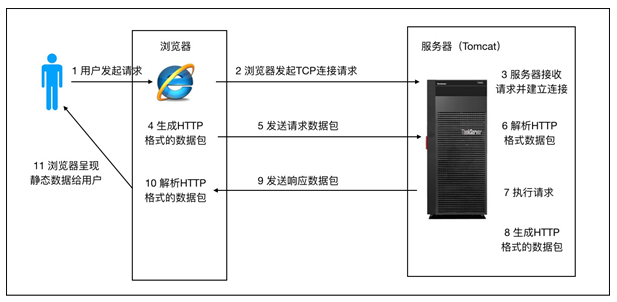
Apache Tomcat Web应⽤服务器

说明：基于 8.5.50 版本的 Tomcat 学习

# 第⼀部分 Tomcat 系统架构与原理剖析

## 第 1 节 浏览器访问服务器的流程



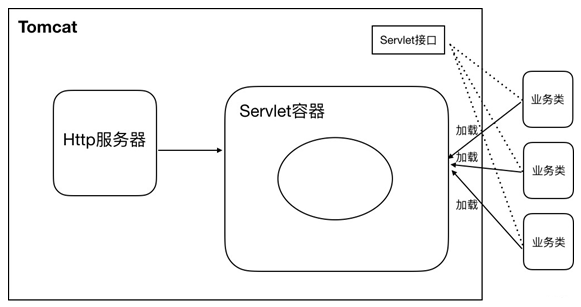
注意：浏览器访问服务器使⽤的是Http协议，Http是应⽤层协议，⽤于定义数据通信的格式，具体的数 据传输使⽤的是TCP/IP协议

## 第 2 节 Tomcat 总体架构

### 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。如果http服务器（tomcat）直接调用业务处理类完成业务处理的话，那么tomcat和业务类耦合在一起了



所以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接⼝调⽤相应的业务类。**Servlet接⼝和Servlet容器这⼀整套内容叫作Servlet规范**。

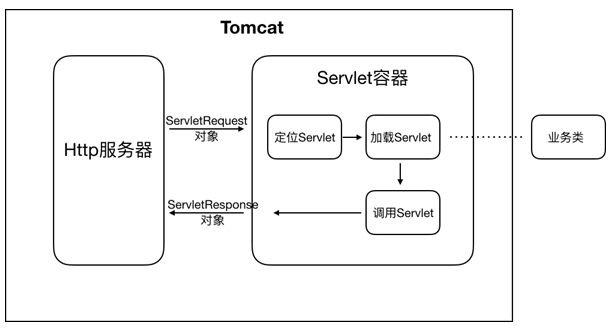
注意：Tomcat既按照Servlet规范的要求去实现了Servlet容器，同时它也具有HTTP服务器的功能。

Tomcat的两个重要身份：

1）http服务器

2）Tomcat是⼀个Servlet容器

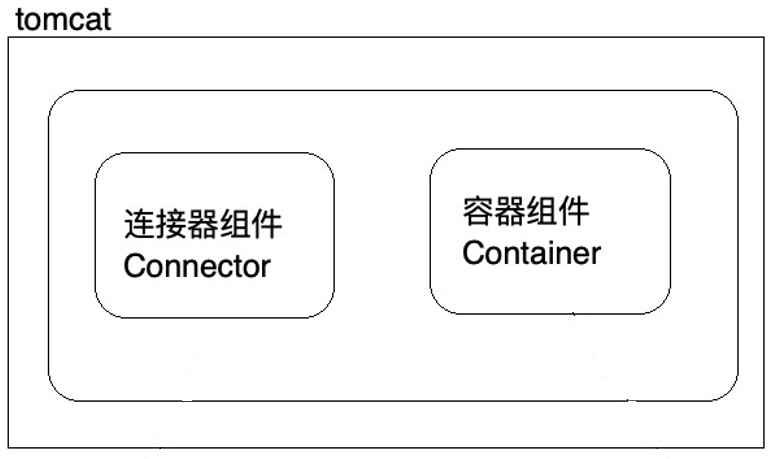
### Tomcat Servlet容器处理流程



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

### 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。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。



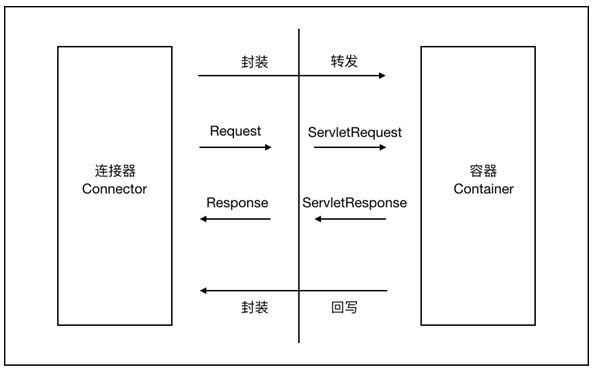
**连接器组件：负责对外交流：** 处理Socket连接，负责⽹络字节流与Request和Response对象的转化；

