# MySQL从入门到放弃

**MySQL从入门到放弃**

- 数据库基础
  - 数据库技术的发展
  - 数据模型
    - 关系数据库的规范化
  - 数据库的体系结构
  - MySQL介绍
  - MySQL存储引擎
- 操作数据表
  - 数据表的创建与查看
  - 修改表结构
    - 添加新字段
    - 修改字段类型
    - 修改字段名（某个字段重新配置）
    - 删除字段
    - 修改表名
    - 复制表
    - 删除表
- MySQL运算
  - 运算符
    - 正则匹配：REGEXP
  - 流程控制语句（进阶）
    - 示例
- 表数据的增删改操作
  - 增
    - 添加一条数据
    - 添加多条数据
    - 使用 `insert...set` 插入数据
    - 插入查询结果
  - 删
    - 通过truncate table语句删除数据
  - 改
- 数据查询
  - 一些基本操作
    - 带 `like` 的字符匹配查询
    - 关键字 `group by` 分组查询
  - 聚合函数查询
  - 连接查询
    - 内连接查询
    - 外连接查询
      - 左外连接
      - 右外连接
  - 子查询

# 数据库基础

## 数据库技术的发展

数据管理技术先后经历了：人工管理 --> 文件系统 --> 数据库系统 三个阶段

20世纪60年代后期以来，计算机应用于管理系统，而且规模越来越大，应用越来越广泛，数据量急剧增长，对共享功能的要求越来越强烈，这样使用文件系统管理数据已经不能满足要求，于是出现了数据库系统来统一管理数据。数据库系统的出现，满足了多用户、多应用共享数据的需求，比文件系统具有明显的优点，标志着数据管理技术的飞跃

## 数据模型

数据模型是数据库系统的核心与基础，是关于描述数据与数据之间的联系、数据的语义、数据一致性约束的概念型工具的集合。数据模型通常由数据结构、数据操作和完整性约束3部分组成：

1. 数据结构：是对系统静态特征的描述，描述对象包括数据的类型、内容、性质和数据之间的相互关系
2. 数据操作：是对系统动态特征的描述，是对数据库各种对象实例的操作
3. 完整性约束：是完整性规则的集合，它定义了给定数据模型中数据及其联系所具有的制约和依存规则

常见的数据库数据模型主要有三种：层次模型、网状模型和关系模型

关系模型：以二维表来描述数据。在关系模型中，每个表有多个字段列和记录行，每个字段列有固定的属性（数字、字符、日期等）。关系模型数据结构简单、清晰、具有很高的数据独立性，是目前主流的数据库数据模型。关系模型的基本术语如下：

- 关系：一个二维表就是一个关系
- 元组：二维表中的一行，即表中的记录
- 属性：二维表中的一列，用类型和值表示
- 域：每个属性取值的变化范围，如性别的域为{男，女}
- 实体完整性约束：约束关系的主键中属性值不能为空值
- 参照完整性约束：关系之间的基本约束

### - 关系数据库的规范化

关系数据库中的每一个关系都要满足一定的规范，根据满足规范的条件不同可以分为五个等级

**第一范式（1NF）**

- 数据组的每个属性只可以包含一个值
- 关系中的每个数组必须包含相同数量的值
- 关系中的每个数组一定不能相同

| 学号 | 姓名 | 性别 | 年龄 | 班级 |
| --- | --- | --- | --- | --- |
| 9527 | 东方 | 男 | 20 | 计算机系3班 |

上面这张表中“班级”列的信息就不是单一的，是可以再分的，不符合第一范式

| 学号 | 姓名 | 性别 | 年龄 | 系别 | 班级 |
| --- | --- | --- | --- | --- | --- |
| 9527 | 东方 | 男 | 20 | 计算机系 | 3班 |

