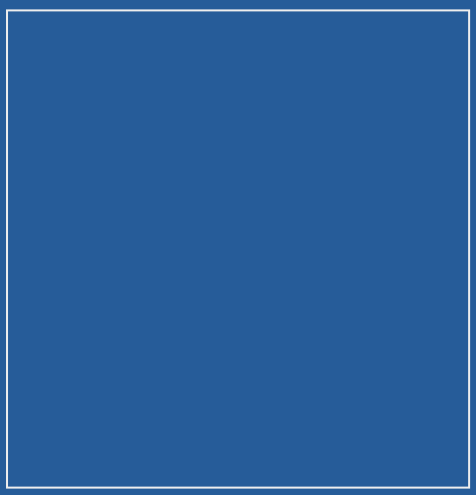


Tomcat源码研究

Tomcat源码研究

目 录

1. Tomcat 源码研究

1.1 Catalina.bat脚本解析

发表时间: 2009-11-06

当 startup 脚本完成环境变量的设置后，就开始调用 catalina.bat 脚本来启动 Tomcat 。 Catalina 脚本的主要任务是根据环境变量和不同的命令行参数，拼凑出完整的 java 命令行，调用 Tomcat 的主类 org.apache.catalina.startup.Bootstrap 来启动 Tomcat 。我们先不解析该脚本，而是写一个简单的测试脚本来调用这个程序，看看测试结果，从而理解该脚本的调用方法。

测试脚本如下：

```
rem 请将 JAVA_HOME 环境变量修改到您的 JDK 安装目录
set JAVA_HOME=C:\Program Files\Java\jdk1.5.0_09
rem 请将 CATALINA_HOME 环境变量修改到您的 Tomcat 安装目录
set CATALINA_HOME=C:\carl\it\tomcat_research\jakarta-tomcat-5.0.28
rem 开始调用 catalina.bat 文件
call %CATALINA_HOME%\bin\catalina.bat
```

我们把上面的脚本保存为 start_tomcat_nothing.bat 文件，然后在 MS-DOS 下执行，我们将看到如下的执行结果。

这个脚本并没有成功启动 Tomcat ，但是它给我们提供调用 catalina.bat 脚本的方法，请阅读上面窗口中的加亮部分。Catalina.bat 的调用方法为 catalina 后面加上具体命令参数，这个命令参数有以下 9 种。

具体解释如下：

debug	在调试器中启动 Tomcat
debug -security	带有安全管理器的调试器中，调用 catalina 脚本来启动 Tomcat
jpda start	调用 catalina 脚本，在 JPDA 调试器中启动 Tomcat
run	在当前窗口内调用 catalina 脚本来启动 Tomcat （不切换窗口）
run -security	带有安全管理的情况下，在当前窗口内调用 catalina 脚本来启动 Tomcat （不切换窗口）
start	开始一个新的 DOS 窗口，并在其中启动 Tomcat （切换至新窗口）
start -security	带有安全管理的情况下，在新的 DOS 窗口中启动 Tomcat （切换至新窗口）
stop	catalina 脚本执行停止命令来关闭 Tomcat

version	您使用的 Tomcat 版本

看完上面的解释，我们对 Tomcat 的启动参数有所了解。好，咱们写一个最简单的脚本来测试一下，拷贝刚才 start_tomcat_nothing.bat 脚本，将它重新命名为 start_tomcat_version.bat ，该脚本的内容和 start_tomcat_nothing.bat 脚本几乎一致，只是最后一行多加了一个 version 命令， start_tomcat_version.bat 脚本全部内容如下：

```
rem 请将 JAVA_HOME 环境变量修改到您的 JDK 安装目录
set JAVA_HOME=C:\Program Files\Java\jdk1.5.0_09
rem 请将 CATALINA_HOME 环境变量修改到您的 Tomcat 安装目录
set CATALINA_HOME=C:\carl\it\tomcat_research\jakarta-tomcat-5.0.28
rem 开始调用 catalina.bat 文件
call %CATALINA_HOME%\bin\catalina.bat version
```

我们可以在 DOS 下观察该脚本的执行结果。

该脚本顺利执行，执行的结果告诉我们当前 Tomcat 的版本号为 5.0.28 。有兴趣的读者朋友可以试试其它的 8 个命令参数，这些命令参数有时对我们非常有用。举例来说，有时候我们正常启动 Tomcat 时， Tomcat 弹出一个 DOS 窗口，但是瞬间消失，我们看不出到底哪里出了问题，也无任何启动日志可看。在这种情况下，我们可以使用 run 命令在同一个窗口内启动 Tomcat ，不让 Tomcat 弹出新的 DOS 窗口，好让我们看看 Tomcat 到底为什么没有启动。测试这个问题的简单方法如下。

首先，请到 Tomcat 安装目录下的 bin 子目录，把 Tomcat 的启动 jar 文件 bootstrap.jar 重命名为 bootstrap_1.jar ，然后点击 startup.bat 文件启动，我们会看到一个小黑窗口闪了一下，但是 Tomcat 并没有正常启动，这是因为 startup.bat 执行的是 Tomcat 的缺省命令 start ，该命令将在开始一个新的 DOS 窗口，并在其中启动 Tomcat 。在这种情况下，我们就要借重于 run 命令了，我们改以下我们上面的 start_tomcat_version.bat 脚本，将 version 命令改为 run 命令，然后另存为 start_tomcat_run.bat ，该脚本全部内容如下：

```
rem 请将 JAVA_HOME 环境变量修改到您的 JDK 安装目录
set JAVA_HOME=C:\Program Files\Java\jdk1.5.0_09
rem 请将 CATALINA_HOME 环境变量修改到您的 Tomcat 安装目录
set CATALINA_HOME=C:\carl\it\tomcat_research\jakarta-tomcat-5.0.28
rem 开始调用 catalina.bat 文件
call %CATALINA_HOME%\bin\catalina.bat run
```

脚本的执行结果如下：

请观察上面窗口中的加亮部分，这部分向我们清楚地展示 Tomcat 的启动错误，没有找到 Tomcat 的启动主类 Bootstrap 。这是因为我们人为地把 Tomcat 的启动 jar 文件包从 bootstrap.jar 重命名为 bootstrap_1.jar ，这个 bootstrap.jar 文件既然不存在，那包含在这个文件的 Bootstrap.class 文件当然也就找不到了。

