

# MySQL 数据模型框架概述

## 1. 数据模型三个层次

层次	名称	描述	主要组件
外层	视图层	用户视角	视图、权限控制
中层	逻辑层	核心数据组织	表、关系、约束
内层	物理层	数据存储	文件、索引、存储引擎

## 2. 关系模型核心组件

组件	描述	示例
表(Table)	二维数据结构	<code>students</code>
行(Row)	一条记录	一个学生的信息
列(Column)	数据属性	<code>name, age</code>
主键(PK)	唯一标识符	<code>student_id</code>
外键(FK)	表间关联	<code>class_id</code>

## 3. 键的类型与约束

类型	作用	SQL语法
主键	唯一标识	<code>PRIMARY KEY</code>
外键	表间关联	<code>FOREIGN KEY REFERENCES</code>
唯一键	防止重复	<code>UNIQUE KEY</code>
非空约束	必须填写	<code>NOT NULL</code>
检查约束	数据验证	<code>CHECK</code>

## 4. 数据完整性类型

完整性类型	描述	实现方式
实体完整性	主键唯一非空	<code>PRIMARY KEY</code>
参照完整性	外键关系有效	<code>FOREIGN KEY</code>
用户定义完整性	业务规则	<code>CHECK</code> , 数据类型

## 5. 关系类型

关系类型	描述	实现方式
一对多	一个A对应多个B	在"多"方设外键
多对多	多个A对应多个B	通过中间表实现

## 6. MySQL存储引擎比较

特性	InnoDB	MyISAM	Memory
事务支持	✓	✗	✗
外键支持	✓	✗	✗
行级锁	✓	✗	✓
崩溃恢复	✓	✗	✗
全文索引	✓ (5.6+)	✓	✗

## 7. 常用数据类型分类

类别	数据类型	描述
数值型	INT, DECIMAL	整数、小数
字符串	VARCHAR, TEXT	可变长度、长文本
日期时间	DATE, DATETIME	日期时间值
枚举集合	ENUM, SET	预定义值列表

## 8. 数据库设计原则

原则	描述	好处
规范化	减少数据冗余	数据一致性
适当反规范化	故意冗余	查询性能
索引策略	合理创建索引	查询速度
数据类型选择	合适的数据类型	存储效率

## 9. ACID特性

特性	描述	保证内容
原子性(Atomicity)	事务全部完成或全部失败	数据操作完整性
一致性(Consistency)	事务前后数据状态一致	业务规则满足
隔离性(Isolation)	并发事务互不干扰	数据并发安全
持久性(Durability)	事务提交后永久保存	数据不丢失

## SQL 语言分类

分类	英文	主要功能	核心关键字
数据查询语言	DQL (Data Query Language)	数据检索	SELECT , FROM , WHERE
数据操作语言	DML (Data Manipulation Language)	数据增删改	INSERT , UPDATE , DELETE
数据定义语言	DDL (Data Definition Language)	数据库对象管理	CREATE , ALTER , DROP
数据控制语言	DCL (Data Control Language)	权限管理	GRANT , REVOKE
事务控制语言	TCL (Transaction Control Language)	事务管理	COMMIT , ROLLBACK , SAVEPOINT

## 基本语法规则

规则类型	描述	示例
大小写不敏感	关键字不区分大小写	SELECT ≡ select ≡ SELECT
分号结束	多条语句用分号分隔	SELECT * FROM users;
注释语法	单行: -- 或 # 多行: /* */	-- 这是注释 /* 多行注释 */
命名规则	使用字母、数字、下划线	table_name , column1

## 各分类详细语法

### 4.1 DQL - 数据查询语言

子句	作用	示例
SELECT	选择列	SELECT name , age

子句	作用	示例
<b>FROM</b>	指定表	FROM students
<b>WHERE</b>	过滤条件	WHERE age > 18
<b>GROUP BY</b>	分组	GROUP BY class_id
<b>HAVING</b>	分组过滤	HAVING COUNT(*) > 5
<b>ORDER BY</b>	排序	ORDER BY score DESC
<b>LIMIT</b>	限制结果	LIMIT 10

完整示例：

sql

```

1 | SELECT department, AVG(salary) as avg_salary
2 | FROM employees
3 | WHERE hire_date > '2020-01-01'
4 | GROUP BY department
5 | HAVING AVG(salary) > 5000
6 | ORDER BY avg_salary DESC
7 | LIMIT 5;

```

## ● DQL-聚合函数

### 1. 介绍

将一列数据作为一个整体，进行纵向计算。

### 2. 常见聚合函数

函数	功能
count	统计数量
max	最大值
min	最小值
avg	平均值
sum	求和

### 3. 语法

```
SELECT 聚合函数(字段列表) FROM 表名 ;
```

## ● DQL-分组查询

### 1. 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 ];
```

### 2. where与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。

#### 注意

- 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。

## HAVING vs WHERE

方面	WHERE	HAVING
执行时机	分组前过滤行	分组后过滤组
使用聚合	不能使用聚合函数	可以使用聚合函数
性能	更高效（减少分组数据量）	分组后执行

## 升序 vs 降序

排序方式	关键字	说明	示例
升序	ASC	从小到大（默认）	ORDER BY age ASC
降序	DESC	从大到小	ORDER BY salary DESC

## 1. 语法

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



### 注意

- 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。

## 4.2 DML - 数据操作语言

语句	语法	示例
<b>INSERT</b>	插入数据	<code>INSERT INTO table (col1,col2) VALUES (val1,val2)</code>
<b>UPDATE</b>	更新数据	<code>UPDATE table SET col1=val1 WHERE condition</code>
<b>DELETE</b>	删除数据	<code>DELETE FROM table WHERE condition</code>

示例：

sql

```
1 -- 插入数据
2 INSERT INTO users (username, email, age)
3 VALUES ('john_doe', 'john@example.com', 25);
4
5 -- 更新数据
6 UPDATE users
7 SET age = 26, email = 'new_john@example.com'
8 WHERE username = 'john_doe';
9
10 -- 删除数据
11 DELETE FROM users
12 WHERE status = 'inactive' AND last_login < '2023-01-01';
```

## 4.3 DDL - 数据定义语言

语句	作用	示例
<b>CREATE</b>	创建对象	<code>CREATE TABLE, CREATE DATABASE</code>
<b>ALTER</b>	修改结构	<code>ALTER TABLE ADD COLUMN</code>
<b>DROP</b>	删除对象	<code>DROP TABLE, DROP DATABASE</code>
<b>TRUNCATE</b>	清空表	<code>TRUNCATE TABLE table_name</code>

示例：

- DDL-数据库操作

- 查询

查询所有数据库

```
SHOW DATABASES;
```

查询当前数据库

```
SELECT DATABASE();
```

- 创建

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

- 删除

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

- 使用

```
USE 数据库名;
```

## ● DDL-表操作-查询

- 查询当前数据库所有表

```
SHOW TABLES;
```

- 查询表结构

```
DESC 表名;
```

- 查询指定表的建表语句

```
SHOW CREATE TABLE 表名;
```

```
1 -- 创建表
2 CREATE TABLE employees (
3     id INT PRIMARY KEY AUTO_INCREMENT,
4     name VARCHAR(100) NOT NULL,
5     salary DECIMAL(10,2),
6     hire_date DATE DEFAULT CURRENT_DATE
7 );
8
9 -- 修改表结构
10 ALTER TABLE employees
11 ADD COLUMN department VARCHAR(50),
12 ADD INDEX idx_department (department);
13
14 -- 删除表
15 DROP TABLE temporary_data;
```

### 创建表的基本语法结构

sql

```

1 CREATE TABLE [IF NOT EXISTS] table_name (
2     column1 datatype [constraints],
3     column2 datatype [constraints],
4     column3 datatype [constraints],
5     ...
6     [table_level_constraints]
7 ) [table_options];

```

### 三、数据类型选择

#### 3.1 数值类型

类型	存储空间	范围	适用场景
TINYINT	1字节	-128~127	状态值、年龄
INT	4字节	±21亿	ID、数量
BIGINT	8字节	±922亿亿	大数据量ID
DECIMAL(m,d)	变长	精确小数	金额、价格
FLOAT	4字节	单精度	科学数据
DOUBLE	8字节	双精度	高精度计算

#### 3.2 字符串类型

类型	最大长度	特点	使用场景
CHAR(n)	255字符	定长，速度快	性别、代码
VARCHAR(n)	65535字符	变长，省空间	姓名、地址
TEXT	64KB	长文本	文章、描述
LONGTEXT	4GB	超长文本	大型内容

#### 3.3 日期时间类型

类型	格式	范围	用途
DATE	YYYY-MM-DD	1000-9999年	生日、日期
TIME	HH:MM:SS	-838~838小时	时间段
DATETIME	YYYY-MM-DD HH:MM:SS	1000-9999年	创建时间
TIMESTAMP	时间戳	1970-2038年	更新时间

## 修改表结构 - 基础操作语法速查表

操作类型	基本语法	示例
添加列	ALTER TABLE 表名 ADD [COLUMN] 列名 数据类型 [约束] [AFTER 列名]	ALTER TABLE users ADD COLUMN age INT AFTER name;
修改列	ALTER TABLE 表名 MODIFY [COLUMN] 列名 新数据类型 [新约束]	ALTER TABLE users MODIFY COLUMN name VARCHAR(100) NOT NULL;
重命名列	ALTER TABLE 表名 CHANGE [COLUMN] 旧列名 新列名 数据类型 [约束]	ALTER TABLE users CHANGE COLUMN phone mobile VARCHAR(20);
删除列	ALTER TABLE 表名 DROP [COLUMN] 列名	ALTER TABLE users DROP COLUMN temp_field;
添加主键	ALTER TABLE 表名 ADD PRIMARY KEY (列名)	ALTER TABLE users ADD PRIMARY KEY (id);
添加外键	ALTER TABLE 表名 ADD FOREIGN KEY (列名) REFERENCES 参考表(参考列)	ALTER TABLE orders ADD FOREIGN KEY (user_id) REFERENCES users(id);
添加唯一约束	ALTER TABLE 表名 ADD UNIQUE (列名)	ALTER TABLE users ADD UNIQUE (email);
删除主键	ALTER TABLE 表名 DROP PRIMARY KEY	ALTER TABLE users DROP PRIMARY KEY;
删除外键	ALTER TABLE 表名 DROP FOREIGN KEY 约束名	ALTER TABLE orders DROP FOREIGN KEY fk_user;
删除索引	ALTER TABLE 表名 DROP INDEX 索引名	ALTER TABLE users DROP INDEX idx_email;
添加索引	ALTER TABLE 表名 ADD INDEX 索引名 (列名)	ALTER TABLE users ADD INDEX idx_name (name);
重命名表	ALTER TABLE 旧表名 RENAME TO 新表名 RENAME TABLE 旧表名 TO 新表名	ALTER TABLE old_users RENAME TO users;
修改引擎	ALTER TABLE 表名 ENGINE = 引擎名	ALTER TABLE users ENGINE = InnoDB;
修改字符集	ALTER TABLE 表名 CONVERT TO CHARACTER SET 字符集	ALTER TABLE users CONVERT TO CHARACTER SET utf8mb4;

## 4.4 DCL - 数据控制语言

语句	作用	示例
<b>GRANT</b>	授予权限	<code>GRANT SELECT, INSERT ON database.* TO user</code>
<b>REVOKE</b>	撤销权限	<code>REVOKE DELETE ON table FROM user</code>

示例：

sql

