零声教育出品 Mark 老师 QQ: 2548898954

初识MySQL

数据库

> **按照数据结构来组织、存储和管理数据的仓库**；是一个长期存储在计算机内的、有组织的、可共享的、统一管理的大量数据的集合；

OLTP

> *OLTP*（*On-Line transaction processing*）翻译为联机事务处理；主要对数据库增删改查；
>
> *OLTP* 主要用来**记录某类业务事件**的发生；数据会以增删改的方式在数据库中进行数据的更新处理操作，要求实时性高、稳定性强、确保数据及时更新成功；

OLAP

> *OLAP*（*On-Line Analytical Processing*）翻译为联机分析处理；主要对数据库查询；
>
> 当数据积累到一定的程度，我们需要对过去发生的事情做一个总结分析时，就需要把**过去一段时间内产生的数据**拿出来进行**统计分析**，从中获取我们想要的信息，为公司做决策提供支持，这时候就是在做 *OLAP* 了；

SQL

定义

> 结构化查询语言(*Structured Query Language*) 简称 *SQL*，是一种特殊目的的编程语言，是一种数据库查询和程序设计语言，用于**存取数据以及查询、更新和管理关系数据库系统。***SQL* 是关系数据库系统的标准语言。
>
> 关系型数据库包括：*MySQL*, *SQL Server*, *Oracle*, *Sybase*, *postgreSQL* 以及 *MS Access*等；
>
> *SQL* 命令包括：*DQL*、*DML*、*DDL*、*DCL*以及*TCL*；

DQL

> *Data Query Language* - 数据查询语言；
>
> `select`：从一个或者多个表中检索特定的记录；

DML

> Data Manipulate Language - 数据操作语言；
>
> `insert`：插入记录；
>
> `update`：更新记录；
>
> `delete`：删除记录；

DDL

> `Data Define Languge` - 数据定义语言；
>
> `create`：创建一个新的表、表的视图、或者在数据库中的对象；
>
> `alter`：修改现有的数据库对象，例如修改表的属性或者字段；
>
> `drop`：删除表、数据库对象或者视图；
>
> truncate

DCL

> `Data Control Language` - 数据控制语言；
>
> `grant`：授予用户权限；
>
> `revoke`：收回用户权限；

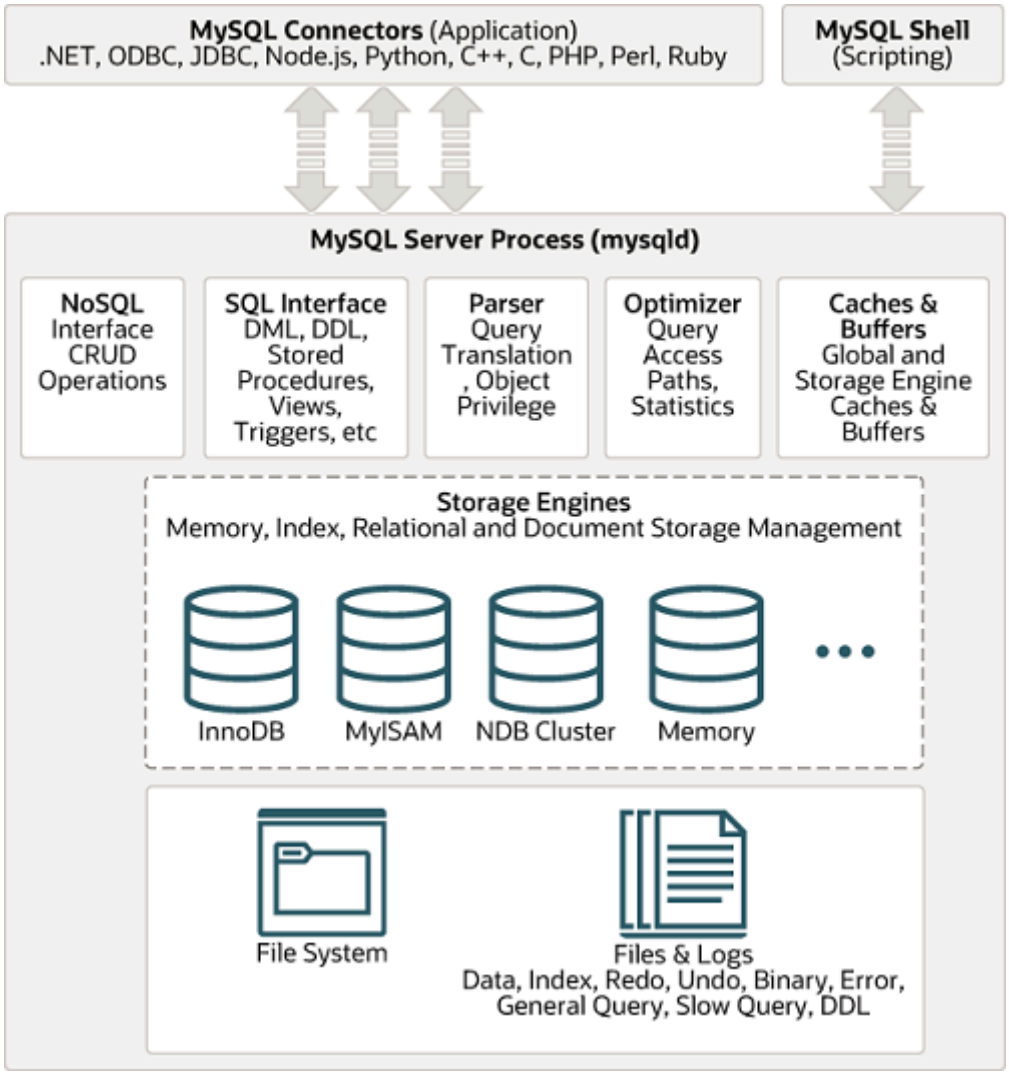
TCL

> `Transaction Control Language` - 事务控制语言；
>
> `commit`：事务提交；
>
> `rollback`：事务回滚；

数据库术语

> 数据库：数据库是一些**关联表的集合**；
>
> 数据表：表是**数据的矩阵**；
>
> 列：一列包含**相同类型的数据**；
>
> 行：或者称为记录是**一组相关的数据**；
>
> 主键：主键是唯一的；一个数据表只能包含一个主键；
>
> 外键：外键用来关联两个表，来保证参照完整性；*MyISAM* 存储引擎本身并不支持外键，只起到注释作用；而 *innoDB* 完整支持外键；
>
> 复合键：或称组合键；将多个列作为一个索引键；
>
> 索引：用于快速访问数据表的数据；索引是对表中的一列或者多列的值进行排序的一种结构；

