

# DDL-数据库操作

SHOW DATABASES;

CREATE DATABASE 数据库名;

USE 数据库名;

SELECT DATABASE();

DROP DATABASE 数据库名;

# DDL-表操作-查询

进入数据库之后, 使用show tables  
也就是说使用use语句之后

show tables;

Desc 表名;

Show create table 表名;

```
CREATE TABLE 表名 (  
    字段1 字段1类型[COMMENT 字段1注释],  
    字段2 字段2类型[COMMENT 字段2注释]  
)
```

最后一个字段后面没有逗号。

```
create table tb_user(  
    id int comment "bianhao",  
    name varchar(50) comment "xingming",  
    age int comment "nianling",  
    gender varchar(1) comment "xingbie"  
    ) comment "yonghubiao";
```

# DDL-表操作-数据类型

## 数值类型

TINYINT  
SMALLINT  
MEDIUMINT  
INT (INTEGER)  
BIGINT  
FLOAT  
DOUBLE  
DECIMAL

数值类型后面加上UNSIGNED表示的是当前是无符号的

age TINYINT UNSIGNED

Score double(4,1)

## 字符串类型

CHAR  
VARCHAR  
TINYBLOB  
TINYTEXT  
BLOB  
TEXT  
MEDIUMBLOB  
MEDIUMTEXT  
LONGBLOB  
LONGTEXT

CHAR(10) 括号里面的数字表示的字符的个数，这是定长的字符串，即便是只放了一个字符其他的也会用空格填充

VARCHAR(10), 变长字符串会根据你输入的字符个数进行空间的分配，但是性能较低

例子：用户名 username，使用VARCHAR是较好的选择。性别 gender，使用CHAR比较好

## 日期时间类型

DATE  
TIME  
YEAR  
DATETIME

TIMESTAMP

例如：描述生日birthday使用date比较好

设计一张员工信息表，要求如下：

- 1 编号（纯数字）
- 2 员工工号（字符串类型，长度不超过10位）
- 3 员工姓名（字符串类型，长度不超过10位）
- 4 性别
- 5 年龄
- 6 身份证号
- 7 入职时间（取值为年月日）

```
create table employersinfo(  
    num int comment "编号",  
    id_job char(10),  
    name varchar(10),  
    gender char(1),  
    age tinyint unsigned,  
    idnumber char(18),  
    startdate date  
);
```

# DDL-表操作-修改

## 添加字段

ALTER TABLE 表名 ADD 字段名 类型 [注释];

## 修改数据类型

ALTER TABLE 表名 MODIFY 字段名 新数据类型;

## 修改字段名和字段类型

ALTER TABLE 表名 CHANGE 旧字段名 新字段名 类型 [COMMENT 注释] [约束];

## 删除字段

ALTER TABLE 表名 DROP 字段名;

## 修改表名

ALTER TABLE 表名 RENAME TO 新表名;

## 删除表

DROP TABLE [IF EXISTS] 表名;

## 删除指定表，并重新创建该表

TRUNCATE TABLE 表名;

功能类别	对应子句	示例
增加内容	ADD	ADD COLUMN age INT
修改内容	MODIFY / CHANGE	MODIFY COLUMN salary DECIMAL(12,2)
删除内容	DROP	DROP COLUMN age
改表名	RENAME TO	RENAME TO staff
加/删约	ADD CONSTRAINT / DROP CONSTRAINT	ADD PRIMARY KEY (id)
修改说明	COMMENT=	COMMENT='说明文字'

# DML-增删改

INSERT  
UPDATE  
DELETE  
增删改

给指定字段添加数据

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

给全部字段添加数据

```
INSERT INTO 表名 VALUES (值1, 值2, ...);
```

批量添加数据

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

```
INSERT INTO 表名 VALUES (值1, 值2, ...), (值1, 值2, ...), (值1, 值2, ...);
```

注意:

插入数据时, 指定的字段顺序需要与值的顺序一一对应。

字符串的日期类型的数据应该包含在引号中。

插入的数据大小, 应该在字段的规定范围内。

```
INSERT INTO emp(num,id_job,name,gender,age,idnumber,startdate)VALUE  
(1,1,'sad','男',30,'466533199906032695','20250302');
```

```
INSERT INTO emp(num,id_job,name,gender,age,idnumber,startdate)VALUE  
(2,2,'sa','男',36,'466533199506032695','20250302');
```

```
Select * from emp;
```

```
UPDATE 表名 SET 字段1=值1,字段2=值2,... [WHERE 条件];
```

注意:

修改语句的条件可以有, 也可以没有, 如果没有条件则会修改整张表的所有数据

DELETE FROM 表名 [WHERE 条件];

注意:

DELETE语句的条件可以有，也可以没有，如果没有条件，则会删除整张的表的所有数据。

DELETE语句不能删除某一个字段的值（可以使用UPDATE语句将那个字段设置为NULL）。

# ALTER关键字

非常好 📌，你又问到了数据库操作的“中级核心指令”—— **ALTER**。

## □ 一、ALTER 的作用（一句话总结）

**ALTER 是 MySQL 的“修改命令”，用于在表已经创建之后，对表结构进行调整、修改、重命名或注释。**

你可以理解成：

CREATE TABLE 是“创建表”，  
ALTER TABLE 是“修改表”。

## ⚙ 二、常见用法分类

### 1️⃣ 添加字段（列）

```
ALTER TABLE employees  
ADD COLUMN age INT COMMENT '员工年龄';
```

📖 表中多出一个名为 age 的新列。

### 2️⃣ 修改字段的数据类型或注释

```
ALTER TABLE employees  
MODIFY COLUMN salary DECIMAL(12,2) COMMENT '员工月薪（元）';
```

📖 MODIFY 表示“修改字段结构”，可以调整数据类型、长度、注释等。

### 3️⃣ 重命名字段

```
ALTER TABLE employees  
CHANGE COLUMN name employee_name VARCHAR(50) COMMENT '员工姓名';
```

📖 CHANGE 不仅可以改字段名，还可以顺便修改字段类型或注释。

### 4️⃣ 删除字段

```
ALTER TABLE employees  
DROP COLUMN age;
```

📖 删除掉指定的列。

### 5️⃣ 重命名表

```
ALTER TABLE employees  
RENAME TO staff;
```

📖 表名从 employees 改为 staff。



## 6 添加或删除约束（如主键、外键）

ALTER TABLE employees

ADD CONSTRAINT pk\_emp\_id PRIMARY KEY (id);

删除外键：

ALTER TABLE employees

DROP FOREIGN KEY fk\_dept\_id;

## 7 修改表注释

ALTER TABLE employees

COMMENT = '公司员工基本信息表';

## 三、理解要点

功能类别	对应子句	示例
增加内容	ADD	ADD COLUMN age INT
修改内容	MODIFY / CHANGE	MODIFY COLUMN salary DECIMAL(12,2)
删除内容	DROP	DROP COLUMN age
改表名	RENAME TO	RENAME TO staff
加/删约束	ADD CONSTRAINT / DROP CONSTRAINT	ADD PRIMARY KEY (id)
修改说明	COMMENT=	COMMENT='说明文字'

## 💡 四、补充说明

- ALTER 不会删除已有数据（除非你删除列）；
- 表结构变化会引起 **锁表** 或 **重建表** 操作（尤其是在大表上）；
- 建议在生产环境操作前先备份。

### ✓ 一句话总结：

ALTER 是用来在表已经存在后修改它的结构的命令，包括添加、修改、删除字段，调整表名或注释等。

是否希望我帮你做一个实际演示？比如一步步展示：

1. 创建表；
2. 用 ALTER 添加新列；
3. 修改列类型；

4. 删除列;  
看每一步的表结构变化?

