Apache Tomcat Web应用服务器课程笔记

（讲师：应癫）

主要课程内容

- 第一部分：Tomcat 系统架构与原理剖析
- 第二部分：Tomcat 服务器核心配置详解
- 第三部分：手写实现迷你版 Tomcat
- 第四部分：Tomcat 源码构建及核心流程源码剖析
- 第五部分：Tomcat 类加载机制剖析
- 第六部分：Tomcat 对 Https 的支持及 Tomcat 性能优化策略

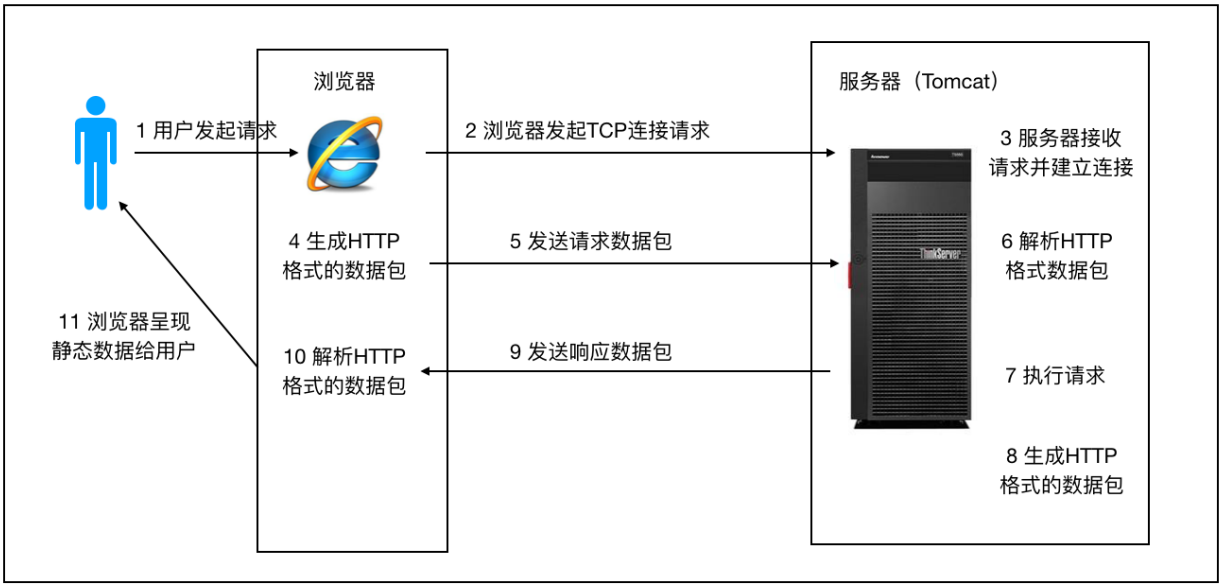
说明：本课程基于 8.5.50 版本的 Tomcat 讲解

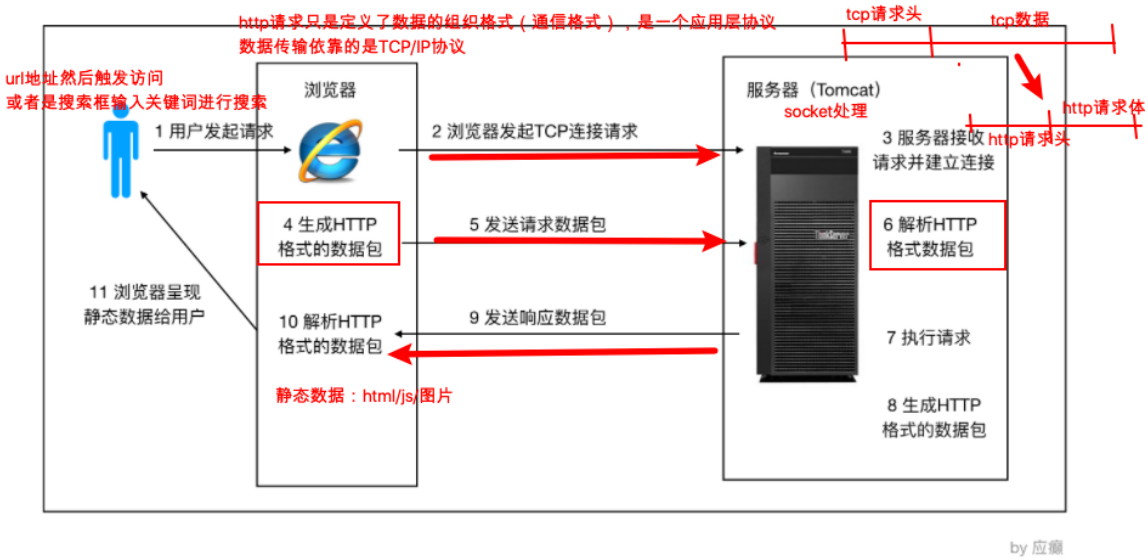
第一部分 Tomcat 系统架构与原理剖析

b/s（浏览器/服务器模式） 浏览器是客户端（发送http请求） ———> 服务器端

第 1 节 浏览器访问服务器的流程

http请求的处理过程





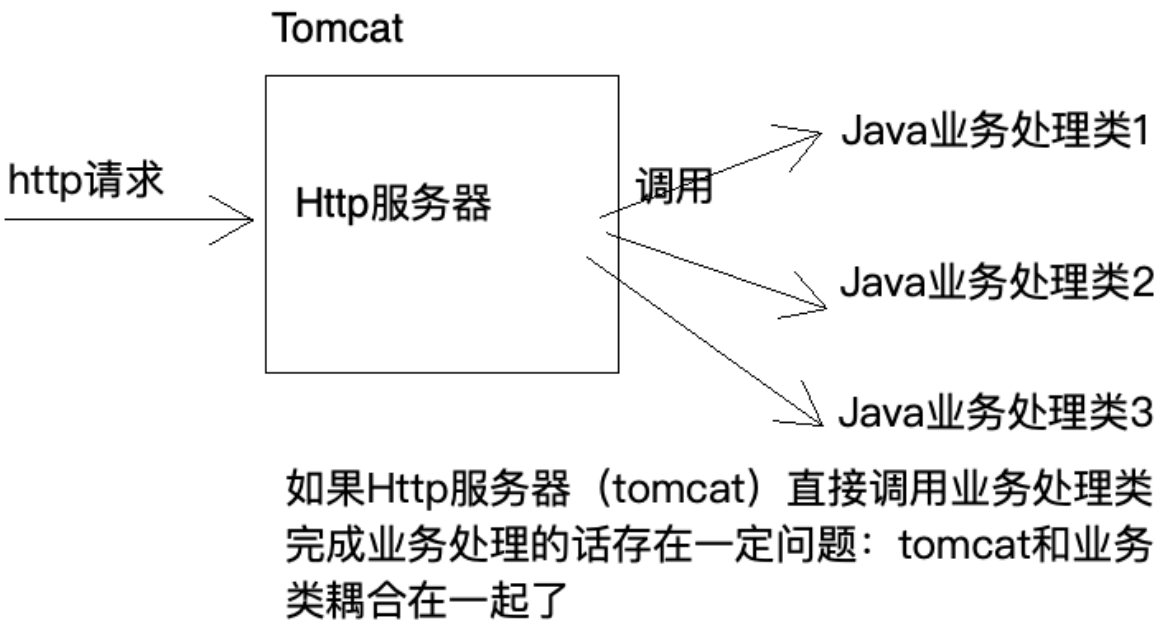
注意：浏览器访问服务器使用的是Http协议，Http是应用层协议，用于定义数据通信的格式，具体的数据传输使用的是TCP/IP协议

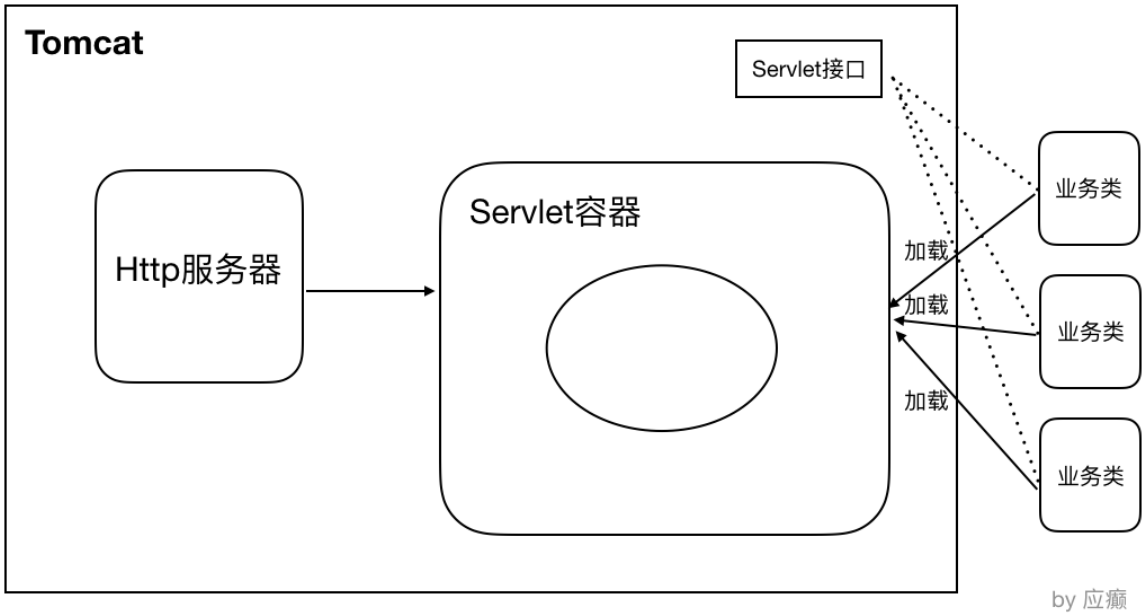
第 2 节 Tomcat 系统总体架构

2.1 Tomcat 请求处理大致过程

Tomcat是一个Http服务器（能够接收并且处理http请求，所以tomcat是一个http服务器）

我们使用浏览器向某一个网站发起请求，发出的是Http请求，那么在远程，Http服务器接收到这个请求之后，会调用具体的程序（Java类）进行处理，往往不同的请求由不同的Java类完成处理。





HTTP 服务器接收到请求之后把请求交给Servlet容器来处理，Servlet 容器通过Servlet接口调用业务类。**Servlet接口和Servlet容器这一整套内容叫作Servlet规范**。

注意：Tomcat既按照Servlet规范的要求去实现了Servlet容器，同时它也具有HTTP服务器的功能。

Tomcat的两个重要身份

1）http服务器

2）Tomcat是一个Servlet容器

2.2 Tomcat Servlet容器处理流程

当用户请求某个URL资源时

1）HTTP服务器会把请求信息使用ServletRequest对象封装起来

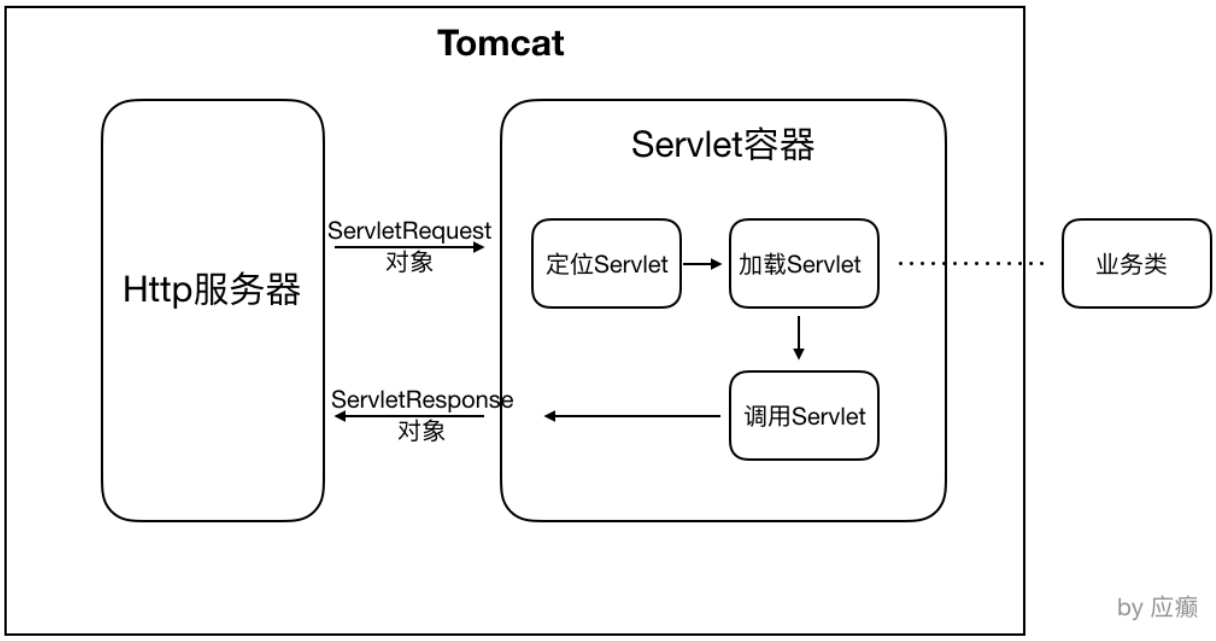
2）进一步去调用Servlet容器中某个具体的Servlet

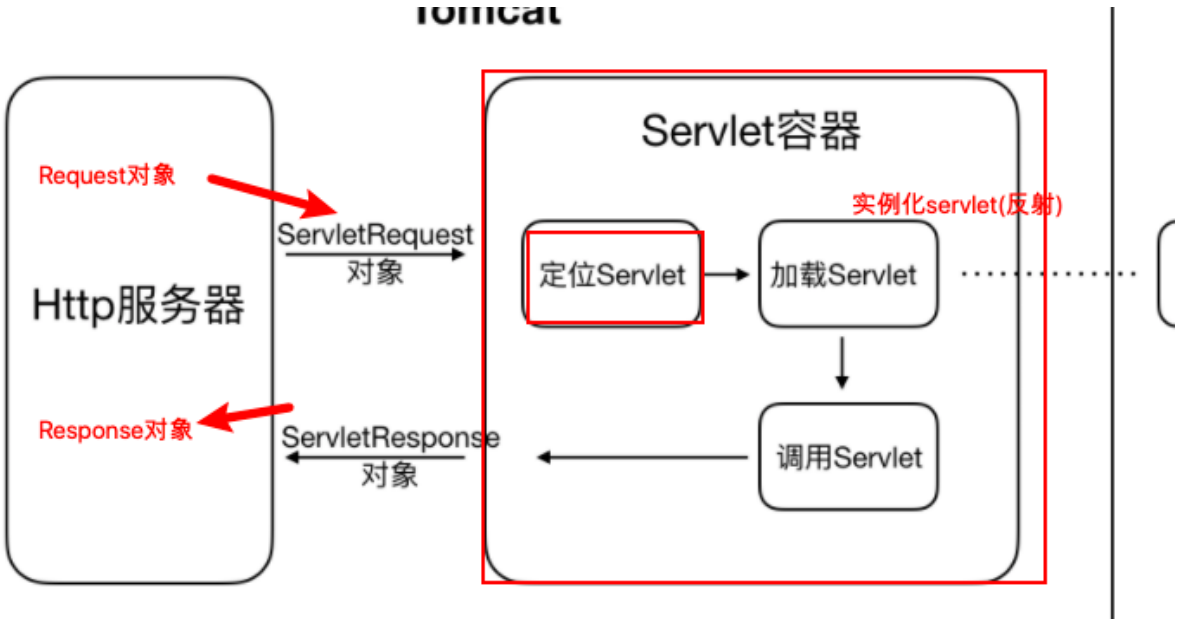
3）在 2）中，Servlet容器拿到请求后，根据URL和Servlet的映射关系，找到相应的Servlet

