JavaWeb后端

经过前面的学习，现在终于可以正式进入到后端的学习当中，不过，我们还是需要再系统地讲解一下HTTP通信基础知识，它是我们学习JavaWeb的基础知识，我们之前已经学习过TCP通信，而HTTP实际上是基于TCP协议之上的应用层协议，因此理解它并不难理解。

打好基础是关键！为什么要去花费时间来讲解计算机网络基础，我们学习一门技术，如果仅仅是知道如何使用却不知道其原理，那么就成了彻头彻尾的“码农”，只知道搬运代码实现功能，却不知道这行代码的执行流程，在遇到一些问题的时候就不知道如何解决，无论是知识层面还是应用层面都得不到提升。

无论怎么样，我们都要明确，我们学习JavaWeb的最终目的是为了搭建一个网站，并且让用户能访问我们的网站并在我们的网站上做一些事情。

计算机网络基础

在计算机网络（谢希仁 第七版 第264页）中，是这样描述万维网的：

> 万维网（World Wide Web）并非是某种特殊的计算机网络，万维网是一个大规模的联机式信息储藏所，英文简称`web`，万维网用**链接**的方法，能够非常方便地从互联网上的一个站点访问另一个站点，从而主动地按需求获取丰富的信息。

这句话说的非常官方，但是也蕴藏着许多的信息，首先它指明，我们的互联网上存在许许多多的服务器，而我们通过访问这些服务器就能快速获取服务器为我们提供的信息（比如打开百度就能展示搜索、打开小破站能刷视频、打开微博能查看实时热点）而这些服务器就是由不同的公司在运营。

其次，我们通过浏览器，只需要输入对应的网址或是点击页面中的一个链接，就能够快速地跳转到另一个页面，从而按我们的意愿来访问服务器。

而书中是这样描述万维网的工作方式：

> 万维网以客户服务器的方式工作，浏览器就是安装在用户主机上的万维网客户程序，万维网文档所驻留的主机则运行服务器程序，因此这台主机也称为万维网服务器。**客户程序向服务器程序发出请求，服务器程序向客户程序送回客户所要的万维网文档**，在一个客户程序主窗口上显示出的万维网文档称为页面。

上面提到的客户程序其实就是我们电脑上安装的浏览器，而服务端就是我们即将要去学习的Web服务器，也就是说，我们要明白如何搭建一个Web服务器并向用户发送我们提供的Web页面，在浏览器中显示的，一般就是HTML文档被解析后的样子。

那么，我们的服务器可能不止一个页面，可能会有很多个页面，那么客户端如何知道该去访问哪个服务器的哪个页面呢？这个时候就需要用到`URL`统一资源定位符。互联网上所有的资源，都有一个唯一确定的URL，比如`http://www.baidu.com`

URL的格式为：

> <协议>://<主机>:<端口>/<路径>
>
> 协议是指采用什么协议来访问服务器，不同的协议决定了服务器返回信息的格式，我们一般使用HTTP协议。
>
> 主机可以是一个域名，也可以是一个IP地址（实际上域名最后会被解析为IP地址进行访问）
>
> 端口是当前服务器上Web应用程序开启的端口，我们前面学习TCP通信的时候已经介绍过了，HTTP协议默认使用80端口，因此有时候可以省略。

> 路径就是我们希望去访问此服务器上的某个文件，不同的路径代表访问不同的资源。

我们接着来了解一下什么是HTTP协议：

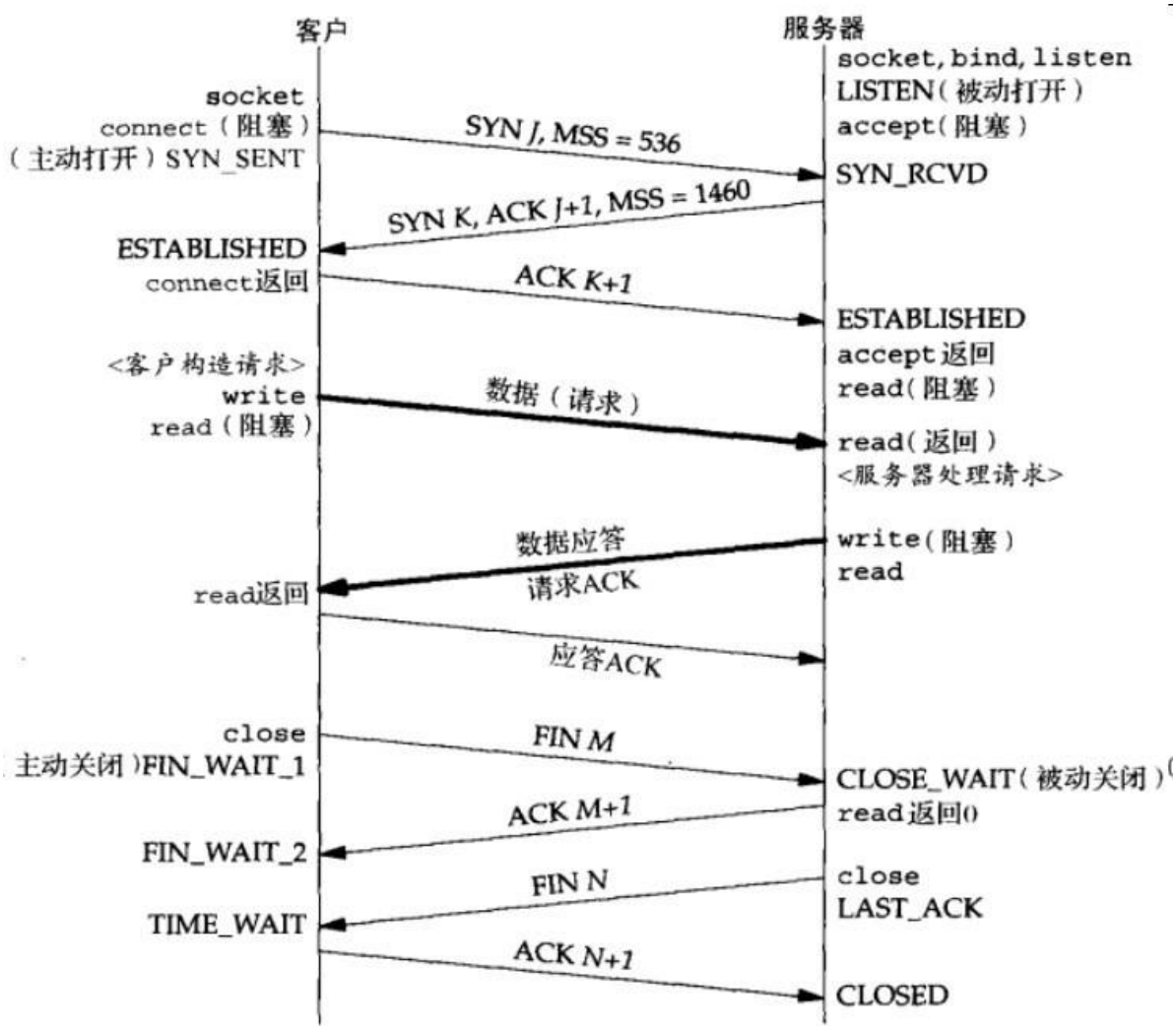
> HTTP是面向事务的应用层协议，它是万维网上能够可靠交换文件的重要基础。HTTP不仅传送完成超文本跳转所需的必须信息，而且也传送任何可从互联网上得到的信息，如文本、超文本、声音和图像。

实际上我们之前访问百度、访问自己的网站，所有的传输都是以HTTP作为协议进行的。

我们来看看HTTP的传输原理：

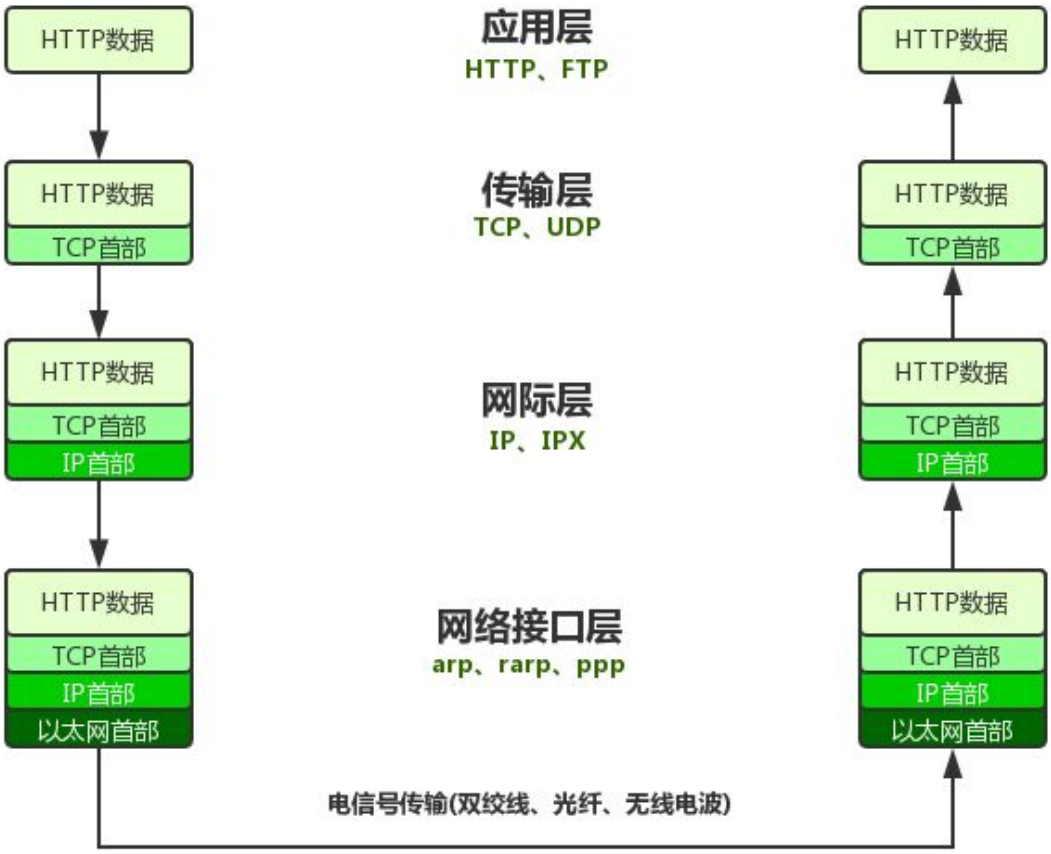
> HTTP使用了面向连接的TCP作为运输层协议，保证了数据的可靠传输。HTTP不必考虑数据在传输过程中被丢弃后又怎样被重传。但是HTTP协议本身是无连接的。也就是说，HTTP虽然使用了TCP连接，但是通信的双方在交换HTTP报文之前不需要先建立HTTP连接。1997年以前使用的是HTTP/1.0协议，之后就是HTTP/1.1协议了。

