南京信息工程大学

《计算机网络》课程设计

题目 HTTP Web 服务器

学年学期 2019-2020 学年第一学期

课程名称 计算机网络课程设计

院　　系 计算机与软件学院

专　　业 计算机科学与技术

学　　号 201733050050

姓　　名 万宇轩

指导教师 黄群

二〇一九 年 十二月 二十 日

1. 引言

1.1. 课题说明

本次课程设计的目标是设计并实现一个基本满足 HTTP 协议的 Web 服务器。在课程设计中，使用 Java 作为本课题的开发平台。

1.2. 技术介绍

1）HTTP 协议

HTTP 协议是 Hyper Text Transfer Protocol（超文本传输协议）的缩写,是一个用于从万维网服务器传输超文本到本地浏览器的传送协议，它基于 TCP/IP 通信协议来传输数据，是一个属于应用层面的面向对象的协议。

HTTP 协议主要工作在 C/S 架构上，一般来说，使用浏览器作为客户端来发送 HTTP 请求，Web 服务器在接收到客户端发来的请求后，会进行响应操作。

HTTP 协议每一次通信过程都是由一个请求体（Request），以及一个响应消息（Response）所组成。

在 Request 中主要包括：请求行（request line）、请求头部（header）、空行和请求数据这四个部分。

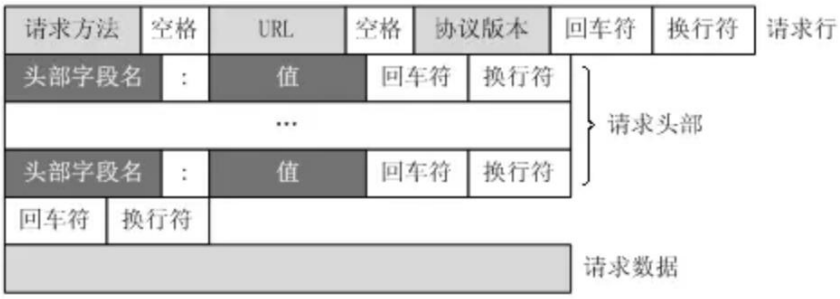

图 1 Request 请求结构

在每一个返回给客户端的 Response 中，也是由四部分所组成，分别是：状态行、消息报头、空行和响应正文。

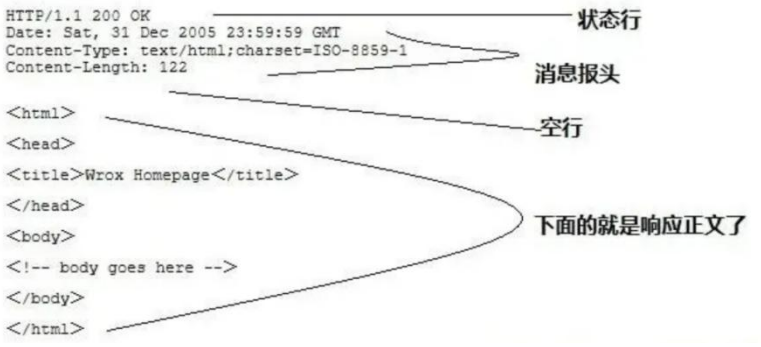

图 2 Response 响应体结构

2）TCP/IP 协议

TCP/IP（Transmission Control Protocol/Internet Protocol）即传输控制协议/网间协议，定义了主机如何连入因特网及数据如何再它们之间传输的标准，从字面意思来看 TCP/IP 是 TCP 和 IP 协议的合称，但实际上 TCP/IP 协议是指因特网整个 TCP/IP 协议族。不同于 ISO 模型的七个分层，TCP/IP 协议参考模型把所有的 TCP/IP 系列协议归类到四个抽象层中：

应用层：TFTP，HTTP，SNMP，FTP，SMTP，DNS，Telnet....

传输层：TCP，UDP

网络层：IP，ICMP，OSPF，EIGRP，IGMP

数据链路层：SLIP，CSLIP，PPP，MTU

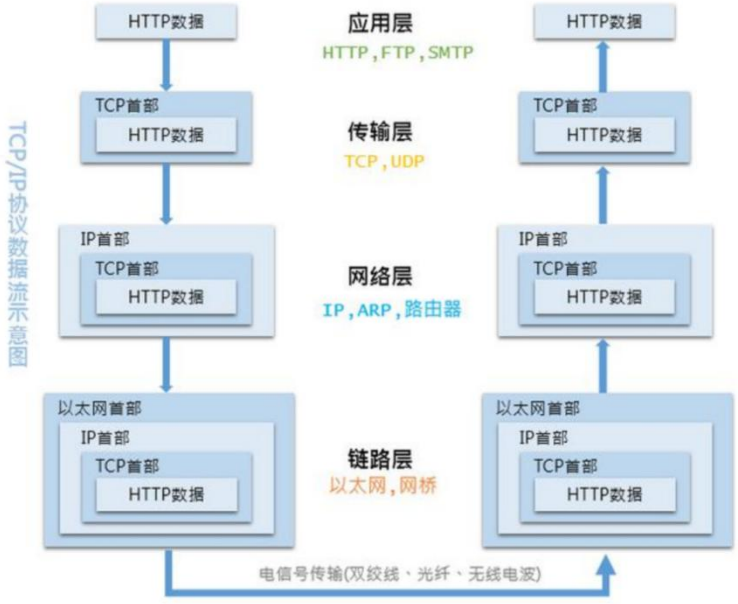

图 3 TCP/IP 协议簇

3）Socket

Socket 是应用层与 TCP/IP 协议族通信的中间软件抽象层，它其实就是是一组接口规范。在设计模式中，Socket 其实就是一个门面模式，即外部与一个子系统的通信必须通过一个统一的门面对象进行。Socket 把复杂的 TCP/IP 协议族隐藏在 Socket 接口后面，对用户来说，一组简单的接口就是全部，让 Socket 去组织数据，以符合指定的协议。

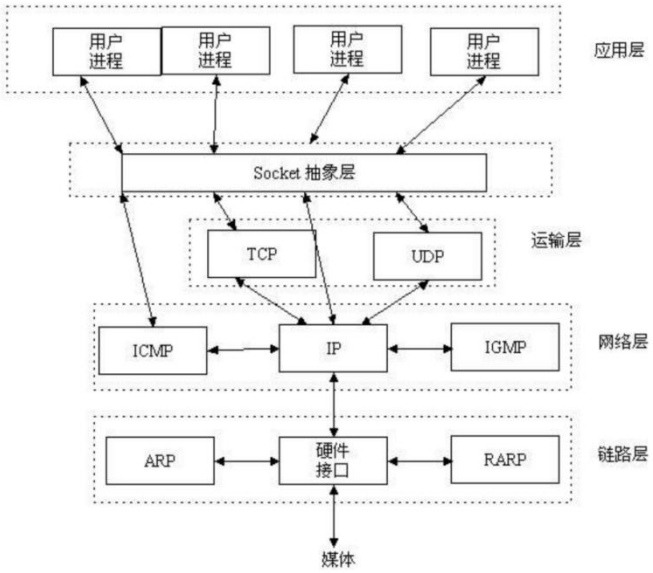

图 4 Socket 接口

4) Servlet

Servlet 是一种独立于平台和协议的服务器端的 Java 应用程序，可以生成动态的 web 页面。它担当 Web 浏览器或其他 HTTP 客户程序发出请求、与 HTTP 服务器上的数据库或应用程序之间交互的中间层。

5) Cookie

Cookie 是浏览器储存在用户电脑上的一小段文本文件，每一个 Web 页面或服务器会通过 Response 来告知浏览器按照一定规范来储存这些信息，之后，浏览器在每一次请求中将这些信息发送至服务器，Web 服务器就可以使用这些信息来识别不同的用户。大多数需要登录的网站在用户验证成功之后都会设置一个 cookie，只要这个 cookie 存在并没有过期，用户就可以自由浏览这个网站的任意页面。

6) Session

Session 在计算机中，尤其是在网络应用中，称为“会话控制”。Session 对象存储特定用户会话所需的属性及配置信息。这样，当用户在应用程序的 Web 页之间跳转时，存储在 Session 对象中的变量将不会丢失，而是在整个用户会话中一直存在下去。当用户请求来自应用程序的 Web 页时，如果该用户还没有会话，则 Web 服务器将自动创建一个 Session 对象。当会话过期或被放弃后，服务器将终止该会话。Session 对象最常见的一个用法就是存储用户的首选项。