4）如果Servlet还没有被加载，就用反射机制创建这个Servlet，并调用Servlet的init方法来完成初始化

5）接着调用这个具体Servlet的service方法来处理请求，请求处理结果使用ServletResponse对象封装

6）把ServletResponse对象返回给HTTP服务器，HTTP服务器会把响应发送给客户端



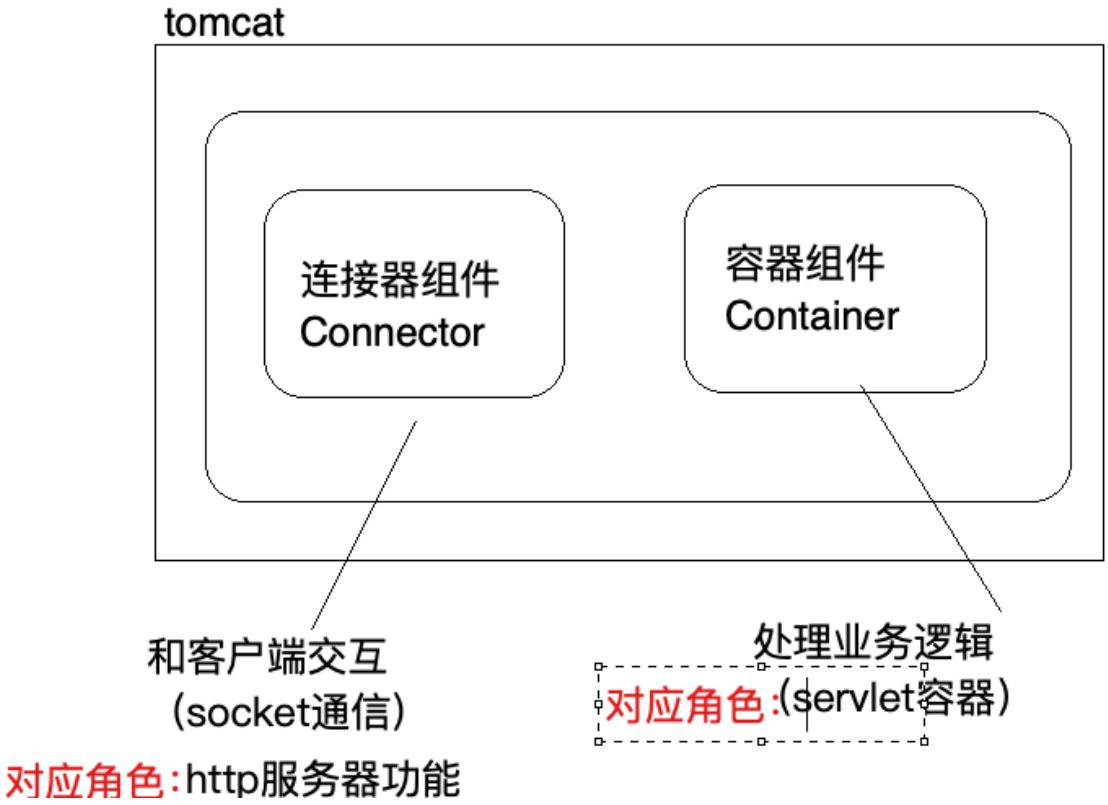


2.3 Tomcat 系统总体架构

通过上面的讲解，我们发现tomcat有两个非常重要的功能需要完成

1）和客户端浏览器进行交互，进行socket通信，将字节流和Request/Response等对象进行转换

2）Servlet容器处理业务逻辑



Tomcat 设计了两个核心组件**连接器（Connector）**和**容器（Container）**来完成 Tomcat 的两大核心功能。

连接器，负责对外交流： 处理Socket连接，负责网络字节流与Request和Response对象的转化；

容器，负责内部处理：加载和管理Servlet，以及具体处理Request请求；

第 3 节 Tomcat 连接器组件 Coyote

3.1 Coyote 简介

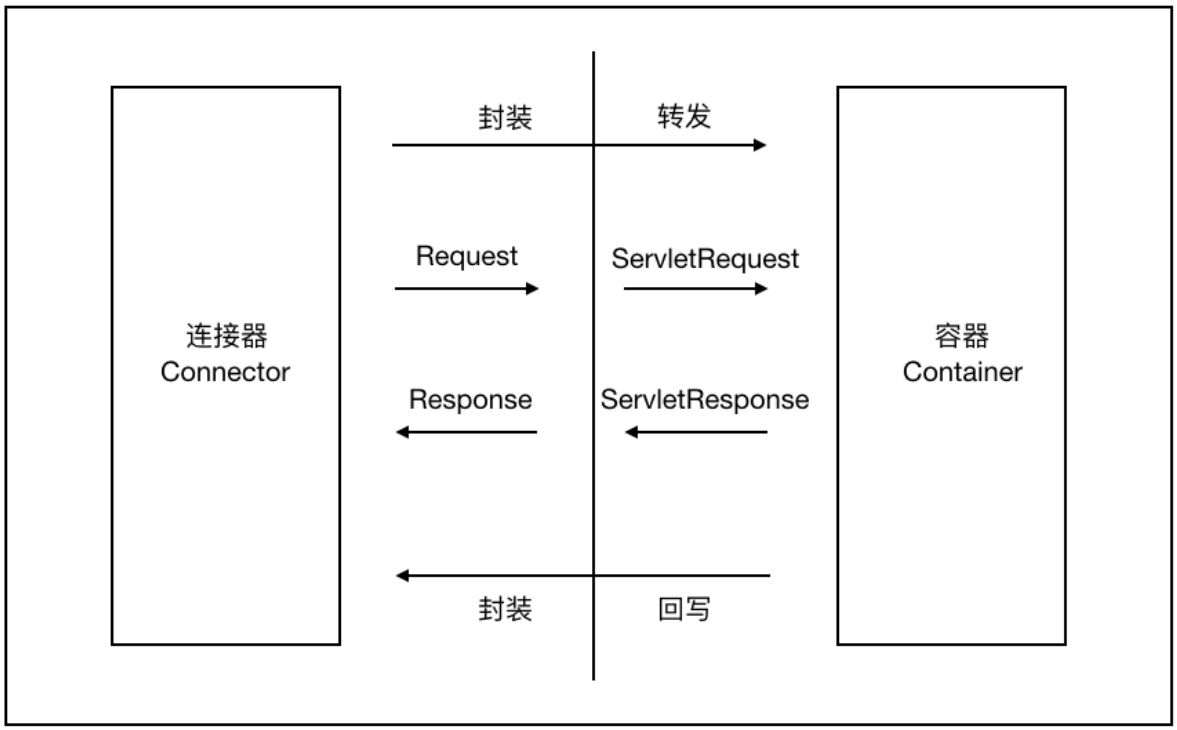
Coyote 是Tomcat 中连接器的组件名称 , 是对外的接口。客户端通过Coyote与服务器建立连接、发送请求并接受响应 。

（1）Coyote 封装了底层的网络通信（Socket 请求及响应处理）

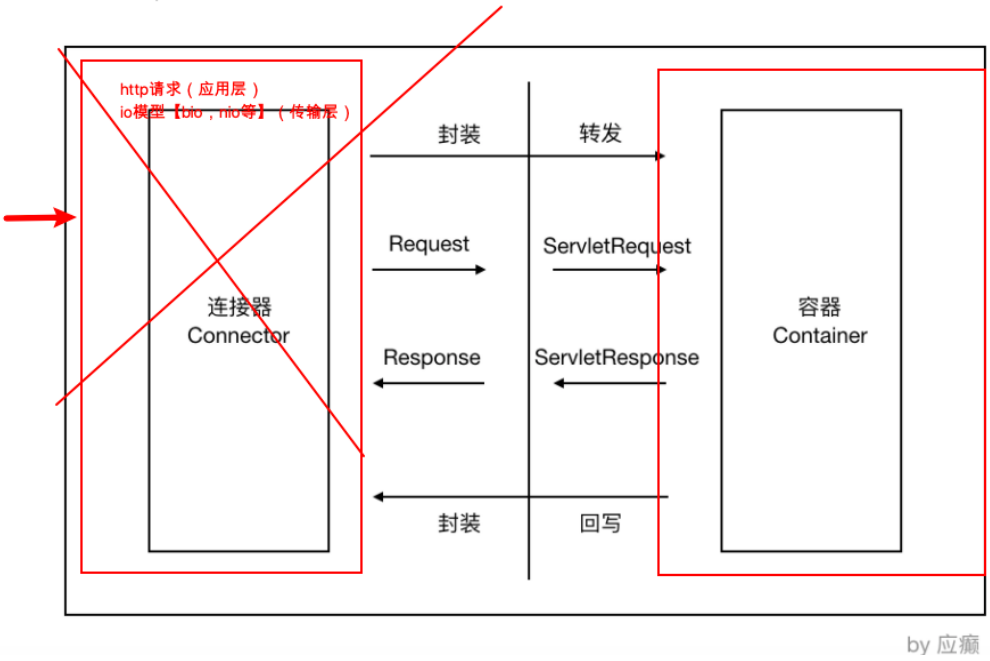
（2）Coyote 使Catalina 容器（容器组件）与具体的请求协议及IO操作方式完全解耦

（3）Coyote 将Socket 输入转换封装为 Request 对象，进一步封装后交由Catalina 容器进行处理，处理请求完成后, Catalina 通过Coyote 提供的Response 对象将结果写入输出流

（4）Coyote 负责的**是具体协议（应用层）和IO（传输层）相关内容**



by 应癫



by 应癫

Tomcat Coyote 支持的 IO模型与协议

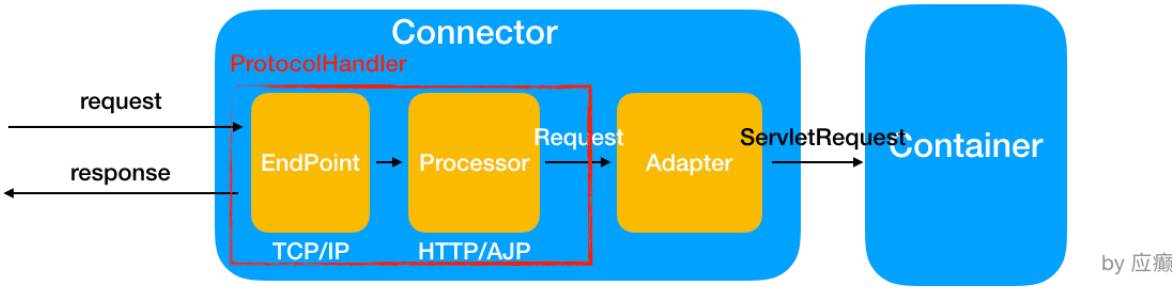
Tomcat支持多种应用层协议和I/O模型，如下：

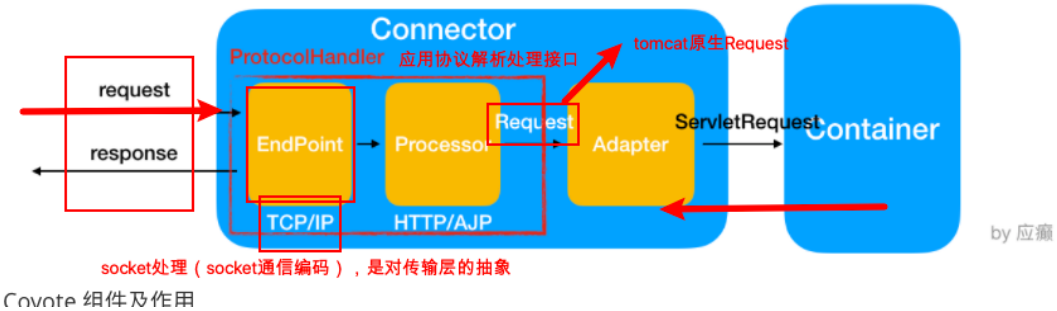
应用层	**应用层协议**	**描述**
	HTTP/1.1	这是大部分Web应用采用的访问协议。
	AJP	用于和WX集成（如Apache），以实现对静态资源的优化以及集群部署，当前支持AJP/1.3。
	HTTP/2	HTTP 2.0大幅度的提升了Web性能。下一代HTTP协议， 自8.5以及9.0版本之后支持。
传输层	**IO模型**	**描述**
	NIO	非阻塞I/O，采用Java NIO类库实现。
	NIO2	异步I/O，采用JDK 7最新的NIO2类库实现。
	APR	采用Apache可移植运行库实现，是C/C++编写的本地库。如果选择该方案，需要单独安装APR库

	应用层协议	描述
应用层	HTTP/1.1	这是大部分Web应用采用的访问协议。 默认的协议是Http/1.1 协议部分
	AJP	用于和WX集成（如Apache），以实现对静态资源的优化以及集群部署，当前支持AJP/1.3。
	HTTP/2	HTTP 2.0大幅度的提升了Web性能。下一代HTTP协议， 自8.5以及9.0版本之后支持。
传输层	IO模型	描述 Bio 同步阻塞IO 现在默认的IO模型是NIO io模型部分
	NIO	非阻塞I/O，采用Java NIO类库实现。
	NIO2	异步I/O，采用JDK 7最新的NIO2类库实现。
	APR	采用Apache可移植运行库实现，是C/C++编写的本地库。如果选择该方案，需要单独安装APR库

在 8.0 之前， Tomcat 默认采用的I/O方式为 BIO，之后改为 NIO。 无论 NIO、NIO2 还是 APR， 在性能方面均优于以往的BIO。 如果采用APR， 甚至可以达到 Apache HTTP Server 的影响性能。