现在，我们再回过头看看这个 catalina.bat 脚本。为简单起见，我们假定该脚本带缺省命令行参数 start ，看看 catalina 脚本的执行流程。如果您对 catalina 的 security 命令有兴趣，请参考 Sun 公司的文档 http://java.sun.com/j2se/1.5.0/docs/guide/security/smPortGuide.html ；如果您对 jpda 命令有兴趣，不妨浏览一下 http://java.sun.com/javase/technologies/core/toolsapis/jpda/ 文档。让我们打开 catalina 脚本，首先请注意这个脚本第二行有一个 setlocal 的命令，这个命令表明 catalina 中的环境变量只在本脚本中起作用，对其它程序和命令不起作用，这就意味着这个脚本中的环境变量是局部变量，不是全局变量，不会影响其它脚本和操作系统环境。

然后我们会看到长达 34 行的注释，这是优秀程序员必须学会的基本功之一。这些注释写得非常简洁明了，详细说明了各个环境变量的意义和用途。紧接着，如果发现 CATALINA_HOME 变量没有定义，该脚本试图设置该变量，这和 startup.bat 的第一节完全类似，在此不再赘述。然后该脚本调用 setclasspath.bat 到 JAVA_HOME 的 bin 目录下寻找 java.exe 、javaw.exe 、 jdb.exe 和 javac.exe 所在的路径，并把这些 exe 文件的文件名和路径赋值到相应的环境变量 _RUNJAVA 、_RUNJAVAW 、 _RUNJDB 和 _RUNJAVAC 中。再接下来， catalina 脚本判断是否定义有环境变量 CATALINA_BASE ， CATALINA_TMPDIR ，如果定义了它们，就执行相应的操作。因为我们在此并没有定义它们，所以执行不到这些操作。然后， catalina 脚本将在本窗口内打印出四个环境变量的值，这四个环境变量我们非常熟悉，一旦启动 Tomcat ，我们必定能看到 CATALINA_BASE ， CATALINA_HOME ， CATALINA_TMPDIR 和 JAVA_HOME 。然后 catalina 脚本根据其后跟的不同命令，拼凑出完整的 JAVA 命令行并执行。下面是该脚本的详细注释：

```
Rem 获得标准的环境变量，因为 setenv.bat 不存在，所以下面这两句不执行
rem Get standard environment variables
if exist "%CATALINA_HOME%\bin\setenv.bat" call "%CATALINA_HOME%\bin\setenv.bat"

Rem 调用 setclasspath.bat 脚本，获得标准的环境变量
rem Get standard Java environment variables
if exist "%CATALINA_HOME%\bin\setclasspath.bat" goto okSetclasspath
echo Cannot find %CATALINA_HOME%\bin\setclasspath.bat
echo This file is needed to run this program
goto end
:okSetclasspath
```

```
set BASEDIR=%CATALINA_HOME%
call "%CATALINA_HOME%\bin\setclasspath.bat"

rem 根据不同情况在 classpath 中加上不同的 jar 包
rem Add on extra jar files to CLASSPATH
if "%JSSE_HOME%" == "" goto noJsse
set CLASSPATH=%CLASSPATH%;%JSSE_HOME%\lib\jcert.jar;%JSSE_HOME%\lib\jnet.jar;%JSSE_HOME%\lib\jsse.jar

:noJsse

Rem 注意下面的这个 jar 文件是 tomcat 的启动包
set CLASSPATH=%CLASSPATH%;%CATALINA_HOME%\bin\bootstrap.jar

rem 我们没有定义 CATALINA_BASE ，下面这节不执行
if not "%CATALINA_BASE%" == "" goto gotBase
set CATALINA_BASE=%CATALINA_HOME%

:gotBase


rem 我们没有定义 CATALINA_ TMPDIR ，下面这节将忽略
if not "%CATALINA_TMPDIR%" == "" goto gotTmpdir
set CATALINA_TMPDIR=%CATALINA_BASE%\temp

:gotTmpdir

Rem 打印 4 个我们非常熟悉的环境变量
rem ----- Execute The Requested Command ---------------------------------------
echo Using CATALINA_BASE:   %CATALINA_BASE%
echo Using CATALINA_HOME:   %CATALINA_HOME%
echo Using CATALINA_TMPDIR: %CATALINA_TMPDIR%
echo Using JAVA_HOME:   %JAVA_HOME%

rem 定义一些环境变量
set _EXECJAVA=%_RUNJAVA%
rem 这是 Tomcat 启动的主类
set MAINCLASS=org.apache.catalina.startup.Bootstrap
```

```
rem 这是 catalina 脚本的缺省命令 start
set ACTION=start
set SECURITY_POLICY_FILE=

set DEBUG_OPTS=
set JPDA=

rem 我们的第一个参数命令是 start ，下面这节将忽略不执行
if not ""%1"" == ""jpda"" goto noJpda
set JPDA=jpda

if not "%JPDA_TRANSPORT%" == "" goto gotJpdaTransport

set JPDA_TRANSPORT=dt_shmem

:gotJpdaTransport

if not "%JPDA_ADDRESS%" == "" goto gotJpdaAddress
set JPDA_ADDRESS=jdbconn
:gotJpdaAddress
shift

:noJpda

rem 我们的第一个参数命令是 start ，所以程序将走到 doStart 标签处执行
if ""%1"" == ""debug"" goto doDebug
if ""%1"" == ""run"" goto doRun
if ""%1"" == ""start"" goto doStart
if ""%1"" == ""stop"" goto doStop
if ""%1"" == ""version"" goto doVersion

rem 如果 catalina 后跟的命令不是 debug, run, start, stop 或 version ，
rem 该脚本将打印出 catalina 的用法，然后结束。
Rem 详情请见我们前面的测试脚本 start_tomcat_nothing.bat 及相应的结果窗口
echo Usage: catalina ( commands ... )
echo commands:
echo   debug   Start Catalina in a debugger
```

```
echo   debug -security   Debug Catalina with a security manager
echo   jpda start   Start Catalina under JPDA debugger
echo   run   Start Catalina in the current window
echo   run -security   Start in the current window with security manager
echo   start   Start Catalina in a separate window
echo   start -security   Start in a separate window with security manager
echo   stop   Stop Catalina
echo   version   What version of tomcat are you running?
goto end

rem 执行 debug 命令，我们的第一个参数命令是 start ，下面这节将忽略不执行
:doDebug
shift
set _EXECJAVA=%_RUNJDB%
set DEBUG_OPTS=-sourcepath "%CATALINA_HOME%\..\..\jakarta-tomcat-catalina\catalina\src\share"

if not ""%1"" == ""-security"" goto execCmd
shift
echo Using Security Manager
set SECURITY_POLICY_FILE=%CATALINA_BASE%\conf\catalina.policy
goto execCmd

:doRun
shift
if not ""%1"" == ""-security"" goto execCmd
shift
echo Using Security Manager
set SECURITY_POLICY_FILE=%CATALINA_BASE%\conf\catalina.policy
goto execCmd

rem 执行 start 命令
:doStart
Rem 将参数列表指针指向下一个参数
shift
rem 设置 Tomcat 启动窗口的标题，缺省值为 Tomcat
if not "%OS%" == "Windows_NT" goto noTitle
set _EXECJAVA=start "Tomcat" %_RUNJAVA%
```

```
goto gotTitle
:noTitle
set _EXECJAVA=start %_RUNJAVA%
:gotTitle


Rem 检查第二个命令参数是否为 -security ，我们没有第二个命令参数，脚本将执行至 execCmd 标签处
if not ""%1"" == ""-security"" goto execCmd
shift

echo Using Security Manager
set SECURITY_POLICY_FILE=%CATALINA_BASE%\conf\catalina.policy
goto execCmd

rem 执行 stop 命令，我们的第一个参数命令是 start ，下面这节将忽略不执行
:doStop
shift
set ACTION=stop
goto execCmd

rem 执行 version 命令，我们的第一个参数命令是 start ，下面这节将忽略不执行
:doVersion
%_EXECJAVA% -classpath "%CATALINA_HOME%\server\lib\catalina.jar" org.apache.catalina.util.ServerInfo
goto end


rem 执行命令，首先看看命令行参数是否不只一个，我们本例只有一个 start 参数，所以下面这节将不执行

:execCmd
rem Get remaining unshifted command line arguments and save them in the
set CMD_LINE_ARGS=
:setArgs
if ""%1""=="""" goto doneSetArgs
set CMD_LINE_ARGS=%CMD_LINE_ARGS% %1
shift

goto setArgs
```

```
:doneSetArgs

Rem 执行 java 命令行
rem Execute Java with the applicable properties
rem 检测 JPDA 和 Security 变量，我们本例没有定义他们，下面两句将忽略
if not "%JPDA%" == "" goto doJpda
if not "%SECURITY_POLICY_FILE%" == "" goto doSecurity

rem 程序将执行下面的 java 命令，在新窗口内启动 Tomcat
%_EXECJAVA% %JAVA_OPTS% %CATALINA_OPTS% %DEBUG_OPTS% -classpath "%CLASSPATH%" -
Dcatalina.base="%CATALINA_BASE%" -Dcatalina.home="%CATALINA_HOME%" -
Djava.io.tmpdir="%CATALINA_TMPDIR%" %MAINCLASS% %CMD_LINE_ARGS% %ACTION%

Rem catalina 至此结束，下面代码我们将忽略。
goto end
:end
```

