景德镇陶瓷大学
科技艺术学院
College of Technology and Art
Jingdezhen Ceramic Institute

本科生毕业设计(论文)

题目：Web 服务器的设计与实现

学　　号：314040300112

姓　　名：朱剑鑫

系　　别：工程系

专　　业：14 计算机科学与技术

完成日期：2018 年 6 月 7 日

指导老师：何福保

分类号_____ 学校代码_____

UDC_____密级_____ 学　　号 314040300112

景德镇陶瓷大学科技艺术学院

本科生毕业设计（论文）

题目：Web 服务器的设计与实现

朱剑鑫

指导老师： 何福保　副教授　科技艺术学院　工程系

申请学位级别：工学学士 专业名称： 14 计算机科学与技术

论文提交日期： 2018 年 6 月 8 日

论文答辩日期 ：2018 年 6 月 8 日

学位授予单位 景德镇陶瓷大学科技艺术学院

答辩委员会主席：_____

论文评阅人：_____

2018 年 6 月 8 日

摘要

万维网发展至今已 25 年有余，其使用之广泛性，在网络世界中有着不可代替的地位。随着移动互联网爆发，用户与日俱增，Web 服务器也面临着诸如更高并发，更大规模计算等挑战。所以，对 Web 服务器的运行机制和架构设计进行深入研究是极其重要的。

服务器采用 Java 为开发语言,Socket 套接字以及多路复用 I/O 模型处理网络请求和数据传输，线程池处理网络并发请求，严格遵循 HTTP 协议规范，进行请求报文的解析和响应报文的构建。使用分层和管道式结构进行系统架构设计，多种设计模式提高代码的可重用性，降低耦合性。支持简单的动态脚本解析能力，来动态生成前端页面。支持动态运行 CGI 程序，且设计了一套基于注解的，零配置的 CGI 应用程序 API，能够更加方便的开发 web 应用程序。

关键词：网络编程　HTTP 协议　Web 服务器　架构设计

Abstract

The development of the WWW has been more than 25 years ago. Its widespread use has an irreplaceable position in the online world. With the outbreak of the mobile Internet and the increasing number of users, web servers also face challenges such as higher concurrency and larger-scale computing. Therefore, it is extremely important to thoroughly study the operating mechanism and architecture design of the web server.

The server adopts Java as the development language, Socket and multiplexed I/O model to process network requests and data transmission, the thread pool processes network concurrent requests, strictly follows the HTTP protocol specification, and performs request message parsing and response messages. The build. Using hierarchical and pipelined architectures for system architecture design, multiple design patterns increase code reusability and reduce coupling. Supports simple dynamic script parsing capabilities to dynamically generate front-end pages. Supports dynamic running of CGI programs, and an annotation-based, zero-configuration CGI application API is designed to make it easier to develop web applications.

Keywords: Network Programming HTTP Protocol WebServer Architecture Design

目录

第一章 绪论

1.1 研究背景与意义

从 1980 年 Tim Berners-Lee 构建 ENQUIRE 项目至今,万维网已经走过了 38 年。在这个全民互联的时代，每天基于 HTTP 协议所传输的数据不计其数。而这些海量数据传输，交互背后，都是由 Web 服务器作为支撑。在移动互联网浪潮下的今天，Web 服务器也面临着更加严峻的挑战。几万甚至几十万每秒的超高并发，以及海量数据的超大计算等对服务器性能的要求也越来越高。Web 服务器严格遵循 HTTP 协议规范进行通信工作，HTTP 又是基于 TCP 协议之上的应用层协议。所以，对网络协议的了解以及对 Web 服务器整个工作流程以及运行机制的研究是非常有必要的。

1.2 课题研究过程

本文采用自底向上的方式，先从网络底层协议开始进行介绍，对 IP 等内核级协议做简单介绍，对 TCP 和 HTTP 协议进行详细说明。然而一款高性能，高可用服务器的瓶颈不仅仅在网络通信上，对操作系统 I/O 模型的选择，对线程模型的处理同样是重中之重。所以需要对 Unix 五种 I/O 模型进行分析对比，了解 I/O 通信过程中同步与异步，阻塞与非阻塞的概念以及优劣；再分析商业 Web 服务器 Tomcat 的 I/O 模型和线程模型。然后对 Java 编程语言以及 JVM 平台的特点进行简单说明；最后剖析 Dam 的整个运行机制。本文旨在剖析一款 Web 服务器从网络协议，到操作系统 I/O 通信，到编码实现阶段所遇到以及要解决的种种问题。

1.3 论文组织结构

本文共分六个章节。

- 第一章：绪论
- 第二章：需求分析，分析服务器基础功能需求与流程架构。
- 第三章：介绍网络协议基础知识与操作系统 I/O 模型以及多线程编程。
- 第四章：介绍服务器架构和编码设计实现
- 第五章：针对服务器所应该具备的功能，进行每个功能点的测试。
- 第六章：总结。

第二章 可行性分析

2.1 技术可行性

Web 服务器作为中间件系统，向下需要与操作系统交互，处理网络和文件 I/O，向上需要为 CGI 程序设计并提供 API，并且具备执行 CGI 程序的能力。在软件设计开发中，也要熟悉常用的架构设计以及设计模式，来保证系统的低耦合性，高可用性和易扩展性。业内也有例如 Nginx，Apache，Tomcat，Jetty 等成熟的开源 Web 服务器，所以在技术上是可行的。

2.1.1 基础技术

服务器将采用 Java 为开发语言，使用 Socket 套接字来监听处理 HTTP 请求和进行网络数据的传输，使用多线程技术并发处理 HTTP 请求，调用操作系统 I/O 流，用于 HTML 页面，CSS 样式表，JS 脚本，图片，音频，视频等浏览器可处理的静态资源文件的读写和日志文件的记录，以及利用缓存技术来缓存 Session 数据，进行状态记录，利用简单的编译原理知识，进行动态脚本的解析。

2.1.2 设计模式

服务器采用了几种设计模式，来提高代码的可重用性，降低系统模块之间的耦合度。例如，服务器采用了观察者模式来处理 handler 链，利用抽象工厂模式来生成 GCI 程序返回的结果处理器，利用策略模式来根据的不同的 HTTP 状态码生成 HTTO 响应头。

2.1.2 架构设计

服务器整体采用分层架构，每一层只解决整个生命周期中的部分问题，且每一层都对上一层隐藏实现细节，有效的降低了整个执行流程的复杂性，使整个架构具有良好的可扩展性，也易于维护。

服务器分为四层，即：Connector 层，Server 层，Handler 层，CGI 程序层。如图 2.1 所示。

2.2 经济可行性

如今大多数的为上层应用提供服务的底层应用软件或者中间件都是开源免费的，软件开源也为软件行业的蓬勃发展起着不可磨灭的作用。此服务器开发出来也是被用来开源学习使用的，所以，在经济上是可行的。

2.3 操作可行性

服务器的启动操作时很简单的，Windows 下点击 bin 目录下的”run.bat”批处理文件，Linux 下点击”run.sh”shell 脚本即可启动并运行服务器。但是要基于

服务器开发 GCI 程序还是需要基本的前端开发知识和 Java 基础的，如果具备以上知识，只需要根据服务器的规范，使用服务器定义好的声明式注解，即可基于服务器开发属于自己的 web 应用。

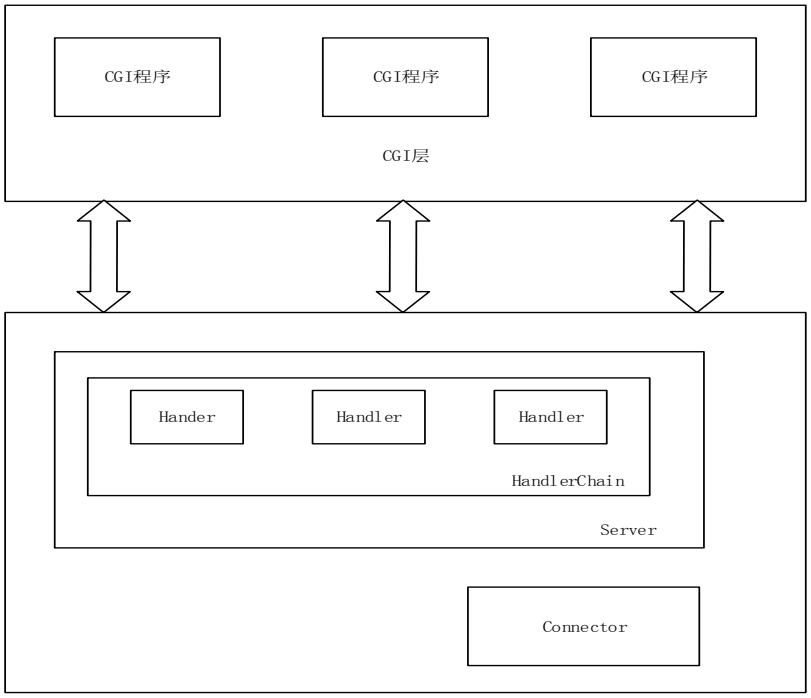

图 2.1 服务器架构设计

第三章 网络协议与 I/O 模型

3.1 TCP/IP 协议

TCP/IP 协议族是一种网络通信模型和全网传输协议族，它是互联网的基本通信体系结构。TCP/IP 协议族不只有 TCP 和 IP 这两个协议，而是从网络层到应用层所的整个协议栈的统称。要进行网络编程，对 TCP/IP 协议族的了解的必不可少的，而 TCP 协议更是整个协议栈中最重要,最复杂,与网络编程相关性最大的一个协议。

3.1.1 TCP 协议定义

TCP(Transmission Control Protocol)协议,位于网络层之上，应用层之下，属于传输层协议，其中 UDP 协议也是传输层中一个重要的协议。TCP 协议是一种面向连接的，可靠地，基于字节流的传输层协议。它在不可靠的网络上提供了一种可靠的网络通信模型。要进行网络应用程序设计，就需要了解 TCP 协议的每个参数，TCP 协议是怎样保证可靠连接，以及流量控制，拥塞控制和性能优化等问题。

3.1.2 TCP 报文格式

表 3.1 为 TCP 数据报结构：

表 3.1 TCP 报文格式

偏移	位 0-3	4-7	8-15	16-31
0	src port(源端口号)			dest port(目的端口号)
32	SYN 序列号			
64	ACK 确认号			
96	报头长度	保留位	标识符	滑动窗口大小
128	校验和			紧急指针
160	选项字段			
160/192+	数据域			

- 源端口号（16Bit）－发送端的端口号
- 目的端口号（16Bit）－接收端的端口号
- 序列号（seq，32Bit）
  - 如果有 SYN，第一个序列码为本序列号+1。
  - 如果没有 SYN，为第一个序列码。
- 确认号（ACK，32 Bit），接收到对方的 SYN+1。
- 报头长度（4 Bit），以 4Byte 为单位计算出的数据段开始地址的偏移值。
- 保留—须置 0

- 标志符
  - URG—表示该数据包是高优先级的，紧急指针字段有效。
  - ACK—确认号，为接收到的 SYN+1。
  - PSH—数据包带有带字段表示希望接收方应尽快将这个报文段交给应用层而不用等待缓冲区装满。
  - RST—重置 TCP 连接。出现差错，拒绝非法报文段或拒绝连接请求时出现。
  - SYN—连接请求或是连接接受请求，用于创建连接和使顺序号同步
  - FIN—释放连接时需要传输的字段。
- 窗口（WIN，16 位长），用于流量控制。接收方最大可接受的数据报大小，即滑动窗口大小。
- 校验和（Checksum，16 位长），强制性的字段。验证 TCP 报文段是否出错。
- 紧急指针（16 位长），本报文段中的紧急数据的最后一个字节的序号。
- 选项字段—最多 40 字节。每个选项开头是 1 字节的整型字段，说明选项的类型。
  - 0：选项表结束
  - 1：无操作用于选项字段之间的字边界对齐。
  - 2：最大报文段长度(MSS)，表示往另一端发送的最大报文长度。当一个连接建立时，连接的双方都要告知对方的 MSS。最终的 IP 数据报通常是 40 字节长：20 字节的 TCP 首部和 20 字节的 IP 首部。[1]
  - 3：滑动窗口扩展因子（wscale），取值为 0-14。用来改变滑动窗口的大小。
  - 4：sackOK：发送端支持并同意使用 SACK 选项。
  - 5：SACK 实际工作的选项。
  - 8：时间戳（10 字节，TCP Timestamps Option，TSopt）
    - 发送端的时间戳（Timestamp Value field，TSval，4 字节）
    - 时间戳回显应答（Timestamp Echo Reply field，TSecr，4 字节）

3.1.3 TCP 协议通信流程

TCP 是一个面向连接的协议。在进行数据传输之前，都必须先建立一条可靠的连接，所以了解 TCP 是如何建立连接以及通信结束后是如何终止连接是非常重要的。

TCP 的连接过程也称为三次握手：

1）请求端发送一个报文到被请求端，报文严格按照 TCP 报文格式定义，包括但不限于以下字段：源端口与目的端口，SYN 序列号，初始序列号(ISN)。

2）被请求端发回包含被请求端的初始序列号的 SYN 报文段，作为应答。同时，将 ACK 设置为客户端的 ISN 加 1 以对客户端的 SYN 报文段进行确认。一个 SYN 将占用一个序号。

3）请求端必须将 ACK 设置为被请求端的 ISN 加 1 以对被请求端的 SYN 报文段进行确认。

这三个过程即可完成连接的建立。

TCP 释放连接的过程也称为四次挥手：

1）主动关闭的一方执行主动关闭，发送一个 FIN 报文段到被动关闭方。

2）被动关闭的一方收到这个 FIN，将发回一个 ACK，ACK 为收到的 FIN+1。

3）被动关闭方发送一个 FIN 报文段到主动关闭一方。

4）主动关闭方收到这个 FIN，并发回一个 ACK，ACK 为收到的 FIN+1。

以上为 TCP 握手和挥手的执行流程。整个 TCP 生命周期的状态迁移如图 3.1 所示。

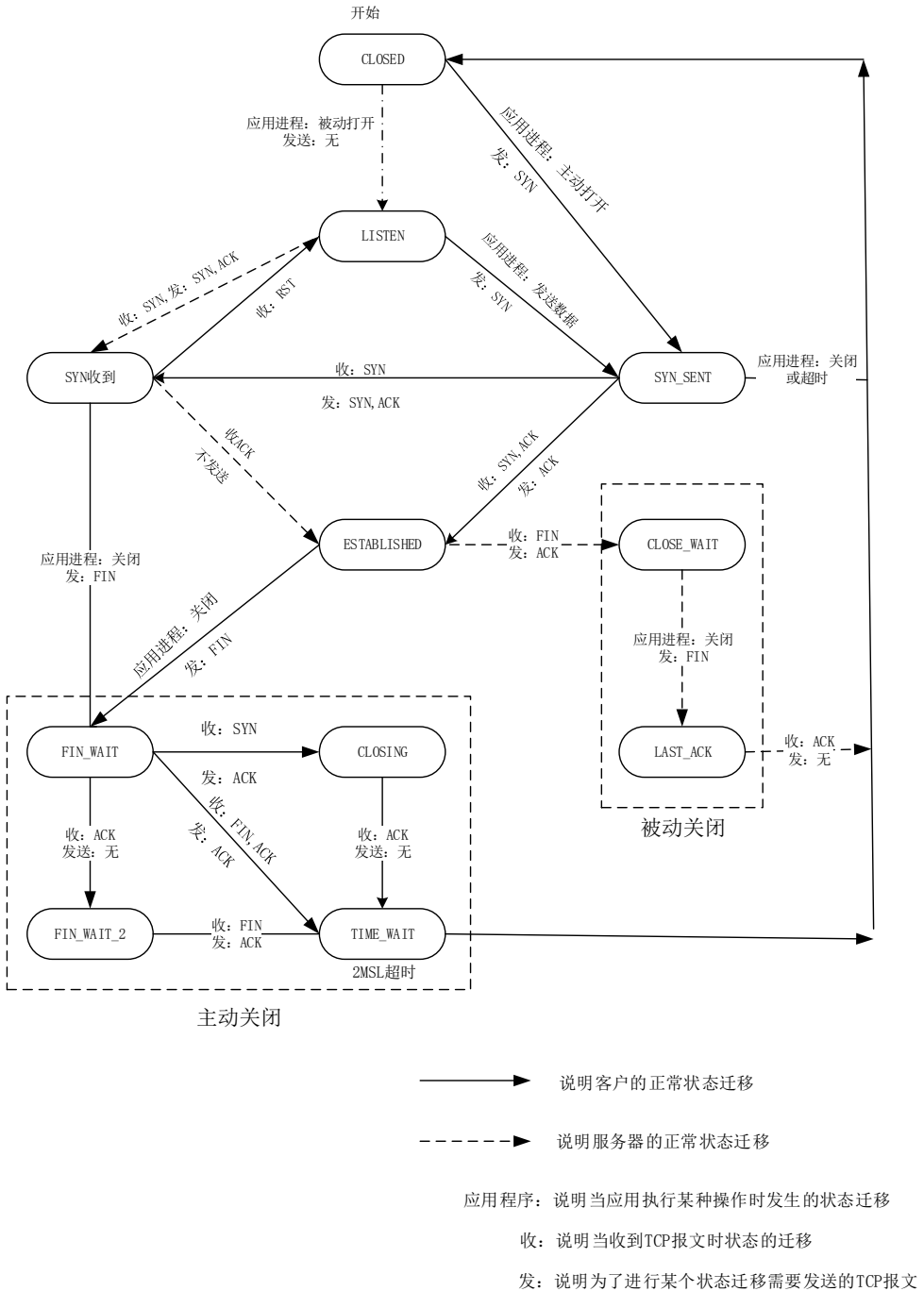

图 3.1 TCP 状态迁移示意图

3.1.4 TCP 协议通信过程中的问题

当然，网络传输通常是不可靠的，而 TCP 协议是可靠地，在网络传输过程中会出现各种各样的问题，例如网络拥塞，数据超时，丢包，半包，粘包等。所以了解 TCP 是运用什么机制和算法保证了网络的可靠传输是非常关键且必要的。