# modify和change有什么区别吗

非常好的问题 📌，这个问题经常会让 MySQL 初学者混淆。

在 ALTER TABLE 语句中，**MODIFY** 和 **CHANGE** 都是用来“修改列”的，但它们之间有重要区别👉

## ❑ 一、两者的区别总结表

对比点	MODIFY	CHANGE
是否必须写新列名	✗ 不需要	✓ 必须写（即使不改名字也要重复写一遍）
能否修改列名	✗ 不能	✓ 可以改列名
能否修改数据类型/约束	✓ 可以	✓ 可以
语法更简洁	✓	✗ 较冗长
推荐使用场景	修改列的属性或注释	修改列名

## ⚙ 二、语法对比

### ❑ MODIFY 用法

ALTER TABLE employees

MODIFY COLUMN salary DECIMAL(12,2) COMMENT '月薪（元）';

📖 说明：

- MODIFY 只修改字段类型、长度、约束或注释；
- **列名保持不变**；
- 语法简单：MODIFY COLUMN <列名> <新定义>。

### ❑ CHANGE 用法

ALTER TABLE employees

CHANGE COLUMN name employee\_name VARCHAR(50) COMMENT '员工姓名';

📖 说明：

- CHANGE 用于**修改列名**；
- 语法要求必须写**旧列名 + 新列名 + 新定义**；
- 即使不想改名字，也要重复写旧名字两次。

例如：

ALTER TABLE employees

CHANGE COLUMN salary salary DECIMAL(12,2) COMMENT '月薪';  
这其实只是修改类型或注释，但依然要写两次 salary。

### □ 三、理解类比（帮你记忆）

动作	英文含义	用法类比
MODIFY	“修改属性”	修改一个人的 <b>身高</b> 但不改名字
CHANGE	“更改名称”	改一个人的 <b>名字</b> （也能顺便改属性）

### ✓ 四、实战建议

- 只改类型或注释时：用 MODIFY  
ALTER TABLE student MODIFY age SMALLINT COMMENT '年龄（岁）';
- 需要改列名时：用 CHANGE  
ALTER TABLE student CHANGE age student\_age SMALLINT COMMENT '学生年龄';

### □ 五、总结一句话

- ◆ MODIFY：修改字段定义，但不改列名。
- ◆ CHANGE：可以修改列名，同时重新定义字段。

是否希望我给你做一个小例子：

我帮你从创建表 → 用 MODIFY 改列类型 → 用 CHANGE 改列名，  
每一步都展示 DESC 的结果？  
这样你会直观看到它们的区别。

# DQL-查询

SELECT 字段列表

FROM 表名列表

WHERE 条件列表

GROUP BY 分组字段列表

HAVING 分组后条件列表

ORDER BY 排序字段列表

LIMIT 分页参数

聚合函数 (count max min avg sum)

查询多个字段

SELECT 字段1, 字段2, ... FROM 表名;

SELECT \* FROM 表名;

设置别名

SELECT 字段1[AS 别名1], 字段2[AS 别名2], ... FROM 表名;

其中 AS可以省略

去除重复记录

SELECT DISTINCT 字段列表 FROM 表名;

条件查询

SELECT 字段列表 FROM 表名 WHERE 条件列表;

使用比较运算符或者逻辑运算符

BETWEEN...AND...

IN(...)

LIKE ( \_ %) \_匹配单个字符, %匹配任意个字符, 要在引号里面

IS NULL

IS NOT NULL

将一列数据作为一个整体, 进行纵向计算

常见聚合函数

count 统计数量

max 最大值

min 最小值

avg 平均值

sum 求和

语法

SELECT 聚合函数(字段列表) FROM 表名;

所有的NULL值不参与聚合函数的运算

# 分组查询

语法

SELECT 字段列表 FROM 表名 [WHERE 条件] GROUP BY 分组字段名 [HAVING 分组后过滤条件]

where 和 having区别

执行时机不同：where是分组之前进行过滤，不满足where条件，不参与分组；而having是分组之后对结果进行过滤。

判断条件不同：where不能对聚合函数进行判断，而having可以。

根据性别分组，统计男员工和女员工的数量

```
select gender,count(*) from emp group by gender;
```

根据性别分组，统计男员工和女员工的平均年龄

```
select gender,avg(age) from emp group by gender;
```

查询年龄小于45的员工，并根据工作地址分组，获取员工数量大于等于3的工作地址

```
select workaddress,count(*) from emp where age<45 group by workaddress having count(*)>=3;
```

执行顺序：where > 聚合函数 > having

分组之后，查询的字段一般为聚合函数和分组字段，查询其他字段无任何意义。

当你在 GROUP BY 中列出多个字段时，

MySQL 会按照**这些字段组合后的唯一值**来分组。

也就是说：

**先按第一个字段分组，再在每个组内按第二个字段分组，以此类推。**

# 排序查询

## 语法

SELECT 字段列表 FROM 表名 ORDER BY 字段1 排序方式1, 字段2 排序方式2, ...;

排序查询支持多字段排序，第一个字段值相同时，才会根据第二个字段进行排序。

排序方式有2种：

ASC：升序（默认值）

DESC：降序

根据年龄对公司员工进行升序排列

```
select * from emp order by age asc;
```

根据入职时间，对员工进行降序排序

```
select * from emp order by entrydate desc;
```

根据年龄对公司员工进行升序排序，年龄相同，再按照入职时间降序排序

```
select * from emp order by age asc, entrydate desc;
```



# 分页查询

SELECT 字段列表 FROM 表名 LIMIT 起始索引, 查询记录数;

起始索引从0开始, 起始索引= (查询页码-1) \*每页显示记录数。

分页查询是数据库的方言, 不同的数据库有不同的实现, MySQL中是LIMIT。

如果查询的是第一页数据, 起始索引可以省略, 直接简写为 limit 10 。

查询第一页员工数据, 每页展示10条记录

```
select * from emp limit 0,10;
```

查询第二页员工数据, 每页展示10条记录

```
select * from emp limit 10,10;
```

案例

查询年龄为20, 21, 22, 23的女性员工的信息

```
select * from emp where gender = '女' and age in(20,21,22,23);
```

查询性别为男, 并且年龄在20-40岁 (含) 的姓名为三个字的员工

```
select * from emp where gender = '男' and age between 20 and 40 and name like '___';
```

统计员工表中, 年龄小于60岁的, 男性员工和女性员工的人数

```
select gender,count(*) from emp where age<60 group by gender;
```