那么既然HTTP是基于TCP进行通信的，我们首先来回顾一下TCP的通信原理：



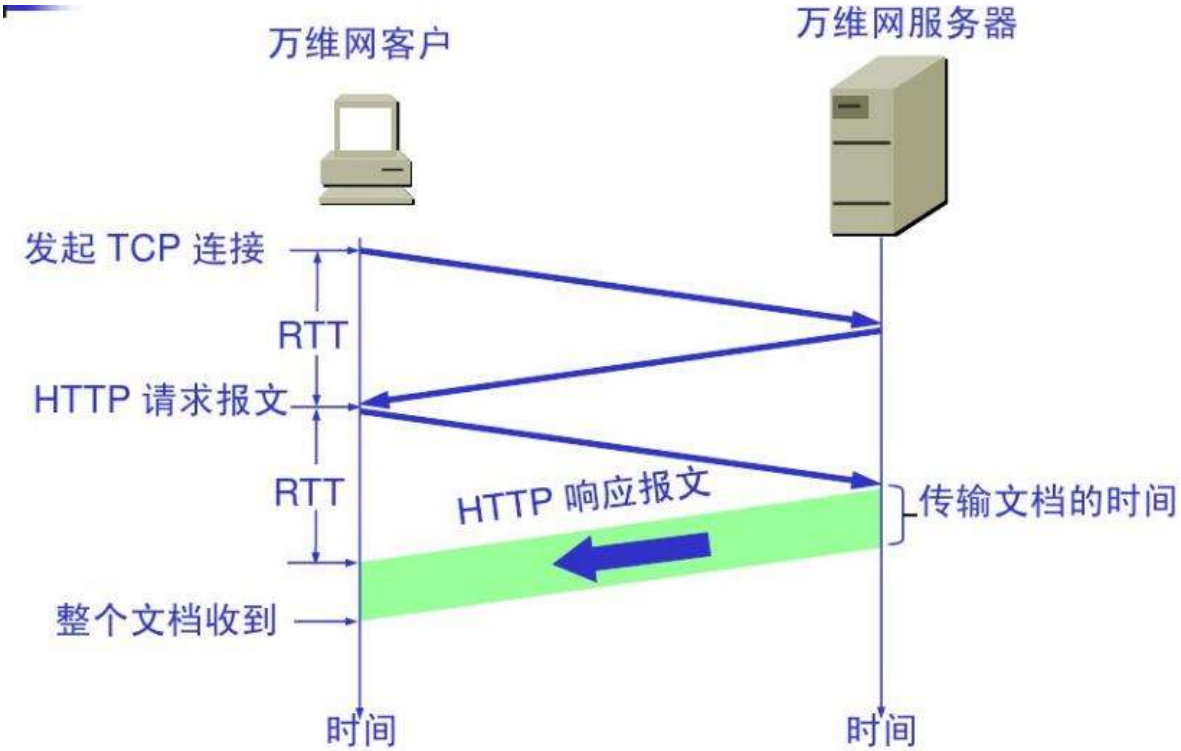
TCP协议实际上是经历了三次握手再进行通信，也就是说保证整个通信是稳定的，才可以进行数据交换，并且在连接已经建立的过程中，双方随时可以互相发送数据，直到有一方主动关闭连接，这时在进行四次挥手，完成整个TCP通信。

而HTTP和TCP并不是一个层次的通信协议，TCP是传输层协议，而HTTP是应用层协议，因此，实际上HTTP的内容会作为TCP协议的报文被封装，并继续向下一层进行传递，而传输到客户端时，会依次进行解包，还原为最开始的HTTP数据。



HTTP使用TCP协议是为了使得数据传输更加可靠，既然它是依靠TCP协议进行数据传输，那么为什么说它本身是无连接的呢？我们来看一下HTTP的传输过程：

> 用户在点击鼠标链接某个万维网文档时，HTTP协议首先要和服务器建立TCP连接。这需要使用三报文握手。当建立TCP连接的三报文握手的前两部分完成后（即经过了一个RTT时间后），万维网客户就把HTTP请求报文作为建立TCP连接的三报文握手中的第三个报文的数据，发送给万维网服务器。服务器收到HTTP请求报文后，就把所请求的文档作为响应报文返回给客户。



因此，我们的浏览器请求一个页面，需要两倍的往返时间。

最后，我们再来了解一下HTTP的报文结构：

由客户端向服务端发送是报文称为请求报文，而服务端返回给客户端的称为响应报文，实际上，整个报文全部是以文本形式发送的，通过使用空格和换行来完成分段。

现在，我们已经了解了HTTP协议的全部基础知识，那么什么是Web服务器呢，实际上，它就是一个软件，但是它已经封装了所有的HTTP协议层面的操作，我们无需关心如何使用HTTP协议通信，而是直接基于服务器软件进行开发，我们只需要关心我们的页面数据如何展示、前后端如何交互即可。

认识Tomcat服务器

Tomcat（汤姆猫）就是一个典型的Web应用服务器软件，通过运行Tomcat服务器，我们就可以快速部署我们的Web项目，并交由Tomcat进行管理，我们只需要直接通过浏览器访问我们的项目即可。

那么首先，我们需要进行一个简单的环境搭建，我们需要在Tomcat官网下载最新的Tomcat服务端程序：https://tomcat.apache.org/download-10.cgi（下载速度可能有点慢）

- 下载：64-bit Windows zip

下载完成后，解压，并放入桌面，接下来需要配置一下环境变量，打开`高级系统设置`，打开`环境变量`，添加一个新的系统变量，变量名称为`JRE_HOME`，填写JDK的安装目录+/jre，比如Zulujdk默认就是：C:\Program Files\Zulu\zulu-8\jre

设置完成后，我们进入tomcat文件夹bin目录下，并在当前位置打开CMD窗口，将startup.sh拖入窗口按回车运行，如果环境变量配置有误，会提示，若没问题，服务器则正常启动。

如果出现乱码，说明编码格式配置有问题，我们修改一下服务器的配置文件，打开`conf`文件夹，找到`logging.properties`文件，这就是日志的配置文件（我们在前面已经给大家讲解过了）将ConsoleHandler的默认编码格式修改为GBK编码格式：

```
java.util.logging.ConsoleHandler.encoding = GBK
```

现在重新启动服务器，就可以正常显示中文了。

服务器启动成功之后，不要关闭，我们打开浏览器，在浏览器中访问：http://localhost:8080/，Tomcat服务器默认是使用8080端口（可以在配置文件中修改），访问成功说明我们的Tomcat环境已经部署成功了。

整个Tomcat目录下，我们已经认识了bin目录（所有可执行文件，包括启动和关闭服务器的脚本）以及conf目录（服务器配置文件目录），那么我们接着来看其他的文件夹：

- lib目录：Tomcat服务端运行的一些依赖，不用关心。
- logs目录：所有的日志信息都在这里。
- temp目录：存放运行时产生的一些临时文件，不用关心。
- work目录：工作目录，Tomcat会将jsp文件转换为java文件（我们后面会讲到，这里暂时不提及）
- webapp目录：所有的Web项目都在这里，每个文件夹都是一个Web应用程序：

我们发现，官方已经给我们预设了一些项目了，访问后默认使用的项目为ROOT项目，也就是我们默认打开的网站。

我们也可以访问example项目，只需要在后面填写路径即可：http://localhost:8080/examples/，或是docs项目（这个是Tomcat的一些文档）http://localhost:8080/docs/

Tomcat还自带管理页面，我们打开：http://localhost:8080/manager，提示需要用户名和密码，由于不知道是什么，我们先点击取消，页面中出现如下内容：

> You are not authorized to view this page. If you have not changed any configuration files, please examine the file `conf/tomcat-users.xml` in your installation. That file must contain the credentials to let you use this webapp.
>
> For example, to add the `manager-gui` role to a user named `tomcat` with a password of `s3cret`, add the following to the config file listed above.
>
> ```
> <role rolename="manager-gui"/>
> <user username="tomcat" password="s3cret" roles="manager-gui"/>
> ```
>
> Note that for Tomcat 7 onwards, the roles required to use the manager application were changed from the single `manager` role to the following four roles. You will need to assign the role(s) required for the functionality you wish to access.
>
> - `manager-gui` - allows access to the HTML GUI and the status pages
> - `manager-script` - allows access to the text interface and the status pages
> - `manager-jmx` - allows access to the JMX proxy and the status pages
> - `manager-status` - allows access to the status pages only
>
> The HTML interface is protected against CSRF but the text and JMX interfaces are not. To maintain the CSRF protection:
>
> - Users with the `manager-gui` role should not be granted either the `manager-script` or `manager-jmx` roles.
> - If the text or jmx interfaces are accessed through a browser (e.g. for testing since these interfaces are intended for tools not humans) then the browser must be closed afterwards to terminate the session.
>
> For more information - please see the Manager App How-To.

现在我们按照上面的提示，去配置文件中进行修改：

```
<role rolename="manager-gui"/>
<user username="admin" password="admin" roles="manager-gui"/>
```

现在再次打开管理页面，已经可以成功使用此用户进行登陆了。登录后，展示给我们的是一个图形化界面，我们可以快速预览当前服务器的一些信息，包括已经在运行的Web应用程序，甚至还可以查看当前的Web应用程序有没有出现内存泄露。

同样的，还有一个虚拟主机管理页面，用于一台主机搭建多个Web站点，一般情况下使用不到，这里就不做演示了。

我们可以将我们自己的项目也放到webapp文件夹中，这样就可以直接访问到了，我们在webapp目录下新建test文件夹，将我们之前编写的前端代码全部放入其中（包括html文件、js、css、icon等），重启服务器。

我们可以直接通过 http://localhost:8080/test/ 来进行访问。

使用Maven创建Web项目

虽然我们已经可以在Tomcat上部署我们的前端页面了，但是依然只是一个静态页面（每次访问都是同样的样子），那么如何向服务器请求一个动态的页面呢（比如显示我们访问当前页面的时间）这时就需要我们编写一个Web应用程序来实现了，我们需要在用户向服务器发起页面请求时，进行一些处理，再将结果发送给用户的浏览器。

注意：这里需要使用终极版IDEA，如果你的还是社区版，就很难受了。

我们打开IDEA，新建一个项目，选择Java Enterprise（社区版没有此选项！）项目名称随便，项目模板选择Web应用程序，然后我们需要配置Web应用程序服务器，将我们的Tomcat服务器集成到IDEA中。配置很简单，首先点击新建，然后设置Tomcat主目录即可，配置完成后，点击下一步即可，依赖项使用默认即可，然后点击完成，之后IDEA会自动帮助我们创建Maven项目。

创建完成后，直接点击右上角即可运行此项目了，但是我们发现，有一个Servlet页面不生效。

需要注意的是，Tomcat10以上的版本比较新，Servlet API包名发生了一些变化，因此我们需要修改一下依赖：

```
<dependency>
    <groupId>jakarta.servlet</groupId>
    <artifactId>jakarta.servlet-api</artifactId>
    <version>5.0.0</version>
    <scope>provided</scope>
</dependency>
```

注意包名全部从javax改为jakarta，我们需要手动修改一下。

感兴趣的可以了解一下为什么名称被修改了：

> Eclipse基金会在2019年对 Java EE 标准的每个规范进行了重命名，阐明了每个规范在Jakarta EE平台未来的角色。
>
> 新的名称Jakarta EE是Java EE的第二次重命名。2006年5月，“J2EE”一词被弃用，并选择了Java EE这个名称。在YouTube还只是一家独立的公司的时候，数字2就就从名字中消失了，而且当时冥王星仍然被认为是一颗行星。同样，作为Java SE 5（2004）的一部分，数字2也从J2SE中删除了，那时谷歌还没有上市。
>
> **因为不能再使用javax名称空间，Jakarta EE提供了非常明显的分界线。**
>
> - Jakarta 9（2019及以后）使用jakarta命名空间。
> - Java EE 5（2005）到Java EE 8（2017）使用javax命名空间。
> - Java EE 4使用javax命名空间。

我们可以将项目直接打包为war包（默认），打包好之后，放入webapp文件夹，就可以直接运行我们通过Java编写的Web应用程序了，访问路径为文件的名称。

Servlet

前面我们已经完成了基本的环境搭建，那么现在我们就可以开始来了解我们的第一个重要类——Servlet。

它是Java EE的一个标准，大部分的Web服务器都支持此标准，包括Tomcat，就像之前的JDBC一样，由官方定义了一系列接口，而具体实现由我们来编写，最后交给Web服务器（如Tomcat）来运行我们编写的Servlet。

那么，它能做什么呢？我们可以通过实现Servlet来进行动态网页响应，使用Servlet，不再是直接由Tomcat服务器发送我们编写好的静态网页内容（HTML文件），而是由我们通过Java代码进行动态拼接的结果，它能够很好地实现动态网页的返回。

当然，Servlet并不是专用于HTTP协议通信，也可以用于其他的通信，但是一般都是用于HTTP。

创建Servlet

那么如何创建一个Servlet呢，非常简单，我们只需要实现`Servlet`类即可，并添加注解`@WebServlet`来进行注册。

```
@WebServlet("/test")
public class TestServlet implements Servlet {
        ...实现接口方法
}
```

我们现在就可以去访问一下我们的页面：http://localhost:8080/test/test

我们发现，直接访问此页面是没有任何内容的，这是因为我们还没有为该请求方法编写实现，这里先不做讲解，后面我们会对浏览器的请求处理做详细的介绍。

除了直接编写一个类，我们也可以在`web.xml`中进行注册，现将类上`@WebServlet`的注解去掉：

```
<servlet>
    <servlet-name>test</servlet-name>
    <servlet-class>com.example.webtest.TestServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>test</servlet-name>
    <url-pattern>/test</url-pattern>
</servlet-mapping>
```