1.2 Tomcat启动遇到的常见问题

发表时间: 2009-11-06

如果一切顺利，我们就可以看到非常熟悉的 Tomcat 窗口 。但是，由于各种原因，我们可能会碰到一些问题， 下面 就具体分析一下 Tomcat 不能正常启动的原因 。

3.1 环境变量设置问题

如果机器上没有安装 JDK 或者环境变量 JAVA_HOME 或 CATALINA_HOME 没有设置正确， Tomcat 就不能正常启动。

3.1.1 下面 这个错误对话框的意思是 Windows 不能发现 “-Djava.endorsed.dirs=” 命令，下面背景的黑窗口的错误是 JAVA_HOME 环境变量应该指向 JDK 而不是 JRE 的根目录（本 例 故意把把 JAVA_HOME 指向 JRE 的根目录，所以产生这个错误）。 在这种情况下，我们在可以重新设置环境变量 JAVA_HOME 并指向 JDK 的安装目录即可。

3.1.2 下面这个错误是因为 CATALINA_HOME 环境变量设置不正确，从而造成 Tomcat 不能正常启动。

```
C:\carl>startup
CATALINA_HOME C:\
The CATALINA_HOME environment variable is not defined correctly
This environment variable is needed to run this program
```

3.1.3 下面这个错误是由于错误地设置了 Java 启动参数 xms256M ，正确的写法应该是 -Xms256M ，请注意大小写。大家知道，在 Java 命令行中， -Xms 表示 JVM 开始启动时所分配的内存大小，而 -Xmx 表示 JVM 运行时最大所能占用的的内存大小，如果您的应用程序所需的内存超过 Xmx 的值， JVM 就会抛出 Out of Memeory(内存不足) 的异常而停止。当然， Xmx 的最大值不应超过物理内存的 70% 。

```
D:\back\blog>start_tomcat_debug.bat
D:\back\blog>set JAVA_OPTS=-xms256M
D:\back\blog>rem 请将 JAVA_HOME 环境变量修改到您的 JDK 安装目录
D:\back\blog>set JAVA_HOME=C:\Program Files\Java\jdk1.5.0_09
D:\back\blog>rem 请将 CATALINA_HOME 环境变量修改到您的 Tomcat 安装目录
D:\back\blog>set CATALINA_HOME=C:\carl\it\tomcat_research\jakarta-tomcat-5.0.28
D:\back\blog>rem 开始调用 catalina.bat 文件
D:\back\blog>call C:\carl\it\tomcat_research\jakarta-tomcat-5.0.28\bin\catalina. bat debug
Using CATALINA_BASE:   C:\carl\it\tomcat_research\jakarta-tomcat-5.0.28
```

Using CATALINA_HOME: C:\carl\it\tomcat_research\jakarta-tomcat-5.0.28
Using CATALINA_TMPDIR: C:\carl\it\tomcat_research\jakarta-tomcat-5.0.28\temp
Using JAVA_HOME: C:\Program Files\Java\jdk1.5.0_09
debug
无效的选项： -xms256M

用法： jdb < 选项 > < 类 > < 参数 > ...

如果遇到类似问题，可以用下面的方法重新设置一下 JVM 启动时的内存参数，在 startup.bat 脚本中加上下面这一行：

set JAVA_OPTS=-Xms256m -Xmx512m

上面这行表示 Tomcat 初始启动内存至少需要 256M ，而最大可能占用内存为为 512M 。有兴趣的读者可以在 startup.bat 里加上这行参数，启动 Tomcat ，您会从 Windows 任务管理器中看到您使用的内存迅速飙升。

3.2 Tomcat 应用服务器的配置问题及其中部署的某个 Web 应用问题

3.2.1 Tomcat 端口冲突，具体错误如下窗口所示。这表明缺省的 8080 端口已经被占有，所以 Tomcat 不能启动，我们可以通过修改 Tomcat 的配置文件 server.xml 来重新定义端口号即可启动，或者停止已经使用 8080 端口的程序。有时我们并没有启动 Tomcat ，也没有其它应用程序占用 8080 端口，但是 Tomcat 还抛出这个错误。这可能是因为上次我们关闭 Tomcat 时，并没有真正关闭 Tomcat 。在这种情况下，请从 Windows 任务管理器中检查一下有无一个 java.exe 的程序正在运行，如果发现，强行 kill java.exe ，然后重启试试。

