# MySQL

存储数据，管理数据。存在500万条数据的限制。

## 数据库分类

### 关系型数据库

MySql、Oracle、Sql Server...

通过表和表之间、行和列之间的关系进行数据的存储。

### 非关系型数据库

redis、mongDB...

通过对象自身的属性来决定。

## DBMS

数据库管理系统。

## 安装MySQL和SQLyog

### MySQL

配置环境变量。

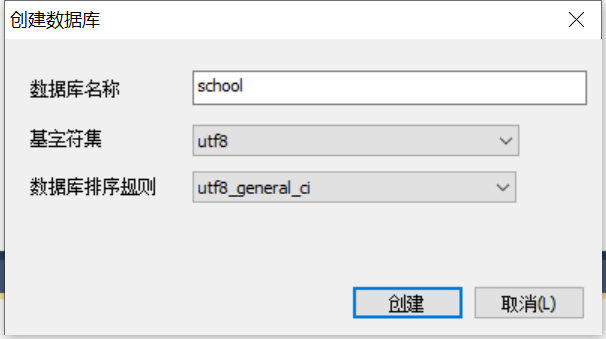
MySQL\MySQL Server 5.5\my.ini 文件修改 utf8。

### SQLyog

安装、注册。

建立连接。

创建数据库：



## 操作数据库

创建数据库

create database [if not exists] testdb;

删除数据库

drop database [if exists] testdb;

使用数据库

use `testdb`; -- 如果需要使用关键字，通过``将关键字包裹

查看所有数据库

show databases;

## 数据库列类型

### 数值

|  |  |  |
| --- | --- | --- |
| 数据类型 | 描述 | 字节数 |
| tinyint | 十分小的数据 | 1个字节 |
| smallint | 较小的数据 | 2个字节 |
| mediumint | 中等大小的数据 | 3个字节 |
| int | 标准整数 | 4个字节 |
| bigint | 较大的数据 | 8个字节 |
| float | 浮点数 | 4个字节 |
| double | 浮点数 | 8个字节 |
| decimal | 字符串形式的浮点数 |  |

### 字符串

|  |  |  |
| --- | --- | --- |
| 数据类型 | 描述 | 字节数 |
| char | 字符串固定大小 | 0-255 |
| varchar | 可变字符串 | 0-65535 |
| tinytext | 微型文本 | 2^8-1 |
| test | 文本 | 2^16-1 |

### 时间日期

|  |  |  |
| --- | --- | --- |
| 数据类型 | 描述 | 格式 |
| date | 日期格式 | YYYY-MM-DD |
| time | 时间格式 | HH:mm:ss |
| datetime | 时间格式 | YYYY-MM-DD HH:mm:ss |
| timestamp | 时间戳 |  |
| year | 年份 |  |

## 数据库字段属性

### Unsigned

无符号的整数。该列不能为负数。

### Zerofill

不足的位数，使用0来填充。

### 自增

在上一条记录的基础上加1。必须是整数类型。

### 非空

该字段不能为null或者不填。

### 默认

该字段的初始值。

## 操作数据库表

### 创建

create table [if not exists] `表名` (  
 `字段名` 列类型 [属性] [索引] [注释]，  
 `字段名` 列类型 [属性] [索引] [注释]，  
 `字段名` 列类型 [属性] [索引] [注释]，  
 ...  
)[表类型][字符集设置][注释];

### 修改

-- 修改表名  
alter table `旧表名` rename as `新表名`;  
-- 增加表的字段  
alter table `表名` add `字段名` 列属性;  
-- 修改表的字段  
alter table `表名` modify `字段名` 列属性;  
-- 修改约束  
alter table `表名` change `字段名` 列属性;

### 删除

-- 删除表的字段  
alter table `表名` drop `字段名`;  
-- 删除表  
drop table if exists `表名`;

## MyISAM 和 InnoDB 的区别

|  |  |  |
| --- | --- | --- |
|  | MyISAM | InnoDB |
| 索引类型 | 非聚簇 | 聚簇 |
| 事务支持 | 不支持 | 支持 |
| 数据表锁定 | 支持 | 支持 |
| 数据行锁定 | 不支持 | 支持 |
| 外键约束 | 不支持 | 支持 |
| 全文索引 | 支持 | 不支持（5.6之后支持） |
| 表空间大小 | 较小 | 较大，约为 MyISAM 的两倍 |

## 数据管理

### 外键

存在外键的时候，删除表的时候，需要先删除引用的从表，再删除主表。

alter table `主表名`   
add constraint `FK\_列名` foreign key(`主表列名`) references `从表名`(`从表列名`);

这种物理外键不推荐，推荐应用层解决该类问题。

### DML

数据管理语言。

#### insert

-- 如果不写字段，那么会默认根据所有字段进行一一匹配  
insert into `表名`[(字段1,字段2,字段3)]   
values (值1,值2,值3),(值4,值5,值6),(值7,值8,值9);

#### update

-- 不指定条件会修改当前表的所有该字段  
update `表名`  
set  
`字段1`=`值1`,  
`字段2`=`值2`,  
`字段3`=`值3`  
where `字段`=`条件`;

#### delete

-- 不指定条件会清空当前表的所有数据  
delete from `表名`  
where `字段`=`条件`;

delete 清空表数据后，重启数据库：

假如该表的引擎为 InnoDB：那么自增列会清空（数据存在内存中，断电即失）；

假如该表的引擎为 MyISAM：自增列会继续从上一个增量开始（数据存在文件中）。

#### truncate

-- 清空表中的所有数据  
truncate table `表名`;

truncate 会重新设置自增列，不会影响事务

### DQL

数据查询语言。

#### 查询语法

select [all | distinct]  
{\* | table.\* | [table.field[as alias]],[table.field2[as alias2]],[...]}  
from table\_name [as table\_alias]  
 [left | right | inner join table\_name2] -- 联合查询  
 [where ...] -- 指定结果需满足的条件  
 [group by ...] -- 指定结果按照哪几个字段来分组  
 [having] -- 过滤分组的记录必须满足的次要条件  
 [order by ...] -- 指定查询记录按一个或多个条件排序  
 [limit {[offset,]row\_count | row\_countOffset offset}]; -- 指定查询记录从哪条至哪条

{} 代表必须，[] 代表可选。

#### where 条件

可以存在多个条件，使用逻辑运算符连接。返回结果都为布尔值。

|  |  |  |
| --- | --- | --- |
| 运算符 | 语法 | 描述 |
| and && | a and b a && b | 逻辑与 |
| or || | a or b a || b | 逻辑或 |
| not ! | not a ! a | 逻辑非 |

