摘 要

WWW(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。如今，我们使用的大部分网络应用，如web应用，app应用，甚至游戏等，其后台都会选择web服务器为数据传输做支撑。更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，高大规模计算等挑战。所以，掌握网络协议基础知识，以及对HTTP协议的协议规范以及通信过程，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。

本文通过对TCP/IP到HTTP协议以及Linux I/O模型的研究，对商用web服务器tomcat的分析，采用Java语言设计制作小型web服务器。

**关键词：网络编程 TCP/IP协议 HTTP协议 Java 服务器 架构设计**

Abstract

The development of the WWW (World Wide Web) has been more than 25 years ago. Its widespread use has an irreplaceable position in the online world. Today, most of the web applications we use, such as web applications, app applications, and even games, will use the web server to support data transmission. With the outbreak of the mobile Internet, users are increasing day by day, and web servers are also facing challenges such as higher concurrency and high-scale computing. Therefore, it is extremely important to deeply study the operating mechanism of the web server for the study of network protocols and the communication principles of the HTTP protocol.

This paper analyzes the TCP/IP to HTTP protocol and the Linux I/O model, analyzes the commercial web server tomcat, and uses the Java language to design a tiny web server.

**Keywords: Network Programming TCP/IP Protocal HTTP Protocal Java Architecture Design**

# 目 录

摘要............................................................................ 1

Abstract........................................................................ 2

1. 绪论..................................................................... 3
   1. 研究的背景与意义.......................................................... 4
   2. 课题研究过程.............................................................. 5
   3. 论文组织结构.............................................................. 6
2. 网络协议介绍............................................................. 7

2.1 OSI网络模型............................................................... 8

2.1.1 OSI七层模型划分....................................................... 9

2.1.2 各层功能定义......................................................... 10

2.2 TCP/IP协议............................................................... 11

2.2.1 TCP协议定义.......................................................... 12

2.2.2 TCP报文格式.......................................................... 13

2.2.3 TCP协议通信流程...................................................... 14

2.2.4 TCP协议通信过程中的问题.............................................. 15

2.2.4.1 Nagle算法........................................................ 16

2.2.4.2 超时重传......................................................... 17

2.2.4.3 拥塞控制......................................................... 18

2.3 HTTP协议................................................................. 19

2.3.1 HTTP协议定义......................................................... 20

2.3.2 HTTP报文格式......................................................... 21

2.3.3 HTTP通信流程......................................................... 22

2.3.4 客户端缓存与状态记录................................................. 23

第三章 I/O模型与多线程编程..................................................... 24

3.1 UNIX I/O模型分析......................................................... 25

3.1.1 同步,异步,阻塞,非阻塞................................................ 26

3.1.2 UNIX下的五种I/O模型................................................. 27

3.2 Tomcat I/O模型与线程模型分析............................................. 28

第四章 服务器设计与实现........................................................ 29

4.1 Java与JVM平台概述....................................................... 30

4.2 服务器架构............................................................... 31

4.2.1 I/O模型与线程模型.................................................... 32

4.2.2 整体架构与执行流程................................................... 33

4.2.3 模块说明............................................................. 34

4.3 服务器编码与实现......................................................... 35

4.3.1 生命周期管理......................................................... 36

4.3.2 NIO编码实现.......................................................... 37

4.3.3 多Reactor模型及编码实现............................................. 38

4.3.4 管道模式与Handler链编码实现......................................... 39

4.3.5 工厂模式与结果处理器创建............................................. 40

4.3.6 策略模式与HTTP响应头的生成.......................................... 41

4.4 运行服务器............................................................... 42

4.5 在服务器上进行web应用程序开发........................................... 43

4.5.1 注解说明............................................................. 44

4.5.2 动态脚本解析......................................................... 45

4.5.3 开发webapp........................................................... 46

第五章 服务器测试.............................................................. 47

5.1 黑盒测试................................................................. 48

第六章 总结.................................................................... 49

6.1 总结..................................................................... 50

致谢........................................................................... 51

参考文献....................................................................... 52

# 第一章 绪论

## 1.1 研究背景与意义

从1980年[Tim](https://zh.wikipedia.org/wiki/%E8%92%82%E5%A7%86%C2%B7%E4%BC%AF%E7%BA%B3%E6%96%AF-%E6%9D%8E) Berners-Lee构建ENQUIRE项目至今，万维网已经走过了28年。从互联网时代的开启到移动互联大潮兴起，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HTTP服务器作为支撑。在互联网浪潮下的今天，HTTP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。HTTP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。所以我们需要对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http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：自底向上对网络协议进行基础介绍
* 第三章：介绍操作系统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：对服务器进行功能上的测试
* 第六章：总结

# 第二章 网络协议介绍

## 2.1 OSI网络模型

开放式系统互连通信参考模型(Open System Interconnection Reference Model,缩写为OSI)，简称OSI模型(OSI Model)，一种概念模型，由国际化标准组织(ISO)提出，一种试图使各种计算机在世界范围内互联为网络的标准框架。[4]

### 2.1.1 OSI七层模型划分

1983年，国际标准化组织(IOS,International Organization for Standardization)发布了著名的**ISO/IEC 7489**标准，它定义了网络互连的7层框架，也就是开放式系统互连参考模型。如下表：

**表 2.1 OSI七层模型划分**

|  |  |  |
| --- | --- | --- |
| **层** | **数据单元** | **功能** |
| 应用层 | Data | web服务与消费者应用程序间的一个接口。[4] |
| 表示层 |  | 数据表示、数据安全、数据压缩。[4] |
| 会话层 |  | 会话层到传输层的映射；会话连接流量控制；数据传输；会话连接恢复和释放；会话连接管理、错误控制。[4] |
| 传输层 | Segment | 用端口号来对上层应用程序进行寻址。[4] |
| 网络层 | Packet | 根据不同网络系统之间路径选择的网络层地址（IP地址）。[4] |
| 数据链路层 | Frame | 在物理层建立，撤消，识别逻辑链路，链路复用和错误检查功能。 通过使用接收系统的硬件或物理地址进行寻址。[4] |
| 物理层 | 比特流(Bit) | 建立、维护和取消物理连接。[4] |