3.2 Coyote 的内部组件及流程





Coyote 组件及作用

Coyote 组件及作用

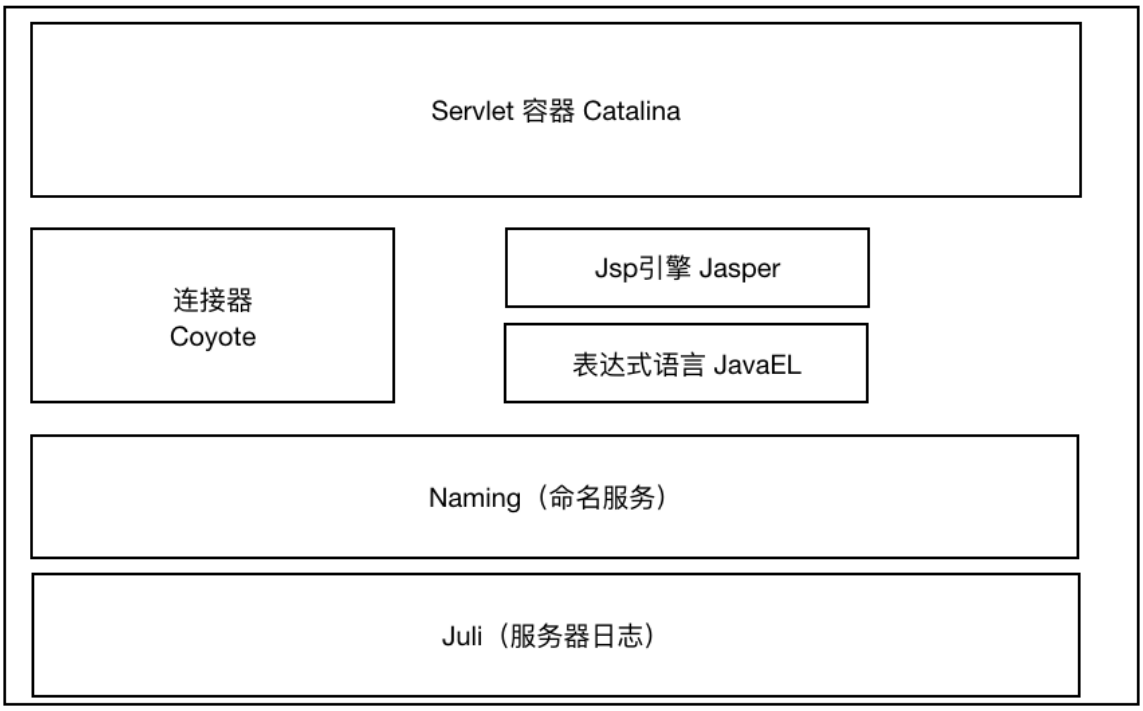
组件	作用描述
EndPoint	EndPoint 是 Coyote 通信端点，即通信监听的接口，是具体Socket接收和发送处理器，是对传输层的抽象，因此EndPoint用来实现TCP/IP协议的
Processor	Processor 是Coyote 协议处理接口 ，如果说EndPoint是用来实现TCP/IP协议的，那么Processor用来实现HTTP协议，Processor接收来自EndPoint的Socket，读取字节流解析成Tomcat Request和Response对象，并通过Adapter将其提交到容器处理，Processor是对应用层协议的抽象
ProtocolHandler	Coyote 协议接口， 通过Endpoint 和 Processor ， 实现针对具体协议的处理能力。Tomcat 按照协议和I/O 提供了6个实现类： AjpNioProtocol ，AjpAprProtocol， AjpNio2Protocol ， Http11NioProtocol ，Http11Nio2Protocol ，Http11AprProtocol
Adapter	由于协议不同，客户端发过来的请求信息也不尽相同，Tomcat定义了自己的Request类来封装这些请求信息。ProtocolHandler接口负责解析请求并生成Tomcat Request类。但是这个Request对象不是标准的ServletRequest，不能用Tomcat Request作为参数来调用容器。Tomcat设计者的解决方案是引入CoyoteAdapter，这是适配器模式的经典运用，连接器调用CoyoteAdapter的Sevice方法，传入的是Tomcat Request对象，CoyoteAdapter负责将Tomcat Request转成ServletRequest，再调用容器

第 4 节 Tomcat Servlet 容器 Catalina

4.1 Tomcat 模块分层结构图及Catalina位置

Tomcat是一个由一系列可配置（conf/server.xml）的组件构成的Web容器，而Catalina是Tomcat的servlet容器。

从另一个角度来说，**Tomcat 本质上就是一款 Servlet 容器**， 因为 Catalina 才是 Tomcat 的核心 ， 其他模块都是为Catalina 提供支撑的。 比如： 通过 Coyote 模块提供链接通信，Jasper 模块提供 JSP 引擎，Naming 提供JNDI 服务，Juli 提供日志服务。

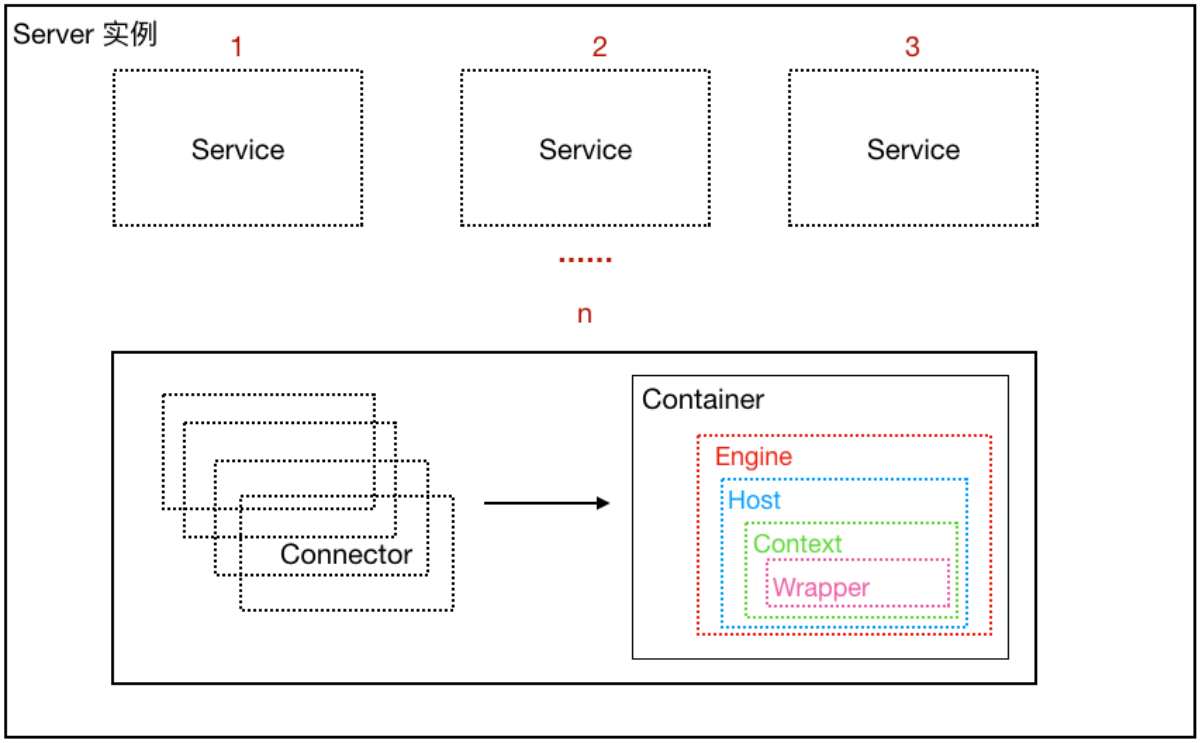


by 应癫

4.2 Servlet 容器 Catalina 的结构

Tomcat（我们往往有一个认识，Tomcat就是一个Catalina的实例，因为Catalina是Tomcat的核心）

Tomcat/Catalina实例



by 应癫

其实，可以认为整个Tomcat就是一个Catalina实例，Tomcat 启动的时候会初始化这个实例，Catalina实例通过加载server.xml完成其他实例的创建，创建并管理一个Server，Server创建并管理多个服务，每个服务又可以有多个Connector和一个Container。

一个Catalina实例（容器）

一个 Server实例（容器）

多个Service实例（容器）

每一个Service实例下可以有多个Connector实例和一个Container实例

- Catalina

  负责解析Tomcat的配置文件（server.xml），以此来创建服务器Server组件并进行管理

- Server

  服务器表示整个Catalina Servlet容器以及其它组件，负责组装并启动Servlaet引擎,Tomcat连接器。Server通过实现Lifecycle接口，提供了一种优雅的启动和关闭整个系统的方式

- Service

  服务是Server内部的组件，一个Server包含多个Service。它将若干个Connector组件绑定到一个Container

- Container

  容器，负责处理用户的servlet请求，并返回对象给web用户的模块

4.3 Container 组件的具体结构

Container组件下有几种具体的组件，分别是Engine、Host、Context和Wrapper。这4种组件（容器）是父子关系。Tomcat通过一种分层的架构，使得Servlet容器具有很好的灵活性。

- Engine

  表示整个Catalina的Servlet引擎，用来管理多个虚拟站点，一个Service最多只能有一个Engine，但是一个引擎可包含多个Host

- Host

  代表一个虚拟主机，或者说一个站点，可以给Tomcat配置多个虚拟主机地址，而一个虚拟主机下可包含多个Context

- Context

  表示一个Web应用程序， 一个Web应用可包含多个Wrapper

- Wrapper

  表示一个Servlet，Wrapper 作为容器中的最底层，不能包含子容器

  上述组件的配置其实就体现在conf/server.xml中。

第二部分 Tomcat 服务器核心配置详解

问题一：去哪儿配置？ 核心配置在tomcat目录下conf/server.xml文件

问题二：怎么配置?

注意：

- Tomcat 作为服务器的配置，主要是 server.xml 文件的配置；
- server.xml中包含了 Servlet容器的相关配置，即 Catalina 的配置；
- Xml 文件的讲解主要是标签的使用

主要标签结构如下：

```
<!--
  Server 根元素，创建一个Server实例，子标签有 Listener、GlobalNamingResources、
Service
-->
<Server>
  <!--定义监听器-->
  <Listener/>
  <!--定义服务器的全局JNDI资源 -->
  <GlobalNamingResources/>
  <!--
      定义一个Service服务，一个Server标签可以有多个Service服务实例
   -->
  <Service/>
</Server>
```

Server 标签

```
<!--
  port：关闭服务器的监听端口
  shutdown：关闭服务器的指令字符串
-->
<Server port="8005" shutdown="SHUTDOWN">
  <!-- 以日志形式输出服务器 、操作系统、JVM的版本信息 -->
  <Listener className="org.apache.catalina.startup.VersionLoggerListener" />
  <!-- Security listener. Documentation at /docs/config/listeners.html
  <Listener className="org.apache.catalina.security.SecurityListener" />
  -->
  <!--APR library loader. Documentation at /docs/apr.html -->
  <!-- 加载（服务器启动） 和 销毁 （服务器停止） APR。 如果找不到APR库， 则会输出日志， 并
不影响        Tomcat启动 -->
  <Listener className="org.apache.catalina.core.AprLifecycleListener"
SSLEngine="on" />
  <!-- Prevent memory leaks due to use of particular java/javax APIs-->
  <!-- 避免JRE内存泄漏问题 -->
  <Listener
className="org.apache.catalina.core.JreMemoryLeakPreventionListener" />
  <!-- 加载（服务器启动） 和 销毁（服务器停止） 全局命名服务 -->
  <Listener
className="org.apache.catalina.mbeans.GlobalResourcesLifecycleListener" />
  <!-- 在Context停止时重建 Executor 池中的线程， 以避免ThreadLocal 相关的内存泄漏 -->
```

```
  <Listener
className="org.apache.catalina.core.ThreadLocalLeakPreventionListener" />


  <!-- Global JNDI resources
       Documentation at /docs/jndi-resources-howto.html
       GlobalNamingResources 中定义了全局命名服务
  -->
  <GlobalNamingResources>
    <!-- Editable user database that can also be used by
         UserDatabaseRealm to authenticate users
    -->
    <Resource name="UserDatabase" auth="Container"
              type="org.apache.catalina.UserDatabase"
              description="User database that can be updated and saved"
              factory="org.apache.catalina.users.MemoryUserDatabaseFactory"
              pathname="conf/tomcat-users.xml" />
  </GlobalNamingResources>

  <!-- A "Service" is a collection of one or more "Connectors" that share
       a single "Container" Note:  A "Service" is not itself a "Container",
       so you may not define subcomponents such as "Valves" at this level.
       Documentation at /docs/config/service.html
   -->
  <Service name="Catalina">
    ...
  </Service>
</Server>
```

Service 标签

```
<!--
  该标签用于创建 Service 实例，默认使用 org.apache.catalina.core.StandardService。
  默认情况下，Tomcat 仅指定了Service 的名称， 值为 "Catalina"。
  Service 子标签为 ： Listener、Executor、Connector、Engine，
  其中：
    Listener 用于为Service添加生命周期监听器，
    Executor 用于配置Service 共享线程池，
    Connector 用于配置Service 包含的链接器，
    Engine 用于配置Service中链接器对应的Servlet 容器引擎
-->
<Service name="Catalina">
  ...
</Service>
```

Executor 标签

```
<!--
```

```
    默认情况下，Service 并未添加共享线程池配置。 如果我们想添加一个线程池， 可以在
<Service> 下添加如下配置：
    name：线程池名称，用于 Connector中指定
    namePrefix：所创建的每个线程的名称前缀，一个单独的线程名称为
namePrefix+threadNumber
    maxThreads：池中最大线程数
    minSpareThreads：活跃线程数，也就是核心池线程数，这些线程不会被销毁，会一直存在
    maxIdleTime：线程空闲时间，超过该时间后，空闲线程会被销毁，默认值为6000（1分钟），单位
毫秒
    maxQueueSize：在被执行前最大线程排队数目，默认为Int的最大值，也就是广义的无限。除非特
殊情况，这个值                    不需要更改，否则会有请求不会被处理的情况发生
    prestartminSpareThreads：启动线程池时是否启动 minSpareThreads部分线程。默认值为
false，即不启动
    threadPriority：线程池中线程优先级，默认值为5，值从1到10
    className：线程池实现类，未指定情况下，默认实现类为
org.apache.catalina.core.StandardThreadExecutor。如果想使用自定义线程池首先需要实现
org.apache.catalina.Executor接口
-->
<Executor name="commonThreadPool"
    namePrefix="thread-exec-"
    maxThreads="200"
    minSpareThreads="100"
    maxIdleTime="60000"
    maxQueueSize="Integer.MAX_VALUE"
    prestartminSpareThreads="false"
    threadPriority="5"
    className="org.apache.catalina.core.StandardThreadExecutor"/>
```

