TOMCAT封神之旅（一）-源码运行及整体架构

原创

gonghaiyu 于 2021-06-13 00:19:12 发布 30 收藏

分类专栏： Tomcat

本文链接：https://blog.csdn.net/gonghaiyu/article/details/116082925

TOMCAT目录结构

主要讲解下tomcat的conf目录。

1. 包括logging.properties，用于配置tomcat的日志信息。
2. server.xml用于配置tomcat容器。
3. tomcat-users.xml用于配置tomcat本身的访问用户及权限。
4. webapps下主要用于tomcat本身的样例信息及欢迎页面。

TOMCAT8源码运行

因本地下载的是tomcat8源码，下面配置tomcat8的源码进行安装运行。先确保本地安装了jdk和idea中配置了maven仓库。

创建tomcat运行目录

先在解压后的目录中创建home目录，然后将conf文件夹、webapps文件夹目录移动到home目录。

将tomcat8.5.42转换为Maven工程，添加 pom.xml 文件

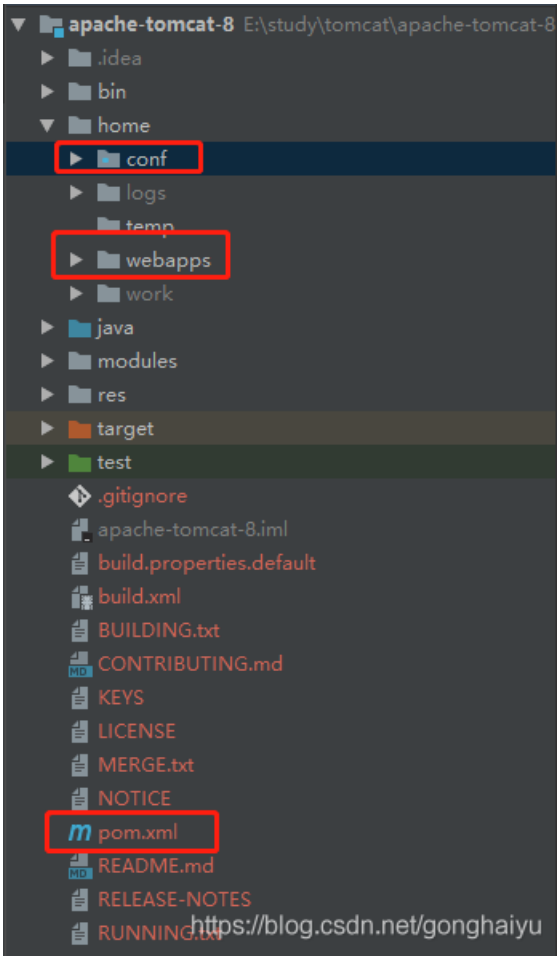
因是采用maven进行编译，先配置pom.xml文件。

```
<?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>Tomcat8</artifactId>
    <name>Tomcat8</name>
    <version>8</version>

    <build>
        <finalName>Tomcat8.0</finalName>
        <sourceDirectory>java</sourceDirectory>
        <testSourceDirectory>test</testSourceDirectory>
        <resources>
            <resource>
```

```
            <directory>java</directory>
        </resource>
    </resources>
    <testResources>
        <testResource>
            <directory>test</directory>
        </testResource>
    </testResources>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>2.3</version>
            <configuration>
                <encoding>UTF-8</encoding>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
        </plugin>
    </plugins>
</build>

<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.easymock</groupId>
        <artifactId>easymock</artifactId>
        <version>4.0.2</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>
```