**容器组件：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

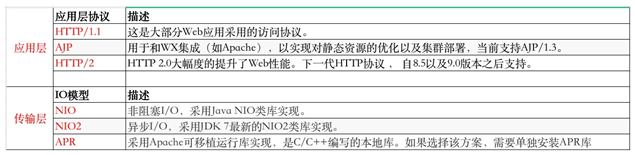
### 3.1 Coyote 简介

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。



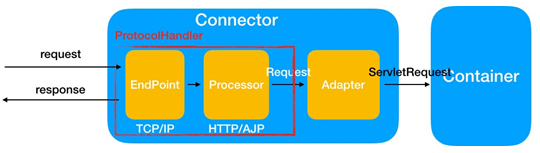
1. **Coyote封装了底层的网络通信（Socket请求及响应处理）**
2. **Coyote使Catalina容器组件与具体的请求协议及IO操作方式完全解耦**
3. **Coyote将Socket输入转换封装为Request对象，进一步封装后交由Catalina容器进行处理，处理请求完成后，Catalina通过Coyote提供的Response对象将结果写入输出流**
4. **Coyote负责的是具体协议（应用层）和IO（传输层）相关内容**

Tomcat Coyote 支持的IO模型和多种应用层协议



在Tomcat 8.0以前，Tomcat默认采用的IO方式是BIO，之后改成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。

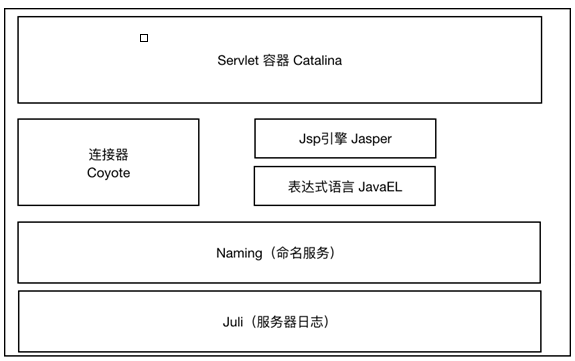
### 3.2 Coyote 的流程及内部组件说明



|  |  |
| --- | --- |
| **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容器组件 Catalina

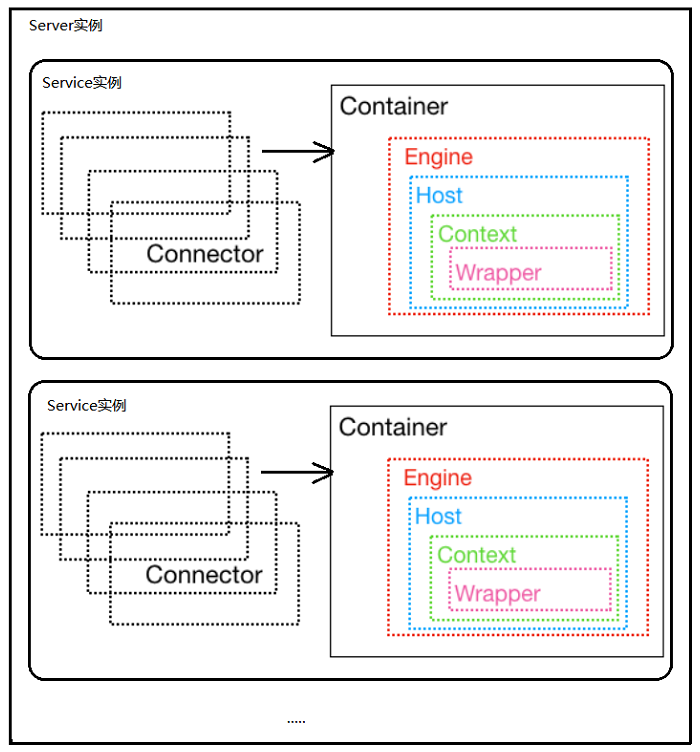
### 4.1 Tomcat 模块分层结构图及Catalina位置



Tomcat本质上就是一款Servlet容器，因为Catalina才是Tomcat的核心，其它模块都是为Catalina提供支撑的。

1. Coyote提供链接通信
2. Jasper提供JSP引擎
3. Naming提供JNDI服务
4. Juli提供日志服务

### 4.2 Servlet 容器 Catalina 的结构



**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，每个服务有可以有多个Connector和一个Container。总的来说：**

**一个Catalina实例**

**一个Server实例**

**多个Service实例**

**每个Service实例，对应多个Connector实例和一个Container实例**

|  |  |
| --- | --- |
| 组件 | 组件作用说明 |
| Catalina | 负责解析Tomat的配置文件Server.xml。以此来创建服务器Server组件并进行管理 |
| Server | 表示整个Catalina Servlet容器以及其他组件，负责组装并启动Servlet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 | 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 | 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

|  |  |
| --- | --- |
| 组件 | 组件作用说明 |
| Engine | 表示整个Catalina的Servlet引擎，用来管理科多个虚拟站点，一个Service最多只能有一个Engine，但是一个引擎课包含多个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 核⼼配置文件server.xml详解

**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><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

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A JP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Executor创建的共享线程池

<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"](http://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 &quot;%r&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）
2. 客户端请求资源，资源分为静态资源（html）和动态资源（Servlet）
3. 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⻚⾯"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⻚

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).getByt

es());