Connector 标签

Connector 标签用于创建链接器实例

默认情况下，server.xml 配置了两个链接器，一个支持HTTP协议，一个支持AJP协议

大多数情况下，我们并不需要新增链接器配置，只是根据需要对已有链接器进行优化

```
<!--
port：
  端口号，Connector 用于创建服务端Socket 并进行监听， 以等待客户端请求链接。如果该属性设置
为0，        Tomcat将会随机选择一个可用的端口号给当前Connector 使用
protocol：
  当前Connector 支持的访问协议。 默认为 HTTP/1.1 ， 并采用自动切换机制选择一个基于 JAVA
NIO 的链接器或者基于本地APR的链接器（根据本地是否含有Tomcat的本地库判定）
connectionTimeOut：
  Connector 接收链接后的等待超时时间， 单位为 毫秒。 -1 表示不超时。
redirectPort：
  当前Connector 不支持SSL请求， 接收到了一个请求， 并且也符合security-constraint 约束，
需要SSL传输，Catalina自动将请求重定向到指定的端口。
executor：
  指定共享线程池的名称， 也可以通过maxThreads、minSpareThreads 等属性配置内部线程池。
```

```
URIEncoding:
  用于指定编码URI的字符编码， Tomcat8.x版本默认的编码为 UTF-8 , Tomcat7.x版本默认为ISO-8859-1
-->
<!--org.apache.coyote.http11.Http11NioProtocol ， 非阻塞式 Java NIO 链接器-->
<Connector port="8080" protocol="HTTP/1.1" connectionTimeout="20000"
redirectPort="8443" />

<Connector port="8009" protocol="AJP/1.3" redirectPort="8443" />
```

可以使用共享线程池

```
<Connector port="8080"
           protocol="HTTP/1.1"
           executor="commonThreadPool"
           maxThreads="1000"
           minSpareThreads="100"
           acceptCount="1000"
           maxConnections="1000"
           connectionTimeout="20000"
           compression="on"
           compressionMinSize="2048"
           disableUploadTimeout="true"
           redirectPort="8443"
           URIEncoding="UTF-8" />
```

Engine 标签

Engine 表示 Servlet 引擎

```
<!--
name： 用于指定Engine 的名称， 默认为Catalina
defaultHost：默认使用的虚拟主机名称， 当客户端请求指向的主机无效时， 将交由默认的虚拟主机处理， 默认为localhost
-->
<Engine name="Catalina" defaultHost="localhost">
  ...
</Engine>
```

Host 标签

Host 标签用于配置一个虚拟主机

```
<Host name="localhost"  appBase="webapps" unpackWARs="true" autoDeploy="true">
    ...
</Host>
```

Context 标签

Context 标签用于配置一个Web应用，如下：

```
<Host name="www.abc.com"  appBase="webapps" unpackWARs="true"
autoDeploy="true">
<!--
  docBase：Web应用目录或者War包的部署路径。可以是绝对路径，也可以是相对于 Host appBase的
相对路径。
  path：Web应用的Context 路径。如果我们Host名为localhost， 则该web应用访问的根路径为：
     http://localhost:8080/web_demo。
-->
    <Context docBase="/Users/yingdian/web_demo" path="/web3"></Context>

    <Valve className="org.apache.catalina.valves.AccessLogValve"
directory="logs"
           prefix="localhost_access_log" suffix=".txt"
           pattern="%h %l %u %t &amp;quot;%r&amp;quot; %s %b" />
</Host>
```

第三部分 手写实现迷你版 Tomcat

名称：Minicat

Minicat要做的事情：作为一个服务器软件提供服务的，也即我们可以通过浏览器客户端发送http请求，Minicat可以接收到请求进行处理，处理之后的结果可以返回浏览器客户端。

1）提供服务，接收请求（Socket通信）

2）请求信息封装成Request对象（Response对象）

3）客户端请求资源，资源分为静态资源（html）和动态资源（Servlet）

4）资源返回给客户端浏览器

我们递进式完成以上需求，提出V1.0、V2.0、V3.0版本的需求

V1.0需求：浏览器请求http://localhost:8080,返回一个固定的字符串到页面"Hello Minicat!"

V2.0需求：封装Request和Response对象，返回html静态资源文件

V3.0需求：可以请求动态资源（Servlet）

完成上述三个版本后，我们的代码如下

- Bootstrap 启动类

```
package server;

import org.dom4j.Document;
import org.dom4j.DocumentException;
import org.dom4j.Element;
import org.dom4j.Node;
```

```
import org.dom4j.io.SAXReader;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.concurrent.*;

/**
 * Minicat的主类
 */
public class Bootstrap {

    /**定义socket监听的端口号*/
    private int port = 8080;

    public int getPort() {
        return port;
    }

    public void setPort(int port) {
        this.port = port;
    }


    /**
     * Minicat启动需要初始化展开的一些操作
     */
    public void start() throws Exception {

        // 加载解析相关的配置，web.xml
        loadServlet();


        // 定义一个线程池
        int corePoolSize = 10;
        int maximumPoolSize =50;
        long keepAliveTime = 100L;
        TimeUnit unit = TimeUnit.SECONDS;
        BlockingQueue<Runnable> workQueue = new ArrayBlockingQueue<>(50);
        ThreadFactory threadFactory = Executors.defaultThreadFactory();
        RejectedExecutionHandler handler = new
ThreadPoolExecutor.AbortPolicy();
```

```
        ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(
                corePoolSize,
                maximumPoolSize,
                keepAliveTime,
                unit,
                workQueue,
                threadFactory,
                handler
        );




        /*
            完成Minicat 1.0版本
            需求：浏览器请求http://localhost:8080,返回一个固定的字符串到页面"Hello Minicat!"
         */
        ServerSocket serverSocket = new ServerSocket(port);
        System.out.println("=====>>>Minicat start on port：" + port);

        /*while(true) {
            Socket socket = serverSocket.accept();
            // 有了socket，接收到请求，获取输出流
            OutputStream outputStream = socket.getOutputStream();
            String data = "Hello Minicat!";
            String responseText = HttpProtocolUtil.getHttpHeader200(data.getBytes().length) + data;
            outputStream.write(responseText.getBytes());
            socket.close();
        }*/


        /**
         * 完成Minicat 2.0版本
         * 需求：封装Request和Response对象，返回html静态资源文件
         */
        /*while(true) {
            Socket socket = serverSocket.accept();
            InputStream inputStream = socket.getInputStream();

            // 封装Request对象和Response对象
            Request request = new Request(inputStream);
            Response response = new Response(socket.getOutputStream());

            response.outputHtml(request.getUrl());
            socket.close();
```

```
        }*/


        /**
         * 完成Minicat 3.0版本
         * 需求：可以请求动态资源（Servlet）
         */
        /*while(true) {
            Socket socket = serverSocket.accept();
            InputStream inputStream = socket.getInputStream();

            // 封装Request对象和Response对象
            Request request = new Request(inputStream);
            Response response = new Response(socket.getOutputStream());

            // 静态资源处理
            if(servletMap.get(request.getUrl()) == null) {
                response.outputHtml(request.getUrl());
            }else{
                // 动态资源servlet请求
                HttpServlet httpServlet =
servletMap.get(request.getUrl());
                httpServlet.service(request,response);
            }

            socket.close();

        }
*/

        /*
            多线程改造（不使用线程池）
         */
        /*while(true) {
            Socket socket = serverSocket.accept();
            RequestProcessor requestProcessor = new
RequestProcessor(socket,servletMap);
            requestProcessor.start();
        }*/



        System.out.println("==========>>>>>>使用线程池进行多线程改造");
        /*
            多线程改造（使用线程池）
         */
        while(true) {

            Socket socket = serverSocket.accept();
```

```
            RequestProcessor requestProcessor = new
RequestProcessor(socket,servletMap);
            //requestProcessor.start();
            threadPoolExecutor.execute(requestProcessor);
        }



    }


    private Map<String,HttpServlet> servletMap = new
HashMap<String,HttpServlet>();

    /**
     * 加载解析web.xml，初始化Servlet
     */
    private void loadServlet() {
        InputStream resourceAsStream =
this.getClass().getClassLoader().getResourceAsStream("web.xml");
        SAXReader saxReader = new SAXReader();

        try {
            Document document = saxReader.read(resourceAsStream);
            Element rootElement = document.getRootElement();

            List<Element> selectNodes =
rootElement.selectNodes("//servlet");
            for (int i = 0; i < selectNodes.size(); i++) {
                Element element =  selectNodes.get(i);
                // <servlet-name>lagou</servlet-name>
                Element servletnameElement = (Element)
element.selectSingleNode("servlet-name");
                String servletName = servletnameElement.getStringValue();
                // <servlet-class>server.LagouServlet</servlet-class>
                Element servletclassElement = (Element)
element.selectSingleNode("servlet-class");
                String servletClass =
servletclassElement.getStringValue();


                // 根据servlet-name的值找到url-pattern
                Element servletMapping = (Element)
rootElement.selectSingleNode("/web-app/servlet-mapping[servlet-name='" +
servletName + "']");
                // /lagou
                String urlPattern = servletMapping.selectSingleNode("url-
pattern").getStringValue();
```

```
                servletMap.put(urlPattern, (HttpServlet)
Class.forName(servletClass).newInstance());

            }



        } catch (DocumentException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }

    }


    /**
     * Minicat 的程序启动入口
     * @param args
     */
    public static void main(String[] args) {
        Bootstrap bootstrap = new Bootstrap();
        try {
            // 启动Minicat
            bootstrap.start();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

- Http协议工具类

```
package server;

/**
 * http协议工具类，主要是提供响应头信息，这里我们只提供200和404的情况
 */
public class HttpProtocolUtil {

    /**
     * 为响应码200提供请求头信息
```

```
     * @return
     */
    public static String getHttpHeader200(long contentLength) {
        return "HTTP/1.1 200 OK \n" +
                "Content-Type: text/html \n" +
                "Content-Length: " + contentLength + " \n" +
                "\r\n";
    }

    /**
     * 为响应码404提供请求头信息(此处也包含了数据内容)
     * @return
     */
    public static String getHttpHeader404() {
        String str404 = "<h1>404 not found</h1>";
        return "HTTP/1.1 404 NOT Found \n" +
                "Content-Type: text/html \n" +
                "Content-Length: " + str404.getBytes().length + " \n" +
                "\r\n" + str404;
    }
}
```

- Request封装类

```
package server;

import java.io.IOException;
import java.io.InputStream;

/**
 * 把请求信息封装为Request对象（根据InputSteam输入流封装）
 */
public class Request {

    private String method; // 请求方式，比如GET/POST
    private String url;  // 例如 /,/index.html

    private InputStream inputStream;  // 输入流，其他属性从输入流中解析出来


    public String getMethod() {
        return method;
    }

    public void setMethod(String method) {
        this.method = method;
    }
```

```
    public String getUrl() {
        return url;
    }

    public void setUrl(String url) {
        this.url = url;
    }

    public InputStream getInputStream() {
        return inputStream;
    }

    public void setInputStream(InputStream inputStream) {
        this.inputStream = inputStream;
    }

    public Request() {
    }


    // 构造器，输入流传入
    public Request(InputStream inputStream) throws IOException {
        this.inputStream = inputStream;

        // 从输入流中获取请求信息
        int count = 0;
        while (count == 0) {
            count = inputStream.available();
        }

        byte[] bytes = new byte[count];
        inputStream.read(bytes);

        String inputStr = new String(bytes);
        // 获取第一行请求头信息
        String firstLineStr = inputStr.split("\\n")[0];  // GET / HTTP/1.1

        String[] strings = firstLineStr.split(" ");

        this.method = strings[0];
        this.url = strings[1];

        System.out.println("=====>>method:" + method);
        System.out.println("=====>>url:" + url);


    }
}
```

- Response封装类

```
package server;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.OutputStream;

/**
 * 封装Response对象，需要依赖于OutputStream
 *
 * 该对象需要提供核心方法，输出html
 */
public class Response {

    private OutputStream outputStream;

    public Response() {
    }

    public Response(OutputStream outputStream) {
        this.outputStream = outputStream;
    }


    // 使用输出流输出指定字符串
    public void output(String content) throws IOException {
        outputStream.write(content.getBytes());
    }


    /**
     *
     * @param path  url，随后要根据url来获取到静态资源的绝对路径，进一步根据绝对路径读取该静态资源文件，最终通过
     *                    输出流输出
     *               /-----> classes
     */
    public void outputHtml(String path) throws IOException {
        // 获取静态资源文件的绝对路径
        String absoluteResourcePath = StaticResourceUtil.getAbsolutePath(path);

        // 输入静态资源文件
        File file = new File(absoluteResourcePath);
        if(file.exists() && file.isFile()) {
            // 读取静态资源文件，输出静态资源
```

```
            StaticResourceUtil.outputStaticResource(new
FileInputStream(file),outputStream);
        }else{
            // 输出404
            output(HttpProtocolUtil.getHttpHeader404());
        }

    }

}
```

- 静态资源请求处理工具类

```
package server;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class StaticResourceUtil {

    /**
     * 获取静态资源文件的绝对路径
     * @param path
     * @return
     */
    public static String getAbsolutePath(String path) {
        String absolutePath =
StaticResourceUtil.class.getResource("/").getPath();
        return absolutePath.replaceAll("\\\\","/") + path;
    }


    /**
     * 读取静态资源文件输入流，通过输出流输出
     */
    public static void outputStaticResource(InputStream inputStream,
OutputStream outputStream) throws IOException {

        int count = 0;
        while(count == 0) {
            count = inputStream.available();
        }

        int resourceSize = count;
        // 输出http请求头,然后再输出具体内容
```

```
 outputStream.write(HttpProtocolUtil.getHttpHeader200(resourceSize).getBytes());

        // 读取内容输出
        long written = 0 ;// 已经读取的内容长度
        int byteSize = 1024; // 计划每次缓冲的长度
        byte[] bytes = new byte[byteSize];

        while(written < resourceSize) {
            if(written  + byteSize > resourceSize) {  // 说明剩余未读取大小不足一个1024长度，那就按真实长度处理
                byteSize = (int) (resourceSize - written);  // 剩余的文件内容长度
                bytes = new byte[byteSize];
            }

            inputStream.read(bytes);
            outputStream.write(bytes);

            outputStream.flush();
            written+=byteSize;
        }
    }
}
```

- 动态资源请求
  - Servlet接口定义

```
package server;

public interface Servlet {

    void init() throws Exception;

    void destory() throws Exception;

    void service(Request request,Response response) throws Exception;
}
```

  - HttpServlet抽象类定义

```
package server;

public abstract class HttpServlet implements Servlet{
```

```
    public abstract void doGet(Request request,Response response);

    public abstract void doPost(Request request,Response response);


    @Override
    public void service(Request request, Response response) throws
Exception {
        if("GET".equalsIgnoreCase(request.getMethod())) {
            doGet(request,response);
        }else{
            doPost(request,response);
        }
    }
}
```

- 业务类Servlet定义LagouServlet