### 2.1.2 各层功能定义

这里只对OSI各层进行功能上的大致阐述，不深究细则，因为每一层实际都是一个复杂的层，都可以作为一个课题单独研究，本文重点不在于此。我们所关心，并且和服务器密切相关的也只在传输层和应用层。

* 应用层：网络应用层是通信用户之间的一个窗口，为用户提供诸如网络管理，文件传输和事务处理等服务。它包含许多独立的，用户共同的服务协议模块。网络应用层是OSI的最高层，为网络用户之间的通信提供专用程序。应用层的内容主要取决于用户的个人需求。这一层设计的主要问题是分布式数据库，分布式计算技术，网络操作系统和分布式操作系统，远程文件传输，电子邮件，终端电话以及远程登录和控制。等待。在2011年之前，国际上没有完整的应用层标准。这是一个非常广泛的研究领域。在OSI的七个级别中，应用层是最复杂的并且包含最多的应用层协议。有些仍在研究和开发之中。。[4]
* 表示层：表示层向上对应用层提供服务，向下接收来自会话层的服务。表示层是一种为应用程序进程间传输的信息提供表示方法的服务。 它只关心正在发送的信息的语法和语义。 表示层需要执行某些特定的功能，主要包括转换不同的数据编码格式，提供数据压缩和解压缩服务以及加密和解密数据。 例如，表示层的协议支持图像格式的显示。[4]
* 会话层：该层也可以被称为会议层或对话层。 在会话层及以上的高层，数据传输单元不再以其他方式命名，并统称为分组。 会话层不参与特定的传输，它包含接入认证。 建立和维护应用程序之间通信的机制，包括会话管理。 如果服务器验证用户登录由会话层完成。[4]
* 传输层：传输层建立在网络层和会话层之间，实质上它是网络体系结构中高低层之间衔接的一个接口层，用一个寻址机制来标识一个特定的应用程序（端口号），传输层不仅是一个单独的结构层，它还是整个分层体系协议的核心，没有传输层整个分层协议就没有意义。[4]
* 网络层：传输层在网络层和会话层之间建立。 它本质上是一个接口层，它连接网络体系结构的高层和低层。 寻址机制用于识别特定的应用程序（端口号）。 传输层不仅是一个独立的结构层，它也是整个分层系统协议的核心，没有传输层的整个层协议就没有意义。[4]
* 数据链路层：在物理层提供的比特流服务的基础上，将比特信息封装成数据帧，该数据帧用于在物理层建立，取消，识别逻辑链路，链路复用和错误检查，并且通过使用接收 系统。 通过硬件地址或物理地址进行寻址，在相邻节点之间建立数据链路，为信道上的帧传输提供无差错控制，并为其上的网络层提供高效的服务。[4]
* 物理层：物理层是OSI分层体系结构中最重要和最基础的层。 它建立在传输媒体的基础上，建立，维护和取消物理连接，实现设备之间的物理接口，以及接收和发送物理层。 一系列比特流，不管信息的含义和信息结构如何。[4]

## 2.2 TCP/IP协议

互联网协议族（Internet Protocol Suite，IPS）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。 它通常被称为TCP/IP协议族。因为这个协议家族的两个核心协议：TCP（传输控制协议）和IP（互联网协议）是这个家族最早通过的标准。 由于分层结构通常用于网络通信协议，因此多层协议协同工作时，它类似于计算机科学中的堆栈，因此也称为TCP / IP协议栈。 这些协议最初起源于美国国防部的ARPAnet项目（缩写为DoD），因此也被称为DoD模型。 该协议系列由Internet工程任务组维护。

### 2.2.1 TCP协议定义

传输控制协议（Transmission Control Protocol,TCP）是一种面向连接的、可靠的、基于字节流的传输层通信协议，由IETF的RFC 793定义。在简化的计算机网络OSI模型中，它完成第四层传输层所指定的功能，用户数据包协议（UDP）是同一层内另一个重要的传输协议。

在因特网协议族（Internet protocol suite）中，TCP层是位于IP层之上，应用层之下的中间层。不同主机的应用层之间经常需要可靠的、像管道一样的连接，但是IP层不提供这样的流机制，而是提供不可靠的包交换。

应用层向TCP层发送用于网络传输的、用8位字节表示的数据流，然后TCP把数据流分区成适当长度的报文段（通常受该计算机连接的网络的数据链路层的最大传输单元（MTU）的限制）。之后TCP把结果包传给IP层，由它来通过网络将包传送给接收端实体的TCP层。TCP为了保证不发生丢包，就给每个包一个序号，同时序号也保证了传送到接收端实体的包的按序接收。然后接收端实体对已成功收到的包发回一个相应的确认（ACK）；如果发送端实体在合理的往返时延（RTT）内未收到确认，那么对应的数据包就被假设为已丢失将会被进行重传。TCP用一个校验和函数来检验数据是否有错误；在发送和接收时都要计算校验和。

### 2.2.2 TCP报文格式

下表为TCP数据包结构：

**表2.2.2 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 |
| **32** | SYN序列号 | | | |
| **64** | ACK确认号 | | | |
| **96** | 报头长度 | 保留 | 标识符 | 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（16位长）－辨识发送连接端口
* 目的端口号（16位长）－辨识接收连接端口
* 序列号（seq，32位长）
  + 如果有SYN(初始序列号)，第一个数据比特的序列码为本序列号+1。
  + 如果没有SYN，则此为第一个数据比特的序列码。
* 确认号（ACK，32位长），期望收到数据的初始序列号+1。
* 报头长度（4位长），以4字节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—为1表示高优先级数据包，紧急指针字段有效。
  + ACK—为1表示确认号字段有效
  + PSH—为1表示数据带有PSH标志，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—为1表示出现差错，需要重置TCP连接。还可用于拒绝非法报文段或拒绝连接请求。
  + SYN—为1表示这是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—为1表示发送方没有数据要传输了，要求释放连接。
* 窗口（WIN，16位长），用于流量控制。表示从确认号开始，接受方可接收的报文字节数，即滑动窗口大小。
* 校验和（Checksum，16位长），强制性的字段。对整个的TCP报文段，以16位字进行计算所得，以验证报文是否出错。
* 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。每个选项的开始是1字节的kind字段，说明选项的类型。
  + 0：选项表结束（1字节）
  + 1：无操作（1字节）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TCP往另一端的最大数据块长度。当一个连接建立时，连接的双方都要通告各自的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：窗口扩大因子（4字节，wscale），取值0-14。用来把TCP的窗口的值左移的位数，以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）