- Nagle 算法

当较小数据包在进行网络传输是，往往会浪费掉 TCP 头部信息，因为每次 TCP 传输时，头部信息占用 40 个字节，而数据包却只有一两个字节的时候，我们需要减少网络发送包的数量来提高传输效率，节省带宽。例如常用的 Telnet 应用就是如此。Nagle 算法主要是用来避免发送较小的数据包。”要求一个 TCP 连接上最多只能有一个未被确认的未完成的分组，该分组在确认到达之前不能发送给其他小分组，相反，TCP 收集这些小分组，并在确认到来时以一个分组的方式发出去”。

这种算法虽然能够降低小数据传输下的网络带宽，但是有时候一些对实时性要求非常高的应用，需要关闭 Nagle 算法。

缺省的，Nagle 算法是默认开启的，如果需要关闭该算法，可使用 TCP_NODELAY 选项来开启或者关闭 Nagle 算法。

- 超时重传

TCP 提供了可靠的网络传输,它使用的方法之一是对接收到的数据进行确认,但是由于网络是不可靠的,数据和确认都有可能在传输过程中丢失,TCP 在发送数据时设置一个定时器，若在定时器超时后还没有收到确认，就认为发送的数据已经丢失了，需要重传该数据。

- 拥塞控制

当数据从一个网络传输速度较快的局域网向一个网络传输较慢的广域网发送数据时便可能发生拥塞,当多个数据流到达一个路由器，而路由器的输出流小鱼这些输入流的总和时也会发生拥塞。

TCP 采用慢启动，拥塞避免算法，快速重传与快速回复算法来解决拥塞问题。

3.2 HTTP 协议

超文本传输协议(HTTP，Hypertext Transfer Protocol)是当今互联网上使用率最高的应用层网络协议。其作用不仅仅是进行最初设计时用于依靠浏览器所进行的超文本传输，也大量的运用于服务端后台，为手机 App，游戏等作为数据支撑，甚至被用来作为微服务之间的中间件 RPC 协议使用。而学习 HTTP 协议最好的资料就是阅读由 Ted Belson 组织协调万维网协议和互联网工程小组共同合作研究，所发布的 RFC 文档，其中 HTTP1.1 协议在 RFC2616 中有详细规范定义。

3.2.1 HTTP 协议定义

HTTP 协议属于应用层协议，是基于传输层 TCP 协议上进行数据传输的。它严格规范了浏览器和服务端的数据传输格式，最初是服务端用来向浏览器传输超文本信息，即 HTML 文件的。HTTP 协议采用了请求/响应模型。浏览器通过 URL 向服务端发

送一个请求头，请求头有一系列的 K/V 对，并且需严格遵循 HTTP 协议规范，服务器收到请求头，并返回相应的响应头和响应正文，同样需要严格遵循 HTTP 协议规范。HTTP 协议的默认端口号为 80。

3.2.2 HTTP 报文格式

浏览器与服务器进行请求头和响应头数据传输时，传输数据需严格按照 HTTP 规范。以下是 HTTP 报文格式。

HTTP 请求报文主要由请求行，请求头部和请求正文三部分组成，如图 3.2 所示。

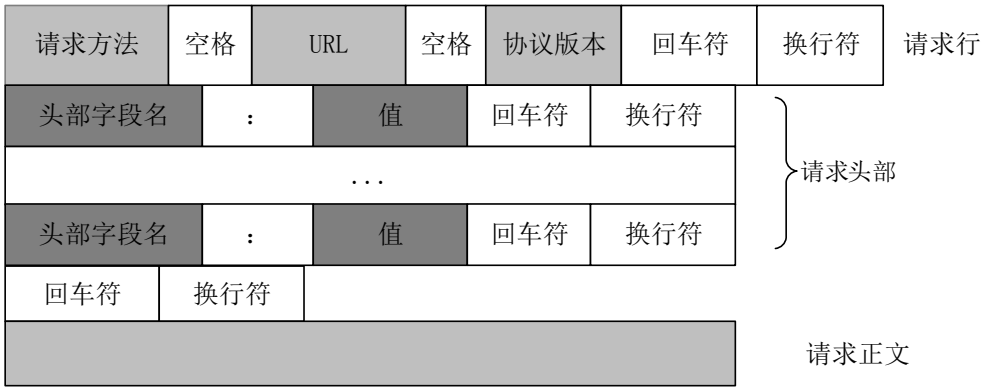

图 3.2 HTTP 请求报文格式

1. 请求行：由三部分组成，请求方法、URL、协议版本号、之间由空格分隔
   - 请求方法：GET、POST、PUT、HEAD、DELETE、OPTIONS、TRACE 以及扩展方法。
   - URL：www.example.com/example
   - 协议版本号：HTTP/主版本号.次版本号，例如：HTTP/1.0 和 HTTP/1.1
2. 请求头部:请求头部为请求报文添加了一些附加信息，由 K/V 对组成。每一行，Key 和 Value 之间用冒号分隔，并且以’\r\n’结束。如表 3.2 所示。

表 3.2 HTTP 请求头说明

请求头	说明
Host	接收请求的服务器地址，可以是 IP:PORT 或者域名
User-Agent	各浏览器内核名称
Connection	指定与连接相关的属性，如 Connection: Keep-Alive
Accept-Charset	浏览器客户端可接收的编码格式
Accept-Encoding	浏览器客户端可接收数据压缩格式
Accept-Language	浏览器客户端可接收的语言

请求头部的最后一行为空行，表示请求头部结束，空行之后就是要传输的正文数据，且这一行必不可少。

3. 请求正文：

GET 请求无请求正文，POST 请求有请求正文。

GET 请求数据格式如图 3.3 所示。

```
GET / HTTP/1.1\r\n
Accept: text/html, application/xhtml+xml, */*\r\n
Accept-Language: zh-CN\r\n
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; rv:11.0) like Gecko\r\n
Accept-Encoding: gzip, deflate\r\n
Host: www.baidu.com\r\n
Connection: Keep-Alive\r\n
 [truncated]Cookie: BD_UPN=1126314351; BD_HOME=0; BAIDUID=A16BF0121D8C32461B34A0C791
\r\n
[Full request URI: http://www.baidu.com/]
[HTTP request 1/1]
```


图 3.3 HTTP GET 请求示例

POST 请求数据格式如图 3.4 所示。

```
POST /q HTTP/1.1\r\n
Connection: close\r\n
Content-Type: application/json\r\n
User-Agent: User-Agent: SogouMobileTool\r\n
Content-Length: 573\r\n
Host: security.ie.sogou.com\r\n
\r\n
[Full request URI: http://security.ie.sogou.com/q]
[HTTP request 1/1]
[Response in frame: 130389]
JavaScript Object Notation: application/json
```


图 3.4 HTTP POST 请求示例

HTTP 响应报文格式：

HTTP 响应报文格式同样由响应正文、响应头部和状态行三部分组成。如图 3.5 所示。

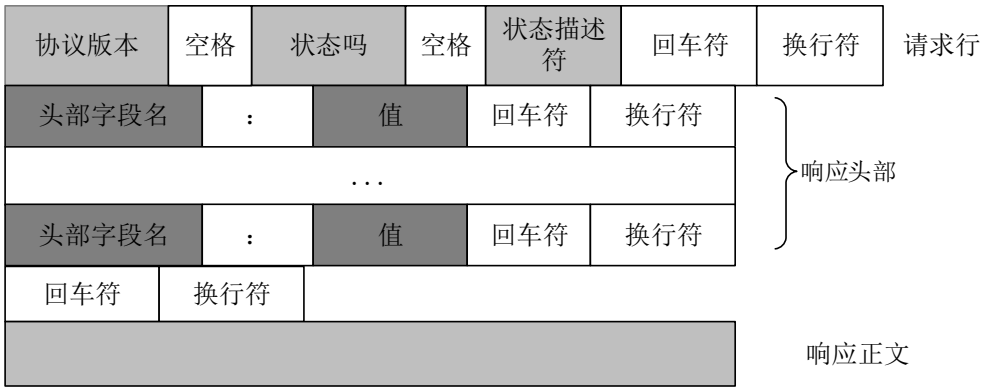

图 3.5 HTTP 响应报文格式

1. 状态行：由三部分组成，分别是：协议版本，状态码，状态描述符，之间由空格隔开。

   常见状态码列举如 3.3 表所示。

表 3.3 HTTP 状态码

状态码	说明
200	响应成功。

302	重定向。
400	客户端请求语法错误。
403	服务端收到客户端请求，但拒绝为其提供服务。
404	请求资源不存在。
500	服务器内部错误。

2. 响应头部：与请求头部类似，为响应报文添加额外的附加信息。
   常见响应头部如表 3.4 所示。

表 3.4 HTTP 响应头说明

响应头	说明
Server	服务器的名称和版本。
Content-Type	响应正文类型。
Content-Length	响应正文长度。
Content-Charset	响应正文使用的编码。
Content-Encoding	响应正文使用的数据压缩格式。
Content-Language	响应正文使用的语言。

响应头数据结构如图 3.6 所示：

```
HTTP/1.1 200 OK\r\n
Date: Fri, 26 Jun 2015 06:07:53 GMT\r\n
Server: Apache\r\n
Last-Modified: Mon, 08 Jun 2015 07:17:50 GMT\r\n
ETag: "75b-517fc707d2380"\r\n
Accept-Ranges: bytes\r\n
Vary: Accept-Encoding,User-Agent\r\n
Content-Encoding: gzip\r\n
Content-Length: 1054\r\n
Connection: Keep-Alive\r\n
Content-Type: image/svg+xml\r\n
\r\n
[HTTP response 1/1]
[Time since request: 0.191899000 seconds]
[Request in frame: 944]
Content-encoded entity body (gzip): 1054 bytes -> 1883 bytes
```


图 3.6 HTTP 响应报文示例

3. 响应正文：服务端返回的响应正文数据，如静态文件(HTML,CSS,JS,IMAGE 等)，动态脚本(PHP,APS,JSP 等)或者 JSON 数据。

3.2.3 HTTP 通信流程

HTTP 协议是基于 TCP 之上的应用层协议，所以 HTTP 协议的整个通信流程建立在 TCP 协议双方建立连接的基础上。整个通信流程如下：

1. 客户端(一般为各浏览器)发送 HTTP 请求报文给服务器
2. TCP 协议封装 HTTP 请求报文，作为 TCP 数据包
3. 双方经过 TCP 建立连接。

4. 服务器解析 TCP 报文并返回相应的 HTTP 响应报文，并将响应报文封装为 TCP 数据包发回客户端
5. 客户端接收服务端返回的 HTTP 响应报文，浏览器解析并展示相应的资源数据。
6. 双方执行 TCP 关闭连接过程。

3.2.4 客户端缓存与状态记录

很多时候，服务端的很多静态资源长时间内是没有变化的，为了节约网络带宽和提高性能，就希望浏览器端能够缓存资源，当服务端资源不发生改变时，服务端将不发送响应正文数据到浏览器，由浏览器读取本地缓存的资源。一个典型的例子如图 3.7 所示。

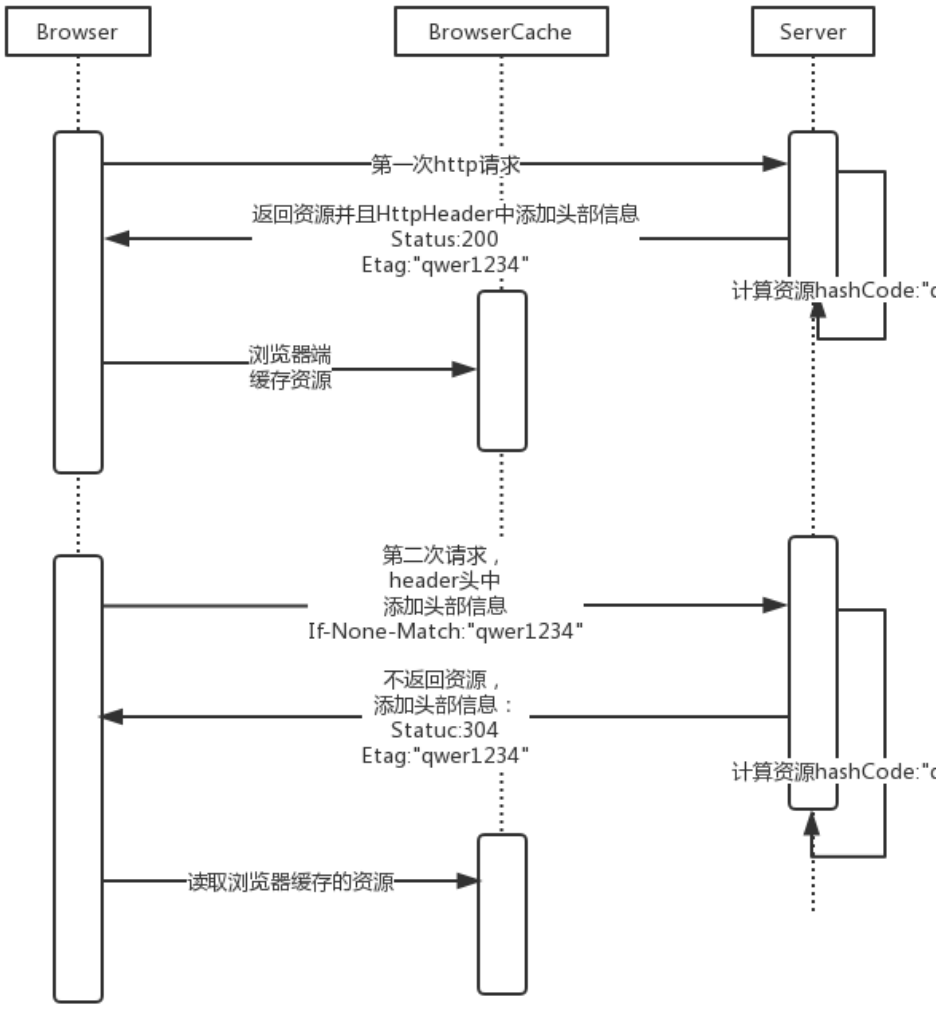

图 3.7 HTTP 缓存示例

然而 HTTP 协议属于无态协议，而很多时候都需要记录用户状态，如登录等。那么它有事通过什么样的方式来记录用户状态呢？浏览器端，将维持 Cookie 数据，然后在每次请求是都会在请求头添加 Cookie：请求行，与服务端进行状态同步。而服务端自己可以在应用程序中缓存 Session 数据结构来记录状态。

