day02-基础

学习目标：

- [ ] 能够使用SQL语句查询数据
- [ ] 能够使用SQL语句进行条件查询
- [ ] 能够使用SQL语句进行排序
- [ ] 能够使用聚合函数
- [ ] 能够使用SQL语句进行分组查询
- [ ] 能够完成数据的备份和恢复
- [ ] 能够使用可视化工具连接数据库，操作数据库
- [ ] 能够说出多表之间的关系及其建表原则
- [ ] 能够理解外键约束

第一章 SQL语句之DQL

语法：查询不会对数据库中的数据进行修改，根据指定的方式来呈现数据。

语法格式：

```
select * | 列名,列名  from 表名 [where 条件表达式]
```

- select 是查询指令，可以读 1 ~ n 行数据；
- 列名换成 * 号，可以查询所有字段数据；
- 使用 where 来指定对应的条件

1.1 准备工作

创建商品表

```
CREATE TABLE products (
    --自增加 AUTO_INCREMENT
    pid INT PRIMARY KEY AUTO_INCREMENT,
    pname VARCHAR(20),  -- 商品名称
    price DOUBLE,       -- 商品价格
    pdate DATE,         -- 日期
    sid VARCHAR(20)     -- 分类ID
);

INSERT INTO products VALUES(NULL,'泰国大榴莲', 98, NULL, 's001');
INSERT INTO products VALUES(NULL,'新疆大枣', 38, NULL, 's002');
INSERT INTO products VALUES(NULL,'新疆切糕', 68, NULL, 's001');
INSERT INTO products VALUES(NULL,'十三香', 10, NULL, 's002');
INSERT INTO products VALUES(NULL,'老干妈', 20, NULL, 's002');
```

1.2 简单查询

```sql
-- 查询所有的商品
select * from product;

-- 查询指定列：商品名和商品价格
select pname,price from product;

-- 别名查询，使用的 as 关键字，as 也可以省略的
-- 使用别名的好处：显示的时候使用识别性更强的名字，本身也不会去影响到表结构
-- 表别名
-- select 字段名 as 字段别名 from 表名
select * from product as p;

-- 列别名
-- select 列名 as 列别名 from 表名
select pname as pn from product;

-- 列和表，同时指定别名
-- select 列名 as 列别名 from 表名 as 表别名

-- 去掉重复值
-- select distinct 字段名 from 表名
select distinct price from product;

--查询结果是表达式（运算查询）：将所有商品的价格 +10 元进行显示
-- select 列名+固定值 from 表名
-- select 列名1 + 列名2 from 表名
select pname, price + 10 from product;
```

1.3 条件查询

使用条件查询，可以根据当下具体情况直查想要的那部分数据，对记录进行过滤。

SQL 语法关键字： WHERE

语法格式：

```sql
select 字段名 from 表名 where 条件;
```

1. 运算符

操作符	描述	实例
=	等号，检测两个值是否相等，如果相等返回true	(A = B) 返回false。
<>, !=	不等于，检测两个值是否相等，如果不相等返回true	(A != B) 返回 true。
>	大于号，检测左边的值是否大于右边的值, 如果左边的值大于右边的值返回true	(A > B) 返回false。
<	小于号，检测左边的值是否小于右边的值, 如果左边的值小于右边的值返回true	(A < B) 返回 true。
>=	大于等于号，检测左边的值是否大于或等于右边的值, 如果左边的值大于或等于右边的值返回true	(A >= B) 返回false。
<=	小于等于号，检测左边的值是否小于于或等于右边的值, 如果左边的值小于或等于右边的值返回true	(A <= B) 返回 true。

```sql
-- 查询商品名称为十三香的商品所有信息
select * from product where pname = '十三香';

-- 查询商品价格 >60 元的所有的商品信息
select * from product where price > 60;
select * from product where price <= 60;

-- 不等于
select * from product where price != 60;
select * from product where price <> 60;
```

2. 逻辑运算符

NOT !	逻辑非
AND &&	逻辑与
OR ||	逻辑或

```sql
select * from product where price > 40 and pid > 3;

select * from product where price > 40 or pid > 3;

select * from product where pid = 3 or pid pid = 5;
```

3. in 关键字

```sql
-- in 匹配某些值中
select * from product where pid in (2,5,8);
-- 不在这些值中
select * from product where pid not in (2,5,8);
```

4. 指定范围中 between...and

```
select * from product where pid between 2 and 10;
```

5. 模糊查询 like 关键字

```
-- 使用 like 实现模糊查询
-- “新”开头
select * from product where pname like '新%';
-- 第二个字符是“新”
select * from product where pname like '_新';
-- 包含“新”
select * from product where pname like '%新%';
```

1.4 排序

语法:

```
select 字段名 from 表名 where 字段 = 值
order by 字段名 [asc | desc]

asc 升序
desc 降序
```

```
-- 查询所有的商品，按价格进行排序
select * from product order by price;

-- 查询名称有新的商品的信息并且按价格降序排序
select * from product where pname like '%新%' order by price desc;
```

1.5 聚合函数（组函数）

```
特点：只对单列进行操作

常用的聚合函数：
sum()：求某一列的和
avg()：求某一列的平均值
max()：求某一列的最大值
min()：求某一列的最小值
count()：求某一列的元素个数

-- 获得所有商品的价格的总和：
select sum(price) from product;

-- 获得所有商品的平均价格：
select avg(price) from product;

-- 获得所有商品的个数：
select count(*) from product;
```

1.6 分组查询

语法格式：

```
SQL 语法关键字：GROUP BY、HAVING

select 字段1, 字段2... from 表名
group by 分组字段
[having 条件];
```

```
-- 根据 cno 字段分组，分组后统计商品的个数
select sid, count(*) from product group by sid;

-- 根据 cno 分组，分组统计每组商品的平均价格，并且平均价格 > 60;
select sid, avg(price) from product group by sid having avg(price) > 60;
```