这样的方式也能注册Servlet，但是显然直接使用注解更加方便，因此之后我们一律使用注解进行开发。只有比较新的版本才支持此注解，老的版本是不支持的哦。

实际上，Tomcat服务器会为我们提供一些默认的Servlet，也就是说在服务器启动后，即使我们什么都不编写，Tomcat也自带了几个默认的Servlet，他们编写在conf目录下的web.xml中：

```
<!-- The mapping for the default servlet -->
    <servlet-mapping>
        <servlet-name>default</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

    <!-- The mappings for the JSP servlet -->
    <servlet-mapping>
        <servlet-name>jsp</servlet-name>
        <url-pattern>*.jsp</url-pattern>
        <url-pattern>*.jspx</url-pattern>
    </servlet-mapping>
```

我们发现，默认的Servlet实际上可以帮助我们去访问一些静态资源，这也是为什么我们启动Tomcat服务器之后，能够直接访问webapp目录下的静态页面。

我们可以将之前编写的页面放入到webapp目录下，来测试一下是否能直接访问。

探究Servlet的生命周期

我们已经了解了如何注册一个Servlet，那么我们接着来看看，一个Servlet是如何运行的。

首先我们需要了解，Servlet中的方法各自是在什么时候被调用的，我们先编写一个打印语句来看看：

```
public class TestServlet implements Servlet {

    public TestServlet(){
        System.out.println("我是构造方法! ");
    }

    @Override
    public void init(ServletConfig servletConfig) throws ServletException {
        System.out.println("我是init");
    }

    @Override
    public ServletConfig getServletConfig() {
        System.out.println("我是getServletConfig");
        return null;
    }

    @Override
    public void service(ServletRequest servletRequest, ServletResponse
servletResponse) throws ServletException, IOException {
        System.out.println("我是service");
    }

    @Override
    public String getServletInfo() {
        System.out.println("我是getServletInfo");
        return null;
    }

    @Override
    public void destroy() {
        System.out.println("我是destroy");
    }
}
```

我们首先启动一次服务器，然后访问我们定义的页面，然后再关闭服务器，得到如下的顺序：

> 我是构造方法!
> 我是init
> 我是service
> 我是service（出现两次是因为浏览器请求了2次，是因为有一次是请求favicon.ico，浏览器通病）
>
> 我是destroy

我们可以多次尝试去访问此页面，但是init和构造方法只会执行一次，而每次访问都会执行的是`service`方法，因此，一个Servlet的生命周期为：

- 首先执行构造方法完成 Servlet 初始化

- Servlet 初始化后调用 **init ()** 方法。
- Servlet 调用 **service()** 方法来处理客户端的请求。
- Servlet 销毁前调用 **destroy()** 方法。
- 最后，Servlet 是由 JVM 的垃圾回收器进行垃圾回收的。

现在我们发现，实际上在Web应用程序运行时，每当浏览器向服务器发起一个请求时，都会创建一个线程执行一次`service`方法，来让我们处理用户的请求，并将结果响应给用户。

我们发现`service`方法中，还有两个参数，`ServletRequest`和`ServletResponse`，实际上，用户发起的HTTP请求，就被Tomcat服务器封装为了一个`ServletRequest`对象，我们得到是其实是Tomcat服务器帮助我们创建的一个实现类，HTTP请求报文中的所有内容，都可以从`ServletRequest`对象中获取，同理，`ServletResponse`就是我们需要返回给浏览器的HTTP响应报文实体类封装。

那么我们来看看`ServletRequest`中有哪些内容，我们可以获取请求的一些信息：

```
@Override
public void service(ServletRequest servletRequest, ServletResponse
servletResponse) throws ServletException, IOException {
    //首先将其转换为HttpServletRequest（继承自ServletRequest，一般是此接口实现）
        HttpServletRequest request = (HttpServletRequest) servletRequest;

        System.out.println(request.getProtocol());  //获取协议版本
        System.out.println(request.getRemoteAddr());  //获取访问者的IP地址
          System.out.println(request.getMethod());   //获取请求方法
        //获取头部信息
        Enumeration<String> enumeration = request.getHeaderNames();
        while (enumeration.hasMoreElements()){
            String name = enumeration.nextElement();
            System.out.println(name + ": " + request.getHeader(name));
        }
}
```

我们发现，整个HTTP请求报文中的所有内容，都可以通过`HttpServletRequest`对象来获取，当然，它的作用肯定不仅仅是获取头部信息，我们还可以使用它来完成更多操作，后面会一一讲解。

那么我们再来看看`ServletResponse`，这个是服务端的响应内容，我们可以在这里填写我们想要发送给浏览器显示的内容：

```
//转换为HttpServletResponse（同上）
HttpServletResponse response = (HttpServletResponse) servletResponse;
//设定内容类型以及编码格式（普通HTML文本使用text/html，之后会讲解文件传输）
response.setHeader("Content-type", "text/html;charset=UTF-8");
//获取Writer直接写入内容
response.getWriter().write("我是响应内容！");
//所有内容写入完成之后，再发送给浏览器
```

现在我们在浏览器中打开此页面，就能够收到服务器发来的响应内容了。其中，响应头部分，是由Tomcat帮助我们生成的一个默认响应头。

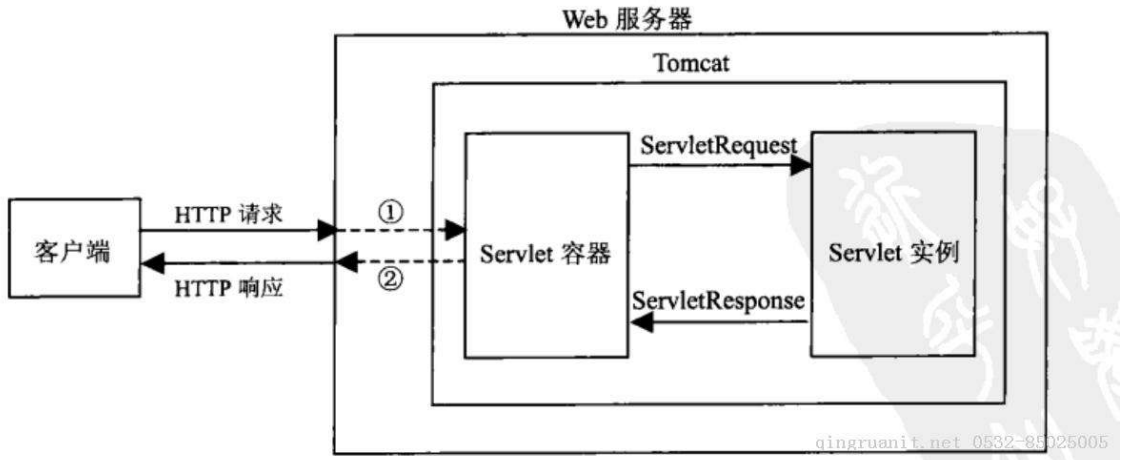


图 1-1　Tomcat 服务器响应客户请求过程

因此，实际上整个流程就已经很清晰明了了。

解读和使用HttpServlet

前面我们已经学习了如何创建、注册和使用Servlet，那么我们继续来深入学习Servlet接口的一些实现类。

首先`Servlet`有一个直接实现抽象类`GenericServlet`，那么我们来看看此类做了什么事情。

我们发现，这个类完善了配置文件读取和Servlet信息相关的的操作，但是依然没有去实现service方法，因此此类仅仅是用于完善一个Servlet的基本操作，那么我们接着来看`HttpServlet`，它是遵循HTTP协议的一种Servlet，继承自`GenericServlet`，它根据HTTP协议的规则，完善了service方法。

在阅读了HttpServlet源码之后，我们发现，其实我们只需要继承HttpServlet来编写我们的Servlet就可以了，并且它已经帮助我们提前实现了一些操作，这样就会给我们省去很多的时间。

```
@Log
@WebServlet("/test")
public class TestServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        resp.setContentType("text/html;charset=UTF-8");
        resp.getWriter().write("<h1>恭喜你解锁了全新玩法</h1>");
    }
}
```

现在，我们只需要重写对应的请求方式，就可以快速完成Servlet的编写。

@WebServlet注解详解

我们接着来看WebServlet注解，我们前面已经得知，可以直接使用此注解来快速注册一个Servlet，那么我们来想细看看此注解还有什么其他的玩法。

首先name属性就是Servlet名称，而urlPatterns和value实际上是同样功能，就是代表当前Servlet的访问路径，它不仅仅可以是一个固定值，还可以进行通配符匹配：

```
@WebServlet("/test/*")
```

上面的路径表示，所有匹配`/test/随便什么`的路径名称，都可以访问此Servlet，我们可以在浏览器中尝试一下。

也可以进行某个扩展名称的匹配：

```
@WebServlet("*.js")
```

这样的话，获取任何以js结尾的文件，都会由我们自己定义的Servlet处理。

那么如果我们的路径为`/`呢?

```
@WebServlet("/")
```

此路径和Tomcat默认为我们提供的Servlet冲突，会直接替换掉默认的，而使用我们的，此路径的意思为，如果没有找到匹配当前访问路径的Servlet，那么久会使用此Servlet进行处理。

我们还可以为一个Servlet配置多个访问路径：

```
@WebServlet({"/test1", "/test2"})
```

我们接着来看loadOnStartup属性，此属性决定了是否在Tomcat启动时就加载此Servlet，默认情况下，Servlet只有在被访问时才会加载，它的默认值为-1，表示不在启动时加载，我们可以将其修改为大于等于0的数，来开启启动时加载。并且数字的大小决定了此Servlet的启动优先级。

```
@Log
@WebServlet(value = "/test", loadOnStartup = 1)
public class TestServlet extends HttpServlet {

    @Override
    public void init() throws ServletException {
        super.init();
        log.info("我被初始化了！");
    }

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        resp.setContentType("text/html;charset=UTF-8");
        resp.getWriter().write("<h1>恭喜你解锁了全新玩法</h1>");
    }
}
```

其他内容都是Servlet的一些基本配置，这里就不详细讲解了。

使用POST请求完成登陆

我们前面已经了解了如何使用Servlet来处理HTTP请求，那么现在，我们就结合前端，来实现一下登陆操作。

我们需要修改一下我们的Servlet，现在我们要让其能够接收一个POST请求：

```
@Log
@WebServlet("/login")
public class LoginServlet extends HttpServlet {
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        req.getParameterMap().forEach((k, v) -> {
            System.out.println(k + ": " + Arrays.toString(v));
        });
    }
}
```

`ParameterMap`存储了我们发送的POST请求所携带的表单数据，我们可以直接将其遍历查看，浏览器发送了什么数据。

现在我们再来修改一下前端：

```
<body>
    <h1>登录到系统</h1>
    <form method="post" action="login">
        <hr>
        <div>
            <label>
                <input type="text" placeholder="用户名" name="username">
            </label>
        </div>
        <div>
            <label>
                <input type="password" placeholder="密码" name="password">
            </label>
        </div>
        <div>
            <button>登录</button>
        </div>
    </form>
</body>
```

通过修改form标签的属性，现在我们点击登录按钮，会自动向后台发送一个POST请求，请求地址为当前地址+/login（注意不同路径的写法），也就是我们上面编写的Servlet路径。

运行服务器，测试后发现，在点击按钮后，确实向服务器发起了一个POST请求，并且携带了表单中文本框的数据。

现在，我们根据已有的基础，将其与数据库打通，我们进行一个真正的用户登录操作，首先修改一下Servlet的逻辑：

```
@Override
protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
    //首先设置一下响应类型
    resp.setContentType("text/html;charset=UTF-8");
    //获取POST请求携带的表单数据
    Map<String, String[]> map = req.getParameterMap();
    //判断表单是否完整
```

```
    if(map.containsKey("username") && map.containsKey("password")) {
        String username = req.getParameter("username");
        String password = req.getParameter("password");

        //权限校验（待完善）
    }else {
        resp.getWriter().write("错误，您的表单数据不完整！");
    }
}
```

