1 Servlet

1.1 继承关系

`javax.servlet.http.HttpServlet` 抽象类继承自 `javax.servlet.GenericServlet` 抽象类。
`javax.servlet.GenericServlet` 抽象类实现 `javax.servlet.Servlet` 接口。

1.2 相关方法

`javax.servlet.Servlet` 接口

```
void init(ServletConfig var1); // 初始化方法
void service(ServletRequest var1, ServletResponse var2); // 服务方法 抽象方法
void destroy(); // 销毁方法
```

`javax.servlet.GenericServlet` 抽象类

```
void service(ServletRequest var1, ServletResponse var2); // 还是抽象的
```

`javax.servlet.http.HttpServlet` 抽象子类实现了 `service` 方法。

```
protected void service(HttpServletRequest req, HttpServletResponse resp) {
    String method = req.getMethod(); // 获取请求方式
    if (method.equals("GET")) {
        this.doGet(req, resp);
    } else if (method.equals("HEAD")) {
        this.doHead(req, resp);
    } else if (method.equals("POST")) {
        this.doPost(req, resp);
    } else if (method.equals("PUT")) {
        this.doPut(req, resp);
    } else if (method.equals("DELETE")) {
        this.doDelete(req, resp);
    } else if (method.equals("OPTIONS")) {
        this.doOptions(req, resp);
    } else if (method.equals("TRACE")) {
        this.doTrace(req, resp);
    } else {
    }
}
```

从源码中可以看出，`service()` 方法根据请求类型，调用对应的 `doXxx()` 方法。

其中 `doPost()` 源码如下

```
protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
    String protocol = req.getProtocol();
    String msg = lStrings.getString("http.method_post_not_supported");
    if (protocol.endsWith("1.1")) {
        resp.sendError(405, msg);
    } else {
        resp.sendError(400, msg);
    }
}
```

所以如果 `HttpServlet` 的子类没有实现 `doPost()` 方法，执行父类的方法，会报错405。

1.3 生命周期

创建如下 `HttpServlet` 的子类

```
public class Demo01Servlet extends HttpServlet {
    public Demo01Servlet() {
        System.out.println("构造函数...");
    }
    @Override
    public void init() throws ServletException {
        System.out.println("初始化函数...");
    }
    @Override
    public void destroy() {
        System.out.println("销毁函数...");
    }
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        System.out.println("doGet()...");
    }
}
```

在 `web.xml` 中添加如下配置

```
<servlet>
    <servlet-name>Demo01Servlet</servlet-name>
    <servlet-class>com.xxyw.servlets.Demo01Servlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>Demo01Servlet</servlet-name>
    <url-pattern>/demo01</url-pattern>
</servlet-mapping>
```

启动Tomcat后，访问 `http://localhost:8080/pro07/demo01` 。从控制台可知，先执行构造方法，后执行初始化方法，然后执行 `doGet()` 方法 。再次访问，只执行 `doGet()` 方法。关闭Tomcat程序，执行销毁方法。

综上，对于一个 `Servlet` 对象，在第一次访问时执行构造方法和初始化方法，然后调用服务方法。服务器关闭才会执行销毁方法。
可以通过如下方式来设置 Servlet 启动的先后顺序，数字越小，启动越靠前，最小值是0。

```
<servlet>
    <servlet-name>Demo01Servlet</servlet-name>
    <servlet-class>com.xxyw.servlets.Demo01Servlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
```

这样在Tomcat服务器启动时就会调用 `Demo01Servlet` 的构造方法和初始化方法。避免第一次访问等待时间较长。

`Servlet` 对象是单例的，线程不安全的。

1.4 HTTP协议

Http是无状态的。

包含请求和响应两部分。

请求包括三个部分。
（1）请求行，包括请求的方式、请求的URL、请求的协议（一般是HTTP1.1）
（2）请求消息头，包含客户端需要告诉服务器的信息，如浏览器型号、本文、能接收内容的类型、发送内容的类型、内容的长度等。
（3）请求主体，get方法没有请求体，有一个queryString；post方式有请求体，form data；json格式有请求体，request payload。

响应包括三个部分。
（1）响应行，包括协议、响应状态码（200）、响应状态（ok）
（2）响应头，包含服务器信息、发送给浏览器的信息（内容的媒体类型、编码、内容长度等）
（3）响应体，响应的实际内容，比如html代码。