注意事项：

① select 语句中的列（非聚合函数列），必须出现在 group by 子句中

② group by 子句中的列，不一定要出现在 select 语句中

③ 聚合函数只能出现 select 语句中或者 having 语句中，一定不能出现在 where 语句中。

having 和 where 的区别：

```
首先，执行的顺序是有先有后。
```

1）where

```
对查询结果进行分组前，将不符合 where 条件的记录过滤掉，然后再分组。
where 后面，不能再使用聚合函数。
```

2）having

```
筛选满足条件的组，分组之后过滤数据。
having 后面，可以使用聚合函数。
```

1.7 分页查询

关键字：limit [offset,] rows

语法格式：

```
select * | 字段列表 [as 别名] from 表名
[where] 条件语句
[group by] 分组语句
[having] 过滤语句
[order by] 排序语句
[limit] 分页语句;

limit offset, length;
offset：开始行数，从 0 开始
length：每页显示的行数
```

limit 关键字不是 SQL92 标准提出的关键字，它是 MySQL 独有的语法。
通过 limit 关键字，MySQL 实现了物理分页。

分页分为逻辑分页和物理分页：

逻辑分页：将数据库中的数据查询到内存之后再进行分页。

物理分页：通过 LIMIT 关键字，直接在数据库中进行分页，最终返回的数据，只是分页后的数据。

```
-- 如果省略第一个参数，默认从 0 开始
select * from product limit 5;

select * from product limit 3, 5;
```

1.8 子查询

定义：子查询允许把一个查询嵌套在另一个查询当中。

子查询，又叫内部查询，相对于内部查询，包含内部查询的就称为外部查询。

子查询可以包含普通select可以包括的任何子句，比如：distinct、 group by、order by、limit、join和union等；

但是对应的外部查询必须是以下语句之一：select、insert、update、delete。

使用的位置：select中、from 后、where 中；group by 和order by 中无实用意义。

举例：查询“化妆品”分类下的商品信息

其他查询语句

union 集合的并集（不包含重复记录）

unionall集合的并集（包含重复记录）

第二章 MySQL图形化开发工具

2.1 安装

提供的Navicat软件可直接使用

2.2 使用

输入用户名、密码，点击连接按钮，进行访问MySQL数据库进行操作

第三章 SQL备份与恢复

3.1 备份

数据库的备份是指将数据库转换成对应的sql文件

1）MySQL命令备份

数据库导出sql脚本的格式：

```
mysqldump  -u用户名 -p密码 数据库名>生成的脚本文件路径
```

例如:

```
mysqldump  -uroot  -proot day02>d:\backup.sql
```

以上备份数据库的命令中需要用户名和密码，即表明该命令要在用户没有登录的情况下使用

2）可视化工具备份

选中数据库，右键 ”备份/导出” ， 指定导出路径，保存成.sql文件即可。

3.2 恢复

数据库的恢复指的是使用备份产生的sql文件恢复数据库，即将sql文件中的sql语句执行就可以恢复数据库内容。

1）命令恢复

使用数据库命令备份的时候只是备份了数据库内容，产生的sql文件中没有创建数据库的sql语句，在恢复数据库之前需要自己动手创建数据库。

- 在数据库外恢复
  - **格式:** `mysql  -uroot  -p密码 数据库名 < 文件路径`
  - 例如: `mysql  -uroot  -proot day02<d:\backup.sql`
- 在数据库内恢复
  - **格式:** `source SQL脚本路径`
  - 例如: `source d:\backup.sql`
  - 注意:使用这种方式恢复数据，首先要登录数据库.

可视化工具恢复

执行的SQL文件，执行即可。

第四章 多表操作

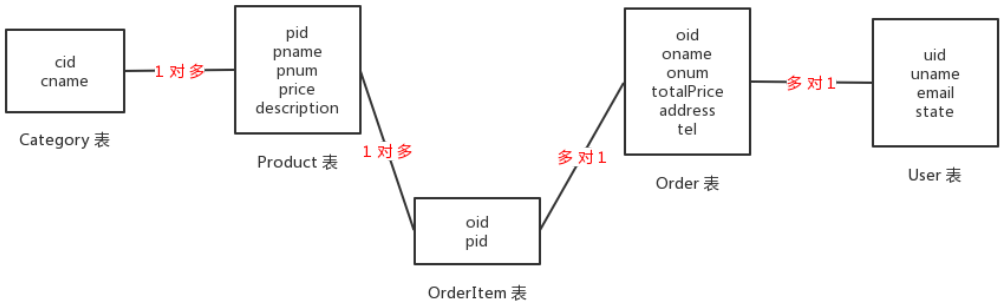
实际开发中，一个项目通常需要很多张表才能完成。例如：一个商城项目就需要分类表(category)、商品表(products)、订单表(orders)等多张表。且这些表的数据之间存在一定的关系，接下来我们将在单表的基础上，一起学习多表方面的知识。

4.1 多表之间的关系

表跟表之间的关系，大家可以理解为是实体跟实体的关系的一种映射。比如，导师与学员，订单与客户，部门与员工等等。

主要关系有三种：

```
一对一：比如，一个男的只能取一个女的当老婆。
一对多：比如，客户与订单，一个客户可以在商城中下多个订单。
多对多：比如，学生与课程，一个学校有很多学生，学生都可以学很多课程。
```



1）一对一关系

在实际工作中，一对一在开发中应用不多，因为一对一完全可以创建成一张表

案例：一个丈夫只能有一个妻子

```
CREATE TABLE wife(
    id  INT PRIMARY KEY ,
    wname  VARCHAR(20),
    sex CHAR(1)
);

CREATE TABLE husband(
    id  INT PRIMARY KEY ,
    hname  VARCHAR(20),
    sex CHAR(1)
);
```

外键唯一

一对一关系创建方式 1 之外键唯一：

添加外键列 wid，指定该列的约束为唯一（不加唯一约束就是一对多关系）

```
ALTER TABLE husband ADD wid INT  UNIQUE;
```

