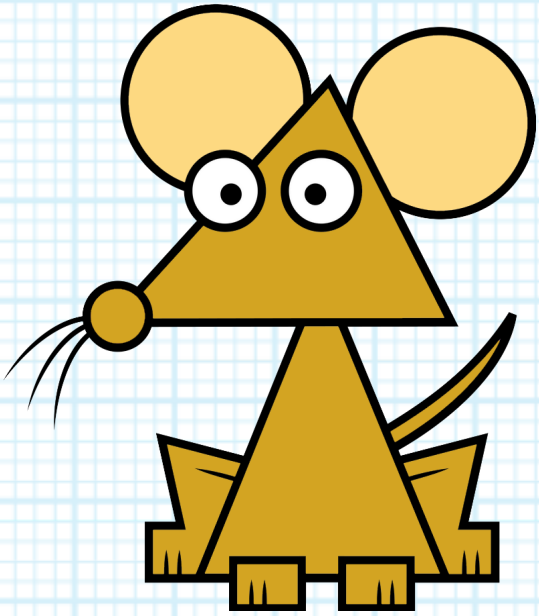


手写Tomcat

自己动手，从零开发一个迷你版Tomcat服务器！

廖雪峰

2025-06-16

https://liaoxuefeng.com/books/jerrymouse/

目录

简介

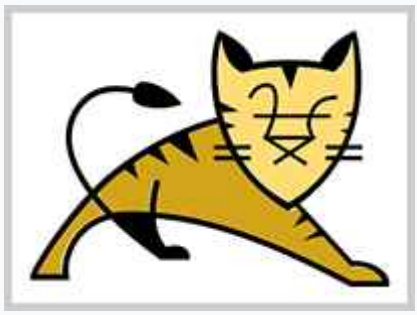
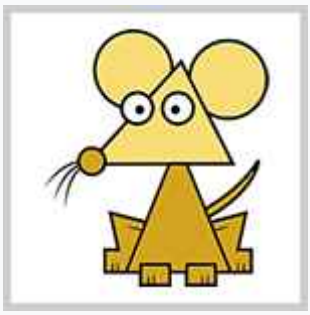
原文链接

对于Java后端开发的同学来说，Tomcat服务器肯定不陌生。开发Java Web App，最后通常都会部署到Tomcat这样的服务器上。

很多同学可能觉得开发Web App是比较容易的，开发Web服务器就比较困难了。实际上，虽然开发Web服务器的难度比开发Web App要高，但也不是高得特别离谱。

对于已经能熟练开发Web App的同学来说，要进一步提升自己的架构水平，不如自己动手，从零开始编写一个Tomcat服务器。

本教程的目标就是以Tomcat服务器为原型，专注于实现一个支持Servlet标准的Web服务器，即实现一个迷你版的Tomcat Server，我们把它命名为Jerrymouse Server，与Tomcat主要区别在于，它俩的图标有所不同：

Tomcat Server	Jerrymouse Server
	

Jerrymouse Server设计目标如下：

- 支持Servlet 6的大部分功能：
  - 支持Servlet组件；
  - 支持Filter组件；
  - 支持Listener组件；
  - 支持Sesssion（仅限Cookie模式）；
  - 不支持JSP；
  - 不支持async模式与WebSocket；
- 可部署一个标准的Web App；
- 不支持同时部署多个Web App；
- 不支持热部署。

我们会一步一步实现一个完整的Web Server，并在此基础上部署一个完整的Web应用程序。

本教程的所有源码均可从GitHub或Gitee下载。

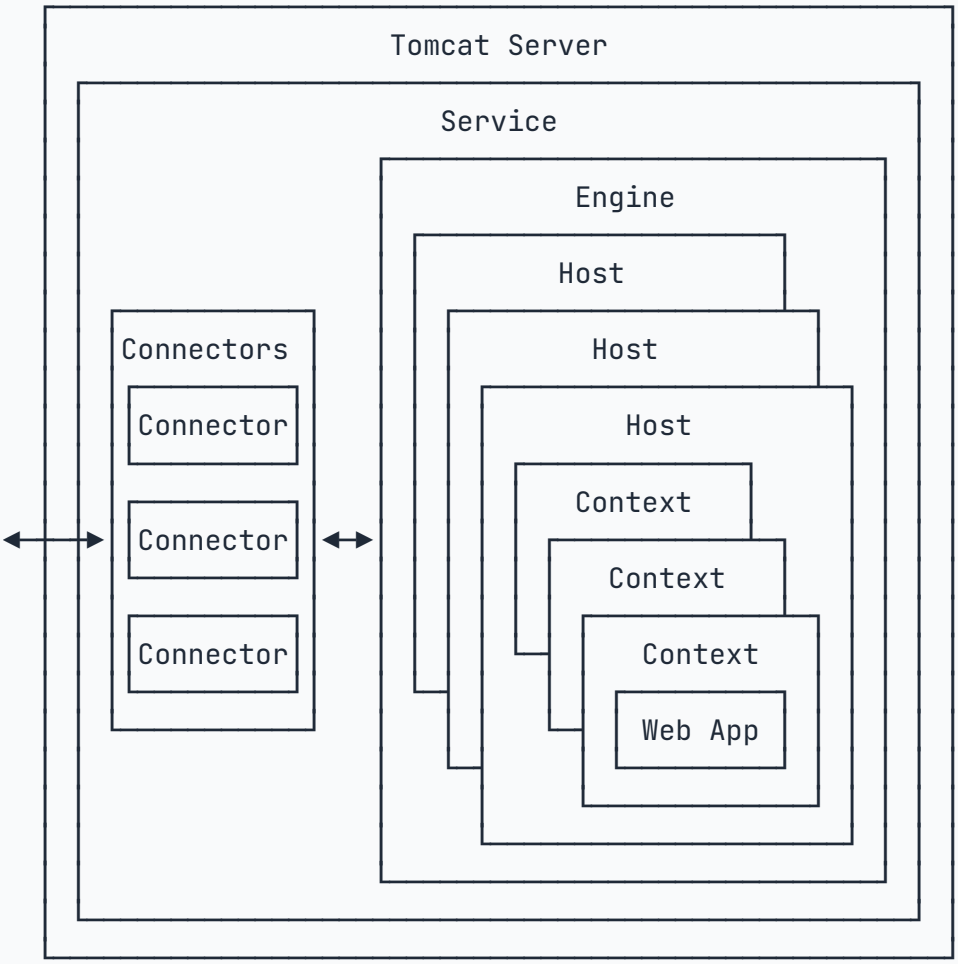
评论

设计服务器架构

原文链接

在开发我们自己的Web服务器之前，我们先看一下Tomcat的架构。

Tomcat是一个开源的Web服务器，它的架构是基于组件的设计，可以将多个组件组合起来使用，用一张图表示如下：



一个Tomcat Server内部可以有多个Service（服务），通常是一个Service。Service内部包含两个组件：

- Connectors：代表一组Connector（连接器），至少定义一个Connector，也允许定义多个Connector，例如，HTTP和HTTPS两个Connector；
- Engine：代表一个引擎，所有HTTP请求经过Connector后传递给Engine。

在一个Engine内部，可以有一个或多个Host（主机），Host可以根据域名区分，在Host内部，又可以有一个或多个Context（上下文），每个Context对应一个Web App。Context是由路径

前缀区分的，如 `/abc` 、 `/xyz` 、 `/` 分别代表3个Web App， `/` 表示的Web App在Tomcat中表示根Web App。

因此，一个HTTP请求：

```
http://www.example.com/abc/hello
```

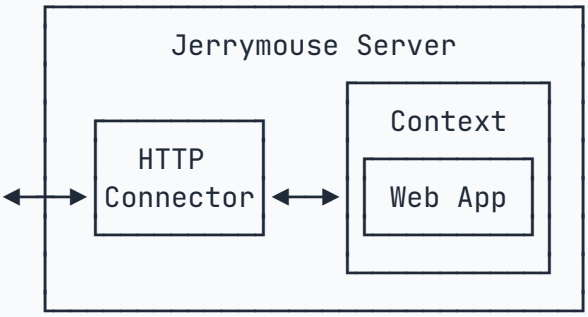
首先根据域名 `www.example.com` 定位到某个Host，然后，根据路径前缀 `/abc` 定位到某个Context，若路径前缀没有匹配到任何Context，则匹配 `/` Context。在Context内部，就是开发者编写的Web App，一个Context仅包含一个Web App。

可见Tomcat Server是一个全功能的Web服务器，它支持HTTP、HTTPS和AJP等多种Connector，又能同时运行多个Host，每个Host内部，还可以挂载一个或多个Context，对应一个或多个Web App。

我们设计Jerrymouse Server就没必要搞这么复杂，可以大幅简化为：

- 仅一个HTTP Connector，不支持HTTPS；
- 仅支持挂载到 `/` 的一个Context，不支持多个Host与多个Context。

因为只有一个Context，所以也只能有一个Web App。Jerrymouse Server的架构如下：

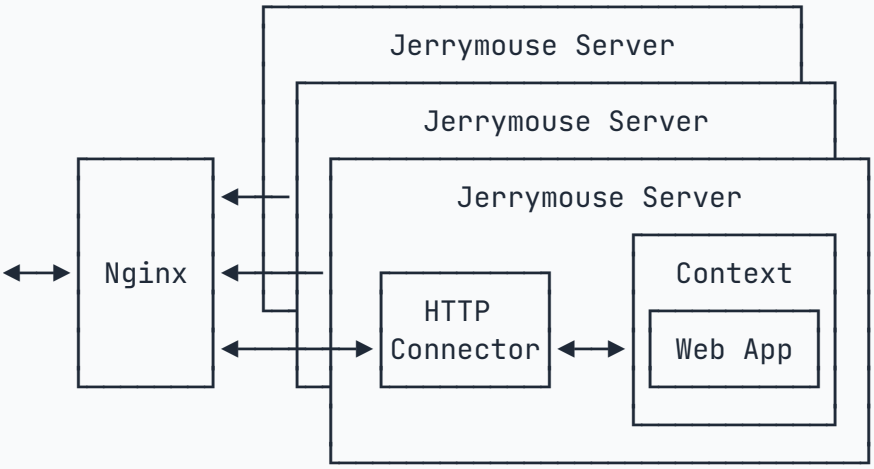


有的同学会担心，如果要运行多个Web App怎么办？

这个问题很容易解决：运行多个Jerrymouse Server就可以运行多个Web App了。

还有的同学会担心，只支持HTTP，如果一定要使用HTTPS怎么办？

HTTPS可以部署在网关，通过网关将HTTPS请求转发为HTTP请求给Jerrymouse Server即可。部署一个Nginx就可以充当网关：



此外，Nginx还可以定义多个Host，根据Host转发给不同的Jerrymouse Server，所以，我们专注于实现一个仅支持HTTP、仅支持一个Web App的Web服务器，把HTTPS、HTTP/2、HTTP/3、Host、Cluster（集群）等功能全部扔给Nginx即可。

评论

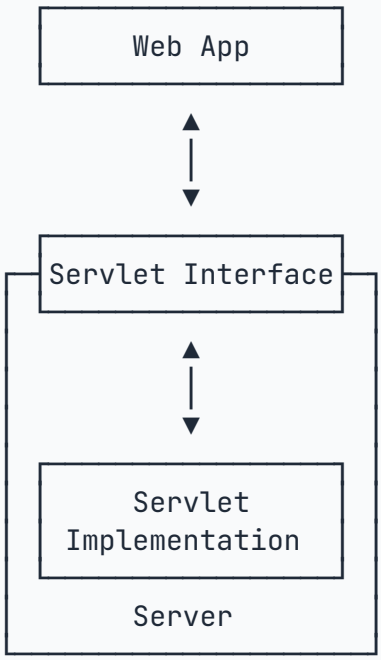
Servlet规范

原文链接

在Java Web应用中，除了Tomcat服务器外，其实还有Jetty、GlassFish、Undertow等多种Web服务器。

一个Java Web App通常打包为 `.war` 文件，并且可以部署到Tomcat、Jetty等多种Web服务器上。为什么一个Java Web App基本上可以无修改地部署到多种Web服务器上呢？原因就在于Servlet规范。

Servlet规范是Java Servlet API的规范，用于定义Web服务器如何处理HTTP请求和响应。Servlet规范有一组接口，对于Web App来说，操作的是接口，而真正对应的实现类，则由各个Web Server实现，这样一来，Java Web App实际上编译的时候仅用到了Servlet规范定义的接口，只要每个Web服务器在实现Servlet接口时严格按照规范实现，就可以保证一个Web App可以正常运行在多种Web服务器上：



对于Web应用程序，Servlet规范是非常重要的。Servlet规范有好几个版本，每个版本都有一些新的功能。以下是一些常见版本的新功能：

Servlet 1.0：定义了Servlet组件，一个Servlet组件运行在Servlet容器（Container）中，通过与容器交互，就可以响应一个HTTP请求；

Servlet 2.0：定义了JSP组件，一个JSP页面可以被动态编译为Servlet组件；

Servlet 2.4：定义了Filter（过滤器）组件，可以实现过滤功能；

Servlet 2.5：支持注解，提供了ServletContextListener接口，增加了一些安全性相关的特性；

Servlet 3.0：支持异步处理的Servlet，支持注解配置Servlet和过滤器，增加了SessionCookieConfig接口；

Servlet 3.1：提供了WebSocket的支持，增加了对HTTP请求和响应的流式操作的支持，增加了对HTTP协议的新特性的支持；

Servlet 4.0：支持HTTP/2的新特性，提供了HTTP/2的Server Push等特性；

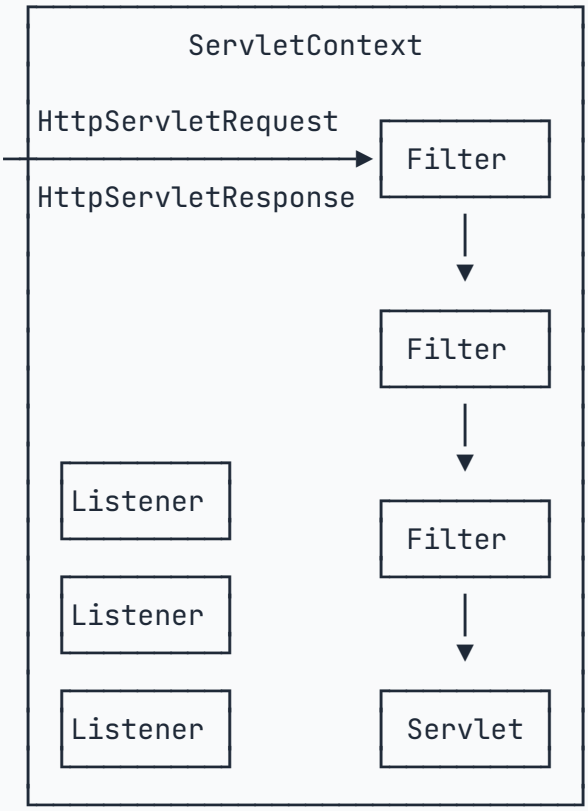
Servlet 5.0：主要是把 `javax.servlet` 包名改成了 `jakarta.servlet` ；