3.2.2 Tomcat 配置文件有错误，产生异常。有时我们在修改 server.xml 文件时，不小心多加或者多删了一个 </>标签 ，可能造成 Tomcat 不能正常启动。下面的窗口说明了这个问题。

3.2.3 Tomcat 中部署的某个 Web 应用发生异常。这时 Tomcat 能正常启动，但在 Tomcat 启动窗口里发现一大堆异常，如 data source 没有正确定义，程序抛出异常等等都有关系。下面的这个例子是 Tomcat 自带的 balancer web 应用程序启动时产生错误。该问题是因为我们错误的修改了 balancer.xml 文件造成的。

3.2.4 Web 应用程序的 jar 文件丢失，如 JDBC 数据库连接文件 class12.jar, mysql.jar 没有拷贝到 Webapp 的 WEB-INF\lib 目录下就会产生异常。尤其要注意的是在 Tomcat5 以前的版本中，公共 jar 文件包通常放在 Tomcat 安装目录下的 common\lib 子目录中，但在 Tomcat6.0 中，这个公共目录改为 lib 子目录。

3.3 未知错误

3.3.1 有时 Tomcat 的启动窗口一闪而过，根本就看不出启动过程中发生了什么错误。 这中间的原因有好多种，最常见的解决办法就是使用 run 命令，打开 startup.bat 文件，找到下面这行：

call "%EXECUTABLE%" start %CMD_LINE_ARGS%

并将它修改为：

call "%EXECUTABLE%" run %CMD_LINE_ARGS%

这样， Tomcat 启动时就不会弹出新窗口，我们就可以从容不迫地观察 Tomcat 的启动错误，并解决问题，请参考上面关于 run 命令的解释。

3.3.2 另外，阅读 Tomcat 的启动日志文件也是我们解决问题的重要办法，缺省的 Tomcat 日志是放在 Tomcat 安装目录的 logs 子目录下。例如下面这段日志说明 JspServletViewer 这个 Web 应用缺少 Map Object 的相关 jar 文件包，从而找不到 com/esri/mo2/map/core/Layout 这个 Java 类。

2008-02-23 11:19:30 StandardContext[/JspServletViewer]Exception sending context initialized event to listener instance of class com.esri.svr.cmn.FileRewriterContextListener
java.lang.NoClassDefFoundError: com/esri/mo2/map/core/Layout
at com.esri.svr.cat.ServiceXMLHandler.startElement ServiceXMLHandler.java:47)
at org.apache.xerces.parsers.AbstractSAXParser.startElement(Unknown Source)
at org.apache.xerces.impl.XMLDocumentFragmentScannerImpl.scanStartElement(Unknown Source)

1.3 Tomat6架构探讨

发表时间: 2009-11-06

下面，我们重点针对 Catalina 子模块，熟悉Tomcat的几个关键组件。

(1) 服务器 (Server)

在 Tomcat 中，服务器代表整个 J2EE 容器，所有的服务及服务上下文均包含在服务器内。我们打开 Tomcat 源代码，可以看到 org.apache.catalina.Server 这个接口，其中比较重要的方法有 initialize(负责 Tomcat 启动前的初始化工作)，还有一些服务 (Services) 管理方法，比如 removeService() 、 addService() 、 findService() 之类的方法。

在 Tomcat 运行时，我们永远只有一个 Server 实例，这不由让我们联想到单例模式 (Singleton Pattern) 。不错，在 Tomcat 中， Server 的实例化工作正是由一个叫 ServerFactory 工厂类完成的，这个工厂类实现了单例设计模式。

比较有意思的是，这个工厂类的产品创建方法名为 getServer() 而不是标准的 createServer() 方法，并且没有加 synchronized 或 synchronized(this) 保护，这是为什么呢？我们知道，在应用单例模式时，需要注意的一个关键点就是多线程的调用问题，如果我们的工厂类在创建单例对象时，这个工厂类有可能被多个线程并发调用的话，那么最好给这个工厂方法加上 synchronized 以避免产生两个不同的产品类实例。如果您想避免 synchronized 的锁机制造成的性能损失，请使用双重检查机制 (double-checked locking) 。所以，如果考虑多线程，这个工厂类的 getServer() 方法应该写成：(红色代码是作者另加上的，源代码中没有)。

```
/**

* Return the singleton <code>Server</code> instance for this JVM.

*/

public static Server getServer() {

if( server==null ){
synchronized (ServerFactory.class) {
if(server==null){
  server=new StandardServer();
}
}
}
return (server);
```

}

为什么 Tomcat 在实现时没有加上面的红色代码呢？这是因为， Tomcat 启动时创建 Server 对象，不可能出现多线程情况，所以就免掉了双重检查。如果我们确信没有多线程调用我们的单例工厂类，我们也可以这样做。

另外，如果您对 ServerFactory 进行调试，您会发现一个非常有趣的现象，这个工厂先执行的不是 create 方法（此处为 getServer 方法），而是 setServer 方法。这意味着这个工厂方法其实并不生产实际产品，实际产品是从别处产生，然后通过 setServer 方法注册到这个 Factory 。当下次有客户请求产品时，这个工厂方法只是简单的把现成的单例产品传给客户。所以这个类其实只需一个单例类足矣，根本没有必要使用工厂模式，所以 Tomcat 的开发者也觉得不好意思使用标准的工厂方法 createProduct ，杀鸡焉用宰牛刀，对吗？

在 Tomcat6.0 中，服务器 (Server) 接口的实现类只有一个，那就是 org.apache.catalina.core. StandardServer 类。这是一个标准的服务器实现类，这个类不但实现了 Server 接口，而且还实现了 Lifecycle 和 MBeanRegistration 接口， Lifecycle 主要提供了服务器的生命周期管理功能，比如说启动、停止等方法，而 MBeanRegistration 接口是为了将 server 注册到 MBean 服务器，以便在 Tomat 运行时，我们能通过 JMX 来管理服务器。

从 Tomcat5.0 开始， Tomcat 的开发人员在 JMX 管理上着实下了一番功夫，争取做到让 Tomcat 具有 JBoss 那样非常强大的管理功能。

(2) 服务 (Service)