添加外键约束

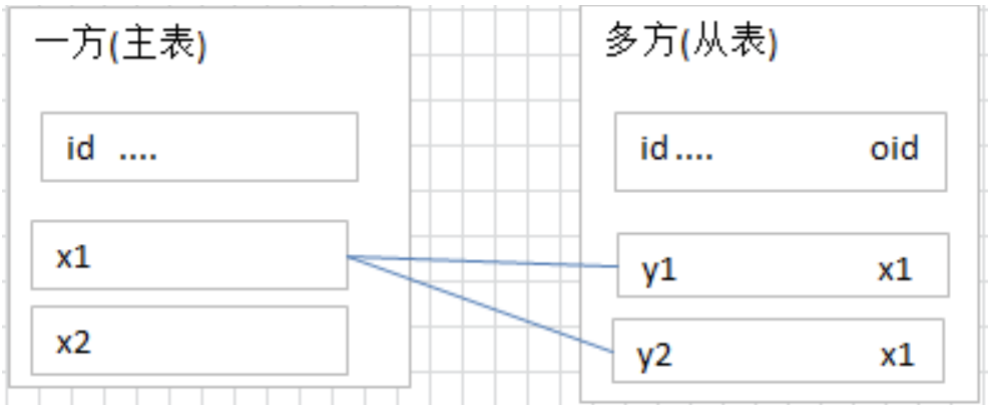
```
alter table husband add foreign key (wid) references wife(id);
```

主键做外键

一对一关系创建方式 2 之主键做外键：（大家下去自己练习）

思路：使用主表的主键作为外键去关联从表的主键

2）一对多关系



常见实例：一个分类对应多个商品，客户和订单，分类和商品，部门和员工.

