计算机网络实验实验报告

学院（系）名称：信息学院

姓名	高星杰	学号	202130220712	专业	计算机科学与技术
班级	21 级 02 班	实验项目	WebServer 的粗略完整实现		
课程名称		计算机网络实验		课程代码	3103009336
实验时间		2023 年 11 月 2 日		实验地点	逸夫楼 C207

考核内容	目的和原理 40	内容及过程分析 30	实验方案设计 10	实验结果（结论正确性以及分析合理性）20	成绩	
各项得分						
考核标准	○原理明确 ○原理较明确 ○原理不明确	○分析清晰 ○分析较清晰 ○分析不清晰	○设计可行 ○设计基本可行 ○设计不可行	○结论正确，分析合理 ○结论正确，分析不充分 ○结论不正确，分析不合理	教师签字：	

1. 实验目的：

（1） 将所学 WebServer 基本理论应用于实践
（2） 学习主流 WebServer——Tomcat 的一些开发理论
（3） 实现基于 Java 的，安全性较高，复用性较高，webserver 和 webapp 解耦合，可控性较高和代码可读性好的功能完善的 WebServer

2. 实验原理：

实验项目用到了众多工具和原理，想要彻底理解本实验项目就一定要先清楚这些原理和概念。

2.1 Java JMX

Java Management Extensions (JMX) 是一项通用、开放的管理和监控技术，可在需要管理和监控的任何地方部署。

我理解的使用场景：运行中的应用程序，你会想：“它到底在做什么？为什么用了这么长时间呢？” 在这些时刻，可能会想如果自己在应用程序中构建了更多的监视功能就好了。例如，在服务器应用程序中，能够查看排队等候处理的任务的数量和类型、当前正在处理的任务、过去一分钟或一小时内的吞吐量统计、平均任务处理时间等。而这些恰好是 JMX 的用武之地
以下是关于 JMX 的一些重要信息：

概述：JMX 技术提供了构建分布式、基于 Web 的、模块化和动态解决方案的工具，用于管理和监控设备、应用程序和服务驱动的网络。它适用于本地或远程管理 Java 应用程序[1][2].

功能：JMX 允许 Java 开发人员将资源封装为 Java 对象，并在分布式环境中将其公开为管

理资源。它具有易于配置、可扩展、可靠且友好的特点，可用于监控和管理 Java 应用程序[3]. 总之，JMX 是一个强大且方便的工具，用于监控和分析 Java 应用程序的性能。在本项目没有用到它分布式的相关功能，只使用了 JMX 检查内存和数据方式。

2.2 Java 多线程

C/S 架构中，多使用多线程进行操作。一个线程负责一个或多个 Socket 连接，完成客户端与服务器的通信交互。服务端每通过 Socket 建立一个连接，就建立一个新线程，并将该连接的相关信息传入新线程，子线程与其他线程互相独立，实现一对多的通信服务。

2.3 Java 面向对象

多态：多态允许不同类的对象对相同的方法做出不同的响应。这增加了代码的灵活性和可扩展性。
继承：继承是一种机制，允许一个类（子类）继承另一个类（父类）的属性和方法。这有助于代码重用和层次结构的构建。
封装：封装是将数据和操作数据的方法封装在一个单元内，以隐藏实现细节并提供更清晰的接口。
抽象：抽象是指创建一个模板或基类，其中包含一些未实现的方法。其他类可以继承这个抽象类并实现这些方法。
类：类是描述一类对象的行为和状态的模板。例如，一个“狗”类可以描述所有狗的共同特征和行为。
对象：对象是类的一个实例，具有状态（属性）和行为（方法）。例如，一只具体的狗就是一个对象。
实例：实例是类的具体对象，通过实例化类来创建。每个实例都有自己的状态和行为。
方法：方法是类中定义的操作，用于改变对象的状态或执行某些功能。
重载：重载是指在同一个类中定义多个具有相同名称但参数不同的方法。
在本项目中用的最多的是抽象类的继承和接口的实现

2.4 TCP 协议

TCP 即传输层通信协议，使用三次握手，四次挥手在端之间建立连接、销毁连接，在连接期间有提供可靠的服务

2.5 Http 协议

HTTP 协议基于 TCP/IP 通信协议传递数据，多用于 C/S 架构中。浏览器作为客户端，向 WEB 服务器发送请求，WEB 服务器根据请求内容，返回响应信息。协议定义了如 GET/POST/PUT/DELETE 等多种请求方法，并设置了各类响应码。C/S 双方通过协议信息，确定访问的资源。

2.6 Socket 套接字

Socket 套接字本质为编程接口，是对 TCP/IP 协议族的封装。Socket 像插座一样，为线路的两台主机建立连接。对于 JAVA 语言，首先需要创建服务器套接字，并开始监听，每当有客户端连接申请时，服务器创建一个新套接字负责与该客户端通信。

2.7 WebServer 的基本理论

（1） Web 服务器的概念：Web 服务器是指驻留于因特网上某种类型计算机的程序，可以处理浏览器等 Web 客户端的请求并返回相应响应，也可以放置网站文件，让全世界浏览；可以放置数据文件，让全世界下载。
（2） Web 服务器的功能：Web 服务器的主要功能是提供 HTTP 服务，即能够理解 URL（网络地址和 HTTP（浏览器用来查看网页的协议）的软件。Web 服务器还可以提供其他服务，如邮件服务、FTP 服务、数据库服务等。

（3） Web 服务器的工作原理：Web 服务器的工作原理并不复杂，一般可分成如下 4 个步骤：连接过程、请求过程、应答过程以及关闭连接 1。连接过程就是 Web 服务器和其浏览器之间所建立起来的一种连接。请求过程就是浏览器向 Web 服务器发送一个请求，包含请求的 URL、方法、头部和正文。应答过程就是 Web 服务器根据请求的 URL 查找对应的文件或动态生成内容，并返回一个应答，包含状态码、头部和正文。关闭连接就是 Web 服务器和浏览器断开连接，释放资源

2.8 设计模式

（1）责任链模式是一种对象的行为模式。在责任链模式里，很多对象由每一个对象对其下家的引用而连接起来形成一条链。请求在这个链上传递，直到链上的某一个对象决定处理此请求。发出这个请求的客户端并不知道链上的哪一个对象最终处理这个请求，这使得系统可以在不影响客户端的情况下动态地重新组织和分配责任。

Handler（抽象处理者）：定义一个处理请求的接口

ConcreteHandler（具体处理者）：处理请求的具体类，或者传给下家

（2）观察者模式原理也很简单，就是你在做事的时候旁边总有一个人在盯着你，当你做的事情是它感兴趣的时候，它就会跟着做另外一些事情。但是盯着你的人必须要到你那去登记，不然你无法通知它。观察者模式通常包含下面这几个角色：

Subject 就是抽象主题：它负责管理所有观察者的引用，同时定义主要的事件操作。

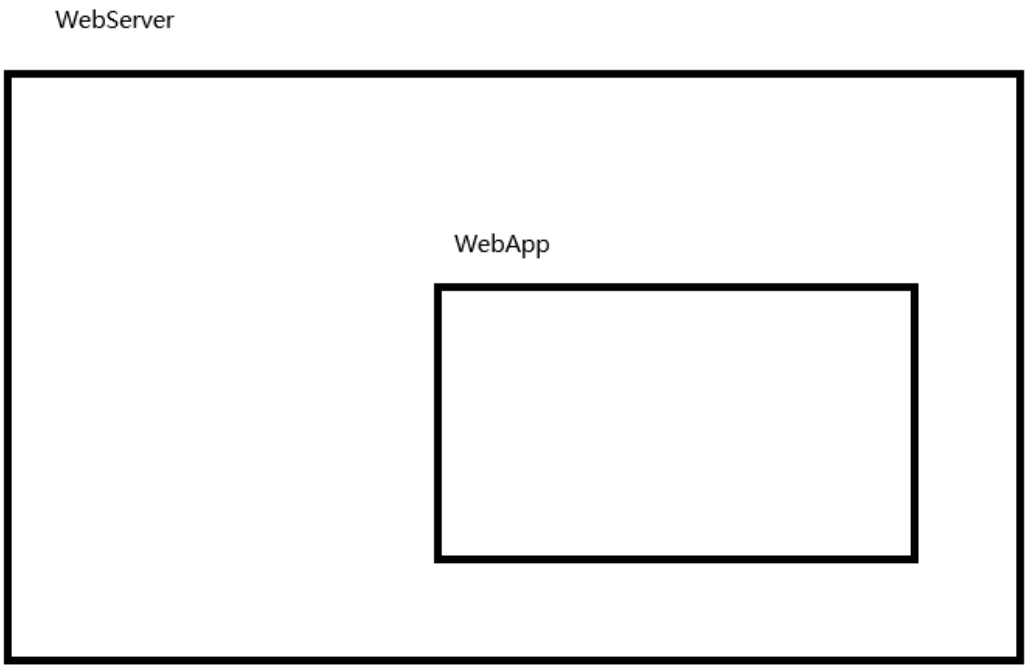
ConcreteSubject 具体主题：它实现了抽象主题的所有定义的接口，当自己发生变化时，会通知所有观察者。