// 读取内容输出

long written = 0 ;// 已经读取的内容⻓度

int byteSize = 1024; // 计划每次缓冲的⻓度

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⻓度，那就按真实⻓度处理

byteSize = (int) (resourceSize - written); // 剩余的⽂件内容

⻓度

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();

}

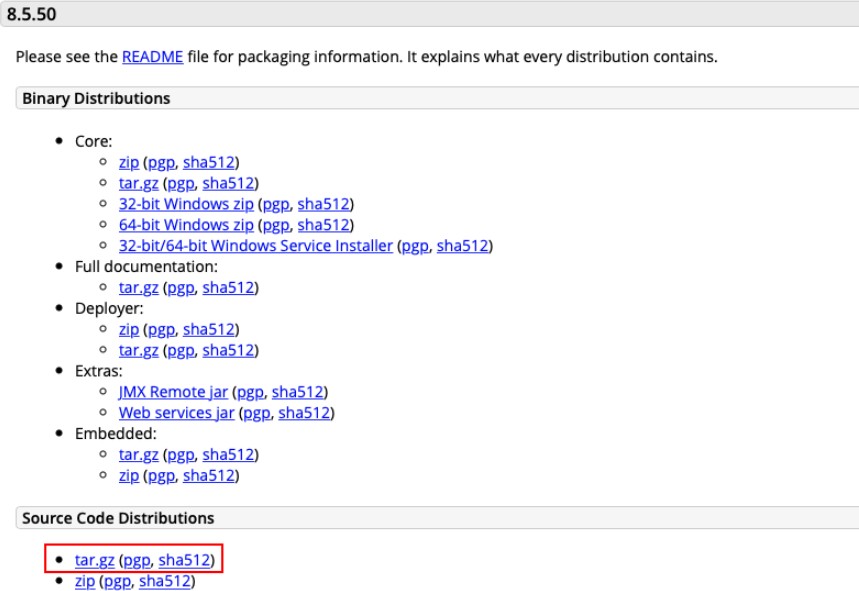
}

}

# 第四部分 Tomcat 核⼼流程源码剖析

## 第 1 节 源码构建步骤

* 1. **下载源码**



* 1. **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"](http://maven.apache.org/POM/4.0.0) xmlns:xsi=["http://www.w3.org/2001/XMLSchema-instance"](http://www.w3.org/2001/XMLSchema-instance) xsi:schemaLocation=["http://maven.apache.org/POM/4.0.0](http://maven.apache.org/POM/4.0.0)

[http://maven.apache.org/xsd/maven-4.0.0.xsd"](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. **导⼊源码⼯程到IDE并进⾏配置**

将源码⼯程导⼊到 IDEA 中

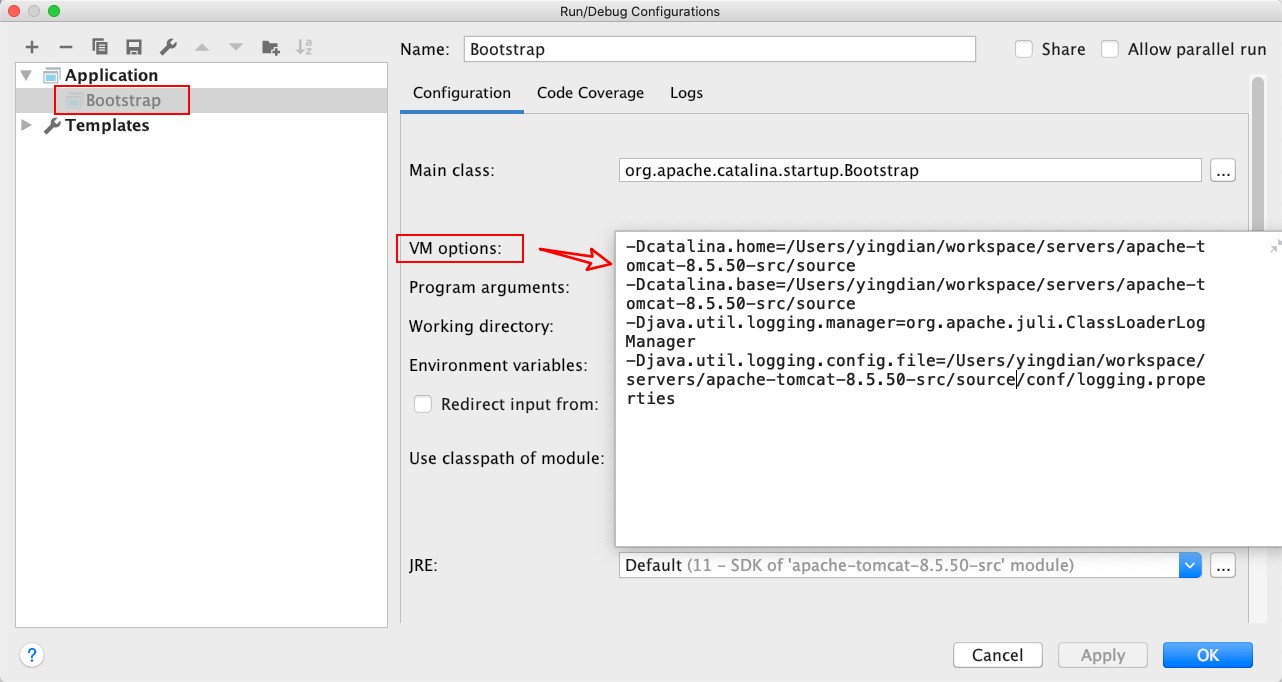
给 tomcat 的源码程序启动类 Bootstrap 配置 VM 参数，因为 tomcat 源码运⾏也需要加载配置⽂件等。