Session 与 Cookie 都是一种用于维持客户端与服务器会话状态的技术，但 Session 的安全性要高于 Cookie，因为 Session 是存放于服务器中，这就提高了数据

的安全性。

2. 功能介绍

本课程设计实现的 HTTP Web 服务器主要实现了以下几个功能：

1. 使用多线程技术，支持多请求的同时访问。
2. 对客户端发送的 HTTP 请求进行解析处理，并返回响应体。
3. 支持服务器内部转发与请求重定向。
4. 实现了 Servlet 容器，可以读取并解析 Servlet 配置，支持 Servlet 编程。
5. 支持静态资源的传输。
6. 实现 Session 机制。

3. 系统设计与实现

3.1. 程序整体流程图

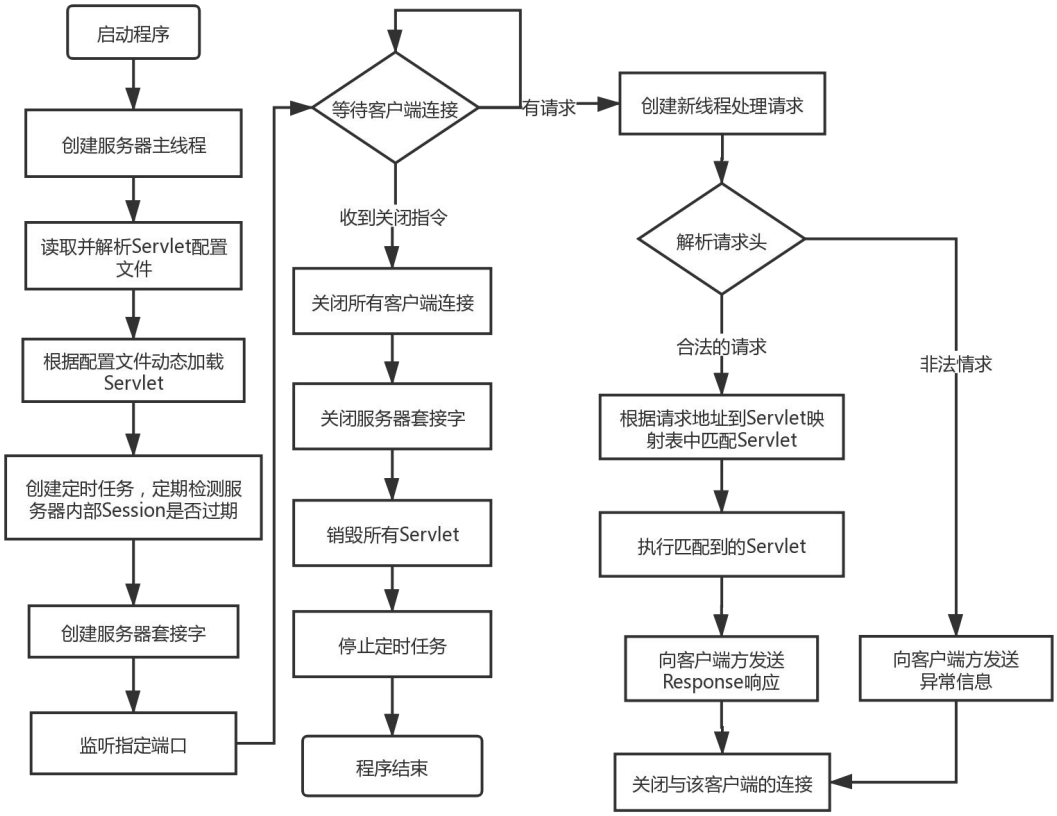

图 5 服务器整体流程图

3.2. 核心模块设计与实现

1）请求解析

对于每一个请求报文的解析主要分为两个部分，请求头部解析与请求体解析。请求头部的解析又分为三个部分，即首行解析，Cookie 解析以及其它部分数据的解

析处理。

当服务器接收到请求后，首先将请求数据全部预读取到数据缓冲区中，以便后期解析处理，然后对请求头的每一行进行解析操作，将解析后的数据使用键值进行保存，头部解析完后，判断是否有 Cookie 字段，如果存在则对 Cookie 字段进行解析，否则判断是否含有请求体数据，如果有则将数据进行解析。

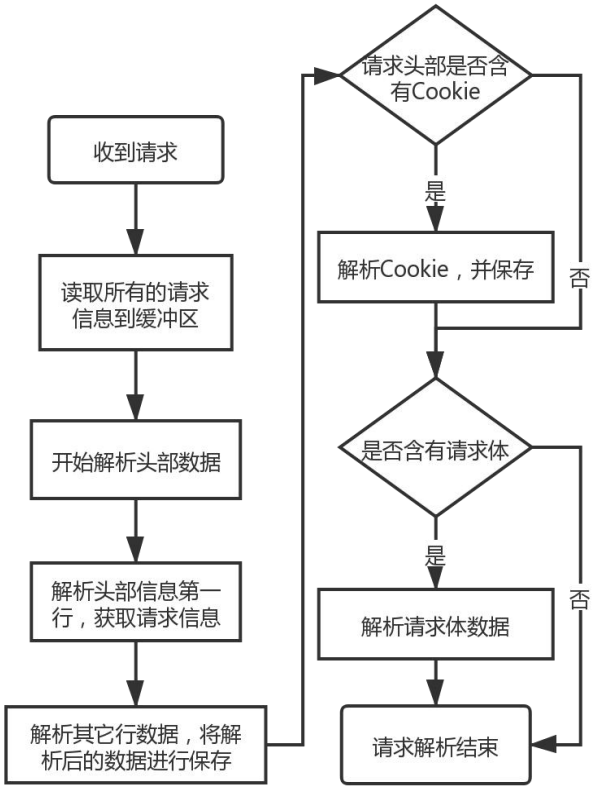

图 6 请求报文解析流程图

请求头部的首行主要由三部分组成，即请求方式，请求路径信息以及协议版本，它们之间使用空格分隔，所以可以使用空格对其进行分割处理，其中请求路径可能包含请求参数等信息，还需要进一步进行解析读取。

```
1.    private void analysis(String line) throws RequestParseException {
2.        //将首行按照空格进行分割
3.        String[] firstLine = line.split(CharContest.BLANK);
4.        //判断请求方式
5.        this.method = RequestMethod.valueOf(firstLine[0].toUpperCase());
6.        log.info("method:{}", this.method);
7.        //解析路径
8.        String pathAndParams = firstLine[1];
9.        int start = pathAndParams.indexOf("?");
10.       if (start != -1) {
11.           //包含参数信息
```

```
            this.url = pathAndParams.substring(0, start);
            log.info("path:{}", this.url);
            String params = pathAndParams.substring(start + 1);
            //解析 url 参数
            this.params = this.parseParams(params);
            log.info("params:{}", this.params);
        } else {
            this.url = pathAndParams;
        }
    }

    //解析参数信息
private HashMap<String, String> parseParams(String params) throws
RequestParseException {
        if (params == null) {
            return new HashMap<>();
        }
        String[] paramList = params.split("&");//将参数进行拆分
        HashMap<String, String> paramsMap = new HashMap<>();
        for (String param : paramList) {
            String[] oneParam = param.split("=");
            if (oneParam.length != 2) {
                throw new RequestParseException();
            }
            paramsMap.put(oneParam[0].trim().toLowerCase(), oneParam[1]);
        }
        return paramsMap;
    }
```

请求头部的其它行的格式比较统一，都是“key:value”的格式，所以解析起来比较方便，可以进行统一解析。

```
//解析 head 的其它行
for (; lineNum < lines.length; lineNum++) {
    if (lines[lineNum].length() == 0) {
        //如果是空行直接退出
        break;
    }
    //提取请求报文的每一行属性值
    int colonIndex = lines[lineNum].indexOf(':');
    if (colonIndex == -1) {
        throw new RequestParseException();
    }
    String key = lines[lineNum].substring(0, colonIndex).toLowerCase().trim()
;
```

```
13.     String value = lines[lineNum].substring(colonIndex + 1).trim();
14.     this.headers.put(key, value);
15. }
```