在上述的标准服务器 (StanderServer.java) 实现代码中，我们可以看到其中有一个 services 的数组，这个数组就是用来存储服务 (Service) 的。所以，我们可以这样理解，一个服务器可能有一至多个服务组成。所谓服务，就是包含一至多个连接器的组件，能够对用户请求作出响应的组件。打开 org.apache.catalina.Service.java 的源代码，我们可以看到其中含有一个连接器数组 (Connector[]) ，这表明一个 Service 有可能包含一个到多个连接器。但所有这些连接器都属于一个引擎 (Engine 或 Container) 。在 Tomcat6 中， org.apache.catalina.Service 接口由 org.apache.catalina.core. StandardService 类来实现的。

(3) 引擎 (Engine)

对一个具体的服务 (service) 来说，引擎是一个用户请求的处理管道，这个管道很特别，因为它只处理 Servlet 请求，在 Tomcat 中，引擎其实就是指 Servlet 引擎。引擎从这些连接器那里接收到 Servlet 请求，然后处理它们，并将响应的结果传回到适当的连接器，从而将响应发送到客户端。简单地说，引擎的功能就是如何处理用户的 Servlet 请求。

org.apache.catalina.Engine 这个接口继承自 org.apache.catalina.Container ，说明引擎是一种特殊的 Container ，是一种专门用来处理 servlet 请求的容器。

(4) 主机 (Host)

对 Tomcat 服务器来说，主机是 Tomcats 所在机器的网络名（域名）。一个引擎可能包含多个主机，主机支持网络别名。例如，用户通过配置 config.xml 里面的主机 (Host) 元素，让 www.abc.com 和 abc.com 指向同一台 Tomcat 应用服务器。

(5) 连接器 (Connector)

在 Tomcat 中，连接器负责和客户端进行请求响应的交流。 Tomcat 中有两种连接器 (Coyote 和 JK 连接器）， Coyote 连接器实现了 Http1.1 协议，我们可以将它理解为 Tomcat 的 Web 服务器部分。 JK 连接器负责处理来自第三方 Web 服务器的请求，并将请求结果发送给第三方 Web 服务器。针对 Apache Httpd Web 服务器， JK 连接器实现了 AJP 协议。

在 Tomcat6.0 中，实现 Coyote 连接器的类是 org.apache.catalina.connector.Connector 。

(6) 上下文 (Context)

上下文代表某一具体的 Web 应用，一个主机可包含多个 Web 应用，所以可有多个 Web 应用上下文，不同的上下文可用不同路径来表示。上下文里含有一些关于该 Web 应用的一些具体信息，比如欢迎页面的文件名， web.xml 文件的位置等等信息。

上下文在 Tomcat 的源码中对应 org.apache.catalina.Context 接口，其具体实现为 org.apache.catalina.core.StandardContext 。

至此为止，我们熟悉了 Tomcat 架构中一些重要组件。下面我们用 UML 类图 (Class Diagram) 来总结一下。

在上面的类图中，我们先撇开Tomcat 组件不谈，首先给我们印象最深刻的一点是：针对接口编程，而非针对具体实现编程 (Program to interface, not implementation) 。人家老外这点确实值得我们学习。上面的类图中，共有 7 个类，其余均为接口，这些类无一例外地调用了接口，而非具体的实现类。 ServerFactory 调用了 Server 接口，而非 StandServer 的实现类； Connector 类调用了 Service 接口和 Container 接口，而没有调用它们的实现类； StandardService 类调用了 Container 接口和 Server 接口，也同样没有调用它们的实现类。所以我们在编程时，也要贯彻这条原则。

在 <<Head First Design Patterns>> 一书里，作者举了个非常生动的例子，请看下面三段代码：

a) 代码片段一

```
Dog d=new Dog();
d.bark();
```

b) 代码片段二

```
Animal animal=new Dog();
```

```
animal.makeSound();
```

c) 代码片段三

```
Animal animal = getAnimal();
animal.makeSound();
```

作者详细解释了上面第三段代码为什么是最好的，而第二段又为什么比第一段好的道理。东扯西拉这么多，现在我们切入正题。

从上面的类图中，我们可以非常清晰地理解 Tomcat 的总体架构：

a) Server(服务器) 是 Tomcat 构成的顶级构成元素，所有一切均包含在 Server 中， Server 的实现类 StandardServer 可以包含一个到多个 Services ；

b) 次顶级元素 Service 的实现类为 StandardService 调用了容器 (Container) 接口，其实是调用了 Servlet Engine(引擎) ，而且 StandardService 类中也指明了该 Service 归属的 Server ；

c) 接下来次级的构成元素就是容器 (Container) ，主机 (Host) 、上下文 (Context) 和引擎 (Engine) 均继承自 Container 接口，所以它们都是容器。但是，它们是有父子关系的，在主机 (Host) 、上下文 (Context) 和引擎 (Engine) 这三类容器中，引擎是顶级容器，直接包含是主机容器，而主机容器又包含上下文容器，所以引擎、主机和上下文从大小上来说又构成父子关系，虽然它们都继承自 Container 接口。

d) 连接器 (Connector) 没有接口 (这可是违反了面向接口编程的原则哟！) ，它直接实现了 Http1.1 协议。连接器将 Service 和 Container 连接起来，首先它需要注册到一个 Service ，它的作用就是把来自客户端的请求转发到 Container(容器) ，这就是它为什么称作连接器的原因。

下面我们来小结一下， Tomcat 的架构从功能的角度，可以分成 5 个子模块，它们分别是 Connector 子模块， Jsper 子模块， Servlet 子模块， Catalina 子模块和 Resource 子模块，每个子模块负责一定的功能；从组件的角度，我们可以看到 Tomcat 中至少有 7 个关键组件，它们 Server 组件、 Service 组件、 Container 组件、 Connector 组件及继承自 Container 组件的 Host 组件、 Engine 组件和 Container 组件，从 UML Class Diagram 中，我们可以非常明确地理解它们的包容关系。到此为止，希望我们能对 Tomcat 的架构有一个比较清晰的认识。

1.4 Tomat6的整体架构

发表时间: 2009-11-06

在上篇文章中，我们已经成功将Tomcat6.0 的源代码导入到 Eclipse IDE 中。现在我们就开始学习 Tomcat 源码。 Tomcat 源代码共有 1000 多个 java 类，代码行数大约 28 万到 30 万行左右。从项目规模上说，可算得上是一个中型项目。要学习理解 Tomcat 源代码，我们有多种办法可行。最原始的一种办法就是，打开 Debugger ，逐行跟踪，看看 Tomcat 如何启动，如何处理客户端请求，如何编译动态 jsp 页面。第二种办法是利用逆向工程，把 Tomcat 的总体类图先描绘出来，然后再结合 sequence diagram ，来学习理解它。我们在这里采取从顶到底的阅读方法，先了解整体架构，然后逐步细化。所谓“纲举目张” ，说的就是这个道理。