MySQL体系结构



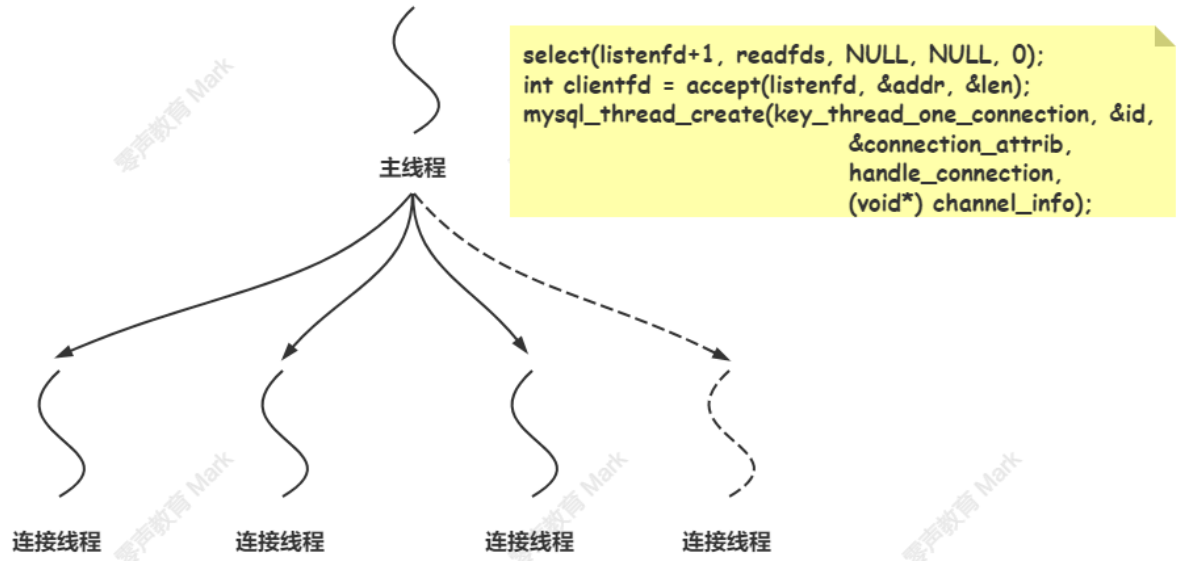
> *MySQL* 由以下几部分组成：
>
> 连接池组件、管理服务和工具组件、*SQL* 接口组件、查询分析器组件、优化器组件、缓冲组件、插件式存储引擎、物理文件。

连接者

> 不同语言的代码程序和 *MySQL* 的交互（SQL交互）；

MySQL 内部连接池

> 管理缓冲用户连接、用户名、密码、权限校验、线程处理等需要缓存的需求；
>
> 网络处理流程：主线程接收连接，接收连接交由连接池处理；
>
> 主要处理方式：IO多路复用 *select* + 阻塞的 *io*；
>
> 需要理解：MySQL 命令处理是多线程并发处理的；




```
select(listenfd+1, readfds, NULL, NULL, 0);
int clientfd = accept(listenfd, &addr, &len);
mysql_thread_create(key_thread_one_connection, &id,
                    &connection_attrib,
                    handle_connection,
                    (void*) channel_info);
```

```
while (1) {
  int n = read(clientfd, buff, sz);
  ...
  do_command();
}
```

> 主线程负责接收客户端连接，然后为每个客户端 *fd* 分配一个连接线程，负责处理该客户端的 *sql* 命令处理；

管理服务和工具组件

> 系统管理和控制工具，例如备份恢复、*MySQL* 复制、集群等；

SQL接口

> 将 *SQL* 语句解析生成相应对象；*DML*，*DDL*，存储过程，视图，触发器等；

查询解析器

> 将 *SQL* 对象交由解析器验证和解析，并生成语法树；

查询优化器

> *SQL* 语句执行前使用查询优化器进行优化；

缓冲组件

> 是一块内存区域，用来弥补磁盘速度较慢对数据库性能的影响；在数据库进行读取页操作，首先将从磁盘读到的页存放在缓冲池中，下一次再读相同的页时，首先判断该页是否在缓冲池中，若在缓冲池命中，直接读取；否则读取磁盘中的页，说明该页被 *LRU* 淘汰了；缓冲池中 *LRU* 采用最近最少使用算法来进行管理；
>
> 缓冲池缓存的数据类型有：索引页、数据页、以及与存储引擎缓存相关的数据（比如*innoDB* 引擎：*undo* 页、插入缓冲、自适应 *hash* 索引、*innoDB* 相关锁信息、数据字典信息等）；

数据库设计三范式

> 为了建立冗余较小、结构合理的数据库，设计数据库时必须遵循一定的规则。在关系型数据库中这种规则就称为范式。范式是符合某一种设计要求的总结。要想设计一个结构合理的关系型数据库，必须满足一定的范式。

范式一

> 确保每列保持原子性；数据库表中的所有字段都是**不可分解**的原子值；
>
> 例如：某表中有一个地址字段，如果经常需要访问地址字段中的城市属性，则需要将该字段拆分为多个字段，省份、城市、详细地址等；

范式二

> 满足范式一的基础上，确保表中的每列都**和主键完全依赖**，而**不能**只与主键的某一**部分依赖**（组合索引）；

订单编号	商品编号	商品名称	数量	单位	价格	客户	所属单位	联系方式
1	1	电脑	1	台	10000	mark	0voice1	13788888888
1	2	手机	2	部	6000	mark	0voice1	13788888888
2	3	ipad	3	部	3000	milo	0voice2	13799999999

订单编号	客户	所属单位	联系方式
1	mark	0voice1	13788888888
2	milo	0voice2	13799999999

订单编号	商品编号	数量
1	1	1
1	2	2
2	3	3

商品编号	商品名称	单位	商品价格
1	电脑	台	10000
2	手机	部	6000
3	ipad	部	3000

范式三

> 满足范式二的基础上，确保每列都和主键**直接相关**，而不是**间接相关**；减少数据冗余；

订单编号	订单项目	负责人	业务员	订单数量	客户编号
1	电脑	mark	秋香	10	1
2	手机	king	贝贝	20	2
3	ipad	darren	柚子	30	1

