rhwayfun专栏

在等待的日子里，刻苦读书，谦卑做人，养得深根，日后才能枝叶茂盛！

个人资料

rhwayfunn

关注 发私信

博客专家

访问： 375972次

积分： 5880

等级： BLOG 6

排名： 第4617名

原创： 212篇 转载： 13篇

译文： 1篇 评论： 244条

博客专栏

计算机面试基础

文章：0篇

阅读：0

Java开发问题汇总

文章：8篇

阅读：6769

互联网协议系列

文章：12篇

阅读：13987

深入理解Tomcat

文章：7篇

阅读：29955

我的微博

深入理解Tomcat系列之一：系统架构

标签： tomcat

2016-03-27 21:24 4251人阅读 评论(1)

分类：

Tomcat源码剖析（6）

前言

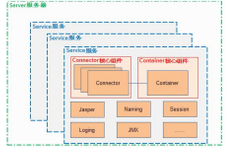
Tomcat是Apache基金组织下的开源项目，性质是一个Web服务器。下面这种情况很普遍：在eclipse床架一个web项目并部署到Tomcat中，启动tomcat，在浏览器中输入一个类似`http://localhost:8080/webproject/anyname.jsp`的url，然后就可以看到我们写好的jsp页面的内容了。一切都是那么自然和顺理成章，然而这一切都是源于tomcat带给我们的，那么在tomcat背后，这一切又是怎么样发生的呢？带着对tomcat工作原理的好奇心，我决定研究一下tomcat的源码，然而部署源码环境的过程却让我心灰意冷，本着搞不定我还真不信的热情，折腾了一个晚上+一个早上，终于把源码源码环境搭建好了。

为了让文章显得更有条理性，我将从以下几个方面说明Tomcat的工作流程：

- 搭建Tomcat源码环境指导
- Tomcat的系统架构
- Tomcat中的核心组件说明
- Servlet工作原理
- 一个例子

Tomcat的系统架构

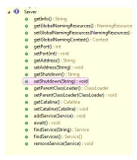
首先我们从一个宏观的角度来看一下Tomcat的系统的架构：



从这张图中可以看到，Tomcat的核心组件就两个Connector和Container（后面还有详细说明），一个Connector+一个Container构成一个Service，Service就是对外提供服务的组件，有了Service组件Tomcat就可以对外提供服务了，但是光有服务还不行，还得有环境让你提供服务才行，所以最外层的Server就为Service提供了生存的土壤。那么这些个组件到底是干嘛用的呢？Connector是一个连接器，主要负责接收请求并把请求交给Container，Container就是一个容器，主要装的是具体处理请求的组件。Service主要是为了关联Container与Connector，一个单独的Container或者一个单独的Connector都不能完整处理一个请求，只有两个结合在一起才能完成一个请求的处理。Server这是负责管理Service集合，从图中我们看到一个Tomcat可以提供多种服务，那么这些Serice就是由Server来管理的，具体的工作包括：对外提供一个接口访问Service，对内维护Service集合，维护Service集合又包括管理Service的生命周期、寻找一个请求的Service、结束一个Service等。以上就是对Tomcat的核心组件的简要说明，下面我们详细看看每一个组件的执行流程：

Server

上面说Server是管理Service接口的，Server是Tomcat的顶级容器，是一个接口，Server接口的标准实现类是StandardServer类，在Server接口中有许多方法，我们重点关注两个方法：addService()和findService(String)。我们先来看看Server接口的全貌：



接着看看addService()和findService(String)的实现代码：

代码清单1-1:

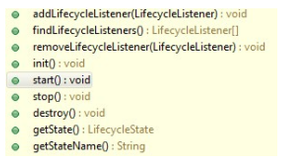
```
/**
 * Add a new Service to the set of defined Services.
 *
 * @param service The Service to be added
 */
@Override
public void addService(Service service) {

    service.setServer(this);

    synchronized (services) {
        Service results[] = new Service[services.length + 1];
        System.arraycopy(services, 0, results, 0, services.length);
        results[services.length] = service;
        services = results;

        if (getState().isAvailable()) {
            try {
                service.start();
            } catch (LifecycleException e) {
                // Ignore
            }
        }

        // Report this property change to interested listeners
```

```
        support.firePropertyChange("service", null, service);
    }

}
```

可以看到，Server使用一个数组来管理Service的，每添加一个Service就把原来的Service拷贝到一个新的数组中，再把新的Service放入Service数组中。所以Server与Service是关联在一起的，那么后面的`getState().isAvailable()`是干嘛的呢？判断状态是否无效，从而决定是否执行service方法。这里说到了状态，就不得不说Tomcat管理各组件生命周期的Lifecycle接口了：

Lifecycle接口

Tomcat中的组件都交给这个接口管理，但是具体组件的生命周期是由包含组件的父容器来管理的，Tomcat中顶级容器管理着Service的生命周期，Service容器又是Connector和Container的父容器，所以这两个组件的生命周期是由Service管理的，Container也有子容器，所以管理着这些子容器的生命周期。这样，只要所有组件都实现了Lifecycle接口，从顶层容器Server开始，就可以控制所有容器的生命周期了。Lifecycle接口中定义了很多状态，在api中详细说明了调用不同方法后的状态转变，同时定义了不同的方法，这些方法在执行后状态会发生相应的改变，在Lifecycle接口中定义了如下方法：



在StandServer中实现了startInernal()方法，就是循环启动StandServer管理的Service的过程，Tomcat的Service都实现了Lifecycle接口，所以被管理的Service都将被通知到，从而执行start()方法，startIntenal()方法是这样的：

