COMMENT '性别：男，女，其他';
-- 将 stu 表中 phone 字段修改为 mobile ，属性保持不变
ALTER TABLE stu CHANGE phone mobile VARCHAR(11) NOT NULL COMMENT '联系电话';

-- 将 stu 表中的 mobile 字段删除
ALTER TABLE stu DROP mobile;

-- 删除数据表 stu
DROP TABLE IF EXISTS stu;
```

第二章：MySQL 数据库的操作

一、DML 语句

(1) DML 的概念

DML 全称为 Data Manipulation Language，表示数据操作语言。主要体现于对表数据的增删改操作。因此 DML 仅包括 INSERT、UPDATE 和 DELEETE 等语句。

(2) INSERT 插入数据

语法:

```sql
-- 需要注意，VALUES后的字段值必须与表名后的字段名一一对应
INSERT INTO 表名(字段名1, 字段名2, ..., 字段名n) VALUES(字段值1, 字段值2, ..., 字段值n);

-- 需要注意，VALUES后的字段值必须与创建表时的字段顺序保持一一对应
INSERT INTO 表名 VALUES(字段值1, 字段值2, ..., 字段值n);

-- 一次性插入多条数据
INSERT INTO 表名(字段名1, 字段名2, ..., 字段名n) VALUES(字段值1, 字段值2, ..., 字段值n),(字段值1, 字段值2, ..., 字段值n), ... , (字段值1, 字段值2, ..., 字段值n);

INSERT INTO 表名 VALUES(字段值1, 字段值2, ..., 字段值n), (字段值1, 字段值2, ..., 字段值n), ..., (字段值1, 字段值2, ..., 字段值n);
```

(3) UPDATE 修改数据

1. WHERE 条件子句:

在 Java 中，条件的表示通常都是使用关系运算符来表示，在 SQL 语句中也是一样，使用 >, <, >=, <=, != 来表示。不同的是，除此之外，SQL 中还可以使用 SQL 专用的关键字来表示条件。这些将在后面的 DQL 语句中详细讲解。

在 Java 中，条件之间的衔接通常都是使用逻辑运算符来表示，在 `SQL` 语句中也是一样，但通常使用 `AND` 来表示逻辑与 (`&&`)，使用 `OR` 来表示逻辑或(`||`)。

示例：

```
WHERE time > 20 && time < 40; <=> WHERE time > 20 and time <40;
```

2. `UPDATE` 语句：

```
UPDATE 表名 SET 字段名1=字段值1[,字段名2=字段值2, ..., 字段名n=字段值n] [WHERE 修改条件]
```

示例：将数据库的学分更该为4，学时更该为15。

```
UPDATE course SET score=4, `time`=15 WHERE name='数据库';
```

(4) `DELETE` 删除数据

语法：

```
DELETE FROM 表名 [WHERE 删除条件];
```

(5) `TRUNCATE` 清空数据

语法：

```
-- 清空表中数据
TRUNCATE [TABLE] 表名;
```

`DELETE` 与 `TRUNCATE` 的区别：

- `DELETE` 语句根据条件删除表中数据，而 `TRUNCATE` 语句则是将表中数据全部清空；如果 `DELETE` 语句要删除表中所有数据，那么在效率上要低于 `TRUNCATE` 语句。
- 如果表中有自增长列，`TRUNCATE` 语句会重置自增长的计数器，但 `DELETE` 语句不会。

- `TRUNCATE` 语句执行后，数据无法恢复，而 `DELETE` 语句执行后，可以使用事务回滚进行恢复。

示例：

```sql
CREATE TABLE IF NOT EXISTS stu_course(
    `number` INT(11) AUTO_INCREMENT PRIMARY KEY NOT NULL COMMENT '课程编号',
    name VARCHAR(20) NOT NULL COMMENT '课程名称',
    score DOUBLE(5, 2) NOT NULL COMMENT '学分'
) ENGINE=InnoDB CHARSET=UTF8 COMMENT '课程表';

ALTER TABLE stu_course RENAME AS course;
ALTER TABLE course ADD `time` INT(3) NOT NULL COMMENT '学时';
ALTER TABLE course MODIFY score DOUBLE(3, 1) NOT NULL COMMENT '学分';

-- 插入数据
INSERT INTO course(`number`, name, score, `time`) VALUES(1, 'JAVA基础', 4, 40);
INSERT INTO course VALUES (2, '数据库', 3, 20);
INSERT INTO course(`number`, name, score, `time`) VALUES (3, 'JSP', 5, 40),(4,'Spring',4,5);
INSERT INTO course VALUES (5, 'Spring Mvc', 2, 50),(6, 'SSM', 3,20);
-- 当列为自增长列或有默认值时，可以不给该列赋值
INSERT INTO course(name, score, `time`) VALUES ('HTML', 4, 20);

-- 将数据库的学分更该为4，学时更该为15
UPDATE course SET score=4, `time`=15 WHERE name='数据库';

-- 删除编号为1的数据
DELETE FROM course WHERE `number`=1;

-- 清空数据
TRUNCATE course;
```

二、DQL 语句

（1）DQL 语句的概念

DQL 全称是 Data Query Language，表示数据查询语言。体现在数据的查询操作上，因此，DQL 仅包括 SELECT 语句。

（2）SELECT 查看数据

语法：

```
SELECT ALL/DISTINCT * | 字段名1 AS 别名1[,字段名2 AS 别名2, ..., 字段名n AS 别名n] FROM 表名 WHERE 查询条件;
```

ALL 表示查询所有满足条件的记录，为默认值，可以省略；

DISTINCT 表示去掉查询结果中重复的记录；

AS 可以给数据列、数据表取一个别名。

（3）比较操作符

操作符	语法	说明
IS NULL	字段名 IS NULL	如果字段的值为NULL，则条件满足
IS NOT NULL	字段名 IS NOT NULL	如果字段的值不为NULL，则条件满足
BETWEEN...AND	字段名 BETWEEN 最小值 AND 最大值	如果字段的值在最小值与最大值之间（闭区间），则条件满足
LIKE	字段名 LIKE '%匹配内容%'	如果字段值包含有匹配内容，则条件满足
IN	字段名 IN(值1，值 2，...， 值n)	如果字段值在值1,值2, ...，值n中，则条件满足

示例：

```sql
CREATE TABLE IF NOT EXISTS stu_course(
    `number` INT(11) AUTO_INCREMENT PRIMARY KEY NOT NULL COMMENT '课程编号',
    name VARCHAR(20) NOT NULL COMMENT '课程名称',
    score DOUBLE(5, 2) NOT NULL COMMENT '学分'
) ENGINE=InnoDB CHARSET=UTF8 COMMENT '课程表';

ALTER TABLE stu_course RENAME AS course;
ALTER TABLE course ADD `time` INT(3) NOT NULL COMMENT '学时';
ALTER TABLE course MODIFY score DOUBLE(3, 1) NOT NULL COMMENT '学分';

-- 插入数据
INSERT INTO course(`number`, name, score, `time`) VALUES(1, 'JAVA基础', 4, 40);
INSERT INTO course VALUES (2, '数据库', 3, 20);
INSERT INTO course(`number`, name, score, `time`) VALUES (3, 'JSP', 5, 40),(4,'Spring',4,5);
INSERT INTO course VALUES (5, 'Spring Mvc', 2, 50),(6, 'SSM', 3,20);
-- 当列为自增长列或有默认值时，可以不给该列赋值
INSERT INTO course(name, score, `time`) VALUES ('HTML', 4, 20);
INSERT INTO course(name, score, `time`) VALUES ('面向对象JAVA', 3, 15);

-- 查找
SELECT score FROM course;
SELECT score,`time` FROM course;
SELECT score FROM course WHERE `number`>=4;
SELECT score,`time` FROM course WHERE name='SSM';
-- 列取别名
SELECT score AS '学分',`time` AS '学时' FROM course WHERE name='SSM';
-- 表取别名
SELECT c.name,c.score FROM course AS c WHERE c.name='JSP';
SELECT c.name AS '课程名称',c.score AS '学分' FROM course AS c WHERE c.name='JSP';
-- 比较操作符
```

```sql
SELECT * FROM course WHERE name IS NOT NULL;
SELECT * FROM course WHERE score BETWEEN 3 AND 4;
SELECT * FROM course WHERE score>=3 AND score<=4;
-- 筛选course表中name字段中包含ing的数据
SELECT * FROM course WHERE name LIKE '%ing%';
-- 筛选course表中name字段中以ing结尾的数据
SELECT * FROM course WHERE name LIKE '%ing';
-- 筛选course表中name字段中以JAVA开头的数据
SELECT * FROM course WHERE name LIKE 'JAVA%';
-- 查询课程名只有三个字符的数据
SELECT * FROM course WHERE name LIKE '___';
-- 查询课程名以S开始只有三个字符的数据
SELECT * FROM course WHERE name LIKE 'S__';
-- 筛选course表中number字段中为1,3,5的数据
SELECT * FROM course WHERE `number` IN(1,3,5);
```

(4) 分组

4,5,6节示例全部按照如下创建的数据库为准:

```sql
USE lesson;
DROP TABLE IF EXISTS student;
CREATE TABLE student(
    no BIGINT(20) AUTO_INCREMENT NOT NULL PRIMARY KEY COMMENT '学号，主键',
    name VARCHAR(20) NOT NULL COMMENT '姓名',
    sex VARCHAR(2) DEFAULT '男' COMMENT '性别',
    age INT(3) DEFAULT 0 COMMENT '年龄',
    score DOUBLE(5, 2) COMMENT '成绩'
) ENGINE=InnoDB CHARSET=UTF8 COMMENT='学生表';

INSERT INTO student VALUES(DEFAULT,'张三','男',20,59);
INSERT INTO student(no, name, sex, age, score) VALUES (DEFAULT, '李四', '女', 19,62);
INSERT INTO student(no, name, sex, age, score) VALUES (DEFAULT, '王五', '其他',21, 62);
INSERT INTO student(no, name, sex, age, score) VALUES (DEFAULT, '龙华', '男', 22,75);
INSERT INTO student(no, name, sex, age, score) VALUES (DEFAULT, '金凤', '女', 18,80);
```

```
INSERT INTO student(no, name, sex, age, score) VALUES (DEFAULT, '张华', '其他',27, 88);
INSERT INTO student(no, name, sex, age, score) VALUES (DEFAULT, '李刚', '男', 30,88);
INSERT INTO student(no, name, sex, age, score) VALUES (DEFAULT, '潘玉明', '女',28, 81);
INSERT INTO student(no, name, sex, age, score) VALUES (DEFAULT, '凤飞飞', '其他',32, 90);
```

1. 分组查询

分组查询所得的结果只是该组中的第一条数据，并不是所有数据。

语法:

```
SELECT ALL/DISTINCT * | 字段名1 AS 别名1[,字段名2 AS 别名2, ..., 字段名n AS 别名n] FROM 表名 WHERE 查询条件 GROUP BY 字段名1，字段名2,..., 字段名n
```

示例:

```
-- 分组查询
-- 查询成绩在80分以上的学生信息，并按性别分组
SELECT * FROM student WHERE score>=80 GROUP BY sex;
-- 查询成绩在60~80分的学生信息，并按性别和年龄分组
SELECT * FROM student WHERE score BETWEEN 60 AND 80 GROUP BY sex,age;
```

2. 聚合函数

`COUNT()`：统计满足条件的数据总条数，可用于任何字段。

`SUM()`：只能用于数值类型的字段或者表达式，计算该满足条件的字段值的总和。

`AVG()`：只能用于数值类型的字段或者表达式，计算该满足条件的字段值的平均值。

`MAX()`：只能用于数值类型的字段或者表达式，计算该满足条件的字段值的最大值。

`MIN()`：只能用于数值类型的字段或者表达式，计算该满足条件的字段值的最小值。

示例：

```
-- 聚合函数
-- 从学生表查询成绩在80分以上的学生人数
SELECT COUNT(*) AS total FROM student WHERE score>=80;
-- 从学生表查询及格的学生人数和总成绩
SELECT COUNT(*) total,SUM(score) totalScore FROM
student WHERE score>=60;
-- 从学生表查询男生、女生、其他类型的学生的平均成绩
SELECT sex,AVG(score) avgScore FROM student GROUP BY
sex;
-- 从学生表查询学生的最大年龄
SELECT MAX(age) FROM student;
-- 从学生表查询学生的最低分
SELECT MIN(score) FROM student;
```

3. 分组查询结果筛选

语法：

```
SELECT ALL/DISTINCT * | 字段名1 AS 别名1[,字段名1 AS 别名1,
..., 字段名n AS 别名n] FROM 表名 WHERE 查询条件 GROUP BY 字
段名1，字段名2,..., 字段名n HAVING 筛选条件
```

分组后如果还需要满足其他条件，则需要使用 `HAVING` 子句来完成。

示例：

```
-- 从学生表查询年龄在20~30之间的学生信息并按性别分组，找出组内平
均分在74分以上的组
SELECT * FROM student WHERE age BETWEEN 20 AND 30 GROUP
BY sex HAVING avg(score)>75;
```

（5）排序

语法：

```
SELECT ALL/DISTINCT * | 字段名1 AS 别名1[,字段名1 AS 别名1,
..., 字段名n AS 别名n] FROM 表名 WHERE 查询条件 ORDER BY 字
段名1 ASC|DESC，字段名2 ASC|DESC,..., 字段名n ASC|DESC
```

`ORDER BY` 必须位于 `WHERE` 条件之后。

`ASC` 表示升序，`DESC` 表示降序。

示例：

```
-- 从学生表查询年龄在18~30岁之间的学生信息并按成绩从高到低排列，如果成绩相同，则按年龄从小到大排列
SELECT * FROM student WHERE age BETWEEN 18 AND 30 ORDER BY score DESC,age ASC;
```

（6）分页

语法：

```
SELECT ALL/DISTINCT * | 字段名1 AS 别名1[,字段名1 AS 别名1, ..., 字段名n AS 别名n] FROM 表名 WHERE 查询条件 LIMIT 偏移量，查询条数
```

`LIMIT` 的第一个参数表示偏移量，也就是跳过的行数。

`LIMIT` 的第二个参数表示查询返回的最大行数，可能没有给定的数量那么多行。

示例：