1.5 会话

Http是无状态，服务器不能区分请求是同一个浏览器还是不同的浏览器发送的。为了让服务器能够区别发送请求的浏览器，引入会话。

```
// 获取请求的session，如果是浏览器第一次请求，服务器会创建Session。
HttpSession session = req.getSession();
session.getId(); // 每个浏览器有唯一的一个SessionId
HttpSession session = req.getSession(true); // 和无参方法效果相同
HttpSession session = req.getSession(false); // 如果没有Session，返回null，不创建
session.isNew(); // 判断Session是否是新创建的
session.getMaxInactiveInterval(); // 获取非激活间隔时长，默认1800秒
session.setMaxInactiveInterval(999); // 设置非激活间隔时长
session.invalidate(); // 强制性让会话立即失效
```

客户端第一次发请求给服务器，服务器获取session，获取不到，则创建新的，然后响应给客户端。
下次客户端给服务器发请求时，会把sessionID带给服务器，服务器就能获取到，那么服务器就判断这一次请求和上次某次请求是同一个客户端，从而能够区分开客户端。

session保存作用域是和具体的某一个session对应的

```
session.setAttribute("name", "juyoujing"); // 按键值对保存数据
Object name = session.getAttribute("name"); // 通过键获取值，返回Object类型
session.removeAttribute("name"); // 删除指定键的数据
```

1.6 转发重定向

服务器内部转发，一次响应过程，客户端不知道内部转发了多少次，浏览器地址栏没有变化。

```
req.getRequestDispatcher("demo01").forward(req, resp);
```

客户端重定向，两次请求响应过程，客户端知道重新向demo01发请求，浏览器地址栏有变化。

```
resp.sendRedirect("demo01");
```

1.7 Thymeleaf

视图模板技术，把Java内存的变量加载到html上，称之为渲染。

1.8 保存作用域

4个
（1）page，页面级别，现在几乎不用
（2）request，一次请求响应范围
（3）session，一次会话范围
（4）application，整个应用程序范围

2 项目实战1

2.1 路径

相对路径、绝对路径

```
<base href="http://localhost:8080/pro10/*" />
<link href="css/shopping.css">
<link th:href="@{/css/shopping.css}">
```

base标签的作用是当前页面上的路径都以此为基础，thymeleaf中使用`th:href="@{/}"`实现。

2.2 编辑修改

```
<td><a th:href="@{/edit.do(fid=${fruit.fid})}" th:text="${fruit.fname}">苹果</a>
</td>
```

在index.html中添加`edit.do`的超链接，传入fid参数。

```
String fidStr = req.getParameter("fid");
if (StringUtil.isNotEmpty(fidStr)) {
    Integer fid = Integer.parseInt(fidStr);
    // 从数据库中获取指定 fid 的水果
    Fruit fruit = fruitDAO.getFruitByFid(fid);
    // 保存到request作用域
    req.setAttribute("fruit", fruit);
    // 跳转到编辑页面
    super.processTemplate("edit", req, resp);
}
```

`EditServlet`中的`doGet`方法获取指定fid的水果信息，保存到request作用域，跳转到edit.html。

```
<form method="post" th:object="${fruit}" th:action="@{/update.do}">
```

edit.html中显示文本框，提交后以post请求发给`update.do`。

`UpdateServlet`把用户修改后的数据根据fid更新到数据库，跳转回index.html。

2.3 删除

```
<td><img src="imgs/del.jpg" th:onclick="|delFruit(${fruit.getFid()})|"
class="delImg"/></td>
```

在index.html的删除图像上添加删除触发事件。

```
function delFruit(fid) {
    if (confirm("是否删除该水果")) {
        window.location.href = 'del.do?fid=' + fid;
    }
}
```

根据传入的fid使用get请求删除对应的水果。`DelServlet`根据fid删除数据库中的水果。

2.4 添加

```
<a th:href="@{/add.do}" style="border:0px solid blue;margin-bottom: 4px;">添加库存
信息</a>
```

在index.html中使用get请求。此时`AddServlet`处理请求，跳转到add.html页面。

```
<form th:action="@{/add.do}" method="post">
```