Observer 观察者：监听主题发生变化相应的操作接口。

2.9 WebServer 和 Webapp 的区别和关系

WebServer 是 Webapp 运行的环境，为 Webapp 提供服务，能够让 coder 专注于对业务逻辑的编写而不是对接收数据的处理的

他们之间的关系可以用图展示：



我理解的本次实验**侧重于对 WebServer 的编写而不是 WebApp**，因为 WebApp 是要有实际应用中的要具体编写的，它是针对具体的业务逻辑编写的，而我们主要还是要实现对 http 协议的处理和响应，实现安全和解耦合，而不是对业务逻辑的代码实现，因为我们也没有那样的复杂的业务场景。

3. 实验分析：

首先根据之前的原理分析，我们的目标是实现一个安全的，可靠的，复用性高的，解耦合的 WebServer（能让开发者放入自己的 webapp），而不是去实现复杂的业务逻辑代码的 WebApp，更不是去实现精美的页面，那是后端开发者和前端开发者干的活。

其次要想设计一个软件程序，就一定要进行需求分析，这里实验分析主要就来分析下 WebServer 的需求：

3.1 WebServer 基本功能需求

（1） 实现多端口扫描（因为承载的 WebApp 不止一个，并且一个 app 也可能使用多个端口）
（2） 实现接收的 http 请求的 url 定向到 WebApp 或者静态资源
（3） 能够限制 WebApp 的资源访问（App 只能访问自己的资源）
（4） 能够实现一些异常的处理的（比如找不到页面，找不到 webapp）
（5） 根据配置选择使用持久连接和非持久连接

3.2 安全和稳定需求

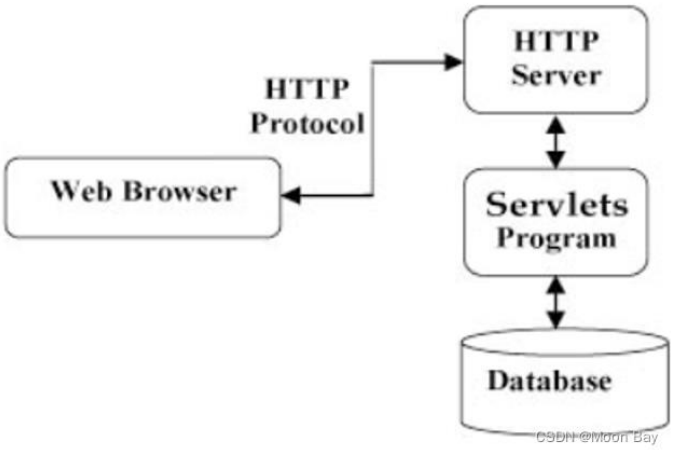
（1） 能够实时查看 WebApp 和各组件的运行状态
（2） 能够实时查看 WebApp 和各组件的运行占用情况
（3） 能够实时查看 WebApp 和各组件的访问情况（防止恶意刷流量）

3.3 解耦合和自定义的需求

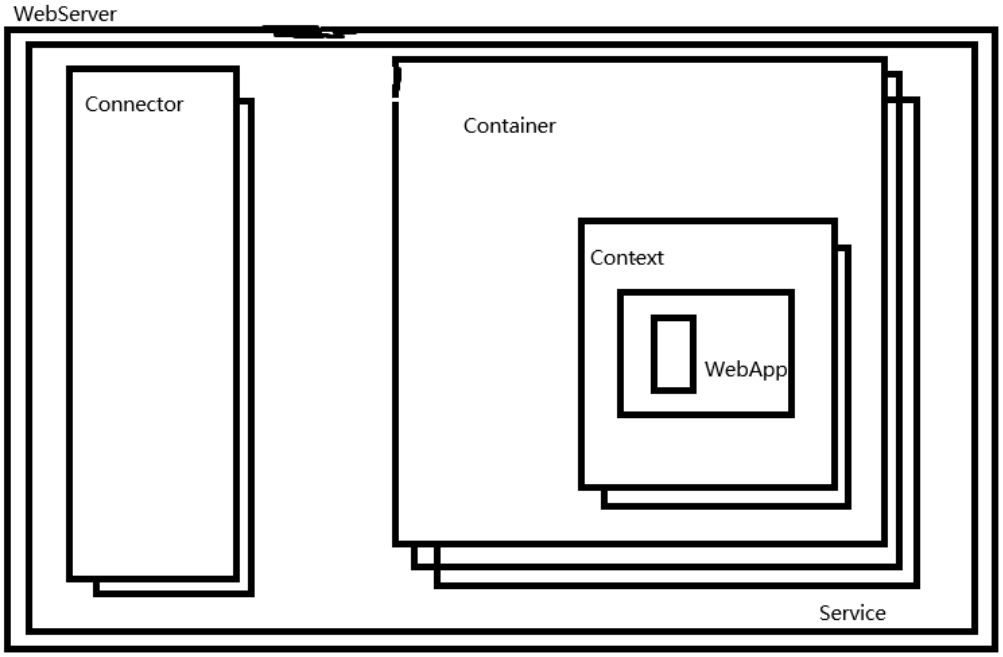
（1） 能够自定义配置文件
（2） WebApp 和 WebServer 解耦合（能够装入 app，编写配置文件后运行 app）

4. 实验设计：

首先**本实验项目的定位是一个 HttpServer（WebServer）如下图，而不是去实现业务逻辑的 WebApp，是为了 WebApp 做服务的（当然后面也会添加一些 WebApp 做测试，但这不是重点）**



下图是本项目的简单结构图（不包括生命周期控制和事件监听机制）



4.1 两大组件 Connector 和 Container 的

Connector 负责接收请求的然后解析请求，然后 Container 负责为调用 WebApp 和为它提供环境。

这里 connector 和 container 都是可以不止一个的，因为 connector 要监听不止一个的端口的 socket 而 container 可以有多个因为可以搭载多个 WebApp。

这两个组件通过责任链模式连接在一起，也就是通过一个流的形式，如下图：

4.2 组件生命周期管理

由于我们使用的是多线程，就要保证每个组件的线程安全，每个组件可能会有不同的组件创建和结束，然后要保证组件的运行不出问题，我们就要实现组件的生命周期管理，通过类似有限状态机的形式来保证服务可靠。

每个组件都有四个状态：init，start，stop，destroy

4.3 组件监管机制

使用监管机制的原因：

首先假如某一天我们忽然被甲方告知，业务大面积瘫痪，这时经过排查，发现由于 bug 导致 socket 连接使用完了没有被释放，导致后续业务没有可用的 socket 连接而超时报错。假如我们使用 JMX 来监控我们系统中 socket 连接池的信息，当 socket 连接池出现短时间内连接被大量使用，这个时候可以搭配我们的监控报警系统，在问题未出现之前就进行响应，可以极大避免上述情况的发生。

其次我们组件基本上数量都不止一个的，而且是在运行是创建的，为了防止应用无限制的生成的占用大量内存和性能，我们需要对组件进行监管，而我们的对组件的监管必须是的实时的否则将毫无意义，所以基于 java JMX 的监管机制就显得非常合适。

基于 JMX 的监管机制的优势：

首先我们可以自定义监管的内容，并且是运行时动态生产监管的内容，不是编译时的，这样所有动态生成的组件都可监管到

其次 JMX 可以使用的一个专用的端口来实现监管的可视化，帮助我们实时的获取组件的信息。

那么我们如何实现组件监管机制呢？

组件监管机制也要基于之前组件生命周期管理机制，我们不仅要获取组件的访问信息，还要获取组件的生命状态，来确保服务器正常服务。

我们要把监管抽象成一个接口让每一个组件都实现他的方法，来实现我们的监管机制，这样监管就嵌入到了每个组件当中，并且有很高的复用性。

4.4 事件监听机制

每个组件的状态会发生变化，变化的时候会抛出一些事件，**我们可以支持定义事件监听器来监听，并消费这些事件。**

我们实现的是简单的事件监听机制

4.5 线程池设计

线程池的设计，就是在一般的线程池上增加这两点：

（1） **我们将线程池也纳入 Lifecycle 生命周期管理**，所以让它实现了 Lifecycle 接口

（2） **引入超时机制：如果线程池中的线程超出了线程，我们就剩下的放到队列中等待，并且添加超时机制。**也就是说当 work queue 满时，会等待指定的时间，如果超时将抛出异常。

5. 实验过程：

5.1 类加载机制

实现的动态加载 webapp 中的类，通过配置文件的设置和规范来实现动态类的加载，让 webServer 更加灵活，webapp 的开发者能够不用管这些复杂的逻辑只用管业务逻辑就行。