```
package server;

import java.io.IOException;

public class LagouServlet extends HttpServlet {
    @Override
    public void doGet(Request request, Response response) {


        try {
            Thread.sleep(100000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        String content = "<h1>LagouServlet get</h1>";
        try {

 response.output((HttpProtocolUtil.getHttpHeader200(content.getBytes()
.length) + content));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void doPost(Request request, Response response) {
        String content = "<h1>LagouServlet post</h1>";
        try {
```

```
 response.output((HttpProtocolUtil.getHttpHeader200(content.getBytes()
.length) + content));
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

        @Override
        public void init() throws Exception {

        }

        @Override
        public void destory() throws Exception {

        }
}
```

- 多线程改造封装的RequestProcessor类

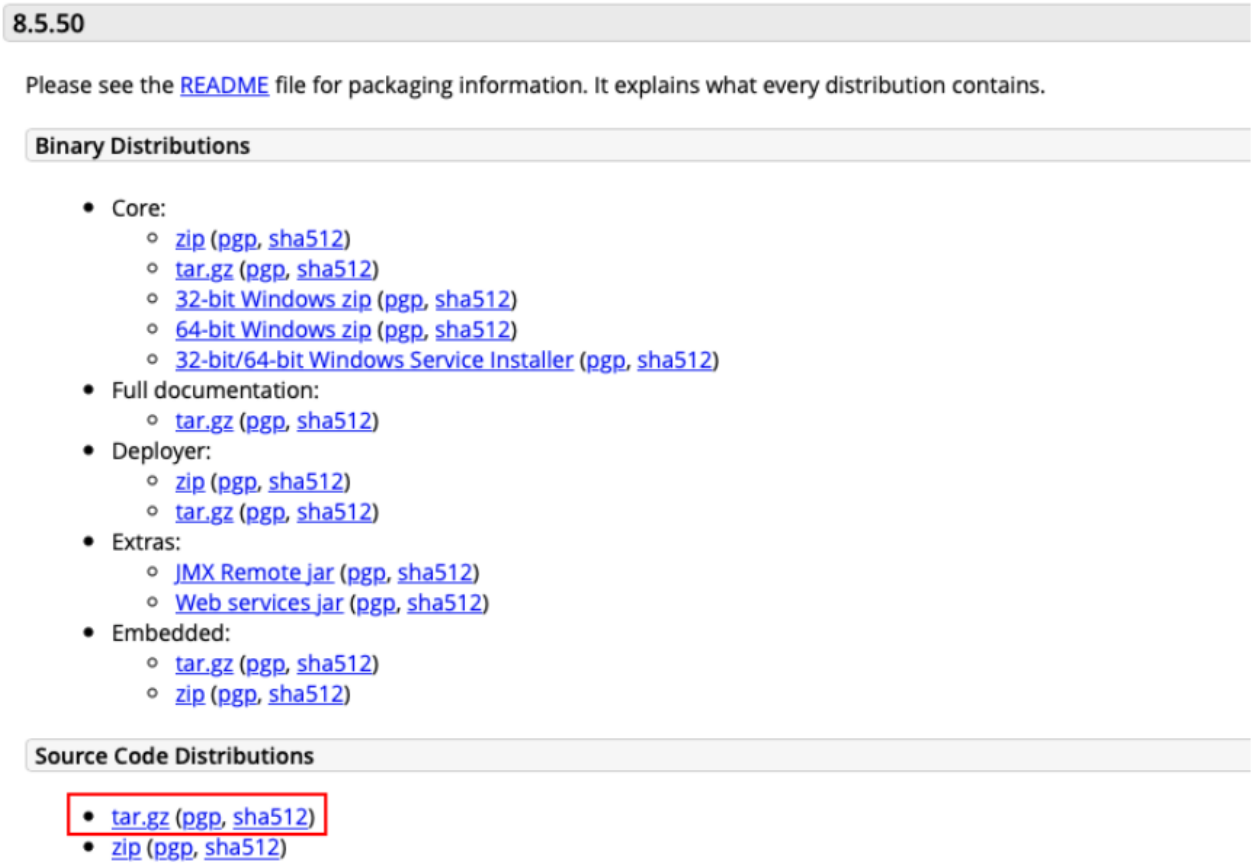
```
package server;

import java.io.InputStream;
import java.net.Socket;
import java.util.Map;

public class RequestProcessor extends Thread {

    private Socket socket;
    private Map<String,HttpServlet> servletMap;

    public RequestProcessor(Socket socket, Map<String, HttpServlet>
servletMap) {
        this.socket = socket;
        this.servletMap = servletMap;
    }

    @Override
    public void run() {
        try{
            InputStream inputStream = socket.getInputStream();

            // 封装Request对象和Response对象
            Request request = new Request(inputStream);
            Response response = new Response(socket.getOutputStream());

            // 静态资源处理
```

```
            if(servletMap.get(request.getUrl()) == null) {
                response.outputHtml(request.getUrl());
            }else{
                // 动态资源servlet请求
                HttpServlet httpServlet =
servletMap.get(request.getUrl());
                httpServlet.service(request,response);
            }

            socket.close();

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

    }
}
```

第四部分 Tomcat 源码构建及核心流程源码剖析

第 1 节 源码构建

1.1 下载源码



1.2 源码导入IDE之前准备工作

- 解压 tar.gz 压缩包，得到目录 apache-tomcat-8.5.50-src
- 进入 apache-tomcat-8.5.50-src 目录，创建一个pom.xml文件，文件内容如下

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>org.apache.tomcat</groupId>
    <artifactId>apache-tomcat-8.5.50-src</artifactId>
    <name>Tomcat8.5</name>
    <version>8.5</version>

    <build>
        <!--指定源目录-->
        <finalName>Tomcat8.5</finalName>
        <sourceDirectory>java</sourceDirectory>
        <resources>
            <resource>
                <directory>java</directory>
            </resource>
        </resources>
        <plugins>
            <!--引入编译插件-->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.1</version>
                <configuration>
                    <encoding>UTF-8</encoding>
                    <source>11</source>
                    <target>11</target>
                </configuration>
            </plugin>
        </plugins>
    </build>

    <!--tomcat 依赖的基础包-->
    <dependencies>
        <dependency>
            <groupId>org.easymock</groupId>
            <artifactId>easymock</artifactId>
            <version>3.4</version>
        </dependency>
        <dependency>
            <groupId>ant</groupId>
            <artifactId>ant</artifactId>
```

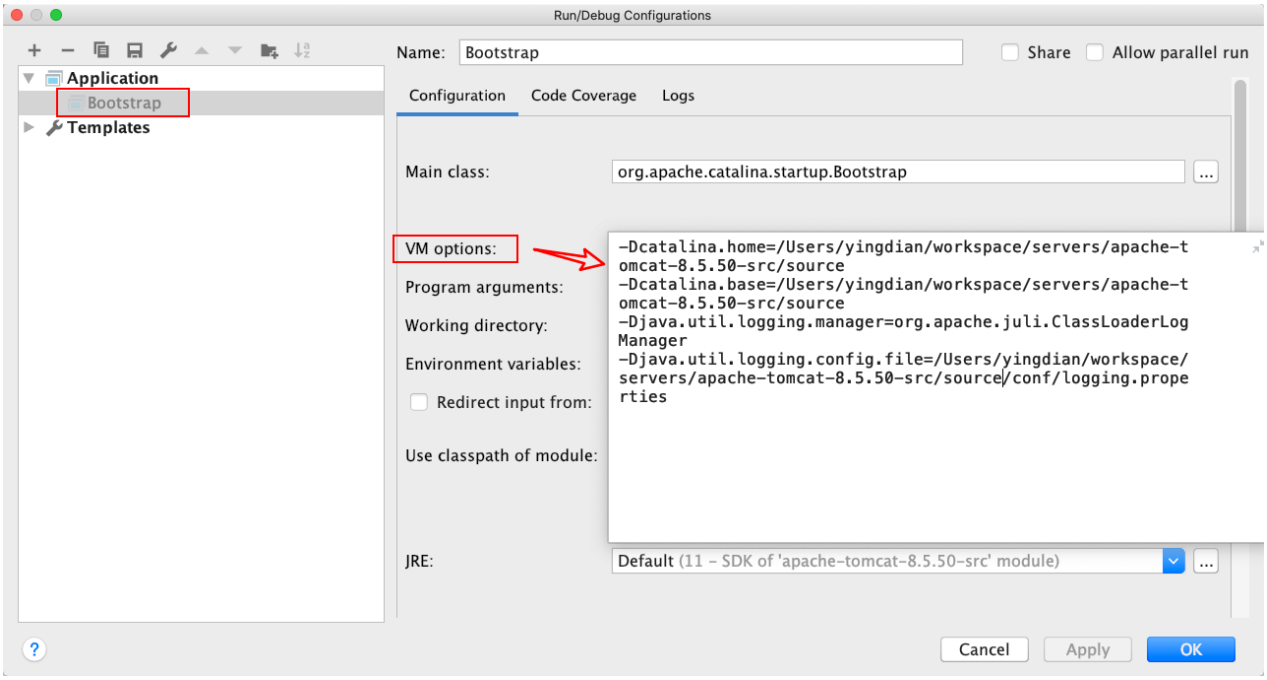
```
            <version>1.7.0</version>
        </dependency>
        <dependency>
            <groupId>wsdl4j</groupId>
            <artifactId>wsdl4j</artifactId>
            <version>1.6.2</version>
        </dependency>
        <dependency>
            <groupId>javax.xml</groupId>
            <artifactId>jaxrpc</artifactId>
            <version>1.1</version>
        </dependency>
        <dependency>
            <groupId>org.eclipse.jdt.core.compiler</groupId>
            <artifactId>ecj</artifactId>
            <version>4.5.1</version>
        </dependency>
        <dependency>
            <groupId>javax.xml.soap</groupId>
            <artifactId>javax.xml.soap-api</artifactId>
            <version>1.4.0</version>
        </dependency>
    </dependencies>
</project>
```

- 在 apache-tomcat-8.5.50-src 目录中创建 **source** 文件夹
- 将 conf、webapps 目录移动到刚刚创建的 **source** 文件夹中

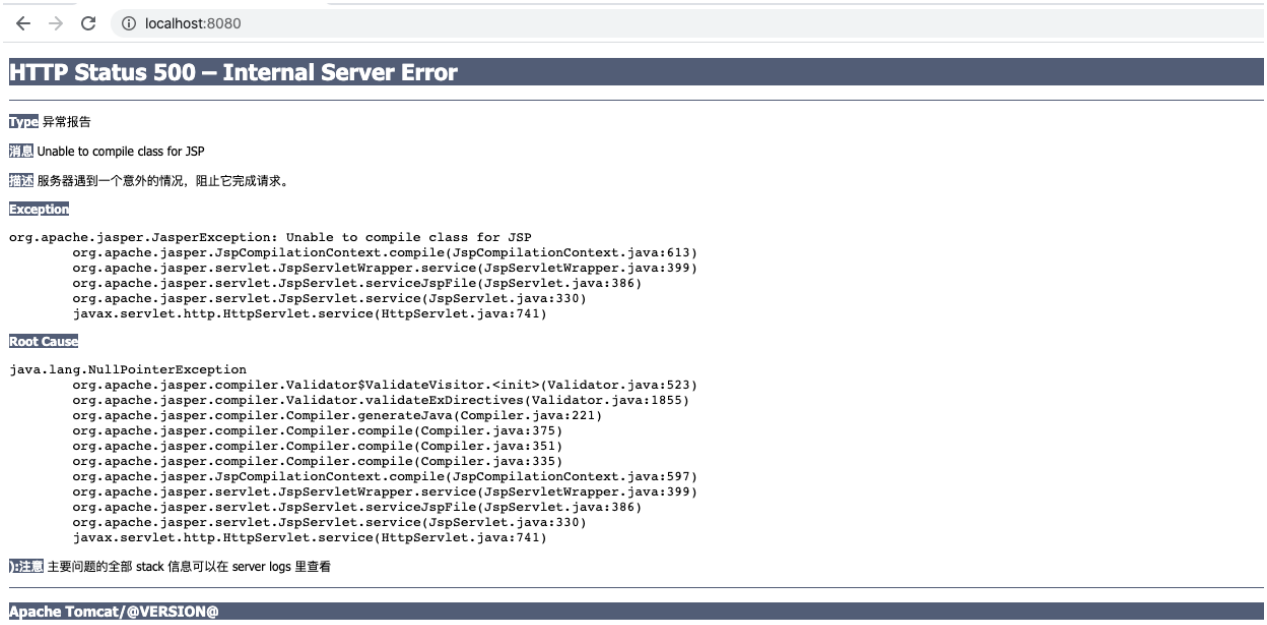
1.3 导入源码工程到IDE并进行配置

- 将源码工程导入到 IDEA 中
- 给 tomcat 的源码程序启动类 Bootstrap 配置 VM 参数，因为 tomcat 源码运行也需要加载配置文件等。

```
-Dcatalina.home=/Users/yingdian/workspace/servers/apache-tomcat-8.5.50-src/source
-Dcatalina.base=/Users/yingdian/workspace/servers/apache-tomcat-8.5.50-src/source
-Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager
-Djava.util.logging.config.file=/Users/yingdian/workspace/servers/apache-tomcat-8.5.50-src/source/conf/logging.properties
```



- 运行 Bootstrap 类的 main 函数，此时就启动了tomcat，启动时候会去加载所配置的 conf 目录下的server.xml等配置文件，所以访问8080端口即可，但此时我们会遇到如下的一个错误



原因是Jsp引擎Jasper没有被初始化，从而无法编译JSP，我们需要在tomcat的源码ContextConfig类中的configureStart方法中增加一行代码将 Jsp 引擎初始化，如下

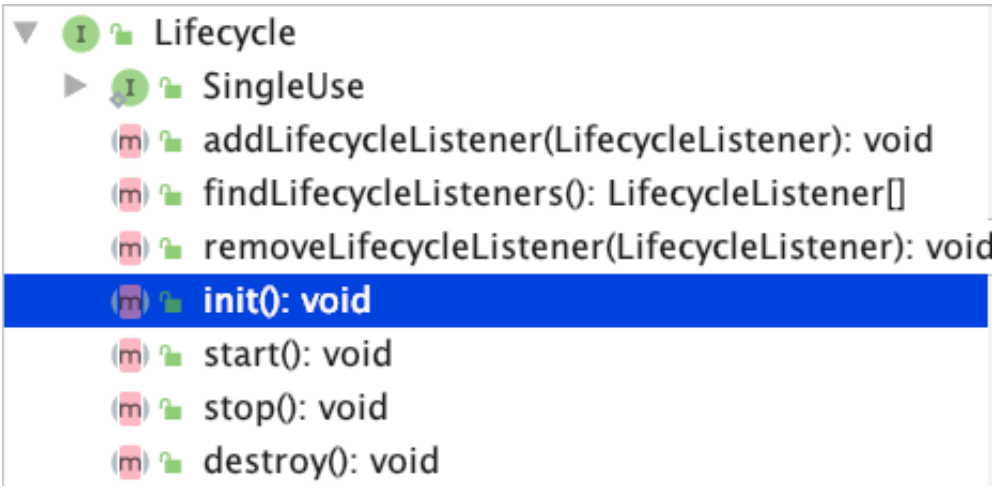
```java
/**
 * Process a "contextConfig" event for this Context.
 */
protected synchronized void configureStart() {
    // Called from StandardContext.start()

    if (log.isDebugEnabled()) {
        log.debug(sm.getString( key: "contextConfig.start"));
    }

    if (log.isDebugEnabled()) {
        log.debug(sm.getString( key: "contextConfig.xmlSettings",
                context.getName(),
                Boolean.valueOf(context.getXmlValidation()),
                Boolean.valueOf(context.getXmlNamespaceAware())));
    }

    webConfig();

    // 初始化jsp解析引擎-jasper
    context.addServletContainerInitializer(new JasperInitializer(),  classes: null);
```