```
1 -- 授予权限
2 GRANT SELECT, INSERT, UPDATE ON company.employees TO 'manager'@'localhost';
3
4 -- 撤销权限
5 REVOKE DELETE ON company.employees FROM 'intern'@'%';
```

操作	命令	示例
创建用户	<code>CREATE USER</code>	<code>CREATE USER 'user'@'host' IDENTIFIED BY 'pass'</code>
授予权限	<code>GRANT</code>	<code>GRANT SELECT ON db.table TO 'user'@'host'</code>
撤销权限	<code>REVOKE</code>	<code>REVOKE DELETE ON db.table FROM 'user'@'host'</code>
查看权限	<code>SHOW GRANTS</code>	<code>SHOW GRANTS FOR 'user'@'host'</code>
删除用户	<code>DROP USER</code>	<code>DROP USER 'user'@'host'</code>

- DCL-管理用户

1. 查询用户

```
USE mysql;  
SELECT * FROM user;
```

2. 创建用户

```
CREATE USER '用户名'@'主机名' IDENTIFIED BY '密码';
```

3. 修改用户密码

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

4. 删除用户

```
DROP USER '用户名'@'主机名';
```

1. 查询权限

```
SHOW GRANTS FOR '用户名'@'主机名';
```

2. 授予权限

```
GRANT 权限列表 ON 数据库名.表名 TO '用户名'@'主机名';
```

3. 撤销权限

```
REVOKE 权限列表 ON 数据库名.表名 FROM '用户名'@'主机名';
```

## 4.5 TCL - 事务控制语言

语句	作用	示例
COMMIT	提交事务	COMMIT;
ROLLBACK	回滚事务	ROLLBACK;
SAVEPOINT	设置保存点	SAVEPOINT point1;

示例：

```

1 START TRANSACTION;
2
3 UPDATE accounts SET balance = balance - 100 WHERE id = 1;
4 UPDATE accounts SET balance = balance + 100 WHERE id = 2;
5
6 -- 如果一切正常
7 COMMIT;
8
9 -- 如果出现错误
10 ROLLBACK;

```

## 5. SQL 运算符

### 5.1 比较运算符

运算符	描述	示例
=	等于	age = 25
<> 或 !=	不等于	status <> 'inactive'
> <	大于小于	salary > 5000
>= <=	大于等于/小于等于	age >= 18
BETWEEN	范围之间	age BETWEEN 18 AND 65
LIKE	模式匹配	name LIKE 'J%'
IN	在列表中	id IN (1, 3, 5)

### 5.2 逻辑运算符

运算符	描述	示例
AND	与	age > 18 AND status = 'active'
OR	或	role = 'admin' OR role = 'manager'
NOT	非	NOT deleted

### 5.3 其他运算符

运算符	描述	示例
IS NULL	为空	email IS NULL
IS NOT NULL	不为空	phone IS NOT NULL

运算符	描述	示例
DISTINCT	去重	SELECT DISTINCT city

## 6. 常用函数分类

函数类型	主要函数	示例
聚合函数	COUNT(), SUM(), AVG(), MAX(), MIN()	SELECT AVG(score) FROM students
字符串函数	CONCAT(), SUBSTRING(), LENGTH(), UPPER()	SELECT CONCAT(first_name, ' ', last_name)
数值函数	ROUND(), CEIL(), FLOOR(), ABS()	SELECT ROUND(price, 2)
日期函数	NOW(), CURDATE(), DATE_ADD(), DATEDIFF()	SELECT DATEDIFF(end_date, start_date)

## 7. SQL 执行顺序

顺序	子句	描述
1	FROM + JOIN	确定数据源
2	WHERE	行级过滤
3	GROUP BY	数据分组
4	HAVING	组级过滤
5	SELECT	选择列
6	ORDER BY	结果排序
7	LIMIT	限制结果集

示例理解：

sql

1   SELECT department, AVG(salary) as avg_sal	-- 5. 选择显示的列
2   FROM employees	-- 1. 从哪张表
3   WHERE hire_date > '2020-01-01'	-- 2. 过滤行
4   GROUP BY department	-- 3. 按部门分组
5   HAVING AVG(salary) > 5000	-- 4. 过滤分组
6   ORDER BY avg_sal DESC	-- 6. 结果排序
7   LIMIT 10;	-- 7. 限制结果数量

# SQL 函数速查表

函数分类	函数名称	语法	功能描述	示例
字符串函数	UPPER	UPPER(string)	转换为大写	UPPER('hello') → 'HELLO'
	LOWER	LOWER(string)	转换为小写	LOWER('HELLO') → 'hello'
	SUBSTRING	SUBSTRING(str, start, length)	截取子字符串	SUBSTRING('Hello', 2, 3) → 'ell'
	CONCAT	CONCAT(str1, str2, ...)	连接字符串	CONCAT('A', 'B') → 'AB'
	LENGTH	LENGTH(string)	字符串长度(字节)	LENGTH('hello') → 5
	CHAR_LENGTH	CHAR_LENGTH(string)	字符数	CHAR_LENGTH('你好') → 2
	REPLACE	REPLACE(str, from, to)	替换字符串	REPLACE('ABC', 'B', 'X') → 'AXC'
	TRIM	TRIM(string)	去除两端空格	TRIM(' hello ') → 'hello'
数值函数	ROUND	ROUND(number, decimals)	四舍五入	ROUND(123.456, 2) → 123.46
	CEIL	CEIL(number)	向上取整	CEIL(123.1) → 124
	FLOOR	FLOOR(number)	向下取整	FLOOR(123.9) → 123
	ABS	ABS(number)	绝对值	ABS(-123) → 123
	POWER	POWER(base, exponent)	幂运算	POWER(2, 3) → 8
	SQRT	SQRT(number)	平方根	SQRT(16) → 4
	MOD	MOD(dividend, divisor)	取余数	MOD(10, 3) → 1
	RAND	RAND()	随机数(0-1)	RAND() → 0.123
日期函数	NOW	NOW()	当前日期时间	NOW() → 2024-01-15 10:30:25
	CURDATE	CURDATE()	当前日期	CURDATE() → 2024-01-15
	CURTIME	CURTIME()	当前时间	CURTIME() → 10:30:25
	DATE_ADD	DATE_ADD(date, INTERVAL n unit)	日期加法	DATE_ADD('2024-01-15', INTERVAL 7 DAY) → '2024-01-22'
	DATE_SUB	DATE_SUB(date, INTERVAL n unit)	日期减法	DATE_SUB('2024-01-15', INTERVAL 1 MONTH) → '2023-12-15'
	DATEDIFF	DATEDIFF(date1, date2)	日期差(天数)	DATEDIFF('2024-01-20', '2024-01-15') → 5

函数分类	函数名称	语法	功能描述	示例
	YEAR	YEAR(date)	提取年份	YEAR('2024-01-15') → 2024
	MONTH	MONTH(date)	提取月份	MONTH('2024-01-15') → 1
	DAY	DAY(date)	提取天数	DAY('2024-01-15') → 15
	DATE_FORMAT	DATE_FORMAT(date, format)	日期格式化	DATE_FORMAT(NOW(), '%Y-%m') → '2024-01'
条件函数	IF	IF(condition, true_val, false_val)	条件判断	IF(age>18, 'Adult', 'Child')
	CASE	CASE WHEN...THEN...END	多条件判断	CASE WHEN score>=90 THEN 'A' END
	COALESCE	COALESCE(val1, val2, ...)	返回第一个非NULL值	COALESCE(NULL, 'default') → 'default'
	NULLIF	NULLIF(val1, val2)	相等返回NULL	NULLIF(0, 0) → NULL
	IFNULL	IFNULL(expr1, expr2)	NULL则返回默认值	IFNULL(NULL, 'N/A') → 'N/A'
聚合函数	COUNT	COUNT(expr)	计数	COUNT(*) → 100
	SUM	SUM(column)	求和	SUM(salary) → 500000
	AVG	AVG(column)	平均值	AVG(salary) → 5000
	MAX	MAX(column)	最大值	MAX(salary) → 10000
	MIN	MIN(column)	最小值	MIN(salary) → 3000
	GROUP_CONCAT	GROUP_CONCAT(expr)	分组连接字符串	GROUP_CONCAT(name) → 'Tom,Jerry'
窗口函数	ROW_NUMBER	ROW_NUMBER() OVER(...)	行号	ROW_NUMBER() OVER(ORDER BY id)
	RANK	RANK() OVER(...)	排名(跳跃)	RANK() OVER(ORDER BY score)
	DENSE_RANK	DENSE_RANK() OVER(...)	排名(连续)	DENSE_RANK() OVER(ORDER BY score)
	LAG	LAG(column, n) OVER(...)	前n行值	LAG(salary, 1) OVER(ORDER BY id)
	LEAD	LEAD(column, n) OVER(...)	后n行值	LEAD(salary, 1) OVER(ORDER BY id)
系统函数	VERSION	VERSION()	MySQL版本	VERSION() → '8.0.33'
	DATABASE	DATABASE()	当前数据库	DATABASE() → 'test_db'

函数分类	函数名称	语法	功能描述	示例
	USER	USER()	当前用户	USER() → 'root@localhost'
	LAST_INSERT_ID	LAST_INSERT_ID()	最后插入ID	LAST_INSERT_ID() → 123

## SQL 约束速查表

约束类型	关键字	级别	功能描述	语法示例
主键约束	PRIMARY KEY	列级/表级	唯一标识记录，非空	id INT PRIMARY KEY PRIMARY KEY (col1, col2)
外键约束	FOREIGN KEY	表级	维护表间引用完整性	FOREIGN KEY (dept_id) REFERENCES departments(id)
唯一约束	UNIQUE	列级/表级	保证列值唯一(允许多个NULL)	email VARCHAR(100) UNIQUE UNIQUE (col1, col2)
非空约束	NOT NULL	列级	列值不能为NULL	name VARCHAR(50) NOT NULL
检查约束	CHECK	列级/表级	自定义数据验证规则	age INT CHECK (age >= 0) CHECK (end_date > start_date)
默认约束	DEFAULT	列级	指定列默认值	status INT DEFAULT 1 created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
自增约束	AUTO_INCREMENT	列级	自动生成唯一标识	id INT AUTO_INCREMENT

## 外键引用操作选项

操作类型	语法	描述
级联删除	ON DELETE CASCADE	主表删除时，从表同步删除
级联更新	ON UPDATE CASCADE	主表更新时，从表同步更新
设为空	ON DELETE SET NULL	主表删除时，从表设为NULL
设为默认	ON DELETE SET DEFAULT	主表删除时，从表设为默认值
限制操作	ON DELETE RESTRICT	禁止删除（默认）
无操作	ON DELETE NO ACTION	无操作（同RESTRICT）

## 🛠 约束管理操作

操作	语法	示例
添加主键	ALTER TABLE 表 ADD PRIMARY KEY (列)	ALTER TABLE users ADD PRIMARY KEY (id)
添加外键	ALTER TABLE 表 ADD FOREIGN KEY (列) REFERENCES 参考表(列)	ALTER TABLE orders ADD FOREIGN KEY (user_id) REFERENCES users(id)
添加唯一	ALTER TABLE 表 ADD UNIQUE (列)	ALTER TABLE users ADD UNIQUE (email)
添加检查	ALTER TABLE 表 ADD CHECK (条件)	ALTER TABLE employees ADD CHECK (age >= 18)
删除主键	ALTER TABLE 表 DROP PRIMARY KEY	ALTER TABLE users DROP PRIMARY KEY
删除外键	ALTER TABLE 表 DROP FOREIGN KEY 约束名	ALTER TABLE orders DROP FOREIGN KEY fk_user
删除唯一	ALTER TABLE 表 DROP INDEX 索引名	ALTER TABLE users DROP INDEX uk_email

## 多表查询

### 连接类型速览

连接类型	关键字	描述	返回结果
内连接	INNER JOIN	返回两个表都匹配的记录	交集
左连接	LEFT JOIN	返回左表所有记录+右表匹配记录	左表全集
右连接	RIGHT JOIN	返回右表所有记录+左表匹配记录	右表全集
全连接	FULL OUTER JOIN	返回两个表所有记录	并集
交叉连接	CROSS JOIN	返回两个表的笛卡尔积	所有组合

### 📋 多表查询速查表

查询类型	语法	描述	使用场景
内连接	SELECT ... FROM A INNER JOIN B ON A.id = B.id	返回匹配的记录	需要两个表都有数据的场景

查询类型	语法	描述	使用场景
左连接	<code>SELECT ... FROM A LEFT JOIN B ON A.id = B.id</code>	返回左表所有+右表匹配	包含主表所有记录，关联表可选
右连接	<code>SELECT ... FROM A RIGHT JOIN B ON A.id = B.id</code>	返回右表所有+左表匹配	包含从表所有记录，主表可选
全连接	<code>LEFT JOIN UNION RIGHT JOIN</code>	返回两个表所有记录	需要两个表完整数据
交叉连接	<code>SELECT ... FROM A CROSS JOIN B</code>	返回笛卡尔积	生成所有组合
自连接	<code>SELECT ... FROM A a1 JOIN A a2 ON a1.id = a2.parent_id</code>	表连接自身	层次数据查询
自然连接	<code>SELECT ... FROM A NATURAL JOIN B</code>	自动按同名列连接	简化同名列连接

## 🔧 连接条件类型

条件类型	示例	描述
等值连接	<code>ON A.id = B.id</code>	最常用的连接类型
非等值连接	<code>ON A.value BETWEEN B.min AND B.max</code>	范围匹配
复合连接	<code>ON A.id = B.id AND A.type = B.type</code>	多条件连接
使用函数	<code>ON UPPER(A.name) = UPPER(B.name)</code>	函数处理后的匹配

## 💡 多表查询最佳实践

实践原则	说明	示例
使用表别名	提高可读性，避免列名冲突	<code>FROM employees e JOIN departments d</code>
明确指定列	避免 <code>SELECT *</code> ，只选择需要的列	<code>SELECT e.name, d.department_name</code>
使用合适的连接	根据业务需求选择连接类型	需要所有员工用 <code>LEFT JOIN</code>
添加索引	为连接字段创建索引	<code>CREATE INDEX idx_dept_id ON employees(department_id)</code>

实践原则	说明	示例
注意NULL处理	连接时考虑NULL值的影响	WHERE d.department_id IS NOT NULL

## 子查询面试速查表

子查询类型	语法示例	使用场景	性能注意
标量子查询	SELECT ... WHERE col = (SELECT ...)	返回单个值比较	确保子查询返回单值
列子查询	SELECT ... WHERE col IN (SELECT ...)	多值匹配	结果集大时考虑 EXISTS
相关子查询	WHERE col = (SELECT ... FROM t2 WHERE t2.id = t1.id)	逐行比较	性能较差，考虑重写
EXISTS	WHERE EXISTS (SELECT 1 ...)	检查存在性	子查询结果集大时优选
NOT EXISTS	WHERE NOT EXISTS (SELECT 1 ...)	检查不存在性	替代NOT IN (

## 事务

### 事务的ACID特性底层实现机制

ACID特性	底层实现机制	关键技术
原子性 (Atomicity)	Undo Log	回滚段、撤销日志
一致性 (Consistency)	Redo Log + Undo Log	双写缓冲区、崩溃恢复
隔离性 (Isolation)	锁 + MVCC	行锁、间隙锁、版本链
持久性 (Durability)	Redo Log	WAL机制、刷盘策略

# 事务操作

- 开启事务