相关代码：

用法：先写 webserver.xml 配置文件

```
<service id="1">
    <port id="1">80</port>
    <port id="2">8083</port>
    <corePoolSize>10</corePoolSize>
    <maxiumSize>20</maxiumSize>
    <keepAliveTime>5000</keepAliveTime>
    <webapp>
        <class>com.hzau.app.test</class>
    </webapp>
</service>
```

然后 webserver 初始化时会读取该文件，然后动态获取创建的类的，实现给类的同时必须要实现 WebApp 这个接口。

```
public interface WebApp {
    新 *
    void service();
}
```

也就是必须要实现一定的业务逻辑

明白了怎么用现在讲这么实现的

内部实现：

```java
public void init() throws LifecycleException {
    for (int j = 0; j < childNodes.getLength(); j++) {
        if (childNodes.item(j).getNodeType() != Node.ELEMENT_NODE) {
            continue;
        }
        String name = childNodes.item(j).getNodeName();
        String classpath = childNodes.item(j).getFirstChild().getNodeValue();
        Class tempClass;
        try {
            tempClass = Class.forName(classpath);
        } catch (ClassNotFoundException e) {
            throw new RuntimeException(e);
        }
        Object studentObj = getObject(tempClass);
        Method service;
        try {
            service = tempClass.getMethod( name: "service");
        } catch (NoSuchMethodException e) {
            throw new RuntimeException(e);
        }
        try {
            service.invoke(studentObj);
        } catch (IllegalAccessException | InvocationTargetException e) {
            throw new RuntimeException(e);
        }
        app.add(tempClass);
    }
}
```

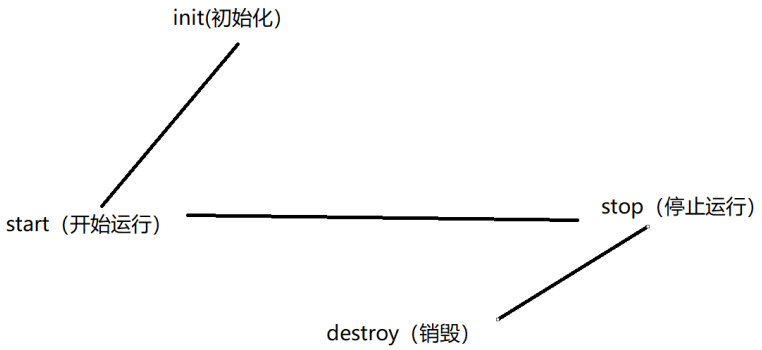
每一个 WebApp 对应一个 Context 容器，也就是上下文

Context 初始化时会读取配置文件中设置了哪些类，然后通过反射机制，来加载类，这样解耦合 WebApp 和 WebServer，并且提供了更加灵活的方法。

总结来说本项目的类加载机制是通过反射机制实现的，在反射机制上进行了封装。

5.2 生命周期的实现

本项目的重点之一就是生命周期控制，受到了 TCP 提供可靠服务的启发。



这里就实现了四个基本状态

```java
public interface LifeCycle {
    /** 第1类：针对监听器 * */
    // 添加监听器
    0 个用法  5 个实现  Kfkcome
    public void addLifecycleListener(LifecycleListener listener);
    // 获取所以监听器
    0 个用法  5 个实现  Kfkcome
    public LifecycleListener[] findLifecycleListeners();
    // 移除某个监听器
    0 个用法  5 个实现  Kfkcome
    public void removeLifecycleListener(LifecycleListener listener);

    /** 第2类：针对控制流程 * */
    // 初始化方法
    4 个用法  5 个实现  Kfkcome
    public void init() throws LifecycleException;
    // 启动方法
    2 个用法  5 个实现  Kfkcome
    public void start() throws LifecycleException;
    // 停止方法，和start对应
    0 个用法  5 个实现  Kfkcome
    public void stop() throws LifecycleException;
    // 销毁方法，和init对应
    0 个用法  5 个实现  Kfkcome
    public void destroy() throws LifecycleException;

    /** 第3类：针对状态 * */
    // 获取生命周期状态
    1 个用法  5 个实现  Kfkcome
    public LifecycleState getState();
    // 获取字符串类型的生命周期状态
    0 个用法  5 个实现  Kfkcome
    public String getStateName();

    4 个用法  1 个实现  Kfkcome
    public void setState(LifecycleState lifecycleState);

}
```

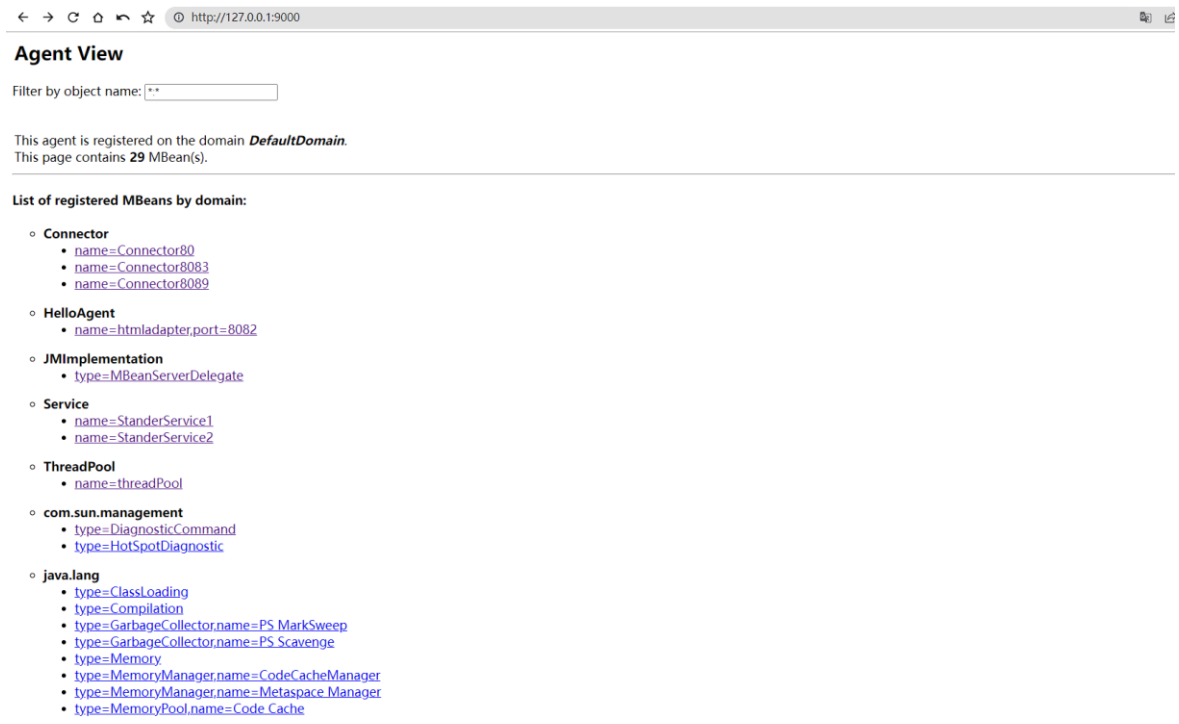
可以看到生命周期有三部分：针对后续监听器，针对流程控制，针对状态的控制的
并且这是一个接口，也就是说后续每个组件都需要根据自己的特定来实现这个接口。