查询所有年龄小于等于35岁员工的姓名和年龄, 并对查询结果按年龄升序排序, 如果年龄相同按入职时间降序排序

```
select name,age from emp where age<=35 order by age asc,entrydate desc;
```

查询性别为男, 且年龄在20-40岁 (含) 以内的前5个员工信息, 对查询的结果按年龄升序排序, 年龄相同按入职时间升序排序

```
select * from emp where gender='男' and age between 20 and 40 order by age asc,entrydate asc limit 5;
```

# 执行顺序

SELECT 字段列表  
FROM 表名列表  
WHERE 条件列表  
GROUP BY 分组字段列表  
HAVING 分组后条件列表  
ORDER BY 排序字段列表  
LIMIT 分页参数

→ 这是编写顺序

FROM  
WHERE  
GROUP BY  
HAVING  
SELECT  
ORDER BY  
LIMIT

→ 这是执行顺序

# DCL

## DCL管理用户

### 查询用户

```
USE mysql;  
SELECT * FROM USER;
```

### 创建用户

```
CREATE USER '用户名'@'主机名' identified by 'password';
```

### 修改用户密码

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

### 删除用户

```
DROP USER '用户名'@'主机名';
```

## DCL 权限控制

### 常用的权限

```
ALL,ALL PRIVILEGES
```

```
SELECT
```

```
INSERT
```

```
UPDATE
```

```
DELETE
```

```
ALTER
```

```
DROP
```

```
CREATE
```

### 查询权限

```
SHOW GRANTS FOR 'USER'@'HOST';
```

### 授予权限

```
GRANT 权限列表 ON 数据库名.表名 TO 'USER'@'HOST';
```

### 撤销权限

```
REVOKE 权限列表 ON 数据库名.表名 FROM 'USER'@'HOST';
```

注意：

多个权限之间，使用逗号分隔

授权时，数据库名和表名可以使用\*进行通配，代表所有。

```
GRANT ALL *.* TO 'USER'@'HOST';
```

# 字符串函数

CONCAT(S1,S2,...,SN) 字符串拼接

LOWER(STR)

UPPER(STR)

LPAD(STR,N,PAD) 用字符串PAD对STR的左边进行填充, 达到N个字符串长度

RPAD(STR,N,PAD)

TRIM(STR) 去掉字符串头部和尾部的空格, 不去掉中间

SUBSTRING(STR,START,LEN) 返回从字符串STR从START位置起的LEN个长度的字符串,索引值是从1开始

```
select concat('hello','mysql');
```

```
select lpad('he',5,'_');
```

企业员工的工号统一为5位, 不足5位的全部在前面填充0

```
update emp set workno = lpad(workno,5,'0');
```

# 数值函数

CEIL(X) 向上取整

FLOOR(X)向下取整

MOD(X,Y) x/y的模

RAND() 生成随机数 0-1 之间

ROUND(X,Y) 求x的四舍五入的值，保留y位小数

```
select ceil(1.1);
```

```
select mod(6,4) 结果是余数
```

生成一个六位数的随机验证码

```
select lpad(round(rand()*100000,0),6,'0');
```

注意随机数可能是0.019255

# 日期函数

CURDATE() 返回当前日期

CURTIME() 返回当前时间

NOW() 放回当前日期和时间

YEAR(date) 获取date的年份 date的类型如下，返回的是一个int类型的值

MONTH(date)

DAY(date)

DATE\_ADD(date,INTERVAL EXPR TYPE) 返回 一个日期/时间值加上一个时间间隔expr后的时间值, TYPE是时间的单位

DATEDIFF(date1,date2) 返回起始时间DATE1 和结束时间DATE2之间的天数，可以返回负数

## 二、date 参数可以是什么类型？

date 可以是以下任意能被 MySQL 识别为日期/时间的类型：

类型	示例值	说明
DATE	'2025-10-19'	只包含日期
DATETIME	'2025-10-19 23:59:59'	包含日期和时间
TIMESTAMP	'2025-10-19 15:00:00'	时间戳类型（会随时区变化）
STRING（可被解析为日期的字符串）	'2025-10-19','20251019'	字符串格式但能被识别为日期
NUMBER（例如 20251019）	20251019	如果能被识别为日期也可用
其他日期函数结果	CURDATE(), NOW(), ADDDATE(...)	函数返回的日期也行

```
select curdate();
```

```
select date_add(now(),INTERVAL 70 DAY);
```

```
select date_add(now(),INTERVAL 70 YEAR);
```

查询所有员工的入职天数，并根据入职天数倒序排列

```
select name,datediff(now(),entrydate) as 'entrydays' from emp order by entrydays;
```

# 流程函数

IF(value,t,f)

如果value为true, 则返回t, 否则返回f

IFNULL(VALUE1,VALUE2)

如果VALUE1不为空, 返回VALUE1, 否则返回VALUE2

CASE WHEN [VAL1] THEN[RES1] ... ELSE [DEFAULT] END

如果VAL1为true, 返回RES1,...,否则返回DEFAULT默认值

CASE [EXPR] WHEN [VAL1] THEN[RES1] ... ELSE [DEFAULT] END

如果EXPR的值等于VAL1, 返回RES1,...,否则返回DEFAULT默认值

```
select if(true,'ok','error');
select ifnull('ok','default');
select ifnull('','default');
select ifnull(null,'default');
```

查询emp表的员工的姓名和工作地址（北京/上海----->一线城市, 其他----->二线城市）

```
select
    name,
    (case workaddress when '北京' then '一线城市' when '上海' then '一线城市' else
    '二线城市' end) as '工作地址'
from emp;
```

统计班级各个学员的成绩, 展示规则如下

>=85 展示优秀

>=60 展示及格

否则 展示不及格

```
select
    id,
    name,
    (case when math>=85 then '优秀' when math>=60 then '及格' else '不及格'
    end) '数学'
from score;
```



# 约束

## NOT NULL

限制该字段的数据不能为NULL

## UNIQUE

保证该字段的所有数据都是唯一的不重复的

## PRIMARY KEY

主键是一行数据的唯一标识，要去非空且唯一

## DEFAULT

保存数据时，如果未指定该字段的值，则采用默认值

## CHECK(8.0.16版本之后)

保证字段值满足某一个条件

## FOREIGN KEY

用来让两张表的数据之间建立连接，保证数据的一致性和完整性

注意：

约束是作用于表中字段上的，可以在创建表/修改表的时候添加约束，对于一个字段可以添加多个约束

### 案例 根据需求，完成表结构的创建

字段名	字段含义	字段类型	约束条件
id	ID唯一标识	int	主键，并且自动增长
name	姓名	varchar(10)	不为空，并且唯一
age	年龄	int	大于0，并且小于等于120
status	状态	char(1)	如果没有指定该值，默认为1
gender	性别	char(1)	无