```
START TRANSACTION 或 BEGIN ;
```

- 提交事务

```
COMMIT ;
```

- 回滚事务

```
ROLLBACK ;
```



## 事务四大特性

- 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。

◦

## 并发问题总结表

问题	生动比喻	本质原因	SQL示例
脏读	用了别人没确认的调料	读取了未提交的数据	事务2读事务1未提交的修改
不可重复读	调料被偷偷换掉	同一行数据被其他事务修改	同一行，两次读结果不同
幻读	凭空多出食材	范围查询的结果集被其他事务增删	同一查询，返回记录数不同
丢失更新	覆盖别人的劳动成果	基于旧值计算并覆盖更新	A的更新被B的更新覆盖

# 解决方案：事务隔离级别

为了解决这些问题，数据库提供了不同的事务隔离级别：

隔离级别	脏读	不可重复读	幻读	丢失更新	性能
读未提交	不允许	不允许	不允许	不允许	★★★★★
读已提交	防止	不允许	不允许	不允许	★★★★
可重复读	防止	防止	不允许	防止	★★★
串行化	防止	防止	防止	防止	★

MySQL默认：可重复读

Oracle/PostgreSQL默认：读已提交

## 设置隔离级别示例：

sql

```
1 -- 设置当前会话的事务隔离级别
2 SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
3
4 -- 开始事务
5 BEGIN;
6 -- 你的SQL操作...
7 COMMIT;
```

## 1. 读未提交（最松的规则，最快但最不安全）

你可以理解成“别人的转账还没真正到账，你却已经看到这笔钱了”。

- 比如小明给你转 1000 元，系统刚发起转账但还没最终确认，你此时查余额就看到多了 1000 元。
- 结果小明的转账突然失败了，系统把这笔钱扣回去了，你之前看到的“余额”就是假的，这就是“脏读”。
- 优点是速度最快，缺点是数据容易不准，一般很少用。

## 2. 读已提交（常用基础规则，平衡速度和准确性）

你可以理解成“只有别人的转账真正到账了，你才能看到这笔钱”。

- 还是小明给你转 1000 元，只有系统显示“转账成功”后，你查余额才能看到多了 1000 元，避免了“脏读”。
- 但有个问题：你第一次查余额是 5000 元，接着小明又转了 2000 元到账，你第二次查就变成了 7000 元——同一时间段内，两次查的结果不一样，这就是“不可重复读”。
- 优点是能防止假数据，速度也比较快，像支付宝、微信支付的日常查询常用这个级别。

### 3. 可重复读（更严的规则，数据更稳定）

你可以理解成“你开始查余额后，不管别人怎么转账，你看到的余额都不变，直到你查完”。

- 你第一次查余额是 5000 元，就算小明转了 2000 元到账，只要你没退出查询页面，刷新多少次看到的还是 5000 元，避免了“不可重复读”。
- 但有个小漏洞：你想查“余额大于 5000 元的账户”，第一次查到 10 个，这时候小明又新增了 1 个余额 6000 元的账户，你再查就变成 11 个——像凭空多了“幻觉”一样，这就是“幻读”。
- 优点是数据稳定性高，能满足大部分业务（比如电商下单、银行对账），MySQL 的默认隔离级别就是这个。

### 4. 串行化（最严的规则，最慢但绝对准确）

你可以理解成“所有操作都排队，一个做完另一个才能来，绝不插队”。

- 你在查“余额大于 5000 元的账户”时，小明想新增账户、小红想转账，都得等你查完才能操作，完全没有“脏读”“不可重复读”“幻读”的问题。
- 优点是数据 100% 准确，缺点是速度特别慢，像春运排队买票一样，只有涉及重大资金（比如银行年底结算）才会偶尔用。

简单总结一下，选择哪个级别，本质是“要不要等”：

隔离级别	通俗理解	适合场景
读未提交	不等，看临时数据	几乎不用
读已提交	不等，但只看真实到账数据	日常查询（支付、购物）
可重复读	开始查后就“冻结”数据	下单、对账
串行化	所有人排队，一个接一个	重大资金结算

## 引擎

### 三大存储引擎对比表

特性	InnoDB	MyISAM	Memory
存储限制	64TB	256TB	受RAM限制
事务支持	✓ 支持	✗ 不支持	✗ 不支持
锁机制	行级锁	表级锁	表级锁
外键	✓ 支持	✗ 不支持	✗ 不支持
崩溃恢复	✓ 强大	◆ 一般	✗ 数据丢失
全文索引	✓ (5.6+)	✓ 支持	✗ 不支持

特性	InnoDB	MyISAM	Memory
适用场景	大多数业务表	读多写少	临时数据

# 索引

## B+树

B + 树通过“阶数”限制节点的键数量，确保单个节点能被一次性加载到内存，减少磁盘 IO 次数。

- 设树的阶数为

$m$

(表示一个节点最多有  $m$  个子节点)，则节点的键数量需满足：

- 根节点：最少 1 个键，最多  $m-1$  个键；
- 非根节点（中间节点、叶子节点）：最少  $\lceil m/2 \rceil - 1$  个键，最多  $m-1$  个键。

- 子节点数量始终比键数量多 1（如 3 个键对应 4 个子节点），保证查询时能通过键的范围定位到唯一子节点。

B+Tree结构规则	生活比喻	规则描述与目的
固定度 (Order= $m$ )	每个目录抽屉容量固定	每个节点最多有 $m$ 个子节点/键，保证了结构的可控性。
半满原则	抽屉不能太空	除根节点外，每个节点至少要有 $\text{ceil}(m/2)$ 个子节点/键，防止树结构退化，保证空间利用率。
节点分裂	抽屉满了就分裂并上报	当一个节点的键数量超过 $m$ 时，会分裂成两个节点，并 promoted 一个键到父节点，这是树长高的唯一方式。
键的拷贝	上报的是“目录项”副本	分裂时，中间键的副本会被推到父节点；原始数据永远在叶子节点。
数据只在叶子	只有最终清单才有具体位置	所有数据指针（或数据本身）都存储在叶子节点。非叶子节点只充当导航索引。
叶子节点链表	最终清单被装订成册	所有叶子节点通过指针形成一个有序的双向链表，极大地优化了范围查询和全表遍历。
平衡树	所有书籍清单都在同一层	由于分裂规则，所有叶子节点都位于相同的深度，保证了查询性能的稳定性。

## 索引分类

- 主键索引：**官方唯一指定的图书编号目录。每本书有且只有一个独一无二的编号（如ISBN），通过它一定能找到唯一的一本书。**一张表只能有一个主键索引。**
- 唯一索引：**类似于主键目录，但地位稍次。比如按“书名”编排的目录，理论上书名也不应重复。它要求索引列的值必须是唯一的。**一张表可以有多个唯一索引。**

- **普通索引：最基础的目录**，没有任何唯一性限制。比如你按“书籍类别”做的一个目录，通过“小说类”可以找到成千上万本书。
- **复合索引：一个多级目录**。比如，一个先按“出版年份”再按“作者姓名”排序的目录。你首先找到2023年的区域，然后在这个区域内，再按作者姓名的顺序找到你要的书。**它遵循“最左前缀原则”**——如果你只用“作者姓名”来查，这个目录就失效了，因为你跳过了第一级“出版年份”。

#### 按数据存储关系分类（目录是否与书籍放在一起）

- **聚簇索引：书籍本身就已经按照这个目录的顺序排列了**。比如，图书馆决定直接把所有书按照ISBN号顺序摆放在书架上。那么，**ISBN目录（索引）和书籍（数据）本身就是一体的**。找到索引就等于找到了数据。**一张表只能有一个聚簇索引**（通常就是主键索引）。
- **非聚簇索引：目录是目录，书籍是书籍**。目录卡片上写的不是书的内容，而是书的位置编号（如3号书架-5层）。你通过目录找到位置，还需要根据这个位置再去一次指定的书架拿书。这个过程被称为**回表**。

## 索引语法

操作	语法	生活比喻	说明
创建普通索引	<code>CREATE INDEX 索引名 ON 表名 (列名);</code>	“管理员，请做一本按[作者]排序的目录！”	最基础的创建索引命令。
创建复合索引	<code>CREATE INDEX 索引名 ON 表名 (列1, 列2);</code>	“管理员，请做一本先按[类别]再按[作者]排序的目录！”	用于多条件查询，注意 <b>最左前缀原则</b> 。
创建唯一索引	<code>CREATE UNIQUE INDEX 索引名 ON 表名 (列名);</code>	“管理员，请做一本按[ISBN]排序的目录，并确保没有重复号！”	索引列的值必须唯一。
删除索引	<code>DROP INDEX 索引名 ON 表名;</code>	“管理员，那个[出版社]目录没人用，撤掉吧！”	删除一个已存在的索引。
创建主键索引	<code>... id INT PRIMARY KEY ...</code>	“这本书的官方编号(ISBN)规则就这么定了！”	在建表时定义，一张表只能有一个。

## 性能分析

### • SQL执行频率

MySQL客户端连接成功后，通过`show [session|global] status`命令可以提供服务器状态信息。通过如下指令，可以查看当前数据库的INSERT、UPDATE、DELETE、SELECT的访问频次：

