

主讲老师: Fox

课前须知:

- 本专题讲解的Tomcat版本: **apache-tomcat-9.0.55**, Tomcat9源码阅读环境搭建: [Tomcat9源码编译](#)
 - 本专题Tomcat安排如下:
 - Tomcat的整体架构及其设计精髓
 - Tomcat的线程模型详解及其调优
 - Tomcat的类加载机制和热加载热部署的实现原理
- 拓展: 对Tomcat源码实现感兴趣的同学可以去看周瑜老师的[Tomcat源码分析专题](#)
- 有道云笔记地址: <https://note.youdao.com/s/Mz5PXNYw>

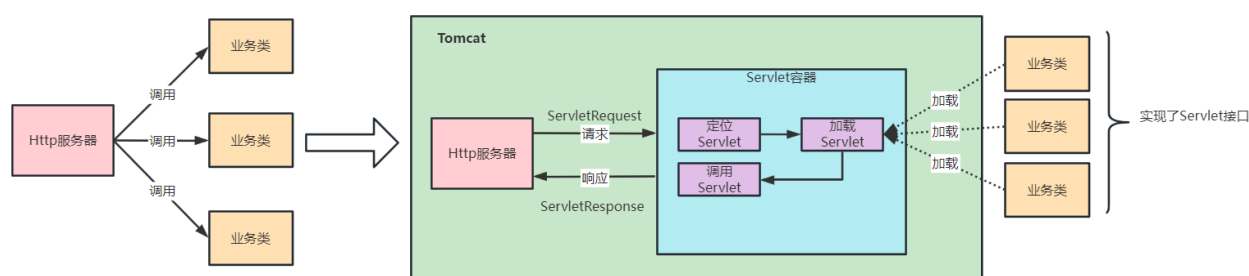
1. Tomcat介绍

官方文档: <https://tomcat.apache.org/tomcat-9.0-doc/index.html>

1.1 Tomcat概念

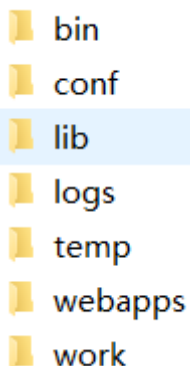
Tomcat是Apache Software Foundation (Apache软件基金会) 开发的一款开源的Java Servlet容器。它是一种Web服务器, 用于在服务器端运行Java Servlet和JavaServer Pages (JSP)技术。它可以为Java Web应用程序提供运行环境, 并通过HTTP协议处理客户端请求。Tomcat也支持多种Web应用程序开发技术, 例如JavaServer Faces (JSF)、Java Persistence API (JPA)等。总的来说, Tomcat是一款高效、稳定和易于使用的Web服务器。

Tomcat核心: Http服务器+Servlet容器



1.2 Tomcat目录结构

Tomcat的解压之后的目录可以看到如下的目录结构



- bin
bin目录主要是用来存放tomcat的脚本，如`startup.sh` , `shutdown.sh`
- conf
 - catalina.policy: Tomcat安全策略文件,控制JVM相关权限,具体可以参考java. security.Permission
 - catalina.properties : Tomcat Catalina行为控制配置文件,比如Common ClassLoader
 - logging.properties : Tomcat日志配置文件, JDK Logging
 - `server.xml` : Tomcat Server配置文件
 - GlobalNamingResources :全局JNDI资源
 - context.xml :全局Context配置文件
 - tomcat-users.xml : Tomcat角色配置文件
 - web.xml : Servlet标准的web.xml部署文件, Tomcat默认实现部分配置入内:
 - org.apache.catalina.servlets.DefaultServlet
 - org.apache.jasper.servlet.JspServlet
- lib
公共类库
- logs
tomcat在运行过程中产生的日志文件
- webapps
用来存放应用程序，当tomcat启动时会去加载webapps目录下的应用程序
- work
用来存放tomcat在运行时的编译后文件，例如JSP编译后的文件

1.3 web应用部署的三种方式

- 拷贝到webapps目录下

```
1 //指定appBase
2 <Host name="localhost" appBase="webapps"
```

- server.xml 的Context标签下配置Context

```
1 <Context docBase="D:\mvc" path="/mvc" reloadable="true" />
```

path:指定访问该Web应用的URL入口 (context-path)

docBase:指定Web应用的文件路径, 可以给定绝对路径, 也可以给定相对于<Host>的appBase属性的相对路径。

reloadable:如果这个属性设为true, tomcat服务器在运行状态下会监视在WEB-INF/classes和WEB-INF/lib目录下class文件的改动, 如果监测到有class文件被更新的, 服务器会自动重新加载Web应用。

- 在\$CATALINA_BASE/conf/[enginename]/[hostname]/ 目录下 (默认conf/Catalina/localhost) 创建xml文件, 文件名就是contextPath。
比如创建mvc.xml, path就是/mvc

```
1 <Context docBase="D:\mvc" reloadable="true" />
```

注意: 想要根路径访问, 文件名为ROOT.xml

2. Tomcat整体架构分析

Tomcat 要实现 2 个核心功能:

- 处理 Socket 连接, 负责网络字节流与 Request 和 Response 对象的转化。
- 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。

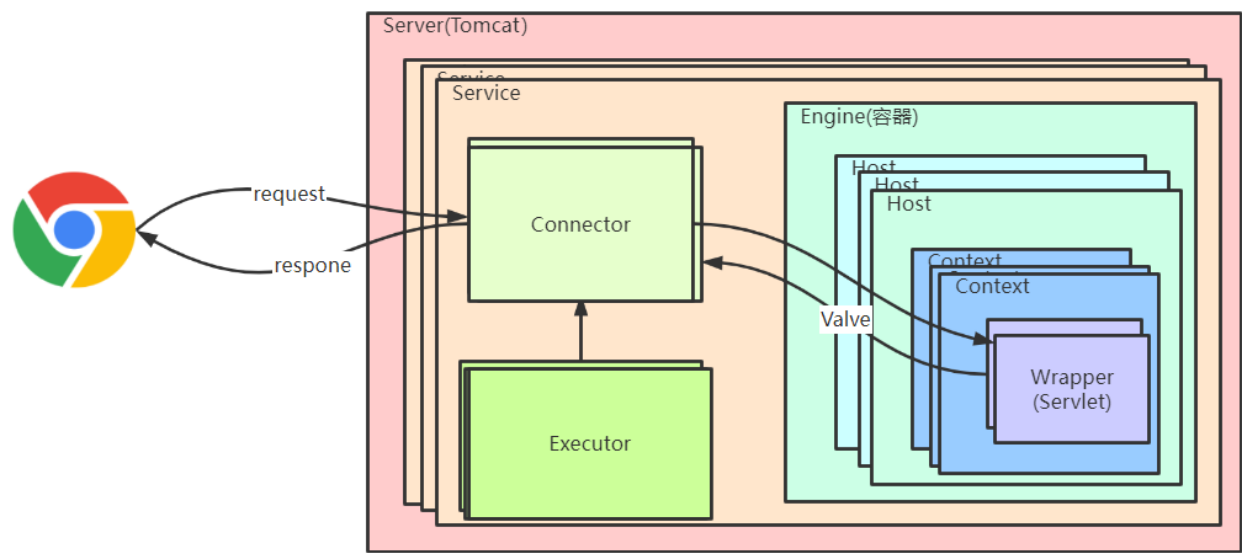
2.1 Tomcat架构图

Tomcat的架构分为以下几个部分:

1. Connector: Tomcat的连接器, 用于接收请求并将其发送给容器。
2. Container: Tomcat的容器, 负责管理Servlet、JSP和静态资源的生命周期。
3. Engine: Tomcat的引擎, 管理容器的生命周期和分配请求。
4. Host: Tomcat的主机, 可以管理多个Web应用程序。
5. Context: Tomcat的上下文, 用于管理单个Web应用程序的配置信息。
6. Servlet: Tomcat的Servlet, 负责处理请求并生成响应。

7. JSP: Tomcat的JSP，用于动态生成Web内容。

总的来说，Tomcat的架构是一个分层的架构，每一层都有其自己的功能和职责。该架构可以提高Tomcat的性能和可维护性，并使得Tomcat可以支持大量的Java Web应用程序。



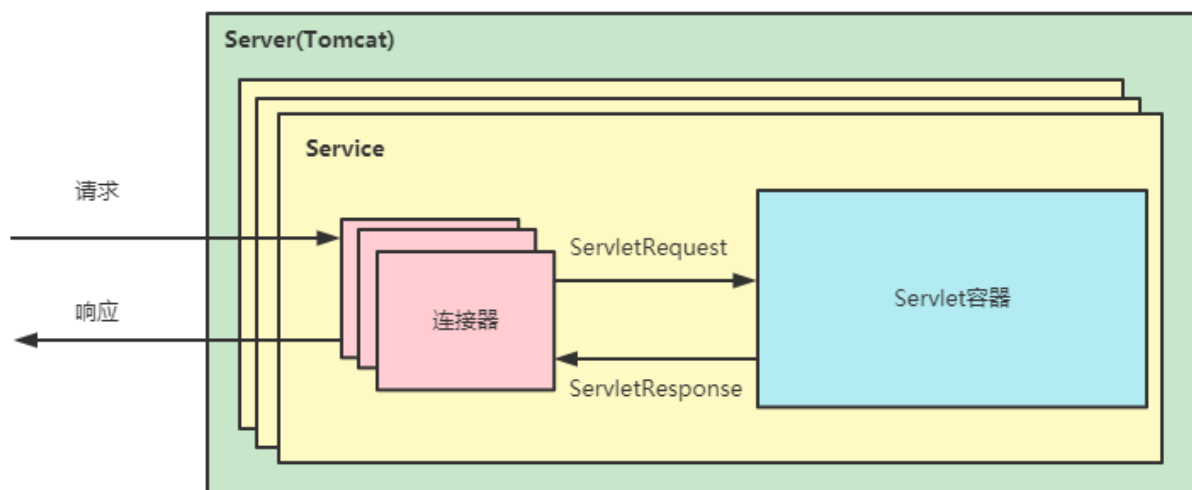
2.2 Tomcat核心组件详解

Server 组件

指的就是整个 Tomcat 服务器，包含多组服务（Service），负责管理和启动各个Service，同时监听 8005 端口发过来的 shutdown 命令，用于关闭整个容器。

Service组件

每个 Service 组件都包含了若干用于接收客户端消息的 Connector 组件和处理请求的 Engine 组件。Service 组件还包含了若干 Executor 组件，每个 Executor 都是一个线程池，它可以为 Service 内所有组件提供线程池执行任务。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。



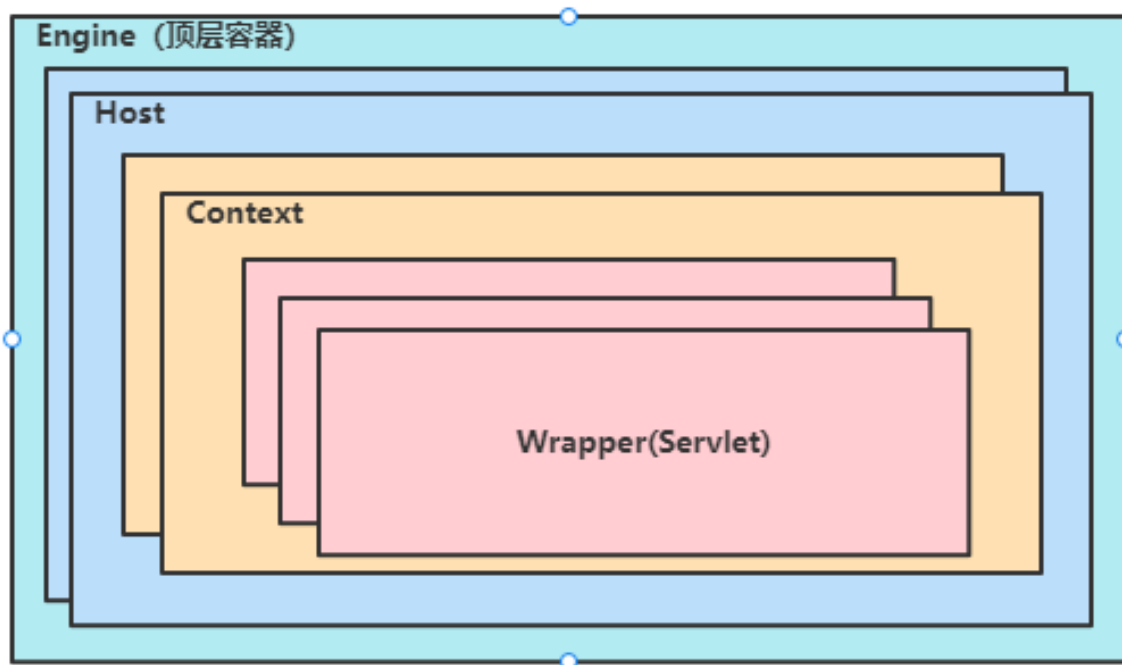
连接器Connector组件

Tomcat 与外部世界的连接器，监听固定端口接收外部请求，传递给 Container，并将Container处理的结果返回给外部。连接器对 Servlet 容器屏蔽了不同的应用层协议及 I/O 模型，无论是 HTTP 还是 AJP，在容器中获取到的都是一个标准的 ServletRequest 对象。

容器Container组件

容器，顾名思义就是用来装载东西的器具，在 Tomcat 里，容器就是用来装载 Servlet 的。Tomcat 通过一种分层的架构，使得 Servlet 容器具有很好的灵活性。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。