### 2.2.3 TCP协议通信流程

TCP是一个面向连接的协议。一方向另一方发送数据之前，都必须先在双方之间建立一条连接，所以需要了解TCP是如何建立连接以及通信结束后是如何终止连接的。

TCP的连接过程也称为三次握手：

1) 请求端（客户端）发送一个SYN段指明打算连接的服务端的端口号，以及初始序列号(ISN)。这个SYN段为报文段1。

2) 服务端发回包含服务端的初始序列号的SYN报文段，作为应答。同时，將确认序号设置为客户端的ISN加1以对客户端的SYN报文段进行确认。一个SYN將占用一个序号。

3) 客戶端必須將确认序号设置为服务端的ISN加1以对服务端的SYN报文段进行确认。

这三个过程即可完成连接的建立。

TCP终止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) 被动关闭一方收到这个FIN，将发回一个ACK，确认序号为收到的序号加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，确认序号为收到的序号加1

以上为TCP握手和挥手的执行流程。下图描述了整个TCP生命周期的状态迁移。

**图2.2 TCP状态迁移示意图**

### 2.2.4 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。

#### 2.2.4.1 Nagle算法

Nagle算法要求一个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。该算法的优越之处在于它是自适应的：确认到达越快，数据也就发送的越快。而在希望减小微小分组数目的低速广域网上，则会发送更小的分组。[1]

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。

在Socket套接字编程中，可使用TCP\_NODELAY选项来开启或者关闭Nagle算法，与实现语言无关。

#### 2.2.4.2 超时重传

TCP提供可靠的传输层,它使用的方法之一就是确认从另一端收到的数据,但是数据和确认都有可能丢失在不可靠的网络传输上,TCP通过在发送时设置一个定时器来解决这种问题,如果在定时器在溢出时还没有收到去人，它就重传该数据。[1]

#### 2.2.4.3 拥塞控制

当数据到达一个大的管道(如一个快速的局域网)并向一个较小的管道(如一个较慢的广域网)发送时便会发生拥塞,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。[1]

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

## 2.3 HTTP协议

超文本传输协议(HTTP，HyperText Transfer Protocol)是当今互联网上使用最多的应用层网络协议。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。

### 2.3.1 HTTP协议定义

HTTP（Hypertext Transfer Protocol）是应用层协议，它适应了分布式超媒体协作系统对灵活性及速度的要求。它是一个一般的、无状态的、基于对象的协议，通过对其请求方法（request methods）进行扩展，可以被用于多种用途，比如命名服务器（name server）及分布式对象管理系统。HTTP的一个特性是其数据表现类型允许系统的构建不再依赖于要传输的数据。

### 2.3.2 HTTP报文格式

HTTP客户端和服务器通过发送纯文本（ASCII）消息进行通信。客户端向服务器发送请求，服务器发送响应。

HTTP请求报文主要有请求行，请求头部和请求正文三部分组成



**图2.3.1HTTP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](http://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有K/V对组成，没对一行，Key和Value之间用冒号分隔，如下表：

**表 2.3 HTTP请求头说明**

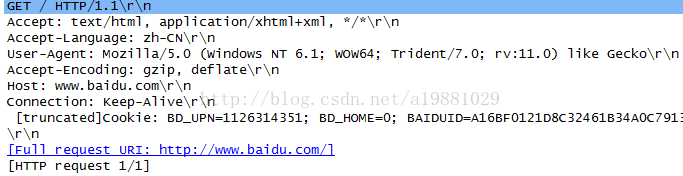
|  |  |
| --- | --- |
| 请求头 | 说明 |
| Host | 接收请求的服务器地址，可以是IP:PORT或者域名 |
| User-Agent | 发送请求的应用程序名称，通常为各浏览器内核名称 |
| Connection | 指定与连接相关的属性，如Connection:Keep-Alive |
| Accept-Charset | 通知服务端可以发送的编码格式 |
| Accept-Encoding | 通知服务端可以发送的数据压缩格式 |
| Accept-Language | 通知服务端可发送的语言 |

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

1. 请求正文：

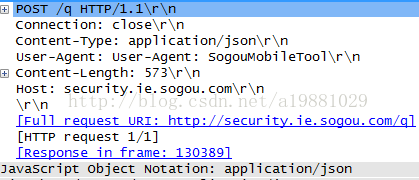
可选部分，如GET请求没有正文，而POST请求有

GET请求：



**图2.3.1 HTTP GET请求示例**

POST请求：



**图2.3.2 HTTP POST请求示例**

HTTP响应报文格式：

HTTP响应报文格式同样由状态行、响应头部、响应正文三部分组成。

**图2.3.2 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：

**表2.3.2 HTTP 状态码**

|  |  |
| --- | --- |
| 状态码 | 说明 |
| 200 | 响应成功。 |
| 302 | 跳转，跳转地址通常在Location中指定。 |
| 400 | 客户端请求由语法错误，无法被服务端识别。 |
| 403 | 服务端收到请求，但拒绝提供服务。 |
| 404 | 请求资源不存在。 |
| 500 | 服务器内部错误。 |

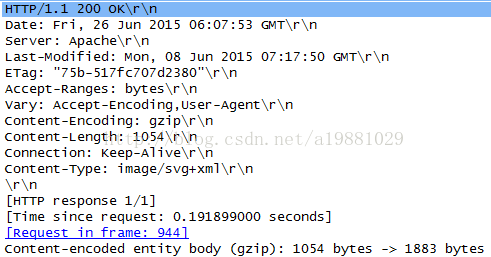
1. 响应头部：与请求头部类似，为响应报文添加附加信息。

常见响应头部如下：

**表2.3.3 HTTP 响应头说明**

|  |  |
| --- | --- |
| 响应头 | 说明 |
| Server | 服务器应用软件的名称和版本 |
| 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 |

