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。

这份文档，主要目的也是把我实践tomcat的整个过程记录下来。

# 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。

## 整体处理流程

我们在开头部分，就要把Tomcat整个架构说清楚。Tomcat接收到一个来自客户端的http请求，内部到底是如何处理的，这个整个流程，要在开头的时候就介绍清楚。

# 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。

## 代码说明

从整体上来看，这章引入的主要是

### 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 |

## 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

## 客户端Session使用场景(Cookie)

为了说清楚Session是怎么使用的，我们先看看Session以Cookie的形式保存在客户端浏览器的内容，究竟是什么。

### Amazon

session到底怎么用呢？我们尝试访问一下Amazon.cn，当我们用账号登录之后，可以看到，几乎每次访问Amazon的后台服务，都会在request http header 中看到Cookie，标识了我在Amazon的登录状态。

其中整体的http header key是”Cookie”；对应http header value是Cookie信息。Cookie信息非常丰富，包括核心的session-id,session-token,以及session-id-time。这些信息字段以分号分隔。格式非常清晰。

服务端接收到Cookie中的session信息，会做出相应的处理，包括身份验证、session超时验证等等。

关于session-id信息，那就是标识某个客户端。

关于session-token，我们要好好讨论一下。对于session-id，我们知道，需要服务端保存这个session-id信息。这样才能验证客户端的身份。但是我们知道，服务端往往要应对成千上万的客户端请求，如果把这些客户端的session-id信息都保存在服务端，那压力无疑是巨大的。那么有一种解决方案是，服务器在生成客户端的session-id之后，将session-id、登陆时间等session信息通过加密算法和秘钥进行加密，生成token。然后将session-id等登陆信息和token一起返回给客户端。客户端将session信息和token保存在客户端cookie，后续请求服务端的时候就发送session信息和token。

服务端接收到session信息和token之后，再次对session信息进行加密计算，判断加密结果和token信息是否一致。从而实现了对session信息的验证。保证客户端上送的session信息是未经篡改的。

|  |
| --- |
| **Cookie:**  session-id=458-9781356-6991254;  ubid-acbcn=460-9604323-6863029;  session-token="/MoPj9tOHgXfCyE+btvyq63b6Zr9WkRz/Jeq0CzWD0fj4FG/Wwm7R3wOL6ZdEgWBXIfIizhciSFGQB3p7s7ERIyigilMOQ79cuDQgpeGuugu+6BCT9KTRV18E1V/F6NAsG7ZNbWGUACD1Em8OToenGFnfO4SaApvRmbQvQRd+bae1gPTPiIq4YaQxIBHAaNoILy73YcSrID450FsMqUCCYWZUY32I6pEj2iaYmhERIk=";  x-acbcn="0lgdztJIjcbGVEGDb6PySdQUz?oRsf7r";  at-main=Atza|IwEBIL-f9Fdn\_AgPS\_U82BGH3h0Zt-SOAAy4YzSr7ch2ITIPM1vEYAwdMwneYiIyA7plX3WzM5Wivkc\_eowUnkAJiR4Xa6-287fagZxrz9slW6aSr4h08-fAXzbg0GcpmhuYEWcQNCGgtvupCU7RW4aAFwzHHDbx-WQs999NVq\_ICMd0\_T\_Qii679RvLRCE9SwEEggHuIUiDYoy4bpaYQlFvL7Y\_;  sess-at-main="q4jzEbQmhNXZlTSZjZEJ4LSgMdX1RC2F8JfvJ4WUCtQ=";  sst-main=Sst1|PQEiDBBVY940w3Sp7gAnUFCoCbBp9M5iKRdVQMGDndzXZI-QE\_HklJfdS8ebJhn4IYZG30EIv16FtQb4ulfz08pUJKmX6OeaXEUykoMAfMbDeTLergHFpJBr-yK9\_8aChxMYZ-Oa\_s3N0lsPrhzhpRZuhZCAJHzGgCAGskSGI4eJtXDtnX-JwPG9HAYtSkHZ5ovbWn6ViaTm1t0NuGzyIYbYOAiYCjY1vTSWjQANK2EO5PyeMoLdYQxGRkUqd\_ih53x4ZvHlQRLS0tEuNzazZ\_OjJrsHoXHNGOMSzTebSEkUCDQ; lc-acbcn=zh\_CN;  i18n-prefs=CNY;  session-id-time=2082729601l |

### 宁波银行网上银行

我们再看一个银行的例子。可以看到，宁波银行网上银行的cookie信息字段更加丰富，安全性做的更好：Cookie下的各个key/value值还做了转码，没办法直观看出cookie中各个字段的意思。同时，还有一个类似Token的字段”DFP\_DEVICEID”，可能是起到Token的作用，用来校验SESSION信息是否经过了篡改。