具体条件判断，使用比较运算符。

|  |  |  |
| --- | --- | --- |
| 运算符 | 语法 | 描述 |
| is null | a is null | 操作符为 null ，结果为真 |
| is not null | a is not null | 操作符不为 null ，结果为真 |
| between and | a between b and c | a 在 b 和 c 之间，结果为真 |
| like | a like b | a 匹配 b，结果为真 |
| in | a in (a1,a2,a3...) | a 等于 a1、a2、a3...的某个值，结果为真 |

like：

%：匹配不限数量的随机字

\_：匹配一个随机字

#### 联表查询

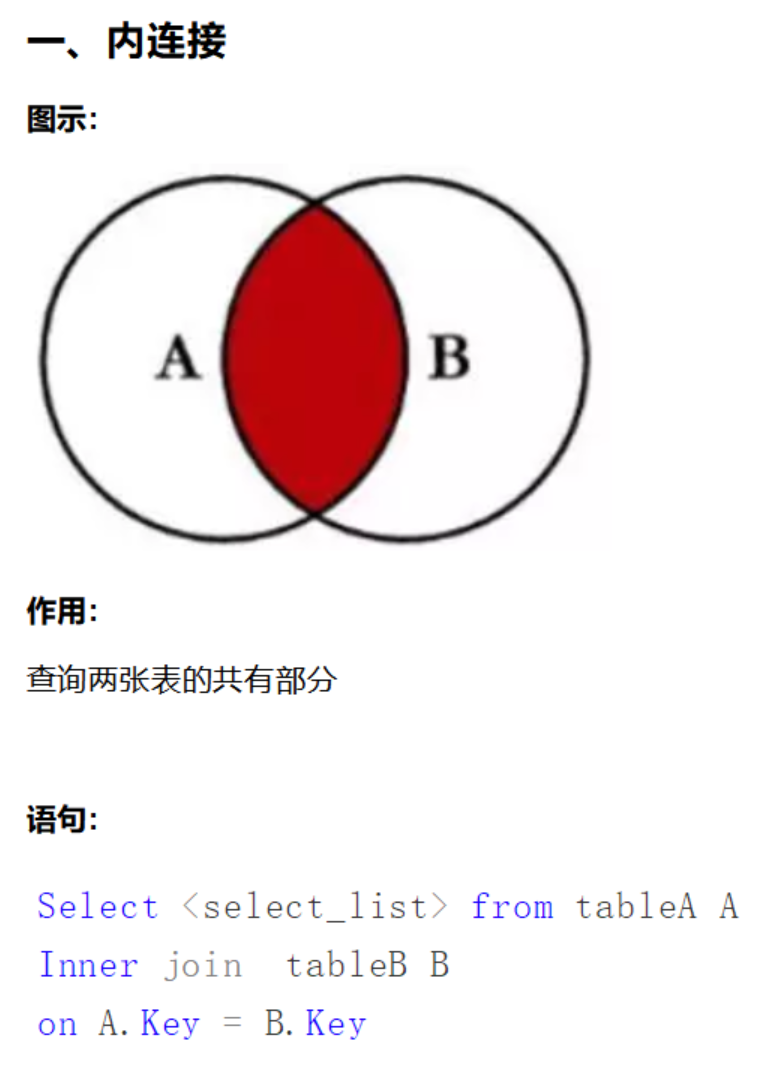


image-20210902100357826

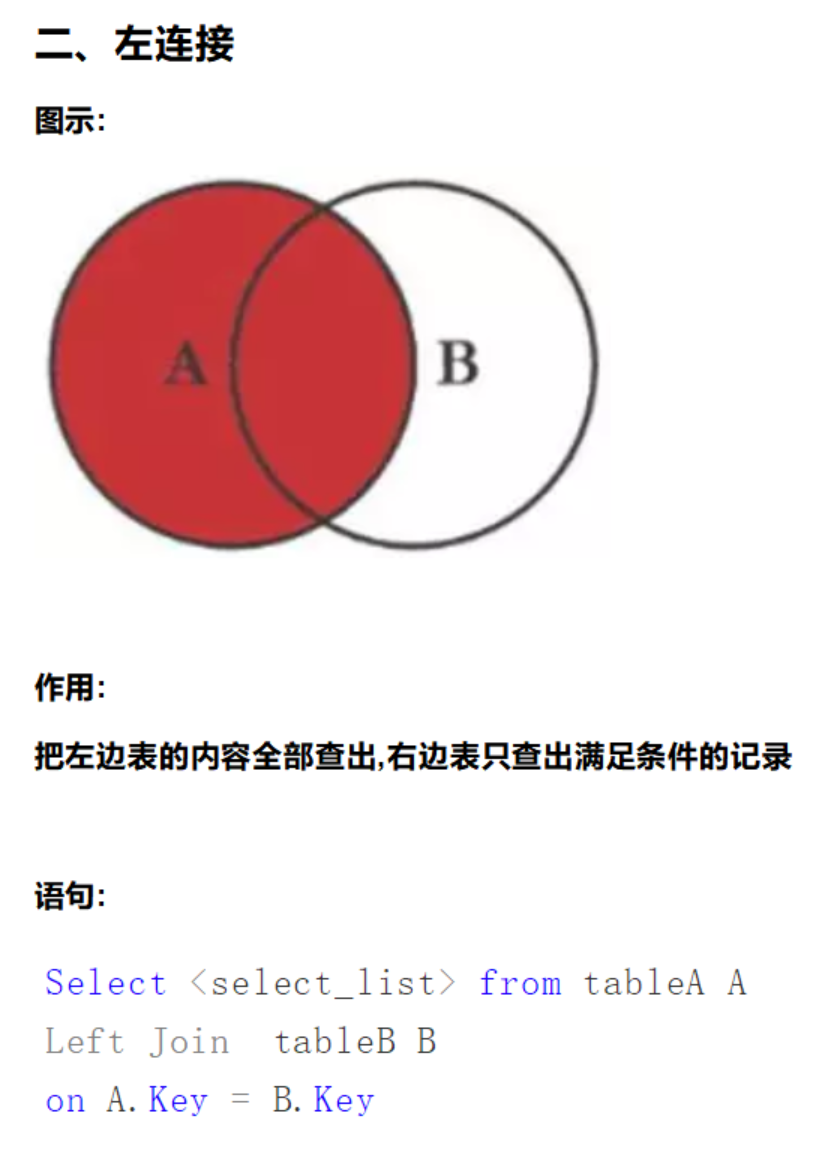


image-20210902100414012

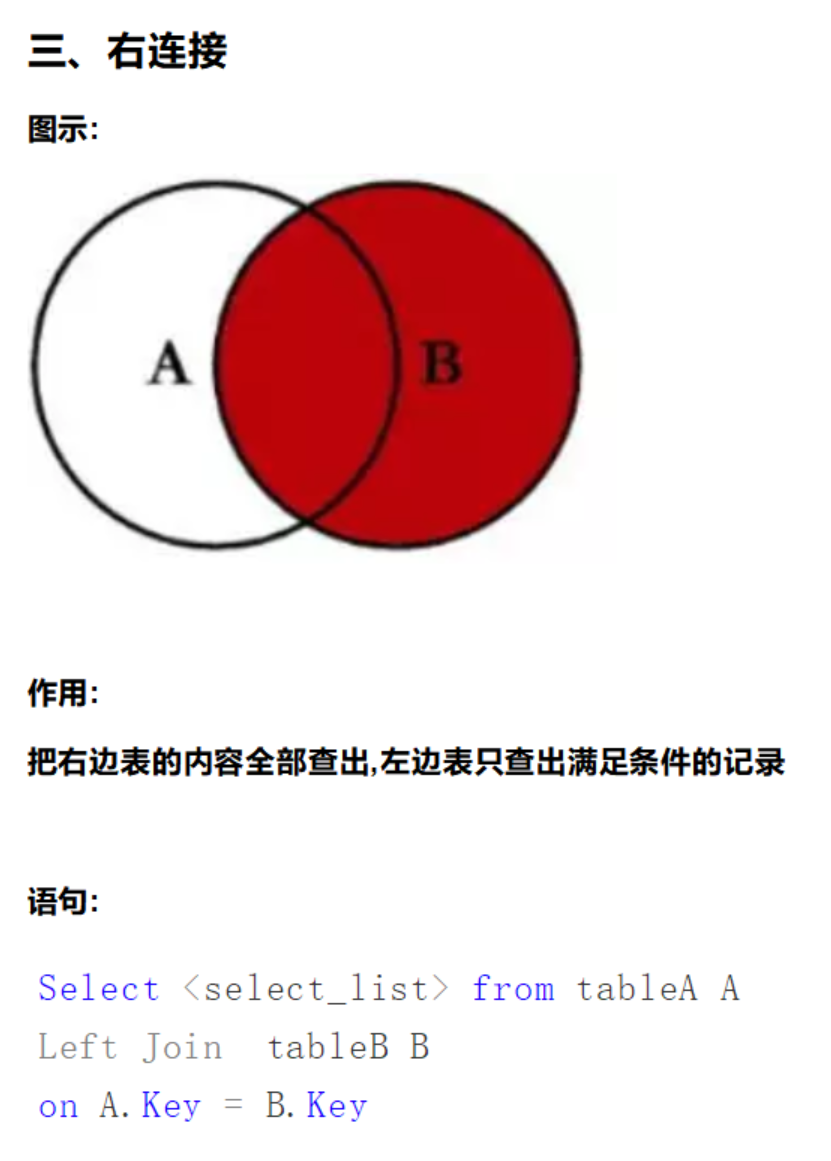


image-20210902100429989

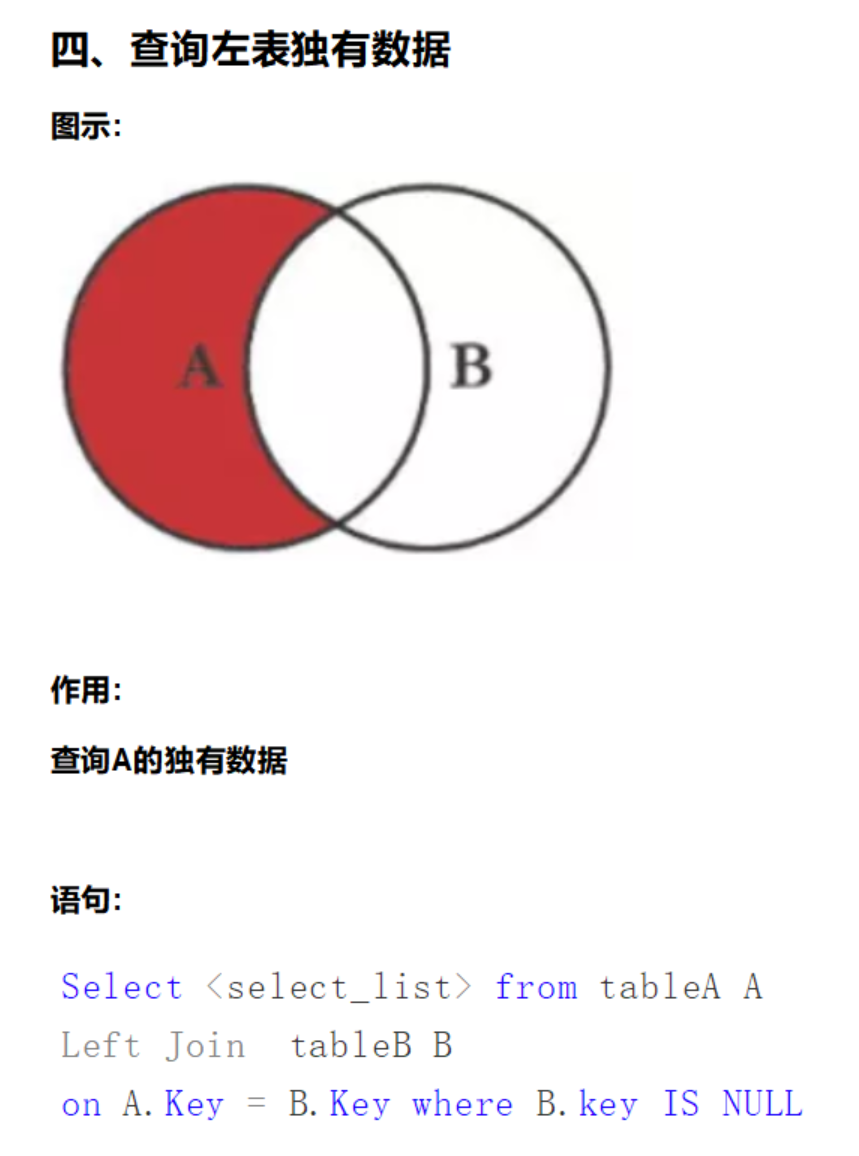