示例：



**图 2.3.3 HTTP 响应报文示例**

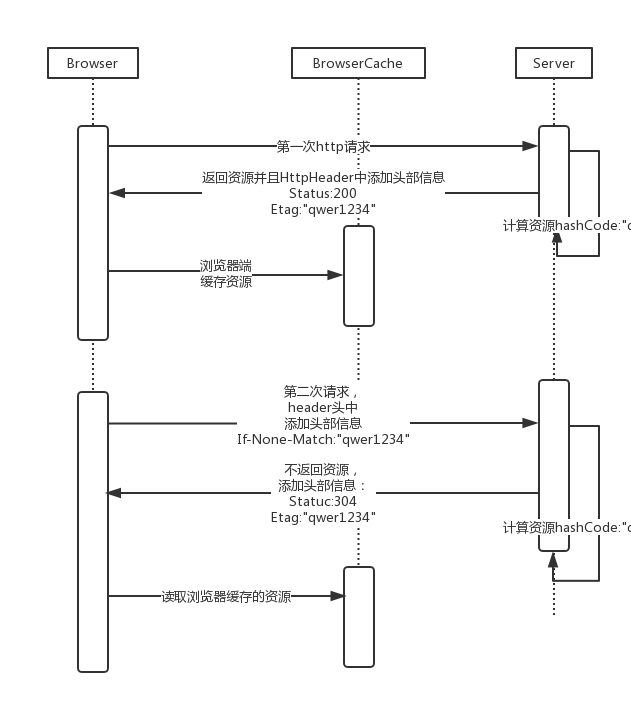
1. 响应正文：服务端返回的响应结果正文数据，如静态文件(HTML,CSS,JS,IMAGE等)，动态脚本(PHP,APS,JSP等)或者JSON数据。

### 2.3.3 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。

### 2.3.4 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，我们就希望浏览器端能够缓存资源，当服务端资源没有变化时，服务端将不发送数据到浏览器，由浏览器读取本地缓存的资源。一个典型的例子如下

**图2.3.4 HTTP缓存示例**

然而HTTP协议属于无态协议，而很多时候我们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。

# 第三章 I/O模型与多线程编程

一款高性能HTTP服务器，关键就在于在高并发请求下的处理能力。而对于操作系统I/O方式的选择以及高可用的多线程并发编程设计是一款服务器性能好坏，处理并发能力强弱的重中之重。所以了解操作系统的I/O模型的区别及优劣和熟练使用多线程编程以及保证线程安全是非常重要的。

## 3.1 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。

### 3.1.1同步，异步，阻塞，非阻塞

我们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我们需要讨论同步，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，我们称之为阻塞。反之，当操作系统不管数据是否就绪，都立即返回信息告知应用程序数据就绪或者出错，应用程序不用处于等待状态，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，我们称之为非阻塞。

**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；当应用程序想操作系统发起一个I/O请求，然后应用程序便可以去做其他时间，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。

### 3.1.2 Unix下的五种I/O模型

一般来说，程序进行输入操作有两步：

1：等待有数据可读；

2：将数据从系统内核拷贝到程序的数据区。

对于一个对Socket的输入操作，第一步一般来说是等待数据从网络上传到本地。当数据包到达的时候，数据将会从网络层拷贝到内核的缓冲区中；第二步是把内核中的数据拷贝到程序的数据区中。[2]

* **阻塞式I/O：**阻塞I/O模式是最普遍使用的I/O 模式。大部分程序使用的都是阻塞模式的I/O。缺省的，一个套接字建立后所处于的模式就是阻塞I/O模式。 对于一个 UDP套接字来说，数据就绪的标志比较简单：

1. 已经收到了一整个数据报
2. 没有收到。

而TCP这个概念就比较复杂，需要附加一些其他的变量。

在下图中，一个进程调用recvfrom，然后系统调用并不返回知道有数据报到达本地系统，然后系统将数据拷贝到进程的缓存中。（如果系统调用收到一个中断信号，则它的调用会被中断）我们称这个进程在调用 recvfro一直到从recvfrom返回这段时间是阻塞的。当recvfrom 正常返回时，我们的进程继续它的操作。[2]

**图3.1.1阻塞式I/O**

* **非阻塞式I/O：**当我们将一个套接字设置为非阻塞模式，我们相当于告诉了系统内核：“当我请求的I/O操作不能够马上完成，你想让我的进程进行休眠等待的时候，不要这么做，请马上返回一个错误给我。”

我们可以参照下图来描述非阻塞模式I/O。

我们开始对 recvfrom 的三次调用，因为系统还没有接收到网络数据，所以内核马上返回一个EWOULDBLOC的错误。第四次我们调用recvfrom函数，一个数据报已经到达了， 内核将它拷贝到我们的应用程序的缓冲区中，然后recvfrom正常返回，我们就可以对接收到的数据进行处理了。当一个应用程序使用了非阻塞模式的套接字，它需要使用一个循环来不停的测试是否一个文件描述符有数据可读（称做polling）。应用程序不停的polling内核来检查是否I/O操作已经就绪。这将是一个极浪费CPU资源的操作。这种模式使用中不是很普遍。[2]

**图3.1.2非阻塞式I/O**

* **I/O复用(select，poll,epoll)：**在使用I/O多路技术的时候，我们调用 select()函数和poll()函数，在调用它们的时候阻塞，而不是我们来调用recvfrom（或 recv）的时候阻塞。

下图说明了它的工作方式。

当我们调用select函数阻塞的时候，select函数等待数据报套接字进入读就绪状态。当select函数返回的时候，也就是套接字可以读取数据的时候。这时候我们就可以调用recvfrom 函数来将数据拷贝到我们的程序缓冲区中。和阻塞模式相比较，select()和 poll()并没有什么高级的地方，而且，在阻塞模式下只需要调用一个函数：读取或发送，在使用了多路复用技术后，我们需要调用两个函数：先调用select()函数或 poll()函数，然后才能进行真正的读写。多路复用的高级之处在于，它能同时等待多个文件描述符，而这些文件描述符（套接字描述符）其中的任意一个进入读就绪状态，select()函数就可以返回。[2]

**图3.1.3 I/O复用**

假设我们运行一个网络客户端程序，要同时处理套接字传来的网络数据又要处理本地的标准输入输出。在我们的程序处于阻塞状态等待标准输入的数据的时候，假如服务器端的程序被kill（或是自己Down掉了），那么服务器程端的TCP 协议会给客户端（我们这端）的TCP协议发送一个FIN数据代表终止连接。但是我们的程序阻塞在等待标准输入的数据上，在它读取套接字数据之前（也许是很长一段时间），它不会看见结束标志，我们就不能够使用阻塞模式的套接字。 IO多路技术一般在下面这些情况中被使用：