5.3 监管机制的实现

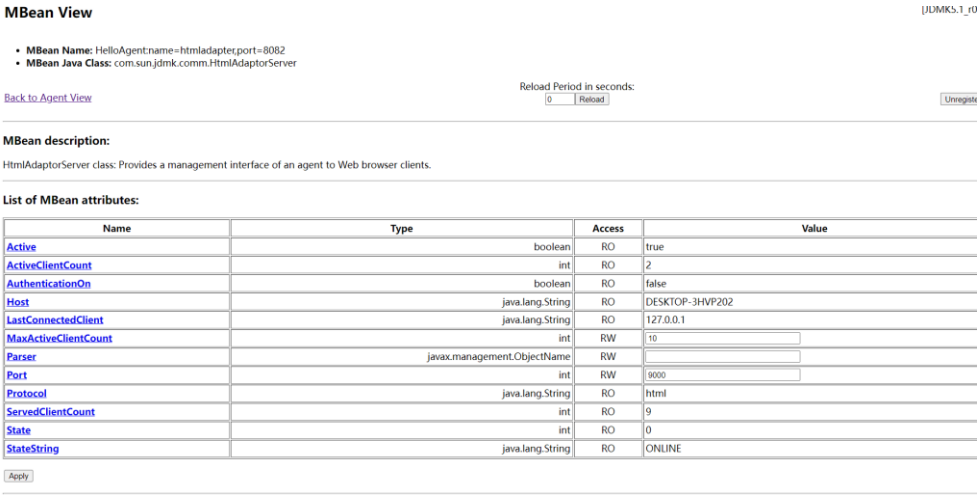
本项目最最重要的一部分监管机制的实现

首先我们监管机制是基于 JMX 的首先看效果：

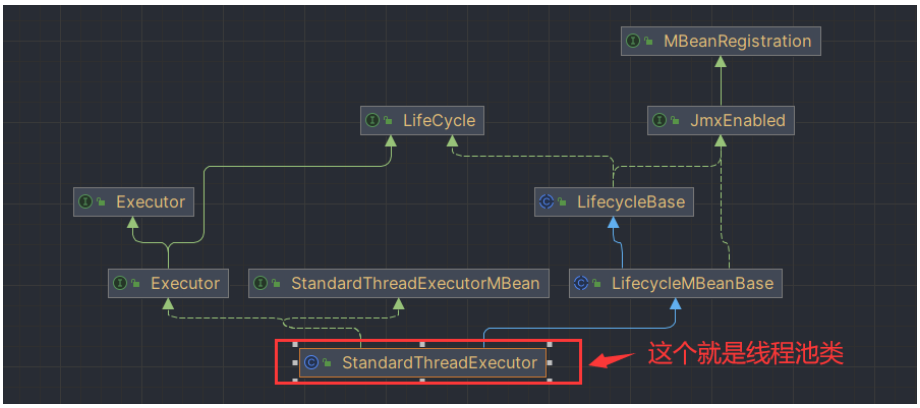
这是后台监管页面，也是一个网页，是封装在 WebServer 中的只要启动本项目的 webserver 就可以直接访问，供使用本 WebServcr 的使用者和 webapp 的开发者参考



里面可以看到各个组件的状态和占用内存



5.4 线程池实现

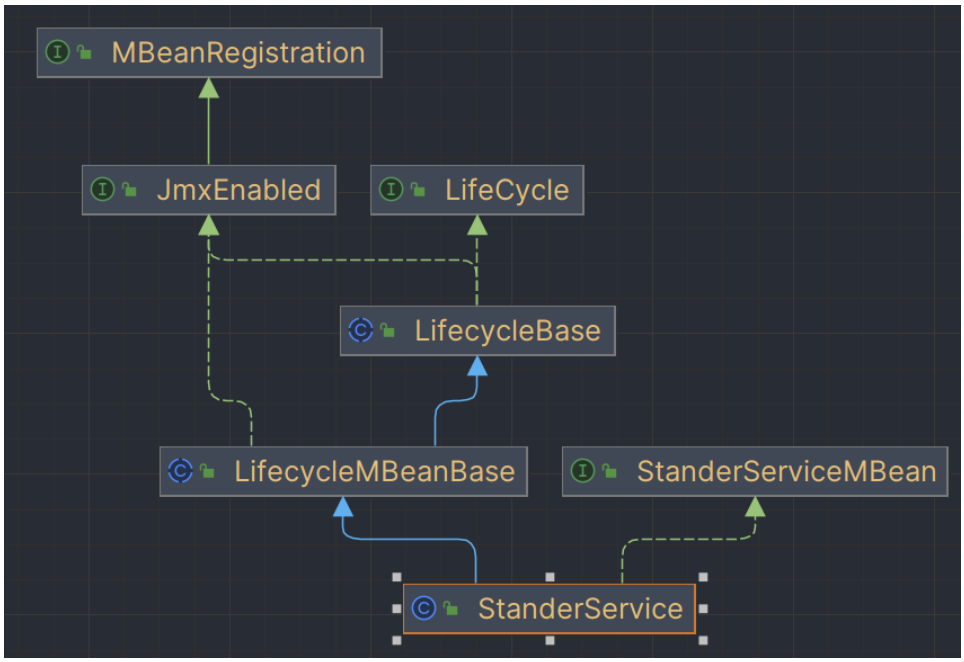


可以看到本项目的线程池实现了生命周期和监管系统的接口，也就意味着，本线程池能够进行监控和生命周期管理，其次就是封装了一般的线程池，使之实现线程队列，如果超出线程最大数量则进入队列等待，同时如果超时则删除。

```java
Kfkcome
@Override
public void execute(Runnable command) {
    if (executor != null) {
        try {
            executor.execute(command);
        } catch (RejectedExecutionException rx) {
            //there could have been contention around the queue
            if (taskqueue.add(command)) {
                Log.error( meg: "standardThreadExecutor.queueFu
            }
        }
    } else {
        Log.error( meg: "standardThreadExecutor.notStarted");
    }
```

5.5 组件的实现

Service

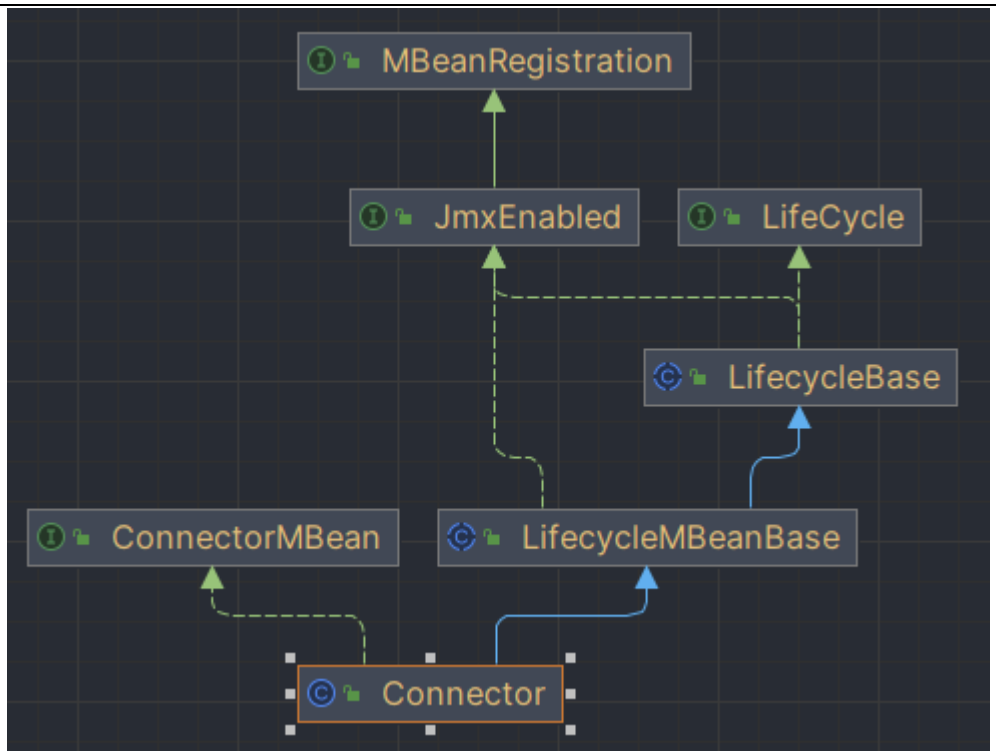


Service 组件用于为 Connector 和 Container 提供容器将两个组件连接在一起，同时也要收到的 JMX 监管和生命周期控制

首先通过 bootstrap 初始化 Service 然后根据用户提供的配置文件，然后根据配置文件初始化 Service 下的各个组件，这个传递就是通过事件监听机制传递的，察觉到 Service 的生命周期状态变化后的，在他之下的组件就自动监察，然后初始化自身。

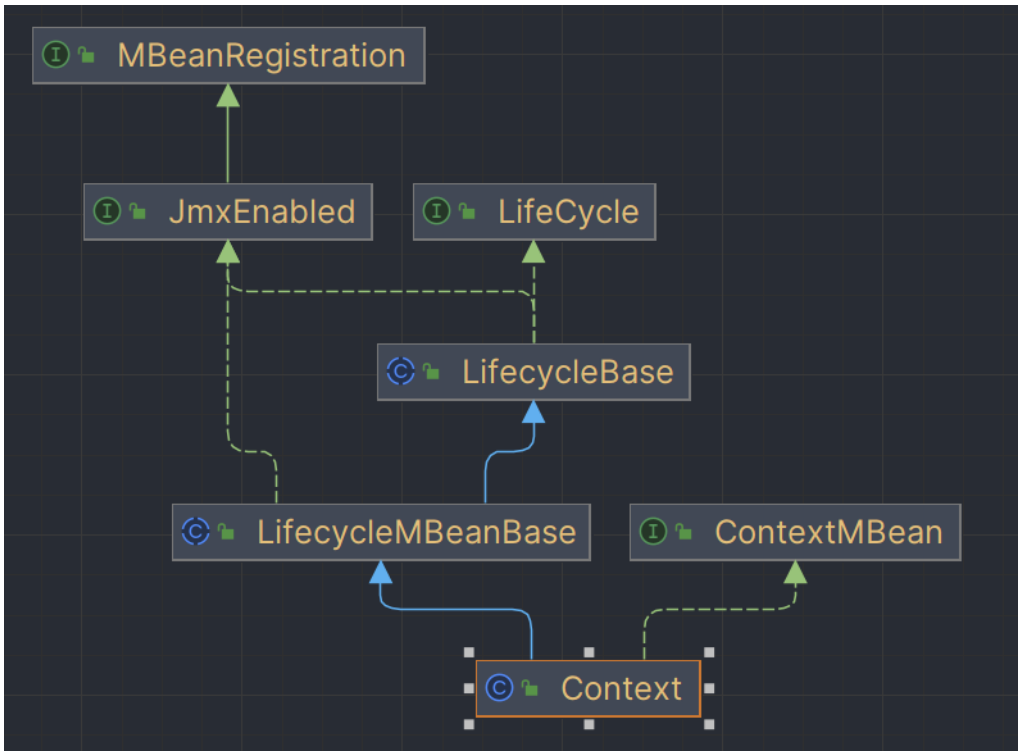
比如 Service 处于 Started 状态后 connector 就可以执行端口扫描，以为响应请求做出准备。