image-20210902100450933

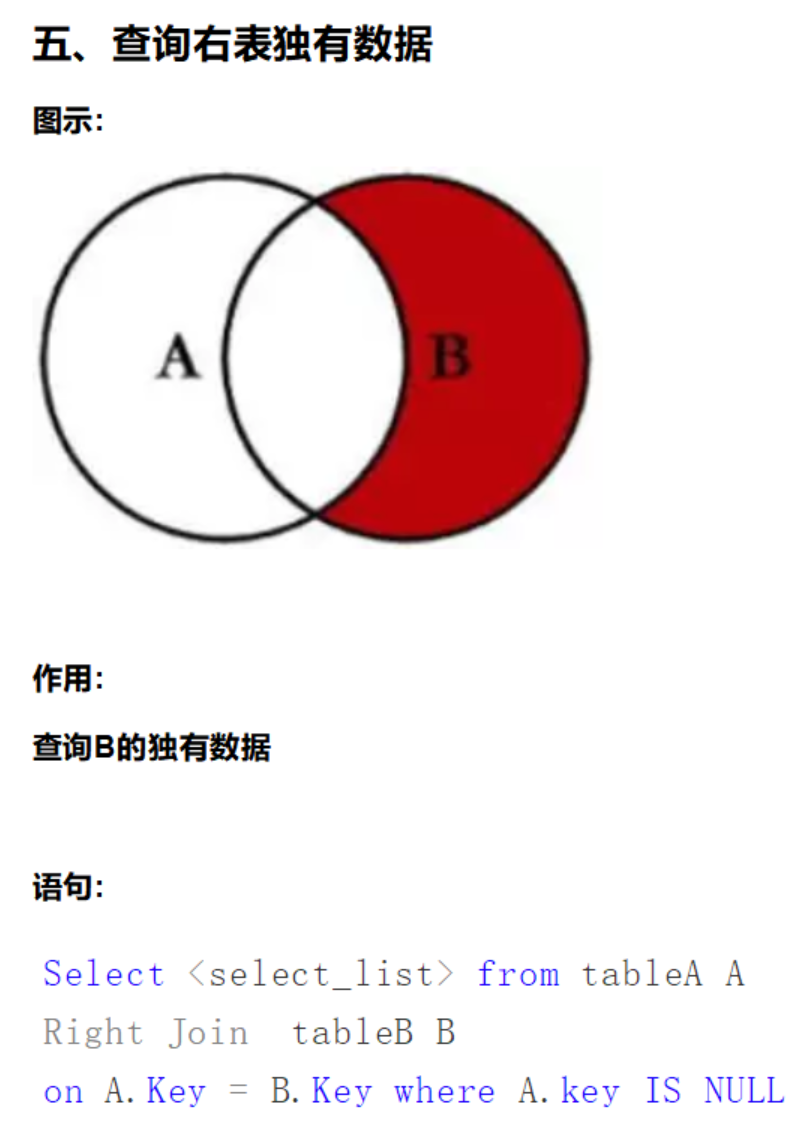


image-20210902100505996

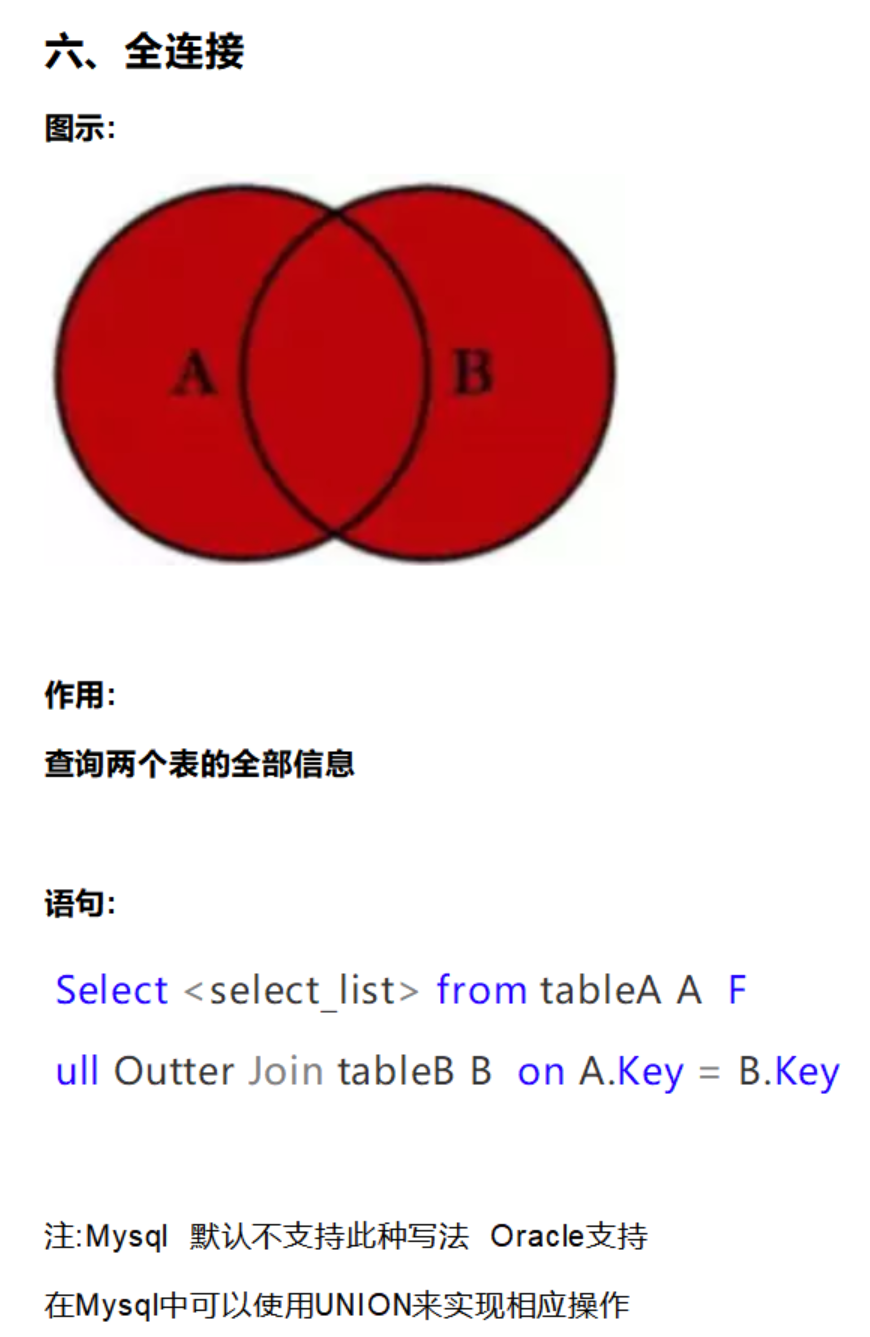


image-20210902100530318



image-20210902100616400

#### 排序

order by

asc：正序

desc：降序

#### 分页

limit

limit((n-1)\*pageSize,pageSize)  
  