3.3 Unix I/O模型分析

在了解 I/O 模型之前，需先对同步，异步，阻塞，非阻塞这几个概念有非常透彻的理解。

3.3.1 同步，异步，阻塞，非阻塞

在讨论什么是同步与异步，阻塞与非阻塞时，该应根据操作系统内核来说的。因为从应用层面来说，很多时候可以通过编码风格来改变程序运行状态，例如轮询，基于事件机制等。

而对于操作系统内核缓冲区到应用程序数据区这一交互过程才是需要讨论同步，异步，阻塞，非阻塞的关键之处。

阻塞与非阻塞：即数据从操作系统内核缓冲区拷贝到应用程序数据区这一时间段内应用程序所处的状态，当应用程序发起一个系统调用，到系统内核缓冲区数据就绪返回数据到应用程序，应用程序一直处于休眠状态，称之为阻塞。反之，当操作系统内核数据是否就绪，都立刻返回信息通知应用程序数据就绪或者直接返回错误信息，应用程序不用处于休眠状态，只需告诉操作系统如果数据未就绪，请返回一个错误信息，不要让我进入休眠状态，称之为非阻塞。

同步与异步：阻塞与非阻塞关注的是数据从操作系统内核拷贝到应用程序缓冲区这一过程中应用程序是否处于休眠状态。同步与异步关注的是消息通信机制，当应用程序主动发起 I/O 调用，并等待操作系统数据就绪返回就如休眠状态(阻塞)或者忙轮询请求操作系统数据是否就绪(非阻塞)都属于同步状态，应用程序并不能去做其他事情；当应用程序向操作系统发起一个 I/O 请求，然后应用程序便可以去做其他事情，只需等待操作系统完成 I/O 操作并且数据就绪时，主动告知应用程序，应用程序接收到操作系统消息，继续执行 I/O 操作即可，称之为异步。

了解完这几个概念之后，就需要了解 Linux 下的几种 I/O 模型了。

3.3.2 Unix 下的五种 I/O 模型

- 阻塞式 I/O：阻塞式 I/O 是使用最广泛的 I/O 模式。绝大多数网络应用程序使用的都是阻塞式 I/O，且大多数语言默认的 Socket 编程，使用的也都是阻塞式 I/O。对于 UDP Socket 来说，数据就绪的标志是相对简单的，即收到整个 UDP 数据报或者，没有收到。而对于 TCP 数据就绪标记就要复杂得多，需要附加一些其他的变量信息。在下图中，网络应用程序调用操作系统内核 API 函数 recvfrom，然后应用程序进入休眠状态，直到操作系统内核数据就绪，然后操作系统将数据拷贝到进程的数据缓存区中。（如果系统调用收到一个中断信号，则它的调用会被中断）这个进程在调用 recvfrom 一直到从 recvfrom 返回这段时间是阻塞的。当 recvfrom 正常返回时，进程继续它的操作。如图 3.8 所示。

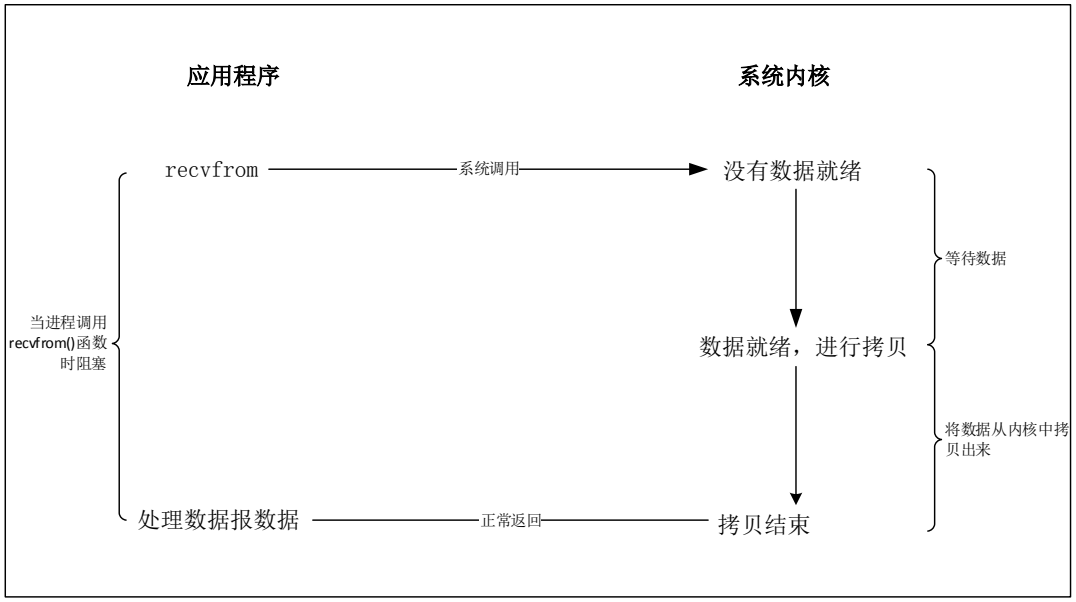

图 3.8 阻塞式 I/O

- 非阻塞式 I/O：将一个套接字设置为非阻塞式 I/O，当操作系统内核数据未就绪是，会直接返回错误信息，而不是让程序进入休眠状态。

图 3.9 描了非阻塞模式 I/O 的系统调用过程。

由于网络应用程序使用了非阻塞式 I/O，前三次调用操作系统内核 API 函数 recvfrom 时，由于操作系统还没有收到网络数据或者内核数据未就绪，则立刻返回一个 EWOULDBLOC 错误信息。当第四层调用 recvfrom 函数时，网络数据到达，操作系统内核数据也已经就绪，则将数据拷贝到应用程序的缓冲区中，recvfrom 正常返回，应用程序对接收到的数据进行处理。如下图所示，非阻塞式 I/O 需要应用程序不断的轮询(polling)内核来检测 I/O 操作是否就绪，这是非常浪费 CPU 资源的。所以这种模式使用的并不是很普遍。

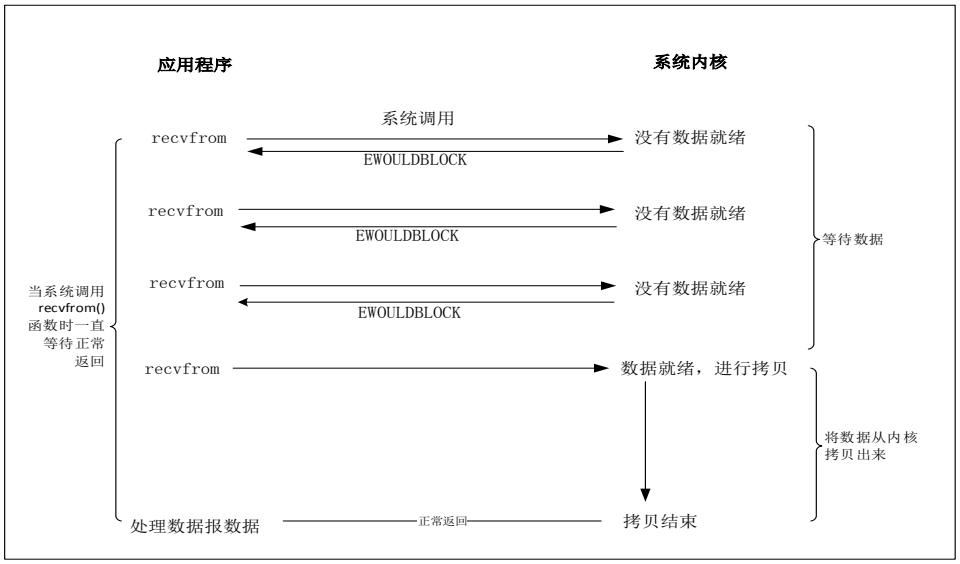

图 3.9 非阻塞式 I/O

- I/O 复用(select，poll,epoll)：I/O 多路复用技术与普通的 I/O 操作所调用的操作系统内核 API 是不同的，采用 I/O 多路复用需要调用 select()和 poll()

函数，当 select()有数据就绪，则再调用 recvfrom 函数即可。当然在调用 select()时同样是阻塞的。而调用 recvfrom 时已经有数据返回，则不会导致应用程序阻塞。下图说明了它的工作方式：

当调用 select()函数时，同样是阻塞的，应用程序需要等待 select()函数 Socket 事件就绪(读事件或者写事件)，当时间就绪时，应用程序在调用 recvfrom 函数来进行数据拷贝操作即可，此过程因为数据已经就绪，不会导致应用程序阻塞。看上去 select()或 poll()只是将阻塞的位置发生转移，并没有提高性能，而且还增加了函数调用的次数。但是，多路复用 I/O 的高级之处就在于其能同时处理多个文件描述符，只要这多个文件描述符其中一个进入事件就绪状态，select()函数就立刻返回。如图 3.10 所示。

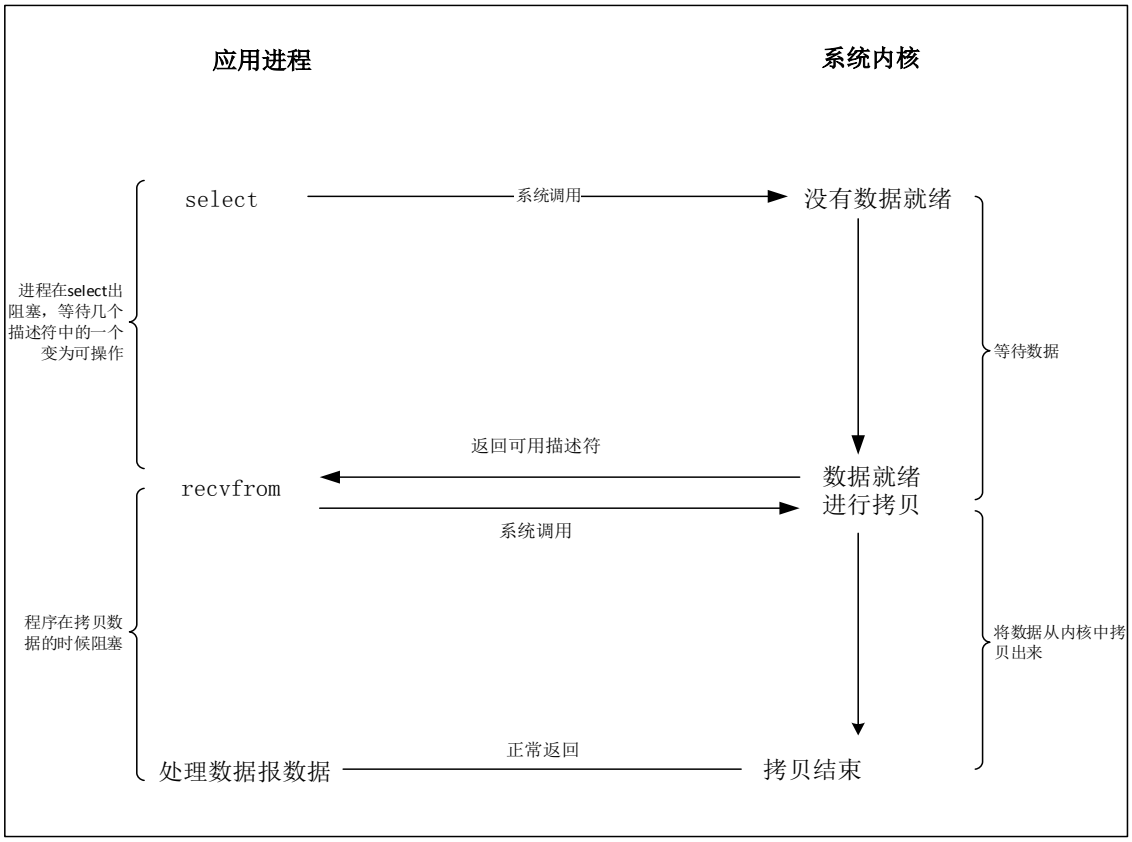

图 3.10 I/O 复用

假如一个正在运行的网络应用程序，要同时处理网络 I/O 和本地文件 I/O。在程序被等待文件 I/O 就绪所阻塞时，假如应用程序被意外停止，那么服务端就会发送一个 FIN 标志来断开 TCP 连接。此时程序被阻塞在等待文件 I/O 上，在处理网络 I/O 之前，都无法接受 FIN 标志服来断开 TCP 连接，就不能够使用阻塞模式的套接字。

以下是 IO 多路复用技术最常用的使用场景：

- 当客户端需要同时处理多个文件描述符，即在网络连接并发数较高的情况下，使用多路复用 I/O 能显著提高性能。
- 服务器需要同时处理已经真在监听的 Socket 和已经连接就绪的 Socket。

- 服务器需要同时使用 UDP 和 TCP 两种协议。
- 服务器需要同时使用多种网络协议。
- I/O 多路复用技术并不局限于网络连接，更是一种事件通知机制的思想，在很多编程中都能够简单这种思想。

- 信号驱动式 I/O(SIGIO)：操作系统在内核文件描述符就绪时，使用一个 SIGIO 信号来通知应用程序。

使用信息驱动式 I/O 需要允许 Socket 使用该模式，然后调用操作系统内核 API 函数 SIGIO。当应用程序调用 SIGIO 时，操作系统会立即返回，此时应用程序可以去做其他事情，只需要等待操作系统返回 SIGIO 信号通知即可，应用程序当收到 SIGIO 信号通知时，再调用 recvfrom 函数进行数据拷贝操作即可。如图 3.11 所示。

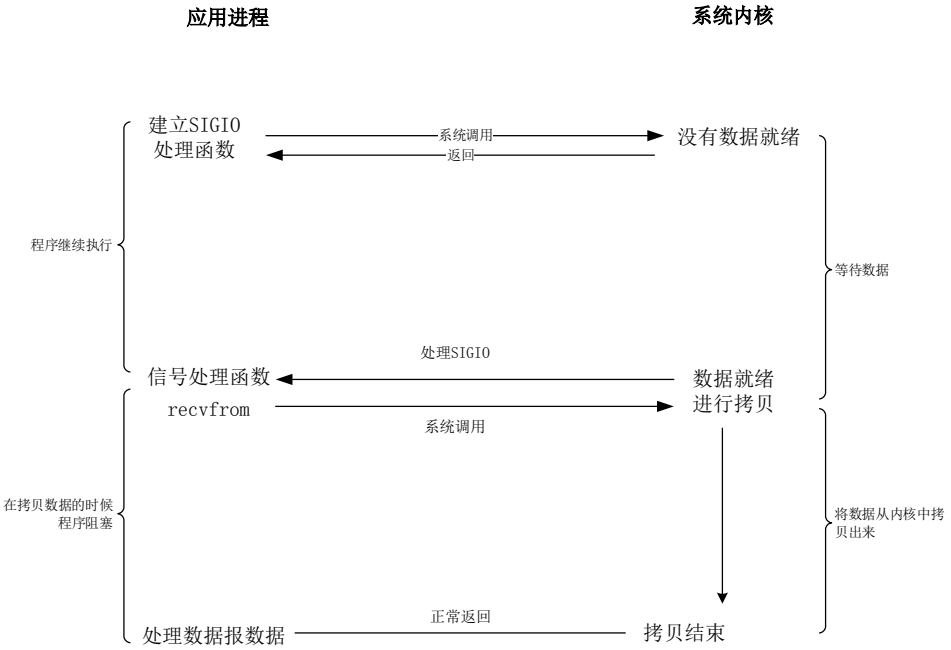

图 3.11 信号驱动式 I/O

信号驱动式 I/O 整个过程都是异步的，应用程序调用 SIGIO 函数后，只需等待操作系统返回 SIGIO 信号，之后再去进行数据拷贝操作即可，在等待操作系统返回 SIGIO 信号期间都是可以去做其他事情的。