总结：有外键的就是多的一方。

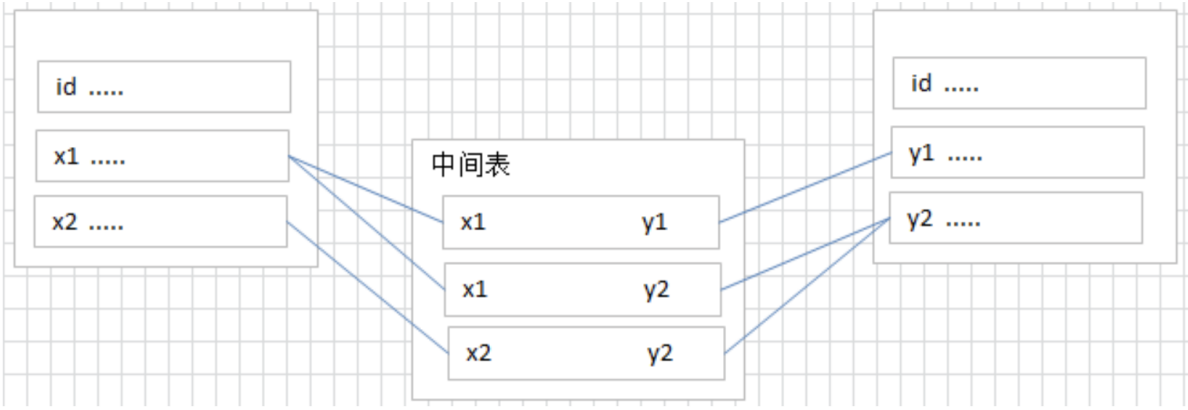
注意事项：一对多关系和一对一关系的创建很类似，唯一区别就是外键不唯一。

一对多关系创建：

- 添加外键列
- 添加外键约束

3）多对多关系

常见实例：学生和课程、用户和角色



注意事项：需要中间表去完成多对多关系的创建，多对多关系其实就是两个一对多关系的组合

多对多关系创建：

- 创建中间表，并在其中创建多对多关系中两张表的外键列
- 在中间表中添加外键约束
- 在中间表中添加联合主键约束

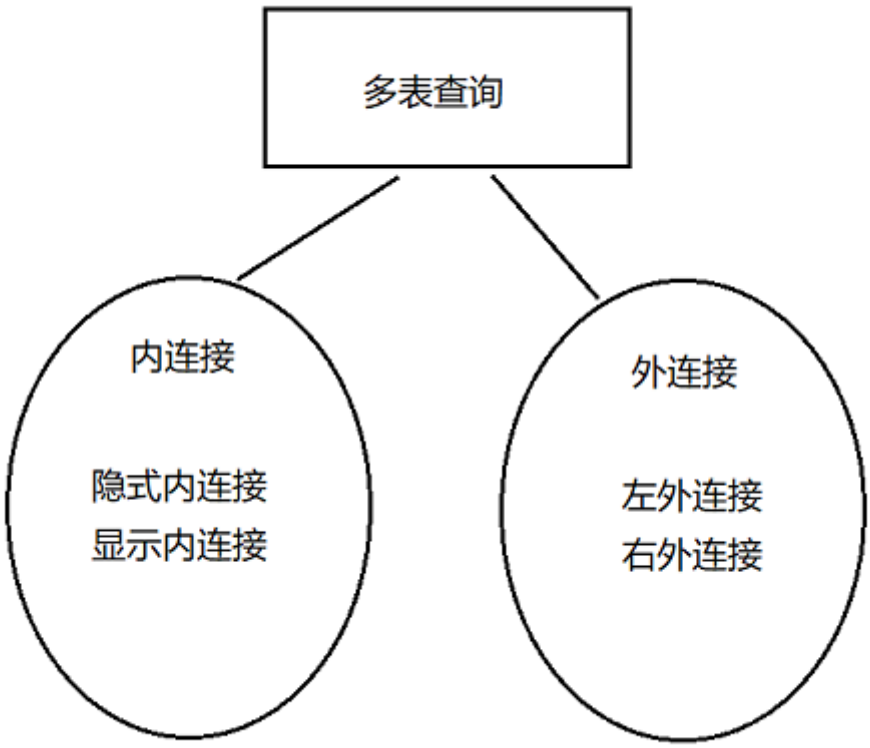
4.2 多表连接查询

4.2.1 多表查询的作用

比如，我们现在有一个员工，希望通过员工查询到对应的部门相关信息，部门名称、部门经理、部门收支等等。

员工 worker --- 部门 department

希望通过一条 SQL 语句进行查询多张相关的表，然后拿到的查询结果，其实是从多张表中综合而来的。比如，我们现在想获取张三的部门经理。



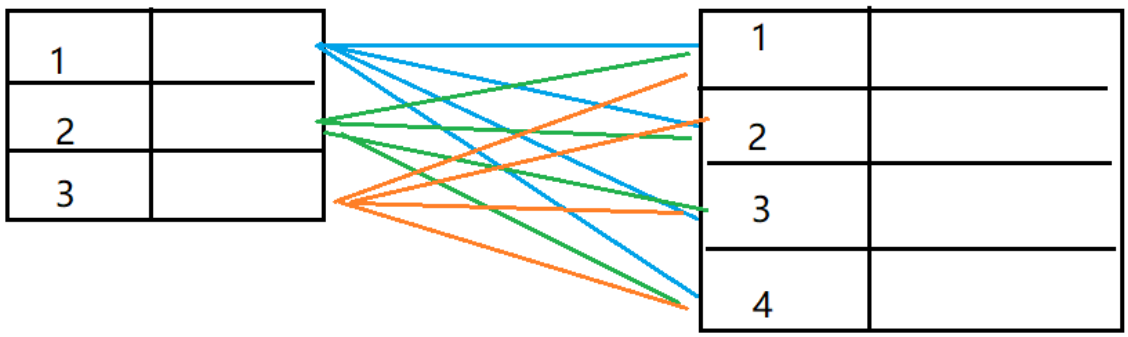
4.2.2 准备数据

```sql
-- 部门表
create table department (
    id int primary key auto_increment,
    name varchar(50)
);
-- 插入部门数据
insert into department(name)
values('技术研发'), ('市场营销'), ('行政财务');

-- 员工表
create table worker (
    id int primary key auto_increment,
    name varchar(50),    -- 名字
    sex char(2),         -- 性别
    money double,        -- 工资
    inWork_date date,    -- 入职时间
    depart_id int,       -- 部门
    foreign key(depart_id) references department(id)
);
-- 插入员工数据
insert into worker(name, sex, money, inWork_date,depart_id)
values('cuihua', '女', 10000, '2019-5-5', 1);
insert into worker(name, sex, money, inWork_date,depart_id)
values('guoqing', '男', 20000, '2018-5-5', 2);
insert into worker(name, sex, money, inWork_date,depart_id)
values('qiangge', '男', 30000, '2018-7-5', 3);
insert into worker(name, sex, money, inWork_date,depart_id)
values('huahua', '女', 10000, '2019-5-5', 1);
```

4.2.3 笛卡尔集

```sql
-- 查询的时候
-- 左边表中的每一条记录和右边表的每一条记录都进行组合了
-- 出现的这种效果，就是笛卡尔集（但具体并不是我们希望得到的查询结果）
select * from worker, department;
```

1）如何产生



2）如何消除

通过上面的分析，我们知道有一些数据其实是无用的，只有满足 worker.depart_id = department.id 这个条件，过滤出来的数据才是我们想要的最终结果。

```
select * from worker, department where id = 3;

-- Column 'id' in where clause is ambiguous
-- select * from worker, department where id = 3;
-- 修改如下
select * from worker, department where worker.depart_id = department.id;
```

4.2.4 内连接

主要是使用左边的表中的记录去匹配右边表中的记录，如果满足条件的话，则显示查询结果。

```
从表.外键 = 主表.主键
```

1）隐式内连接（使用 where 关键字来指定条件）

```
select * from worker, department where worker.depart_id = department.id;
```

2）显示内连接（使用 inner..join..on 语句）

```
-- 1. 查询两张表
-- 使用 on 来指定条件
-- inner 关键字是可以省略的
select * from worker join department
on worker.depart_id = department.id ;

-- 2. 想查一个叫 cuihua 的人
-- 使用别名
select * from worker w join department d
on w.depart_id = d.id
where w.name = 'cuihua';

-- 3. 查询部分字段、
select w.id 员工编号, w.name 员工姓名, w.money 工资, d.name 部门名称 from worker
w join department d
on w.depart_id = d.id
where w.name = 'cuihua';
```

4.2.5 外连接

1）左外连接

使用 left outer join .. on

```
select 字段 from 左表 left outer join 右表 .. on 条件
```

```sql
-- 左外连接

-- 参考，内连接
select * from department d inner join worker w on d.id = w.depart_id;

-- 左外连接
-- outer 关键字是可以省略的
select * from department d left outer join worker w on d.id = w.depart_id;
```

当我们使用左边表中的记录去匹配右边表中的记录，如果满足条件的话，则显示；否则，显示为 null；简单理解：在之前内连接的基础上，先保证左边的数据全部展示，再去找右表中的数据。

2）右外连接

使用 right outer join .. on

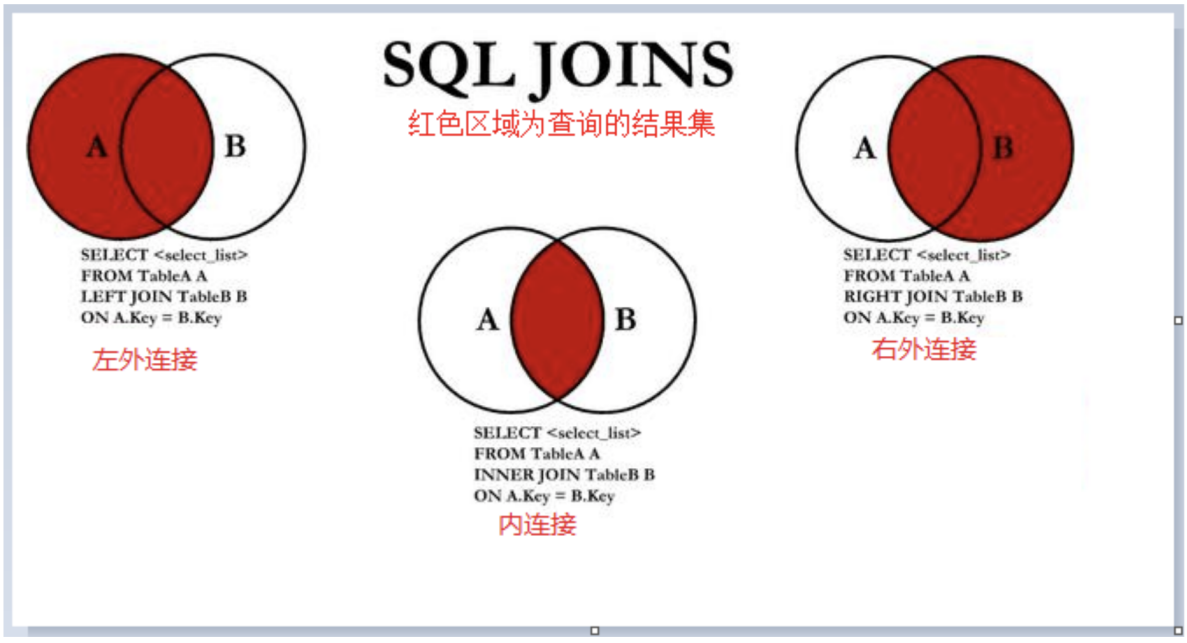
```sql
select 字段 from 左表 right outer join 右表.. on 条件
```

```sql
-- 右外连接
select * from department d right outer join worker w on d.id = w.depart_id;
```