```
-- 从学生表分页查询成绩及格的学生信息，每页显示3条，查询第2页学生信息
SELECT * FROM student WHERE score>=60 LIMIT 0, 3;
-- 第一个参数为偏移量，第二个参数为显示数据量
SELECT * FROM student WHERE score>=60 LIMIT 3, 3;
SELECT * FROM student WHERE score>=60 LIMIT 6, 3;
```

如果一个查询中包含分组、排序和分页，那么它们之间必须按照分组 -> 排序 -> 分页的先后顺序排列。

三、`MySQL` 常用函数

（1）常用数学函数

函数	说明	示例
ABS(X)	返回X的绝对值。	SELECT ABS(-2);
FLOOR(X)	返回不大于X的最大数。	SELECT FLOOR(1.3)
CEIL(X)	返回不小于X的最大数。	SELECT CEIL(1.3);
TRUNCATE(X, D)	返回数值X保留小数点后D位的值，截断时不进行四舍五入。	SELECT TRUNCATE(1.2326, 3);
ROUND(X)	返回离X最近的整数，截断时进行四舍五入。	SELECT ROUND(1.8);
ROUND(X, D)	保留X小数点后D位的值，截断时要四舍五入	SELECT ROUND(1.2325, 3);
RAND()	返回0~1的随机数。	SELECT RAND();
MOD(N, M)	返回N除以M后的余数。	SELECT MOD(9, 2);

(2) 常用字符串函数

函数	说明	示例
CHAR_LENGTH(str)	计算字符串字符个数。	SELECT CHAR_LENGTH('字符串ABC');
LENGTH(str)	返回值为字符串的长度，单位为字节。	SELECT LENGTH('字符串ABC');
CONCAT(s1, s2, ...)	将多个字符串拼在一起，若其中任意一个为NULL则返回值为NULL。	SELECT CONCAT('ad', 'min');
LOWER(str) LCASE(str)	将字符串中的字母全部转换成小写。	SELECT LOWER('ABC'); SELECT LCASE('ABC');
UPPER(str) UCASE(str)	将字符串中的字母全部转换成大写。	SELECT UPPER('abc'); SELECT UCASE('abc');

函数	说明	示例
LEFT(s, n) RIGHT(s, n)	返回字符串s从最左（右）边开始的n个字符。	SELECT LEFT('abcdefg', 5); SELECT RIGHT('abcdefg', 5);
LTRIM(s) RTRIM(s)	返回字符串s，其左（右）边的空格全部被删除。	SELECT LTRIM(' abcd'); SELECT RTRIM('abcdefgh ');
TRIM(s)	返回字符串s，删除其两边空格。	SELECT TRIM(' abc ');
REPLACE(s, s1, s2)	返回一个字符串，用字符串s2替换字符串s中的所有字符串s1。	SELECT REPLACE('ababc', 'ab', 'd');
SUBSTRING(s, n, len)	从字符串s中返回一个第n个字符开始长度为len的字符串。	SELECT SUBSTRING('abcdef', 2, 3);

示例：

```
-- 查询计科有多少人
SELECT COUNT(*) FROM stu WHERE LEFT(class,2)='计科';
-- 查询计科和软工各有多少人
SELECT LEFT(class, 2), COUNT(*) FROM stu GROUP BY LEFT(class, 2);
-- 查询名字有四个字的学生
SELECT * FROM stu WHERE CHAR_LENGTH(`name`)=4;
-- 查询成绩能够被10整除的考试信息
SELECT * FROM score WHERE MOD(score, 10)=0;
```

(3) 日期与时间函数

函数	说明	示例
CURDATE() CURRENT_DATE()	返回当前日期。	SELECT CURDATE();
CURTIME() CURRENT_TIME()	返回当前时间。	SELECT CURTIME();
NOW() CURRENT_TIMESTAMP() SYSDATE()	返回当前日期和时间。	SELECT NOW();
YEAR(d)	返回给定日期d中的年。	SELECT YEAR(NOW());
MONTH(d)	返回给定日期d的月份，范围1~12。	SELECT MONTH(NOW());
DAYOFMONTH(d)	返回给定日期d是当月的第几天。	DAYOFMONTH(NOW());
HOUR(d)	返回给定日期d的小时数。	SELECT HOUR(NOW());
MINUTE(d)	返回给定日期d的分钟数。	SELECT MINUTE(NOW());
SECOND(d)	返回给定日期d的秒数。	SELECT SECOND(NOW());

函数	说明	示例
ADDDATE(d, n)	返回给定日期d加上n天的日期。	SELECT ADDDATE(NOW(), 3);
TIMESTAMPDIFF(INTERVAL expr type, d1, d2)	返回给定日期d1与d2的时间差。	SELECT TIMESTAMPDIFF(YEAR, '2019-12-14', '2024-9-23');
DATE_FORMAT(d, f)	返回给定日期格式的字符串。	SELECT DATE_FORMAT(NOE(), '%Y-%m-%d %H:%i:%s');

示例:

```
SELECT CURRENT_DATE();
SELECT CURRENT_TIME();
SELECT
YEAR(NOW()),MONTH(NOW()),DAYOFMONTH(NOW()),DAYOFWEEK(NOW()),
HOUR(NOW()),MINUTE(NOW()),SECOND(NOW());
SELECT ADDDATE(NOW(),3), ADDDATE(NOW(),-3);
SELECT TIMESTAMPDIFF(YEAR,'2020-10-10',NOW());
SELECT DATE_FORMAT(NOW(),'%Y-%m-%d %h:%i:%s');
-- 查询年龄在20岁以上的学生信息
SELECT * FROM stu WHERE
TIMESTAMPDIFF(YEAR,birthday,NOW()) > 35;
-- 查询今天过生日的学生信息
SELECT * FROM stu WHERE MONTH(birthday)=MONTH(NOW())
AND DAYOFMONTH(birthday)=DAYOFMONTH(NOW());
-- 查询本周过生日的学生信息
SELECT RIGHT(DATE_FORMAT(ADDDATE(NOW(),-
DAYOFWEEK(NOW())),'%Y-%m-%d'),5);
SELECT RIGHT(DATE_FORMAT(ADDDATE(NOW(),7-
DAYOFWEEK(NOW())),'%Y-%m-%d'),5);
SELECT * FROM stu WHERE RIGHT(birthday,5) >
RIGHT(DATE_FORMAT(ADDDATE(NOW(),-
DAYOFWEEK(NOW())),'%Y-%m-%d'),5)
```

```
17  AND RIGHT(birthday,5) <=
    RIGHT(DATE_FORMAT(ADDDATE(NOW(),7-
    DAYOFWEEK(NOW())),'%Y-%m-%d'),5);
```

（4）条件判断函数

1. IF 函数

IF 语句：

```
1  IF(条件,表达式1,表达式2)
```

如果条件满足，则执行表达式1，否则执行表达式2。

IFNULL 语句：

```
1  IFNULL(字段, 表达式)
```

如果字段值为空，则使用表达式，否则，使用字段值。

2. CASE...WHEN 函数

CASE WHEN 语句：

```
1  CASE WHEN 条件1 THEN 表达式1 [WHEN 条件2 THEN 表达式2 ...]
   ELSE 表达式n END;
```

如果条件1满足，则使用表达式1；【如果条件2满足，则使用表达式2，...】否则，使用表达式n。相当于 Pascal 中的多重 `if..then..else` 语句。

CASE ... WHEN语句：

```
1  CASE 表达式 WHEN 值1 THEN 表达式1 [WHEN 值2 THEN 表达式2
   ...] ELSE 表达式n END;
```

如果表达式的执行结果为值1，则使用表达式1；【执行结果为值2，则使用表达式2，...】否则，使用 表达式n。相当于Java中的 `switch` 语句。

示例：

```
1  -- 将学生成绩展示为及格和不及格
```

```sql
SELECT id,stu_name,course, IF(score>=60, '及格', '不及格') score FROM score;
-- 将未参加考试的学生成绩展示为缺考
SELECT id,stu_name,course, IFNULL(score, '缺考') score FROM score;


SELECT stu_name,course,(CASE WHEN (course='Java') THEN score ELSE 0 END) Java FROM score;
-- 查询每位学生的各课程成绩，行转列
SELECT
    stu_name,
    course,
    MAX(CASE WHEN (course='Java') THEN score ELSE 0 END) Java,
    MAX(CASE WHEN (course='Html') THEN score ELSE 0 END) Html,
    MAX(CASE WHEN (course='Jsp') THEN score ELSE 0 END) Jsp,
    MAX(CASE WHEN (course='Spring') THEN score ELSE 0 END) Spring
FROM score GROUP BY stu_name;

SELECT
    stu_name,
    course,
    MAX(CASE course WHEN 'Java' THEN score ELSE 0 END) Java,
    MAX(CASE course WHEN 'Html' THEN score ELSE 0 END) Html,
    MAX(CASE course WHEN 'Jsp' THEN score ELSE 0 END) Jsp,
    MAX(CASE course WHEN 'Spring' THEN score ELSE 0 END) Spring
FROM score GROUP BY stu_name;

-- 查询各班级性别人数
SELECT
        class,
        SUM(CASE sex WHEN 0 THEN 1 ELSE 0 END) '男',
        SUM(CASE sex WHEN 1 THEN 1 ELSE 0 END) '女',
```

```
32          SUM(CASE sex WHEN 2 THEN 1 ELSE 0 END) '其他'
33  FROM stu GROUP BY class;
```

(5) 其他函数

1. 数字格式化函数

语法:

```
1  SELECT FORMAT(X,D); --将数字X格式化，将X保留到小数点后D位，截断时要进行四舍五入。
```

2. 系统信息函数

函数	说明	示例
VERSION()	获取数据库的版本号。	SELECT VERSION();
CONNECTION_ID()	获取服务器的连接数。	SELECT CONNECTION_ID();
DATABASE() SCHEMA()	获取当前数据库名。	SELECT DATABASE(); SELECT SCHEMA();

函数	说明	示例
USER() SYSTEM_USER() SESSION_USER() CURRENT_USER() CURRENT_USER	获取当前用户名。	SELECT USER(); SELECT CURRENT_USER();

示例：

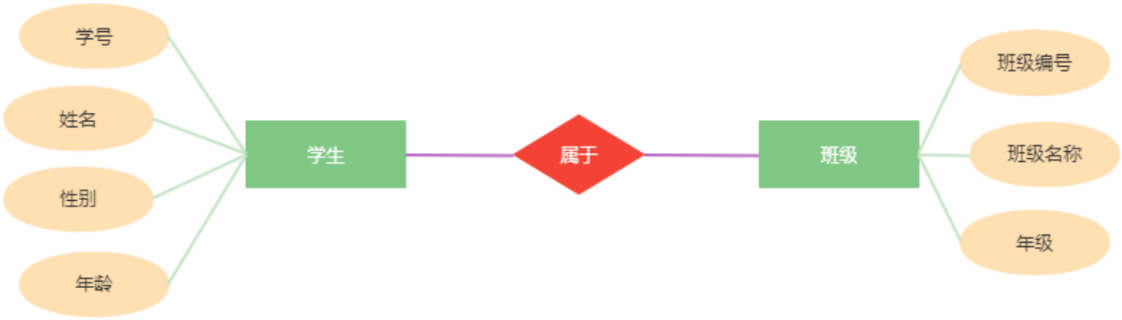
```
SELECT ROUND(1.2345, 3);
SELECT
CURRENT_USER(),CURRENT_USER,USER(),SESSION_USER(),SYSTEM_USER();
SELECT VERSION();
SELECT CONNECTION_ID();
SELECT DATABASE();
```

四、多表查询

（1）表与表之间的关系

1. 表与表之间的关系

数据表是用来描述实体信息的，比如可以使用数据表来描述学生信息，也可以用数据表来描述班级信息，这样就会存在学生表和班级表。而学生和班级显然存在着一种关系：



这种关系在数据库中体现就称之为表与表之间的关系。数据库通过主外键关联关系来体现表与表之间的关联关系。

2. 主外键关联关系

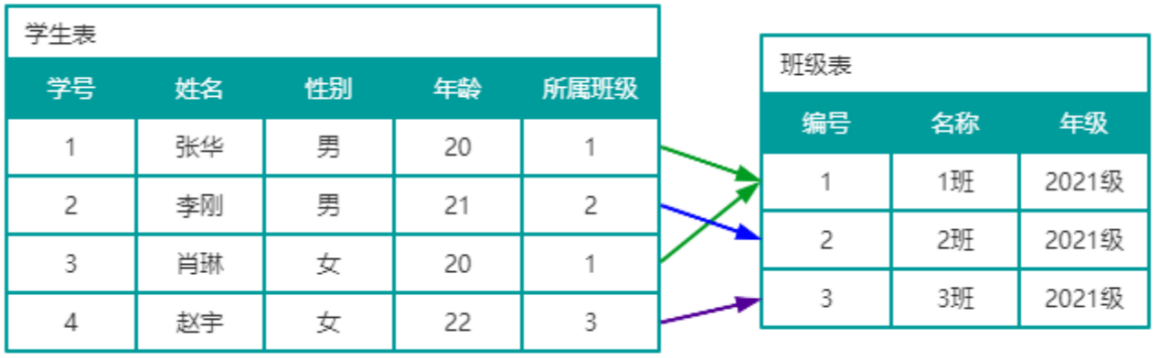
学生表

学号	姓名	性别	年龄
1	张华	男	20
2	李刚	男	21
3	肖琳	女	20
4	赵宇	女	22

班级表

编号	名称	年级
1	1班	2021级
2	2班	2021级
3	3班	2021级

如图所示，此时学生表和班级表并没有任何关系，然而实际上学生和班级是存在归属关系。可以在学生表中添加一个字段，表名该学生所属班级，该字段值使用的是班级表中的主键，在学生表中称之为外键。这样学生表中的所属班级（外键）与班级表中的编号（主键）就产生关联关系，这种关联关系称为主外键关联关系。



3. 主外键关联关系的定义

```
DROP TABLE IF EXISTS class;
CREATE TABLE class(
    id INT(11) AUTO_INCREMENT PRIMARY KEY NOT NULL COMMENT '班级编号',
    name VARCHAR(30) NOT NULL COMMENT '班级名称',
    grade VARCHAR(30) NOT NULL DEFAULT '男' COMMENT '年级'
) ENGINE=INNODB CHARSET=UTF8 COMMENT='学生表';

DROP TABLE IF EXISTS stu;
CREATE TABLE stu(
    number BIGINT(20) NOT NULL COMMENT '学号',
    name VARCHAR(30) NOT NULL COMMENT '姓名',
    sex VARCHAR(2) NOT NULL DEFAULT '男' COMMENT '性别',
```

```
    age TINYINT(3) UNSIGNED DEFAULT 0 COMMENT '年龄',
    class_id INT(11) NOT NULL COMMENT '所属班级',
--  指定number为主键
    PRIMARY KEY(number),
--  指定class_id为外键，关联class表中的id字段
    FOREIGN KEY (class_id) REFERENCES class(id)
) ENGINE=INNODB CHARSET=UTF8 COMMENT='学生表';
```

4. 约束

主键约束:

```
-- 添加主键约束：保证数据的唯一性
ALTER TABLE 表名 ADD PRIMARY KEY(字段名1,字段名2, ..., 字段名n);

-- 删除主键约束
ALTER TABLE 表名 DROP PRIMARY KEY;
```

外键约束:

```
-- 添加外键约束
ALTER TABLE 表名1 ADD CONSTRAINT 外键名称 FOREIGN KEY(表名1的字段名) REFERENCES 表名2(表名2的字段名);

-- 删除外键约束
ALTER TABLE 表名 DROP FOREIGN KEY 外键名称;
```

唯一约束:

```
-- 为字段添加唯一约束
ALTER TABLE 表名 ADD CONSTRAINT 约束名称 UNIQUE(字段名1, 字段名2, ..., 字段名n);

-- 删除字段的唯一约束
ALTER TABLE 表名 DROP KEY 约束名称;
```

非空约束:

```sql
-- 为字段添加非空约束
ALTER TABLE 表名 MODIFY 字段名 列类型 NOT NULL;

-- 删除字段非空约束
ALTER TABLE 表名 MODIFY 字段名 列类型 NULL;
```

默认值约束:

```sql
-- 为字段添加默认值
ALTER TABLE 表名 ALTER 字段名 SET DEFAULT 默认值;

-- 删除字段的默认值
ALTER TABLE 表名 ALTER 字段名 DROP DEFAULT;
```

自增约束:

```sql
-- 为字段添加自增约束
ALTER TABLE 表名 MODIFY 字段名 列类型 AUTO_INCREMENT;