这张表就是符合第一范式的学生信息表，它将原“班级”列的信息拆分到“系别”列和“班级”列中

**第二范式（2NF）**

第二范式是在第一范式的基础上建立起来的，满足第二范式必先满足第一范式。第二范式要求数据库表中的每个实体必须可以被唯一地区分。在学生信息表中，由于每个学生编号都是唯一的，因此每个学生可以被唯一的区分，这个唯一属性列被称为主关键字或主键。第二范式要求实体的属性必须完全依赖于主关键字，即不能存在仅依赖主关键字一部分的属性，如果存在，那么这个属性和主关键字的这一部分应该分离出来形成一个新的实体

这里以“员工工资信息表”为例，若以（员工编码、岗位）为组合关键字（复合主键），就会存在以下决定关系：

```
（员工编码，岗位）-->（决定）（姓名、年龄、学历、基本工资、绩效工资、奖金）
```

在上面的决定关系中，还可以进一步拆分为如下两种决定关系：

```
（员工编码）-->（决定）（姓名、年龄、学历）
（岗位）-->（决定）（基本工资）
# 真正需要复合主键的属性只有一下几种
（员工编码，岗位）-->（决定）（绩效工资、奖金）
```

根据上面的关系对照，可以把信息表更改为如下3张表：

```
员工档案表：EMPLOYEE（员工编码、姓名、年龄、学历）
岗位工资表：QUARTERS（岗位、基本工资）
员工工资表：PAY（员工编码、岗位、绩效工资、奖金）
```

**第三范式（3NF）**

第三范式要求关系表中不存在非关键字列对任意候选关键字列的传递函数依赖。以员工信息表为例

```
（员工编码）-->（决定）（姓名、年龄、部门编码、部门经理）
```

上面这个关系表是符合第二范式的（单主键肯定是满足第二范式的），但它不符合第三范式，因为该关系表内部隐含着如下决定关系：

```
（员工编码）-->（决定）（部门编码）-->（决定）（部门经理）
```

上面的关系表存在非关键字段“部门经理”对关键字段“员工编码”的传递函数依赖。对于上面这种关系，可以把这个关系表更改为如下两个关系表：

```
员工信息表：EMPLOYEE（员工编码、姓名、年龄、部门编码）
部门信息表：DEPARTMENT（部门编码、部门经理）
```

对于关系型数据库的设计，理想的设计目标是按照“规范化”原则存储数据，因为这样做能够消除数据冗余、更新异常、插入异常和删除异常

**关系数据库的设计原则**

- 数据库内数据文件的数据组织应获得最大限度的共享、最小的冗余度、消除数据及数据依赖关系中的冗余部分，使依赖于同一个数据模型的数据达到有效的分离
- 保证输入、修改数据时数据的一致性与正确性
- 保证数据与使用数据的应用程序之间的高度独立性

**实体与关系**

实体是指客观存在并可相互区别的事物，实体既可以是实际的事物，也可以是抽象的概念或关系。实体之间有三种关系，分别如下：

- 一对一关系：指表A中的一条记录在表B中有且只有一条相匹配的记录。在一对一关系中，大部分相关信息都在一个表中
- 一对多关系：指表A中的行可以在表B中有许多匹配行。但是表B中的行只能在表A中有一个匹配行
- 多对多关系：指关系中的每个表的行在相关表中具有多个匹配行。在数据库中，多对多关系的建立是依靠第三张表来实现的，连接表包含相关的两个表的主键列，然后从两个相关表的主键列分别创建与连接表中的匹配列的关系

  可以给一个多对多的例子：治理门户中的治理计划和治理规则的对应关系就是多对多的，即：一条治理计划可以对应多个规则，每个规则也可以同时出现在不同的治理计划中

  我们会有一张计划表 (govern_plan)，也有一张规则表 (govern_rule)，分别存储计划和规则的相关属性，但我们如何存储二者的对应关系呢?

  为了存储这种多对多关系，我们只能在引入一张表 (govern_plan_rule_relation)，这张表的作用只有一个，就是表示二者的对应关系，这样，无论我们是希望根据计划找规则，还是根据规则找用到它的计划，都可以通过这张表实现

优秀提问1：既然两张表是一对一的关系，为什么不直接放在一张表里?

