tomcat 学习 |tomcat 中组件结构设计

作者：xiaodaojava

原文链接：https://ld246.com/article/1565622311320

来源网站：链滴

许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

开头说两句

小刀博客: https://www.lixiang.red

小刀公众号: 程序员学习大本营

学习背景

在前面几篇文章,我们一起学习了tomcat中的server.xml，类加载器,组件默认值,digester解析server.xml并初步初始化等基础知识点

https://www.lixiang.red/articles/2019/08/11/1565515601658.html

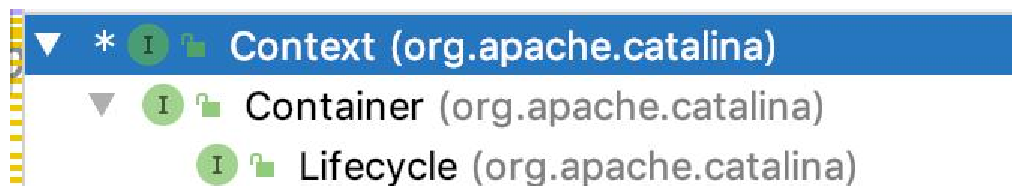
下面我们就要真正的走进源码,去看一看这些组件是如何实现的,今天我们一起学习tomcat中组件的设计

源码中的这些组件

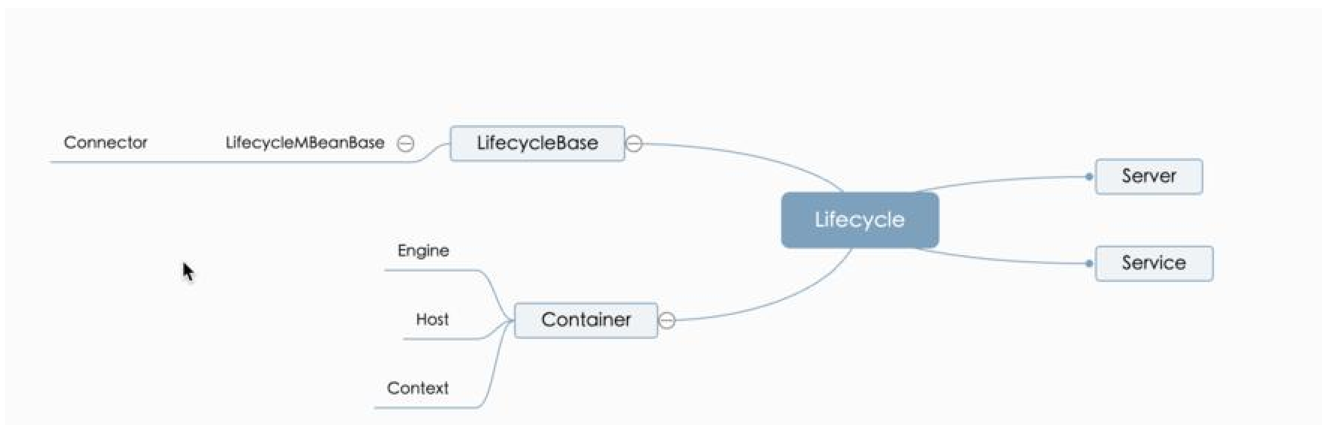
通过下图我们可以看到,在我们直接使用的Context,Service,Server上面还有一层接口: Container 和 Lifcycle

我们说接口是有什么什么能力,现在我们就整理下这些组件的接口

以内层的Context为例,我们在idea中可以看到如下继承图:



同样,我们去对比其他的组件,也会发现他们都类似于继承这些接口,几大常用组件的继承关系如下所示:



我们可以通过idea的类图工具,查看类的继承以集接口相关的方法,对着类名点右键,然后可以看到相关图

```
Container extends Lifecycle {
Translate                        ^⌘U
Translate and Replace...         ^⌘O
Show Context Actions             ⌥⏎
Copy Reference                   ⌥⇧⌘C
Paste                            ⌘V
Paste from History...            ⇧⌘V
Paste without Formatting         ⌥⇧⌘V
Column Selection Mode            ⇧⌘8
Find Usages                      ⌥F7
Refactor
Folding
Analyze
Go To
Generate...                      ⌘N
Reveal in Finder
Open in Terminal
Local History
Git
Compare with Clipboard
Diagrams                         Show Diagram...        ⌥⇧⌘U
                                 Show Diagram Popup...  ⌥⌘U
编码规约扫描                     ⌥⇧⌘J
关闭实时检测功能
Open on GitHub
Create Gist...
Open on GitLab
```

展示变量

展示方法

Lifecycle

addLifecycleListener(LifecycleListener)	void
findLifecycleListeners()	LifecycleListener[]
removeLifecycleListener(LifecycleListener)	void
init()	void
start()	void
stop()	void
destroy()	void
getState()	LifecycleState
getStateName()	String

Container

getLogger()	Log
getLogName()	String
getObjectName()	ObjectName
getDomain()	String
getMBeanKeyProperties()	String
getPipeline()	Pipeline
getCluster()	Cluster
setCluster(Cluster)	void
getBackgroundProcessorDelay()	int
setBackgroundProcessorDelay(int)	void
getName()	String
setName(String)	void
getParent()	Container
setParent(Container)	void
getParentClassLoader()	ClassLoader
setParentClassLoader(ClassLoader)	void
getRealm()	Realm
setRealm(Realm)	void
backgroundProcess()	void
addChild(Container)	void
addContainerListener(ContainerListener)	void
addPropertyChangeListener(PropertyChangeListener)	void
findChild(String)	Container

Lifecycle接口

我们可以对上图中的接口做一个划分,如下所示

Lifecycle	
addLifecycleListener(LifecycleListener)	void
findLifecycleListeners()	LifecycleListener[]
removeLifecycleListener(LifecycleListener)	void
init()	void
start()	void
stop()	void
destroy()	void
getState()	LifecycleState
getStateName()	String

上面三个方法是 Listener相关的

中间三个方法是自身生命周期相关的

下面两个方法是获取自身状态的

listener

监听器,每一组件,有一组监听器,在组件本身达到某一状态时,可以循环监听器list，然后这些注册监听器组件,再根据监听到的状态进行相关的动作行为.

我们以server为例:

Container × Lifecycle.java × LifecycleBase.java × StandardServer.java ×

```
758     */
759     @Override
760     protected void startInternal() throws LifecycleException {
761
762         fireLifecycleEvent(CONFIGURE_START_EVENT, data: null);
763         setState(LifecycleState.STARTING);
764
765         globalNamingResources.start();
766
767         // Start our defined Services
768         synchronized (servicesLock) {
769             for (int i = 0; i < services.length; i++) {
770                 services[i].start();
```

/**
  * 循环监听器List，去执行不同的动作

原文链接：tomcat 学习 |tomcat 中组件结构设计

```
 *
 * @param type  Event type
 * @param data  Data associated with event.
 */
protected void fireLifecycleEvent(String type, Object data) {
    LifecycleEvent event = new LifecycleEvent(this, type, data);
    for (LifecycleListener listener : lifecycleListeners) {
        listener.lifecycleEvent(event);
    }
}
```