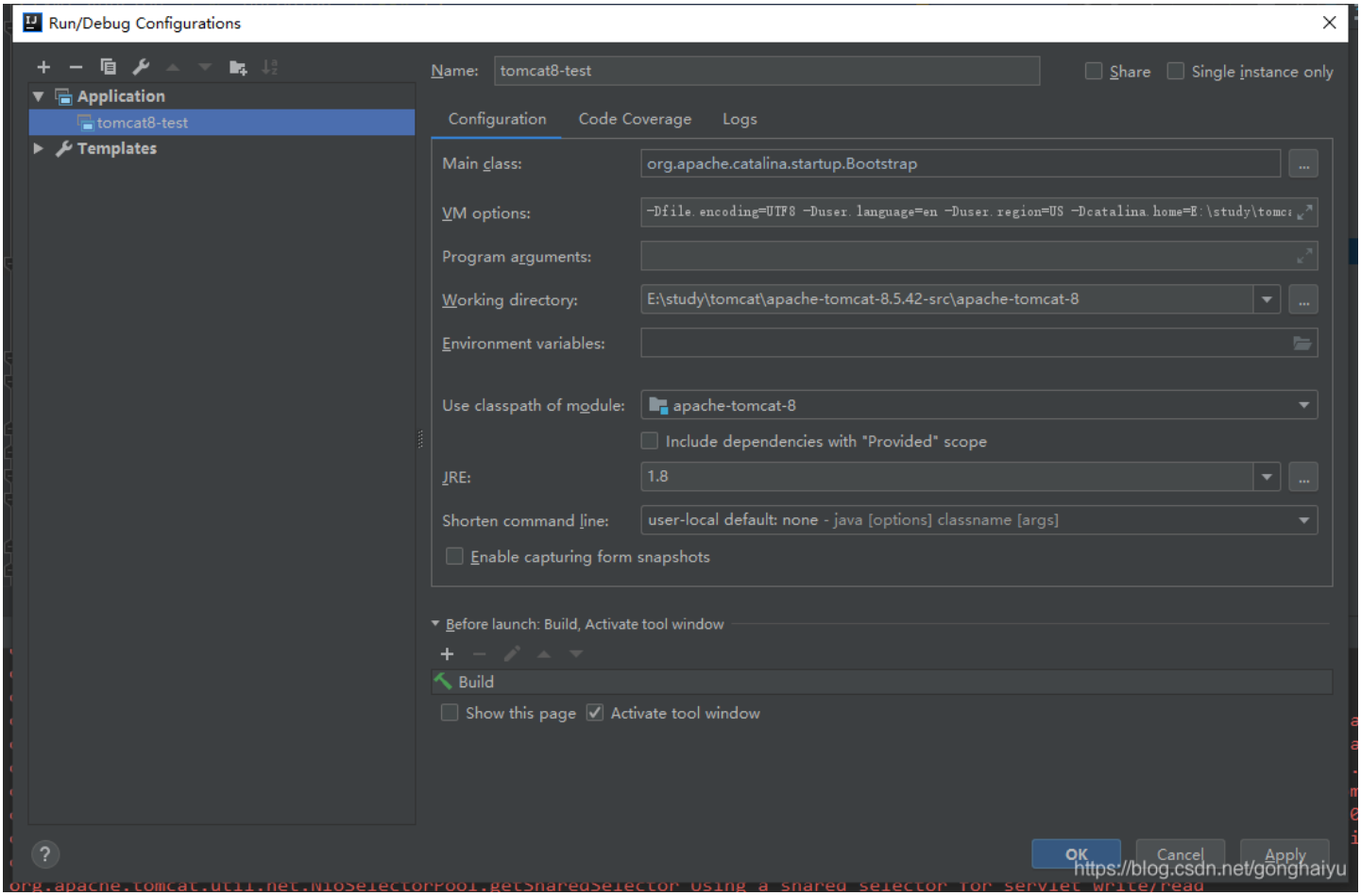
进行完上面两步操作后，目录结构如下。



配置启动类

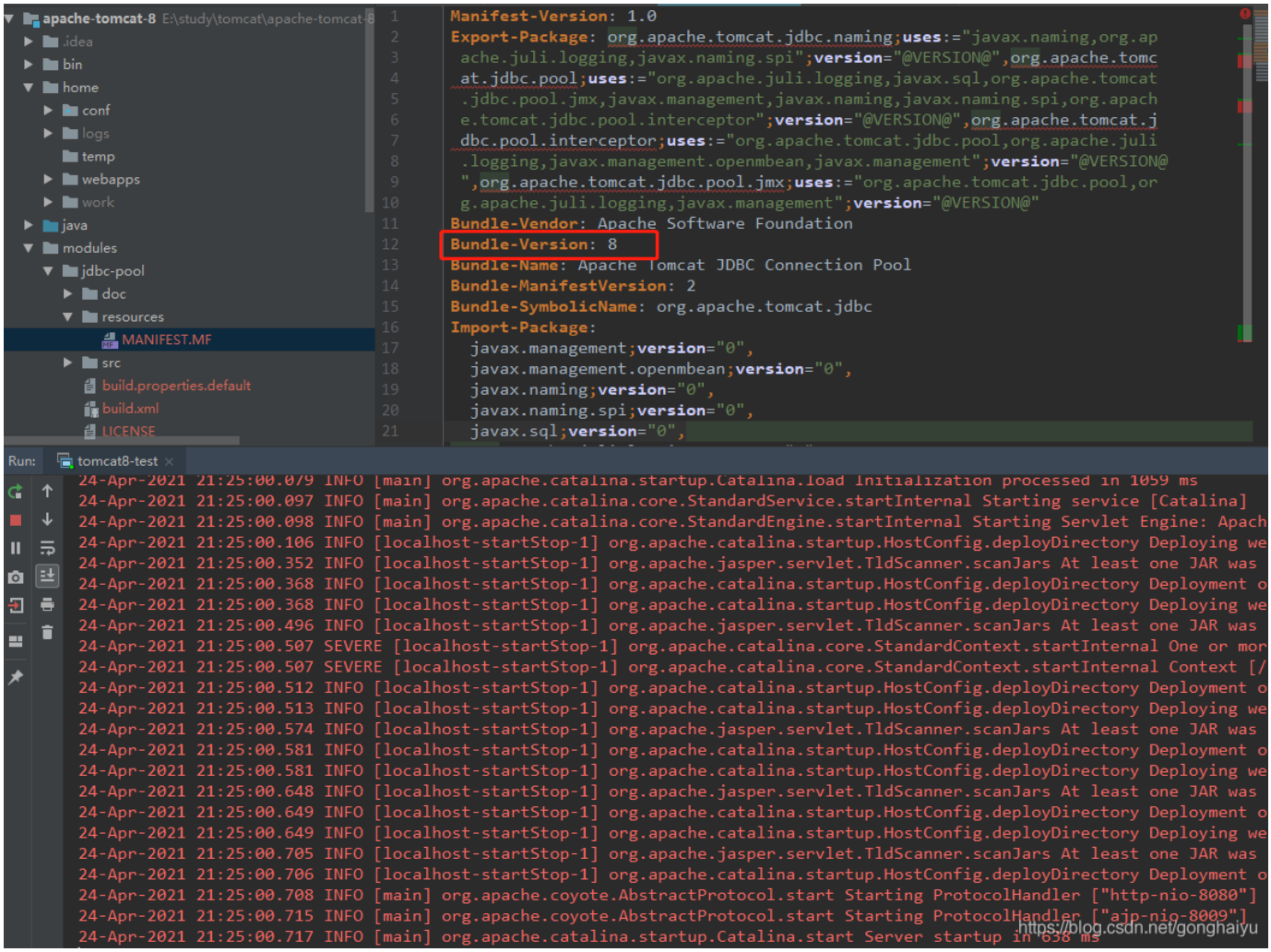
按下图配置启动类org.apache.catalina.startup.Bootstrap后，并增加启动参数。

```
-Dfile.encoding=UTF8
-Duser.language=en
-Duser.region=US
-Dcatalina.home=E:\study\tomcat\apache-tomcat-8.5.42-src\apache-tomcat-8\home
-Dcatalina.base=E:\study\tomcat\apache-tomcat-8.5.42-src\apache-tomcat-8\home
-Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager
-Djava.util.logging.config.file=E:\study\tomcat\apache-tomcat-8.5.42-src\apache-tomcat-8\home\conf\logging.prope
rties
```


Run/Debug Configurations
Application
tomcat8-test
Templates
Name: tomcat8-test
Share
Single instance only
Configuration
Code Coverage
Logs
Main class: org.apache.catalina.startup.Bootstrap
VM options: -Dfile.encoding=UTF8 -Duser.language=en -Duser.region=US -Dcatalina.home=E:\study\tomca
Program arguments:
Working directory: E:\study\tomcat\apache-tomcat-8.5.42-src\apache-tomcat-8
Environment variables:
Use classpath of module: apache-tomcat-8
Include dependencies with "Provided" scope
JRE: 1.8
Shorten command line: user-local default: none - java [options] classname [args]
Enable capturing form snapshots
Before launch: Build, Activate tool window
Build
Show this page
Activate tool window
OK
Cancel
Apply
https://blog.csdn.net/gonghaiyu