接下来我们再去编写Mybatis的依赖和配置文件，创建一个表，用于存放我们用户的账号和密码。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="${驱动类（含包名）}"/>
                <property name="url" value="${数据库连接URL}"/>
                <property name="username" value="${用户名}"/>
                <property name="password" value="${密码}"/>
            </dataSource>
        </environment>
    </environments>
</configuration>
```

```
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.5.7</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.27</version>
</dependency>
```

配置完成后，在我们的Servlet的init方法中编写Mybatis初始化代码，因为它只需要初始化一次。

```
SqlSessionFactory factory;
@SneakyThrows
@Override
public void init() throws ServletException {
    factory = new
SqlSessionFactoryBuilder().build(Resources.getResourceAsReader("mybatis-
config.xml"));
}
```

现在我们创建一个实体类以及Mapper来进行用户信息查询：

```
@Data
public class User {
    String username;
    String password;
}
```

```
public interface UserMapper {

    @Select("select * from users where username = #{username} and password = #{password}")
    User getUser(@Param("username") String username, @Param("password") String password);
}
```

```
<mappers>
    <mapper class="com.example.dao.UserMapper"/>
</mappers>
```

好了，现在完事具备，只欠东风了，我们来完善一下登陆验证逻辑：

```
//登陆校验（待完善）
try (SqlSession sqlSession = factory.openSession(true)){
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);
    User user = mapper.getUser(username, password);
    //判断用户是否登陆成功，若查询到信息则表示存在此用户
    if(user != null){
        resp.getWriter().write("登陆成功！");
    }else {
        resp.getWriter().write("登陆失败，请验证您的用户名或密码！");
    }
}
```

现在再去浏览器上进行测试吧！

注册界面其实是同理的，这里就不多做讲解了。

上传和下载文件

首先我们来看看比较简单的下载文件，首先将我们的icon.png放入到resource文件夹中，接着我们编写一个Servlet用于处理文件下载：

```
@WebServlet("/file")
public class FileServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
      resp.setContentType("image/png");
      OutputStream outputStream = resp.getOutputStream();
      InputStream inputStream = Resources.getResourceAsStream("icon.png");

    }
}
```

为了更加快速地编写IO代码，我们可以引入一个工具库：

```
<dependency>
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
    <version>2.6</version>
</dependency>
```

使用此类库可以快速完成IO操作：

```
resp.setContentType("image/png");
OutputStream outputStream = resp.getOutputStream();
InputStream inputStream = Resources.getResourceAsStream("icon.png");
//直接使用copy方法完成转换
IOUtils.copy(inputStream, outputStream);
```

现在我们在前端页面添加一个链接，用于下载此文件：

```
<hr>
<a href="file" download="icon.png">点我下载高清资源</a>
```

下载文件搞定，那么如何上传一个文件呢？

首先我们编写前端部分：

```
<form method="post" action="file" enctype="multipart/form-data">
    <div>
        <input type="file" name="test-file">
    </div>
    <div>
        <button>上传文件</button>
    </div>
</form>
```

注意必须添加`enctype="multipart/form-data"`，来表示此表单用于文件传输。

现在我们来修改一下Servlet代码：

```
@MultipartConfig
@WebServlet("/file")
```

```
public class FileServlet extends HttpServlet {

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        try(FileOutputStream stream = new
FileOutputStream("/Users/nagocoler/Documents/IdeaProjects/WebTest/test.png")){
            Part part = req.getPart("test-file");
            IOUtils.copy(part.getInputStream(), stream);
            resp.setContentType("text/html;charset=UTF-8");
            resp.getWriter().write("文件上传成功！");
        }
    }
}
```

注意，必须添加`@MultipartConfig`注解来表示此Servlet用于处理文件上传请求。

现在我们再运行服务器，并将我们刚才下载的文件又上传给服务端。

使用XHR请求数据

现在我们希望，网页中的部分内容，可以动态显示，比如网页上有一个时间，旁边有一个按钮，点击按钮就可以刷新当前时间。

这个时候就需要我们在网页展示时向后端发起请求了，并根据后端响应的结果，动态地更新页面中的内容，要实现此功能，就需要用到JavaScript来帮助我们，首先在js中编写我们的XHR请求，并在请求中完成动态更新：

```
function updateTime() {
    let xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function() {
        if (xhr.readyState === 4 && xhr.status === 200) {
            document.getElementById("time").innerText = xhr.responseText
        }
    };
    xhr.open('GET', 'time', true);
    xhr.send();
}
```

接着修改一下前端页面，添加一个时间显示区域：

```
<hr>
<div id="time"></div>
<br>
<button onclick="updateTime()">更新数据</button>
<script>
    updateTime()
</script>
```

最后创建一个Servlet用于处理时间更新请求：

```
@WebServlet("/time")
public class TimeServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy年MM月dd日
HH:mm:ss");
        String date = dateFormat.format(new Date());
        resp.setContentType("text/html;charset=UTF-8");
        resp.getWriter().write(date);
    }
}
```

现在点击按钮就可以更新了。

GET请求也能传递参数，这里做一下演示。

重定向与请求转发

当我们希望用户登录完成之后，直接跳转到网站的首页，那么这个时候，我们就可以使用重定向来完成。当浏览器收到一个重定向的响应时，会按照重定向响应给出的地址，再次向此地址发出请求。

实现重定向很简单，只需要调用一个方法即可，我们修改一下登陆成功后执行的代码：

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

调用后，响应的状态码会被设置为302，并且响应头中添加了一个Location属性，此属性表示，需要重定向到哪一个网址。

现在，如果我们成功登陆，那么服务器会发送给我们一个重定向响应，这时，我们的浏览器会去重新请求另一个网址。这样，我们在登陆成功之后，就可以直接帮助用户跳转到用户首页了。

那么我们接着来看请求转发，请求转发其实是一种服务器内部的跳转机制，我们知道，重定向会使得浏览器去重新请求一个页面，而请求转发则是服务器内部进行跳转，它的目的是，直接将本次请求转发给其他Servlet进行处理，并由其他Servlet来返回结果，因此它是在进行内部的转发。

```
req.getRequestDispatcher("/time").forward(req, resp);
```

现在，在登陆成功的时候，我们将请求转发给处理时间的Servlet，注意这里的路径规则和之前的不同，我们需要填写Servlet上指明的路径，并且请求转发只能转发到此应用程序内部的Servlet，不能转发给其他站点或是其他Web应用程序。

现在再次进行登陆操作，我们发现，返回结果为一个405页面，证明了，我们的请求现在是被另一个Servlet进行处理，并且请求的信息全部被转交给另一个Servlet，由于此Servlet不支持POST请求，因此返回405状态码。

那么也就是说，该请求包括请求参数也一起被传递了，那么我们可以尝试获取以下POST请求的参数。

现在我们给此Servlet添加POST请求处理，直接转交给Get请求处理：

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

再次访问，成功得到结果，但是我们发现，浏览器只发起了一次请求，并没有再次请求新的URL，也就是说，这一次请求直接返回了请求转发后的处理结果。

那么，请求转发有什么好处呢？它可以携带数据！

```
req.setAttribute("test", "我是请求转发前的数据");
req.getRequestDispatcher("/time").forward(req, resp);
```

```
System.out.println(req.getAttribute("test"));
```

通过`setAttribute`方法来给当前请求添加一个附加数据，在请求转发后，我们可以直接获取到该数据。

重定向属于2次请求，因此无法使用这种方式来传递数据，那么，如何在重定向之间传递数据呢？我们可以使用即将要介绍的ServletContext对象。

最后总结，两者的区别为：

- 请求转发是一次请求，重定向是两次请求
- 请求转发地址栏不会发生改变， 重定向地址栏会发生改变
- 请求转发可以共享请求参数 ，重定向之后，就获取不了共享参数了
- 请求转发只能转发给内部的Servlet

了解ServletContext对象

ServletContext全局唯一，它是属于整个Web应用程序的，我们可以通过`getServletContext()`来获取到此对象。

此对象也能设置附加值：

```
ServletContext context = getServletContext();
context.setAttribute("test", "我是重定向之前的数据");
resp.sendRedirect("time");
```

```
System.out.println(getServletContext().getAttribute("test"));
```

因为无论在哪里，无论什么时间，获取到的ServletContext始终是同一个对象，因此我们可以随时随地获取我们添加的属性。

它不仅仅可以用来进行数据传递，还可以做一些其他的事情，比如请求转发：

```
context.getRequestDispatcher("/time").forward(req, resp);
```

它还可以获取根目录下的资源文件（注意是webapp根目录下的，不是resource中的资源）

初始化参数

初始化参数类似于初始化配置需要的一些值，比如我们的数据库连接相关信息，就可以通过初始化参数来给予Servlet，或是一些其他的配置项，也可以使用初始化参数来实现。

我们可以给一个Servlet添加一些初始化参数：

```
@WebServlet(value = "/login", initParams = {
        @WebInitParam(name = "test", value = "我是一个默认的初始化参数")
})
```

它也是以键值对形式保存的，我们可以直接通过Servlet的`getInitParameter`方法获取：

```
System.out.println(getInitParameter("test"));
```

但是，这里的初始化参数仅仅是针对于此Servlet，我们也可以定义全局初始化参数，只需要在web.xml编写即可：

```
<context-param>
    <param-name>lbwnb</param-name>
    <param-value>我是全局初始化参数</param-value>
</context-param>
```

我们需要使用ServletContext来读取全局初始化参数：

```
ServletContext context = getServletContext();
System.out.println(context.getInitParameter("lbwnb"));
```

有关ServletContext其他的内容，我们需要完成后面内容的学习，才能理解。

Cookie

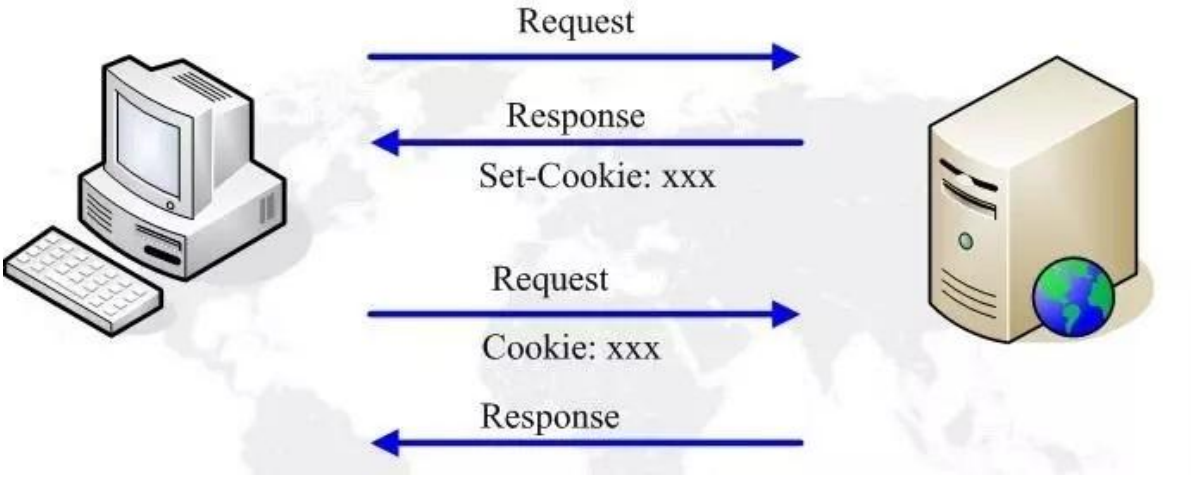
什么是Cookie？不是曲奇，它可以在浏览器中保存一些信息，并且在下次请求时，请求头中会携带这些信息。

我们可以编写一个测试用例来看看：

```
Cookie cookie = new Cookie("test", "yyds");
resp.addCookie(cookie);
resp.sendRedirect("time");
```