**答：给一个例子——考虑今日头条用数据库存新闻的场景。用户发的文章内容往往比较大，但标题往往会比较短。如果都放到一张表里，那么这张表会非常大，而查询大表是非常耗时的。而如果把文章内容单独抽出来存在一张表里，那么存放标题的表就会变得非常小，查询效率提升会特别明显。对于小字段小数据量的数据库来说，放到一张表里确实可以，但是对于大字段的数据库，要添加删除修改某一个字段，由于数据库大，字段多，数据库找到这个字段耗时久，造成锁表等问题。大数据量的表分离存储对于索引创建、读写分离、分割等操作都有优势**

## 数据库的体系结构

数据库的三级模式结构：模式、外模式、内模式

- 模式也称为逻辑模式或概念模式，是数据库中全体数据的逻辑结构和特征的描述，是所有用户的公共数据视图。一个数据库只有一个模式，模式处于三级结构的中间层
- 外模式：也称用户模式，是数据库用户（包括应用程序猿和最终用户）能够看见和使用的局部数据的逻辑结构和特征的描述，是数据库用户的数据视图，是与某一应用有关的数据的逻辑表示。外模式是模式的子集，一个数据库可以有多个外模式

- 内模式：也称存储模式，是数据物理结构和存储方式的描述，是数据在数据库内部的表示方式。一个数据库只有一个内模式

## MySQL介绍

MySQL是目前最为流行的开放源代码的数据库管理系统，目前属于Oracle公司。MySQL数据库可以称得上是目前运行速度最快的SQL数据库，是一个真正的多用户、多线程SQL数据库服务器。MySQL主要目标是快捷、便捷和易用。MySQL被广泛地应用在Internet上的中小型网站中。由于其体积小、速度快、总体拥有成本低，尤其在开发源代码这一特点，成为多数中小型网站为了降低网站总体拥有成本而选择的重要指标

### MySQL的特性

- 使用C和C++语言编写，并使用了多种编译器进行测试，保证源代码的可移植性
- 支持AIX、FreeBSD、HP-UX、Linux、Mac OS、Windows等多种操作系统
- 为多种编程语言提供了API。包括C、C++、Python、Java、PHP等
- 支持多线程，充分利用CPU资源
- 优化的SQL查询算法，有效地提高查询速度
- 提供TCP/IP、ODBC、JDBC等多种数据库连接途径
- 提供用于管理、检查、优化数据库操作的管理工具
- 可以处理拥有上千万条记录的大型数据库

## MySQL存储引擎

存储引擎其实就是如何存储数据、如何为存储的数据建立索引和如何更新、查询数据等技术的实现方法。因为在关系数据库中数据是以表的形式存储的，所以存储引擎也可以成为表类型。在Oracle和SQL Server等数据库中只有一种存储引擎，所有数据存储管理机制都是一样的；而MySQL数据库提供了多种存储引擎，用户可以根据不同的需求为数据表选择不同的存储引擎，用户也可以根据需要编写自己的存储引擎

在MySQL中，可以用 `show engines` 语句查询MySQL中支持的存储引擎

```
show engines;
```

打印出的结果比较乱，就不给出了。查询默认引擎的命令：

```
show variables like '%storage_engine%';
+---------------------------------+-----------+
| Variable_name                   | Value     |
+---------------------------------+-----------+
| default_storage_engine          | InnoDB    |
| default_tmp_storage_engine      | InnoDB    |
| disabled_storage_engines        |           |
| internal_tmp_mem_storage_engine | TempTable |
+---------------------------------+-----------+
4 rows in set (0.01 sec)
```

MySQL的存储引擎一般默认为InnoDB

### InnoDB存储引擎

InnoDB给MySQL的表提供了事务、回滚、崩溃修复能力和多版本并发控制的事务安全。InnoDB是第一个提供外键约束的表引擎。InnoDB存储引擎中自持自动增长列AUTO_INCREMENT。自动增长列的值不能为空，且值必须唯一。MySQL规定自增列必须为主键。在插入时，如果自动增长列不输入值，则插入的值为自动增长后的值；如果输入的是0或空（NULL），则插入的值也是增长后的值；如果插入一个确定的值，且该值在之前没有出现过，则可以直接插入