先会使用右边的表中的记录去匹配左边表的记录，如果满足条件，则展示；否则，则显示 null；简单理解：在原来内连接的基础上，保证右边表中的全部数据都展示。

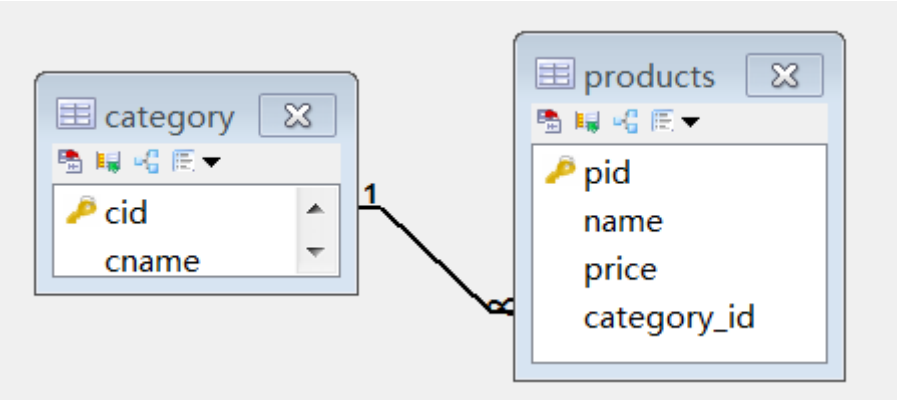
4.2.6 小结：连接区别

下面通过一张图说明连接的区别:



4.3 一对多操作案例

分析



- category分类表，为一方，也就是主表，必须提供主键cid
- products商品表，为多方，也就是从表，必须提供外键category_id

实现：分类和商品

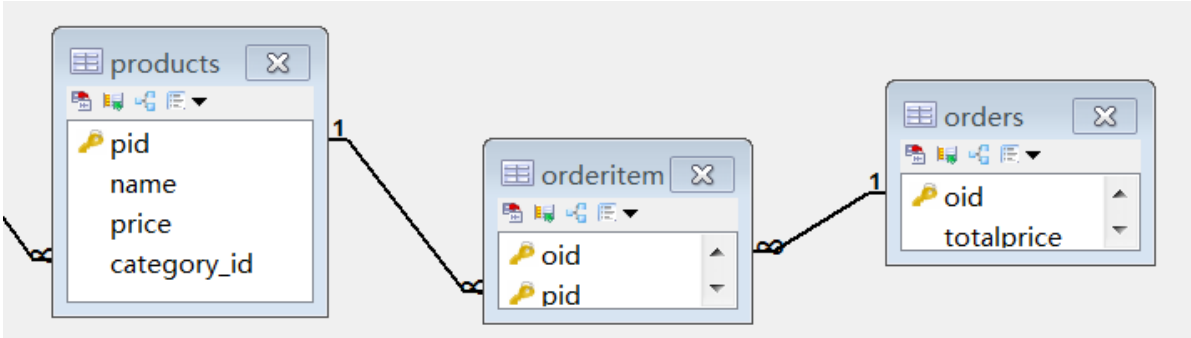
```sql
#创建分类表
create table category(
  cid varchar(32) PRIMARY KEY ,
  cname varchar(100) -- 分类名称
);

# 商品表
CREATE TABLE `products` (
  `pid` varchar(32) PRIMARY KEY  ,
  `name` VARCHAR(40) ,
  `price` DOUBLE
);

#添加外键字段
alter table products add column category_id varchar(32);

#添加约束
alter table products add constraint product_fk foreign key (category_id)
references category (cid);
```

操作

```sql
#1 向分类表中添加数据
INSERT INTO category (cid ,cname) VALUES('c001','服装');

#2 向商品表添加普通数据,没有外键数据，默认为null
INSERT INTO products (pid,pname) VALUES('p001','商品名称');

#3 向商品表添加普通数据，含有外键信息(category表中存在这条数据)
INSERT INTO products (pid ,pname ,category_id) VALUES('p002','商品名称
2','c001');

#4 向商品表添加普通数据，含有外键信息(category表中不存在这条数据) -- 失败,异常
INSERT INTO products (pid ,pname ,category_id) VALUES('p003','商品名称
2','c999');

#5 删除指定分类(分类被商品使用) -- 执行异常
DELETE FROM category WHERE cid = 'c001';
```

4.4 多对多操作案例

分析



- 商品和订单多对多关系，将拆分成两个一对多。
- products商品表，为其中一个一对多的主表，需要提供主键pid
- orders 订单表，为另一个一对多的主表，需要提供主键oid
- orderitem中间表，为另外添加的第三张表，需要提供两个外键oid和pid

实现：订单和商品

```
#商品表[已存在]

#订单表
create table `orders`(
  `oid` varchar(32) PRIMARY KEY ,
  `totalprice` double   #总计
);

#订单项表
create table orderitem(
  oid varchar(50),-- 订单id
  pid varchar(50)-- 商品id
);

#订单表和订单项表的主外键关系
alter table `orderitem` add constraint orderitem_orders_fk foreign key (oid) references orders(oid);

#商品表和订单项表的主外键关系
alter table `orderitem` add constraint orderitem_product_fk foreign key (pid) references products(pid);

#联合主键（可省略）
alter table `orderitem` add primary key (oid,pid);
```

