轻量级 J2EE 框架应用

E 5　A Simple Controller with DAO pattern

学号：SA18225170

姓名：李朝喜

报告撰写时间：2018/12/27

文档目录

1. 主题概述

1.1 J2EE DAO

DAO（Data Access Object）是一个数据访问接口，数据访问：顾名思义就是与数据库打交道。夹在业务逻辑与数据库资源中间。

在核心 J2EE 模式中是这样介绍 DAO 模式的：为了建立一个健壮的 J2EE 应用，应该将所有对数据源的访问操作抽象封装在一个公共 API 中。用程序设计的语言来说，就是建立一个接口，接口中定义了此应用程序中将会用到的所有事务方法。在这个应用程序中，当需要和数据源进行交互的时候则使用这个接口，并且编写一个单独的类来实现这个接口在逻辑上对应这个特定的数据存储。

1.2 JDBC

JDBC（Java DataBase Connectivity，Java 数据库连接）是一种用于执行 SQL 语句的 Java API，可以为多种关系数据库提供统一访问，它由一组用 Java 语言编写的类和接口组成。JDBC 提供了一种基准，据此可以构建更高级的工具和接口，使数据库开发人员能够编写数据库应用程序。

Java 数据库连接体系结构是用于 Java 应用程序连接数据库的标准方法。JDBC 对 Java 程序员而言是 API，对实现与数据库连接的服务提供商而言是接口模型。作为 API，JDBC 为程序开发提供标准的接口，并为数据库厂商及第三方中间件厂商实现与数据库的连接提供了标准方法。

1.3 MySQL

MySQL 是一个关系型数据库管理系统，由瑞典 MySQL AB 公司开发，目前属于 Oracle 旗下产品。MySQL 是最流行的关系型数据库管理系统之一，在 WEB 应用方面，MySQL 是最好的 RDBMS (Relational Database Management System，关系数据库管理系统) 应用软件。

MySQL 所使用的 SQL 语言是用于访问数据库的最常用标准化语言。MySQL 软件采用了双授权政策，分为社区版和商业版，由于其体积小、速度快、总体拥有成本低，尤其是开放源码这一特点，一般中小型网站的开发都选择 MySQL 作为网站数据库。

1.4 PostgreSQL

PostgreSQL 是以加州大学伯克利分校计算机系开发的 POSTGRES，现在已经更名为 PostgreSQL，以版本 4.2 为基础的免费的对象关系型数据库管理系统（ORDBMS）。

PostgreSQL 支持大部分 SQL 标准并且提供了许多其他现代特性：复杂查询、外键、触发器、视图、事务完整性、MVCC。同样，PostgreSQL 可以用许多方法扩展，比如增加新的数据类型、函数、操作符、聚集函数、索引。

2. 假设

需要注意，实验 E5 是在 E4 基础上进行的，所以在做 E5 前请先完成 E4，否则可能对项目上下文等内容不清楚。

2.1 知识背景

根据我的实际情况，我认为在至少具备以下知识背景的前提下进行实验，更有助于较好的完成实验。

- 具备基本的 SQL 编程能力；
- 理解 Java 基础知识，并且具备基本的 Java 编程能力，如：理解 Java 中对象封装、继承等概念，以及具备 JavaSE 基本开发能力；
- 了解 JDBC 相关概念，能够使用 JDBC 与数据库进交互（CURD）；

2.2 环境背景

本次实验是在 Windows 10 系统下进行的，部分操作可能对其他系统并不适用。此外，请确保你的电脑至少有 2G 的内存，否则在同时运行多款软件时，可能会出现卡顿现象。

具体要求：

- 已安装并且配置好 Java 环境：Java SE Development Kit 11.0.1
- 已安装好 Eclipse：eclipse-jee-2018-09-win32-x86_64.zip
- 已安装好 Tomcat：https://tomcat.apache.org/download-90.cgi
- 已安装好 Maven：https://maven.apache.org/download.cgi
- 已安装好 MySQL：https://dev.mysql.com/downloads/mysql/
- 已安装好 PostgreSQL：
  https://www.enterprisedb.com/downloads/postgres-postgresql-downloads

注意：软件安装以及相关配置不在本文档涉及范围内，对于如：Java 环境变量配置、Maven 修改镜像源、Eclipse 中配置 Maven 等问题，请读者自行查阅资料。