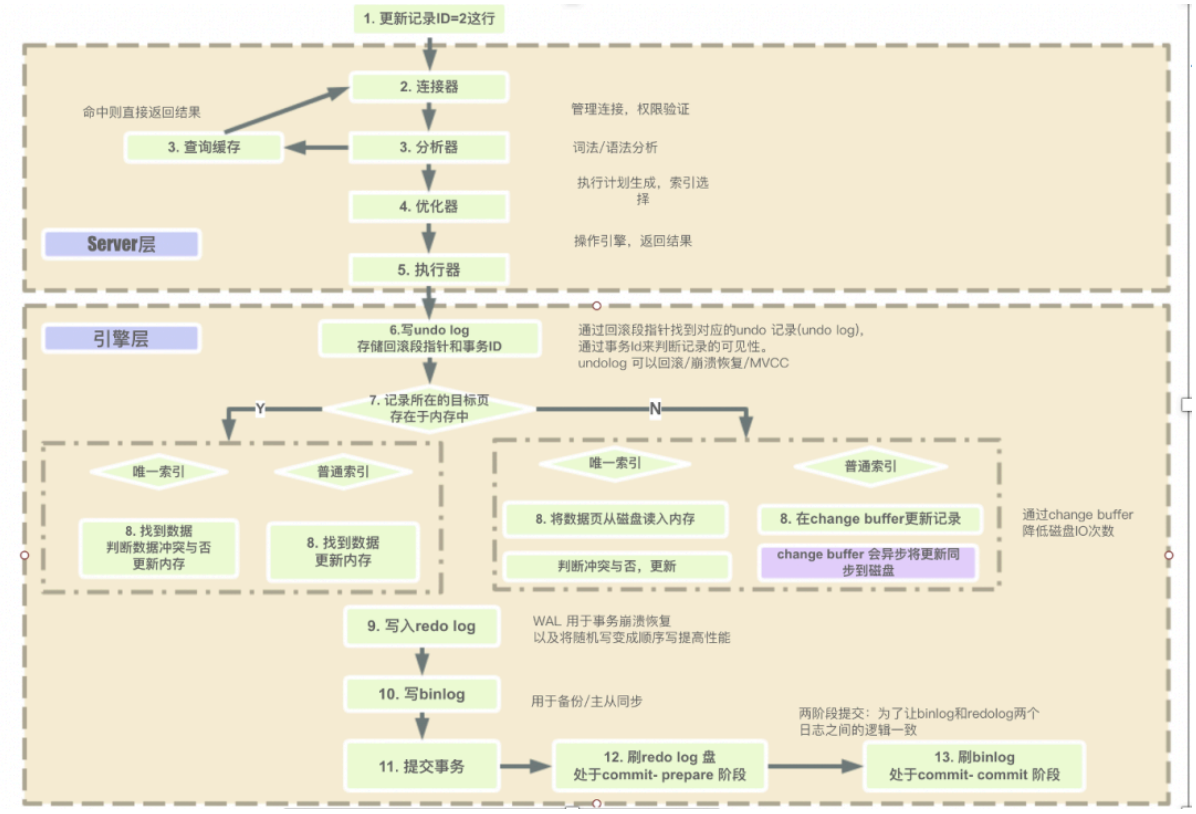
客户编号	客户名称	所属公司	联系方式
1	vico	0voice1	13766666666
2	milo	0voice2	13799999999

反范式

> 范式可以避免数据冗余，减少数据库的空间，减小维护数据完整性的麻烦；但是采用数据库范式化设计，可能导致数据库业务涉及的**表变多**，并且造成**更多的联表查询**，将导致整个系统的**性能降低**；因此**基于性能考虑**，可能需要进行反范式设计；

CRUD

执行过程



创建数据库

```
CREATE DATABASE `数据库名` DEFAULT CHARACTER SET utf8; # 字符集设置为 utf8
```

删除数据库

```
DROP DATABASE `数据库名`;
```

选择数据库

```
USE `数据库名`;
```

创建表

```
CREATE TABLE `table_name` (column_name column_type);

CREATE TABLE IF NOT EXISTS `0voice_tbl` (
   `id` INT UNSIGNED AUTO_INCREMENT COMMENT '编号',
   `course` VARCHAR(100) NOT NULL COMMENT '课程',
   `teacher` VARCHAR(40) NOT NULL COMMENT '讲师',
   `price` DECIMAL(8,2) NOT NULL COMMENT '价格',
   PRIMARY KEY ( `id` ),  ## not null unique
)ENGINE=innoDB DEFAULT CHARSET=utf8 COMMENT = '课程表';
```

删除表

```
DROP TABLE `table_name`;    # 把数据和表都删除
```

清空数据表

```
TRUNCATE TABLE `table_name`; -- 截断表 以页为单位（至少有两行数据），有自增索引的话，从初始值开始累加
DELETE TABLE `table_name`; -- 逐行删除，有自增索引的话，从之前值继续累加
```

增

```
INSERT INTO `table_name`(`field1`, `field2`, ..., `fieldn`) VALUES (value1, value2, ..., valuen);
INSERT INTO `0voice_tbl` (`course`, `teacher`, `price`) VALUES ('C/C++Linux服务器开发/高级架构师', 'Mark', 7580.0);
```

删

```
DELETE FROM `table_name` [WHERE Clause];

DELETE FROM `0voice_tbl` WHERE id = 3;
```

改

```
UPDATE table_name SET field1=new_value1, field2=new_value2 [, fieldn=new_valuen]

UPDATE `0voice_tbl` SET `teacher` = 'Mark' WHERE id = 2;
-- 累加
UPDATE `0Voice_tbl` set `age` = `age` + 1 WHERE id = 2;
```

查

```
SELECT field1, field2,...fieldN FROM table_name
[WHERE Clause]
```

高级查询

准备

```
DROP TABLE IF EXISTS `class`;
CREATE TABLE `class` (
  `cid` int(11) NOT NULL AUTO_INCREMENT,
  `caption` varchar(32) NOT NULL,
  PRIMARY KEY (`cid`)
) ENGINE=innoDB AUTO_INCREMENT=5 DEFAULT CHARSET=utf8;

# innoDB 有外键约束   myisam 注释的作用
DROP TABLE IF EXISTS `course`;
CREATE TABLE `course` (
  `cid` int(11) NOT NULL AUTO_INCREMENT,
  `cname` varchar(32) NOT NULL,
  `teacher_id` int(11) NOT NULL,
  PRIMARY KEY (`cid`),
  KEY `fk_course_teacher` (`teacher_id`),
  CONSTRAINT `fk_course_teacher` FOREIGN KEY (`teacher_id`) REFERENCES `teacher`
(`tid`)
) ENGINE=innoDB AUTO_INCREMENT=5 DEFAULT CHARSET=utf8;

DROP TABLE IF EXISTS `score`;
CREATE TABLE `score` (
  `sid` int(11) NOT NULL AUTO_INCREMENT,
  `student_id` int(11) NOT NULL,
  `course_id` int(11) NOT NULL,
  `num` int(11) NOT NULL,
  PRIMARY KEY (`sid`),
  KEY `fk_score_student` (`student_id`),
  KEY `fk_score_course` (`course_id`),
  CONSTRAINT `fk_score_course` FOREIGN KEY (`course_id`) REFERENCES `course`
(`cid`),
  CONSTRAINT `fk_score_student` FOREIGN KEY (`student_id`) REFERENCES `student`
(`sid`)
) ENGINE=innoDB AUTO_INCREMENT=53 DEFAULT CHARSET=utf8;

DROP TABLE IF EXISTS `student`;
CREATE TABLE `student` (
```

```
  `sid` int(11) NOT NULL AUTO_INCREMENT,
  `gender` char(1) NOT NULL,
  `class_id` int(11) NOT NULL,
  `sname` varchar(32) NOT NULL,
  PRIMARY KEY (`sid`),
  KEY `fk_class` (`class_id`),
  CONSTRAINT `fk_class` FOREIGN KEY (`class_id`) REFERENCES `class` (`cid`)
) ENGINE=innoDB AUTO_INCREMENT=17 DEFAULT CHARSET=utf8;

DROP TABLE IF EXISTS `teacher`;
CREATE TABLE `teacher` (
  `tid` int(11) NOT NULL AUTO_INCREMENT,
  `tname` varchar(32) NOT NULL,
  PRIMARY KEY (`tid`)
) ENGINE=innoDB AUTO_INCREMENT=6 DEFAULT CHARSET=utf8;
```