当执行server的start时,就会去向所有的监听器传播 CONFIGURE_START_EVENT这个事件.但是监听对这个事件做不做响应,就是对应的实际监听器所做的决定,如下图所示,HostConfig ,在接收到事件时,判断类型,对不同类型的事件,做不同的处理. 基本上Config都继承了 LifecycleListener 这个接口

```
 * @param event The lifecycle event that has occurred
 */
@Override
public void lifecycleEvent(LifecycleEvent event) {

    // Identify the host we are associated with
    try {
        host = (Host) event.getLifecycle();
        if (host instanceof StandardHost) {
            setCopyXML(((StandardHost) host).isCopyXML());
            setDeployXML(((StandardHost) host).isDeployXML());
            setUnpackWARs(((StandardHost) host).isUnpackWARs());
            setContextClass(((StandardHost) host).getContextClass());
        }
    } catch (ClassCastException e) {
        log.error(sm.getString( key: "hostConfig.cce", event.getLifecycle()), e);
        return;
    }

    // Process the event that has occurred
    if (event.getType().equals(Lifecycle.PERIODIC_EVENT)) {
        check();
    } else if (event.getType().equals(Lifecycle.BEFORE_START_EVENT)) {
        beforeStart();
    } else if (event.getType().equals(Lifecycle.START_EVENT)) {
        start();
    } else if (event.getType().equals(Lifecycle.STOP_EVENT)) {
        stop();
    }
}
```

组件本身的生命周期

中间四个方法,代表着组件的四种状态:init(),初始化, start() 启动,stop()停止,destory(销毁),这些通过字意思就能猜出来.tomcat官方给了一张生命周期和状态对应的流程图:

原文链接：tomcat 学习 |tomcat 中组件结构设计

```
Lifecycle.java    Container    LifecycleBase.java    HostConfig.java
28   * <pre>
29   *            start()
30   *  -----------------------------
31   *  |                           |
32   *  | init()                    |
33   * NEW -»-- INITIALIZING        |
34   * | |           |              |     ------------------«-----------------------
35   * | |           |auto          |     |                                        |
36   * | |          \|/    start() \|/   \|/     auto          auto         stop() |
37   * | |      INITIALIZED --»-- STARTING_PREP --»- STARTING --»- STARTED --»---  |
38   * | |         |                                                            |  |
39   * | |destroy()|                                                            |  |
40   * | --»-----«--    ------------------------«--------------------------------  ^
41   * |     |          |                                                          |
42   * |     |         \|/          auto                 auto              start() |
43   * |     |     STOPPING_PREP ----»---- STOPPING ------»----- STOPPED -----»-----
44   * |    \|/                               ^                     |  ^
45   * |     |               stop()           |                     |  |
46   * |     |       --------------------------                     |  |
47   * |     |       |                                              |  |
48   * |     |       |    destroy()                       destroy() |  |
49   * |     |    FAILED ----»------ DESTROYING ---«-----------------  |
50   * |     |                        ^     |                          |
51   * |     |     destroy()          |     |auto                      |
52   * |     --------»-----------------    \|/                         |
53   * |                                 DESTROYED                     |
54   * |                                                               |
55   * |                            stop()                             |
56   * ----»-----------------------------»------------------------------
57   *
58   * Any state can transition to FAILED.
59   *
60   * Calling start() while a component is in states STARTING_PREP, STARTING or
61   * STARTED has no effect.
```

重点不在于上面的图,而在于下面话,组件通过调用不同的方法,使自身达到不同的状态,然后调用监听器lis，去通知其他的组件/配置做相应的处理,以Server 这个组件为例:,在前几篇中,我们讲tomcat 启动流程时候,分析到这里, 会调用server.init(),实际上就是调用了生命周期的第一个阶段方法

Catalina.java × | Lifecycle.java × | Container × | LifecycleBase.java

```
639
640        // Start the new server
641        try {
642            getServer().init();
643        } catch (LifecycleException e) {
644            if (Boolean.getBoolean( name
645                throw new java.lang.Errc
646            } else {
647                log.error( message: "Cata
648            }
649        }
650
651        long t2 = System.nanoTime();
```

我们可以看到, init执行的LifecycleBase类中的init方法,最终,是执行的StandardServer中的initInterna方法,

Lifecycle.java | Container | LifecycleBase.java | StandardServer.java | HostConfig.java

Match Case | Words | Regex ? | 5 matches

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

    try {
        setStateInternal(LifecycleState.INITIALIZING, data: null, check: false);
        initInternal();
        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);
    }
}
```

由各个子类去重写