此时启动会报错，

1. TestCookieFilter中的CookieFilter 找不到，两种方案：1、注解注释；2补全这个代码。这里采用注释这个测试类。
2. 第二个错是编译时，这个MANIFEST.MF报错。按照红框中修改即可。

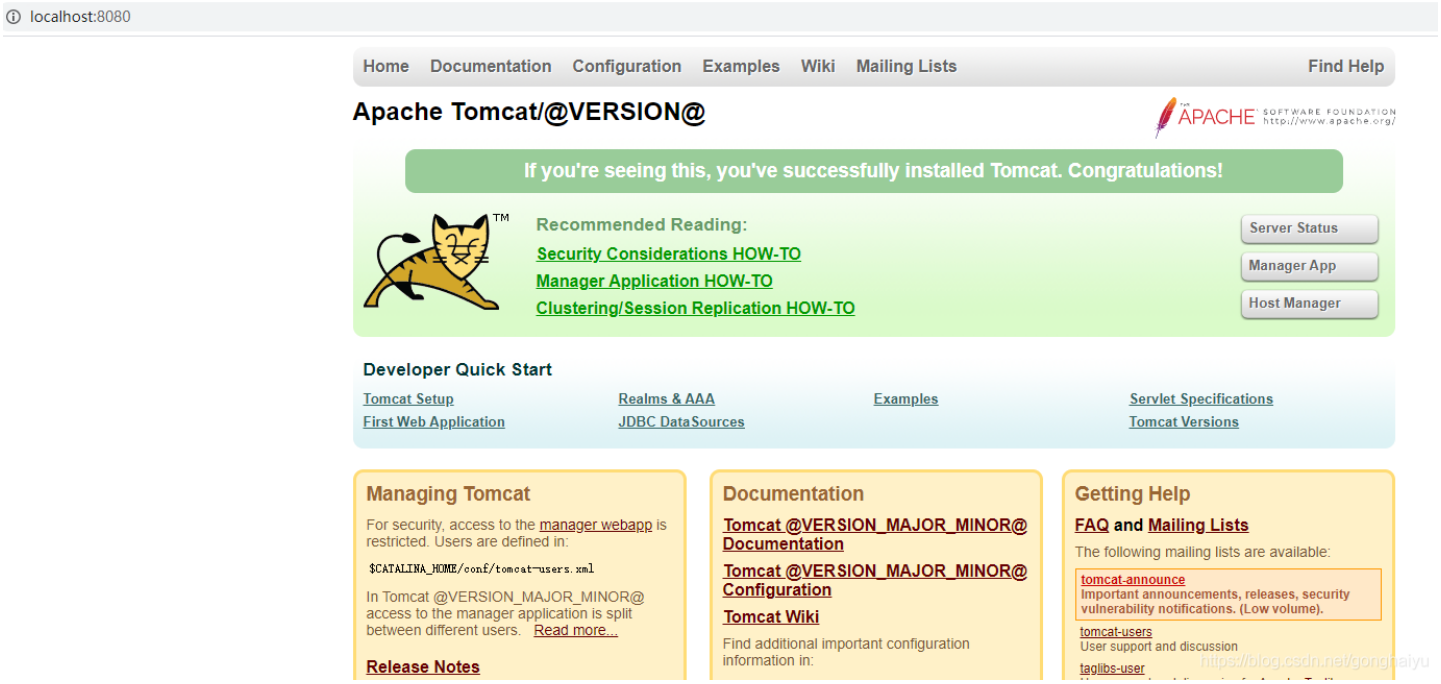


启动

此时启动，访问页面时会报错。需要配置jsp解析器，添加以下代码。

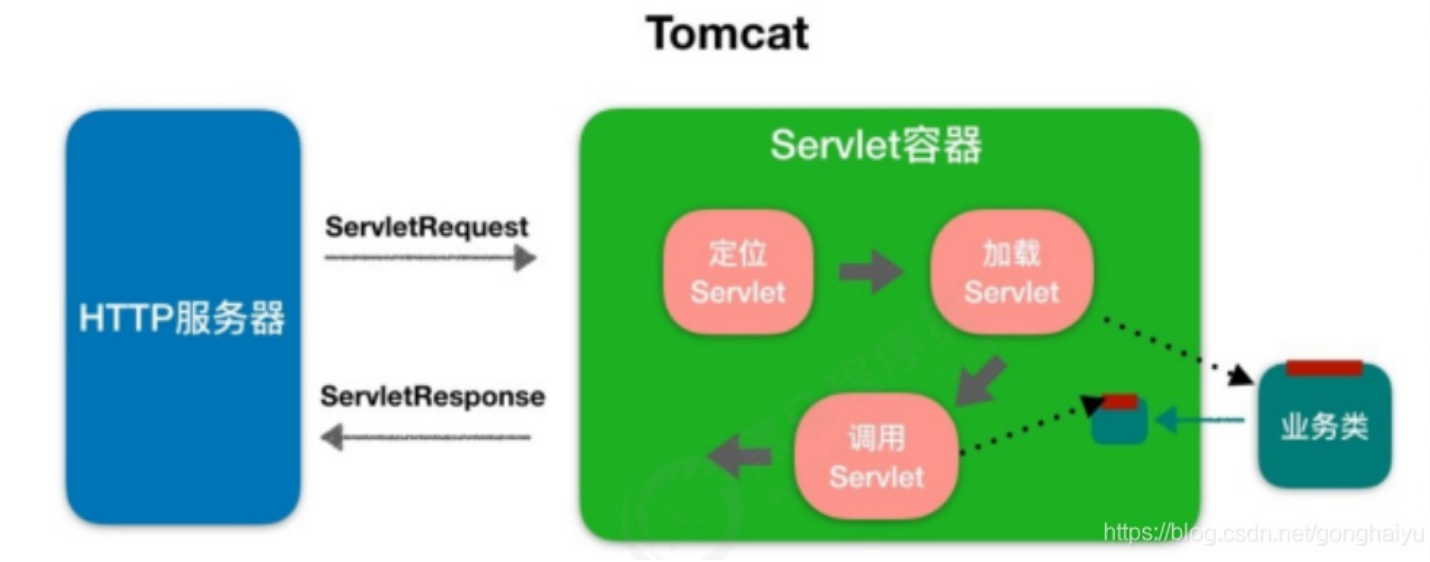
```java
/**
     * Process a "contextConfig" event for this Context.
     */
    protected synchronized void configureStart() {
        // Called from StandardContext.start()

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

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

        webConfig();
        // 配置jsp解析器
        context.addServletContainerInitializer(new JasperInitializer(), null);
       // 配置jsp解析器
        if (!context.getIgnoreAnnotations()) {
            applicationAnnotationsConfig();
        }
```

再次启动



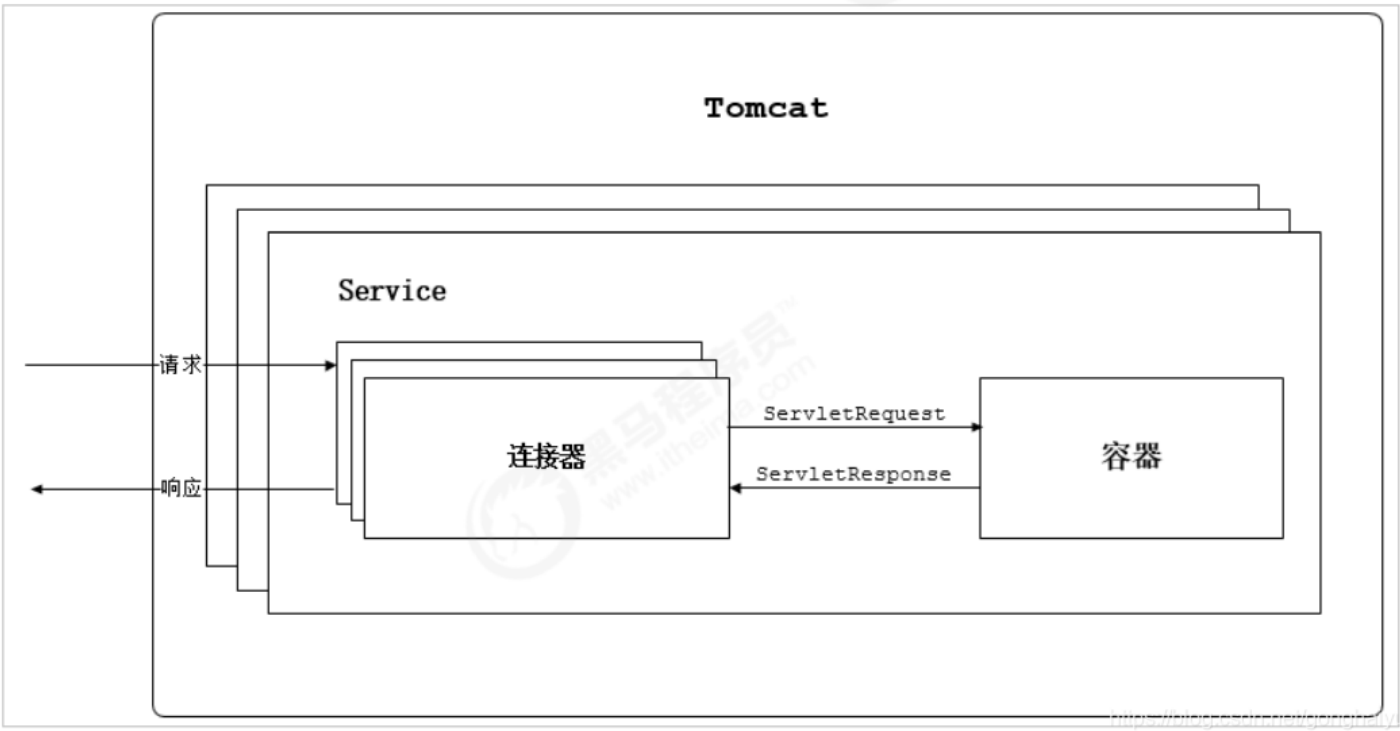
Servlet容器工作流程

为了解耦，HTTP服务器不直接调用Servlet，而是把请求交给Servlet容器来处理，那Servlet容器又是怎么工作的呢？
当客户请求某个资源时，HTTP服务器会用一个ServletRequest对象把客户的请求信息封装起来，然后调用Servlet容器的service方法，Servlet容器拿到请求后，根据请求的URL和Servlet的映射关系，找到相应的Servlet，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，接着调用Servlet的service方法来处理请求，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。



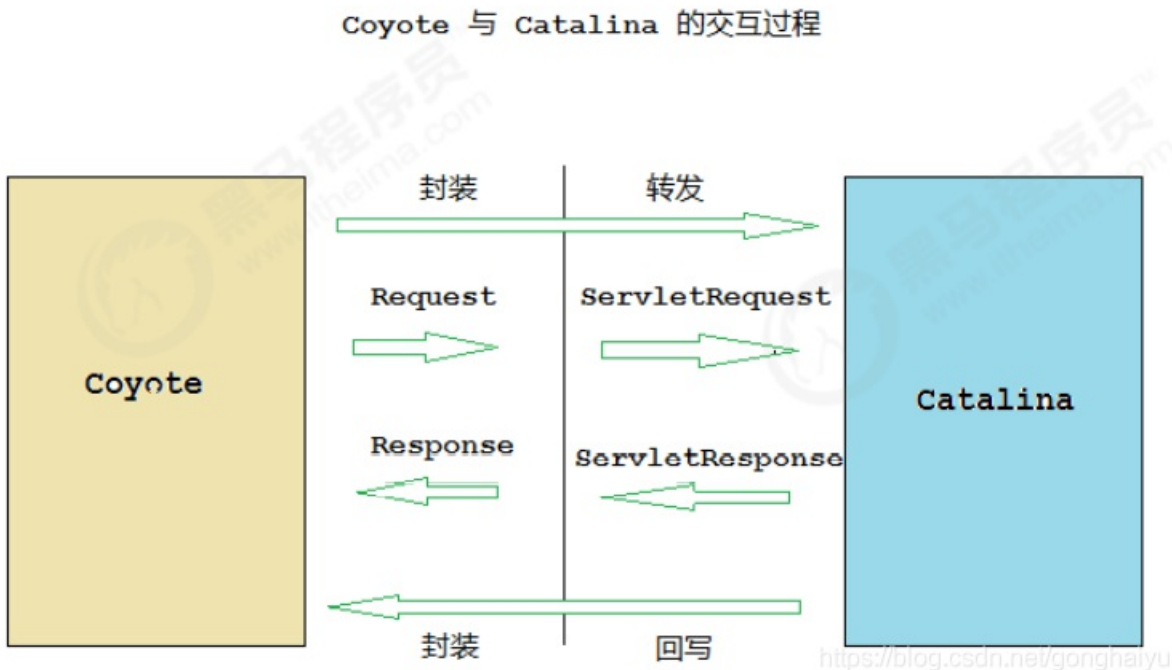
Tomcat整体架构

我们已经了解了Tomcat要实现两个核心功能：
1） 处理Socket连接，负责网络字节流与Request和Response对象的转化。
2） 加载和管理Servlet，以及具体处理Request请求。
因此Tomcat设计了两个核心组件连接器（Connector）和容器（Container）来分别做这两件事情。连接器负责对外交流，容器负责内部处理。