操作

```
#1 向商品表中添加数据
INSERT INTO products (pid,pname) VALUES('p003','商品名称');

#2 向订单表中添加数据
INSERT INTO orders (oid ,totalprice) VALUES('x001','998');
INSERT INTO orders (oid ,totalprice) VALUES('x002','100');

#3向中间表添加数据(数据存在)
```

```sql
INSERT INTO orderitem(pid,oid) VALUES('p001','x001');
INSERT INTO orderitem(pid,oid) VALUES('p001','x002');
INSERT INTO orderitem(pid,oid) VALUES('p002','x002');

#4删除中间表的数据
DELETE FROM orderitem WHERE pid='p002' AND oid = 'x002';

#5向中间表添加数据(数据不存在) -- 执行异常
INSERT INTO orderitem(pid,oid) VALUES('p002','x003');

#6删除商品表的数据 -- 执行异常
DELETE FROM products WHERE pid = 'p001';
```

day03-基础

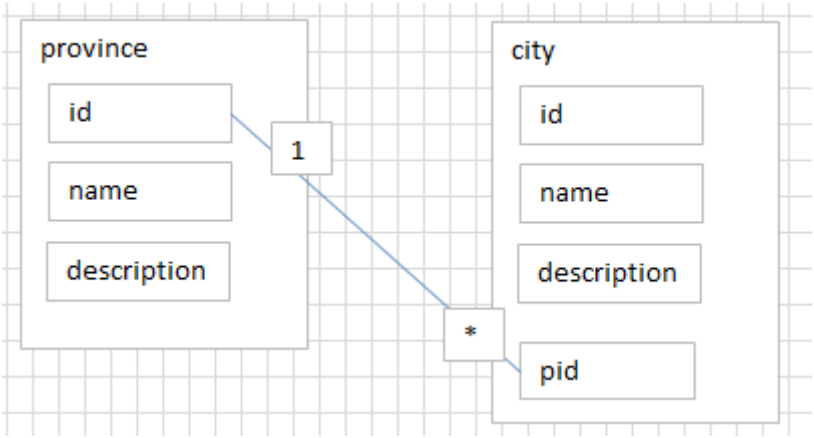
学习目标:

- [ ] 能够使用内连接进行多表查询
- [ ] 能够使用外连接进行多表查询
- [ ] 能够使用子查询进行多表查询

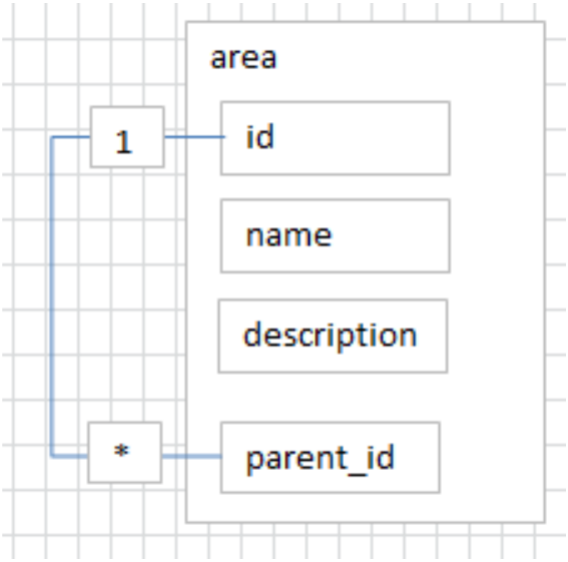
第一章 多表关系实战

1.1 实战1：省和市

- 方案1：多张表，一对多



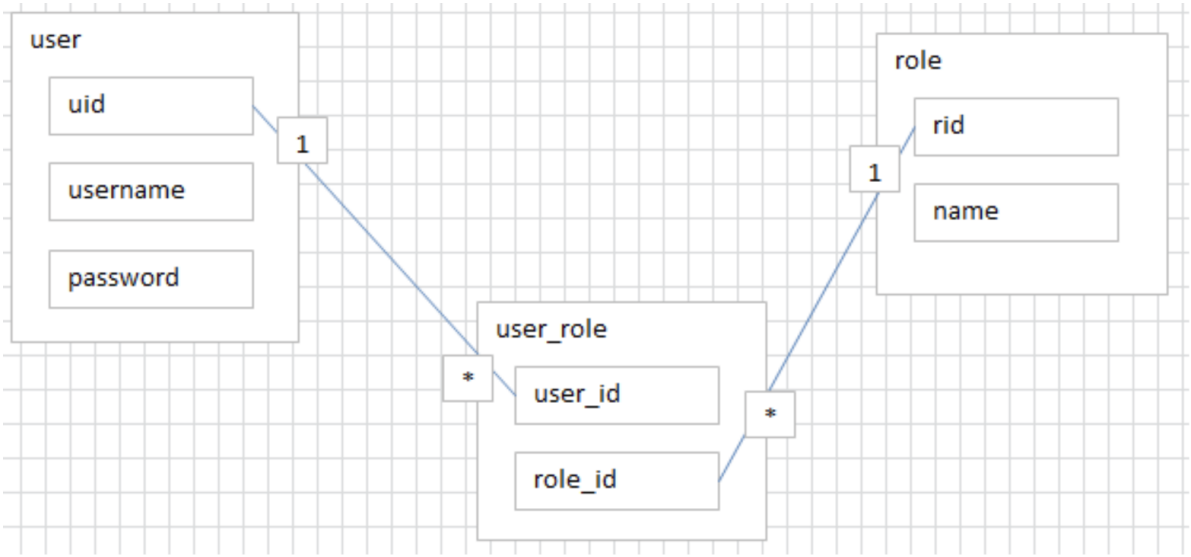
- 方案2：一张表，自关联一对多




```
id=1
name='北京'
p_id = null;
---
id=2
name='昌平'
p_id=1
---
id=3
name='大兴'
p_id=1
---
id=3
name='上海'
p_id=null
---
id=4
name='浦东'
p_id=3
```

1.2 实战2：用户和角色

- 多对多关系



第二章 多表查询之子查询

提供表结构如下：

```sql
-- 部门表
create table department (
    id int primary key auto_increment,
    name varchar(50)
);
-- 员工表
create table worker (
    id int primary key auto_increment,
    name varchar(50),    -- 名字
    sex char(2),         -- 性别
    money double,        -- 工资
    inWork_date date,    -- 入职时间
    depart_id int,       -- 部门
    foreign key(depart_id) references department(id)
);
```

2.1 初始化数据

```sql
-- 插入部门数据
insert into department(name)
values('技术研发'), ('市场营销'), ('行政财务');

-- 插入员工数据
insert into worker(name, sex, money, inWork_date,depart_id)
values('cuihua', '女', 10000, '2019-5-5', 1);
insert into worker(name, sex, money, inWork_date,depart_id)
values('guoqing', '男', 20000, '2018-5-5', 2);
insert into worker(name, sex, money, inWork_date,depart_id)
values('qiangge', '男', 30000, '2018-7-5', 3);
insert into worker(name, sex, money, inWork_date,depart_id)
values('huahua', '女', 10000, '2019-5-5', 1);
```

2.2 什么是子查询?