在add.html中使用post请求把表单信息发给`AddServlet`，把水果对象存储到数据库中。

2.5 分页

```
<input type="button" value="首页" class="btn" th:onclick="|page(1)|"
th:disabled="${session.pageNum == 1}"/>
<input type="button" value="上一页" class="btn"
th:onclick="|page(${session.pageNum - 1})|" th:disabled="${session.pageNum ==
1}"/>
<input type="button" value="下一页" class="btn"
th:onclick="|page(${session.pageNum + 1})|" th:disabled="${session.pageNum ==
session.pageCount}"/>
<input type="button" value="尾页" class="btn"
th:onclick="|page(${session.pageCount})|" th:disabled="${session.pageNum ==
session.pageCount}"/>
```

使用disabled标签让按钮不可以点击，避免出现非法页数。

```
function page(pageNum) {
    window.location.href = 'index?pageNum=' + pageNum;
}
```

函数`page()`根据传入的页面编号使用get请求主页面。

```
public List<Fruit> getFruitListByPageKey(String keyword, Integer pageNum) {
    return super.executeQuery("select * from t_fruit where fname like ? or
remark like ? limit ? , 5 ", "%" + keyword + "%", "%" + keyword + "%", (pageNum
- 1) * 5);
}
```

`IndexServlet`中根据页面编号计算出SQL语句中的起始编号，从数据库中读取对应页的数据。

2.6 查找

```
<form th:action="@{/index}" method="post" style="float:left;width:60%;margin-
left: 20%;">
    <input type="hidden" name="oper" value="search"/>
    请输入关键字：<input type="text" name="keyword" th:value="${session.keyword}"/>
    <input type="submit" value="查询" class="btn">
</form>
```

在index.html页面有一个form表单能够把查询关键字和查询操作发给后端。

如果是search操作，`IndexServlet`会把keyword保存到session，下次上下页请求从session读取keyword。

3 MVC

对水果库存管理系统的优化。

3.1 一个Servlet

在fruit.servlets包下有`AddServlet`、`DelServlet`、`IndexServlet`等，每种操作对应一个Servlet，我们可以把这些Servlet合并到`FruitServlet`中，对于Fruit相关的请求`add.do`、`del.do`等都使用`fruit.do?operator=add`这种方法。FruitServlet根据参数`operator`判断是哪种操作，使用switch判断，调用对应的同名方法。

3.2 反射调用请求的方法

如果增加方法，需要增加switch的判断。不用switch，使用反射，找到FruitServlet中同名的方法，使用invoke反射调用同名方法。

3.3 DispatcherServlet

除了Fruit有增删改查，购物车等的Servlet实现都是类似的，可以继续提取，抽象出DispatcherServlet。

对于一个url请求，如`http://localhost:8080/pro15/fruit.do`提取出`fruit`并通过反射获取对应的`FruitServlet`，之后优化成`FruitController`。所以我们需要一个记录fruit和FruitController对应关系的文件，即applicationContext.xml。

```
<?xml version="1.0" encoding="utf-8" ?>
<beans>
    <bean id="fruit" class="com.xxyw.fruit.controllers.FruitController"></bean>
</beans>
```

XML是可扩展的标记语言，超文本标记语言HTML是XML的一个子集。
XML由XML声明、DTD文档类型定义、XML正文三部分组成。

`DispatcherServlet`继承自`ViewBaseServlet`，在init()方法中，读取xml获得请求字符串对应的Controller类的全类名，通过反射获取类的对象，保存在Map中。

```
InputStream inputStream =
getClass().getClassLoader().getResourceAsStream("applicationContext.xml");
DocumentBuilderFactory documentBuilderFactory =
DocumentBuilderFactory.newInstance();
DocumentBuilder documentBuilder = documentBuilderFactory.newDocumentBuilder();
Document document = documentBuilder.parse(inputStream);

// 获取所有的bean节点
NodeList beanNodeList = document.getElementsByTagName("bean");
for (int i = 0; i < beanNodeList.getLength(); i++) {
    Node beanNode = beanNodeList.item(i);
    if (beanNode.getNodeType() == Node.ELEMENT_NODE) {
        Element beanElement = (Element) beanNode;
        String beanId = beanElement.getAttribute("id");
        String className = beanElement.getAttribute("class");
        Class<?> controllerBeanClass = Class.forName(className);
        Object beanObj = controllerBeanClass.newInstance();
        beanMap.put(beanId, beanObj);
    }
}
```