* 当一个客户端需要同时处理多个文件描述符的输入输出操作的时候（一般来说是 标准的输入输出和网络套接字）， I/O多路复用技术将会有机会得到使用。当程序需要同时进行多个套接字的操作的时候。
* 如果一个 TCP 服务器程序同时处理正在侦听网络连接的套接字和已经连接好的套接字。
* 如果一个服务器程序同时使用TCP和UDP协议。
* 如果一个服务器同时使用多种服务并且每种服务可能使用不同的协议（比如 inetd 就是这样的）。
* I/O 多路服用技术并不只局限与网络程序应用上。几乎所有的程序都可以找到应用 I/O 多路复用的地方。
* **信号驱动式I/O(SIGIO)：**我们可以使用信号，让内核在文件描述符就绪的时候使用SIGIO信号通知我们。

我们将这种模式称为信号驱动I/O模式。使用这种模式，我们首先需要允许套接字使用信号驱动I/O，还要安装一个SIGIO的处理函数。在这种模式下，系统调用将会立即返回，然后我们的程序可以继续做其他的事情。当数据就绪的时候，系统会向我们的进程发送一个SIGIO信号。这样我们就可以SIGIO信号的处理函数中进行I/O操作（或是我们在函数中通知主函数有数据可读）。

对于信号驱动I/O模式，它的先进之处在于它在等待数据的时候不会阻塞程序可以做自己的事情。当有数据到达的时候，系统内核会向程序发送一个 SIGIO 信号进行通知，这样我们的程序就可以获得更大的灵活性，因为我们不必为等待数据进行额外的编码。[2]

**图3.1.4 信号驱动式I/O**

信号I/O可以使内核在某个文件描述符发生改变的时候发信号通知我们的程序。异步I/O可以提高我们程序进行I/O读写的效率。通过使用它，当我们的程序进行I/O操作的时候，内核可以在初始化I/O操作后立即返回，在进行I/O操作的同时，我们的程序可以做自己的事情，直到 I/O 操作结束，系统内核给我们的程序发消息通知。

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。

为了在一个套接字上使用信号驱动I/O操作，下面这三步是所必须的。

(1)一个和SIGIO信号的处理函数必须设定。

(2)套接字的拥有者必须被设定。一般来说是使用fcntl函数的 F\_SETOWN参数来进行设定拥有者。

(3)套接字必须被允许使用异步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)：**当我们运行在异步I/O模式下时，我们如果想进行I/O操作，只需要告诉内核我们要进行I/O操作，然后内核会马上返回。具体的I/O和数据的拷贝全部由内核来完成，我们的程序可以继续向下执行。当内核完成所有的I/O操作和数据拷贝后，内核将通知我们的程序。

异步I/O和信号驱动I/O的区别是：

* + 信号驱动I/O模式下，内核在操作可以被操作的时候通知给我们的应用程序发送SIGIO消息。
  + 异步I/O模式下，内核在所有的操作都已经被内核操作结束之后才会通知我们的应用程序。

如下图，当我们进行一个 IO 操作的时候，我们传递给内核我们的文件描述符，我们的缓存区指针和缓存区的大小，一个偏移量offset，以及在内核结束所有操作后和我们联系的方法。这种调用也是立即返回的，我们的程序不需要阻塞住来等待数据的就绪。我们可以要求系统内核在所有的操作结束后（包括从网络上读取信息，然后拷贝到我们提供给内核的缓存区中）给我们发一个消息。[2]

**图3.1.5 异步I/O**

* **几种I/O模式的比较**

下面这个表格对这几种I/O模式进行了对比。



**3.1.6 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。

## 3.2 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，们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，有助于帮助我们更好的了解服务器。

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，应用于并发数相对低的场景。每个socket对应一个线程，用来接收socket(http)请求以及进行数据处理，再将响应发送回客户端。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 Web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切及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，故其一个重要的特性是平台无关系，即我们所说的一次编写，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

本届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/0模型与线程模型

服务器采用IO多路复用模型，上节分析linux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。其线程模型图如下。

**图4.2.1 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各层次。

* 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。

执行流程图如下所示：



**图4.2.2 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。

流程图如下：



**图4.2.3 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，介绍如果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。没个状态的说明如下：

* 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：

1. **package** org.dam.utils.lifecycle;
3. **import** java.util.\*;
4. **import** java.util.concurrent.CopyOnWriteArrayList;
6. /\*\*
7. \* Created by geeche on 2018/2/3.
8. \*/
9. **public** **class** ContainerLifeCycle **extends** AbstractLifeCycle **implements** Destroyable {
11. **private** **final** List<Bean> beans = **new** CopyOnWriteArrayList<Bean>();
13. **private** **boolean** started = **false**;
15. @Override
16. **protected** **void** doStart() **throws** Exception {
17. **for**(Bean bean : beans){
18. **if**(bean.bean **instanceof** LifeCycle){
19. LifeCycle lc = (LifeCycle)bean.bean;
20. **if**(!lc.isRunning()){
21. lc.start();
22. }
23. }
24. }
25. started = **true**;
26. **super**.doStart();
27. }
29. @Override
30. **protected** **void** doStop() **throws** Exception {
31. started = **false**;
32. **super**.doStop();
33. **final** List<Bean> reverse = **new** ArrayList<Bean>(beans);
34. Collections.reverse(reverse);
35. **for**(Bean bean : reverse){
36. **if**(bean.bean **instanceof** LifeCycle){
37. LifeCycle lc = (LifeCycle)bean;
38. **if**(!lc.isRunning()){
39. lc.stop();
40. }
41. }
42. }
43. started = **true**;
44. **super**.doStart();
45. }
47. @Override
48. **public** **void** destroy() {
49. **final** List<Bean> reverse = **new** ArrayList<Bean>(beans);
50. Collections.reverse(reverse);
51. **for**(Bean bean : reverse){
52. **if**(bean.bean **instanceof** Destroyable){
53. Destroyable de = (Destroyable) bean;
54. de.destroy();
55. }
56. }
57. }
59. **public** **boolean** Contains(Object obj){
60. **for**(Bean bean : beans){
61. **if**(bean.bean == obj){
62. **return** **true**;
63. }
64. }
65. **return** **false**;
66. }