Servlet 6.0：继续增加一些新功能，并废除一部分功能。

目前最新的Servlet版本是6.0，我们开发Jerrymouse Server也是基于最新的Servlet 6.0。

Servlet处理流程

当Servlet容器接收到用户的HTTP请求后，由容器负责把请求转换为 `HttpServletRequest` 和 `HttpServletResponse` 对象，分别代表HTTP请求和响应，然后，经过若干个Filter组件后，到达最终的Servlet组件，由Servlet组件完成HTTP处理，将响应写入 `HttpServletResponse` 对象：



其中，`ServletContext`代表整个容器的信息，如果容器实现了`ServletContext`接口，也可以把`ServletContext`可以看作容器本身。`ServletContext`、`HttpServletRequest`和`HttpServletResponse`都是接口，具体实现由Web服务器完成。`Filter`、`Servlet`组件也是接口，但具体实现由Web App完成。此外，还有一种`Listener`接口，可以监听各种事件，但不直接参与处理HTTP请求，具体实现由Web App完成，何时调用则由容器决定。因此，针对Web App的三大组件：`Servlet`、`Filter`和`Listener`都是运行在容器中的组件，只有容器才能主动调用它们。（此处略去JSP组件，因为我们不打算支持JSP）

对于Jerrymouse服务器来说，开发服务器就必须实现Servlet容器本身，容器实现`ServletContext`接口，容器内部管理若干个`Servlet`、`Filter`和`Listener`组件。

对每个请求，需要创建`HttpServletRequest`和`HttpServletResponse`实例，查找并匹配合适的一组`Filter`和一个`Servlet`，让它们处理HTTP请求。

在处理过程中，会产生各种事件，容器负责将产生的事件发送到`Listener`组件处理。

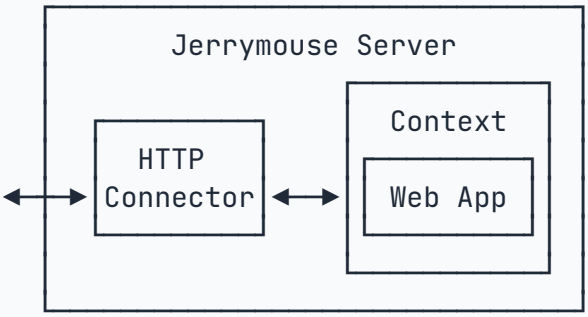
以上就是我们编写Servlet容器按照Servlet规范所必须的全部功能。

评论

实现HTTP服务器

原文链接

我们设计的Jerrymouse Server的架构如下：



在实现Servlet支持之前，我们先实现一个HTTP Connector。

所谓Connector，这里可以简化为一个能处理HTTP请求的服务器，HTTP/1.x协议是基于TCP连接的一个简单的请求-响应协议，首先由浏览器发送请求：

```
GET /hello HTTP/1.1
Host: www.example.com
User-Agent: curl/7.88.1
Accept: */*

```

请求头指出了请求的方法 `GET` ，主机 `www.example.com` ，路径 `/hello` ，接下来服务器解析请求，输出响应：

```
HTTP/1.1 200 OK
Server: Simple HttpServer/1.0
Date: Fri, 07 Jul 2023 23:15:09 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 22
Connection: keep-alive

<h1>Hello, world.</h1>
```

响应返回状态码 `200` ，每个响应头 `Header: Value` ，最后是以 `\r\n\r\n` 分隔的响应内容。

所以我们编写HTTP Server实际上就是：

1. 监听TCP端口，等待浏览器连接；
2. 接受TCP连接后，创建一个线程处理该TCP连接：
    1. 接收浏览器发送的HTTP请求；
    2. 解析HTTP请求；
    3. 处理请求；
    4. 发送HTTP响应；
    5. 重复1～4直到TCP连接关闭。

整个流程不复杂，但是处理步骤比较繁琐，尤其是解析HTTP请求，是个体力活，不但要去读HTTP协议手册，还要做大量的兼容性测试。

所以我们选择直接使用JDK内置的 `jdk.httpserver` 。 `jdk.httpserver` 从JDK 9开始作为一个公开模块可以直接使用，它的包是 `com.sun.net.httpserver` ，主要提供以下几个类：

- HttpServer：通过指定IP地址和端口号，定义一个HTTP服务实例；
- HttpHandler：处理HTTP请求的核心接口，必须实现 `handle(HttpExchange)` 方法；
- HttpExchange：可以读取HTTP请求的输入，并将HTTP响应输出给它。

一个能处理HTTP请求的简单类实现如下：

```
class SimpleHttpHandler implements HttpHandler {
    final Logger logger = LoggerFactory.getLogger(getClass());

    @Override
    public void handle(HttpExchange exchange) throws IOException {
        // 获取请求方法、URI、Path、Query等:
        String method = exchange.getRequestMethod();
        URI uri = exchange.getRequestURI();
        String path = uri.getPath();
        String query = uri.getRawQuery();
        logger.info("{}: {}?{}", method, path, query);
        // 输出响应的Header:
        Headers respHeaders = exchange.getResponseHeaders();
        respHeaders.set("Content-Type", "text/html; charset=utf-8");
        respHeaders.set("Cache-Control", "no-cache");
        // 设置200响应:
        exchange.sendResponseHeaders(200, 0);
        // 输出响应的内容:
        String s = "<h1>Hello, world.</h1><p>" +
```

```
LocalDateTime.now().withNano(0) + "</p>";
        try (OutputStream out = exchange.getResponseBody()) {
            out.write(s.getBytes(StandardCharsets.UTF_8));
        }
    }
}
```

可见，`HttpExchange`封装了HTTP请求和响应，我们不再需要解析原始的HTTP请求，也无需构造原始的HTTP响应，而是通过`HttpExchange`间接操作，大大简化了HTTP请求的处理。

最后写一个`SimpleHttpServer`把启动`HttpServer`、处理请求连起来：

```
public class SimpleHttpServer implements HttpHandler, AutoCloseable {
    final Logger logger = LoggerFactory.getLogger(getClass());

    public static void main(String[] args) {
        String host = "0.0.0.0";
        int port = 8080;
        try (SimpleHttpServer connector = new SimpleHttpServer(host, port))
{
            for (;;) {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    break;
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    final HttpServer httpServer;
    final String host;
    final int port;

    public SimpleHttpServer(String host, int port) throws IOException {
        this.host = host;
        this.port = port;
        this.httpServer = HttpServer.create(new
InetSocketAddress("0.0.0.0", 8080), 0, "/", this);
```

```
        this.httpServer.start();
        logger.info("start jerrymouse http server at {}:{}", host, port);
    }

    @Override
    public void close() {
        this.httpServer.stop(3);
    }

    @Override
    public void handle(HttpExchange exchange) throws IOException {
        ...
    }
}
```

运行后打开浏览器，访问 `http://localhost:8080` ，可以看到如下输出：

这样，我们就成功实现了一个简单的HTTP Server。

参考源码

可以从GitHub或Gitee下载源码。

GitHub

评论

实现Servlet服务器

原文链接

在上一节，我们已经成功实现了一个简单的HTTP服务器，但是，好像和Servlet没啥关系，因为整个操作都是基于 `HttpExchange` 接口做的。

而Servlet处理HTTP的接口是基于 `HttpServletRequest` 和 `HttpServletResponse` ，前者负责读取HTTP请求，后者负责写入HTTP响应。

怎么把基于 `HttpExchange` 的操作转换为基于 `HttpServletRequest` 和 `HttpServletResponse` ？答案是使用Adapter模式。

首先我们定义 `HttpExchangeAdapter` ，它持有一个 `HttpExchange` 实例，并实现 `HttpExchangeRequest` 和 `HttpExchangeResponse` 接口：

```
interface HttpExchangeRequest {
    String getRequestMethod();
    URI getRequestURI();
}

interface HttpExchangeResponse {
    Headers getResponseHeaders();
    void sendResponseHeaders(int rCode, long responseLength) throws
IOException;
    OutputStream getResponseBody();
}

public class HttpExchangeAdapter implements HttpExchangeRequest,
HttpExchangeResponse {
    final HttpExchange exchange;

    public HttpExchangeAdapter(HttpExchange exchange) {
        this.exchange = exchange;
    }

    // 实现方法
    ...
}
```

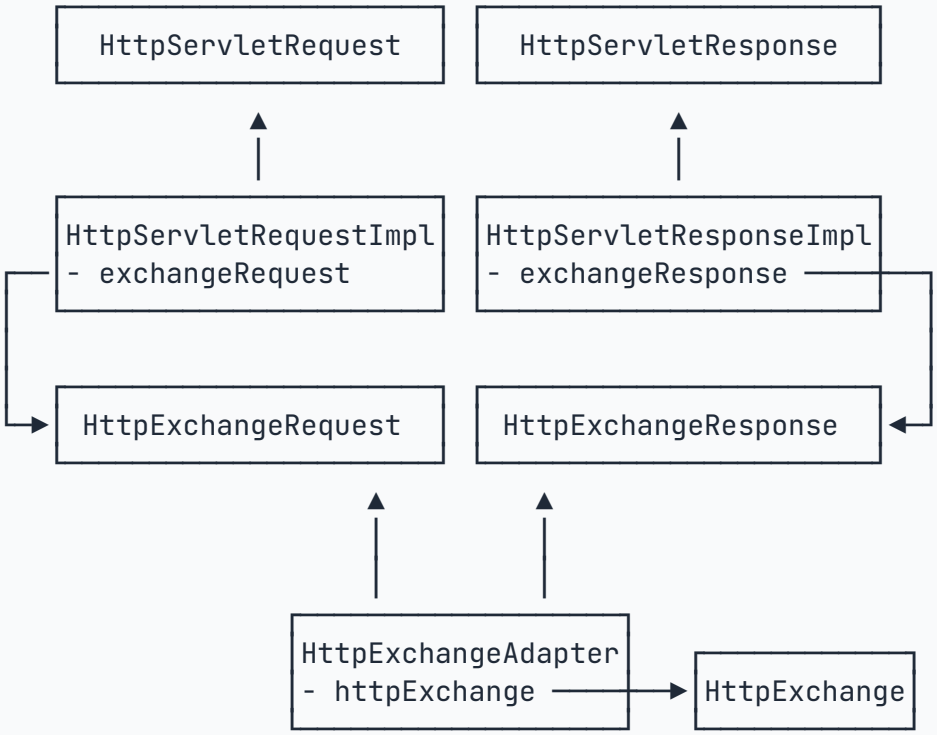
紧接着我们编写 `HttpServletRequestImpl`，它内部持有 `HttpServletRequest`，并实现了 `HttpServletRequest` 接口：

```
public class HttpServletRequestImpl implements HttpServletRequest {
    final HttpExchangeRequest exchangeRequest;

    public HttpServletRequestImpl(HttpExchangeRequest exchangeRequest) {
        this.exchangeRequest = exchangeRequest;
    }

    // 实现方法
    ...
}
```

同理，编写 `HttpServletResponseImpl` 如下：

```
public class HttpServletResponseImpl implements HttpServletResponse {
    final HttpExchangeResponse exchangeResponse;

    public HttpServletResponseImpl(HttpExchangeResponse exchangeResponse) {
        this.exchangeResponse = exchangeResponse;
    }

    // 实现方法
    ...
}
```

用一个图表示从 `HttpExchange` 转换为 `HttpServletRequest` 和 `HttpServletResponse` 如下：



接下来我们改造处理HTTP请求的 `HttpHandler` 接口：

```
public class HttpConnector implements HttpHandler {
    @Override
    public void handle(HttpExchange exchange) throws IOException {
        var adapter = new HttpExchangeAdapter(exchange);
        var request = new HttpServletRequestImpl(adapter);
        var response = new HttpServletResponseImpl(adapter);
        process(request, response);
    }

    void process(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
        // TODO
    }
}
```

在 `handle(HttpExchange)` 方法内部，我们创建的对象如下：

1. HttpExchangeAdapter实例：它内部引用了jdk.httpserver提供的HttpExchange实例；

2. HttpServletRequestImpl实例：它内部引用了HttpExchangeAdapter实例，但是转换为HttpExchangeRequest接口；

3. HttpServletResponseImpl实例：它内部引用了HttpExchangeAdapter实例，但是转换为HttpExchangeResponse接口。

所以实际上创建的实例只有3个。最后调用 `process(HttpServletRequest, HttpServletResponse)` 方法，这个方法内部就可以按照Servlet标准来处理HTTP请求了，因为方法参数是标准的Servlet接口：

```
void process(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    String name = request.getParameter("name");
    String html = "<h1>Hello, " + (name == null ? "world" : name) + ".
</h1>";
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    pw.write(html);
    pw.close();
}
```

目前，我们仅实现了代码调用时用到的 `getParameter()` 、 `setContentType()` 和 `getWriter()` 这几个方法。如果补全 `HttpServletRequest` 和 `HttpServletResponse` 接口所有的方法定义，我们就得到了完整的 `HttpServletRequest` 和 `HttpServletResponse` 接口实现。

运行代码，在浏览器输入 `http://localhost:8080/?name=World` ，结果如下：

参考源码

可以从GitHub或Gitee下载源码。

GitHub

小结

为了实现Servlet服务器，我们必须把jdk.httpserver提供的输入输出 `HttpExchange` 转换为Servlet标准定义的 `HttpServletRequest` 和 `HttpServletResponse` 接口，转换方式是Adapter模式；

转换后的 `HttpExchangeAdapter` 类再用 `HttpExchangeRequest` 和 `HttpExchangeResponse` 把读取和写入功能分开，使得结构更加清晰。