约束关键字
PRIMARY KEY, AUTO_INCREMENT
NOT NULL, UNIQUE
CHECK
DEFAULT

```
create table user(
    id int primary key auto_increment,
    name varchar(10) not null unique,
    age int check (age>0 && age<=120),
    status char(1) default '1',
    gender char(1)
) comment 'user';
```

```
insert into user(name,age,status,gender) values ('tom1',19,'1','男'),('tom2',18,'0','男');
```

```
insert into user(name,age,status,gender) values ('tom3',19,'1','男');
```

id是自增的，如果插入数据失败也会向系统申请一个id值，再插入下一条数据的id会比上一条有效的数据大2

# 外键删除更新行为

## ● 删除/更新行为

行为	说明
NO ACTION	当在父表中删除/更新对应记录时，首先检查该记录是否有对应外键，如果有则不允许删除/更新。(与 RESTRICT 一致)
RESTRICT	当在父表中删除/更新对应记录时，首先检查该记录是否有对应外键，如果有则不允许删除/更新。(与 NO ACTION 一致)
CASCADE	当在父表中删除/更新对应记录时，首先检查该记录是否有对应外键，如果有，则也删除/更新外键在子表中的记录。
SET NULL	当在父表中删除对应记录时，首先检查该记录是否有对应外键，如果有则设置子表中该外键值为null（这就要求该外键允许取null）。
SET DEFAULT	父表有变更时，子表将外键列设置成一个默认的值 (Innodb不支持)

当在父表中删除/更新对应记录时，首先检查该记录是否有对应外键，如果有则不允许删除/更新。(与 RESTRICT 一致)

当在父表中删除/更新对应记录时，首先检查该记录是否有对应外键，如果有则不允许删除/更新。(与 NO ACTION 一致)

当在父表中删除/更新对应记录时，首先检查该记录是否有对应外键，如果有，则也删除/更新外键在子表中的记录。

当在父表中删除对应记录时，首先检查该记录是否有对应外键，如果有则设置子表中该外键值为null（这就要求该外键允许取null）

父表有变更时，子表将外键列设置成一个默认的值 (Innodb不支持)

```
ALTER TABLE 表名 ADD CONSTRAINT 外键名称 FOREIGN KEY (外键字段) REFERENCES 主表名 (主表字段名) ON UPDATE CASCADE ON DELETE CASCADE;
```

```
alter table emp add constraint fk_emp_dept_id foreign key (dept_id) references  
dept(id) on update cascade on delete cascade;
```

# 多表关系

## 多表关系

### ● 一对多(多对一)

- 案例: 部门 与 员工的关系
- 关系: 一个部门对应多个员工, 一个员工对应一个部门
- 实现: 在**多的一方建立外键, 指向一的一方的主键**

员工表(emp)

id	name	age
1	张无忌	20
2	杨逍	33
3	赵敏	18
4	常遇春	43

部门表(dept)

id	name
1	研发部
2	市场部
3	财务部
4	销售部

员工表(emp) N

id	name	age	dept_id
1	张无忌	20	1
2	杨逍	33	1
3	赵敏	18	2
4	常遇春	43	2

1 部门表(dept)

id	name
1	研发部
2	市场部
3	财务部
4	销售部

### ● 多对多

- 案例: 学生 与 课程的关系
- 关系: 一个学生可以选修多门课程, 一门课程也可以供多个学生选择
- 实现: **建立第三张中间表, 中间表至少包含两个外键, 分别关联两方主键**



## ● 一对一

➤ 案例: 用户 与 用户详情的关系

➤ 关系: 一对一关系, 多用于单表拆分, 将一张表的基础字段放在一张表中, 其他详情字段放在另一张表中, 以提升操作效率

id	name	age	gender	phone	degree	major	primaryschool	middleschool	university
1	黄渤	45	1	18800001111	本科	舞蹈	静安区第一小学	静安区第一中学	北京舞蹈学院
2	冰冰	35	2	18800002222	硕士	表演	朝阳区第一小学	朝阳区第一中学	北京电影学院
3	码云	55	1	18800008888	本科	英语	杭州市第一小学	杭州市第一中学	杭州师范大学
4	李彦宏	50	1	18800009999	本科	应用数学	阳泉第一小学	阳泉区第一中学	清华大学

id	name	age	gender	phone
1	黄渤	45	1	18800001111
2	冰冰	35	2	18800002222
3	码云	55	1	18800008888
4	李彦宏	50	1	18800009999

用户基本信息表(tb\_user)

id	degree	major	primaryschool	middleschool	university
1	本科	舞蹈	静安区第一小学	静安区第一中学	北京舞蹈学院
2	硕士	表演	朝阳区第一小学	朝阳区第一中学	北京电影学院
3	本科	英语	杭州市第一小学	杭州市第一中学	杭州师范大学
4	本科	应用数学	阳泉第一小学	阳泉区第一中学	清华大学

用户教育信息表(tb\_user\_edu)

➤ 实现: 在任意一方加入外键, 关联另外一方的主键, 并且设置外键为唯一的(UNIQUE)

# 外键约束

- 概念

外键用来让两张表的数据之间建立连接，从而保证数据的一致性和完整性。

## PRIMARY KEY

## FOREIGN KEY

id	name	age	job	salary	entrydate	managerid	dept_id
1	金庸	66	总裁	20000	2000-01-01	<null>	5
2	张无忌	20	项目经理	12500	2005-12-05	1	1
3	杨逍	33	开发	8400	2000-11-03	2	1
4	韦一笑	48	开发	11000	2002-02-05	2	1
5	常遇春	43	开发	10500	2004-09-07	3	1

员工表 emp

## PRIMARY KEY

id	name
1	研发部
2	市场部
3	财务部
4	销售部
5	总经办

部门表 dept

子表/从表

父表/主表

注意：目前上述的两张表，在数据库层面，并未建立外键关联，所以是无法保证数据的一致性和完整性的。它们只是在逻辑上有关系，需要建立外键关联。

## 添加外键

create table 表名(

    字段名 数据类型,

    ...

    [constraint] [外键名称] foreign key (外键字段名) references 主表(主表列名)

);

ALTER TABLE 表名 ADD CONSTRAINT 外键名称 FOREIGN KEY (外键字段名)  
REFERENCES 主表 (主表列名);

```
alter table emp add constraint fk_emp_dept_id foreign key (dept_id) references  
dept(id);
```

建立外键连接之后再删除父表中的某一条数据的时候会报错

删除外键

ALTER TABLE 表名 DROP FOREIGN KEY 外键名称;

# 内连接

内连接查询语法:

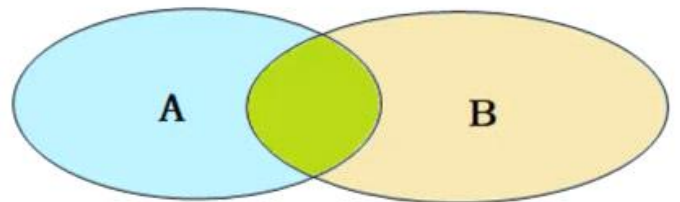
## ➤ 隐式内连接

```
SELECT 字段列表 FROM 表1,表2 WHERE 条件 ...;
```