基础查询

```
-- 全部查询
SELECT * FROM student;
-- 只查询部分字段
SELECT `sname`, `class_id` FROM student;
-- 别名  列明  不要用关键字
SELECT `sname` AS '姓名' , `class_id` AS '班级ID' FROM student;
-- 把查询出来的结果的重复记录去掉
SELECT distinct `class_id` FROM student;
```

条件查询

```
-- 查询姓名为 邓洋洋 的学生信息
SELECT * FROM `student` WHERE `name` = '邓洋洋';
-- 查询性别为 男，并且班级为 2 的学生信息
SELECT * FROM `student` WHERE `gender`="男" AND `class_id`=2;
```

范围查询

```
-- 查询班级id 1 到 3 的学生的信息
SELECT * FROM `student` WHERE `class_id` BETWEEN 1 AND 3;
```

判空查询

```
# is null 判断造成索引失效
# 索引 B+ 树
SELECT * FROM `student` WHERE `class_id` IS NOT NULL;   #判断不为空
SELECT * FROM `student` WHERE `class_id` IS NULL;       #判断为空

SELECT * FROM `student` WHERE `gender` <> '';           #判断不为空字符串
SELECT * FROM `student` WHERE `gender` = '';            #判断为空字符串
```

模糊查询

```
-- 使用 like关键字，"%"代表任意数量的字符，”_”代表占位符
-- 查询名字为 m 开头的学生的信息
SELECT * FROM `teacher` WHERE `tname` LIKE '谢%';
-- 查询姓名里第二个字为 小 的学生的信息
SELECT * FROM `teacher` WHERE `tname` LIKE '_小%';
```

分页查询

```
-- 分页查询主要用于查看第N条 到 第M条的信息，通常和排序查询一起使用
-- 使用limit关键字，第一个参数表示从条记录开始显示，第二个参数表示要显示的数目。表中默认第一条记录的参数为0。
-- 查询第二条到第三条内容
SELECT * FROM `student` LIMIT 1,2;
```

查询后排序

```
-- 关键字：order by field,  asc:升序, desc:降序
SELECT * FROM `score` ORDER BY `num` ASC;
-- 按照多个字段排序
SELECT * FROM `score` ORDER BY `course_id` DESC, `num` DESC;
```

聚合查询

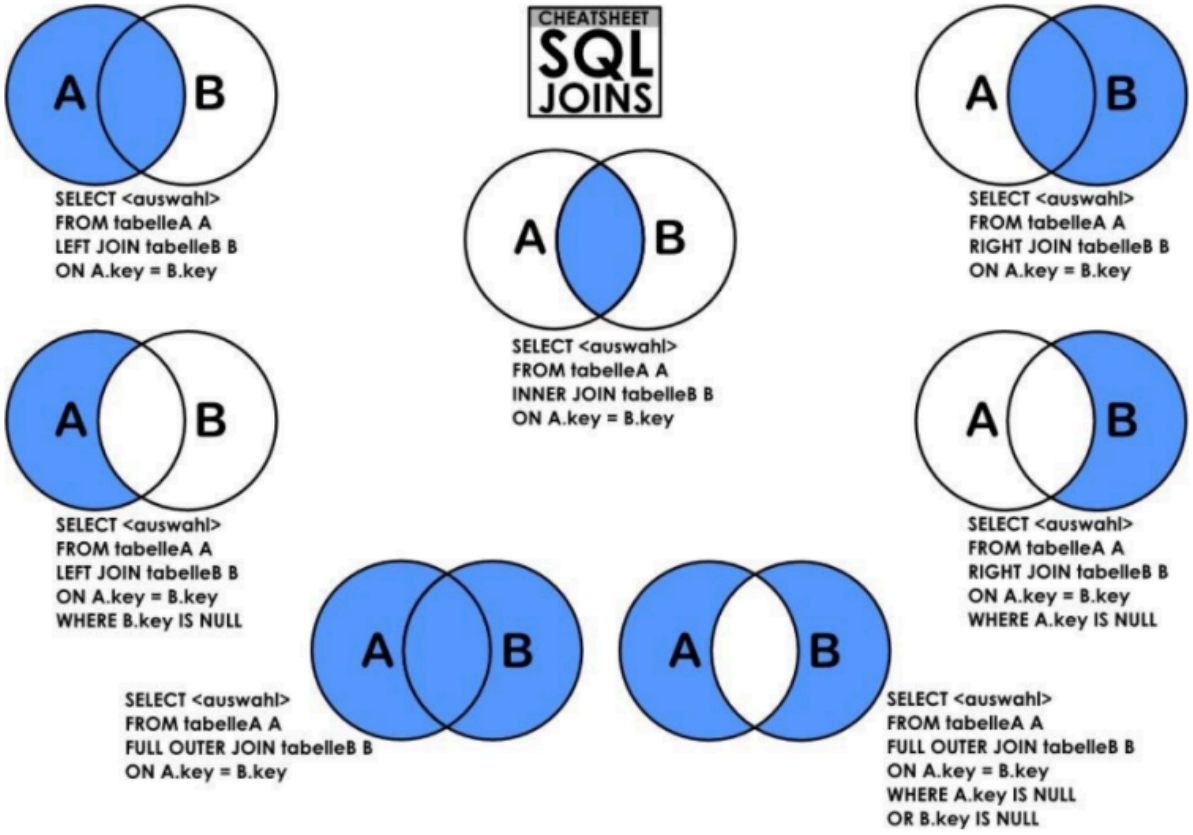
聚合函数	描述
`sum()`	计算某列的总和
`avg()`	计算某列的平均值
`max()`	计算某列的最大值
`min()`	计算某列的最小值
`count()`	计算某列的行数

```
SELECT sum(`num`) FROM `score`;
SELECT avg(`num`) FROM `score`;
SELECT max(`num`) FROM `score`;
SELECT min(`num`) FROM `score`;
SELECT count(`num`) FROM `score`;
```