InnoDB存储引擎中支持外键，外键所在的表为子表，外键所依赖的表为父表。父表中被子表外键所关联的字段必须为主键。当删除、更新父表信息时，子表也必须有相应的改动

InnoDB存储引擎的优势在于提供了良好的事务管理、崩溃修复能力和并发控制。缺点是其读写速率稍差，占用的数据空间相对比较大。InnoDB表是如下情况的理想引擎：

- 更新密集的表：InnoDB特别适合处理多重并发的更新请求
- 事务：InnoDB是唯一支持事务的标准，这是管理敏感数据的必须软件
- 自动灾难恢复：InnoDB表能自动从灾难中恢复，MyISAM恢复的过程要长很多

Oracle的InnoDB存储引擎广泛应用于基于MySQL的Web、电子商务、金融系统、健康护理以及零售应用。因为InnoDB可提供高效的ACID独立性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）兼容事务处理能力，以及独特的高性能和具有可扩展性的架构要素

**MyISAM存储引擎**

MyISAM存储引擎是MySQL中常见的存储引擎，曾是MySQL的默认存储引擎。MyISAM存储引擎的表存储成三个文件。文件的名字与表名相同，扩展名包括frm（存储表的结构）、MYD（存储数据）、MYI（存储索引）

MyISAM存储引擎的优势在于占用空间小、处理速度快；缺点是不支持事物的完整性和并发性

# 操作数据表

## 数据表的创建与查看

为了系统学习，我们先创建一个新的数据库：db_admin

```
mysql> create database db_admin;
Query OK, 1 row affected (0.00 sec)
mysql> use db_admin;
Database changed
```

创建一个用户表：tb_admin

```
# tb_admin (id, user, password, create_time)
create table tb_admin (
    id int auto_increment primary key,
    user varchar(30) not null,
    password varchar(30) not null,
    createtime datetime
);
Query OK, 0 rows affected (0.01 sec)
```

可以使用 `desc` 关键字查看表结构

```
desc tb_admin
+------------+-------------+------+-----+---------+----------------+
| Field      | Type        | Null | Key | Default | Extra          |
+------------+-------------+------+-----+---------+----------------+
| id         | int         | NO   | PRI | NULL    | auto_increment |
| user       | varchar(30) | NO   |     | NULL    |                |
| password   | varchar(30) | NO   |     | NULL    |                |
| createtime | datetime    | YES  |     | NULL    |                |
+------------+-------------+------+-----+---------+----------------+
4 rows in set (0.01 sec)
```

### 修改表结构

- 添加新字段

```
alter table tb_admin add email varchar(50) not null
```

- 修改字段类型

```
alter table tb_admin modify user varchar(40)
```

- 修改字段名（某个字段重新配置）

```
alter table tb_admin change column user username varchar(50) not null
```

- 删除字段

```
alter table tb_admin drop column email
```

- 修改表名

```
# 写法1
alter table tb_admin rename as tb_administrator
# 写法2
rename table tb_administrator to tb_admin
```

- 复制表

```
# 复制表结构
create table duplicate_db like tb_admin
```

```
# 复制表内容
create table duplicate_db_content as select * from tb_admin
```

- 删除表

```
drop table if exists duplicate_db
```

# MySQL运算

## 运算符

⚠️MySQL中，AND的优先级要大于OR

| 优先级 | 运算符 |
| --- | --- |
| 1 | ! |
| 2 | ~ |
| 3 | ^ |
| 4 | *, /, DIV, %, MOD |
| 5 | +, - |
| 6 | >>, << |
| 7 | & |
| 8 | \| |
| 9 | =, <=>, <, <=, >, >=, !=, in, is, null, like, regexp |
| 10 | between and, case, when, then, else |
| 11 | not |
| 12 | &&, and |
| 13 | \|\|, or, nor |
| 14 | := |