|  |
| --- |
| **Cookie:**  \_trs\_uv=kke2stws\_352\_12v8; Hm\_lvt\_8b9480950a6cadd80a66f238d3e4542e=1611670160; Hm\_lpvt\_8b9480950a6cadd80a66f238d3e4542e=1611670160; SF\_cookie\_15=29469227; JSESSIONID=0000DogtGcZdExS57PDc2cYNjNM:-1; Hm\_lvt\_077a59fb58754888f7db312d1e4c8296=1611670162; Hm\_lpvt\_077a59fb58754888f7db312d1e4c8296=1611670162; web\_vist\_num=1;  si=0fdeb02d7a554b70a5dc49facd1f1389;  is\_si\_expire=0;  iss\_webanalytics\_id=ab8497c472ff4c8d81c5bc769fcb0be0;  nu=1;  BSFIT\_OkLJUJ=38R4HHP55SWXS41X; BSFIT\_SMART=B1D20ADBC2616DBCF719F3131F869B2E; BSFIT\_DEVICEID=bda80fb285c6a0eba0bbdba3894e1320f909f1961182a2e4c803381248a0e7c7;  BSFIT\_EXPIRATION=1612248382857; BSFIT\_TRACEID=60102292f89829f14aaa0048;  fp\_ver=4.8.6;  DFP\_EXPIRATION=1611701555788; DFP\_OkLJUJ=FIabtpLqsBKMygKiu3\_oFzGuqXcuhgi\_; DFP\_DEVICEID=VuuanUc1GkN3xYSXnQUC\_7y57tCi5AsX0Du-mPCgD2S36DIhfyE5Mx-QG5dtVc6HAKHSS5XpbPzuvmlGfXPruhIEhhoTaOJXF8lMW\_7U82J3gZENnZAU6jm1fMoo0VaLG-\_D3wy3r5vJf5MkfXiiym7OO\_WD7yQ7 |

### 科管系统

我们再来看一个内网的例子，宁波银行科管系统。这个系统由于是内部管理系统，因此session机制就非常简单，就只要通过session-id来标识客户端就行了。

### session使用场景总结

从以上例子可以看到，

## 服务端Session使用场景(Sevlet如何使用Session)

了解了Session的使用场景，我们再来到服务端。

### 创建Session

在服务端，当Servlet接收到来自客户端的登录请求(往往带来了用户名和密码)之后，会判断登录请求是否合法。如果合法，就创建Session信息，相当于为这个客户端的本次登录动作创建一个会话标识。对应到代码中就是：

|  |
| --- |
| MySessionServlet.java  protected void doGet(HttpServletRequest req, HttpServletResponse resp)  {      Session session = (Session) req.getSession();      String sid = session.getId();  } |

备注：具体创建session对象的逻辑，参考HttpServletRequestBase.doGetSession()调用Manager创建Session。这里不展开了。

### 设置Session属性

Servlet端根据实际业务需求，设置session的各个属性，比如上一个小节Cookie中的各个key/value信息，比如user-id/session-token/expire信息等等。安全级别高的，Cookie内容就多。安全级别低的Cookie内容就少一些。

servlet执行完成后，会将session信息(包括session-id以及各个session属性)以Cookie的形式，放到http response header中，返回给客户端，保存在客户端浏览器中

### session验证

下次客户端再来访问服务端后台服务的时候，会带上Cookie信息，后台Servlet会根据Cookie中的session信息验证客户端的身份，并且验证客户端访问是否合法(比如是否篡改了Cookie信息)、Session是否过期，等等。安全等级高的，session验证逻辑复杂一些，安全等级低的，验证逻辑相对简单。

### Session超时、删除

服务端会定期验证session是否过期，把过期的session删除

### Session管理

对于那些高并发的场景，服务端可能会有成千上万的session对象，服务端会通过技术手段把过多的session持久化到本地或者数据库。

## 代码总体说明

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 |

## 代码详细说明

我们在”客户端Session使用场景”/”服务端Session使用场景”中已经搞清楚了Session的使用方式，接下来，我们看看在tomcat端，是如何处理Session的。

### 1.客户第一次登陆

客户第一次登陆后，会访问一个专门验证登陆信息的Servlet。Servlet开发者会在这个Servlet中创建Session。Servlet接收到来自客户端的登录请求后，会验证登录信息。一旦验证通过，就为这个用户这次登录创建一个session。

代码参考：

|  |
| --- |
| MySessionServet.doGet() |

### 2.tomcat端创建session对象实例

代码参考：

|  |
| --- |
| HttpRequestBase.doGetSession()  ManagerBase.createSession()  1.new StandardSession()  2.setId()  3.设置session参数(比如最大存活时间) |

