How Tomcat Works

# 前言

最佳伴侣：

https://github.com/Aresyi/HowTomcatWorks

这个是一个国人阅读这本书的代码实践，非常不错。这个作者也翻译了这本书。

# Overview

这本书讨论的是Tomcat4.1/5.0，介绍这款开源软件的内部实现机制。Tomcat是一个复杂的系统，由许多组件组成。这本书介绍了一个总体的场景，介绍了tomcat各个组件。

# Chap1 A Simple Web Server

这章介绍web server(HTTP server)是如何运作的。通过代码实例，介绍怎么搭建一个http server。

|  |
| --- |
| 官方实现： |

|  |
| --- |
| 自己实现：  HttpServer.java //  Request.java  Response.java |

# Chap2 A Simple Servlet Container

这章介绍如何开发一个简单的servlet容器。然后，我们会进化成一个功能更加复杂的servlet容器。

|  |
| --- |
| 官方实现： |

|  |
| --- |
| 自己实现：  PrimitiveServlet.java // 基础的Servlet处理类，实现了javax.servlet.Servlet接口  HttpServer1.java // 这个httpserver在之前版本的基础上，加入了对Servlet的处理  StaticResourcePrcessor.java // the processor to deal with the static resource |

# Chap3 Connector

Tomcat主要包含两大模块：Container和Connector。其中Container是Java容器，Connector是用于创建requests/responses。

这章介绍Connector.。Connector主要是遵循servlet规范，产生以下两个对象：

javax.servlet.http.HttpServletRequest

javax.servlet.http.HttpServletResponse

供servlet容器消费这两个对象

|  |
| --- |
| 官方实现： |

|  |
| --- |
| 自己实现：  BootStrap.java  HttpConnector.java  HttpProcessor.java process() parseRequest() |

# Chap4 Tomcat Default Connector

Chap3中开发的一个简单的Connector,主要是帮助我们理解Tomcat的Default Connector。

Connector的功能是什么呢？

1.等待客户端的HTTP请求；

2.生成请求对象/返回对象： HttpServletRequest/HttpServletResponse

3.将请求对象/返回对象传递给servlet容器(Container)

Connector是一个单独的实现组件，和servlet container是解耦的，因此是可替换的。只要实现如下接口就行了：org.apache.catalina.Connector, 这个思想我们要好好学习一下。

Connector需要兼容HTTP各个版本。

比如需要支持HTTP1.1版本的新功能：Persistent Connection/Chunked Encoding/100 Status

为了支持并发，Connector需要管理一个线程池，这个线程池中每个线程是之前的HttpProcessor，HttpProcessor用于处理单个Http请求。

|  |
| --- |
| 官方实现：  Connector.java // base class of connector  核心的方法包括：getContainer()/setContainer()/createRequest()/createResonse()  Request.java // implementation in tomcat, implement the interface of HttpServletRequest in javax  Response.java // implementation in tomcat, implement the interface of HttpServletResponse in javax |

|  |
| --- |
| 自己实现：  SimpleContainer.java  BootStrap.java |

SimpleContainer.java中，主要是实现了Container interface的invoke()方法：

invoke(Request request, Response response)

这个方法很清楚，就是调用一个connector实现，处理请求，返回响应内容。

到这章为止，我们通过分析Connector的实现，在一个默认的Container中，调用Connector的实现逻辑，实现了处理Servlet Request，并返回Servlet Response的功能。

# Chap5 Container

Container代表的是Java容器，主要是接收servlet请求，然后响应给客户端。container主要包括如下的接口：

org.apache.catalina.Container

有四类Container:

Engine, Host, Context, and Wrapper

这四个接口是继承了Container接口。所以这四个接口都代表一种类型的Container。

这章讨论Context, and Wrapper 。

Container采用的是pipeline机制。

|  |
| --- |
| 官方实现：  Container.java // the interface 代表一个Java容器容器  Engine.java // the interface that extends from the interface Container  Host.java // the interface that extends from the interface Container  Context.java // 代表web application，可以包含多个Wrapper  Wrapper.java // 代表servlet definition，包含整个servlet生命周期，包括init/service/destroy  Pipeline.java // interface  Valve.java // interface 这个接口代表一次客户端请求中的一个组件  Contained.java // the interface |

我们分析一下Container接口有哪些重要方法：

|  |
| --- |
| invoke(Request request, Response response) // 这个方法很清楚，就是调用一个connector实现，处理请求，返回响应内容 |

|  |
| --- |
| 自己实现：  SimpleLoader  SimplePipeline  SimpleWrapper  SimpleContextValve  SimpleContextMapper  SimpleContext  BootStrap1  BootStrap2 |