3. 实现或证明

3.1 项目包结构及简要描述

SimpleController

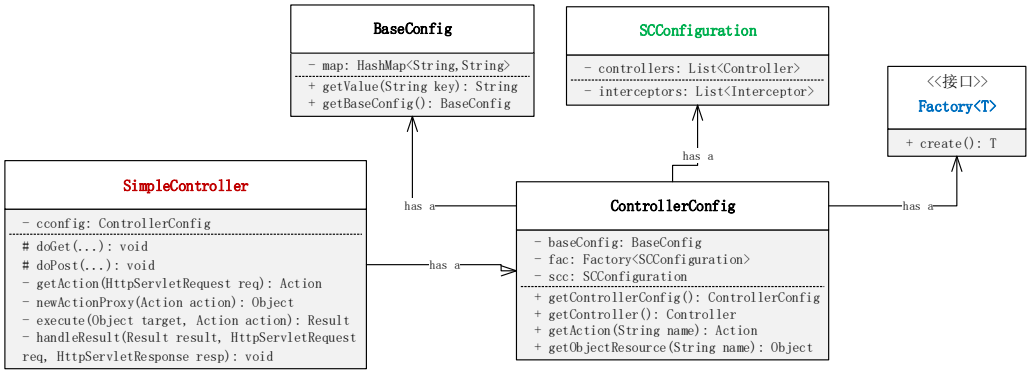
```
|- src/main/java
   |- southday.j2eework.sc.ustc.controller          # 包根目录
      |- SimpleController.java                      # Servlet，用于处理*.sc的请求
      |- action                                     # action包，里面包含了Action接口定义及默认实现类等
         |- ActionIface.java                        # Action接口
         |- ActionSupport.java                      # Action的默认实现类，框架使用者可以继承该类
         |- DefaultAction.java                      # 默认Action，当未找到匹配的action时，就返回DefaultAction
      |- config                                     # Config包，里面包含了项目所用的公共资源对象类，通常是单例实现
         |- BaseConfig.java                         # 基本资源配置类，包括UseSC中资源文件(.jsp、.xml)的位置，以及一些公共使用的配置属性
         |- ControllerConfig.java                   # Controller对象配置类，通过扫描UseSC中的controller.xml文件，来获得全局共享的Controller对象
         |- bean                                    # bean包，包含与UseSC中controller.xml文件里定义内容相对应的JavaBean对象类
            |- Controller.java                      # 与controller.xml中定义的<controller>标签相对应
            |- Action.java                          # 与controller.xml中定义的<action>标签相对应
            |- Result.java                          # 与controller.xml中定义的<result>标签相对应
            |- Interceptor.java                     # 与controller.xml中定义的<interceptor>标签相对应
            |- InterceptorRef.java                  # 与controller.xml中定义的<interceptor-ref>标签相对应
            |- SCConfiguration.java                 # 与controller.xml中定义的<sc-configuration>标签相对应
         |- factory                                 # factory包，工厂模式，里面包含了用于创建对象的工厂类
            |- ActionFactory.java                   # 用于生成Action对象的工厂
            |- ResultFactory.java                   # 用于生成Result对象的工厂
            |- ControllerFactory.java               # 用于生成Controller对象的工厂
            |- SCConfigurationFactory.java          # 用于生成SCConfiguration对象的工厂
            |- dom4j                                # dom4j包，基于DOM4J 的 XML解析来实现SCConfiguration对象创建的工厂
               |- DOM4JSCConfigurationFactory.java  # 基于DOM4J的 XML解析来创建SCConfiguration对象的具体实现类
      |- dao                                        # dao包，包含各类DAO
         |- BaseDAO.java                            # 基本DAO，作为其他DAO的父类存在
      |- factory                                    # factory包，目前包含工厂的抽象定义（接口Factory）
         |- Factory.java                            # 抽象工厂，提供接口：T create() throws Exception;
      |- interceptor                                # interceptor包，用于存放拦截器的相关类、接口等
         |- ParameterInterceptor.java               # 参数拦截器，目前仅用于给Action代理对象（动态）填充参数
      |- proxy                                      # proxy包，包含与代理机制相关的类
         |- ProxyFactory.java                       # 代理工厂，用于生成代理类对象
         |- cglib                                   # cglib包，关于Action的动态代理，使用cglib技术来实现
            |- LogActionProxy.java                  # 针对日志记录的Action代理拦截器
      |- transformer                                # transformer包，包含各类转换器
         |- XML2HTMLTransformer.java                # 将xml转为html的转换器，目前使用xslt技术实现
      |- util                                       # util包，包含项目工具类
         |- CommonUtil.java                         # 包含普遍被使用的、公用的方法，如：获取类加载路径、检查String类型参数等
         |- FileUtil.java                           # 包含与文件资源处理相关的常用方法，如：关闭资源，加载properties配置等
         |- ReflectUtil.java                        # 包含与反射处理相关的常用方法
```

UseSC

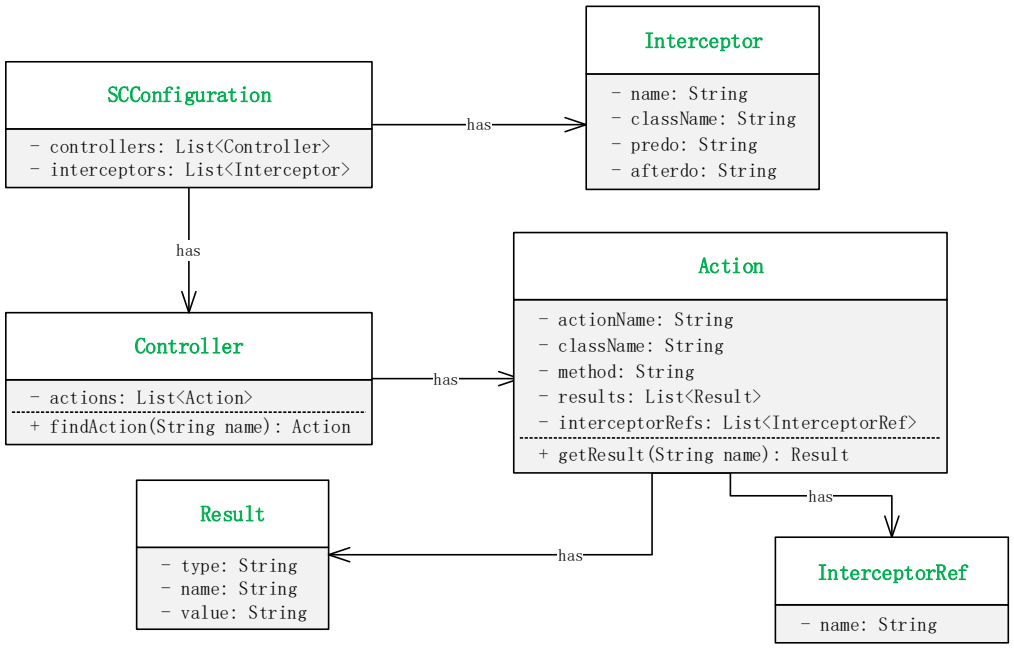
```
|- src/main/java
   |- southday.j2eework.water.ustc                 # 包根目录
      |- action                                    # action包，包含了具体业务对应的action类
         |- LoginAction.java                       # 登陆业务对应的Action
         |- RegisterAction.java                    # 注册业务对应的Action
         |- LogoutAction.java                      # 退出登陆对应的Action
      |- bean                                      # bean包
         |- User.java                              # UserBean
      |- db                                        # db包，服务端管理数据库资源的包
         |- SimpleUserDB.java                      # 自制的简易User数据库，内部使用ConcurrentHashMap实现，单例模式
      |- interceptor                               # interceptor包，包含拦截器相关类
         |- LogInterceptor.java                    # 日志记录拦截器
      |- log                                       # log包，包含与日志记录操作相关的类
         |- LogProperties.java                     # 用于加载和保存log.properties配置文件中的信息
         |- LogWriter.java                         # 用于写日志
         |- bean                                   # bean包，包含日志文件xml中相对应的bean对象类
            |- Log.java                            # 与log.xml中定义的<log>标签相对应
            |- ActionLog.java                      # 与log.xml中定义的<action>标签相对应
      |- service                                   # service包，包含模型层（业务逻辑处理）的相关类
         |- UserService.java                       # 接口，定义了与User有关的相关业务逻辑的抽象方法
         |- impl                                   # impl包，包含了针对模型层接口的具体实现类
            |- UserServiceImpl.java                # UserService接口的具体实现类
      |- dao                                       # dao包，包含各类DAO
         |- UserDAO.java                           # 实现BaseDAO的具体类，包含query, insert, update, delete等方法的实现
   |- src/main/resources                           # Web项目的资源文件包
      |- config                                    # config包，包含了开发者自定义的配置文件
         |- controller.xml                         # controller.xml，里面配置了Action的执行策略、结果等信息，会被SimpleController项目扫描使用
         |- files-locations.properties             # files-locations.properties，里面包含了各资源文件的存放位置信息，会被SimpleController项目扫描使用
         |- log.properties                         # log.properties，里面包含了关于日志记录的一些配置信息，比如：日志文件保存位置等
         |- jdbc-mysql.properties                  # jdbc-mysql.properties，里面包含了DBCP用于创建、管理数据库连接的配置属性（针对MySQL）
         |- jdbc-postgresql.properties             # jdbc-postgresql.properties，里面包含了 DBCP用于创建、管理数据库连接的配置属性（针对PostgreSQL）
   |- src/main/webapp                              # Web项目部署包
      |- welcome.jsp                               # 项目首页 jsp
      |- WEB-INF                                   # Web配置包
         |- web.xml                                # Web项目配置文件
      |- pages                                     # pages包，存放返回结果页面
         |- failure.jsp                            # “请求失败”页面
         |- no-req-resource.jsp                    # “为找到请求资源”页面
         |- unkonwn-action.jsp                     # “未知Action”页面
         |- welcome.jsp                            # “欢迎”页面（用户已登陆）
         |- success_view.xml                       # “登陆成功”页面的XML配置
         |- success_view.xsl                       # “登陆成功”页面的XSL配置
```