连接器 - Coyote

Coyote 是Tomcat的连接器框架的名称 , 是Tomcat服务器提供的供客户端访问的外部接口。客户端通过Coyote与服务器建立连接、发送请求并接受响应 。
Coyote 封装了底层的网络通信（Socket 请求及响应处理），为Catalina 容器提供了统一的接口，使Catalina 容器与具体的请求协议及IO操作方式完全解耦。Coyote 将Socket 输入转换封装为 Request 对象，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 。
Coyote 作为独立的模块，只负责具体协议和IO的相关操作， 与Servlet 规范实现没有直接关系，因此即便是 Request 和 Response 对象也并未实现Servlet规范对应的接口， 而是在Catalina 中将他们进一步封装为ServletRequest 和 ServletResponse 。



IO模型与协议

在Coyote中， Tomcat支持的多种I/O模型和应用层协议，具体包含哪些IO模型和应用层协议，请看下表：
Tomcat 支持的IO模型（自8.5/9.0 版本起，Tomcat 移除了 对 BIO 的支持）：

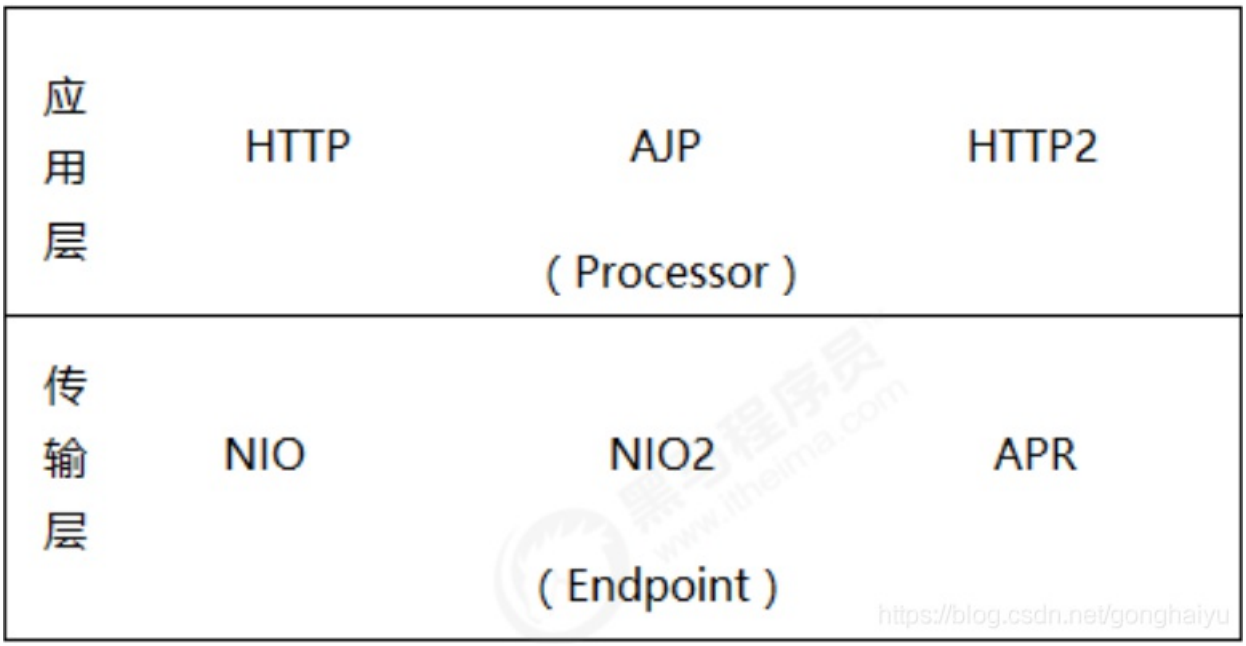
IO模型	描述
NIO	非阻塞I/O，采用Java NIO类库实现。
NIO2	异步I/O，采用JDK 7最新的NIO2类库实现。
APR	采用Apache可移植运行库实现，是C/C++编写的本地库。如果选择该方案，需要单独安装APR库。

Tomcat 支持的应用层协议 ：

应用层协议	描述

议	
HTTP/1.1	这是大部分Web应用采用的访问协议。
AJP	用于和Web服务器集成（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。

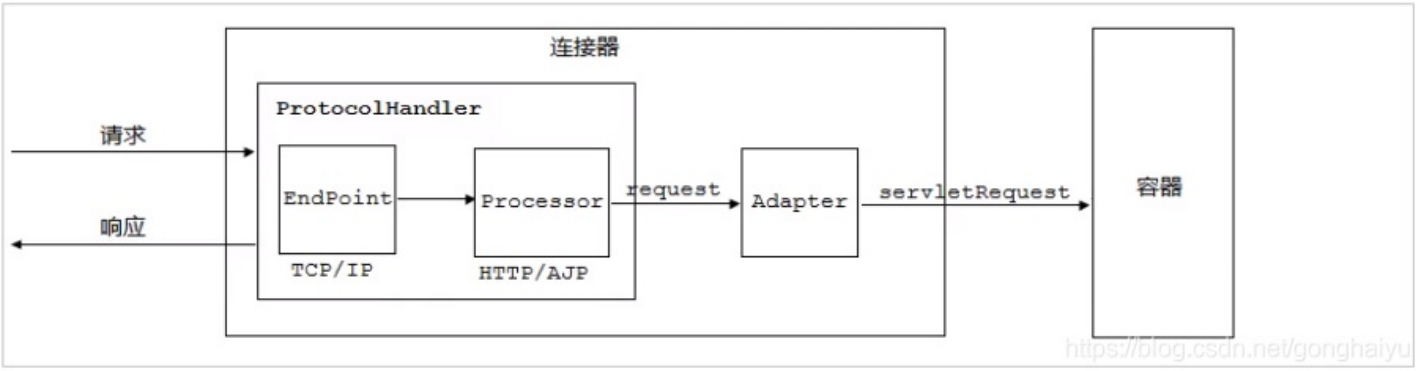
协议分层：



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。

Tomcat为了实现支持多种I/O模型和应用层协议，一个容器可能对接多个连接器，就好比一个房间有多个门。但是单独的连接器或者容器都不能对外提供服务，需要把它们组装起来才能工作，组装后这个整体叫作Service组件。这里请你注意，Service本身没有做什么重要的事情，只是在连接器和容器外面多包了一层，把它们组装在一起。Tomcat内可能有多个Service，这样的设计也是出于灵活性的考虑。通过在Tomcat中配置多个Service，可以实现通过不同的端口号来访问同一台机器上部署的不同应用。

连接器组件



连接器中的各个组件的作用如下：

1. EndPoint

1. EndPoint：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。
2. Tomcat 并没有EndPoint 接口，而是提供了一个抽象类AbstractEndpoint， 里面定义了两个内部类：Acceptor和SocketProcessor。Acceptor用于监听Socket连接请求。SocketProcessor用于处理接收到的Socket请求，它实现Runnable接口，在Run方法里调用协议处理组件Processor进行处理。为了提高处理能力，SocketProcessor被提交到线程池来执行。而这个线程池叫作执行器（Executor)，我在后面的专栏会详细介绍Tomcat如何扩展原生的Java线程池。