## ➤ 显式内连接

```
SELECT 字段列表 FROM 表1 [INNER] JOIN 表2 ON 连接条件 ...;
```

内连接查询的是两张表交集的部分



给表取了别名之后在where后面也要使用别名

```
select * from emp, dept;
```

该查询语句会将两个表的内容做一个笛卡尔积之后展示出来

```
select * from emp, dept where emp.dept_id = dept.id;
```

该查询语句会将两个表按照键进行匹配，取交集，第一个表里面的dept\_id字段的值如果在第二个表的id字段中没有对应，则不会被查询出来

-- 1. 查询每一个员工的姓名，及关联的部门的名称 (隐式内连接实现)

```
select emp.name, dept.name from emp, dept where emp.dept_id = dept.id;
```

-- 2. 查询每一个员工的姓名，及关联的部门的名称 (显式内连接实现)

```
select emp.name, dept.name from emp inner join dept on emp.dept_id = dept.id;
```



# 外连接

外连接查询语法:

## ➤ 左外连接

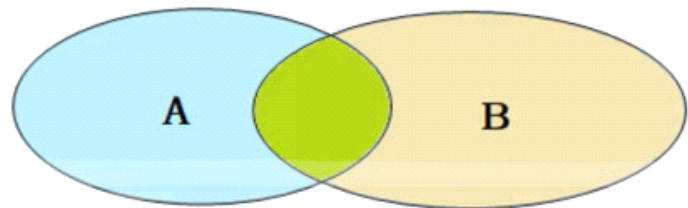
```
SELECT 字段列表 FROM 表1 LEFT [OUTER] JOIN 表2 ON 条件 ... ;
```

相当于查询表1(左表)的所有数据 包含 表1和表2交集部分的数据

## ➤ 右外连接

```
SELECT 字段列表 FROM 表1 RIGHT [OUTER] JOIN 表2 ON 条件 ... ;
```

相当于查询表2(右表)的所有数据 包含 表1和表2交集部分的数据



-- 1. 查询emp表的所有数据, 和对应的部门信息(左外连接)

```
select * from emp left join dept on emp.dept_id = dept.id;
```

这时左边表中如果dept\_id有空值也会查询出来, 后面的 dept表中的数据会被填充成NULL

-- 2. 查询dept表的所有数据, 和对应的员工信息(右外连接)

```
select * from emp right join dept on emp.dept_id = dept.id;
```

# 自连接

## 自连接查询语法:

```
SELECT 字段列表 FROM 表A 别名A JOIN 表A 别名B ON 条件 ... ;
```

自连接查询，可以是内连接查询，也可以是外连接查询。

-- 1. 查询员工 及其 所属领导的名字

```
select * from emp e1 join emp e2 on e1.managerid = e2.id;
```

```
select e1.name, e2.name from emp e1 join emp e2 on e1.managerid = e2.id;
```

-- 2. 查询所有员工 emp 及其领导的名字 emp , 如果员工没有领导, 也需要查询出来

```
select * from emp e1 left join emp e2 on e1.managerid = e2.id;
```

```
select e1.name, e2.name from emp e1 left join emp e2 on e1.managerid = e2.id;
```

# 联合查询

对于union查询，就是把多次查询的结果合并起来，形成一个新的查询结果集。

```
SELECT 字段列表 FROM 表A ...  
UNION [ ALL ]  
SELECT 字段列表 FROM 表B ....;
```

UNION ALL 不会去重，  
联合查询要有相同的列

对于联合查询的多张表的列数必须保持一致，字段类型也需要保持一致。

-- 1. 将薪资低于 5000 的员工，和 年龄大于 50 岁的员工全部查询出来.

```
select * from emp where salary<5000;
```

```
select * from emp where age>50;
```

```
select * from emp where salary<5000;
```

```
select * from emp where age>50;
```

```
select * from emp where salary<5000 or age>50;
```

```
select * from emp where salary<5000
```

```
union all
```

```
select * from emp where age>50;
```

# 子查询

- 概念：SQL语句中嵌套SELECT语句，称为**嵌套查询**，又称**子查询**。

```
SELECT * FROM t1 WHERE column1 = (SELECT column1 FROM t2);
```

子查询外部的语句可以是INSERT / UPDATE / DELETE / SELECT 的任何一个。

- 根据子查询结果不同，分为：
  - 标量子查询（子查询结果为单个值）
  - 列子查询(子查询结果为一列)
  - 行子查询(子查询结果为一行)
  - 表子查询(子查询结果为多行多列)

- 根据子查询位置，分为：WHERE之后、FROM之后、SELECT 之后。

如果子查询返回了空值 NULL，匹配结果可能会变成 “不确定 (UNKNOWN)”

# 标量子查询

- 标量子查询

子查询返回的结果是单个值（数字、字符串、日期等），最简单的形式，这种子查询成为**标量子查询**。

常用的操作符：= <> > >= < <=

-- 1. 查询 "销售部" 的所有员工信息

```
select * from dept where name='销售部';  
select * from emp where dept_id=4;  
select * from emp where dept_id=(select id from dept where name='销售部');
```

-- 2. 查询在 "方东白" 入职之后的员工信息

```
select entrydate from emp where name='方东白';  
select * from emp where entrydate>(select entrydate from emp where name='方东白');
```



# 列子查询

## ● 列子查询

子查询返回的结果是一列（可以是多行），这种子查询称为**列子查询**。

常用的操作符：IN 、 NOT IN 、 ANY 、 SOME 、 ALL

操作符	
IN	在指定的集合范围之内，多选一
NOT IN	不在指定的集合范围之内
ANY	子查询返回列表中，有任意一个满足即可
SOME	与ANY等同，使用SOME的地方都可以使用ANY
ALL	子查询返回列表的所有值都必须满足

-- 1. 查询 "销售部" 和 "市场部" 的所有员工信息

```
select id from dept where dept.name in ('销售部','市场部');
```

```
select * from emp where emp.dept_id in (select id from dept where dept.name in ('销售部','市场部'));
```

-- 2. 查询比 财务部 所有人工资都高的员工信息

```
select id from dept where dept.name='财务部';
```

```
select salary from emp where dept_id=(select id from dept where dept.name='财务部');
```

```
select * from emp where salary>all(select salary from emp where dept_id=(select id from dept where dept.name='财务部'));
```

-- 3. 查询比研发部其中任意一人工资高的员工信息

```
select id from dept where dept.name='研发部';
```

```
select salary from emp where dept_id=(select id from dept where dept.name='研发部');  
select * from emp where salary > any (select salary from emp where dept_id=(select id from dept  
where dept.name='研发部'));
```



# 行子查询

- 行子查询

子查询返回的结果是一行（可以是多列），这种子查询称为行子查询。

常用的操作符：= 、<> 、IN 、NOT IN

-- 1. 查询与 "张无忌" 的薪资及直属领导相同的员工信息；

```
select * from emp where (salary,managerid)=(select emp.salary,emp.managerid from emp where name='张无忌');
```

# 表子查询

表子查询一般在from之后，将查询到的表当作是一张临时表

## ● 表子查询

子查询返回的结果是多行多列，这种子查询称为**表子查询**。

常用的操作符：IN

-- 1. 查询与 "鹿杖客", "宋远桥" 的职位和薪资相同的员工信息

```
select emp.job,emp.salary from emp where name in ('鹿杖客','宋远桥');
```

```
select * from emp where (job,salary) in (select emp.job,emp.salary from emp where name in ('鹿杖客','宋远桥'));
```

-- 2. 查询入职日期是 "2006-01-01" 之后的员工信息, 及其部门信息

```
select * from emp where entrydate>'2006-01-01';
```

```
select dept.name from dept where id in (select dept_id from emp where entrydate>'2006-01-01');
```

```
select * from emp left join dept on emp.dept_id = dept.id;
```

```
select * from (select * from emp where entrydate>'2006-01-01') as aaa left join dept on  
aaa.dept_id=dept.id;
```

# 子查询执行顺序1

非常棒的问题 🎯!

