# 7、书城第八阶段：

## 1、使用 Filter 过滤器拦截/pages/manager/所有内容，实现权限检查

Filter 代码：

```
public class ManagerFilter implements Filter {

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {

    }

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain
filterChain) throws IOException, ServletException {
        HttpServletRequest httpServletRequest = (HttpServletRequest) servletRequest;

        Object user = httpServletRequest.getSession().getAttribute("user");

        if (user == null) {

httpServletRequest.getRequestDispatcher("/pages/user/login.jsp").forward(servletRequest,servletRes
ponse);
        } else {
            filterChain.doFilter(servletRequest,servletResponse);
        }
    }

    @Override
    public void destroy() {

    }
}
```

web.xml 中的配置：

```
<filter>
    <filter-name>ManagerFilter</filter-name>
    <filter-class>com.atguigu.filter.ManagerFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>ManagerFilter</filter-name>
    <url-pattern>/pages/manager/*</url-pattern>
```

```
    <url-pattern>/manager/bookServlet</url-pattern>
</filter-mapping>
```

# 2、ThreadLocal 的使用

ThreadLocal 的作用，它可以解决多线程的数据安全问题。

ThreadLocal 它可以给当前线程关联一个数据（可以是普通变量，可以是对象，也可以是数组，集合）

ThreadLocal 的特点：

1、ThreadLocal 可以为当前线程关联一个数据。（它可以像 Map 一样存取数据，key 为当前线程）

2、每一个 ThreadLocal 对象，只能为当前线程关联一个数据，如果要为当前线程关联多个数据，就需要使用多个 ThreadLocal 对象实例。

3、每个 ThreadLocal 对象实例定义的时候，一般都是 static 类型

4、ThreadLocal 中保存数据，在线程销毁后。会由 JVM 虚拟自动释放。

测试类：

```
public class OrderService {

    public void createOrder(){
        String name = Thread.currentThread().getName();
        System.out.println("OrderService 当前线程[" + name + "]中保存的数据是：" +
ThreadLocalTest.threadLocal.get());
        new OrderDao().saveOrder();
    }

}

public class OrderDao {

    public void saveOrder(){
        String name = Thread.currentThread().getName();
        System.out.println("OrderDao 当前线程[" + name + "]中保存的数据是：" +
ThreadLocalTest.threadLocal.get());
    }

}

public class ThreadLocalTest {

//    public static Map<String,Object> data = new Hashtable<String,Object>();
    public static ThreadLocal<Object> threadLocal = new ThreadLocal<Object>();

    private static Random random = new Random();
```

```
    public static class Task implements  Runnable {
        @Override
        public void run() {
            // 在 Run 方法中，随机生成一个变量（线程要关联的数据），然后以当前线程名为 key 保存到 map 中
            Integer i = random.nextInt(1000);
            // 获取当前线程名
            String name = Thread.currentThread().getName();
            System.out.println("线程["+name+"]生成的随机数是：" + i);
//            data.put(name,i);
            threadLocal.set(i);

            try {
                Thread.sleep(3000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            new OrderService().createOrder();

            // 在 Run 方法结束之前，以当前线程名获取出数据并打印。查看是否可以取出操作
//            Object o = data.get(name);
            Object o = threadLocal.get();
            System.out.println("在线程["+name+"]快结束时取出关联的数据是：" + o);
        }
    }
    public static void main(String[] args) {
        for (int i = 0; i < 3; i++){
            new Thread(new Task()).start();
        }
    }
}
```

