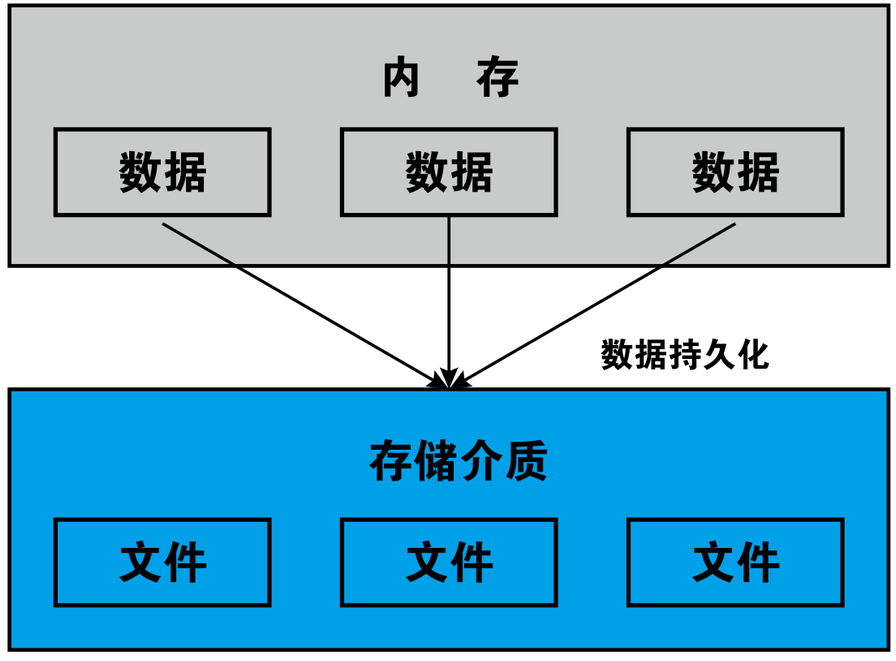
目　录

一　JDBC 概述

1.1　数据的持久化

持久化 (persistence)：把数据保存到可掉电式存储设备中以供之后使用。大多数情况下，特别是企业级应用，数据持久化意味着将内存中的数据保存到硬盘上加以“固化”，而持久化的实现过程大多通过各种关系数据库来完成。

持久化的主要作用是将内存中的数据存储在关系型数据库中，当然也可以存储在磁盘文件、XML 数据文件中。



1.2 Java 中的数据存储技术

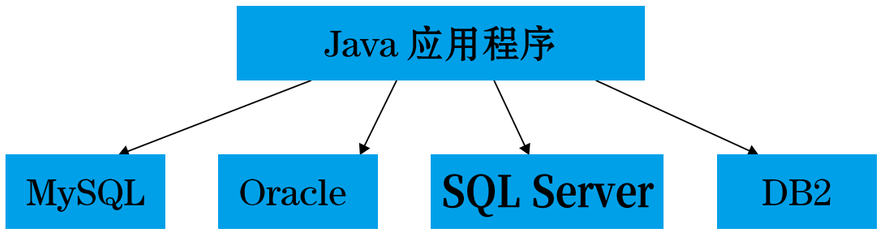
· 在 Java 中，数据库存取技术可分为如下几类：
  · JDBC 直接访问数据库；
  · JDO(Java Data Object) 技术；
  · 第三方 O/R 工具，如 Hibernate,Mybatis 等；
· JDBC 是 java 访问数据库的基石，JDO、Hibernate、MyBatis 等只是更好的封装了 JDBC。

1.3 JDBC 介绍

· JDBC(Java Database Connectivity) 是一个独立于特定数据库管理系统、通用的 SQL 数据库存取和操作的公共接口（一组 APl），定义了用来访问数据库的标准 Java 类库，（java.sql,javax.sql）使用这些类库可以以一种标准的方法、方便地访问数据库资源。

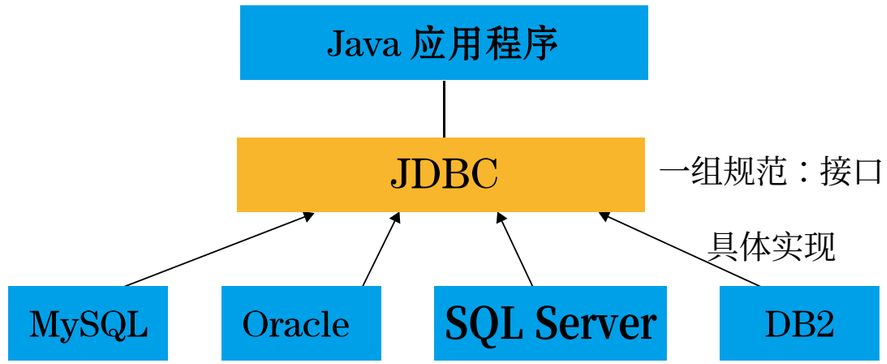
· JDBC 为访问不同的数据库提供了一种统一的途径，为开发者屏蔽了一些细节问题。

· JDBC 的目标是使 Java 程序员使用 JDBC 可以连接任何提供了 JDBC 驱动程序的数据库系统，这样就使得程序员无需对特定的数据库系统的特点有过多的了解，从而大大简化和加快了开发过程。
如果没有 JDBC，那么 Java 程序访问数据库时是这样的：



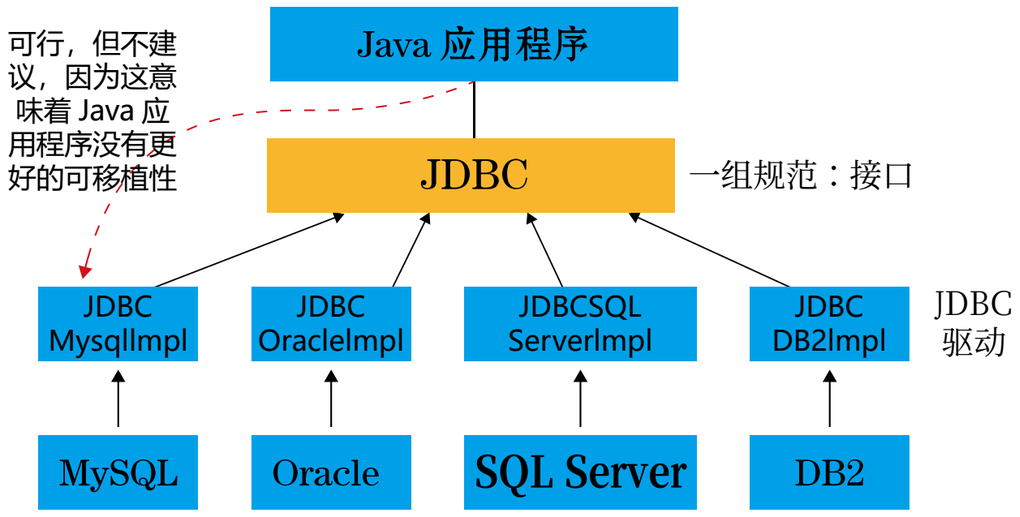
我们认为的连接应该是这样的

有了 JDBC，Java 程序访问数据库时是这样的：



真实的连接应该是这样的

总结如下：



1.4 JDBC 体系结构

· JDBC 接口（API）包括两个层次：

· 面向应用的 API：Java API，抽象接口，供应用程序开发人员使用（连接数据库，执行 SQL 语句，获得结果）。

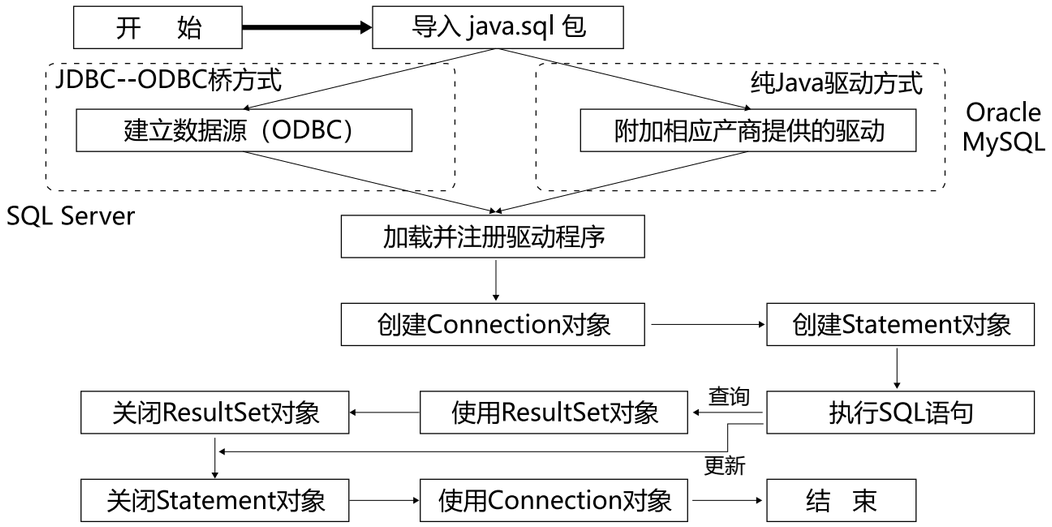
· 面向数据库的 API：Java Driver API，供开发商开发数据库驱动程序用。

JDBC 是 sun 公司提供一套用于数据库操作的接口，java 程序员只需要面向这套接口编程即可。

不同的数据库厂商，需要针对这套接口，提供不同实现。不同的实现的集合，即为不同数据库的驱动。

————面向接口编程

1.5 JDBC 程序编写步骤



补充：ODBC（Open Database Connectivity，开放式数据库连接），是微软在 Windows 平台下推出的。使用者在程序中只需要调用 ODBCAPI，由 ODBC 驱动程序将调用转换成为对特定的数据库的调用请求。

二　获取数据库连接

2.1 要素一：Driver 接口实现类

2.1.1 Driver 接口介绍

· java.sql.Driver 接口是所有 JDBC 驱动程序需要实现的接口。这个接口是提供给数据库厂商使用的，不同数据库厂商提供不同的实现。

· 在程序中不需要直接去访问实现了 Driver 接口的类，而是由驱动程序管理器类 (java.sql.DriverManager) 去调用这些 Driver 实现。

Oracle 的驱动：oracle.jdbc.driver.OracleDriver

mySql 的驱动： com.mysql.jdbc.Driver

2.1.2 加载与注册 JDBC 驱动

· 加载驱动：加载 JDBC 驱动需调用 Class 类的静态方法 forName()，向其传递要加载的 JDBC 驱动的类名。

· Class.forName(“ com.mysql.jdbc.Driver ”);

· 注册驱动：DriverManager 类是驱动程序管理器类，负责管理驱动程序

· 使用 DriverManager.registerDriver(com.mysql.jdbc.Driver) 来注册驱动

· 通常不用显式调用 DriverManager 类的 registerDriver() 方法来注册驱动程序类的实例，因为 Driver 接口的驱动程序类都包含了静态代码块，在这个静态代码块中，会调用 DriverManager.registerDriver() 方法来注册自身的一个实例。下图是 MySQL 的 Driver 实现类的源码：

```
public class Driver extends NonRegisteringDriver implements java.
sql.Driver {
    static {
        try {
            java.sql.DriverManager.registerDriver(new Driver());
        } catch (SQLException e) {
            throw new RuntimeException("Can't register driver!")
        }
    }
```

2.2 要素二：URL

· JDBC URL 用于标识一个被注册的驱动程序，驱动程序管理器通过这个 URL 选择正确的驱动程序，从而建立到数据库的连接。
· JDBC URL 的标准由三部分组成，各部分间用冒号分隔。
· jdbc：子协议：子名称
· 协议：JDBC URL 中的协议总是 jdbc
· 子协议：子协议用于标识一个数据库驱动程序
· 子名称：一种标识数据库的方法。子名称可以依不同的子协议而变化，用子名称的目的是为了定位数据库提供足够的信息。包含主机名（对应服务端的 ip 地址），端口号，数据库名
· 举例：

jdbc:mysql://localhost:3306/test
协议　子协议　子名称

· 几种常用数据库的 JDBC URL
· MySQL 的连接 URL 编写方式：
· jdbc:mysql:// 主机名称 :mysql 服务端口号 / 数据库名称 ? 参数 = 值 & 参数 = 值
· jdbc:mysql://localhost:3306/atguigu
· jdbc:mysql://localhost:3306/atguigu?useUnicode=true&characterEncoding=utf8（如果 JDBC 程序与服务器端的字符集不一致，会导致乱码，那么可以通过参数指定服务器端的字符集）
· jdbc:mysql://localhost:3306/atguigu?user=root&password=123456
· Oracle 9i 的连接 URL 编写方式：
· jdbc:oracle:thin:@ 主机名称 :oracle 服务端口号 : 数据库名称
· jdbc:oracle:thin:@localhost:1521:atguigu
· SQLServer 的连接 URL 编写方式：
· jdbc:sqlserver:// 主机名称 :sqlserver 服务端口号 :DatabaseName= 数据库名称
· jdbc:sqlserver://localhost:1433:DatabaseName=atguigu

2.3 要素三：用户名和密码

· user,password 可以用 “ 属性名 = 属性值 ” 方式告诉数据库。
· 可以调用 DriverManager 类的 getConnection() 方法建立到数据库的连接。

