## 课程目标

1、掌握 Web 服务器 Tomcat 的运行原理。

2、掌握 Netty API 的基本应用。

## 内容定位

1、熟练使用 Tomcat 部署 Web 项目。

2、熟悉 Java 网络编程。

## 1、环境准备

Netty 作为底层通信框架，用来实现 Web 容器自然也不难，我们先介绍一下整体实现思路。我们知道，Tomcat 是基于 J2EE 规范的 Web 容器，主要入口是 web.xml 文件。web.xml 文件中主要配置 Servlet、Filter、Listener 等，而 Servlet、Filter、Listener 在 J2EE 中只是抽象的实现，具体业务逻辑由开发者来实现。本章内容，就以最常用的 Servlet 为例来详细展开。

### 1.1 定义 GPServlet 抽象类

首先，我们创建 GPServlet 类。我们都知道 GPServlet 生命周期中最常用的方法是 doGet()方法和 doPost()方法，而 doGet()方法和 doPost()方法是 service()方法的分支实现，看下面的简易版 Servlet 源码实现。

```
package com.gupaoedu.vip.netty.tomcat.bio.http;

public abstract class GPServlet {

    public void service(GPRequest request,GPResponse response) throws Exception{
        if("GET".equalsIgnoreCase(request.getMethod())){
            doGet(request,response);
        }else {
            doPost(request,response);
        }
    }


    public abstract void doGet(GPRequest request,GPResponse response) throws Exception;

    public abstract void doPost(GPRequest request,GPResponse response) throws Exception;
```

```
}
```

从上面的代码中，我们看到，doGet()方法和 doPost()方法中有两个参数 GPRequest 和 GPResponse 对象，这两个对象是由 Web 容器创建的，主要是对底层 Socket 的输入输出的封装。其中 GPRequest 是对 Input 的封装，GPResponse 是对 Output 的封装。

## 1.2 创建用户业务代码

下面基于 GPServlet 来实现两个业务逻辑 FirstServlet 和 SecondServlet。FirstServlet 类的实现代码如下。

```
package com.gupaoedu.vip.netty.tomcat.bio.servlet;

import com.gupaoedu.vip.netty.tomcat.bio.http.GPRequest;
import com.gupaoedu.vip.netty.tomcat.bio.http.GPResponse;
import com.gupaoedu.vip.netty.tomcat.bio.http.GPServlet;

public class FirstServlet extends GPServlet {
    public void doGet(GPRequest request, GPResponse response) throws Exception {
        this.doPost(request,response);
    }

    public void doPost(GPRequest request, GPResponse response) throws Exception {
       response.write("This is first servlet from BIO.");
    }
}
```

SecondServlet 类的实现代码如下。

```
package com.gupaoedu.vip.netty.tomcat.bio.servlet;

import com.gupaoedu.vip.netty.tomcat.bio.http.GPRequest;
import com.gupaoedu.vip.netty.tomcat.bio.http.GPResponse;
import com.gupaoedu.vip.netty.tomcat.bio.http.GPServlet;

public class SecondServlet extends GPServlet {
    public void doGet(GPRequest request, GPResponse response) throws Exception {
        this.doPost(request,response);
    }

    public void doPost(GPRequest request, GPResponse response) throws Exception {
        response.write("This is second servlet from BIO.");
    }
}
```

## 1.3 完成 web.properties 配置

为了简化操作，我们用 web.properties 文件代替 web.xml 文件，具体内容如下。

```
servlet.one.className=com.gupaoedu.vip.netty.tomcat.bio.servlet.FirstServlet
servlet.one.url=/firstServlet.do
```

```
servlet.two.className=com.gupaoedu.vip.netty.tomcat.bio.servlet.SecondServlet
servlet.two.url=/secondServlet.do
```

上述代码分别给两个 Servlet 配置了/firstServlet.do 和/secondServlet.do 的 URL 映射。

# 2、基于传统 I/O 手写 Tomcat

下面我们来看 GPRequest 和 GPResponse 的基本实现。

## 2.1 创建 GPRequest 对象

GPRequest 主要就是对 HTTP 的请求头信息进行解析。我们从浏览器发送一个 HTTP 请求，如在浏览器地址栏中输入 http://localhost:8080，后台服务器获取的请求其实就是一串字符串，具体格式如下。

```
GET / HTTP/1.1
Host: localhost:8080
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/75.0.3770.142
Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v
=b3
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-CN,zh;q=0.9
```