### 3.将新创建的sesion对象实例纳入Manager管理

代码参考：

|  |
| --- |
| StandardSession.setId()  manager.add(this); |

### 4.新创建的session，以Cookie的形式，返回给客户端。

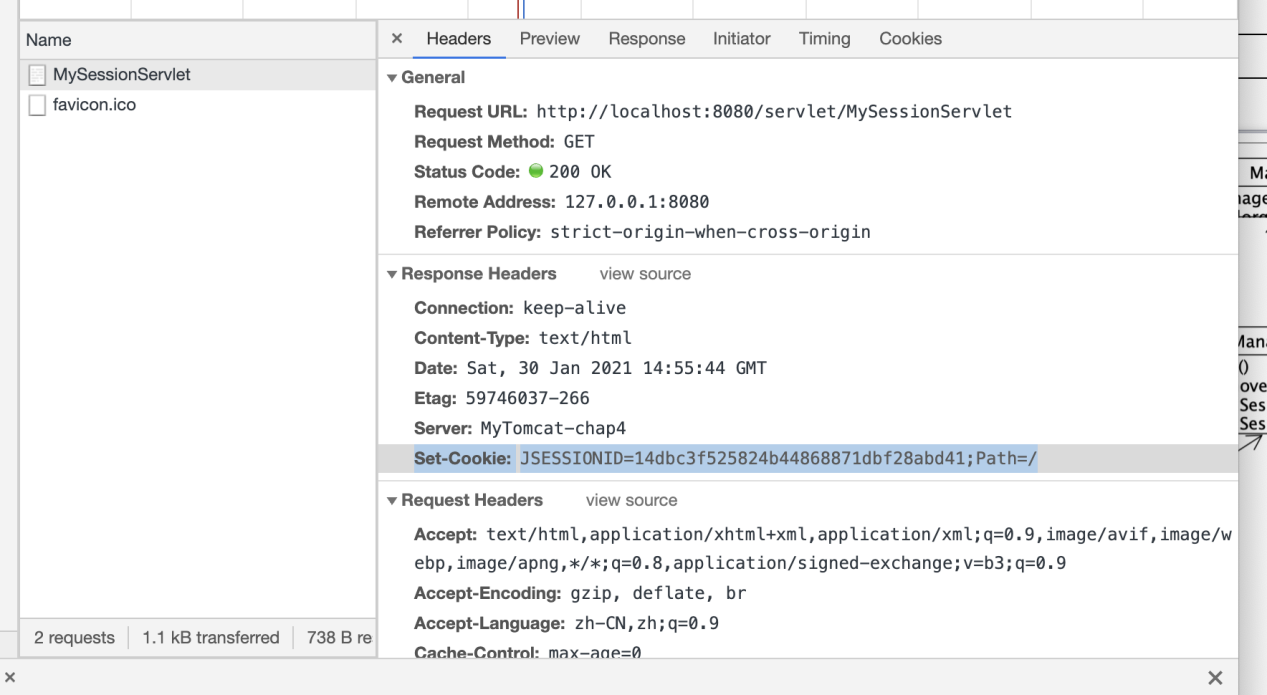
具体代码参考：

|  |
| --- |
| HttpResponseBase.sendHeaders() |

新创建的sesssion，要以Cookie的形式返回给客户端，必须满足以下一个条件：

|  |
| --- |
| 1.Request对象中能够取到session对象  HttpServletRequest.getSession()  这个在HttpServletRequest调用Manager创建session后，就会有自带一个session对象  2.session.isNew()为true  说明这个session是新创建的。这个值在ManagerBase.createSession()中设置为true。  需要注意的是，只有在session刚创建的时候，isNew为true，客户端后续来访问的时候，这个值当然是false。否则客户端Cookie会频繁更新。  3.session.isValid()为true  说明这个session未过期失效。  这个值在ManagerBase.createSession()中设置为true。  需要注意的是，这个只是设置为初始值true，后续Manager会定期管理session，一旦发现session超时，或者session非法，就会设置为false;  4.getContext().getCookies()为true  就是StandardContext.getCookies()为true |

最终客户端浏览器保留的Session信息的效果是这样的：



### 5.后续客户端访问tomcat 无需再次登录

具体代码参考

|  |
| --- |
| MySessionPlainServlet.doGet() |

如果之前已经登录过的话，这个Servlet就能获取到登录信息。

一般来说，判断客户端Servlet请求的Session会话是否合法不会放在Servlet中，一般会放到Filter中统一判断。

那么，tomcat如何判断客户端是第一次访问创建session，还是之前已经创建了session呢？这里需要特别说明，在tomcat接收到客户端http请求，执行servlet之前，就会统一判断当前客户端的会话信息。如果之前有登录过，就调用一下session.access()信息，将session.isNew()设置为false。具体参考：

|  |
| --- |
| 1.SimpleContextValve.invoke()  备注：后续好像会把这块逻辑迁移到StandardHostValve.invoke()中  2.StandardSession.access() |

### 6.Manager定期把过期的Session踢掉

Manager处理过期session的流程是这样的：

1. 记录lastAccessTime

每次客户端访问tomcat服务端servlet的时候，tomcat端都会记录这个客户端对应的session的lastAccessTime

代码参考：

|  |
| --- |
| SimpleContextValve.invoke()  StandardSession.access() |

1. Manager异步线程检测超时session

StandardManager启动的时候，会启动一个异步线程。异步线程会定期检测这个Manager管理的session，如果发现有session的lastAccessTime和现在的时间差距过大(maxInactiveInterval)，就将这个session设置为expire。

代码参考：

|  |
| --- |
| StandardManager.run()  StandardManager.processExpires() |

备注：maxInactiveInterval目前定义在:

|  |
| --- |
| Constants.MAX\_INACTIVE\_INTERVAL |

后续建议放到配置文件。

(3)Session回收

一旦某个session被设置为expire超时，就回收这个session。回收逻辑为：先将当前session对象的各个属性设置为初始值，再将当前session从Manager中remove掉，最后再放到Manager的回收队列中，供后续创建新session重复利用。

代码参考：

|  |
| --- |
| StandardSession.expire()  StandardSession.recycle()  ManagerBase.recycle() |