每一个请求体的 Cookie 是存放在请求头的 Cookie 字段中，这个字段是由一个个 Cookie 键值对所组成，解析时首先按照分号将它们进行分割，然后读取每一个键值对即可。

```
1.  if (this.headers.containsKey("cookie")) {
2.      //如果存在 cookie，则对 cookie 进行解析操作
3.      String cookies = this.headers.get("cookie");
4.      //使用分号进行分割
5.      String[] cookieList = cookies.split(SEMICOLON);
6.      for (String rawCookie : cookieList) {
7.          //遍历键值对
8.          String[] cookieNameVal = rawCookie.trim().split(EQUATION);
9.          //按照等号分割
10.         if (cookieNameVal.length < 2) {
11.             //不合法，则抛弃该 cookie
12.             continue;
13.         }
14.         //将 cookie 进行实例化
15.         Cookie cookie = new Cookie(cookieNameVal[0], cookieNameVal[1]);
16.         cookieMap.put(cookieNameVal[0], cookie);
17.     }
18. }
```

对于请求体只对表单提交的数据进行解析，其它类型的数据不做处理，交由用户处理。

```
1.  //解析 body 体数据
2.  private void parseBody(String body) throws RequestParseException {
3.      //将 body 数据读取出来
4.      int len = Math.min(body.length(), Integer.parseInt(this.getHeader("conte
    nt-length")));
5.      this.body = body.substring(0, len);
6.      String contentType = getHeader("content-type");
7.      if (contentType.equals(HTTPConstant.TYPE_FORM) || contentType.equals(HTT
    PConstant.TYPE_TEXT)) {
8.          //表单格式，则直接调用表单解析
9.          this.params.putAll(this.parseParams(this.body));
10.     }
11.     log.info("body 数据：{}", this.body);
12. }
```

2）请求响应

当每一次请求处理完成后，Servlet 将调用响应 Response 的 write 函数，如果该响应有数据要发送给客户端，则该函数会将服务器的响应数据写入到响应体中，同时将数据的类型以及长度添加到响应头中，随后 write 函数将调用 buildHeader 函数来构建响应头部信息，主要包含响应日期，编码格式，响应体数据结构类型，响应体长度以及 Cookie 等信息。

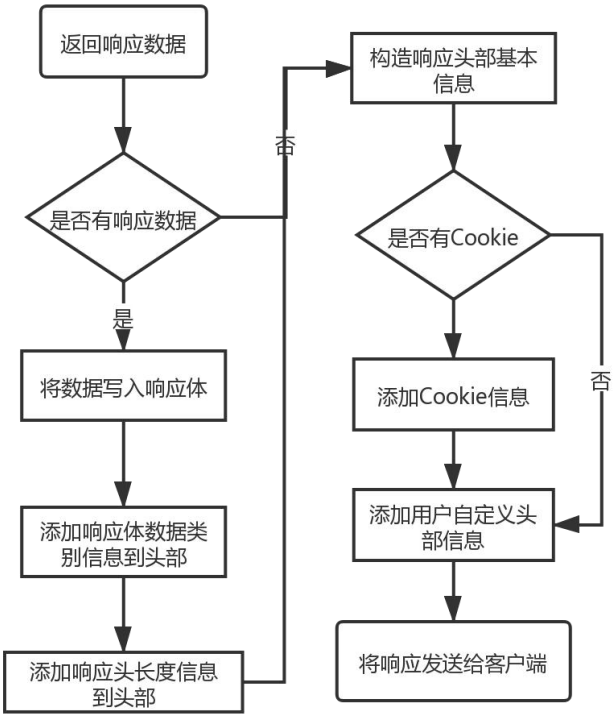

图 7 响应构造流程图

```
1.  //构造 response 的头部信息
2.  private void buildHeader() {
3.      //添加默认头部信息
4.      response.append("HTTP/1.1").append(BLANK).
5.              append(httpStatus.getCode()).append(BLANK).append(httpStatus).ap
    pend(CRLF);
6.       response.append("Date:").append(BLANK).append(new Date()).append(CRLF);

7.      if (charset != null) {
8.          String type = contentType;
9.          int charSetIndex = contentType.indexOf("charset");
10.         if (charSetIndex != -1) {
11.             //存在 charset，将原来的 charset 替换掉
12.             type = contentType.substring(0, charSetIndex);
13.         }
14.         if (type.contains(SEMICOLON)) {
15.             contentType = type + "charset=" + charset;
16.         } else {
```

```
            contentType = type + SEMICOLON + "charset=" + charset;
        }
    }
     response.append("Content-Type:").append(BLANK).append(contentType).append(CRLF);
    if (hasBody) {
        //如果有 body 的话，添加 body 体长度信息
        response.append("Content-Length:").append(BLANK).append(body.length).append(CRLF);
    }
    if (!cookies.isEmpty()) {
        //如果有 cookie 的话
        StringBuilder cookieStr = new StringBuilder();
        for (Cookie cookie : cookies) {
            cookieStr.append(cookie.toString()).append(BLANK);
        }
         response.append("Set-Cookie:").append(BLANK).append(cookieStr).append(CRLF);
    }
    for (Header header : headers) {//添加用户头部信息
         response.append(header.getKey()).append(":").append(BLANK).append(header.getVal()).append(CRLF);
    }
    response.append(CRLF);//添加空行
    log.info("header:{}", response);
}
```

```
private void sendToClient() {
    //发送数据给客户端
    try {
        this.buildHeader();
        byte[] header = response.toString().getBytes(StandardCharsets.UTF_8);

        this.writer.write(header);
        if (hasBody) {
            this.writer.write(body);
        }
        this.writer.flush();
    } catch (IOException e) {
        e.printStackTrace();
        log.error("写回客户端失败！");
    }
}
```

3）Session 机制

Session 机制主要用于保存用户在服务器的会话状态，它需要和 Cookie 机制进行配合使用。客户端浏览器在每一次访问服务器的时候都会在 Cookie 中携带 Session ID，服务器通过读取 Cookie 中的 Session ID，从而识别该请求属于哪一个用户，以此将同一个用户的不同请求都视为同一个会话。

Session 机制默认是不开启状态，如果需要开启，则需要调用 Request 的 getSession 函数，同时它有一个 Boolean 类型的参数，表示如果 Session 不存在，则进行创建操作。

在调用 getSession 函数后，首先会判断 Cookie 中是否存在 Session ID，如果存在则读取 ID，并到 ServletContext 的 Session 表中查找对应 Session ID 的 Session 实例；如果不存在，则会根据传入的 Boolean 参数来判断是否需要创建 Session。

在创建 Session 的时候，会调用 ServletContext 中的 createSession 函数来创建用户 Session，并将创建的 Session 实例保存到 ServletContext 的 Session 表中，同时会调用 Response 的 addCookie 函数，将生成的 Session ID 作为 Cookie 值传给浏览器。

由于 Session 有着过期机制，即如果保存在 ServletContext 的 Session 表中的 Session 实例过期了，那么将会被清除，所以还有一个定时任务用于定期检查 Session 表中的实例是否过期，如果过期则进行销毁操作。

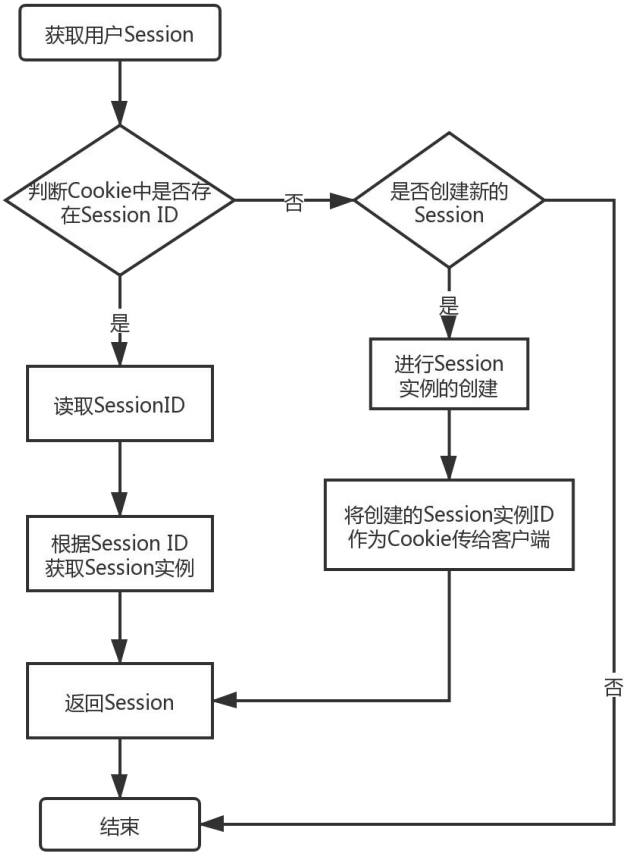

图 8 Session 获取流程图

```
1.  public HttpSession getSession(boolean createIfNone) {
2.      if (this.session != null) {
3.          //如果 Session 已经有了，则直接返回
4.          return this.session;
5.      }
6.      if (cookieMap.containsKey(SESSION)) {
7.          //判断 Cookie 中是否存在 Session ID
8.          Cookie cookie = cookieMap.get(SESSION);
9.          String sessionId = cookie.getValue();//获取 sessionId
10.          HttpSession currentSession = WebApplicationContext.getServletContext
    ().
11.                  getSession(sessionId);//使用 ID 获取 Session 实例
12.          if (currentSession != null) {
13.              this.session = currentSession;
14.              return this.session;
15.          }
16.      }
17.      if (createIfNone) {
18.          //如果不存在，则创建
19.          this.session = WebApplicationContext.getServletContext().createSessi
    on(this.container.getResponse());
20.      }
21.      return this.session;
22.  }
```