3.2 UML 类图

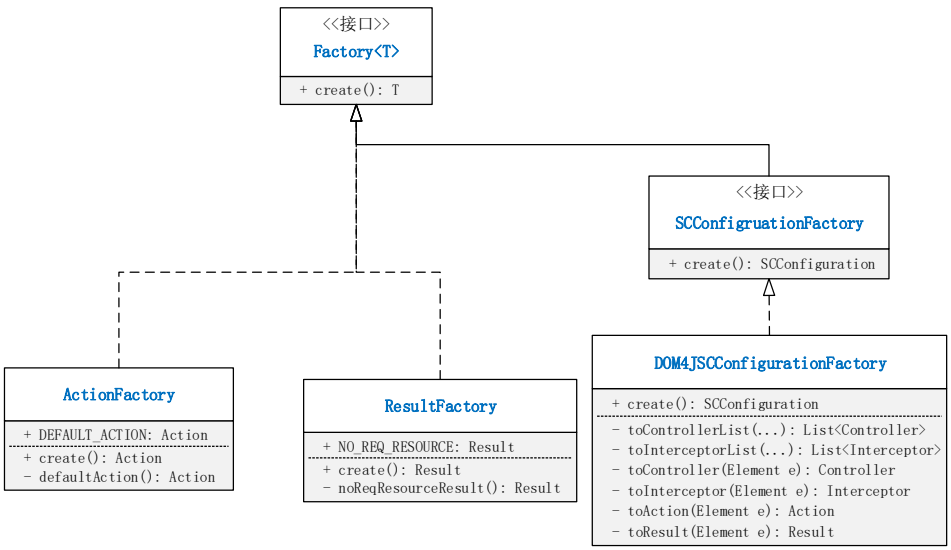
3.2.1 SimpleController UML 类图



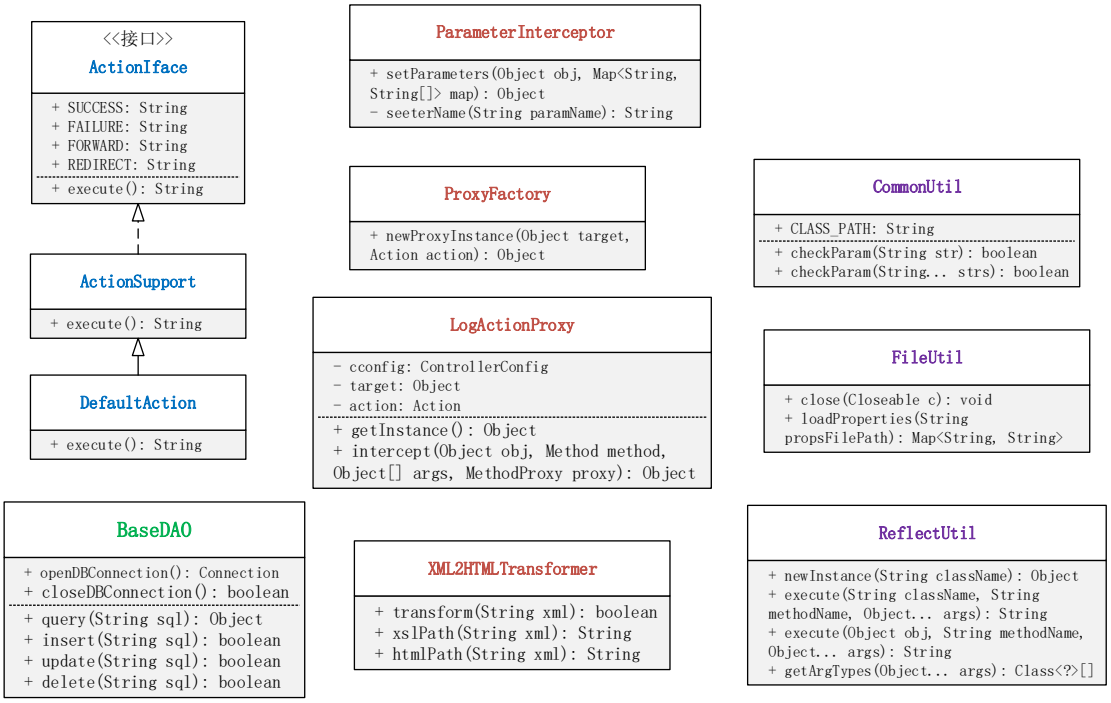
Servlet 相关类-UML 类图



contoller.xml 对应的相关 Bean 类-UML 类图

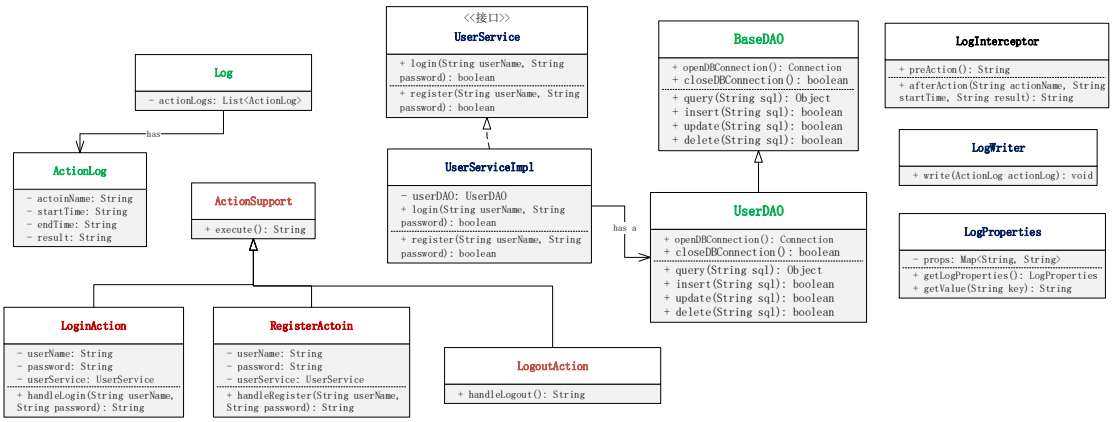


Factory-UML 类图



其他相关类-UML 类图

3.2.2 UseSC UML 类图



3.3 引入 DBCP

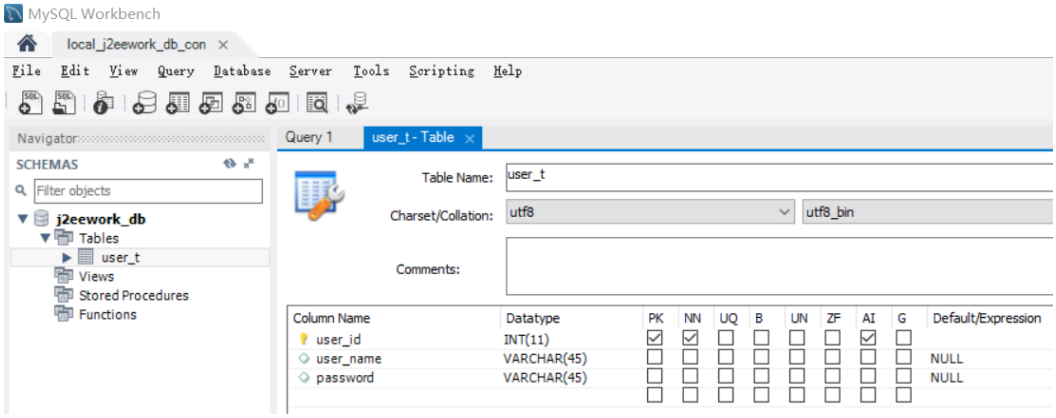
DBCP（DataBase Connection Pool）是一个开源的数据库连接池。由 Apache 开发，通过数据库连接池，可以让程序自动管理数据库连接的释放和断开。

在 UseSC 工程的 pom.xml 中添加如下依赖：

```
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-dbcp2</artifactId>
    <version>2.5.0</version>
</dependency>
```

3.4 MySQL 部分

1）创建数据库 j2eework_db，在该数据库中创建表 user_t，表中创建字段：user_id、user_name、password，如下：



2）创建用户 lcx，并对其授权，SQL 如下：

```
# 创建用户
CREATE USER 'lcx'@'localhost' IDENTIFIED BY 'lcx';
# 授权
GRANT ALL PRIVILEGES ON j2eework_db.* TO 'lcx'@'localhost';
# 一定要刷新权限
FLUSH PRIVILEGES;
```

3）在 UseSC 工程的 pom.xml 里添加 MySQL-JDBC 连接包：

```
<!--mysql 数据库的 jdbc 连接包-->
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <!-- mysql-8 需要 Connector/J 8.0.9 or higher -->
  <version>8.0.11</version>
</dependency>