# Chap6 Lifecycle

Tomcat启动的时候，需要启动很多组件；Tomcat关闭的时候，也需要关闭很多组件。这些组件的启动和关闭，由Lifecycle这个接口来管理。

|  |
| --- |
| 官方实现：  LifecycleEvent.java  LifecycleSupport.java |

|  |
| --- |
| 自己实现：  SimpleContextLifecyleListener.java  SimpleLoader.java  SimplePipeline.java  SimpleWrapper.java |

# Chap7 Logger

Tomcat中，日志相关的核心接口是Logger

|  |
| --- |
| 官方实现：  Logger.java // interface  LoggerBase.java // base class  SystemOutLogger.java  SystemErrLogger.java |

|  |
| --- |
| 自己实现： |

# Chap8 Loader

所谓的Loader，就是Tomcat中，加载Servlet类的类加载器。

## Java Class Loader

这个小结介绍了Java Class Loader加载Java Class类的的顺序，介绍得非常清楚，实际加载顺序为：

1. Bootstrap
2. Extension
3. System Class loader

我们具体说明一下：

1. Bootstrp

所谓的bootstrap，其实就是我们启动JVM运行Java程序要用到的核心类，具体代码可以参考我们本地JDK1.8环境的Java源码：

/Library/Java/JavaVirtualMachines/jdk1.8.0\_71.jdk/Contents/Home/src/java

1. Extension

所谓的extension，就是一些扩展的类，一般是Java实现厂商扩展的一些方便开发的类，参考本地的扩展类：

/Library/Java/JavaVirtualMachines/jdk1.8.0\_71.jdk/Contents/Home/src/javax

1. System Class loader

就是我们当前Java程序指定的classpath

比如对于IDEA工程来说，就是out/production目录

对于MyEclipse来书，就是bin目录

还有就是我们手工指定的classpath，比如

export CLASSPATH=.:$JAVA\_HOME/lib/dt.jar:$JAVA\_HOME/lib/tools.jar

1. User specified class

最后才去用户指定的loader目录找要执行的class类

比如我们通过 URLClassLoader初始化loader类的时候，指定的加载路径：

file:/Users/zhoushuo/Documents/workspace/TomcatWin/WEB-INF/classes/

以上的加载顺序是从1到4。

## The Loader Interface

## The Reload Interface

## The WebappLoader Class

## The WebappClassLoader Class

|  |
| --- |
| 官方实现：  Loader.java // interface  Reloader.java // interface  WebappLoader.java // an implementation for Loader interface  ResourceEntry.java // used for class load caching |

|  |
| --- |
| 自己实现：  SimpleContextConfig.java |

## 代码分析

重点代码有两个：

1.

## 要关注的问题

我们看Loader这块内容时，要关注这几个问题：

1. reload的时候，如何保证cache中加载的class重新加载？
2. 一个tomcat下，部署的不同应用，如何实现隔离？
3. 一个tomcat下，部署的不同应用，如何共享公共的jar包？
4. 如何限制tomcat上部署的某个应用，只能访问本应用范围内的类，仅限于WEB-INF/classes和WEB-INF/lib这两个目录？

## 总结

# Chap9 Session Management

Session管理主要有Manager.java这个接口来实现。Session manager做的事情主要是创建session/销毁session/更新session。

实现类为：ManagerBase/StandardManager/PersistentManager/DistributedManager

Session本身的接口为：Session.java/HttpSession.java.实现类为:StandardSession.java。

另外还有一个接口就是Store

.java，用来将session持久化到本地。对应的实现类为：

|  |
| --- |
| 官方实现：  **Manager.java** // main interface of session management  ManagerBase.java // implementation of Manager interface  StandardManager // implementation of Manager interface  PersistentManager // implementation of Manager interface  DistributedManager // implementation of Manager interface  **Session.java** // Session相关的interface  HttpSession.java // session interface of servlet  StandardSession.java // an implementation of Session/HttpSession interface  Store.java // interface to provide a permanent store  StoreBase.java // implementation of Store interface |

|  |
| --- |
| 自己实现： |

# Chap10 Security

|  |
| --- |
| 官方实现：  Realm.java // the interface to authenticate the user  Principal.java // the interface to  Generic Principal.java // the implementation fo Principal interface |

|  |
| --- |
| 自己实现：  SimpleContextConfig.java  SimpleRealm.java  SimpleUserDatabaseRealm.java |

# Chap11 StandardWrapper

之前我们介绍了Container/Context/Wrapper，其中Container接口代表了Java容器，Context和Wrapper这两个接口都继承了这个接口。