2.4 数据库连接方式举例

```
// jar 包要装 mysql-connector-java-8.0 以上版本
public class ConnectionTest {
	// 方式一
	public static void main(String[] args) throws SQLException {
		Driver driver = new com.mysql.cj.jdbc.Driver();
		// url:http://localhost:8080/gmall/keyboard.jpg
		// jdbc:mysql: 协议
		// localhost:ip 地址
		// 3306：默认 mysql 的端口号
		// mysql:mysql 的存储 mysql 本身一些信息的数据库
		String url = "jdbc:mysql://localhost:3306/mysql?characterEncoding=utf8&useSSL=false&serverTimezone=UTC&rewriteBatchedStatements=true";
		// 将用户名和密码封装在 Properties 中
		Properties info = new Properties();
		info.setProperty("user", "root");
		info.setProperty("password", "abc123");

		Connection conn = driver.connect(url, info);
		System.out.println(conn);
	}

	@Test
	// 方式二：对方式一的迭代：在如下的程序中不出现第三方 api，使得程序具有更好的移植性
	public void tese2() throws Exception {
		//1. 获取 Driver 实现类对象，使用反射
		Class clazz = Class.forName("com.mysql.cj.jdbc.Driver");
		Driver driver = (Driver) clazz.newInstance();
		//2. 提供要连接的数据库
		String url = "jdbc:mysql://localhost:3306/mysql?characterEncoding=utf8&useSSL=false&serverTimezone=UTC&rewriteBatchedStatements=true";
		//3. 提供需要连接的用户名和密码
		Properties info = new Properties();
		info.setProperty("user", "root");
		info.setProperty("password", "abc123");
		//4. 获取连接
```

```
        Connection conn = driver.connect(url, info);
        System.out.println(conn);
    }

    @Test
    // 方式三：使用 DriverManager 替换 Driver
    public void test3() throws Exception {
        //1. 获取 Driver 实现类对象
        Class clazz = Class.forName("com.mysql.cj.jdbc.Driver");
        Driver driver = (Driver) clazz.newInstance();

        //2. 提供另外三个连接的基本信息
        String url = "jdbc:mysql://localhost:3306/mysql?characterEncoding=utf8&useSSL=false&serverTimezone=UTC&rewriteBatchedStatements=true";
        String user = "root";
        String password = "abc123";
        // 注册驱动
        DriverManager.registerDriver(driver);
        // 获取连接
        Connection conn = DriverManager.getConnection(url, user, password);
        System.out.println(conn);
    }

    @Test
    // 方式四，可以只是加载驱动，不用显式的注册驱动过了
    public void test4() throws Exception {
        //1. 提供三个连接的基本信息
        String url = "jdbc:mysql://localhost:3306/mysql?characterEncoding=utf8&useSSL=false&serverTimezone=UTC&rewriteBatchedStatements=true";
        String user = "root";
        String password = "abc123";

        //2. 加载 Driver
        Class.forName("com.mysql.cj.jdbc.Driver");
        // 相对于方式三，可以省略如下操作
//        Driver driver = (Driver) clazz.newInstance();
        // 注册驱动
```



```
//        DriverManager.registerDriver(driver);

        // 在 mysql 的 Driver 实现类中，声明了如下操作
//        static {
//            try {
//                DriverManager.registerDriver(new Driver());
//            } catch (SQLException var1) {
//                throw new RuntimeException("Can't register driver!");
//            }
//        }

        // 获取连接
        Connection conn = DriverManager.getConnection(url, user, password);
        System.out.println(conn);
    }

    @Test
    // 方式五：将数据库连接需要的 4 个基本信息声明在配置文件中，通过读取配置文件的方式，获取连接
    public void test5() throws Exception {
        // 读取配置文件中的 4 个基本信息
        InputStream is = ConnectionTest.class.getClassLoader().getResourceAsStream("propertiesTest");
        Properties pro = new Properties();
        pro.load(is);
        String url = pro.getProperty("url");
        String user = pro.getProperty("user");
        String password = pro.getProperty("password");
        String driverClass = pro.getProperty("driverClass");
        //2. 加载驱动
        Class.forName(driverClass);
        // 获取连接
        Connection conn = DriverManager.getConnection(url, user, password);
        System.out.println(conn);
    }
}
```

其中，配置文件声明在工程的 src 目录下：【jdbc.properties】

```
url=jdbc:mysql://localhost:3306/mysql?characterEncoding=utf8&use
SSL=false&serverTimezone=UTC&rewriteBatchedStatements=true
user=root
password=abc123
driverClass=com.mysql.cj.jdbc.Driver
```

说明：使用配置文件的方式保存配置信息，在代码中加载配置文件

使用配置文件的好处：

①实现了代码和数据的分离，如果需要修改配置信息，直接在配置文件中修改，不需要深入代码；

②如果修改了配置信息，省去重新编译的过程。

三 PreparedStatement

3.1 操作和访问数据库

使用 PreparedStatement 实现 CRUD 操作

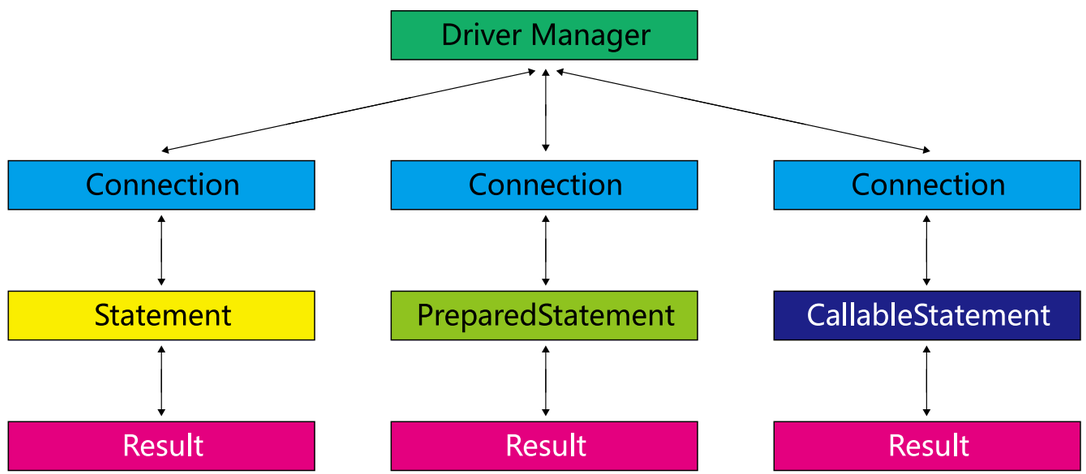
· 数据库连接被用于向数据库服务器发送命令和 SQL 语句，并接受数据库服务器返回的结果。其实一个数据库连接就是一个 Socket 连接。

· 在 java.sql 包中有 3 个接口分别定义了对数据库的调用的不同方式：

· Statement：用于执行静态 SQL 语句并返回它所生成结果的对象。

· PrepatedStatement：SQL 语句被预编译并存储在此对象中，可以使用此对象多次高效地执行该语句。

· CallableStatement：用于执行 SQL 存储过程。



3.2 使用 Statement 操作数据表的弊端

· 通过调用 Connection 对象的 createStatement() 方法创建该对象。该对象用于执行静态的 SQL 语句，并且返回执行结果。
· Statement 接口中定义了下列方法用于执行 SQL 语句：
int excuteUpdate(String sql)：执行更新操作 INSERT、UPDATE、DELETE
ResultSet executeQuery(String sql)：执行查询操作 SELECT
· 但是使用 Statement 操作数据表存在弊端：
    · 问题一：存在拼串操作，繁琐
    · 问题二：存在 SQL 注入问题
· SQL 注入是利用某些系统没有对用户输入的数据进行充分的检查，而在用户输入数据中注入非法的 SQL 语句段或命令 (如：SELECT user, password FROM user_table WHERE user='a' OR 1 = ' AND password = ' OR '1' ='1') ，从而利用系统的 SQL 引擎完成恶意行为的做法。
· 对于 Java 而言，要防范 SQL 注入，只要用 PreparedStatement(从 Statement 扩展而来) 取代 Statement 就可以了。
· 代码演示：

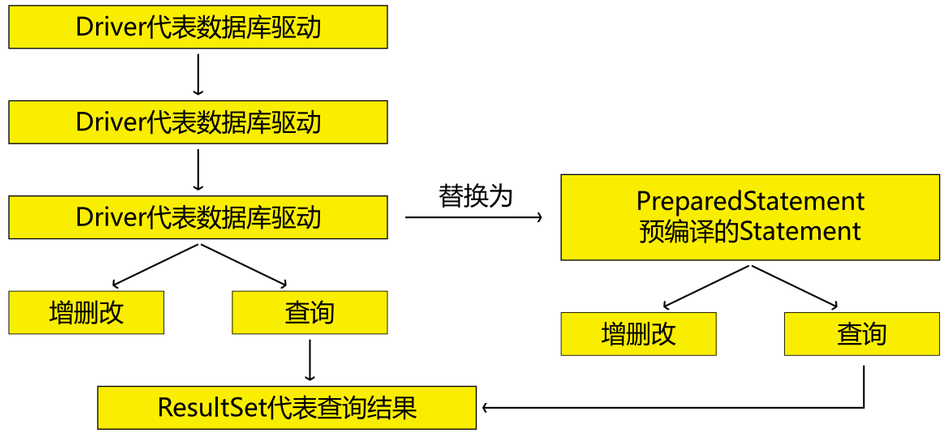
```
public class StatementTest {
    // 使用 Statement 的弊端：需要拼写 sql 语句，并且存在 SQL 注入的问题
    @Test
    public void testLogin() {
        Scanner scanner = new Scanner(System.in);
        System.out.print(" 请输入用户名 ");
        String user = scanner.next();
        System.out.print(" 请输入密码 ");
        String password = scanner.next();
        //SELECT user, password FROM user_table WHERE user = '1' or ' AND password = '=1 or '1' = '1';
        String sql = "SELECT user, password FROM user_table WHERE user = '" + user + "' AND password = '123456'";
        User returnUser = get(sql, User.class);
        if(returnUser != null){
            System.out.println(" 登录成功。");
        }else {
            System.out.println(" 用户名或密码错误。");
        }
    }
```



```
    // 使用 Statement 实现对数据表的查询操作
    public <T> T get(String sql, Class<T> clazz) {
        T t = null;
        Connection conn = null;
        Statement st = null;
        ResultSet rs = null;
        try {
            // 1. 加载配置文件
            InputStream is = StatementTest.class.getClassLoader().getResourceAsStream("jdbc.properties");
            Properties pros = new Properties();
            pros.load(is);
            // 2. 读取配置信息
            String user = pros.getProperty("user");
            String password = pros.getProperty("password");
            String url = pros.getProperty("url");
            String driverClass = pros.getProperty("driverClass");
            // 3. 加载驱动
            Class.forName(driverClass);
            // 4. 获取连接
            conn = DriverManager.getConnection(url, user, password);
            st = conn.createStatement();
            rs = st.executeQuery(sql);
            // 获取结果集的元数据
            ResultSetMetaData rsmd = rs.getMetaData();
            // 获取结果集的列数
            int columnCount = rsmd.getColumnCount();
            if (rs.next()) {
                t = clazz.newInstance();
                for (int i = 0; i < columnCount; i++) {
                    // //1. 获取列的名称
                    // String columnName = rsmd.getColumnName(i+1);
                    // 1. 获取列的别名
                    String columnName = rsmd.getColumnLabel(i + 1);
                    // 2. 根据列名获取对应数据表中的数据
                    Object columnVal = rs.getObject(columnName);
```

```
                        // 3. 将数据表中得到的数据，封装进对象
                        Field field = clazz.getDeclaredField
(columnName);
                        field.setAccessible(true);
                        field.set(t, columnVal);
                    }
                    return t;
                }
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                // 关闭资源
                if (rs != null) {
                    try {
                        rs.close();
                    } catch (SQLException e) {
                        e.printStackTrace();
                    }
                }
                if (st != null) {
                    try {
                        st.close();
                    } catch (SQLException e) {
                        e.printStackTrace();
                    }
                }

                if (conn != null) {
                    try {
                        conn.close();
                    } catch (SQLException e) {
                        e.printStackTrace();
                    }
                }
            }

            return null;
        }
    }
```


综上



3.3 PreparedStatement 的使用

3.3.1 PreparedStatement 介绍

· 可以通过调用 Connection 对象的 preparedStatement(String sql) 方法获取 PreparedStatement 对象；

· PreparedStatement 接口是 Statement 的子接口，它表示一条预编译过的 SQL 语句；

· PreparedStatement 对象所代表的 SQL 语句中的参数用问号 (?) 来表示，调用 PreparedStatement 对象的 setXxx() 方法来设置这些参数， setXxx() 方法有两个参数，第一个参数是要设置的 SQL 语句中的参数的索引 (从 1 开始)，第二个是设置的 SQL 语句中的参数的值

3.3.2 PreparedStatement vs Statement

· 代码的可读性和可维护性。

· PreparedStatement 能最大可能提高性能：

· DBServer 会对预编译语句提供性能优化。因为预编译语句有可能被重复调用，所以语句在被 DBServer 的编译器编译后的执行代码被缓存下来，那么下次调用时只要是相同的预编译语句就不需要编译，只要将参数直接传入编译过的语句执行代码中就会得到执行。

· 在 statement 语句中，即使是相同操作但因为数据内容不一样，所以整个语句本身不能匹配，没有缓存语句的意义。事实是没有数据库会对普通语句编译后的执行代码缓存。这样每执行一次都要对传入的语句编译一次。

· (语法检查，语义检查，翻译成二进制命令，缓存)

· PreparedStatement 可以防止 SQL 注入。

3.3.3 Java 与 SQL 对应数据类型转换表

Java类型	SQL类型
boolean	BIT
byte	TINYINT
short	SMALLINT
int	INTEGER
long	BIGINT
String	CHAR,VARCHAR,LONGVARCHAR
byte array	BINARY , VAR BINARY
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP

3.3.4 使用 PreparedStatement 实现增 / 删 / 改操作

· 可以通过调用 Connection 对象的 preparedStatement(String sql) 方法获取 PreparedStatement 对象；

· PreparedStatement 接口是 Statement 的子接口，它表示一条预编译过的 SQL 语句；

· PreparedStatement 对象所代表的 SQL 语句中的参数用问号 (?) 来表示，调用 PreparedStatement 对象的 setXxx() 方法来设置这些参数， setXxx() 方法有两个参数，第一个参数是要设置的 SQL 语句中的参数的索引 (从 1 开始)，第二个是设置的 SQL 语句中的参数的值


```
public class PraperedStatementUpdataTest {
    @Test
    public void test() {
        // 在 customer 表中添加一条记录
        Connection conn = null;
        PreparedStatement ps = null;
        try {
            // 1. 读取配置文件中的 4 个基本信息
            InputStream is = ClassLoader.getSystemClassLoader().
getResourceAsStream("jdbc.properties");
            Properties pro = new Properties();
            pro.load(is);
            String user = pro.getProperty("user");
            String url = pro.getProperty("url");
            String password = pro.getProperty("password");
            String driverClass = pro.getProperty("driverClass");
            // 2. 加载驱动
            Class.forName(driverClass);
            // 3. 获取连接
            conn = DriverManager.getConnection(url, user,
password);
            //System.out.println(conn);

            // 4. 预编译 sql 语句，返回 PreparedStatement 的实例
            String sql = "insert into customers(name, email, brith)
values(?, ?, ?) ";// 占位符
            ps = conn.prepareStatement(sql);
            // 5. 填充占位符
            ps.setString(1, " 哪吒 ");
            ps.setString(2, "nezha@gmail.com");
            SimpleDateFormat sdf = new SimpleDateFormat
("yyyy-MM-dd");
            java.util.Date date = sdf.parse("1000-01-01");
            ps.setDate(3, new Date(date.getTime()));
            // 6. 执行操作
            ps.execute();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            // 7. 资源关闭
```

```
            try {
                if (ps != null)
                    ps.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
            try {
                if (conn != null)
                    conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
}

@Test
public void testUpdata() {
        // 修改 customer 表中的一条记录
        Connection conn = null;
        PreparedStatement ps = null;
        try {
            // 1. 获取数据库的连接
            conn = JDBCUtils.getConnection();
            // 2. 预编译 sql 语句，返回 PreparedStatement 的实例
            String sql = "updata customer set name = ? where id
    = ?";
            ps = conn.prepareStatement(sql);
            // 3. 填充占位符
            ps.setObject(1, " 莫扎特 ");
            ps.setObject(2, 18);
            // 4. 执行
            ps.execute();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            // 5. 关闭资源
            JDBCUtils.closeResource(conn, ps);
        }
    }
}
```