Connector



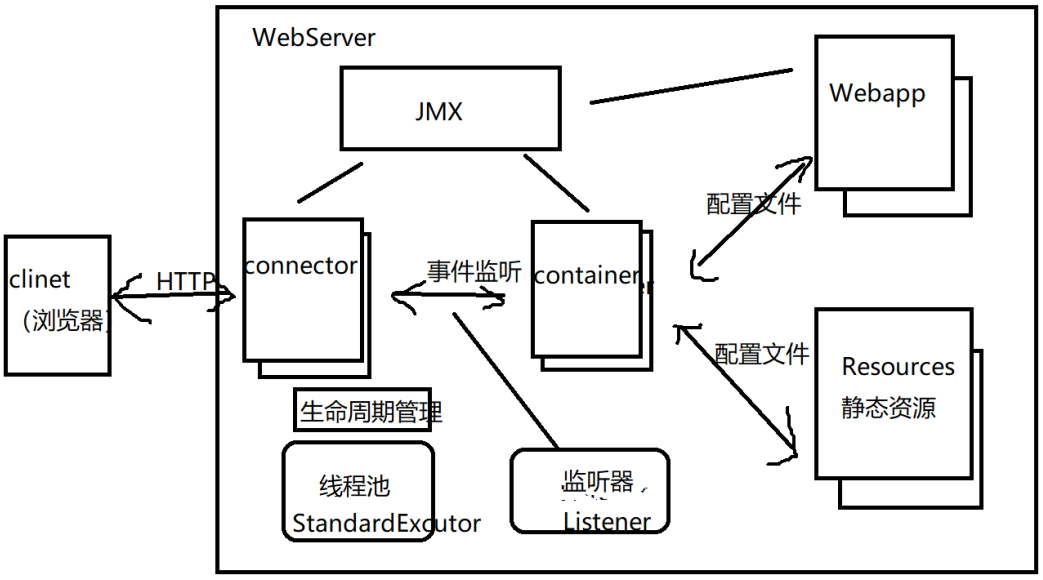
Connector 的设计模式也类似于 Service 的，只不过 Connector 是由线程池运行接管的，因为的要支持不止一个用户去访问，而限制用户的访问是在线程池中实现的。

Context



Context 的实现也于前面组件实现类似，不同的地方是，Context 要与外部 webapp 打交道，为 webapp 提供服务，也就是 Context 也必须由线程池提供服务的，以便实现多个 webapp 同时在一个 webServer 中的运行。

5.6 总体结构图



每个组件都实现了 JMX 监管和生命周期管理，我们可以通过后台网页直接的查看他们的属性和生命状态

5.7 WebServer 和 WebApp 的解耦合的实现

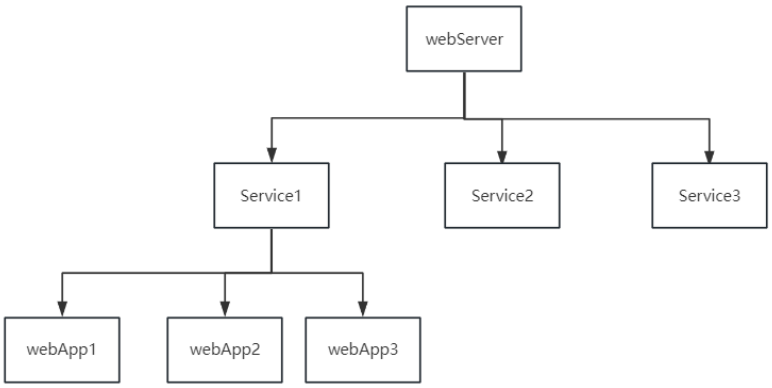
（1） 配置文件

配置文件是由我们设计的，让开发者遵循的

下面介绍一下我们设计的配置文件结构：

```xml
<?xml version = "1.0" encoding = "UTF-8" ?>
<WebServer>
    <service id="1">
        <port id="1">80</port>
        <port id="2">8083</port>
        <corePoolSize>10</corePoolSize>
        <maxiumSize>20</maxiumSize>
        <keepAliveTime>5000</keepAliveTime>
        <webapp>
            <class>com.hzau.app.test</class>
        </webapp>
    </service>
    <service id="2">
        <port id="1">8089</port>
    </service>
</WebServer>
```

Xml 的结构的



首先是配置的 Service 的一些属性比如端口号， 线程池的属性等等，然后配置 Webapp 的一些属性，比如路径等。

其中修改端口和线程池的属性，用户只用修改的配置文件就可以实现，这样让 webServer 作为一个整体不必被破坏，同时也提供了多个自定义的选项

（2） 解析配置文件

```
private void init() {
    Document document = ParsersXML.getDocumentBuilder( path: "src/main/resources/webserver.xml");
    NodeList servers = document.getElementsByTagName("service");
    for(int i=0;i<servers.getLength();i++){
        Node item = servers.item(i);
        String nodeValue = item.getAttributes().item( index: 0).getNodeValue();
        StanderService standerService = new com.hzau.Service.ServiceImpl.StanderService(Integer.parseInt(nodeValue),item.getChildNodes());
        try {
            standerService.init();
        } catch (LifecycleException e) {
            throw new RuntimeException(e);
        }
        services.add(standerService);
    }

}
```

首先在启动引导程序初始化的中读取到配置文件，然后交给 Service 进行初始化，每一个 Service 的对应一个配置节点

```
public void init() throws LifecycleException {
    Log.info( meg: id + "号Service正在初始化");
    int corePoolSize = 5;
    int maxiumSize = 10;
    int keepAliveTime = 5000;
    for (int i = 0; i < serviceValue.getLength(); i++) {
        Node item = serviceValue.item(i);
        if (item.getNodeType() == Node.ELEMENT_NODE) {
            String nodeName = item.getNodeName();
            String nodeValue = item.getFirstChild().getNodeValue();
            Log.info( meg: "读取到配置文件：" + nodeName + " : " + nodeValue);
            if (nodeName.equals("port")) {
                port.add(Integer.parseInt(nodeValue));
                connectors.add(new Connector(Integer.parseInt(nodeValue)));
            }
            if (nodeName.equals("corePoolSize"))
                corePoolSize = Integer.parseInt(nodeValue);
            if (nodeName.equals("maxiumSize"))
                maxiumSize = Integer.parseInt(nodeValue);
            if (nodeName.equals("keepAliveTime"))
                keepAliveTime = Integer.parseInt(nodeValue);
            if (nodeName.equals("webapp")) {
                NodeList childNodes = item.getChildNodes();
                contexts.add(new Context(childNodes));
            }
        }
    }
```

然后解析 Service 的一些属性，最后初始化 Context 等组件

（3） WebApp 的规范——实现接口

```
public interface WebApp {
    0 个用法  1 个实现  新 *
    void getService();
    0 个用法  1 个实现  新 *
    void putService();
    0 个用法  1 个实现  新 *
    void deleteService();
    0 个用法  1 个实现  新 *
    void postService();
}
```

5.8 实验结果：

（1）持久连接和非持久连接的代码上的区别

在非持久连接的代码上我们只要在 Connector 进行访问生成 resquest 后不进行关闭 socket，并且在 response 返回数据时直接在头中添加 constant-length，这样 webApp 开发者能够不用管其他的只要在配置文件中设置好持久连接，那么就会自动配置好的。