- 重启 Tomcat，正常访问即可。至此，Tomcat 源码构建完毕。

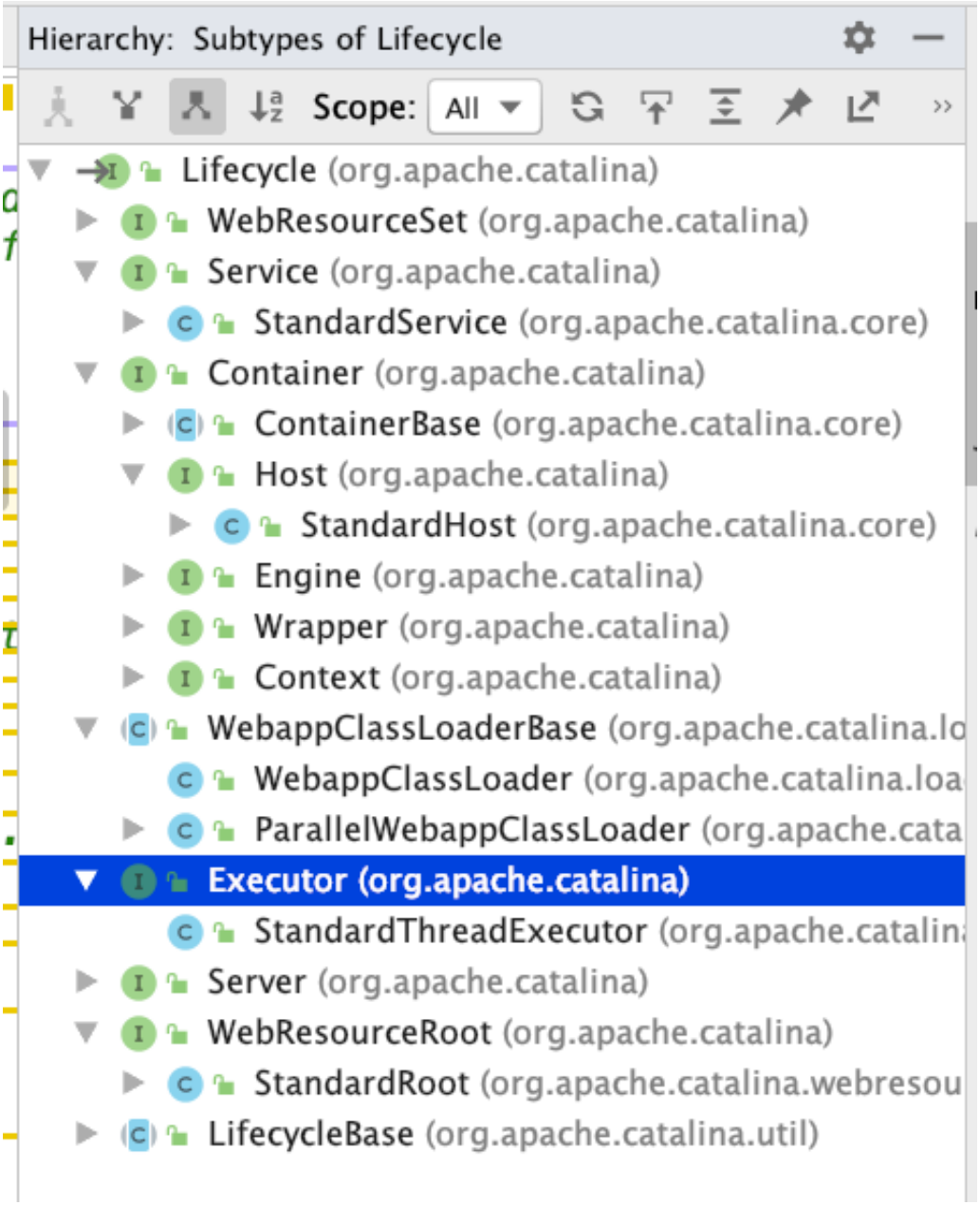
第 2 节 核心流程源码剖析

Tomcat中的各容器组件都会涉及创建、销毁等，因此设计了生命周期接口Lifecycle进行统一规范，各容器组件实现该接口。

Lifecycle生命周期接口主要方法示意



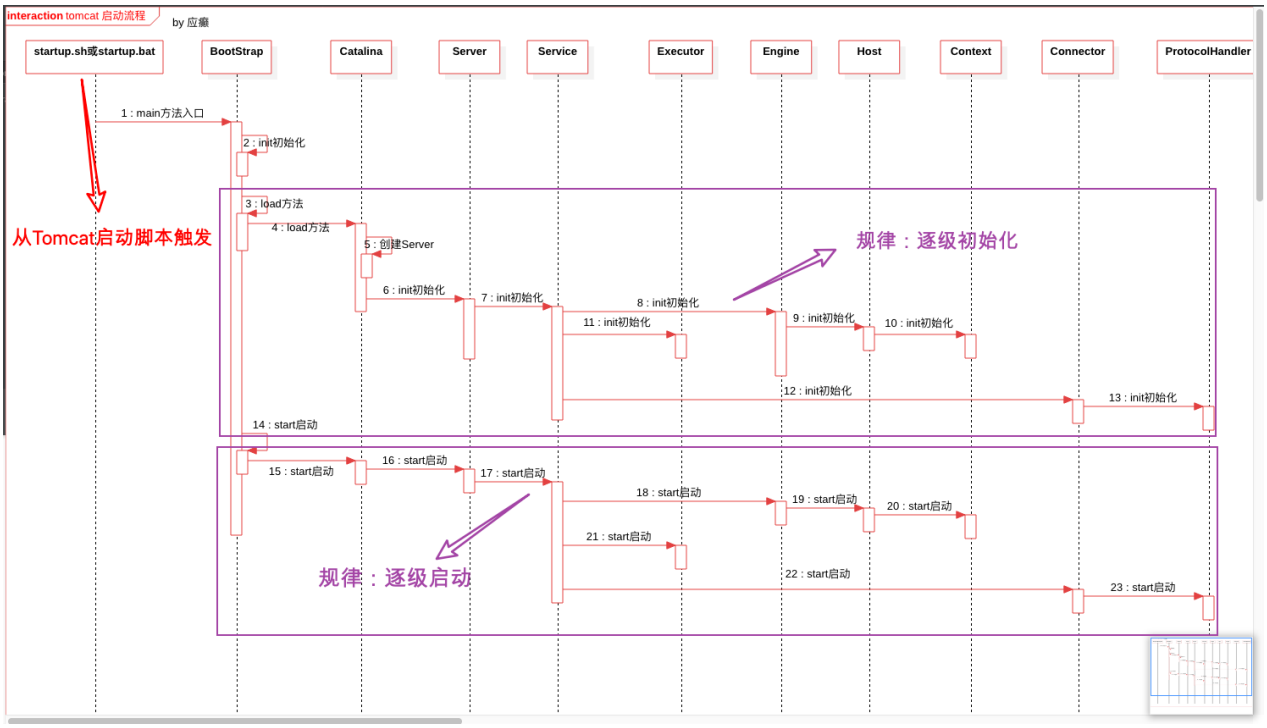
Lifecycle生命周期接口继承体系示意



核心流程源码剖析

源码追踪部分我们关注两个流程：Tomcat启动流程和Tomcat请求处理流程

Tomcat启动流程



Tomcat请求处理流程

- 请求处理流程分析

tomcat请求处理流程: 当一个servlet请求到来的时候，tomcat是通过怎样的机制定位到servlet并且执行的

url：http://localhost:8080/web_demo/resume/addresume

锁定servlet

Mapper（映射的意思，这里不是集合）组件完成url和Host、Context、Wrapper等容器的映射

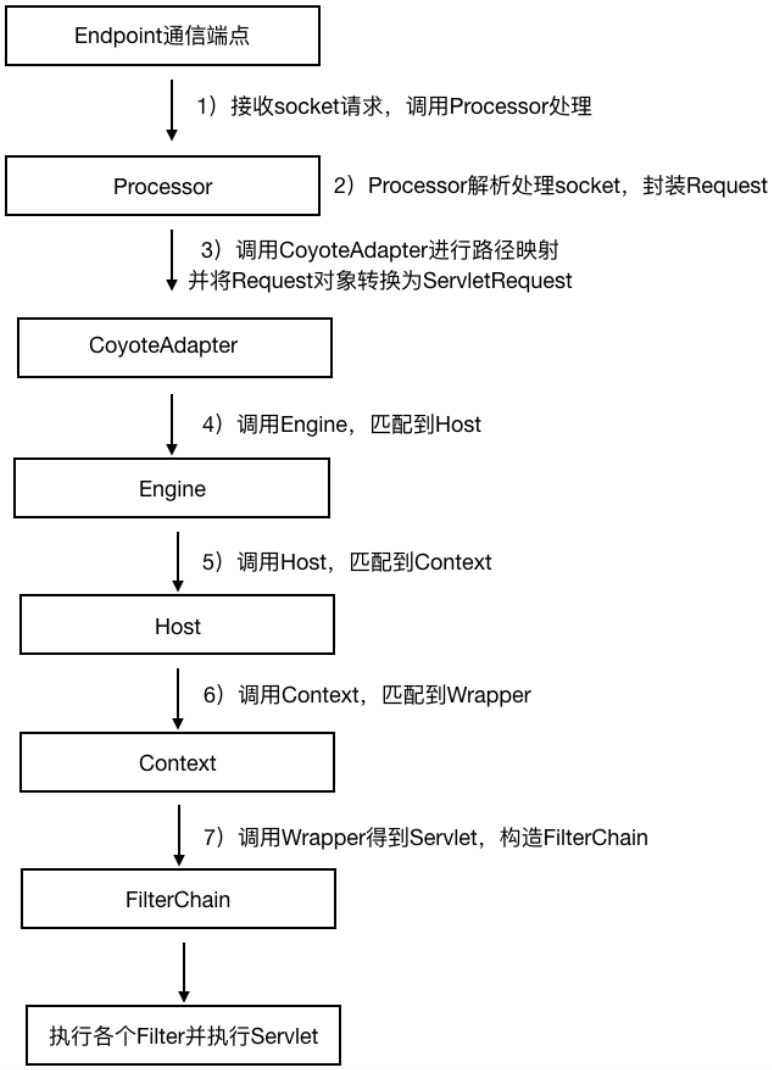
Mapper组件体系结构

web应用案例——>部署到tomcat软件中（不是源代码工程）

最终，希望的是把web应用案例部署到tomcat源代码工程中

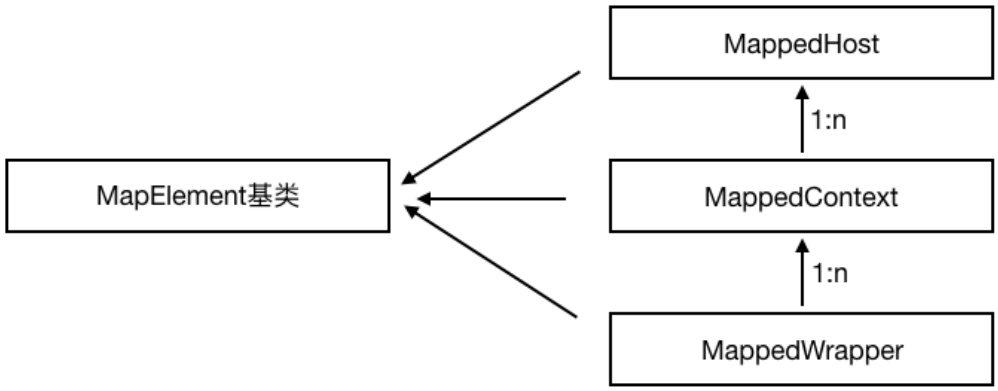
- 请求处理流程示意图

by 应癫



- Mapper组件体系结构

by 应癫　Tomcat Mapper组件体系结构



第五部分 Tomcat 类加载机制剖析

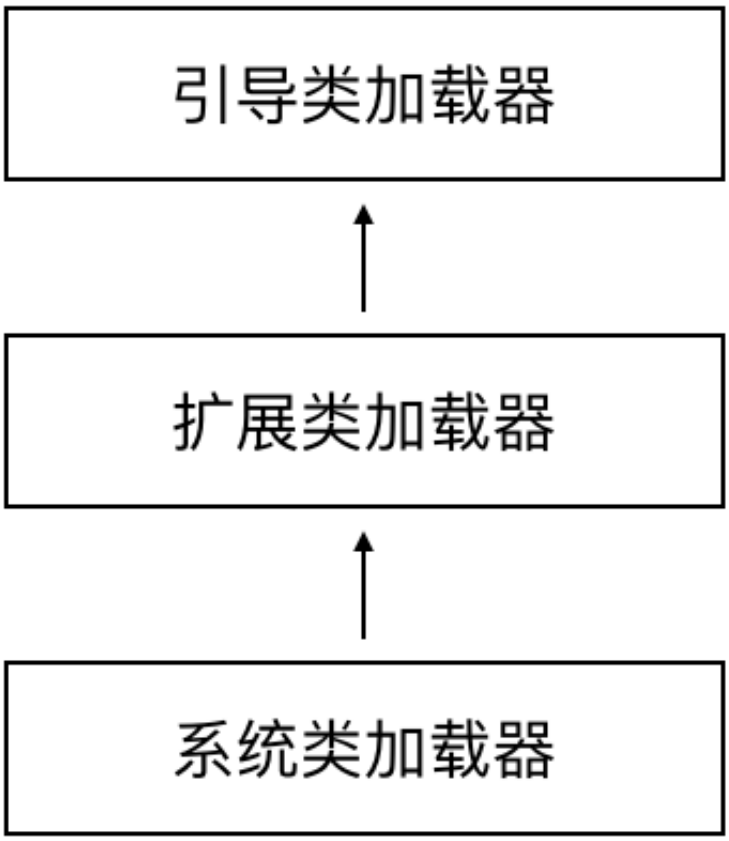
Java类（.java）—> 字节码文件(.class) —> 字节码文件需要被加载到jvm内存当中（这个过程就是一个类加载的过程）