- Engine：引擎，Servlet 的顶层容器，用来管理多个虚拟站点，一个 Service 最多只能有一个 Engine；
- Host：虚拟主机，负责 web 应用的部署和 Context 的创建。可以给 Tomcat 配置多个虚拟主机地址，而一个虚拟主机下可以部署多个 Web 应用程序；
- Context：Web 应用上下文，包含多个 Wrapper，负责 web 配置的解析、管理所有的 Web 资源。一个 Context 对应一个 Web 应用程序。
- Wrapper：表示一个 Servlet，最底层的容器，是对 Servlet 的封装，负责 Servlet 实例的创建、执行和销毁。



2.3 结合Server.xml理解Tomcat架构

我们可以通过 Tomcat 的 server.xml 配置文件来加深对 Tomcat 架构的理解。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。

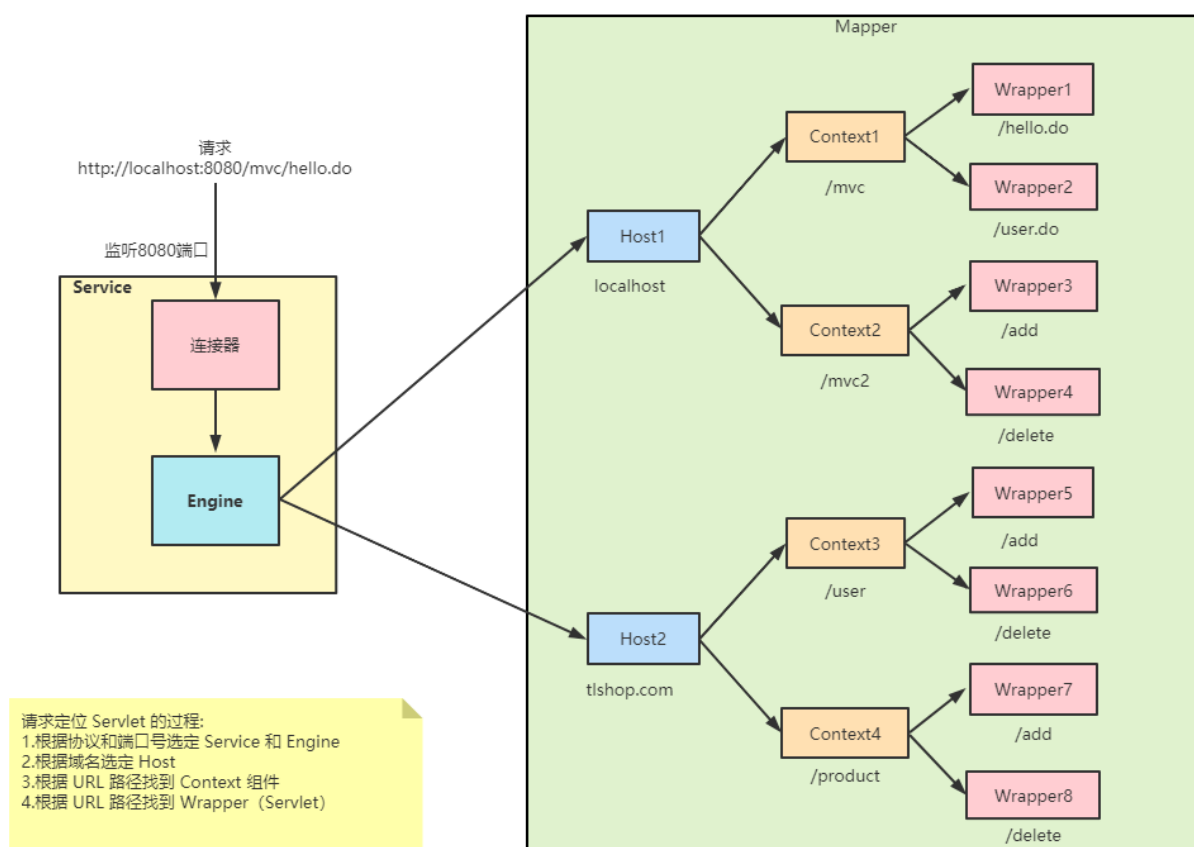
```
1 <Server>    //顶层组件，可以包括多个Service
2     <Service> //顶层组件，可包含一个Engine，多个连接器
3         <Connector/> //连接器组件，代表通信接口
4         <Engine> //容器组件，一个Engine组件处理Service中的所有请求，包含多个Host
5             <Host> //容器组件，处理特定的Host下客户请求，可包含多个Context
6                 <Context/> //容器组件，为特定的Web应用处理所有的客户请求
7             </Host>
8         </Engine>
9     </Service>
10 </Server>
```

Tomcat启动期间会通过解析 server.xml，利用反射创建相应的组件，所以xml中的标签和源码一一对应。

思考：Tomcat 是怎么确定请求是由哪个 Wrapper 容器里的 Servlet 来处理的呢？

2.4 请求定位 Servlet 的过程

Tomcat 是用 Mapper 组件来完成这个任务的。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 里去寻找，就能定位到一个 Servlet。一个请求 URL 最后只会定位到一个 Wrapper 容器，也就是一个 Servlet。



3. Tomcat 架构设计精髓分析

3.1 Connector 高内聚低耦合设计

优秀的模块化设计应该考虑高内聚、低耦合：

- 高内聚是指相关度比较高的功能要尽可能集中，不要分散。
- 低耦合是指两个相关的模块要尽可能减少依赖的部分和降低依赖的程度，不要让两个模块产生强依赖。

Tomcat 连接器需要实现的功能：

- 监听网络端口。
- 接受网络连接请求。
- 读取请求网络字节流。
- 根据具体应用层协议（HTTP/AJP）解析字节流，生成统一的 Tomcat Request 对象。

- 将 Tomcat Request 对象转成标准的 ServletRequest。
- 调用 Servlet 容器，得到 ServletResponse。
- 将 ServletResponse 转成 Tomcat Response 对象。
- 将 Tomcat Response 转成网络字节流。
- 将响应字节流写回给浏览器。

分析连接器详细功能列表，我们会发现连接器需要完成 3 个高内聚的功能：

- 网络通信。
- 应用层协议解析。
- Tomcat Request/Response 与 ServletRequest/ServletResponse 的转化。