一个查询的结果作为另一个查询的条件。

有查询的嵌套，内部的查询就是子查询。

子查询，需要使用小括号包含起来。

```
-- 查询技术研发部门有哪些员工
-- 1. 先查询所有的员工
select * from worker where depart_id = 1;
-- 2. 查询部门
select id from department where name = '技术研发';

-- 使用子查询的方式，来统一查询对应的数据
select * from worker where depart_id = (select id from department where name = '技术研发');
```

2.3 常见三种做法

1）单行单列

也就是结果只有一个

```
select 指定的字段 from 表 where 字段 = (子查询)
-- 谁的工资最高?
-- 1. 先把最高工资找出来
select MAX(money) from worker;
-- 2. 再去员工表中，把对应的员工信息查出来
select * from worker where money = (select MAX(money) from worker);

-- 谁的工资少于平均工资?
-- 1. 先把平均工资算出来
select AVG(money) from worker;
-- 2. 再去员工表中，把对应的员工信息查出来
select * from worker where money < (select AVG(money) from worker);
```

2）多行单列

当我们在处理多行单列的时候，有可能会出现多个值，这时候可以类似数组或集合一样处理，在 SQL 中，使用 in 关键字即可。

```
-- 查询那些工资大于 12000 的人都来自哪些部门
-- 1. 先查大于 5000 的员工对应的部门 id
select depart_id from worker where money > 12000;
-- 2. 根据部门的编号，再找出部门的名字
-- 你查找到的记录，多于 1 行了 Subquery returns more than 1 row
-- select name from department where id = (select depart_id from worker where money > 12000);
select name from department where id in (select depart_id from worker where money > 12000);


-- 查询行政财务和技术研发中的所有员工的信息
```

```sql
-- 1. 先根据名字来查找 id
select id from department where name in ('行政财务', '技术研发');
-- 2. 再去查询相关的员工
select * from worker where depart_id in (select id from department where name in ('行政财务', '技术研发'));
```

3）多行多列

当你的子查询只要是多列，那么它肯定在 from 后面是以一张表存在的。

```sql
select 字段 from (子查询) 表别名 where 条件;
```

子查询在这里要作为表，然后还要给一个别名，如果不这样处理的话，就没办法访问到表中的字段。

```sql
-- 从 2019 年后入职的员工和相关部门信息
-- 1. 2019-1-1 后的时间
select * from worker where inWork_date >= '2019-01-01';
-- 2. 当我们从上面查找到对应的员工，则可以通过员工的 depart_id 找到对应的部门信息
select * from department d, (select * from worker where inWork_date >= '2019-01-01') w
where d.id = w.depart_id;

-- 将上面的例子，换成内连接实现
select * from worker inner join department
on worker.depart_id = department.id
where inWork_date >= '2019-01-01';
```

第三章 事务

3.1 事务原理

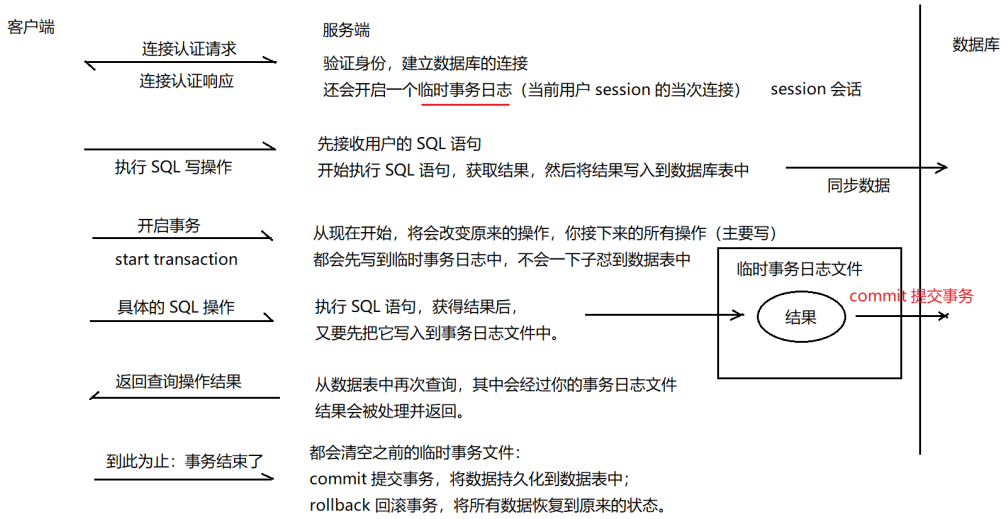
事务：要么成功，要么不成功。

转账，A - 500，B + 500。转账的过程中，有可能会有一些突发的情况，导致转账操作会出现一些意料不到的问题。所以，在这里，我们需要建立一个通道，在通道中完成的操作，要么成功，要么不成功的时候及时回滚数据，避免造成大面积的业务混乱。

```sql
-- 账户表
create table bankCount(
  id int primary key auto_increment,
  name varchar(50),
  money double
);


-- 添加数据
insert into bankCount(name, money)
values ('cuihua', 1000), ('banban', 2000);

-- 翠花给班班转钱 500
update bankCount set money = money - 500 where name = 'cuihua';
update bankCount set money = money + 500 where name = 'banban';

-- 提供一个事务通道，让转账的操作稳妥一些，如果中途出现问题，及时回滚数据，不要造成数据丢失
```