-Dcatalina.home=/Users/yingdian/workspace/servers/apache-tomcat-8.5.50- src/source

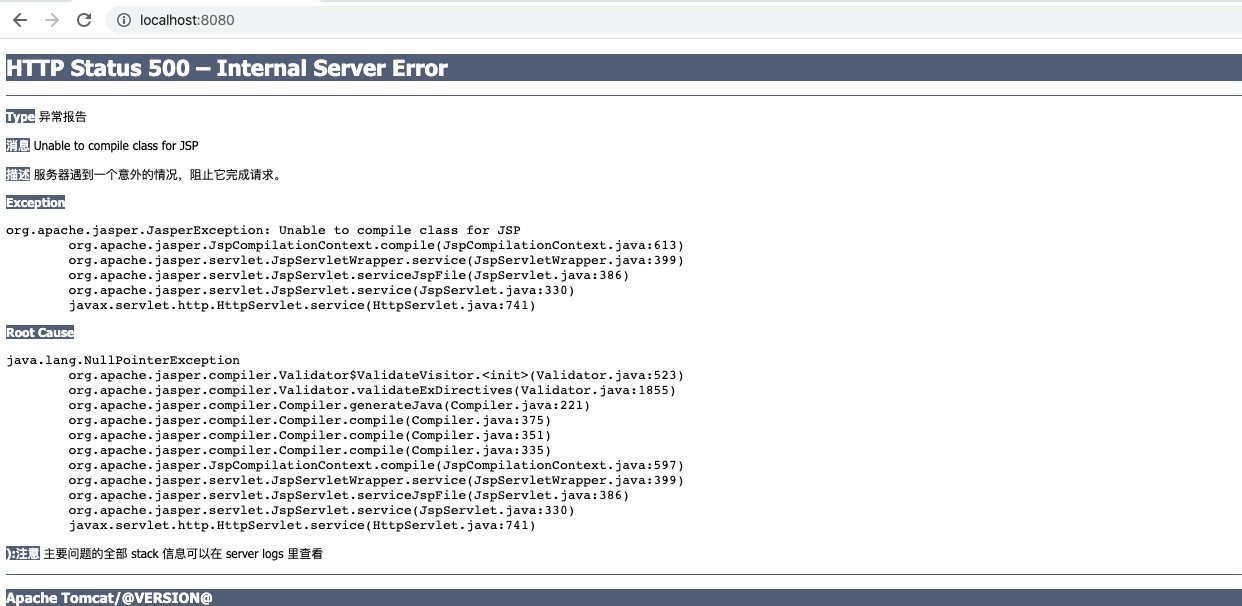
-Dcatalina.base=/Users/yingdian/workspace/servers/apache-tomcat-8.5.50- src/source

-Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager

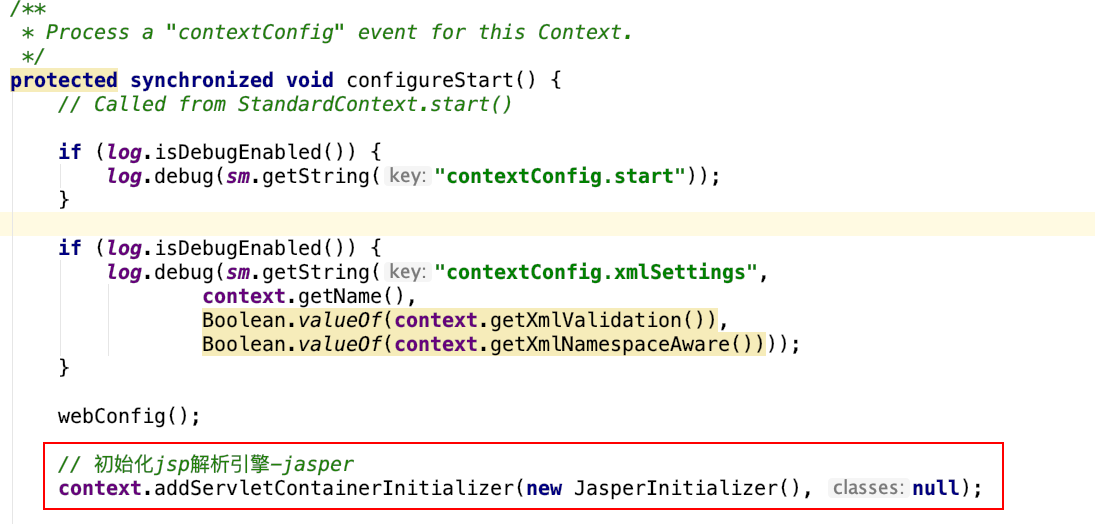
-Djava.util.logging.config.file=/Users/yingdian/workspace/servers/apache- tomcat-8.5.50-src/source/conf/logging.properties