- 正则匹配：REGEXP

```
# 检测字段govern_objective是否以'o'开头
select govern_objective, govern_objective regexp'o' from fort_govern_plan
```

```
# 检测字段create_time是否满足该正则表达式
select create_time, create_time regexp'.*02-26.*' from fort_govern_plan
```

## 流程控制语句（进阶）

- 示例

假设我们有一张表，保存用户的姓名和性别，其中性别用int类型表示，1为男，2为女，我们先建表如下：

```
create table t_user (
    id int primary key auto_increment comment 'id',
    sex tinyint not null default 1 comment 'sex',
    name varchar(16) not null default '' comment 'name'
) comment 'user table';

insert into t_user values (1, 1, '路人甲Java'), (2, 1, '张学友'), (3, 2, '王祖贤'), (4, 1, '郭富城'), (5, 2, '李嘉欣');
```

```
select * from t_user;

+----+-----+---------------+
| id | sex | name          |
+----+-----+---------------+
|  1 |   1 | 路人甲Java    |
|  2 |   1 | 张学友        |
|  3 |   2 | 王祖贤        |
|  4 |   1 | 郭富城        |
|  5 |   2 | 李嘉欣        |
+----+-----+---------------+
5 rows in set (0.00 sec)
```

现在我们有这样一个需求，我们要通过查询t_user表返回：编号、性别（男、女）、姓名（即我们需要将数据库中表示性别的数字转化成字符返回）

在MySQL中不要乱用，因为在MySQL中有特别的意义，尽量避免使用

```
# 解法1: If条件判断
select id as 编号, if(sex = 1, '男', '女') as 性别, name as 姓名 from t_user;
```

```
# 解法2: Switch
select id as 编号,
(
    case sex
        when 1 then '男'
        when 2 then '女'
    end
) as 性别, name as 姓名
from t_user;
```

```
# 使用存储过程，传入男女，改成1，2存入数据表
delimiter $
create procedure proc1(id int, sex_str varchar(8), name varchar(16))
    begin
    declare v_sex tinyint unsigned;
    case sex_str
        when '男' then
            set v_sex = 1;
        when '女' then
            set v_sex = 2;
    end case;
    insert into t_user values (id, v_sex, name);
end $
```

```
# 调用存储过程
call proc1(6, '男', '郭富城')
```

```
+----+-----+---------------+
| id | sex | name          |
+----+-----+---------------+
|  1 |   1 | 路人甲Java    |
|  2 |   1 | 张学友        |
|  3 |   2 | 王祖贤        |
|  4 |   1 | 郭富城        |
|  5 |   2 | 李嘉欣        |
|  6 |   1 | 郭富城        |
+----+-----+---------------+
6 rows in set (0.00 sec)
```

```
/* 使用函数之前，确保设置一下配置 */
set global log_bin_trust_function_creators=TRUE;

/* 使用函数 返回男女 */
delimiter //
create function sex_function(sex tinyint unsigned)
returns varchar(8)
begin
declare v_sex varchar(8);
case sex
when 1 then set v_sex := '男';
else set v_sex := '女';
end case;
return v_sex;
end //
delimiter ;
```

```
select sex, sex_function(sex) as 性别, name from t_user
```

新的需求：现在我们要新增和更新用户数据，但我们不知道我穿传入的数据是不是已经存在了（如果传入id=3，那么只需要更新，不需要新建），我们可以在后端先进行这个判断，但我们也可以直接利用MySQL的流程控制解决这个问题

```
drop procedure if exists proc2;
delimiter //
create procedure proc2(v_id int, v_sex varchar(8), v_name varchar(16), out result tinyint)
begin
declare v_count tinyint default 0;
select count(id) into v_count from t_user where id = v_id;
if v_count > 0 then
begin
declare lsex tinyint;
select if(v_sex='男', 1, 2) into lsex;
update t_user set sex = lsex, name = v_name where id = v_id;
select row_count() into result;
end;
else
begin
declare lsex tinyint;
select if (v_sex='男', 1, 2) into lsex;
insert into t_user values (v_id, lsex, v_name);
select 0 into result;
end;
end if;
end //
delimiter ;
```