```
SHOW GLOBAL STATUS LIKE 'Com_____';
```

```
mysql> show global status like 'Com_____';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| Com_binlog | 0 |
| Com_commit | 0 |
| Com_delete | 0 |
| Com_import | 0 |
| Com_insert | 0 |
| Com_repair | 0 |
| Com_revoke | 0 |
| Com_select | 11 |
| Com_signal | 0 |
| Com_update | 0 |
| Com_xa_end | 0 |
+-----+-----+
```

# 慢查询日志

## 总结表格

组成部分	生活比喻	在MySQL中的含义与命令
慢查询日志	图书馆的“读者求助记录本”	记录所有执行缓慢的SQL语句的日志文件。
开启/关闭	打开或合上记录本	<code>SET GLOBAL slow_query_log = 1;</code> (开启) <code>SET GLOBAL slow_query_log = 0;</code> (关闭)
时间阈值	“超过5分钟才记录”的规定	<code>SET GLOBAL long_query_time = 5;</code> (单位: 秒)
日志文件	记录本本身	通常名为 <code>hostname-slow.log</code> , 路径由 <code>slow_query_log_file</code> 参数指定。
分析工具	馆长助理, 负责汇总报告	<code>mysqldumpslow</code> : MySQL自带, 简单汇总。 <code>pt-query-digest</code> : 第三方工具, 功能强大, 分析更细致。
未使用索引查询	“连目录都没用的找书请求, 也要记录!”	<code>log_queries_not_using_indexes = 1</code> (记录所有未走索引的查询)

核心价值与流程	说明
为什么用它?	<b>1. 定位问题:</b> 精准找到导致系统缓慢的具体SQL。 <b>2. 优化依据:</b> 为索引优化、SQL重构提供明确目标。 <b>3. 监控预警:</b> 持续监控系统性能变化。
标准工作流	<b>1. 开启日志 -&gt; 2. 运行系统</b> (收集一段时间数据) <b>-&gt; 3. 分析日志</b> (用工具) <b>-&gt; 4. 优化SQL</b> (如加索引、改写查询) <b>-&gt; 5. 验证效果</b> (再次EXPLAIN)

总而言之, 慢查询日志就是你派去数据库里的“侦探”, 它会把所有执行缓慢的SQL“罪犯”都给你揪出来。你的任务就是根据它提供的线索, 去“审判”和“优化”这些SQL语句。

## profile详情

## 总结表格

操作/概念	语法/命令	生活比喻	作用与解读
开启分析	<code>SET SESSION profiling = 1;</code>	开启CT扫描仪	开启当前会话的SQL语句性能分析功能。
执行SQL	执行任意 <code>SELECT/UPDATE</code> 等	让病人做动作	让被分析的SQL语句运行一次。
查看报告列表	<code>SHOW PROFILES;</code>	查看所有病人的体检编号	列出所有被记录的SQL及其总耗时和ID。

操作/概念	语法/命令	生活比喻	作用与解读
查看详细报告	SHOW PROFILE FOR QUERY [ID];	查看某个病人的详细CT报告	核心步骤，显示该SQL在每个执行阶段的精确耗时。
关键瓶颈指标	Sending data	搬书时间	最常见瓶颈。时间过长意味着数据扫描/传输量大。
	System lock	等待钥匙的时间	时间过长意味着存在锁竞争。
	Sorting result	整理书籍顺序的时间	时间过长应考虑为 ORDER BY 字段建立索引。
与EXPLAIN 的关系	互补工具	计划书 vs. 执行报告	EXPLAIN 看数据库打算怎么干；SHOW PROFILE 看它实际干得怎么样，时间花在哪儿了。

## explain

### 总结表格： EXPLAIN 快速诊断指南

字段	重点关注	好的表现	坏的表现	优化方向
type	最关键指标	const, eq_ref, ref, range	ALL, index	为查询条件添加索引
key	是否使用索引	显示索引名称	NULL	创建合适的索引
rows	扫描数据量	数值很小	数值很大	优化查询条件，提高索引选择性
Extra	执行细节	Using index	Using temporary, Using filesort	优化GROUP BY/ORDER BY，使用覆盖索引

## EXPLAIN 使用心法

1. 首要目标：避免 type: ALL (全表扫描)
2. 次要目标：减少 rows (扫描行数)
3. 高级目标：争取 Extra: Using index (覆盖索引)
4. 警惕信号：Using temporary 和 Using filesort

# 索引失效

## 总结表格：索引失效场景速查

失效场景	错误示例	正确写法	根本原因
违反最左前缀	INDEX(a,b): WHERE b=1	WHERE a=1 AND b=1	索引结构要求从左到右匹配
索引列计算	WHERE price*2>100	WHERE price>50	计算后无法使用索引排序
索引列函数	WHERE YEAR(date)=2023	WHERE date BETWEEN...	函数处理破坏索引顺序
NOT LIKE / <>	WHERE name<>'张三'	WHERE name>'张三'	非等值查询无法精确定位
OR连接非索引列	WHERE a=1 OR b=2 (b无索引)	分开查询用UNION	优化器选择全表扫描更优
隐式类型转换	WHERE varchar_col=123	WHERE varchar_col='123'	类型不匹配导致比较失败
LIKE左模糊	WHERE name LIKE '%张'	WHERE name LIKE '张%'	索引按前缀组织，后缀无法查找
范围查询阻断	WHERE a>1 AND b=2	WHERE a=1 AND b>2	范围查询后索引列无法有效使用
数据分布倾斜	WHERE gender='男' (占90%)	-	优化器认为全表扫描更快
IN子查询	WHERE id IN(subquery)	改用JOIN	子查询结果集不确定

## 范围查询阻断规则总结

查询条件	索引使用	关键原因
a > 1 AND b = 2	只有 a 用到索引	范围查询后，后续字段在范围内无序
a >= 1 AND b = 2	a和b 都用到了索引	包含等值片段 (a=1)，在等值片段内后续字段有序
a BETWEEN 1 AND 3 AND b = 2	a和b 都用到了索引	BETWEEN 包含边界值，在边界等值处后续字段有序
a LIKE '张%' AND b = 2	只有 a 用到索引	LIKE 是范围查询，后续字段在范围内无序

# 一句话记忆法则

范围查询会阻断联合索引的后续匹配，但如果范围查询包含等值边界，在边界值处匹配可以继续。

## 使用规则

### SQL提示

SQL提示	语法	强度	生活比喻	使用场景
USE INDEX	USE INDEX (idx_name)	★☆☆弱	"我建议走这条路"	多个索引可选时表达偏好
IGNORE INDEX	IGNORE INDEX (idx_name)	★★☆中	"千万别走这条路"	优化器选择了错误索引时
FORCE INDEX	FORCE INDEX (idx_name)	★★★强	"必须走这条路！"	确信某个索引最优时

### 索引使用规则总结表格

规则类别	概念	生活比喻	语法示例	优点	缺点	适用场景
覆盖索引 vs 回表	覆盖索引：查询字段全部在索引中	查字典时，答案就在目录页上，无需翻正文	CREATE INDEX idx_cover ON users(name, age); SELECT name, age FROM users;	极高性能，无需回表，减少IO	索引占用空间稍大	高频率查询，且查询字段固定
	回表查询：需回到主键索引取完整数据	先查目录找到页码，再翻到正文看详情	CREATE INDEX idx_name ON users(name); SELECT * FROM users WHERE name='张三';	索引较小	性能较低，需要两次查找	需要查询非索引字段
前缀索引	对字段的前N个字符创建索引	只记名字的前3个字来区分人员	CREATE INDEX idx_email_prefix ON users(email(10));	节省存储空间，提高写入速度	无法覆盖索引，无法排序分组	文本字段很长时（地址、URL、描述）
单列索引	单个字段的索引	分开的姓名目录、电话目录、地址目录	CREATE INDEX idx_name ON users(name); CREATE INDEX idx_phone ON users(phone);	简单灵活，维护成本低	多条件查询效率低	单字段查询，字段更新频繁

规则类别	概念	生活比喻	语法示例	优点	缺点	适用场景
联合索引	多个字段组合的索引	统一的【姓氏+名字+城市】综合目录	<pre>CREATE INDEX idx_composite ON users(last_name, first_name, city);</pre>	多条件查询高效，支持排序优化	维护成本高，占用空间大	多字段组合查询，需要排序分组

## 设计原则

### 索引设计原则

1. 针对于数据量较大，且查询比较频繁的表建立索引。
2. 针对于常作为查询条件 (where)、排序 (order by)、分组 (group by) 操作的字段建立索引。
3. 尽量选择区分度高的列作为索引，尽量建立唯一索引，区分度越高，使用索引的效率越高。  
↓
4. 如果是字符串类型的字段，字段的长度较长，可以针对于字段的特点，建立前缀索引。
5. 尽量使用联合索引，减少单列索引，查询时，联合索引很多时候可以覆盖索引，节省存储空间，避免回表，提高查询效率。
6. 要控制索引的数量，索引并不是多多益善，索引越多，维护索引结构的代价也就越大，会影响增删改的效率。
7. 如果索引列不能存储NULL值，请在创建表时使用NOT NULL约束它。当优化器知道每列是否包含NULL值时，它可以更好地确定哪个索引最有效地用于查询。

## 优化

### 总结表格

优化场景	核心问题	优化方案	性能提升
插入数据	频繁IO开销	批量插入 + 顺序主键	提升10-100倍
主键优化	页分裂碎片	自增主键 + 长度控制	提升插入性能
order by	文件排序	索引排序 + 覆盖索引	避免临时文件
group by	临时表排序	索引分组 + 联合索引	减少排序开销
limit	大偏移量	游标分页 + 覆盖索引	提升分页性能
count	全表扫描	count(1) + 近似计数	减少IO操作
update	锁竞争	索引条件 + 短事务	避免锁升级

# 黄金法则总结

1. 索引是万金油：90%的优化问题都可以通过索引解决
2. 批量优于单条：减少网络IO和事务开销
3. 覆盖索引优先：避免回表查询和文件排序
4. 锁要精细控制：使用索引避免锁升级
5. 分页要聪明：大偏移量时使用游标分页

## 1. 插入数据

insert : 批量插入、手动控制事务、主键顺序插入

大批量插入: load data local infile

## 2. 主键优化

主键长度尽量短、顺序插入      AUTO\_INCREMENT      UUID

## 3. order by优化

using index: 直接通过索引返回数据，性能高

using filesort: 需要将返回的结果在排序缓冲区排序

## 4. group by优化



索引，多字段分组满足最左前缀法则

## 5. limit优化

覆盖索引 + 子查询

## 6. count优化

性能: count(字段) < count(主键 id) < count(1) ≈ count(\*)

## 7. update优化

尽量根据主键/索引字段进行数据更新

高级软件人才拉

# 视图/存储过程/触发器

## 总结表格

知识点	核心概念	语法	应用场景
视图介绍	虚拟表，查询封装	<pre>CREATE VIEW view_name AS SELECT...</pre>	简化复杂查询，数据抽象

知识点	核心概念	语法	应用场景
CASCADED检查	严格检查所有层级	WITH CASCADED CHECK OPTION	需要严格数据一致性的场景
LOCAL检查	只检查当前层级	WITH LOCAL CHECK OPTION	灵活的权限控制场景
视图更新	满足条件可更新基础表	UPDATE view SET ...	通过视图维护基础数据
视图作用	安全、简化、抽象	-	系统架构设计，权限管理

## 视图选择指南

1. 需要严格数据一致性 → 使用 CASCADED
2. 需要灵活权限控制 → 使用 LOCAL
3. 简化复杂查询 → 创建查询视图
4. 数据安全保护 → 创建列限制视图
5. 多表关联抽象 → 创建关联视图

- 创建