分组查询

```
-- 分组加group_concat
SELECT `gender`, group_concat(`age`) as ages FROM `student` GROUP BY `gender`;
-- 可以把查询出来的结果根据某个条件来分组显示
SELECT `gender` FROM `student` GROUP BY `gender`;
-- 分组加聚合
SELECT `gender`, count(*) as num FROM `student` GROUP BY `gender`;
-- 分组加条件
SELECT `gender`, count(*) as num FROM `student` GROUP BY `gender` HAVING num > 6;
```

联表查询



INNER JOIN

> **只取两张表有对应关系的记录**

```
SELECT
    cid
FROM
    `course`
INNER JOIN `teacher` ON course.teacher_id = teacher.tid;
```

LEFT JOIN

> 在内连接的基础上**保留左表没有对应关系的记录**

```sql
SELECT
    course.cid
FROM
    `course`
LEFT JOIN `teacher` ON course.teacher_id = teacher.tid;
```

RIGHT JOIN

> 在内连接的基础上**保留右表没有对应关系的记录**

```sql
SELECT
    course.cid
FROM
    `course`
RIGHT JOIN `teacher` ON course.teacher_id = teacher.tid;
```

子查询/合并查询

单行子查询

```sql
select * from course where teacher_id = (select tid from teacher where tname =
'谢小二老师')
```

多行子查询

> 多行子查询即返回多行记录的子查询
>
> **IN 关键字**：运算符可以检测结果集中是否存在某个特定的值，如果检测成功就执行外部的查询。
>
> **EXISTS 关键字**：内层查询语句不返回查询的记录。而是返回一个真假值。如果内层查询语句查询到满足条件的记录，就返回一个真值（`true`），否则，将返回一个假值（`false`）。当返回的值为 `true` 时，外层查询语句将进行查询；当返回的为 `false` 时，外层查询语句不进行查询或者查询不出任何记录。
>
> **ALL 关键字**：表示满足所有条件。使用 *ALL* 关键字时，只有满足内层查询语句返回的所有结果，才可以执行外层查询语句。
>
> **ANY 关键字**：允许创建一个表达式，对子查询的返回值列表，进行比较，只要满足内层子查询中的，任意一个比较条件，就返回一个结果作为外层查询条件。
>
> **在 FROM 子句中使用子查询**：子查询出现在 `from` 子句中，这种情况下将子查询当做一个临时表使用。

```sql
select * from student where class_id in (select cid from course where teacher_id = 2);

select * from student where exists(select cid from course where cid = 5);

SELECT
    student_id,
    sname
FROM
    (SELECT * FROM score WHERE course_id = 1 OR course_id = 2) AS A
      LEFT JOIN student ON A.student_id = student.sid;
```

正则表达式

选项	说明（自动加匹配二字）	例子	匹配值示例
^	文本开始字符	'^b'匹配以字母b开头的字符串	book, big, banana, bike
.	任何单个字符	'b.t'匹配任何b和t之间有一个字符	bit, bat, but, bite
*	0个或多个在它前面的字符	'f*n'匹配字符n前面有任意n个字符f	fn, fan, faan, abcn
+	前面的字符一次或多次	'ba+'匹配以b开头后面紧跟至少一个a	ba, bay, bare, battle
<字符串>	包含指定字符串的文本	'fa'	fan, afa, faad
[字符集合]	字符集合中的任一个字符	'[xz]'匹配x或者z	dizzy, zebra, x-ray, extra
[^]	不在括号中的任何字符	'[^abc]'匹配任何不包含a、b或c的字符串	desk, fox, f8ke
字符串{n}	前面的字符串至少n次	b{2}匹配2个或更多的b	bbb, bbbb, bbbbbb
字符串{n,m}	前面的字符串至少n次，至多m次	b{2,4}匹配最少2个，最多4个b	bb, bbb, bbbb

```sql
SELECT * FROM `teacher` WHERE `tname` REGEXP '^谢';
```

视图

定义

> 视图（`view`）是一种虚拟存在的表，是一个逻辑表，本身并不包含数据。其内容由查询定义。
>
> 基表：用来创建视图的表叫做基表；
>
> 通过视图，可以展现基表的部分数据；

> 视图数据来自定义视图的查询中使用的表，使用视图动态生成；

优点

> 简单：使用视图的用户完全不需要关心后面对应的表的结构、关联条件和筛选条件，对用户来说已经是过滤好的复合条件的结果集。
>
> 安全：使用视图的用户只能访问他们被允许查询的结果集，对表的权限管理并不能限制到某个行某个列，但是通过视图就可以简单的实现。
>
> 数据独立：一旦视图的结构确定了，可以屏蔽表结构变化对用户的影响，源表增加列对视图没有影响；源表修改列名，则可以通过修改视图来解决，不会造成对访问者的影响。

语法

```
CREATE VIEW <视图名> AS <SELECT语句>
```

案例

```
-- 创建视图
-- 查询“c++高级”课程比“音视频”课程成绩高的所有学生的学号;
CREATE VIEW view_test1 AS SELECT
A.student_id
FROM
    (
        SELECT
            student_id,
            num
        FROM
            score
        WHERE
            course_id = 1
    ) AS A -- 12
LEFT JOIN (
    SELECT
        student_id,
        num
    FROM
        score
    WHERE
        course_id = 2
) AS B -- 11
ON A.student_id = B.student_id
WHERE
    A.num >
IF (isnull(B.num), 0, B.num);
```

作用

- 可复用，减少重复语句书写；类似程序中函数的作用；
- 重构利器

> 假如因为某种需求，需要将 *user* 拆成表 *usera* 和表 *userb*；如果应用程序使用 *sql* 语句：`select * from user` 那就会提示该表不存在；若此时创建视图 `create view user as select a.name,a.age,b.sex from usera as a, userb as b where a.name=b.name;`，则只需要更改数据库结构，而不需要更改应用程序；

- 逻辑更清晰，屏蔽查询细节，关注数据返回；
- 权限控制，某些表对用户屏蔽，但是可以给该用户通过视图来对该表操作；

流程控制

IF

```
IF condition THEN
    ...
ELSEIF condition THEN
    ...
ELSE
    ...
END IF
```