你现在问的是:

“子查询能不能放在 SELECT 后面?”

✓ **答案是: 可以!**

但它有**特定的使用方式和限制**。我们一步一步讲 🗣️

## □ 一、SELECT 后的子查询是什么?

放在 SELECT 后面的子查询, 通常称为 **标量子查询 (Scalar Subquery)**。

它的特点是:

- ✓ **返回单个值 (一个行、一列)**,  
这个值可以直接参与计算、显示或比较。

## □ 二、基本语法

SELECT

column1,

(SELECT 单个值 FROM 其它表 WHERE 条件) AS 别名

FROM 表名;

📖 注意:

- 这个子查询在每一行都会执行一次;
- 子查询必须只返回一行一列, 否则会报错:  
Error 1242: Subquery returns more than 1 row

## 🎯 三、举例说明

假设我们有两个表:

**表 1: employees**

id	name	dept_id
1	Alice	10
2	Bob	20
3	Carol	10

**表 2: departments**

id	dept_name
10	Sales

## ✓ 示例 1：在 SELECT 中嵌入子查询显示部门名

```
SELECT
  name,
  (SELECT dept_name
   FROM departments d
   WHERE d.id = e.dept_id) AS department
FROM employees e;
```

输出：

name	department
------	------------

Alice	Sales
-------	-------

Bob	IT
-----	----

Carol	Sales
-------	-------

💡 说明：

对 employees 的每一行，子查询会执行一次，去 departments 表查出部门名。

## ✓ 示例 2：计算某行的占比或差值

假设我们有一个销售表 sales：

id	salesman	amount
1	Alice	1000
2	Bob	2000
3	Carol	1500

我们想计算每个人的销售额占总销售的比例：

```
SELECT
  salesman,
  amount,
  amount / (SELECT SUM(amount) FROM sales) AS ratio
FROM sales;
```

结果：

salesman	amount	ratio
Alice	1000	0.2
Bob	2000	0.4
Carol	1500	0.3

## □ 四、和其他位置子查询的对比

放置位置	名称	是否允许多行返回	用途
WHERE	条件子查询	✓ (可配合 IN、EXISTS)	过滤行
FROM	派生表子查询	✓ (多行多列)	生成临时表
SELECT	标量子查询	✗ (只能返回单值)	计算、展示新列
HAVING	条件子查询	✓	聚合过滤

## △ 五、注意事项

1. 子查询必须返回单个值 (否则会报错) ;
2. 效率可能较低, 因为会对每一行执行一次;
3. 如果查询结果可能为空, 建议用 COALESCE() 或 IFNULL() 做空值处理;
4. 复杂场景下可以改写为 JOIN 提高性能。

### ✓ 总结一句话:

子查询可以放在 SELECT 后面,  
这种子查询叫做 **标量子查询 (scalar subquery)** ,  
它在每一行计算一个值, 可以当作一列输出或参与表达式。

是否希望我帮你画一张图, 展示同一个 SQL 里子查询在 SELECT、FROM、WHERE 三个位置的执行区别? 这张图会让你一眼看懂三者的作用顺序。

# 子查询执行顺序2

非常好的问题 🙌!

这条语句正是一个经典的“**标量子查询在 SELECT 子句中执行**”的例子。

我们来一步一步、**从执行逻辑的角度**（而不是书写顺序）拆解它 🙌

## 💡 一、完整语句

```
SELECT
  name,
  (SELECT dept_name
   FROM departments d
   WHERE d.id = e.dept_id) AS department
FROM employees e;
```

## 📦 二、整体目标

从 employees 表中取出每个员工的姓名 name,  
并通过子查询查出这个员工对应的部门名称 dept\_name。

也就是：

对 employees 表的每一行员工记录，都执行一次内部子查询来获取部门名。

## ⚙️ 三、执行顺序详解（逻辑顺序）

MySQL 的语句执行顺序并不是从上到下，而是按逻辑步骤进行的。

我们来看这条语句内部的流程。

### 步骤 1：确定主查询的 FROM 来源

FROM employees e

📖 MySQL 首先确定要从哪个表取数据。

这里是从 employees 表中取出所有员工的记录。

### 步骤 2：逐行读取 employees 表中的每一行记录

假设表内容是：

id	name	dept_id
1	Alice	10
2	Bob	20
3	Carol	10

MySQL 会一行一行地处理：

- 当前行：(id=1, name='Alice', dept\_id=10)
- 下一行：(id=2, name='Bob', dept\_id=20)
- 以此类推...

### 步骤 3：对每一行执行 SELECT 子句

对于每一行，MySQL 会计算要输出的列：

```
SELECT
  name,
  (SELECT dept_name
   FROM departments d
   WHERE d.id = e.dept_id) AS department
```

### 步骤 4：执行子查询 (SELECT dept\_name ...)

现在重点来了 📌

在处理当前员工（比如 Alice）这一行时：

- e.dept\_id = 10
- 于是子查询变成：

```
SELECT dept_name
FROM departments d
WHERE d.id = 10;
```

### 步骤 5：执行子查询并返回单个值

假设 departments 表是：

id	dept_name
10	Sales
20	IT

那么对于 Alice 的这行：

子查询返回 dept\_name = 'Sales'

于是当前行的输出为：

name	department
Alice	Sales

### 步骤 6：处理下一行

接下来处理第二行 Bob：

- e.dept\_id = 20
- 子查询就变成：

```
SELECT dept_name FROM departments d WHERE d.id = 20;
```

返回 'IT'

输出第二行:

**name    department**

Bob     IT

以此类推直到所有行完成。

## 🔄 七、总结执行顺序（逻辑层面）

步骤	操作	说明
1	执行 FROM employees e	确定主查询数据来源
2	逐行读取 employees 表记录	一行一行处理
3	对每行执行 SELECT 子句	计算输出列
4	执行子查询 (SELECT dept_name ... WHERE d.id = e.dept_id)	利用当前行的 dept_id 查部门名
5	子查询返回单个值	作为当前行 department 列的结果
6	输出当前行结果	输出到结果集
7	循环处理下一行	重复直到 employees 表遍历完