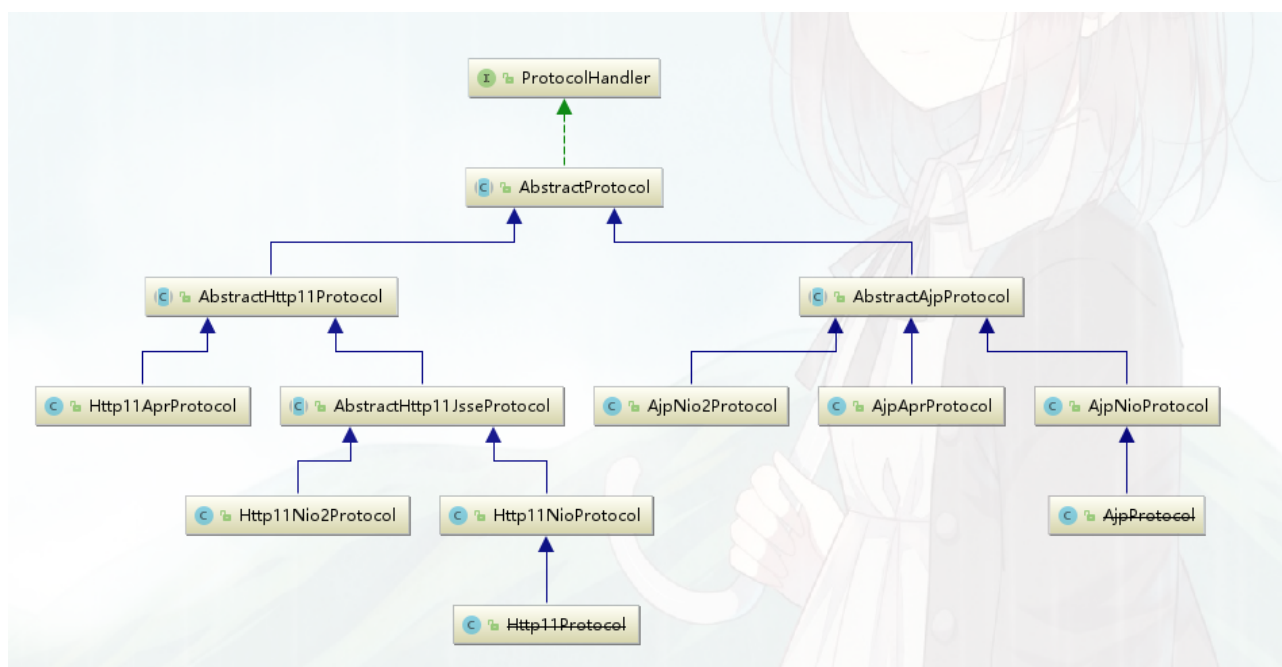
因此 Tomcat 的设计者设计了 3 个组件来实现这 3 个功能，分别是 **EndPoint**、**Processor** 和 **Adapter**。

- EndPoint 负责提供字节流给 Processor；
- Processor 负责提供 Tomcat Request 对象给 Adapter；
- Adapter 负责提供 ServletRequest 对象给容器。

组件之间通过抽象接口交互。这样做的好处是封装变化。这是面向对象设计的精髓，将系统中经常变化的部分和稳定的部分隔离，有助于增加复用性，并降低系统耦合度。

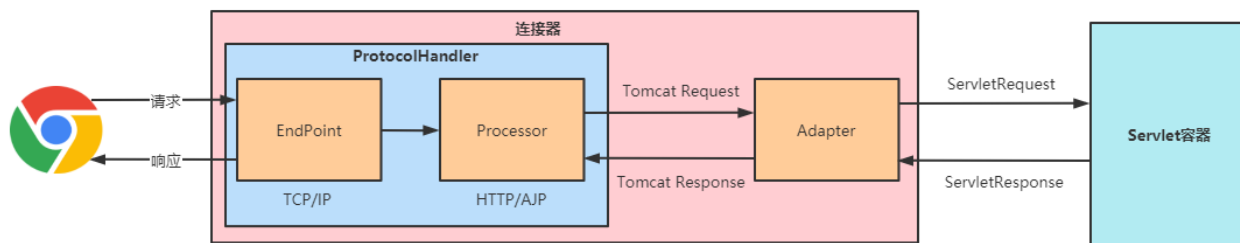
由于 I/O 模型和应用层协议可以自由组合，比如 NIO + HTTP 或者 NIO2 + AJP。Tomcat 的设计者将网络通信和应用层协议解析放在一起考虑，设计了一个叫 **ProtocolHandler** 的接口来封装这两种变化点。各种协议和通信模型的组合有相应的具体实现类。比如：Http11NioProtocol 和 AjpNioProtocol。

除了这些变化点，系统也存在一些相对稳定的部分，因此 **Tomcat 设计了一系列抽象基类来封装这些稳定的部分**，抽象基类 AbstractProtocol 实现了 ProtocolHandler 接口。每一种应用层协议有自己的抽象基类，比如 AbstractAjpProtocol 和 AbstractHttp11Protocol，具体协议的实现类扩展了协议层抽象基类。



ProtocolHandler

连接器用 ProtocolHandler 来处理网络连接和应用层协议，包含了 2 个重要部件：EndPoint 和 Processor。



连接器用 ProtocolHandler 接口来封装通信协议和 I/O 模型的差异，ProtocolHandler 内部又分为 EndPoint 和 Processor 模块，EndPoint 负责底层 Socket 通信，Processor 负责应用层协议解析。连接器通过适配器 Adapter 调用容器。

EndPoint

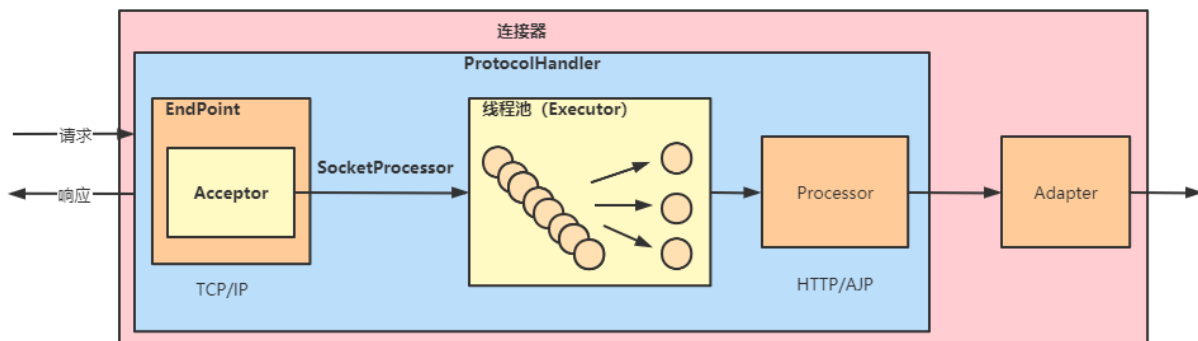
EndPoint 是通信端点，即通信监听的接口，是具体的 Socket 接收和发送处理器，是对传输层的抽象，因此 EndPoint 是用来实现 TCP/IP 协议的。

EndPoint 是一个接口，对应的抽象实现类是 AbstractEndpoint，而 AbstractEndpoint 的具体子类，比如在 NioEndpoint 和 Nio2Endpoint 中，有两个重要的子组件：Acceptor 和 SocketProcessor。其中 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)。