CASE

```
-- 相当于switch语句
CASE value
    WHEN value THEN ...
    WHEN value THEN ...
    ELSE ...
END CASE
```

WHILE

```
WHILE condition DO
...
END WHILE;
```

LEAVE

```
-- 相当于break
LEAVE label;
```

示例

```
-- LEAVE语句退出循环或程序块，只能和BEGIN ... END，LOOP，REPEAT，WHILE语句配合使用
-- 创建存储过程
DELIMITER //
CREATE PROCEDURE example_leave(OUT sum INT)
BEGIN
    DECLARE i INT DEFAULT 1;
    DECLARE s INT DEFAULT 0;

    while_label:WHILE i<=100 DO
        SET s = s+i;
        SET i = i+1;
        IF i=50 THEN
            -- 退出WHILE循环
            LEAVE while_label;
        END IF;
    END WHILE;

    SET sum = s;
END
//
DELIMITER ;

-- 调用存储过程
CALL example_leave(@sum);
SELECT @sum;
```

ITERATE

```
-- 相当于 continue
ITERATE label
```

LOOP

```
-- 相当于 while(true) {...}
LOOP
    ...
END LOOP
-- 可以通过LEAVE语句退出循环
```

示例

```
-- 创建存储过程
DELIMITER //
CREATE PROCEDURE example_loop(OUT sum INT)
BEGIN
    DECLARE i INT DEFAULT 1;
    DECLARE s INT DEFAULT 0;

    loop_label:LOOP
        SET s = s+i;
        SET i = i+1;
```

```
        IF i>100 THEN
            -- 退出LOOP循环
            LEAVE loop_label;
        END IF;
    END LOOP;

    SET sum = s;
END
//
DELIMITER ;

-- 调用存储过程
CALL example_loop(@sum);
SELECT @sum;
```

REPEAT

```
-- 相当于 do .. while(condition)
REPEAT
    ...
    UNTIL condition
END REPEAT
```

示例

```
DELIMITER //
CREATE PROCEDURE example_repeat(OUT sum INT)
BEGIN
    DECLARE i INT DEFAULT 1;
    DECLARE s INT DEFAULT 0;

    REPEAT
        SET s = s+i;
        SET i = i+1;

        UNTIL i > 100
    END REPEAT;

    SET sum = s;
END
//
DELIMITER ;

-- 调用存储过程
CALL example_repeat(@sum);
SELECT @sum;
```

触发器

> 触发器是否具备事务性?

定义

> 触发器（*trigger*）是 *MySQL* 提供给程序员和数据分析员来保证数据完整性的一种方法，它是与表事件相关的特殊的存储过程，它的执行不是由程序调用，也不是手工启动，而是由事件来触发，比如当对一个表进行 *DML* 操作（`insert`，`delete`，`update`）时就会激活它执行。

4要素

> 监视对象：`table`
>
> 监视事件：`insert`、`update`、`delete`
>
> 触发时间：`before` , `after`
>
> 触发事件：`insert`、`update`、`delete`

语法

```
CREATE TRIGGER trigger_name
trigger_time trigger_event
ON tbl_name FOR EACH ROW
    [trigger_order]
trigger_body -- 此处写执行语句

-- mysql  c/c++  function   udf   动态库

-- trigger_body: 可以一个语句，也可以是多个语句；多个语句写在 BEGIN ... END 间
-- trigger_time: { BEFORE | AFTER }
-- trigger_event: { INSERT | UPDATE | DELETE }
-- trigger_order: { FOLLOWS | PRECEDES } other_trigger_name
```

准备

```
CREATE TABLE `work` (
    `id` INT PRIMARY KEY auto_increment,
    `address` VARCHAR (32)
) DEFAULT charset = utf8 ENGINE = innoDB;

CREATE TABLE `time` (
    `id` INT PRIMARY KEY auto_increment,
    `time` DATETIME
) DEFAULT charset = utf8 ENGINE = innoDB;

CREATE TRIGGER trig_test1 AFTER INSERT
ON `work` FOR EACH ROW
INSERT INTO `time` VALUES(NULL,NOW());
```

NEW 和 OLD

> 在 `INSERT` 型触发器中，`NEW` 用来表示将要（`BEFORE`）或已经（`AFTER`）插入的新数据；
>
> 在 `DELETE` 型触发器中，`OLD` 用来表示将要或已经被删除的原数据；

> 在 `UPDATE` 型触发器中，`OLD` 用来表示将要或已经被修改的原数据，`NEW` 用来表示将要或已经修改为的新数据；

```
NEW.columnName （columnName为相应数据表某一列名）
OLD.columnName （columnName为相应数据表某一列名）
```

案例

> 在下订单的时候，对应的商品的库存量要相应的减少，即买几个商品就减少多少个库存量。

准备

```
CREATE TABLE `goods` (
    `id` INT PRIMARY KEY auto_increment,
    `name` VARCHAR (32),
    `num` SMALLINT DEFAULT 0
);

CREATE TABLE `order` (
    `id` INT PRIMARY KEY auto_increment,
    `goods_id` INT,
    `quantity` SMALLINT COMMENT '下单数量'
);

INSERT INTO goods VALUES (NULL, 'C++', 40);

INSERT INTO goods VALUES (NULL, 'C', 63);

INSERT INTO goodS VALUES (NULL, 'mysql', 87);

INSERT INTO `order` VALUES (NULL, 1, 3);

INSERT INTO `order` VALUES (NULL, 2, 4);
```

需求1

> 客户修改订单购买的数量，在原来购买数量的基础上减少2个；

```
-- delimiter
-- delimiter是mysql分隔符，在mysql客户端中分隔符默认是分号 ;。如果一次输入的语句较多，并且语句中间有分号，这时需要重新指定一个特殊的分隔符。通常指定 $$ 或 ||
delimiter //
CREATE TRIGGER trig_order_1 AFTER INSERT
ON `order` FOR EACH ROW
BEGIN
    UPDATE goods SET num = num - 2 WHERE id = 1;
END//
delimiter ;
INSERT
```

需求2

> 客户修改订单购买的数量，商品表的库存数量自动改变；

```
delimiter //
CREATE TRIGGER trig_order_2 BEFORE UPDATE
ON `order` FOR EACH ROW
BEGIN
    UPDATE goods SET num=num+old.quantity-new.quantity WHERE id = new.goods_id;
END
//
delimiter ;
-- 测试
UPDATE `order` SET quantity = quantity+2 WHERE id = 1;
```