## ⚠ 八、补充说明

1. 子查询在 SELECT 中属于**相关子查询 (correlated subquery)**，因为它引用了外层表的列 e.dept\_id;
2. 它会为外层的**每一行**执行一次;
3. 如果子查询返回多行，会报错：  
Error 1242: Subquery returns more than 1 row
4. 如果想提升性能，可以改写成 JOIN：  
SELECT e.name, d.dept\_name  
FROM employees e  
JOIN departments d ON e.dept\_id = d.id;

### ✓ 一句话总结：

这条语句的执行顺序是：

先从 employees 取出一行 →

然后执行子查询用该行的 dept\_id 去查部门名 →

返回结果 →



继续下一行，直到全部完成。

是否希望我帮你画一张“执行流程图”，展示外层循环和子查询执行的一对多关系？（像双层循环一样的结构，很直观。）

# 事务

查看/设置事务提交方式

```
select @@autocommit;
```

如果结果为1, 则说明事务提交方式是自动提交的

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

设置提交方式为手动提交

设置手动提交之后, 需要在语句之后执行

commit; 语句, 执行之后事务才会提交到MySQL

rollback; 回滚, 当事务执行异常之后进行回滚, 确保数据恢复到操作之前

另外一种开启事务的方式

- 开启事务

```
START TRANSACTION 或 BEGIN ;
```

- 提交事务

```
COMMIT ;
```

- 回滚事务

```
ROLLBACK ;
```

start transaction; 或者  
start begin;

# 事务特性和并发问题

- 原子性 (Atomicity)：事务是不可分割的最小操作单元，要么全部成功，要么全部失败。
- 一致性 (Consistency)：事务完成时，必须使所有的数据都保持一致状态。
- 隔离性 (Isolation)：数据库系统提供的隔离机制，保证事务在不受外部并发操作影响的独立环境下运行。
- 持久性 (Durability)：事务一旦提交或回滚，它对数据库中的数据的改变就是永久的。

并发问题是指2个或者以上事务同时操作一个数据库引发的问题

问题	描述
脏读	一个事务读到另外一个事务还没有提交的数据。
不可重复读	一个事务先后读取同一条记录，但两次读取的数据不同，称之为不可重复读。
幻读	一个事务按照条件查询数据时，没有对应的数据行，但是在插入数据时，又发现这行数据已经存在，好像出现了“幻影”。

隔离级别	脏读	不可重复读	幻读
Read uncommitted	√	√	√
Read committed	×	√	√
Repeatable Read(默认)	×	×	√
Serializable	×	×	×

-- 查看事务隔离级别

```
SELECT @@TRANSACTION_ISOLATION;
```

-- 设置事务隔离级别

```
SET [SESSION|GLOBAL] TRANSACTION ISOLATION LEVEL {READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE}
```

在使用start transaction开启一个事务之后，后面使用update修改了一条数据，此时还没有提交事务，但是再使用select查询该条数据时发现数据已经被修改了，这是为什么

非常棒的问题 🎯

这说明你已经在理解 **MySQL 事务的底层行为** —— 尤其是**可见性** (visibility) 和**隔离级别** (isolation level) 的问题。

我们一步步拆解来看 🗨️

## ❑ 一、你描述的场景复现

假设你做了以下操作：

```
START TRANSACTION;
```

```
UPDATE employees SET salary = 10000 WHERE id = 1;
```

```
-- 还没有 COMMIT
```

```
SELECT salary FROM employees WHERE id = 1;
```

结果：

🗨️ 你发现 SELECT 查询出的 salary 已经是 10000（修改后的值）。

## ✓ 二、为什么会这样？

**这是因为：事务内部的查询能看到自己在同一事务中修改过的数据。**

换句话说，**同一个事务中的所有操作共享一个“事务视图”**，

而这个视图里包括：

- 事务开始时的快照数据（未修改的行）；
- 当前事务自己新写入或修改的行。

所以：

在同一个事务里，**自己修改的数据对自己立即可见**，  
即使事务还没提交。

这叫做：

**“读己写 (Read Your Own Writes) ” 原则。**

## ❑ 三、MySQL 的事务隔离级别与可见性关系

MySQL 默认使用的隔离级别是：

```
REPEATABLE READ
```

这个级别的规则是：

情况	是否可见
同一个事务自己修改的数据	✓ 可见
其他事务未提交的修改	✗ 不可见
其他事务已提交的修改	取决于隔离级别

## 四、举个两个事务的例子来对比

### 事务A

START TRANSACTION;

UPDATE employees SET salary = 10000

WHERE id = 1;

SELECT salary FROM employees WHERE

id = 1; → 看到10000 ✓

(此时事务A未提交)

COMMIT;

◆ 说明:

- A 能看到自己的修改 (事务内一致性) ;
- B 看不到 A 未提交的修改 (隔离性) 。

### 事务B

SELECT salary FROM employees WHERE

id = 1; → 仍看到旧值 ✗

再查询一次 → 看到10000 ✓

## 五、类比理解

你可以把事务想成一个**独立的小世界 (快照)** :

- 在这个小世界中, 你修改的东西立即生效 (你能看到) ;
- 但在提交之前, **外部世界 (别的事务) 看不到**。

## 六、总结一句话

### 现象

在事务内, 自己能看到自己刚改的数据

但其他事务看不到这些修改

### 原因

事务中的修改对当前事务立即可见 (Read Your Own Writes)

因为修改尚未提交, 未进入全局可见状态

💡 一句话总结:

在 MySQL 中, 事务内部查询能看到自己尚未提交的修改, 这属于事务的一致性特征, 不受隔离级别影响。

如果你愿意, 我可以帮你演示一个**两事务对比实验 (Session A / Session B)** ,

让你清楚看到不同事务间数据可见性的变化。  
要我写那段 SQL 代码吗？

# 为什么会发生脏读问题

非常棒的问题 🙌

你现在已经在理解数据库事务的\*\*隔离性 (Isolation) \*\*本质问题了。

“脏读 (Dirty Read)” 是事务并发问题中最经典的一种，下面我们把它彻底讲清楚👉

## □ 一、什么是“脏读” (Dirty Read)

**脏读**是指：一个事务读到了另一个事务还没有提交的修改数据。

这些“未提交”的数据被称为**脏数据 (dirty data)**，

因为对方事务随时可能 **回滚 (ROLLBACK)**，

那你刚读到的数据就会“失效”或“消失”。

## □ 二、举个例子你就明白了

时间	事务A	事务B
T1	START TRANSACTION;	
T2	UPDATE accounts SET balance = balance - 100 WHERE id = 1;	
T3	(此时A还没提交)	SELECT balance FROM accounts WHERE id = 1; → 读到了 <b>修改后的余额 -100</b> 🌟
T4	ROLLBACK; (撤销修改)	
T5		B 读到的数据根本不存在了 😞

🔑 说明：

- 事务B在T3时读到的余额是**事务A未提交的临时状态**；
- A回滚后，B读到的数据成了“幻觉”，这就是**脏读**。

## ⚙️ 三、为什么会发生脏读？

原因是：

👉 **隔离级别太低**，事务之间没有被完全隔离。

在 MySQL 中，只有在隔离级别为 **READ UNCOMMITTED** 时，才会出现脏读。

隔离级别	是否可能脏读
READ UNCOMMITTED	✓ 可能
READ COMMITTED	✗ 不会
REPEATABLE READ	✗ 不会 (MySQL默认)