在 GPRequest 获得输入内容之后，对这一串满足 HTTP 的字符信息进行解析。我们来看 GPRequest 简单直接的代码实现。

```
package com.gupaoedu.vip.netty.tomcat.bio.http;

import java.io.InputStream;

public class GPRequest {

    private String method;
    private String url;

    public GPRequest(InputStream is) {

        try {
            //拿到 HTTP 协议的具体内容
            String content = "";
            byte[] buff = new byte[1024];
            int len = 0;
            if ((len = is.read(buff)) > 0) {
                content = new String(buff, 0, len);
            }
```

```
            String line = content.split("\\n")[0];
            String [] arr = line.split("\\s");

            this.method = arr[0];
            this.url = arr[1].split("\\?")[0];


//            System.out.println(content);

        }catch (Exception e){
            e.printStackTrace();
        }

    }

    public String getUrl(){
        return this.url;
    }
    public String getMethod(){
        return this.method;
    }

}
```

在上面的代码中，GPRequest主要提供了getUrl()方法和getMethod()方法。输入流InputStream作为GPRequest的构造参数传入，在构造函数中，用字符串切割的方法提取请求方式和URL。

## 2.2 创建GPResponse对象

接下来看GPResponse的实现，与GPRequest的实现思路类似，就是按照HTTP规范从Output输出格式化的字符串，来看代码。

```
package com.gupaoedu.vip.netty.tomcat.bio.http;

import java.io.OutputStream;

public class GPResponse {
    private OutputStream out;

    public GPResponse(OutputStream os) {
        this.out = os;
    }

    public void write(String s) throws Exception{
        StringBuilder sb = new StringBuilder();
        sb.append("HTTP/1.1 200 ok\n")
                .append("Content-Type: text/html;\n")
                .append("\r\n")
                .append(s);
        out.write(sb.toString().getBytes());
    }

}
```

上面的代码中，输出流 OutputStream 作为 GPResponse 的构造参数传入，主要提供了一个 write()方法。通过 write()方法按照 HTTP 规范输出字符串。

## 2.3 创建 GPTomcat 启动类

前面 4.2.1 和 4.2.2 两节只是对 J2EE 规范的再现，接下来就是真正 Web 容器的实现逻辑，分为三个阶段：初始化阶段、服务就绪阶段、接受请求阶段。

第一阶段：初始化阶段，主要是完成对 web.xml 文件的解析。

```
package com.gupaoedu.vip.netty.tomcat.bio;

import com.gupaoedu.vip.netty.io.bio.tomcat.http.GPRequest;
import com.gupaoedu.vip.netty.io.bio.tomcat.http.GPResponse;
import com.gupaoedu.vip.netty.io.bio.tomcat.http.GPServlet;

import java.io.FileInputStream;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.HashMap;
import java.util.Map;
import java.util.Properties;

/**
 * Created by Tom.
 */
public class GPTomcat {
    private int port = 8080;
    private ServerSocket server;
    private Map<String,GPServlet> servletMapping = new HashMap<String,GPServlet>();

    private Properties webxml = new Properties();


    private void init(){

        //加载 web.xml 文件，同时初始化 ServletMapping 对象
        try{
            String WEB_INF = this.getClass().getResource("/").getPath();
            FileInputStream fis = new FileInputStream(WEB_INF + "web.properties");

            webxml.load(fis);

            for (Object k : webxml.keySet()) {

                String key = k.toString();
                if(key.endsWith(".url")){
                    String servletName = key.replaceAll("\\.url$", "");
                    String url = webxml.getProperty(key);
                    String className = webxml.getProperty(servletName + ".className");
                    //单实例，多线程
                    GPServlet obj = (GPServlet)Class.forName(className).newInstance();
```

```
                servletMapping.put(url, obj);
            }

        }


    }catch(Exception e){
        e.printStackTrace();
    }

  }
}
```

上面代码中，首先从 WEB-INF 读取 web.properties 文件并对其进行解析，然后将 URL 规则和 GPServlet 的对应关系保存到 servletMapping 中。

第二阶段：服务就绪阶段，完成 ServerSocket 的准备工作。在 GPTomcat 类中增加 start()方法。

```
  public void start(){

    //1.加载配置文件，初始化 ServletMapping
    init();

    try {
        server = new ServerSocket(this.port);

        System.out.println("GPTomcat 已启动，监听的端口是：" + this.port);

        //2.等待用户请求，用一个死循环来等待用户请求
        while (true) {
            Socket client = server.accept();
            //3.HTTP 请求，发送的数据就是字符串——有规律的字符串（HTTP）
            process(client);

        }

    } catch (Exception e) {
        e.printStackTrace();
    }
  }
```