在service()方法中，从请求的url中提取出字符串，从`beanMap`中获得对应的Controller对象，然后通过反射得到执行的方法，进一步获取方法的参数名称，使用`req.getParameter(paraName)`获得url请求中的参数，特判request、response、session参数和Integer类型的参数，使用invoke方法调用，根据返回值（String类型）进行视图跳转。

此时FruitController中操作的方法简化如下，以修改为例

```
private String edit(Integer fid, HttpServletRequest req) {
    if (fid != null) {
        Fruit fruit = fruitDAO.getFruitByFid(fid);
        req.setAttribute("fruit", fruit);
        return "edit";
    }
    return "error";
}
private String update(Integer fid, String fname, Integer price, Integer fcount,
String remark) {
    boolean result = fruitDAO.updateFruit(new Fruit(fid, fname, price, fcount,
remark));
    System.out.println(result ? "添加成功" : "添加失败");
    return "redirect:fruit.do";
}
```

edit()、update()方法一般不需要request、response参数，没有抛出异常，返回值为String，如果以`redirect:`开头，重定向新的请求，没有则跳转到同名的html页面。

参数由DispatcherServlet获取传入，不再需要从request获取，只需要执行对应的数据库操作即可。

3.4 servlet api

3.4.1 初始化参数

GenericServlet有2个init()方法

```
public void init(ServletConfig config) throws ServletException {
    this.config = config;
    this.init();
}

public void init() throws ServletException {
}
```

继承HttpServlet的Servlet可以重写init()方法，在初始化的时候获取初始化参数。

```
public void init() throws ServletException {
    ServletConfig config = getServletConfig();
    String initValue = config.getInitParameter("hello");
    System.out.println("initValue = " + initValue);
}
```

使用方法`getServletConfig()`可以获取ServletConfig对象，调用其`getInitParameter()`方法从初始化参数中获取对应的值。

```
<servlet>
    <servlet-name>Demo01Servlet</servlet-name>
    <servlet-class>com.xxyw.servlets.Demo01Servlet</servlet-class>
    <init-param>
        <param-name>hello</param-name>
        <param-value>world</param-value>
    </init-param>
</servlet>
<servlet-mapping>
    <servlet-name>Demo01Servlet</servlet-name>
    <url-pattern>/demo01</url-pattern>
</servlet-mapping>
```

初始化参数在web.xml中的`<servlet>`标签内的`<init-param>`中设置。

如果是以注解的形式配置servlet

```
@WebServlet(urlPatterns = {"/demo02"},
        initParams = {
                @WebInitParam(name = "hello", value = "world"),
                @WebInitParam(name = "uname", value = "jim")
        })
```

使用如上格式，在`@WebServlet`中的`initParams`参数中设置`@WebInitParam`。

3.4.2 上下文参数

还可以通过ServletContext获取上下文参数。

```
ServletContext servletContext = getServletContext();
String location = servletContext.getInitParameter("contextConfigLocation");
System.out.println("location = " + location);
```

在init()方法中，使用`getServletContext()`方法获取ServletContext对象，调用其`getInitParameter()`方法获取上下文参数的值。

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:applicationContext.xml</param-value>
</context-param>
```

上下文参数在web.xml中的配置如上。
还可以在service()方法中获取。

```
protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
    ServletContext servletContext = req.getServletContext();
    String location = servletContext.getInitParameter("contextConfigLocation");
    System.out.println("location = " + location);

    ServletContext servletContext1 = req.getSession().getServletContext();
    String location1 =
servletContext1.getInitParameter("contextConfigLocation");
    System.out.println("location1 = " + location1);
}
```

通过request或session获取ServletContext对象。

3.5 业务层

MVC，Model模型、View视图、Controller控制器
视图层用于数据展示和用户交互。
控制层能够接收客户端的请求，具体的业务功能需要借助模型组件完成。

模型层有pojo/vo值对象、dao数据访问对象、bo业务对象3种。

数据访问对象DAO是单精度方法或细粒度方法，一个方法只考虑一个操作。
业务对象BO的方法属于业务方法，实际业务比较复杂，一个业务功能可能包括多个DAO方法。

3.6 IOC

耦合/依赖
指软件系统中层与层之间存在依赖，如service层中需要DAO层的实现类，controller层需要service层的实现类。

控制反转IOC
在controller中，我们使用`private FruitService fruitService = new FruitServiceImpl();`创建service对象。如果创建语句出现在controller中的某个方法内部，那么fruitService的作用域（生命周期）是这个方法。如果是controller类的成员变量，fruitService的作用域（生命周期）是这个controller实例。我们通过在applicationContext.xml中定义这个fruitService，通过解析XML，产生fruitService实例，存放在beanMap中，beanMap存放在一个BeanFactory中，转移（改变）了之前service实例、dao实例等的生命周期，控制权从程序员转移到BeanFactory，这个现象称之为控制反转。

依赖注入DI
将controller中的代码`private FruitService fruitService = new FruitServiceImpl();`
改写成`private FruitService fruitService = null;`
在applicationContext.xml中配置