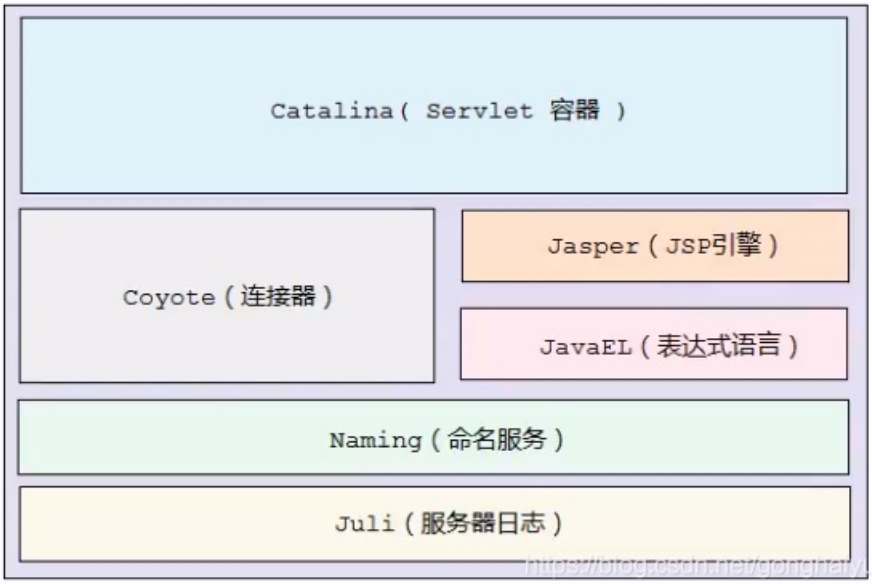
2. Processor
   Processor： Coyote 协议处理接口，如果说EndPoint是用来实现TCP/IP协议的，那么Processor用来实现HTTP协议，Processor接收来自EndPoint的Socket，读取字节流解析成Tomcat Request和Response对象，并通过Adapter将其提交到容器处理，Processor是对应用层协议的抽象。
3. ProtocolHandler
   ProtocolHandler： Coyote 协议接口， 通过Endpoint 和 Processor， 实现针对具体协议的处理能力。Tomcat 按照协议和I/O 提供了6个实现类： AjpNioProtocol，AjpAprProtocol， AjpNio2Protocol， Http11NioProtocol，Http11Nio2Protocol，Http11AprProtocol。我们在配置tomcat/conf/server.xml 时， 至少要指定具体的ProtocolHandler , 当然也可以指定协议名称， 如： HTTP/1.1，如果安装了APR，那么将使用Http11AprProtocol， 否则使用 Http11NioProtocol 。
4. 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“存放”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，也就意味着，不能用TomcatRequest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TomcatRequest对象，CoyoteAdapter负责将Tomcat Request转成ServletRequest，再调用容器的Service方法。

容器 - Catalina

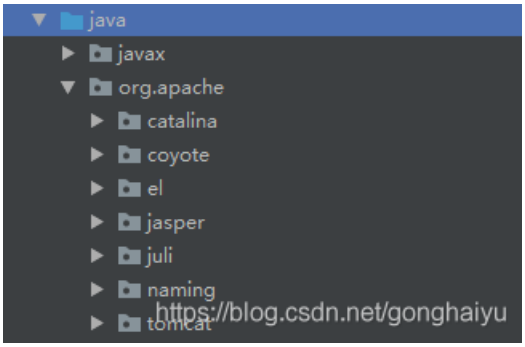
Tomcat是一个由一系列可配置的组件构成的Web容器，而Catalina是Tomcat的servlet容器。
Catalina 是Servlet 容器实现，包含了之前讲到的所有的容器组件，以及后续章节涉及到的安全、会话、集群、管理等Servlet 容器架构的各个方面。它通过松耦合的方式集成Coyote，以完成按照请求协议进行数据读写。同时，它还包括我们的启动入口、Shell程序等。

Catalina 地位

Tomcat 的模块分层结构图， 如下：

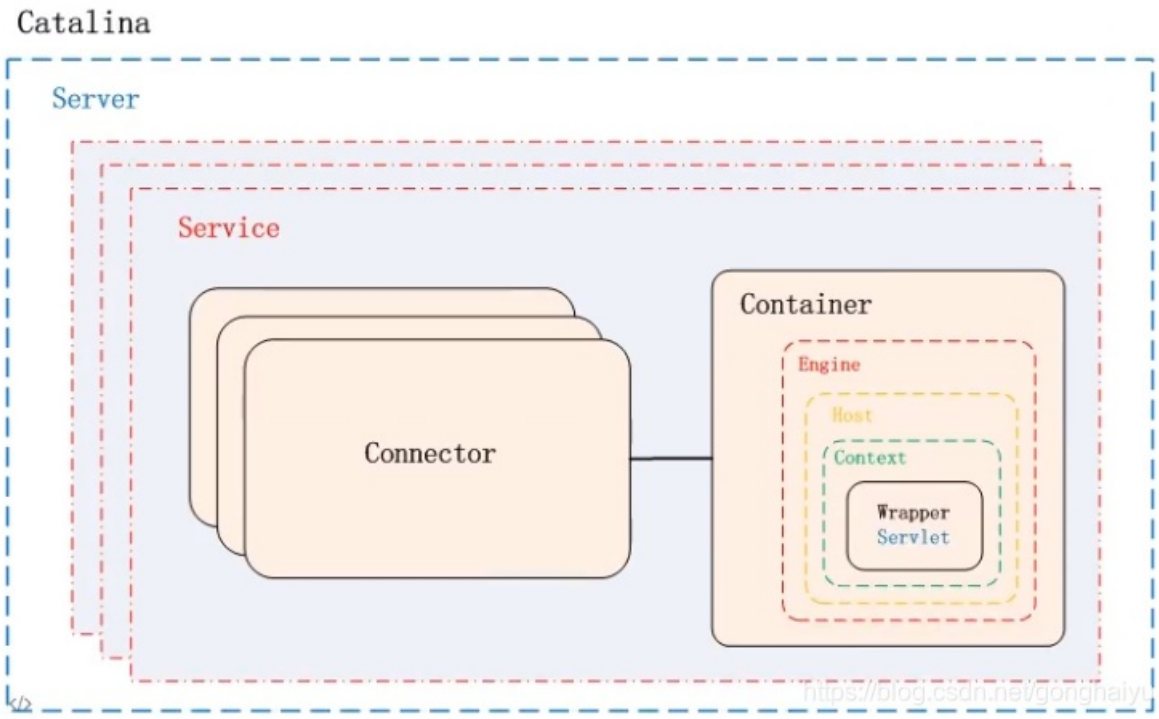


上面的结构与包结构是相互匹配的，如下所示。



Catalina 结构

Catalina 的主要组件结构如下：



如上图所示，Catalina负责管理Server，而Server表示着整个服务器。Server下面有多个服务Service，每个服务都包含着多个连接器组件Connector（Coyote 实现）和一个容器组件Container。在Tomcat 启动的时候， 会初始化一个Catalina的实例。

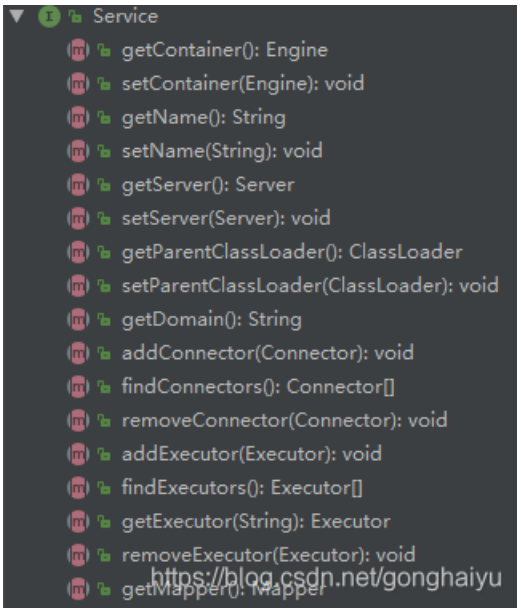
上面这个图与service.xml文件是相匹配的。

```
<?xml version="1.0" encoding="UTF-8"?>
<Server port="8005" shutdown="SHUTDOWN">
  <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 -->
  <Listener className="org.apache.catalina.core.AprLifecycleListener" SSLEngine="on" />
  <!-- Prevent memory leaks due to use of particular java/javax APIs-->
  <Listener className="org.apache.catalina.core.JreMemoryLeakPreventionListener" />
  <Listener className="org.apache.catalina.mbeans.GlobalResourcesLifecycleListener" />
  <Listener className="org.apache.catalina.core.ThreadLocalLeakPreventionListener" />

  <GlobalNamingResources>
    <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>

  <Service name="Catalina">

    <Connector port="8080" protocol="HTTP/1.1"
               connectionTimeout="20000"
               redirectPort="8443" />
    <Connector port="8009" protocol="AJP/1.3" redirectPort="8443" />
    <Engine name="Catalina" defaultHost="localhost">

      <Realm className="org.apache.catalina.realm.LockOutRealm">

        <Realm className="org.apache.catalina.realm.UserDatabaseRealm"
               resourceName="UserDatabase"/>
      </Realm>

      <Host name="localhost"  appBase="webapps"
            unpackWARs="true" autoDeploy="true">
        <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>
    </Engine>
  </Service>
</Server>
```