评论

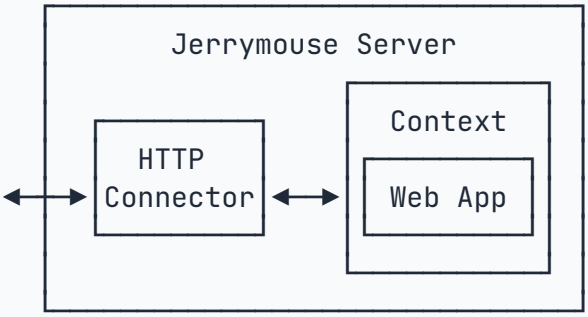
实现Servlet组件

原文链接

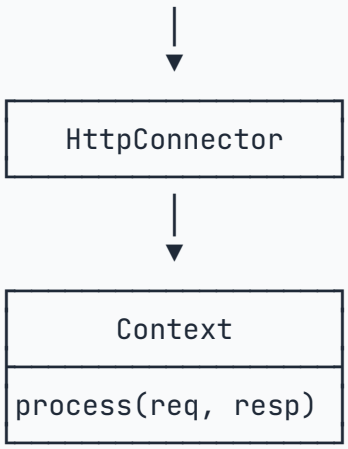
现在，我们已经成功实现了一个 `HttpConnector` ，并且，将jdk.httpserver提供的输入输出 `HttpExchange` 转换为Servlet标准定义的 `HttpServletRequest` 和 `HttpServletResponse` 接口，最终处理方法如下：

```
void process(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    // TODO
}
```

这样，我们就有了处理 `HttpServletRequest` 和 `HttpServletResponse` 的入口，回顾一下Jerrymouse设计的架构图：



我们让 `HttpConnector` 持有一个Context实例，在Context定义 `process(req, resp)` 方法：



这个Context组件本质上可以视为Servlet规范定义的 `ServletContext` ，而规范定义的Servlet、Filter、Listener等组件，就可以让 `ServletContext` 管理，后续的服务器设计就简化为如何实现 `ServletContext` ，以及如何管理Servlet、Filter、Listener等组件。

评论

实现ServletContext

原文链接

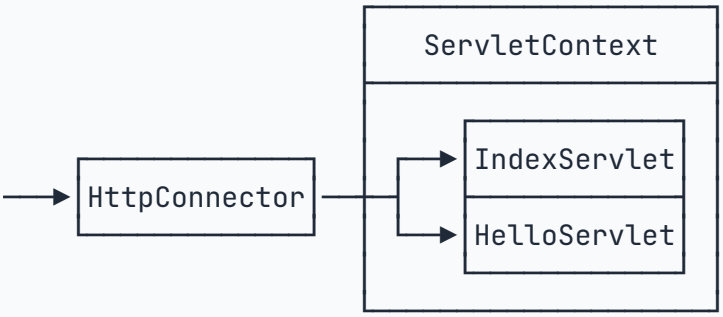
在Java Web应用程序中，`ServletContext`代表应用程序的运行环境，一个Web应用程序对应一个唯一的`ServletContext`实例，`ServletContext`可以用于：

- 提供初始化和全局配置：可以从`ServletContext`获取Web App配置的初始化参数、资源路径等信息；
- 共享全局数据：`ServletContext`存储的数据可以被整个Web App的所有组件读写。

既然`ServletContext`是一个Web App的全局唯一实例，而Web App又运行在Servlet容器中，我们在实现`ServletContext`时，完全可以把它当作Servlet容器来实现，它在内部维护一组Servlet实例，并根据Servlet配置的路由信息将请求转发给对应的Servlet处理。假设我们编写了两个Servlet：

- IndexServlet：映射路径为`/`；
- HelloServlet：映射路径为`/hello`。

那么，处理HTTP请求的路径如下：



下面，我们来实现`ServletContext`。首先定义`ServletMapping`，它包含一个Servlet实例，以及将映射路径编译为正则表达式：

```
public class ServletMapping {
    final Pattern pattern; // 编译后的正则表达式
    final Servlet servlet; // Servlet实例
    public ServletMapping(String urlPattern, Servlet servlet) {
        this.pattern = buildPattern(urlPattern); // 编译为正则表达式
        this.servlet = servlet;
```

```
    }
}
```

接下来实现 `ServletContext` ：

```
public class ServletContextImpl implements ServletContext {
    final List<ServletMapping> servletMappings = new ArrayList<>();
}
```

这个数据结构足够能让我们实现根据请求路径路由到某个特定的Servlet：

```
public class ServletContextImpl implements ServletContext {
    ...
    // HTTP请求处理入口：
    public void process(HttpServletRequest request, HttpServletResponse
response) throws IOException, ServletException {
        // 请求路径：
        String path = request.getRequestURI();
        // 搜索Servlet：
        Servlet servlet = null;
        for (ServletMapping mapping : this.servletMappings) {
            if (mapping.matches(path)) {
                // 路径匹配：
                servlet = mapping.servlet;
                break;
            }
        }
        if (servlet == null) {
            // 未匹配到任何Servlet显示404 Not Found：
            PrintWriter pw = response.getWriter();
            pw.write("<h1>404 Not Found</h1><p>No mapping for URL: " + path
+ "</p>");
            pw.close();
            return;
        }
        // 由Servlet继续处理请求：
        servlet.service(request, response);
    }
}
```

这样我们就实现了 `ServletContext` ！

不过，细心的同学会发现，我们编写的两个Servlet：`IndexServlet` 和 `HelloServlet` ，还没有被添加到 `ServletContext` 中。那么问题来了：Servlet在什么时候被初始化？

答案是在创建 `ServletContext` 实例后，就立刻初始化所有的Servlet。我们编写一个 `initialize()` 方法，用于初始化Servlet：

```
public class ServletContextImpl implements ServletContext {
    Map<String, ServletRegistrationImpl> servletRegistrations = new
HashMap<>();
    Map<String, Servlet> nameToServlets = new HashMap<>();
    List<ServletMapping> servletMappings = new ArrayList<>();

    public void initialize(List<Class<?>> servletClasses) {
        // 依次添加每个Servlet:
        for (Class<?> c : servletClasses) {
            // 获取@WebServlet注解:
            WebServlet ws = c.getAnnotation(WebServlet.class);
            Class<? extends Servlet> clazz = (Class<? extends Servlet>) c;
            // 创建一个ServletRegistration.Dynamic:
            ServletRegistration.Dynamic registration =
this.addServlet(AnnoUtils.getServletName(clazz), clazz);

registration.addMapping(AnnoUtils.getServletUrlPatterns(clazz));

registration.setInitParameters(AnnoUtils.getServletInitParams(clazz));
        }
        // 实例化Servlet:
        for (String name : this.servletRegistrations.keySet()) {
            var registration = this.servletRegistrations.get(name);
            registration.servlet.init(registration.getServletConfig());
            this.nameToServlets.put(name, registration.servlet);
            for (String urlPattern : registration.getMappings()) {
                this.servletMappings.add(new ServletMapping(urlPattern,
registration.servlet));
            }
            registration.initialized = true;
        }
    }
```

```
    @Override
    public ServletRegistration.Dynamic addServlet(String name, Servlet
servlet) {
        var registration = new ServletRegistrationImpl(this, name,
servlet);
        this.servletRegistrations.put(name, registration);
        return registration;
    }
}
```

从Servlet 3.0规范开始，我们必须要提供 `addServlet()` 动态添加一个Servlet，并且返回 `ServletRegistration.Dynamic` ，因此，我们在 `initialize()` 方法中调用 `addServlet()` ，完成所有Servlet的创建和初始化。

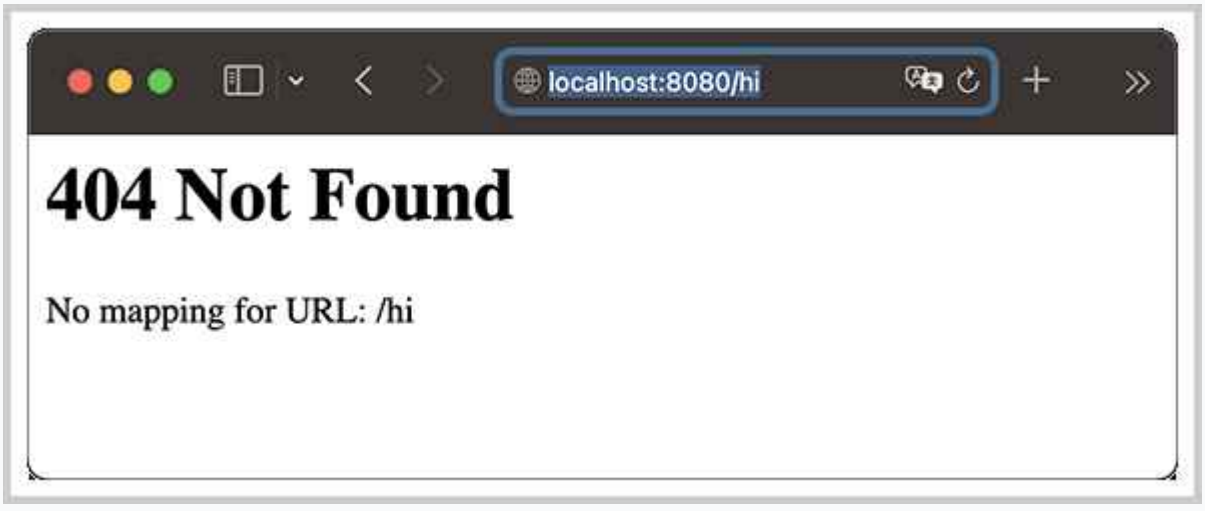
最后我们修改 `HttpConnector` ，实例化 `ServletContextImpl` ：

```
public class HttpConnector implements HttpHandler {
    // 持有ServletContext实例:
    final ServletContextImpl servletContext;
    final HttpServer httpServer;

    public HttpConnector() throws IOException {
        // 创建ServletContext:
        this.servletContext = new ServletContextImpl();
        // 初始化Servlet:
        this.servletContext.initialize(List.of(IndexServlet.class,
HelloServlet.class));
        ...
    }

    @Override
    public void handle(HttpExchange exchange) throws IOException {
        var adapter = new HttpExchangeAdapter(exchange);
        var request = new HttpServletRequestImpl(adapter);
        var response = new HttpServletResponseImpl(adapter);
        // process:
        this.servletContext.process(request, response);
    }
}
```

运行服务器，输入 `http://localhost:8080/` ，查看 `IndexServlet` 的输出：

输入 `http://localhost:8080/hello?name=Bob` ，查看 `HelloServlet` 的输出：

输入错误的路径，查看404输出：



可见，我们已经成功完成了 `ServletContext` 和所有Servlet的管理，并实现了正确的路由。

有的同学会问：Servlet本身应该是Web App开发人员实现，而不是由服务器实现。我们在服务器中却写死了两个Servlet，这显然是不合理的。正确的方式是从外部war包加载Servlet，但是这个问题我们放到后面解决。

参考源码

可以从GitHub或Gitee下载源码。

GitHub

小结

编写Servlet容器时，直接实现 `ServletContext` 接口，并在内部完成所有Servlet的管理，就可以实现根据路径路由到匹配的Servlet。

评论

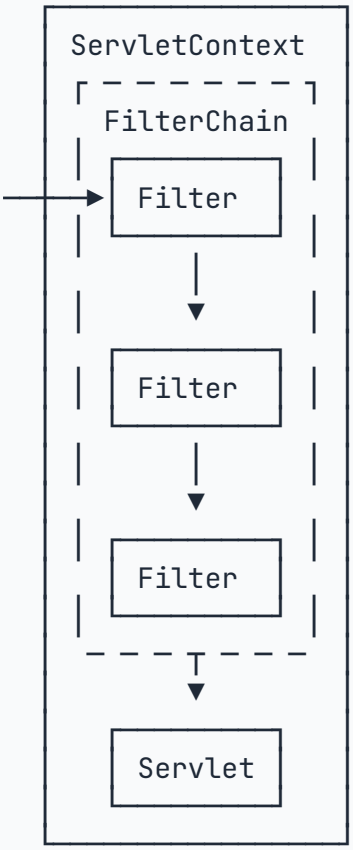
实现FilterChain

原文链接

上一节我们实现了 `ServletContext` ，并且能够管理所有的Servlet组件。本节我们继续增加对Filter组件的支持。

Filter是Servlet规范中的一个重要组件，它的作用是在HTTP请求到达Servlet之前进行预处理。它可以被一个或多个Filter按照一定的顺序组成一个处理链（FilterChain），用来处理一些公共逻辑，比如打印日志、登录检查等。

Filter还可以有针对性地拦截或者放行HTTP请求，本质上一个 `FilterChain` 就是一个责任链模式。在Servlet容器中，处理流程如下：



这里有几点需要注意：

1. 最终处理请求的Servlet是根据请求路径选择的；
2. Filter链上的Filter是根据请求路径匹配的，可能匹配0个或多个Filter；
3. 匹配的Filter将组成FilterChain进行调用。

下面，我们首先将 `Filter` 纳入 `ServletContext` 中管理。和 `ServletMapping` 类似，先定义 `FilterMapping` ，它包含一个 `Filter` 实例，以及将映射路径编译为正则表达式：

```
public class FilterMapping {
    final Pattern pattern; // 编译后的正则表达式
    final Filter filter;

    public FilterMapping(String urlPattern, Filter filter) {
        this.pattern = buildPattern(urlPattern); // 编译为正则表达式
        this.filter = filter;
    }
}
```

接着，根据Servlet规范，我们需要提供 `addFilter()` 动态添加一个 `Filter` ，并且返回 `FilterRegistration.Dynamic` ，所以需要在 `ServletContext` 中实现相关方法：

```
public class ServletContextImpl implements ServletContext {
    Map<String, FilterRegistrationImpl> filterRegistrations = new HashMap<>
();
    Map<String, Filter> nameToFilters = new HashMap<>();
    List<FilterMapping> filterMappings = new ArrayList<>();

    // 根据Class Name添加Filter:
    @Override
    public FilterRegistration.Dynamic addFilter(String name, String
className) {
        return addFilter(name, Class.forName(className));
    }

    // 根据Class添加Filter:
    @Override
    public FilterRegistration.Dynamic addFilter(String name, Class<?
extends Filter> clazz) {
        return addFilter(name, clazz.newInstance());
    }

    // 根据Filter实例添加Filter:
    @Override
    public FilterRegistration.Dynamic addFilter(String name, Filter filter)
{
```

```
        var registration = new FilterRegistrationImpl(this, name, filter);
        this.filterRegistrations.put(name, registration);
        return registration;
    }
    ...
}
```