69. **public** **boolean** addBean(Object obj){
70. **return** addBean(obj,!((obj **instanceof** LifeCycle) && ((LifeCycle)obj).isStarted()));
71. }
72. **public** **boolean** addBean(Object obj,**boolean** managed){
73. **if**(Contains(obj)){
74. **return** **false**;
75. }
76. Bean bean = **new** Bean(obj);
77. bean.managed = managed;
78. beans.add(bean);
80. **if**(obj **instanceof** LifeCycle){
81. LifeCycle lc = (LifeCycle)obj;
82. **if**(managed && started){
83. **try**{
84. lc.start();
85. }**catch** (Exception e){
86. **throw** **new** RuntimeException (e);
87. }
88. }
89. }
90. **return** **true**;
91. }
93. **public** **void** manage(Object bean){
94. **for**(Bean b : beans){
95. **if**(b.bean == bean){
96. b.managed = **true**;
97. **return**;
98. }
99. }
100. **throw** **new** IllegalArgumentException();
101. }
103. **public** **void** ummanage(Object bean){
104. **for**(Bean b : beans){
105. **if**(b.bean == bean){
106. b.managed = **false**;
107. **return**;
108. }
109. }
110. **throw** **new** IllegalArgumentException();
111. }
113. **public** Collection<Object> getBeans(){
114. **return** getBeans(Object.**class**);
115. }
117. **public** <T>List<T> getBeans(Class<T> clazz){
118. ArrayList<T> beanRes = **new** ArrayList<T>();
119. **for**(Bean bean : beans){
120. **if**(clazz.isInstance(bean.bean)){
121. beanRes.add((T)bean.bean);
122. }
123. }
124. **return** beanRes;
125. }
127. **public** <T>T getBean(Class<T> clazz){
128. **for**(Bean bean : beans){
129. **if**(clazz.isInstance(bean.bean)){
130. **return** (T)bean.bean;
131. }
132. }
133. **return** **null**;
134. }
136. **public** **void** removeBeans(){
137. beans.clear();
138. }
140. **public** **boolean** removeBean(Object obj){
141. Iterator<Bean> iterator = beans.iterator();
142. **while** (iterator.hasNext()){
143. Bean bean = iterator.next();
144. **if**(bean.bean == obj){
145. beans.remove(obj);
146. **return** **true**;
147. }
148. }
149. **return** **false**;
150. }
152. **private** **class** Bean{
153. **final** Object bean;
154. **volatile** **boolean** managed = **true**;
155. Bean(Object bean){
156. **this**.bean = bean;
157. }
158. }
160. }

整个关系类图如下：



**图 4.3.1 生命周期UML类图**

### 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(),**interestOps**,**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*** }  
  
}

以下为整个流程时序图：



**图 4.3.3 Reactor时序图**

### 4.3.4 管道模式与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类图如下：



**图 4.3.4 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\_Server\_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***.getDesc());  
 }  
 **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web app中五种返回类型，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。基于以上我们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类图如下：



**图 4.3.5 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\_Server\_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，然而我们需要根据不同的状态生成不同的http响应头信息。这里介绍的是我们如何通过策略模式，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类如下：



**图 4.3.6 响应头生成策略模式UML类图**

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：

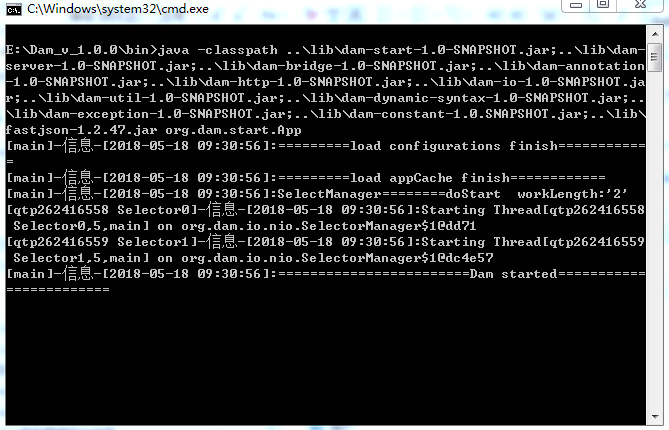
**public void** generate(){  
 **if**(HttpHelper.*is20X*(**statusCode**)){  
 **responseStrategy** = **new** OKResponseStrategy(**handleResult**,**request**,**response**);  
 }  
 **if**(HttpHelper.*is30X*(**statusCode**)){  
 **responseStrategy** = **new** RedirectReponseStrategy(**request**,**response**,(String) **extend**);  
 }  
 **if**(HttpHelper.*is40X*(**statusCode**)){  
 **responseStrategy** = **new** ClientErrorResponseStrategy(**request**,**response**,HttpHelper.*handleError*((String)**extend**));  
 }  
 **if**(HttpHelper.*is50X*(**statusCode**)){  
 **responseStrategy** = **new** ServerErrorResponseStrategy(**request**,**response**,HttpHelper.*handleError*((String)**extend**));  
 }  
 **responseStrategy**.doGenerate();  
}

## 4.4运行服务器

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:web app所在的目录。
* Readme.md:应用程序说明解释。

而我们只需点击bin目录下的run脚本即可运行服务器，当我们看到Dam started字符即说明服务器启动成功，如下图所示：



**图 4.4 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web应用程序开发

服务器所做的开发工作完成之后，我们迫不及待就想要开发一款web应用，让其运行在我们的服务器之上，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我们所需要的一些信息，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

我们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。

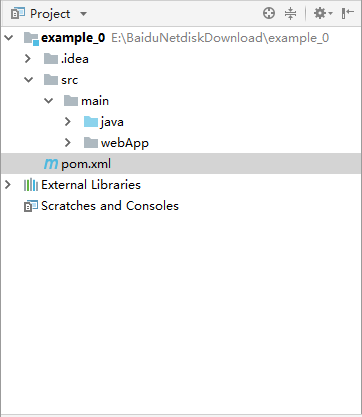
首先我们在脚本内定义一个变量语法为:{$}

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

### 4.5.3 开发webApp

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。

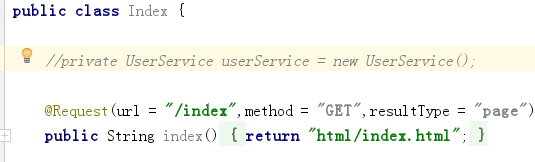
1. 首先我们需要新建一个项目，这里我们同样采用maven进行构建，当然你也可以手动导入响应jar包。新建之后的项目目录结构如下：