Catalina 各个组件的职责：

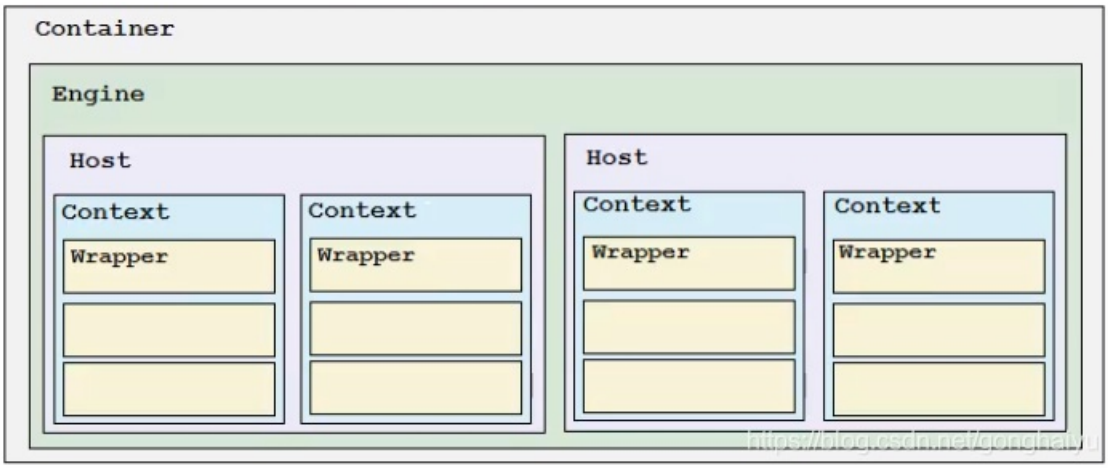
组件	职责
Catalina	负责解析Tomcat的配置文件，以此来创建服务器Server组件，并根据命令来对其进行管理
Server	服务器表示整个Catalina Servlet容器以及其它组件，负责组装并启动Servlet引擎,Tomcat连接器。Server通过实现Lifecycle接口，提供了一种优雅的启动和关闭整个系统的方式
Service	服务是Server内部的组件，一个Server包含多个Service。它将若干个Connector组件绑定到一个Container（Engine）上
Connector	连接器，处理与客户端的通信，它负责接收客户请求，然后转给相关的容器处理，最后向客户返回响应结果
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

service中的方法：



Container 结构

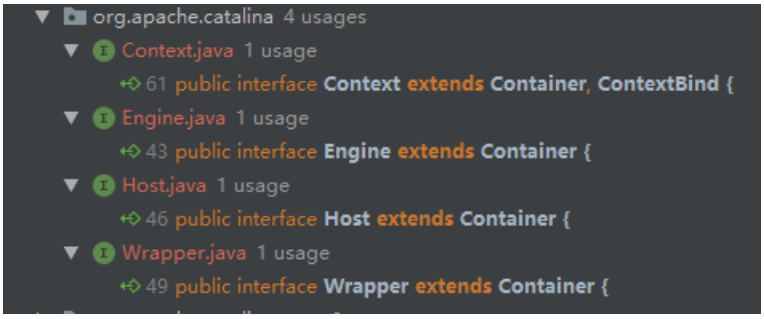
Tomcat设计了4种容器，分别是Engine、Host、Context和Wrapper。这4种容器不是平行关系，而是父子关系。， Tomcat通过一种分层的架构，使得Servlet容器具有很好的灵活性。

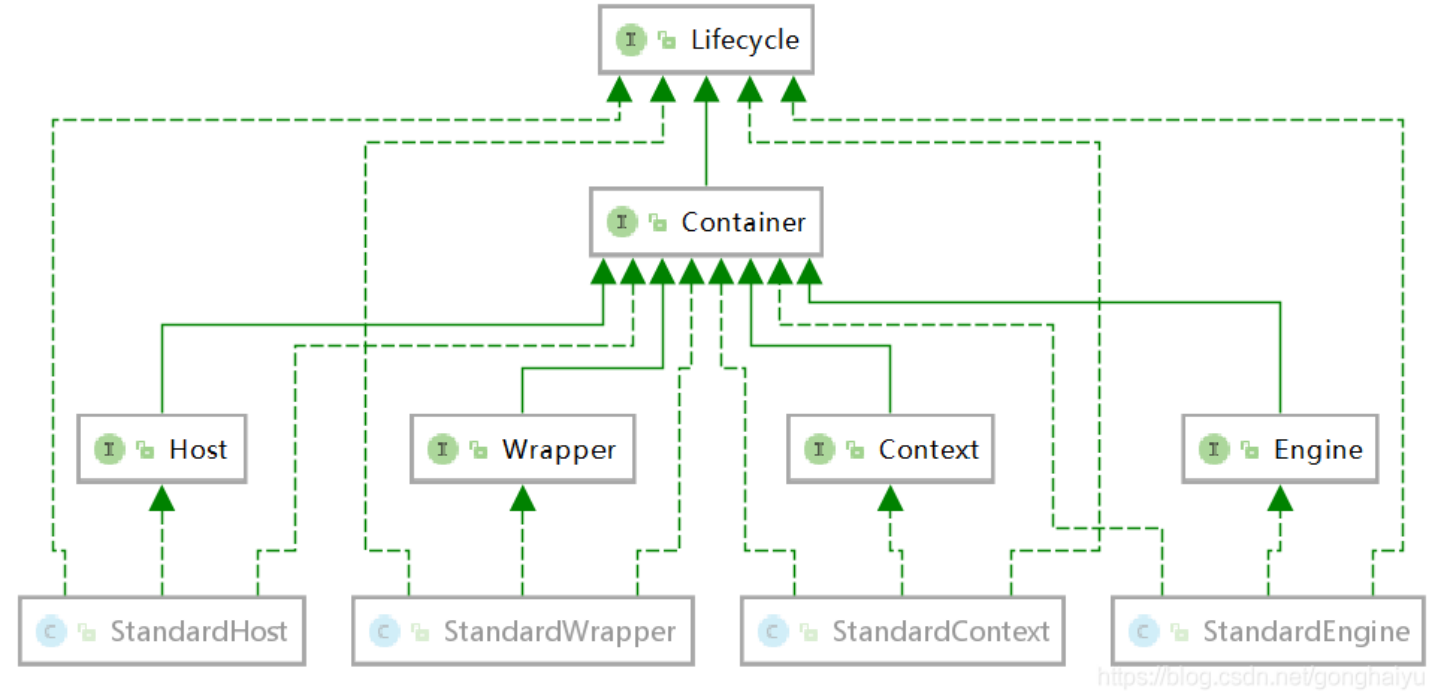


各个组件的含义：

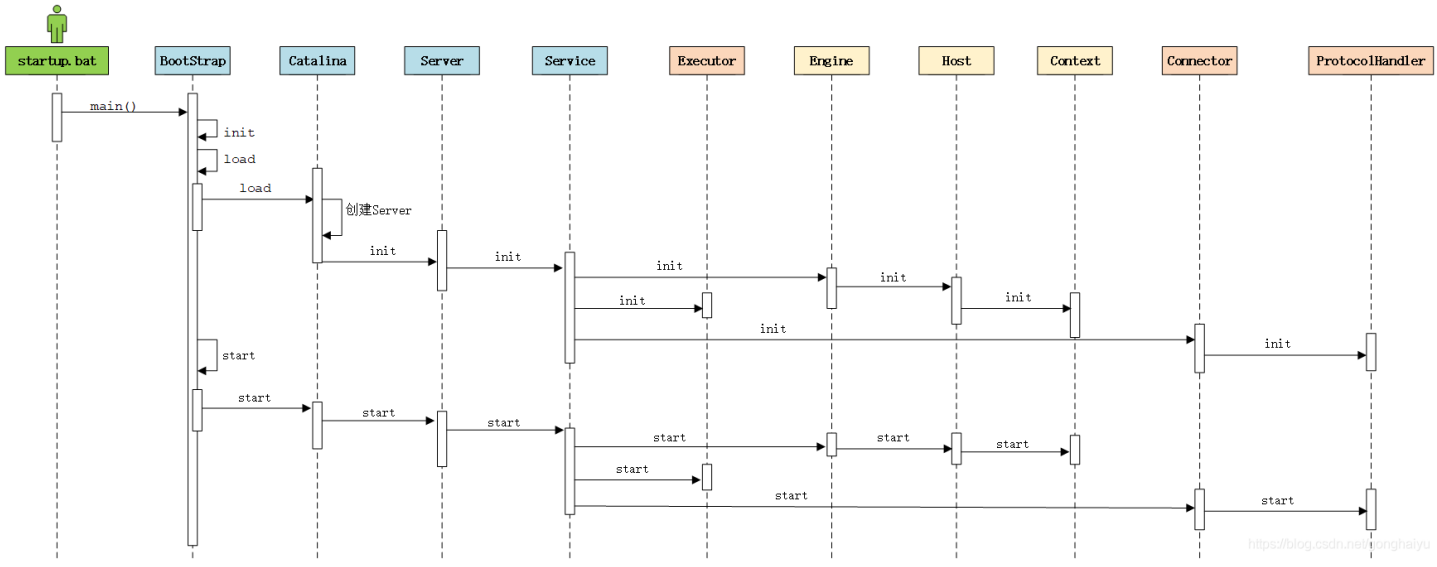
容器	描述
Engine	表示整个Catalina的Servlet引擎，用来管理多个虚拟站点，一个Service最多只能有一个Engine，但是一个引擎可包含多个Host
Host	代表一个虚拟主机，或者说一个站点，可以给Tomcat配置多个虚拟主机地址，而一个虚拟主机下可包含多个Context
Context	表示一个Web应用程序， 一个Web应用可包含多个Wrapper
Wrapper	表示一个Servlet，Wrapper 作为容器中的最底层，不能包含子容器