# 3、使用 Filter 和 ThreadLocal 组合管理事务

## 3.1、使用 ThreadLocal 来确保所有 dao 操作都在同一个 Connection 连接对象中完成

原理分析图：


JdbcUtils 工具类的修改：

```
public class JdbcUtils {

    private static DruidDataSource dataSource;
    private static ThreadLocal<Connection> conns = new ThreadLocal<Connection>();

    static {
        try {
            Properties properties = new Properties();
            // 读取 jdbc.properties 属性配置文件
            InputStream inputStream =
JdbcUtils.class.getClassLoader().getResourceAsStream("jdbc.properties");
            // 从流中加载数据
            properties.load(inputStream);
            // 创建 数据库连接 池
            dataSource = (DruidDataSource) DruidDataSourceFactory.createDataSource(properties);

        } catch (Exception e) {
            e.printStackTrace();
        }

    }

    /**
     * 获取数据库连接池中的连接
     * @return 如果返回 null,说明获取连接失败<br/>有值就是获取连接成功
     */
    public static Connection getConnection(){
        Connection conn = conns.get();
```

```
        if (conn == null) {
            try {
                conn = dataSource.getConnection();//从数据库连接池中获取连接
                conns.set(conn); // 保存到 ThreadLocal 对象中，供后面的 jdbc 操作使用
                conn.setAutoCommit(false); // 设置为手动管理事务
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
        return conn;
    }

    /**
     * 提交事务，并关闭释放连接
     */
    public static void commitAndClose(){
        Connection connection = conns.get();
        if (connection != null) { // 如果不等于 null，说明 之前使用过连接，操作过数据库
            try {
                connection.commit(); // 提交 事务
            } catch (SQLException e) {
                e.printStackTrace();
            } finally {
                try {
                    connection.close(); // 关闭连接，资源资源
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
        }
        // 一定要执行 remove 操作，否则就会出错。（因为 Tomcat 服务器底层使用了线程池技术）
        conns.remove();
    }

    /**
     * 回滚事务，并关闭释放连接
     */
    public static void rollbackAndClose(){
        Connection connection = conns.get();
        if (connection != null) { // 如果不等于 null，说明 之前使用过连接，操作过数据库
            try {
                connection.rollback();//回滚事务
            } catch (SQLException e) {
                e.printStackTrace();
            } finally {
                try {
                    connection.close(); // 关闭连接，资源资源
                } catch (SQLException e) {
                    e.printStackTrace();
```

```
                }
            }
        }
        // 一定要执行remove操作，否则就会出错。（因为Tomcat服务器底层使用了线程池技术）
        conns.remove();
    }


    /**
     * 关闭连接，放回数据库连接池
     * @param conn

    public static void close(Connection conn){
        if (conn != null) {
            try {
                conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    } */

}
```

修改 BaseDao

```
public abstract class BaseDao {

    //使用DbUtils操作数据库
    private QueryRunner queryRunner = new QueryRunner();

    /**
     * update() 方法用来执行：Insert\Update\Delete语句
     *
     * @return 如果返回-1,说明执行失败<br/>返回其他表示影响的行数
     */
    public int update(String sql, Object... args) {

        System.out.println(" BaseDao 程序在[" +Thread.currentThread().getName() + "]中");

        Connection connection = JdbcUtils.getConnection();
        try {
            return queryRunner.update(connection, sql, args);
        } catch (SQLException e) {
            e.printStackTrace();
            throw new RuntimeException(e);
        }
    }

    /**
```

```
     * 查询返回一个 javaBean 的 sql 语句
     *
     * @param type 返回的对象类型
     * @param sql  执行的 sql 语句
     * @param args sql 对应的参数值
     * @param <T>  返回的类型的泛型
     * @return
     */
    public <T> T queryForOne(Class<T> type, String sql, Object... args) {
        Connection con = JdbcUtils.getConnection();
        try {
            return queryRunner.query(con, sql, new BeanHandler<T>(type), args);
        } catch (SQLException e) {
            e.printStackTrace();
            throw new RuntimeException(e);
        }
    }

    /**
     * 查询返回多个 javaBean 的 sql 语句
     *
     * @param type 返回的对象类型
     * @param sql  执行的 sql 语句
     * @param args sql 对应的参数值
     * @param <T>  返回的类型的泛型
     * @return
     */
    public <T> List<T> queryForList(Class<T> type, String sql, Object... args) {
        Connection con = JdbcUtils.getConnection();
        try {
            return queryRunner.query(con, sql, new BeanListHandler<T>(type), args);
        } catch (SQLException e) {
            e.printStackTrace();
            throw new RuntimeException(e);
        }
    }

    /**
     * 执行返回一行一列的 sql 语句
     * @param sql   执行的 sql 语句
     * @param args  sql 对应的参数值
     * @return
     */
    public Object queryForSingleValue(String sql, Object... args){

        Connection conn = JdbcUtils.getConnection();

        try {
            return queryRunner.query(conn, sql, new ScalarHandler(), args);
```