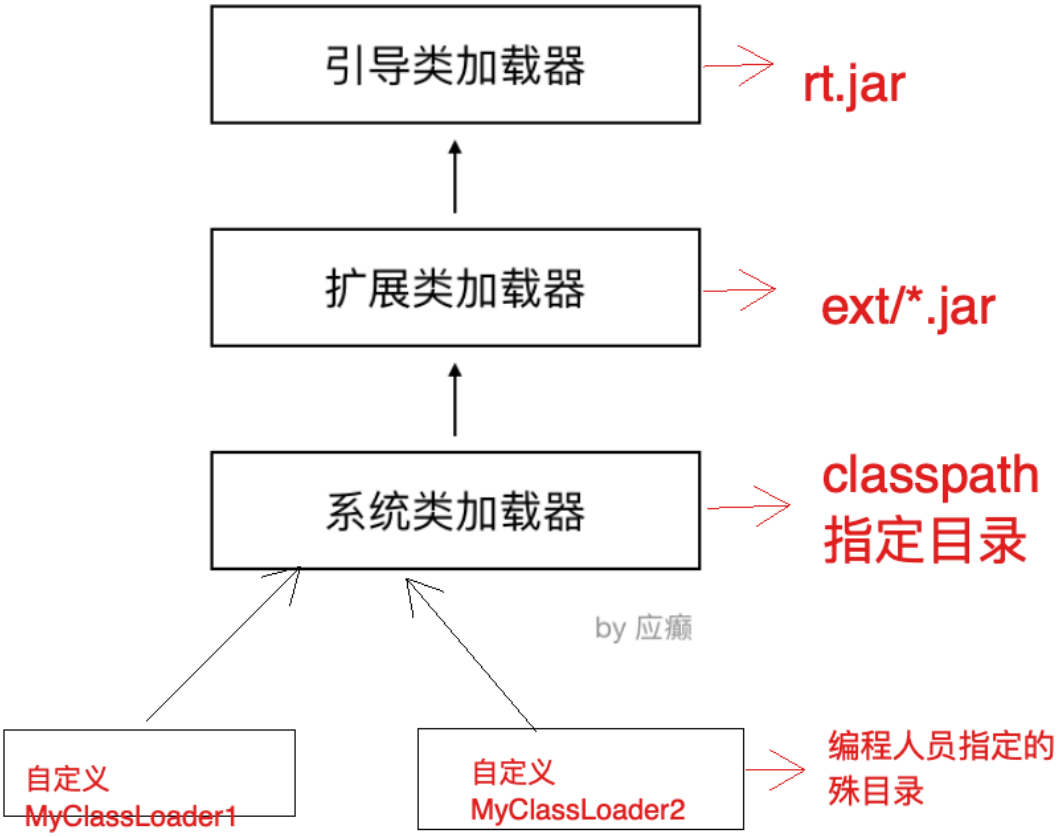
类加载器（ClassLoader，说白了也是一个类，jvm启动的时候先把类加载器读取到内存当中去，其他的类（比如各种jar中的字节码文件，自己开发的代码编译之后的.class文件等等））

要说 Tomcat 的类加载机制，首先需要来看看 Jvm 的类加载机制，因为 Tomcat 类加载机制是在 Jvm 类加载机制基础之上进行了一些变动。

第 1 节 JVM 的类加载机制

JVM 的类加载机制中有一个非常重要的角色叫做类加载器（ClassLoader），类加载器有自己的体系，Jvm内置了几种类加载器，包括：引导类加载器、扩展类加载器、系统类加载器，他们之间形成父子关系，通过 Parent 属性来定义这种关系，最终可以形成树形结构。





类加载器	作用
引导启动类加载器 BootstrapClassLoader	c++编写，加载java核心库 java.*,比如rt.jar中的类，构造ExtClassLoader和AppClassLoader
扩展类加载器 ExtClassLoader	java编写，加载扩展库 JAVA_HOME/lib/ext目录下的jar中的类，如classpath中的jre ，javax.*或者java.ext.dir指定位置中的类
系统类加载器 SystemClassLoader/AppClassLoader	默认的类加载器，搜索环境变量 classpath 中指明的路径

另外：用户可以自定义类加载器（Java编写，用户自定义的类加载器，可加载指定路径的 class 文件）

当 JVM 运行过程中，用户自定义了类加载器去加载某些类时，会按照下面的步骤（父类委托机制）

1） 用户自己的类加载器，把加载请求传给父加载器，父加载器再传给其父加载器，一直到加载器树的顶层

2 ）最顶层的类加载器首先针对其特定的位置加载，如果加载不到就转交给子类

3 ）如果一直到底层的类加载都没有加载到，那么就会抛出异常 ClassNotFoundException

因此，按照这个过程可以想到，如果同样在 classpath 指定的目录中和自己工作目录中存放相同的 class，会优先加载 classpath 目录中的文件

第 2 节 双亲委派机制

2.1 什么是双亲委派机制

当某个类加载器需要加载某个.class文件时，它首先把这个任务委托给他的上级类加载器，递归这个操作，如果上级的类加载器没有加载，自己才会去加载这个类。

2.2 双亲委派机制的作用

- 防止重复加载同一个.class。通过委托去向上面问一问，加载过了，就不用再加载一遍。保证数据安全。
- 保证核心.class不能被篡改。通过委托方式，不会去篡改核心.class，即使篡改也不会去加载，即使加载也不会是同一个.class对象了。不同的加载器加载同一个.class也不是同一个.class对象。这样保证了class执行安全（如果子类加载器先加载，那么我们可以写一些与java.lang包中基础类同名的类， 然后再定义一个子类加载器，这样整个应用使用的基础类就都变成我们自己定义的类了。）

  Object类 -----> 自定义类加载器（会出现问题的，那么真正的Object类就可能被篡改了）

第 3 节 Tomcat 的类加载机制

Tomcat 的类加载机制相对于 Jvm 的类加载机制做了一些改变。

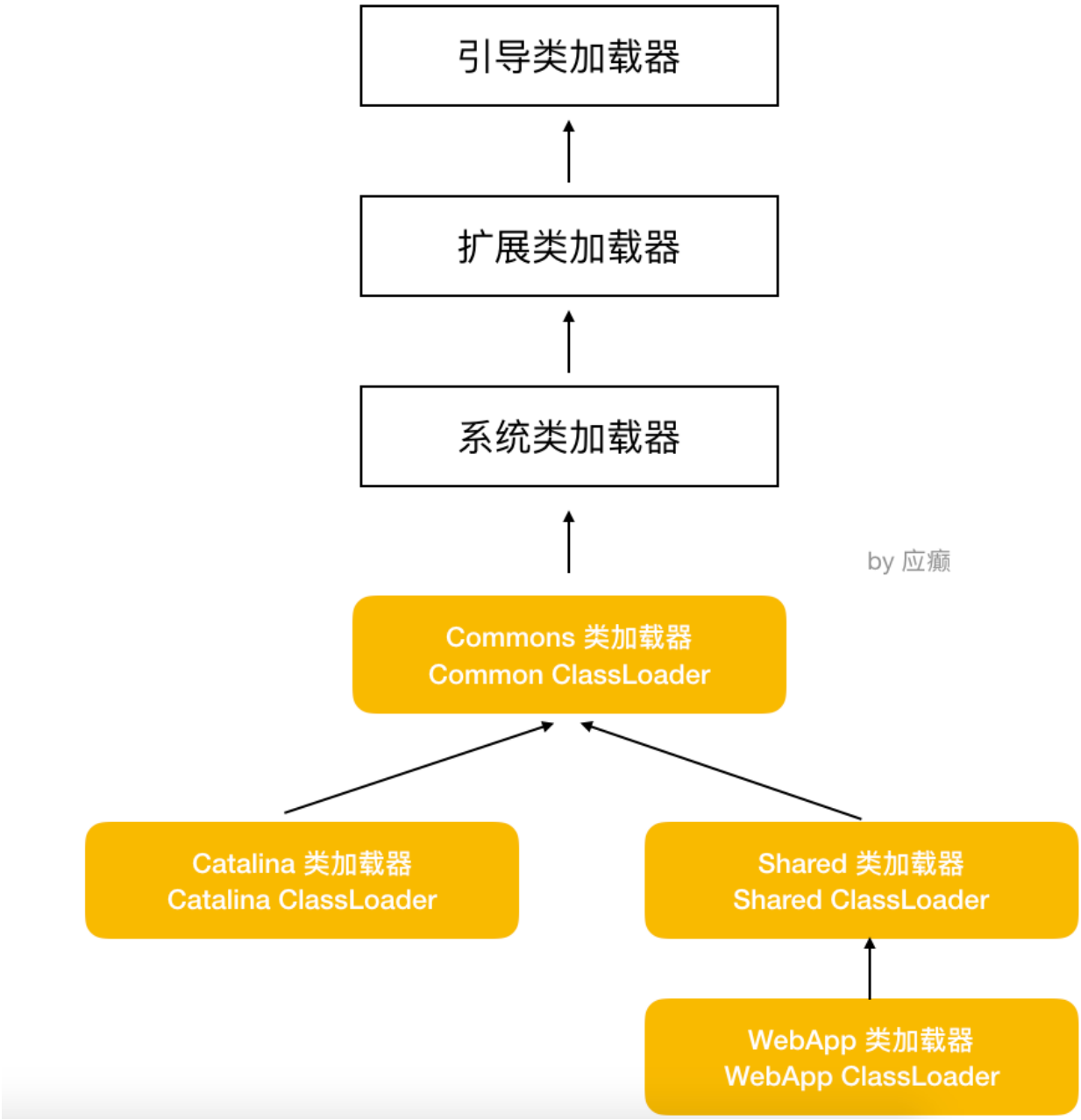
没有严格的遵从双亲委派机制，也可以说打破了双亲委派机制

比如：有一个tomcat，webapps下部署了两个应用

app1/lib/a-1.0.jar com.lagou.edu.Abc

app2/lib/a-2.0.jar com.lagou.edu.Abc

不同版本中Abc类的内容是不同的，代码是不一样的



- 引导类加载器 和 扩展类加载器 的作用不变
- 系统类加载器正常情况下加载的是 CLASSPATH 下的类，但是 Tomcat 的启动脚本并未使用该变量，而是加载tomcat启动的类，比如bootstrap.jar，通常在catalina.bat或者catalina.sh中指定。位于CATALINA_HOME/bin下
- Common 通用类加载器加载Tomcat使用以及应用通用的一些类，位于CATALINA_HOME/lib下，比如servlet-api.jar
- Catalina ClassLoader 用于加载服务器内部可见类，这些类应用程序不能访问
- Shared ClassLoader 用于加载应用程序共享类，这些类服务器不会依赖
- Webapp ClassLoader，每个应用程序都会有一个独一无二的Webapp ClassLoader，他用来加载本应用程序 /WEB-INF/classes 和 /WEB-INF/lib 下的类。

  tomcat 8.5 默认改变了严格的双亲委派机制

  - 首先从 Bootstrap Classloader加载指定的类
  - 如果未加载到，则从 /WEB-INF/classes加载
  - 如果未加载到，则从 /WEB-INF/lib/*.jar 加载

- 如果未加载到，则依次从 System、Common、Shared 加载（在这最后一步，遵从双亲委派机制）

第六部分 Tomcat 对 Https 的支持及 Tomcat 性能优化策略

第 1 节 Tomcat 对 HTTPS 的支持

Https是用来加强数据传输安全的

1.1 HTTPS 简介

https 编辑

本词条由"科普中国"科学百科词条编写与应用工作项目 审核 。

HTTPS （全称：Hyper Text Transfer Protocol over SecureSocket Layer），是以安全为目标的 HTTP 通道，在HTTP的基础上通过传输加密和身份认证保证了传输过程的安全性 [1] 。HTTPS 在HTTP 的基础下加入SSL 层，HTTPS 的安全基础是 SSL，因此加密的详细内容就需要 SSL。 HTTPS 存在不同于 HTTP 的默认端口及一个加密/身份验证层（在 HTTP与 TCP 之间）。这个系统提供了身份验证与加密通讯方法。它被广泛用于万维网上安全敏感的通讯，例如交易支付等方面 [2] 。

中文名	超文本传输安全协议	目　标	安全的HTTP通道
外文名	HTTPS	应　用	安全数据传输

Http超文本传输协议，明文传输 ，传输不安全，https在传输数据的时候会对数据进行加密

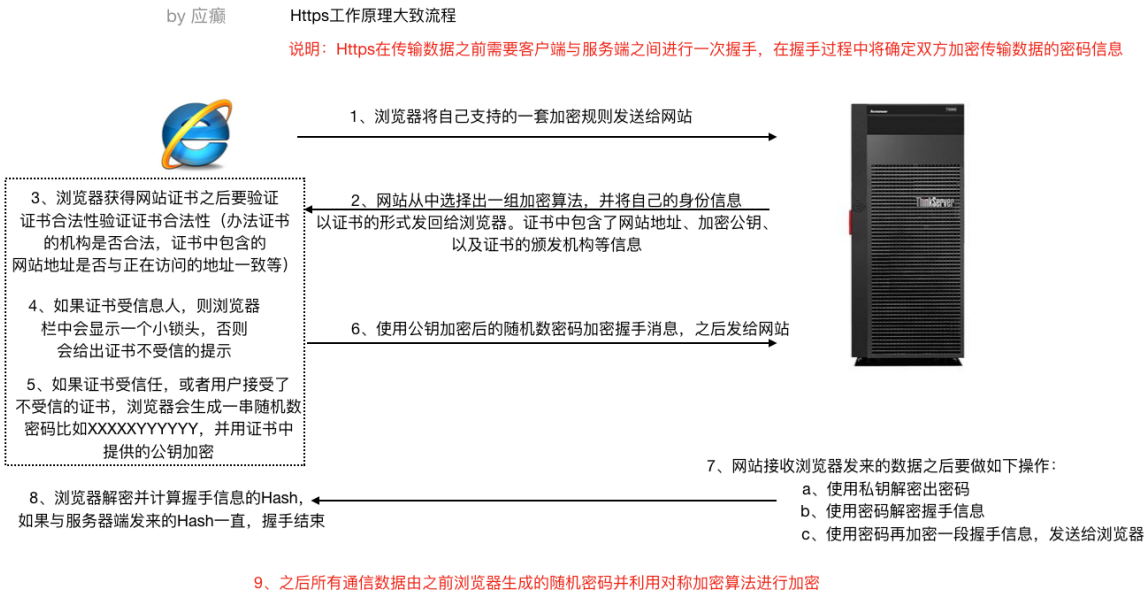
ssl协议

TLS(transport layer security)协议

HTTPS和HTTP的主要区别

- HTTPS协议使用时需要到电子商务认证授权机构（CA）申请SSL证书
- HTTP默认使用8080端口，HTTPS默认使用8443端口
- HTTPS则是具有SSL加密的安全性传输协议，对数据的传输进行加密，效果上相当于HTTP的升级版
- HTTP的连接是无状态的，不安全的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比HTTP协议安全