通用的增删改操作

updata 模块

```
public class PraperedStatementUpdataTest {
    // 通用的增删改操作
    public void updata(String sql, Object... args) {// sql 中占位符的个
数与可变形参相同
        Connection conn = null;
        PreparedStatement ps = null;
        try {
            // 1. 获取数据库的连接
            conn = JDBCUtils.getConnection();
            // 2. 预编译 sql 语句，返回 PreparedStatement 的实例
            ps = conn.prepareStatement(sql);
            // 3. 填充占位符
            for (int i = 0; i < args.length; i++) {
                ps.setObject(i + 1, args[i]);// 小心参数声明错误
            }
            // 4. 执行
            ps.execute();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            // 5. 关闭资源
            JDBCUtils.closeResource(conn, ps);
        }
    }
}
```

JDBCUtils 模块

```
// 操作数据库的工具类
public class JDBCUtils {
	/** 获取数据库连接
	 * @throws Exception
	 */
	public static Connection getConnection() throws Exception {
		// 1. 读取配置文件中的 4 个基本信息
		InputStream is = ClassLoader.getSystemClassLoader().
getResourceAsStream("jdbc.properties");
		Properties pro = new Properties();
		pro.load(is);
		String user = pro.getProperty("user");
		String url = pro.getProperty("url");
		String password = pro.getProperty("password");
		String driverClass = pro.getProperty("driverClass");
		// 2. 加载驱动
		Class.forName(driverClass);
		// 3. 获取连接
		Connection conn = DriverManager.getConnection(url, user,
password);
		return conn;
	}
	public static void closeResource(Connection conn, Statement
ps) {
		// 关闭连接的操作
		try {
			if (ps != null)
				ps.close();
		} catch (SQLException e) {
			e.printStackTrace();
		}
		try {
			if (conn != null)
				conn.close();
		} catch (SQLException e) {
			e.printStackTrace();
		}
	}
}
```


3.4 ResultSet 与 ResultSetMetaData

3.4.1 ResultSet

· 查询需要调用 PreparedStatement 的 executeQuery() 方法，查询结果是一个 ResultSet 对象；

· ResultSet 对象以逻辑表格的形式封装了执行数据库操作的结果集，ResultSet 接口由数据库厂商提供实现；

· ResultSet 返回的实际上就是一张数据表。有一个指针指向数据表的第一条记录的前面。

· ResultSet 对象维护了一个指向当前数据行的游标，初始的时候，游标在第一行之前，可以通过 ResultSet 对象的 next() 方法移动到下一行。调用 next() 方法检测下一行是否有效。若有效，该方法返回 true，且指针下移。相当于 Iterator 对象的 hasNext() 和 next() 方法的结合体。

· 当指针指向一行时，可以通过调用 getXxx(int index) 或 getXxx(int columnName) 获取每一列的值。

  · 例如：getInt(1), getString("name")

  · 注意：Java 与数据库交互涉及到的相关 Java API 中的索引都从 1 开始。

· ResultSet 接口的常用方法：

· boolean next()

· getString()

· …

3.4.2 ResultSetMetaData

· 可用于获取关于 ResultSet 对象中列的类型和属性信息的对象；

· ResultSetMetaData meta = rs.getMetaData()；

  · getColumnName(int column)：获取指定列的名称；

  · getColumnLabel(int column)：获取指定列的别名；

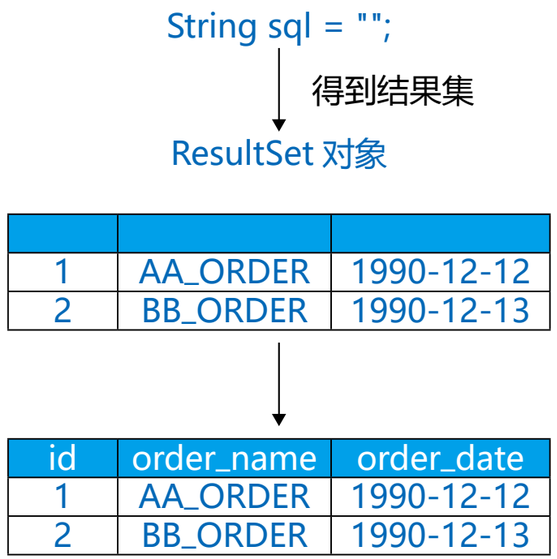
  · getColumnCount()：返回当前 ResultSet 对象中的列数。

  · getColumnTypeName(int column)：检索指定列的数据库特定的类型名称。

  · getColumnDisplaySize(int column)：指示指定列的最大标准宽度，以字符为单位。

  · isNullable(int column)：指示指定列中的值是否可以为 null。

  · isAutoIncrement(int column)：指示是否自动为指定列进行编号，这样这些列仍然是只读的。



问题 1：得到结果集后 , 如何知道该结果集中有哪些列 ？ 列名是什么？

需要使用一个描述 ResultSet 的对象， 即 ResultSetMetaData

问题 2：关于 ResultSetMetaData

1. 如何获取 ResultSetMetaData： 调用 ResultSet 的 getMetaData() 方法即可

2. 获取 ResultSet 中有多少列：调用 ResultSetMetaData 的 getColumnCount() 方法

3. 获取 ResultSet 每一列的列的别名是什么：调用 ResultSetMetaData 的 getColumnLabel() 方法

3.5 资源的释放

· 释放 ResultSet, Statement, Connection。

· 数据库连接（Connection）是非常稀有的资源，用完后必须马上释放，如果 Connection 不能及时正确的关闭将导致系统宕机。Connection 的使用原则是尽量晚创建，尽量早的释放。

· 可以在 finally 中关闭，保证及时其他代码出现异常，资源也一定能被关闭。

3.6 针对 Customer 表的查询操作

CustomerQuery 模块

```
// 针对于 Customer 表的查询操作
public class CustomerForQuery {

    @Test
    public void testQuery() {
        Connection conn = null;
        PreparedStatement ps = null;
        ResultSet resultSet = null;
        try {
            conn = JDBCUtils.getConnection();
            String sql = "select id , name , email, brith from customers where id";
            ps = conn.prepareStatement(sql);
            ps.setObject(1, 1);
            // 执行并返回结果集
            resultSet = ps.executeQuery();
            // 处理结果集
            if (resultSet.next()) {// 判断结果集的下一条是否有数据，如果有数据返回 true，如果返回 false，指针不会下移
                // 获取当前这条数据的各个字段值
                int id = resultSet.getInt(1);
                String name = resultSet.getString(2);
                String email = resultSet.getString(3);
                Date birth = resultSet.getDate(4);

                // 方式一
                // System.out.println("id =" + id + ",name =" + name + ",email" + email +
                // ",birth" + birth);
                // 方式二
                // Object[] data = new Object[]{id, name, email, birth};
                // 方式三：将数据封装成一个对象（推荐）
                Customer customer = new Customer(id, name, email, birth);
```

```
                System.out.println(customer);
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            // 关闭资源
            JDBCUtils.closeResource(conn, ps, resultSet);
        }
    }

    @Test
    public void testQueryForCustomer() {
        String sql = "select id , name , email, brith from customers
where id = ?";
        Customer customer = queryForCustomer(sql, 13);
        System.out.println(customer);

        sql = "select id , name , email, brith from customers where
id = ?";
        Customer customer1 = queryForCustomer(sql, " 周杰伦 ");
        System.out.println(customer1);
}

public Customer queryForCustomer(String sql, Object... args) {
    Connection conn = null;
    PreparedStatement ps = null;
    ResultSet rs = null;
    try {
        conn = JDBCUtils.getConnection();
        ps = conn.prepareStatement(sql);
        for (int i = 0; i < args.length; i++) {
            ps.setObject(i + 1, args[i]);
        }
        rs = ps.executeQuery();
        // 通过结果集的元数据：ResultSetMetaData
        ResultSetMetaData rsmd = rs.getMetaData();
        // 通过 ResultSetMetaData 获取结果集中的列
        int columnCount = rsmd.getColumnCount();
        if (rs.next()) {
            Customer cust = new Customer();
```



```
            for (int i = 0; i < columnCount; i++) {
                Object columnValue = rs.getObject(i + 1);
                // 获取每个列的列名
                String columnName = rsmd.
getColumnName(i + 1);
                // 给 cust 对象指定的某个属性，赋值为 value
                Field field = Customer.class.
getDeclaredField(columnName);
                field.setAccessible(true);
                field.set(cust, columnValue);
            }
            return cust;
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        JDBCUtils.closeResource(conn, ps, rs);
    }
    return null;
    }
}
```

Cusomer 模块

```
/**
 * ORM 编程思想（Object relationnl mapping)
 * 一个表对应一个 Java 类 表中的一条记录对应 java 类中的一个对象
 * 表中的一个字段对应 java 类中的一个属性
 */
public class Customer {
    private int id;
    private String name;
    private String email;
    private Date birth;

    public Customer() {
    }

    public Customer(int id, String name, String email, Date birth) {
        this.id = id;
        this.name = name;
```

```
        this.email = email;
        this.birth = birth;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public Date getBirth() {
        return birth;
    }
    public void setBirth(Date birth) {
        this.birth = birth;
    }

    @Override
    public String toString() {
        return "Customer{" + "id=" + id + ", name='" + name + '\''
+ ", email='" + email + '\'' + ", birth=" + birth + '}';
    }
}
```


3.7 针对 Order 表的查询操作

OrderQuery 模块

```
public class OrderForQuery {
    @Test
    public void test() {
        Connection conn = null;
        PreparedStatement ps = null;
        ResultSet rs = null;
        try {
            conn = JDBCUtils.getConnection();
            String sql = "select order_Id, order_name, order_date
from `order` where order_id = ?";
            ps = conn.prepareStatement(sql);
            ps.setObject(1, 1);

            rs = ps.executeQuery();
            if (rs.next()) {
                int id = (int) rs.getObject(1);
                String name = (String) rs.getObject(2);
                Date date = (Date) rs.getObject(3);

                Order order = new Order(id, name, date);
                System.out.println(order);
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            JDBCUtils.closeResource(conn, ps, rs);
        }
    }

    @Test
    public void testOrderFroQuery() {
        String sql = "select order_id orderId, order_name
orderName, order_date orderDate from `order` where order_id = ?";
        Order order = orderFroQuery(sql, 1);
        System.out.println(order);
    }
```

```java
// 针对于 Order 表的通用查询操作
public Order orderFroQuery(String sql, Object... args) {
        Connection conn = null;
        PreparedStatement ps = null;
        ResultSet rs = null;
        try {
                conn = JDBCUtils.getConnection();
                ps = conn.prepareStatement(sql);
                for (int i = 0; i < args.length; i++) {
                        ps.setObject(i + 1, args[i]);
                }

                // 执行，获取结果集
                rs = ps.executeQuery();
                ResultSetMetaData rsmd = rs.getMetaData();
                // 获取列数
                int columnCount = rsmd.getColumnCount();
                if (rs.next()) {
                        Order order = new Order();
                        for (int i = 0; i < columnCount; i++) {
                                // 获取每个列的列值，通过 ResultSet
                                Object columnValue = rs.getObject(i + 1);
                                // 通过 ResultSetMetaData
                                // 获取每个列的列名，getCatalogName() --
不推荐使用
                                // 获取每个列的别名，getColumnLabel()
                                // String catalogName = rsmd.
getCatalogName(i + 1);
                                String catalogLabel = rsmd.
getColumnLabel(i + 1);

                                // 通过反射，将对象指定名 catalogName 的属
性赋值为指定的值 columnValue
                                Field field = Order.class.
getDeclaredField(catalogLabel);
                                field.setAccessible(true);
                                field.set(order, columnValue);
                        }
                        return null;
                }
```



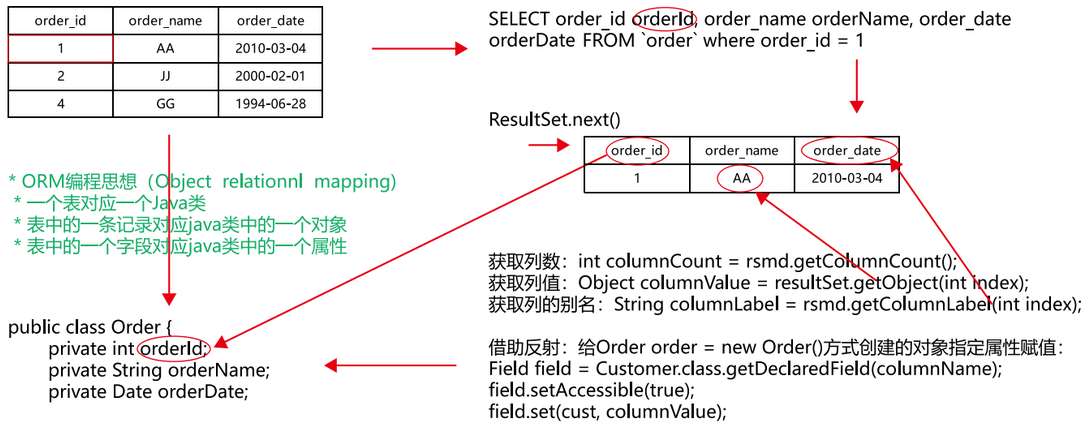
```java
                } catch (Exception e) {
                        e.printStackTrace();
                } finally {
                        JDBCUtils.closeResource(conn, ps, rs);
                }
                return null;
        }
}
```