再添加一个 `initFilters()` 方法用于向容器添加 `Filter` ：

```
public class ServletContextImpl implements ServletContext {
    ...
    public void initFilters(List<Class<?>> filterClasses) {
        for (Class<?> c : filterClasses) {
            // 获取@WebFilter注解:
            WebFilter wf = c.getAnnotation(WebFilter.class);
            // 添加Filter:
            FilterRegistration.Dynamic registration =
this.addFilter(AnnoUtils.getFilterName(clazz), clazz);
            // 添加URL映射:

registration.addMappingForUrlPatterns(EnumSet.of(DispatcherType.REQUEST),
true, AnnoUtils.getFilterUrlPatterns(clazz));
            // 设置初始化参数:

registration.setInitParameters(AnnoUtils.getFilterInitParams(clazz));
        }
        for (String name : this.filterRegistrations.keySet()) {
            // 依次处理每个FilterRegistration.Dynamic:
            var registration = this.filterRegistrations.get(name);
            // 调用Filter.init()方法:
            registration.filter.init(registration.getFilterConfig());
            this.nameToFilters.put(name, registration.filter);
            // 将Filter定义的每个URL映射编译为正则表达式:
            for (String urlPattern : registration.getUrlPatternMappings())
{
                this.filterMappings.add(new FilterMapping(urlPattern,
registration.filter));
            }
        }
    }
```

```
    ...
}
```

这样，我们就完成了对Filter组件的管理。

下一步，是改造 `process()` 方法，把原来直接把请求扔给 `Servlet` 处理，改成先匹配 `Filter` ，处理后再扔给最终的 `Servlet` ：

```
public class ServletContextImpl implements ServletContext {
    ...
    public void process(HttpServletRequest request, HttpServletResponse
response) throws IOException, ServletException {
        // 获取请求路径:
        String path = request.getRequestURI();
        // 查找Servlet:
        Servlet servlet = null;
        for (ServletMapping mapping : this.servletMappings) {
            if (mapping.matches(path)) {
                servlet = mapping.servlet;
                break;
            }
        }
        if (servlet == null) {
            // 404错误:
            PrintWriter pw = response.getWriter();
            pw.write("<h1>404 Not Found</h1><p>No mapping for URL: " + path
+ "</p>");
            pw.close();
            return;
        }
        // 查找Filter:
        List<Filter> enabledFilters = new ArrayList<>();
        for (FilterMapping mapping : this.filterMappings) {
            if (mapping.matches(path)) {
                enabledFilters.add(mapping.filter);
            }
        }
        Filter[] filters = enabledFilters.toArray(Filter[]::new);
        // 构造FilterChain实例:
        FilterChain chain = new FilterChainImpl(filters, servlet);
        // 由FilterChain处理:
```

```
        chain.doFilter(request, response);
    }
    ...
}
```

注意上述 `FilterChain` 不仅包含一个 `Filter[]` 数组，还包含一个 `Servlet`，这样我们调用 `chain.doFilter()` 时，在 `FilterChain` 中最后一个处理请求的就是 `Servlet`，这样设计可以简化我们实现 `FilterChain` 的代码：

```
public class FilterChainImpl implements FilterChain {
    final Filter[] filters;
    final Servlet servlet;
    final int total; // Filter总数量
    int index = 0; // 下一个要处理的Filter[index]

    public FilterChainImpl(Filter[] filters, Servlet servlet) {
        this.filters = filters;
        this.servlet = servlet;
        this.total = filters.length;
    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response)
throws IOException, ServletException {
        if (index < total) {
            int current = index;
            index++;
            // 调用下一个Filter处理:
            filters[current].doFilter(request, response, this);
        } else {
            // 调用Servlet处理:
            servlet.service(request, response);
        }
    }
}
```

注意 `FilterChain` 是一个递归调用，因为在执行 `Filter.doFilter()` 时，需要把 `FilterChain` 自身传进去，在执行 `Filter.doFilter()` 之前，就要把 `index` 调整到正确的值。

我们编写两个测试用的Filter：

- LogFilter：匹配 `/*` ，打印请求方法、路径等信息；
- HelloFilter：匹配 `/hello` ，根据请求参数决定放行还是返回403错误。

在初始化ServletContextImpl时将Filter加进去，先测试 `http://localhost:8080/` ：

观察后台输出， `LogFilter` 应该起作用：

```
16:48:00.304 [HTTP-Dispatcher] INFO  c.i.j.engine.filter.LogFilter -- GET:
/
```

再测试 `http://localhost:8080/hello?name=Bob` ：

观察后台输出， `HelloFilter` 和 `LogFilter` 应该起作用：

```
16:49:31.409 [HTTP-Dispatcher] INFO  c.i.j.engine.filter.HelloFilter --
Check parameter name = Bob
```

```
16:49:31.409 [HTTP-Dispatcher] INFO  c.i.j.engine.filter.LogFilter -- GET: /hello
```

最后测试 `http://localhost:8080/hello?name=Jim` ：

可以看到， `HelloFilter` 拦截了请求，返回403错误，最终的 `HelloServlet` 并没有处理该请求。

现在，我们就成功地在 `ServletContext` 中实现了对 `Filter` 的管理，以及根据每个请求，构造对应的 `FilterChain` 来处理请求。目前还有几个小问题：

一是和Servlet一样，Filter本身应该是Web App开发人员实现，而不是由服务器实现。我们在在服务器中写死了两个Filter，这个问题后续解决；

二是Servlet规范并没有规定多个Filter应该如何排序，我们在实现时也没有对Filter进行排序。如果要按固定顺序给Filter排序，从Servlet规范来说怎么排序都可以，通常是按 `@WebFilter` 定义的 `filterName` 进行排序，Spring Boot提供的一个 `FilterRegistrationBean` 允许开发人员自己定义Filter的顺序。

参考源码

可以从GitHub或Gitee下载源码。

GitHub

小结

实现 `FilterChain` 时，要首先在 `ServletContext` 内完成所有Filter的初始化和映射，然后，根据请求路径匹配所有合适的Filter和唯一的Servlet，构造 `FilterChain` 并处理请求。

评论

实现HttpSession

原文链接

HttpSession是Java Web App的一种机制，用于在客户端和服务器之间维护会话状态信息。

Session原理

当客户端第一次请求Web应用程序时，服务器会为该客户端创建一个唯一的Session ID，该ID本质上是一个随机字符串，然后，将该ID存储在客户端的一个名为 `JSESSIONID` 的Cookie中。与此同时，服务器会在内存中创建一个 `HttpSession` 对象，与Session ID关联，用于存储与该客户端相关的状态信息。

当客户端发送后续请求时，服务器根据客户端发送的名为 `JSESSIONID` 的Cookie中获得Session ID，然后查找对应的 `HttpSession` 对象，并从中读取或继续写入状态信息。

Session用途

Session主要用于维护一个客户端的会话状态。通常，用户成功登录后，可以通过如下代码创建一个新的 `HttpSession` ，并将用户ID、用户名等信息放入 `HttpSession` ：

```
@WebServlet(urlPatterns = "/login")
public class LoginServlet extends HttpServlet {
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        String username = req.getParameter("username");
        String password = req.getParameter("password");
        if (loginOk(username, password)) {
            // 登录成功，获取Session:
            HttpSession session = req.getSession();
            // 将用户名放入Session:
            session.setAttribute("username", username);
            // 返回首页:
            resp.sendRedirect("/");
        } else {
            // 登录失败:
            resp.sendRedirect("/error");
```

```
        }
    }
}
```

在其他页面，可以随时获取 `HttpSession` 并取出用户信息，然后在页面展示给用户：

```
@WebServlet(urlPatterns = "/")
public class IndexServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        // 获取Session:
        HttpSession session = req.getSession();
        // 从Session中取出用户名:
        String username = (String) session.getAttribute("username");
        if (username == null) {
            // 未获取到用户名，说明未登录:
            resp.sendRedirect("/login");
        } else {
            // 获取到用户名，说明已登录:
            String html = "<p>Welcome, " + username + "!</p>";
            resp.setContentType("text/html");
            PrintWriter pw = resp.getWriter();
            pw.write(html);
            pw.close();
        }
    }
}
```

当用户登出时，需要调用 `HttpSession` 的 `invalidate()` 方法，让会话失效，这样，用户将重新回到未登录状态，因为后续调用 `req.getSession()` 将返回一个新的 `HttpSession` ，从这个新的 `HttpSession` 取出的 `username` 将是 `null` 。

HttpSession的生命周期

第一次调用 `req.getSession()` 时，服务器会为该客户端创建一个新的 `HttpSession` 对象；

后续调用 `req.getSession()` 时，服务器会返回与之关联的 `HttpSession` 对象；

调用 `req.getSession().invalidate()` 时，服务器会销毁该客户端对应的 `HttpSession` 对象；

当客户端一段时间内没有新的请求，服务器会根据Session超时自动销毁超时的 `HttpSession` 对象。

HttpSession接口

`HttpSession` 是一个接口，Java的Web应用调用 `HttpServletRequest` 的 `getSession()` 方法时，需要返回一个 `HttpSession` 的实现类。

了解了以上关于 `HttpSession` 的相关规范后，我们就可以开始实现对 `HttpSession` 的支持。

首先，我们需要一个 `SessionManager` ，用来管理所有的Session：

```
public class SessionManager {
    // 引用ServletContext:
    ServletContextImpl servletContext;
    // 持有SessionID -> Session:
    Map<String, HttpSessionImpl> sessions = new ConcurrentHashMap<>();
    // Session默认过期时间(秒):
    int inactiveInterval;

    // 根据SessionID获取一个Session:
    public HttpSession getSession(String sessionId) {
        HttpSessionImpl session = sessions.get(sessionId);
        if (session == null) {
            // Session未找到，创建一个新的Session:
            session = new HttpSessionImpl(this.servletContext, sessionId,
inactiveInterval);
            sessions.put(sessionId, session);
        } else {
            // Session已存在，更新最后访问时间:
            session.lastAccessedTime = System.currentTimeMillis();
        }
        return session;
    }

    // 删除Session:
    public void remove(HttpSession session) {
        this.sessions.remove(session.getId());
    }
}
```

`SessionManager` 由 `ServletContextImpl` 持有唯一实例。

再编写一个 `HttpSession` 的实现类 `HttpSessionImpl` ：

```
public class HttpSessionImpl implements HttpSession {

    ServletContextImpl servletContext; // ServletContext
    String sessionId; // SessionID
    int maxInactiveInterval; // 过期时间(s)
    long creationTime; // 创建时间(ms)
    long lastAccessedTime; // 最后一次访问时间(ms)
    Attributes attributes; // getAttribute/setAttribute
}
```

然后，我们分析一下用户调用Session的代码：

```
HttpSession session = request.getSession();
session.invalidate();
```

由于 `HttpSession` 是从 `HttpServletRequest` 获得的，因此，必须在 `HttpServletRequestImpl` 中引用 `ServletContextImpl` ，才能访问 `SessionManager` ：

```
public class HttpServletRequestImpl implements HttpServletRequest {
    // 引用ServletContextImpl:
    ServletContextImpl servletContext;
    // 引用HttpServletResponse:
    HttpServletResponse response;

    @Override
    public HttpSession getSession(boolean create) {
        String sessionId = null;
        // 获取所有Cookie:
        Cookie[] cookies = getCookies();
        if (cookies != null) {
            // 查找JSESSIONID:
            for (Cookie cookie : cookies) {
                if ("JSESSIONID".equals(cookie.getName())) {
                    // 拿到Session ID:
                    sessionId = cookie.getValue();
                    break;
```

```
                }
            }
        }
        // 未获取到SessionID，且create=false，返回null:
        if (sessionId == null && !create) {
            return null;
        }
        // 未获取到SessionID，但create=true，创建新的Session:
        if (sessionId == null) {
            // 如果Header已经发送，则无法创建Session，因为无法添加Cookie:
            if (this.response.isCommitted()) {
                throw new IllegalStateException("Cannot create session for
response is commited.");
            }
            // 创建随机字符串作为SessionID:
            sessionId = UUID.randomUUID().toString();
            // 构造一个名为JSESSIONID的Cookie:
            String cookieValue = "JSESSIONID=" + sessionId + "; Path=/;
SameSite=Strict; HttpOnly";
            // 添加到HttpServletResponse的Header:
            this.response.addHeader("Set-Cookie", cookieValue);
        }
        // 返回一个Session对象:
        return this.servletContext.sessionManager.getSession(sessionId);
    }

    @Override
    public HttpSession getSession() {
        return getSession(true);
    }
    ...
}
```

对 `HttpServletRequestImpl` 的改造主要是加入了 `ServletContextImpl` 和 `HttpServletResponse` 的引用：可以通过前者访问到 `SessionManager` ，而创建的新的SessionID需要通过后者把Cookie发送到客户端，因此，在 `HttpConnector` 中，做相应的修改如下：

```
public class HttpConnector implements HttpHandler {
    ...
```

```
    @Override
    public void handle(HttpExchange exchange) throws IOException {
        var adapter = new HttpExchangeAdapter(exchange);
        var response = new HttpServletResponseImpl(adapter);
        // 创建Request时，需要引用servletContext和response:
        var request = new HttpServletRequestImpl(this.servletContext,
adapter, response);
        // process:
        try {
            this.servletContext.process(request, response);
        } catch (Exception e) {
            logger.error(e.getMessage(), e);
        }
    }
}
```

当用户调用 `session.invalidate()` 时，要让Session失效，就需要从 `SessionManager` 中移除：

```
public class HttpSessionImpl implements HttpSession {
    ...
    @Override
    public void invalidate() {
        // 从SessionManager中移除:
        this.servletContext.sessionManager.remove(this);
        this.sessionId = null;
    }
    ...
}
```

最后，我们还需要实现Session的自动过期。由于我们管理的Session实际上是以 `Map<String, HttpSession>` 存储的，所以，让Session自动过期就是定期扫描所有的Session，然后根据最后一次访问时间将过期的Session自动删除。给 `SessionManager` 加一个 `Runnable` 接口，并启动一个Daemon线程：

```
public class SessionManager implements Runnable {
    ...
    public SessionManager(ServletContextImpl servletContext, int interval)
{
        ...
```

```
        // 启动Daemon线程:
        Thread t = new Thread(this);
        t.setDaemon(true);
        t.start();
    }

    // 扫描线程:
    @Override
    public void run() {
        for (;;) {
            // 每60秒扫描一次:
            try {
                Thread.sleep(60_000L);
            } catch (InterruptedException e) {
                break;
            }
            // 当前时间:
            long now = System.currentTimeMillis();
            // 遍历Session:
            for (String sessionId : sessions.keySet()) {
                HttpSession session = sessions.get(sessionId);
                // 判断是否过期:
                if (session.getLastAccessedTime() +
session.getMaxInactiveInterval() * 1000L < now) {
                    // 删除过期的Session:
                    logger.warn("remove expired session: {}, last access
time: {}", sessionId,
DateUtils.formatDateTimeGMT(session.getLastAccessedTime()));
                    session.invalidate();
                }
            }
        }
    }
```