**循环控制语句**

```
delete from test1;
drop procedure if exists proc3;
delimiter //
create procedure proc3(v_count int)
  begin
    declare i int default 1;
    a: while i <= v_count do
      insert into test1 values (i);
      set i = i + 1
    end
  end //
delimiter ;
```

**添加leave控制语句**

```
drop procedure if exists proc4;
delimiter //
create procedure proc4(v_count int)
  begin
    declare i int default 1;
    a: while i <= v_count do
      insert into test1 values (i);
      if i = 10 then
        leave a;
      end if;
      set i = i + 1
    end while;
  end //
delimiter ;
```

## 表数据的增删改操作

先查看一下目标表结构：

```
+------------+-------------+------+-----+---------+----------------+
| Field      | Type        | Null | Key | Default | Extra          |
+------------+-------------+------+-----+---------+----------------+
| id         | int         | NO   | PRI | NULL    | auto_increment |
| username   | varchar(50) | NO   |     | NULL    |                |
| password   | varchar(30) | NO   |     | NULL    |                |
| createtime | datetime    | YES  |     | NULL    |                |
+------------+-------------+------+-----+---------+----------------+
4 rows in set (0.00 sec)
```

### 增

- 添加一条数据

```
insert into tb_admin values (1, 'wzh', '123456', now())
```

```
insert into tb_admin (username, password, createtime) values ('wzhdx', '111', now())
```

- 添加多条数据

```
insert into tb_admin values (7, 'aaa', 'aaa', now()), (8, 'bbb', 'bbb', now())
```

```
insert into tb_admin (username, password, createtime) values ('ccc', 'ccc', now()), ('ddd',
'ddd', now())
```

- 使用 `insert...set` 插入数据

```
insert into tb_admin set username='eee', password='eee', createtime=now()
```

- 插入查询结果

```
insert into tb_admin (username, password) select user, pass from tb_mrbook
```

## 删

**在实际应用中，执行删除操作时，执行删除的条件一般应该为数据的id，而不是具体的字段，这样可以避免一些错误发生**

```
delete from tb_admin where username = 'eee'
```

- 通过truncate table语句删除数据

```
/* 先复制一张表 */
create table duplicate_tb as select * from tb_admin
```

```
/* 删除表中的所有数据 */
truncate table duplicate_tb
```

```
/* 删除表 */
drop table if exists duplicate_tb
```

`delete` 语句和 `truncate` 语句的区别：

- 使用truncate语句后，表中的auto_increment计数器将被重新设置为该列的初始值
- 对于参与了索引和视图的表，不能使用truncate table语句来删除数据，应该使用delete语句
- truncate table操作与delete操作使用的系统和事务日志资源少。delete语句每删除一行都会在事务日志中添加一行记录，而truncate table语句是通过释放存储表数据所用的数据页来删除数据的，因此只在事务日志中记录页的释放

## 改

**改是一种比较危险的操作，使用之前一定要确保where子句的正确性**

```
update tb_admin set password = '540269653' where username = 'wzh'
```

# 数据查询

## 一些基本操作

```
select * from fort_record;
select * from fort_record where plan_id = 121;
select * from fort_record where create_time < '2021-05-30';
select * from fort_record where create_time between '2021-05-01' and '2021-05-31';
select * from fort_record where plan_id in (117, 118, 119, 120);
select * from fort_record where complete_time < now();
select * from fort_record where update_time is null;
select * from fort_record where update_time < '2021-05-15' and create_time < '2021-05-01';
select distinct create_time from fort_record;
select * from fort_record order by original_vcore_sum desc;
select * from fort_record order by original_vcore_sum desc limit 10;
```

### - 带 like 的字符匹配查询

```
/* 搜索create_time字段中包含'06‘的行 */
select * from fort_record where create_time like '%06%';
```

```
/* 搜索create_time字段中包含‘06’且‘06’不是结尾的元组 */
select * from fort_record where create_time like '%06_%';
```