-- 为字段删除自增约束
ALTER TABLE 表名 MODIFY 字段名 列类型;
```

示例:

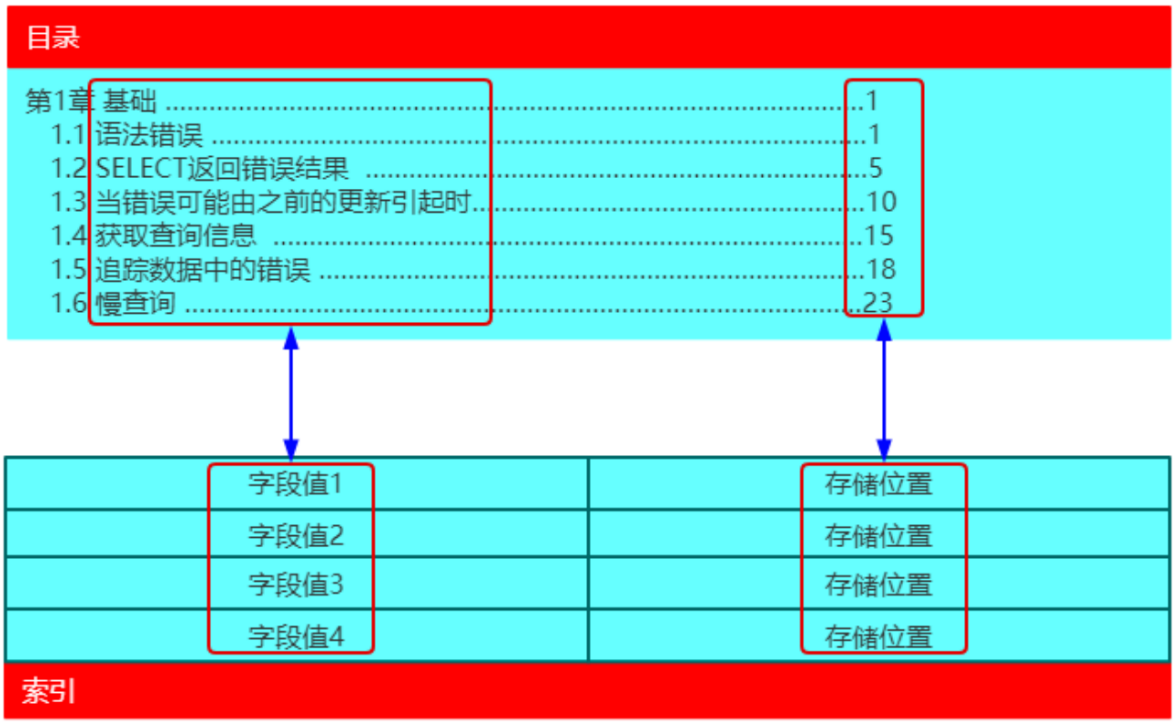
```sql
ALTER TABLE stu DROP PRIMARY KEY;
ALTER TABLE stu ADD PRIMARY KEY(number);


ALTER TABLE stu DROP FOREIGN KEY stu_ibfk_1;
ALTER TABLE stu ADD CONSTRAINT fk_class_id FOREIGN KEY(class_id) REFERENCES class(id);

ALTER TABLE stu ADD CONSTRAINT un_name UNIQUE(name);
INSERT INTO `lesson`.stu (`name`,sex,age,class_id) VALUES ('张三', '男', 20, 1)
-- Duplicate entry '张三' for key 'stu.un_name' 唯一数据

ALTER TABLE stu DROP KEY un_name;

ALTER TABLE stu ALTER sex DROP DEFAULT;
ALTER TABLE stu ALTER sex SET DEFAULT '女';
```

```
16
17 ALTER TABLE stu MODIFY number BIGINT(20) NOT NULL
   AUTO_INCREMENT;
18 ALTER TABLE stu MODIFY number BIGINT(20) NOT NULL;
```

（2）索引

1. 索引的概念

在关系数据库中，索引是一种单独的、物理的对数据库表中一列或多列的值进行排序的一种存储结构，它是表中一列或多列值的集合和相应的指向表中物理标识这些值的数据页的逻辑指针清单。



索引可以对比书籍的目录来理解。

2. 索引的作用

- 保证数据的准确性；
- 提高检索速度；
- 提高系统性能。

3. 索引的类型

- 唯一索引（UNIQUE）：不可以出现相同的值，可以有NULL值；
- 普通索引（INDEX）：允许出现相同的索引内容；
- 主键索引（PRIMARY KEY）：不允许出现相同的值；

- 全文索引（FULLTEXT INDEX）：`InnoDB` 不支持，可以针对值中的某个单词，但效率确实不高；
- 组合索引：实质上是将多个字段建到一个索引里，列值的组合必须唯一。

4. 索引的操作

```
-- 创建索引
ALTER TABLE 表名 ADD INDEX 索引名称 (字段名1, 字段名2, ..., 字段名n);

-- 创建全文索引
ALTER TABLE 表名 ADD FULLTEXT 索引名称 (字段名1, 字段名2, ..., 字段名n);

-- 查看索引
SHOW INDEX FROM 表名;

-- 删除索引
ALTER TABLE 表名 DROP INDEX 索引名称;
```

示例：

```
ALTER TABLE stu ADD INDEX sexIndex(sex);
SHOW INDEX FROM stu;
ALTER TABLE stu DROP INDEX sexIndex;
```

5. 使用索引的注意事项

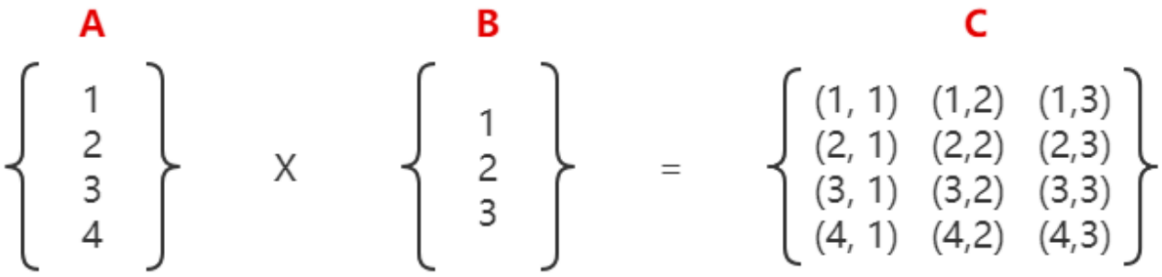
- 虽然索引大大提高了查询速度，但也会降低更新表的速度，比如对表进行 `INSERT`，`UPDATE` 和 `DELETE` 操作，此时，数据库不仅要保存数据，还要保存一下索引文件；
- 建立索引会占用磁盘空间的索引文件。如果索引创建过多（尤其是在字段多、数据量大的表上创建索引），就会导致索引文件过大，这样反而会降低数据库性能。因此，索引要建立在经常进行查询操作的字段上；
- 不要在列上进行运算（包括函数运算），这会忽略索引的使用；
- 不建议使用 `like` 操作，如果非使用不可，注意正确的使用方式。`like '%查询内容%'` 不会使用索引，而 `like '查询内容%'` 可以使用索引；

- 避免使用 `IS NULL`、`NOT IN`、`<>`、`!=`、`OR` 操作，这些操作都会忽略索引而进行全表扫描。

（3）多表查询

1. 笛卡尔积

笛卡尔积又称为笛卡尔乘积，由笛卡尔提出，表示两个集合相乘的结果。



笛卡尔积与多表查询有什么关系呢？每一张表可以看做是一个数据的集合，多表关联串时，这些表中的数据就会形成笛卡尔积。

2. 内连接

内连接相当于在笛卡尔积的基础上加上了连接条件。当没有连接条件时，内连接上升为笛卡尔积。

```
SELECT 字段名1, 字段名2, ..., 字段名n FROM 表1 [INNER] JOIN 表2 [ON 连接条件];

SELECT 字段名1, 字段名2, ..., 字段名n FROM 表1, 表2 [WHERE 关联条件 AND 查询条件];
```

3. 外连接

外连接涉及到两张表：主表和从表，要查询的信息主要来自于哪张表，哪张表就是主表。

外连接查询的结果为主表中所有的记录。如果从表中有和它匹配的，则显示匹配的值，这部分相当于内连接查询出来的结果；如果从表中没有和它匹配的，则显示 `null`。

外连接查询的结果 = 内连接的结果 + 主表中有的而内连接结果中没有的记录。

外连接分为左外连接和右外连接两种。左外连接使用 `LEFT JOIN` 关键字，`LEFT JOIN` 左边的是主表；右外连接使用 `RIGHT JOIN` 关键字，`RIGHT JOIN` 右边的是主表。

左外连接：

```
SELECT 字段名1, 字段名2, ..., 字段名n FROM 主表 LEFT JOIN 从表 [ON 连接条件];
```

右外连接：

```
SELECT 字段名1, 字段名2, ..., 字段名n FROM 从表 RIGHT JOIN 主表 [ON 连接条件];
```

示例：

```
SELECT COUNT(*) FROM score;
SELECT COUNT(*) FROM stu;
-- 笛卡尔积中的数据个数
SELECT COUNT(*) FROM stu, score;

-- 内连接
SELECT COUNT(*) FROM stu INNER JOIN score ON stu.id=score.stu_id;
SELECT COUNT(*) FROM stu, score WHERE stu.id=score.stu_id;

-- 外连接
SELECT * FROM stu a LEFT JOIN score b ON a.id=b.stu_id WHERE score IS NULL;
SELECT * FROM stu a RIGHT JOIN score b ON a.id=b.stu_id;S
```

（4）子查询

1. 子查询的概念

子查询就是嵌套在其他查询中的查询。因此，子查询出现的位置只有3种情况：在 `SELECT... FROM` 之间、在 `FROM...WHERE` 之间、在 `WHERE` 之后。

2. `SELECT ... FROM` 之间

执行时机是在查询结果出来之后。

3. `FROM ... WHERE` 之间

执行时机是一开始就执行。

4. `WHERE` 之后

执行时机是一开始就执行。

示例:

```
-- 查询stu表所有学生信息，并将性别按男、女、其他展示
SELECT
    id,
    `name`,
    (SELECT text FROM dict WHERE type='sex' AND value=sex) sex, -- 子查询
    birthday,
    class
FROM
    stu;

-- 查询年龄与Java成绩都与强鸿晖的年龄与Java成绩相同的学生信息
SELECT
    c.*,d.*
FROM
    stu c
    INNER JOIN score d ON c.id = d.stu_id
    INNER JOIN (
    SELECT
        TIMESTAMPDIFF(YEAR,a.birthday,NOW()) age,
        b.score
    FROM
        stu a
        INNER JOIN score b ON a.id = b.stu_id
    WHERE
        `name` = '强鸿晖'
        AND b.course = 'Java'
```

```
        ) e ON TIMESTAMPDIFF(YEAR,c.birthday,NOW())=
e.age
    AND d.score = e.score
WHERE
    d.course = 'Java';

-- 查询Java成绩最高的学生信息
SELECT
    *
FROM
    stu a
    INNER JOIN score b ON a.id = b.stu_id
WHERE
    b.score =(
    SELECT
        MAX( score )
    FROM
        score
    WHERE
        course = 'Java'
    )
    AND b.course = 'Java';
```

五、存储过程，函数，触发器和视图

（1）变量

在 MySQL 中，变量分为四种类型，即局部变量、用户变量、会话变量和全局变量。其中局部变量和用户变量在实际应用中使用较多，会话变量和全局变量使用较少，因此作为了解即可。

1. 全局变量

MySQL 全局变量会影响服务器整体操作，当服务启动时，它将所有全局变量初始化为默认值。要想更改全局变量，必须具有管理员权限。其作用域为服务器的整个生命周期。

全局变量操作语法：

```sql
-- 查看所有全局变量
SHOW GLOBAL VARIABLES;

-- 设置某个全局变量
SET GLOBAL 变量名 = 状态值;
SET @@GLOBAL.变量名 = 状态值;

-- 查询某个全局变量
SELECT @@GLOBAL.变量名;
SHOW GLOBAL VARIABLES LIKE '%变量名%';
```

2. 会话变量

MySQL 会话变量是服务器为每个连接的客户端维护的一系列变量。其作用域仅限于当前连接，因此，会话变量是独立的。

会话变量操作语法：

```sql
-- 查看所有会话变量
SHOW SESSION VARIABLES;

-- 设置某个会话变量
SET SESSION 变量名 = 状态值;
SET @@SESSION.变量名 = 状态值;

-- 省略变量类型时默认为会话变量
-- SESSION关键字可以省略或者用LOCAL代替
SET 变量名 = 状态值;

-- 查询某个会话变量
SELECT @@变量名;
SHOW SESSION VARIABLES LIKE '%变量名%';
SELECT @@LOCAL.变量名;
```

3. 用户变量

MySQL 用户变量，MySQL 中用户变量不用提前申明，在用的时候直接用 `@变量名` 使用就可以了。其作用域为当前连接。

```
-- 修改用户变量，在用SET进行赋值时可以用“=”或“:=”两种赋值符号赋值
SET @变量名 = 属性值;
SELECT 属性值 INTO @变量名;

-- 查询用户变量
SELECT @变量名;

-- 修改后查询用户变量，在用SELECT进行赋值时只能用“:=”符号赋值
SELECT @变量名 := 属性值;
```

4. 局部变量

MySQL 局部变量，只能用在 `BEGN/END` 语句块中，比如存储过程中的 `BEGN/END` 语句块。

```
BEGIN
-- 定义局部变量
    DECLARE i INT(11) DEFAULT 0;
--  修改局部变量
    SET i = 10;
    SELECT i := 20;
    SELECT 30 INTO i;
END
```

示例：

```
-- 查看所有全局变量
SHOW GLOBAL VARIABLES;
-- 设置某个全局变量
SET GLOBAL sql_warnings = ON;
SET @@GLOBAL.sql_warnings = OFF;
-- 查询某个全局变量
SELECT @@GLOBAL.sql_warnings;
SHOW GLOBAL VARIABLES LIKE '%sql_warnings%';

-- 查看所有会话变量
SHOW SESSION VARIABLES;
-- 设置某个会话变量
SET SESSION auto_increment_increment = 5;
SET @@SESSION.auto_increment_increment = 2;
```

```
-- 省略变量类型时默认为会话变量
-- SESSION关键字可以省略或者用LOCAL代替
SET auto_increment_increment = 1;
-- 查询某个会话变量
SELECT @@auto_increment_increment;
SHOW SESSION VARIABLES LIKE
'%auto_increment_increment%';
SELECT @@LOCAL.auto_increment_increment;

-- 修改用户变量
SET @a = 5;
SELECT 1 INTO @a;
-- 查询用户变量
SELECT @a;
-- 修改后查询用户变量
SELECT @a := 10;
-- 将stu表中的数据从id101开始，新设置一个字段num，从1开始自增
SELECT (SELECT @INDEX := @INDEX + 1) num, a.* FROM
score a,(SELECT @INDEX := 0) b WHERE id > 100;

BEGIN
-- 定义局部变量
    DECLARE i INT(11) DEFAULT 0;
--  修改局部变量
    SET i = 10;
    SELECT i := 20;
    SELECT 30 INTO i;
END
```

（2）存储过程

1. 存储过程的概念与优点

存储过程的概念：

在大型数据库系统中，存储过程是一组为了完成特定功能而存储在数据库中的 SQL 语句集，一次编译后永久有效。

存储过程的优点：

- 运行速度快：在存储过程创建的时候，数据库已经对其进行了一次解析和优化。存储过程一旦执行，在内存中就 会保留一份这个存储过程，

下次再执行同样的存储过程时，可以从内存中直接调用，所以执行速度会比普通 SQL 快。

- 减少网络传输：存储过程直接就在数据库服务器上跑，所有的数据访问都在数据库服务器内部进行，不需要传输数据到其他服务器，所以会减少一定的网络传输。
- 增强安全性： 提高代码安全，防止 SQL 被截获、篡改。

3. 存储过程的使用

存储过程使用语法：