存储过程

定义

> *SQL* 语句需要先编译然后执行，而存储过程（*Stored Procedure*）是一组为了完成特定功能的 *SQL* 语句集，经编译后存储在数据库中，用户通过指定存储过程的名字并给定参数（如果该存储过程带有参数）来调用执行它。
>
> 存储过程是可编程的函数，在数据库中创建并保存，可以由 *SQL* 语句和控制结构组成。当想要在不同的应用程序或平台上执行相同的函数，或者封装特定功能时，存储过程是非常有用的。数据库中的存储过程可以看做是对编程中面向对象方法的模拟，它允许控制数据的访问方式。

特点

- 能完成较复杂的判断和运算进行有限的编程
- 可编程行强，灵活
- SQL 编程的代码可重复使用
- 执行的速度相对快一些
- 减少网络之间的数据传输，节省开销

语法

```
CREATE PROCEDURE  过程名([[IN|OUT|INOUT] 参数名 数据类型[,[IN|OUT|INOUT] 参数名 数据类型…]]) [特性 ...] 过程体
```

> 存储过程根据需要可能会有**输入、输出、输入输出参数**，如果有多个参数用 "," 分割开。
>
> `MySQL` 存储过程的参数用在存储过程的定义，共有三种参数类型 `IN`，`OUT`，`INOUT`。
>
> `IN`：参数的值必须在调用存储过程时指定，在存储过程中修改该参数的值不能被返回，可以设置默认值
>
> `OUT`：该值可在存储过程内部被改变，并可返回
>
> `INOUT`：调用时指定，并且可被改变和返回
>
> 过程体的开始与结束使用 `BEGIN` 与 `END` 进行标识。

案例

```
DELIMITER //
      CREATE PROCEDURE proc_test1()
BEGIN
      SELECT current_time();
      SELECT current_date();
END
//
DELIMITER ;
call proc_test1();
```

`IN`

```
DELIMITER //
CREATE PROCEDURE proc_in_param (IN p_in INT)
BEGIN
    SELECT
        p_in ;
    SET p_in = 2 ; SELECT
        p_in ;
    END ;//
DELIMITER ;


-- 调用
SET @p_in = 1;

CALL proc_in_param (@p_in);

-- p_in虽然在存储过程中被修改，但并不影响@p_id的值
SELECT @p_in;=1
```

`OUT`

```
DELIMITER //
  CREATE PROCEDURE proc_out_param(OUT p_out int)
    BEGIN
      SELECT p_out;
      SET p_out=2;
      SELECT p_out;
    END;
//
DELIMITER ;
-- 调用
SET @p_out=1;
CALL proc_out_param(@p_out);
SELECT @p_out; -- 2
```

`INOUT`

```
DELIMITER //
  CREATE PROCEDURE proc_inout_param(INOUT p_inout int)
    BEGIN
      SELECT p_inout;
      SET p_inout=2;
      SELECT p_inout;
    END;
//
DELIMITER ;
#调用
SET @p_inout=1;
CALL proc_inout_param(@p_inout) ;
SELECT @p_inout; -- 2
```

游标

> 游标是针对行操作的，对从数据库中 `select` 查询得到的结果集的每一行可以进行分开的独立的相同或者不相同的操作。
>
> 对于取出多行数据集，需要针对每行操作；可以使用游标；游标常用于存储过程、函数、触发器、事件；
>
> 游标相当于迭代器

定义游标

```
DECLARE cursor_name CURSOR FOR select_statement;
```

打开游标

```
OPEN cursor_name;
```

取游标数据

```
FETCH cursor_name INTO var_name[,var_name,.....]
```

关闭游标

```
CLOSE curso_name;
```

释放

```
DEALLOCATE cursor_name;
```

设置游标结束标志

```
DECLARE done INT DEFAULT 0;
DECLARE CONTINUE HANDLER FOR NOT FOUND
SET done = 1; -- done 为标记为
```

案例

```
CREATE PROCEDURE proc_while (
    IN age_in INT,
    OUT total_out INT
)
BEGIN
-- 创建 用于接收游标值的变量
DECLARE p_id,p_age,p_total INT ;
DECLARE p_sex TINYINT ;
-- 注意:接收游标值为中文时,需要给变量 指定字符集utf8
DECLARE p_name VARCHAR (32) CHARACTER SET utf8 ; -- 游标结束的标志
DECLARE done INT DEFAULT 0 ; -- 声明游标
DECLARE cur_teacher CURSOR FOR SELECT
    teacher_id,
    teacher_name,
    teacher_sex,
    teacher_age
FROM
    teacher
WHERE
    teacher_age > age_in ; -- 指定游标循环结束时的返回值
DECLARE CONTINUE HANDLER FOR NOT found
SET done = 1 ; -- 打开游标
OPEN cur_teacher ; -- 初始化 变量
SET p_total = 0 ; -- while 循环
WHILE done != 1 DO
    FETCH cur_teacher INTO p_id,
    p_name,
    p_sex,
    p_age ;
IF done != 1 THEN

SET p_total = p_total + 1 ;
END
IF ;
END
WHILE ; -- 关闭游标
CLOSE cur_teacher ; -- 将累计的结果复制给输出参数
SET total_out = p_total ;
END//
delimiter ;

-- 调用
SET @p_age =20;
CALL proc_while(@p_age, @total);
SELECT @total;
```

权限管理

创建用户

```
CREATE USER username@host IDENTIFIED BY password;
```

> `host` 指定该用户在哪个主机上可以登陆，如果是本地用户可用 `localhost`，如果想让该用户可以从任意远程主机登陆，可以使用通配符 `%`;

授权

```
GRANT privileges ON databasename.tablename TO 'username'@'host' WITH GRANT
OPTION;
```

> `privileges`：用户的操作权限，如 `SELECT`，`INSERT`，`UPDATE` 等，如果要授予所的权限则使用 ALL;
>
> `databasename.tablename` 如果是 `*.*` 表示任意数据库以及任意表;
>
> `WITH GRANT OPTION` 这个选项表示该用户可以将自己拥有的权限授权给别人。注意：经常有人在创建操作用户的时候不指定 `WITH GRANT OPTION` 选项导致后来该用户不能使用 `GRANT` 命令创建用户或者给其它用户授权。
> 如果不想这个用户有这个 grant 的权限，则不要加该 `WITH GRANT OPTION` 选项;

对视图授权

```
GRANT select, SHOW VIEW ON `databasename`.`tablename`  to 'username'@'host';
```