创建 Session：

```
1.  //创建 session
2.  public HttpSession createSession(HttpServletResponse response) {
3.      String sessionId = UUID.randomUUID().toString();
4.      HttpSession session = new HttpSession(sessionId);
5.      this.sessionMap.put(sessionId, session);
6.      response.addCookie(new Cookie(SESSION, sessionId));
7.      return session;
8.  }
```

检测 Session 是否过期：

```
1.  public void detectSessionIsValid() {
2.      //检测 Session 是否过期
3.      if (!sessionMap.isEmpty()) {
4.          Set<String> sessionKeySet = sessionMap.keySet();
5.          for (String key : sessionKeySet) {
6.              HttpSession session = sessionMap.get(key);
7.              //计算当前时间减去 session 的最大过期时间，该结果表示，如果该 session 是
    不过期的，那么它的活动时间应该在 lastAccessedTime 与 now 之间。
```

```
8.              Instant lastAccessedTime = Instant.now().minusSeconds(session.ge
    tMaxInactiveInterval());
9.              log.info("检测 session，sessionID:{}",session.getId());
10.             if (lastAccessedTime.isAfter(session.getLastAccessed())) {
11.                 //如果 session 最后的活动时间在这之前，表示已经过期了
12.                 session.invalidate();
13.             }
14.         }
15.     } else {
16.         log.info("当前没有 session");
17.     }
18. }
```

4）Servlet 容器

本系统所实现的 Servlet 部分主要分为：Servlet 配置文件的读取与解析、请求地址与 Servlet 的匹配，Servlet 的动态加载等。

Servlet 的配置信息都保存在 web.xml 中，用户可以在其中配置自定义的 Servlet 信息，每一个 Servlet 配置项由两部分所组成，一个是 url 地址到 servlet-name 的映射，另一个是 servlet-name 到 servlet-class 的映射，这样 Servlet 容器就可以通过匹配 URL 地址来匹配相应的 Servlet。

```
1.  /**
2.   * 解析 web.xml 的配置文件
3.   */
4.  private void parseConfig() {
5.      SAXReader reader = new SAXReader();
6.      try {
7.          //加载配置文件
8.           Document document = reader.read(ServletContext.class.getResourceAsSt
    ream("/web.xml"));
9.          Element root = document.getRootElement();
10.         //保存 servlet 名字与 servlet 类名的映射
11.         Iterator<Element> servletList = root.elementIterator("servlet");
12.         while (servletList.hasNext()) {
13.             //遍历所有的 Servlet
14.             Element servlet = servletList.next();
15.             String key = servlet.elementText("servlet-name");
16.             String value = servlet.elementText("servlet-class");
17.              servlets.put(key.trim(), new ServletHolder(value));//将 servlet 名
    字与 servlet 类名保存为键值对
18.         }
19.         //遍历 servlet 名字与 url 地址的映射
```

```
20.          Iterator<Element> servletMappingList = root.elementIterator("servlet
    -mapping");
21.          while (servletMappingList.hasNext()) {
22.              Element servletMap = servletMappingList.next();
23.              String key = servletMap.elementText("url-pattern");
24.              String value = servletMap.elementText("servlet-name");
25.              servletMapping.put(key, value);//将 servlet 名字与 url 地址保存为键值
    对
26.          }
27.      } catch (DocumentException e) {
28.          e.printStackTrace();
29.          log.error("加载 servlet 配置项失败");
30.      }
31. }
```

从请求地址到指定 Servlet 的匹配分为两部分，一个是精确匹配，另一个是模糊匹配。在系统中保存有两张映射表，一个是 servletMapping 映射表，用于 URL 地址到 servlet-name 的映射,另一个是 servlets 映射表,用于 servlet-name 到 servlet-class 的映射，这两个映射表的关系基本与配置文件系保持一致。