```xml
<?xml version="1.0" encoding="utf-8" ?>
<beans>
    <bean id="fruitDAO" class="com.xxyw.fruit.dao.impl.FruitDAOImpl"/>
    <bean id="fruitService"
class="com.xxyw.fruit.service.impl.FruitServiceImpl">
        <property name="fruitDAO" ref="fruitDAO"/>
    </bean>
    <bean id="fruit" class="com.xxyw.fruit.controllers.FruitController">
        <property name="fruitService" ref="fruitService"/>
    </bean>
</beans>
```

在DispatcherServlet中的init()方法中通过BeanFactory读取配置文件，生成实例，并根据`<property>`标签把实例需要的其他实例通过反射赋值，实现依赖注入。

3.7 Filter

过滤器，实现Filter接口，重写其中的init()、doFilter()、destroy()方法。

可以通过注解`@WebFilter("fruit.do")`配置，也可以通过web.xml配置`<filter>`、`<filter-mapping>`，和servlet类似。也可以使用通配符，如`@WebFilter("*.do")`拦截所有以`.do`结尾的请求。

在doFilter()方法中

```java
public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException, ServletException {
    System.out.println("A");
    filterChain.doFilter(servletRequest, servletResponse);
    System.out.println("A2");
}
```

语句`filterChain.doFilter(servletRequest, servletResponse);`之前的内容在同名的Servlet方法之前执行，之后的方法在同名的Servlet执行完之后执行。

多个过滤器形成过滤器链，如果是注解方式进行配置，拦截顺序按照全类名，如果是xml中配置，按照配置顺序。执行顺序是A、B、C、demo、C2、B2、A2。

可以在filter中配置字符编码。

3.8 事务管理

对于DAO中的一个方法，要么全部执行成功，否则回滚。

类似的，一个service方法中有多个DAO方法，也应当是要么全部执行成功，要么回滚。

创建一个`OpenSessionInViewFilter`过滤器，通过`ThreadLocal`获取数据库的`Connection`对象，如果捕获到异常，回滚。要确保DAO层、Service层、Controller层、DispatcherServlet都是抛出异常，这样Filter中才能捕获异常进行回滚。

```java
try {
    TransactionManager.beginTrans();
    System.out.println("开启事务...");
    filterChain.doFilter(servletRequest, servletResponse);
    TransactionManager.commit();
```

```
        System.out.println("提交事务...");
    } catch (Exception e) {
        e.printStackTrace();
        try {
            TransactionManager.rollback();
            System.out.println("回滚事务...");
        } catch (SQLException ex) {
            ex.printStackTrace();
        }
    }
```

在`OpenSessionInViewFilter`中的`doFilter()`方法中，通过`TransactionManager`实现事务的开启、提交和回滚。

`TransactionManager`中通过`ThreadLocal`获取同一个数据库连接对象`Connection`，确保一次Service方法的多个DAO方法使用的是同一个Connection，这样才能回滚整个Service方法。

3.9 监听器

有8种监听器
ServletContextListener、HttpSessionListener、ServletRequestListener
ServletContextAttributeListener、HttpSessionAttributeListener、ServletRequestAttributeListener
HttpSessionBindingListener、HttpSessionActivationListener

可以把原来在DispatcherServlet中的BeanFactory的构造函数放在ServletContextListener的监听事件中，服务器启动时就会读取配置文件创建容器。

```
public void contextInitialized(ServletContextEvent servletContextEvent) {
    ServletContext application = servletContextEvent.getServletContext();
    String path = application.getInitParameter("contextConfigLocation");
    BeanFactory beanFactory = new ClassPathXmlApplicationContext(path);
    application.setAttribute("beanFactory", beanFactory);
}
```

ServletContextListener的子类ContextLoaderListener中的contextInitialized如上，监听到服务器启动就会从web.xml中读取初始化参数配置文件地址，然后读取配置文件创建容器。