（4） get/put/delete/post 的实现

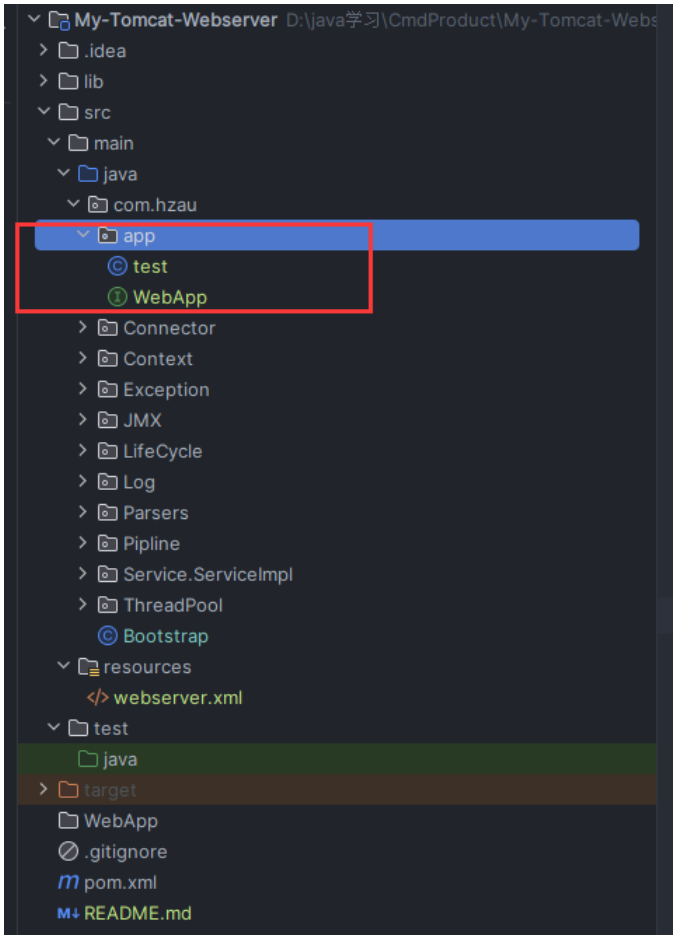
这几个的区别是 method 不同，我们可以在 connector 进行 request 封装时进行解析一下然后分别交给实现了 WebApp 接口的 webapp 的 get put delete post 方法调用封装好的 request 编写业务逻辑代码即可

```
public class HttpServletRequest extends Socket {
    0 个用法
    Map<String,String> header;
    2 个用法
    String method;
    2 个用法
    String requestURL;
    1 个用法
    String protocol;

    1 个用法  ⚠ Kfkcome
    public HttpServletRequest(Socket socket){
        InputStream inputStream = null;
        try {
            inputStream = socket.getInputStream();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        byte[] read = new byte[800];
        try {
            inputStream.read(read);

        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        String data = new String(read);
        Log.info(data);
        String[] split = data.split( regex: "\r\n");
        method= split[0].split( regex: " ")[0];
        requestURL=split[0].split( regex: " ")[1];
        protocol=split[0].split( regex: " ")[2];
```

（5） **在编写的 WebServer 中放置 webapp 测试**

放入之前作为练习实现的 2 个 webapp，一个作为前端给客户端发送页面等静态资源，一个作为后端做数据数据处理，可以实现基本的登录，注册等功能，因为这些功能都是基于 post put get delete 的 http 请求实现的，更加复杂功能也可以实现但是没有必要，因为再怎么复杂也是基于 get post delete put 实现的，**我们的重点是学习和实现 webServer 的容器而不是 WebApp。**

1. 将 webapp 放入指定文件夹



2. 将静态资源放入指定文件夹

3. 编写配置文件

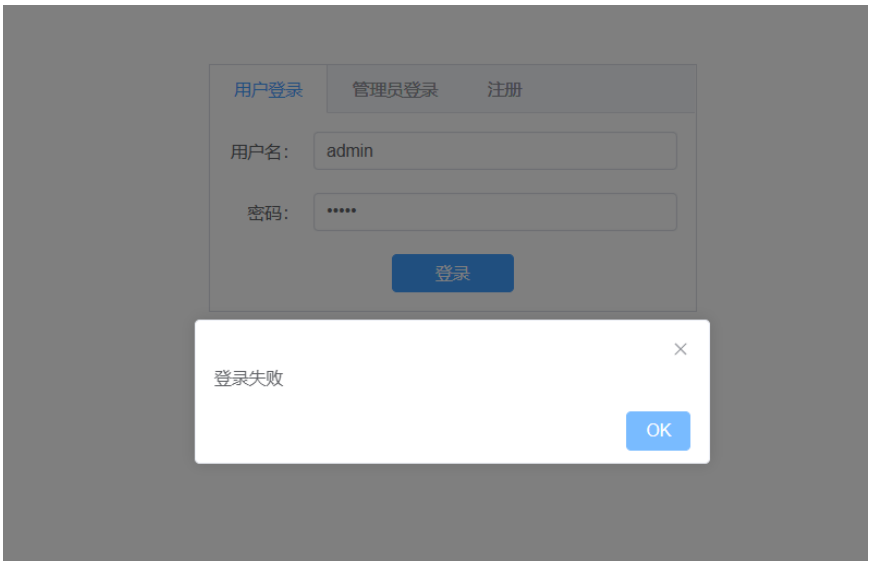
```xml
<?xml version = "1.0" encoding = "UTF-8"?>
<WebServer>
    <service id="1">
        <port id="1">80</port>
        <port id="2">8083</port>
        <corePoolSize>10</corePoolSize>
        <maxiumSize>20</maxiumSize>
        <keepAliveTime>5000</keepAliveTime>
        <webapp>
            <class>com.hzau.app.test</class>
        </webapp>
    </service>
    <service id="2">
        <port id="1">8089</port>
    </service>
</WebServer>
```

4. 运行

```
Thu Nov 16 10:30:43 CST 2023[info] : 1号Service正在初始化
Thu Nov 16 10:30:43 CST 2023[info] : 读取到配置文件：port : 80
Thu Nov 16 10:30:44 CST 2023[info] : 读取到配置文件：port : 8083
Thu Nov 16 10:30:44 CST 2023[info] : 读取到配置文件：corePoolSize : 10
Thu Nov 16 10:30:44 CST 2023[info] : 读取到配置文件：maxiumSize : 20
Thu Nov 16 10:30:44 CST 2023[info] : 读取到配置文件：keepAliveTime : 5000
Thu Nov 16 10:30:44 CST 2023[info] : 读取到配置文件：webapp :

Thu Nov 16 10:30:44 CST 2023[info] : ThreadPool:name=threadPool  成功注册进MBeanServer
Thu Nov 16 10:30:44 CST 2023[info] : Service:name=StanderService1  成功注册进MBeanServer
Thu Nov 16 10:30:44 CST 2023[info] : 80号Connect正在初始化
Thu Nov 16 10:30:44 CST 2023[info] : Connector:name=Connector80  成功注册进MBeanServer
Thu Nov 16 10:30:44 CST 2023[info] : 8083号Connect正在初始化
Thu Nov 16 10:30:44 CST 2023[info] : Connector:name=Connector8083  成功注册进MBeanServer
Thu Nov 16 10:30:44 CST 2023[info] : 服务成功执行成功
Thu Nov 16 10:30:44 CST 2023[info] : 2号Service正在初始化
Thu Nov 16 10:30:44 CST 2023[info] : 读取到配置文件：port : 8089
Thu Nov 16 10:30:44 CST 2023[warn] : lifecycleMBeanBase.registerFail : ThreadPool:name=threadPool  ThreadPool:name=threadPool
Thu Nov 16 10:30:44 CST 2023[info] : ThreadPool:name=threadPool  成功注册进MBeanServer
Thu Nov 16 10:30:44 CST 2023[info] : Service:name=StanderService2  成功注册进MBeanServer
Thu Nov 16 10:30:44 CST 2023[info] : 8089号Connect正在初始化
Thu Nov 16 10:30:44 CST 2023[info] : Connector:name=Connector8089  成功注册进MBeanServer
```

可以看到配置文件再运行时生效了，WebServer 监听了配置文件设置的端口号，启动了生命周期管理和监管系统

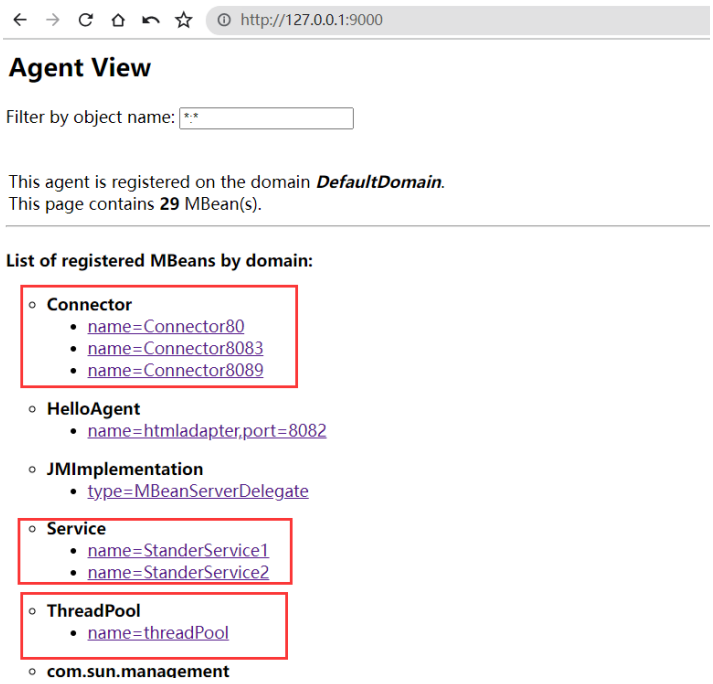
网页界面：在浏览器输入 127.0.0.1:8083(是我们刚才配置的 webapp 的端口号)