Processor

Processor 用来实现 HTTP/AJP 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。

Processor 是一个接口，定义了请求的处理等方法。它的抽象实现类 AbstractProcessor 对一些协议共有的属性进行封装，没有对方法进行实现。具体的实现有 AJPProcessor、HTTP11Processor 等，这些具体实现类实现了特定协议的解析方法和请求处理方式。



EndPoint 接收到 Socket 连接后，生成一个 SocketProcessor 任务提交到线程池去处理，SocketProcessor 的 Run 方法会调用 Processor 组件去解析应用层协议，Processor 通过解析生成 Request 对象后，会调用 Adapter 的 Service 方法。

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 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 Service 方法，传入的是 Tomcat Request 对象，CoyoteAdapter 负责将 Tomcat Request 转成 ServletRequest，再调用容器的 Service 方法。

设计复杂系统的基本思路：

首先要分析需求，根据高内聚低耦合的原则确定子模块，然后找出子模块中的变化点和不变点，用接口和抽象基类去封装不变点，在抽象基类中定义模板方法，让子类自行实现抽象方法，也就是具体子类去实现变化点。

3.2 父子容器组合模式设计

思考：Tomcat 设计了 4 种容器，分别是 Engine、Host、Context 和 Wrapper，Tomcat 是怎么管理这些容器的？

Tomcat 采用组合模式来管理这些容器。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。

Container 接口定义如下：

```
1 public interface Container extends Lifecycle {  
2     public void setName(String name);  
     public Container getParent();  
}
```

```
4     public void setParent(Container container);
5     public void addChild(Container child);
6     public void removeChild(Container child);
7     public Container findChild(String name);
8 }
```

3.3 Pipeline-Valve 责任链模式设计

连接器中的 Adapter 会调用容器的 Service 方法来执行 Servlet，最先拿到请求的是 Engine 容器，Engine 容器对请求做一些处理后，会把请求传给自己子容器 Host 继续处理，依次类推，最后这个请求会传给 Wrapper 容器，Wrapper 会调用最终的 Servlet 来处理。那么这个调用过程具体是怎么实现的呢？答案是使用 Pipeline-Valve 管道。

Pipeline-Valve 是责任链模式，责任链模式是指在一个请求处理的过程中有很多处理者依次对请求进行处理，每个处理者负责做自己相应的处理，处理完之后将再调用下一个处理者继续处理。

为什么要使用管道机制？

在一个比较复杂的大型系统中，如果一个对象或数据流需要进行繁杂的逻辑处理，我们可以选择在一个大的组件中直接处理这些繁杂的业务逻辑，这个方式虽然达到目的，但扩展性和可重用性较差，因为可能牵一发而动全身。更好的解决方案是采用管道机制，用一条管道把多个对象(阀门部件)连接起来，整体看起来就像若干个阀门嵌套在管道中一样，而处理逻辑放在阀门上。

Valve接口设计

理解它的设计，第一步就是阀门设计。Valve 表示一个处理点，比如权限认证和记录日志。

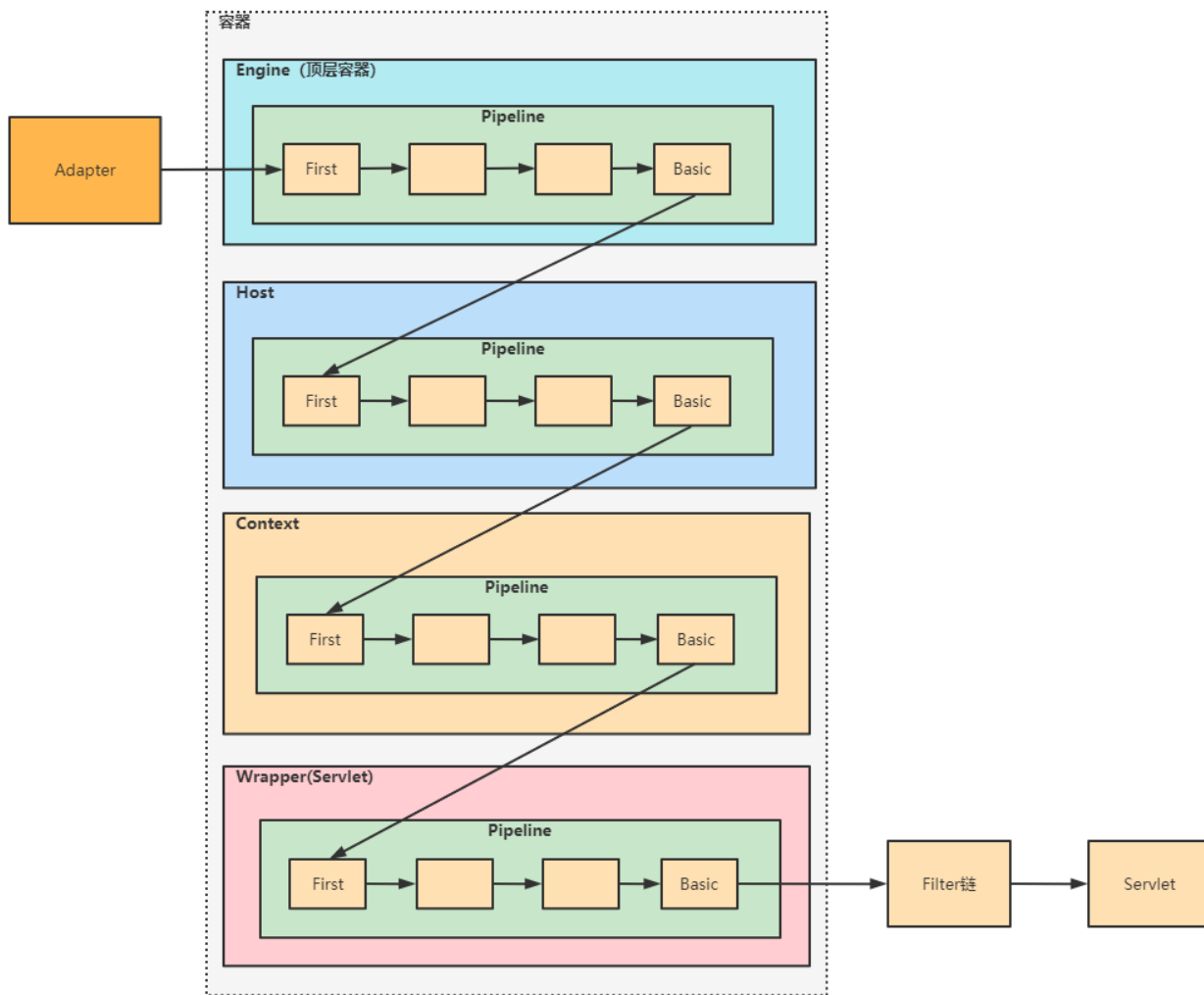
```
1 public interface Valve {
2     public Valve getNext();
3     public void setNext(Valve valve);
4     public void invoke(Request request, Response response) throws IOException,
5         ServletException;
6 }
```