```

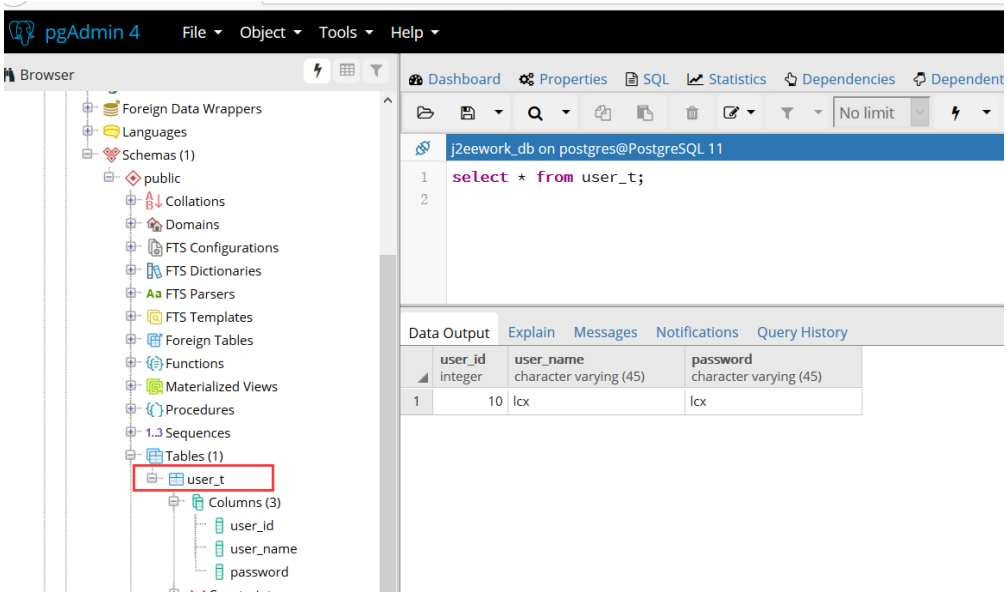
4）在 UseSC 工程的 config 目录下添加 jdbc-mysql.properties 文件，内容如下：

```
# 连接设置
driverClassName=com.mysql.cj.jdbc.Driver
url=jdbc:mysql://127.0.0.1:3306/j2eework_db?
username=lcx
password=lcx

# <!-- 初始化连接 -->
initialSize=10
# 最大连接数量
maxActive=50
# <!-- 最大空闲连接 -->
maxIdle=20
# <!-- 最小空闲连接 -->
```

```
minIdle=5
# <!-- 超时等待时间以毫秒为单位 6000 毫秒/1000 等于 60 秒 -->
maxWait=60000

# JDBC 驱动建立连接时附带的连接属性属性的格式必须为这样：[属性名=property;]
# 注意：“user” 与 “password” 两个属性会被明确地传递，因此这里不需要包含他们。
connectionProperties=autoReconnect=true;autoReconnectForPools=true;useUnicode=true;characterEncoding=utf-8;useSSL=false;serverTimezone=UTC

# 指定由连接池所创建的连接的自动提交（auto-commit）状态。
defaultAutoCommit=true

# driver default 指定由连接池所创建的连接的事务级别（TransactionIsolation）。
#可用值为下列之一：（详情可见 javadoc）NONE,READ_UNCOMMITTED, READ_COMMITTED, REPEATABLE_READ, SERIALIZABLE
defaultTransactionIsolation=READ_UNCOMMITTED

# 表示 DBCP 连接池自动管理应程序中使用完毕的连接
removeAbandonedOnMaintenance=true
removeAbandonedOnBorrow=true
# 表示一个连接在程序中使用完毕后，若在 10 秒之内没有再次使用，则 DBCP 连接池回收该连接
removeAbandonedTimeout=10
```