备注：Session回收的策略非常不错，提升了创建新session的效率，也减轻了JVM垃圾回收的压力。以后碰到类似的场景，可以参考tomcat对于session对象的处理方式。

### 7.Manager定期把内存中过多的Session持久化

具体代码参考：

## tomcat创建session的整体调用链路

我们单独用一个小节，说明一下session的调用链路

首先我们先说明一下session在servlet中是如何使用的：

servlet开发人员在servlet中调用session一般是这样的(具体参考MySessionServlet.java)：

|  |
| --- |
| protected void doGet(HttpServletRequest req, HttpServletResponse resp)  {  Session session = (Session) req.getSession();  String sid = session.getId();  ......  } |

可以看到，是通过HttpServletRequest调用的。

那么在HttpServletRequest中是如何取到session对象的呢？

这就要转到：HttpRequestBase.java

|  |
| --- |
| getSession()->doGetSession() |

备注：如何从HttpServletRequest 转到HttpRequestBase，请参考chap4\_request相关的内容，这里不展开了。

我们在看HttpRequestBase.doGetSession()方法相关的代码：

|  |
| --- |
| if (context != null)     manager = context.getManager();  .......  session = manager.createSession();  if (session != null)      return (session.getSession());  ...... |

备注：context的manager对象哪里来呢？当然是在bootstrap中指定对应的manager实现类啦

可以很清楚看到，在这里，由context获取Session Manager类

然后由Session Manager获取具体的session对象。

这样，整个session创建的链路就非常清晰了：

1. servlet开发人员在servlet类中，通过HttpServletRequest 接口使用session对象；
2. HttpServletRequest接口的实现类(HttpRequestBase)，通过context获取Manager
3. 由Manager创建session对象。

具体的代码可以参考：

|  |
| --- |
| /Users/zhoushuo/Documents/workspace/SimpleTomcat/simpletomcat-chap9/UML/Session\_create.uxf |