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

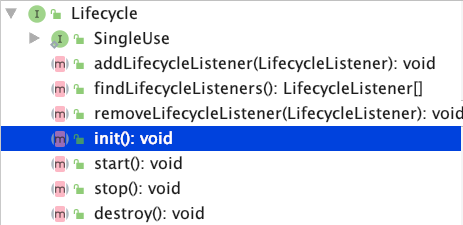


重启 Tomcat，正常访问即可。⾄此，Tomcat 源码构建完毕。

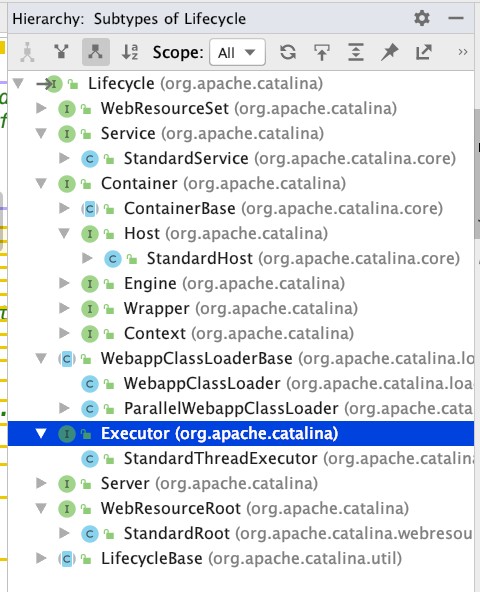
## 第 2 节 核⼼流程源码剖析

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**



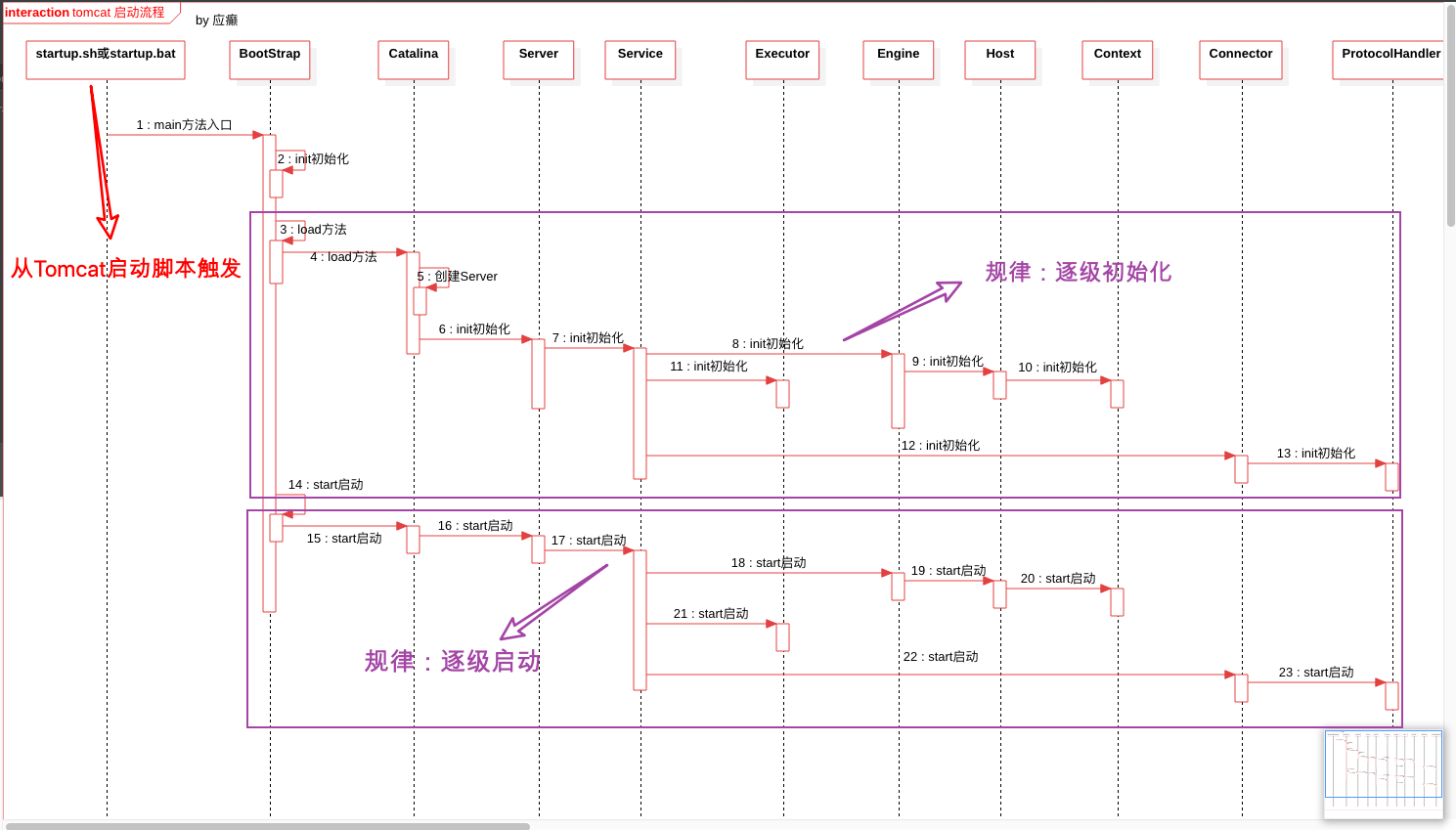
**Lifecycle⽣命周期接⼝继承体系示意**



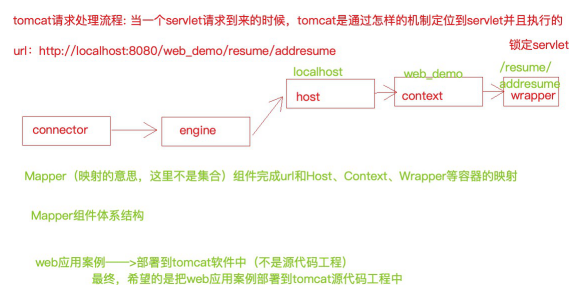
**核⼼流程源码剖析**

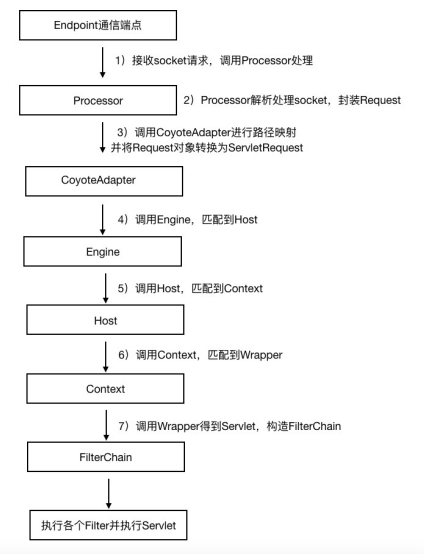
源码追踪部分我们关注两个流程：Tomcat启动流程和Tomcat请求处理流程

### 2.1 Tomcat启动流程

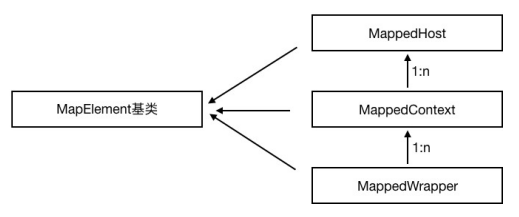


