（1）什么叫MVC模式，试以第九次作业为例，分析MyShop当中的MVC实现，并说说这种模式的优点。

MVC模式，即Model-View-Controller模式，是一种软件设计模式，用于将应用程序的逻辑层（Model）、用户界面（View）和控制逻辑（Controller）分离。这种模式通过将不同的关注点分离，可以提高代码的可维护性和可扩展性。

以第九次作业的一部分代码为例：

Model（模型）

模型负责处理与应用程序的业务逻辑和数据相关的内容。它直接管理数据、逻辑和规则。

在这个例子中，模型部分包括：

- **GoodsSingle.java**：这是一个Java Bean，包含商品的属性和getter/setter方法。

```
public class GoodsSingle implements Serializable {
    private int id;
    private String name;
    private double price;
    private int num;
    private String imgname;
    public GoodsSingle() {}
    // getters and setters
}
```

- **MyTools.java**：这是一个工具类，用于处理商品数据的加载和保存。

```
public class MyTools {
    public static String path = "";
    public static void setPath(String path) {
        MyTools.path = path;
    }
    public static void saveGoods_txt(ArrayList<GoodsSingle> goods) throws Exception {
        // implementation
    }
    public static ArrayList<GoodsSingle> loadGoods_txt() throws IOException {
        // implementation
    }
    public static void appendGoods_txt(GoodsSingle single) throws Exception {
        // implementation
    }
}
```

View（视图）

视图负责显示数据和与用户交互。它不包含任何业务逻辑，只是简单地将数据呈现给用户。

在这个例子中，视图部分包括：

- **addgoods.jsp**：这是一个JSP页面，用于呈现商品添加表单，并处理用户输入。

```jsp
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>添加商品</title>
    <script>
        function uploadImage() {
            var file = document.getElementById('image').files[0];
            const img = document.getElementById('uploadedImage');
            if (file) {
                const reader = new FileReader();
                reader.onload = function (e) {
                    img.src = e.target.result;
                    img.style.display = 'block';
                };
                reader.readAsDataURL(file);
            }
        }
    </script>
</head>
<body>
<h1>添加商品</h1>
<form action="AddGoodsServlet" method="post" enctype="multipart/form-data">
    <!-- form fields -->
    <input type="submit" value="添加商品">
</form>
</body>
</html>
```

Controller（控制器）

控制器负责响应用户输入，并调用模型和视图来完成用户的请求。它处理应用程序的输入逻辑和流控制。

在这个例子中，控制器部分包括：

- **AddGoodsServlet.java**：这是一个Servlet，处理来自`addgoods.jsp`表单的POST请求，将数据传递给模型，并返回结果给视图。

```java
@WebServlet("/AddGoodsServlet")
@MultipartConfig //告诉容器这个 servlet 能够处理 multipart/form-data 类型的请求
public class AddGoodsServlet extends HttpServlet {
    // implementation
}
```

MVC模式的优点

1. **分离关注点**：将业务逻辑、用户界面和控制逻辑分离，使代码更加清晰和易于维护。
2. **可扩展性**：因为视图和模型是分离的，所以可以轻松地更改视图而不影响模型和控制器，反之亦然。
3. **可测试性**：模型和控制器可以独立于视图进行测试，提高了代码的可测试性。
4. **重用性**：模型可以被多个视图重用，减少了代码重复。

通过使用MVC模式，应用程序的结构更加合理，维护和扩展更加方便，从而提高了代码的质量和开发效率。

（2）转发和重定向有什么区别，它们各自适用于那些典型情景?

转发（Forward）和重定向（Redirect）是处理HTTP请求的重要机制，它们在实现上有明显的区别，并适用于不同的典型情景。

转发（Forward）

定义

转发是**服务器端**操作，它将请求从一个Servlet或JSP页面转发到另一个Servlet或JSP页面。转发后，**浏览器地址栏不会发生变化**，用户感知不到这次请求的转发。**转发过程中，始终是同一个请求**，原本作用域为request的属性在转发后仍然可以调用，因为仍然是同一个请求。

实现方式

```
RequestDispatcher dispatcher = request.getRequestDispatcher("targetPage.jsp");
dispatcher.forward(request, response);
```