```
CREATE [OR REPLACE] VIEW 视图名称[(列名列表)] AS SELECT语句 [ WITH [CASCADED | LOCAL] CHECK OPTION ]
```

- 查询

```
查看创建视图语句: SHOW CREATE VIEW 视图名称;
查看视图数据: SELECT * FROM 视图名称 .....
```

- 修改

```
方式一: CREATE [OR REPLACE] VIEW 视图名称[(列名列表)] AS SELECT语句 [ WITH [CASCADED | LOCAL] CHECK OPTION ]
方式二: ALTER VIEW 视图名称[(列名列表)] AS SELECT语句 [ WITH [CASCADED | LOCAL] CHECK OPTION ]
```

- 删除

```
DROP VIEW [IF EXISTS] 视图名称 [视图名称] ...
```

## 存储过程

### 创建存储过程

sql

```
1 -- 修改分隔符（避免与SQL语句中的分号冲突）
2 DELIMITER $$

3
4 -- 创建存储过程
5 CREATE PROCEDURE 过程名称(参数列表)
6 BEGIN
7     -- SQL语句块
8 END$$

9
10 -- 恢复分隔符
11 DELIMITER ;
```

## 系统变量分类：

- 全局系统变量：影响整个MySQL服务器
- 会话系统变量：只影响当前连接

sql

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

3
4 -- 查看所有会话变量
5 SHOW SESSION VARIABLES;

6
7 -- 查看具体变量
8 SELECT @@global.auto_increment_increment;
9 SELECT @@session.autocommit;

10
11 -- 设置会话变量
12 SET SESSION autocommit = 0;
13 SET @@session.auto_increment_increment = 2;
```

## 用户定义变量

### 生活比喻：顾客的临时偏好

用户变量就像顾客本次就餐的临时要求

### 特点：

- 以@开头，会话级别有效
- 无需声明，直接使用

sql

```
1 -- 设置用户变量
2 SET @user_count = 10;
3 SET @company_name := '阿里巴巴';
4
5 -- 在SQL中使用
6 SELECT @user_count;
7 SELECT * FROM employees WHERE company = @company_name;
8
9 -- 通过查询设置变量
10 SELECT COUNT(*) INTO @total FROM employees;
11 SELECT @total;
```

## 局部变量

### 生活比喻：厨师做菜时的临时变量

局部变量就像厨师做菜时临时用的碗和调料，只在当前菜谱（存储过程）内有效

### 语法：

sql

```
1 | DECLARE 变量名 数据类型 [DEFAULT 默认值];
```

### 示例：

sql

```
1 DELIMITER $$ 
2 CREATE PROCEDURE CalculateBonus()
3 BEGIN
4     -- 声明局部变量
5     DECLARE base_salary DECIMAL(10,2);
6     DECLARE bonus_rate DECIMAL(3,2) DEFAULT 0.1;
7     DECLARE total_bonus DECIMAL(10,2);
8
9     -- 使用变量
10    SET base_salary = 5000.00;
11    SET total_bonus = base_salary * bonus_rate;
12
13    SELECT total_bonus AS bonus;
14 END$$
15 DELIMITER ;
```

## 50. 进阶-存储过程-if判断

语法：

sql

```
1 IF 条件 THEN  
2     -- 语句  
3 ELSEIF 条件 THEN  
4     -- 语句  
5 ELSE  
6     -- 语句  
7 END IF;
```

示例：根据成绩评级

sql

```
1 DELIMITER $$  
2 CREATE PROCEDURE CheckGrade(IN score INT)  
3 BEGIN  
4     DECLARE grade_level VARCHAR(10);  
5  
6     IF score >= 90 THEN  
7         SET grade_level = '优秀';  
8     ELSEIF score >= 80 THEN  
9         SET grade_level = '良好';  
10    ELSEIF score >= 60 THEN  
11        SET grade_level = '及格';  
12    ELSE  
13        SET grade_level = '不及格';  
14    END IF;  
15  
16    SELECT grade_level AS result;  
17 END$$  
18 DELIMITER ;  
19  
20 -- 调用  
21 CALL CheckGrade(85); -- 返回: 良好
```

## 51. 进阶-存储过程-参数(IN,OUT,INOUT)

三种参数类型：

类型	作用	生活比喻
IN	输入参数（默认）	给厨师的食材（只进不出）
OUT	输出参数	厨师做好的菜品（只出不进）
INOUT	输入输出参数	可重复使用的容器（既进又出）

示例：

sql

```
1 DELIMITER $$  
2 CREATE PROCEDURE EmployeeOperation(  
3     IN emp_id INT,                      -- 输入：员工ID  
4     OUT emp_name VARCHAR(50),           -- 输出：员工姓名  
5     INOUT salary_adjust DECIMAL(10,2)  -- 输入输出：薪资调整  
6 )  
7 BEGIN  
8     -- 根据输入参数查询输出参数  
9     SELECT name INTO emp_name FROM employees WHERE id = emp_id;  
10  
11    -- 处理输入输出参数  
12    SET salary_adjust = salary_adjust * 1.1;  -- 涨薪10%  
13  
14    -- 更新薪资  
15    UPDATE employees SET salary = salary * salary_adjust WHERE id = emp_id;  
16 END$$  
17 DELIMITER ;  
18  
19 -- 调用  
20 SET @adjust = 1.05;  
21 CALL EmployeeOperation(1, @name, @adjust);  
22 SELECT @name, @adjust;  -- 查看输出结果
```

## 52. 进阶-存储过程-case

两种语法形式：

形式1：值匹配

sql

```
1 DELIMITER $$  
2 CREATE PROCEDURE GetWeekdayName(IN day_num INT)  
3 BEGIN  
4     DECLARE day_name VARCHAR(10);  
5  
6     CASE day_num  
7         WHEN 1 THEN SET day_name = 'Monday';  
8         WHEN 2 THEN SET day_name = 'Tuesday';  
9         WHEN 3 THEN SET day_name = 'Wednesday';  
10        WHEN 4 THEN SET day_name = 'Thursday';  
11        WHEN 5 THEN SET day_name = 'Friday';  
12        WHEN 6 THEN SET day_name = 'Saturday';  
13        WHEN 7 THEN SET day_name = 'Sunday';  
14        ELSE SET day_name = 'Invalid';  
15    END CASE;  
16  
17    SELECT day_name;  
18 END$$  
19 DELIMITER ;
```

## 形式2：条件匹配

sql

```
1 DELIMITER $$  
2 CREATE PROCEDURE EvaluateSalary(IN salary DECIMAL(10,2))  
3 BEGIN  
4     DECLARE level VARCHAR(20);  
5  
6     CASE  
7         WHEN salary > 20000 THEN SET level = '高级';  
8         WHEN salary > 10000 THEN SET level = '中级';  
9         WHEN salary > 5000 THEN SET level = '初级';  
10        ELSE SET level = '实习';  
11    END CASE;  
12  
13    SELECT level AS salary_level;  
14 END$$  
15 DELIMITER ;
```

## 53. 进阶-存储过程-循环-while

语法：

sql

```
1 WHILE 条件 DO  
2     -- 循环体  
3 END WHILE;
```

示例：批量插入测试数据

sql

```
1 DELIMITER $$  
2 CREATE PROCEDURE InsertTestData()  
3 BEGIN  
4     DECLARE i INT DEFAULT 1;  
5  
6     WHILE i <= 100 DO  
7         INSERT INTO test_table (id, name) VALUES (i, CONCAT('User', i));  
8         SET i = i + 1;  
9     END WHILE;  
10    END$$  
11    DELIMITER ;
```

## 54. 进阶-存储过程-循环-repeat

语法：

sql

```
1 REPEAT  
2     -- 循环体  
3 UNTIL 条件 END REPEAT;
```

特点：先执行后判断（至少执行一次）

示例：

sql

```
1 DELIMITER $$  
2 CREATE PROCEDURE RepeatDemo()  
3 BEGIN  
4     DECLARE counter INT DEFAULT 1;  
5  
6     REPEAT  
7         INSERT INTO log_table (message) VALUES (CONCAT('Execution ', counter));  
8         SET counter = counter + 1;  
9     UNTIL counter > 5 END REPEAT;  
10 END$$  
11 DELIMITER ;
```

## 55. 进阶-存储过程-循环-loop

语法：

sql

```
1 循环标签: LOOP  
2     -- 循环体  
3     IF 条件 THEN  
4         LEAVE 循环标签;    -- 退出循环  
5     END IF;  
6  
7     ITERATE 循环标签;    -- 继续下一次循环  
8 END LOOP;
```

## 示例：灵活循环控制

sql

```
1 DELIMITER $$  
2 CREATE PROCEDURE LoopDemo()  
3 BEGIN  
4     DECLARE i INT DEFAULT 0;  
5  
6     my_loop: LOOP  
7         SET i = i + 1;  
8  
9         -- 跳过偶数  
10        IF i % 2 = 0 THEN  
11            ITERATE my_loop;  
12        END IF;  
13  
14        -- 插入奇数  
15        INSERT INTO numbers (value) VALUES (i);
```

```
16
17      -- 退出条件
18      IF i >= 10 THEN
19          LEAVE my_loop;
20      END IF;
21  END LOOP my_loop;
22 END$$
23 DELIMITER ;
```