首先，我们可以从功能的角度将 Tomcat 源代码分成 5 个子模块，它们分别是：

1) Jsper 子模块

这个子模块负责 jsp 页面的解析， jsp 属性的验证，同时也负责将 jsp 页面动态转换为 java 代码并编译成 class 文件。在 Tomcat 源代码中，凡是属于 org.apache.jasper 包及其子包中的源代码都属于这个子模块；

2) Servlet 和 Jsp 规范的实现模块

这个子模块的源代码属于 javax.servlet 包及其子包，如我们非常熟悉的 javax.servlet.Servlet 接口、javax.servet.http.HttpServlet 类及 javax.servlet.jsp.HttpJspPage 就位于这个子模块中；

3) Catalina 子模块

这个子模块包含了所有以 org.apache.catalina 开头的 java 源代码。该子模块的任务是规范了 Tomcat 的总体架构，定义了 Server 、 Service 、 Host 、 Connector 、 Context 、 Session 及 Cluster 等关键组件及这些组件的实现，这个子模块大量运用了 Composite 设计模式。同时也规范了 Catalina 的启动及停止等事件的执行流程。从代码阅读的角度看，这个子模块是我们阅读和学习的重点。

4) Connectors 子模块

如果说上面三个子模块实现了 Tomcat 应用服务器的话，那么这个子模块就是 Web 服务器的实现。所谓连接器 (Connector) 就是一个连接客户和应用服务器的桥梁，它接收用户的请求，并把用户请求包装成标准的 Http 请求（包含协议名称，请求头

Head ，请求方法是 Get 还是 Post 等等）。同时，这个子模块还按照标准的 Http 协议，负责给客户端发送响应页面，比如在请求页面未发现时，connector就会给客户端浏览器发送标准的 Http 404 错误响应页面。

Tomcat 实现了两类连接器，除了上述实现了 Http1.1 协议的 Coyote 连接器外，还有一种 JK 连接器，JK 连接器是将Tomcat 和第三方 Web 服务器（如 Apache 或 IIS Web 服务器）连接起来， Tomcat 此时充当应用服务器的角色，负责处理和解释 Jsp 及 Servlet 请求。

Coyote 连接器的源代码位于以 org.apache.coyote 开头的包中， JK 连接器的代码位于以 org.apache.jk 开头的包中。

另外， Tomcat 虽然实现了 Web 服务器的功能，但是其实现不是非常完美，效率不高，所以在生产环境中，我们通常要将 Tomcat 和 Apache Web Server 配合使用，尽量利用它们各自的优势。

5) Resource 子模块

这个子模块包含一些资源文件，如 Server.xml 及 Web.xml 配置文件。严格说来，这个子模块不包含 java 源代码，但是它还是 Tomcat 编译运行所必需的。

上面我们从模块组件的角度，简单介绍了 Tomcat 的子模块划分及其相应的功能。下面我们简单以图示意之。

从上面的Tomcat 子模块示意图中，我们可以看到，来自客户端的请求首先由 Connector 子模块进行处理，然后根据情况或者发送到第三方的 Web 服务器，或者转发到 Jsper 模块进行处理，或者转发到 Jsp/Servlet 子模块处理。总体说来，Tomcat 通过下面三种方式处理来自客户端的请求：

(1) 如果客户端发出静态页面请求，如果没有配置第三方 Web 服务器，此时客户端的请求直接交由 Coyote Connector 子模块处理，然后返回结果；如果配有第三方应用服务器，那么客户的请求直接由第三方应用服务器响应，然后返回静态记过页面。客户端请求的执行过程如图中绿线所示。

(2) 如果客户端请求 Jsp 页面，该请求首先转发到发送 Coyote 连接器（在没有配置第三方 Web 服务器的情况下），或者经过第三方 Web 服务器将客户请求转发到 JK 连接器；然后该 Jsp 请求将交给 Jsper 子模块处理， Jsper 将根据情况验证编译

该 Jsp 页面，最后由 Jsp/Servlet 模块对客户请求进行处理。 Jsp 请求处理完毕，服务器首先把响应结果发送给连接器子模块，连接器子模块根据情况或将响应结果页面发送到第三方 Web 服务器，或者直接发送响应结果页面到客户端。

(3) 如果客户请求 Servlet ， Tomcat 的处理流程和 Jsp 页面的请求执行流程基本类似，只不过少了一个 Jsper 子模块处理罢了。

1.5 JMX在Tomcat中的应用（一）

发表时间: 2009-11-06

一、 JMX 简单介绍

Tomcat 从 5.0 版本开始引入 JMX ，力图使 JMX 成为 Tomcat 未来版本的管理工具和平台。首先，让我们来对 JMX 做一个简单了解。 JMX 是 Java Management Extension 的缩写，可译为 Java 管理工具扩展，扩展的意思就是 JMX 不包含在标准的 J2SE 中，我们必须要另外下载 JMX RI 的实现。不过，这种把 JMX 排除在 J2SE 之外的情况已经成为历史了， J2SE5.0 和 J2SE6.0 都已经包含了 JMX 的标准实现。这说明， JMX 已经成为 J2SE 不可分割的一部分，另外一方面， JMX 已经成为 Java 平台上管理工具的事实标准，在业界广泛使用。例如， JBOSS 就是以 JMX 为微内核， Web 应用模块和其它功能模块都可热插拔到这个微内核，将 JMX 的管理功能发挥得淋漓尽致。从当前业界使用情况看， JMX 中的 X(Extension ，扩展）应该去掉，改名为 Java Management Standard Platform (JMSP ， Java 管理标准平台）更加合适。为了向下兼容，我们姑且还是称之为 JMX 吧。

JMX 要管理的对象是什么呢，是资源。什么是资源，资源是指企业中的的各种应用软件和平台，举例来说，一个公司内部可能有许多应用服务器、若干 Web 服务器、一台至多台的数据库服务器及文件服务器等等，那么，如果我们想监视数据库服务器的内存使用情况，或者我们想更改应用服务器上 JDBC 最大连接池的数目，但我们又不想重启数据库和应用服务器，这就是典型意义上的资源管理，即对我们的资源进行监视 (Monitoring ，查看) 和管理 (Management ，更改 ），这种监视和更改不妨碍当前资源的正常运行。对资源进行适当的监测和管理，可以让我们的 IT 资源尽可能的平稳运行，可以为我们的客户提供真正意思上的 24 × 7 服务。在资源耗尽或者在硬件出故障之前，我们就可以通过管理工具监测到，并通过管理工具进行热调整和插拔。独孤九剑，料敌机先，适当的资源管理就是我们料敌机先的工具，可以让我们立于 IT 服务的不败之地。在 Sun 公司提出 JMX(JSR174) 以前，人们通常都是使用 SNMP 对网络上的资源进行管理。 SNMP 的主要问题是入门门槛太高，不容易使用。所以 Sun 提出了 JSR174 倡议并且提供了一套 JMX 的参考实现。