我们也可以再通过Tomcat的server.xml配置文件来加深对Tomcat容器的理解。Tomcat采用了组件化的设计，它的构成组件都是可配置的，其中最外层的是Server，其他组件按照一定的格式要求配置在这个顶层容器中。
那么，Tomcat是怎么管理这些容器的呢？你会发现这些容器具有父子关系，形成一个树形结构，你可能马上就想到了设计模式中的组合模式。没错，Tomcat就是用组合模式来管理这些容器的。具体实现方法是，所有容器组件都实现了Container接口，因此组合模式可以使得用户对单容器对象和组合容器对象的使用具有一致性。这里单容器对象指的是最底层的Wrapper，组合容器对象指的是上面的Context、Host或者Engine。





Tomcat 启动流程



主要目的：加载Tomcat的配置文件，初始化容器组件 ，监听对应的端口号， 准备接受客户端请求。

启动源码解析

Lifecycle

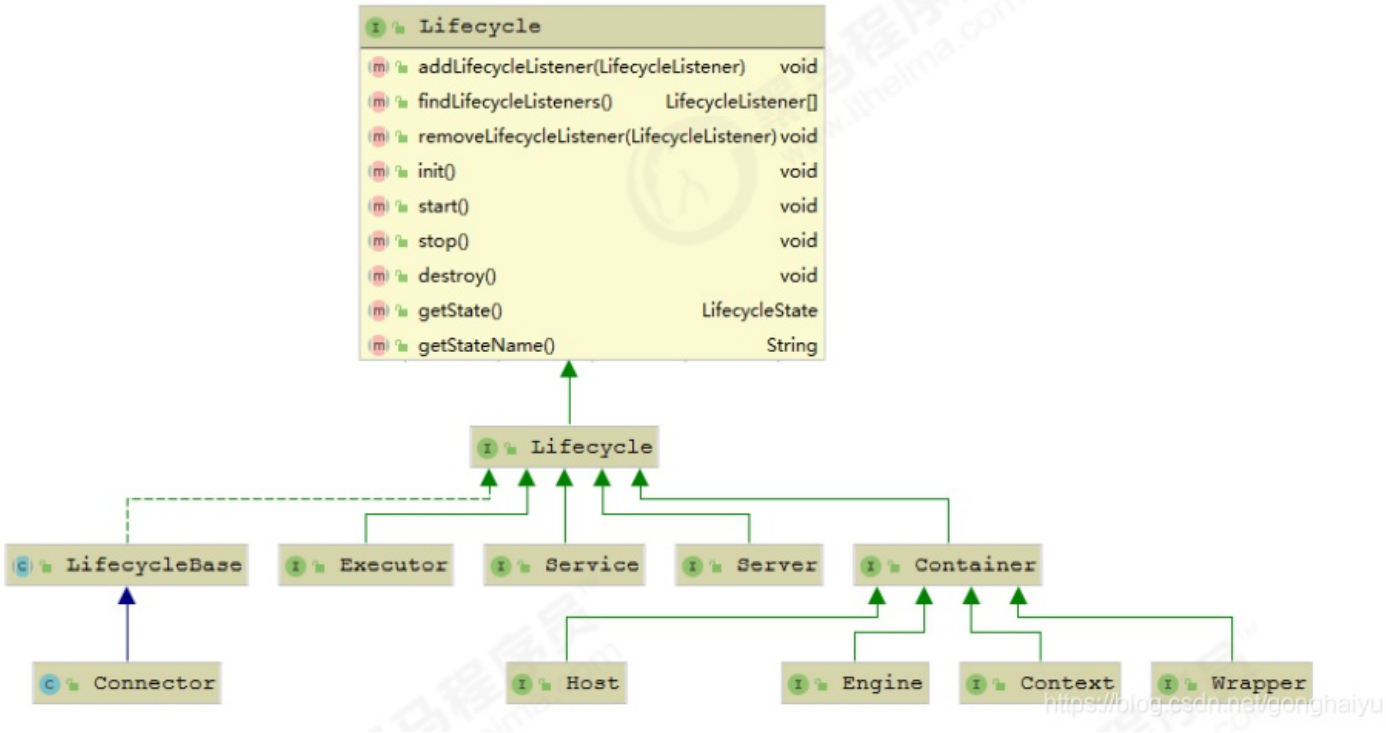
由于所有的组件均存在初始化、启动、停止等生命周期方法，拥有生命周期管理的特性， 所以Tomcat在设计的时候， 基于生命周期管理抽象成了一个接口 Lifecycle ， 而组件 Server、Service、Container、Executor、Connector 组件 ， 都实现了一个生命周期的接口，从而具有了以下生命周期中的核心方法：

1） init（）：初始化组件

2） start（）：启动组件

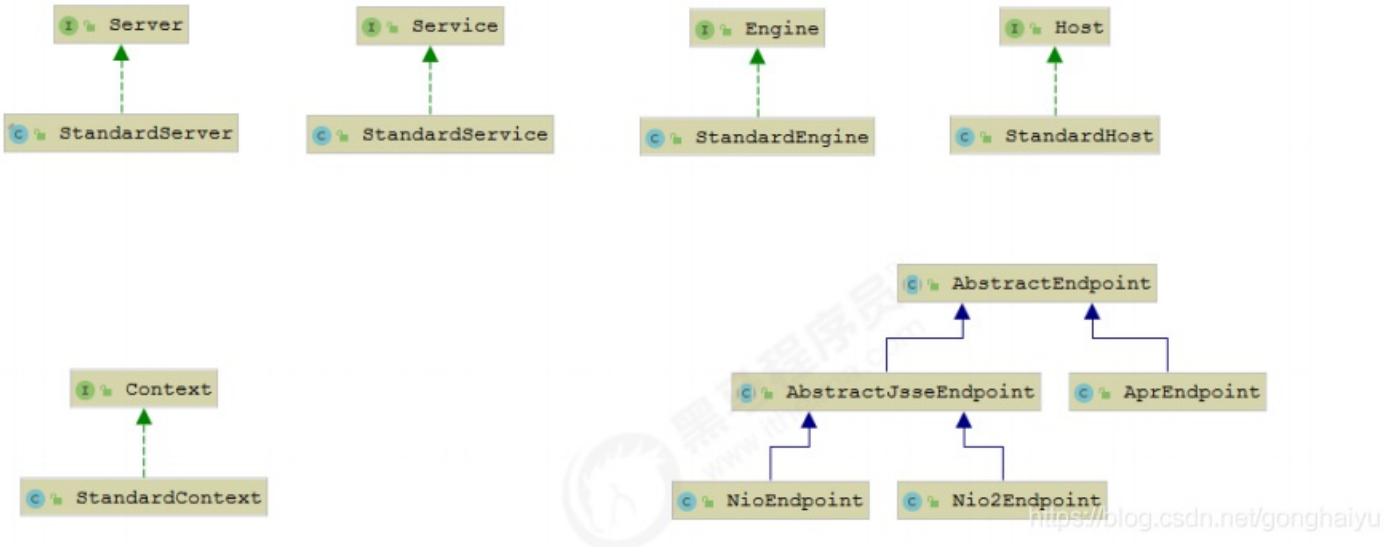
3） stop（）：停止组件

4） destroy（）：销毁组件



各组件的默认实现

上面我们提到的Server、Service、Engine、Host、Context都是接口， 下图中罗列了这些接口的默认实现类。当前对于 Endpoint 组件来说，在Tomcat中没有对应的Endpoint接口， 但是有一个抽象类 AbstractEndpoint ，其下有三个实现类： NioEndpoint、Nio2Endpoint、AprEndpoint ， 这三个实现类，分别对应于前面讲解链接器 Coyote时， 提到的链接器支持的三种IO模型：NIO，NIO2，APR ， Tomcat8.5版本中，默认采用的是 NioEndpoint。