-- n：当前页  
-- pageSize：页面大小

#### 子查询

在 where 语句中嵌套一个子查询语句。

## MySQL函数

### 常用函数

-- 绝对值  
select abs(-8);  
  
-- 向上取整  
select ceiling(9.4);  
  
-- 向下取整  
select floor(9.4);  
  
-- 0~1 之前的随机数  
select rand();  
  
-- 判断一个数的符号  
-- 负数返回-1，0返回0，整数返回1  
select sign(-10);  
  
-- 字符串长度  
select char\_length('测试字符串长度');  
  
-- 字符串拼接  
select concat('你','好','阿');  
  
-- 字符串替换  
select insert('设计模式',1,2,'不懂');  
  
-- 转小写字母  
select lower('Hello,World!');  
  
-- 转大写字母  
select upper('Hello,World!');  
  
-- 当前日期 yyyy-MM-DD HH:mm:ss  
select now();  
  
-- 系统当前用户  
select user();  
  
-- MySQL版本  
select version();

### 聚合函数

|  |  |
| --- | --- |
| 函数 | 描述 |
| count() | 计数 |
| sum() | 求和 |
| avg() | 平均值 |
| max() | 最大值 |
| min() | 最小值 |

count(字段)：会忽略所有的 null 值  
count(\*)：计算行数（所有字段），不会忽略 null 值  
count(1)：计算行数（一个字段），不会忽略 null 值

## MD5加密

MD5：不可逆，相同的值加密后的 md5 值是一样的。

-- 插入数据时加密  
insert into `表名` values(`id字段`,`姓名字段`,md5(`密码字段`));  
-- 修改历史数据为加密  
update `表名` set `密码字段`=md5(`密码字段`);  
  
-- 校验  
select \* from `表名` where `姓名字段`=`姓名` and `密码字段` = md5(`密码`);

## 事务

### 事务管理

#### 原子性（Atomicity）

原子性是指事务是一个不可分割的工作单位，事务中的操作要么都发生，要么都不发生。

#### 一致性（Consistency）

事务前后数据的完整性必须保持一致。

#### 隔离性（Isolation）

事务的隔离性是多个用户并发访问数据库时，数据库为每一个用户开启的事务，不能被其他事务的操作数据所干扰，多个并发事务之间要相互隔离。

#### 持久性（Durability）

持久性是指一个事务一旦被提交，它对数据库中数据的改变就是永久性的，接下来即使数据库发生故障也不应该对其有任何影响。

### 事务隔离级别

#### 脏读

指一个事务读取了另外一个事务未提交的数据。

#### 不可重复读

在一个事务内读取表中的某一行数据，多次读取结果不同。

（这个不一定是错误，只是某些场合不对）

#### 虚读(幻读)

是指在一个事务内读取到了别的事务插入的数据，导致前后读取数量总量不一致。

（一般是行影响，如下图所示：多了一行）

### MySQL 事务

MySQL 默认开启事务。

操作步骤：

-- 关闭自动提交事务  
set autocommit = 0;  
  
-- 事务开启  
start transaction;  
  
-- 手动提交  
commit;  
  
-- 手动回滚  
rollback;  
  
-- 开启自动提交事务  
set autocommit = 1;  
  
-- 创建保存点  
savepoint `保存点名`;  
  
-- 回滚到保存点  
rollback to savepoint `保存点名`;  
  
-- 删除设置的保存点  
release savepoint `保存点名`;

## 索引

索引是帮助 MySQL 搞笑获取数据的数据结构。

### 优缺点

优点：可以快速检索，减少I/O次数，加快检索速度；根据索引分组和排序，可以加快分组和排序。

缺点：索引本身也是表，因此会占用存储空间，一般来说，索引表占用的空间是数据表的1.5倍；索引表的维护和创建需要时间成本，这个成本随着数据量增大而增大；构建索引会降低数据表的修改操作（删除，添加，修改）的效率，因为在修改数据表的同时还需要修改索引表。

### 分类

#### 主键索引

主索引，根据主键pk\_clolum（length）建立索引，**不允许重复，不允许空值**。

alter table `table\_name` add primary key pk\_index\_col(`col`);

#### 唯一索引

用来建立索引的列的值必须是**唯一的，允许空值**。

alter table `table\_name` add unique unique\_index\_col(`col`);

#### 普通索引

用表中的普通列构建的索引，没有任何限制。

alter table `table\_name` add index index\_col(`col`);

#### 全文索引

用大文本对象的列构建的索引。

alter table `table\_name` add fulltext index ft\_index\_col(`col`);

#### 组合索引

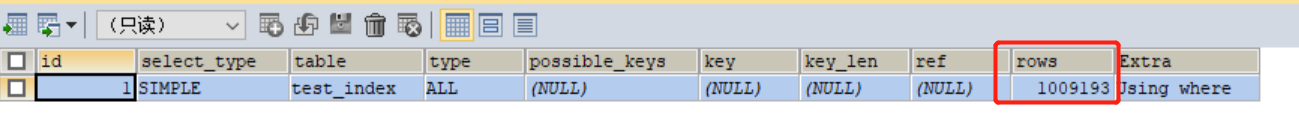
用多个列组合构建的索引，这多个列中的值不允许有空值。

alter table `table\_name` add index index\_name(`col1`,`col2`,`clo3`);

### 测试索引

-- 自定义函数，创建100W条数据  
DELIMITER $$  
CREATE FUNCTION mock\_data()  
RETURNS INT   
BEGIN  
 DECLARE num INT DEFAULT 1000000;  
 DECLARE i INT DEFAULT 0;  
 WHILE i<num  
 DO  
 INSERT test\_index(`name`,`sex`,`email`,`phone`,`address`)  
 VALUE (CONCAT('张三',i),'1','123456789@qq.com','13666667777',CONCAT('我是地址',i));  
 SET i = i+1;  
 END WHILE;  
 RETURN i;  
END;  
-- 执行函数  
SELECT mock\_data();