HTTPS工作原理



1.2 Tomcat 对 HTTPS 的支持

1） 使用 JDK 中的 keytool 工具生成免费的秘钥库文件(证书)。

```
keytool -genkey -alias lagou -keyalg RSA -keystore lagou.keystore
```

```
localhost:conf yingdian$ keytool -genkey -alias lagou -keyalg RSA -keystore lagou.keystore
输入密钥库口令：
再次输入新口令：
您的名字与姓氏是什么？
  [Unknown]:  www.abc.com
您的组织单位名称是什么？
  [Unknown]:  lagou
您的组织名称是什么？
  [Unknown]:  lagou
您所在的城市或区域名称是什么？
  [Unknown]:  beijing
您所在的省/市/自治区名称是什么？
  [Unknown]:  beijing
该单位的双字母国家/地区代码是什么？
  [Unknown]:  cn
CN=www.abc.com, OU=lagou, O=lagou, L=beijing, ST=beijing, C=cn是否正确？
  [否]:  y
```

和网站域名保持一致

2） 配置conf/server.xml

```
<Connector port="8443" protocol="org.apache.coyote.http11.Http11NioProtocol"
    maxThreads="150" schema="https" secure="true" SSLEnabled="true">
    <SSLHostConfig>
        <Certificate
certificateKeystoreFile="/Users/yingdian/workspace/servers/apache-tomcat-
8.5.50/conf/lagou.keystore" certificateKeystorePassword="lagou123"  type="RSA"
/>
    </SSLHostConfig>
</Connector>
```

4）使用https协议访问8443端口（https://localhost:8443）。

第 2 节 Tomcat 性能优化策略

系统性能的衡量指标，主要是响应时间和吞吐量。

1）响应时间：执行某个操作的耗时；

2) 吞吐量：系统在给定时间内能够支持的事务数量，单位为TPS（Transactions PerSecond的缩写，也就是事务数/秒，一个事务是指一个客户机向服务器发送请求然后服务器做出反应的过程。

Tomcat优化从两个方面进行

1）JVM虚拟机优化（优化内存模型）

2）Tomcat自身配置的优化（比如是否使用了共享线程池？IO模型？）

学习优化的原则

提供给大家优化思路，没有说有明确的参数值大家直接去使用，必须根据自己的真实生产环境来进行调整，调优是一个过程

2.1 虚拟机运行优化（参数调整）

Java 虚拟机的运行优化主要是内存分配和垃圾回收策略的优化：

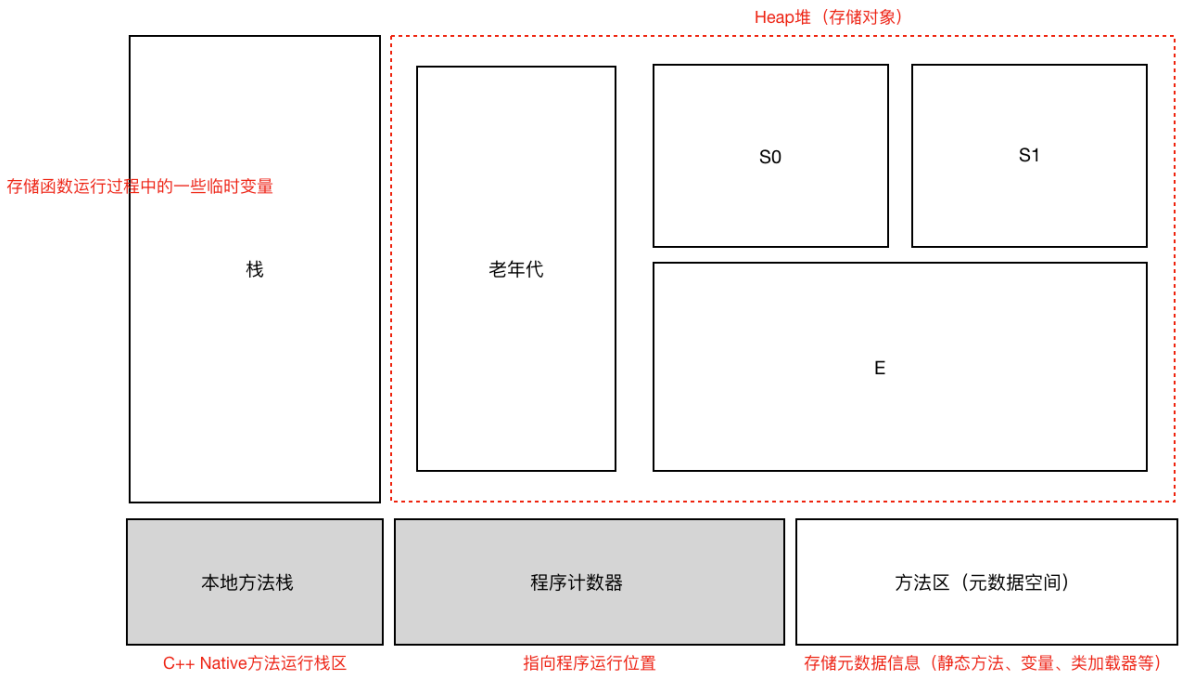
- 内存直接影响服务的运行效率和吞吐量
- 垃圾回收机制会不同程度地导致程序运行中断（垃圾回收策略不同，垃圾回收次数和回收效率都是不同的）

1） Java 虚拟机内存相关参数

参数	参数作用	优化建议
-server	启动Server，以服务端模式运行	服务端模式建议开启
-Xms	最小堆内存	建议与-Xmx设置相同
-Xmx	最大堆内存	建议设置为可用内存的80%
-XX:MetaspaceSize	元空间初始值	
-XX:MaxMetaspaceSize	元空间最大内存	默认无限
-XX:NewRatio	年轻代和老年代大小比值，取值为整数，默认为2	不需要修改
-XX:SurvivorRatio	Eden区与Survivor区大小的比值，取值为整数，默认为8	不需要修改

JVM内存模型回顾

by 应癫　JVM 内存模型主体示意图



注意：栈、本地方法栈、程序计数器是线程私有的；堆区和方法区是全局共享的

参数调整示例

```
JAVA_OPTS="-server -Xms2048m -Xmx2048m -XX:MetaspaceSize=256m -
XX:MaxMetaspaceSize=512m"
```

调整后查看可使用JDK提供的内存映射工具

```
localhost:bin yingdian$ jhsdb jmap --heap --pid 8481
Attaching to process ID 8481, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 11.0.5+10-LTS

using thread-local object allocation.
Garbage-First (G1) GC with 4 thread(s)

Heap Configuration:
   MinHeapFreeRatio         = 40
   MaxHeapFreeRatio         = 70
   MaxHeapSize              = 2147483648 (2048.0MB)
   NewSize                  = 1363144 (1.2999954223632812MB)
   MaxNewSize               = 1287651328 (1228.0MB)
   OldSize                  = 5452592 (5.1999969482421875MB)
   NewRatio                 = 2
   SurvivorRatio            = 8
   MetaspaceSize            = 268435456 (256.0MB)
   CompressedClassSpaceSize = 528482304 (504.0MB)
   MaxMetaspaceSize         = 536870912 (512.0MB)
   G1HeapRegionSize         = 1048576 (1.0MB)

Heap Usage:
G1 Heap:
   regions  = 2048
   capacity = 2147483648 (2048.0MB)
   used     = 52320840 (49.89704132080078MB)
   free     = 2095162808 (1998.1029586791992MB)
   2.4363789707422256% used
G1 Young Generation:
Eden Space:
   regions  = 39
   capacity = 101711872 (97.0MB)
   used     = 40894464 (39.0MB)
   free     = 60817408 (58.0MB)
   40.20618556701031% used
Survivor Space:
   regions  = 11
   capacity = 11534336 (11.0MB)
   used     = 11534336 (11.0MB)
   free     = 0 (0.0MB)
   100.0% used
G1 Old Generation:
   regions  = 0
   capacity = 2034237440 (1940.0MB)
   used     = 0 (0.0MB)
   free     = 2034237440 (1940.0MB)
   0.0% used
```

使用内存映射工具查看tomcat中jvm内存配置

2） 垃圾回收（GC）策略

垃圾回收性能指标

- 吞吐量：工作时间（排除GC时间）占总时间的百分比， 工作时间并不仅是程序运行的时间，还包含内存分配时间。
- 暂停时间：由垃圾回收导致的应用程序停止响应次数/时间。

垃圾收集器

- 串行收集器（Serial Collector）

  单线程执行所有的垃圾回收工作， 适用于单核CPU服务器

  工作进程-----|（单线程）垃圾回收线程进行垃圾收集|---工作进程继续

- 并行收集器（Parallel Collector）

工作进程-----|（多线程）垃圾回收线程进行垃圾收集|---工作进程继续

又称为吞吐量收集器（关注吞吐量）， 以并行的方式执行年轻代的垃圾回收， 该方式可以显著降低垃圾回收的开销(指多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态)。适用于多处理器或多线程硬件上运行的数据量较大的应用

- 并发收集器（Concurrent Collector）

  以并发的方式执行大部分垃圾回收工作，以缩短垃圾回收的暂停时间。适用于那些响应时间优先于吞吐量的应用， 因为该收集器虽然最小化了暂停时间(指用户线程与垃圾收集线程同时执行,但不一定是并行的，可能会交替进行)， 但是会降低应用程序的性能

- CMS收集器（Concurrent Mark Sweep Collector）

  并发标记清除收集器， 适用于那些更愿意缩短垃圾回收暂停时间并且负担的起与垃圾回收共享处理器资源的应用

- G1收集器（Garbage-First Garbage Collector）

  适用于大容量内存的多核服务器， 可以在满足垃圾回收暂停时间目标的同时， 以最大可能性实现高吞吐量(JDK1.7之后)

垃圾回收器参数

参数	描述
-XX:+UseSerialGC	启用串行收集器
-XX:+UseParallelGC	启用并行垃圾收集器，配置了该选项，那么 -XX:+UseParallelOldGC默认启用
-XX:+UseParNewGC	年轻代采用并行收集器，如果设置了 -XX:+UseConcMarkSweepGC选项，自动启用
-XX:ParallelGCThreads	年轻代及老年代垃圾回收使用的线程数。默认值依赖于JVM使用的CPU个数
-XX:+UseConcMarkSweepGC（CMS）	对于老年代，启用CMS垃圾收集器。 当并行收集器无法满足应用的延迟需求是，推荐使用CMS或G1收集器。启用该选项后， -XX:+UseParNewGC自动启用。
-XX:+UseG1GC	启用G1收集器。 G1是服务器类型的收集器， 用于多核、大内存的机器。它在保持高吞吐量的情况下，高概率满足GC暂停时间的目标。

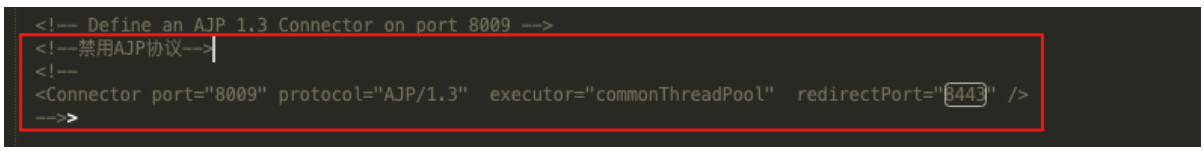
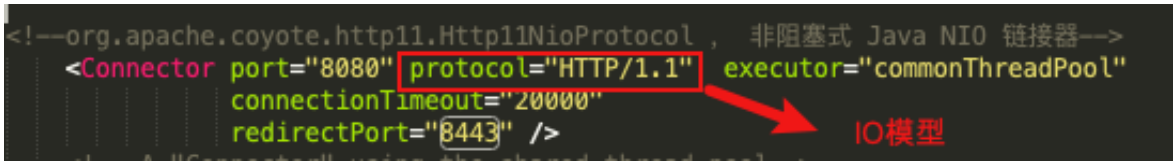
在bin/catalina.sh的脚本中 , 追加如下配置 :

```
JAVA_OPTS="-XX:+UseConcMarkSweepGC"
```

2.2 Tomcat 配置调优

Tomcat自身相关的调优

- 调整tomcat线程池


name="commonThreadPool"
maxThreads="200"
minSpareThreads="100"
maxIdleTime="60000"
maxQueueSize="Integer.MAX_VALUE"
prestartminSpareThreads="false"
threadPriority="5"
Connector连接器使用线程池
<Connector port="8080" protocol="HTTP/1.1" executor="commonThreadPool"
connectionTimeout="20000"
redirectPort="8443" />
<!-- A "Connector" using the shared thread pool-->

<!-- Define an AJP 1.3 Connector on port 8009 -->
<!--禁用AJP协议-->
<!--
<Connector port="8009" protocol="AJP/1.3" executor="commonThreadPool" redirectPort="8443" />
-->


- 调整tomcat的连接器

  调整tomcat/conf/server.xml 中关于链接器的配置可以提升应用服务器的性能。

参数	说明
maxConnections	最大连接数，当到达该值后，服务器接收但不会处理更多的请求， 额外的请求将会阻塞直到连接数低于maxConnections 。可通过ulimit -a 查看服务器限制。对于CPU要求更高(计算密集型)时，建议不要配置过大 ; 对于CPU要求不是特别高时，建议配置在2000左右(受服务器性能影响)。 当然这个需要服务器硬件的支持
maxThreads	最大线程数,需要根据服务器的硬件情况，进行一个合理的设置
acceptCount	最大排队等待数,当服务器接收的请求数量到达maxConnections ， 此时Tomcat会将后面的请求，存放在任务队列中进行排序， acceptCount指的就是任务队列中排队等待的请求数 。 一台Tomcat的最大的请求处理数量，是maxConnections+acceptCount

- 禁用 AJP 连接器

- 调整 IO 模式

  Tomcat8之前的版本默认使用BIO（阻塞式IO），对于每一个请求都要创建一个线程来处理，不适合高并发；Tomcat8以后的版本默认使用NIO模式（非阻塞式IO）

当Tomcat并发性能有较高要求或者出现瓶颈时，我们可以尝试使用APR模式，APR（Apache Portable Runtime）是从操作系统级别解决异步IO问题，使用时需要在操作系统上安装APR和Native（因为APR原理是使用使用JNI技术调用操作系统底层的IO接口）

- 动静分离

  可以使用Nginx+Tomcat相结合的部署方案，Nginx负责静态资源访问，Tomcat负责Jsp等动态资源访问处理（因为Tomcat不擅长处理静态资源）。