### 2.2 Tomcat请求处理流程





### 2.3 Mapper组件体系结构



# 第五部分 Tomcat 类加载机制剖析

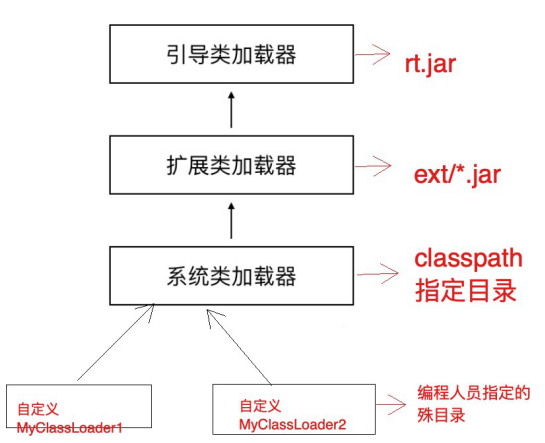
Java类（.java）—> 字节码⽂件(.class) —> 字节码⽂件需要被加载到jvm内存当中（这个过程就是⼀个类加载的过程）

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））

PS： 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⻆⾊叫做类加载器（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）**

## 第 2 节 双亲委派机制

* 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。

* 1. **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

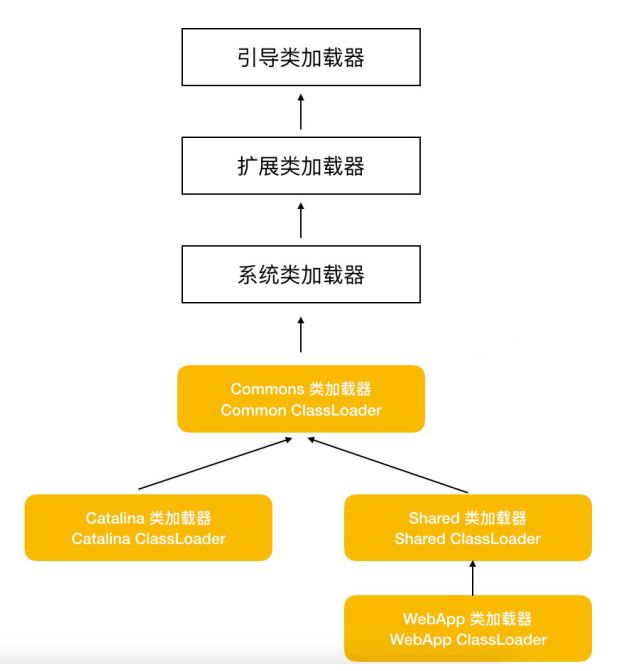
Tomcat 的类加载机制相对于 Jvm 的类加载机制做了一些改变。没有严格的遵从双亲委派机制，也可以说打破了双亲委派机制