基于 Berkeley 接口的 Socket 信号驱动 I/O 使用信号 SIGIO。有的系统 SIGPOLL 信号， 它也是相当于 SIGIO 的。以下是在一个 Socket 上使用信号驱动式 I/O 的步骤说明：

(1)设定一个能够处理 SIGIO 信号的函数。

(2)设定 Socket 拥有者。一般来说是使用 fcntl 函数的 F_SETOWN 参数来进行设定拥有者。

(3)允许 Socket 使用异步 I/O。一般是通过调用 fcntl 函数的 F_SETFL 命令，O_ASYNC 为参数来实现。

- 异步 I/O(POSIX 的 aio_系列函数)：异步 I/O 从应用程序向操作系统内核发起

一个异步 I/O 操作，到读取网络数据整个阶段都是异步的。操作系统在完成所有的 I/O 操作，并且数据拷贝完成之后通知应用程序来处理网络数据即可。异步 I/O 与信号驱动式 I/O 的不通之处就在于，数据拷贝都由操作系统完成，而不需要应用程序调用 recvfrom 函数，因为调用 recvfrom 函数进行数据拷贝也是阻塞的。异步 I/O 可以真正意义上被称为异步且非阻塞的。

如图 3.12 所示，当应用程序向操作系统北河发起一个异步 I/O 操作时，需要向操作系统传递文件描述符，缓冲区指针和缓冲区大小等参数数据，以及需要定义内核结束所有操作所有执行的回调函数。之后应用程序就可以去做其他事请，当操作系统回调我们定义好的函数，应用程序再执行回调函数逻辑即可。

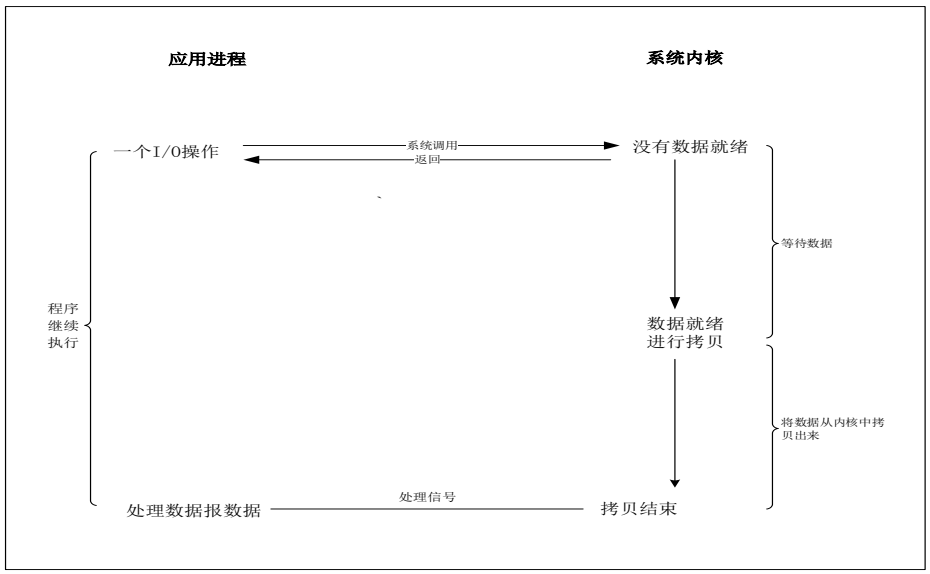

图 3.12 异步 I/O

- 几种 I/O 模式的比较

图 3.13 比较了五种 I/O 模型。

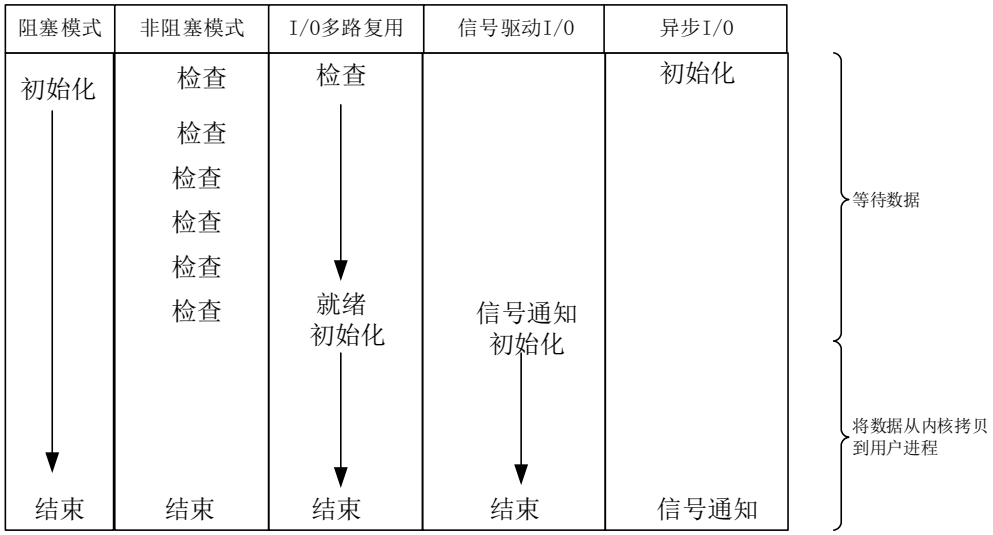

图 3.13 I/O 模型对比

以上就是 Linux 下五种 I/O 模式，以及五种 I/O 模式的区别，开发一款高性能服务器，选择合适的 I/O 模式是最关键的因素之一。

3.4 Tomcat I/O 模型与线程模型分析

上节分析了 Unix 下的五种 I/O 模型，在熟悉了 Unix 下的各种 I/O 模型之后，想要开发一款高性能，高可用的商用服务器还需要深厚的多线程编程功底。于是，不妨了解一款成熟的商用服务器，研究其源码。学习其 I/O 模式，多线程模型以及程序架构，有助于帮助更好的了解服务器。

Tomcat 拥有两种 I/O 模型，BIO 以及 NIO。两种 I/O 分别应用于不用的并发场景。

BIO 即阻塞式 I/O，应用于并发数相对少的场景。每个 socket 对应一个线程，用来接收 socket(HTTP)请求以及进行数据 decode，encode 操作，再将响应数据发送回客户端。由线程池统一管理所有线程资源，避免频繁创建线程所带来的开销。

NIO 即非阻塞 I/O，也就是多路复用 I/O。对于 Java 语言来说，只需使用 NIO 对应的 JDK 即可，其底层对于不同的操作系统有不同的处理方式(Linux2.6 之前是 select,poll,2.6 之后为 epoll,Windows 是 IOCP)。其可用于并发数较高的场景下。

第四章 服务器设计与实现

4.1 Java 与 JVM 平台概述

Java 是一门面向对象的工程性强类型语言，它不仅吸收面向对象的可重用，可扩展，健壮性等优点，还摒弃了指针等难以理解的概念。最为突出的是其垃圾回收机制，不需要编程人员手动申请内存，释放内存，往往这些操作都是危险且极其容易造成内存泄漏或内存溢出的，内存的申请与释放操作都由 JVM 来完成，极大降低了编程难度。Java 不仅可以用来编写桌面应用程序，Web 应用程序，嵌入式系统，更被广泛运用于服务端，分布式，中间件等领域。

Java 语言是基于 JVM 的，故其中一个重要的特性是平台无关性，即所说的一次编写，到处运行特性。

JVM(Java Virtual Machine) 是一种用于计算设备的规范，它是一个虚构出来的计算机，是通过在实际的计算机上仿真模拟各种计算机功能来实现的。

4.2 服务器整体架构

本节将介绍服务器的整体架构，服务器所采用的 IO 模型以及线程模型以及各模块之间的作用。阐述服务器的整个运行流程，介绍服务器是怎样从接受一个 TCP 请求，如何解析 HTTP 协议信息请求头，以及如何处理请求头，最后将处理后的结果返回到客户端。

4.2.1 I/O 模型与线程模型

服务器采用 IO 多路复用模型，上节分析 Unix 五种 IO 模型的区别以及优劣，故这里不重复解释。虽然 NIO 并不意味着高性能，但是在大多数情况下 NIO 模型的性能是要高于普通的阻塞式 IO 的，尤其是在处理高并发连接的情况下，当然在连接数少的时候 NIO 性能可能和阻塞式 IO 相差无几，甚至因为线程池对于线程资源申请的开销，性能还要略低。最好的办法是实现两种 IO 模式，根据并发数或者配置文件的方式，让用户根据自身 web 应用的情况选择合适的 IO 模型。

服务器采用多 Reactor 模型。即多个 Selector 监听请求注册时间，Selector 数量需通过配置文件让用户根据自身机器硬件配置选择 Selector 数量，默认为 2*CPU-1，使得性能最大化。然后再为 TCP 的 encode 和 decode 分配子线程去处理每个子时间。其线程模型图 4.1 所示。

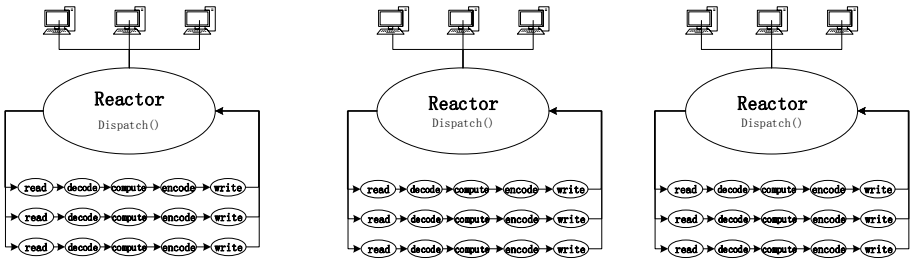

图 4.1 多 Reactor 模型

4.2.2 整体架构与执行流程

服务器主要采用分层模式与管道/过滤器模式架构。将整个冗长的工作流程分为 4 个
层次。

- 第一层：Connector 即 IO 层，主要工作为监听并处理 TCP 连接，并将返回的字节流通过 TCP 写回客户端。
- 第二层：Server 层，主要工作为数据的 decode 与 encode，即解析从 IO 层接收的 TCP 数据，解码后封装成 Request 对象交给 Handler 层处理，以及获取 Handler 层传递回来的 Response 对象，将数据进行编码返回给 IO 层。
- 第三层：Handler 层，次层采用管道/过滤器模式，将 Server 层的 Request 对象经过一系列的 Handler 链处理，最终选择是否直接返回静态资源或错误信息或交给 Web App 进行用户业务逻辑处理。将返回数据封装成 Response 对象返回给 Server 层。
- 第四层：App 层，次层严格来说已经不属于服务器的整体架构内，需要用户开发自己的 web App，服务器只是通过动态代理查询是否有可匹配的 web App，如果有则执行用户 web App 自身的业务逻辑，反之则返回正确的错误信息。
  执行流程图如图 4.2 所示

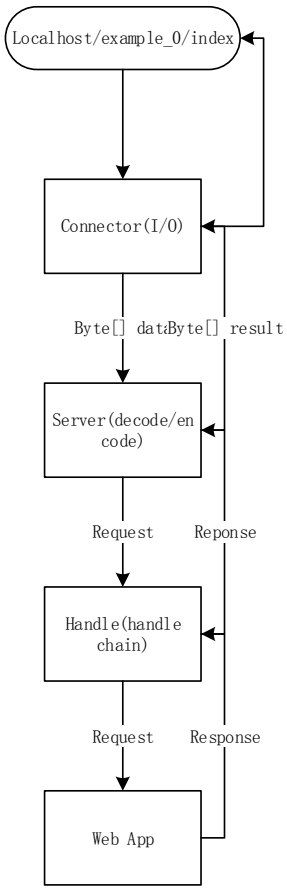

图 4.2 层次架构流程图

在第三层 Handler 层中，采用了管道/过滤器架构模式，将 Request 进过 HttpHandler->SessionHandler->ApploadHandler 三个基础 handler 链按照单向链表先后顺序处理后，再将 Request 传递给 web App 处理。Handler 的核心方法返回值类型为布尔类型，倘若链中的其中一个 handler 返回 false 或者抛出异常则 handler 链不向下执行。

如：HttpHandler 监听到 Reuqest 中的 URL 为静态资源类型，则直接返回 false，将静态资源返回给 Server，或者通过判断浏览器是否缓存，直接返回响应头信息 Server。而不向下执行其他的 Handler。

流程图如图 4.3 所示。

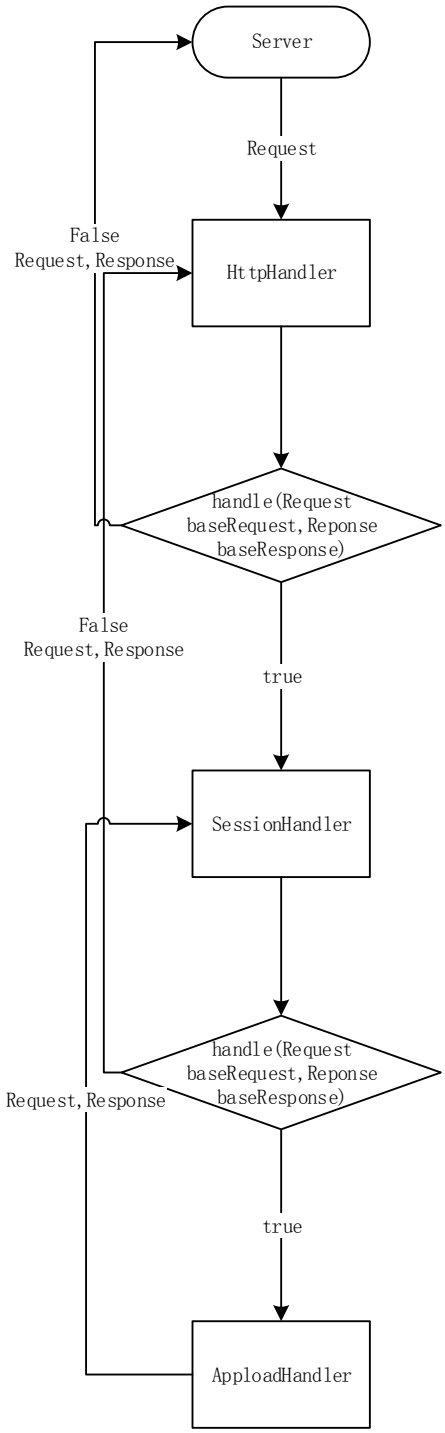

图 4.3 管道 handler 链流程图

4.2.3 模块说明

整个项目架构分为 9 个模块:

dam-annotation, dam-bridge, dam-synamic-syntax, dam-exception, dam-http, dam-io, dam-server, dam-start, dam-util。

每个模块的作用 说明如下:

- dam-annotatio: 所有注解类都在这个模块之中，用于开发 web 应用程序所用的声明式注解。

- dam-bridge：桥梁模块，用于连接 server 模块与 webapp 之间通信的桥梁。
- dam-synamic-syntax：动态脚本解析模块，用于解析自定义的.tsp 动态脚本文件。
- dam-exception：所有的自定义异常类。
- dam-http：定义 HTTP 请求，响应所需要的数据结构，以及根据 RFC2616 所定义 HTTP 协议消息枚举，常量等。
- dam-io：采用多路复用 IO，监听 TCP 连接，处理网络事件，将 HTTP 请求数据传递给 Server 模块，以及将 Server 模块返回的数据写回客户端。
- dam-server:解析并处理HTTP消息,并将消息封装成Response对象交给handler链进行一系列的处理。
- dam-start：启动模块。
- dam-util：工具模块。

4.3 服务器编码与实现

本小节主要对服务器核心功能的编码实现进行讲解，介绍如何实现组件的生命周期，Java NIO 的 API 的基本使用，线程池的管理，以及设计模式的使用。

4.3.1 生命周期管理

服务器从接收一次浏览器发出的 HTTP 请求到将 HTTP 响应写回给浏览器定义为一个完整的生命周期。服务器中的组件 Server，Handler，Web App 都有自己的生命周期，对应着 HTTP 请求的开始于结束。