## 56. 进阶-存储过程-游标-cursor

### 生活比喻：阅读书籍

游标就像手指逐行阅读书籍，一次处理一行数据

### 游标使用步骤：

1. 声明游标
2. 打开游标
3. 获取数据
4. 处理数据
5. 关闭游标

### 示例：批量处理员工数据

sql

```
1  DELIMITER $$
2  CREATE PROCEDURE ProcessEmployees()
3  BEGIN
4      DECLARE done INT DEFAULT 0;
5      DECLARE emp_id INT;
6      DECLARE emp_name VARCHAR(50);
7      DECLARE emp_salary DECIMAL(10,2);
8
9      -- 1. 声明游标
10     DECLARE emp_cursor CURSOR FOR
11         SELECT id, name, salary FROM employees;
12
13     -- 2. 声明异常处理（见下一节）
14     DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
15
16     -- 创建临时结果表
17     CREATE TEMPORARY TABLE temp_results (
18         id INT,
19         name VARCHAR(50),
20         new_salary DECIMAL(10,2)
```

```

21 );
22
23 -- 3. 打开游标
24 OPEN emp_cursor;
25
26 -- 4. 循环获取数据
27 read_loop: LOOP
28   FETCH emp_cursor INTO emp_id, emp_name, emp_salary;
29   IF done = 1 THEN
30     LEAVE read_loop;
31   END IF;
32
33   -- 处理数据: 薪资大于5000的员工加薪5%
34   IF emp_salary > 5000 THEN
35     SET emp_salary = emp_salary * 1.05;
36   END IF;
37
38   -- 插入结果表
39   INSERT INTO temp_results VALUES (emp_id, emp_name, emp_salary);
40 END LOOP;
41
42 -- 5. 关闭游标
43 CLOSE emp_cursor;
44
45 -- 返回结果
46 SELECT * FROM temp_results;
47
48 -- 清理
49 DROP TEMPORARY TABLE temp_results;
50 END$$
51 DELIMITER ;

```

## 57. 进阶-存储过程-条件处理程序-handler

### 生活比喻：应急预案

条件处理程序就像餐厅的应急预案，当出现问题时自动执行

### 语法：

sql

1 | **DECLARE** 处理类型 **HANDLER FOR** 条件类型 处理语句；

## 处理类型：

- CONTINUE：继续执行后续语句
- EXIT：退出BEGIN...END块

## 条件类型：

- SQLSTATE '状态码'
- MySQL错误码
- 条件名称
- SQLWARNING
- NOT FOUND
- SQLEXCEPTION

## 完整示例：

sql

```
1  DELIMITER $$  
2  CREATE PROCEDURE SafeDataProcessing()  
3  BEGIN  
4      DECLARE done INT DEFAULT 0;  
5      DECLARE continue_handler INT DEFAULT 1;  
6  
7      -- 各种异常处理程序  
8      DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;  
9      DECLARE CONTINUE HANDLER FOR SQLEXCEPTION  
10     BEGIN  
11         GET DIAGNOSTICS CONDITION 1 @sqlstate = RETURNED_SQLSTATE;  
12         INSERT INTO error_log (error_message) VALUES (@sqlstate);  
13         SET continue_handler = 0;  
14     END;  
15  
16     DECLARE EXIT HANDLER FOR SQLSTATE '45000'  
17     BEGIN  
18         ROLLBACK;  
19         SELECT '事务已回滚' AS result;  
20     END;  
21  
22     -- 业务逻辑  
23     START TRANSACTION;  
24  
25     -- 数据处理...  
26     IF continue_handler = 0 THEN  
27         SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = '处理失败';  
28     END IF;  
29  
30     COMMIT;  
31     SELECT '处理完成' AS result;
```

```
32 | END$$  
33 | DELIMITER ;
```

# 存储函数

## 总结表格

知识点	核心语法	应用场景
存储过程创建	CREATE PROCEDURE name() BEGIN ... END	封装复杂业务逻辑
系统变量	@global.var_name, @session.var_name	服务器配置管理
用户变量	@var_name := value	会话级数据共享
局部变量	DECLARE var_name type [DEFAULT value]	过程内部计算
IF判断	IF...THEN...ELSEIF...ELSE...END IF	条件分支处理
参数	IN/OUT/INOUT param_name type	输入输出控制
CASE	CASE WHEN...THEN...ELSE...END CASE	多条件判断
WHILE	WHILE condition DO...END WHILE	条件循环
REPEAT	REPEAT...UNTIL condition END REPEAT	至少执行一次循环
LOOP	Label: LOOP...LEAVE Label...END LOOP	灵活循环控制
游标	DECLARE CURSOR FOR...FETCH INTO	逐行处理结果集
异常处理	DECLARE HANDLER FOR condition action	错误处理和恢复

## 与存储过程的主要区别

特性	存储过程	存储函数
返回值	可以有多个OUT参数	必须返回一个值
SQL中使用	使用 CALL 调用	可以在SELECT等语句中直接使用
主要用途	执行复杂业务逻辑	进行计算和数据处理
参数类型	IN, OUT, INOUT	只有IN参数（默认）
事务处理	支持事务	通常不支持

## 2. 存储函数的基本语法

### 创建语法

sql

```
1 DELIMITER $$  
2  
3 CREATE FUNCTION 函数名称(参数列表)  
4 RETURNS 返回值类型  
5 [特征]  
6 BEGIN  
7     -- 函数体  
8     RETURN 返回值;  
9 END$$  
10  
11 DELIMITER ;
```

### 特征选项：

- `DETERMINISTIC`：确定性函数（相同输入总是相同输出）
- `READS SQL DATA`：函数中会读取数据
- `MODIFIES SQL DATA`：函数中会修改数据

## 触发器

### 什么是触发器？

定义：触发器是一种特殊的存储过程，它在特定的数据库事件（INSERT、UPDATE、DELETE）发生时自动执行。

### 核心特性：

1. **自动执行**：无需手动调用，满足条件时自动触发
2. **事件驱动**：基于数据变更事件
3. **前后触发**：可以在操作前或操作后执行
4. **行级或语句级**：可以每行触发或每条SQL触发

## 触发器基本语法

sql

```
1 DELIMITER $$  
2  
3 CREATE TRIGGER 触发器名称  
4 触发时机 触发事件 ON 表名称  
5 FOR EACH ROW -- 行级触发器  
6 BEGIN  
7     -- 触发器逻辑  
8 END$$  
9  
10 DELIMITER ;
```

## 触发时机：

- BEFORE：在操作执行之前
- AFTER：在操作执行之后

## 触发事件：

- INSERT
- UPDATE
- DELETE

## 触发器管理命令

### 查看触发器

sql

```
1 -- 查看所有触发器  
2 SHOW TRIGGERS;  
3  
4 -- 查看特定表的触发器  
5 SHOW TRIGGERS LIKE 'table_name';  
6  
7 -- 查看触发器定义  
8 SHOW CREATE TRIGGER trigger_name;  
9  
10 -- 从信息模式查看  
11 SELECT * FROM information_schema.TRIGGERS  
12 WHERE TRIGGER_SCHEMA = 'your_database';
```

## 删除触发器

sql

```
1 | DROP TRIGGER IF EXISTS trigger_name;
```

## 实际应用场景总结

场景类型	触发器类型	用途	示例
审计日志	AFTER INSERT/UPDATE/DELETE	记录数据变更历史	用户操作日志、薪资变更记录
数据校验	BEFORE INSERT/UPDATE	确保数据质量	年龄验证、邮箱格式检查
数据同步	AFTER INSERT/UPDATE/DELETE	维护数据一致性	统计表自动更新、归档数据
业务规则	BEFORE UPDATE/DELETE	强制执行业务规则	防止修改已完成订单、软删除
自动填充	BEFORE INSERT/UPDATE	自动设置字段值	创建时间、更新时间

## 触发器最佳实践

- 保持简洁：触发器逻辑应该简单明了
- 性能考虑：避免在触发器中执行复杂查询
- 错误处理：合理使用SIGNAL抛出错误
- 避免循环：注意触发器之间的循环调用
- 文档记录：记录触发器的用途和逻辑

## 注意事项

- 一个表同类触发器执行顺序：
  - BEFORE触发器 → 数据库操作 → AFTER触发器
  - 同类触发器按创建顺序执行
- 事务中的触发器：
  - 如果触发器失败，整个事务会回滚
  - 触发器中的SIGNAL会导致事务回滚
- 性能影响：
  - 触发器会增加数据库操作的开销
  - 在高并发场景下需要谨慎使用

# 锁

## 全局锁

### 什么是全局锁？

定义：锁定整个数据库实例，让数据库处于只读状态。

### 全局锁命令：

sql

```
1 -- 加全局锁  
2 FLUSH TABLES WITH READ LOCK;  
3  
4 -- 释放全局锁  
5 UNLOCK TABLES;
```

### 全局锁的影响：

sql

```
1 -- 加锁后，这些操作会被阻塞：  
2 INSERT INTO users (name) VALUES ('张三'); -- ✗ 失败  
3 UPDATE accounts SET balance = 1000; -- ✗ 失败  
4 DELETE FROM orders WHERE status = 'old'; -- ✗ 失败  
5  
6 -- 这些操作仍然可以执行：  
7 SELECT * FROM users; -- ✓ 可以  
8 SHOW TABLES; -- ✓ 可以
```

### 查看锁状态：

sql

```
1 -- 查看当前锁信息  
2 SHOW PROCESSLIST;  
3 -- 或  
4 SELECT * FROM performance_schema.metadata_locks;
```

## 使用全局锁进行一致性备份：

### 方法1：传统全局锁备份

sql

```
1 -- 步骤1: 加全局读锁
2 FLUSH TABLES WITH READ LOCK;
3
4 -- 步骤2: 获取一致性位点 (用于主从复制)
5 SHOW MASTER STATUS;
6
7 -- 步骤3: 执行备份 (在另一个会话中)
8 -- mysqldump -u root -p database_name > backup.sql
9
10 -- 步骤4: 释放锁
11 UNLOCK TABLES;
```

### 方法2：InnoDB在线备份（推荐）

sql

```
1 -- 使用mysqldump的single-transaction参数
2 mysqldump --single-transaction -u root -p database_name > backup.sql
```

## 全局锁备份的优缺点：

优点：

- 数据绝对一致
- 操作简单

缺点：

- 整个数据库只读，业务停摆
- 备份期间无法处理任何写操作
- 不适合大型数据库（锁定时间太长）

## 表级锁

分类	表共享读锁 (Table Read Lock)	表独占写锁 (Table Write Lock)
生活化比喻	多人可同时进入会议室查阅资料，但不能修改资料	一人独占会议室，既能查阅也能修改资料，其他人无法进入