第三阶段：接受请求阶段，完成每一次请求的处理。在 GPTomcat 中增加 process()方法的实现。

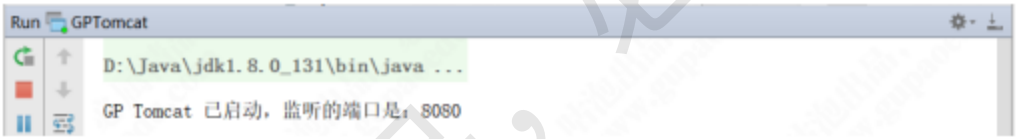
```
  private void process(Socket client) throws Exception {

    InputStream is = client.getInputStream();
    OutputStream os = client.getOutputStream();

    //4.Request(InputStrean)/Response(OutputStrean)
    GPRequest request = new GPRequest(is);
    GPResponse response = new GPResponse(os);

    //5.从协议内容中获得 URL，把相应的 Servlet 用反射进行实例化
    String url = request.getUrl();

    if(servletMapping.containsKey(url)){
```

```
            //6.调用实例化对象的 service()方法，执行具体的逻辑 doGet()/doPost()方法
            servletMapping.get(url).service(request,response);
        }else{
            response.write("404 - Not Found");
        }


        os.flush();
        os.close();

        is.close();
        client.close();
    }
```

每次客户端请求过来以后，从 servletMapping 中获取其对应的 Servlet 对象，同时实例化 GPRequest 和 GPResponse 对象，将 GPRequest 和 GPResponse 对象作为参数传入 service()方法，最终执行业务逻辑。最后，增加 main()方法。

```
    public static void main(String[] args) {
        new GPTomcat().start();
    }
```

服务启动后，运行效果如下图所示。




## 3、基于 Netty 重构 Tomcat 实现

了解了传统的 I/O 实现方式之后，我们发现 Netty 版本的实现就比较简单了，来看具体的代码实现。

### 3.1 重构 GPTomcat 逻辑

话不多说，直接看代码。

```
package com.gupaoedu.vip.netty.tomcat.nio;

import com.gupaoedu.vip.netty.tomcat.nio.http.GPRequest;
import com.gupaoedu.vip.netty.tomcat.nio.http.GPResponse;
import com.gupaoedu.vip.netty.tomcat.nio.http.GPServlet;
import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.*;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioServerSocketChannel;
import io.netty.handler.codec.http.*;

import java.io.FileInputStream;
import java.net.ServerSocket;
```

```
import java.util.HashMap;
import java.util.Map;
import java.util.Properties;

//Netty 就是一个同时支持多协议的网络通信框架
public class GPTomcat {
    //打开 Tomcat 源码，全局搜索 ServerSocket

    private int port = 8080;

    private Map<String,GPServlet> servletMapping = new HashMap<String,GPServlet>();

    private Properties webxml = new Properties();

    private void init(){

        //加载 web.xml 文件，同时初始化 ServletMapping 对象
        try{
            String WEB_INF = this.getClass().getResource("/").getPath();
            FileInputStream fis = new FileInputStream(WEB_INF + "web.properties");

            webxml.load(fis);

            for (Object k : webxml.keySet()) {

                String key = k.toString();
                if(key.endsWith(".url")){
                    String servletName = key.replaceAll("\\.url$", "");
                    String url = webxml.getProperty(key);
                    String className = webxml.getProperty(servletName + ".className");
                    GPServlet obj = (GPServlet)Class.forName(className).newInstance();
                    servletMapping.put(url, obj);
                }
            }

        }catch(Exception e){
            e.printStackTrace();
        }
    }

    public void start(){

        init();


        //Netty 封装了 NIO 的 Reactor 模型，Boss，Worker
        //Boss 线程
        EventLoopGroup bossGroup = new NioEventLoopGroup();
        //Worker 线程
        EventLoopGroup workerGroup = new NioEventLoopGroup();
        try {

            //1.创建对象
            ServerBootstrap server = new ServerBootstrap();


            //2.配置参数
            //链路式编程
            server.group(bossGroup, workerGroup)
                //主线程处理类，看到这样的写法，底层就是用反射
                .channel(NioServerSocketChannel.class)
                //子线程处理类，Handler
                .childHandler(new ChannelInitializer<SocketChannel>() {
```

```
            //客户端初始化处理
              protected void initChannel(SocketChannel client) throws Exception {
                //无锁化串行编程
                //Netty 对 HTTP 的封装，对顺序有要求
                //HttpResponseEncoder 编码器
                //责任链模式，双向链表 Inbound OutBound
                client.pipeline().addLast(new HttpResponseEncoder());
                //HttpRequestDecoder 解码器
                client.pipeline().addLast(new HttpRequestDecoder());
                //业务逻辑处理
                client.pipeline().addLast(new GPTomcatHandler());
            }
        })
        //针对主线程的配置 分配线程最大数量 128
        .option(ChannelOption.SO_BACKLOG, 128)
        //针对子线程的配置 保持长连接
        .childOption(ChannelOption.SO_KEEPALIVE, true);

        //3.启动服务器
        ChannelFuture f = server.bind(port).sync();
        System.out.println("GPTomcat 已启动，监听的端口是：" + port);
        f.channel().closeFuture().sync();
    }catch (Exception e){
        e.printStackTrace();
    }finally {
        //关闭线程池
        bossGroup.shutdownGracefully();
        workerGroup.shutdownGracefully();
    }
}

public class GPTomcatHandler extends ChannelInboundHandlerAdapter {
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {

        if (msg instanceof HttpRequest){
            System.out.println("hello");
            HttpRequest req = (HttpRequest) msg;

            //转交给我们自己的 Request 实现
            GPRequest request = new GPRequest(ctx,req);
            //转交给我们自己的 Response 实现
            GPResponse response = new GPResponse(ctx,req);
            //实际业务处理
            String url = request.getUrl();

            if(servletMapping.containsKey(url)){
                servletMapping.get(url).service(request, response);
            }else{
                response.write("404 - Not Found");
            }

        }
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {

    }

}

public static void main(String[] args) {
```

```
        new GPTomcat().start();
    }

}
```