OrderQuery 模块

```java
public class Order {
        private int orderId;
        private String orderName;
        private Date orderDate;

        public Order() {
        }

        public Order(int orderId, String orderName, Date orderDate) {
                this.orderId = orderId;
                this.orderName = orderName;
                this.orderDate = orderDate;
        }

        public int getOrderId() {
                return orderId;
        }

        public void setOrderId(int orderId) {
                this.orderId = orderId;
        }

        public String getOrderName() {
                return orderName;
        }

        public void setOrderName(String orderName) {
                this.orderName = orderName;
        }
```

```
    public Date getOrderDate() {
        return orderDate;
    }

    public void setOrderDate(Date orderDate) {
        this.orderDate = orderDate;
    }

    @Override
    public String toString() {
        return "Order{" +
                "orderId=" + orderId +
                ", orderName='" + orderName + '\'' +
                ", orderDate=" + orderDate +
                '}';
    }
}
```

图解查询操作的流程



3.8 针对针对不同表的通用查询操作

```
public class preparedStatementQuery {

    @Test
    public void testGetForList() {
        String sql = "select id, name, email from customers where id < ?";
        List<Customer> list = getForList(Customer.class, sql, 12);
        list.forEach(System.out::println);

        String sql2 = "select order_id orderId, order_name orderName, from `order` where order_id < ?";
        List<Order> orderList = getForList(Order.class, sql2, 5);
        orderList.forEach(System.out::println);
    }

    // 针对不同表的通用查询操作，返回表中的多条记录
    public <T> List<T> getForList(Class<T> clazz, String sql, Object... args) {
        Connection conn = null;
        PreparedStatement ps = null;
        ResultSet rs = null;
        try {
            conn = JDBCUtils.getConnection();
            ps = conn.prepareStatement(sql);
            for (int i = 0; i < args.length; i++) {
                ps.setObject(i + 1, args[i]);
            }
            rs = ps.executeQuery();
            // 通过结果集的元数据：ResultSetMetaData
            ResultSetMetaData rsmd = rs.getMetaData();
            // 通过 ResultSetMetaData 获取结果集中的列
            int columnCount = rsmd.getColumnCount();
            // 创建集合对象
            ArrayList<T> list = new ArrayList<>();
            while (rs.next()) {
                T t = clazz.newInstance();
                // 处理结果集一行数据中的每一个列，给 t 对象指定的属性赋值
```

```
                    for (int i = 0; i < columnCount; i++) {
                        Object columnValue = rs.getObject(i + 1);
                        // 获取每个列的列名
                        String columnName = rsmd.
getColumnName(i + 1);
                        // 给 cust 对象指定的某个属性，赋值为 value
                        Field field = clazz.
getDeclaredField(columnName);
                        field.setAccessible(true);
                        field.set(t, columnValue);
                    }
                    list.add(t);
                }
                return list;
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                JDBCUtils.closeResource(conn, ps, rs);
            }
            return null;
        }

        @Test
        public void testGetInstance() {
            String sql = "select id, name, email from customers where
id = ?";
            Customer customer = getInstance(Customer.class, sql, 12);
            System.out.println(customer);

            String sql2 = "select order_id orderId, order_name
orderName, from `order` where order_id = ?";
            Order order = getInstance(Order.class, sql2, 1);
            System.out.println(sql2);
        }

        // 针对不同表的通用查询操作，返回表中的一条记录
        public <T> T getInstance(Class<T> clazz, String sql, Object...
args) {
            Connection conn = null;
            PreparedStatement ps = null;
```



```
            ResultSet rs = null;
            try {
                conn = JDBCUtils.getConnection();
                ps = conn.prepareStatement(sql);
                for (int i = 0; i < args.length; i++) {
                    ps.setObject(i + 1, args[i]);
                }
                rs = ps.executeQuery();
                // 通过结果集的元数据：ResultSetMetaData
                ResultSetMetaData rsmd = rs.getMetaData();
                // 通过 ResultSetMetaData 获取结果集中的列
                int columnCount = rsmd.getColumnCount();
                if (rs.next()) {
                    T t = clazz.newInstance();
                    for (int i = 0; i < columnCount; i++) {
                        Object columnValue = rs.getObject(i + 1);
                        // 获取每个列的列名
                        String columnName = rsmd.
getColumnName(i + 1);
                        // 给 cust 对象指定的某个属性，赋值为 value
                        Field field = clazz.
getDeclaredField(columnName);
                        field.setAccessible(true);
                        field.set(t, columnValue);
                    }
                    return t;
                }
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                JDBCUtils.closeResource(conn, ps, rs);
            }
            return null;
        }
    }
```

演示使用 PreparedStatement 代替 Statement，解决 SQL 注入问题

除了解决 Statement 的拼串、sql 问题之外，PreparedStatement 还有哪些好处呢？

1.PreparedStatement 操作 Blob 的数据，而 Statement 做不到。

2.PneparedStatement 可以实现更高效的批量操作。

```
public class PreparedStatementTest {
    @Test
    public void testLogin() {
        Scanner scanner = new Scanner(System.in);
        System.out.print(" 请输入用户名 ");
        String user = scanner.next();
        System.out.print(" 请输入密码 ");
        String password = scanner.next();
        // SELECT user, password FROM user_table WHERE user =
'1' or ' AND password = '=1 or '1' = '1';
        String sql = "SELECT user, password FROM user_table
WHERE user = ? AND password = ?";
        User returnUser = getInstance(User.class, sql, user,
password);
        if (returnUser != null) {
            System.out.println(" 登录成功。");
        } else {
            System.out.println(" 用户名或密码错误。");
        }
    }

    public <T> T getInstance(Class<T> clazz, String sql, Object...
args) {
        Connection conn = null;
        PreparedStatement ps = null;
        ResultSet rs = null;
        try {
            conn = JDBCUtils.getConnection();
            ps = conn.prepareStatement(sql);
            for (int i = 0; i < args.length; i++) {
                ps.setObject(i + 1, args[i]);
            }
            rs = ps.executeQuery();
            // 通过结果集的元数据：ResultSetMetaData
            ResultSetMetaData rsmd = rs.getMetaData();
```



```
            // 通过 ResultSetMetaData 获取结果集中的列
            int columnCount = rsmd.getColumnCount();
            if (rs.next()) {
                T t = clazz.newInstance();
                for (int i = 0; i < columnCount; i++) {
                    Object columnValue = rs.getObject(i + 1);
                    // 获取每个列的列名
                    String columnName = rsmd.
getColumnName(i + 1);
                    // 给 cust 对象指定的某个属性，赋值为 value
                    Field field = clazz.
getDeclaredField(columnName);
                    field.setAccessible(true);
                    field.set(t, columnValue);
                }
                return t;
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            JDBCUtils.closeResource(conn, ps, rs);
        }
        return null;
    }
}
```

3.9　JDBC API 小结

· 两种思想：
　　· 面向接口编程的思想；
　　· ORM 思想 (object relational mapping)；
　　　　· 一个数据表对应一个 java 类；
　　　　· 表中的一条记录对应 java 类的一个对象；
　　　　· 表中的一个字段对应 java 类的一个属性；
　　· sql 是需要结合列名和表的属性名来写。注意起别名。

· 两种技术：
　　· JDBC 结果集的元数据：ResultSetMetaData；
　　　　· 获取列数：getColumnCount()；
　　　　· 获取列的别名：getColumnLabel()；
　　· 通过反射，创建指定类的对象，获取指定的属性并赋值。

四　操作 BLOB 类型字段

4.1　MySQL BLOB 类型

· MySQL 中，BLOB 是一个二进制大型对象，是一个可以存储大量数据的容器，它能容纳不同大小的数据。

· 插入 BLOB 类型的数据必须使用 PreparedStatement，因为 BLOB 类型的数据无法使用字符串拼接写的。

· MySQL 的四种 BLOB 类型 (除了在存储的最大信息量上不同外，他们是等同的)。

· 实际使用中根据需要存入的数据大小定义不同的 BLOB 类型。

· 需要注意的是：如果存储的文件过大，数据库的性能会下降。

· 如果在指定了相关的 Blob 类型以后，还报错：xxx too large，那么在 mysql 的安装目录下，找 my.ini 文件加上如下的配置参数： max_allowed_packet=16M。同时注意：修改了 my.ini 文件之后，需要重新启动 mysql 服务。

类型	大小 (单位：字节)
TinyBlob	最大 255
Blob	最大 65K
MediumBlob	最大 16M
LongBlob	最大 4G

4.2 使用 PraperedStatement 操作 Blob 类型的数据

```
public class BlobTest {
    // 向数据表 customers 中插入 Blob 类型的字段
    @Test
    public void testInstance() throws Exception {
        Connection conn = JDBCUtils.getConnection();
        String sql = "inster into customers(name, email, birth,
photo)values(?, ?, ?, ?)";
        PreparedStatement ps = conn.prepareStatement(sql);
        ps.setObject(1, " 张宇豪 ");
        ps.setObject(2, "zhang@qq.com");
        ps.setObject(3, "1992-09-08");
        FileInputStream is = new FileInputStream(new
File("zhangyuhao.jpg"));
        ps.setBlob(4, is);

        ps.execute();

        JDBCUtils.closeResource(conn, ps);
    }

    @Test
    // 查询数据表 customers 中 Blob 类型的字段
    public void testQuery() {
        Connection conn = null;
        PreparedStatement ps = null;
        InputStream is = null;
        FileOutputStream fos = null;
        ResultSet rs = null;
        try {
            conn = JDBCUtils.getConnection();
            String sql = "select id, name, email, birth, photo from
customers where id = ?";
            ps = conn.prepareStatement(sql);
            ps.setInt(1, 21);
            is = null;
            fos = null;
```



```
            rs = ps.executeQuery();
            if (rs.next()) {
                int id = rs.getInt("id");
                String name = rs.getString("name");
                String email = rs.getString("email");
                Date birth = rs.getDate("birth");
                Customer cust = new Customer(id, name, email,
birth);

                System.out.println(cust);
                // 将 Blob 类型的字段下载下来，以文件的方式保存
在本地
                Blob photo = rs.getBlob("photo");
                is = photo.getBinaryStream();
                fos = new FileOutputStream("zhangyuhao.jpg");
                byte[] buffer = new byte[1024];
                int len;
                while ((len = is.read(buffer)) != -1) {
                    fos.write(buffer, 0, len);
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                if (is != null)
                    is.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
            try {
                if (fos != null)
                    fos.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
            JDBCUtils.closeResource(conn, ps, rs);
        }
    }
}
```

五　批量插入

5.1　批量执行 SQL 语句

当需要成批插入或者更新记录时，可以采用 Java 的批量更新机制，这一机制允许多条语句一次性提交给数据库批量处理。通常情况下比单独提交处理更有效率；

JDBC 的批量处理语句包括下面三个方法：

- addBatch(String)：添加需要批量处理的 SQL 语句或是参数；
- executeBatch()：执行批量处理语句；
- clearBatch(): 清空缓存的数据

通常我们会遇到两种批量执行 SQL 语句的情况：

- 多条 SQL 语句的批量处理；
- 一个 SQL 语句的批量传参；

5.2　高效的批量插入

使用 PreparedStatement 实现批量数据的操作

update、delete 本身就具有批量操作的效果。
此时的批量操作，主要指的是批量插入。使用 PreparedStatement 如何实现更高效的批量插入？

题目：向 goods 表中插入 20000 条数据

```
CREATE TABLE goods(
	id INT PRIMARY KEY AUTO_INCREMENT,
	NAME VARCHAR(25)
);
```

方式一：使用 Statement

```
Connection conn = JDBCUtils.getConnection();
Statement st = conn.createStatement();
for(int i = 1;i <=20000;i++){
	String sql = "insert into goods(name)values('name_" + i + "')";
	st.execute(sql);
}
```



```
public class InsterTest {
	@Test
	// 批量插入的方式二：使用 PreparedStatement
	public void test() {
		Connection conn = null;
		PreparedStatement ps = null;
		try {
			long start = System.currentTimeMillis();
			conn = JDBCUtils.getConnection();
			String sql = "inster into goods(name) Values(?)";
			ps = conn.prepareStatement(sql);
			for (int i = 0; i < 20000; i++) {
				ps.setObject(1, "name_" + i);
				ps.execute();
			}
			long end = System.currentTimeMillis();
			System.out.println(" 花费的时间为 " + (end - start));
		} catch (Exception e) {
			e.printStackTrace();
		} finally {
			JDBCUtils.closeResource(conn, ps);
		}
	}

	@Test
	// 批量插入的方式三：
	// 修改 1：使用 addBatch() / executeBatch() / clearBatch()
	// 修改 2：mysql 服务器默认是关闭批处理的，我们需要通过一个参数，让 mysql 开启批处理的支持。
	//rewriteBatchedStatements=true 写在配置文件的 url 后面
	// 修改 3：使用更新的 mysql 驱动：mysql-connector-java-5.1.37-bin.jar
	public void test2() {
		Connection conn = null;
		PreparedStatement ps = null;
		try {
			long start = System.currentTimeMillis();
			conn = JDBCUtils.getConnection();
			String sql = "inster into goods(name) Values(?)";
			ps = conn.prepareStatement(sql);
```

```
            for (int i = 0; i < 20000; i++) {
                ps.setObject(1, "name_" + i);
                // 1. “攒” sql
                ps.addBatch();
                if (i % 500 == 0) {
                    // 2. 执行 Batch
                    ps.executeBatch();
                    // 3. 清空 Batch
                    ps.clearBatch();
                }
            }
            long end = System.currentTimeMillis();
            System.out.println(" 花费的时间为 " + (end - start));
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            JDBCUtils.closeResource(conn, ps);
        }
    }
```



```
                    // 3. 清空 Batch
                    ps.clearBatch();
                }
            }
            // 提交数据
            conn.commit();
            long end = System.currentTimeMillis();
            System.out.println(" 花费的时间为 " + (end - start));
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            JDBCUtils.closeResource(conn, ps);
        }
    }
}
```

```
    @Test
    // 批量插入的方式四：在三的基础上操作使用 Connection 的
setAutoCommit(false) / commit()
    public void test3() {
        Connection conn = null;
        PreparedStatement ps = null;
        try {
            long start = System.currentTimeMillis();
            conn = JDBCUtils.getConnection();
            // 设置不允许自动提交数据
            conn.setAutoCommit(false);
            String sql = "inster into goods(name) Values(?)";
            ps = conn.prepareStatement(sql);
            for (int i = 0; i < 20000; i++) {
                ps.setObject(1, "name_" + i);
                // 1. “攒” sql
                ps.addBatch();
                if (i % 500 == 0) {
                    // 2. 执行 Batch
                    ps.executeBatch();
```