```
<listener>
    <listener-class>com.xxyw.myssm.listeners.ContextLoaderListener</listener-
class>
</listener>

<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>applicationContext.xml</param-value>
</context-param>
```

web.xml中的Listener和配置文件地址的初始化参数如上。Listener还可以使用注解配置。

4 Cookie

Cookie保存在客户端（浏览器），可以设置有效时长。应用有记住用户名密码等。

```
Cookie cookie = new Cookie("uname", "jim");
response.addCookie(cookie);
cookie.setMaxAge(60 * 60 * 24 * 10);
```

已上代码就是创建一个Cookie并通过响应返回给浏览器，设置有效期10天。

5 Kaptcha

用于生成图片验证码。

使用步骤
（1）添加jar，在web.xml中配置相关属性
（2）在html上使用 `url-pattern` ，在后端通过Session的 `KAPTCHA_SESSION_KEY` 获取正确的验证码，和前端发送过来的用户输入的比较，进行后续处理。

在web.xml中的配置如下

```
<servlet>
    <servlet-name>KaptchaServlet</servlet-name>
    <servlet-class>com.google.code.kaptcha.servlet.KaptchaServlet</servlet-
class>
    <init-param>
        <param-name>kaptcha.border.color</param-name>
        <param-value>red</param-value>
    </init-param>
    <init-param>
        <param-name>kaptcha.textproducer.char.string</param-name>
        <param-value>abcdefg</param-value>
    </init-param>
    <init-param>
        <param-name>kaptcha.noise.impl</param-name>
        <param-value>com.google.code.kaptcha.impl.NoNoise</param-value>
    </init-param>
    <init-param>
        <param-name>kaptcha.image.width</param-name>
        <param-value>120</param-value>
    </init-param>
    <init-param>
        <param-name>kaptcha.image.height</param-name>
        <param-value>40</param-value>
    </init-param>
    <init-param>
        <param-name>kaptcha.textproducer.font.size</param-name>
        <param-value>28</param-value>
    </init-param>
</servlet>
<servlet-mapping>
    <servlet-name>KaptchaServlet</servlet-name>
    <url-pattern>/kaptch.jpg</url-pattern>
</servlet-mapping>
```

可以在 `init-param` 配置边框颜色、字符集范围、图片是否有噪声、图片长宽、字体大小等。
其中的 `url-pattern` 就是html中引用图片的地址。

```
<img th:src="@{/kaptch.jpg}" alt="" />
```

如上，在regist.html中引用。

```
Object kaptchaObj = session.getAttribute("KAPTCHA_SESSION_KEY");
if (kaptchaObj == null || !kaptchaObj.equals(verifyCode)) {}
```

在Controller中的regist方法中通过Session获取真实值，和前端传来的 `verifyCode` 比较再进行相关处理。

6 前后端分离

使用vue和axios实现前后端分析，异步发送请求，后端的数据以json格式发送给前端。

以修改书城项目的购物车页面为例。

```
window.onload = function () {
    var vue = new Vue({
        el: "#cart_div",
        data: {
            cart: {}
        },
        methods: {
            getCart: function () {
                axios({
                    method: "POST",
                    url: "cart.do",
                    params: {
                        operate: "getCart"
                    }
                }).then(function (value) {
                    vue.cart = value.data
                }).catch(function (reason) {

                })
            },
            editBuyCount: function (cartItemId, buyCount) {
                axios({
                    method: "POST",
                    url: "cart.do",
                    params: {
                        operate: "editBuyCount",
                        cartItemId: cartItemId,
                        buyCount: buyCount
                    }
                }).then(function (value) {
                    vue.getCart()
                }).catch(function (reason) {

                })
            }
        },
        mounted: function () {
            this.getCart()
```

```
        }
    })
}
```

对于id为 `cart_div` 的购物车div，在数据装载mounted时，调用 `getCart()` 方法向服务器端发送异步请求获取购物车信息，接收到数据后赋值给data中的cart变量。在前端使用v-bind和{{}}把data中的cart的值显示在html上。

在购物车页面上增加减少商品数量的时候，调用Vue对象methods中的 `editBuyCount()` 方法，该方法向服务器发送异步请求修改购物车项的数量，服务器端可以不返回内容，浏览器只需要再调用一次 `getCart()` 方法重新获取购物车信息即可。