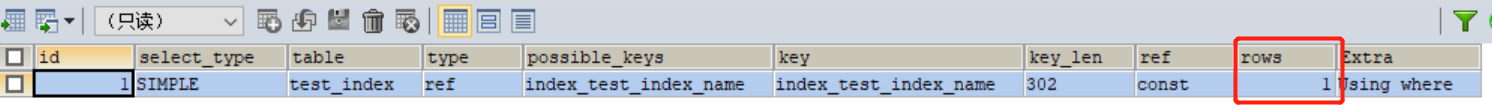
-- 查看sql执行效率  
DESCRIBE SELECT \* FROM test\_index WHERE NAME = '张三9999';



可以看到总共查询了 1009193 行数据。

-- 创建索引  
ALTER TABLE test\_index ADD INDEX index\_test\_index\_name(`name`);

-- 再次查询sql执行效率  
DESCRIBE SELECT \* FROM test\_index WHERE NAME = '张三9999';



可以看到只查询了 1 行数据。

### 增加索引原则

1、索引不是越多越好。

2、不要对经常变动的数据加索引。

3、小数据量的表不需要加索引。

4、索引一般加在常用来查询的字段上。

### 索引实现原理

#### 哈希索引

只有memory（内存）存储引擎支持哈希索引，哈希索引用索引列的值计算该值的hashCode，然后在hashCode相应的位置存执该值所在行数据的物理位置，因为使用散列算法，因此访问速度非常快，但是一个值只能对应一个hashCode，而且是散列的分布方式，因此哈希索引不支持范围查找和排序的功能。

#### 全文索引

FULLTEXT（全文）索引，仅可用于 MyISAM 和 InnoDB，针对较大的数据，生成全文索引非常的消耗时间和空间。对于文本的大对象，或者较大的 CHAR 类型的数据，如果使用普通索引，那么匹配文本前几个字符还是可行的，但是想要匹配文本中间的几个单词，那么就要使用 LIKE %word% 来匹配，这样需要很长的时间来处理，响应时间会大大增加，这种情况，就可使用时 FULLTEXT 索引了，在生成 FULLTEXT 索引时，会为文本生成一份单词的清单，在索引时根据这个单词的清单来索引。

#### B树索引

BTree 是平衡搜索多叉树，设树的度为 2d（d>1），高度为 h，那么 BTree 要满足以下条件：

1、每个叶子结点的高度一样，等于 h；

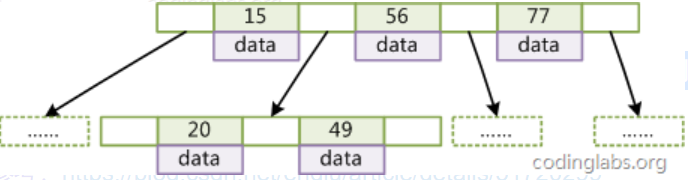
2、每个非叶子结点由 n-1 个 key 和 n 个指针 point 组成，其中 d<=n<=2d ，key 和 point 相互间隔，结点两端一定是 key；

3、叶子结点指针都为 null；

4、非叶子结点的 key 都是 [key,data] 二元组，其中 key 表示作为索引的键，data 为键值所在行的数据。

在 BTree 的结构下，就可以使用二分查找的查找方式，查找复杂度为 h\*log(n)，一般来说树的高度是很小的，一般为 3 左右，因此 BTree 是一个非常高效的查找结构。

BTree 结构：



#### B+树索引

B+Tree 是 BTree 的一个变种，设 d 为树的度数，h 为树的高度，B+Tree 和 BTree 的不同主要在于：

1、B+Tree 中的非叶子结点不存储数据，只存储键值；

2、B+Tree 的叶子结点没有指针，所有键值都会出现在叶子结点上，且 key 存储的键值对应 data 数据的物理地址；

3、B+Tree 的每个非叶子节点由 n 个键值 key 和 n 个指针 point 组成。

**B+Tree对比BTree的优点：**

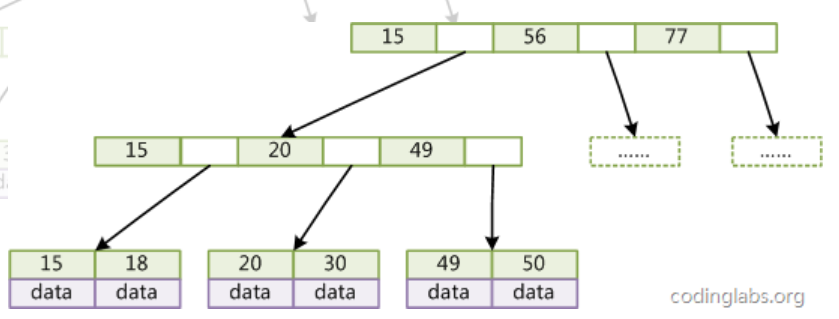
1、磁盘读写代价更低

一般来说 B+Tree 比 BTree 更适合实现外存的索引结构，因为存储引擎的设计专家巧妙的利用了外存（磁盘）的存储结构，即磁盘的最小存储单位是扇区（sector），而操作系统的块（block）通常是整数倍的 sector，操作系统以页（page）为单位管理内存，一页（page）通常默认为4K，数据库的页通常设置为操作系统页的整数倍，因此**索引结构的节点被设计为一个页的大小**，然后利用外存的“预读取”原则，每次读取的时候，把整个节点的数据读取到内存中，然后在内存中查找，已知内存的读取速度是外存读取 I/O 速度的几百倍，那么提升查找速度的关键就在于尽可能少的磁盘 I/O，那么可以知道，每个节点中的 key 个数越多，那么树的高度越小，需要 I/O 的次数越少，因此一般来说 B+Tree 比 BTree 更快，因为 B+Tree 的非叶节点中不存储 data，就可以存储更多的 key。