Pipeline接口设计

由于Pipeline是为容器设计的，所以它在设计时加入了一个Contained接口, 就是为了制定当前Pipeline所属的容器

```
1  public interface Pipeline extends Contained {
2
3      // 基础的处理阀
4      public Valve getBasic();
5      public void setBasic(Valve valve);
6
7
8      // 对节点（阀门）增删查
9      public void addValve(Valve valve);
10     public Valve[] getValves();
11     public void removeValve(Valve valve);
12
13
14     // 获取第一个节点，遍历的起点，所以需要有这方法
15     public Valve getFirst();
16
17
18     // 是否所有节点（阀门）都支持处理Servlet3异步处理
19     public boolean isAsyncSupported();
20
21
22     // 找到所有不支持Servlet3异步处理的阀门
23     public void findNonAsyncValves(Set<String> result);
24 }
```

Pipeline 中维护了 Valve 链表，Valve 可以插入到 Pipeline 中，对请求做某些处理。整个调用链的触发是 Valve 来完成的，Valve 完成自己的处理后，调用 getNext.invoke() 来触发下一个 Valve 调用。**每一个容器都有一个 Pipeline 对象**，只要触发这个 Pipeline 的第一个 Valve，这个容器里 Pipeline 中的 Valve 就都会被调用到。Basic Valve 处于 Valve 链表的末端，它是 Pipeline 中必不可少的一个 Valve，负责调用下层容器的 Pipeline 里的第一个 Valve。



整个调用过程由连接器中的 Adapter 触发的，它会调用 Engine 的第一个 Valve：

```
1 //org.apache.catalina.connector.CoyoteAdapter#service
2 // Calling the container
3 connector.getService().getContainer().getPipeline().getFirst().invoke(request,
  response);
```

Wrapper 容器的最后一个 Valve 会创建一个 Filter 链，并调用 doFilter() 方法，最终会调到 Servlet 的 service 方法。

```
1 //org.apache.catalina.core.StandardWrapperValve#invoke
2 filterChain.doFilter(request.getRequest(), response.getResponse());
```

Valve 和 Filter 的区别：

- Valve 是 Tomcat 的私有机制，与 Tomcat 的基础架构 /API 是紧耦合的。Servlet API 是公有的标准，所有的 Web 容器包括 Jetty 都支持 Filter 机制。
- Valve 工作在 Web 容器级别，拦截所有应用的请求；而 Servlet Filter 工作在应用级别，只能拦截某个 Web 应用的所有请求。

对比两种责任链模式

管道/阀门	过滤器链/过滤器
管道 (Pipeline)	过滤器链 (FilterChain)
阀门 (Valve)	过滤器 (Filter)
底层实现为具有头 (first) 、尾 (basic) 指针的单向链表	底层实现为数组
Valve的核心方法invoke(request,response)	Filter核心方法doFilter(request,response,chain)
pipeline.getFirst().invoke(request,response)	filterchain.doFilter(request,response)

3.4 Tomcat生命周期设计

通过对Tomcat架构的分析，我们知道了Tomcat 都有哪些组件，组件之间是什么样的关系，以及Tomcat 是怎么处理一个 HTTP 请求的。如果能让Tomcat能够对外提供服务，我们需要创建、组装并启动Tomcat组件；在服务停止的时候，我们还需要释放资源，销毁Tomcat组件，这是一个动态的过程。**Tomcat 需要动态地管理这些组件的生命周期。**

在我们实际的工作中，如果你需要设计一个比较大的系统或者框架时，你同样也需要考虑这几个问题：**如何统一管理组件的创建、初始化、启动、停止和销毁？如何做到代码逻辑清晰？如何方便地添加或者删除组件？如何做到组件启动和停止不遗漏、不重复？**

一键式启停：LifeCycle 接口

系统设计就是要找到系统的变化点和不变点。这里的不变点就是每个组件都要经历创建、初始化、启动这几个过程，这些状态以及状态的转化是不变的。而变化点是每个具体组件的初始化方法，也就是启动方法是不一样的。因此，**我们把不变点抽象出来成为一个接口，这个接口跟生命周期有关，叫作 LifeCycle。**LifeCycle 接口里应该定义这么几个方法：init()、start()、stop() 和 destroy()，每个具体的组件去实现这些方法。

```
1 public interface Lifecycle {
2     /** 第1类：针对监听器 */
3     // 添加监听器
4     public void addLifecycleListener(LifecycleListener listener);
5     // 获取所有监听器
6     public LifecycleListener[] findLifecycleListeners();
7     // 移除某个监听器
```

```

8     public void removeLifecycleListener(LifecycleListener listener);
9
10    /** 第2类：针对控制流程 */
11    // 初始化方法
12    public void init() throws LifecycleException;
13    // 启动方法
14    public void start() throws LifecycleException;
15    // 停止方法，和start对应
16    public void stop() throws LifecycleException;
17    // 销毁方法，和init对应
18    public void destroy() throws LifecycleException;
19
20    /** 第3类：针对状态 */
21    // 获取生命周期状态
22    public LifecycleState getState();
23    // 获取字符串类型的生命周期状态
24    public String getStateName();
25 }

```

在父组件的 init() 方法里需要创建子组件并调用子组件的 init() 方法。同样，在父组件的 start() 方法里也需要调用子组件的 start() 方法，因此调用者可以无差别的调用各组件的 init() 方法和 start() 方法，这就是**组合模式**的使用，并且只要调用最顶层组件，也就是 Server 组件的 init() 和 start() 方法，整个 Tomcat 就被启动起来了。

可扩展性：LifeCycle 事件

因为各个组件 init() 和 start() 方法的具体实现是复杂多变的，比如在 Host 容器的启动方法里需要扫描 webapps 目录下的 Web 应用，创建相应的 Context 容器，如果将来需要增加新的逻辑，直接修改 start() 方法？这样会违反开闭原则，那如何解决这个问题呢？开闭原则说的是为了扩展系统的功能，你不能直接修改系统中已有的类，但是你可以定义新的类。

组件的 init() 和 start() 调用是由它的父组件的状态变化触发的，上层组件的初始化会触发子组件的初始化，上层组件的启动会触发子组件的启动，因此我们把组件的生命周期定义成一个个状态，把状态的转变看作是一个事件。而事件是有监听器的，在监听器里可以实现一些逻辑，并且监听器也可以方便的添加和删除，这就是典型的**观察者模式**。