分类	表共享读锁 (Table Read Lock)	表独占写锁 (Table Write Lock)
加锁语句	<code>LOCK TABLES table_name READ;</code>	<code>LOCK TABLES table_name WRITE;</code>
当前会话权限	可执行读操作 ( <code>SELECT</code> )，不可执行写操作 ( <code>INSERT/UPDATE/DELETE</code> )	可执行读操作 ( <code>SELECT</code> ) 和写操作 ( <code>INSERT/UPDATE/DELETE</code> )
其他会话权限	可执行读操作 ( <code>SELECT</code> )，写操作会被阻塞 (等待锁释放)	读、写操作均被阻塞 (完全等待锁释放)
锁兼容性	- 与其他读锁兼容 (多会话可同时加读锁) - 与写锁不兼容 (已有读锁时，写锁请求会阻塞)	- 与其他读锁不兼容 (已有写锁时，读锁请求会阻塞) - 与其他写锁不兼容 (同一时间只能有一个写锁)
释放锁	<code>UNLOCK TABLES;</code> (释放所有表锁)	<code>UNLOCK TABLES;</code> (释放所有表锁)
适用场景	只读查询场景 (如报表生成)，需要保证读数据时不被修改，但允许其他会话同时读	全表更新场景 (如批量修改数据)，需要独占表以避免并发冲突，确保修改过程中数据不被干扰

## 元数据锁

### 一、核心定义：MDL 是什么？

- 全称：Metadata Lock（元数据锁）。
- 作用：专门保护表的元数据（表结构、索引、字段类型等），而非表中的业务数据。
- 管理方式：完全由 MySQL 自动控制，无需手动加锁 / 解锁 —— 操作触发时自动获取，操作结束后自动释放。

### 二、核心锁规则：读锁与写锁的“互斥逻辑”

MDL 只有两种锁类型，规则简单但关键，直接决定并发行为：

锁类型	触发场景	核心权限	与其他锁的兼容性
MDL 读锁	执行查询 (如 <code>SELECT</code> )	允许查询，不允许改结构	与其他读锁兼容 (多会话可同时查)；与写锁互斥 (查时不能改结构)
MDL 写锁	修改表结构 (如 <code>ALTER TABLE</code> )	允许改结构，不允许其他操作	与读锁、写锁均互斥 (改结构时，既不能查，也不能同时改其他结构)

# 意向锁

## 为什么需要意向锁?

问题：如果要对表加写锁，需要检查表中是否有行级读锁。逐行检查效率太低。

解决方案：意向锁在表级别标记“这个表中有行被锁定了”。

### 意向锁类型：

#### 1. 意向共享锁 (IS)

sql

```
1 -- 当给某行加共享锁时，会自动给表加IS锁
2 SELECT * FROM users WHERE id = 1 LOCK IN SHARE MODE;
3 -- 自动：行级S锁 + 表级IS锁
```

#### 2. 意向排他锁 (IX)

sql

```
1 -- 当给某行加排他锁时，会自动给表加IX锁
2 SELECT * FROM users WHERE id = 1 FOR UPDATE;
3 -- 自动：行级X锁 + 表级IX锁
```

# 行级锁

## 行级锁知识总结表格

锁类型	锁定目标	锁模式	兼容性	作用	SQL示例
共享锁 (S Lock)	单行记录	S	与S锁兼容， 与X锁不兼容	允许并发 读，阻止写	SELECT ... LOCK IN SHARE MODE
排他锁 (X Lock)	单行记录	X	与所有锁都不 兼容	独占访问， 阻止读写	SELECT ... FOR UPDATE
间隙锁 (Gap Lock)	记录之 间的间 隙	X, GAP	与间隙锁兼 容，与插入不 兼容	防止幻读， 阻止范围内 插入	自动在RR级别使用
临键锁 (Next-Key Lock)	记录+前 一个间 隙	X (默认)	与间隙锁部分 兼容	行锁+间隙 锁，全面防 幻读	SELECT ... FOR UPDATE (RR级别)

锁类型	锁定目标	锁模式	兼容性	作用	SQL示例
记录锁 (Record Lock)	单行记录	X, REC_NOT_GAP	与间隙锁兼容	只锁记录，不锁间隙	特殊场

## 不同查询场景的锁范围表格

查询场景	记录存在情况	锁定类型	锁定范围示例	阻止的插入
等值查询 id=10	记录存在	临键锁	(5, 10]	插入6-10
等值查询 id=12	记录不存在	间隙锁	(10, 15)	插入11-14
范围查询 id>8 AND id<18	-	临键锁	(5, 20]	插入6-19
最大记录查询 id>100	-	间隙锁	(20, +∞)	插入21以上

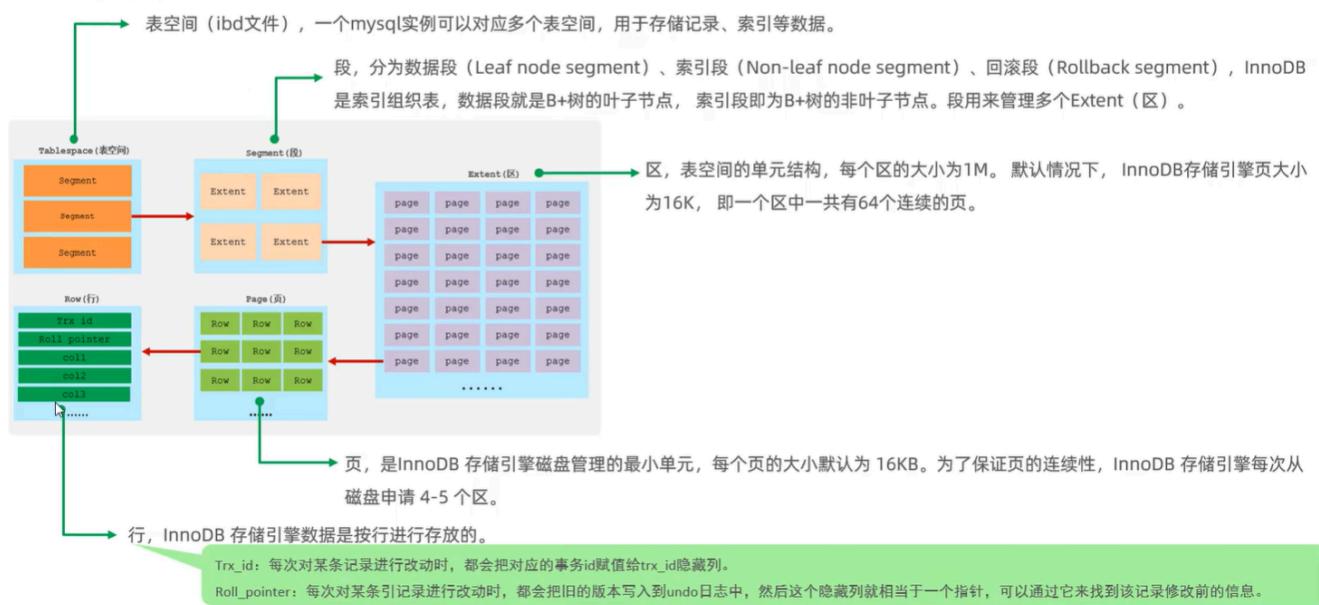
## InnoDB逻辑存储结构

### 生活化比喻：大型图书馆的书籍管理

想象一个超大型图书馆：

- 表空间：整个图书馆大楼
- 段：不同区域（中文区、外文区、期刊区）
- 区：每个书架
- 页：书架上的一格
- 行：具体的每一本书

### 逻辑存储结构



# InnoDB引擎整体架构概览

## 生活化比喻：大型物流仓库系统

想象一个高效的物流仓库：

- **内存结构**: 仓库的临时分拣区（高速操作）
- **磁盘结构**: 仓库的正式货架（永久存储）
- **后台线程**: 仓库的工作人员（自动化处理）

## 内存结构组成：

text

- |   |                                 |
|---|---------------------------------|
| 1 | 内存结构                            |
| 2 | — 缓冲池 (Buffer Pool) - 最重要的部分    |
| 3 | — 更改缓冲区 (Change Buffer)         |
| 4 | — 自适应哈希索引 (Adaptive Hash Index) |
| 5 | — 日志缓冲区 (Log Buffer)            |

## 缓冲池内部结构：

text

- |   |                                |
|---|--------------------------------|
| 1 | 缓冲池 (Buffer Pool)              |
| 2 | — 数据页 (Data Pages) - 表数据       |
| 3 | — 索引页 (Index Pages) - 索引数据     |
| 4 | — 自适应哈希索引 (AHI) - 加速查询         |
| 5 | — 锁信息 (Lock Info) - 行锁信息       |
| 6 | — 数据字典 (Data Dictionary) - 元数据 |

## 更改缓冲区 (Change Buffer)

### 生活比喻：快递代收点

- 小件包裹先放代收点，积累到一定量再统一配送
- 减少频繁往返仓库的次数

## 自适应哈希索引 (Adaptive Hash Index)

### 生活比喻：快递员记忆的常用地址

- 快递员记住经常送货的地址，不用每次都查地图
- 但地址太多记不住，只记最常用的

# 日志缓冲区 (Log Buffer)

## 生活比喻：快递单据临时堆放区

- 快递员先批量记录单据，积累到一定量再统一录入系统

## 磁盘结构组成：

text

1	磁盘结构
2	— 系统表空间 (System Tablespace)
3	— 独立表空间 (File-per-table Tablespace)
4	— 通用表空间 (General Tablespace)
5	— 撤销表空间 (Undo Tablespace)
6	— 临时表空间 (Temporary Tablespace)
7	— 重做日志 (Redo Log)

## 1. 重做日志 (Redo Log) - 崩溃恢复关键

### 生活比喻：仓库操作日志

- 记录所有货物进出操作
- 停电后根据日志恢复现场

## 撤销表空间 (Undo Tablespace)

### 生活比喻：操作撤销记录本

- 记录每个操作的逆操作
- 用于回滚和MVCC

## 后台线程类型：

text

1	后台线程
2	— Master Thread                   - 主线程，协调工作
3	— IO Thread                       - 处理IO请求
4	— Purge Thread                   - 清理Undo日志
5	— Page Cleaner Thread          - 刷脏页

客户端请求



SQL接口



InnoDB引擎

— 内存结构 (高速处理)
— 缓冲池 (数据缓存)
— 更改缓冲区 (索引优化)

```

|   └── 日志缓冲区 (事务日志)
|       └── 自适应哈希索引 (查询加速)

|   └── 磁盘结构 (持久化存储)
|       └── 表空间 (数据存储)
|           └── 重做日志 (崩溃恢复)
|               └── 撤销表空间 (事务回滚)

└── 后台线程 (自动化维护)
    ├── Master Thread (调度)
    ├── IO Thread (IO处理)
    ├── Purge Thread (垃圾回收)
    └── Page Cleaner Thread (内存管理)

```

## MVCC

特性/组件	生动比喻	在 MVCC 中的作用
隐藏字段	报表上的“最后修改人工号”、“上一版档案号”	<code>DB_TRX_ID</code> 标识数据版本, <code>DB_ROLL_PTR</code> 形成历史版本链。
Undo Log	公司档案室	存储数据的历史版本, 用于版本链回溯和事务回滚。
ReadView	我的“阅读视角”	决定一个事务能看到哪个版本的数据。它是实现隔离性的关键。
版本链	用线串起来的糖葫芦, 新版在前, 旧版在后	通过回滚指针将数据的所有历史版本串联起来, 方便按规则遍历。
隔离级别差异	<b>RC</b> (读已提交): 实时快照 <b>RR</b> (可重复读): 定格照片	<b>RC</b> : 每次 SELECT 都生成新 ReadView, 能看到其他已提交的事务。 <b>RR</b> : 只在第一次 SELECT 时生成 ReadView, 后续查询复用, 保证读一致性。