## Session

session接口主要是保存session相关的信息。最关键的方法为expire()，包含了session超时的逻辑。

整体结构如下：

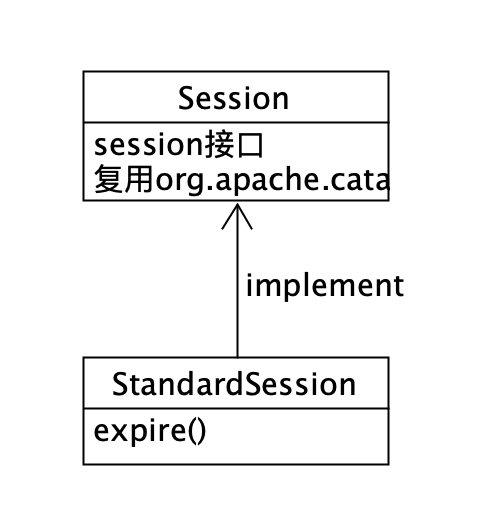


图1 session架构

## Manager

Managger主要是用于管理Session实例。

Manager整体架构如下：

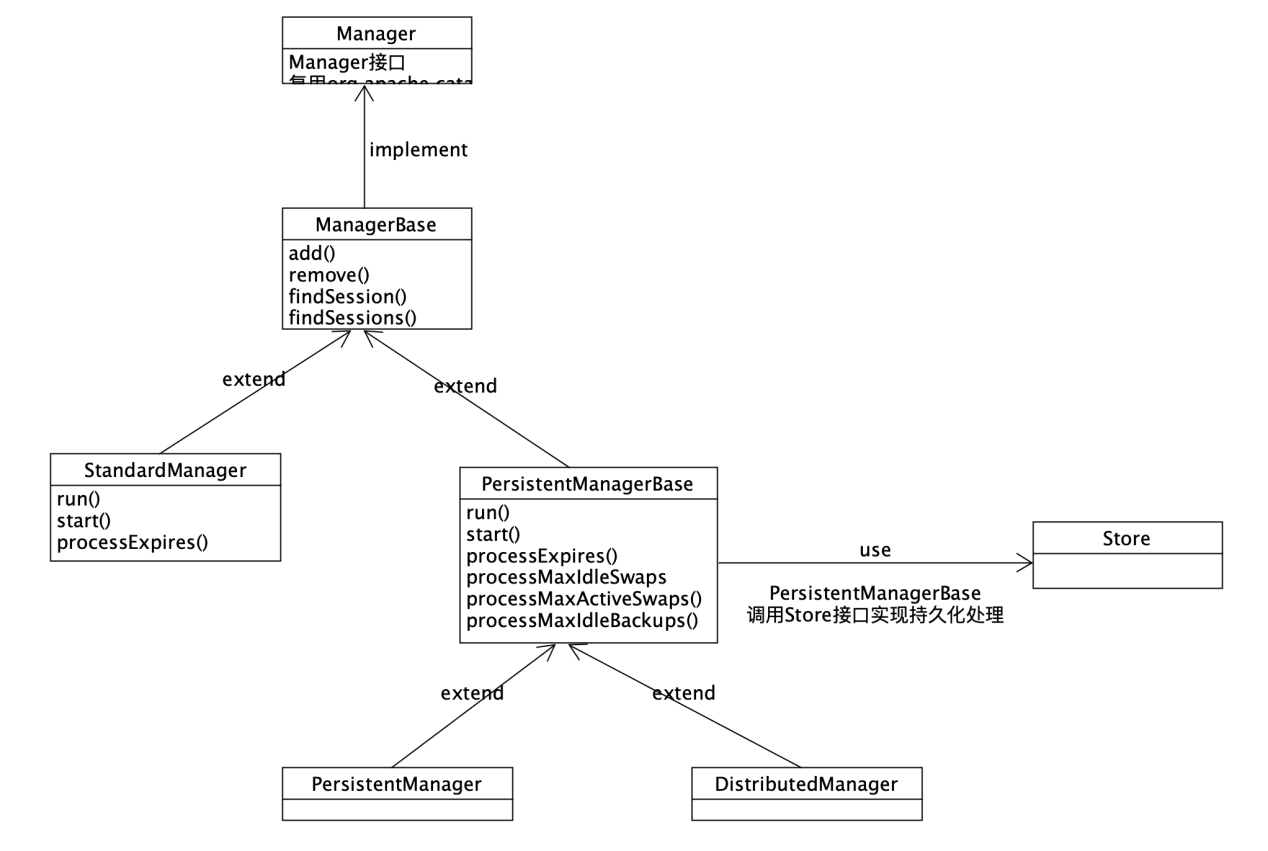


图2 Manager架构

### ManagerBase

ManagerBase.java是所有Manager实现类的基类，包含重要的方法为：

add()

remove()

findSession()

findSessions()

createSession()

### StandardManager

StandardManager.java是Manger接口的一种标准实现。最重要的方法包括：

run()

start()

processExpires()

StandardManager主要做两个事情：

1. run()的意思是StandardManager会启动一个异步线程，调用processExpires()方法，定期检测过期的session；
2. start()的意思是StandardManager会实现LifeCycle接口，在StandardManager对象启动的时候，自动从本地会从tomcat本地目录下读取一个文件：SESSIONS.ser。这个文件保存了持久化在tomcat本地目录下的session信息。然后StandardManager会把这些持久化的session信息加载到内存；

那么，StandardManger是怎么启动的呢？具体可以参考：

|  |
| --- |
| StandardContext.start() |

也就是说，StandardManger是在Context中启动的。当然，也可以在BootStrap中创建StandardManger的时候启动，但是BootStrap中的代码太冗余了。

### PersistentManagerBase

PersistentManagerBase和StandardManager差不多，也会启动异步线程定期检测过期的session对象。

|  |
| --- |
| run()  start()  processExpires() |

不同的是，正如名称显示的那样，PersistentManagerBase会对session对象进行持久化处理。对session对象实现持久化管理是啥意思呢？就是内存中的session对象空闲了很久，或者活跃的session对象太多达到了MaxActive的配置数量。那么PersistentManagerBase会把这些session对象 **swap out**到持久层(文件系统或者数据库)。保证内存中的session不会过多，影响tomcat JVM的性能。后续要用到session的时候，再通过**swap in**，从持久层加载到内存供servlet使用。

PersistentManagerBase涉及持久化的方法包括：

|  |
| --- |
| processMaxIdleSwaps  processMaxActiveSwaps()  processMaxIdleBackups() |

这个三个方法分别代表：

1. 一旦某个线程空闲时间过长，就Swap out to persistent；
2. 一旦PersistentManagerBase管理的活跃session数超过了某个阈值，就Swap out to persistent；
3. 一旦PersistentManagerBase管理的空闲session数超过了某个阈值，就备份到persistent。这样就算服务器宕机，后续重启的时候还能重新加载到内存中。

### PersistentManager

这个是标准的持久化Manager，就是一个名字而已，大部分实现方法都是继承自

PersistentManagerBase。

### DistributedManager

这个manager虽然也是继承自PersistentManagerBase，但是新增了分布式功能。意思就是在创建session对象之后，能够将session对象同步到整个tomcat集群各个节点上。