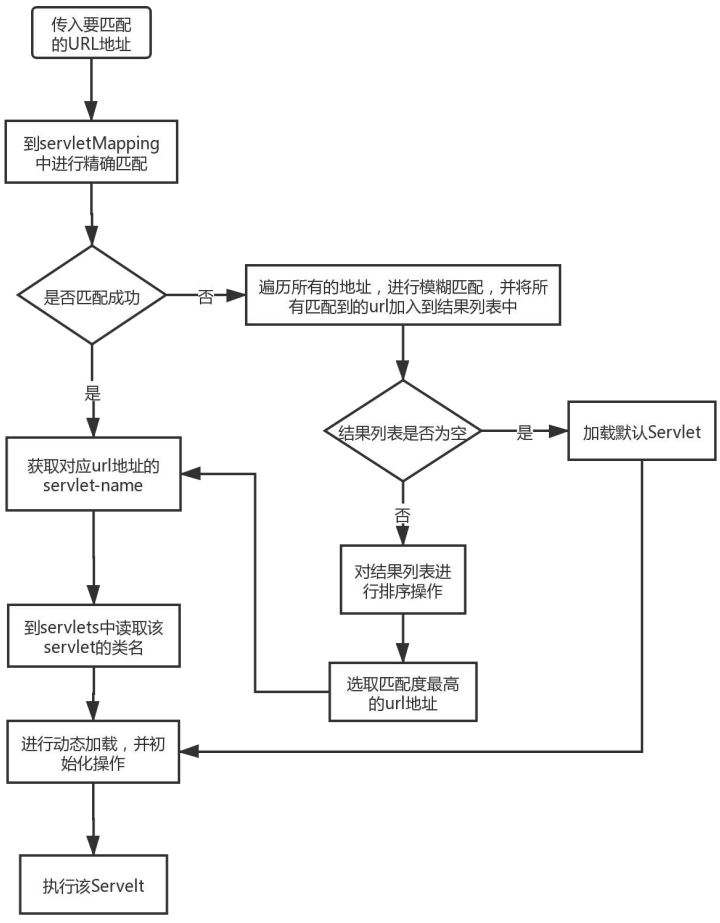

图 9 Servlet 匹配流程图

系统在得到请求的 URL 地址后，首先会到 servletMapping 中进行精确匹配，寻找与当前 URL 地址完全一致的 servlet-name，如果找到了，则再到 servlets 中寻找对应的 servlet-class，最后根据类名，使用 Java 中的反射技术，动态加载该匹配到的 Servlet;如果精确匹配的时候没有找到与 URL 匹配的 Servlet,那么将会进行模糊匹配，对于每一个经过模糊匹配后符合条件的 URL 地址，都会被放入到匹配结果列表中，然后对列表中所有的地址按照匹配度进行排序操作，最后将匹配度最高的 Servlet 进行动态加载与初始化操作；如果精确匹配与模糊匹配都失败了，则会返回一个默认的 Servlet，即 DefaultServlet，该 Servlet 主要用于静态资源的处理操作。

```
1.  public Servlet getServletByUrl(String url) throws ServletNotFoundException {
2.      log.info("要匹配的 URL:{}", url);
3.      //根据 url 地址来进行精确匹配对应的 servlet
4.      String servletName = servletMapping.get(url);
5.      if (servletName == null) {
6.          //根据 url 地址进行模糊匹配
7.          List<String> matchedPath = new ArrayList<>();//所有匹配的路径
8.          Set<String> needToMatchSet = servletMapping.keySet();//所有需要进行匹配的路径
9.          for (String needToMatchPath : needToMatchSet) {
10.             if (matcher.match(needToMatchPath, url)) {
11.                 matchedPath.add(needToMatchPath);
12.             }
13.         }
14.         if (!matchedPath.isEmpty()) {
15.             //如果非空，也就是说有路径匹配成功
16.             //对所有的路径进行排序处理，取出匹配度最高的
17.             Comparator<String> pathComparator = matcher.getPatternComparator(url);
18.             matchedPath.sort(pathComparator);
19.             servletName = servletMapping.get(matchedPath.get(0));//获取匹配度最高的
20.         } else {
21.             //说明没有一个符合要求的，使用默认的 servlet
22.             return defaultServlet;
23.         }
24.     }
25.     ServletHolder servletHolder = servlets.get(servletName);
26.     if (servletHolder == null) {
27.         throw new ServletNotFoundException();
28.     }
```

```
29.     //根据匹配的 servletName 加载 servlet
30.     Servlet servlet = servletHolder.getServlet();
31.     if (servlet == null) {
32.         //如果 servlet 没有实例化，则进行加载
33.         servlet = initServlet(servletHolder.getServletClass());
34.         servletHolder.setServlet(servlet);
35.     }
36.     return servlet;
37. }
38.
39. private Servlet initServlet(String ServletClassName) throws ServletNotFoundE
    xception {
40.     try {
41.         //使用反射进行动态加载
42.          Servlet servlet = (Servlet) Class.forName(ServletClassName).getDecla
    redConstructor().newInstance();
43.         //调用前初始化
44.         servlet.init();
45.         return servlet;
46.     } catch (ClassNotFoundException e) {
47.         e.printStackTrace();
48.         log.error("加载并初始化 servlet 失败");
49.         throw new ServletNotFoundException();
50.      } catch (IllegalAccessException | InstantiationException | NoSuchMethodE
    xception | InvocationTargetException e) {
51.         e.printStackTrace();
52.     }
53.     return null;
54. }
```

4. 运行效果

图 10 客户端运行

图 11 客户端运行

图 12 服务器运行

5. 总结

在这次课程设计中，实现了一个基本满足 HTTP 协议的 Web 服务器，包含简单的动态页面加载，Session 会话机制，以及简单的 Servlet 容器等。

这个课程设计总共大概写了有 1 个礼拜左右，当然由于要上课，所以也不是天天写的，但也确实花了不少时间。通过本次课程设计，运用到了计算机网络课程里面学习到的应用层的相关知识，并且进一步了解和熟悉了 HTTP 协议的工作方式，同时也积累了基于 Java 平台的多线程及网络应用程序开发的经验，了解了 Servlet 的实现机制。

附录

本次课程设计核心源代码

Server 类：

```
1.  package com.yaser.core.network.server;
2.
3.  import com.yaser.core.http.context.ServletContext;
4.  import com.yaser.core.http.context.WebApplicationContext;
5.  import lombok.Getter;
6.
7.  import java.io.IOException;
8.  import java.net.ServerSocket;
9.
10. //服务器抽象类
11. public abstract class Server {
12.     @Getter
13.     protected ServerSocket serverSocket;
14.     @Getter
15.     protected boolean isRunning = true;
16.     @Getter
17.     protected ServletContext servletContext;
18.
19.     public Server(int port) {
20.         try {
21.             //创建一个服务器进行接受请求
22.             serverSocket = new ServerSocket(port);
23.             servletContext = WebApplicationContext.getServletContext();
24.         } catch (IOException e) {
25.             e.printStackTrace();
26.             System.out.println("Server start failed!");
27.         }
28.     }
29.
30.     public abstract void start();
31.
32.     public void close() {
33.         try {
34.             this.isRunning = false;
35.             this.serverSocket.close();
36.             this.servletContext.destroy();
37.         } catch (IOException e) {
38.             e.printStackTrace();
39.         }
```

```
    }
}
```

BioServer 类：

```
package com.yaser.core.network.server;

import com.yaser.core.network.handler.BioServerHandler;
import com.yaser.core.network.handler.Handler;
import lombok.extern.slf4j.Slf4j;

import java.util.Scanner;

@Slf4j
//服务器
public class BioServer extends Server {
    public BioServer(int port) {
        super(port);
    }

    @Override
    public void start() {
        //创建服务器处理程序对象
        Handler handler = new BioServerHandler(this);
        handler.start();
        Scanner scanner = new Scanner(System.in);
        String cmd;
        while (scanner.hasNext()) {
            cmd = scanner.nextLine();
            if (cmd.equals("EXIT")) {
                handler.close();
                this.close();
                break;
            }
        }
    }
}
```

BioServerHandler 类：

```
package com.yaser.core.network.handler;

import com.yaser.core.exception.exceptions.ServletException;
import com.yaser.core.http.request.HttpServletRequest;
import com.yaser.core.http.response.HttpServletResponse;
import com.yaser.core.http.servlet.ServletContainer;
import com.yaser.core.network.server.Server;
```

```
import lombok.extern.slf4j.Slf4j;

import java.io.IOException;

//服务器处理程序
@Slf4j
public class BioServerHandler extends Handler {
    public BioServerHandler(Server server) {
        super(server);
    }

    public void start() {
        while (this.server.isRunning()) {
            HttpServletResponse response = null;
            try {
                this.client = this.server.getServerSocket().accept();
                //先初始化 response 再初始化 request，因为在解析 request 的时候可能出现异常，避免出现异常后 response 却未初始化
                response = new HttpServletResponse(client.getOutputStream());

                //读取请求数据流，并解析
                 HttpServletRequest request = new HttpServletRequest(client.getInputStream());
                //从线程池中分配线程进行执行
                 pool.execute(new ServletContainer(servletContext, response, request, this));
            } catch (IOException e) {
                e.printStackTrace();
                log.error("IO Exception");
            } catch (ServletException e) {
                this.exceptionHandler.handle(e, response, this);
            }
        }

    }
}
```

Handler 类：

```
package com.yaser.core.network.handler;


import com.yaser.core.exception.ExceptionHandler;
import com.yaser.core.http.context.ServletContext;
import com.yaser.core.network.server.Server;
import lombok.Getter;
```

```java
import lombok.extern.slf4j.Slf4j;

import java.io.IOException;
import java.net.Socket;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

@Slf4j
public abstract class Handler {
    protected ExecutorService pool;
    protected Socket client;
    protected Server server;
    protected ServletContext servletContext;

    @Getter
    protected ExceptionHandler exceptionHandler;

    public Handler(Server server) {
        this.server = server;
        this.servletContext = server.getServletContext();
        exceptionHandler = new ExceptionHandler();
        //创建一个线程池
        pool = Executors.newFixedThreadPool(10);
    }

    public Socket getClient() {
        return client;
    }

    public abstract void start();

    public void close() {
        try {
            log.info("close socket");
            if (!client.isInputShutdown()) {
                this.client.shutdownInput();
            }
            if (!client.isOutputShutdown()) {
                this.client.shutdownOutput();
            }
            if (!client.isClosed()) {
                this.client.close();
            }
        } catch (IOException e) {
```

```
            e.printStackTrace();
            log.error("关闭 socket 时出现异常!");
        }
    }
}
```