3.5 PostgreSQL 部分

1）创建数据库 j2eework_db，在该数据库中创建表 user_t，表中创建字段：user_id、user_name、password，如下：



2）创建用户 lcx，并对其授权，SQL 如下：

```
-- 创建用户 lcx
CREATE USER lcx PASSWORD 'lcx';
-- 给用户 lcx 授权
GRANT ALL ON user_t TO lcx;
-- 序列授权
GRANT ALL ON user_t_user_id_seq TO lcx;
```

3）在 UseSC 工程的 pom.xml 里添加 PostgreSQL-JDBC 连接包：

```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>42.2.5</version>
</dependency>
```

4）在 UseSC 工程的 config 目录下添加 jdbc-postgresql.properties 文件，配置和上面的 jdbc-mysql.properties 基本类似，这里只给出变化的部分，如下：

```
# 连接设置
driverClassName=org.postgresql.Driver
url=jdbc:postgresql://127.0.0.1:5432/j2eework_db?
username=lcx
password=lcx
```

3.6 BaseDAO & UserDAO

3.6.1 BaseDAO

在 SimpleController 工程中新建包：southday.j2eework.sc.ustc.controller.dao，在包中创建 BaseDAO 抽象类，内容如下：

```
public abstract class BaseDAO {
    public abstract Connection openDBConnection();
    public abstract boolean closeDBConnection();

    public abstract Object query(String sql);
    public abstract boolean insert(String sql);
    public abstract boolean update(String sql);
    public abstract boolean delete(String sql);
}
```

注：由于使用 DBCP 来管理数据库连接，并且连接属性 driver、url、userName、userPassword 等都放在了 properties 文件中，所以在 BaseDAO 中都不定义这些字段。

3.6.2 UserDAO

在 UseSC 工程中新建包：southday.j2eework.water.ustc.dao，在包中创建 UserDAO 类（单例，继承 BaseDAO），重写相关方法，内容如下：

```java
public class UserDAO extends BaseDAO {
    private static final String PROPERTIES = "jdbc-postgresql.properties";
    private BasicDataSource ds;

    private UserDAO() {
        String propsFilePath = getPathOfJDBCProps();
        try {
            ds = BasicDataSourceFactory.createDataSource(FileUtil.getProperties(propsFilePath));
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    private static class UserDAOHolder {
        private static UserDAO usrDAO = new UserDAO();
    }

    public static UserDAO getUserDAO() {
        return UserDAOHolder.usrDAO;
    }

    @Override
    public Connection openDBConnection() {
        return null;
    }

    @Override
    public boolean closeDBConnection() {
        return false;
    }

    @Override
    public Object query(String sql) {
        Object result = null;
        try {
            Connection conn = ds.getConnection();
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(sql);
            if (rs.next())
                result = rs.getString("password");
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return result;
    }
```

```java
    @Override
    public boolean insert(String sql) {
        try {
            Connection conn = ds.getConnection();
            conn.setAutoCommit(false);
            Statement stmt = conn.createStatement();
            stmt.executeUpdate(sql);
            conn.commit();
            conn.close();
            return true;
        } catch (SQLException e) {
            e.printStackTrace();
            return false;
        }
    }

    @Override
    public boolean update(String sql) {
        // TODO Auto-generated method stub
        return false;
    }

    @Override
    public boolean delete(String sql) {
        // TODO Auto-generated method stub
        return false;
    }

    private String getPathOfJDBCProps() {
        return BaseConfig.getBaseConfig().getFilePathInClassesDIR(PROPERTIES);
    }
}
```

注：由于数据库连接使用 DBCP 来管理，并且考虑到 Connection 的并发安全问题，就不重写 openDBConnection()和 closeDBConnection()方法，而是改为每次要执行 sql 都从连接池中获取一个 Connection，执行完后将连接归还（conn.close()）给 DBCP。

3.7 修改 UserServiceImpl 逻辑

之前的 UserServiceImpl 是使用模拟的数据库 SimpleUserDB 来实现相关逻辑，现在改为使用 UserDAO 来完成与数据库相关的操作。

1）login，构造 sql，根据 user_name 查询 password，如果返回为 null，则返回 false；否则判断参数中的 password 是否和返回的 password 相同，相同返回 true，否则返回 false。

2）register，构造 sql，直接执行 insert 语句，user_id 在数据库中设置了自增，所以只需插入 user_name 和 password 属性值。具体代码如下：