```
for (Cookie cookie : req.getCookies()) {
    System.out.println(cookie.getName() + ": " + cookie.getValue());
}
```

我们可以观察一下，在`HttpServletResponse`中添加Cookie之后，浏览器的响应头中会包含一个`Set-Cookie`属性，同时，在重定向之后，我们的请求头中，会携带此Cookie作为一个属性，同时，我们可以直接通过`HttpServletRequest`来快速获取有哪些Cookie信息。



还有这么神奇的事情吗？那么我们来看看，一个Cookie包含哪些信息：

- name - Cookie的名称，Cookie一旦创建，名称便不可更改
- value - Cookie的值，如果值为Unicode字符，需要为字符编码。如果为二进制数据，则需要使用BASE64编码
- maxAge - Cookie失效的时间，单位秒。如果为正数，则该Cookie在maxAge秒后失效。如果为负数，该Cookie为临时Cookie，关闭浏览器即失效，浏览器也不会以任何形式保存该Cookie。如果为0，表示删除该Cookie。默认为-1。
- secure - 该Cookie是否仅被使用安全协议传输。安全协议。安全协议有HTTPS，SSL等，在网络上传输数据之前先将数据加密。默认为false。
- path - Cookie的使用路径。如果设置为“/sessionWeb/”，则只有contextPath为“/sessionWeb”的程序可以访问该Cookie。如果设置为“/”，则本域名下contextPath都可以访问该Cookie。注意最后一个字符必须为“/”。
- domain - 可以访问该Cookie的域名。如果设置为“.google.com”，则所有以“google.com”结尾的域名都可以访问该Cookie。注意第一个字符必须为“.”。
- comment - 该Cookie的用处说明，浏览器显示Cookie信息的时候显示该说明。
- version - Cookie使用的版本号。0表示遵循Netscape的Cookie规范，1表示遵循W3C的RFC 2109规范

我们发现，最关键的其实是`name`、`value`、`maxAge`、`domain`属性。

那么我们来尝试修改一下maxAge来看看失效时间：

```
cookie.setMaxAge(20);
```

设定为20秒，我们可以直接看到，响应头为我们设定了20秒的过期时间。20秒内访问都会携带此Cookie，而超过20秒，Cookie消失。

既然了解了Cookie的作用，我们就可以通过使用Cookie来实现记住我功能，我们可以将用户名和密码全部保存在Cookie中，如果访问我们的首页时携带了这些Cookie，那么我们就可以直接为用户进行登陆，如果登陆成功则直接跳转到首页，如果登陆失败，则清理浏览器中的Cookie。

那么首先，我们先在前端页面的表单中添加一个勾选框：

```
<div>
    <label>
        <input type="checkbox" placeholder="记住我" name="remember-me">
        记住我
    </label>
</div>
```

接着，我们在登陆成功时进行判断，如果用户勾选了记住我，那么就讲Cookie存储到本地：

```
if(map.containsKey("remember-me")){   //若勾选了勾选框，那么会此表单信息
    Cookie cookie_username = new Cookie("username", username);
    cookie_username.setMaxAge(30);
    Cookie cookie_password = new Cookie("password", password);
    cookie_password.setMaxAge(30);
    resp.addCookie(cookie_username);
    resp.addCookie(cookie_password);
}
```

然后，我们修改一下默认的请求地址，现在一律通过`http://localhost:8080/yyds/login`进行登陆，那么我们需要添加GET请求的相关处理：

```
@Override
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
    Cookie[] cookies = req.getCookies();
    if(cookies != null){
        String username = null;
        String password = null;
        for (Cookie cookie : cookies) {
            if(cookie.getName().equals("username")) username =
cookie.getValue();
            if(cookie.getName().equals("password")) password =
cookie.getValue();
        }
        if(username != null && password != null){
            //登陆校验
            try (SqlSession sqlSession = factory.openSession(true)){
                UserMapper mapper = sqlSession.getMapper(UserMapper.class);
                User user = mapper.getUser(username, password);
                if(user != null){
                    resp.sendRedirect("time");
                    return;   //直接返回
                }
            }
        }
    }
    req.getRequestDispatcher("/").forward(req, resp);   //正常情况还是转发给默认的
Servlet帮我们返回静态页面
}
```

现在，30秒内都不需要登陆，访问登陆页面后，会直接跳转到time页面。

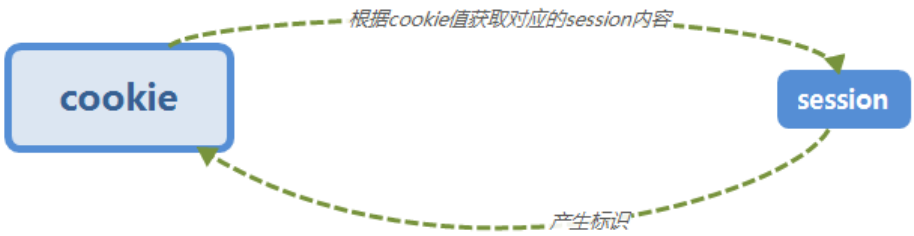
现在已经离我们理想的页面越来越接近了，但是仍然有一个问题，就是我们的首页，无论是否登陆，所有人都可以访问，那么，如何才可以实现只有登陆之后才能访问呢？这就需要用到Session了。

Session

由于HTTP是无连接的，那么如何能够辨别当前的请求是来自哪个用户发起的呢？Session就是用来处理这种问题的，每个用户的会话都会有一个自己的Session对象，来自同一个浏览器的所有请求，就属于同一个会话。

但是HTTP协议是无连接的呀，那Session是如何做到辨别是否来自同一个浏览器呢？Session实际上是基于Cookie实现的，前面我们了解了Cookie，我们知道，服务端可以将Cookie保存到浏览器，当浏览器下次访问时，就会附带这些Cookie信息。

Session也利用了这一点，它会给浏览器设定一个叫做`JSESSIONID`的Cookie，值是一个随机的排列组合，而此Cookie就对应了你属于哪一个对话，只要我们的浏览器携带此Cookie访问服务器，服务器就会通过Cookie的值进行辨别，得到对应的Session对象，因此，这样就可以追踪到底是哪一个浏览器在访问服务器。



那么现在，我们在用户登录成功之后，将用户对象添加到Session中，只要是此用户发起的请求，我们都可以从`HttpSession`中读取到存储在会话中的数据：

```
HttpSession session = req.getSession();
session.setAttribute("user", user);
```

同时，如果用户没有登录就去访问首页，那么我们将发送一个重定向请求，告诉用户，需要先进行登录才可以访问：

```
HttpSession session = req.getSession();
User user = (User) session.getAttribute("user");
if(user == null) {
    resp.sendRedirect("login");
    return;
}
```

在访问的过程中，注意观察Cookie变化。

Session并不是永远都存在的，它有着自己的过期时间，默认时间为30分钟，若超过此时间，Session将丢失，我们可以在配置文件中修改过期时间：

```
<session-config>
    <session-timeout>1</session-timeout>
</session-config>
```

我们也可以在代码中使用`invalidate`方法来使Session立即失效：

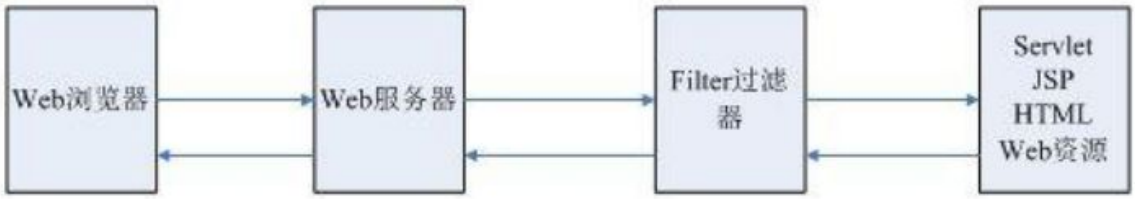
```
session.invalidate();
```

现在，通过Session，我们就可以更好地控制用户对于资源的访问，只有完成登陆的用户才有资格访问首页。

Filter

有了Session之后，我们就可以很好地控制用户的登陆验证了，只有授权的用户，才可以访问一些页面，但是我们需要一个一个去进行配置，还是太过复杂，能否一次性地过滤掉没有登录验证的用户呢？

过滤器相当于在所有访问前加了一堵墙，来自浏览器的所有访问请求都会首先经过过滤器，只有过滤器允许通过的请求，才可以顺利地到达对应的Servlet，而过滤器不允许的通过的请求，我们可以自由地进行控制是否进行重定向或是请求转发。并且过滤器可以添加很多个，就相当于添加了很多堵墙，我们的请求只有穿过层层阻碍，才能与Servlet相拥，像极了爱情。



添加一个过滤器非常简单，只需要实现Filter接口，并添加`@WebFilter`注解即可：

```
@WebFilter("/*")   //路径的匹配规则和Servlet一致，这里表示匹配所有请求
public class TestFilter implements Filter {
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException, ServletException {

    }
}
```

这样我们就成功地添加了一个过滤器，那么添加一句打印语句看看，是否所有的请求都会经过此过滤器：

```
HttpServletRequest request = (HttpServletRequest) servletRequest;
System.out.println(request.getRequestURL());
```

我们发现，现在我们发起的所有请求，一律需要经过此过滤器，并且所有的请求都没有任何的响应内容。

那么如何让请求可以顺利地到达对应的Servlet，也就是说怎么让这个请求顺利通过呢？我们只需要在最后添加一句：

```
filterChain.doFilter(servletRequest, servletResponse);
```

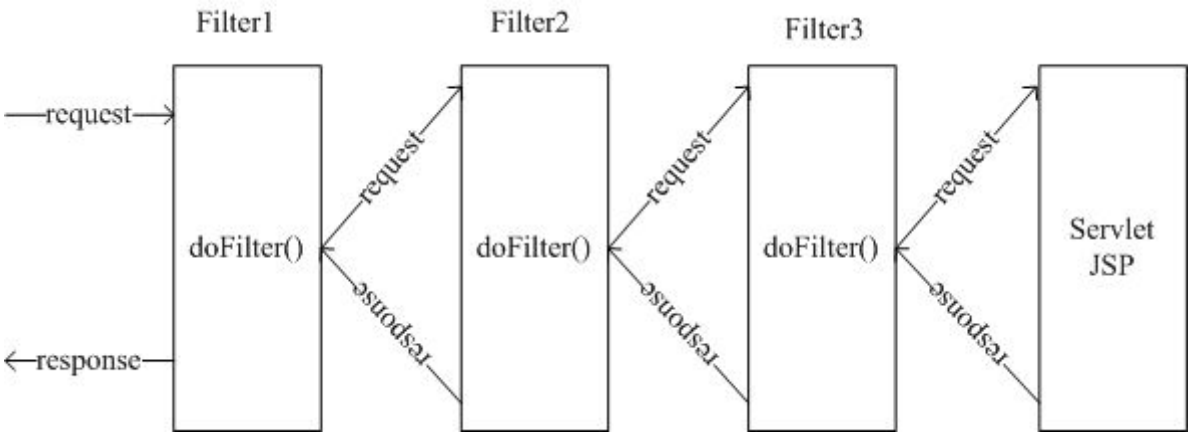
那么这行代码是什么意思呢？

由于我们整个应用程序可能存在多个过滤器，那么这行代码的意思实际上是将此请求继续传递给下一个过滤器，当没有下一个过滤器时，才会到达对应的Servlet进行处理，我们可以再来创建一个过滤器看看效果：

```
@WebFilter("/*")
public class TestFilter2 implements Filter {
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException, ServletException {
        System.out.println("我是2号过滤器");
        filterChain.doFilter(servletRequest, servletResponse);
    }
}
```

由于过滤器的过滤顺序是按照类名的自然排序进行的，因此我们将第一个过滤器命名进行调整。

我们发现，在经过第一个过滤器之后，会继续前往第二个过滤器，只有两个过滤器全部经过之后，才会到达我们的Servlet中。



实际上，当`doFilter`方法调用时，就会一直向下直到Servlet，在Servlet处理完成之后，又依次返回到最前面的Filter，类似于递归的结构，我们添加几个输出语句来判断一下：

```
@Override
public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException, ServletException {
    System.out.println("我是2号过滤器");
    filterChain.doFilter(servletRequest, servletResponse);
    System.out.println("我是2号过滤器，处理后");
}
```

```
@Override
public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException, ServletException {
    System.out.println("我是1号过滤器");
    filterChain.doFilter(servletRequest, servletResponse);
    System.out.println("我是1号过滤器，处理后");
}
```

最后验证我们的结论。

同Servlet一样，Filter也有对应的HttpFilter专用类，它针对HTTP请求进行了专门处理，因此我们可以直接使用HttpFilter来编写：