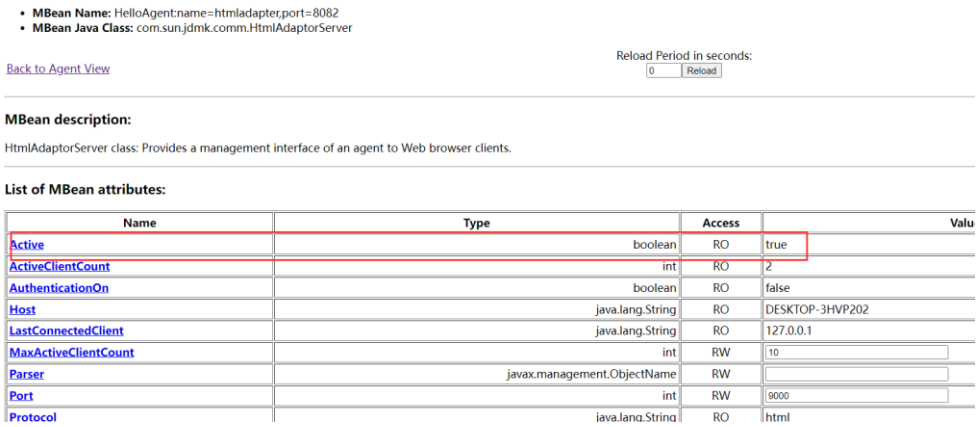
也可以发现本项目的 webserver 既可以作为前端容器也可以作为后端的容器，有一定的普

适性。因为既可以容纳静态资源，也可以容纳 webapp。

查看后台：



都已经正确注册到管理后台中



正处于活跃状态

6. 结论与分析：

6.1 遇到问题与解决：

1. JMX 使用的问题：

JMX 注册到 JMXserver 中有问题，

要符合 JMX 的命名规范（注册资源的类要在实现类后加 MBean 并且放在同一个目录中）才能正确注册

2. 持续连接的问题

刚开始在代码中实现出现问题，我认为持续连接只要不把 Socket 关闭即可，但后面运行中发现不仅需要保持 Socket 连接，还要用一个线程继续轮询有无数据发送。添加该功能后得到解决，并且在返回中还要添加 KeepAlive 的头，并且返回时还有 constant-length（不让浏览器一直查询等待。

3. Java 读取 xml 文件

一开始本来想设计 txt 作为配置文件，但是这样不够规范且容易出错，所以还是用了 xml 格式的文件，但并不会使用 Java 读取，之后在网上查找资料才学会。

4. 配置文件路径问题

一开始考虑到 webServer 的完整性，想把的配置文件自定义放到其他文件夹中，发现在更改 webserver 文件位置后，不能运行，所以最终将配置文件放在了内部，并且用相对路径访问

6.2 感想与感悟

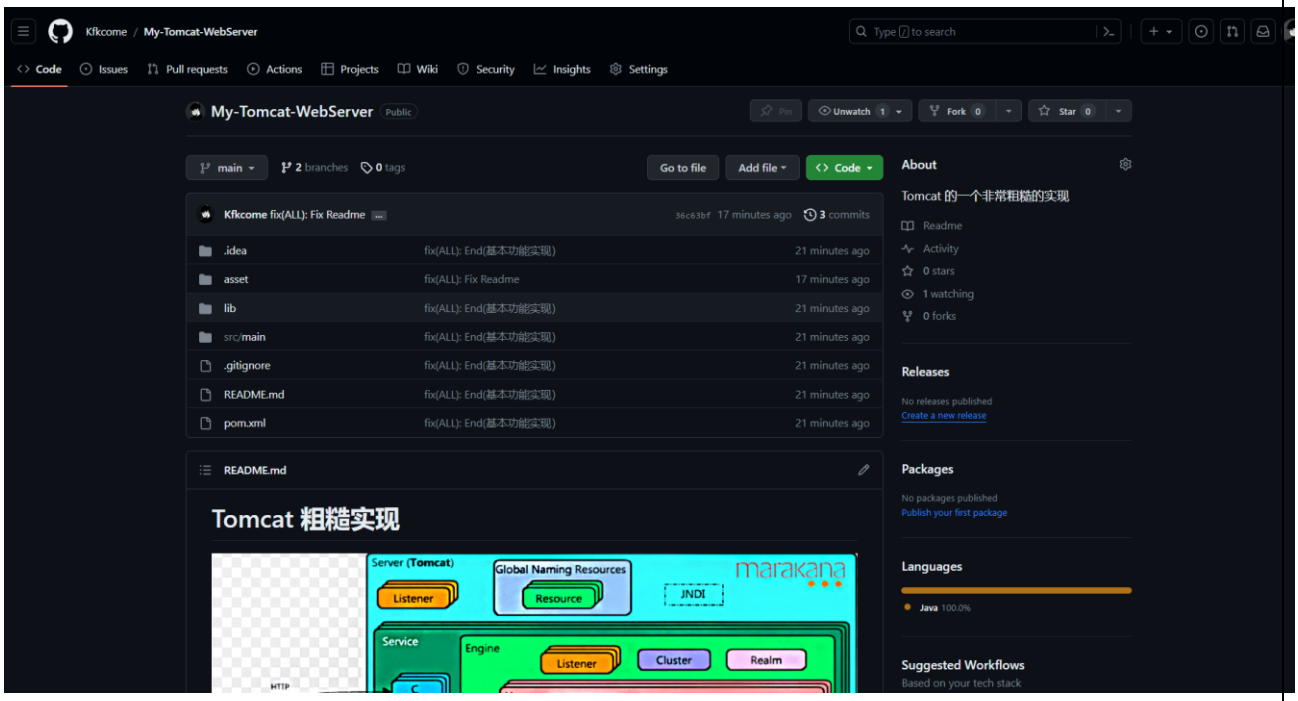
我本次实验已经实现了我之前汇报 ppt 上提到的绝大部分功能稳定安全，权限，解耦合，但是对比发展多时的成熟的 Tomcat 又显的十分简陋，Tomcat 有更厉害的分布式性能优化和监管机制，有更加强大的稳定服务，我还需要继续学习。当我在写此次大作业时，我不禁感叹，先辈的伟大，我们现在走的路都是先辈已经走过的，众多困难已经被先辈们解决了。但是这并不代表我们不需要再走一遍，我们只有走过先辈的路，才能在他们的基础上走的更远。

“学的越多，不知道的就越多”，真的是这样，虽然我之前在课下学习了一些后端知识，但是经过不断的学习的，愈发觉得的不懂的越来越多，希望以后能够不断学习，学海无涯苦作舟。

7. 项目代码

本着互联网开源的精神将完整项目代码放到了本人 github 上

https://github.com/Kfkcome/My-Tomcat-WebServer



这里由于篇幅问题只放了一部分代码，全部代码请访问上述网址：

启动引导程序 bootstrap 程序入口

```java
package com.hzau;

import com.hzau.Exception.LifecycleException;
import com.hzau.JMX.Registry;
import com.hzau.Parsers.ParsersXML;
import com.hzau.Service.ServiceImpl.StanderService;
import com.hzau.Service.ServiceImpl.StanderServiceMBean;
import com.sun.jdmk.comm.HtmlAdaptorServer;
import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

import javax.management.*;
import java.lang.management.ManagementFactory;
import java.util.ArrayList;

/**
 * 类功能描述：启动引导程序
 *
 * @author new
 * @date 2023/11/13
 */
public final class Bootstrap {
    private static final Object daemonLock = new Object();//守护进程锁
    private final ArrayList<StanderService> services = new ArrayList<>();

    private void init() {
        Document document =
ParsersXML.getDocumentBuilder("src/main/resources/webserver.xml");
        NodeList servers = document.getElementsByTagName("service");
        for(int i=0;i<servers.getLength();i++){
            Node item = servers.item(i);
            String nodeValue =
item.getAttributes().item(0).getNodeValue();
            StanderService standerService = new
com.hzau.Service.ServiceImpl.StanderService(Integer.parseInt(nodeValue
),item.getChildNodes());
            try {
                standerService.init();
            } catch (LifecycleException e) {
                throw new RuntimeException(e);
            }
            services.add(standerService);
        }
```

```
    }
    private void start(){
        for (StanderService service : services) {
            try {
                service.start();
            } catch (LifecycleException e) {
                throw new RuntimeException(e);
            }
        }
    }

    public static void main(String[] args) throws
NotCompliantMBeanException, InstanceAlreadyExistsException,
MBeanRegistrationException, MalformedObjectNameException {
        synchronized (daemonLock)//守护进程锁保证唯一性
        {

            Bootstrap bootstrap = new Bootstrap();
            bootstrap.init();
            ObjectName adapterName = new
ObjectName("HelloAgent:name=htmladapter,port=8082");
            //端口号默认 8082
            HtmlAdaptorServer adapter = new HtmlAdaptorServer();
            adapter.setPort(9000);
            Registry.getRegistry().registerMBean(adapter, adapterName);
            bootstrap.start();
            adapter.start();
        }
    }
}
```