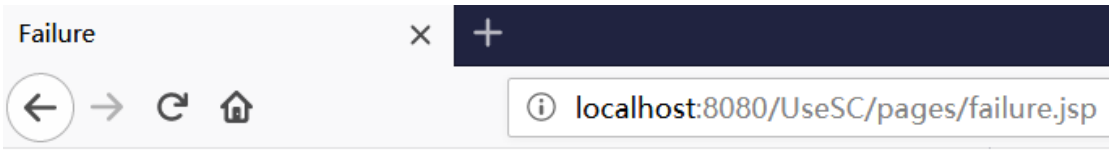
```
public class UserServiceImpl implements UserService {
    private UserDAO userDAO = UserDAO.getUserDAO();

    private UserServiceImpl() {}

    private static class UserServiceImplHolder {
        private static UserService userService = new UserServiceImpl();
    }

    public static UserService getUserService() {
        return UserServiceImplHolder.userService;
    }

    @Override
    public boolean login(String userName, String password) throws Exception {
        if (!CommonUtil.checkParam(userName, password))
            return false;
        String sql = "SELECT password FROM user_t WHERE user_name='" + userName + "'";
        Object res = userDAO.query(sql);
        if (res == null)
            return false;
        return password.equals((String)res);
    }

    @Override
    public boolean register(String userName, String password) throws Exception {
        if (!CommonUtil.checkParam(userName, password))
            return false;
        StringBuilder sb = new StringBuilder();
        sb.append("INSERT INTO user_t(user_name, password) VALUES(");
        sb.append("'" + userName + "',");
        sb.append("'" + password + "')");
        return userDAO.insert(sb.toString());
    }
}
```

3.8 测试结果

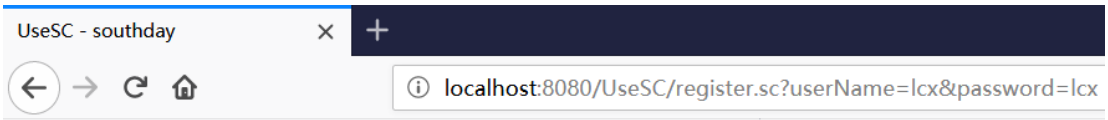
基于 MySQL 还是 PostgreSQL，只需要修改 UserDAO 类中的静态字段 PROPERTIES 就行，可改为：jdbc-mysql.properties 或 jdbc-postgresql.properties。

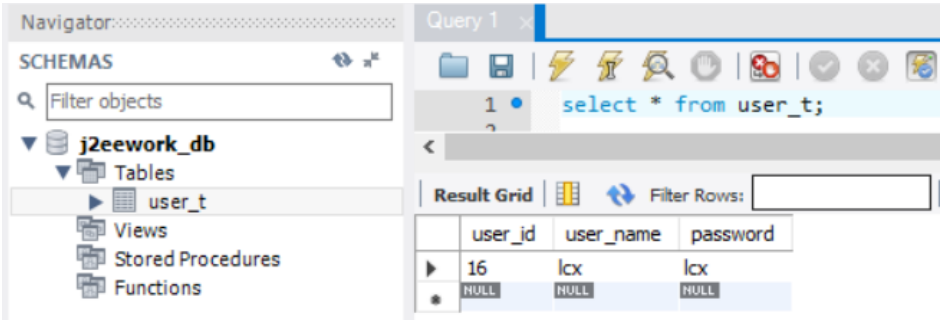
3.8.1 基于 MySQL 测试

1）测试登陆：http://localhost:8080/UseSC/login.sc?userName=lcx&password=lcx，失败，用户账号 lcx 不存在；

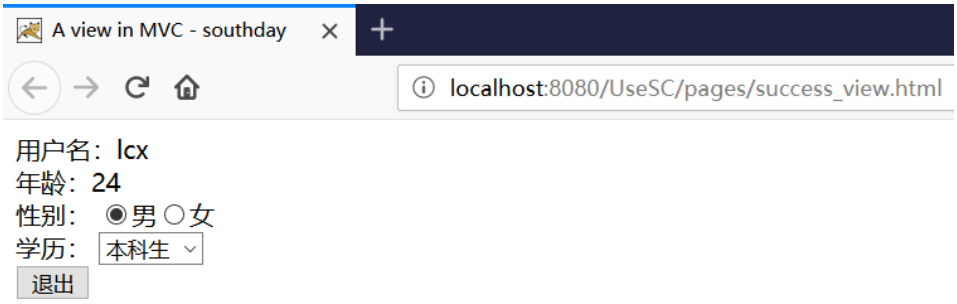


2）测试注册：http://localhost:8080/UseSC/register.sc?userName=lcx&password=lcx，注册成功，跳转到 welcome 页面；

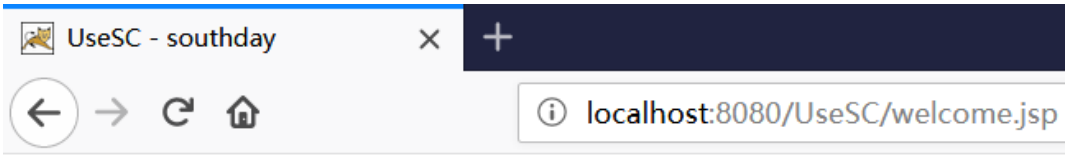




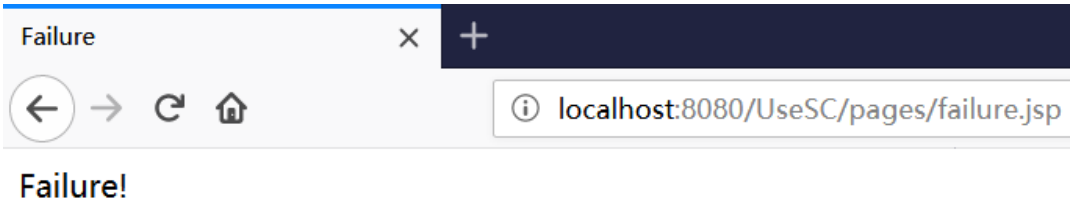
3）注册成功后，用 lcx 账号登陆：http://localhost:8080/UseSC/login.sc?userName=lcx&password=lcx，登陆成功，跳转到 success_view.html 页面；