服务器将生命周期定为 5 个阶段，Running,Started,Stopping,Stopped,Failed。每个状态的说明如下：

- Running:组件正在启动，即接收 HTTP 请求后开始初始化组件。
- Started:组件已启动，即组件正在执行响应运算逻辑。
- Stopping:组件正在停止，组件执行完运算逻辑，将运算结果返回。
- Stopped：组件已停止，需要释放相应资源。
- Failed：组件异常。

首先定义 Lifecycle 接口，然后通过 AbstractLifeCycle 抽象类实现该接口，在抽象类中用状态机转换算法进行生命周期状态管理，然后 ContainerLifeCycle 又继承自 AbstractLifeCycle，然后定义 doStart()和 doStop()核心函数，由这两个核心函数来启动组件并改变生命周期状态，并且该类统一管理所有继承了 ContainerLifeCycle 的 Java Bean。ContainerLifeCycle 执行 doStart()或者 doStop()方法来遍历组件，调用组件所重写的 doStart()和 doStop()函数。

ContainerLifeCycle 类的代码如下：

```
package org.dam.utils.lifecycle;

import java.util.*;
```

```
import java.util.concurrent.CopyOnWriteArrayList;

/**
 * Created by geeche on 2018/2/3.
 */
public class ContainerLifeCycle extends AbstractLifeCycle implements
Destroyable {

    private final List<Bean> beans = new CopyOnWriteArrayList<Bean>();

    private boolean started = false;

    @Override
    protected void doStart() throws Exception {
        for(Bean bean : beans){
            if(bean.bean instanceof LifeCycle){
                LifeCycle lc = (LifeCycle)bean.bean;
                if(!lc.isRunning()){
                    lc.start();
                }
            }
        }
        started = true;
        super.doStart();
    }

    @Override
    protected void doStop() throws Exception {
        started = false;
        super.doStop();
        final List<Bean> reverse = new ArrayList<Bean>(beans);
        Collections.reverse(reverse);
        for(Bean bean : reverse){
            if(bean.bean instanceof LifeCycle){
                LifeCycle lc = (LifeCycle)bean;
                if(!lc.isRunning()){
                    lc.stop();
                }
            }
        }
        started = true;
        super.doStart();
    }
    @Override
    public void destroy() {
        final List<Bean> reverse = new ArrayList<Bean>(beans);
```

```
        Collections.reverse(reverse);
        for(Bean bean : reverse){
            if(bean.bean instanceof Destroyable){
                Destroyable de = (Destroyable) bean;
                de.destroy();
            }
        }
    }

    public boolean Contains(Object obj){
        for(Bean bean : beans){
            if(bean.bean == obj){
                return true;
            }
        }
        return false;
    }


    public boolean addBean(Object obj){
        return addBean(obj,!((obj instanceof LifeCycle) &&
((LifeCycle)obj).isStarted()));
    }
    public boolean addBean(Object obj,boolean managed){
        if(Contains(obj)){
           return false;
        }
        Bean = new Bean(obj);
        bean.managed = managed;
        beans.add(bean);

        if(obj instanceof LifeCycle){
            LifeCycle lc = (LifeCycle)obj;
            if(managed && started){
                try{
                    lc.start();
                }catch (Exception e){
                    throw new RuntimeException (e);
                }
            }
        }
        return true;
    }
    public void manage(Object bean){
        for(Bean b : beans){
            if(b.bean == bean){
```

```
                b.managed = true;
                return;
            }
        }
        throw new IllegalArgumentException();
    }

    public void ummanage(Object bean){
        for(Bean b : beans){
            if(b.bean == bean){
                b.managed = false;
                return;
            }
        }
        throw new IllegalArgumentException();
    }

    public Collection<Object> getBeans(){
        return getBeans(Object.class);
    }

    public <T>List<T> getBeans(Class<T> clazz){
        ArrayList<T> beanRes = new ArrayList<T>();
        for(Bean : beans){
            if(clazz.isInstance(bean.bean)){
                beanRes.add((T)bean.bean);
            }
        }
        return beanRes;
    }

    public <T>T getBean(Class<T> clazz){
        for(Bean : beans){
            if(clazz.isInstance(bean.bean)){
                return (T)bean.bean;
            }
        }
        return null;
    }

    public void removeBeans(){
        beans.clear();
    }
    public boolean removeBean(Object obj){
        Iterator<Bean> iterator = beans.iterator();
        while (iterator.hasNext()){
```

```
                Bean = iterator.next();
                if(bean.bean == obj){
                    beans.remove(obj);
                    return true;
                }
            }
            return false;
        }

        private class Bean{
            final Object bean;
            volatile boolean managed = true;
            Bean(Object bean){
                this.bean = bean;
            }
        }

}
```

整个关系类图如图 4.4 所示。

4.3.2 NIO 编码实现

服务器对于 TCP 连接统一采用 NIO 的方式。下面介绍 Java NIO 中基本 API 的在编码过程中的使用：

首先打开 Socket 连接并绑定端口：

```
public void open() throws IOException {
    synchronized (this){
            if(acceptorChannel == null){
            acceptorChannel = ServerSocketChannel.open();
            acceptorChannel.configureBlocking(true);
            acceptorChannel.socket().setReuseAddress(getReuseAddress());
            InetSocketAddress address = getHost() == null ? new
InetSocketAddress(getPort()) : new InetSocketAddress(getHost(),getPort());
            acceptorChannel.socket().bind(address);
            localPort = acceptorChannel.socket().getLocalPort();
            if(localPort < 0){
                throw new IOException("Server channel not bound");
            }
            addBean(acceptorChannel);
        }
    }
}
```

监听 TCP 连接并注册相关事件：

```
@Override
protected void accept(int acceptId) throws IOException,
InterruptedException {
    ServerSocketChannel serverSocketChannel;
    synchronized (this){
        serverSocketChannel = acceptorChannel;
    }
    if(serverSocketChannel != null && serverSocketChannel.isOpen() &&
selectorManager.isStarted()){
        SocketChannel socketChannel = serverSocketChannel.accept();
        socketChannel.configureBlocking(false);
        Socket socket = socketChannel.socket();
        configure(socket);
        selectorManager.register(socketChannel);
    }
}
```

更新事件：

```
@Override
public void doUpdateKey(){
    synchronized (this){
        if(getChannel().isOpen()){
            if(interestOps > 0){
                if(key == null && !key.isValid()){
                    SelectableChannel sc =
(SelectableChannel)getChannel();
                    if(sc.isRegistered()){
                        updateKey();
                    }else{
                        try {
                            key =
((SelectableChannel)getChannel()).register(worker.getSelector(),interes
tOps,this);
                        }catch (IOException e){
                            if(key != null && key.isValid()){
                                key.cancel();
                            }
```

```
                            if(open){
                                worker.destroyEndPoint(this);
                            }
                            open = false;
                            key = null;
                        }
                    }
                }else{
                    Logger.INFO("doUpdayeKey register event to
key,interestOps'{}'",interestOps);
                    key.interestOps(interestOps);
                }
            }else{
                if(key != null && key.isValid()){
                    key.interestOps(0);
                }else{
                    key = null;
                }
            }
        }else{
            if(key != null && key.isValid()){
                key.cancel();
            }
            if(open){
                open = false;
                worker.destroyEndPoint(this);
            }
            key = null;
        }
    }

}
```

Socket 相关配置

```
protected void configure(Socket socket) {
    try {
        socket.setTcpNoDelay(true);
        socket.setSoLinger(true,2);
```

```
    }catch (Exception e){
        Logger.ERROR("socket configure error:{}",socket);
    }
}
```

数据读写

```
@Override
public int fill(Buffer buffer) {
    if(inShut){
        return -1;
    }
    int len = 0;
    if(buffer instanceof NioBuffer) {
        NioBuffer nioBuffer = (NioBuffer)buffer;
        ByteBuffer byteBuffer = nioBuffer.byteBuffer();
        synchronized (byteBuffer){
            try {
                int bufferSize = NioBuffer.DEFAULT_BUFFER_SIZE;
                len = channel.read(byteBuffer);
            }catch (IOException e){
            }finally {
                if(!isInputShutdown()){
                    try {
                        shutdownInput();
                    } catch (IOException e) {

                    }
                }
            }
        }
    }
    return len;
}
@Override
public int flush(Buffer buffer) throws IOException{
    int len = 0;
    if(buffer instanceof NioBuffer){
```

```
            final NioBuffer nioBuffer = (NioBuffer)buffer;
            final ByteBuffer byteBuffer = nioBuffer.byteBuffer();
            buffer.flip();
            len = channel.write(byteBuffer);
        }
        return len;
    }
```

关闭连接：

```
protected final void shutdownChannelInput()throws IOException{
    inShut = true;
    if(channel.isOpen()){
        if(socket != null){
            try {
                if(!socket.isInputShutdown()){
                    socket.shutdownInput();
                }
            }catch (SocketException e){
                //TODO
            }finally {
                if(outShut){
                    close();
                }
            }
        }
    }
}
protected final void shutdownChannelInput()throws IOException{
    inShut = true;
    if(channel.isOpen()){
        if(socket != null){
            try {
                if(!socket.isInputShutdown()){
                    socket.shutdownInput();
                }
            }catch (SocketException e){
                //TODO
            }finally {
```

```
                if(outShut){
                    close();
                }
            }
        }
    }
}
```

以上就是 NIO 基本 API 在项目中的使用。

4.3.3 多 Reactor 模型及编码实现

接下来看看如何采用线程池+多 Selector 实现的多 Reactor 模型。

1. 使用并发阻塞队列 ConcurrentLinkedQueue 储存所有的事件，保证线程安全性.
2. 根据当前线程的索引选取响应的 SelectorWorker，使用死循环执行当前 SelectorWorker 的 dowork 方法
3. SelectorWorker 中 dowork()遍历阻塞队列，取出里面的事件，将事件交给子线程去处理，知道队列为空，并且 Selector 中的感兴趣的操作为 0 结束 doWork()函数。
4. 所有线程的申请统一交给自定义的线程池 QuerueThreadPool 处理。

核心代码如下：

```
@Override
protected void doStart() throws Exception {
    workers = new SelectorWorker[workLength];
    for(int i=0;i<workers.length;i++){
        workers[i] = new SelectorWorker(i);
    }
    super.doStart();
    Logger.INFO("SelectManager========doStart
workLength:'{}'",getWorkLength());
    for(int i=0;i<getWorkLength();i++){
        final int id = i;
        boolean working = dispatch(new Runnable() {
            @Override
            public void run() {
                String name = Thread.currentThread().getName();
                int priority = Thread.currentThread().getPriority();
                try {
                    SelectorWorker selectorWorkers[] = workers;
```

```
                    if(selectorWorkers == null){
                        return;
                    }
                    SelectorWorker worker = selectorWorkers[id];
                    Thread.currentThread().setName(name+" Selector"+id);
                    if (getSelectorPriorityDelta()!=0)

Thread.currentThread().setPriority(Thread.currentThread().getPriority()
+getSelectorPriorityDelta());
                    Logger.INFO("Starting {} on
{}",Thread.currentThread(),this);
                    while (isRunning()){
                        try {
                            worker.doWork();
                        } catch(IOException e) {

Logger.INFO(Logger.printStackTraceToString(e.fillInStackTrace()));
                        } catch(Exception e)
Logger.INFO(Logger.printStackTraceToString(e.fillInStackTrace()));
                        }
                    }
                } finally {
                    Logger.INFO("Stopped {} on
{}",Thread.currentThread(),this);
                    Thread.currentThread().setName(name);
                    if (getSelectorPriorityDelta()!=0)
                        Thread.currentThread().setPriority(priority);
                }
            }
        });
    }
}
public void doWork() throws IOException{
    try {
        selectedThread = Thread.currentThread();
        final Selector currentSelector = selector;
        if(currentSelector == null){
```

```
            return;
        }
        Object work;
        int works = workQueue.size();
        while ( works-- >0 && (work = workQueue.poll())!=null){
            Logger.INFO(">>>>>>>dowork<<<<<<<<");
            Channel channel = null;
            SelectionKey key = null;
            if(work instanceof EndPoint){
                Logger.INFO("do work endpoint:{}",work);
                final SelectChannelEndPoint endPoint =
(SelectChannelEndPoint)work;
                channel = endPoint.getChannel();
                endPoint.doUpdateKey();
            }else if(work instanceof ChannelAndAttachment){
                final ChannelAndAttachment caa = (ChannelAndAttachment)
work;
                final SelectableChannel sc = caa.channel;
                channel = sc;
                final Object att = caa.attachment;
                if( (sc instanceof SocketChannel) &&
((SocketChannel)sc).isConnected() ){
                    key = ((SocketChannel)
sc).register(selector,SelectionKey.OP_READ,att);
                    SelectChannelEndPoint endPoint =
createEndPoint((SocketChannel)sc,key);
                    key.attach(endPoint);
                    endPoint.schedule();
                }else if(channel.isOpen()){
                    key = sc.register(selector,SelectionKey.OP_CONNECT);
                }
            }else if(work instanceof SocketChannel){
                Logger.INFO("newly socketChannel:{}",work);
                final SocketChannel socketChannel = (SocketChannel)work;
                channel = socketChannel;
                key =
socketChannel.register(selector,SelectionKey.OP_READ,null);
```

```
                SelectChannelEndPoint endPoint =
createEndPoint(socketChannel,key);
                Logger.INFO("newly endPoint:{}",endPoint);
                key.attach(endPoint);
                endPoint.schedule();
            }else if(work instanceof ChangeTask){
                ((Runnable)work).run();
            }
        }
        int select = selector.selectNow();
        if(select == 0 && selector.selectedKeys().isEmpty()){

        }

        if(selector == null || !selector.isOpen()){
            return;
        }
        for(SelectionKey selectionKey : selector.selectedKeys()){
            Logger.INFO("current selectionKey:{} event is read : '{}' or
write :'{}'",

selectionKey,selectionKey.isReadable(),selectionKey.isWritable());
            SocketChannel socketChannel = null;
            try {
                if(!selectionKey.isValid()){
                    selectionKey.cancel();
                    SelectChannelEndPoint endPoint =
(SelectChannelEndPoint)selectionKey.attachment();
                    if(endPoint!=null){
                        endPoint.doUpdateKey();
                    }
                    continue;
                }
                Object attachment = selectionKey.attachment();
                if(attachment instanceof SelectChannelEndPoint){
                    if(selectionKey.isWritable() ||
selectionKey.isReadable()){
```

```
                        final SelectChannelEndPoint endPoint =
(SelectChannelEndPoint)attachment;
                        Logger.INFO("selectKey attched
endpoint:{},readBlock:'{}' and writeBlock:'{}'",

endPoint,endPoint.isReadBloking(),endPoint.isWriteBloking());
                        endPoint.schedule();
                    }
                }else if(selectionKey.isConnectable()){
                    socketChannel = (SocketChannel)
selectionKey.channel();
                    boolean connected = false;
                    try {
                        connected = socketChannel.finishConnect();
                    }catch (Exception e){
                        Logger.ERROR("");
                    }finally {
                        if(connected){
                selectionKey.interestOps(SelectionKey.OP_READ);
                            SelectChannelEndPoint endPoint =
createEndPoint(socketChannel,selectionKey);
                            selectionKey.attach(endPoint);
                            endPoint.schedule();
                        }else{
                            selectionKey.cancel();
                            socketChannel.close();
                        }
                    }
                }else{
                    socketChannel =
(SocketChannel)selectionKey.channel();
                    SelectChannelEndPoint endPoint =
createEndPoint(socketChannel,selectionKey);
                    selectionKey.attach(endPoint);
                    if(selectionKey.isReadable()){
                        endPoint.schedule();
                    }
```

```
                    selectionKey = null;
                }
            }catch (Exception e){

                if(selectionKey != null && !(selectionKey.channel()
instanceof ServerSocketChannel) &&

                        selectionKey.isValid()){

                    selectionKey.cancel();

                }
            }
        }

        currentSelector.selectedKeys().clear();
        selectedThread = null;

    }catch (IOException e){

        //TODO doWork IOException

    }

}
```