六　数据库事务

6.1　数据库事务介绍

· 事务：一组逻辑操作单元，使数据从一种状态变换到另一种状态。
　　· 一组逻辑操作单元：一个或多个 DML 操作。
· 事务处理（事务操作）：保证所有事务都作为一个工作单元来执行，即使出现了故障，都不能改变这种执行方式。当在一个事务中执行多个操作时，要么所有的事务都被提交 (commit)，那么这些修改就永久地保存下来；要么数据库管理系统将放弃所作的所有修改，整个事务回滚 (rollback) 到最初状态。
· 数据一旦提交，就不可回滚
· 哪些操作会导致数据的自动提交？
　　· DDL 操作一旦执行，都会自动提交。
　　　　· set autocommit=false 对 DDL 操作失效。
　　· DML 默认情况下，一旦执行，就会自动提交。
　　　　· 我们可以通过 set autocommit = false 的方式取消 DML 操作的自动提交。
　　· 默认在关闭连接时，会自动的提交数据
· 为确保数据库中数据的一致性，数据的操纵应当是离散的成组的逻辑单元：当它全部完成时，数据的一致性可以保持，而当这个单元中的一部分操作失败，整个事务应全部视为错误，所有从起始点以后的操作应全部回退到开始状态。

6.2　JDBC 事务处理

· 数据一旦提交，就不可回滚。
· 数据什么时候意味着提交？
　　· 当一个连接对象被创建时，默认情况下是自动提交事务：每次执行一个 SQL 语句时，如果执行成功，就会向数据库自动提交，而不能回滚。
　　· 关闭数据库连接，数据就会自动的提交。如果多个操作，每个操作使用的是自己单独的连接，则无法保证事务。即同一个事务的多个操作必须在同一个连接下。
· JDBC 程序中为了让多个 SQL 语句作为一个事务执行：
　　· 调用 Connection 对象的 setAutoCommit(false); 以取消自动提交事务；
　　· 在所有的 SQL 语句都成功执行后，调用 commit()； 方法提交事务；
　　· 在出现异常时，调用 rollback(); 方法回滚事务
· 若此时 Connection 没有被关闭，还可能被重复使用，则需要恢复其自动提交状态 setAutoCommit(true)。尤其是在使用数据库连接池技术时，执行 close() 方法前，建议恢复自动提交状态。

```
public class TransactionTest {

    @Test
    // 针对于数据表 user_table 来说
    // 用户 AA 给 BB 用户转账 100
// update user_table set balance = balance - 100 where user = "AA";
// update user_table set balance = balance + 100 where user = "BB";
    public void testUpdate() {
        // ********************************************* 未考虑数据库事物的版本 *********************************
        String sql = "update user_table set balance = balance - 100 where user =?";
        updata(sql, "AA");
        // 模拟网络异常
        System.out.println(10 / 0);

        String sql2 = "update user_table set balance = balance + 100 where user =?";
        updata(sql2, "BB");
        System.out.println(" 转账成功 ");

    }
```

```
    // 通用的增删改操作 --- version 1.0 版本
    public int updata(String sql, Object... args) {
// sql 中占位符的个数与可变形参相同
        Connection conn = null;
        PreparedStatement ps = null;
        try {
            // 1. 获取数据库的连接
            conn = JDBCUtils.getConnection();
            // 2. 预编译 sql 语句，返回 PreparedStatement 的实例
            ps = conn.prepareStatement(sql);
            // 3. 填充占位符
            for (int i = 0; i < args.length; i++) {
                ps.setObject(i + 1, args[i]);// 小心参数声明错误
            }
            // 4. 执行
            return ps.executeUpdate();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            // 5. 关闭资源
            JDBCUtils.closeResource(conn, ps);
        }
        return 0;
    }
    // ********************************************* 考虑数据库事物后的
转账操作 *********************************
    @Test
    public void testUpdateWithTx(){
        Connection conn = null;
        try {
            conn = Util.JDBCUtils.getConnection();
            //1. 取消数据的自动提交
            conn.setAutoCommit(false);
            System.out.println(conn.setAutoCommit());;
             String sql = "update user_table set balance = balance
- 100 where user =?";
            updata(conn, sql, "AA");

            // 模拟网络异常
//            System.out.println(10/0);
```



```
            String sql2 = "update user_table set balance = balance
+ 100 where user =?";
            updata(conn, sql2, "BB");
            System.out.println(" 转账成功 ");
            //2. 提交数据
            conn.commit();
        } catch (Exception e) {
            e.printStackTrace();
            //3. 回滚数据
            try {
                conn.rollback();
            } catch (SQLException ex) {
                ex.printStackTrace();
            }
        } finally {
            Util.JDBCUtils.closeResource(conn, null);
        }
    }
    // 通用的增删改操作 --- version 2.0 版本
    public int updata(Connection conn, String sql, Object... args) {
// sql 中占位符的个数与可变形参相同
        PreparedStatement ps = null;
        try {
            // 1. 预编译 sql 语句，返回 PreparedStatement 的实例
            ps = conn.prepareStatement(sql);
            // 2. 填充占位符
            for (int i = 0; i < args.length; i++) {
                ps.setObject(i + 1, args[i]);// 小心参数声明错误
            }
            // 3. 执行
            return ps.executeUpdate();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            // 4. 关闭资源
            JDBCUtils.closeResource(null, ps);
        }
        return 0;
    }
}
```

6.3 事务的 ACID 属性

· 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）：事务必须使数据库从一个一致性状态变换到另外一个一致性状态。

· 隔离性（Isolation）：事务的隔离性是指一个事务的执行不能被其他事务干扰，即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。

· 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，接下来的其他操作和数据库故障不应该对其有任何影响。

6.3.1 数据库的并发问题

· 对于同时运行的多个事务，当这些事务访问数据库中相同的数据时，如果没有采取必要的隔离机制，就会导致各种并发问题：

· 脏读：对于两个事务 T1，T2，T1 读取了已经被 T2 更新但还没有被提交的字段。之后，若 T2 回滚，T1 读取的内容就是临时且无效的。

· 不可重复读：对于两个事务 T1，T2，T1 读取了一个字段，然后 T2 更新了该字段。之后，T1 再次读取同一个字，值就不同了。

· 幻读：对于两个事务 T1，T2，T1 从一个表中读取了一个字段，然后 T2 在该表中插入了一些新的行。之后，如 果 T1 再次读取同一个表，就会多出几行。

· 数据库事务的隔离性：数据库系统必须具有隔离并发运行各个事务的能力，使它们不会相互影响，避免各种并发问题。

· 一个事务与其他事务隔离的程度称为隔离级别。数据库规定了多种事务隔离级别，不同隔离级别对应不同的干扰程度，隔离级别越高，数据一致性就越好，但并发性越弱。

6.3.2 四种隔离级别

· 数据库提供的 4 种事务隔离级别：

隔离级别	描述
READ UNCOMMITTED（读未提交数据）	允许事务读取未被其他事物提交的变更 . 脏读，不可重复读和幻读的问题都会出现
READ COMMITED（读已提交数据）	只允许事务读取已经被其它事务提交的变更 . 可以避免脏读，但不可重复读和幻读问仍然可能出现
REPEATABLE READ（可重复读）	确保事务可以多次从一个字段中读取相同的值，在这个事务持续期间，禁止其他事物对这个字段进行更新可以避免脏读和不可重复读，但幻读的问题仍然存在
SERIALIZABLE（串行化）	确保事务可以从一个表中读取相同的行在这个事务持续期间，禁止其他事务对该表执行插入，更新和删除操作，所有并发问题都可以避免，但性能十分低下

· Oracle 支持的 2 种事务隔离级别：READ COMMITED, SERIALIZABLE。Oracle 默认的事务隔离级别为： READ COMMITED 。

· Mysql 支持 4 种事务隔离级别。Mysql 默认的事务隔离级别为：REPEATABLE READ。

6.3.3 在 MySql 中设置隔离级别

· 每启动一个 mysql 程序 , 就会获得一个单独的数据库连接 . 每个数据库连接都有一个全局变量 @@tx_isolation，表示当前的事务隔离级别。

· 查看当前的隔离级别：

```
SELECT @@tx_isolation;
```

· 设置当前 mySQL 连接的隔离级别：

```
set transaction isolation level read committed;
```

· 设置数据库系统的全局的隔离级别：

```
set global transaction isolation level read committed;
```

· 补充操作：

· 创建 mysql 数据库用户：

```
create user tom identified by 'abc123';
```

· 授予权限

```
# 授予通过网络方式登录的 tom 用户，对所有库所有表的全部权限，密码设为 abc123.
grant all privileges on *.* to tom@'%' identified by 'abc123';
# 给 tom 用户使用本地命令行方式，授予 atguigudb 这个库下的所有表的插删改查的权限。
grant select,insert,delete,update on atguigudb.* to tom@localhost identified by 'abc123';
```

Java 代码演示并设置数据库的隔离级别

```
public class TransactionTest {
    @Test
    public void testTransactionSelect() throws Exception {
        Connection conn = Util.JDBCUtils.getConnection();
        // 获取当前连接的隔离级别
        System.out.println(conn.getTransactionIsolation());
        // 设置数据库的隔离级别
        conn.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
        // 取消自动提交数据
        conn.setAutoCommit(false);
        String sql = "select user, password, balance from user_table where user = ?";
        User user = getInstance(conn, User.class, sql, "CC");
        System.out.println(user);
    }

    @Test
    public void testTransactionUpdate() throws Exception {
        Connection conn = Util.JDBCUtils.getConnection();
        // 取消自动提交数据
        conn.setAutoCommit(false);
        String sql = "update user_table set balance = ? where user = ?";
        updata(conn, sql, 5000);
        Thread.sleep(15000);
        System.out.println(" 修改结束。");
    }
```



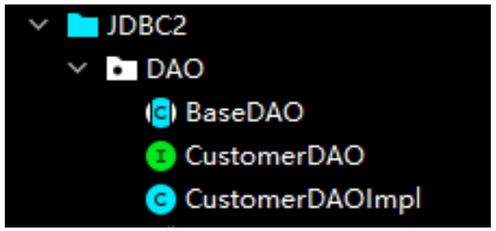
```
    // 通用的查询操作，用于返回数据表中的一条记录（version -- 2.0：考虑上事务）
    public <T> T getInstance(Connection conn, Class<T> clazz, String sql, Object... args) {
        PreparedStatement ps = null;
        ResultSet rs = null;
        try {
            ps = conn.prepareStatement(sql);
            for (int i = 0; i < args.length; i++) {
                ps.setObject(i + 1, args[i]);
            }
            rs = ps.executeQuery();
            // 通过结果集的元数据：ResultSetMetaData
            ResultSetMetaData rsmd = rs.getMetaData();
            // 通过 ResultSetMetaData 获取结果集中的列
            int columnCount = rsmd.getColumnCount();
            if (rs.next()) {
                T t = clazz.newInstance();
                for (int i = 0; i < columnCount; i++) {
                    Object columnValue = rs.getObject(i + 1);
                    // 获取每个列的列名
                    String columnName = rsmd.getColumnName(i + 1);
                    // 给 cust 对象指定的某个属性，赋值为 value
                    Field field = clazz.getDeclaredField(columnName);
                    field.setAccessible(true);
                    field.set(t, columnValue);
                }
                return t;
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            JDBCUtils.closeResource(null, ps, rs);
        }
        return null;
    }
}
```

七　DAO 及相关实现类

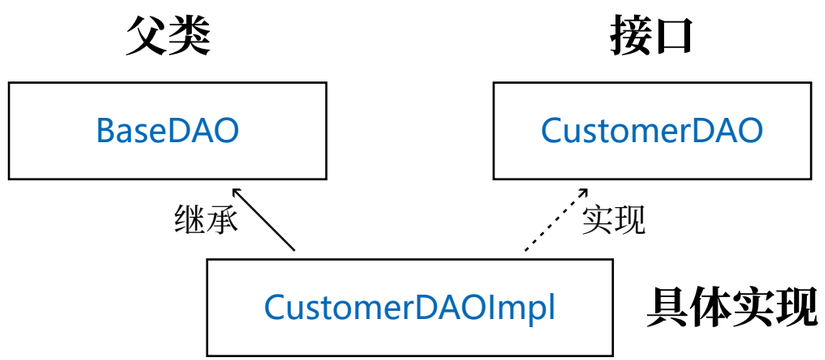
·DAO：Data Access Object 访问数据信息的类和接口，包括了对数据的 CRUD（Create、Retrival、Update、Delete），而不包含任何业务相关的信息。有时也称作：BaseDAO