4）在 success_view.html 页面点击“退出”，返回到首页（welcome.jsp）；



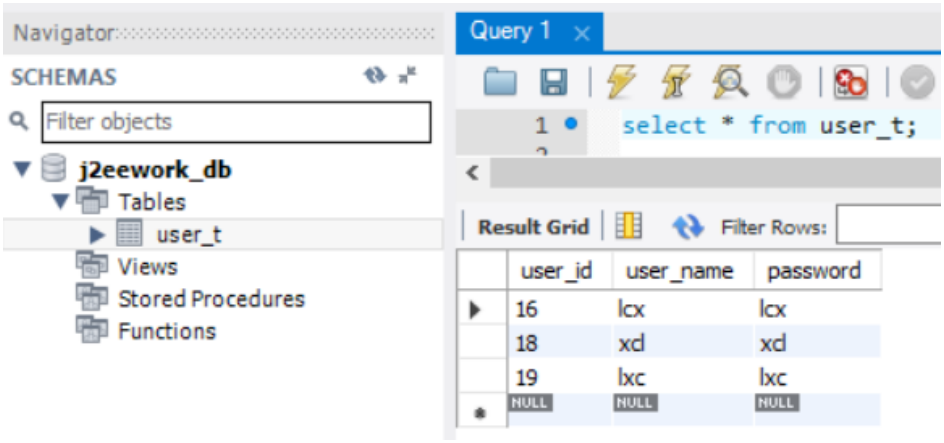
5）注册成功后，使用 lcx 账号登陆，填写错误密码 xcl，http://localhost:8080/UseSC/login.sc?userName=lcx&password=xcl，登陆失败，因为密码错误；



6）使用<userName, password>=<xcl, xcl>, <lxc, lxc>注册账号：

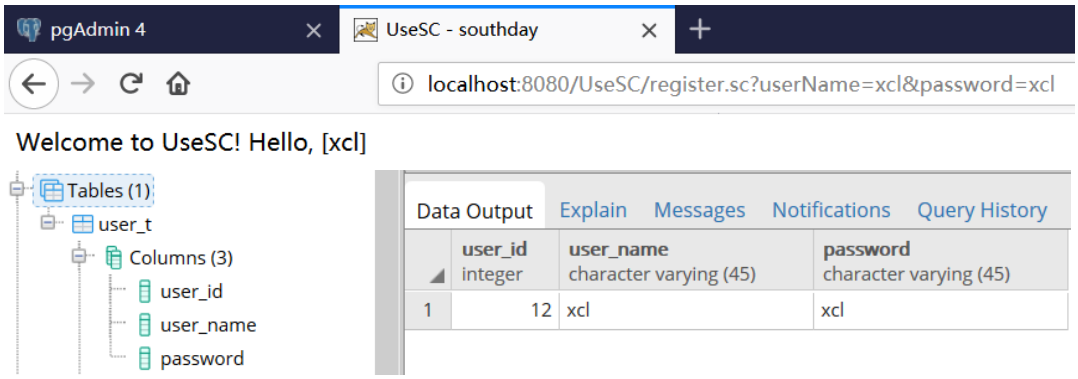
- http://localhost:8080/UseSC/register.sc?userName=xcl&password=xcl
- http://localhost:8080/UseSC/register.sc?userName=lxc&password=lxc

注册成功，数据库 user_t 表显示如下：

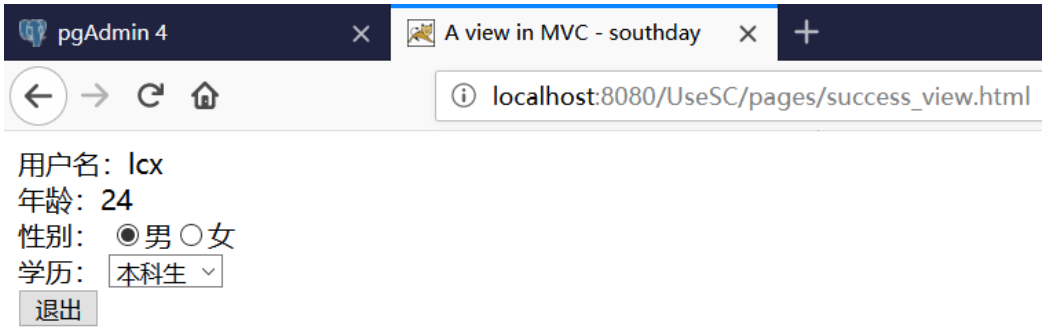


3.8.2 基于 PostgreSQL 测试

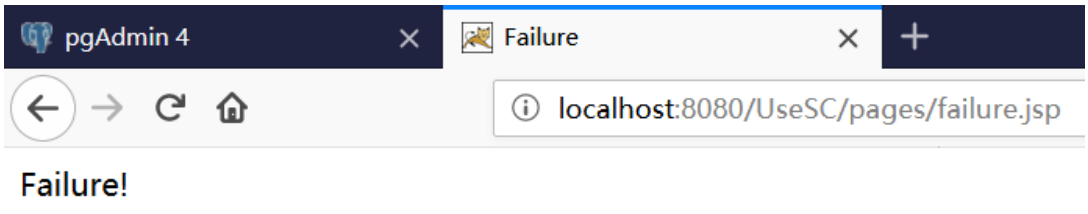
1）测试注册：http://localhost:8080/UseSC/register.sc?userName=xcl&password=xcl，注册成功，跳转到 welcome 页面；



2）注册成功后，用 xcl 账号登陆：http://localhost:8080/UseSC/login.sc?userName=xcl&password=xcl，登陆成功，跳转到 success_view.html 页面；



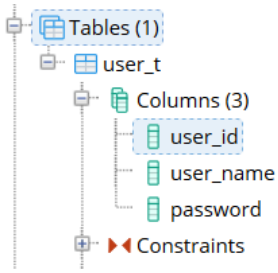
3）注册成功后，使用 xcl 账号登陆，填写错误密码 lcx，http://localhost:8080/UseSC/login.sc?userName=xcl&password=lcx，登陆失败，因为密码错误；



4）使用<userName, password>=<lcx, lcx>, <cxl, cxl>注册账号：

- http://localhost:8080/UseSC/register.sc?userName=lcx&password=lcx
- http://localhost:8080/UseSC/register.sc?userName=cxl&password=cxl

注册成功，数据库 user_t 表显示如下：



Data Output | Explain | Messages | Notifications | Query History

	user_id integer	user_name character varying (45)	password character varying (45)
1	12	xcl	xcl
2	13	lcx	lcx
3	14	cxl	cxl