## Store

接下来，我们介绍一下Store的体系，Store的代码架构如下：

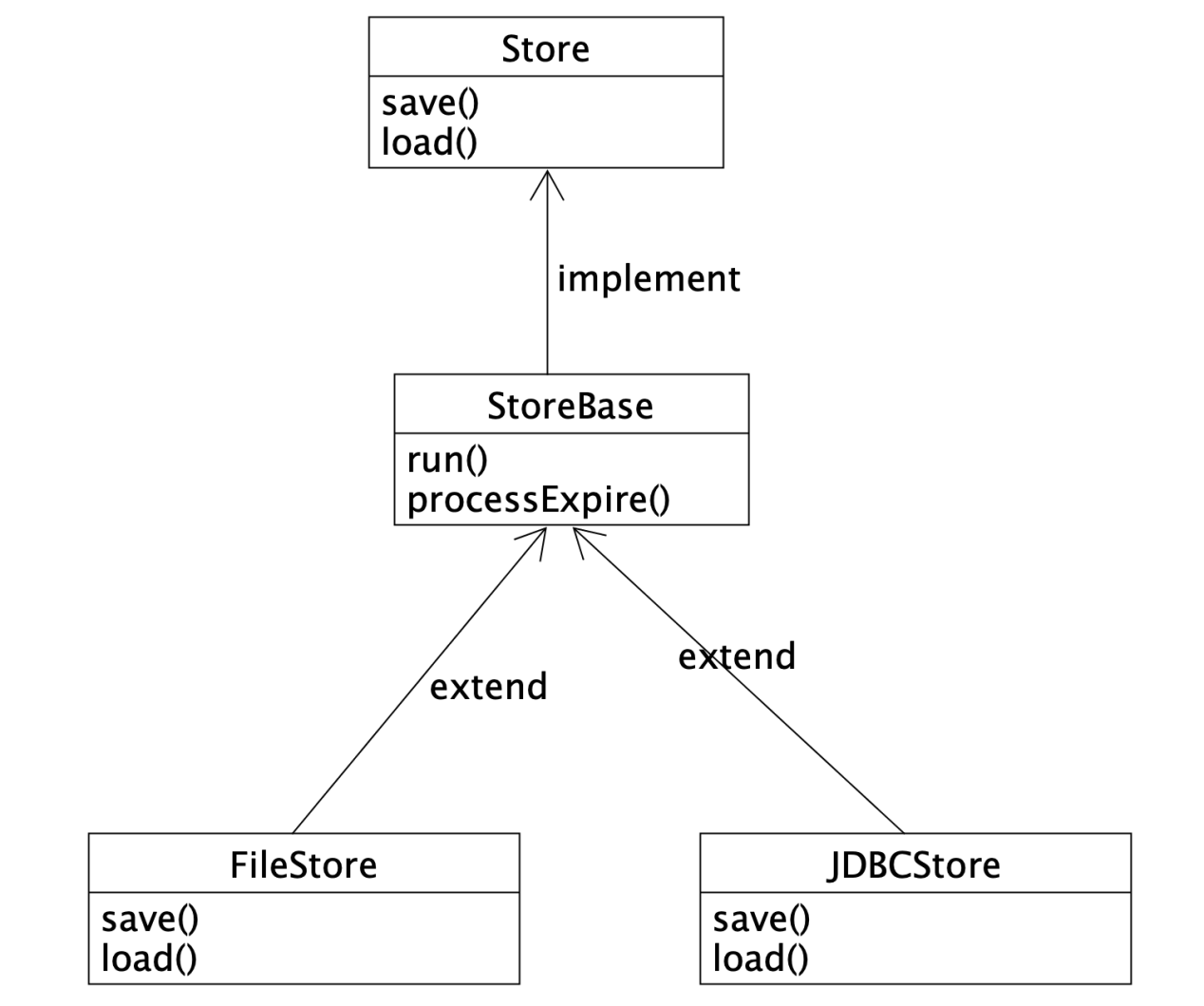


图3 Store架构

其中，Store.java是Store相关类的接口，定义了Store的各个规范。其中最重要的两个接口方法是save()/load()。

StoreBase是各个Store相关实现类的基类。主要做的事情，就是启动一个异步线程，定期检测由Store管理的持久化层中的session列表，将过期的session踢掉。防止持久化层中的session越来越多。这个清理过期session的逻辑，和之前在StandardManager中实现的类似。

对于这个Store体系，我们要搞清楚，Store是在哪里创建的？Store是在哪里启动的？

1. Store是在哪里创建的

我们可以在tomcat启动的时候，指定Manager对应的Store(参考BootStrap.java)，也可以在其他组件中初始化Manager的时候，指定对应的Store(参考StandardContext.start())。

1. Store是在哪里启动的

Store是在PersistentManagerBase启动的时候启动的，具体代码可以参考PersistentManagerBase.start()。

那么PersistentManagerBase又是在哪里启动的呢？是在StantdardContext.start()中启动的。

### StoreBase

继承Store接口，但是并不实现load()/save()方法。

StoreBase主要实现的方法为：：

run()

processExpire()

启动一个单独的线程，检测过期的session

### FileStore

FileStore和JDBCStore这两个具体的Store接口实现类，都继承了StoreBase。其中最关键的方法为：

save()

load()

remove()

keys()

其中FileStore将session对象持久化到文件系统，按照sessionid.session的格式存储。

JDBCStore将session持久化到数据库。

### JDBCStore

## 本章目标

代码梳理完了，session的大致实现思路也调研完毕。无非就是在启动BootStrap.java中选择一个Manager/Store的具体实现类，实现servlet的session功能。

本章最终要实现的目标，要达到的功能为：

1. servlet类能够能够存取session(完善ManagerBase)
2. 能够定期将过期的session回收(完善StandardManager)
3. 能够将session对象持久化到磁盘、数据库(完善PersistentManagerBase)
4. 能够实现将session对象持久化到tomcat集群(完善DistributedManager)
5. 能够实现将session对象持久化到redis(新增处理类：RedisStore)

## 测试

从这章开始，我们开始引入测试的概念。主要是对tomcat新增的功能进行验证。

### Session Test Client

Session Test Client。什么意思呢？就是我们这章的主题是Session，我们可以看到，除了Session的基本功能之外，这章有很大一部分内容，是提升Session的可用性。即在实际场景中，可能会有很多客户端访问tomcat服务端servlet，因此就会产生很多的session。针对这类实际场景，设计了很多功能，比如将过期sesssion回收、将session实例持久化到本地、持久化到数据库，在分布式场景中将Session同步到分布式各个节点，等等。