```
public abstract class HttpFilter extends GenericFilter {
    private static final long serialVersionUID = 7478463438252262094L;

    public HttpFilter() {
    }

    public void doFilter(ServletRequest req, ServletResponse res, FilterChain
chain) throws IOException, ServletException {
        if (req instanceof HttpServletRequest && res instanceof
HttpServletResponse) {
            this.doFilter((HttpServletRequest)req, (HttpServletResponse)res,
chain);
        } else {
            throw new ServletException("non-HTTP request or response");
        }
    }

    protected void doFilter(HttpServletRequest req, HttpServletResponse res,
FilterChain chain) throws IOException, ServletException {
        chain.doFilter(req, res);
    }
}
```

那么现在，我们就可以给我们的应用程序添加一个过滤器，用户在未登录情况下，只允许静态资源和登陆页面请求通过，登陆之后畅行无阻：

```
@WebFilter("/*")
public class MainFilter extends HttpFilter {
    @Override
    protected void doFilter(HttpServletRequest req, HttpServletResponse res,
FilterChain chain) throws IOException, ServletException {
        String url = req.getRequestURL().toString();
        //判断是否为静态资源
        if(!url.endsWith(".js") && !url.endsWith(".css") &&
!url.endsWith(".png")){
            HttpSession session = req.getSession();
            User user = (User) session.getAttribute("user");
            //判断是否未登陆
            if(user == null && !url.endsWith("login")){
                res.sendRedirect("login");
                return;
            }
        }
        //交给过滤链处理
        chain.doFilter(req, res);
    }
}
```

现在，我们的页面已经基本完善为我们想要的样子了。

当然，可能跟着教程编写的项目比较乱，大家可以自己花费一点时间来重新编写一个Web应用程序，加深对之前讲解知识的理解。我们也会在之后安排一个编程实战进行深化练习。

Listener

监听器并不是我们学习的重点内容，那么什么是监听器呢？

如果我们希望，在应用程序加载的时候，或是Session创建的时候，亦或是在Request对象创建的时候进行一些操作，那么这个时候，我们就可以使用监听器来实现。

	ServletContext域	HttpSession域	ServletRequest域
域对象的创建与销毁	ServletContextListener	HttpSessionListener	ServlerRequestListener
域对象内的属性的变化	ServletContextAttributeListener	HttpSessionAttributeListener	ServlerRequestAttributeListener

默认为我们提供了很多类型的监听器，我们这里就演示一下监听Session的创建即可：

```
@WebListener
public class TestListener implements HttpSessionListener {
    @Override
    public void sessionCreated(HttpSessionEvent se) {
        System.out.println("有一个Session被创建了");
    }
}
```

有关监听器相关内容，了解即可。

了解JSP页面与加载规则

前面我们已经完成了整个Web应用程序生命周期中所有内容的学习，我们已经完全了解，如何编写一个Web应用程序，并放在Tomcat上部署运行，以及如何控制浏览器发来的请求，通过Session+Filter实现用户登陆验证，通过Cookie实现自动登陆等操作。到目前为止，我们已经具备编写一个完整Web网站的能力。

在之前的教程中，我们的前端静态页面并没有与后端相结合，我们前端页面所需的数据全部需要单独向后端发起请求获取，并动态进行内容填充，这是一种典型的前后端分离写法，前端只负责要数据和显示数据，后端只负责处理数据和提供数据，这也是现在更流行的一种写法，让前端开发者和后端开发者各尽其责，更加专一，这才是我们所希望的开发模式。

JSP并不是我们需要重点学习的内容，因为它已经过时了，使用JSP会导致前后端严重耦合，因此这里只做了解即可。

JSP其实就是一种模板引擎，那么何谓模板引擎呢？顾名思义，它就是一个模板，而模板需要我们填入数据，才可以变成一个页面，也就是说，我们可以直接在前端页面中直接填写数据，填写后生成一个最终的HTML页面返回给前端。

首先我们来创建一个新的项目，项目创建成功后，删除Java目录下的内容，只留下默认创建的jsp文件，我们发现，在webapp目录中，存在一个`index.jsp`文件，现在我们直接运行项目，会直接访问这个JSP页面。

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>
<!DOCTYPE html>
<html>
<head>
    <title>JSP - Hello World</title>
</head>
<body>
<h1><%= "Hello World!" %>
</h1>
<br/>
<a href="hello-servlet">Hello Servlet</a>
</body>
</html>
```

但是我们并没有编写对应的Servlet来解析啊，那么为什么这个JSP页面会被加载呢？

实际上，我们一开始提到的两个Tomcat默认的Servlet中，一个是用于请求静态资源，还有一个就是用于处理jsp的：

```
<!-- The mappings for the JSP servlet -->
    <servlet-mapping>
        <servlet-name>jsp</servlet-name>
        <url-pattern>*.jsp</url-pattern>
        <url-pattern>*.jspx</url-pattern>
    </servlet-mapping>
```

那么，JSP和普通HTML页面有什么区别呢，我们发现它的语法和普通HTML页面几乎一致，我们可以直接在JSP中编写Java代码，并在页面加载的时候执行，我们随便找个地方插入：

```
<%
    System.out.println("JSP页面被加载");
%>
```

我们发现，请求一次页面，页面就会加载一次，并执行我们填写的Java代码。也就是说，我们可以直接在此页面中执行Java代码来填充我们的数据，这样我们的页面就变成了一个动态页面，使用`<%=  %>`来填写一个值：

```
<h1><%= new Date() %></h1>
```

现在访问我们的网站，每次都会创建一个新的Date对象，因此每次访问获取的时间都不一样，我们的网站已经算是一个动态的网站的了。

虽然这样在一定程度上上为我们提供了便利，但是这样的写法相当于整个页面既要编写前端代码，也要编写后端代码，随着项目的扩大，整个页面会显得难以阅读，并且现在都是前后端开发人员职责非常明确的，如果要编写JSP页面，那就必须要招一个既会前端也会后端的程序员，这样显然会导致不必要的开销。

那么我们来研究一下，为什么JSP页面能够在加载的时候执行Java代码呢？

首先我们将此项目打包，并在Tomcat服务端中运行，生成了一个文件夹并且可以正常访问。

我们现在看到`work`目录，我们发现这个里面多了一个`index_jsp.java`和`index_jsp.class`，那么这些东西是干嘛的呢，我们来反编译一下就啥都知道了：

```
public final class index_jsp extends org.apache.jasper.runtime.HttpJspBase  //继承自HttpServlet
    implements org.apache.jasper.runtime.JspSourceDependent,
                 org.apache.jasper.runtime.JspSourceImports {

 ...

  public void _jspService(final jakarta.servlet.http.HttpServletRequest request, final jakarta.servlet.http.HttpServletResponse response)
      throws java.io.IOException, jakarta.servlet.ServletException {

    if (!jakarta.servlet.DispatcherType.ERROR.equals(request.getDispatcherType())) {
      final java.lang.String _jspx_method = request.getMethod();
      if ("OPTIONS".equals(_jspx_method)) {
        response.setHeader("Allow","GET, HEAD, POST, OPTIONS");
        return;
      }
      if (!"GET".equals(_jspx_method) && !"POST".equals(_jspx_method) && !"HEAD".equals(_jspx_method)) {
        response.setHeader("Allow","GET, HEAD, POST, OPTIONS");
        response.sendError(HttpServletResponse.SC_METHOD_NOT_ALLOWED, "JSP 只允许 GET、POST 或 HEAD。Jasper 还允许 OPTIONS");
        return;
      }
    }

    final jakarta.servlet.jsp.PageContext pageContext;
    jakarta.servlet.http.HttpSession session = null;
    final jakarta.servlet.ServletContext application;
    final jakarta.servlet.ServletConfig config;
    jakarta.servlet.jsp.JspWriter out = null;
    final java.lang.Object page = this;
    jakarta.servlet.jsp.JspWriter _jspx_out = null;
    jakarta.servlet.jsp.PageContext _jspx_page_context = null;


    try {
      response.setContentType("text/html; charset=UTF-8");
      pageContext = _jspxFactory.getPageContext(this, request, response,
             null, true, 8192, true);
      _jspx_page_context = pageContext;
      application = pageContext.getServletContext();
      config = pageContext.getServletConfig();
      session = pageContext.getSession();
      out = pageContext.getOut();
      _jspx_out = out;

      out.write("\n");
      out.write("\n");
      out.write("<!DOCTYPE html>\n");
      out.write("<html>\n");
      out.write("<head>\n");
      out.write("    <title>JSP - Hello World</title>\n");
      out.write("</head>\n");
```

```
      out.write("<body>\n");
      out.write("<h1>");
      out.print( new Date() );
      out.write("</h1>\n");

    System.out.println("JSP页面被加载");

      out.write("\n");
      out.write("<br/>\n");
      out.write("<a href=\"hello-servlet\">Hello Servlet</a>\n");
      out.write("</body>\n");
      out.write("</html>");
    } catch (java.lang.Throwable t) {
      if (!(t instanceof jakarta.servlet.jsp.SkipPageException)){
        out = _jspx_out;
        if (out != null && out.getBufferSize() != 0)
          try {
            if (response.isCommitted()) {
              out.flush();
            } else {
              out.clearBuffer();
            }
          } catch (java.io.IOException e) {}
        if (_jspx_page_context != null)
_jspx_page_context.handlePageException(t);
        else throw new ServletException(t);
      }
    } finally {
      _jspxFactory.releasePageContext(_jspx_page_context);
    }
  }
}
```

我们发现，它是继承自`HttpJspBase`类，我们可以反编译一下jasper.jar（它在tomcat的lib目录中）来看看:

```
package org.apache.jasper.runtime;

import jakarta.servlet.ServletConfig;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import jakarta.servlet.jsp.HttpJspPage;
import java.io.IOException;
import org.apache.jasper.compiler.Localizer;

public abstract class HttpJspBase extends HttpServlet implements HttpJspPage {
    private static final long serialVersionUID = 1L;

    protected HttpJspBase() {
    }

    public final void init(ServletConfig config) throws ServletException {
        super.init(config);
```

```
        this.jspInit();
        this._jspInit();
    }

    public String getServletInfo() {
        return Localizer.getMessage("jsp.engine.info", new Object[]{"3.0"});
    }

    public final void destroy() {
        this.jspDestroy();
        this._jspDestroy();
    }

    public final void service(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        this._jspService(request, response);
    }

    public void jspInit() {
    }

    public void _jspInit() {
    }

    public void jspDestroy() {
    }

    protected void _jspDestroy() {
    }

    public abstract void _jspService(HttpServletRequest var1,
HttpServletResponse var2) throws ServletException, IOException;
}
```

实际上，Tomcat在加载JSP页面时，会将其动态转换为一个java类并编译为class进行加载，而生成的Java类，正是一个Servlet的子类，而页面的内容全部被编译为输出字符串，这便是JSP的加载原理，因此，JSP本质上依然是一个Servlet!

如果同学们感兴趣的话，可以查阅一下其他相关的教程，本教程不再讲解此技术。

使用Thymeleaf模板引擎

虽然JSP为我们带来了便捷，但是其缺点也是显而易见的，那么有没有一种既能实现模板，又能兼顾前后端分离的模板引擎呢?