4. 结论

4.1 总结

通过本次实验，我学习了 DAO 的相关知识，认识并使用了开源数据库连接池 DBCP。不仅如此，还接触了之前没用过的 PostgreSQL，收获很多。在重写 E4 代码时，还遇到了意外的惊喜（4.2.1 中提到）。虽然时间很紧，但还是尽量做到每次实验都有些进步，无论是学习新知识还是温故而知新，厚积而薄发。

4.2 问题及看法

4.2.1 由类加载顺序引起的问题

在写 E5 时，我重写了 E4 中 FileUtil.java 的部分代码，结果运行时报错：

很明显，我的文件路径是拼接的，而前面的值为 null（默认值），也就是说没有初始化。我看了下代码，FileUtil 中的这两行：

写法 1：

```
public class FileUtil {
    private static final BaseConfig baseConfig = BaseConfig.getBaseConfig();
    public static final String CLASSES_PATH = getClassesPath();
    /**
```

我把这两行换了下顺序，改为如下，代码就正常运行了。

写法 2：

```
public class FileUtil {
    public static final String CLASSES_PATH = getClassesPath();
    private static final BaseConfig baseConfig = BaseConfig.getBaseConfig();
    /**
```

为什么会这样呢？原因是我的 BaseConfig 类中也使用到了 FileUtil.CLASS_PATH，代码如下：

```
public class BaseConfig {
    private static final String FILES_LOCATIONS_PROPS_PATH = getPathOfFilesLocationsProps();
    private Map<String, String> props = null;

    private BaseConfig() {
        try {
            props = FileUtil.loadProperties(FILES_LOCATIONS_PROPS_PATH);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
```

```
private static String getPathOfFilesLocationsProps() {
    return FileUtil.CLASSES_PATH + "config/files-locations.properties";
}
```

写法 1：

1）加载 FileUtil 时，会去加载 BaseConfig（因为 baseConfig 定义为静态）；

2）而加载 BaseConfig 时 ，BaseConfig 里面又有个方法：getPathOfFilesLocationsProps()，里面用到了 FileUtil.CLASS_PATH，而此时的 FileUtil.CLASS_PATH 还是默认值：null；

3）导致 BaseConfig 类中的 FILES_LOCATIONS_PROPS_PATH 变量值为：nullconfig\file-locations.properties；

4）此时一读取文件，就发现找不到路径，所以报错！

写法 2：

先加载（并初始化）CLASS_PATH 的值，所以后面加载 BaseConfig 时不会出现 null。

为了避免这类情况的发生，我调整了代码：

1）将方法从 FileUtil 类中抽离出来，放到 BaseConfig.java 中：

```
public String getFilePathInClassesDIR(String fileName) {
    return FileUtil.CLASSES_PATH + getValue(fileName);
}
```

2）然后其他类引用如下：

```
private static String getPathOfControllerXML() {
    return BaseConfig.getBaseConfig().getFilePathInClassesDIR("controller.xml");
}
```

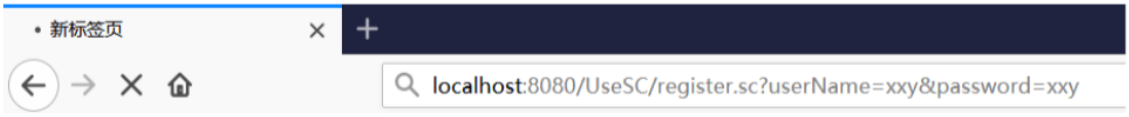
3）这样一来，FileUtil 中就不用引入 BaseConfig 类，职责也比较明确！

4.2.2 未归还连接引起的请求阻塞问题

之前在查 DBCP 的使用教程时，看到 DBCP 是可以自动管理连接的，并且它里面的类多数都是实现 AutoCloseable 接口，即：自动释放连接。所以我写代码时，获取到连接后就没有 conn.close()。

测试的时候发现：连续注册了 10 个用户后，连接池中 Connection 资源用光了，请求一直阻塞在 ds.getConnection()，页面无响应。这是控制台输出的日志：

```
[register-Thread-9] start log...
[register-Thread-9] end log.
insert: begin...ds.getConnection()
insert: end.....ds.getConnection()
[register-Thread-10] start log...
[register-Thread-10] end log.
insert: begin...ds.getConnection()
insert: begin...ds.getConnection()
insert: begin...ds.getConnection()
insert: begin...ds.getConnection()
```



原因是我以为使用了 DBCP 连接池就不需要我自己写代码去关闭连接 conn.close()，查了资料后发现：

1）应用程序中主动关闭连接（执行 conn.close()），该连接交回给 DBCP 连接池，由连接池管理该连接；

2）若应用程序不主动关闭，则需要配置 DBCP 自动释放连接，如下：

```
# 表示DBCP连接池自动管理应程序中使用完毕的连接
removeAbandonedOnMaintenance=true
removeAbandonedOnBorrow=true
# 表示一个连接在程序中使用完毕后，若在1秒之内没有再次使用，则DBCP连接池回收该连接
# 通常不会配置为1，这里仅为了测试
removeAbandonedTimeout=1
```

我就是属于既不主动归还连接，也不让 DBCP 自动管理连接的类型，最终导致无法获取到 Connection。我认为，实际写代码中应该尽量用第 1 种（主动归还连接的方式），因为第 2 种自动管理存在不确定性；或者两者结合（双保险），用第 2 种作为保险方法，以防止某些开发者忘记释放连接。

5. 参考文献

[1] JDBC 连接数据库（二）——连接池：https://www.cnblogs.com/xiaotiaosi/p/6398371.html
[2] DBCP 数据库连接池的简单使用：https://www.cnblogs.com/sunseine/p/5947448.html
[3] MySql 数据库连接池专题：https://www.cnblogs.com/aspirant/p/6747238.html
[4] PostgreSQL 数据类型：https://blog.csdn.net/timo1160139211/article/details/78461347
[5] PostgreSQL 如何为主键创建自增序列(Sequences)：
https://blog.csdn.net/timo1160139211/article/details/78191470
[6] PostgreSQL 学习手册(角色和权限)：
https://www.cnblogs.com/stephen-liu74/archive/2012/05/18/2302639.html
[7] PostgreSQL 用户、数据库及表的管理、操作与授权：
https://blog.csdn.net/qq_24879495/article/details/78039712