```
-- 声明分隔符
[DELIMITER $$]
CREATE PROCEDURE 存储过程名称 ([IN | OUT | INOUT] 参数名1 数据类型, [[IN | OUT |INOUT] 参数名2 数据类型, ..., [IN | OUT | INOUT] 参数名n 数据类型])
-- IN表示输入，OUT表示输出，INOUT表示输入同时输入

-- 语句块开始
BEGIN
-- SQL语句集
END[$$]
-- 还原分隔符
[DELIMITER ;]

-- 调用存储过程
CALL 存储过程名(参数1, 参数2, ...);
```

示例：使用存储过程完成银行转账业务

```
DROP PROCEDURE IF EXISTS transferMoney;
-- 创建函数
DELIMITER //
CREATE PROCEDURE transferMoney(IN transferFrom BIGINT, IN transferTo BIGINT, IN money DOUBLE(20,3))
BEGIN
--  用于标记转账是否成功
    DECLARE result TINYINT(1) DEFAULT 0;
    UPDATE account SET balance = balance - money WHERE account=transferFrom AND balance >= money;
    IF ROW_COUNT() = 1 THEN
```

```
        UPDATE account SET balance = balance + money
WHERE account=transferTo;
        IF ROW_COUNT() = 1 THEN
            SET result = 1;
        END IF;
    END IF;
--  查询结果
    SELECT result;
END //
DELIMITER ;
-- 调用函数
CALL transferMoney(123456, 123457, 2000);
```

如果转账账户已经将钱转出去，而在执行目标账户增加余额的时候出现了异常或者目标账户输入错误，此时应该怎么办呢？

MySQL 对数据的操作提供了事务的支持，用来保证数据的一致性，可以有效的解决此类问题。

（3）事务

1. 事务的概念

事务(Transaction)是访问并可能操作各种数据项的一个数据库操作序列，这些操作要么全部执行，要么全部不执行，是一个不可分割的工作单位。事务由事务开始与事务结束之间执行的全部数据库操作组成。

2. 事务的特性（ACID）

- 原子性（Atomicity）：事务的各元素是不可分的（原子的）。它们是一个整体。要么都执行，要么都不执行。
- 一致性（Consistency）：当事务完成时，必须保证所有数据保持一致状态。当转账操作完成时，所有账户的总金额应该保持不变，此时数据处于一致性状态；如果总金额发生了改变，说明数据处于非一致性状态。
- 隔离性（Isolation） 对数据操作的多个并发事务彼此独立，互不影响。比如两个用户同时都在进行转账操作，但彼此都不影响对方。
- 持久性（Durability） 对于已提交事务，系统必须保证该事务对数据库的改变不被丢失，即使数据库出现故障。

3. 事务的操作

事务操作语法：

```
--  开启事务
START TRANSACTION;

--  回滚事务（不执行事务，回退至事务未开始时）
ROLLBACK;

--  提交事务
COMMIT;

-- 发生SQLEXCEPTION时的处理方式：CONTINUE，EXIT

-- 即使有异常发生，也会执行后面的语句
CONTINUE;

-- 有异常发生时，直接退出当前存储过程
EXIT;

--  声明SQLEXCEPTION处理器，当有SQLEXCEPTION发生时，错误标识符的值设为0
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SET result = 0;
```

示例：见上述存储过程示例的描述与问题。

```
DROP PROCEDURE IF EXISTS transferMoney;
-- 创建函数
DELIMITER //
CREATE PROCEDURE transferMoney(IN transferFrom BIGINT, IN transferTo BIGINT, IN money DOUBLE(20,3), OUT result TINYINT(1))
BEGIN
--  用于标记转账是否成功
--  DECLARE result TINYINT(1) DEFAULT 0;

-- 发生SQLEXCEPTION时的处理方式：CONTINUE，EXIT
-- CONTINUE表示即使有异常发生，也会执行后面的语句
-- EXIT表示，有异常发生时，直接退出当前存储过程
--  声明SQLEXCEPTION处理器，当有SQLEXCEPTION发生时，错误标识符的值设为0
```

```
13      DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SET
    result = 0;
14  --  开启事务
15      START TRANSACTION;
16      UPDATE account SET balance = balance - money WHERE
    account=transferFrom AND balance >= money;
17      IF ROW_COUNT() = 1 THEN
18          UPDATE account SET balance = balance + money
    WHERE account=transferTo;
19          IF ROW_COUNT() = 1 THEN
20              SET result = 1;
21          ELSE SET result = 0;
22          END IF;
23          ELSE SET result = 0;
24      END IF;
25  --  回滚事务
26      IF result = 0 THEN ROLLBACK;
27  --  提交事务
28      ELSE COMMIT;
29      END IF;
30  --  查询结果
31  --  SELECT result;
32  END //
33  DELIMITER ;
34  -- 调用函数
35  -- 将输出结果返回给变量@rs
36  CALL transferMoney(123456, 123457, 2000, @rs);
37  SELECT @rs;
```

（4）自定义函数

1. 函数与自定义函数的概念

函数：函数就是在大型数据库系统中，一组为了完成特定功能而存储在数据库中的 SQL 语句集，一次编译后永久有效。

自定义函数：MySQL 本身提供了一些内置函数，这些函数给我们日常的开发和数据操作带来了很大的便利，比如聚合函数 `SUM()`、`AVG()` 以及日期时间函数等。但这并不能完全满足开发的需要，有时我们需要一个函数来完成一些复杂功能的实现，而 MySQL 中又没有这样的函数，因此，我们需要自定义函数来实现。

2. 自定义函数的使用

自定义函数的基本语法：

```
-- 声明分隔符
[DELIMITER $$]

CREATE FUNCTION 函数名称 ([参数名1 数据类型, 参数名2 数据类型, ..., 参数名n 数据类型])
RETURNS 数据类型
DETERMINISTIC | NO SQL | READS SQL DATA | MODIFIES SQL DATA | CONTAINS SQL
-- 函数特征：
-- DETERMINISTIC：不确定
-- NO SQL：没有SQL语句
-- READS SQL DATA：只读数据，不会修改数据
-- MODIFIES SQL DATA：需要修改数据
-- CONTAINS SQL：包含SQL语句

-- 语句块开始
BEGIN
    -- SQL 语句集
    RETURN 结果;
-- 语句块结束
END [$$]

-- 还原分隔符
[DELIMITER ;]
```

示例：使用函数实现求score表中的成绩最大差值

```sql
-- 使用函数实现求score表中的成绩最大差值
SELECT MAX(score) - MIN(score) FROM score;

DELIMITER //
CREATE FUNCTION getMaxDiff()
RETURNS DOUBLE(5,2)
READS SQL DATA
BEGIN
    RETURN (SELECT MAX(score) - MIN(score) FROM score);
END //
DELIMITER ;
SELECT getMaxDiff();
```

循环结构语法：

```sql
-- WHILE 循环
WHILE 循环条件 DO
    -- SQL 语句集
END WHILE;

-- REPEAT 循环（类似于do while循环）
REPEAT
    -- SQL 语句集
UNTIL 循环终止条件 END REPEAT;

-- LOOP 标号循环
标号: LOOP
        -- SQL 语句集
        IF 循环终止条件 THEN LEAVE 标号;
        END IF;
    END LOOP;
```

示例1：使用函数实现求0~给定的任意整数的累加和

```sql
-- WHILE循环
DELIMITER //
CREATE FUNCTION getTotal1(maxNum INT(11))
RETURNS INT(11)
NO SQL
BEGIN
```

```
    DECLARE total INT(11) DEFAULT 0;
    DECLARE i INT(11) DEFAULT 0;
    WHILE i <= maxNum DO
        SET total = total + i;
        SET i = i + 1;
    END WHILE;
    RETURN total;
END //
DELIMITER ;
SELECT getTotal1(100);

-- REPEAT 循环
DELIMITER //
CREATE FUNCTION getTotal2(maxNum INT(11))
RETURNS INT(11)
NO SQL
BEGIN
    DECLARE total INT(11) DEFAULT 0;
    DECLARE i INT(11) DEFAULT 0;
    REPEAT
        SET total = total + i;
        SET i = i + 1;
    UNTIL i > maxNum END REPEAT;
    RETURN total;
END //
DELIMITER ;
SELECT getTotal2(100);

-- LOOP 标号
DROP FUNCTION IF EXISTS getTotal3;
DELIMITER //
CREATE FUNCTION getTotal3(maxNum INT(11))
RETURNS INT(11)
NO SQL
BEGIN
    DECLARE total INT(11) DEFAULT 0;
    DECLARE i INT(11) DEFAULT 0;
    a: LOOP
        SET total = total + i;
        SET i = i + 1;
        IF i > maxNum THEN LEAVE a;
```

```
        END IF;
    END LOOP;
    RETURN total;
END //
DELIMITER ;
SELECT getTotal3(100);
```

示例2：使用函数实现生成一个指定长度的随机字符串

```
DELIMITER //
CREATE FUNCTION randomStr(len INT(11))
RETURNS VARCHAR(255)
NO SQL
BEGIN
    DECLARE s VARCHAR(50) DEFAULT
'abcdefghigklmnopqrstuvwxyz0123456789';
    DECLARE rs VARCHAR(255) DEFAULT '';
    DECLARE i INT(11) DEFAULT 0;
    DECLARE position INT(11);
    WHILE i < len DO
        SELECT ROUND(RAND() * 36) INTO position;
        SET rs = CONCAT(rs,SUBSTRING(s, position, 1));
        SET i = i + 1;
  END WHILE;
    RETURN rs;
END //
DELIMITER ;
SELECT randomStr(10);
```

（5）触发器

1. 触发器的概念

触发器（trigger）是用来保证数据完整性的一种方法，由事件来触发，比如当对一个表进行增删改操作时就会被激活执行。经常用于加强数据的完整性约束和业务规则。

2. 触发器的类型

触发器类型	NOW和OLD的使用
INSERT 触发器	NEW 表示新增的数据
UPDATE 触发器	OLD 表示修改前的数据，NEW 表示修改后的数据
DELETE 触发器	OLD 表示被删除的数据

3. 触发器的使用

触发器语法：

```
-- 删除触发器
DROP TIGGER [IF EXISTS] 触发器名称;
-- 创建触发器
-- 触发时机为BEFORE或AFTER，即在某事件前触发还是之后触发
-- 触发事件，为INSTER，UPDATE或DELETE
[DELIMITER $$]
CREATE TRIGGER 触发器名称 BEFORE|AFTER
INSERT|UPDATE|DELETE ON 表名 FOR EACH ROW
BEGIN
-- 执行的SQL操作
END [$$]
DELIMITER ;
```

以下示例均以 `lesson` 表导入的数据为例。

示例1：现有商品表goods和订单表order，每一个订单的生成都意味着商品数量的减少，请使用触发器完成这一过程。

```
-- region 区域
-- agent 代理商
-- sales 销售人员
-- order 订单
-- goods 货物
-- 删除触发器
DROP TRIGGER IF EXISTS addOreder;
-- 创建触发器
DELIMITER //
CREATE TRIGGER addOrder AFTER INSERT ON `order` FOR
EACH ROW
BEGIN
```

```
-- 新货物数量 = 原数量 - 新增订单购买数
    UPDATE goods SET number = number - NEW.sale_count
WHERE id=NEW.goods_id;
END //
DELIMITER ;

INSERT INTO `order` (`goods_id`, `sales_id`,
`sale_count`, `created_time`, `state`) VALUES (1, 1,
6, '2024-08-16', 1);
```

示例2：现有商品表goods和订单order，每一个订单的取消都意味着商品数量的增加，请使用触发器完成这一过程。

```
-- 删除触发器
DROP TRIGGER IF EXISTS deleteOreder;
-- 创建触发器
DELIMITER //
CREATE TRIGGER deleteOrder AFTER DELETE ON `order` FOR
EACH ROW
BEGIN
-- 新货物数量 = 原数量 + 旧订单（被取消订单）购买数
    UPDATE goods SET number = number + OLD.sale_count
WHERE id=OLD.goods_id;
END //
DELIMITER ;

DELETE FROM `order` WHERE id=350003;
```

示例3：现有商品表goods和订单表order，每一个订单购买数量的更新都意味着商品数量的变动，请使用触发器完成这一过程。

```
-- 删除触发器
DROP TRIGGER IF EXISTS updateOreder;
-- 创建触发器
DELIMITER //
CREATE TRIGGER updateOrder AFTER UPDATE ON `order` FOR
EACH ROW
BEGIN
    DECLARE changeNum INT(11) DEFAULT 0;
--  差值 = 新订单购买数量 - 原订单购买数量（可能为负数）
    SET changeNum = NEW.sale_count - OLD.sale_count;
```

```
--  新数量 = 原数量 - 差值
--  这里用NEW或OLD都可以，因为订单编号不变
    UPDATE goods SET number = number - changeNum WHERE id=OLD.goods_id;
END //
DELIMITER ;

UPDATE `order` SET sale_count = sale_count + 2 WHERE id=20;
UPDATE `order` SET sale_count = sale_count - 4 WHERE id=20;
```

（6）视图

1. 视图的概念

视图是一张虚拟表，本身并不存储数据，当 SQL 操作视图时所有数据都是从其他表中查出来。

实际上，查询得到的数据表就是视图，子查询就是在视图里进行查询。

2. 视图的使用

视图语法：

```
-- 创建视图
CREATE VIEW 视图名 AS SELECT 列1[,列2,...] FROM 表名 WHERE 条件;
-- 创建或更新视图（即可以创建也可以更新，功能包含上面的语句）
CREATE OR REPLACE VIEW 视图名 AS SELECT 列1[,列2,...] FROM 表名 WHERE 条件;
-- 删除视图
DROP VIEW IF EXISTS 视图名;
```

3. 视图的作用与示例

视图并不能提升查询速度，只是方便了业务开发，但同时也加大了数据库服务器的压力，因此，需 要合理的使用视图。

- 定制用户数据，聚焦特定的数据。

示例1：如果频繁获取销售人员编号、姓名和代理商名称，可以创建视图。

```sql
-- 删除视图
DROP VIEW IF EXISTS salesInfo;
-- 创建或更新视图
CREATE OR REPLACE VIEW salesInfo AS
SELECT
    a.id,
    a.`name` saleName,
    b.`name` agentName
FROM
    sales a,
    agent b
WHERE
    a.agent_id = b.id;
-- 测试代码
SELECT id, saleName FROM salesInfo;
```

- 简化数据操作。例如，进行关联查询时，涉及到的表可能会很多，这时写的 SQL 语句可能会很长，如果这 个动作频繁发生的话，可以创建视图。

示例2：

```sql
-- 删除视图
DROP VIEW IF EXISTS searchOrderDetail;
-- 创建视图
CREATE OR REPLACE VIEW searchOrderDetail AS
SELECT
    a.id regionId,
    a.`name` regionName,
    b.id agentId,
    b.`name` agentName,
    c.id saleId,
    c.`name` saleName,
    d.sale_count saleCount,
    d.created_time createdTime,
    e.`name` goodsName
FROM
    region a,
    agent b,
    sales c,
    `order` d,
```