图 4.5 为整个流程时序图。

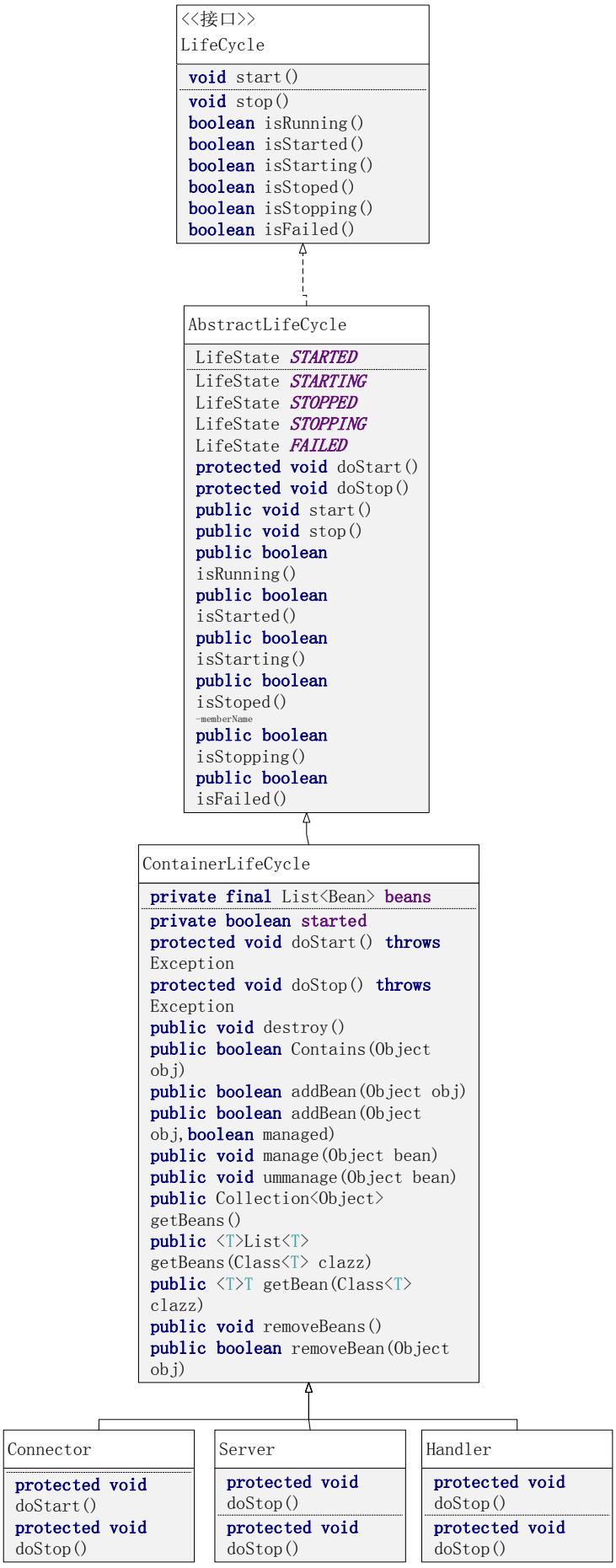

图 4.4 生命周期 UML 类图

4.3.4 管道架构与 Handler 链编码实现

Connector 的核心计算逻辑处理完成后的主要逻辑就是整个 Handler 链的处理了。Handler 链采用了管道模式。首先定义 Handler 接口，接口中核心函数为:handle(Request baseRequest,Response baseResponse)。然后 Server 实现 Handler 接口，重写 handle 方法，方法内调用 HandlerWrapper 的 handle 方法。AbstractHandler 抽象类实现 Handler 接口,处理 Handler 中 Server 对象的 getter 和 setter 方法，并定义抽象方法 handle(Request baseRequest,Response baseResponse)交给其子类去实现。HandlerWrapper 继承 AbstractHandler，实现其 handle()方法，并且 HandlerWrapper 类持有整个 Handler 链的一个链表集合，然后其 handle()方法遍历整个 Handler 链,并执行每个 Handler 的 handle()方法。其 UML 类图如图 4.6 所示。

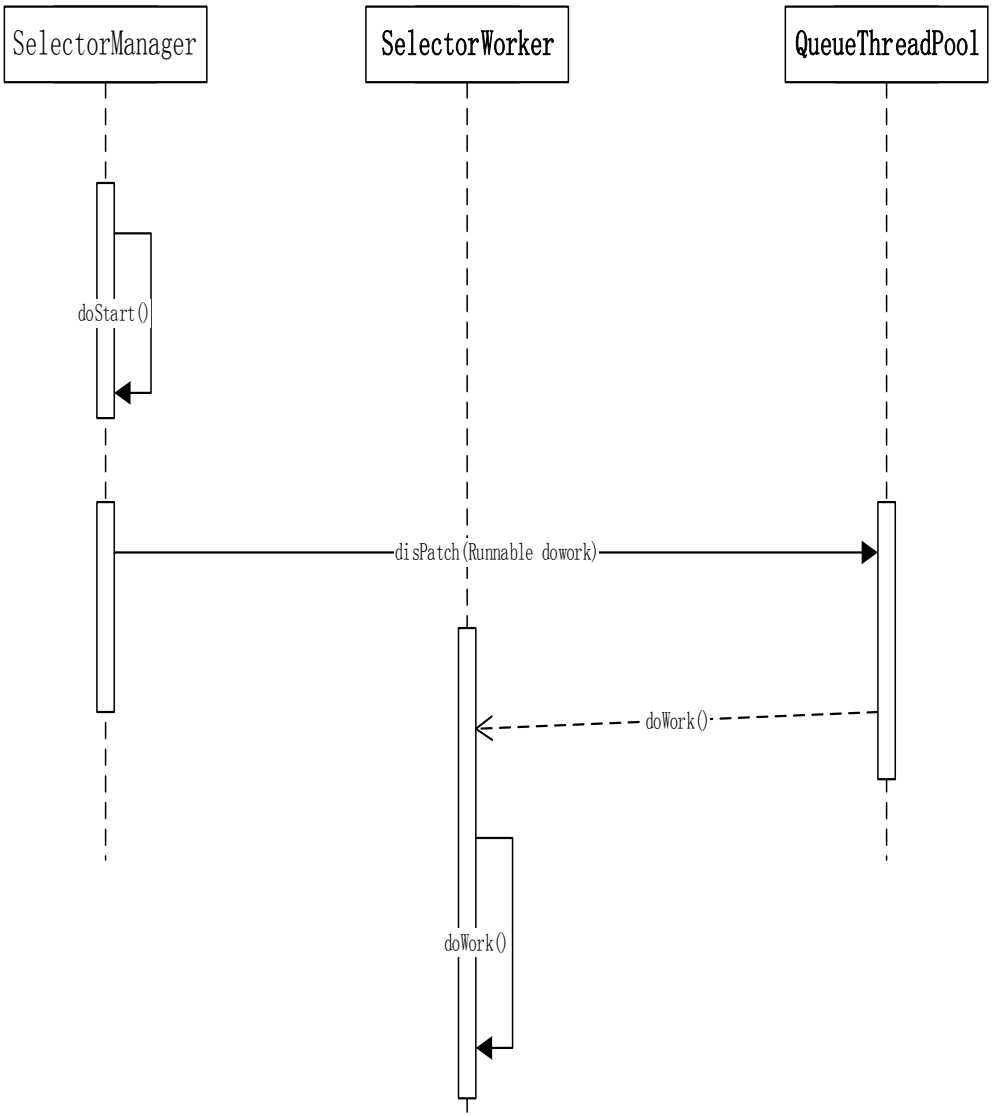

图 4.5 Reactor 时序图

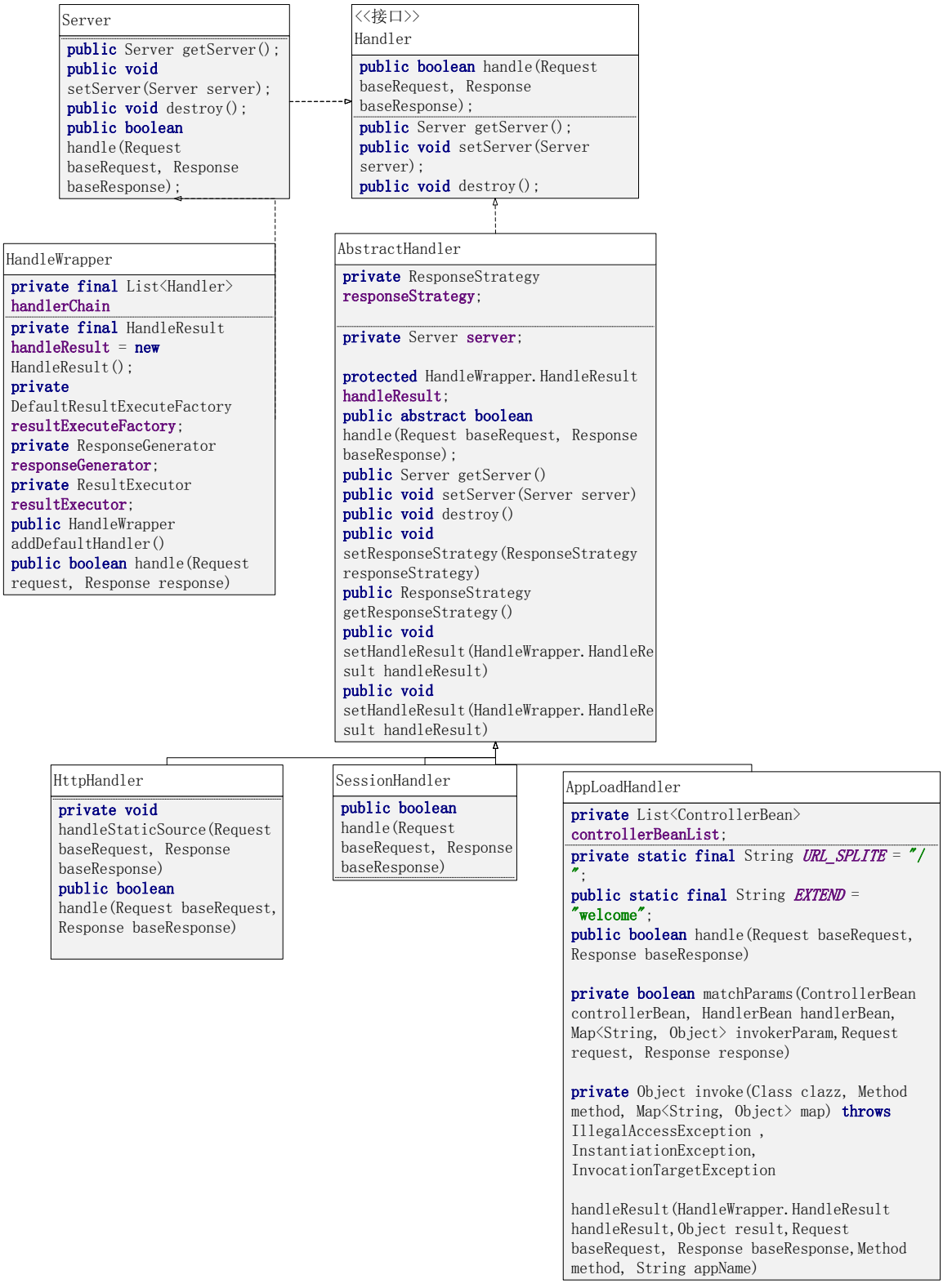

图 4.6 Handler 管道 UML 类图

编码实现如下：

```
@Override
public boolean handle(Request request, Response response) {
    for(Handler handler : handlerChain){
        try {
```

```
                if(!handler.handle(request,response)){
                    Logger.INFO("execute-[{}]
back:",handler.getClass().getName());
                    break;
                }
            }catch (ClientException e){
                handleResult.setResultType(RESULT_TYPE_CLIENT_ERROR);
                handleResult.setExtend(e.getMessage());
                break;
            }catch (ServerException e){
                handleResult.setResultType(RESULT_TYPE_SERVER_ERROR);
                handleResult.setExtend(e.getMessage());
                break;
            }
        }
        Logger.INFO("handle chain over
resultHandle:{}",handleResult.toString());
        resultExecuteFactory = new DefaultResultExecuteFactory(response);
        resultExecutor =
resultExecuteFactory.createResultExecute(handleResult);
        try {
            handleResult.setBody(resultExecutor.execute());
        } catch (VariableIllegalException e) {
            handleResult.setBody(StringUtil.format("handle dynamic page
error:{}",
                    Logger.printStackTraceToString(e)).getBytes());

response.setHttpHeader(STATUS,HttpConstant.HttpStatusCode.Internal_Serv
er_Error.getDesc());
            handleResult.setResultStatus(RESULT_TYPE_SERVER_ERROR);
        } catch (IOException e) {
            handleResult.setBody(StringUtil.format("handle static source
error:{}",
                    Logger.printStackTraceToString(e)).getBytes());
            handleResult.setResultStatus(RESULT_TYPE_CLIENT_ERROR);

response.setHttpHeader(STATUS,HttpConstant.HttpStatusCode.Not_Found.get
```

```
Desc());
        }
        responseGenerator = new
ResponseGenerator(handleResult,request,response);
        responseGenerator.generate();
        response.generateHeaderBytes();
        return true;
}
```

4.3.5 工厂模式与结果处理器创建

服务器规定了 CGI 程序的五种返回类型，以及 handle 链中出现异常时的客户端异常或者服务的异常等返回结果的处理。基于以上可以很自然的想到创建型设计模式-工厂模式来创建对应的结果处理器，来达到可扩展和低耦合的效果。

URL 类图如图 4.7 所示。

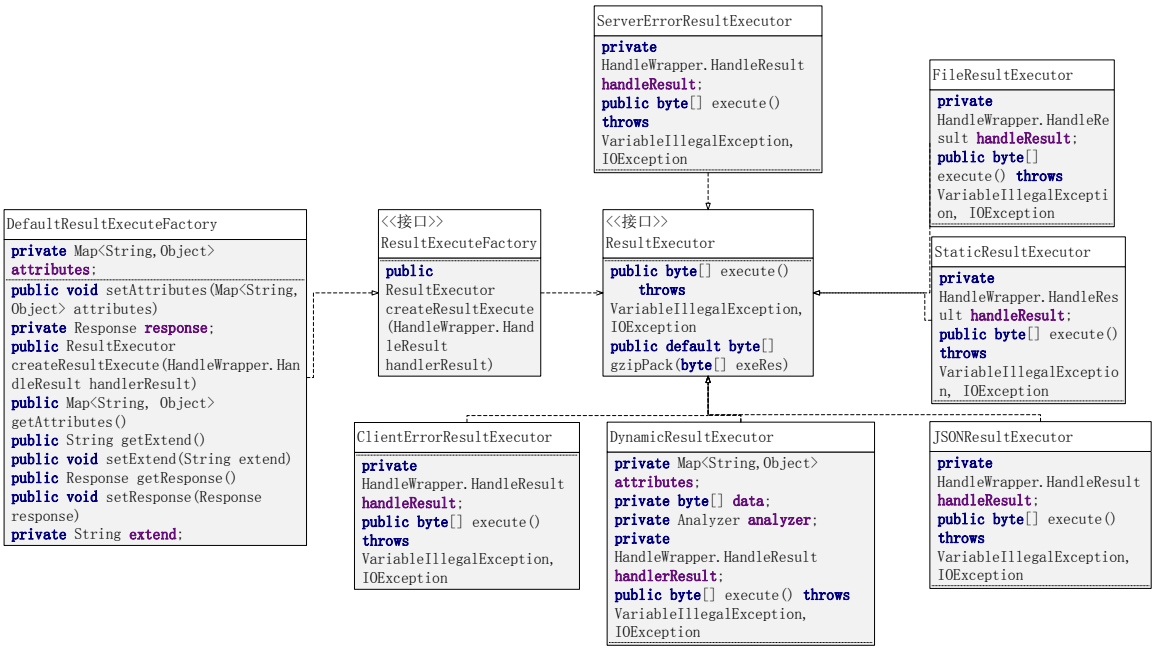

图 4.7 结果处理器创建工厂 UML 类图

对于结果处理器的创建核心代码如下：

```
public ResultExecutor createResultExecute(HandleWrapper.HandleResult
handlerResult) {
    if(RESULT_TYPE_STATIC.equals(handlerResult.getResultType())){
        return new StaticResultExecutor(handlerResult);
    }else if(RESULT_TYPE_DYNAMIC.equals(handlerResult.getResultType())){
        return new DynamicResultExecutor(handlerResult,attributes);
    }else if(RESULT_TYPE_JSON.equals(handlerResult.getResultType())){
        return new JSONResultExecutor(handlerResult);
```

```
    }else
if(RESULT_TYPE_REDIRECT.equals(handlerResult.getResultType())){
        return new RedirectResultExecutor(handlerResult);
    }else
if(RESULT_TYPE_CLIENT_ERROR.equals(handlerResult.getResultType())){
        return new ClientErrorResultExecutor(handlerResult);
    }else
if(RESULT_TYPE_SERVER_ERROR.equals(handlerResult.getResultType())){
        return new ServerErrorResultExecutor(handlerResult);
    }else
if(RESULT_TYPE_FILE.equalsIgnoreCase(handlerResult.getResultType())){
        return new StaticResultExecutor(handlerResult);
    }

handlerResult.setResultStatus(HttpConstant.HttpStatusCode.Internal_Serv
er_Error.getDesc());
    return new ResultExecutor() {
        @Override
        public byte[] execute() {
            return "result type not found".getBytes();
        }
    };
}
```