2、查询速度更稳定

由于 B+Tree 非叶子节点不存储数据（data），因此所有的数据都要查询至叶子节点，而叶子节点的高度都是相同的，因此所有数据的查询速度都是一样的。

B+Tree 结构：



## 数据导入导出

### 导出

#### 可视化

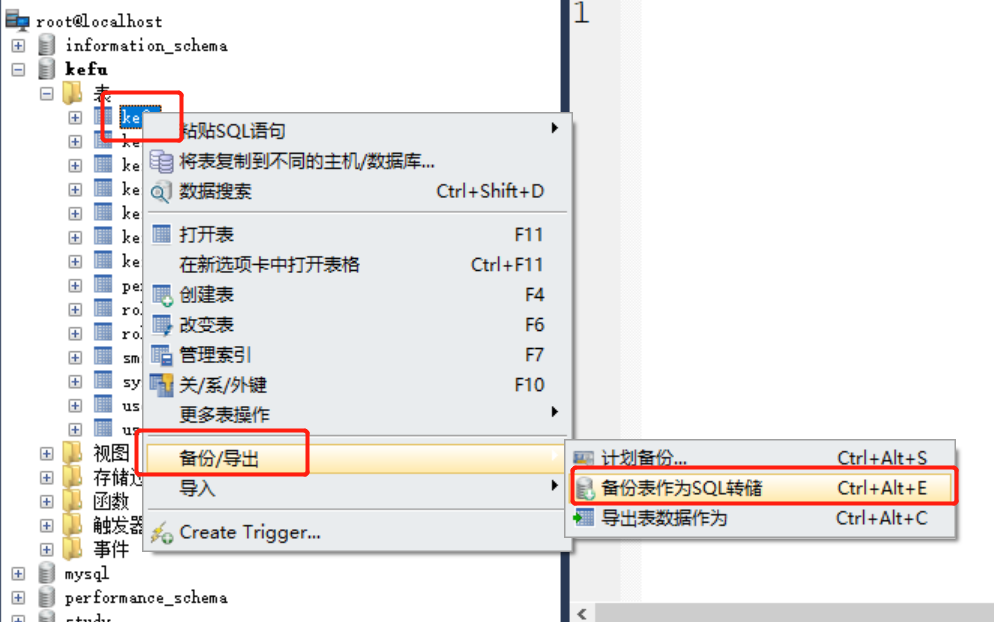


image-20210902172942872

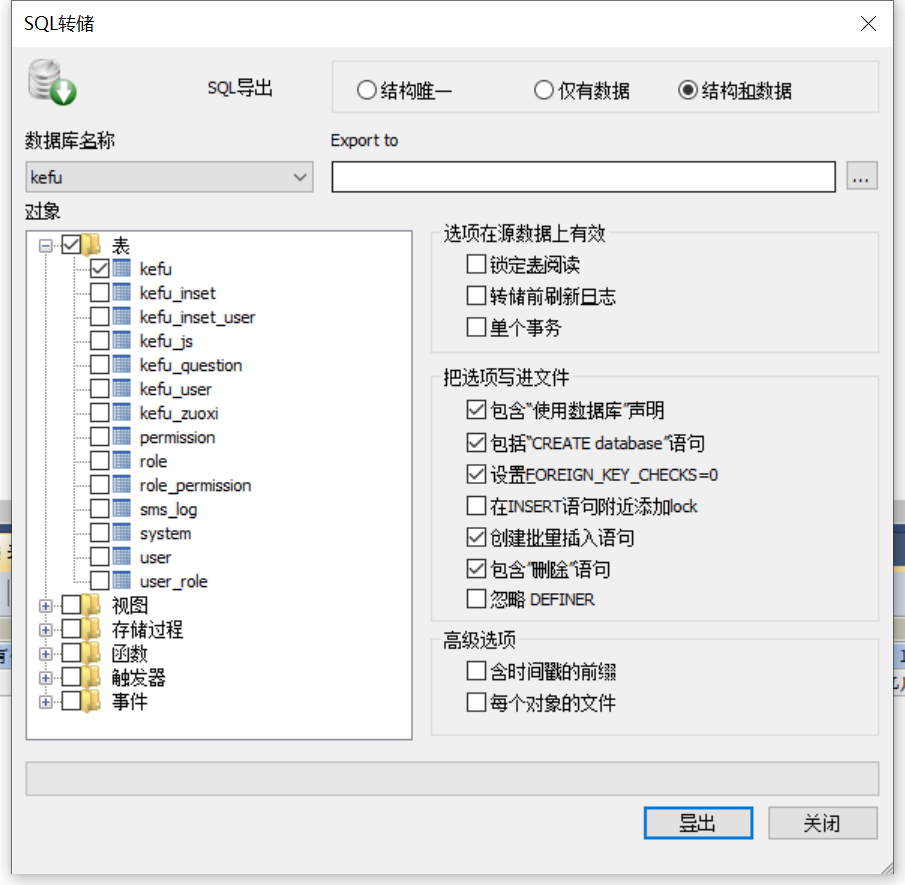


image-20210902172956156

#### 命令行

mysqldump -u root -p123456 kefu kefu > D:/testexport.sql

### 导入

#### 可视化

选中需要导入的数据库，直接拖动文件，全部执行即可。

#### 命令行

登录数据库，进入需要执行的数据库。

source D:/testexport.sql

## 数据库设计

1、收集信息，分析需求。

2、标识实体。

3、标识实体之间的关系。

## 数据库三大范式

### 第一范式1NF

数据表中的每一列(字段)，必须是不可拆分的最小单元，也就是确保每一列的原子性，而不是集合。

### 第二范式2NF

满足1NF的基础上，要求：表中的所有列，都必需依赖于主键，而不能有任何一列与主键没有关系（一个表只描述一件事情）。第二范式消除表的无关数据。

### 第三范式3NF

满足2NF的基础上，任何非主属性不依赖于其它非主属性（在2NF基础上消除传递依赖）（也表明不允许数据存在冗余的现象）。

### 范式和性能的问题

关联查询的表不得超过三张表。

1、商业化的需求和目标，数据库的性能更加重要。

2、规范性能的问题时，适当参考三大范式。

3、给某些表增加冗余字段，将多表查询变为单表查询。

4、增加计算列，减少数据库函数的调用。

## JDBC

JDBC 是 sun 公司提供的一套接口，接口都有调用者和实现者，面向接口调用，面向接口写实现类，都属于面向接口编程。

public static void main(String[] args) throws ClassNotFoundException, SQLException {  
 // 加载驱动  
 Class.forName("com.mysql.jdbc.Driver");  
 // 用户信息  
 // url格式：协议://主机ip:端口号/数据库名?参数1&参数2&参数3  
 String url = "jdbc:mysql://localhost:3306/kefu?useUnicode=true&characterEncoding=utf8&useSSL=true";  
 String username = "root";  
 String password = "123456";  
 // 创建连接  
 Connection connection = DriverManager.getConnection(url, username, password);  
 // 创建sql对象  
 Statement statement = connection.createStatement();  
 // 执行sql，获取结果  
 String sql = "select \* from role";  
 ResultSet resultSet = statement.executeQuery(sql);  
 // 处理结果  
 while (resultSet.next()) {  
 System.out.println("-----------------start--------------------");  
 System.out.println(resultSet.getString("id"));  
 System.out.println(resultSet.getString("name"));  
 System.out.println(resultSet.getString("description"));  
 System.out.println("-----------------end--------------------");  
 }  
 // 释放连接  
 resultSet.close();  
 statement.close();  
 connection.close();  
}