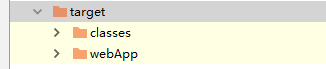
**图 4.5.3 webapp结果示意图**

src/main/java目录下为我们的java源文件。Webapp下为我们的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。

1. 新建一个Java类，我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/”。如下图所示：



1. 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下如：



**图4.5.3 编译后包结果示意图**

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

1. 在我们服务器www目录下新建一个文件夹，该文件夹为我们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

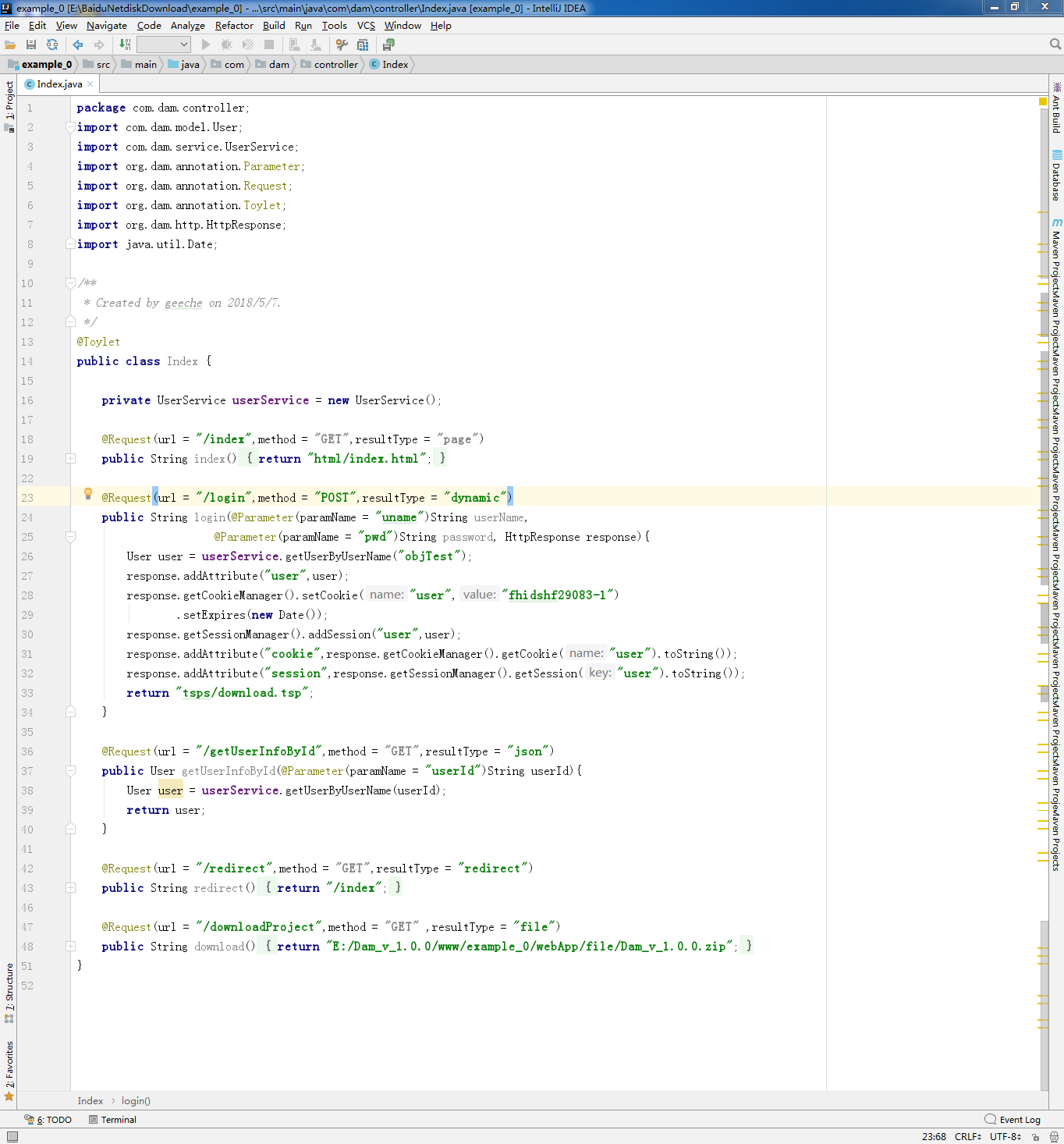
本节将使用一个按照Dam服务器规范的web应用程序demo，实际演示服务器是否能够正常运行。