代码清单1-2：

```
/**
 * Start nested components ({@link Service}s) and implement the requirements
 * of {@link org.apache.catalina.util.LifecycleBase#startInternal()}.
 *
 * @exception LifecycleException if this component detects a fatal error
 *  that prevents this component from being used
 */
@Override
protected void startInternal() throws LifecycleException {

    fireLifecycleEvent(CONFIGURE_START_EVENT, null);
    setState(LifecycleState.STARTING);

    globalNamingResources.start();

    // Start our defined Services
    synchronized (services) {
        for (int i = 0; i < services.length; i++) {
            services[i].start();
        }
    }
}
```

现在所有的Service就会收到通知继而执行start方法。如果一个Service不允许被使用将会抛出一个LifecycleException异常。

stopIntenal()会通知所有Service执行stop方法，具体处理流程与startIntenal()方法类似。这个执行过程涉及一个非常重要的设计模式，就是观察者模式。

现在我们已经能够知道了容器通过Lifecycle接口管理容器的生命周期，那么在父容器的状态改变具体是怎么样通知给子容器的呢？回到代码清单1-2，我们注意到有一个`fireLifecycleEvent()`方法，fireLifecycleEvent()的执行流程如下：

1. 调用LifecycleBase的fireLifecycleEvent(LifecycleListener listener)方法，LifecycleBase是一个抽象类，实现了Lifecycle接口
2. 继续调用LifecycleSupport（是一个辅助完成对已经注册监听器的事件通知类，不可被继承，使用final)的fireLifecycleEvent(String type, Object data)方法
3. 完成事件通知

fireLifecycleEvent(String type, Object data)的方法如下：

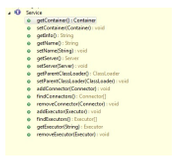
代码清单1-3：

```
/**
 * Notify all lifecycle event listeners that a particular event has
 * occurred for this Container.  The default implementation performs
 * this notification synchronously using the calling thread.
 *
 * @param type Event type
 * @param data Event data
 */
public void fireLifecycleEvent(String type, Object data) {

    LifecycleEvent event = new LifecycleEvent(lifecycle, type, data);
    LifecycleListener interested[] = listeners;
    for (int i = 0; i < interested.length; i++)
        interested[i].lifecycleEvent(event);

}
```

所以，具体事件的通知是由LifecycleListener接口的lifecycleEvent方法完成的，各实现类可以根据不同的情况实现不同的事件监听逻辑

Service

Service是具体提供服务的接口，一个Service包装了Connector和一个Container，在Tomcat中这点是如何实现的呢？Service是一个接口，其标准实现类是StandardService，下面是这两个类的鸟瞰图：



这里，我们只关心与Connector和Container最紧密的方法：setContainer()和addConnector()方法，先看一下setContainer()方法的源码：

代码清单2-1：

```
/**
 * Set the <code>Container</code> that handles requests for all
 * <code>Connectors</code> associated with this Service.
 *
 * @param container The new Container
 */
@Override
public void setContainer(Container container) {

    Container oldContainer = this.container;
    if ((oldContainer != null) && (oldContainer instanceof Engine))
        ((Engine) oldContainer).setService(null);
    this.container = container;
    if ((this.container != null) && (this.container instanceof Engine))
        ((Engine) this.container).setService(this);
    if (getState().isAvailable() && (this.container != null)) {
        try {
            this.container.start();
        } catch (LifecycleException e) {
            // Ignore
        }
    }
    if (getState().isAvailable() && (oldContainer != null)) {
        try {
            oldContainer.stop();
        } catch (LifecycleException e) {
            // Ignore
        }
    }

    // Report this property change to interested listeners
    support.firePropertyChange("container", oldContainer, this.container);

}
```

从代码中可以看到这个方法主要的任务是设置一个Container容器来处理一个或者多个Connector传送过来的请求。首先判断当前的Service是否已经关联了Container容器，如果已经关联了就去除这个关联关系。如果原来的Container容器已经启动了就终止其生命周期，结束运行并设置新的关联关系，这个新的Container容器开始新的生命周期。最后把这个过程通知给感兴趣的事件监听程序。

下面看看addConnector的方法：

代码清单2-2：

```
/**
 * Add a new Connector to the set of defined Connectors, and associate it
 * with this Service's Container.
 *
 * @param connector The Connector to be added
 */
@Override
public void addConnector(Connector connector) {

    synchronized (connectors) {
        connector.setService(this);
        Connector results[] = new Connector[connectors.length + 1];
        System.arraycopy(connectors, 0, results, 0, connectors.length);
        results[connectors.length] = connector;
        connectors = results;

        if (getState().isAvailable()) {
            try {
                connector.start();
```

```
            } catch (LifecycleException e) {
                log.error(sm.getString(
                        "standardService.connector.startFailed",
                        connector), e);
            }
        }

        // Report this property change to interested listeners
        support.firePropertyChange("connector", null, connector);
    }

}
```

执行过程也比较清楚：用一个同步代码块包住connectors数组，首先设置connector与container和service的关联关系，然后让connector开始新的生命周期，最后通知感兴趣的事件监听程序。注意到Connector的管理和Server管理Service一样都使用了数组拷贝并把新的数组赋给当前的数组，从而间接实现了动态数组。之所以使用数组我想可能是出于性能的考虑吧。

查看评论

苍海一薯 1楼 2017-04-11 20:24发表

多个Connector+一个Container构成一个Service

网站客服 杂志客服 微博客服 webmaster@csdn.net 400-660-0108 | 北京创新乐知信息技术有限公司 版权所有 | 江苏知之为计算机有限公司 |

江苏乐知网络技术有限公司

京 ICP 证 09002463 号 | Copyright © 1999-2017, CSDN.NET, All Rights Reserved