其中Context代表web application，Wrapper代表servlet定义。Context可以包含多个Wrapper。

这章我们仔细讲讲Wrapper接口的实现。这个实现就是StandardWrapper。这个实现类当然需要有一些处理servlet的方法，比如加载servlet，获取ServletContext()，获取ServletConfig信息等。

|  |
| --- |
| 官方实现：  StandardWrapper.java // Wrapper接口的一个实现类  StandardWrapperValve.java // Valve接口的一个实现类，用于StandardWrapper中处理当前servlet中filter的各个事情  ApplicationFileterConfig.java // 实现了servlet FilterConfig接口，管理Filter实例  ApplicationFileterChain.java // 实现了servlet FilterChain接口，执行Filter实例 |

|  |
| --- |
| 自己实现： |

# Chap12 StandardContext

Context代表web application:应用。Context可以包含多个Wrapper。这章介绍Context接口的一个实现:StandardContext。

一般来说，StandardContext还支持reload功能，就是一旦web.xml或者WEB-INF下的class有改变，会自动把这些改变加载进来。

|  |
| --- |
| 官方实现：  StandardContext.java // an implementation of Context interface  ContainerBase.java // StandardContext.java的基类  ContainerBackgroundProcessor.java // 这个类是ContainerBase类的一个内部类，主要处理一些异步进程(比如定期检测class是否有更新/web.xml是否有更新等)。 |

|  |
| --- |
| 自己实现： |

# Chap13 Host and Engine

Host是这样的，如果你要在Tomcat上部署多个应用(Context)，就会用到host。

Engine是代表整个tomcat Catalinna的servlet引擎，是顶层Container。

总而言之，这些Container从顶层往下的关系为：

Engine->Host->Context->Wrapper

|  |
| --- |
| 官方实现：  Host.java // the interface extends from Container.java  StandardHost.java // the implementation of Host interface  StandardHostValve.java // the default valve of StandardHost  Engine.java // the interface  StandardEngine.java // the implementation of Engine interface |

|  |
| --- |
| 自己实现： |

# Chap14 Server and Service

Server代表了整个tomcat servlet 容器。也就是tomcat进程(默认端口为8080).我们可以通过Server start来启动tomcat下所有的container components。也可以通过Server stop把这些container关闭。

Server包含Service，通过Service管理像Container/connector这类的组件。

|  |
| --- |
| 官方实现：  Server.java // interface  StandardServer.java // the implementation of Server.java  Service.java  StandardService.java |
|  |

|  |
| --- |
| 自己实现： |

# Chap15 Digester

Digester是一个apache commons组件。这个组件主要是处理XML文件的。

那么Digester在tomcat中的作用是什么呢？

我们看看ContextConfig.java文件。里面有两个Digest对象：

1. contextDigester
2. webDigester

其中webDigester用来处理web.xml。

contextDigester可能是用来处理其他配置文件。当然，我们知道Tomcat还有很多配置文件，比如config目录下有很多，server.xml。

|  |
| --- |
| 官方实现：  ContextConfig.java // Context的配置i |

|  |
| --- |
| 自己实现： |

# Chap16 Shutdown Hook

主要介绍如何优雅地关闭Tomcat。

|  |
| --- |
| 官方实现：  CatalinaShutdownHook.java // 定义在Catalina.java中的一个类 |

|  |
| --- |
| 自己实现：  ShutdownHookDemo.java //  ShutdownHook.java //  MySwingApp.java //  MySwingAppWithShutdownHook.java // |

# Chap17 Tomcat Startup

这章介绍Tomcat如何启动。涉及的类包括：Catalina.java/BootStrap.java

此外，还介绍了Tomcat在Windows/Unix系统下的启动脚本。我们只要了解Tomcat在Unix下的启动脚本就行了。为了让读者更好地理解启动脚本，居然还讲解了shell的一些基本用法，可以说也是很贴心了。

Tomcat启动脚本在tomcat/bin/start.sh，但是真正的启动脚本是catalina.sh

Tomcat停止脚本为shutdown.sh。

后续我如果要实现自己的Tomcat，也可以模仿这个，创建启停脚本。

|  |
| --- |
| 官方实现：  Catalina.java // 启动的方法主要是start()/stop()  BootStrap.java  start.sh  shutdown.sh |

|  |
| --- |
| 自己实现： |

# Chap18 Deployer

这章介绍Tomcat如何部署应用(主要是如何部署war包)。

但是奇怪的是，PDF中介绍的接口和实现类在tomcat5中一个也没有找到。我们尝试下载Tomcat4试试。

|  |
| --- |
| 官方实现： |

|  |
| --- |
| 自己实现： |

# Chap19 Manager Servlet

# Chap20 JMX-Based Management