·作用：为了实现功能的模块化，更有利于代码的维护和升级。

·下面是尚硅谷 JavaWeb 阶段书城项目中 DAO 使用的体现：



·层次结构：



7.1　优化前

BaseDAO 模块

```
//DAO：date(base) access object
// 封装了针对于数据表的通用操作
public abstract class BaseDAO {
	// 通用的增删改操作 （version -- 2.0：考虑上事务）
	public int updata(Connection conn, String sql, Object... args) {
// sql 中占位符的个数与可变形参相同
		PreparedStatement ps = null;
		try {
			// 1. 预编译 sql 语句，返回 PreparedStatement 的实例
			ps = conn.prepareStatement(sql);
			// 2. 填充占位符
			for (int i = 0; i < args.length; i++) {
				ps.setObject(i + 1, args[i]);// 小心参数声明错误
			}
			// 3. 执行
			return ps.executeUpdate();
		} catch (Exception e) {
			e.printStackTrace();
		} finally {
			// 4. 关闭资源
			JDBCUtils.closeResource(null, ps);
		}
		return 0;
	}
	// 通用的查询操作，用于返回数据表中的一条记录（version -- 2.0：
考虑上事务）
	public <T> T getInstance(Connection conn, Class<T> clazz,
String sql, Object... args) {
		PreparedStatement ps = null;
		ResultSet rs = null;
		try {
			ps = conn.prepareStatement(sql);
			for (int i = 0; i < args.length; i++) {
				ps.setObject(i + 1, args[i]);
			}
```

```
            rs = ps.executeQuery();
            // 通过结果集的元数据：ResultSetMetaData
            ResultSetMetaData rsmd = rs.getMetaData();
            // 通过 ResultSetMetaData 获取结果集中的列
            int columnCount = rsmd.getColumnCount();
            if (rs.next()) {
                T t = clazz.newInstance();
                for (int i = 0; i < columnCount; i++) {
                    Object columnValue = rs.getObject(i + 1);
                    // 获取每个列的列名
                    String columnName = rsmd.
getColumnName(i + 1);
                    // 给 cust 对象指定的某个属性，赋值为 value
                    Field field = clazz.
getDeclaredField(columnName);
                    field.setAccessible(true);
                    field.set(t, columnValue);
                }
                return t;
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            JDBCUtils.closeResource(null, ps, rs);
        }
        return null;
    }

    // 针对不同表的通用查询操作，返回表中的多条记录构成的集合
 (version -- 2.0：考虑上事务)
    public <T> List<T> getForList(Connection conn, Class<T> clazz,
String sql, Object... args) {
        PreparedStatement ps = null;
        ResultSet rs = null;
        try {
            ps = conn.prepareStatement(sql);
            for (int i = 0; i < args.length; i++) {
                ps.setObject(i + 1, args[i]);
            }
            rs = ps.executeQuery();
```



```
            // 通过结果集的元数据：ResultSetMetaData
            ResultSetMetaData rsmd = rs.getMetaData();
            // 通过 ResultSetMetaData 获取结果集中的列
            int columnCount = rsmd.getColumnCount();
            // 创建集合对象
            ArrayList<T> list = new ArrayList<>();
            while (rs.next()) {
                T t = clazz.newInstance();
                // 处理结果集一行数据中的每一个列，给 t 对象指定
的属性赋值
                for (int i = 0; i < columnCount; i++) {
                    Object columnValue = rs.getObject(i + 1);
                    // 获取每个列的列名
                    String columnName = rsmd.
getColumnName(i + 1);
                    // 给 cust 对象指定的某个属性，赋值为 value
                    Field field = clazz.
getDeclaredField(columnName);
                    field.setAccessible(true);
                    field.set(t, columnValue);
                }
                list.add(t);
            }
            return list;
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            JDBCUtils.closeResource(null, ps, rs);
        }
        return null;
    }

    // 用于查询特殊值的通用方法
    public <E> E getValue(Connection conn, String sql, Object...
args) {
        PreparedStatement ps = null;
        ResultSet rs = null;
        try {
            ps = conn.prepareStatement(sql);
            for (int i = 0; i < args.length; i++) {
```

```
                ps.setObject(i + 1, args[i]);
            }
            rs = ps.executeQuery();
            if (rs.next()) {
                return (E) rs.getObject(1);
            }
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            Util.JDBCUtils.closeResource(null, ps, rs);
        }
        return null;
    }
}
```

CustomerDAO 模块

```
// 此接口用于规范针对于 Customer 表的常用操作
public interface CustomerDAO {
    // 将 cust 对象添加到数据库中
    void insert(Connection conn, Customer cust);
    // 针对指定的 id，删除表中的一条记录
    void deleteById(Connection conn, int id);
    // 针对于内存中的 cust 对象，去修改数据表中指定的记录
    void updateById(Connection conn, Customer cust);
    // 针对于指定的 id 查询得到对应的 Customer 对象
    Customer getCustomerById(Connection conn, int id);
    // 查询表中的所有记录构成的集合
    List<Customer> getAll(Connection conn);
    // 返回数据表中数据的条目数
    long getCount(Connection conn);
    // 返回数据表中最大的生日
    Date getMaxBirth(Connection conn);
}
```


CustomerDAOImpl 模块

```
public class CustomerDAOImpl extends BaseDAO implements
CustomerDAO {

    @Override
    public void insert(Connection conn, Customer cust) {
        String sql = "insert into customers(name, email, birth)
values(?, ?, ?)";
        updata(conn, sql, cust.getName(), cust.getEmail(), cust.
getBirth());
    }

    @Override
    public void deleteById(Connection conn, int id) {
        String sql = "delete from customers where id = ?";
        updata(conn, sql, id);
    }

    @Override
    public void updateById(Connection conn, Customer cust) {
        String sql = "update customers set name = ? , email = ? ,
birth = ? where = ?";
        updata(conn, sql, cust.getName(), cust.getEmail(), cust.
getBirth(), cust.getId());
    }

    @Override
    public Customer getCustomerById(Connection conn, int id) {
        String sql = "select id, name, email, birth from customers
where id = ?";
        Customer customer = getInstance(conn, Customer.class,
sql, id);
        return customer;
    }
```

```
    @Override
    public List<Customer> getAll(Connection conn) {
        String sql = "select id, name, email, birth from customers";
        List<Customer> list = getForList(conn, Customer.class, sql);
        return list;
    }

    @Override
    public long getCount(Connection conn) {
        String sql = "select count( * ) from customers";
        return getValue(conn, sql);
    }

    @Override
    public Date getMaxBirth(Connection conn) {
        String sql = "select max(birth) from customers";
        return getValue(conn, sql);
    }
}
```


CustomerDAOImplTest 模块

```
class CustomerDAOImplTest {
    private CustomerDAOImpl dao = new CustomerDAOImpl();

    @Test
    void insert() {
        Connection conn = null;
        try {
            conn = JDBCUtils.getConnection();
            Customer cust = new Customer(1, " 于小飞 ",
"xiaofei@126.com", new Date(166564321L));
            dao.insert(conn, cust);
            System.out.println(" 添加成功。 ");
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            JDBCUtils.closeResource(conn, null);
        }
    }

    @Test
    void deleteById() {
        Connection conn = null;
        try {
            conn = JDBCUtils.getConnection();
            dao.deleteById(conn, 13);
            System.out.println(" 删除成功。 ");
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            JDBCUtils.closeResource(conn, null);
        }
    }

    @Test
    void updateById() {
        Connection conn = null;
        try {
            conn = JDBCUtils.getConnection();
```

```
            Customer cust = new Customer(18, " 贝多芬 ",
"beiduofen@126.com", new Date(231654231L));
            dao.updateById(conn, cust);
            System.out.println(" 修改成功 ");
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            JDBCUtils.closeResource(conn, null);
        }
    }

    @Test
    void getCustomerById() {
        Connection conn = null;
        try {
            conn = JDBCUtils.getConnection();
            Customer cust = dao.getCustomerById(conn, 19);
            System.out.println(cust);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            JDBCUtils.closeResource(conn, null);
        }
    }

    @Test
    void getAll() {
        Connection conn = null;
        try {
            conn = JDBCUtils.getConnection();
            List<Customer> list = dao.getAll(conn);
            list.forEach(System.out::println);
            System.out.println("");
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            JDBCUtils.closeResource(conn, null);
        }
    }
```



```
@Test
void getCount() {
    Connection conn = null;
    try {
        conn = JDBCUtils.getConnection();
        long count = dao.getCount(conn);
        System.out.println(" 表中的记录数为：" + count);
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        JDBCUtils.closeResource(conn, null);
    }
}

@Test
void getMaxBirth() {
    Connection conn = null;
    try {
        conn = JDBCUtils.getConnection();
        Date maxBirth = dao.getMaxBirth(conn);
        System.out.println(" 最大的生日为：" + maxBirth);
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        JDBCUtils.closeResource(conn, null);
    }
}
}
```

7.2 优化后

BaseDAO 模块

```
//DAO：date(base) access object
// 封装了针对于数据表的通用操作
public abstract class BaseDAO<T> {
	private Class<T> clazz = null;

	// public BaseDAO() {
	// }
	{
		// 获取当前 BaseDAO 的子类继承父类中的泛型
		Type genericSuperclass = this.getClass().
getGenericSuperclass();
		ParameterizedType parameterizedType =
(ParameterizedType) genericSuperclass;
		Type[] typeArguments = parameterizedType.
getActualTypeArguments();// 获取父类的泛型参数
		clazz = (Class<T>) typeArguments[0];// 泛型的第一个参数
	}

	// 通用的增删改操作 （version -- 2.0：考虑上事务）
	public int updata(Connection conn, String sql, Object... args) {
// sql 中占位符的个数与可变形参相同
		PreparedStatement ps = null;
		try {
			// 1. 预编译 sql 语句，返回 PreparedStatement 的实例
			ps = conn.prepareStatement(sql);
			// 2. 填充占位符
			for (int i = 0; i < args.length; i++) {
				ps.setObject(i + 1, args[i]);// 小心参数声明错误
			}
			// 3. 执行
			return ps.executeUpdate();
		} catch (Exception e) {
			e.printStackTrace();
		} finally {
			// 4. 关闭资源
			JDBCUtils.closeResource(null, ps);
		}
		return 0;
	}
	// 通用的查询操作，用于返回数据表中的一条记录（version -- 2.0：
考虑上事务）
	public T getInstance(Connection conn, String sql, Object... args) {
		PreparedStatement ps = null;
		ResultSet rs = null;
		try {
			ps = conn.prepareStatement(sql);
			for (int i = 0; i < args.length; i++) {
				ps.setObject(i + 1, args[i]);
			}
			rs = ps.executeQuery();
			// 通过结果集的元数据：ResultSetMetaData
			ResultSetMetaData rsmd = rs.getMetaData();
			// 通过 ResultSetMetaData 获取结果集中的列
			int columnCount = rsmd.getColumnCount();
			if (rs.next()) {
				T t = clazz.newInstance();
				for (int i = 0; i < columnCount; i++) {
					Object columnValue = rs.getObject(i + 1);
					// 获取每个列的列名
					String columnName = rsmd.
getColumnName(i + 1);
					// 给 cust 对象指定的某个属性，赋值为 value
					Field field = clazz.
getDeclaredField(columnName);
					field.setAccessible(true);
					field.set(t, columnValue);
				}
				return t;
			}
		} catch (Exception e) {
			e.printStackTrace();
		} finally {
			JDBCUtils.closeResource(null, ps, rs);
		}
		return null;
	}
```

```
    // 针对不同表的通用查询操作，返回表中的多条记录构成的集合
 (version -- 2.0：考虑上事务)
    public List<T> getForList(Connection conn, String sql, Object...
args) {
        PreparedStatement ps = null;
        ResultSet rs = null;
        try {
            ps = conn.prepareStatement(sql);
            for (int i = 0; i < args.length; i++) {
                ps.setObject(i + 1, args[i]);
            }
            rs = ps.executeQuery();
            // 通过结果集的元数据：ResultSetMetaData
            ResultSetMetaData rsmd = rs.getMetaData();
            // 通过 ResultSetMetaData 获取结果集中的列
            int columnCount = rsmd.getColumnCount();
            // 创建集合对象
            ArrayList<T> list = new ArrayList<>();
            while (rs.next()) {
                T t = clazz.newInstance();
                // 处理结果集一行数据中的每一个列，给 t 对象指定
的属性赋值
                for (int i = 0; i < columnCount; i++) {
                    Object columnValue = rs.getObject(i + 1);
                    // 获取每个列的列名
                    String columnName = rsmd.
getColumnName(i + 1);
                    // 给 cust 对象指定的某个属性，赋值为 value
                    Field field = clazz.
getDeclaredField(columnName);
                    field.setAccessible(true);
                    field.set(t, columnValue);
                }
                list.add(t);
            }
            return list;
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            JDBCUtils.closeResource(null, ps, rs);
```



```
        }
        return null;
    }

    // 用于查询特殊值的通用方法
    public <E> E getValue(Connection conn, String sql, Object...
args) {
        PreparedStatement ps = null;
        ResultSet rs = null;
        try {
            ps = conn.prepareStatement(sql);
            for (int i = 0; i < args.length; i++) {
                ps.setObject(i + 1, args[i]);
            }
            rs = ps.executeQuery();
            if (rs.next()) {
                return (E) rs.getObject(1);
            }
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            Util.JDBCUtils.closeResource(null, ps, rs);
        }
        return null;
    }
}
```

CustomerDAOImpl 模块

```
public class CustomerDAOImpl extends BaseDAO<Customer>
implements CustomerDAO {

	@Override
	public void insert(Connection conn, Customer cust) {
		String sql = "insert into customers(name, email, birth)
values(?, ?, ?)";
		updata(conn, sql, cust.getName(), cust.getEmail(), cust.
getBirth());
	}

	@Override
	public void deleteById(Connection conn, int id) {
		String sql = "delete from customers where id = ?";
		updata(conn, sql, id);
	}

	@Override
	public void updateById(Connection conn, Customer cust) {
		String sql = "update customers set name = ? , email = ? ,
birth = ? where = ?";
		updata(conn, sql, cust.getName(), cust.getEmail(), cust.
getBirth(), cust.getId());
	}

	@Override
	public Customer getCustomerById(Connection conn, int id) {
		String sql = "select id, name, email, birth from customers
where id = ?";
		Customer customer = getInstance(conn, sql, id);
		return customer;
	}

	@Override
	public List<Customer> getAll(Connection conn) {
		String sql = "select id, name, email, birth from customers";
		List<Customer> list = getForList(conn, sql);
		return list;
	}

	@Override
	public long getCount(Connection conn) {
		String sql = "select count( * ) from customers";
		return getValue(conn, sql);
	}

	@Override
	public Date getMaxBirth(Connection conn) {
		String sql = "select max(birth) from customers";
		return getValue(conn, sql);
	}
}
```

CustomerDAO 略

CustomerDAOImplTest 略

八　数据库连接池

8.1　JDBC 数据库连接池的必要性

· 在使用开发基于数据库的 web 程序时，传统的模式基本是按以下步骤：

· 在主程序（如 servlet、beans）中建立数据库连接；

· 进行 sql 操作

· 断开数据库连接

· 这种模式开发，存在的问题：

· 普通的 JDBC 数据库连接使用 DriverManager 来获取，每次向数据库建立连接的时候都要将 Connection 加载到内存中，再验证用户名和密码 (得花费 0.05s ～ 1s 的时间)。需要数据库连接的时候，就向数据库要求一个，执行完成后再断开连接。这样的方式将会消耗大量的资源和时间。数据库的连接资源并没有得到很好的重复利用。若同时有几百人甚至几千人在线，频繁的进行数据库连接操作将占用很多的系统资源，严重的甚至会造成服务器的崩溃。

· 对于每一次数据库连接，使用完后都得断开。否则，如果程序出现异常而未能关闭，将会导致数据库系统中的内存泄漏，最终将导致重启数据库。（回忆：何为 Java 的内存泄漏？）