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



1，引导类加载器和扩展类加载器的作用不变

2，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，而是去加载tomcat启动的类，比如bootstrap.jar，通常在catalina.bat或catalina.sh中指定，位于CATALINA\_HOME/bin下

3，Common通用类加载器加载Tomcat使用和应用通用的一些类，位于CATALINA\_HOME/bin下，比如servlet-api.jar

4，Catalina类加载器 用于加载服务器内部可见类，这些类应用程序不能访问

5，Shared类加载器 用于加载应用程序共享类，这些类服务器不会依赖

6，Webapp类加载器 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（在这最后一部，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

## 第 1 节 Tomcat 对 HTTPS 的⽀持

Https是⽤来加强数据传输安全的

### HTTPS 简介



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

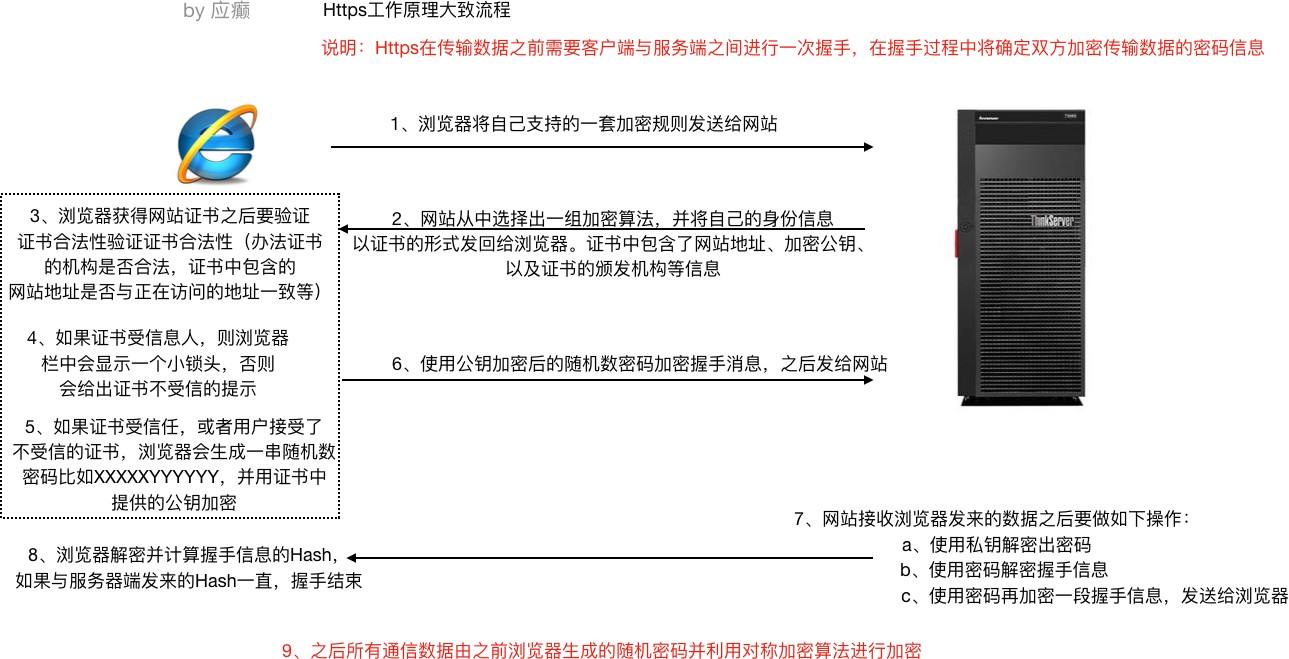
**HTTPS和HTTP的主要区别**

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

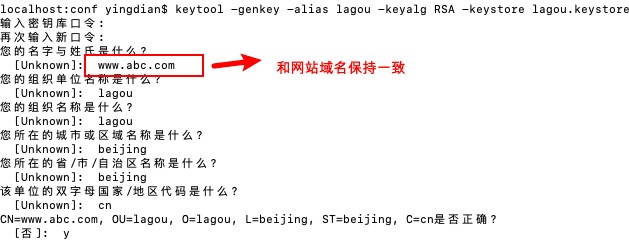
HTTP的连接是⽆状态的，不安全的；HTTPS协议是由SSL+HTTP协议构建的可进⾏加密传输、身 份认证的⽹络协议，⽐HTTP协议安全