我们实践了这些功能之后，要验证这些功能。怎么验证呢？靠手工登录浏览器访问tomcat，创建Session，那是不现实的。 所以我们要搞一个测试程序，模拟客户端访问Tomcat服务端创建session(s)。

如果只是要模拟session创建，那实现还是比较方便的，只要通过http client调用我们的测试servlet(MySessionServlet)就行了。具体实现参考：

|  |
| --- |
| test.client.SessionClient.java |

但是要模拟真实用户操作就比较困难了，这个要好好想想。真实客户不仅会创建session，还会不定期地访问这个session，如果长期不访问session，还会造成session过期，然后重新登录。

### 测试案例

围绕Session，可以设计各种各样的测试案例:

|  |  |  |
| --- | --- | --- |
| index | Test case | Comment |
| 1 | session create | 创建session |
| 2 | session access | 客户端访问session |
| 3 | findSession |  |
| 4 | processExpire | session超时 |
| 5 | persistent | 将session持久化 |

从总体来说，要验证tomcat处理session的情况，主要是通过各类Session Manager打印异步线程处理的结果，比如：

|  |
| --- |
| 1.StandardManager 打印异步线程处理session的结果  StandardManager.printSessionStatus()  2.PersistentManagerBase 打印异步线程处理session的结果  PersistentManagerBase.printSessionStatus()  3.StoreBase 异步线程处理持久化层中session的结果  StoreBase.printStoreBaseSessionStatus() |

要打印这些session结果，要理清各种session保持的参数：

|  |
| --- |
| 这些参数主要保存在：  Constants.java  关键的参数包括：  MAX\_INACTIVE\_INTERVAL  MIN\_IDLE\_SWAP  MAX\_IDLE\_SWAP  MAX\_ACTIVE\_SESSIONS  MAX\_IDLE\_BACKUP  只有把这些参数调整好了，才能够有可能检测到session的状态。  为了保证测试到各个session处理的功能，我们将这些参数调整为：   1. **MAX\_INACTIVE\_INTERVAL**   session最长空闲时间，session一旦空闲时间超过了这个值，就设置为expire。如果session在内存中，就由Manager处理；如果session在内存中，由StoreBase处理。  在PersistentManagerBase处理逻辑中，如果这个值设置得比MAX\_IDLE\_SWAP大。那么，在Manager处理expire前，这个session必然已经被持久化了。那么应该是由StoreBase来处理expire。所以在PersistentManagerBase中，procssExpires()这个方法一般来说是没啥用了。  目前设置为**60s**   1. **MIN\_IDLE\_SWAP**   session至少空闲多少时间，session空闲时间至少要达到这个值，才能被Manager持久化。这个参数和MAX\_ACTIVE\_SESSIONS搭配使用.  由PersistentManagerBase.processMaxActiveSwaps()处理，对于那些达到一定空闲时间，且数量超过MAX\_ACTIVE\_SESSIONS的session，放到持久化层中去。  设置为10s   1. **MAX\_IDLE\_SWAP**   session空闲时间超过这个数值，就被放到持久化  设置为30   1. **MAX\_ACTIVE\_SESSIONS**   内存中最多保留多少session，一旦超过这个数值，就被Manager放到持久化层  设置为50   1. **MAX\_IDLE\_BACKUP**   内存中的session一旦超过这个数值，就备份到持久化  设置为20 |

搞清楚这些参数之后，我们在实际测试测试过程中，也碰到了一些问题。总体来说，测试的秘诀是，要先测试单个功能，再测整体功能。

比如我们要验证PersistentManagerBase.processMaxIdleSwaps()功能，就把其他所有的异步线程都注释掉，只测试这一个功能，看PersistentManager能否将内存中空闲时间过久的session swap out 到持久化层中去。

等这个功能测通后，再测试其他功能。当然，优先测试和processMaxIdleSwaps()相关的功能，比如StoreBase.processExpire()功能(将持久化层中空闲过久的session expire掉)。

这样，逐个功能测试通过之后，再把这些异步线程的功能串联起来做整体的测试。

### 自动测试

自动测试框架主要是采用我们之前设计的TestUI框架。当然也可以采用Maven的测试框架。

### 压力测试

对于Session这块功能来说，显然是要做压力测试的。因为我们知道，将session信息保存在tomcat服务端的做法，显然对服务端的压力是很大的，一旦客户端过多，会占用服务端过多的(内存)资源。

本章大部分的内容，包括PersistentManager、Store，都是关于如何保证在高并发、集群的模式下tomcat服务端状态正常的。

如何做压力测试呢？当然是模拟有很多客户端登录tomcat服务器，在服务端保存了很多很多的session对象信息。

大量的session占用了内存资源，我们如何来估算session占用多少内存呢？我们可以看一下StandardSession对象中有多少属性就能大致知道一个Session对象占用多少资源了。Session对象包括sid/createTime/updateTime/attribute等等。