ServletContext 类：

```
package com.yaser.core.http.context;

import com.yaser.core.constant.HTTPConstant;
import com.yaser.core.exception.exceptions.ServletNotFoundException;
import com.yaser.core.http.conversation.Cookie;
import com.yaser.core.http.conversation.HttpSession;
import com.yaser.core.http.conversation.HttpSessionCleaner;
import com.yaser.core.http.response.HttpServletResponse;
import com.yaser.core.http.servlet.Servlet;
import com.yaser.core.http.servlet.ServletHolder;
import lombok.extern.slf4j.Slf4j;
import org.dom4j.Document;
import org.dom4j.DocumentException;
import org.dom4j.Element;
import org.dom4j.io.SAXReader;
import org.springframework.util.AntPathMatcher;

import java.lang.reflect.InvocationTargetException;
import java.time.Instant;
import java.util.*;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

import static com.yaser.core.constant.CharContest.SESSION;


/**
 * Servlet 上下文环境
 * 主要工作：
 * 1.服务器启动的时候读取 servlet 配置文件
 * 2.将所有的 servlet 信息数据静态保存
 */
@Slf4j
public class ServletContext {
    /**
     * url -> servlet 别名
```

```
39.      * 一个 url 对应一个 servlet
40.      * 一个 servlet 可以有多个 url 对应
41.      */
42.     private HashMap<String, String> servletMapping;
43.
44.     private HashMap<String, ServletHolder> servlets;
45.     //路径匹配器
46.     private AntPathMatcher matcher;
47.
48.     private Servlet defaultServlet;
49.
50.     private Map<String, HttpSession> sessionMap;
51.
52.     private ScheduledExecutorService scheduledService;
53.
54.     public ServletContext() {
55.         init();
56.     }
57.
58.     private void init() {
59.         //读取 web.xml 配置文件，并加载 servlet
60.         this.servletMapping = new HashMap<>();
61.         this.servlets = new HashMap<>();
62.         this.matcher = new AntPathMatcher();
63.         this.sessionMap = new ConcurrentHashMap<>();
64.         this.parseConfig();
65.         //加载默认 servlet
66.         try {
67.              this.defaultServlet = this.initServlet(this.servlets.get(HTTPCon
    stant.DEFAULT_SERVLET).getServletClass());
68.         } catch (ServletNotFoundException e) {
69.             e.printStackTrace();
70.         }
71.         this.scheduledService = Executors.newSingleThreadScheduledExecutor();

72.          this.scheduledService.scheduleAtFixedRate(new HttpSessionCleaner(),
    0, 15, TimeUnit.SECONDS);
73.     }
74.
75.     public HttpSession getSession(String sessionId) {
76.         return sessionMap.get(sessionId);
77.     }
78.
```

```
79.     public Servlet getServletByUrl(String url) throws ServletNotFoundExcepti
    on {
80.         log.info("要匹配的 URL:{}", url);
81.         //根据 url 地址来进行精确匹配对应的 servlet
82.         String servletName = servletMapping.get(url);
83.         if (servletName == null) {
84.             //根据 url 地址进行模糊匹配
85.             List<String> matchedPath = new ArrayList<>();//所有匹配的路径
86.              Set<String> needToMatchSet = servletMapping.keySet();//所有需要进
    行匹配的路径
87.             for (String needToMatchPath : needToMatchSet) {
88.                 if (matcher.match(needToMatchPath, url)) {
89.                     matchedPath.add(needToMatchPath);
90.                 }
91.             }
92.             if (!matchedPath.isEmpty()) {
93.                 //如果非空，也就是说有路径匹配成功
94.                 //对所有的路径进行排序处理，取出匹配度最高的
95.                  Comparator<String> pathComparator = matcher.getPatternCompar
    ator(url);
96.                 matchedPath.sort(pathComparator);
97.                  servletName = servletMapping.get(matchedPath.get(0));//获取匹
    配度最高的
98.             } else {
99.                 //说明没有一个符合要求的，使用默认的 servlet
100.                  return defaultServlet;
101.             }
102.         }
103.         ServletHolder servletHolder = servlets.get(servletName);
104.         if (servletHolder == null) {
105.             throw new ServletNotFoundException();
106.         }
107.         //根据匹配的 servletName 加载 servlet
108.         Servlet servlet = servletHolder.getServlet();
109.         if (servlet == null) {
110.             //如果 servlet 没有实例化，则进行加载
111.             servlet = initServlet(servletHolder.getServletClass());
112.             servletHolder.setServlet(servlet);
113.         }
114.         return servlet;
115.     }
116.
117.     private Servlet initServlet(String ServletClassName) throws ServletNotF
    oundException {
```

```
118.         try {
119.             //使用反射进行动态加载
120.              Servlet servlet = (Servlet) Class.forName(ServletClassName).get
    DeclaredConstructor().newInstance();
121.             //调用前初始化
122.             servlet.init();
123.             return servlet;
124.         } catch (ClassNotFoundException e) {
125.             e.printStackTrace();
126.             log.error("加载并初始化 servlet 失败");
127.             throw new ServletNotFoundException();
128.          } catch (IllegalAccessException | InstantiationException | NoSuchMe
    thodException | InvocationTargetException e) {
129.             e.printStackTrace();
130.         }
131.         return null;
132.     }
133.
134.     /**
135.      * 解析 web.xml 的配置文件
136.      */
137.     private void parseConfig() {
138.         SAXReader reader = new SAXReader();
139.         try {
140.             //加载配置文件
141.              Document document = reader.read(ServletContext.class.getResourc
    eAsStream("/web.xml"));
142.             Element root = document.getRootElement();
143.             //保存 servlet 名字与 servlet 类名的映射
144.             Iterator<Element> servletList = root.elementIterator("servlet");

145.             while (servletList.hasNext()) {
146.                 //遍历所有的 Servlet
147.                 Element servlet = servletList.next();
148.                 String key = servlet.elementText("servlet-name");
149.                 String value = servlet.elementText("servlet-class");
150.                     servlets.put(key.trim(), new ServletHolder(value));//将
    servlet 名字与 servlet 类名保存为键值对
151.             }
152.             //遍历 servlet 名字与 url 地址的映射
153.              Iterator<Element> servletMappingList = root.elementIterator("se
    rvlet-mapping");
154.             while (servletMappingList.hasNext()) {
155.                 Element servletMap = servletMappingList.next();
```

```
156.                 String key = servletMap.elementText("url-pattern");
157.                 String value = servletMap.elementText("servlet-name");
158.                 servletMapping.put(key, value);//将 servlet 名字与 url 地址保存为键值对
159.             }
160.         } catch (DocumentException e) {
161.             e.printStackTrace();
162.             log.error("加载 servlet 配置项失败");
163.         }
164.     }
165.
166.     //销毁所有的 Servlet
167.     public void destroyServlet() {
168.         servlets.values().forEach(servletHolder -> {
169.             Servlet servlet = servletHolder.getServlet();
170.             if (servlet != null) {
171.                 servletHolder.getServlet().destroy();
172.             }
173.         });
174.     }
175.
176.     //创建 session
177.     public HttpSession createSession(HttpServletResponse response) {
178.         String sessionId = UUID.randomUUID().toString();
179.         HttpSession session = new HttpSession(sessionId);
180.         this.sessionMap.put(sessionId, session);
181.         response.addCookie(new Cookie(SESSION, sessionId));
182.         return session;
183.     }
184.
185.     public void removeSession(String sessionId) {
186.         this.sessionMap.remove(sessionId);
187.     }
188.
189.     public void detectSessionIsValid() {
190.         //检测 Session 是否过期
191.         if (!sessionMap.isEmpty()) {
192.             Set<String> sessionKeySet = sessionMap.keySet();
193.             for (String key : sessionKeySet) {
194.                 HttpSession session = sessionMap.get(key);
195.                 //计算当前时间减去 session 的最大过期时间，该结果表示，如果该 session 是不过期的，那么它的活动时间应该在 lastAccessedTime 与 now 之间。
196.                 Instant lastAccessedTime = Instant.now().minusSeconds(session.getMaxInactiveInterval());
```

```
                log.info("检测 session，sessionID:{}",session.getId());
                 if (lastAccessedTime.isAfter(session.getLastAccessed())) {

                    //如果 session 最后的活动时间在这之前，表示已经过期了
                    session.invalidate();
                }
            }
        } else {
            log.info("当前没有 session");
        }
    }
    public void destroy(){
        this.destroyServlet();
        this.scheduledService.shutdownNow();
    }
}
```

HttpSession 类：

```
package com.yaser.core.http.conversation;

import com.yaser.core.http.context.WebApplicationContext;
import lombok.Getter;
import lombok.Setter;
import lombok.extern.slf4j.Slf4j;

import java.time.Instant;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

import static com.yaser.core.constant.HTTPConstant.DEFAULT_MAX_INACTIVE_INTERVAL;

@Slf4j
public class HttpSession {
    @Getter
    private String id;//sessionId

    private Map<String, Object> attribute;//保存在 session 中的属性值
    @Getter
    private Instant lastAccessed;
    @Getter
    @Setter
    private int maxInactiveInterval;//最大过期时间，单位是秒
    //当前 session 是否合法，因为在 session 过期到被删除的期间，session 还有可能被使用，所以为了防止被使用，使用该字段做标识符
```

```
26.     private boolean isValid;
27. 
28.     public HttpSession(String id) {
29.         this.id = id;
30.         this.attribute = new ConcurrentHashMap<>();
31.         this.lastAccessed = Instant.now();
32.         this.isValid = true;
33.         this.maxInactiveInterval = DEFAULT_MAX_INACTIVE_INTERVAL;
34.     }
35. 
36.     public void setAttribute(String name, Object val) {
37.         if (this.isValid) {
38.             this.attribute.put(name, val);
39.             this.lastAccessed = Instant.now();
40.         } else {
41.             throw new IllegalStateException("session has invalidated");
42.         }
43.     }
44. 
45.     public Object getAttribute(String name) {
46.         if (this.isValid) {
47.             this.lastAccessed = Instant.now();
48.             return this.attribute.get(name);
49.         }
50.         throw new IllegalStateException("session has invalidated");
51.     }
52. 
53.     public void invalidate() {
54.         this.isValid = false;
55.         this.attribute.clear();
56.         log.info("当前 session 已过期，sessionID:{}", id);
57.         WebApplicationContext.getServletContext().removeSession(this.id);
58.     }
59. 
60.     public void removeAttribute(String name) {
61.         if (this.isValid) {
62.             this.attribute.remove(name);
63.             this.lastAccessed = Instant.now();
64.         } else {
65.             throw new IllegalStateException("session has invalidated");
66.         }
67.     }
68. }
```