```java
protected abstract void initInternal() throws LifecycleException;
```

```java
// 初始化我们的service
    for (int i = 0; i < services.length; i++) {
        services[i].init();
    }
```

同样,我们可以去看到,start也是一个类似的过程,所以这四个方法, 是贯穿整个tomcat生命周期的,推动tomcat的运行

本身状态

可以通过下图看使用方法:组件.getState().isAvailable(), 去判断组件是否在可用状态,通过LifecycleSta e源码可以看到,只在三种状态下available是可用的

```java
public LifecycleState getState();

/**
 * Obtain a textual repre
 * for JMX. The format of
 * should not be relied u
 * component state, use {
 *
 * @return The name of th
 */
public String getStateNam
```

Some Usages of getState() in All Places **(Only 100 usages shown)**

File	Line	Code
ManagerServlet.java	1008	} else if (deployed!=null && !deployed.getState().isAvailable()) {
ManagerServlet.java	1040	if (context.getState().isAvailable()) {
ManagerServlet.java	1382	if (context.getState().isAvailable())
HTMLHostManagerServlet.java	364	if (host.getState().isAvailable()) {
HostManagerServlet.java	569	if (host.getState().isAvailable()) {
HostManagerServlet.java	631	if (!host.getState().isAvailable()) {
MapperListener.java	111	if (!LifecycleState.NEW.equals(host.getState())) {
MapperListener.java	157	if (child.getState().isAvailable()) {
MapperListener.java	165	if (child.getParent().getState().isAvailable()) {
MapperListener.java	307	if (container.getState().isAvailable()) {
MapperListener.java	480	if (w.getParent().getState().isAvailable()) {
MapperListener.java	487	if (c.getParent().getState().isAvailable()) {
ManagerBase.java	356	if (!getState().equals(LifecycleState.NEW)) {
PersistentManagerBase.java	925	if (!getState().isAvailable() || maxIdleSwap < 0)
PersistentManagerBase.java	967	if (!getState().isAvailable() || minIdleSwap < 0 || getMaxActiveSessions() < 0) {