具体来说就是在 LifeCycle 接口里加入两个方法：添加监听器和删除监听器。

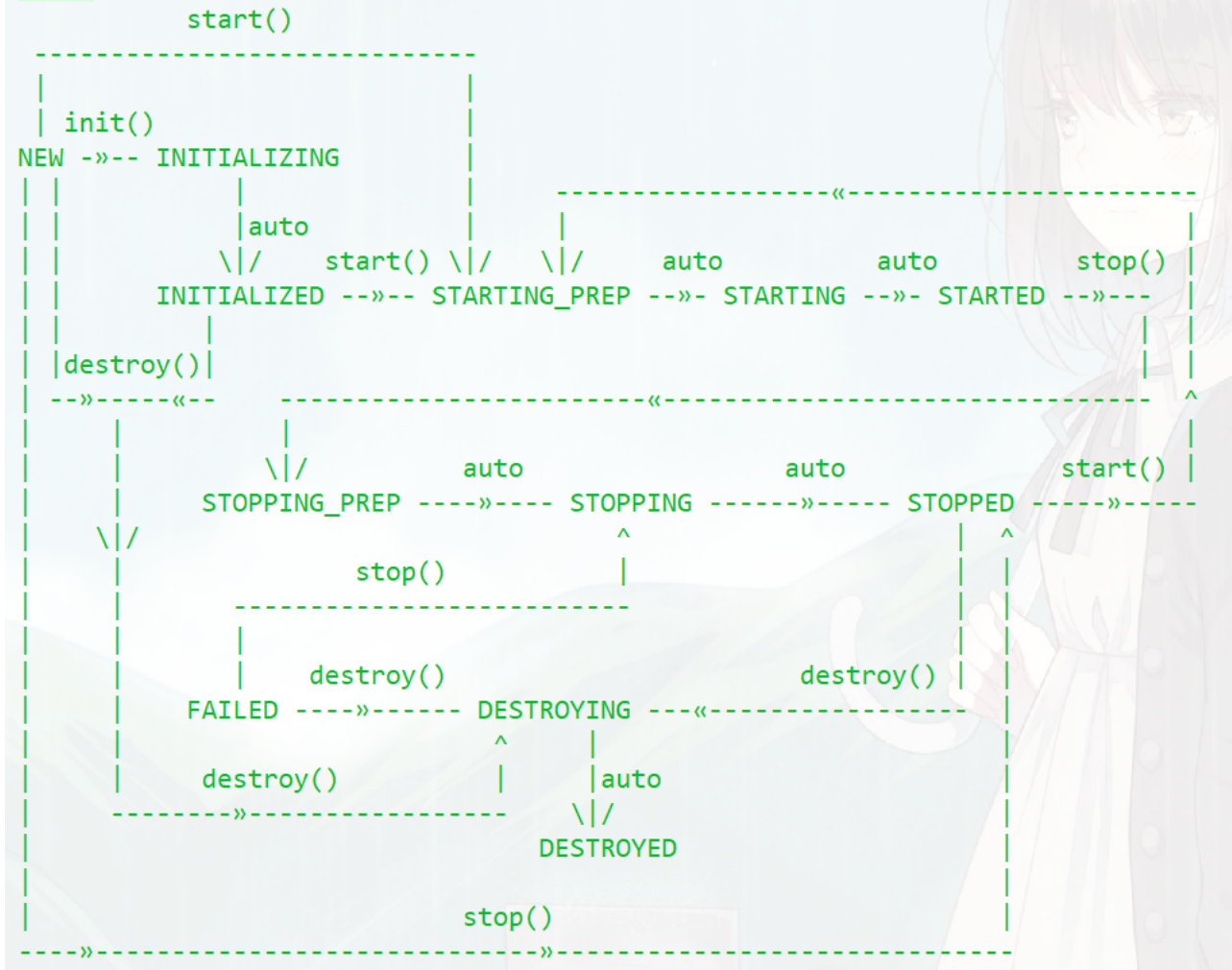
I Lifecycle

```
(m) addLifecycleListener(LifecycleListener)    void
(m) findLifecycleListeners()                    LifecycleListener[]
(m) removeLifecycleListener(LifecycleListener)  void
```

我们还需要定义一个 Enum 来表示组件有哪些状态，以及处在什么状态会触发什么样的事件。

```
public enum LifecycleState {
    // 组件刚刚创建时，即在LifecycleBase实例构造完成时的状态。
    NEW(false, null),
    // 组件初始化过程中的状态。
    INITIALIZING(false, Lifecycle.BEFORE_INIT_EVENT),
    // 组件初始化完成时的状态。
    INITIALIZED(false, Lifecycle.AFTER_INIT_EVENT),
    // 组件启动前的状态。
    STARTING_PREP(false, Lifecycle.BEFORE_START_EVENT),
    // 组件启动过程中的状态。
    STARTING(true, Lifecycle.START_EVENT),
    // 组件启动完成的状态。
    STARTED(true, Lifecycle.AFTER_START_EVENT),
    // 组件停止前的状态。
    STOPPING_PREP(true, Lifecycle.BEFORE_STOP_EVENT),
    // 组件停止过程中的状态。
    STOPPING(false, Lifecycle.STOP_EVENT),
    // 组件停止完成的状态。
    STOPPED(false, Lifecycle.AFTER_STOP_EVENT),
    // 组件销毁过程中的状态。
    DESTROYING(false, Lifecycle.BEFORE_DESTROY_EVENT),
    // 组件销毁后的状态。
    DESTROYED(false, Lifecycle.AFTER_DESTROY_EVENT),
    // 组件启动、停止过程中出现异常的状态。
    FAILED(false, null);
}
```