```
    goods e
WHERE
    a.id = b.region_id
AND b.id = c.agent_id
AND c.id = d.sales_id
AND d.goods_id = e.id;
-- 测试代码
SELECT *FROM searchOrderDetail;
```

- 提高安全性能。例如，用户密码属于隐私数据，用户不能直接查看密码。可以使用视图过滤掉这一字段

示例3：

```
-- 删除视图
DROP VIEW IF EXISTS userInfo;
-- 创建视图
CREATE OR REPLACE VIEW userInfo AS
SELECT
    username,
    salt,
    failure_times,
    last_log_time
FROM
    `user`
-- 测试代码（会报错）
-- SELECT `password` FROM userInfo;
```

六、数据库设计

（1）设计数据库

1. 实体

实体就是软件开发过程中所涉及到的事物，通常都是一类数据对象的个体。

2. 数据库设计

数据库设计就是将实体与实体之间的关系进行规划和结构化的过程。

设计数据库的作用：

当存储的数据比较少的时候，当然不需要对数据库进行设计。但是，当对数据的需求量越来越大时，对数据库的设计就很有必要性了！如果数据库的设计不当，会造成数据冗余、修改复杂、操作数据异常等问题。而好的数据库设计，则可以减少不必要的数据冗余，通过合理的数据规划提高系统的性能。

3. 设计数据库的方法

1. **收集信息**

   在确定客户要做什么之后，收集一切相关的信息，尽量不遗漏任何信息。

2. **标识实体**

   实体一般是名词，每个实体只描述一件事情，不能重复出现含义相同的实体。

3. **标识实体的详细属性**

   标识每个实体需要存储的详细信息。

4. **标识实体之间的关系**

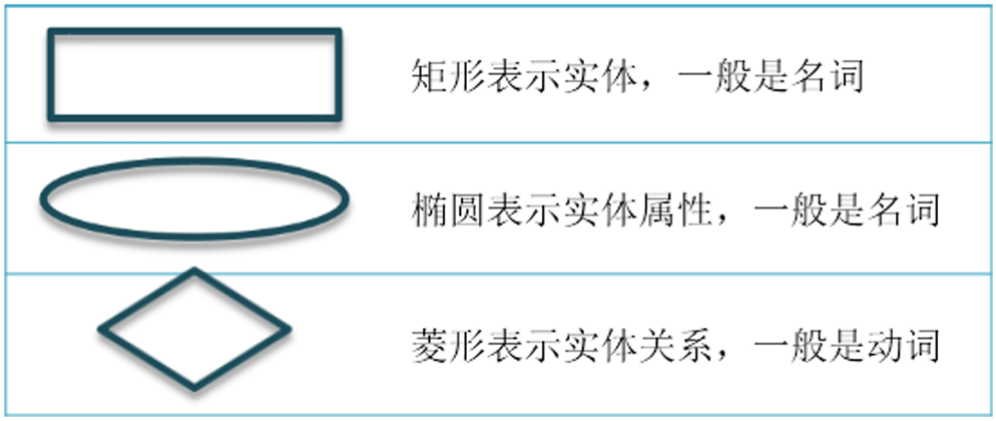
   理清实体与实体之间的关系。

（2）ER 图

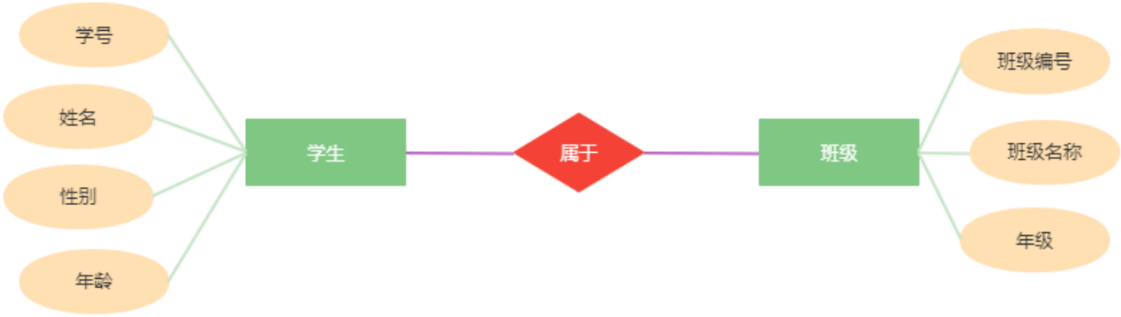
1. ER 图的概念

ER（Entity Relational）图：实体关系图。通常是整个体系的实体关系图。

2. 绘制 ER 图的方法



示例：见表与表的关系



（3）数据库模型图

1. 关系模式

实体关系的描述称为关系模式，关系模式通常使用二维表的形式表示。

示例：

学生（学号，姓名，性别，年龄，所属班级）

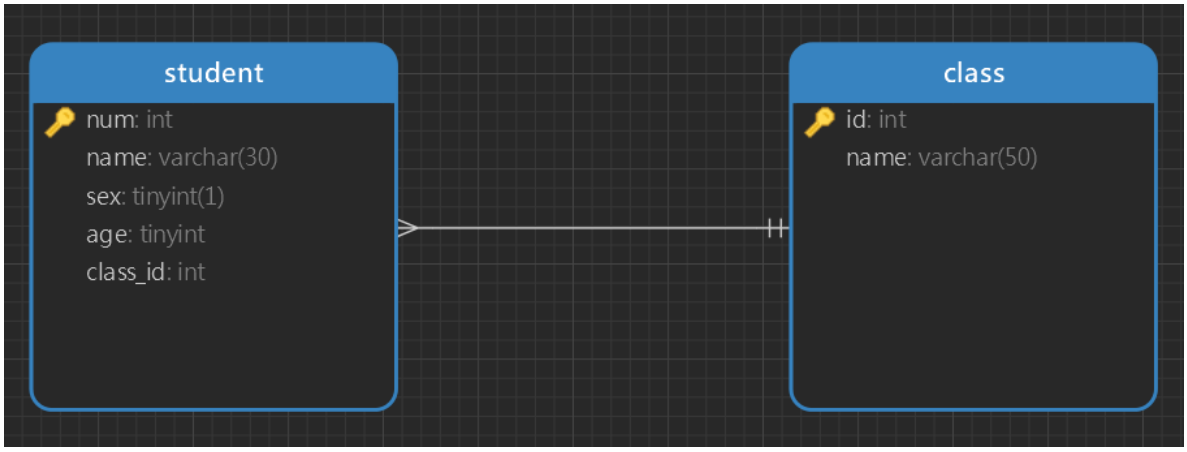
班级（班级编号， 班级名称）

2. 关系模式转换数据库模型图

将关系模式使用Navicat工具转换为数据库模型图，转换步骤如下：

- 将各实体转换为对应的表，将各属性转换为各表对应的列；
- 标识每个表的主键列；
- 在表之间建立主外键，体现实体。

示例：



（4）数据库三大范式

1. 第一范式

第一范式是最基本的范式，确保每列保持原子性，也就是每列不可再分。

示例：

address
四川省成都市
陕西省西安市
广东省广州市
甘肃省兰州市

province	city
四川省	成都市
陕西省	西安市
广东省	广州市
甘肃省	兰州市

左侧表不满足第一范式，因为该列可以再分，右侧表满足第一范式。

2. 第二范式

第二范式是在第一范式的基础上，每张表的属性完全依赖于主键，也就是每张表只描述一件事情。

示例：

学生表

id	name	sex	age	class
1	张华	男	20	计科1班
2	金凤	女	22	计科2班
3	李刚	男	21	软工1班
4	龙强	男	24	软工2班

班级表

id	name
1	计科1班
2	计科2班
3	软工1班
4	软工2班

学生表

id	name	sex	age	class_id
1	张华	男	20	1
2	金凤	女	22	2
3	李刚	男	21	3
4	龙强	男	24	4

上方的表，班级列不依赖于主键id，不满足第二范式。下方拆成两个表则满足第二范式。

3. 第三范式

第三范式是在第二范式的基础上，确保每列都直接依赖于主键，而不是间接依赖于主键，也就是不能存在传递依赖。比如A依赖于B，B依赖于C，这样A就间接依赖于C。

示例：

学生表

id	name	sex	age	major	tuition
1	张华	男	20	计算机专业	8000
2	金凤	女	22	软件工程专业	9000
3	李刚	男	21	电子商务专业	7000
4	龙强	男	24	临床医学专业	16000

专业表

id	name	tuition
1	计算机专业	8000
2	软件工程专业	9000
3	电子商务专业	7000
4	临床医学专业	16000

学生表

id	name	sex	age	major_id
1	张华	男	20	1
2	金凤	女	22	2
3	李刚	男	21	3
4	龙强	男	24	4

上方表中，专业列依赖于id，学费列依赖于专业列，不满足第二范式和第三范式。下方拆成两个表则满足第三范式。

练习：

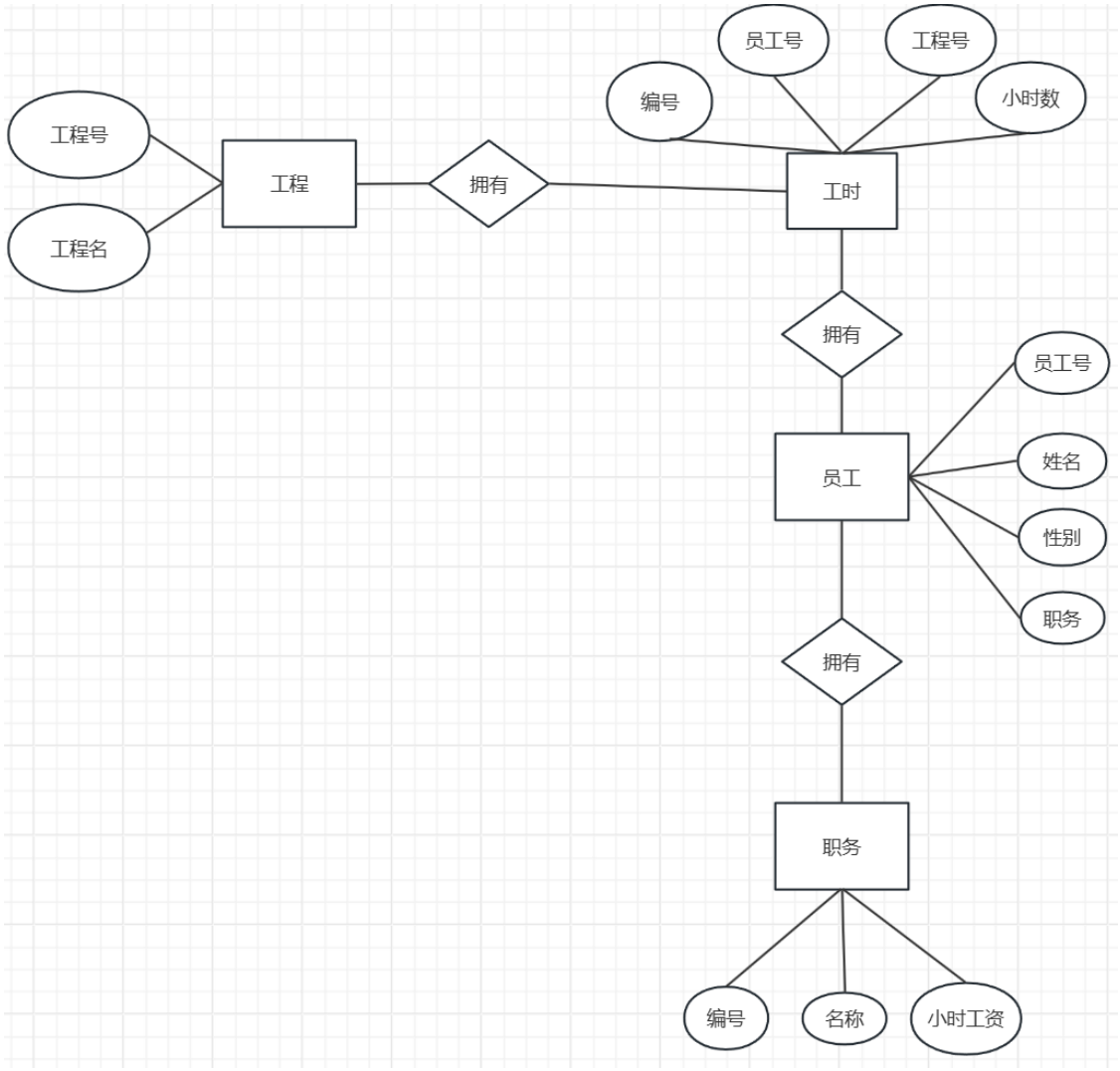
假设某建筑公司要设计一个数据库。公司的业务规则概括说明如下：

- 公司承担多个工程项目，每一项工程有：工程号、工程名称、施工人员等；
- 公司有多名职工，每一名职工有：职工号、姓名、性别、职务（工程师、技术员）等；
- 公司按照工时和小时工资率支付工资，小时工资率由职工的职务决定（例如，技术员的小时工资率与工程师不同）

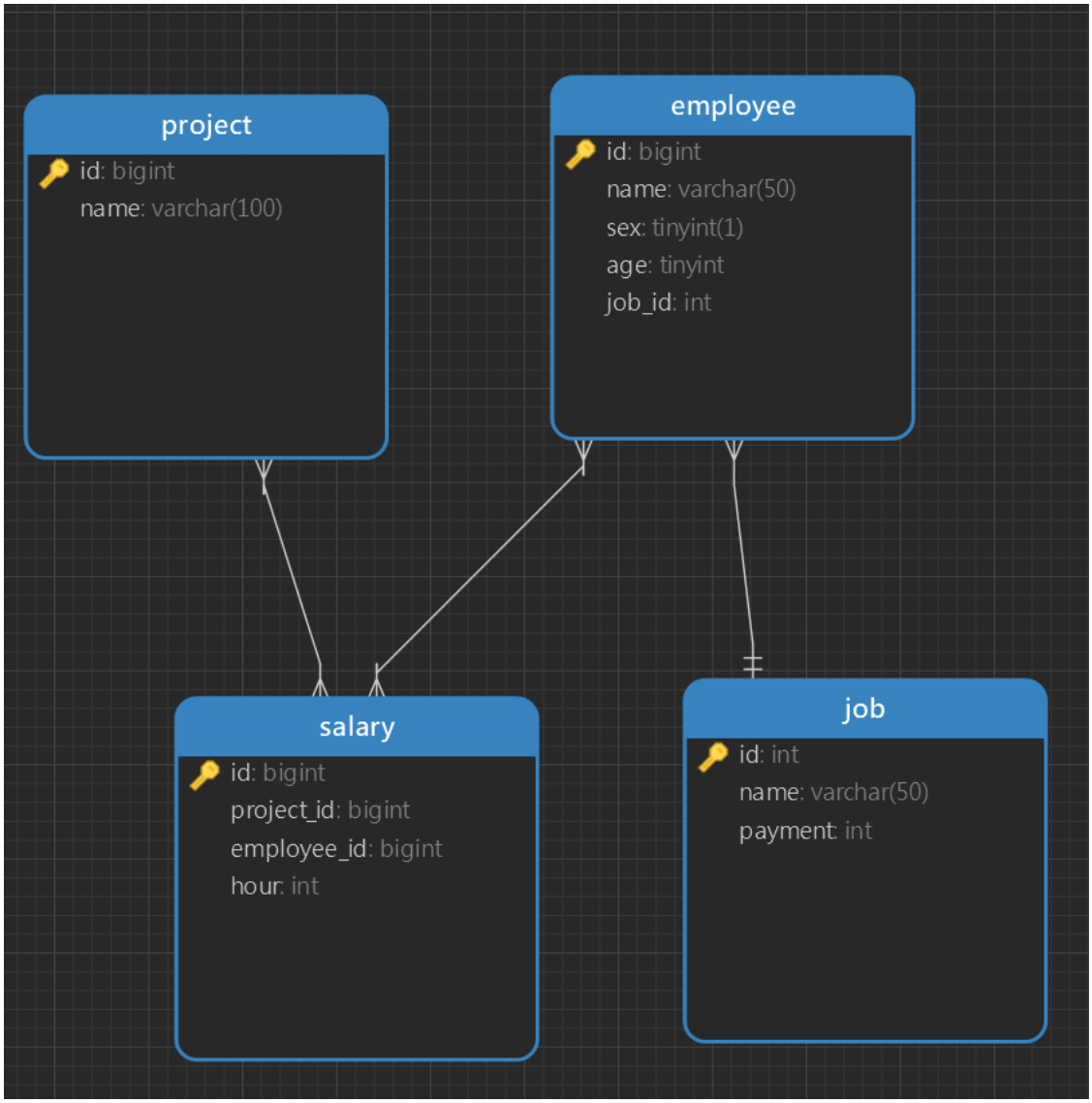
分析：

1. 找出实体（工程、员工、职务、工时）

2. 找出实体关系，并绘制ER图；



3. 将ER图转换为数据库模型图；



4. 使用三大范式规范数据库设计。

4. 三大范式使用注意事项

在实际开发过程中，为了满足性能的需要，数据库的设计可能会打破数据库三大范式的约束。

以空间换时间：当数据库中存储的数据越来越多时，查询效率会下降，为了提升了查询效率，可能会在表中增加新的字段，此时，数据库的设计就不再满足三大范式。

七、JDBC

（1）JDBC

1. JDBC 的概念

JDBC (Java Database Connection) 是 Java 数据库连接技术的简称，提供连接数据库的能力。

2. JDBC API

Java 作为目前世界上最流行的高级开发语言，当然不可能考虑去实现各种数据库的连接与操作。但 Java 语言的开发者对数据库的连接与操作提供了相关的接口，供各大数据库厂商去实现。这些接口位于 `java.sql` 包中。

`Driver` 驱动：

`java.sql.Driver`：数据库厂商提供的 JDBC 驱动包中必须包含该接口的实现，该接口中就包含连接数据库的功能。