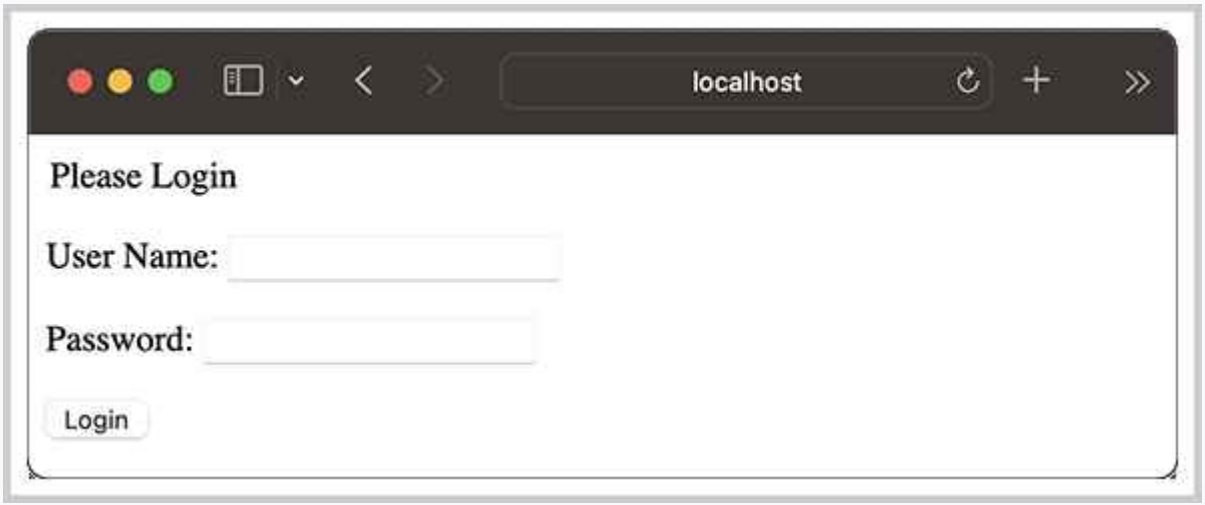
将 `HttpServletRequest` 和 `HttpServletResponse` 与Cookie相关的实现方法补全，我们就得到了一个基于Cookie的 `HttpSession` 实现！

最后需要注意的一点是，和 `HttpServletRequest` 不同，访问 `HttpServletRequest` 实例的一定是一个线程，因此， `HttpServletRequest` 的 `getAttribute()` 和 `setAttribute()` 不需要同步，底层存储用 `HashMap` 即可。但是，访问 `HttpSession` 实例的可能是多线程，所以，

`HttpSession` 的 `getAttribute()` 和 `setAttribute()` 需要实现并发访问，底层存储用 `ConcurrentHashMap` 即可。

测试HttpSession

访问 `IndexServlet` ，第一次访问时，将获取到新的 `HttpSession` ，此时， `HttpSession` 没有用户信息，因此显示登录表单：



登录成功后，可以看到用户名已放入 `HttpSession` ， `IndexServlet` 从 `HttpSession` 获取到用户名后将用户名显示出来：

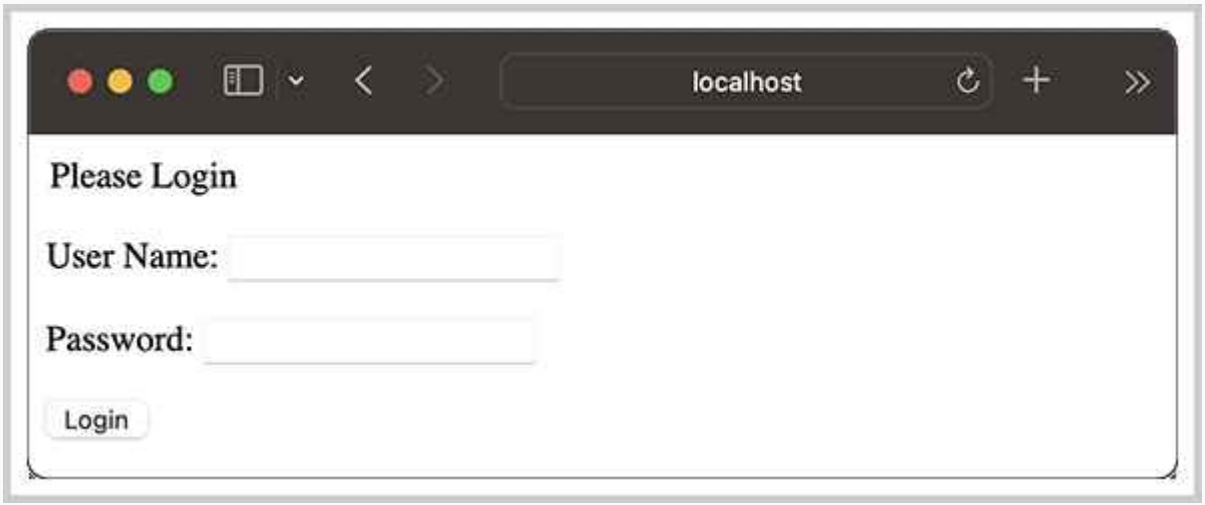
刷新页面， `IndexServlet` 仍将显示登录的用户名，因为根据Cookie拿到相同的SessionID后，获取的 `HttpSession` 是同一个实例。

由于我们设定的 `HttpSession` 过期时间是10分钟，等待至少10分钟，观察控制台输出：

```
21:41:38.001 [HTTP-Dispatcher] INFO  c.i.j.engine.filter.LogFilter -- GET:
/
```

```
21:42:05.586 [HTTP-Dispatcher] INFO  c.i.j.engine.filter.LogFilter -- GET:
/
21:52:15.578 [Thread-0] WARN  c.i.j.engine.SessionManagerImpl -- remove
expired session: 899eb456-5aa3-40d4-8c64-ddc97d39c0d2, last access time:
Fri, 14 Jul 2023 13:42:05 GMT
```

大约在21:52:15时清理了过期的Session，最后一次访问时间是21:42:05（注意时间需要经过时区调整），再次刷新页面将显示登录表单：



参考源码

可以从GitHub或Gitee下载源码。

GitHub

小结

使用Cookie模式实现 `HttpSession` 时，需要实现一个 `HttpSessionManager` ，它在内部维护一个Session ID到 `HttpSession` 实例的映射；

`HttpSessionManager` 通过定期扫描所有 `HttpSession` ，将过期的 `HttpSession` 自动删除，因此，Session自动失效的时间不是特别精确；

由于没有对 `HttpSession` 进行持久化处理，重启服务器后，将丢失所有用户的Session。如果希望重启服务器后保留用户的Session，则需要将Session数据持久化到文件或数据库，此功能要求用户放入 `HttpSession` 的Java对象必须是可序列化的；

因为Session不容易扩展，因此，大规模集群的Web App通常自己管理Cookie来实现登录功能，这样，将用户状态完全保存在浏览器端，不使用Session，服务器就可以做到无状态集群。

评论

实现Listener

原文链接

在Java Web App中，除了Servlet、Filter和HttpSession外，还有一种Listener组件，用于事件监听。

Listener原理

Listener是Java Web App中的一种事件监听机制，用于监听Web应用程序中产生的事件，例如，在 `ServletContext` 初始化完成后，会触发 `contextInitialized` 事件，实现了 `ServletContextListener` 接口的Listener就可以接收到事件通知，可以在内部做一些初始化工作，如加载配置文件，初始化数据库连接池等。实现了 `HttpSessionListener` 接口的Listener可以接收到Session的创建和消耗事件，这样就可以统计在线用户数。

Listener机制是基于观察者模式实现的，即当某个事件发生时，Listener会接收到通知并执行相应的操作。

Listener类型

Servlet规范定义了很多种Listener接口，常用的Listener包括：

- `ServletContextListener`：用于监听 `ServletContext` 的创建和销毁事件；
- `HttpSessionListener`：用于监听 `HttpSession` 的创建和销毁事件；
- `ServletRequestListener`：用于监听 `ServletRequest` 的创建和销毁事件；
- `ServletContextAttributeListener`：用于监听 `ServletContext` 属性的添加、修改和删除事件；
- `HttpSessionAttributeListener`：用于监听 `HttpSession` 属性的添加、修改和删除事件；
- `ServletRequestAttributeListener`：用于监听 `ServletRequest` 属性的添加、修改和删除事件。

下面我们就来实现上述常用的Listener。

首先我们需要在 `ServletContextImpl` 中注册并管理所有的Listener，所以用不同的 `List` 持有注册的Listener：

```
public class ServletContextImpl implements ServletContext {
    ...
    private List<ServletContextListener> servletContextListeners = null;
    private List<ServletContextAttributeListener>
servletContextAttributeListeners = null;
    private List<ServletRequestListener> servletRequestListeners = null;
    private List<ServletRequestAttributeListener>
servletRequestAttributeListeners = null;
    private List<HttpSessionAttributeListener>
httpSessionAttributeListeners = null;
    private List<HttpSessionListener> httpSessionListeners = null;
    ...
}
```

然后，实现 `ServletContext` 的 `addListener()` 接口，用于注册Listener：

```
public class ServletContextImpl implements ServletContext {
    ...
    @Override
    public void addListener(String className) {
        addListener(Class.forName(className));
    }

    @Override
    public void addListener(Class<? extends EventListener> clazz) {
        addListener(clazz.newInstance());
    }

    @Override
    public <T extends EventListener> void addListener(T t) {
        // 根据Listener类型放入不同的List:
        if (t instanceof ServletContextListener listener) {
            if (this.servletContextListeners == null) {
                this.servletContextListeners = new ArrayList<>();
            }
            this.servletContextListeners.add(listener);
        } else if (t instanceof ServletContextAttributeListener listener) {
            if (this.servletContextAttributeListeners == null) {
                this.servletContextAttributeListeners = new ArrayList<>();
            }
```

```
            this.servletContextAttributeListeners.add(listener);
        } else if ...
            ...代码略...
        } else {
            throw new IllegalArgumentException("Unsupported listener: " +
t.getClass().getName());
        }
    }
    ...
}
```

接下来，就是在合适的时机，触发这些Listener。以 `ServletContextAttributeListener` 为例，统一触发的方法放在 `ServletContextImpl` 中：

```
public class ServletContextImpl implements ServletContext {
    ...
    void invokeServletContextAttributeAdded(String name, Object value) {
        logger.info("invoke ServletContextAttributeAdded: {} = {}", name,
value);
        if (this.servletContextAttributeListeners != null) {
            var event = new ServletContextAttributeEvent(this, name,
value);
            for (var listener : this.servletContextAttributeListeners) {
                listener.attributeAdded(event);
            }
        }
    }

    void invokeServletContextAttributeRemoved(String name, Object value) {
        logger.info("invoke ServletContextAttributeRemoved: {} = {}", name,
value);
        if (this.servletContextAttributeListeners != null) {
            var event = new ServletContextAttributeEvent(this, name,
value);
            for (var listener : this.servletContextAttributeListeners) {
                listener.attributeRemoved(event);
            }
        }
    }
```

```
    void invokeServletContextAttributeReplaced(String name, Object value) {
        logger.info("invoke ServletContextAttributeReplaced: {} = {}",
name, value);
        if (this.servletContextAttributeListeners != null) {
            var event = new ServletContextAttributeEvent(this, name,
value);
            for (var listener : this.servletContextAttributeListeners) {
                listener.attributeReplaced(event);
            }
        }
    }
    ...
}
```

当Web App的任何组件调用 `ServletContext` 的 `setAttribute()` 或 `removeAttribute()` 时，就可以触发事件通知：

```
public class ServletContextImpl implements ServletContext {
    ...
    @Override
    public void setAttribute(String name, Object value) {
        if (value == null) {
            removeAttribute(name);
        } else {
            Object old = this.attributes.setAttribute(name, value);
            if (old == null) {
                // 触发attributeAdded:
                this.invokeServletContextAttributeAdded(name, value);
            } else {
                // 触发attributeReplaced:
                this.invokeServletContextAttributeReplaced(name, value);
            }
        }
    }

    @Override
    public void removeAttribute(String name) {
        Object old = this.attributes.removeAttribute(name);
        // 触发attributeRemoved:
        this.invokeServletContextAttributeRemoved(name, old);
```

```
    }
    ...
}
```

其他事件触发也是类似的写法，此处不再重复。

测试Listener

为了测试Listener机制是否生效，我们还需要先编写不同类型的Listener，例如，`HelloHttpSessionAttributeListener` 实现如下：

```
@WebListener
public class HelloHelloHttpSessionAttributeListener implements
HttpSessionAttributeListener {

    final Logger logger = LoggerFactory.getLogger(getClass());

    @Override
    public void attributeAdded(HttpSessionBindingEvent event) {
        logger.info(">>> HttpSession attribute added: {} = {}",
event.getName(), event.getValue());
    }

    @Override
    public void attributeRemoved(HttpSessionBindingEvent event) {
        logger.info(">>> HttpSession attribute removed: {} = {}",
event.getName(), event.getValue());
    }

    @Override
    public void attributeReplaced(HttpSessionBindingEvent event) {
        logger.info(">>> HttpSession attribute replaced: {} = {}",
event.getName(), event.getValue());
    }
}
```

然后在 `HttpConnector` 中注册所有的Listener：

```
List<Class<? extends EventListener>> listenerClasses =
List.of(HelloHttpSessionAttributeListener.class, ...);
```

```
for (Class<? extends EventListener> listenerClass : listenerClasses) {
    this.servletContext.addListener(listenerClass);
}
```

启动服务器，在浏览器中登录或登出，观察日志输出，在每个请求处理前后，可以看到`ServletRequestListener`的创建和销毁事件：

```
08:58:23.944 [HTTP-Dispatcher] INFO  c.i.j.e.l.HelloServletRequestListener
-- >>> ServletRequest initialized: HttpServletRequestImpl@71a49a97[GET:/]
...
08:58:24.008 [HTTP-Dispatcher] INFO  c.i.j.e.l.HelloServletRequestListener
-- >>> ServletRequest destroyed: HttpServletRequestImpl@71a49a97[GET:/]
```

在第一次访问页面和登出时，可以看到`HttpSessionListener`的创建和销毁事件：

```
08:58:23.947 [HTTP-Dispatcher] INFO  c.i.j.e.l.HelloHttpSessionListener --
>>> HttpSession created:
com.itranswarp.jerrymouse.engine.HttpSessionImpl@15037a31
...
08:58:36.766 [HTTP-Dispatcher] INFO  c.i.j.e.l.HelloHttpSessionListener --
>>> HttpSession destroyed:
com.itranswarp.jerrymouse.engine.HttpSessionImpl@15037a31
```