适用情景

1. 在同一个请求内进行操作： 适用于在服务器端不同资源之间**共享请求和响应对象**（如数据），如在表单验证失败后转发回表单页面并显示错误信息。
2. 避免浏览器刷新问题： 由于浏览器**地址栏不变**，用户刷新页面不会重新提交数据，适合处理表单数据提交的结果展示。
3. 请求处理逻辑内部的跳转： 多个Servlet或JSP需要协作处理一个请求时，可以使用转发来保持**请求对象的一致性。**

重定向（Redirect）

定义

重定向是**客户端**操作，服务器向客户端发送一个HTTP状态码302及新的URL，客户端（浏览器）接收到后，会**重新发起一个新的请求**到新的URL。**浏览器地址栏会发生变化**，用户可以看到URL的变化。**重定向后属于新的独立请求**，原本作用域为request的属性在重定向后无法访问，因为已经是一个新的请求。

实现方式

```
response.sendRedirect("targetPage.jsp");
```

适用情景

1. 客户端新请求： 适用于需要客户端发起一个**全新的请求**时，如用户登录后重定向到首页。
2. 避免重复提交表单： 在表单提交后，使用重定向来防止用户刷新页面导致表单重复提交。
3. 跨域重定向： 可以重定向到**不同的服务器**或域名下的资源，这**在转发中是无法实现的。**

总结

- **转发（Forward）：** 服务器内部跳转，**请求和响应对象共享**，**浏览器地址栏不变。**适用于内部页面间的数据传递和显示。
- **重定向（Redirect）：** 客户端重新请求，地址栏变化，**不能共享请求和响应对象。**适用于提交表单后的跳转和**跨域访问。**

（3）JSP中include指令与include动作在实现上有什么区别，请举例说明。

在JSP中，`include`指令和`include`动作都是用于将一个文件包含到当前页面中，但它们在实现方式和应用场景上有一些重要的区别。

1. include 指令 (`<%@ include %>`)

实现方式

- **静态包含**：`include`指令在页面被翻译和编译前将包含的文件内容**直接嵌入**到包含它的JSP页面中。这意味着包含的文件在编译时就已经被处理。这点和一些编程语言（如C、C++）相似，相当于直接把内容原封不动嵌入。
- 语法：`<%@ include file="relativeURL" %>`

示例

```
<%@ include file="header.jsp" %>
<html>
<body>
    <h1>Main Content</h1>
</body>
</html>
```

在这个例子中，`header.jsp`的内容会在编译阶段被**静态包含**到主JSP页面中。如果`header.jsp`包含了一些**静态**的HTML或者脚本，它们会在主页面中生成**静态内容。**

优点

功能强大，所包含的代码可以含有总体上影响主页面的JSP构造，比如属性、方法的定义和文档类型的设定。

缺点

被包含的页面发生更改，就要对应更改主页面（并重新编译）。

2. include 动作（`<jsp:include>`）

实现方式

- **动态包含**：`include`动作在页面执行时**动态地**将包含的文件插入到当前页面。这意味着**每次请求**都会重新处理包含的文件内容。
- 语法：`<jsp:include page="relativeURL" flush="true" />`

示例

```
<jsp:include page="header.jsp" flush="true" />
<html>
<body>
    <h1>Main Content</h1>
</body>
</html>
```

在这个例子中，`header.jsp`的内容会在运行时被包含进来。如果`header.jsp`包含一些动态生成的内容，例如数据库查询结果，它们会在每次请求时生成最新的内容并插入到主页面中。

主要区别总结

- 处理时机：
  - `include`指令：在翻译和编译时**静态**包含文件内容。
  - `include`动作：在运行时**动态**包含文件内容。
- 应用场景：
  - `include`指令：适合包含**静态**资源（如页眉、页脚等不常变动的部分）。
  - `include`动作：适合包含**动态**资源（如需要每次请求都更新的内容）。
- 影响范围：
  - `include`指令：包含文件**可以影响主页面的编译**，被包含文件中的声明和指令会在主页面中生效。
  - `include`动作：包含文件的声明和指令**不会影响主页面的编译**，仅在运行时生效。
- 处理效率：
  - `include`指令：在编译时包含文件内容，所以这种方式效率更高，不会在每次请求时重新包含文件。
  - `include`动作：每次请求都会动态包含文件内容，所以这种方式在处理静态文件时效率较低，但在需要动态内容时是必须的。
- 参数传递：
  - 动态包含可以给被包含的页面传递参数。
  - 静态包含不能给被包含的页面传递参数。
- 包含地址：
  - 动态包含的地址可以是变量。