Thymeleaf（百里香叶）是一个适用于Web和独立环境的现代化服务器端Java模板引擎，官方文档：https://www.thymeleaf.org/documentation.html。

那么它和JSP相比，好在哪里呢，我们来看官网给出的例子：

```
<table>
  <thead>
    <tr>
```

```
      <th th:text="#{msgs.headers.name}">Name</th>
      <th th:text="#{msgs.headers.price}">Price</th>
    </tr>
  </thead>
  <tbody>
    <tr th:each="prod: ${allProducts}">
      <td th:text="${prod.name}">Oranges</td>
      <td th:text="${#numbers.formatDecimal(prod.price, 1, 2)}">0.99</td>
    </tr>
  </tbody>
</table>
```

我们可以在前端页面中填写占位符，而这些占位符的实际值则由后端进行提供，这样，我们就不用再像JSP那样前后端都写在一起了。

那么我们来创建一个例子感受一下，首先还是新建一个项目，注意，在创建时，勾选Thymeleaf依赖。

首先编写一个前端页面，名称为`test.html`，注意，是放在resource目录下，在html标签内部添加`xmlns:th="http://www.thymeleaf.org"`引入Thymeleaf定义的标签属性：

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <div th:text="${title}"></div>
</body>
</html>
```

接着我们编写一个Servlet作为默认页面：

```
@WebServlet("/index")
public class HelloServlet extends HttpServlet {

    TemplateEngine engine;
    @Override
    public void init() throws ServletException {
        engine = new TemplateEngine();
        ClassLoaderTemplateResolver r = new ClassLoaderTemplateResolver();
        engine.setTemplateResolver(r);
    }

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        Context context = new Context();
        context.setVariable("title", "我是标题");
        engine.process("test.html", context, resp.getWriter());
    }
}
```

我们发现，浏览器得到的页面，就是已经经过模板引擎解析好的页面，而我们的代码依然是后端处理数据，前端展示数据，因此使用Thymeleaf就能够使得当前Web应用程序的前后端划分更加清晰。

虽然Thymeleaf在一定程度上分离了前后端，但是其依然是在后台渲染HTML页面并发送给前端，并不是真正意义上的前后端分离。

Thymeleaf语法基础

那么，如何使用Thymeleaf呢?

首先我们看看后端部分，我们需要通过`TemplateEngine`对象来将模板文件渲染为最终的HTML页面:

```
TemplateEngine engine;
@Override
public void init() throws ServletException {
    engine = new TemplateEngine();
    //设定模板解析器决定了从哪里获取模板文件，这里直接使用ClassLoaderTemplateResolver表示加载内部资源文件
    ClassLoaderTemplateResolver r = new ClassLoaderTemplateResolver();
    engine.setTemplateResolver(r);
}
```

由于此对象只需要创建一次，之后就可以一直使用了。接着我们来看如何使用模板引擎进行解析:

```
@Override
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
    //创建上下文，上下文中包含了所有需要替换到模板中的内容
    Context context = new Context();
    context.setVariable("title", "<h1>我是标题</h1>");
    //通过此方法就可以直接解析模板并返回响应
    engine.process("test.html", context, resp.getWriter());
}
```

操作非常简单，只需要简单几步配置就可以实现模板的解析。接下来我们就可以在前端页面中通过上下文提供的内容，来将Java代码中的数据解析到前端页面。

接着我们来了解Thymeleaf如何为普通的标签添加内容，比如我们示例中编写的:

```
<div th:text="${title}"></div>
```

我们使用了`th:text`来为当前标签指定内部文本，注意任何内容都会变成普通文本，即使传入了一个HTML代码，如果我希望向内部添加一个HTML文本呢？我们可以使用`th:utext`属性:

```
<div th:utext="${title}"></div>
```

并且，传入的title属性，不仅仅只是一个字符串的值，而是一个字符串的引用，我们可以直接通过此引用调用相关的方法:

```
<div th:text="${title.toLowerCase()}"></div>
```

这样看来，Thymeleaf既能保持JSP为我们带来的便捷，也能兼顾前后端代码的界限划分。

除了替换文本，它还支持替换一个元素的任意属性，我们发现，`th:`能够拼接几乎所有的属性，一旦使用`th:属性名称`，那么属性的值就可以通过后端提供了，比如我们现在想替换一个图片的链接：

```
@Override
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
    Context context = new Context();
    context.setVariable("url",
"http://n.sinaimg.cn/sinakd20121/600/w1920h1080/20210727/a700-
adf8480ff24057e04527bdfea789e788.jpg");
    context.setVariable("alt", "图片就是加载不出来啊");
    engine.process("test.html", context, resp.getWriter());
}
```

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <img width="700" th:src="${url}" th:alt="${alt}">
</body>
</html>
```

现在访问我们的页面，就可以看到替换后的结果了。

Thymeleaf还可以进行一些算术运算，几乎Java中的运算它都可以支持：

```
<div th:text="${value % 2}"></div>
```

同样的，它还支持三元运算：

```
<div th:text="${value % 2 == 0 ? 'yyds' : 'lbwnb'}"></div>
```

多个属性也可以通过`+`进行拼接，就像Java中的字符串拼接一样，这里要注意一下，字符串不能直接写，要添加单引号：

```
<div th:text="${name}+' 我是文本 '+${value}"></div>
```

Thymeleaf流程控制语法

除了一些基本的操作，我们还可以使用Thymeleaf来处理流程控制语句，当然，不是直接编写Java代码的形式，而是添加一个属性即可。

首先我们来看if判断语句，如果if条件满足，则此标签留下，若if条件不满足，则此标签自动被移除：

```
@Override
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
    Context context = new Context();
    context.setVariable("eval", true);
    engine.process("test.html", context, resp.getWriter());
}
```

```
<div th:if="${eval}">我是判断条件标签</div>
```

`th:if`会根据其中传入的值或是条件表达式的结果进行判断，只有满足的情况下，才会显示此标签，具体的判断规则如下：

- 如果值不是空的：
    - 如果值是布尔值并且为`true`。
    - 如果值是一个数字，并且是非零
    - 如果值是一个字符，并且是非零
    - 如果值是一个字符串，而不是“错误”、“关闭”或“否”
    - 如果值不是布尔值、数字、字符或字符串。
- 如果值为空，th:if将计算为false

`th:if`还有一个相反的属性`th:unless`，效果完全相反，这里就不演示了。

我们接着来看多分支条件判断，我们可以使用`th:switch`属性来实现：

```
<div th:switch="${eval}">
    <div th:case="1">我是1</div>
    <div th:case="2">我是2</div>
    <div th:case="3">我是3</div>
</div>
```

只不过没有default属性，但是我们可以使用`th:case="*"`来代替：

```
<div th:case="*">我是Default</div>
```

最后我们再来看看，它如何实现遍历，假如我们有一个存放书籍信息的List需要显示，那么如何快速生成一个列表呢？我们可以使用`th:each`来进行遍历操作：

```
@Override
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
    Context context = new Context();
    context.setVariable("list", Arrays.asList("伞兵一号的故事", "倒一杯卡布奇诺", "玩游戏要啃着玩", "十七张牌前的电脑屏幕"));
    engine.process("test.html", context, resp.getWriter());
}
```

```
<ul>
    <li th:each="title : ${list}" th:text="'《'+${title}+'》'"></li>
</ul>
```

`th:each`中需要填写 "单个元素名称 : ${列表}"，这样，所有的列表项都可以使用遍历的单个元素，只要使用了`th:each`，都会被循环添加。因此最后生成的结果为：

```
<ul>
        <li>《伞兵一号的故事》</li>
        <li>《倒一杯卡布奇诺》</li>
        <li>《玩游戏要啃着玩》</li>
        <li>《十七张牌前的电脑屏幕》</li>
    </ul>
```

我们还可以获取当前循环的迭代状态，只需要在最后添加`iterStat`即可，从中可以获取很多信息，比如当前的顺序：

```
<ul>
    <li th:each="title, iterStat : ${list}" th:text="${iterStat.index}+'.
《'+${title}+'》'"></li>
</ul>
```

状态变量在`th:each`属性中定义，并包含以下数据：

- 当前*迭代索引*，以0开头。这是`index`属性。
- 当前*迭代索引*，以1开头。这是`count`属性。
- 迭代变量中的元素总量。这是`size`属性。
- 每个迭代的*迭代变量*。这是`current`属性。
- 当前迭代是偶数还是奇数。这些是`even/odd`布尔属性。
- 当前迭代是否是第一个迭代。这是`first`布尔属性。
- 当前迭代是否是最后一个迭代。这是`last`布尔属性。

通过了解了流程控制语法，现在我们就可以很轻松地使用Thymeleaf来快速替换页面中的内容了。

Thymeleaf模板布局

在某些网页中，我们会发现，整个网站的页面，除了中间部分的内容会随着我们的页面跳转而变化外，有些部分是一直保持一个状态的，比如打开小破站，我们翻动评论或是切换视频分P的时候，变化的仅仅是对应区域的内容，实际上，其他地方的内容会无论内部页面如何跳转，都不会改变。

Thymeleaf就可以轻松实现这样的操作，我们只需要将不会改变的地方设定为模板布局，并在不同的页面中插入这些模板布局，就无需每个页面都去编写同样的内容了。现在我们来创建两个页面：

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <div class="head">
        <div>
            <h1>我是标题内容，每个页面都有</h1>
        </div>
        <hr>
    </div>
    <div class="body">
```

```
        <ul>
            <li th:each="title, iterStat : ${list}"
th:text="${iterStat.index}+'.《'+${title}+'》'"></li>
        </ul>
    </div>
</body>
</html>
```

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <div class="head">
        <div>
            <h1>我是标题内容，每个页面都有</h1>
        </div>
        <hr>
    </div>
    <div class="body">
        <div>这个页面的样子是这样的</div>
    </div>
</body>
</html>
```

接着将模板引擎写成工具类的形式：

```
public class ThymeleafUtil {

    private static final TemplateEngine engine;
    static  {
        engine = new TemplateEngine();
        ClassLoaderTemplateResolver r = new ClassLoaderTemplateResolver();
        engine.setTemplateResolver(r);
    }

    public static TemplateEngine getEngine() {
        return engine;
    }
}
```

```
@WebServlet("/index2")
public class HelloServlet2 extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        Context context = new Context();
        ThymeleafUtil.getEngine().process("test2.html", context,
resp.getWriter());
    }
}
```

现在就有两个Servlet分别对应两个页面了，但是这两个页面实际上是存在重复内容的，我们要做的就是将这些重复内容提取出来。

我们单独编写一个`head.html`来存放重复部分：

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org" lang="en">
<body>
    <div class="head" th:fragment="head-title">
        <div>
            <h1>我是标题内容，每个页面都有</h1>
        </div>
        <hr>
    </div>
</body>
</html>
```

现在，我们就可以直接将页面中的内容快速替换：

```
<div th:include="head.html::head-title"></div>
<div class="body">
    <ul>
        <li th:each="title, iterStat : ${list}" th:text="${iterStat.index}+'.
《'+${title}+'》'"></li>
    </ul>
</div>
```

我们可以使用`th:insert`和`th:replace`和`th:include`这三种方法来进行页面内容替换，那么`th:insert`和`th:replace`（和`th:include`，自3.0年以来不推荐）有什么区别？

- `th:insert`最简单：它只会插入指定的片段作为标签的主体。
- `th:replace`实际上将标签直接*替换*为指定的片段。
- `th:include`和`th:insert`相似，但它没有插入片段，而是只插入此片段*的内容*。

你以为这样就完了吗？它还支持参数传递，比如我们现在希望插入二级标题，并且由我们的子页面决定：

```
<div class="head" th:fragment="head-title">
    <div>
        <h1>我是标题内容，每个页面都有</h1>
        <h2>我是二级标题</h2>
    </div>
    <hr>
</div>
```

稍加修改，就像JS那样添加一个参数名称：