事务的原理：

当我们在 MySQL 中，如果开启了事务，那么你所有的操作都会临时被保存在事务日志中，只有遇上 commit（提交）命令的时候，才会同步到数据表中。如果遇上 rollback 和 断开连接，那么它都会去清空你的事务日志。



3.2 手动提交

1）核心 SQL 语句

```
开启事务：start transaction
提交事务：commit
回滚事务：rollback
```

2）实现过程

```
第一步：开启事务
第二步：执行你的 SQL 语句
第三步：提交事务
第四步：如果出现问题的话，则回滚事务（数据）
```

3）事务提交

```
第一步：开启事务
start transaction;

第二步：执行你的 SQL 语句
update bankCount set money = money - 500 where name = 'cuihua';
update bankCount set money = money + 500 where name = 'banban';

第三步：如果出现问题的话，则回滚事务（数据）
commit;
```

4）事务回滚

```
第一步：开启事务
start transaction;

第二步：执行你的 SQL 语句
update bankCount set money = money - 500 where name = 'cuihua';
update bankCount set money = money + 500 where name = 'banban';

第三步：提交事务
rollback;
```

3.3 自动提交

MySQL 默认情况下，每一条 DML 语句都是一个单独的事务，都会对应的开启一个事务，当你执行的时候，同时自动默认提交事务。

1）自动提交事务

```
-- 随便一个 DML 语句都是会自动提交事务的
update bankCount set money = money + 500 where name = 'banban';
```

2）取消自动提交

@@autocommit 原来是 1 的，如果你要取消的话，则设置为 0 即可（但不建议）

```
set @@autocommit = 0;
```

3.4 事务

1）事务的特性 ACID

a. 原子性：Atomicity 在每一个事务中，都可以看成是一个整体，不能将其再度分解，所有的操作，要么一起成功，要么一起失败。

b. 一致性：Consistency 在事务执行前数据的所有状态跟执行后的数据状态应该是一样的。比如，转账前两个账户的金额总和应该跟转账后两个账户的总金额都是一样的。

c. 隔离性：Isolation 多个事务之间不能相互影响，必须保证其操作的单独性，否则会出现一些串改的情况，执行的时候应该保持隔离的状态。

d. 持久性：Durability 如果我们的事务执行成功之后，它将把数据永久性存储到数据库中，哪怕设备关机之后，也是能够保存下来的。

2）隔离级别可能会出现的问题

a. 脏读：其中一个事务读取到了另外一个事务中的数据（尚未提交的数据）。

b. 不可重复读：一个事务中两次读取数据的时候，发现数据的内容不一样。要求，多次读取数据的时候，在一个事务中读出的都应该是一样的。一般是由于 update 操作引发，所以将来执行的时候要特别注意。

c. 幻读：一般都是 insert 或者 delete 操作的时候会出现这个问题。一个事务中，两次读取数据的时候，发现数据的数量不一样。要求，在一个事务中多次去读取数据的时候都应该是一样的。

3）MySQL 的四种隔离级别

如果你将来使用 命令行 来设置隔离级别的时候，只有在当次会话中是有效的。只要你关掉了窗口，隔离级别会即时恢复到默认状况 --- RR。

- read uncommitted 读未提交
- read committed 读已提交
- repeatable read 可重复读（默认）
- serializable 串行化

级别	级别名字	脏读	不可重复读	幻读	备注
1	read uncommitted	✓	✓	✓	
2	read committed		✓	✓	
3	repeatable read			✓	select @@tx_isolation
4	serializable				

当你设置的级别越高，性能就越差，但安全性非常高（建议，有选择使用，一般就只用默认就好）

3.5 案例演示（了解）

1）脏读

先打开窗口，设置隔离级别：

```
set global transaction isolation level read uncommitted;
```

如果设置的级别是“读未提交”，其实会造成一个脏读的问题。脏读，会导致一个事务读取到了另外一个事务中的数据，其实非常危险。

解决方案：提升你的隔离级别。

```
set global transaction isolation level read committed;
```

当你设置成 read commintted 就不能读取到另外一个事务中的数据了。

只有当第一个事务提交了数据之后，第二个事务才能够去读取到数据。

read committed 可以避免数据的脏读。

2）不可重复读

如果将来，你写的 SQL 语句，发现第一次查询的时候，是一个结果，第二次查询的时候又是另外一个结果。一般都是最后一次查询的才是正确的，有时候第一次不正确的结果会被误用，就会给用户不好的体验。

订票：PC 端 --- App 端 --- 短信　　如果说每次查询的结果不一样的话，则会导致推送用户信息的时候，呈现的数据不同步。

解决方案：将你的级别设置为 repeatable read 可重复读，也就是 mysql 默认的级别（不建议修改）。

3）幻读（课后自己演示）

```
set global transaction isolation level serializable ;
```

第四章 DCL

- DDL create、alter、drop
- DML insert、update、delete
- DQL select、show
- DCL grant、revoke

4.1 创建用户

如果将来创建一个新的用户，它并不会拥有与 root 用户一样的权限，root 是超级管理员，所有的权限它都有。

```
create user '用户名'@'主机名' identified by '密码';
CREATE USER '用户名'@'主机名' IDENTIFIED WITH mysql_native_password  BY '密码';
create user 'cuihua'@'localhost' identified by '1234';
create user 'huahua'@'localhost' identified by '1234';
```

4.2 授权

如果想要使用这些新增的用户，则需要授予一定的权限。

```
grant 权限1, 权限2,..., 权限N on 数据库名.表名 to '用户'@'主机名'
-- cuihua 的权限
-- 如果希望在某个数据库下所有的表都能用的话，则建议写成  数据库名.*
grant create, alter, insert, update, select on hello.* to
'cuihua'@'localhost';
-- 简单的赋权限方法
grant all on *.* to 'huahua'@'localhost';
```

4.3 撤销授权

```
revoke all on hello.* from 'cuihua'@'localhost';
```

4.4 查看权限

```
-- 查看权限
show grants for '用户名'@'主机名';
```

4.5 删除用户

```
drop user '用户名'@'主机名';
```

4.6 修改用户的密码

```
ALTER USER '用户名'@'主机名' IDENTIFIED WITH mysql_native_password BY '新密码';
```