从技术上说， JMX 整体架构可分为三层，即资源植入层 (Instrumentation Level ，可能有更好的译法？）、代理层 (Agent Level) 和管理层 (Manager Level) ，简述如下：

资源植入层 (Instrumentation Level) ：该层包含 MBeans 及这些 MBeans 所管理的资源， MBeans 是一个 Java 对象，它必须实现 JMX 规范中规定的接口。按照 JMX 规范，在 MBeans 对象的接口中，我们可以指定管理层可以访问资源的哪些属性，可以调用资源的哪些方法，并且，在资源的属性发生变化是，我们的 MBeans 可以发出消息，通知对这些属性变化感兴趣的其它对象。 JMX 规范定义了四种 MBeans ，它们分别是标准 MBeans(Standard MBeans) 、动态 MBeans(Dynamic MBeans) 、开放 MBeans(Open MBeans) 和模态 MBeans(Model MBeans) 。

代理层 (Agent Level) ：代理层的目的就是要把 MBeans 中实现的接口暴露给管理层，该层通常由 MBean Server

和 Agent Services 构成， MBean Server 就是一个 MBeans 对象注册器，所有的资源 MBeans 都注册到这个 MBean Server ，对象管理器或者其它的管理层应用程序可以通过访问 MBean Server ，从而可以访问 MBean Server 中注册的 MBeans ，当然也就可以监视和管理和这些 MBeans 绑定的资源。

管理层 (Manager Level) ：又称之为分布式服务层 (Distributed Services) ，顾名思义，该层主要包含一些管理应用程序，这些程序可以访问和操作 JMX 代理层 (Agent Level) 。这些管理应用程序可以是一个 Web 应用，也可能是一个 Java SWT 应用程序。

1.6 JMX在Tomcat中的应用（二）

发表时间: 2009-11-06

下面，我们举一个简单的例子，理解一下 JMX 中中的各个概念。我们家有一个中央热水系统 (Central Heater System) ，它是我们家的一个资源，现在我们想通过 JMI 进行管理。现有的代码如下所示，当然，为简单起见，我们略去了一些 JNI 调用代码，因为厂家提供的 API 是用 C 语言写的。

a) 热水器接口（CentralHeaterInf .java）的现有代码：

```
package carl.test.jmx;

/**
* The interface of Central Heater
* @author carlwu
*
*/
public interface CentralHeaterInf {

/**
* The heater is provided by British Gas Company
*/
public final static String HEATER_PROVIDER = "British Gas Company" ;

/**
* Get current temperature of heater
* @return the temperature of the heater
*/
public int getCurrentTemperature();


/**
* Set the new temperature
* @param newTemperature
*/
public void setCurrentTemperature( int newTemperature);
```

```
/**
* Turn on the heater
*/
public void turnOn();


/**
* Turn off the heater
*/
public void turnOff();

}
```

b) 热水器实现代码的现有代码 (CentralHeaterImpl .java)

```
/**
* The implemenation of Central Heater
* @author carlwu
*
*/
package carl.test.jmx;

public class CentralHeaterImpl implements CentralHeaterInf {

int currentTemperature ;
public int getCurrentTemperature() {
  return currentTemperature ;
}

public void setCurrentTemperature( int newTemperature) {
  currentTemperature =newTemperature;
}

public void turnOff() {
  System. out .println( "The heater is off. " );
```

```
}

public void turnOn() {
  System. out .println( "The heater is on. " );
}

}
```

1.1 资源植入层 (Instrumentation Level) 代码示例

我们如何让 JMX 对我们的中央热水器进行管理呢？首先，我们并不想让远程管理者能够关闭我们的中央热水器，因为热水器一旦关上，我们再也无法访问厂家提供的 API 。既然不能关闭它，我们的 MBeans 中也就不需要打开 (turnOn) 方法。所以，我们简单定义的 MBeans 接口如下：

```
package carl.test.jmx;

/**
* @author carlwu
*
*/

public interface CentralHeaterImplMBean {

/**
* return the heater provider
* @return
*/
public String getHeaterProvider();

/**
* Get current temperature of heater
* @return the temperature of the heater
*/
public int getCurrentTemperature();
```

```
/**
* Set the new temperature
* @param newTemperature
*/
public void setCurrentTemperature( int newTemperature);

/**
* Print the current temperature of the heater
* @return the string of current temperature
*/
public String printCurrentTemperature();

}
```

上面的 MBean 接口及其简单，意义也非常明显，我们只向管理程序公开热水器的生产厂家（该属性为只读，管理程序不能更改热水器的生产厂家），但管理程序可以获取并更改当前热水器的温度，并且可以打印出热水器的当前温度。

接下来，我们要做的，就是更改我们已有的 CentralHeaterImpl.java 代码，让它实现 CentralHeaterImplMBean 接口，同时实现 CentralHeaterImplMBean MBean 中规定的所有方法。 CentralHeaterImpl.java 更改后的源代码如下：

```
/**
* The implemenation of Central Heater
* @author carlwu
*
*/
package carl.test.jmx;


public class CentralHeaterImpl implements CentralHeaterInf,CentralHeaterImplMBean {

int currentTemperature ;

public int getCurrentTemperature() {
  return currentTemperature ;
```

```
}

public void setCurrentTemperature( int newTemperature) {
  currentTemperature =newTemperature;
}

public void turnOff() {
  System. out .println( "The heater is off. " );
}

public void turnOn() {
  System. out .println( "The heater is on. " );

}



public String getHeaterProvider() {
  // TODO Auto-generated method stub
  return HEATER_PROVIDER ;

}



public String printCurrentTemperature() {

  String printMsg= "Current temperature is:" + currentTemperature ;
  System. out .println(printMsg);
  return printMsg;
}

}
```

到此为止，我们的资源植入层 (Instrumentation Level) 的代码全部完成，它主要由一个 MBean(CentralHeaterImplBean) 及其实现类 CentralHeaterImpl 组成，在 CentralHeaterImplBean 这个 MBean 接

口中，我们说明了要向管理程序暴露的属性和方法，在本例中，我们的管理程序可以访问热水器的生产厂家信息，同时还可以获取和设置并打印热水器的温度。在 MBean 的实现类中，我们实现了 MBean 接口中规定的所有方法。