· 这种开发不能控制被创建的连接对象数，系统资源会被毫无顾及的分配出去，如连接过多，也可能导致内存泄漏，服务器崩溃。

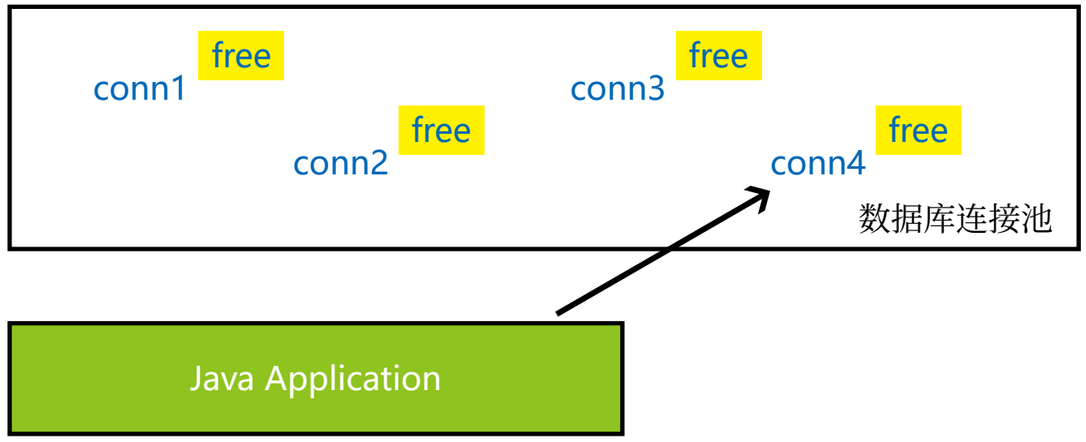
8.2　数据库连接池技术

· 为解决传统开发中的数据库连接问题，可以采用数据库连接池技术。

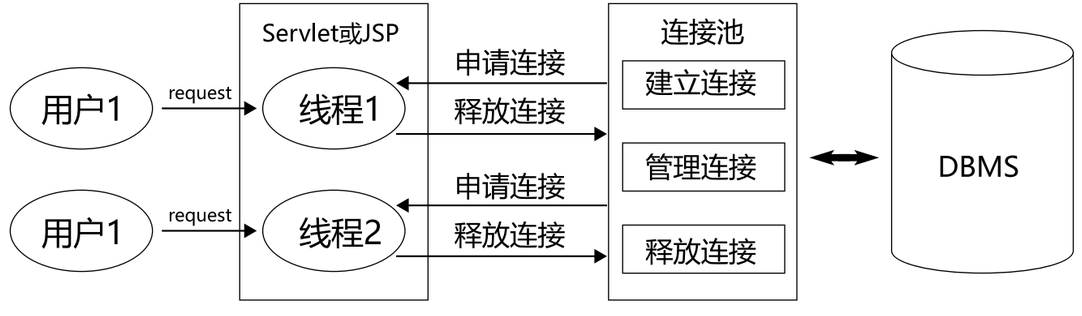
· 数据库连接池的基本思想：就是为数据库连接建立一个“ 缓冲池 ”。预先在缓冲池中放入一定数量的连接，当需要建立数据库连接时，只需从“ 缓冲池 ” 中取出一个，使用完毕之后再放回去。

· 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，而不是重新建立一个。

· 数据库连接池在初始化时将创建一定数量的数据库连接放到连接池中，这些数据库连接的数量是由最小数据库连接数来设定的。无论这些数据库连接是否被使用，连接池都将一直保证至少拥有这么多的连接数量。连接池的最大数据库连接数量限定了这个连接池能占有的最大连接数，当应用程序向连接池请求的连接数超过最大连接数量时，这些请求将被加入到等待队列中。



工作原理：



· 数据库连接池技术的优点

1. 资源重用

由于数据库连接得以重用，避免了频繁创建，释放连接引起的大量性能开销。在减少系统消耗的基础上，另一方面也增加了系统运行环境的平稳性。

2. 更快的系统反应速度

数据库连接池在初始化过程中，往往已经创建了若干数据库连接置于连接池中备用。此时连接的初始化工作均已完成。对于业务请求处理而言，直接利用现有可用连接，避免了数据库连接初始化和释放过程的时间开销，从而减少了系统的响应时间。

3. 新的资源分配手段

对于多应用共享同一数据库的系统而言，可在应用层通过数据库连接池的配置，实现某一应用最大可用数据库连接数的限制，避免某一应用独占所有的数据库资源。

4. 统一的连接管理，避免数据库连接泄漏

在较为完善的数据库连接池实现中，可根据预先的占用超时设定，强制回收被占用连接，从而避免了常规数据库连接操作中可能出现的资源泄露。

8.3 多种开源的数据库连接

· JDBC 的数据库连接池使用 javax.sql.DataSource 来表示，DataSource 只是一个接口，该接口通常由服务器 (Weblogic, WebSphere, Tomcat) 提供实现，也有一些开源组织提供实现：

· **DBCP** 是 Apache 提供的数据库连接池。tomcat 服务器自带 dbcp 数据库连接池。速度相对 c3p0 较快，但因自身存在 BUG，Hibernate3 已不再提供支持。

· **C3P0** 是一个开源组织提供的一个数据库连接池，速度相对较慢，稳定性还可以。hibernate 官方推荐使用。

· **Proxool** 是 sourceforge 下的一个开源项目数据库连接池，有监控连接池状态的功能，稳定性较 c3p0 差一点。

· **BoneCP** 是一个开源组织提供的数据库连接池，速度快

· **Druid** 是阿里提供的数据库连接池，据说是集 DBCP、C3P0、Proxool 优点于一身的数据库连接池，但是速度不确定是否有 BoneCP 快。

· DataSource 通常被称为数据源, 它包含连接池和连接池管理两个部分，习惯上也经常把 DataSource 称为连接池。

· DataSource 用来取代 DriverManager 来获取 Connection，获取速度快，同时可以大幅度提高数据库访问速度。

· 特别注意：

· 数据源和数据库连接不同，数据源无需创建多个，它是产生数据库连接的工厂，因此整个应用只需要一个数据源即可。

· 当数据库访问结束后，程序还是像以前一样关闭数据库连接：conn.close()；但 conn.close() 并没有关闭数据库的物理连接，它仅仅把数据库连接释放，归还给了数据库连接池。

8.3.1 C3P0 数据库连接池

```
public class C3P0 {
    @Test
    // 方式一
    public void testGetConnection() throws Exception {
        // 获取 C3P0 数据库连接池
        ComboPooledDataSource cpds = new
ComboPooledDataSource();
        cpds.setDriverClass("com.mysql.cj.jdbc.Driver");
        cpds.setJdbcUrl("jdbc:mysql://localhost:3306/mysql");
        cpds.setUser("root");
        cpds.setPassword("abc123");

        // 通过设置相关的参数，对数据库连接池进行管理
        // 设置初始时数据库连接池中的连接数
        cpds.setInitialPoolSize(10);
        Connection conn = cpds.getConnection();
        System.out.println(conn);
        // 销毁 C3P0 数据库连接池
        DataSources.destroy(cpds);
    }

    @Test
    // 方式二：使用配置文件
    public void testGetConnection2() throws Exception {
        ComboPooledDataSource cpds = new ComboPooledDataS
ource("helloc3p0");
        Connection conn = cpds.getConnection();
        System.out.println(conn);
    }
}
```


c3p0-config.xml

```
<c3p0-config>
    <!-- This app is massive! -->
    <named-config name="helloc3p0">
        <!-- 提供获取连接的 4 个基本信息 -->
        <property name="driverClass">com.mysql.cj.jdbc.Driver</
property>
        <property name="jdbcUrl">jdbc:mysql://localhost:3306/
mysql</property>
        <property name="user">root</property>
        <property name="password">abc123</property>
        <!-- 进行数据库连接池管理的基本信息 -->
        <!-- 当数据库连接池中的连接数不够时，c3p0 一次性向数据库
服务器申请的连接数 -->
        <property name="acquireIncrement">5</property>
        <!-- c3p0 数据库连接池中初始化时的连接数 -->
        <property name="initialPoolSize">10</property>
        <!-- c3p0 数据库连接池维护最少连接数 -->
        <property name="minPoolSize">10</property>
        <!-- c3p0 数据库连接池维护最多连接数 -->
        <property name="maxPoolSize">100</property>
        <!-- c3p0 数据库连接池最多维护的 Statement 的个数 -->
        <property name="maxStatements">50</property>
        <!-- 每个连接中可以最多使用的 Statement 的个数 -->
        <property name="maxStatementsPerConnection">2</
property>

    </named-config>
</c3p0-config>
```

8.3..2 DBCP 数据库连接池

· DBCP 是 Apache 软件基金组织下的开源连接池实现，该连接池依赖该组织下的另一个开源系统：Common - pool。如需使用该连接池实现，应在系统中增加如下两个 jar 文件：

· Commons - dbcp.jar：连接池的实现

· Commons - pool.jar：连接池实现的依赖库

· Tomcat 的连接池正是采用该连接池来实现的。该数据库连接池既可以与应用服务器整合使用，也可由应用程序独立使用。

· 数据源和数据库连接不同，数据源无需创建多个，它是产生数据库连接的工厂，因此整个应用只需要一个数据源即可。

· 当数据库访问结束后，程序还是像以前一样关闭数据库连接：conn.close(); 但上面的代码并没有关闭数据库的物理连接，它仅仅把数据库连接释放，归还给了数据库连接池。

· 配置属性说明：

属性	默认值	说明
initialSize0		连接池启动时创建的初始化连接数量
maxActive8		连接池中可同时连接的最大的连接数
maxIdle8		连接池中最大的空闲的连接数，超过的空闲连接将被释放，如果设置为负数表示不限制
minIdle0		连接池中最小的空闲的连接数，低于这个数量会被创建新的连接。该参数越接近maxIdle，性能越好，因为连接的创建和销毁，都是需要消耗资源的；但是不能太大。
maxWait	无限制	最大等待时间，当没有可用连接时，连接池等待连接释放的最大时间，超过该时间限制会抛出异常，如果设置-1表示无限等待
poolPreparedStatements	FALSE	开启池的Statement是否prepared
maxOpenPreparedStatements	无限制	开启池的prepared 后的同时最大连接数
minEvictableIdleTimeMillis		连接池中连接，在时间段内一直空闲， 被逐出连接池的时间
removeAbandonedTimeout	300	超过时间限制，回收没有用(废弃)的连接
removeAbandonedF	ALSE	超过removeAbandonedTimeout时间后，是否进 行没用连接（废弃）的回收


```java
// 测试 DBCP 的数据库连接池技术
public class DBCP {
    @Test
    // 方式一：不推荐
    public void testGetConnection() throws Exception {
        // 创建了 DBCP 的数据库连接池
        BasicDataSource source = new BasicDataSource();
        // 设置基本信息
        source.setDriverClassName("com.mysql.cj.jdbc.Driver");
        source.setUrl("jdbc:mysql://localhost:3306/mysql");
        source.setUsername("root");
        source.setPassword("abc123");

        // 还可以设置其他设计数据库连接池管理的相关属性
        source.setInitialSize(10);
        source.setMaxActive(10);
        Connection conn = source.getConnection();
        System.out.println(conn);
    }

    @Test
    // 方式二：使用配置文件 ( 推荐 )
    public void testGetConnection2() throws Exception {
        Properties pros = new Properties();
        // 方式 1:
        // InputStream is =
        // ClassLoader.getSystemClassLoader().
getResourceAsStream("dbcp.properties");
        // 方式 2:
        FileInputStream is = new FileInputStream(new File("H:\\
JAVA\\JDBC\\JDBC3\\lib\\dbcp.properties"));
        pros.load(is);
        // 创建一个 DBCP 数据库连接池
        DataSource source = BasicDataSourceFactory.
createDataSource(pros);
        Connection conn = source.getConnection();
        System.out.println(conn);
    }
```

```
        // 方式三：使用配置文件 ( 推荐 )
        private static DataSource source;
        static {
            try {
                Properties pros = new Properties();
                FileInputStream is = new FileInputStream(new
File("H:\\JAVA\\JDBC\\JDBC3\\lib\\dbcp.properties"));
                pros.load(is);
                // 创建一个 DBCP 数据库连接池
                source = BasicDataSourceFactory.
createDataSource(pros);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        public static Connection testGetConnection3() throws Exception
{
            Connection conn = source.getConnection();
            System.out.println(conn);
            return conn;
        }
}
```

dbcp.properties

```
driverClassName=com.mysql.cj.jdbc.Driver
url=jdbc:mysql://localhost:3306/mysql?useUnicode=true&characterE
ncoding=UTF-8&useSSL=false&serverTimezone=Asia/Shanghai&zer
oDateTimeBehavior=CONVERT_TO_NULL&allowPublicKeyRetrieval=t
rue&useSSL=false
username=root
password=abc123

initialSize=10
```


8.3.3 Druid（德鲁伊）数据库连接池

Druid 是阿里巴巴开源平台上一个数据库连接池实现，它结合了 C3P0、DBCP、Proxool 等 DB 池的优点，同时加入了日志监控，可以很好的监控 DB 池连接和 SQL 的执行情况，可以说是针对监控而生的 DB 连接池，可以说是目前最好的连接池之一。

```
public class Druid {
    @Test
    public void getConnection() throws Exception {
        Properties pros = new Properties();
        InputStream is = ClassLoader.getSystemClassLoader().
getResourceAsStream("lib\\Druid.properties");
        pros.load(is);
        DataSource source = DruidDataSourceFactory.
createDataSource(pros);
        Connection conn = source.getConnection();
        System.out.println(conn);
    }
}
```

Druid.properties

```
url=jdbc:mysql://localhost:3306/mysql?useUnicode=true&characterE
ncoding=UTF-8&useSSL=false&serverTimezone=Asia/Shanghai&zer
oDateTimeBehavior=CONVERT_TO_NULL&allowPublicKeyRetrieval=t
rue&useSSL=false
username=root
password=abc123
driverClassName=com.mysql.cj.jdbc.Driver

initialSize=10
maxActive=10
```