- 静态包含的地址是常量。

详细示例

假设有两个文件，分别是 `header.jsp` 和 `main.jsp`。

header.jsp

```
<% String headerMessage = "Welcome to My Website!"; %>
<h1><%= headerMessage %></h1>
```

使用 include 指令的 main.jsp

```
<%@ include file="header.jsp" %>
<html>
<body>
    <p>This is the main content of the page.</p>
</body>
</html>
```

使用 include 动作的 main.jsp

```
<jsp:include page="header.jsp" flush="true" />
<html>
<body>
    <p>This is the main content of the page.</p>
</body>
</html>
```

- 包含方式：
  - 使用`include`指令时，`header.jsp`的内容在编译时被包含，所以`headerMessage`变量会在主页面中可用。
  - 使用`include`动作时，`header.jsp`的内容在运行时被包含，所以`headerMessage`变量仅在`header.jsp`中可用，主页面中不会有该变量。

备注：当包含的文件是静态文件时，两种包含方式的结果不会有区别，但是效率会有区别。

（4）较为复杂的web应用在实现时通常分为哪三层？每层各自具有什么功能？这种三层模式和MVC模式有什么区别？

在实现较为复杂的Java Web应用时，**通常采用三层架构（Three-Tier Architecture）**。这三层分别是：

1. **表示层（Presentation Layer）**：
   - **功能**：负责与用户进行交互，接收用户输入并呈现处理结果。主要包含用户界面组件，例如HTML、CSS、JavaScript、JSP（Java Server Pages）、Servlets等。

- **示例**：用户通过浏览器访问一个表单页面，输入数据并提交，这些数据会传递给后端进行处理，处理结果再返回给用户显示。

2. **业务逻辑层（Business Logic Layer）**：
   - **功能**：负责处理应用程序的具体业务逻辑。包含业务规则、业务流程和业务数据处理。通常通过Spring等框架来实现。
   - **示例**：从表示层接收到用户请求后，进行各种业务操作，如验证数据、计算、调用其他服务等，然后将处理结果传递给数据访问层或表示层。
3. **数据访问层（Data Access Layer）**：
   - **功能**：负责与数据库进行交互，执行数据的存取操作。通常使用JDBC（Java Database Connectivity）、ORM（Object-Relational Mapping）框架如Hibernate、MyBatis等。
   - **示例**：业务逻辑层需要读取或保存数据时，调用数据访问层的组件，数据访问层执行相应的SQL操作并返回结果。

三层架构与MVC模式的区别

三层架构和**MVC模式**（Model-View-Controller）是两种不同的架构模式，但它们在Web应用开发中可以互相结合使用。

1. **三层架构**：
   - **目标**：主要关注应用程序的分层结构，强调职责分离，便于维护和扩展。
   - **层次**：分为表示层、业务逻辑层和数据访问层。
   - **特点**：每一层都只负责特定的职责，表示层处理用户交互，业务逻辑层处理业务规则，数据访问层处理数据存取。
2. **MVC模式**：
   - **目标**：关注用户界面和业务逻辑的分离，主要用于组织用户界面代码。
   - **组件**：分为模型（Model）、视图（View）、控制器（Controller）。
   - 特点：
     - **Model（模型）**：处理数据和业务逻辑。
     - **View（视图）**：负责显示数据。
     - **Controller（控制器）**：处理用户输入，并更新模型和视图。

区别与结合

- **关注点不同**：三层架构关注整体**应用的分层**，而MVC模式更关注**用户界面层次**的分离。
- **层次关系**：MVC模式中的模型层（Model）可以对应三层架构中的业务逻辑层和数据访问层，视图层（View）对应表示层（Presentation Layer），控制器（Controller）作为协调者可以在表示层和业务逻辑层之间进行交互。
- **结合使用**：在实际应用中，通常会将MVC模式应用于表示层（Presentation Layer），以实现用户界面代码的组织，而在更大的架构设计中，再将表示层、业务逻辑层和数据访问层进行分离，形成一个完整的三层架构。