```java
STARTING(true, Lifecycle.START_EVENT),
```

原文链接：tomcat 学习 |tomcat 中组件结构设计

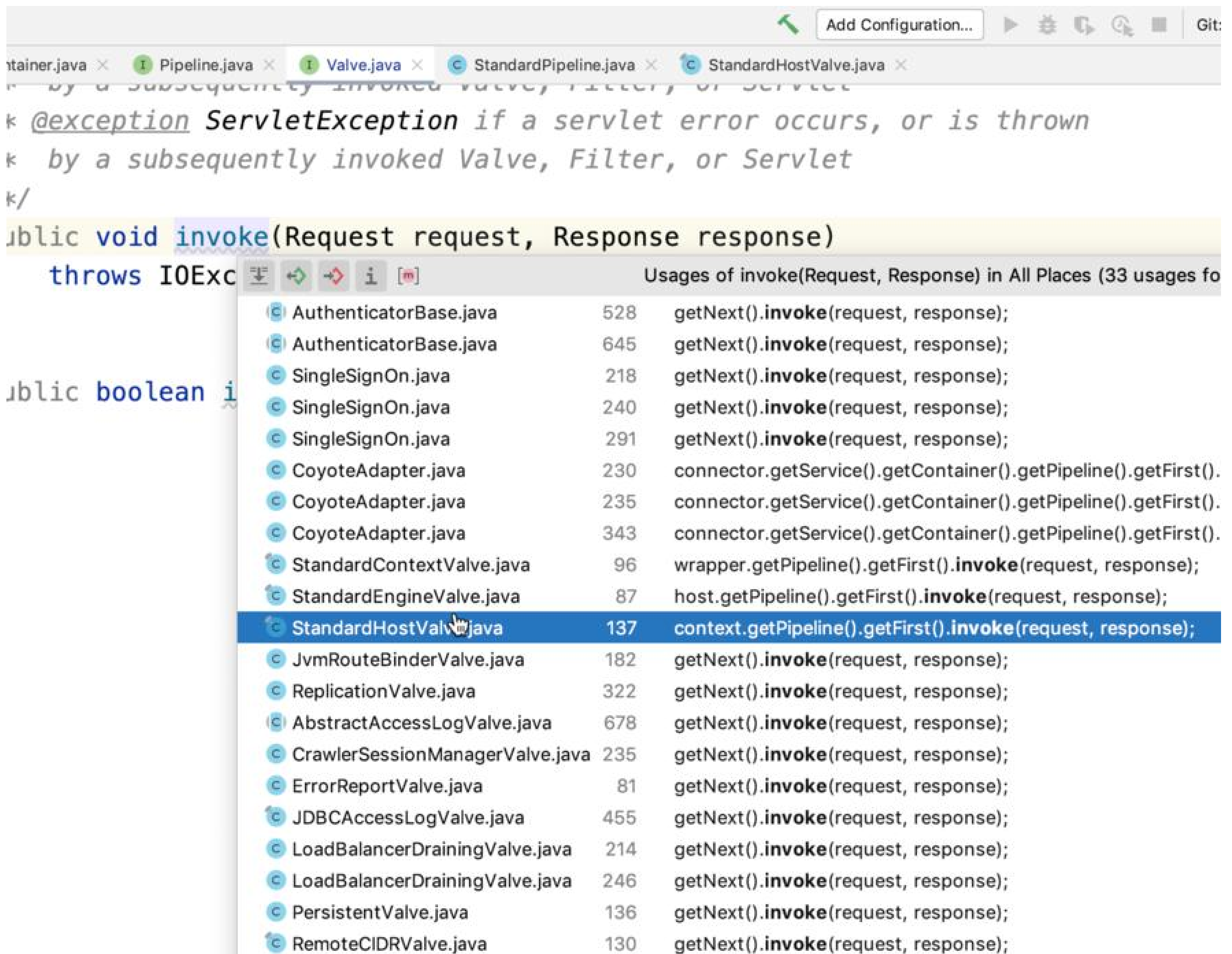
```
    STARTED(true, Lifecycle.AFTER_START_EVENT),
    STOPPING_PREP(true, Lifecycle.BEFORE_STOP_EVENT),
```

Container 接口,pipeline,valve

通过其名字,我们可以大致猜测,是表示一个容器,那么做为一个容器,他有自己的名字,有子容器(Children) 但是这些容器本身并没有处理业务逻辑的功能,所以,一个容器还会绑定一个执行链.

Tomcat中定义了Pipeline, valve来帮助Container来处理业务,每个Container组件通过执行一个pipeli e里面的valve 来执行业务.对于每个容器,都会有一个默认的valve在最底层,最后来执行.如果用户/使用没有自定义的话,就会使用默认的.

我们可以看到在valve的方法中, invoke方法,已经是和业务/具体处理是相关联的了



我们以StandardEngineValve为例,可以看到,对于同一个request和response,他在Invoke方法中,又调了host的valve的inove

```
@Override
    public final void invoke(Request request, Response response)
        throws IOException, ServletException {

        // Select the Host to be used for this Request
        Host host = request.getHost();
        if (host == null) {
            response.sendError
```

```
            (HttpServletResponse.SC_BAD_REQUEST,
             sm.getString("standardEngine.noHost",
                          request.getServerName()));
        return;
    }
    if (request.isAsyncSupported()) {
        request.setAsyncSupported(host.getPipeline().isAsyncSupported());
    }

    // 调用host的valve去继续处理request,response
    host.getPipeline().getFirst().invoke(request, response);

}
```

后面我们学习也就是以对request, response的处理为主

最后说两句

今天我们学习的tomcat组件的接口结构,是后面学习的基础,在学习过程中,小伙伴们有什么问题,可以和刀一起交流:best396975802

原文链接：tomcat 学习 |tomcat 中组件结构设计