详细配置参数：

配置	缺省	说明
name		配置这个属性的意义在于，如果存在多个数据源，监控的时候 可以通过名字来区分开来。如果没有配置，将会生成一个名字， 格式是："DataSource-" + System.identityHashCode(this)
url		连接数据库的url，不同数据库不一样。例如：mysql： jdbc:mysql://10.20.153.104:3306/druid2 oracle： jdbc:oracle:thin:@10.20.149.85:1521:ocnauto
username		连接数据库的用户名
password		连接数据库的密码。如果你不希望密码直接写在配置文件中， 可以使用ConfigFilter。详细看这里：https://github.com/ alibaba/druid/wiki/%E4%BD%BF%E7%94%A8ConfigFilter
driverClassName		根据url自动识别 这一项可配可不配，如果不配置druid会根据url 自动识别dbType，然后选择相应的driverClassName(建议配置下)
initialSize	0	初始化时建立物理连接的个数。初始化发生在显示调用init方法， 或者第一次getConnection时
maxActive	8	最大连接池数量
maxIdle	8	已经不再使用，配置了也没效果
minIdle		最小连接池数量
maxWait		获取连接时最大等待时间，单位毫秒。配置了maxWait之后， 缺省启用公平锁，并发效率会有所下降，如果需要可以通过 配置useUnfairLock属性为true使用非公平锁。
poolPreparedStatements	FALSE	是否缓存preparedStatement，也就是PSCache。PSCache对支持 游标的数据库性能提升巨大，比如说oracle。在mysql下建议关闭。
maxOpenPreparedStatements	-1	要启用PSCache，必须配置大于0，当大于0时，poolPreparedStatements 自动触发修改为true。在Druid中，不会存在Oracle下PSCache占用 内存过多的问题，可以把这个数值配置大一些，比如说100
validationQuery		用来检测连接是否有效的sql，要求是一个查询语句。如果 validationQuery为null，testOnBorrow、testOnReturn、 testWhileIdle都不会其作用。
testOnBorrow	TRUE	申请连接时执行validationQuery检测连接是否有效，做了这个配置会降低性能。
testOnReturn	FALSE	归还连接时执行validationQuery检测连接是否有效，做了这个配置会降低性能。
testWhileIdle	FALSE	建议配置为true，不影响性能，并且保证安全性。申请连接的时候检 测，如果空闲时间大于timeBetweenEvictionRunsMillis，执行 validationQuery检测连接是否有效。
timeBetweenEvictionRunsMillis		有两个含义： 1)Destroy线程会检测连接的间隔时间 2)testWhileIdle的判断依据，详细看testWhileIdle属性的说明
numTestsPerEvictionRun		不再使用，一个DruidDataSource只支持一个EvictionRun
minEvictableIdleTimeMillis		
connectionInitSqls		物理连接初始化的时候执行的sql
exceptionSorter		根据dbType自动识别 当数据库抛出一些不可恢复的异常时，抛弃连接
filters		属性类型是字符串，通过别名的方式配置扩展插件，常用的插件有： 监控统计用的filter:stat日志用的filter:log4j防御sql注入的filter:wall
proxyFilters		类型是List，如果同时配置了filters和proxyFilters，是组合关系，并非替换关系

JDBCUtils

```
public class JDBCUtils {
    public static Connection getConnection() throws Exception {
        //1. 读取配置文件中的 4 个基本信息
            InputStream is = ClassLoader.getSystemClassLoader().
getResourceAsStream("jdbc.properties");
        Properties pro = new Properties();

        pro.load(is);
        String user = pro.getProperty("user");
        String url = pro.getProperty("url");
        String password = pro.getProperty("password");
        String driverClass = pro.getProperty("driverClass");

        //2. 加载驱动
        Class.forName(driverClass);
        //3. 获取连接
         Connection conn = DriverManager.getConnection(url, user,
password);
        return conn;
    }

    // 使用 C3P0 的数据库连接池技术
    // 数据库连接池只需提供一个即可
     private static ComboPooledDataSource cpds = new ComboPool
edDataSource("helloc3p0");

    public static Connection getConnection2() throws Exception {
        Connection conn = cpds.getConnection();
        return conn;
    }

    // 使用 DBCP 数据库连接池技术获取数据库连接
    private static DataSource source;
    static {
        try {
            Properties pros = new Properties();
                FileInputStream is = new FileInputStream(new
File("H:\\JAVA\\JDBC\\JDBC3\\lib\\dbcp.properties"));
```

```
                pros.load(is);
                // 创建一个 DBCP 数据库连接池
                source = BasicDataSourceFactory.
createDataSource(pros);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        public static Connection testGetConnection3() throws Exception
{
            Connection conn = source.getConnection();
            System.out.println(conn);
            return conn;
        }

        // 使用 Druid 数据库连接池技术获取数据库连接
        private static DataSource source2;
        static {
            try {
                Properties pros = new Properties();
                 InputStream is = ClassLoader.getSystemClassLoader().
getResourceAsStream("lib\\Druid.properties");
                pros.load(is);
                source2 = DruidDataSourceFactory.
createDataSource(pros);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        public static Connection getConnection4() throws Exception {
            Connection conn = source.getConnection();
            return conn;
        }

        public static void closeResource(Connection conn, Statement ps,
ResultSet rs) {
```



```
            /**
             * 关闭连接的操作
             */
            try {
                if (ps != null) ps.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
            try {
                if (conn != null) conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
            try {
                if (rs != null) rs.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
    }
}
```

九 Apache-DBUtils

9.1 Apache-DBUtils 简介

Apache-DBUtils 实现 CRUD 操作

· commons-dbutils 是 Apache 组织提供的一个开源 JDBC 工具类库，它是对 JDBC 的简单封装，学习成本极低，并且使用 dbutils 能极大简化 jdbc 编码的工作量，同时也不会影响程序的性能。

· API 介绍：

· org.apache.commons.dbutils.QueryRunner

· org.apache.commons.dbutils.ResultSetHandler

· 工具类：org.apache.commons.dbutils.DbUtils

9.2 主要 API 的使用

9.2.1 DbUtils

· DbUtils ：提供如关闭连接、装载 JDBC 驱动程序等常规工作的工具类，里面的所有方法都是静态的。主要方法如下：

· public static void close(…) throws java.sql.SQLException： DbUtils 类提供了三个重载的关闭方法。这些方法检查所提供的参数是不是 NULL，如果不是的话，它们就关闭 Connection、Statement 和 ResultSet。

· public static void closeQuietly(…): 这一类方法不仅能在 Connection、Statement 和 ResultSet 为 NULL 情况下避免关闭，还能隐藏一些在程序中抛出的 SQLEeception。

· public static void commitAndClose(Connection conn)throws SQLException：用来提交连接的事务，然后关闭连接。

· public static void commitAndCloseQuietly(Connection conn)：用来提交连接，然后关闭连接，并且在关闭连接时不抛出 SQL 异常。

· public static void rollback(Connection conn)throws SQLException：允许 conn 为 null，因为方法内部做了判断

· public static void rollbackAndClose(Connection conn)throws SQLException

· rollbackAndCloseQuietly(Connection)

· public static boolean loadDriver(java.lang.String driverClassName)：这一方装载并注册 JDBC 驱动程序，如果成功就返回 true。使用该方法，你不需要捕捉这个异常 ClassNotFoundException。

9.2.2 QueryRunner 类

· 该类简单化了 SQL 查询，它与 ResultSetHandler 组合在一起使用可以完成大部分的数据库操作，能够大大减少编码量。

· QueryRunner 类提供了两个构造器：

· 默认的构造器；

· 需要一个 javax.sql.DataSource 来作参数的构造器。

· QueryRunner 类的主要方法：

· 更新：

· public int update(Connection conn, String sql, Object... params) throws SQLException: 用来执行一个更新（插入、更新或删除）操作。

· .....

· 插入

· public T insert(Connection conn,String sql,ResultSetHandler rsh, Object... params) throwsSQLException：只支持 INSERT 语句，其中 rsh - The handler used to create the result object from the ResultSet of auto-generated keys. 返回值： An object generated by the handler. 即自动生成的键值

· .....

· 批处理

· public int[] batch(Connection conn,String sql,Object[][] params) throws SQLException：INSERT,UPDATE, or DELETE 语句
public T insertBatch(Connection conn,String sql,ResultSetHandler rsh,Object[][] params)throws SQLException：只支持 INSERT 语句

· .....

· 查询

· public Object query(Connection conn, String sql, ResultSetHandler rsh,Object... params) throws SQLException：执行一个查询操作，在这个查询中，对象数组中的每个元素值被用来作为查询语句的置换参数。该方法会自行处理 PreparedStatement 和 ResultSet 的创建和关闭。

· .....

commons-dbutils 是 Apache 组织提供的一个开源 JDBC 工具类库，封装了针对于数据库的增删改查操作。

```
public class QueryRunnerTest {
    @Test
    // 测试插入
    public void testInsert() {
        Connection conn = null;
        try {
            QueryRunner runner = new QueryRunner();
            conn = JDBCUtils.getConnection4();
            String sql = "insert into customers(name, email, birth)
values(?, ?, ?)";
            int insertCount = runner.update(conn, sql, " 周周 ",
"zz@126.com", "1990-01-01");
            System.out.println(" 添加了 " + insertCount + " 条记录 ");
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            JDBCUtils.closeResource(conn, null, null);
        }
    }

    @Test
    // 测试查询
    // BeanHandler：是 ResultSetHandler 接口的实现类，用于封装表
中的一条记录
    public void testQuery() {
        Connection conn = null;
        try {
            QueryRunner runner = new QueryRunner();
            conn = JDBCUtils.getConnection4();
            String sql = "select id, name, email, birth from
customers where id = ?";
            BeanHandler<Customer> handler = new
BeanHandler<>(Customer.class);
            Customer customer = runner.query(conn, sql, handler,
23);
            System.out.println(customer);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            JDBCUtils.closeResource(conn, null, null);
```



```
        }
    }

    @Test
    // BeanHandler：是 ResultSetHandler 接口的实现类，用于封装表
中的多条记录构成的集合
    public void testQuery2() {
        Connection conn = null;
        try {
            QueryRunner runner = new QueryRunner();
            conn = JDBCUtils.getConnection4();
            String sql = "select id, name, email, birth from
customers where id < ?";
            BeanListHandler<Customer> handler = new
BeanListHandler<>(Customer.class);
            List<Customer> list = runner.query(conn, sql, handler,
23);
            list.forEach(System.out::println);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            JDBCUtils.closeResource(conn, null, null);
        }
    }

    @Test
    // MapHandler：是 ResultSetHandler 接口的实现类，对应表中的
一条记录
    // 将字段及相应的字段值作为 map 中的 key 和 value
    public void testQuery3() {
        Connection conn = null;
        try {
            QueryRunner runner = new QueryRunner();
            conn = JDBCUtils.getConnection4();
            String sql = "select id, name, email, birth from
customers where id 《 ?";
            MapHandler handler = new MapHandler();
            Map<String, Object> map = runner.query(conn, sql,
handler, 23);
            System.out.println(map);
```

```
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                JDBCUtils.closeResource(conn, null, null);
            }
        }

        @Test
        // MapListHandler：是 ResultSetHandler 接口的实现类，对应表中的多条记录
        // 将字段及相应的字段值作为 map 中的 key 和 value，将这些 map 添加到 List 中
        public void testQuery4() {
            Connection conn = null;
            try {
                QueryRunner runner = new QueryRunner();
                conn = JDBCUtils.getConnection4();
                String sql = "select id, name, email, birth from customers where id < ?";
                MapListHandler handler = new MapListHandler();
                List<Map<String, Object>> list = runner.query(conn, sql, handler, 23);
                list.forEach(System.out::println);
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                JDBCUtils.closeResource(conn, null, null);
            }
        }
    }
```


9.2.3 ResultSetHandler 接口及实现类

· 该接口用于处理 java.sql.ResultSet，将数据按要求转换为另一种形式。
· ResultSetHandler 接口提供了一个单独的方法：Object handle (java.sql.ResultSet .rs)。
· 接口的主要实现类：
  · **ArrayHandler：** 把结果集中的第一行数据转成对象数组。
  · **ArrayListHandler：** 把结果集中的每一行数据都转成一个数组，再存放到 List 中。
  · **BeanHandler：** 将结果集中的第一行数据封装到一个对应的 JavaBean 实例中。
  · **BeanListHandler：** 将结果集中的每一行数据都封装到一个对应的 JavaBean 实例中，存放到 List 里。
  · **ColumnListHandler：** 将结果集中某一列的数据存放到 List 中。
  · **KeyedHandler(name)：** 将结果集中的每一行数据都封装到一个 Map 里，再把这些 map 再存到一个 map 里，其 key 为指定的 key。
  · **MapHandler：** 将结果集中的第一行数据封装到一个 Map 里，key 是列名，value 就是对应的值。
  · **MapListHandler:** 将结果集中的每一行数据都封装到一个 Map 里，然后再存放到 List
  · **ScalarHandler：** 查询单个值对象

```
public class QueryRunnerTest {
    @Test
    // ScalarHandler：用于查询特殊值
    public void testQuery5() {
        Connection conn = null;
        try {
            QueryRunner runner = new QueryRunner();
            conn = JDBCUtils.getConnection4();
            String sql = "select count( * ) from customers";
            ScalarHandler handler = new ScalarHandler();
            Long count = (Long) runner.query(conn, sql, handler);
            System.out.println(count);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            JDBCUtils.closeResource(conn, null, null);
        }
    }
```

```
    @Test
    // ScalarHandler：用于查询最大的生日
    public void testQuery6() {
        Connection conn = null;
        try {
            QueryRunner runner = new QueryRunner();
            conn = JDBCUtils.getConnection4();
            String sql = "select max(birth) from customers";
            ScalarHandler handler = new ScalarHandler();
            Date maxBirth = (Date) runner.query(conn, sql,
handler);
            System.out.println(maxBirth);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            JDBCUtils.closeResource(conn, null, null);
        }
    }

    @Test
    // 自定义 ResultSetHandler 的实现类
    public void testQuery7() {
        Connection conn = null;
        try {
            QueryRunner runner = new QueryRunner();
            conn = JDBCUtils.getConnection4();
            String sql = "select id, name, email, birth from
customers where id = ?";
            ResultSetHandler<Customer> handler = new
ResultSetHandler<Customer>() {
                @Override
                public Customer handle(ResultSet rs) throws
SQLException {
                    // return null;
                    if (rs.next()) {
                        int id = rs.getInt("id");
                        String name = rs.getString("name");
                        String email = rs.getString("email");
                        Date birth = rs.getDate("birth");
```



```
                        Customer customer = new
Customer(id, name, email, birth);
                        return customer;
                    }
                    return null;
                }
            };
            Customer customer = runner.query(conn, sql, handler,
23);
            System.out.println(customer);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            JDBCUtils.closeResource(conn, null, null);
        }
    }
}

// 使用 dbutils.jar 中提供的 DbUtils 工具类，实现资源的关闭
    public static void closeResource2(Connection conn, Statement
ps, ResultSet rs) {
        DbUtils.closeQuietly(conn);
        DbUtils.closeQuietly(ps);
        DbUtils.closeQuietly(rs);
    }
```

JavaWeb 技术概览