其他事件的触发也可以在日志中找到，这说明我们成功地实现了Servlet规范的Listener机制。

参考源码

可以从GitHub或Gitee下载源码。

GitHub

小结

Servlet规范定义了各种Listener组件，我们支持了其中常用的大部分`EventListener`组件；

Listener组件由`ServletContext`统一管理，并提供统一调度入口方法；

通知机制允许多线程同时调用，如果要防止并发调用Listener的回调方法，需要Listener组件本身在内部做好同步。

评论

加载Web App

原文链接

到目前为止，我们已经实现了 `ServletContext` 容器，支持 `Servlet` 、 `Filter` 和 `Listener` 组件，支持 `HttpSession` ，但是，加载 `Servlet` 、 `Filter` 和 `Listener` 组件时，是写死在服务器里面的 `IndexServlet` 、 `LogFilter` 和 `HelloHttpSessionListener` 这样的类。

而一个正常的Web服务器是从外部加载这些组件的，根据Servlet规范，Web App开发者完成了 `Servlet` 、 `Filter` 和 `Listener` 等组件后，需要按规范把它们打包成 `.war` 文件。 `.war` 文件本质上就是一个jar包，但它的目录组织如下：

```
hello-webapp
├── WEB-INF
│   ├── classes
│   │   └── com
│   │       └── example
│   │           ├── filter
│   │           │   └── LogFilter.class
│   │           ├── listener
│   │           │   ├── HelloHttpSessionListener.class
│   │           │   └── HelloServletContextAttributeListener.class
│   │           ├── servlet
│   │           │   ├── HelloServlet.class
│   │           │   └── IndexServlet.class
│   │           └── util
│   │               └── DateUtil.class
│   └── lib
│       ├── logback-classic-1.4.6.jar
│       ├── logback-core-1.4.6.jar
│       └── slf4j-api-2.0.4.jar
├── contact.html
└── favicon.ico
```

Servlet规范规定，一个 `.war` 包解压后，目录 `/WEB-INF/classes` 存放所有编译后的 `.class` 文件，目录 `/WEB-INF/lib` 存放所有依赖的第三方jar包，其他文件可按任意目录存放。

Web服务器通常会提供一个用于访问文件的Servlet，对于以 `/WEB-INF/` 开头的路径，Web服务器会拒绝访问，其他路径则按正常文件访问，因此，路径 `/contact.html` 可以被访问到，而路径 `/WEB-INF/contact.html` 则不能被访问到。注意这个限制是针对浏览器发出的请求的路径限制，如果在Servlet内部读写 `/WEB-INF/` 目录下的文件则没有任何限制。利用这个限制，很多MVC框架的模版页通常会存放在 `/WEB-INF/templates` 目录下。

以上是关于 `.war` 包的目录规范。我们要把写死的 `Servlet` 、 `Filter` 和 `Listener` 组件从服务器项目中摘出来，单独实现一个 `.war` 包，然后，我们需要实现服务器启动后动态加载 `war` 包，就实现了一个比较完善的Web服务器。

评论

实现ClassLoader

原文链接

要通过Web服务器加载 `war` 包，我们首先要了解JVM的ClassLoader（类加载器）的机制。

在Java中，所有的类，都是由ClassLoader加载到JVM中执行的，但JVM中不止一种ClassLoader。写个简单的程序就可以测试：

```
public class Main {
    public static void main(String[] args) {
        System.out.println(String.class.getClassLoader()); // null
        System.out.println(DataSource.class.getClassLoader()); //
PlatformClassLoader
        System.out.println(Main.class.getClassLoader()); // AppClassLoader
    }
}
```

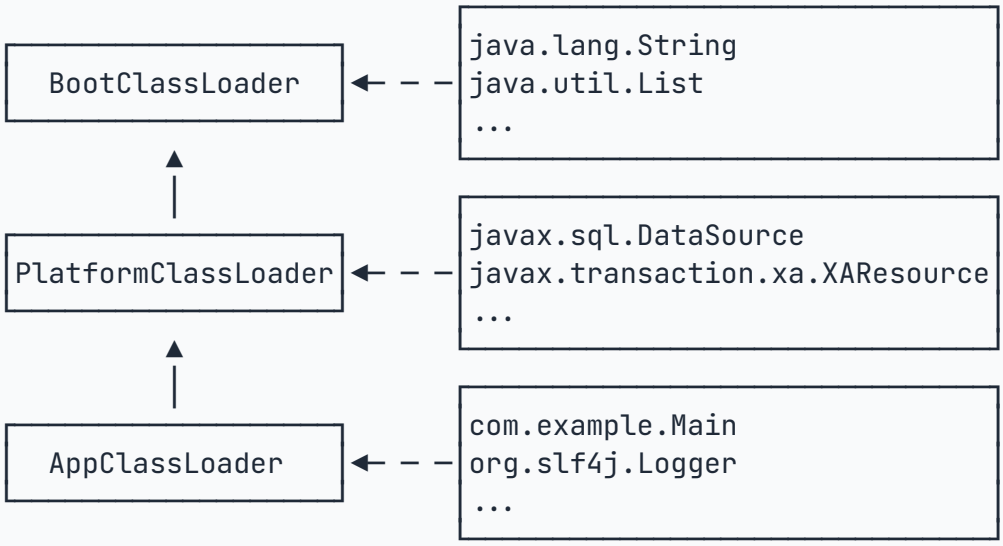
对于Java核心类，如 `java.lang.String` ，返回 `null` 表示使用的是JVM内部的启动类加载器（ `BootClassLoader` ），对于非核心的JDK类，如 `javax.sql.DataSource` ，使用的是 `PlatformClassLoader` ，对于用户编写的类，如 `Main` ，使用的是 `AppClassLoader` 。

我们通常说的ClassPath机制，即JVM应该在哪些目录和哪些jar包里去找Class，实际上说的是 `AppClassLoader` 使用的ClassPath，这3个ClassLoader按优先级排序如下：

1. BootClassLoader
2. PlatformClassLoader
3. AppClassLoader

用AppClassLoader加载一个Class时，它首先会委托父级ClassLoader尝试加载，如果加载失败，才尝试自己加载，这就是JVM的ClassLoader使用的双亲委派模型，它是为了防止用AppClassLoader加载用户自己编写的 `java.lang.String` 导致破坏JDK的核心类。

因此，对于一个Class来说，它始终关联着一个加载它自己的ClassLoader：



现在，假设我们完成了Jerrymouse服务器的开发，那么最后得到的就是 `jerrymouse.jar` 这样的jar包，如果要运行一个 `hello-webapp.war` ，我们期待的命令行如下：

```
$ java -jar jerrymouse.jar --war hello-webapp.war
```

上述命令行的classpath实际上是 `jerrymouse.jar` ，服务器的类均可以被JVM的 `AppClassLoader` 加载，但是， `AppClassLoader` 无法加载 `hello-webapp.war` 在 `/WEB-INF/classes` 存放的 `.class` 文件，也无法加载在 `/WEB-INF/lib` 存放的jar文件，原因是它们均不在classpath中，且运行期无法修改classpath。

因此，我们必须自己编写ClassLoader，才能加载到 `hello-webapp.war` 里的 `.class` 文件和 `jar` 包。

编写ClassLoader

为了加载 `war` 包里的 `.class` 文件和 `jar` 包，我们定义一个 `WebAppClassLoader` 。直接从 `ClassLoader` 继承不是不可以，但是要自己编写的代码太多。ClassLoader看起来很复杂，实际上就是想办法以任何方式拿到 `.class` 文件的用 `byte[]` 表示的内容，然后用 `ClassLoader` 的 `defineClass()` 获得JVM加载后的 `Class` 实例。大多数ClassLoader都是基于文件的加载，因此，JDK提供了一个 `URLClassLoader` 方便编写从文件加载的ClassLoader：

```
public class WebAppClassLoader extends URLClassLoader {

    public WebAppClassLoader(Path classPath, Path libPath) throws
IOException {
        super("WebAppClassLoader", createUrls(classPath, libPath),
```

```
ClassLoader.getSystemClassLoader());
    }

    // 返回一组URL用于搜索class:
    static URL[] createUrls(Path classPath, Path libPath) throws
IOException {
        List<URL> urls = new ArrayList<>();
        urls.add(toDirURL(classPath));
        Files.list(libPath).filter(p ->
p.toString().endsWith(".jar")).sorted().forEach(p -> {
            urls.add(toJarURL(p));
        });
        return urls.toArray(URL[]::new);
    }

    static URL toDirURL(Path p) {
        // 将目录转换为URL:
        ...
    }

    static URL toJarURL(Path p) {
        // 将jar包转换为URL:
        ...
    }
}
```

只要传入正确的目录和一组jar包，`WebAppClassLoader` 就可以加载到对应的 `.class` 文件。

下一步是修改启动流程，先解析命令行参数 `--war` 拿到 `war` 包的路径，然后解压到临时目录，获取到 `/tmp/xxx/WEB-INF/classes` 路径以及 `/tmp/xxx/WEB-INF/lib` 路径，就可以构造 `WebAppClassLoader` 了：

```
Path classesPath = ...
Path libPath = ...
ClassLoader classLoader = new WebAppClassLoader(classesPath, libPath);
```

接下来，需要获取到所有的 `Servlet` 、 `Filter` 和 `Listener` 组件，因此需要在 `WebAppClassLoader` 的范围内扫描所有 `.class` 文件：

```
Set<Class<?>> classSet = ... // 扫描获得所有Class
```

修改 `HttpConnector` ，传入 `ClassLoader` 和扫描的Class，就可以把所有 `Servlet` 、 `Filter` 和 `Listener` 添加到 `ServletContext` 中。这样，我们就把写死的Servlet组件从服务器中移除掉，并实现了从外部war包动态加载Servlet组件。

设置ContextClassLoader

在 `HttpConnector` 中，我们还需要对 `handler()` 方法进行改进，正确设置线程的ContextClassLoader（上下文类加载器）：

```
public void handle(HttpExchange exchange) throws IOException {
    var adapter = new HttpExchangeAdapter(exchange);
    var response = new HttpServletResponseImpl(this.config, adapter);
    var request = new HttpServletRequestImpl(this.config,
this.servletContext, adapter, response);
    try {
        // 将线程的上下文类加载器设置为WebAppClassLoader:
        Thread.currentThread().setContextClassLoader(this.classLoader);
        this.servletContext.process(request, response);
    } catch (Exception e) {
        logger.error(e.getMessage(), e);
    } finally {
        // 恢复默认的线程的上下文类加载器:
        Thread.currentThread().setContextClassLoader(null);
        response.cleanup();
    }
}
```

为什么需要设置线程的ContextClassLoader？执行 `handle()` 方法的线程是由线程池提供的，线程池是 `HttpConnector` 创建的，因此， `handle()` 方法内部加载的任何类都是由 `AppClassLoader` 加载的，而我们希望加载的类是由 `WebAppClassLoader` 从解压的 `war` 包中加载，此时，就需要设置线程的上下文类加载器。

举例说明：

当我们在一个方法中调用 `Class.forName()` 时：

```
Object createInstance(String className) {
    Class<?> clazz = Class.forName(className);
```

```
    return clazz.newInstance();
}
```

正常情况下，将由 `AppClassLoader` 负责查找 `Class` ，显然是找不到war包解压后存放在 `classes` 和 `lib` 目录里的类，只有我们自己写的 `WebAppClassLoader` 才能找到，因此，必须设置正确的线程上下文类加载器：

```
Object createInstance(String className) {
    Thread.currentThread().setContextClassLoader(this.classLoader);
    Class<?> clazz = Class.forName(className);
    Thread.currentThread().setContextClassLoader(null);
    return clazz.newInstance();
}
```

最后，完善所有接口的实现类，我们就成功开发了一个迷你版的Tomcat服务器！

参考源码

可以从GitHub或Gitee下载源码。

GitHub

小结

开发Web服务器时，需要编写自定义的ClassLoader，才能从war包中加载 `.class` 文件；

处理Servlet请求的线程必须正确设置ContextClassLoader。

评论

部署Web App

原文链接

现在，我们已经实现了 `WebAppClassLoader` ，就可以启动Web Server、加载war包。

先编写一个简单的 `hello-webapp` ，实现Servlet、Filter和Listener如下：

- HelloServlet：输出一个简单的 `Hello, {name}` ；
- LoginServlet：使用Session实现登录功能；
- LogoutServlet：使用Session实现登出功能；
- LogFilter：打印日志的Filter；
- 若干Listener：用于监听各种事件。

用Maven打包为标准的war包，我们得到一个 `hello-webapp-1.0.war` 文件。

启动Web Server并加载war文件，使用以下命令：

```
$ java -jar /path/to/jerrymouse-1.0.0.jar -w /path/to/hello-webapp-1.0.war
```