### PreparedStatement 对象

将传递过来的参数当作字符，如果存在转移字符，该字符会被直接转义。

public static void main(String[] args) throws ClassNotFoundException, SQLException {  
 // 加载驱动  
 Class.forName("com.mysql.jdbc.Driver");  
 // 用户信息  
 // url格式：协议://主机ip:端口号/数据库名?参数1&参数2&参数3  
 String url = "jdbc:mysql://localhost:3306/kefu?useUnicode=true&characterEncoding=utf8&useSSL=true";  
 String username = "root";  
 String password = "123456";  
 // 创建连接  
 Connection connection = DriverManager.getConnection(url, username, password);  
 // 创建sql对象  
 String sql = "select \* from role where id = ?";  
 PreparedStatement preparedStatement = connection.prepareStatement(sql);  
 // 给参数进行赋值  
 preparedStatement.setInt(1, 1);  
 // 执行sql，获取结果  
 ResultSet resultSet = preparedStatement.executeQuery();  
 // 处理结果  
 while (resultSet.next()) {  
 System.out.println("-----------------start--------------------");  
 System.out.println(resultSet.getString("id"));  
 System.out.println(resultSet.getString("name"));  
 System.out.println(resultSet.getString("description"));  
 System.out.println("-----------------end--------------------");  
 }  
 // 释放连接  
 resultSet.close();  
 preparedStatement.close();  
 connection.close();  
}

### JDBC 操作事务

public static void main(String[] args) throws ClassNotFoundException, SQLException {  
 // 加载驱动  
 Class.forName("com.mysql.jdbc.Driver");  
 // 用户信息  
 // url格式：协议://主机ip:端口号/数据库名?参数1&参数2&参数3  
 String url = "jdbc:mysql://localhost:3306/kefu?useUnicode=true&characterEncoding=utf8&useSSL=true";  
 String username = "root";  
 String password = "123456";  
  
 Connection connection = null;  
 PreparedStatement preparedStatement = null;  
 ResultSet resultSet = null;  
 try {  
 // 创建连接  
 connection = DriverManager.getConnection(url, username, password);  
 // 开启手动提交事务  
 connection.setAutoCommit(false);  
 // 创建sql对象  
 String sql = "update role set id = ? and name = ? and description=?";  
 preparedStatement = connection.prepareStatement(sql);  
 // 给参数进行赋值  
 preparedStatement.setInt(1, 1);  
 preparedStatement.setString(2, "李思思");  
 preparedStatement.setString(3, "我是一个描述");  
 // 执行sql  
 preparedStatement.execute();  
 // 出现异常时，因为开启了事务，会进行回滚，rollback() 可以不用显示定义  
 int i = 10 / 0;  
 // 手动提交事务  
 connection.commit();  
 } catch (Exception e) {  
 if (null != connection)  
 connection.rollback();  
 e.printStackTrace();  
 } finally {  
 // 释放连接  
 if (null != resultSet) {  
 resultSet.close();  
 }  
 if (null != preparedStatement) {  
 preparedStatement.close();  
 }  
 if (null != connection) {  
 // 关闭手动提交事务  
 connection.setAutoCommit(true);  
 connection.close();  
 }  
 }  
}

### 数据库连接池

池化技术：准备一些预先的资源，进程过来就连接预先准备好的。

数据库连接池负责分配、管理和释放数据库连接，它允许应用程序重复使用一个现有的数据库连接，而不是再重新建立一个；释放空闲时间超过最大空闲时间的数据库连接来避免因为没有释放数据库连接而引起的数据库连接遗漏。这项技术能明显提高对数据库操作的性能。

#### 参数（Druid）

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数值 | 参数描述 |
| initialSize | 3 | 初始化配置 |
| minIdle | 3 | 最小连接数 |
| maxActive | 15 | 最大连接数 |
| maxWait | 5000 | 获取连接超时时间（单位：ms） |
| timeBetweenEvictionRunsMillis | 90000 | 连接有效性检测时间（单位：ms） |
| testOnBorrow | false | 获取连接检测 |
| testOnReturn | false | 归还连接检测 |
| minEvictableIdleTimeMillis | 1800000 | 最大空闲时间(单位ms) |
| testWhileIdle | true | 在获取连接后，确定是否要进行连接空间时间的检查 |

#### 参数（DBCP）

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数值 | 参数描述 |
| initialSize | 3 | 初始化配置 |
| minIdle | 3 | 最小连接数 |
| maxIdle | 15 | 最大空闲连接 |
| maxTotal | 15 | 最大连接数 |
| maxWaitMillis | 5000 | 获取连接超时时间（单位：ms） |
| timeBetweenEvictionRunsMillis | 90000 | 心跳检测时间（单位：ms） |
| minEvictableIdleTimeMillis | 1800000 | 最大空闲时间（单位：ms） |
| testOnBorrow | FALSE | 获取连接检测 |
| testOnReturn | FALSE | 归还连接检测 |
| numTestsPerEvictionRun | -1 | 空闲连接检查的个数 |
| testWhileIdle | TRUE | 是否开启对空闲连接的检查 |

#### 参数（C3P0）

|  |  |  |
| --- | --- | --- |
| 参数名称 | 参数值 | 参数描述 |
| initialPoolSize | 3 | 初始化配置 |
| minPoolSize | 3 | 最小连接数 |
| maxPoolSize | 15 | 最大连接数 |
| acquireIncrement | 1 | 每次获取的个数 |
| checkoutTimeout | 5000 | 获取连接超时时间（单位：ms） |
| idleConnectionTestPeriod | 90 | 心跳检测时间（单位：s） |
| maxIdleTime | 1800 | 最大空闲时间（单位：s） |
| testConnectionOnCheckout | FALSE | 获取连接检测 |
| testConnectionOnCheckin | FALSE | 归还连接检测 |
| numHelperThreads | 1 | 默认开启连接数 |