**HTTPS⼯作原理**



### 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

1. 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）

1. 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

### 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

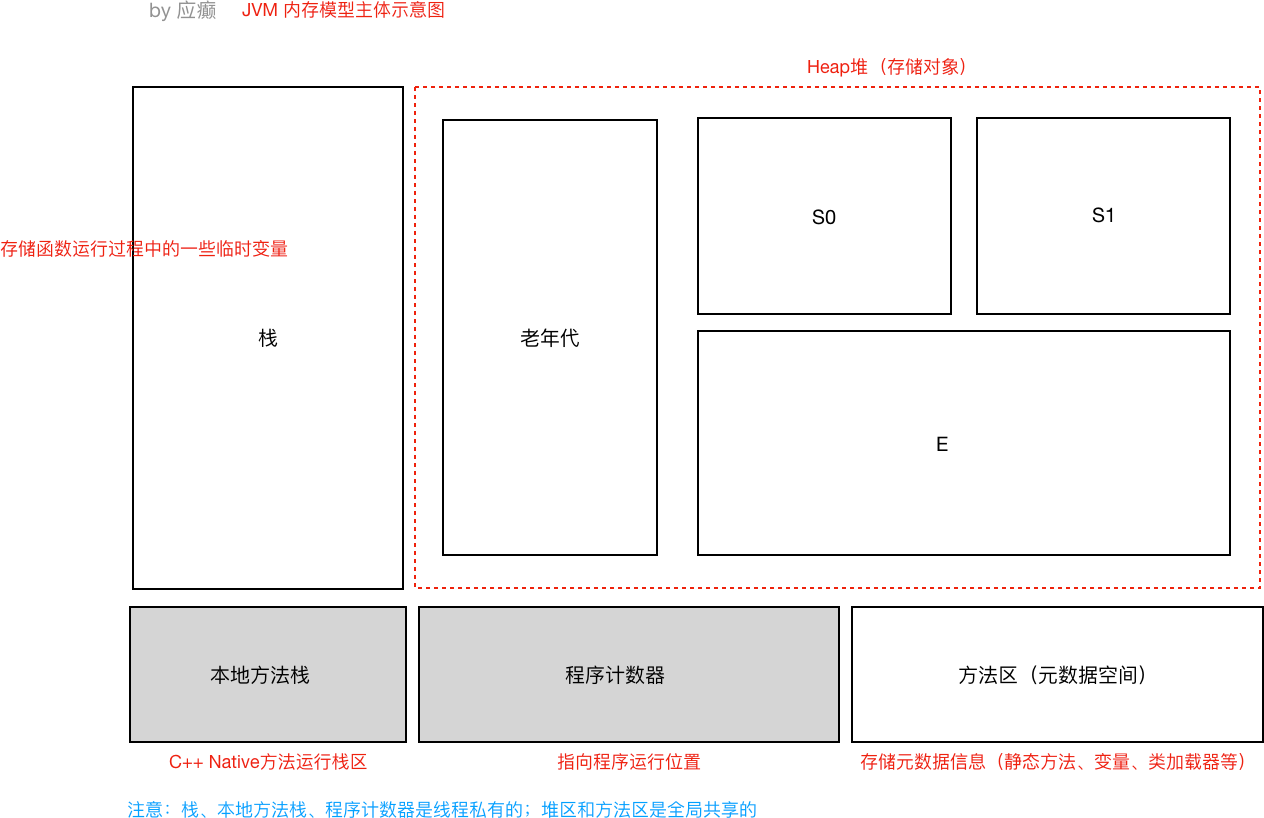
垃圾回收机制会不同程度地导致程序运⾏中断（垃圾回收策略不同，垃圾回收次数和回收效率都是

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 |

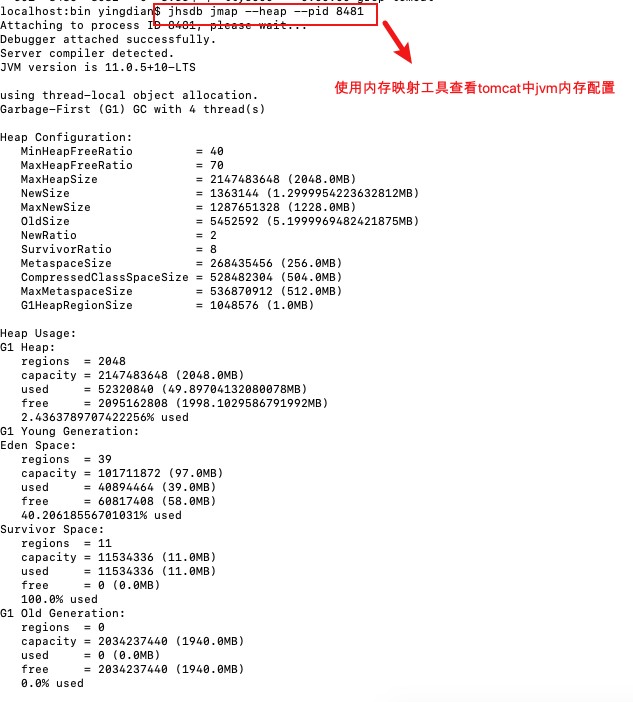
**JVM内存模型回顾**



**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**



1. 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"

### Tomcat 配置调优

##### 调整Tomcat线程池



##### 调整Tomcat的连接器

可以选择合适的连接器

|  |  |
| --- | --- |
| **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

AJP连接器默认开启，可以选择关闭，而且最近发现了漏洞。

##### 调整IO模式

Tomcat8之前的版本默认都是使用BIO（阻塞式IO），这种方式对于每一个请求都要创建一个线程来处理，不适合高并发场景；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（Apache Portable Runtime）模式，APR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JNI技术调用操作系统底层的IO接口

##### 配合Nginx动静分离

可以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）