4.3.6 策略模式与 HTTP 响应头生成

根据 RFC2616 规定，HTTP 响应状态码分为四类，分别为 20X,30X,40X,50X，其具体功能已在第二章协议分析阶段介绍过了，然而需要根据不同的状态生成不同的 HTTP 响应头信息。这里介绍的是如何通过策略模式，来对服务器返回的状态码，来执行相应的算法策略，生成相应的 HTTP 响应头信息。

其 UML 类图 4.8 所示。

ResponseGenerator 类持有 ResponseStrategy 的引用，其核心方法就是根据不通的状态码生成响应的策略算法，进行 HTTP 响应头的生成逻辑运算。代码如下：

```
public void generate(){
    if(HttpHelper.is20X(statusCode)){
        responseStrategy = new
OKResponseStrategy(handleResult,request,response);
    }
```

```
    if(HttpHelper.is30X(statusCode)){
        responseStrategy = new
RedirectReponseStrategy(request,response,(String) extend);
    }
    if(HttpHelper.is40X(statusCode)){
        responseStrategy = new
ClientErrorResponseStrategy(request,response,HttpHelper.handleError((St
ring)extend));
    }
    if(HttpHelper.is50X(statusCode)){
        responseStrategy = new
ServerErrorResponseStrategy(request,response,HttpHelper.handleError((St
ring)extend));
    }
    responseStrategy.doGenerate();
}
```

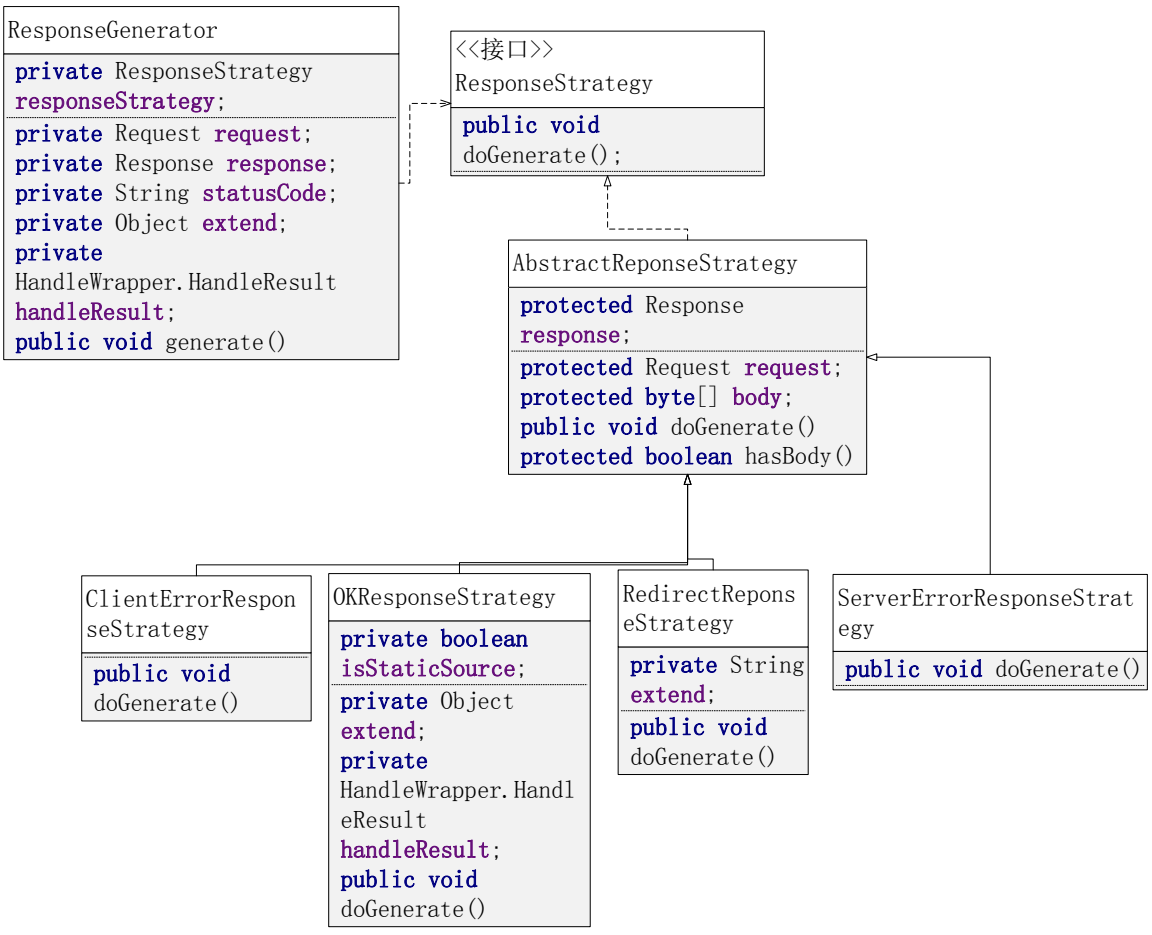

图 4.8 响应头生成策略模式 UML 类图

4.4 启动服务器

服务器应用程序打包为一个.zip 或者.tar 文件，方便 Windows 平台和类 Unix 平台解压,这也是为什么大多数软件的压缩格式都是这两种，因为类 Unix 平台对于 rar 格式并不原生支持。解压完之后的文件结构如下：

- /bin:里面有 run.bat 和 run.sh 可执行脚本，分别对应 windows 平台的批处理脚本和 linux 平台的 shell 脚本，用来启动服务器。
- /config：配置文件，包含了一些基本配置，如项目路径，app 资源文件路径，服务器名称等等。
- /lib:应用程序打包后的 jar 包全在这个目录下。
- /logs:日志文件。
- /temp:临时文件。
- /www:CGI 应用程序所在的目录。
- Readme.md:应用程序说明解释。

而只需点击 bin 目录下的 run 脚本即可运行服务器，当看到 Dam started 字符即说明服务器启动成功，如图 4.9 所示。

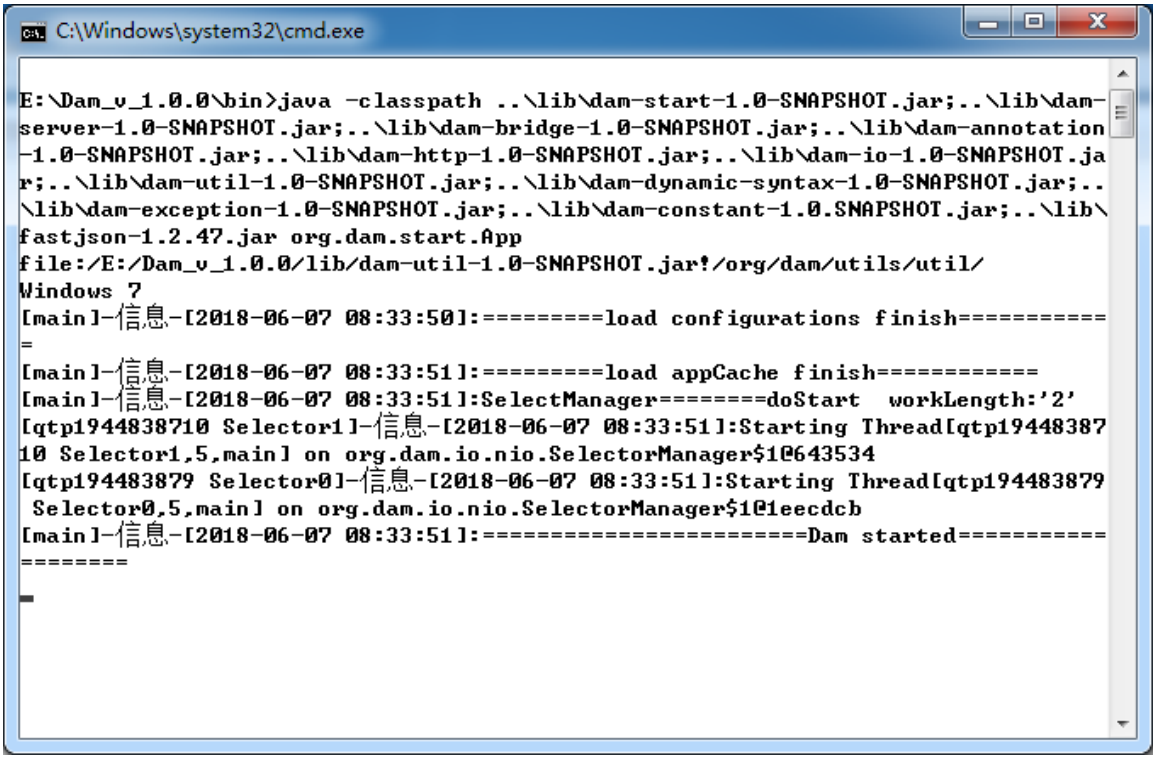

图 4.9 服务器启动示意图

或者从 github 下载源码，项目采用 maven 进行构建，所以你需要将其导入为 maven 项目，然后运行 start 模块中的 App 类中的 main 函数即可启动服务器。

4.5 在服务器上进行 CGI 程序开发

服务器所做的开发工作完成之后，迫不及待就想要开发一款 web 应用，让其运行在服务器之上，这也是对服务器最好的验证。这节将介绍如何基于 Dam 服务器，开发 web 应用程序。

4.5.1 注解说明

服务器采用 0 配置的方式来开发 web 应用程序，这不像 Tomcat 一样，需要进行繁琐的配置，添加各种映射关系。所依靠的就是通过注解的方式来标志所需要的一些信息，一下就是开发 web 应用程序所需要了解的所有注解及其对应的含义。

- @Toylet——类注解，声明该类为一个控制器，用与处理 HTTP 请求。
- @Request——方法注解，声明该方法对对应的具体的每一个 HTTP 请求，其中参数 url 表示该方法处理的具体的 HTTP 请求 url，参数 method 为 HTTP 请求 method，即:GET,POST,.HEAD,PUT,OPTIONS 等。参数 resultType 返回类型，应严格遵循服务器中定义的五种返回类型
  1. page——返回类型为一个静态 html 页面。
  2. dynamic——返回类型为一个动态.tsp 脚本
  3. json——返回类型为 json 字符串
  4. redirect——返回类型为重定向
  5. file——返回一个文件
- @Parameter——给注解为一个方法内参数注解,与 HTTP 请求参数一一对应，当 HTTP 请求参数为空时，将采用 defaultValue 为默认值。

以上就是开发 web 应用程序所需要了解的所有注解，了解这些注解的用法之后即可在服务器之上开发属于自己的 web 应用程序了。

4.5.2 动态脚本解析

熟悉的动态脚本例如 php，jsp，asp 等都是有服务器后台数据动态解析生成返回给浏览器的，php 是经过 C 语言编写的解释器进行运行期解释执行的，所以在效率上来说是不够高的，jsp 是在编译器变异成 servlet 类字节码运行的。所以一款服务器对于动态脚本的支持是必不可少的，服务器只对简单的单变量脚本进行解析，并不支持复杂语言结果，因为那相当于重写开发一款编程语言，对编译原理等知识需要有非常深的理解，项目的重点也不在于此，能力上也有限。这里只对 Dam 服务器所支持的脚本语法做个简单介绍。

首先在脚本内定义一个变量语法为:{$}

- {$var}:基本数据类型，如 int，string，boolean 等
- {$var.get(0)}：List 类型
- {$var.getUsername()}：对象

4.5.3 开发 CGI 程序

本小节将用上一节中所介绍的注解，实战性开发一款 web 应用程序。该 demo 既可以讲解如何基于 Dam 进行 web 应用程序开发，也可以用于第五章黑盒测试该服务器是否能够正常运行。

一、 首先需要新建一个项目，这里同样采用 maven 进行构建，当然你也可以手动导入响应 jar 包。新建之后的项目目录结构如图 4.10 所示。

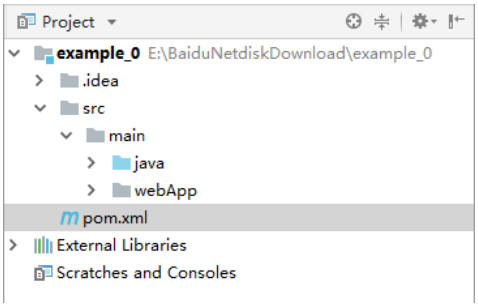

图 4.10 webapp 结果示意图

src/main/java 目录下为 java 源文件。webApp 下为 web 资源文件，例如 html,css,js,tsp 动态脚本,图片等,这里当然也能通过配置改变资源文件路径。

二、 新建一个 Java 类，这里新建了一个名为 Index 的 Java 类。然后需要在给类上声明注解@Toylet。然后新建一个方法，也为 index()即可，返回值为 String 类型。然后在该方法上声明 @Request(url=”/index” ,method=”GET”,resultType=”page”) 注解，该方法返回值为”/html/index.html”。如图 4.10 所示。

```
public class Index {

    //private UserService userService = new UserService();

    @Request(url = "/index",method = "GET",resultType = "page")
    public String index() { return "html/index.html"; }
```

图 4.10 CGI 程序控制器类

编译该项目,这里采用 mvn compile 进行编译,编译后的文件目录结构如图 4.11 所示。

图 4.11 编译后包结果示意图

classes 文件即编译过后的 Java 字节码文件，webApp 文件为资源文

三、 在服务器 www 目录下新建一个文件夹，该文件夹为 web 应用程序的名称，然后将编译过后的字节码文件个资源文件复制到新建的文件夹下,然后确定 run 脚本即可。

第五章 服务器测试

本节将使用一个按照 Dam 服务器规范的 web 应用程序 demo，实际演示服务器是否能够正常运行。

5.1 设计 CGI 程序 Demo

服务器最重要的还动态执行 CGI 程序的能力，所以需要开发一个 Demo，通过 Demo 来验证服务器所具备的功能。Demo 基本上包含了对服务器所有功能的测试，包括但不限于：对 html, css, js, 图片等静态文件的测试；通过 List，String，对象等类型的变量测试. tsp 动态脚本，其中包含了对 cookie，session 的输出测试；对重定向的测试；对客户端缓存的测试；对直接序列化 json 输出的测试，对文件下载的测试。