组件生命周期状态变化如下：

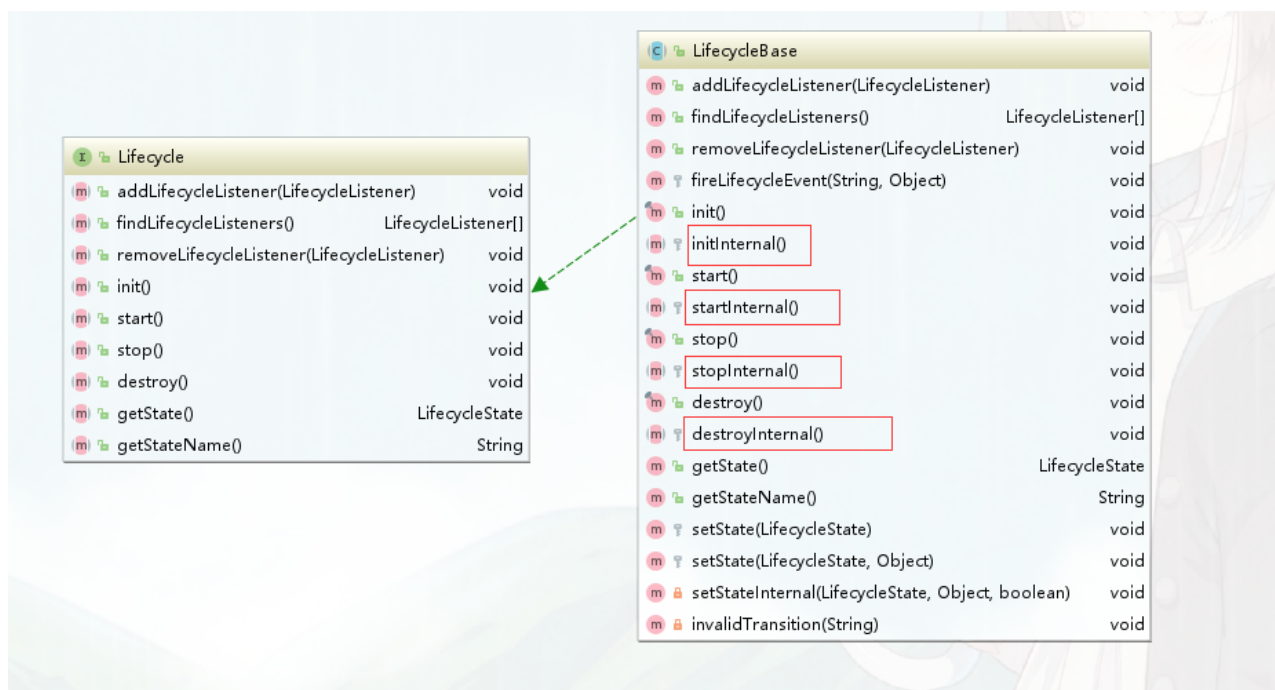


重用性：LifeCycleBase 抽象基类

有了接口，我们就要用类去实现接口。一般来说实现类不止一个，不同的类在实现接口时往往会有一些相同的逻辑，如果让各个子类都去实现一遍，就会有重复代码。那子类如何重用这部分逻辑呢？其实就是定义一个基类来实现共同的逻辑，然后让各个子类去继承它，就达到了重用的目的。而基类中往往会定义一些抽象方法，所谓的抽象方法就是说基类不会去实现这些方法，而是调用这些方法来

实现骨架逻辑。抽象方法是留给各个子类去实现的，并且子类必须实现，否则无法实例化。

Tomcat 定义一个基类 `LifeCycleBase` 来实现 `LifeCycle` 接口，把一些公共的逻辑放到基类中去，比如生命状态的转变与维护、生命周期事件的触发以及监听器的添加和删除等，而子类就负责实现自己的初始化、启动和停止等方法。为了避免跟基类中的方法同名，我们把具体子类的实现方法改个名字，在后面加上 `Internal`，叫 `initInternal()`、`startInternal()` 等。



`LifecycleBase` 实现了 `Lifecycle` 接口中所有的方法，还定义了相应的抽象方法交给具体子类去实现，这是典型的模板设计模式（骨架抽象类和模板方法）。

`LifecycleBase` 的 `init()` 方法实现：

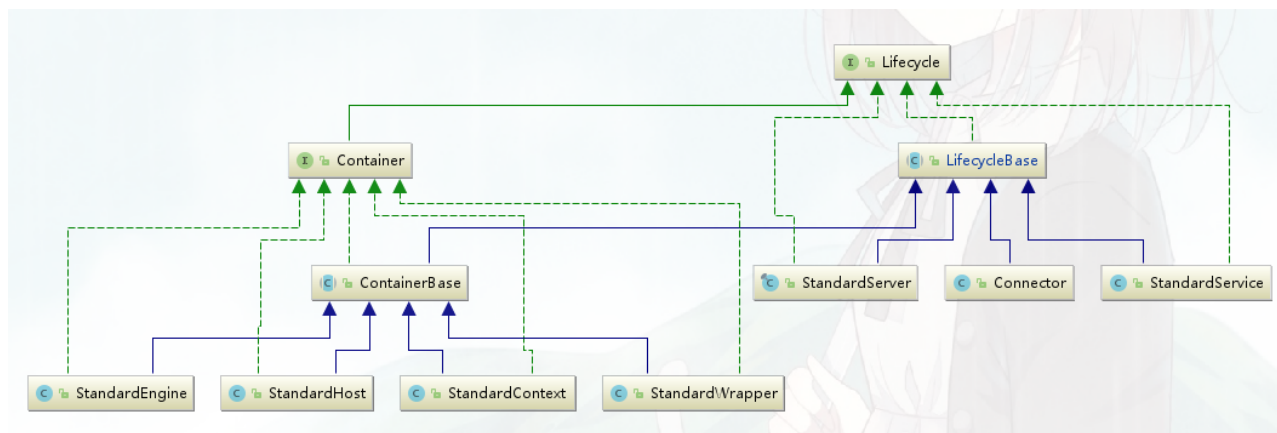
```
@Override
public final synchronized void init() throws LifecycleException {
    // 状态检查
    if (!state.equals(LifecycleState.NEW)) {
        invalidTransition(Lifecycle.BEFORE_INIT_EVENT);
    }

    try {
        // 触发INITIALIZING事件的监听器
        setStateInternal(LifecycleState.INITIALIZING, data: null, check: false);
        // 调用具体子类的初始化方法
        initInternal();
        // 触发INITIALIZED事件的监听器
        setStateInternal(LifecycleState.INITIALIZED, data: null, check: false);
    } catch (Throwable t) {
        ExceptionUtils.handleThrowable(t);
        setStateInternal(LifecycleState.FAILED, data: null, check: false);
        throw new LifecycleException(
            sm.getString(key: "lifecycleBase.initFail", toString()), t);
    }
}
```

思考：是什么时候、谁把监听器注册进来的呢？

- Tomcat 自定义了一些监听器，这些监听器是父组件在创建子组件的过程中注册到子组件的。比如 `MemoryLeakTrackingListener` 监听器，用来检测 Context 容器中的内存泄漏，这个监听器是 Host 容器在创建 Context 容器时注册到 Context 中的。
- 我们还可以在 `server.xml` 中定义自己的监听器，Tomcat 在启动时会解析 `server.xml`，创建监听器并注册到容器组件。

生命周期总体类图



StandardServer、StandardService 是 Server 和 Service 组件的具体实现类，它们都继承了 LifecycleBase。StandardEngine、StandardHost、StandardContext 和 StandardWrapper 是相应容器组件的具体实现类，因为它们都是容器，所以继承了 ContainerBase 抽象基类，而 ContainerBase 实现了 Container 接口，也继承了 LifecycleBase 类，它们的生命周期管理接口和功能接口是分开的，这也符合设计中**接口分离的原则**。

Tomcat 为了实现一键式启停以及优雅的生命周期管理，并考虑到了可扩展性和可重用性，将面向对象思想和设计模式发挥到了极致，分别运用了**组合模式、观察者模式、骨架抽象类和模板方法**。如果你需要维护一堆具有父子关系的实体，可以考虑使用**组合模式**。观察者模式听起来“高大上”，其实就是当一个事件发生后，需要执行一连串更新操作。传统的实现方式是在事件响应代码里直接加更新逻辑，当更新逻辑加多了之后，代码会变得臃肿，并且这种方式是紧耦合的、侵入式的。**观察者模式实现了低耦合、非侵入式的通知与更新机制**。模板方法在抽象基类中经常用到，用来实现通用逻辑。