确保路径正确，就可以看到如下输出：

```
10:20:07.586 [main] INFO  com.itranswarp.jerrymouse.Start -- extract
'/Users/liaoxuefeng/Git/Github/jerrymouse/step-by-step/hello-
webapp/target/hello-webapp-1.0.war' to
'/var/folders/np/n5bzqjhs2l521tn0bkcqvzcc0000gn/T/_jm_10421889768839940273'
10:20:07.619 [main] INFO  com.itranswarp.jerrymouse.Start -- set web root:
/var/folders/np/n5bzqjhs2l521tn0bkcqvzcc0000gn/T/_jm_10421889768839940273
10:20:07.943 [main] INFO  c.i.j.classloader.WebAppClassLoader -- set
classes path:
/var/folders/np/n5bzqjhs2l521tn0bkcqvzcc0000gn/T/_jm_10421889768839940273/W
EB-INF/classes
10:20:07.944 [main] INFO  c.i.j.classloader.WebAppClassLoader -- set jar
path:
/var/folders/np/n5bzqjhs2l521tn0bkcqvzcc0000gn/T/_jm_10421889768839940273/W
EB-INF/lib/logback-classic-1.4.6.jar
10:20:07.944 [main] INFO  c.i.j.classloader.WebAppClassLoader -- set jar
path:
/var/folders/np/n5bzqjhs2l521tn0bkcqvzcc0000gn/T/_jm_10421889768839940273/W
```

```
EB-INF/lib/logback-core-1.4.6.jar
10:20:07.944 [main] INFO  c.i.j.classloader.WebAppClassLoader -- set jar
path:
/var/folders/np/n5bzqjhs2l521tn0bkcqvzcc0000gn/T/_jm_10421889768839940273/W
EB-INF/lib/slf4j-api-2.0.4.jar
10:20:07.954 [main] INFO  com.itranswarp.jerrymouse.Start -- Found
@WebServlet: com.itranswarp.sample.web.HelloServlet
10:20:07.955 [main] INFO  com.itranswarp.jerrymouse.Start -- Found
@WebServlet: com.itranswarp.sample.web.LoginServlet
10:20:07.956 [main] INFO  com.itranswarp.jerrymouse.Start -- Found
@WebServlet: com.itranswarp.sample.web.LogoutServlet
10:20:07.960 [main] INFO  com.itranswarp.jerrymouse.Start -- Found
@WebFilter: com.itranswarp.sample.web.filter.LogFilter
10:20:07.962 [main] INFO  com.itranswarp.jerrymouse.Start -- Found
@WebListener:
com.itranswarp.sample.web.listener.HelloHttpSessionAttributeListener
10:20:07.963 [main] INFO  com.itranswarp.jerrymouse.Start -- Found
@WebListener: com.itranswarp.sample.web.listener.HelloHttpSessionListener
10:20:07.964 [main] INFO  com.itranswarp.jerrymouse.Start -- Found
@WebListener:
com.itranswarp.sample.web.listener.HelloServletContextAttributeListener
10:20:07.965 [main] INFO  com.itranswarp.jerrymouse.Start -- Found
@WebListener:
com.itranswarp.sample.web.listener.HelloServletContextListener
10:20:07.966 [main] INFO  com.itranswarp.jerrymouse.Start -- Found
@WebListener:
com.itranswarp.sample.web.listener.HelloServletRequestAttributeListener
10:20:07.967 [main] INFO  com.itranswarp.jerrymouse.Start -- Found
@WebListener:
com.itranswarp.sample.web.listener.HelloServletRequestListener
10:20:08.003 [main] ERROR com.itranswarp.jerrymouse.Start -- load class
'ch.qos.logback.core.net.LoginAuthenticator' failed: NoClassDefFoundError:
jakarta/mail/Authenticator
10:20:08.038 [main] INFO  c.i.j.connector.HttpConnector -- starting
jerrymouse http server at 0.0.0.0:8080...
10:20:08.044 [main] INFO  c.i.j.engine.ServletContextImpl -- set web root:
/var/folders/np/n5bzqjhs2l521tn0bkcqvzcc0000gn/T/_jm_10421889768839940273
10:20:08.044 [main] INFO  c.i.j.engine.ServletContextImpl -- auto register
@WebListener:
com.itranswarp.sample.web.listener.HelloServletRequestAttributeListener
10:20:08.045 [main] INFO  c.i.j.engine.ServletContextImpl -- auto register
```

```
@WebListener:
com.itranswarp.sample.web.listener.HelloHttpSessionAttributeListener
10:20:08.045 [main] INFO  c.i.j.engine.ServletContextImpl -- auto register
@WebListener:
com.itranswarp.sample.web.listener.HelloServletContextListener
10:20:08.045 [main] INFO  c.i.j.engine.ServletContextImpl -- auto register
@WebListener: com.itranswarp.sample.web.listener.HelloHttpSessionListener
10:20:08.045 [main] INFO  c.i.j.engine.ServletContextImpl -- auto register
@WebListener:
com.itranswarp.sample.web.listener.HelloServletContextAttributeListener
10:20:08.045 [main] INFO  c.i.j.engine.ServletContextImpl -- auto register
@WebListener:
com.itranswarp.sample.web.listener.HelloServletRequestListener
```

从日志信息可知，`war`包被自动解压到临时目录，然后，初始化`WebAppClassLoader`，定位`classes`目录和`lib`目录下的所有jar包，自动扫描所有class文件，找出Servlet、Filter和Listener组件并自动注册，我们就可以在浏览器测试页面：

并在后台观察Listener输出：

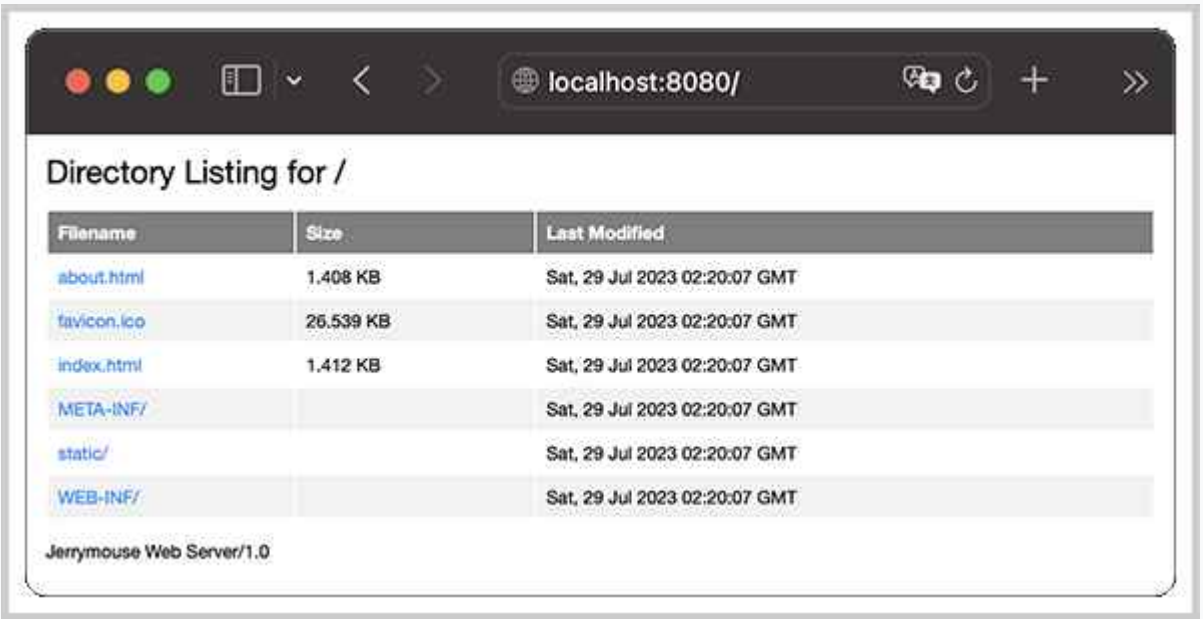
```
10:20:49.380 [pool-1-thread-11] DEBUG c.i.j.engine.ServletContextImpl --
process /login by filter
[com.itranswarp.sample.web.filter.LogFilter@7bf37a4a], servlet
com.itranswarp.sample.web.LoginServlet@2a156b2d
10:20:49.380 [pool-1-thread-11] DEBUG c.i.j.engine.ServletContextImpl --
invoke ServletRequestInitialized: request =
HttpServletRequestImpl@24d2901f[GET:/login]
10:20:49.380 [pool-1-thread-11] INFO  c.i.s.w.l.HelloServletRequestListener
-- >>> ServletRequest initialized:
```

```
HttpServletRequestImpl@24d2901f[GET:/login]
10:20:49.380 [pool-1-thread-11] INFO  c.i.sample.web.filter.LogFilter --
[GET] /login
10:20:49.383 [pool-1-thread-11] DEBUG c.i.j.engine.ServletContextImpl --
invoke HttpSessionCreated: session = HttpSessionImpl@5c0656b[id=ef81240c-
12ac-414b-96e6-fe843951fd1f]
10:20:49.384 [pool-1-thread-11] INFO  c.i.s.w.l.HelloHttpSessionListener --
>>> HttpSession created: HttpSessionImpl@5c0656b[id=ef81240c-12ac-414b-
96e6-fe843951fd1f]
10:20:49.385 [pool-1-thread-11] DEBUG c.i.j.engine.ServletContextImpl --
invoke ServletRequestDestroyed: request =
HttpServletRequestImpl@24d2901f[GET:/login]
10:20:49.385 [pool-1-thread-11] INFO  c.i.s.w.l.HelloServletRequestListener
-- >>> ServletRequest destroyed:
HttpServletRequestImpl@24d2901f[GET:/login]
10:21:04.777 [pool-1-thread-12] DEBUG c.i.j.engine.ServletContextImpl --
process /login by filter
[com.itranswarp.sample.web.filter.LogFilter@7bf37a4a], servlet
com.itranswarp.sample.web.LoginServlet@2a156b2d
10:21:04.778 [pool-1-thread-12] DEBUG c.i.j.engine.ServletContextImpl --
invoke ServletRequestInitialized: request =
HttpServletRequestImpl@73a3895e[POST:/login]
10:21:04.778 [pool-1-thread-12] INFO  c.i.s.w.l.HelloServletRequestListener
-- >>> ServletRequest initialized:
HttpServletRequestImpl@73a3895e[POST:/login]
10:21:04.778 [pool-1-thread-12] INFO  c.i.sample.web.filter.LogFilter --
[POST] /login
10:21:04.779 [pool-1-thread-12] DEBUG c.i.j.engine.ServletContextImpl --
invoke HttpSessionAttributeAdded: username = bob, session =
HttpSessionImpl@5c0656b[id=ef81240c-12ac-414b-96e6-fe843951fd1f]
10:21:04.780 [pool-1-thread-12] INFO
c.i.s.w.l.HelloHttpSessionAttributeListener -- >>> HttpSession attribute
added: username = bob
10:21:04.781 [pool-1-thread-12] DEBUG c.i.j.engine.ServletContextImpl --
invoke ServletRequestDestroyed: request =
HttpServletRequestImpl@73a3895e[POST:/login]
10:21:04.781 [pool-1-thread-12] INFO  c.i.s.w.l.HelloServletRequestListener
-- >>> ServletRequest destroyed:
HttpServletRequestImpl@73a3895e[POST:/login]
10:21:04.790 [pool-1-thread-13] DEBUG c.i.j.engine.ServletContextImpl --
process /login by filter
```

```
[com.itranswarp.sample.web.filter.LogFilter@7bf37a4a], servlet
com.itranswarp.sample.web.LoginServlet@2a156b2d
10:21:04.790 [pool-1-thread-13] DEBUG c.i.j.engine.ServletContextImpl --
invoke ServletRequestInitialized: request =
HttpServletRequestImpl@455863b5[GET:/login]
10:21:04.790 [pool-1-thread-13] INFO  c.i.s.w.l.HelloServletRequestListener
-- >>> ServletRequest initialized:
HttpServletRequestImpl@455863b5[GET:/login]
10:21:04.790 [pool-1-thread-13] INFO  c.i.sample.web.filter.LogFilter --
[GET] /login
10:21:04.791 [pool-1-thread-13] DEBUG c.i.j.engine.ServletContextImpl --
invoke ServletRequestDestroyed: request =
HttpServletRequestImpl@455863b5[GET:/login]
10:21:04.791 [pool-1-thread-13] INFO  c.i.s.w.l.HelloServletRequestListener
-- >>> ServletRequest destroyed:
HttpServletRequestImpl@455863b5[GET:/login]
10:21:05.798 [pool-1-thread-14] DEBUG c.i.j.engine.ServletContextImpl --
process /logout by filter
[com.itranswarp.sample.web.filter.LogFilter@7bf37a4a], servlet
com.itranswarp.sample.web.LogoutServlet@20675c59
10:21:05.798 [pool-1-thread-14] DEBUG c.i.j.engine.ServletContextImpl --
invoke ServletRequestInitialized: request =
HttpServletRequestImpl@2ea0a625[GET:/logout]
10:21:05.798 [pool-1-thread-14] INFO  c.i.s.w.l.HelloServletRequestListener
-- >>> ServletRequest initialized:
HttpServletRequestImpl@2ea0a625[GET:/logout]
10:21:05.798 [pool-1-thread-14] INFO  c.i.sample.web.filter.LogFilter --
[GET] /logout
10:21:05.798 [pool-1-thread-14] DEBUG c.i.j.engine.ServletContextImpl --
invoke ServletContextAttributeRemoved: username = bob, session =
HttpSessionImpl@5c0656b[id=ef81240c-12ac-414b-96e6-fe843951fd1f]
10:21:05.798 [pool-1-thread-14] INFO
c.i.s.w.l.HelloHttpSessionAttributeListener -- >>> HttpSession attribute
removed: username = bob
10:21:05.798 [pool-1-thread-14] DEBUG c.i.j.engine.ServletContextImpl --
invoke HttpSessionDestroyed: session = HttpSessionImpl@5c0656b[id=ef81240c-
12ac-414b-96e6-fe843951fd1f]
10:21:05.798 [pool-1-thread-14] INFO  c.i.s.w.l.HelloHttpSessionListener --
>>> HttpSession destroyed: HttpSessionImpl@5c0656b[id=ef81240c-12ac-414b-
96e6-fe843951fd1f]
10:21:05.799 [pool-1-thread-14] DEBUG c.i.j.engine.ServletContextImpl --
```

```
invoke ServletRequestDestroyed: request =
HttpServletRequestImpl@2ea0a625[GET:/logout]
10:21:05.799 [pool-1-thread-14] INFO  c.i.s.w.l.HelloServletRequestListener
-- >>> ServletRequest destroyed:
HttpServletRequestImpl@2ea0a625[GET:/logout]
```

此外，因为我们没有定义映射到 `/` 的Servlet，因此，Web Server自动注册一个内置的 `DefaultServlet` ，用于显示目录：



按Ctrl+C关闭服务器，由于我们在Web Server启动时设置了ShutdownHook，所以临时目录会被自动删除。

参考源码

可以从GitHub或Gitee下载源码。

GitHub

小结

通过加载war包，我们就可以完整地启动Web Server，运行一个Web App。

评论

部署Spring Web App

原文链接

现在，我们编写的Jerrymouse Server运行一个简单的 `hello-webapp` 没问题，那么复杂的Web App呢？

我们来编写一个基于Spring MVC的Web App。

首先，我们要加载Spring内置的 `DispatcherServlet` ，如果用传统的 `web.xml` 配置，则可以编写如下配置文件：

```
<?xml version="1.0"?>
<web-app>
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextClass</param-name>
            <param-
value>org.springframework.web.context.support.AnnotationConfigWebApplicatio
nContext</param-value>
        </init-param>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>com.itranswarp.sample.AppConfig</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

但是，我们已经干掉了通过 `web.xml` 配置的方式，只支持自动扫描 `@WebServlet` ，因此，需要编写一个继承自 `DispatcherServlet` 的 `AppDispatcherServlet` ：

```
@WebServlet(
    urlPatterns = "/",
```

```
    initParams = {
        @WebInitParam(name = "contextClass", value =
"org.springframework.web.context.support.AnnotationConfigWebApplicationCont
ext"),
        @WebInitParam(name = "contextConfigLocation", value =
"com.itranswarp.sample.AppConfig") })
public class AppDispatcherServlet extends DispatcherServlet {
}
```

这样，Jerrymouse Server会自动扫描到 `AppDispatcherServlet` ，然后，根据 `@WebServlet` 的配置，启动Spring容器，类型为 `AnnotationConfigWebApplicationContext` ，配置类 `com.itranswarp.sample.AppConfig` ，所以还需要编写配置类 `AppConfig` ：