```
//根据给定的数据库url地址连接数据库
Connection connect(String url, java.util.Properties
info) throws SQLException;
```

`DriverManager` 驱动管理器：

`java.sql.DriverManager`：数据库厂商的提供的 JDBC 驱动交给 `DriverManager` 来管理， `DriverManager` 主要负责获取数据库连接对象 `Connection`。

```
 //通过给定的账号、密码和数据库地址获取一个连接
public static Connection getConnection(String url,
String user,
                                       String password)
throws SQLException
```

`Connection` 连接：

`java.sql.Connection`：连接接口，数据库厂商提供的 JDBC 驱动包中必须包含该接口的实现，该接口主要提供与数据库的交互功能。

```
//创建一个SQL语句执行对象
Statement createStatement() throws SQLException;

 //创建一个预处理SQL语句执行对象
PreparedStatement prepareStatement(String sql) throws
SQLException;

 //创建一个存储过程SQL语句执行对象
CallableStatement prepareCall(String sql) throws
SQLException;
```

```
9
10   //设置该连接上的所有操作是否执行自动提交
11  void setAutoCommit(boolean autoCommit) throws
    SQLException;
12
13   //提交该连接上至上次提交以来所作出的所有更改
14  void commit() throws SQLException;
15
16   //回滚事务，数据库回滚到原来的状态
17  void rollback() throws SQLException;
18
19   //关闭连接
20  void close() throws SQLException;
21
22   //设置事务隔离级别
23  void setTransactionIsolation(int level) throws
    SQLException;
24  //不支持事务
25  int TRANSACTION_NONE             = 0;
26   //读取未提交的数据
27  int TRANSACTION_READ_UNCOMMITTED = 1;
28   //读取已提交的数据
29  int TRANSACTION_READ_COMMITTED   = 2;
30   //可重复读
31  int TRANSACTION_REPEATABLE_READ  = 4;
32   //串行化
33  int TRANSACTION_SERIALIZABLE     = 8;
```

`Statement` 执行器：

java.sql.Statement： SQL语句执行接口，数据库厂商提供的 JDBC 驱动包中必须包含该接口的实现，该接口主要提供执行数据库厂商提供的 SQL 语句的功能。

```
1  //执行查询，得到一个结果集
2  ResultSet executeQuery(String sql) throws
   SQLException;
3
4   //执行更新，得到受影响的行数
5  int executeUpdate(String sql) throws SQLException;
6
```

```
 //关闭SQL语句执行器
void close() throws SQLException;

 //将SQL语句添加到批处理执行SQL列表中
void addBatch( String sql ) throws SQLException;

 //执行批处理，返回列表中每一条SQL语句的执行结果
int[] executeBatch() throws SQLException;
```

`ResultSet` 结果集：

`java.sql.ResultSet`：查询结果集接口，数据库厂商提供的 JDBC 驱动包中必须包含该接口的实现，该接口主要提供查询结果的获取功能。

```
//光标从当前位置（默认位置位为0）向前移动一行，如果存在数据，则返回true，否则返回false
 boolean next() throws SQLException;

//光标从当前位置（默认位置位为0）向后移动一行，如果存在数据，则返回true，否则返回false
boolean previous() throws SQLException;

//关闭结果集
void close() throws SQLException;

//获取指定列的字符串值，传参可以是索引也可以是列名
String getString(int columnIndex) throws SQLException;
String getString(String columnName) throws SQLException;

//获取指定列的布尔值，传参可以是索引也可以是列名
boolean getBoolean(int columnIndex) throws SQLException;
boolean getBoolean(String columnName) throws SQLException;

//获取指定列的整数值，传参可以是索引也可以是列名
int getInt(int columnIndex) throws SQLException;
int getInt(String columnName) throws SQLException;

//获取指定列的对象，传参可以是索引也可以是列名
```

```
23 Object getObject(int columnIndex, Class type) throws
   SQLException;
24 Object getObject(String columnName, Class type) throws
   SQLException;
25
26 //获取结果集元数据：查询结果的列名称、列数量、列别名等等
27 ResultSetMetaData getMetaData() throws SQLException;
```

3. JDBC 的操作步骤

1. **引入驱动包**

   新建工程后，将 `mysql-connector-java.jar` 引入工程中。

2. **加载驱动**

```
//MySQL 5.0
 //Class.forName("com.mysql.jdbc.Driver");
 //MySQL 8.0
 Class.forName("com.mysql.cj.jdbc.Driver");
```

3. **获取连接**

```
Connection connection =
DriverManager.getConnection(url, username,
password);
```

4. **在连接上创建 SQL 语句执行器**

```
Statement statement = connection.createStatement();
```

5. **执行 SQL 语句**

```
//使用执行器查询并得到一个结果集
ResultSet rs = statement.executeQuery(sql);
 while(rs.next()){
 //获取列信息
}

 //更新
int affectedRows = statement.executeUpdate();
```

6. **释放资源**

```
rs.close();
statement.close();
connection.close();
```

示例:

```
package com.ssh.jdbc;

public class Account {

    private String account;

    private double balance;

    private int state;

    public Account(String account, double balance, int state) {
        this.account = account;
        this.balance = balance;
        this.state = state;
    }

    public Account() {

    }

    public String getAccount() {
        return account;
    }

    public void setAccount(String account) {
        this.account = account;
    }

    public double getBalance() {
        return balance;
    }

    public void setBalance(double balance) {
        this.balance = balance;
```

```
    }

    public int getState() {
        return state;
    }

    public void setState(int state) {
        this.state = state;
    }

    @Override
    public String toString() {
        return "Account{" +
                "account='" + account + '\'' +
                ", balance=" + balance +
                ", state=" + state +
                '}';
    }
}
```

```
package com.ssh.jdbc;

import java.sql.*;
import java.util.ArrayList;
import java.util.List;

public class JdbcTest {

    public static void main(String[] args) {
        //使用jdbc连接技术
        //mysql://localhost:3306 使用的是mysql数据库协议访问本地计算机3306端口
        String url = "jdbc:mysql://localhost:3306/lesson?serverTimezone=Asia/Shanghai";
        String username = "root";
        String password = "root";
        List<Account> accounts = new ArrayList<Account>();
        try {
            //加载驱动
```

```
            Class.forName("com.mysql.cj.jdbc.Driver");
            //获取连接
            Connection conn =
DriverManager.getConnection(url, username, password);
            //在连接上创建SQL语句执行器
            Statement s = conn.createStatement();

            //SQL语句
            String updateSql = "UPDATE account SET
balance = balance + 1000 WHERE account = 123456";
            //执行更新时返回的是受影响的行数
            int affectedRows =
s.executeUpdate(updateSql);
            System.out.println(affectedRows);

            //SQL语句
            String sql = "SELECT account,balance,state
FROM account";
            //使用执行器得到一个结果集
            ResultSet rs = s.executeQuery(sql);
            //循环执行SQL语句并收集结果
            while (rs.next()) { //光标向下移动
                //通过列名获取列的值
                String account =
rs.getString("account");
                double balance = rs.getDouble(2);
                int state = rs.getInt("state");
                Account account1 = new
Account(account, balance, state);
                accounts.add(account1);
            }
            //关闭结果集
//              rs.close();
            //关闭执行器
            s.close();
            //关闭连接
            conn.close();
        } catch (ClassNotFoundException e) {
            throw new RuntimeException(e);
        } catch (SQLException e) {
            throw new RuntimeException(e);
```

```
        }
        accounts.forEach(System.out::println);
    }
}
```

4. 预处理 SQL

在日常开发中，我们经常会根据用户输入的信息从数据库中进行数据筛选，对于 `goods` 表有如下操作：

```
package com.ssh.jdbc;

import java.sql.*;
import java.util.Scanner;

public class PrepareStatementTest {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("请输入商品名称：");
        String goodsName = sc.nextLine();
        goodsName = "'" + goodsName +"'";
        String url =
"jdbc:mysql://localhost:3306/lesson?
serverTimezone=Asia/Shanghai";
        String username = "root";
        String password = "root";
        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
            Connection conn =
DriverManager.getConnection(url, username, password);
            Statement s = conn.createStatement();
            String sql = "SELECT
id,name,number,price,agent_id FROM goods WHERE name =
" + goodsName + "LIMIT 0, 20";
            ResultSet rs = s.executeQuery(sql);
            while (rs.next()) {
                long id = rs.getLong("id");
                String name = rs.getString("name");
                int number = rs.getInt("number");
                double price = rs.getDouble("price");
```

```
                long agentId = rs.getLong("agent_id");
                System.out.println(id + " " + name + "
" + number + " " + price + " " + agentId);
            }
            rs.close();
            s.close();
            conn.close();
        } catch (ClassNotFoundException e) {
            throw new RuntimeException(e);
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
}
```

当用户输入 `小米10' or 1='1` 时，代码执行后，SQL 语句就变成了：

```
SELECT id,name,number,price,agent_id FROM goods WHERE
name = '小米10' or 1='1';
```

明显查询的结果发生了变化，这样的情况被称作为 SQL 注入。为了防止 SQL 注入，Java 提供了 `PreparedStatement` 接口对 SQL 进行预处理，该接口是 `Statement` 接口的子接口，其常用方法如下：

```
//获取PreparedStatement接口对象
PreparedStatement ps =
connection.prepareStatement(sql);

//执行查询，得到一个结果集
ResultSet executeQuery() throws SQLException;

 //执行更新，得到受影响的行数
int executeUpdate() throws SQLException;

 //使用给定的整数值设置给定位置的参数
void setInt(int parameterIndex, int x) throws
SQLException;

 //使用给定的长整数值设置给定位置的参数
void setLong(int parameterIndex, long x) throws
SQLException;
```

```
15
16  //使用给定的双精度浮点数值设置给定位置的参数
17 void setDouble(int parameterIndex, double x) throws
   SQLException;
18
19  //使用给定的字符串值设置给定位置的参数
20 void setString(int parameterIndex, String x) throws
   SQLException;
21
22  //使用给定的对象设置给定位置的参数
23 void setObject(int parameterIndex, Object x) throws
   SQLException;
24
25  //获取结果集元数据
26 ResultSetMetaData getMetaData() throws SQLException;
```

使用 `PreparedStatement` 时， SQL 语句中的参数一律使用 `?` 号来进行占位，然后通过调用 `setXxx()` 方法来对占位的 `?` 号进行替换。从而将参数作为一个整体进行查询。

更改后的示例：

```
1  package com.ssh.jdbc;
2
3  import java.sql.*;
4  import java.util.Scanner;
5
6  public class PrepareStatementTest {
7
8      public static void main(String[] args) {
9          Scanner sc = new Scanner(System.in);
10         System.out.println("请输入商品名称：");
11         String goodsName = sc.nextLine();
12         String url =
   "jdbc:mysql://localhost:3306/lesson?
   serverTimezone=Asia/Shanghai";
13         String username = "root";
14         String password = "root";
15         try {
16             Class.forName("com.mysql.cj.jdbc.Driver");
```

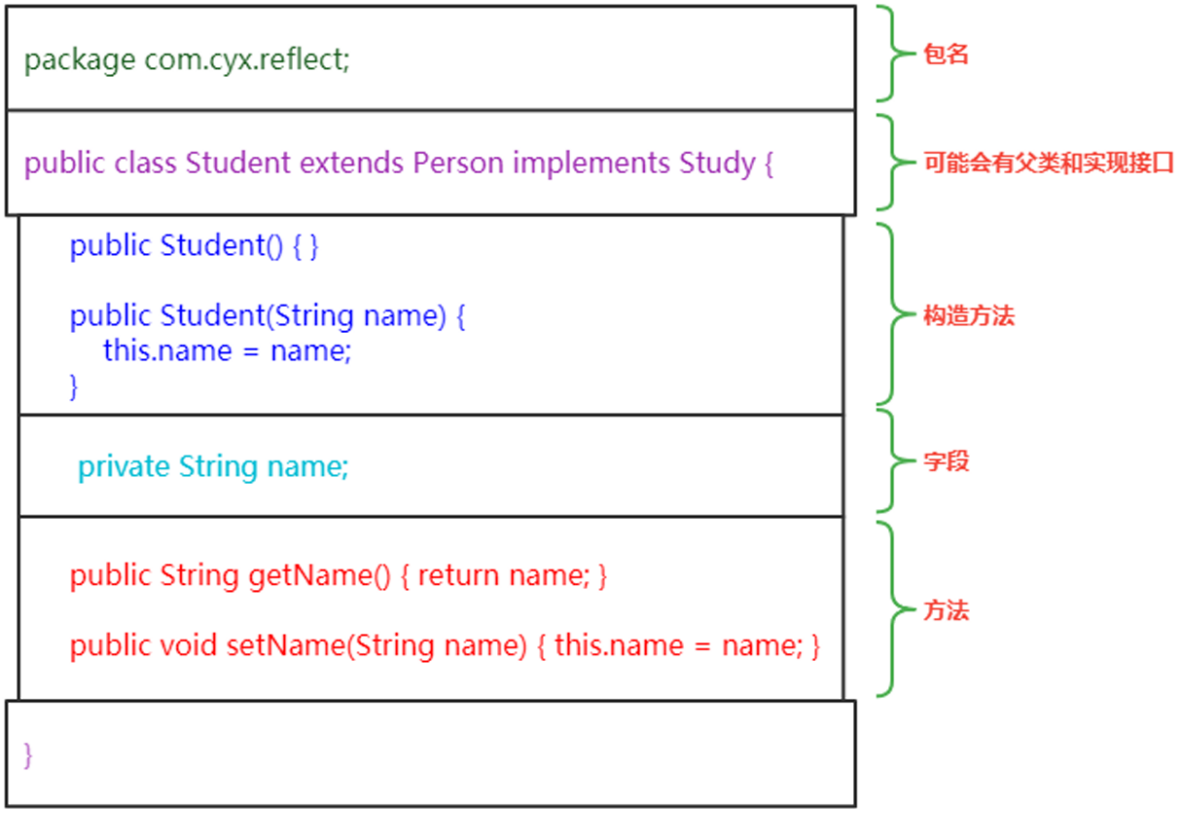
```
            Connection conn = DriverManager.getConnection(url, username, password);
            String sql = "SELECT id,name,number,price,agent_id FROM goods WHERE name = ? LIMIT 0, 20";
            //创建预处理执行器
            PreparedStatement ps = conn.prepareStatement(sql);
            //设置占位符替换的值
            ps.setString(1, goodsName);
            ResultSet rs = ps.executeQuery();
            while (rs.next()) {
                long id = rs.getLong("id");
                String name = rs.getString("name");
                int number = rs.getInt("number");
                double price = rs.getDouble("price");
                long agentId = rs.getLong("agent_id");
                System.out.println(id + " " + name + " " + number + " " + price + " " + agentId);
            }
            rs.close();
            ps.close();
            conn.close();
        } catch (ClassNotFoundException e) {
            throw new RuntimeException(e);
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
}
```

(2) 反射

1. `Class` 类的概述

我们编写的 Java 程序先经过编译器编译，生成 class 文件，而 class 文件的执行场所是在 JVM 中，那么 JVM 如何存储我们编写的类的信息?

一个类的组成部分：



定义一个类来描述所有共同特征：