- ‘%’可以匹配一个或多个字符，可以代表任意长度的字符串，长度可以为0
- ‘_’只匹配一个字符

### - 关键字 group by 分组查询

一直感觉 group by 非常奇怪，又说不上来在哪里有问题，这次发现了异常的地方在于：

单独使用 group by 关键字，查询结果中只显示每组的一条记录

```
select plan_id, create_time from fort_record group by create_time;
```

```
+---------+-------------+
| plan_id | create_time |
+---------+-------------+
|     168 | 0000-00-00  |
|     117 | 2021-04-13  |
|     121 | 2021-04-15  |
|     123 | 2021-04-16  |
|     125 | 2021-04-19  |
+---------+-------------+
5 rows in set (0.00 sec)
```

实际场景下，我们肯定是需要每组的所有记录而不是一条记录，因此在这里引入了 `group_concat` 函数

```
select group_concat(plan_id), create_time from fort_record group by create_time;
```

```
+-----------------------------------------------------+-------------+
| group_concat(plan_id)                               | create_time |
+-----------------------------------------------------+-------------+
| 168,171                                             | 0000-00-00  |
| 117,118                                             | 2021-04-13  |
| 122,121                                             | 2021-04-15  |
| 123,124                                             | 2021-04-16  |
| 134,135,136,137,133,132,131,130,128,127,126,125,129 | 2021-04-19  |
+-----------------------------------------------------+-------------+
5 rows in set (0.00 sec)
```

**聚合函数查询**

```
select count(*) from fort_record;
select sum(original_vcore_sum) from fort_record;
select max(original_vcore_sum) from fort_record;
select min(original_vcore_sum) from fort_record;
select avg(original_vcore_sum) from fort_record;
```

**连接查询**

连接是把不同表的记录连到一起的最普遍的方法

- 内连接查询

内连接是最普遍的连接类型，而且是最匀称的，因为他们要求构成连接的每一部分的每个表的匹配，不匹配的行将被排除。内连接最常见的例子是相等连接，也就是连接后几张表的某个字段相同

```
/* 举个例子：考虑一个治理方案会涉及多个资产，我们现在需要拿到治理方案id，治理方案的创建者以及治理方案圈出的资产，这就需要通过内连接查询 */
select a.creator, a.id as plan_id, b.owner from fort_govern_plan a, fort_asset_info b where a.id = b.plan_id;
```

- 外连接查询

与内连接不同，外连接是指使用OUTER JOIN关键字将两个表连接起来。外连接生成的结果集不仅包含符合条件的行数据，而且还包括左表、右表或两边全连接表中所有的数据行

**左外连接**

左外连接（left join）是指将坐标中的所有数据分别与右表中的每条数据进行连接组合，返回的结果除内连接的数据外，还包括左表中不符合条件的数据，并在右表的相应列中添加 `null` 值

```
/* 举例：现在我们需要资产表中的所有数据，除此以外，因为每条资产会对应一个方案id，我们还希望通过方案id查处这个方案的创建者，也就是说我们需要多一列数据，来自方案表 */
select a.*, b.creator from fort_asset_info a left join fort_govern_plan b on a.plan_id = b.id;
```

在实际情况中，可能有些资产表中的数据找不到plan_id与之对应（因为我们可能手动删了某些方案），因此如果我们用内连接的话，这类资产就会被抛弃掉，但我们不希望它们被抛弃（因为我们需要资产表的所有数据，我们只是想在这些数据的基础上增加一列，如果找不到匹配的我们就把这一列设置为 `null` ，在上面的语境下，即：将b.creator设置为空

个人理解：内连接的目的是匹配数据；外连接的目的是扩充，我们除了需要一张表的数据以外，还需要可能存在的另一列数据，这一列数据来自其他表

**右外连接**

右外连接（right join）是指将有表中的所有数据分别与左表中的每条数据进行连接组合，返回的结果除内连接的数据外，还包括右表中不符合条件的数据，并在左表中的相应列中添加 `null`

## 子查询

子查询就是 `select` 查询是另一个查询的附属，在外面一层的查询中使用里面一层查询产生的结果集。这样就不是执行两个独立的查询，而是执行包含一个子查询的单独查询