```
@Configuration
@ComponentScan
@EnableWebMvc
@EnableTransactionManagement
@PropertySource("classpath:/jdbc.properties")
public class AppConfig {
    @Bean
    WebMvcConfigurer createWebMvcConfigurer() {
        return new WebMvcConfigurer() {
            @Override
            public void addResourceHandlers(ResourceHandlerRegistry
registry) {

registry.addResourceHandler("/static/**").addResourceLocations("/static/");

registry.addResourceHandler("/favicon.ico").addResourceLocations("/");
            }
        };
    }

    @Bean
    ViewResolver createViewResolver(@Autowired ServletContext
servletContext) {
        var engine = new PebbleEngine.Builder().autoEscaping(true)
                // loader:
                .loader(new Servlet5Loader(servletContext))
                // build:
```

```
                .build();
        var viewResolver = new PebbleViewResolver(engine);
        viewResolver.setPrefix("/WEB-INF/templates/");
        viewResolver.setSuffix("");
        return viewResolver;
    }

    @Bean
    DataSource createDataSource(@Value("${jdbc.driver}") String jdbcDriver,
@Value("${jdbc.url}") String jdbcUrl,
            @Value("${jdbc.username}") String jdbcUsername,
@Value("${jdbc.password}") String jdbcPassword) {
        HikariConfig config = new HikariConfig();
        config.setDriverClassName(jdbcDriver);
        config.setJdbcUrl(jdbcUrl);
        config.setUsername(jdbcUsername);
        config.setPassword(jdbcPassword);
        config.addDataSourceProperty("autoCommit", "false");
        config.addDataSourceProperty("connectionTimeout", "5");
        config.addDataSourceProperty("idleTimeout", "60");
        return new HikariDataSource(config);
    }

    @Bean
    JdbcTemplate createJdbcTemplate(@Autowired DataSource dataSource) {
        return new JdbcTemplate(dataSource);
    }

    @Bean
    PlatformTransactionManager createTxManager(@Autowired DataSource
dataSource) {
        return new DataSourceTransactionManager(dataSource);
    }
}
```

`AppConfig`是标准的Spring配置类，我们正常配置MVC、ViewResolver、JDBC相关的DataSource、JdbcTemplate和PlatformTransactionManager，以及Spring容器需要的Service、Controller等，就可以实现一个完整的Spring MVC的WebApp。

用Maven编译`spring-webapp`，得到`spring-webapp-1.0.war`文件。用以下命令运行：

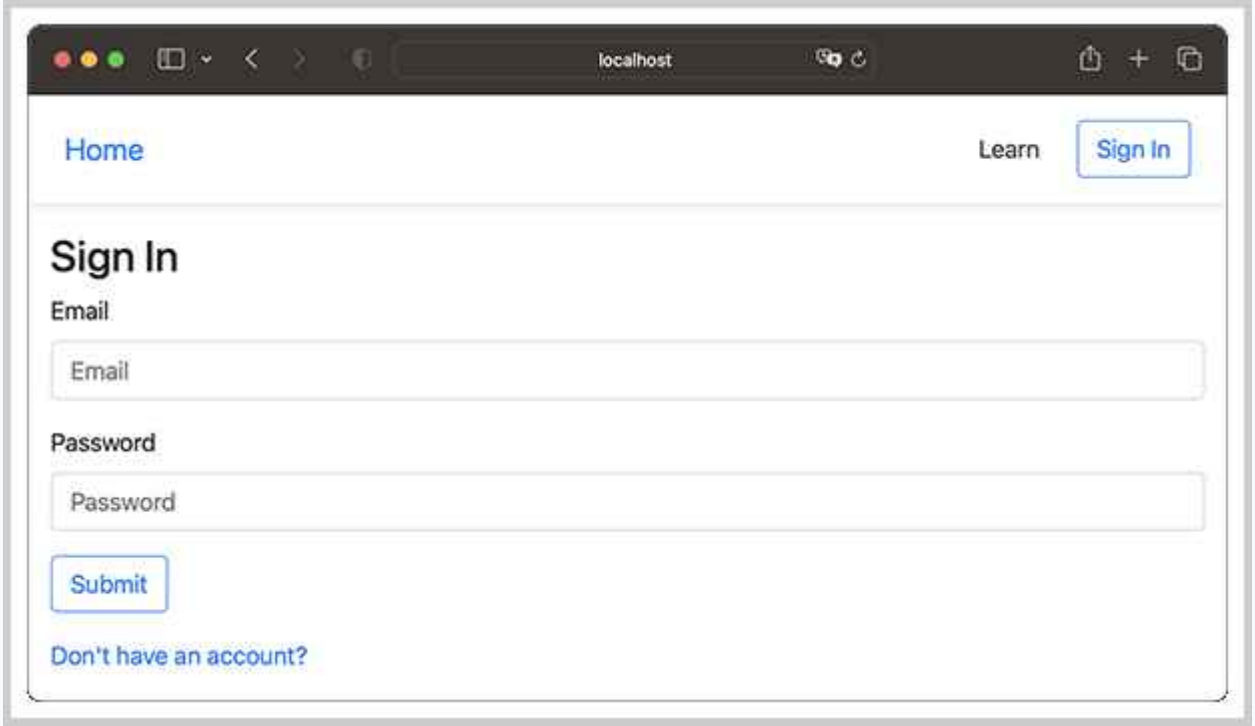
```
$ java -jar /path/to/jerrymouse-1.0.0.jar -w /path/to/spring-webapp-1.0.war
```

观察 `AppDispatcherServlet` 的输出：

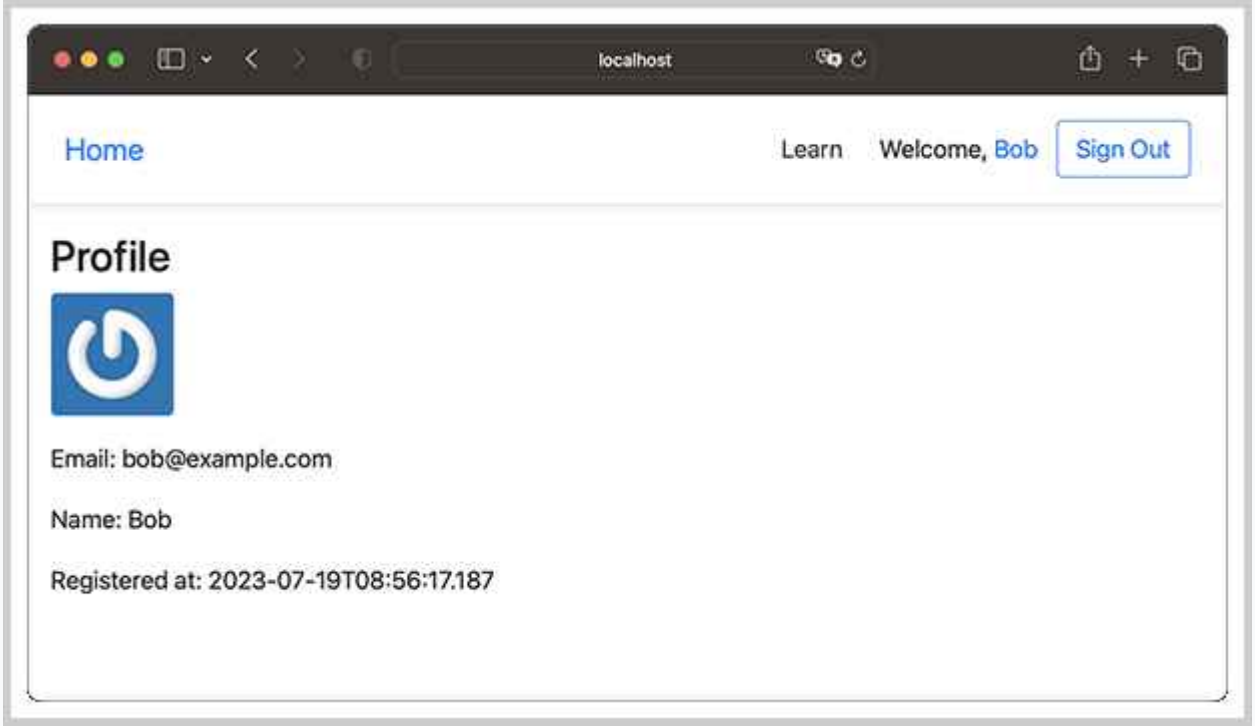
```
10:48:08.200 [main] DEBUG o.s.w.c.s.AnnotationConfigWebApplicationContext -
- Refreshing WebApplicationContext for namespace 'appDispatcherServlet-
servlet'
...
10:48:08.223 [main] DEBUG c.i.j.engine.ServletContextImpl -- invoke
ServletContextAttributeAdded:
org.springframework.web.context.support.ServletContextScope =
org.springframework.web.context.support.ServletContextScope@2616b618
10:48:08.225 [main] DEBUG o.s.b.f.s.DefaultListableBeanFactory -- Creating
shared instance of singleton bean
'org.springframework.context.annotation.internalConfigurationAnnotationProc
essor'
...
10:48:08.705 [main] DEBUG c.i.sample.AppDispatcherServlet -- Detected
AcceptHeaderLocaleResolver
10:48:08.705 [main] DEBUG c.i.sample.AppDispatcherServlet -- Detected
FixedThemeResolver
10:48:08.706 [main] DEBUG c.i.sample.AppDispatcherServlet -- Detected
org.springframework.web.servlet.view.DefaultRequestToViewNameTranslator@559
fd5ec
10:48:08.706 [main] DEBUG c.i.sample.AppDispatcherServlet -- Detected
org.springframework.web.servlet.support.SessionFlashMapManager@bb12f41
10:48:08.707 [main] DEBUG c.i.j.engine.ServletContextImpl -- invoke
ServletContextAttributeAdded:
org.springframework.web.servlet.FrameworkServlet.CONTEXT.appDispatcherServl
et = WebApplicationContext for namespace 'appDispatcherServlet-servlet',
started on Sat Jul 29 10:48:08 CST 2023
10:48:08.707 [HikariPool-1 connection adder] DEBUG
com.zaxxer.hikari.pool.HikariPool -- HikariPool-1 - Added connection
org.hsqldb.jdbc.JDBCConnection@4b7d3c3b
10:48:08.707 [main] DEBUG c.i.sample.AppDispatcherServlet --
enableLoggingRequestDetails='false': request parameters and headers will be
masked to prevent unsafe logging of potentially sensitive data
10:48:08.707 [main] INFO  c.i.sample.AppDispatcherServlet -- Completed
initialization in 511 ms
```

注意到 `AppDispatcherServlet` 是绑定到 `/` 的，因此，所有请求均全部由Spring提供的 `DispatcherServlet` 处理，包括静态文件。

访问 `http://localhost:8080` ，观察MVC的输出：



观察登录后的输出：



可见，Jerrymouse Server可以正常运行基于Spring MVC的Web App。不过，我们无法使用Spring提供的 `async` 相关功能，例如 `DeferredResult` ，也不能使用WebSocket相关功能。

参考源码

可以从GitHub或Gitee下载源码。

GitHub

小结

通过正确配置Spring提供的 `DispatcherServlet` ，我们可以用Jerrymouse Server运行基于Spring MVC的Web App。

评论

常见问题

原文链接

本节列出开发Servlet服务器时需要注意的一些常见问题。

如何正确实现getOutputStream()和getWriter()?

根据Servlet规范，`getOutputStream()`和`getWriter()`在一次HTTP处理中只能二选一，不能都调用，因此，HttpServletResponse内部会用`callOutput`记录调用状态：

- null：未调用`getOutputStream()`和`getWriter()`；
- Boolean.TRUE：已调用`getOutputStream()`；
- Boolean.FALSE：已调用`getWriter()`。

违反调用规则会抛出`IllegalStateException`。

HttpServletResponse为什么要实现cleanup()?

因为Web App可能不会调用`getOutputStream()`或`getWriter()`，而是直接设置Header后返回：

```
resp.setStatus(403);
```

此时，`HttpConnector`要调用`cleanup()`，如果发现没有发送Header，则需要立刻发送Header，否则浏览器无法收到响应。

此外，根据HttpConnector的实现方式，基于JDK的`HttpExchange`的`OutputStream`也需要关闭（但不一定会关闭对应的TCP连接）。

如何对Servlet排序?

Servlet需要根据映射进行排序，遵循以下原则：

- 路径长的优先级高，例如，`/auth/login`排在`/auth`前；
- 前缀匹配比后缀匹配优先级高，例如，`/auth/*`排在`*.do`前。

如何处理"/"映射?

根据Servlet规范，`/`相当于`/*`，但还是有所不同，因为`/`表示默认的Servlet，即所有规则均不匹配时，最后匹配`/`。

如果一个Web App没有提供`/`映射，则Web Server可以自动提供一个默认的映射到`/`的Servlet。Jerrymouse和Tomcat类似，提供一个浏览文件的`DefaultServlet`。

如何处理静态文件?

处理静态文件时，将URL路径`/path/to/file.doc`转换为本地文件路径`${webroot}/path/to/file.doc`，然后根据扩展名设置正确的`Content-Type`，读取文件内容，发送即可。

需要注意的是，Servlet规范规定，不允许访问`/WEB-INF/`开头的URL，因此，遇到访问`/WEB-INF/*`的请求时，直接返回404错误码。

如何对Filter排序?

Servlet规范没有对Filter排序的要求，但我们在实现时还是按`@WebFilter`的`filterName()`进行排序，这样Web App可以根据名称调整Filter的顺序。

如何启用虚拟线程?

默认情况下，Jerrymouse Server采用线程池模式，要启用虚拟线程，可以加上配置项，以创建不同类型的`ExecutorService`：

```
ExecutorService executor = config.server.enableVirtualThread ?
        Executors.newVirtualThreadPerTaskExecutor() :
        new ThreadPoolExecutor(0, config.server.threadPoolSize, 0L,
TimeUnit.MILLISECONDS, new LinkedBlockingQueue<>());
```

评论

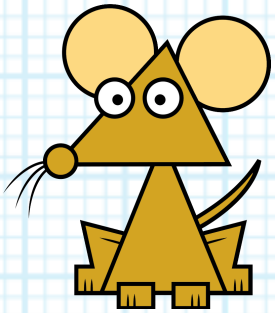
期末总结

原文链接

终于到了期末总结的时刻了！

通过开发一个迷你版Tomcat服务器，相信大家对Java Web开发又有了更深刻的理解。通过自己从零开始手写Jerrymouse Server，写完后应该完全可胜任Java架构师这样的高级职位！

评论



手写Tomcat

自己动手，从零开发一个迷你版Tomcat服务器！

Author: 廖雪峰

Version: 2025-06-16

Website: https://liaoxuefeng.com/books/jerrymouse/