## 5.1黑盒测试

黑盒测试，顾名思义就是在不需要关注程序本身的代码运行逻辑，只关注功能的健壮性即可。符合我们对Dam服务器演示测试的需求。

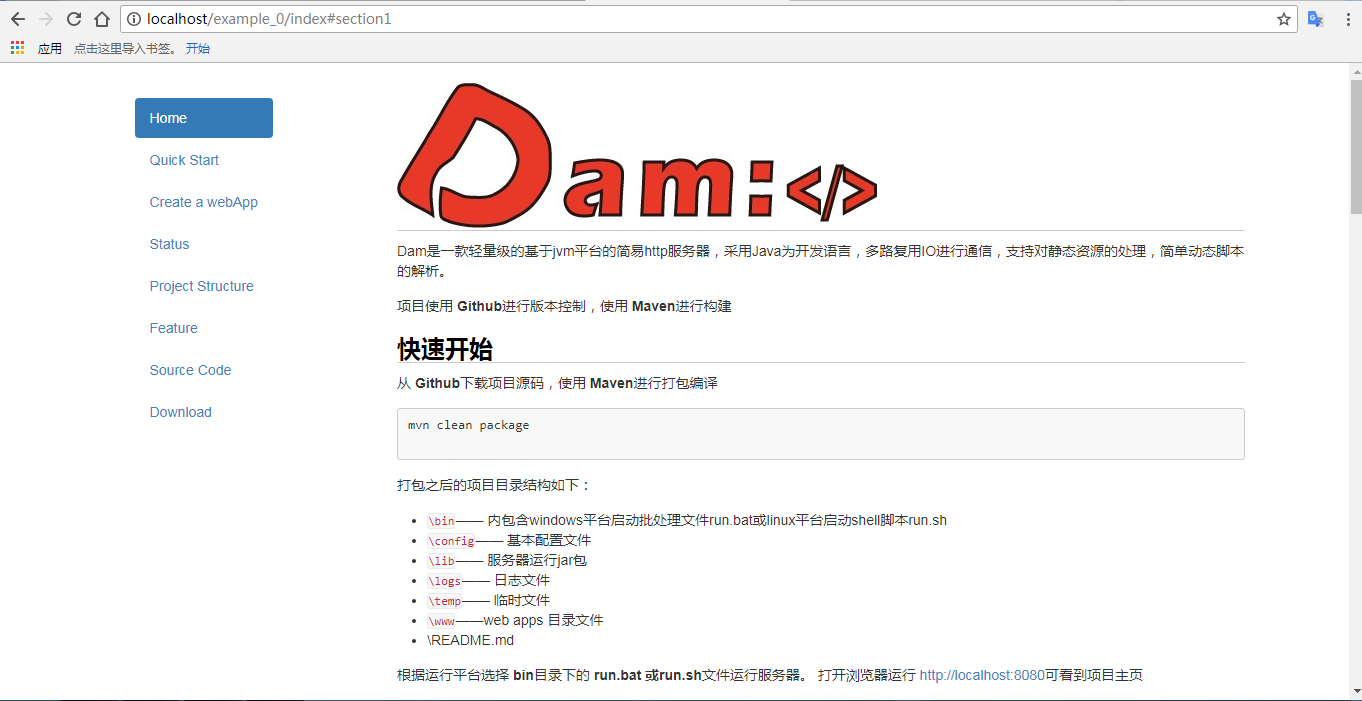
首先是我们的Demo，控制器类包含了对6中返回类型的测试，对静态文件，动态脚本的测试，对重定向，客户端缓存，cookie，session等功能的测试。

类文件如下图所示：



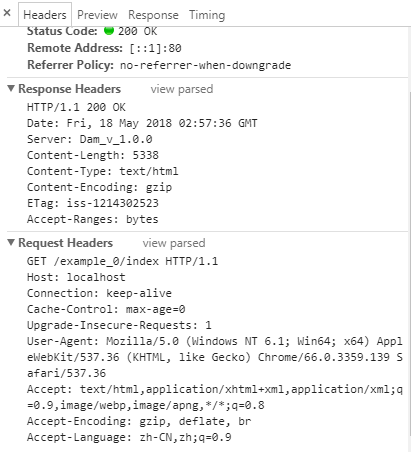
**图 5.1.1 demo控制器类**

第一个函数index()对应的url为”/index”，返回值类page，一个静态页面。我们浏览器输入localhost/example\_0/index验证结果截图如下：



**图 5.1.2 demo主页**

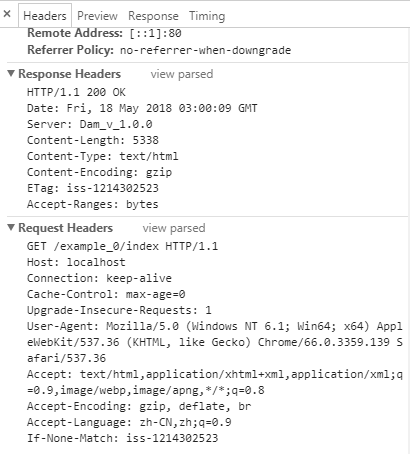
然后我们继续看第一次进入index页面时http请求头与响应头如下图所示：



**图 5.1.3 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。

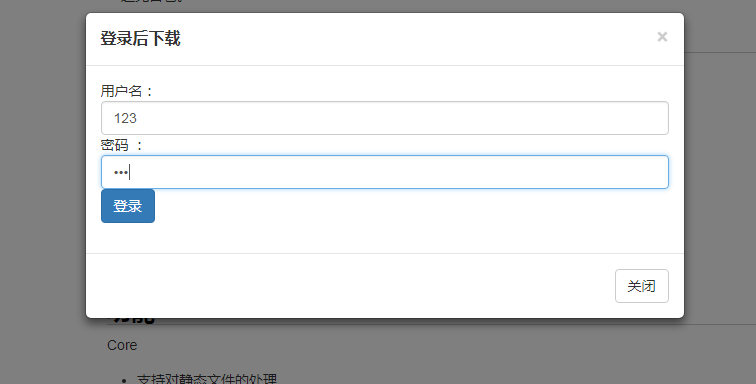
第二次请求index页面请求头与响应头如下图所示：



**图 5.1.4 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。

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。我们从index页面表单登录执行login函数，结果如下图所示：



**图 5.1.5 登录**



**图 5.1.6 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。首先我们看看动态脚本代码：



**图 5.1.7 .tsp脚本**

结果如我们预期的一样动态脚本中的${}变量均显示正常。

在看看对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与我们服务端的也一样。

以上均通过测试。

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。

我们在浏览器输入localhost/example\_0/getUserById?userId=1,结果显示如下图：



**图 5.1.8 json结果返回页**

和预期一样返回的为一个json字符串。

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

第五个函数download()返回类型为file类型，该功能即文件下载功能。我们点击login跳转页面后的Dam\_v\_1\_0\_0.zip即可下载文件。结果显示如下图所示：



**图 5.1.9 下载成功页**

左下角文件下载成功，结果与预期一致。

以上就是对整个服务器的测试结果。与我们所预期的全部一致。

# 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io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http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

整过毕设过程设计内容广泛，一路下来遇到了很多砍，感谢老师一直以来对自己的指导，感谢父母对自己的鼓励，感谢自己坚持下来解决一个又一个难点，更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。

# 参考文献

1. W.RICHARD STEVENS 著 范建华 等译. TCP/IP详解,卷1：协议.北京:机械工业出版社,2004.4
2. W.RICHARD STEVENS Bill FENNER ANDREW M.RUDOFF著 杨继张译.UNIX网络编程.北京:清华大学出版社,2017.1
3. 上野宣 著 于均良译.图解HTTP.北京:人名邮电出版社,1014.4
4. 严体华 著.网络管理员(第三版).北京:清华大学出版社,2011,9