刷新权限

```
-- 修改权限后需要刷新权限
FLUSH PRIVILEGES;
```

远程连接

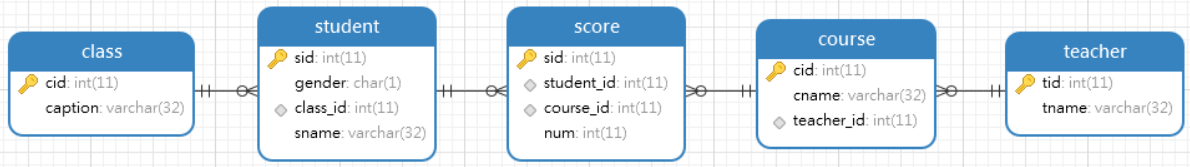
修改配置

> 注释 `mysqld.cnf` 中 `bind-address`，修改 `mysql.user` 表，然后重启 mysql

```
-- mysqld.cnf
# vi /etc/mysql/mysql.conf.d/mysqld.cnf
#bind-address=127.0.0.1
```

```
-- 修改user表
select `user`, `host` from `mysql`.`user`;
update user set host='%' where user='root';
```

作业



- 查询平均成绩大于60分的同学的学号和平均成绩;

```
SELECT
    student_id,
    AVG(num) AS avg_num
FROM
    score
GROUP BY
    student_id
HAVING
    avg_num > 60;
```

- 查询 'c++高级架构' 课程比 '音视频' 课程成绩高的所有学生的学号;

```
SELECT
    A.student_id
FROM
    (
        (
            SELECT
                student_id,
                num
            FROM
                score
            WHERE
                course_id = (
                    SELECT
                        cid
                    FROM
                        course
                    WHERE
                        cname = 'c++高级架构'
                )
        ) AS A
        INNER JOIN (
            SELECT
                student_id,
                num
            FROM
                score
            WHERE
                course_id = (
                    SELECT
                        cid
                    FROM
                        course
                    WHERE
```

```
                        cname = '音视频'
                )
        ) AS B ON A.student_id = B.student_id
    )
WHERE
    A.num > B.num;

# 如果只报了c++高级，也满足条件，那么 sql 语句如下：

SELECT
    A.student_id
FROM
    (
        (
            SELECT
                student_id,
                num
            FROM
                score
            WHERE
                course_id = (
                    SELECT
                        cid
                    FROM
                        course
                    WHERE
                        cname = "c++高级架构"
                )
        ) AS A
        LEFT JOIN (
            SELECT
                student_id,
                num
            FROM
                score
            WHERE
                course_id = (
                    SELECT
                        cid
                    FROM
                        course
                    WHERE
                        cname = "音视频"
                )
        ) AS B ON A.student_id = B.student_id
    )
WHERE
    A.num > IFNULL(B.num, 0);
```

- 查询所有同学的学号、姓名、选课数、总成绩；

```
SELECT
    sid,
    sname,
    A.cnt,
    A.sum_sco
```

```
FROM
    student
LEFT JOIN (
    SELECT
        student_id,
        count(course_id) AS cnt,
        sum(num) AS sum_sco
    FROM
        score
    GROUP BY
        student_id
) AS A ON a.student_id = sid;
```

- 查询没学过 '谢小二' 老师课的同学的学号、姓名;

```
SELECT
    sid,
    sname
FROM
    student
WHERE
    sid NOT IN (
        SELECT DISTINCT
            student_id
        FROM
            score
        WHERE
            course_id IN (
                SELECT
                    cid
                FROM
                    course
                left join
                    teacher
                ON
                    course.teacher_id = teacher.tid
                        WHERE
                            teacher.tname = '谢小二老师'
            )
    );

# 将 in 和 not in 优化为 联表查询
SELECT
    sid,
    sname
FROM
    student
LEFT JOIN (
    SELECT DISTINCT
        A.student_id
    FROM
        (
            SELECT
                student_id,
                course_id
            FROM
```

```
                score
        ) AS A
    LEFT JOIN (
        SELECT
            cid
        FROM
            course
        WHERE
            teacher_id = (
                SELECT
                    tid
                FROM
                    teacher
                WHERE
                    tname = '谢小二老师'
            )
    ) AS B ON A.course_id = B.cid
    WHERE
        B.cid IS NULL
) AS c ON student.sid = c.student_id;
```

- 查询学过课程编号为 '1' 并且也学过课程编号为 '2' 的同学的学号、姓名；

```
# 既可以用分组也可以联合查询
SELECT
    sid,
    sname
FROM
    student
RIGHT JOIN (
    SELECT
        student_id
    FROM
        score
    WHERE
        course_id = 1
    OR course_id = 2
    GROUP BY
        student_id
    HAVING
        count(course_id) = 2
) AS A ON A.student_id = sid;
```

- 查询学过 '谢小二' 老师所教的所有课的同学的学号、姓名；

```
SELECT
    sid,
    sname
FROM
    student
RIGHT JOIN (
    SELECT DISTINCT
        student_id
    FROM
        score
```

```
    WHERE
        course_id IN (
            SELECT
                cid
            FROM
                course
            LEFT JOIN teacher ON course.teacher_id = teacher.tid
            WHERE
                teacher.tname = '谢小二老师'
        )
) AS A ON student.sid = A.student_id;
```

- 查询有课程成绩小于 60 分的同学的学号、姓名;

```
SELECT
    sid,
    sname
FROM
    student
RIGHT JOIN (
        SELECT DISTINCT
            student_id
        FROM
            score
        WHERE
            num < 60
    ) as A ON A.student_id = sid;
```

- 查询没有学全所有课的同学的学号、姓名;

```
SELECT
    sid,
    sname
FROM
    student
RIGHT JOIN (
    SELECT
        student_id
    FROM
        score
    GROUP BY
        student_id
    HAVING
        count(course_id) < (SELECT count(1) FROM course)
) AS A ON sid = A.student_id;
```

- 查询至少有一门课与学号为 '1' 的同学所学相同的同学的学号和姓名;

```
SELECT
    sid,
    sname
FROM
    student
RIGHT JOIN (
    SELECT DISTINCT
```

```
        student_id
    FROM
        score
    WHERE
        student_id != 1
    AND course_id IN (
        SELECT
            course_id
        FROM
            score
        WHERE
            student_id = 1
    )
) AS A ON A.student_id = student.sid;

# 如果添加一个条件，需要知道有多少门相同
```

- 查询至少学过学号为 '1' 同学所有课的其他同学学号和姓名；

```
SELECT
    sid,
    sname
FROM
    student
RIGHT JOIN (
    SELECT
        student_id,
        COUNT(1) AS cnt
    FROM
        score
    WHERE
        student_id != 1
    AND course_id IN (
        SELECT
            course_id
        FROM
            score
        WHERE
            student_id = 1
    )
    GROUP BY
        student_id
    HAVING
        cnt >= (
            SELECT
                count(1)
            FROM
                score
            WHERE
                student_id = 1
        )
) AS A ON A.student_id = student.sid;
```