## 🔗 四、为什么会读到未提交的数据？

在 READ UNCOMMITTED 模式下：

- 事务读取数据**直接访问最新的物理行版本**（InnoDB行缓冲）；
- 不会判断这条数据是不是“未提交”的；
- 因此会把别的事务正在修改中的数据读出来。

这就是所谓的：

**没有加隔离锁的读操作。**

## □ 五、图示理解

事务A: 修改了一行 → (还没提交)



事务B: 直接读取那一行最新版本（而不是旧快照）



A回滚，修改被撤销



B读到的数据变成了“脏数据”

## ✓ 六、总结一句话

项目	内容
现象	一个事务读到了另一个事务未提交的数据
原因	隔离级别过低（READ UNCOMMITTED）
危害	数据不一致，读到“假数据”
解决方法	使用 READ COMMITTED 或更高隔离级别

### 📖 一句话记忆：

脏读 = 读到了别人“没洗干净”的数据（还没提交的修改）。

要不要我顺带帮你画一个「四种事务隔离级别对比表」，展示每种级别下可能发生的三种并发问题（脏读、不可重复读、幻读）？

那张表记一次就终身不忘。



# 为什么会出现不可重复读

非常好的问题 📌

“**不可重复读 (Non-repeatable Read)**” 是数据库并发事务中常见的一类问题。  
我们先来看它**为什么会发生**，再讲它出现的**具体原因和机制**。

## 一、定义

不可重复读是指：在同一个事务中，**两次读取同一条记录**，结果却**不一致**。

例如：

事务A：

```
START TRANSACTION;
```

```
SELECT balance FROM account WHERE id = 1; -- 第一次读，结果是100
```

事务B（在A还没结束前执行）：

```
UPDATE account SET balance = 200 WHERE id = 1;
```

```
COMMIT;
```

事务A再次执行：

```
SELECT balance FROM account WHERE id = 1; -- 第二次读，结果是200
```

☞ 同一个事务中两次读到的结果不一样，这就是**不可重复读**。

## 二、发生原因

根本原因在于：

**不同事务之间并发执行时，某个事务修改并提交了另一个事务正在读取的数据。**

换句话说：

- 第一个事务读取了某条数据；
- 第二个事务修改了这条数据并提交；
- 第一个事务再次读取同样的数据时，得到了新的值。

这就是因为第一个事务**无法保证两次读取数据时数据的一致性**。

## 三、与隔离级别的关系

MySQL (InnoDB) 中四种事务隔离级别如下：

隔离级别	脏读	不可重复读	幻读
READ UNCOMMITTED	✓可能	✓可能	✓可能
READ COMMITTED	✗不会	✓可能	✓可能
REPEATABLE READ	✗不会	✗不会	✓可能 (InnoDB已解决)
SERIALIZABLE	✗不会	✗不会	✗不会

不可重复读主要发生在 **READ COMMITTED** 隔离级别下。

## 四、为什么在 REPEATABLE READ 下不会发生？

在 InnoDB 中使用了 MVCC (多版本并发控制) 技术。

在第一次查询时，事务会创建一个**一致性视图 (snapshot)**，之后同一个事务的所有查询都基于这个视图执行。

即使其他事务修改并提交了数据，这个事务仍然“看不到”新数据。

所以在 REPEATABLE READ 下，第二次读取依然会得到第一次看到的版本，避免了不可重复读。

### ✓ 总结一句话：

**不可重复读发生是因为一个事务在读取数据的过程中，另一个事务修改并提交了该数据，导致前者再次读取时得到不同结果。**

是否想让我帮你画一张图，直观展示“不可重复读”是怎么发生的吗？

# 为什么会出现幻读

非常棒的问题 🙌

幻读 (Phantom Read) 是事务并发问题中最“微妙”的一种，它和不可重复读非常相似，但本质上影响的是“行的数量”，而不是数据值本身。我们来详细讲清楚它的原理与发生机制



## □ 一、幻读的定义

幻读 (Phantom Read) 指的是：

在同一个事务中，两次执行相同的查询语句，第二次查询时却出现了新增或删除的“行”（记录数不同）。

也就是说：

- 第一次查：返回某些行。
- 第二次查（相同条件）：返回的行数比第一次多（或少）。

这些“突然出现/消失”的行，就像“幻觉”一样，所以叫幻读。

## ■ 二、举个例子

假设有一张表：

```
CREATE TABLE orders (  
  id INT PRIMARY KEY,  
  amount DECIMAL(10,2)  
);
```

### 事务A：

```
START TRANSACTION;
```

```
SELECT * FROM orders WHERE amount > 100; -- 第一次查，返回3条记录
```

### 同时事务B执行：

```
INSERT INTO orders VALUES (4, 200);    -- 插入一条满足 amount>100 的新行  
COMMIT;
```

### 事务A再执行：

```
SELECT * FROM orders WHERE amount > 100; -- 第二次查，返回4条记录
```

☞ 第一次3条，第二次4条，事务A“看到了”一条幻影记录。

## ⚙ 三、幻读产生的原因

幻读的根本原因是：

其他事务在当前事务的查询范围内，插入或删除了符合条件的数据行。

换句话说，事务A并没有锁住“查询的范围”，而只锁住了已存在的行。

因此，当事务B在该范围内新增/删除行时，A的结果集就发生了变化。

## 🔒 四、隔离级别与幻读

隔离级别	是否可能出现幻读
READ UNCOMMITTED	✓ 可能
READ COMMITTED	✓ 可能
REPEATABLE READ (默认)	△ 理论可能，但InnoDB中已避免
SERIALIZABLE	✗ 不会

### 📌 重点：

在 InnoDB 中（MySQL默认存储引擎），即使隔离级别是 **REPEATABLE READ**，也能通过 **Next-Key Lock（临键锁）** 机制避免幻读。

## 📄 五、InnoDB 如何防止幻读

InnoDB 使用了一种称为 **Next-Key Lock** 的锁机制（行锁 + 间隙锁）：

- **行锁 (Record Lock)**：锁定已存在的记录。
- **间隙锁 (Gap Lock)**：锁定两条记录之间的间隙。
- **Next-Key Lock** = 行锁 + 间隙锁。

当事务A执行：

```
SELECT * FROM orders WHERE amount > 100 FOR UPDATE;
```

时，InnoDB 不仅锁住现有的行，还锁住了满足条件的“范围间隙”，防止其他事务插入新的行。

这样就不会再出现“幻影”数据。

## ✓ 总结

问题	说明
幻读是什么	同一事务两次查询结果，行数不一致（出现/消失了记录）
原因	其他事务插入或删除了满足条件的行
和不可重复读区别	不可重复读是“值”变了，幻读是“行”变了
解决方式	InnoDB 的 Next-Key Lock 或 SERIALIZABLE 隔离级别

是否希望我帮你画一张图，对比一下「不可重复读」和「幻读」发生的区别？  
（图会非常直观地显示行锁与间隙锁的不同效果。）