```java
public class Class {
    private String name; //类名

    private Package pk; //包名

    private Constructor[] constructors; //构造方法，因为可能存在多个，所以使用数组

    private Field[] fields; //字段（成员变量），因为可能存在多个，所以使用数组

    private Method[] methods; //方法，因为可能存在多个，所以使用数组

    private Class<?> interfaces; //实现的接口，因为可能存在多个，所以使用数组

    private Class<?> superClass; //继承的父类

    //省略getter/setter
}
```

有 `Class` 对象的类型：

- 外部类，成员内部类，静态内部类，局部内部类，匿名内部类
- 接口
- 数组
- 枚举类型
- 注解
- 基本数据类型
- `void`

为什么要设计这样的类：

因为我们编写的程序从本质上来说也是文件， JVM 加载类的过程相当于对文件内容进行解析，解析内容就需要找到共有特征（`Class` 类定义），然后再将这特征（使用 `Class` 对象）存储起来，在使用的时候再取出来。通过 `Class` 对象反向推到我们编写的类的内容，然后再进行操作， 这个过程就称为反射。

在 JDK 中已经提供了这样的类：`java.lang.Class`，因此，我们不需要再来设计，只需要学习它即 可。

`Class` 类也是类，继承于 `Object`，该类的对象不是 `new` 出来的，而是由系统创建的，存放在堆中。

对于某个对象的 `Class` 的类对象，在内存中只有一份，因为类只加载一次。

每个类的实例都会记住自己是由哪个 `Class` 实例所生成。

类的字节码二进制数据，是放在方法区的，有的地方称为类的元数据（包括方法代码，变量名，方法名，访问权限等）。

2. `Class` 类的方法

获取 `Class` 类的方法：

```
Class<类名> clazz = 类名.class;

Class<类名> clazz = 对象名.getClass();

Class<类名> clazz = clazz.getSuperClass();

Class clazz = Class.forName("类的全限定名");  //类的全限定名=包名 + "." + 类名

Class clazz = 包装类.TYPE;

Class clazz = 数据类型.class;
```

`Class` 类的常用方法:

```
//获取当前Class对象的父类的Class对象
public native Class<? super T> getSuperclass();

//获取当前Class对象的接口
public Class<?>[] getInterfaces();

//获取该类的类加载器
public ClassLoader getClassLoader();

//获取类中使用public修饰的字段
public Field[] getFields() throws SecurityException;

//获取类中定义的所有字段
public Field[] getDeclaredFields() throws SecurityException;

//通过给定的字段名获取类中定义的字段
public Field getField(String name) throws NoSuchFieldException, SecurityException;

//获取类中使用public修饰的方法
public Method[] getMethods() throws SecurityException;

//获取类中定义的所有方法
public Method[] getDeclaredMethods() throws SecurityException;
```

```
24
25  //通过给定的方法名和参数列表类型获取类中定义的方法
26  public Method getDeclaredMethod(String name, Class<?
    >... parameterTypes) throws NoSuchMethodException,
    SecurityException;
27
28  //获取类中使用public修饰的构造方法
29  public Constructor<?>[] getConstructors() throws
    SecurityException;
30
31  //通过给定的参数列表类型获取类中定义的构造方法
32  public Constructor<T> getConstructor(Class<?>...
    parameterTypes) throws NoSuchMethodException,
    SecurityException;
33
34  //获取类的全限定名
35  public String getName();
36
37  //获取类所在的包
38  public Package getPackage();
39
40  //判断该类是否是基本数据类型
41  public native boolean isPrimitive();
42
43  //判断该类是否是接口
44  public native boolean isInterface();
45
46  //判断该类是否是数组
47  public native boolean isArray();
48
49  //通过类的无参构造创建一个实例
50  public T newInstance() throws InstantiationException,
    IllegalAccessException;
51
52  java.lang.reflect.AccessibleObject
53  //修改访问权限，开启或禁用访问安全检查的开关
54  //参数值为true表示反射的对象在使用时取消访问检查，提高访问效率，
    参数值为false表示反射的对象执行访问检查，默认为false
55  public void setAccessible(boolean flag) throws
    SecurityException;
56
```

```
//描述特征的类的方法
//获取属性值
public Object get(Object obj);

//获取属性名
public String getName();

//设置属性值
public void set(Object obj, Object value);
```

示例:

```
package com.ssh.jdbc.reflection;

import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.Arrays;

/**
 * @author 我
 * @version 1.0
 */
public class ReflectionTest {

    public static void main(String[] args) {
        //构建一个学生对象，并为每个字段赋值
        Class<Student> clazz = Student.class;
        try {
            Constructor<? extends Student> c =
clazz.getDeclaredConstructor();
            //无参构造是私有方法，因此需要先修改访问权限
            c.setAccessible(true);
            Student s = c.newInstance();
            Field nameFiled =
clazz.getDeclaredField("name");
            nameFiled.setAccessible(true);
            //给指定的对象的该字段赋值
            nameFiled.set(s, "李四");
```

```
            Field ageFiled = clazz.getDeclaredField("age");
            ageFiled.setAccessible(true);
            ageFiled.set(s, 20);

            //组装方法名
            String fieldName = nameFiled.getName();
            String methodName = "get" + fieldName.substring(0, 1).toUpperCase() + fieldName.substring(1);
            //通过方法名来获取方法
            Method m = clazz.getDeclaredMethod(methodName);
            //方法是属于成员的，不能直接调用，所以通过反射来调用方法
            String name = (String) m.invoke(s);
            System.out.println(name);
            System.out.println(s);

            methodName = "set" + fieldName.substring(0, 1).toUpperCase() + fieldName.substring(1);
            m = clazz.getDeclaredMethod(methodName, nameFiled.getType());
            //反射调用set方法来修改字段值
            m.invoke(s, "王五");
            System.out.println(s);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /**
     * 获取方法
     */
    private static void getMethod(){
        Class<Student> clazz = Student.class;
        //获取所有定义的方法
        Method[] methods = clazz.getDeclaredMethods();
        for (Method method : methods) {
```

```
            System.out.print(method.getModifiers() + " " + method.getName() + " (");
            Class[] types = method.getParameterTypes();
            for (Class c : types) {
                System.out.print(c.getName() + ",");
            }
            System.out.println(")");
        }

 System.out.println("================================");

        //获取所有定义的方法以及继承与它们的方法
        methods = clazz.getMethods();
        for (Method method : methods) {
            System.out.print(method.getModifiers() + " " + method.getName() + " (");
            Class[] types = method.getParameterTypes();
            for (Class c : types) {
                System.out.print(c.getName() + ",");
            }
            System.out.println(")");
        }

 System.out.println("================================");

        //根据方法参数列表获取方法
        try {
            Method m = clazz.getDeclaredMethod("setName", String.class);
            System.out.print(m.getModifiers() + " " + m.getName() + " (");
            Class[] types = m.getParameterTypes();
            for (Class c : types) {
                System.out.print(c.getName() + ",");
            }
            System.out.println(")");
        } catch (NoSuchMethodException e) {
```

```
            throw new RuntimeException(e);
        }
    }

    /**
     * 获取字段
     */
    private static void getField() {
        Class<Student> clazz = Student.class;
        //获取所有字段
        Field[] fields = clazz.getDeclaredFields();
        for (Field f : fields) {
            System.out.println(f.getModifiers() + " "
+ f.getType() + " " + f.getName());
        }
        //获取公开的字段
        fields = clazz.getFields();
        for (Field f : fields) {
            System.out.println(f.getModifiers() + " "
+ f.getType() + " " + f.getName());
        }
        //通过字段名获取字段
        try {
            Field f = clazz.getDeclaredField("name");
            System.out.println(f.getModifiers() + " "
+ f.getType() + " " + f.getName());
        } catch (NoSuchFieldException e) {
            throw new RuntimeException(e);
        }
    }

    /**
     * 获取构造方法
     */
    private static void getConstructor() {
        Class<Student> clazz = Student.class;
        //获取在类中定义的所有构造方法
        Constructor[] constructors =
clazz.getDeclaredConstructors();
        for (Constructor c : constructors) {
```

```
126             System.out.println(c.getModifiers());
    //访问修饰符用数字来表示：1为public。2为private
127             String name = c.getName();  //构造方法的名字
128             Class[] types = c.getParameterTypes();
    //构造方法中参数的数据类型
129             System.out.print(name + " ");
130 
    System.out.println(Arrays.toString(types));
131         }
132 
    System.out.println("==========================
    ==");
133         //获取用public修饰的构造方法
134         constructors = clazz.getConstructors();
135         for (Constructor c : constructors) {
136             System.out.println(c.getModifiers());
    //访问修饰符用数字来表示
137             String name = c.getName();  //构造方法的名字
138             Class[] types = c.getParameterTypes();
    //构造方法中参数的数据类型
139             System.out.print(name + " ");
140 
    System.out.println(Arrays.toString(types));
141         }
142 
    System.out.println("==========================
    ==");
143         //根据参数类型获取用public修饰的构造方法
144         try {
145             Constructor c =
    clazz.getConstructor(String.class, int.class);
146             System.out.println(c.getModifiers());
    //访问修饰符用数字来表示
147             String name = c.getName();  //构造方法的名字
148             Class[] types = c.getParameterTypes();
    //构造方法中参数的数据类型
149             System.out.print(name + " ");
150 
    System.out.println(Arrays.toString(types));
151         } catch (NoSuchMethodException e) {
152             throw new RuntimeException(e);
```

```
153             }
154         }
155 
156         /**
157          * 获取Class类
158          */
159         private static void getClazz(){
160             Class<Student> c1 = Student.class;
161             System.out.println(c1.getName());
162             Student stu = new Student("张三", 20);
163             Class<? extends Student> c2 = stu.getClass();
164             System.out.println(c2.getName());
165             //获取父类
166             Class<? super Student> c3 =
    c1.getSuperclass();
167             System.out.println(c3.getName());
168             try {
169                 Class c4 =
    Class.forName("com.ssh.jdbc.reflection.Student");
170                 System.out.println(c4.getName());
171             } catch (ClassNotFoundException e) {
172                 throw new RuntimeException(e);
173             }
174             Class c5 = Integer.TYPE;
175             System.out.println(c5.getName());
176             Class c6 = int.class;
177             System.out.println(c6.getName());
178         }
179 }
```

2. 反射与配置文件

通过外部文件配置，在不修改源码的情况下，来控制程序，也符合设计模式中的开闭原则。

```
1 package com.ssh.reflection.entity;
2 
3 /**
4  * @author 申书航
5  * @version 1.0
6  */
```

```java
public class Cat {

    private String name = "Tom";

    public int age = 2;

    public Cat(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public Cat() {

    }

    public void greet() {
        System.out.println(name + " Hello Cat");
    }

    public void cry() {
        System.out.println(name + " Meow");
    }
}
```

```
classFullPath=com.ssh.reflection.entity.Cat
method=greet
```

```java
package com.ssh.reflection;

import java.io.FileInputStream;
import java.io.IOException;
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.util.Properties;

/**
 * @author 申书航
 * @version 1.0
 */
public class ReflectionTest1 {
```

```
15
16      public static void main(String[] args) {
17
18          String path =
"E:\\JavaCode\\java\\chapter19\\src\\com\\ssh\\reflect
ion\\resource\\re.properties";
19
20          Properties prop = new Properties();
21          try {
22              prop.load(new FileInputStream(path));
23              String classFullPath =
prop.get("classFullPath").toString();
24              String methodName =
prop.get("method").toString();
25              System.out.println("classFullPath: " +
classFullPath);
26              System.out.println("methodName: " +
methodName);
27
28              Class clazz =
Class.forName(classFullPath);
29              Object o = clazz.newInstance();
30              System.out.println("o的运行类型: " +
o.getClass());
31              Method method =
clazz.getMethod(methodName);
32              method.invoke(o);   // 执行方法
33
34              // getField()方法不能获取私有属性
35 //              Field name = clazz.getField("name");
36              Field age = clazz.getField("age");
37              System.out.println("age的值: " +
age.get(o));
38              age.set(o, 3);   // 设置属性值
39              System.out.println("age的值: " +
age.get(o));
40
41              Constructor constructor =
clazz.getConstructor();
42              System.out.println("constructor: " +
constructor);
```

```
43
     System.out.println(clazz.getConstructor(String.class,
    int.class));
44
45          } catch (Exception e) {
46              e.printStackTrace();
47          }
48      }
49  }
```

3. 反射与数据库

数据库查询出的每一条数据基本上都会封装为一个对象，数据库中的每一个字段值都会存储在对象相应的属性中。如果查询结果的每一个字段都与对象中的属性名保持一致，那么就可以使用反射来完成万能查询。

`JdbcUtil` 构建演示：

```
1  package com.ssh.jdbc.reflection;
2
3  import java.lang.reflect.Constructor;
4  import java.lang.reflect.Field;
5  import java.lang.reflect.Method;
6  import java.sql.*;
7  import java.util.ArrayList;
8  import java.util.List;
9
10 public class JdbcUtil {
11
12     private static final String url =
    "jdbc:mysql://localhost:3306/lesson?
    serverTimezone=Asia/Shanghai";
13     private static final String username = "root";
14     private static final String password = "root";
15
16     static {
17         try {
18
     Class.forName("com.mysql.cj.jdbc.Driver");
19         } catch (ClassNotFoundException e) {
20             e.printStackTrace();
21             System.out.println("驱动程序加载失败");
```