## 隐藏字段

隐藏字段名	生活化比喻	核心作用	备注
<code>DB_TRX_ID</code> (事务 ID)	<b>最后修改者的工号</b>	标识这个数据版本是由哪个事务创建的。是MVCC判断数据可见性的最关键依据。	每次修改都会更新。
<code>DB_ROLL_PTR</code> (回滚指针)	<b>上一版档案在历史库的存放地址</b>	形成数据的版本链。通过它, 可以找到任何历史版本, 是实现“多版本”的关键。	像链表指针, 把各个版本串起来。
<code>DB_ROW_ID</code> (行ID)	<b>档案的内部编号 (保底措施)</b>	当表没有主键时, InnoDB自动创建它来作为每行记录的唯一标识。	如果表有主键, 则此字段不存在。

## MVCC-实现原理

### ● readview



不同的隔离级别, 生成ReadView的时机不同:

- READ COMMITTED: 在事务中每一次执行快照读时生成ReadView。
- REPEATABLE READ: 仅在事务中第一次执行快照读时生成ReadView, 后续复用该ReadView。

数据版本的 DB_TRX_ID	与ReadView比较结果	含义	是否可见
50	$50 < \text{min\_trx\_id}(88)$ <input checked="" type="checkbox"/>	修改者在你开始前就已完成工作	<input checked="" type="checkbox"/> 可见
90	在 $[\text{min}, \text{max}]$ 之间, 但不在 $\text{m\_ids}$ 中	修改者在你开始期间完成工作	<input checked="" type="checkbox"/> 可见
95	在 $[\text{min}, \text{max}]$ 之间, 且在 $\text{m\_ids}$ 中 <input checked="" type="checkbox"/>	修改者正在工作中	<input checked="" type="checkbox"/> 不可见
101	$101 \geq \text{max\_trx\_id}(101)$ <input checked="" type="checkbox"/>	修改者在你之后才开始工作	<input checked="" type="checkbox"/> 不可见

## RC&RR

特性	READ COMMITTED (RC)	REPEATABLE READ (RR)
ReadView 生成时机	每次 SELECT	仅第一次 SELECT
生动比喻	实时直播	定时快照
数据可见性	每次都能看到其他已提交事务的最新结果	只能看到第一次 SELECT 时已经提交的数据状态
是否会出现“不可重复读”	会	不会
原理核心	ReadView 反映查询瞬间的数据库状态	ReadView 反映事务开始后第一个查询瞬间的数据库状态

# MySQL管理

## 系统数据库

Mysql数据库安装完成后，自带了一下四个数据库，具体作用如下：

数据库	含义
mysql	存储MySQL服务器正常运行所需要的各种信息（时区、主从、用户、权限等）
information_schema	提供了访问数据库元数据的各种表和视图，包含数据库、表、字段类型及访问权限等
performance_schema	为MySQL服务器运行时状态提供了一个底层监控功能，主要用于收集数据库服务器性能参数
sys	包含了一系列方便DBA和开发人员利用performance_schema性能数据库进行性能调优和诊断的视图

## 常用工具

### ● mysql

该mysql不是指mysql服务，而是指mysql的客户端工具。

语法：

```
mysql [options] [database]
```

选项：

-u, --user=name	#指定用户名
-p, --password[=name]	#指定密码
-h, --host=name	#指定服务器IP或域名
-P, --port=port	#指定连接端口
-e, --execute=name	#执行SQL语句并退出

-e选项可以在Mysql客户端执行SQL语句，而不用连接到MySQL数据库再执行，对于一些批处理脚本，这种方式尤其方便。

示例：

```
mysql -uroot -p123456 db01 -e "select * from stu";
```

### ● mysqldump

mysqldump客户端工具用来备份数据库或在不同数据库之间进行数据迁移。备份内容包含创建表，及插入表的SQL语句。

语法：

```
mysqldump [options] db_name [tables]
mysqldump [options] --database/-B db1 [db2 db3...]
mysqldump [options] --all-databases/-A
```

连接选项：

-u, --user=name	指定用户名
-p, --password[=name]	指定密码
-h, --host=name	指定服务器ip或域名
-P, --port=#	指定连接端口

输出选项：

--add-drop-database	在每个数据库创建语句前加上drop database语句
--add-drop-table	在每个表创建语句前加上drop table语句，默认开启；不开启(--skip-add-drop-table)
-n, --no-create-db	不包含数据库的创建语句
-t, --no-create-info	不包含数据表的创建语句
-d --no-data	不包含数据
-T, --tab=name	自动生成两个文件：一个.sql文件，创建表结构的语句；一个.txt文件，数据文件

## 常用工具

- mysqlimport/source

mysqlimport 是客户端数据导入工具，用来导入mysql&gt;ump 加 -T 参数后导出的文本文件。

语法：

```
mysqlimport [options] db_name textfile1 [textfile2...]
```

示例：

```
mysqlimport -uroot -p2143 test /tmp/city.txt
```

如果需要导入sql文件,可以使用mysql中的source 指令：

语法：

```
source /root/xxxx.sql
```

## 1. mysql

Mysql 客户端工具, -e 执行SQL并退出

## 2. mysqladmin

Mysql管理工具

## 3. mysqlbinlog

二进制日志查看工具

## 4. mysqlshow

查看数据库、表、字段的统计信息

## 5. mysqldump

数据备份工具

## 6. mysqlimport/source

数据导入工具

# 运维

## 核心知识总结表格

日志类型	生活化比喻	记录内容	主要用途	生产环境建议
错误日志	故障报警器	服务器启动、关闭信息和运行时的错误、警告。	排错首选。当数据库无法启动或出现严重错误时查看。	必须开启，并定期检查。

日志类型	生活化比喻	记录内容	主要用途	生产环境建议
二进制日志	全量高清监控录像	所有更改数据的SQL语句或数据行变化（增、删、改）。	主从复制、数据恢复（PITR）。	强烈建议开启，这是数据安全的重要保障。
查询日志	访客流水账	所有客户端连接和执行的SQL语句（包括查询）。	全局审计、跟踪特定时间点的所有数据库操作。	默认关闭。仅在深度调试时临时开启，否则性能开销大。
慢查询日志	效率分析报告	执行时间超过阈值的SELECT语句（也可配置记录增删改）。	数据库性能分析与优化。 找出并优化慢SQL。	建议开启。合理设置阈值（如1-2秒），定期分析。

日志类型	默认状态	核心配置项	如何查看/使用	关键操作命令/工具
错误日志	开启	log_error	直接查看日志文件内容	SHOW VARIABLES LIKE 'log_error'; tail -f <log_file_path>
二进制日志	关闭	log-bin, binlog-format	1. 查看日志列表 2. 用于数据恢复	SHOW BINARY LOGS; mysqlbinlog 工具
查询日志	关闭	general_log, general_log_file	直接查看日志文件内容	SET GLOBAL general_log = 'ON'; (仅用于调试)
慢查询日志	关闭	slow_query_log, long_query_time	1. 查看日志文件 2. 使用工具分析	mysqldumpslow 工具 pt-query-digest 工具

## 主从复制核心知识总结表格

环节	生活化角色	核心任务	关键配置/命令
概述	总裁 (Master) 与秘书 (Slave)	数据备份、读写分离、负载均衡	理解架构和目标
原理	三步工作法	1. 总裁记日志 2. 通讯员拿日志 3. 执行员执行日志	Binary Log -> I/O Thread -> Relay Log -> SQL Thread
主库配置	给总裁配日志本	1. 开启日志 2. 创建秘书账号 3. 记录起始点	server-id, log-bin CREATE USER ... REPLICATION SLAVE SHOW MASTER STATUS;
从库配置	指导秘书工作	1. 设置身份 2. 指定总裁 3. 开始工作	server-id CHANGE MASTER TO ... START SLAVE; SHOW SLAVE STATUS\G

环节	生活化角色	核心任务	关键配置/命令
测试	检查工作成果	在主库写，在从库读	CREATE / INSERT (主库) -> SELECT (从库)

## 分库分表

### 核心知识总结表格

拆分方式	生活化比喻	拆分对象	核心思想	解决的核心问题
垂直分库	建立专业仓库（用户库、订单库）	库，按业务模块	专库专用，业务解耦	业务耦合高，单库并发压力大
垂直分表	拆分档案袋（常用信息 vs 详细信息）	表，按列/字段	冷热数据分离	单表字段过多，IO效率低
水平分库	按地区建分库（华北库、华南库）	库，按数据行	数据分片，分布式存储	单库存储/性能瓶颈，海量数据
水平分表	按月份分抽屉（1月表、2月表）	表，按数据行	单表数据切分	单表数据量过大，SQL查询慢

一句话总结选择策略：

- 先垂直分库，把业务模块拆开。
- 如果某个业务的表字段太多，再考虑垂直分表。
- 如果某个业务的单表数据量太大，再考虑水平分表（在单库内）。
- 如果单个数据库服务器整体扛不住压力，就进行水平分库，将表分布到不同服务器。

## 分片规则

### 分片规则核心知识总结表格

分片规则	生活化比喻	核心原理	优点	缺点
范围分片	按学区号段分配	按分片键的连续范围分配	易管理，范围查询快	容易产生热点，数据分布可能不均
取模分片	按工号尾数分宿舍	$\text{hash}(\text{key}) \% N$	数据分布均匀	扩缩容代价大（需要重新hash）
一致性Hash	旋转的令牌环	数据和节点映射到哈希环，顺时针查找	扩缩容数据迁移量小	实现复杂，可能有数据倾斜（需虚拟节点）
枚举分片	按省份分配	根据枚举值直接映射	配置简单，直观	不灵活，需预知枚举值
应用指定	老板特批	由应用程序代码逻辑决定	极度灵活	与业务代码耦合，维护难

分片规则	生活化比喻	核心原理	优点	缺点
固定/字符串 Hash	更公平的抽签	使用更优的Hash函数使分布均匀	数据分布更均衡	复杂度增加
按天/月分片	按日期归档文件	按时间范围分配	便于数据生命周期管理	存在时间热点（读写集中在最新分片）

选择策略小结：

- 追求均匀分布和简单性：首选 取模分片（如果分片数量基本确定）。
- 需要频繁扩缩容：首选 一致性Hash算法。
- 业务有明显的范围特征（如时间、地域）：考虑 范围分片 或 枚举分片。
- 需要极致的灵活性和业务定制：才考虑 应用指定算法。

## 读写分离

### 核心知识总结表格

主题	生活化比喻	核心目标	关键技术点	优缺点
读写分离介绍	设立“问讯处”	分摊主库压力，提升系统整体吞吐量	写操作发往主库，读操作发往从库	优点：提升性能，提高可用性。 缺点：存在数据延迟（主从同步需要时间）。
一主一从架构	一个总裁配一个秘书	基础的高可用与读写分离	1. 搭建主从复制 2. 应用层或中间件实现SQL路由	实现简单，是经典基础架构。但主库是单点故障。
双主双从架构	联席总裁制度	消除单点故障，实现更高可用性	1. 配置双主复制 2. 解决自增ID冲突 3. 每个主再配置从库	优点：高可用性极高。 缺点：架构复杂，配置和维护成本高。