## 思考

我们在这章中讨论了tomcat标准的session处理方式。但是要紧跟时代步伐，实践一下通过redis缓存处理session的方式。

## Tips

本章有哪些温馨提示呢？

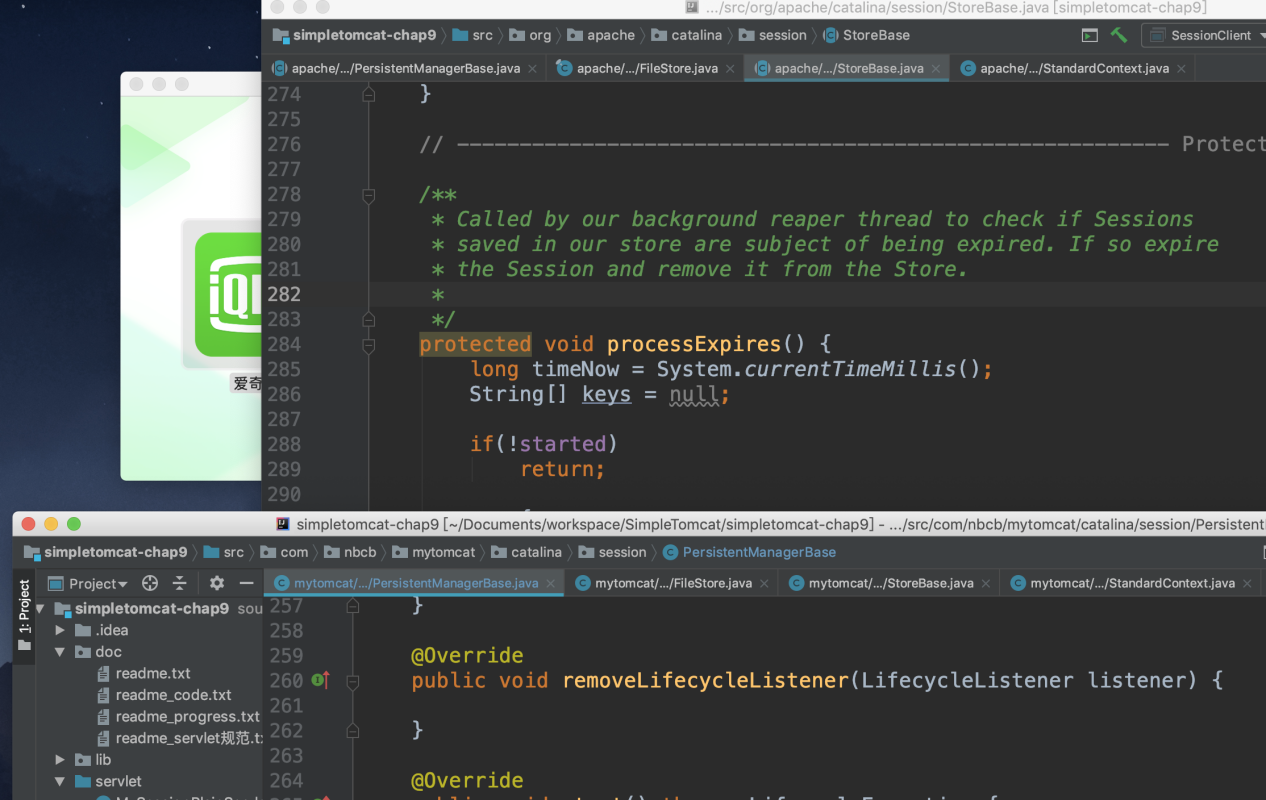
### 1.尽量不要修改catalina官方接口

尽量不要修改org.apache官方定义的接口，比如把某个接口方法的返回参数从Array改为List，这可能需要改动好多好多实现类的代码，得不偿失。

因为接口的本质，是定义了一套规范，规范了各个组件之间如何进行交互。对于接口中的方法，无论是参数还是返回值，都有严格的要求。一旦修改了接口，说明我们的规范有所改动。这对于那些接口的调用方、接口的实现类，都需要同步修改。

### 2.新开窗口比对代码

为了更好地参考org.apache官方实现，比如我们比对catalina官方的Session类(org.apache.catalina.session.StandardSession)和我们自己实现的StandardSession的代码，就可以把官方的代码拉出来形成单独的一个窗口：



## 3.git提交前要确认所有提交的目录

我们执行git commit 提交文件之前，要再三确认要提交的内容：

git status

不要把那些编译文件class、日志文件啥的都提交上去。

## 总结

我们对session这章内容做一个总结。总体来看，session解决了客户端以http协议访问servlet，如何保持状态的问题，是tomcat核心功能之一。但凡是那种需要客户端登录的web服务场景，都会用到session的功能。

一路学习下来收获还是蛮多的。我们了解了session运行原理，session生命周期，session管理方式，如何解决高并发场景下的session持久化方案。

但是，我们学习这章内容，不能能仅仅满足于实现session功能。还要从tomcat实现session功能的整体架构中，提炼出通用的session解决方案。并且有能力将这套架构移植到其他需要类似功能的系统中去。比如你开发数据库软件，那就不可避免要用到session的功能，因为针对每个访问数据库的请求，数据库端都会创建一个session来处理这个SQL请求。

# Chap10 Security

这章主要介绍安全相关的内容。

|  |
| --- |
| 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