类文件如图 5.1 所示。

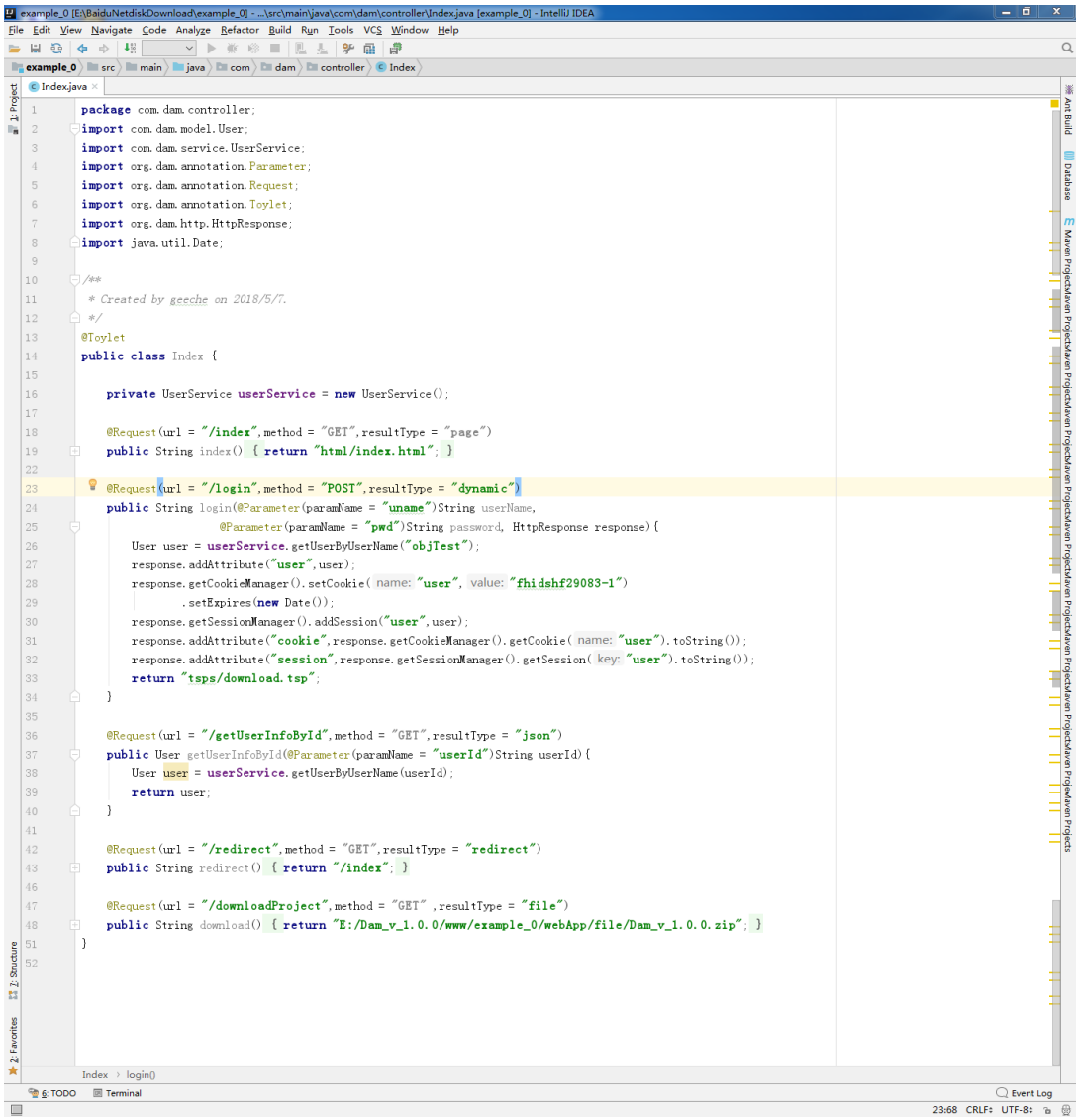

图 5.1 demo 控制器类

5.2 功能测试

通过上一节编写的 CGI 程序，来验证服务器所具备的功能，浏览器需要以这样的 URL 格式来进行 CGI 程序访问：localhost/CGI 应用程序名/Request 注解中的 url 项。

5.2.1 静态文件与缓存

浏览器输入 localhost/example_0/index，该用例包含了对 html，css，js，图片等静态资源的测试。验证结果截图如图 5.2 所示。

图 5.2 demo 主页

浏览器第一次请求这个页面的请求头和响应头如图 5.2 所示。

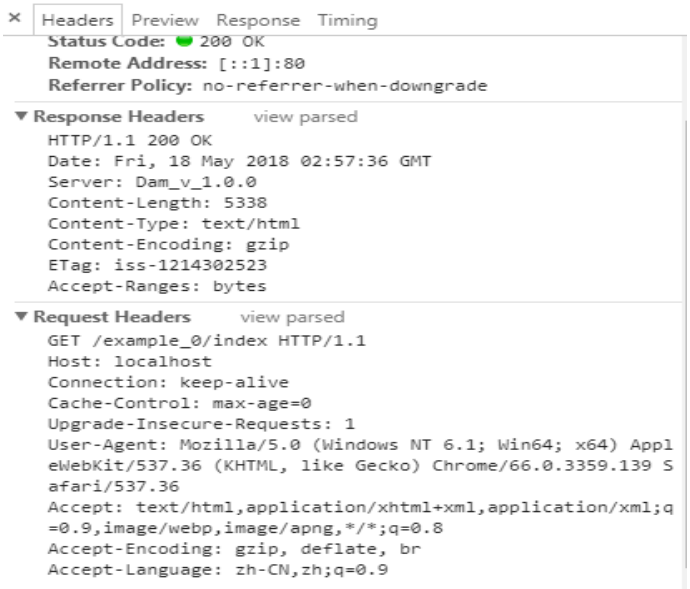

图 5.3 第一次请求时的头信息

注意在响应头加入了 Etag 字段，该字段即对浏览器段做缓存用的。第二次请求

相同页面的请求头响应头信息如图 5.4 所示。

图 5.4 第二次请求时的头信息

请求头 If-None-Match 字段的值只要与响应头 Etag 字段的值相同，并且浏览器端开起缓存，服务器将不返回数据，浏览器将读取本地缓存。

5.2.2 post 表单提交，动态脚本及 cookie，session

第二个函数 login 是一个 post 请求，所以包含了对 post 表单的处理，cookie 和 session 的处理。从 index 页面表单登录执行 login 函数，结果如图 5.5 所示。

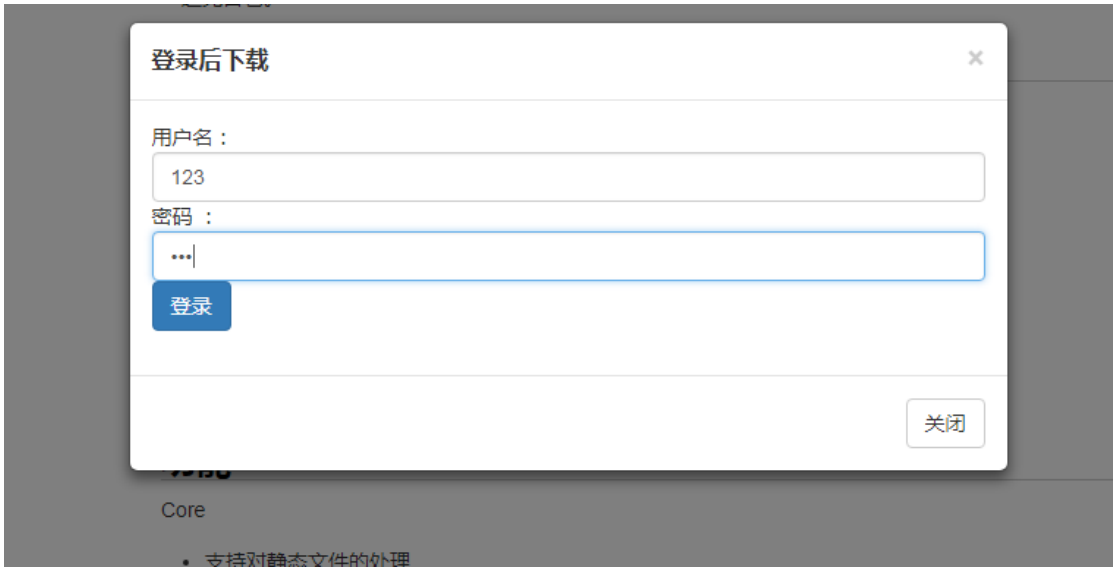

图 5.5 登录

点击登录后跳转页面如图 5.6 所示。

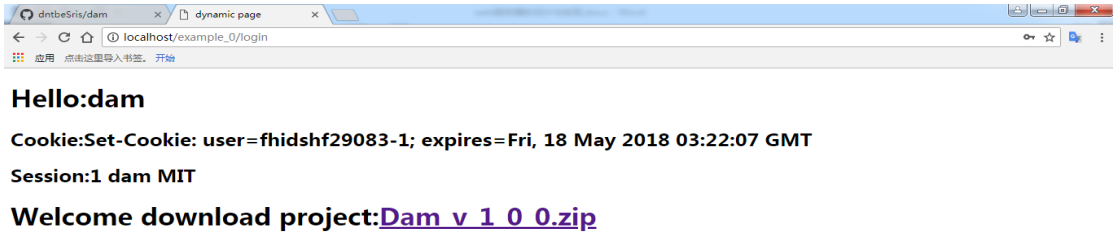

图 5.6 登录成功及下载页

这个处理过程包含了对 post 表单的处理，对动态脚本的解析。首先看看动态脚本代码如图 5.7 所示。

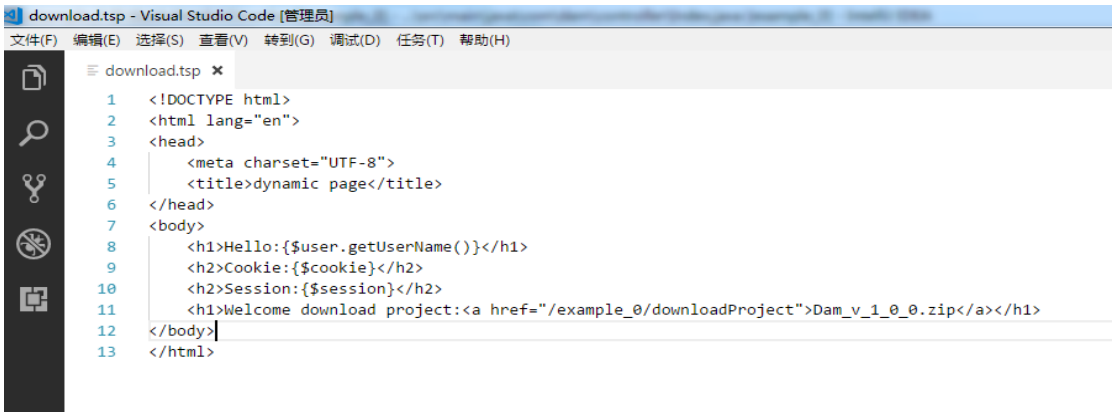

图 5.7 .tsp 脚本

结果如预期的一样动态脚本中的${}变量均显示正常。再看看对 cookie 的支持。页面显示的 cookie 与浏览器响应头显示的一样。页面打印的 session 与服务端的也一样。

以上均通过测试。

5.2.3 重定向支持

第四个函数为 redirect(),返回值为重定向类型。浏览器输入对应 url：localhost/example_0/redirect，结果显示跳转到 index 页面，与预期结果一致。

5.2.4 输出序列化 json 串

第三个函数 getUserInfoById(),返回值为 json 类型，对于 ajax 请求或者说前后端分离来说通常都是返回 json 类型。

在浏览器输入 localhost/example_0/getUserById?userId=1,结果显示如图 5.8 所示：

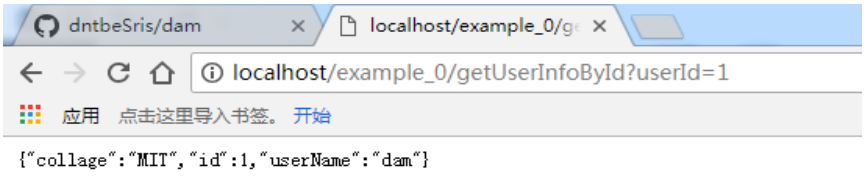

图 5.8 json 结果返回页

和预期一样返回的为一个 json 字符串。

5.2.5 文件下载

第五个函数 download()返回类型为 file 类型，该功能即文件下载功能。点击 login 跳转页面后的 Dam_v_1_0_0.zip 即可下载文件。结果显示如图 5.9 所示：

图 5.9 下载成功页

左下角文件下载成功，结果与预期一致。

第六章 总结

6.1 总结

整个服务器开发设计下来，所涉及到的知识基本上涵盖了本科时期所有的专业基础知识。从计算机网络中的协议基础，到操作系统中的 IO，多线程知识，程序设计中大量数据结构的使用，以及动态脚本处理所涉及到的编译原理的知识，以及软件工程当中利用各种设计模式降低程序之间和模块之间的耦合度等等。从最初的资料收集，参照 RFC2616 去研究网络协议细节，阅读 Linux 高级网络编程去了解操作系统的 I/O 模型，阅读商业服务器 Jetty，Tomcat 等源码去了解服务器编码与架构设计，到自己动手编码实现，最终也完成了一款 Web 服务器雏形，实现了基本功能。整个过程下来不仅在编码能力上有长足的进度，也提升了自己在在程序设计架构上的能力，同样也巩固了计算机专业基础知识。

致谢

时光荏苒，大学生涯即将结束。经过半年时间的磨砺，最终完成了毕业设计和毕业论文。半年制作毕业设计和毕业论文的过程，是一个反复遇到困难，解决困难的过程，期间得到了许多关怀与帮助，此刻要向他们表达我最真挚的感谢。

首先要感谢我的导师何福保老师，这几个月来的悉心指导，告诉我论文的撰写规范，应该注意的细节，甚至字字句句把关，提出很多中肯的指导意见。其次感谢大学四年来的任课老师，让我有丰富的专业知识完成整个毕业设计。

同时也要感谢大学四年与我互勉互励的同窗们，感谢这个信息大爆炸时代，让我能够对信息，知识唾手可得，能够检索问题并站在前人的肩膀上解决问题。

最后感谢参与毕设答辩的各位评审老师和所有帮助过我的人，你们的鼓励是我前进的动力。我会继续砥砺前行，继续努力，奋斗。

参考文献

[1] W.RICHARD STEVENS 著 范建华 等译. TCP/IP 详解,卷 1：协议[M].北京:机械工业出版社,2004.

[2] James F.Kurose,Keuth W.Ross 著 陈鸣 译. 计算机网络自顶向下方法.第四版[M].北京:机械工业出版社,2009.

[3] W.RICHARD STEVENS,Bill FENNER,ANDREW M.RUDOFF 著 杨继张译. Unix 网络编程[M].北京:清华大学出版社,2006.

[4] 上野宣 著 于均良译. 图解 HTTP[M].北京:人名邮电出版社,2014.

[5] David Gourley Brian Totty 著 陈涓 赵振平 译. HTTP 权威指南[M].北京:人民邮电出版社,2012.

[6] 严体华 著.网络管理员.第三版[M].北京:清华大学出版社,2011.

[7] Brain Goetz,Tim Peierls,Joshua Bloch,Joseph Bowbeer,David Homles,Doug Lea 著 童云兰 等译. Java 并发编程实战[M].北京:机械工业出版社,2012.

[8] Norman Maurer,Marvin Allen Wolfthal 著. Netty in Action[M].Greenwitch:Manning Publications,2015.

[9] Eric Freeman 著 O'Reilly Taiwan 公司 译. Head First 设计模式[M].北京:中国电力出版社,2007.

[10]Alfred V.Aho,Monica S.Lam,Ravi Sethi,Jeffrey D.Ullman 著 赵建华 郑滔 戴新宇 译.编译原理[M].北京:机械工业出版社,2008.

[11] Elliotte Rusty Harold 著 朱涛江 林剑 译. Java 网络编程.第三版.北京:中国电力出版社,2005-11.

[12] Joshua Bloch 著 俞黎敏 译. Effective Java[M].第二版.北京:机械工业出版社,2009.

[13] Krik Knoernschild 著 张卫滨 译. Java 应用架构设计[M].北京：机械工业出版社,2013.

[14] 杨小娇. 轻量级高并发 Web 服务器的研究与实现[D].南京邮电大学,2014.

[15] 封相远. 基于 Linux 操作系统的 Web 服务器的设计与实现[D].天津大学,2007.

[16] 梁苏彬. 基于集群的高可用 HTTP 服务器[D].四川大学,2004.