```
<div class="head" th:fragment="head-title(sub)">
    <div>
        <h1>我是标题内容，每个页面都有</h1>
        <h2 th:text="${sub}"></h2>
    </div>
    <hr>
</div>
```

现在直接在替换位置添加一个参数即可：

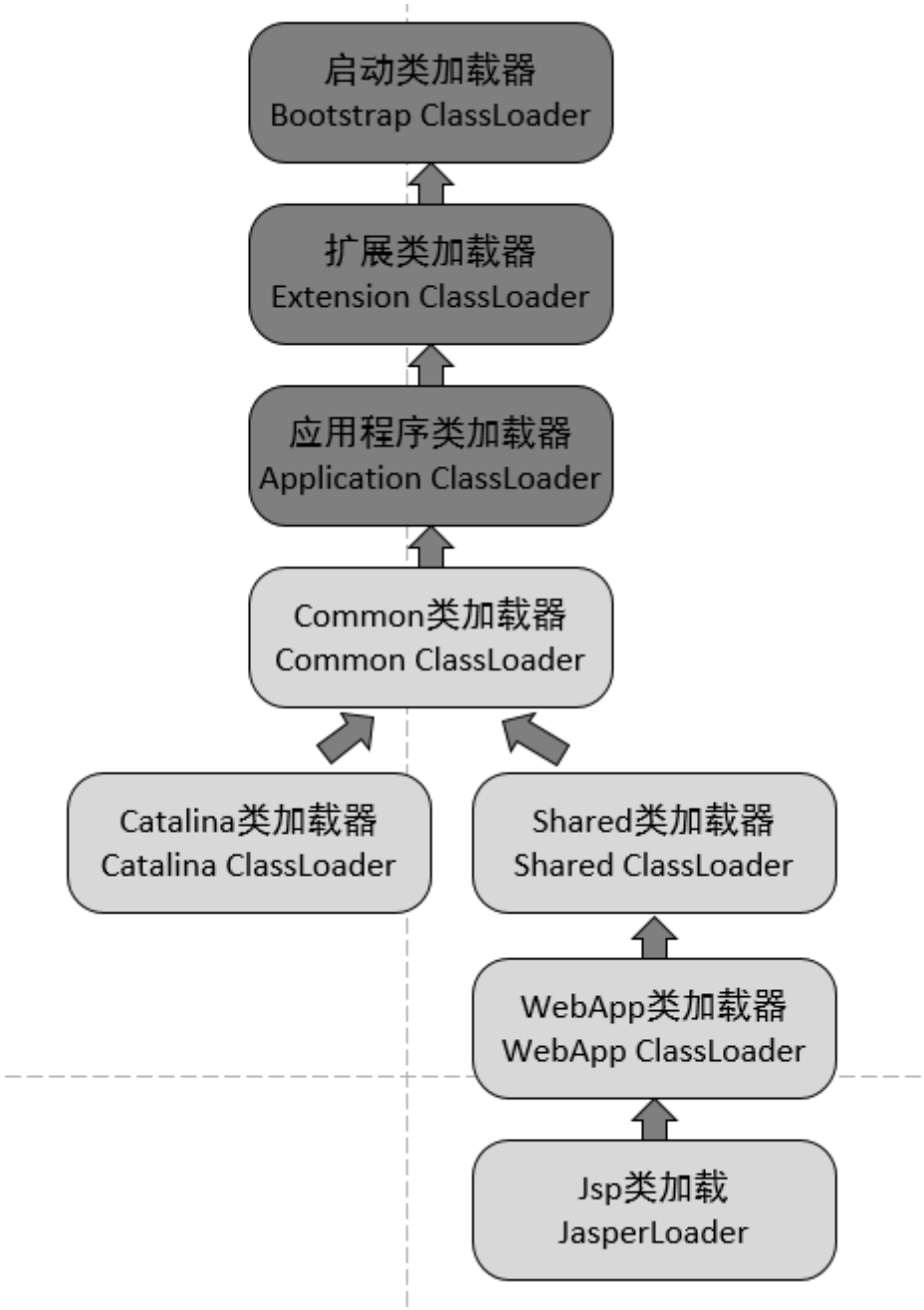
```
<div th:include="head.html::head-title('这个是第1个页面的二级标题')"></div>
<div class="body">
    <ul>
        <li th:each="title, iterStat : ${list}" th:text="${iterStat.index}+'.
《'+${title}+'》'"></li>
    </ul>
</div>
```

这样，不同的页面还有着各自的二级标题。

探讨Tomcat类加载机制

有关JavaWeb的内容，我们就聊到这里，在最后，我们还是来看一下Tomcat到底是如何加载和运行我们的Web应用程序的。

Tomcat服务器既然要同时运行多个Web应用程序，那么就必须要实现不同应用程序之间的隔离，也就是说，Tomcat需要分别去加载不同应用程序的类以及依赖，还必须保证应用程序之间的类无法相互访问，而传统的类加载机制无法做到这一点，同时每个应用程序都有自己的依赖，如果两个应用程序使用了同一个版本的同一个依赖，那么还有必要去重新加载吗，带着诸多问题，Tomcat服务器编写了一套自己的类加载机制。

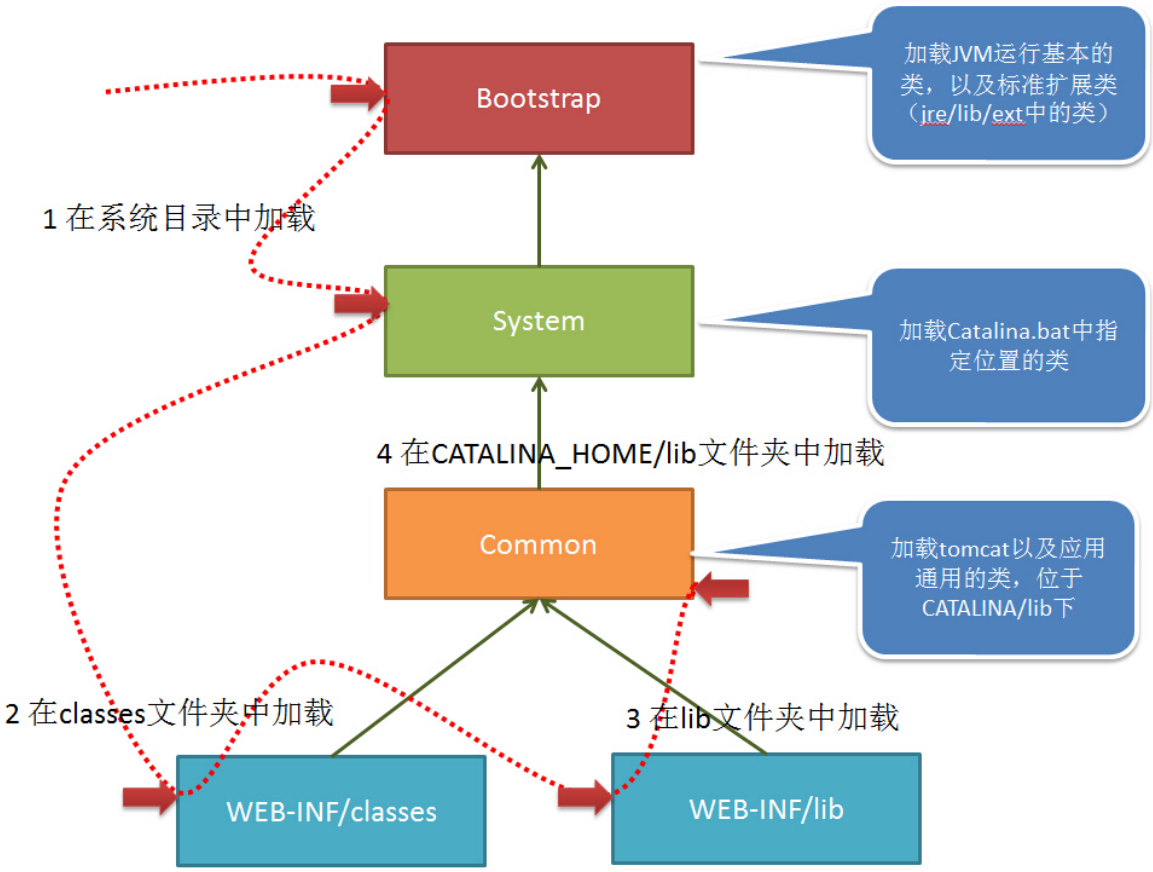


首先我们要知道，Tomcat本身也是一个Java程序，它要做的是去动态加载我们编写的Web应用程序中的类，而要解决以上提到的一些问题，就出现了几个新的类加载器，我们来看看各个加载器的不同之处：

- Common ClassLoader：Tomcat最基本的类加载器，加载路径中的class可以被Tomcat容器本身以及各个Web应用程序访问。
- Catalina ClassLoader：Tomcat容器私有的类加载器，加载路径中的class对于Web应用程序不可见。
- Shared ClassLoader：各个Web应用程序共享的类加载器，加载路径中的class对于所有Web应用程序可见，但是对于Tomcat容器不可见。
- Webapp ClassLoader：各个Web应用程序私有的类加载器，加载路径中的class只对当前Web应用程序可见，每个Web应用程序都有一个自己的类加载器，此加载器可能存在多个实例。
- JasperLoader：JSP类加载器，每个JSP文件都有一个自己的类加载器，也就是说，此加载器可能会存在多个实例。

通过这样进行划分，就很好地解决了我们上面所提到的问题，但是我们发现，这样的类加载机制，破坏了JDK的双亲委派机制（在JavaSE阶段讲解过），比如Webapp ClassLoader，它只加载自己的class文件，它没有将类交给父类加载器进行加载，也就是说，我们可以随意创建和JDK同包同名的类，岂不是就出问题了？

难道Tomcat的开发团队没有考虑到这个问题吗?



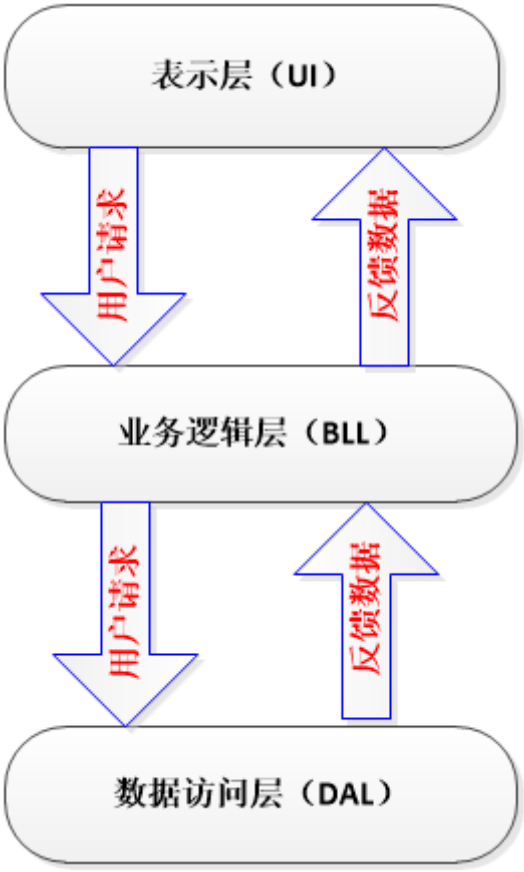
实际上，WebAppClassLoader的加载机制是这样的：WebAppClassLoader 加载类的时候，绕开了AppClassLoader，直接先使用 ExtClassLoader 来加载类。这样的话，如果定义了同包同名的类，就不会被加载，而如果是自己定义 的类，由于该类并不是JDK内部或是扩展类，所有不会被加载，而是再次回到WebAppClassLoader进行加载，如果还失败，再使用AppClassloader进行加载。

实战：编写图书管理系统

图书管理系统需要再次迎来升级，现在，我们可以直接访问网站来操作图书，这里我们给大家提供一个前端模板直接编写，省去编写前端的时间。

本次实战使用到的框架：Servlet+Mybatis+Thymeleaf

注意在编写的时候，为了使得整体的代码简洁高效，我们严格遵守三层架构模式：



就是说，表示层只做UI，包括接受请求和相应，给模板添加上下文，以及进行页面的解析，最后响应给浏览器；业务逻辑层才是用于进行数据处理的地方，表示层需要向逻辑层索要数据，才能将数据添加到模板的上下文中；数据访问层一般就是连接数据库，包括增删改查等基本的数据库操作，业务逻辑层如果需要从数据库取数据，就需要向数据访问层请求数据。

当然，贯穿三大层次的当属实体类了，我们还需要创建对应的实体类进行数据的封装，以便于在三层架构中进行数据传递。

接下来，明确我们要实现的功能，也就是项目需求：

- 图书管理员的登陆和退出（只有登陆之后才能进入管理页面）
- 图书的列表浏览（包括书籍是否被借出的状态也要进行显示）以及图书的添加和删除
- 学生的列表浏览
- 查看所有的借阅列表，添加借阅信息

结束语

首先祝贺各位顺利完成了JavaWeb相关知识的学习。

本教程创作的动力离不开各位观众姥爷们的支持，我们也会在后面为大家录制更多的Java技术栈教程，如果您喜欢本系列视频的话，直接用三连狠狠的砸向UP主吧！

虽然我们现在已经学会了如何去编写一个网站，但是实际上，这样的开发模式已经过时（不过拿去当毕设当期末设计直接无敌好吧），我们还需要继续深入了解更加现代化的开发模式，这样我们才有机会参与到企业的项目开发当中。

希望在后续的视频中，还能看到各位的身影，完结撒花！