生命周期管理

```
package LifeCycle;
import  Exception.LifecycleException;

/**
 * 类功能描述：生命周期管理
 *
 * @author new
 * @date 2023/11/14
 */
public interface LifeCycle {
    /** 第 1 类：针对监听器 **/
    // 添加监听器
    public void addLifecycleListener(LifecycleListener listener);
```

```
    // 获取所以监听器
    public LifecycleListener[] findLifecycleListeners();
    // 移除某个监听器
    public void removeLifecycleListener(LifecycleListener listener);

    /** 第 2 类：针对控制流程 **/
    // 初始化方法
    public void init() throws LifecycleException;
    // 启动方法
    public void start() throws LifecycleException;
    // 停止方法，和 start 对应
    public void stop() throws LifecycleException;
    // 销毁方法，和 init 对应
    public void destroy() throws LifecycleException;

    /** 第 3 类：针对状态 **/
    // 获取生命周期状态
    public LifecycleState getState();
    // 获取字符串类型的生命周期状态
    public String getStateName();

}
```

Service 组件部分代码

```
package com.hzau.Service.ServiceImpl;


import com.hzau.Connector.Connector;
import com.hzau.Context.Context;
import com.hzau.Exception.LifecycleException;
import com.hzau.JMX.LifecycleMBeanBase;

import com.hzau.LifeCycle.LifecycleListener;
import com.hzau.LifeCycle.LifecycleState;
import com.hzau.Log.Log;

import com.hzau.ThreadPool.StandardThreadExecutor;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.ArrayList;

public class StanderService extends LifecycleMBeanBase implements
```

```
StanderServiceMBean {
    private int id;
    NodeList serviceValue;
    ArrayList<Integer> port;
    ArrayList<Connector> connectors;
    ArrayList<Context>contexts;
    StandardThreadExecutor standardThreadExecutor;

    public StanderService(int id, NodeList serviceValue) {
        this.id = id;
        this.serviceValue = serviceValue;
        port = new ArrayList<>();
        connectors = new ArrayList<>();
        contexts=new ArrayList<>();
    }

    @Override
    protected String getDomainInternal() {
        return null;
    }

    @Override
    protected String getObjectNameKeyProperties() {
        return "name=StanderService" + id;
    }

    @Override
    public void addLifecycleListener(LifecycleListener listener) {

    }

    @Override
    public LifecycleListener[] findLifecycleListeners() {
        return new LifecycleListener[0];
    }

    @Override
    public void removeLifecycleListener(LifecycleListener listener) {

    }

    @Override
    public void init() throws LifecycleException {
        Log.info(id + "号 Service 正在初始化");
        int corePoolSize = 5;
```

```
        int maxiumSize = 10;
        int keepAliveTime = 5000;
        for (int i = 0; i < serviceValue.getLength(); i++) {
            Node item = serviceValue.item(i);
            if (item.getNodeType() == Node.ELEMENT_NODE) {
                String nodeName = item.getNodeName();
                String nodeValue = item.getFirstChild().getNodeValue();
                Log.info("读取到配置文件: " + nodeName + " : " + nodeValue);
                if (nodeName.equals("port")) {
                    port.add(Integer.parseInt(nodeValue));
                    connectors.add(new
Connector(Integer.parseInt(nodeValue)));
                }
                if (nodeName.equals("corePoolSize"))
                    corePoolSize = Integer.parseInt(nodeValue);
                if (nodeName.equals("maxiumSize"))
                    maxiumSize = Integer.parseInt(nodeValue);
                if (nodeName.equals("keepAliveTime"))
                    keepAliveTime = Integer.parseInt(nodeValue);
                if (nodeName.equals("webapp")) {
                    NodeList childNodes = item.getChildNodes();
                    contexts.add(new Context(childNodes));
                }
            }
        }
        standardThreadExecutor = new StandardThreadExecutor(corePoolSize,
maxiumSize, keepAliveTime);
        standardThreadExecutor.init();


        this.setDomain("Service");
        this.register(this, getObjectNameKeyProperties());
        for (Connector connector : connectors) {
            connector.init();
        }
        for (Context context : contexts) {
            context.init();
        }
        setState(LifecycleState.INITIALIZED);
    }

    @Override
    public void start() throws LifecycleException {
        for (Connector connector : connectors) {
```

```
            Runnable r = new Runnable() {
                @Override
                public void run() {
                    try {
                        connector.start();
                    } catch (LifecycleException e) {
                        Log.error("运行出错");
                        throw new RuntimeException(e.getMessage());
                    }
                }
            };
            standardThreadExecutor.execute(r);
            setState(LifecycleState.STARTED);
        }
    }
}
```

Connector 组件部分代码

```
package com.hzau.Connector;



import com.hzau.Exception.LifecycleException;
import com.hzau.JMX.LifecycleMBeanBase;
import com.hzau.LifeCycle.LifecycleListener;
import com.hzau.LifeCycle.LifecycleState;
import com.hzau.Log.Log;
import com.hzau.Pipline.HttpServletRequest;

import java.io.IOException;
import java.io.InputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;

;


public class Connector extends LifecycleMBeanBase implements
ConnectorMBean{

    ArrayList<LifecycleListener> listeners;
    int port;
    ServerSocket serverSocket;
    HttpServletRequest httpServletRequest;
    public Connector(int port){
```

```
        this.port=port;
    }

    @Override
    protected String getDomainInternal() {
        return "Connector";
    }

    @Override
    protected String getObjectNameKeyProperties() {
        //用时间保证唯一性
        return "name=Connector"+port;
    }

    @Override
    public void addLifecycleListener(LifecycleListener listener) {
        listeners.add(listener);
    }

    @Override
    public LifecycleListener[] findLifecycleListeners() {
        return new LifecycleListener[0];
    }

    @Override
    public void init() throws LifecycleException {
        Log.info(port + "号 Connect 正在初始化");
        register(this,getObjectNameKeyProperties());

        try {
            serverSocket = new ServerSocket(port);

        } catch (IOException e) {
            Log.error("该端口被占用无法使用");
            throw new RuntimeException(e);
        }
    }
    @Override
    public void start() throws LifecycleException {
        setState(LifecycleState.STARTED);
        try {
            Socket accept = serverSocket.accept();
            httpServletRequest=new HttpServletRequest(accept);
            Log.info(port+"端口接收到请求连接");
        } catch (IOException e) {
```

```
            throw new RuntimeException(e);
        }
    }
    @Override
    public LifecycleState getState() {
        return lifecycleState;
    }

    @Override
    public String getStateName() {
        return lifecycleState.getLifecycleEvent();
    }
    @Override
    public void removeLifecycleListener(LifecycleListener listener) {
        listeners.remove(listener);
    }
}
```

JMX 组件管理注册

```
package com.hzau.JMX;

import javax.management.MBeanServer;
import javax.management.ObjectName;

public interface MBeanRegistration  {

    // 在注册之前执行的方法，如果发生异常，MBean 不会注册到 MBean Server 中
    public ObjectName preRegister(MBeanServer server,
                                  ObjectName name) throws Exception;

    // 在注册之后执行的方法，比如注册失败提供报错信息
    public void postRegister(Boolean registrationDone);


    // 在卸载前执行的方法
    public void preDeregister() throws Exception ;

    // 在执行卸载之后的方法
    public void postDeregister();

 }
package com.hzau.JMX;

import javax.management.MBeanServer;
import javax.management.MBeanServerFactory;
```

```java
import java.lang.management.ManagementFactory;

public class Registry {
    public static MBeanServer mBeanServer;
    public static MBeanServer getRegistry()
    {
        if (mBeanServer==null){
            mBeanServer= ManagementFactory.getPlatformMBeanServer();
        }
        return mBeanServer;
    }
}
```

(针对实验结果，对其正确性、创新性进行分析；写出遇到的问题及其解决方案，本次实验心得体会）

【过程记录（源程序、测试用例、测试结果等）】