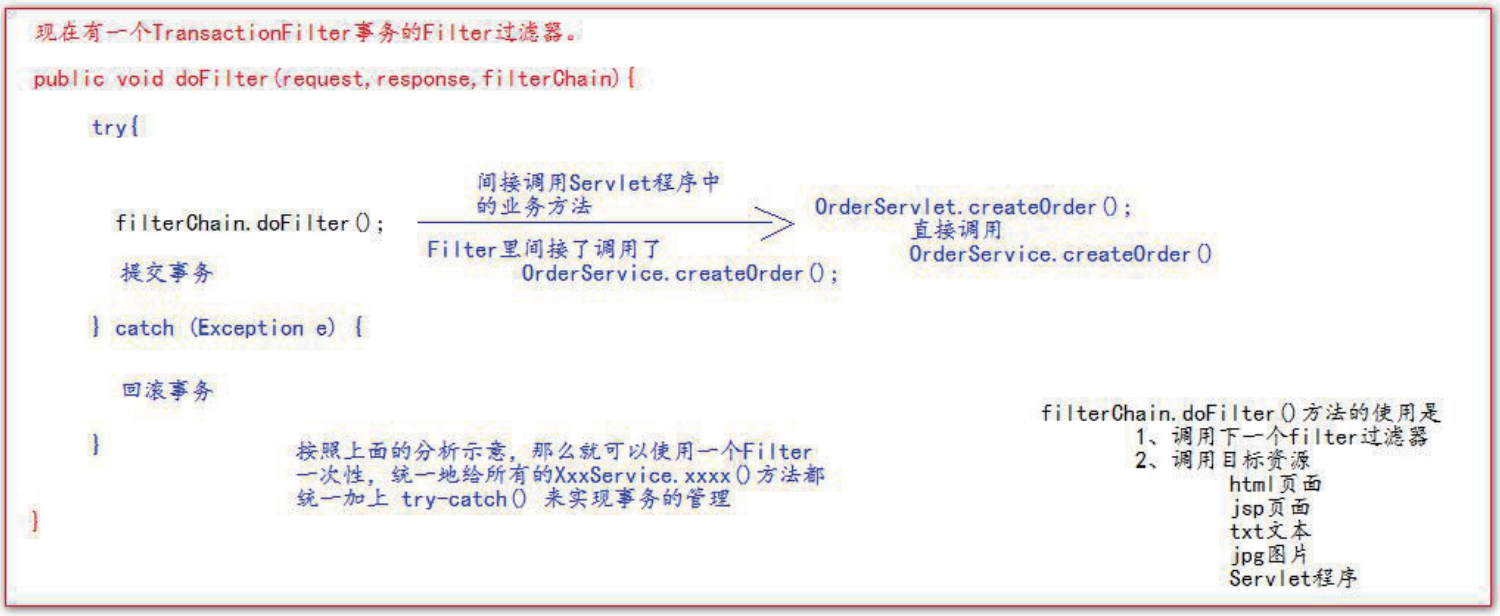
```
        } catch (SQLException e) {
            e.printStackTrace();
            throw new RuntimeException(e);
        }

    }

}
```

## 3.2、使用 Filter 过滤器统一给所有的 Service 方法都加上 try-catch。来进行实现的管理。

原理分析图：




Filter 类代码：

```
public class TransactionFilter implements Filter {

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain
filterChain) throws IOException, ServletException {
        try {
            filterChain.doFilter(servletRequest,servletResponse);
            JdbcUtils.commitAndClose();// 提交事务
        } catch (Exception e) {
            JdbcUtils.rollbackAndClose();//回滚事务
            e.printStackTrace();
        }
    }
```

```
}
```

在 web.xml 中的配置：

```
<filter>
    <filter-name>TransactionFilter</filter-name>
    <filter-class>com.atguigu.filter.TransactionFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>TransactionFilter</filter-name>
    <!-- /* 表示当前工程下所有请求 -->
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

一定要记得把 BaseServlet 中的异常往外抛给 Filter 过滤器

```
public abstract class BaseServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException,
IOException {
        doPost(req, resp);
    }

    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException,
IOException {
        // 解决post请求中文乱码问题
        // 一定要在获取请求参数之前调用才有效
        req.setCharacterEncoding("UTF-8");

        String action = req.getParameter("action");
        try {
            // 获取action业务鉴别字符串，获取相应的业务 方法反射对象
            Method method = this.getClass().getDeclaredMethod(action, HttpServletRequest.class,
HttpServletResponse.class);
//            System.out.println(method);
            // 调用目标业务 方法
            method.invoke(this, req, resp);
        } catch (Exception e) {
            e.printStackTrace();
            throw new RuntimeException(e);// 把异常抛给 Filter 过滤器
        }
    }

}
```

## 3.3、将所有异常都统一交给 Tomcat，让 Tomcat 展示友好的错误信息页面。

在 web.xml 中我们可以通过错误页面配置来进行管理。

```
<!--error-page 标签配置，服务器出错之后，自动跳转的页面-->
<error-page>
    <!--error-code 是错误类型-->
    <error-code>500</error-code>
    <!--location 标签表示。要跳转去的页面路径-->
    <location>/pages/error/error500.jsp</location>
</error-page>

<!--error-page 标签配置，服务器出错之后，自动跳转的页面-->
<error-page>
    <!--error-code 是错误类型-->
    <error-code>404</error-code>
    <!--location 标签表示。要跳转去的页面路径-->
    <location>/pages/error/error404.jsp</location>
</error-page>
```