```
        }
    }

    public static void main(String[] args) {
        String sql = "SELECT id, name, number, price, agent_id agentId FROM goods WHERE name LIKE ? AND price > ?";
        Object[] params = {"%魅%", 1000};
        List<Goods> goodsList = query(sql, Goods.class, params);
//        goodsList.forEach(System.out::println);

        sql = "SELECT id, name, region_id regionId FROM agent WHERE name LIKE ?";
        params = new Object[]{"%小米%"};
        List<Agent> agents = query(sql, Agent.class, params);
        agents.forEach(System.out::println);
    }

    /**
     * 万能更新
     * @param sql
     * @param params
     * @return
     */
    public static int update(String sql, Object... params) {
        int result = 0;
        Connection conn = null;
        PreparedStatement ps = null;
        try {
            conn = DriverManager.getConnection(url,username,password);
            ps = createPreparedStatement(conn, sql, params);
            result = ps.executeUpdate();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        } finally {
            close(ps, conn);
```

```
        }
        return result;
    }

    /**
     * 创建SQL语句执行器
     * @param conn
     * @param sql
     * @param params
     * @return
     * @throws SQLException
     */
    private static PreparedStatement createPreparedStatement(Connection conn, String sql, Object... params) throws SQLException {
        PreparedStatement ps = conn.prepareStatement(sql);
        if (params != null && params.length > 0) {
            for (int i = 0; i < params.length; i++) {
                ps.setObject(i + 1, params[i]);
            }
        }
        return ps;
    }

    /**
     * 关闭连接，执行器，结果集
     * @param closeables
     */
    private static void close(AutoCloseable... closeables) {
        if (closeables != null && closeables.length > 0) {
            for (AutoCloseable ac : closeables) {
                if (ac != null) {
                    try {
                        ac.close();
                    } catch (Exception e) {
                        throw new RuntimeException(e);
                    }
```

```
                }
            }
        }
    }

    /**
     * 万能查询通过反射实现，必须保证类中定义的字段名与查询结果展示的列名称保持一致
     * @param sql
     * @param clazz
     * @param params
     * @return
     * @param <T>
     */
    public static<T> List<T> query(String sql, Class<T> clazz, Object...params) {
        List<T> dataList = new ArrayList<>();
        Connection conn = null;
        PreparedStatement ps = null;
        ResultSet rs = null;
        try {
            conn = DriverManager.getConnection(url,username,password);
            ps = createPreparedStatement(conn, sql, params);
            rs = ps.executeQuery();
            while(rs.next()){
                T t = createInstance(clazz, rs);
                dataList.add(t);
            }
        } catch (Exception e) {
            throw new RuntimeException(e);
        } finally {
            close(rs, ps, conn);
        }
        return dataList;
    }

    /**
     * 生成一个对象
     * @param clazz
```

```
127         * @param rs
128         * @return
129         * @param <T>
130         * @throws Exception
131         */
132        private static <T> T createInstance(Class<T>
    clazz, ResultSet rs) throws Exception {
133            //获取无参构造
134            Constructor<T> c = clazz.getConstructor();
135            //创建对象
136            T t = c.newInstance();
137            //获取类中定义的字段
138            Field[] fields = clazz.getDeclaredFields();
139            for (Field field : fields) {
140                //组装方法名
141                String fieldName = field.getName();
142    //                    set id -> set + I + d == setId
143                String methodName = "set" +
    fieldName.substring(0, 1).toUpperCase() +
    fieldName.substring(1);
144                //根据组装的方法名和参数类型获取方法
145                Method m = clazz.getMethod(methodName,
    field.getType());
146                try {
147                    Object value =
    rs.getObject(fieldName, field.getType());
148                    m.invoke(t,value);
149                }
150                catch (Exception e) {
151                }
152            }
153            return t;
154        }
155
156    //    /**
157    //     * 查询货物
158    //     * @return
159    //     */
160    //    public static List<Goods> getGoods() {
161    //        List<Goods> goodsList = new ArrayList<>();
162    //        try {
```

```
163 //            Connection conn =
    DriverManager.getConnection(url,username,password);
164 //            String sql = "SELECT
    id,name,number,price,agent_id FROM goods WHERE name
    LIKE ? AND price > ?";
165 //            PreparedStatement ps =
    conn.prepareStatement(sql);
166 //            ps.setString(1, "%小米%");
167 //            ps.setDouble(2, 1000.00);
168 //            ResultSet rs = ps.executeQuery();
169 //            while(rs.next()){
170 //                Goods goods = new Goods();
171 //                goods.setId(rs.getLong("id"));
172 //
    goods.setName(rs.getString("name"));
173 //
    goods.setNumber(rs.getInt("number"));
174 //
    goods.setPrice(rs.getDouble("price"));
175 //
    goods.setAgentId(rs.getLong("agent_id"));
176 //                goodsList.add(goods);
177 //            }
178 //            rs.close();
179 //            ps.close();
180 //            conn.close();
181 //        } catch (Exception e) {
182 //            throw new RuntimeException(e);
183 //        }
184 //        goodsList.forEach(System.out::println);
185 //        return goodsList;
186 //    }
187 //
188 //    /**
189 //     * 查询代理
190 //     * @return
191 //     */
192 //    public static List<Agent> getAgents() {
193 //        List<Agent> agents = new ArrayList<>();
194 //        try {
```

```
195 //            Connection conn =
    DriverManager.getConnection(url,username,password);
196 //            String sql = "SELECT id,name,region_id
    FROM agent WHERE name LIKE ?";
197 //            PreparedStatement ps =
    conn.prepareStatement(sql);
198 //            ps.setString(1,"%小米%");
199 //            ResultSet rs = ps.executeQuery();
200 //            while(rs.next()){
201 //                Agent agent = new Agent();
202 //                agent.setId(rs.getInt("id"));
203 //
    agent.setName(rs.getString("name"));
204 //
    agent.setRegionId(rs.getInt("region_id"));
205 //                agents.add(agent);
206 //            }
207 //            rs.close();
208 //            ps.close();
209 //            conn.close();
210 //        } catch (Exception e) {
211 //            throw new RuntimeException(e);
212 //        }
213 //        agents.forEach(System.out::println);
214 //        return agents;
215 //    }
216 }
```

（3）分层开发

1. 分层

分层的优点：

- 分层后每一层只专注于自己所做的事情，能够提高作业质量；
- 便于分工协作，提高作业效率；
- 便于业务拓展；
- 方便问题排查。

分层的特点：

- 上层制定任务，下层接受任务，上层安排下层做事，但下层不能安排上层做事；
- 下层只需要汇报做事的结果，不需要汇报做事的过程。

2. 三层结构

生活中的分层也可以应用于软件开发中，软件开发分层主要分为三层：

- 界面层，又称控制层（controller）：与用户进行交互，主要负责数据采集和展示；
- 业务逻辑层（service）：负责处理功能模块的业务逻辑，以及界面层和数据访问层的数据流转；
- 数据访问层（data access object => dao）：只负责与数据库进行交互

2. 分层原则

软件分层开发也具有生活中分层的特点，这些特点被称之为分层原则：

- 封装性原则：每层只向外公开接口，但隐藏了内部实现细节；
- 顺序访问原则：下层为上层服务，但下层不能使用上层服务；
- 开闭原则：对扩展开放，对修改关闭。

3. 分层开发的优点

软件分层开发的好处：

- 各层专注于自己所做的事情，便于提高开发质量；
- 便于分工协作，提高开发效率；
- 便于程序扩展；
- 便于代码复用；
- 易于维护。

4. 分层开发案例

示例：使用分层开发完成用户注册与登录功能。

```
package com.ssh.layer.model;

public class User {

    private String username;

```

```
    private String password;

    private String salt;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getSalt() {
        return salt;
    }

    public void setSalt(String salt) {
        this.salt = salt;
    }
}
```

```java
package com.ssh.layer.dao;

import com.ssh.layer.model.User;

/**
 * 数据访问层接口
 */
public interface UserDao {

    int saveUser(String username, String password, String salt);

    User getUserByUsername(String username);
}
```

```java
package com.ssh.layer.dao.impl;

import com.ssh.layer.dao.UserDao;
import com.ssh.layer.model.User;
import com.ssh.layer.util.JdbcUtil;

import java.util.List;

public class UserDaoImpl implements UserDao {

    @Override
    public int saveUser(String username, String password, String salt) {
        String sql = "INSERT INTO `user` (`username`, `password`, `salt`) VALUES (?, ?, ?)";
        Object[] params = {username, password, salt};
        return JdbcUtil.update(sql, params);
    }

    @Override
    public User getUserByUsername(String username) {
        String sql = "SELECT username, password, salt FROM user WHERE username = ?";
        List<User> users = JdbcUtil.query(sql, User.class, username);
```

```
        return users.size() == 0 ? null : users.get(0);
    }
}
```

```
package com.ssh.layer.service;

/**
 * 业务层接口
 */
public interface UserService {

    String register(String username, String password);

    String login(String username, String password);
}
```

```
package com.ssh.layer.service.impl;

import com.ssh.layer.dao.UserDao;
import com.ssh.layer.dao.impl.UserDaoImpl;
import com.ssh.layer.model.User;
import com.ssh.layer.service.UserService;
import com.ssh.layer.util.MD5;

public class UserServiceImpl implements UserService {

    /**
     * 处理业务需要获取数据，因此需要使用数据访问层
     */
    private UserDao userDao = new UserDaoImpl();

    @Override
    public String register(String username, String password) {
        String salt = MD5.randString(30);
        String encrypt = MD5.encrypt(password, salt);
        int affectedRows = userDao.saveUser(username, encrypt, salt);
        return affectedRows == 1 ? "注册成功" : "注册失败";
```

```
    }

    @Override
    public String login(String username, String password) {
        User user = userDao.getUserByUsername(username);
        if (user == null) {
            return "账号不存在";
        }
        String salt = user.getSalt();
        String encrypt = MD5.encrypt(password, salt);
        return encrypt.equals(user.getPassword()) ? "登录成功" : "密码错误";
    }
}
```

```
package com.ssh.layer.controller;

import com.ssh.layer.service.UserService;
import com.ssh.layer.service.impl.UserServiceImpl;

/**
 * 控制层
 */
public class UserController {

    /**
     * 控制层调用业务层完成业务处理
     */
    private UserService userService = new UserServiceImpl();

    public String register(String username, String password) {
        return userService.register(username, password);
    }

    public String login(String username, String password) {
```

```
        return userService.login(username, password);
    }
}
```

```
package com.ssh.layer.util;

import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.sql.*;
import java.util.ArrayList;
import java.util.List;

public class JdbcUtil {

    private static final String url =
"jdbc:mysql://localhost:3306/lesson?
serverTimezone=Asia/Shanghai";
    private static final String username = "root";
    private static final String password = "root";

    static {
        try {

 Class.forName("com.mysql.cj.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
            System.out.println("驱动程序加载失败");
        }
    }

    /**
     * 万能更新
     * @param sql
     * @param params
     * @return
     */
    public static int update(String sql, Object...
params) {
        int result = 0;
        Connection conn = null;
```

```
        PreparedStatement ps = null;
        try {
            conn = DriverManager.getConnection(url,username,password);
            ps = createPreparedStatement(conn, sql, params);
            result = ps.executeUpdate();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        } finally {
            close(ps, conn);
        }
        return result;
    }

    /**
     * 创建SQL语句执行器
     * @param conn
     * @param sql
     * @param params
     * @return
     * @throws SQLException
     */
    private static PreparedStatement createPreparedStatement(Connection conn, String sql, Object... params) throws SQLException {
        PreparedStatement ps = conn.prepareStatement(sql);
        if (params != null && params.length > 0) {
            for (int i = 0; i < params.length; i++) {
                ps.setObject(i + 1, params[i]);
            }
        }
        return ps;
    }

    /**
     * 关闭连接，执行器，结果集
     * @param closeables
     */
```

```
69      private static void close(AutoCloseable...
closeables) {
70          if (closeables != null && closeables.length >
0) {
71              for (AutoCloseable ac : closeables) {
72                  if (ac != null) {
73                      try {
74                          ac.close();
75                      } catch (Exception e) {
76                          throw new
RuntimeException(e);
77                      }
78                  }
79              }
80          }
81      }
82
83      /**
84       * 万能查询通过反射实现，必须保证类中定义的字段名与查询结果
展示的列名称保持一致
85       * @param sql
86       * @param clazz
87       * @param params
88       * @return
89       * @param <T>
90       */
91      public static<T> List<T> query(String sql,
Class<T> clazz, Object...params) {
92          List<T> dataList = new ArrayList<>();
93          Connection conn = null;
94          PreparedStatement ps = null;
95          ResultSet rs = null;
96          try {
97              conn =
DriverManager.getConnection(url,username,password);
98              ps = createPreparedStatement(conn, sql,
params);
99              rs = ps.executeQuery();
100             while(rs.next()){
101                 T t = createInstance(clazz, rs);
102                 dataList.add(t);
```

```
            }
        } catch (Exception e) {
            throw new RuntimeException(e);
        } finally {
            close(rs, ps, conn);
        }
        return dataList;
    }

    /**
     * 生成一个对象
     * @param clazz
     * @param rs
     * @return
     * @param <T>
     * @throws Exception
     */
    private static <T> T createInstance(Class<T>
clazz, ResultSet rs) throws Exception {
        //获取无参构造
        Constructor<T> c = clazz.getConstructor();
        //创建对象
        T t = c.newInstance();
        //获取类中定义的字段
        Field[] fields = clazz.getDeclaredFields();
        for (Field field : fields) {
            //组装方法名
            String fieldName = field.getName();
//                    set id -> set + I + d == setId
            String methodName = "set" +
fieldName.substring(0, 1).toUpperCase() +
fieldName.substring(1);
            //根据组装的方法名和参数类型获取方法
            Method m = clazz.getMethod(methodName,
field.getType());
            try {
                Object value =
rs.getObject(fieldName, field.getType());
                m.invoke(t,value);
            }
            catch (Exception e) {
```

```
            }
        }
        return t;
    }
}
```

```
package com.ssh.layer.util;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.Base64;
import java.util.Random;

public class MD5 {

    private static final char[] chars = {

 'a','b','c','d','e','f','g','h','i','j','k','l',

 'm','n','o','p','q','r','s','t','u','v','w','x',
            'y','z',
'0','1','2','3','4','5','6','7','8','9'
    };

    public static String randString(int length) {
        Random r = new Random();
        StringBuilder sb = new StringBuilder(length);
        for (int i = 0; i < length; i++) {
            int index = r.nextInt(chars.length);
            sb.append(chars[index]);
        }
        return sb.toString();
    }

    public static String encrypt(String password,
String secret) {
        try {
            MessageDigest md5 =
MessageDigest.getInstance("MD5");
            int len1 = password.length();
            int len2 = secret.length();
```

```
            String str = password.substring(0,len1) + secret.substring(0,len2) + password.substring(len1) + secret.substring(len2);
            byte[] data = str.getBytes();
            byte[] result = md5.digest(data);
            return Base64.getEncoder().encodeToString(result);
        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
        }
        return "";
    }
}
```

```
package com.ssh.layer.starter;

import com.ssh.layer.controller.UserController;

import java.util.Scanner;

public class Launcher {

    public static void main(String[] args) {
        UserController controller = new UserController();
        Scanner sc = new Scanner(System.in);
//        System.out.println("请输入注册账号：");
//        String username = sc.next();
//        System.out.println("请输入注册密码：");
//        String password = sc.next();
//        String result = controller.register(username, password);
//        System.out.println(result);

        System.out.println("请输入登录账号：");
        String username1 = sc.next();
        System.out.println("请输入登录密码：");
        String password1 = sc.next();
        String loginResult = controller.login(username1, password1);
        System.out.println(loginResult);
```

```
25      }
26  }
```