HttpServletRequest 类：

```java
package com.yaser.core.http.request;

import com.yaser.core.constant.CharContest;
import com.yaser.core.constant.HTTPConstant;
import com.yaser.core.enumeration.RequestMethod;
import com.yaser.core.exception.exceptions.RequestInvalidException;
import com.yaser.core.exception.exceptions.RequestParseException;
import com.yaser.core.http.context.WebApplicationContext;
import com.yaser.core.http.conversation.Cookie;
import com.yaser.core.http.conversation.HttpSession;
import com.yaser.core.http.servlet.ServletContainer;
import lombok.Getter;
import lombok.Setter;
import lombok.extern.slf4j.Slf4j;

import java.io.BufferedInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.URLDecoder;
import java.nio.charset.StandardCharsets;
import java.util.Arrays;
import java.util.HashMap;

import static com.yaser.core.constant.CharContest.*;

@Slf4j
public class HttpServletRequest {
    @Setter
    private ServletContainer container;
    @Getter
    private RequestMethod method = null;//请求方式

    private String url = "";//请求路径信息

    private String forwardUrl;

    private boolean forward;
    private HashMap<String, String> params = new HashMap<>();//请求参数信息

     private HashMap<String, String> headers = new HashMap<>();//请求头部信息

    private HashMap<String, Cookie> cookieMap = new HashMap<>();//cookie
    @Getter
```

```
44.     private String body = "";//body 体信息
45.
46.     private HashMap<String, Object> attributes = new HashMap<>();//用户属性
47.
48.     private HttpSession session;//用户会话
49.
50.     public HttpSession getSession() {
51.         return getSession(true);
52.     }
53.
54.     public HttpSession getSession(boolean createIfNone) {
55.         if (this.session != null) {
56.             //如果 Session 已经有了，则直接返回
57.             return this.session;
58.         }
59.         if (cookieMap.containsKey(SESSION)) {
60.             //判断 Cookie 中是否存在 Session ID
61.             Cookie cookie = cookieMap.get(SESSION);
62.             String sessionId = cookie.getValue();//获取 sessionId
63.              HttpSession currentSession = WebApplicationContext.getServletCon
    text().
64.                     getSession(sessionId);//使用 ID 获取 Session 实例
65.             if (currentSession != null) {
66.                 this.session = currentSession;
67.                 return this.session;
68.             }
69.         }
70.         if (createIfNone) {
71.             //如果不存在，则创建
72.              this.session = WebApplicationContext.getServletContext().createS
    ession(this.container.getResponse());
73.         }
74.         return this.session;
75.     }
76.
77.     public HttpServletRequest() {
78.         method = RequestMethod.GET;
79.     }
80.
81.     public HttpServletRequest(InputStream in) throws RequestInvalidException,
     RequestParseException {
82.         this.parseRequest(readRequest(in));
83.     }
84.
```

```
85.     //读取全部的输入流数据，并返回
86.     public byte[] readRequest(InputStream in) throws RequestInvalidException
    {
87.         BufferedInputStream bin = new BufferedInputStream(in);
88.         byte[] buf = null;
89.         try {
90.             buf = new byte[bin.available()];
91.             //将数据读取到缓冲区中
92.             int len = bin.read(buf);
93.             if (len <= 0) {
94.                 throw new RequestInvalidException();
95.             }
96.         } catch (IOException e) {
97.             e.printStackTrace();
98.         }
99.         return buf;
100.     }
101.
102.     public String getHeader(String key) {
103.         return headers.get(key.toLowerCase());
104.     }
105.
106.     public String getServletPath() {
107.         //如果进行转发，则返回转发的地址，否则返回浏览器请求的地址
108.         if (this.forward) {
109.             return this.forwardUrl;
110.         }
111.         return this.url;
112.     }
113.
114.     public Object getAttribute(String key) {
115.         return this.attributes.get(key.toLowerCase());
116.     }
117.
118.     public void setAttribute(String key, Object val) {
119.         this.attributes.put(key, val);
120.     }
121.
122.     //解析 body 体数据
123.     private void parseBody(String body) throws RequestParseException {
124.         //将 body 数据读取出来
125.         int len = Math.min(body.length(), Integer.parseInt(this.getHeader("
    content-length")));
126.         this.body = body.substring(0, len);
```

```
127.         String contentType = getHeader("content-type");
128.          if (contentType.equals(HTTPConstant.TYPE_FORM) || contentType.equal
    s(HTTPConstant.TYPE_TEXT)) {
129.             //表单格式，则直接调用表单解析
130.             this.params.putAll(this.parseParams(this.body));
131.         }
132.         log.info("body 数据：{}", this.body);
133.     }
134.
135.     //解析头部信息
136.     private void parseHeader(String[] lines) throws RequestParseException {

137.         int lineNum = 0;
138.         //先解析第一行数据
139.         this.analysis(lines[lineNum++]);
140.         //解析 head 的其它行
141.         for (; lineNum < lines.length; lineNum++) {
142.             if (lines[lineNum].length() == 0) {
143.                 //如果是空行直接退出
144.                 break;
145.             }
146.             //提取请求报文的每一行属性值
147.             int colonIndex = lines[lineNum].indexOf(':');
148.             if (colonIndex == -1) {
149.                 throw new RequestParseException();
150.             }
151.              String key = lines[lineNum].substring(0, colonIndex).toLowerCas
    e().trim();
152.              String value = lines[lineNum].substring(colonIndex + 1).trim();

153.             this.headers.put(key, value);
154.         }
155.         log.info("headers:{}", this.headers);
156.         if (this.headers.containsKey("cookie")) {
157.             //如果存在 cookie，则对 cookie 进行解析操作
158.             String cookies = this.headers.get("cookie");
159.             //使用分号进行分割
160.             String[] cookieList = cookies.split(SEMICOLON);
161.             for (String rawCookie : cookieList) {
162.                 //遍历键值对
163.                  String[] cookieNameVal = rawCookie.trim().split(EQUATION);

164.                 //按照等号分割
165.                 if (cookieNameVal.length < 2) {
```

```
166.                     //不合法，则抛弃该 cookie
167.                     continue;
168.                 }
169.                 //将 cookie 进行实例化
170.                  Cookie cookie = new Cookie(cookieNameVal[0], cookieNameVal[
    1]);
171.                 cookieMap.put(cookieNameVal[0], cookie);
172.             }
173.         }
174.     }
175.
176.     private void parseRequest(byte[] data) throws RequestInvalidException,
    RequestParseException {
177.         String mergedData = new String(data, StandardCharsets.UTF_8);//使用
    UTF-8 进行编码
178.         //将数据进行解析
179.         String[] lines = URLDecoder.decode(mergedData, StandardCharsets.UTF
    _8).split(CharContest.CRLF);
180.         if (lines.length <= 1) {
181.             throw new RequestInvalidException();
182.         }
183.         log.info("Request 数据读取完毕");
184.         log.info("行数：{}", lines.length);
185.         log.info("数据：{}", Arrays.toString(lines));
186.         log.info("============================");
187.         this.parseHeader(lines);
188.         String len = this.getHeader("content-length");
189.         if (len != null && Integer.parseInt(len.trim()) != 0) {
190.             log.info("要解析 body");
191.             parseBody(lines[lines.length - 1]);
192.         } else {
193.             log.info("不要解析 body");
194.         }
195.     }
196.
197.
198.     //解析提交方式，路径以及协议
199.     private void analysis(String line) throws RequestParseException {
200.         //将首行按照空格进行分割
201.         String[] firstLine = line.split(CharContest.BLANK);
202.         //判断请求方式
203.         this.method = RequestMethod.valueOf(firstLine[0].toUpperCase());
204.         log.info("method:{}", this.method);
205.         //解析路径
```

```
206.         String pathAndParams = firstLine[1];
207.         int start = pathAndParams.indexOf("?");
208.         if (start != -1) {
209.             //包含参数信息
210.             this.url = pathAndParams.substring(0, start);
211.             log.info("path:{}", this.url);
212.             String params = pathAndParams.substring(start + 1);
213.             //解析 url 参数
214.             this.params = this.parseParams(params);
215.             log.info("params:{}", this.params);
216.         } else {
217.             this.url = pathAndParams;
218.         }
219.     }
220.
221.     //解析参数信息
222.     private HashMap<String, String> parseParams(String params) throws Reque
    stParseException {
223.         if (params == null) {
224.             return new HashMap<>();
225.         }
226.         String[] paramList = params.split("&");//将参数进行拆分
227.         HashMap<String, String> paramsMap = new HashMap<>();
228.         for (String param : paramList) {
229.             String[] oneParam = param.split("=");
230.             if (oneParam.length != 2) {
231.                 throw new RequestParseException();
232.             }
233.             paramsMap.put(oneParam[0].trim().toLowerCase(), oneParam[1]);
234.         }
235.         return paramsMap;
236.     }
237.
238.     public RequestDispatcher getRequestDispatcher(String url) {
239.         return new HttpRequestDispatcher(url);
240.     }
241.
242.     public String getParameter(String key) {
243.         return params.get(key.toLowerCase());
244.     }
245.
246.     public Cookie getCookie(String name) {
247.         return cookieMap.get(name);
248.     }
```

```
249.
250.     public void setForward(String forwardUrl) {
251.         this.forwardUrl = forwardUrl;
252.         this.forward = true;
253.     }
254. }
```