ProtocolHandler： Coyote协议接口，通过封装Endpoint和Processor， 实现针对具体协议的处理功能。Tomcat按照协议和IO提供了6个实现类。

AJP协议（Apache JServ Protocol）：

1） AjpNioProtocol：采用NIO的IO模型。

2） AjpNio2Protocol：采用NIO2的IO模型。

3） AjpAprProtocol：采用APR的IO模型，需要依赖于APR库。

HTTP协议：

1） Http11NioProtocol：采用NIO的IO模型，默认使用的协议（如果服务器没有安装APR）。

2） Http11Nio2Protocol：采用NIO2的IO模型。

3） Http11AprProtocol：采用APR的IO模型，需要依赖于APR库。

源码入口

通过catilina.sh中的代码查看，可以知道tomcat的main方法入口在BootStrap类下。每一级的组件除了完成自身的处理外，还要负责调用子组件响应的生命周期管理方法，组件与组件之间是松耦合的，因为我们可以很容易的通过配置文件进行修改和替换。

```
/**
     * Main method and entry point when starting Tomcat via the provided
     * scripts.
     *
     * @param args Command line arguments to be processed
     */
    public static void main(String args[]) {

        if (daemon == null) {
            // Don't set daemon until init() has completed
            Bootstrap bootstrap = new Bootstrap();
            try {
                bootstrap.init();
            } catch (Throwable t) {
                handleThrowable(t);
                t.printStackTrace();
                return;
            }
            daemon = bootstrap;
        } else {
            // When running as a service the call to stop will be on a new
            // thread so make sure the correct class loader is used to prevent
            // a range of class not found exceptions.
```

```
            Thread.currentThread().setContextClassLoader(daemon.catalinaLoader);
        }

        try {
            String command = "start";
            if (args.length > 0) {
                command = args[args.length - 1];
            }

            if (command.equals("startd")) {
                args[args.length - 1] = "start";
                daemon.load(args);
                daemon.start();
            } else if (command.equals("stopd")) {
                args[args.length - 1] = "stop";
                daemon.stop();
            } else if (command.equals("start")) {
                daemon.setAwait(true);
                daemon.load(args);
                daemon.start();
                if (null == daemon.getServer()) {
                    System.exit(1);
                }
            } else if (command.equals("stop")) {
                daemon.stopServer(args);
            } else if (command.equals("configtest")) {
                daemon.load(args);
                if (null == daemon.getServer()) {
                    System.exit(1);
                }
                System.exit(0);
            } else {
                log.warn("Bootstrap: command \"" + command + "\" does not exist.");
            }
        } catch (Throwable t) {
            // Unwrap the Exception for clearer error reporting
            if (t instanceof InvocationTargetException &&
                    t.getCause() != null) {
                t = t.getCause();
            }
            handleThrowable(t);
            t.printStackTrace();
            System.exit(1);
        }

    }
```

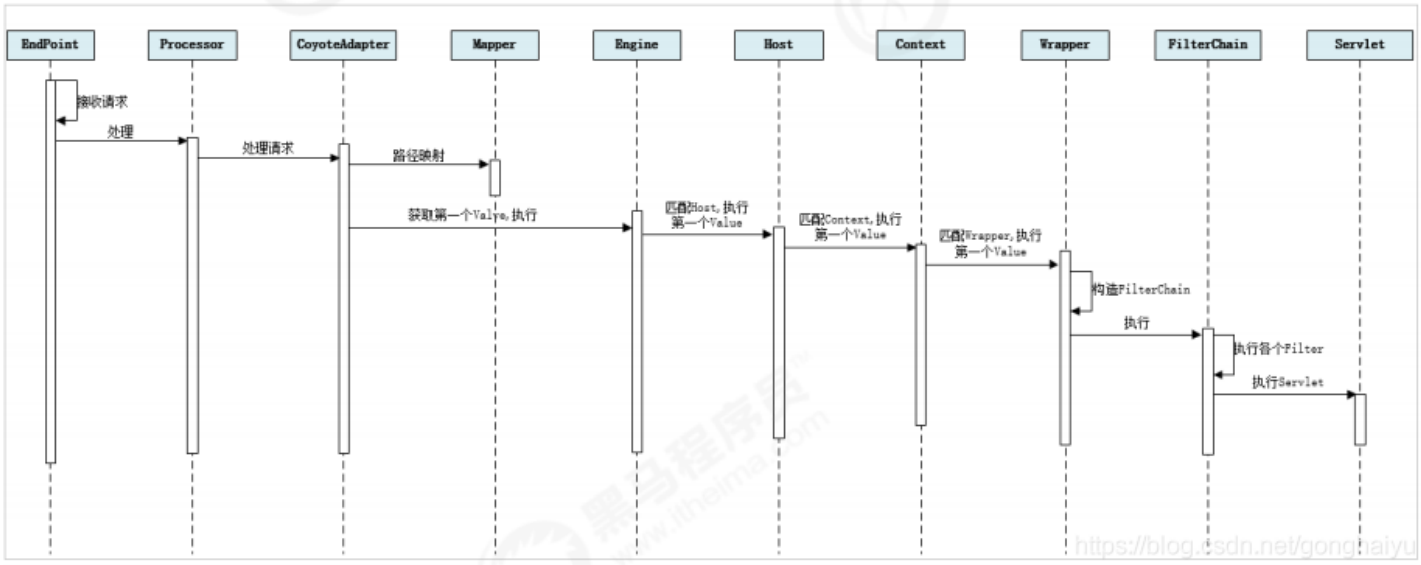
Tomcat请求处理流程

请求流程

设计了这么多层次的容器，Tomcat是怎么确定每一个请求应该由哪个Wrapper容器里的Servlet来处理的呢？答案是，Tomcat是用Mapper组件来完成这个任务的。这一点与Mybatis中如何定位到查询的语句思路是一致的。
Mapper组件的功能就是将用户请求的URL定位到一个Servlet，它的工作原理是：Mapper组件里保存了Web应用的配置信息，其实就是容器组件与访问路径的映射关系，比如Host容器里配置的域名、Context容器里的Web应用路径，以及Wrapper容器里Servlet映射的路径，你可以想象这些配置信息就是一个多层次的Map。

当一个请求到来时，Mapper组件通过解析请求URL里的域名和路径，再到自己保存的Map里去查找，就能定位到一个Servlet。
请你注意，一个请求URL最后只会定位到一个Wrapper容器，也就是一个Servlet。
下面的示意图中， 就描述了 当用户请求链接 http://www.itcast.cn/bbs/findAll 之后, 是如何找到最终处理业务逻辑的servlet。

下面我们再来解析一下， 从Tomcat的设计架构层面来分析Tomcat的请求处理。



步骤如下:

1. Connector组件Endpoint中的Acceptor监听客户端套接字连接并接收Socket。
2. 将连接交给线程池Executor处理，开始执行请求响应任务。
3. Processor组件读取消息报文，解析请求行、请求体、请求头，封装成Request对象。
4. Mapper组件根据请求行的URL值和请求头的Host值匹配由哪个Host容器、Context容器、Wrapper容器处理请求。
5. CoyoteAdaptor组件负责将Connector组件和Engine容器关联起来，把生成的Request对象和响应对象Response传递到Engine容器中，调用 Pipeline。
6. Engine容器的管道开始处理，管道中包含若干个Valve、每个Valve负责部分处理逻辑。执行完Valve后会执行基础的Valve–StandardEngineValve，负责调用Host容器的Pipeline。
7. Host容器的管道开始处理，流程类似，最后执行 Context容器的Pipeline。
8. Context容器的管道开始处理，流程类似，最后执行 Wrapper容器的Pipeline。
9. Wrapper容器的管道开始处理，流程类似，最后执行 Wrapper容器对应的Servlet对象的处理方法。

参考

tomcat8源码编译：https://www.cnblogs.com/bjlhx/p/14348256.html