代码的基本思路和基于传统 I/O 手写的版本一致，不再赘述。

### 3.2 重构 GPRequest 逻辑

我们先来看代码。

```
package com.gupaoedu.vip.netty.tomcat.nio.http;

import io.netty.channel.ChannelHandlerContext;
import io.netty.handler.codec.http.HttpRequest;
import io.netty.handler.codec.http.QueryStringDecoder;

import java.util.List;
import java.util.Map;

public class GPRequest {

    private ChannelHandlerContext ctx;

    private HttpRequest req;

    public GPRequest(ChannelHandlerContext ctx, HttpRequest req) {
        this.ctx = ctx;
        this.req = req;
    }

    public String getUrl() {
        return req.uri();
    }

    public String getMethod() {
        return req.method().name();
    }

    public Map<String, List<String>> getParameters() {
        QueryStringDecoder decoder = new QueryStringDecoder(req.uri());
        return decoder.parameters();
    }

    public String getParameter(String name) {
        Map<String, List<String>> params = getParameters();
        List<String> param = params.get(name);
        if (null == param) {
            return null;
        } else {
            return param.get(0);
        }
    }
}
```

和基于传统的 I/O 手写的版本一样，提供 getUrl()方法和 getMethod()方法。在 Netty 的版本中，我们增加了

getParameter()的实现，供大家参考。

### 3.3 重构 GPResponse 逻辑

还是继续看代码。

```
package com.gupaoedu.vip.netty.tomcat.nio.http;

import io.netty.buffer.Unpooled;
import io.netty.channel.ChannelHandlerContext;
import io.netty.handler.codec.http.*;

public class GPResponse {

    //SocketChannel 的封装
    private ChannelHandlerContext ctx;

    private HttpRequest req;

    public GPResponse(ChannelHandlerContext ctx, HttpRequest req) {
     this.ctx = ctx;
     this.req = req;
    }

    public void write(String out) throws Exception {
     try {
         if (out == null || out.length() == 0) {
          return;
         }
         //设置 HTTP 及请求头信息
         FullHttpResponse response = new DefaultFullHttpResponse(
          //设置版本为 HTTP 1.1
          HttpVersion.HTTP_1_1,
          //设置响应状态码
          HttpResponseStatus.OK,
          //将输出内容编码格式设置为 UTF-8
          Unpooled.wrappedBuffer(out.getBytes("UTF-8")));

         response.headers().set("Content-Type", "text/html;");

         ctx.write(response);
     } finally {
         ctx.flush();
         ctx.close();
     }
    }
}
```

相对于基于传统的 I/O 手写的版本而言，主要变化就是利用 Netty 对 HTTP 的默认支持，可以使用现成的 API。

### 3.4 运行效果演示

启动容器，我们在浏览器地址栏中输入 http://localhost:8080/firstServlet.do，可以得到如下图所示的结果。


在浏览器地址栏中输入 http://localhost:8080/secondServlet.do，可以得到如下图所示的结果。