HttpServletResponse 类：

```
1.  package com.yaser.core.http.response;
2.
3.  import com.yaser.core.constant.HTTPConstant;
4.  import com.yaser.core.enumeration.HttpStatus;
5.  import com.yaser.core.http.conversation.Cookie;
6.  import com.yaser.core.http.servlet.ServletContainer;
7.  import lombok.Setter;
8.  import lombok.extern.slf4j.Slf4j;
9.
10. import java.io.BufferedOutputStream;
11. import java.io.IOException;
12. import java.io.OutputStream;
13. import java.nio.charset.StandardCharsets;
14. import java.util.ArrayList;
15. import java.util.Date;
16.
17. import static com.yaser.core.constant.CharContest.*;
18.
19. @Slf4j
20. public class HttpServletResponse {
21.     @Setter
22.     private ServletContainer container;
23.     private BufferedOutputStream writer;
24.     private StringBuilder response;
25.     private ArrayList<Header> headers;//用户添加的头部信息
26.
27.     public void setContentType(String contentType) {
28.         this.contentType = contentType;
29.     }
30.
31.     private String contentType;
32.     private String charset;
33.     private HttpStatus httpStatus;
34.     private byte[] body;
35.     private boolean hasBody = false;
36.     private ArrayList<Cookie> cookies;
37.
```

```
38.     public HttpServletResponse(OutputStream out) {
39.         writer = new BufferedOutputStream(out);
40.         response = new StringBuilder();
41.         headers = new ArrayList<>();
42.         httpStatus = HttpStatus.OK;
43.         contentType = HTTPConstant.DEFAULT_CONTENT_TYPE;
44.         cookies = new ArrayList<>();
45.         charset = null;
46.     }
47.
48.     //构造 response 的头部信息
49.     private void buildHeader() {
50.         //添加默认头部信息
51.         response.append("HTTP/1.1").append(BLANK).
52.                 append(httpStatus.getCode()).append(BLANK).append(httpStatus)
    .append(CRLF);
53.          response.append("Date:").append(BLANK).append(new Date()).append(CRL
    F);
54.         if (charset != null) {
55.             String type = contentType;
56.             int charSetIndex = contentType.indexOf("charset");
57.             if (charSetIndex != -1) {
58.                 //存在 charset，将原来的 charset 替换掉
59.                 type = contentType.substring(0, charSetIndex);
60.             }
61.             if (type.contains(SEMICOLON)) {
62.                 contentType = type + "charset=" + charset;
63.             } else {
64.                 contentType = type + SEMICOLON + "charset=" + charset;
65.             }
66.         }
67.          response.append("Content-Type:").append(BLANK).append(contentType).a
    ppend(CRLF);
68.         if (hasBody) {
69.             //如果有 body 的话，添加 body 体长度信息
70.              response.append("Content-Length:").append(BLANK).append(body.len
    gth).append(CRLF);
71.         }
72.         if (!cookies.isEmpty()) {
73.             //如果有 cookie 的话
74.             StringBuilder cookieStr = new StringBuilder();
75.             for (Cookie cookie : cookies) {
76.                 cookieStr.append(cookie.toString()).append(BLANK);
77.             }
```

```
78.                 response.append("Set-Cookie:").append(BLANK).append(cookieStr).a
    ppend(CRLF);
79.         }
80.         for (Header header : headers) {//添加用户头部信息
81.                 response.append(header.getKey()).append(":").append(BLANK).appen
    d(header.getVal()).append(CRLF);
82.         }
83.         response.append(CRLF);//添加空行
84.         log.info("header:{}", response);
85.     }
86.
87.
88.     public void setHttpStatus(HttpStatus httpStatus) {
89.         this.httpStatus = httpStatus;
90.     }
91.
92.     public void addHeader(Header header) {
93.         this.headers.add(header);
94.     }
95.
96.
97.     public void write(String data) {
98.         //将用户传入的数据进行写回操作
99.         body = data.getBytes(StandardCharsets.UTF_8);
100.         hasBody = true;
101.         this.write();
102.     }
103.
104.     public void write(byte[] data) {
105.         body = data;
106.         hasBody = true;
107.         this.write();
108.     }
109.
110.     public void addCookie(Cookie cookie) {
111.         this.cookies.add(cookie);
112.     }
113.
114.     public void write() {
115.         this.sendToClient();
116.     }
117.
118.     //重定向
119.     public void sendRedirect(String url) {
```

```
120.         log.info("重定向至：{}", url);
121.         this.addHeader(new Header("Location", url));
122.         this.setHttpStatus(HttpStatus.MOVED_TEMPORARILY);
123.         this.sendToClient();
124.     }
125.
126.     private void sendToClient() {
127.         //发送数据给客户端
128.         try {
129.             this.buildHeader();
130.              byte[] header = response.toString().getBytes(StandardCharsets.U
    TF_8);
131.             this.writer.write(header);
132.             if (hasBody) {
133.                 this.writer.write(body);
134.             }
135.             this.writer.flush();
136.         } catch (IOException e) {
137.             e.printStackTrace();
138.             log.error("写回客户端失败！");
139.         }
140.     }
141.
142.     public void setCharacterEncoding(String charset) {
143.         this.charset = charset;
144.     }
145. }
```

ResourceHandler 类：

```
1.  package com.yaser.core.resource;
2.
3.  import com.yaser.core.exception.exceptions.ResourceNotFoundException;
4.  import lombok.Getter;
5.  import lombok.extern.slf4j.Slf4j;
6.
7.  import java.io.File;
8.  import java.io.FileInputStream;
9.  import java.io.IOException;
10. import java.net.URISyntaxException;
11. import java.net.URL;
12.
13. //静态资源处理
14. @Slf4j
15. public class ResourceHandler {
16.     @Getter
```

```
17.     private String resourceName;
18.     public byte[] getResourceByUrl(String url) throws ResourceNotFoundExcept
    ion {
19.         URL resourceUrl = ResourceHandler.class.getResource(url);
20.         if (resourceUrl == null) {
21.             log.error("{}资源未找到", url);
22.             throw new ResourceNotFoundException();
23.         }
24.         byte[] resource = null;
25.         try {
26.             File file = new File(resourceUrl.toURI());
27.             this.resourceName = file.getName();
28.             FileInputStream fis = new FileInputStream(file);
29.             resource = new byte[fis.available()];
30.             fis.read(resource);
31.             fis.close();
32.         } catch (URISyntaxException | IOException e) {
33.             e.printStackTrace();
34.         }
35.         return resource;
36.     }
37. }
```