然而，在上面的实现中，我们更改已有的 CentralHeaterImpl.java 代码。但从代码编写的角度看，这种做法违反了软件设计的基本原则 — 开闭原则。我们已有的 CentralHeaterImpl.java 类已经经过多次测试，消除了所有的 Bug ，现在为了支持 JMX ，我又增加方法，又修改代码，这会让原本运行得很好的系统重新产生 Bug 。您不妨思考一下，如何不修改 CentralHeaterImpl 类的代码，但又让使 JMX 能够管理我们家的热水器呢？请参考本文的附录，看看您的想法是否比我提供的参考实现高明些？

1.2 代理层 (Agent Level) 示例代码

```
import java.lang.management.ManagementFactory;
import javax.management.MBeanServer;
import javax.management.ObjectName;

/**
* @author carlwu
*
*/
public class CentralHeaterAgent {
private static MBeanServer mBeanServer ;
/**
* @param args
*/

public static void main(String[] args) throws Exception {

  ObjectName oname;
  // get the default MBeanServer from Management Factory

  mBeanServer = ManagementFactory.getPlatformMBeanServer ();
  // try {
  // create a instance of CentralHeaterImpl class
  CentralHeaterInf centralHeater = new CentralHeaterImpl();
```

```
  // assign a Object name to above instance
  oname = new ObjectName( "MyHome:name=centralheater" );

  // register the instance of CentralHeaterImpl to MBeanServer
  mBeanServer .registerMBean(centralHeater, oname);

  System. out .println( "Press any key to end our JMX agent..." );
  System. in .read();

}

}
```

您可以看到，上面的代理层代码异常简单。前面讲过，代理层中最重要的对象就是 MBeanServer ，我们可以把 MBeanServer 理解为一个全局的 HashMap ，所有的 MBeans 都通过唯一的名字注册到这个 HashMap ，这个 HashMap 可以跨越 JVM 访问，甚至可以通过 RMI 、 Http 及其它手段跨越网络传输到其它机器，让其它机器也能访问这个 MBeanServer 中注册的对象。下面我们稍微理解一下代理层代码，在 main() 方法中，

a) 首先我们从 ManagementFactory 的工厂方法中获得 MBeanServer 对象；

b) 然后实例化我们的热水器对象，注意这个对象声明为 CentralHeaterInf ，而不是 CentralHeaterImplMBean 。JMX 规范并没有规定对象声明，只要这个对象实现了一个以 SomethingMBean 命名的接口或父类即可；

c) 接下来通过 new ObjectName(String) 构造函数给我们的 MBean 一个全局的名字，这个名字一般的格式是：" 域名 : 属性 1=*, 属性 2=*,..." 构成；

d) 第四步，我们调用 MBeanServer 的 regiesterBean 方法，通过第三步声明的全局名字把我们的 MBean 实例注册到 MBeanServer 。

这几步都非常简单明了。下面我们在 Eclipse 中运行代理层代码，运行时，请加上下面几个 JVM 运行时参数：

```
-Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.port=9999
-Dcom.sun.management.jmxremote.ssl="false"
-Dcom.sun.management.jmxremote.authenticate="false"
```

这四个 JVM 运行时参数的意义是， MBeanServer 允许其它管理程序通过 RMI 方式访问， RMI 端口是 9999 ，RMI 不使用 SSL 协议，也不需要验证。 Eclipse 的 Run 窗口如下：

然后，请在上面的窗口中点击 Run(运行) 按钮，运行代理层程序。

1.3 管理层代码

管理层代码编写起来其实也比较简单，但如果您要求界面比较完美，并且您也不想卷入到 AWT 加 Swing 的面条代码中，您最好直接使用 JDK 自带的 JConsole.exe 程序，这个程序位于 JDK\bin 目录下，可直接运行。下面我们观察管理程序在远程和本地运行情况。

a) 远程运行 JConsole 管理程序

请双击 JConsole.exe 或者通过命令行在本机上启动 JConsole.exe ，在 JConsole 的连接界面，选择远程连接，然后输入 RMI 地址和端口号，本例为 localhost:9999 ，注意确保我们上面编写的 CentralHeaterAgent 代理处于运行状态，远程连接界面如下图所示：

连接成功后，请点击 MBean 标签，并展开 MyHome 节点，我们可以观察到 CentralHeaterImplMBean 中暴露给管理程序所有的属性和方法，如下图所示：

我们在 CentralHeaterImplMBean 接口中规定， CurrentTemperature 属性是可以更改的，所以上图中 CurrentTemperature 的值显示为绿色，表示远程管理者可以调节；但 HeaterProvider(生产厂家) 的属性是不能更改的，所以其值显示为灰色。现在，我们以远程管理用户的身份，把 CurrentTemperature 属性的值改为 25 ，并按回车或者点击刷新按钮，接下来您可以在上面的界面中，调用操作方法 printCurrentTemperature() ，您会在弹出的对话框中看到 “ Current temperature is:25 ” 的字样，这说明我们的温度更改成功。请注意这是通过远程 RMI 完成的。

b) 本地运行 JConsole 管理程序

请关闭上步中打开的 JConsole ，然后重新运行 JConsole.exe 程序，选择本地进程中的 carl.test.jmx.CentralHeaterAgent 程序，并单击 “连接” 按钮，图示如下。

您在本地的管理程序中可以观察到， MyHome 节点下的 centralheater 的 CurrentTemperature 的值已经改为 25 ，这个更改时通过远程方式完成的。

到此为止， JMX 的小例子就结束了，您可能有些疑惑。 a) 首先， JMX 从表现形式上看似 RMI 或者 EJB ，是一种分布式计算，对吗？ b) 其次，我们在注册 MBean 时，开始声明了一个 MBean 对象，然后把这个对象注册到 MBeanServer ，那么，所有的操作都只能操纵这个对象，对吧？假设我们在程序的其它地方，又新创建了这个 MBean 的另一个实例，这时，我们如何管理这个新创建的实例呢？

我们先来思考一下 JMX 和 RMI/EJB 的区别， RMI/EJB 是分布式计算的一种，它们通过 Stub 和 Skeleon 的方式，在服务器和客户端之间传输 Bean 实例，所以客户端必须知道服务器端 Bean 的接口；客户端可以获得服务

器端的实例对象，并能调用这个实例对象的方法，被调用的方法其实是在客户端运行的，方法的运行需要占用客户端资源。但 JMX 不同， JMX 管理程序（类似于 EJB/RMI 的客户端）不需要了解服务器中 Bean 的任何接口的信息，更不需要从服务器上获取正在运行的实例，所有方法的调用均在服务器端完成，管理程序只是用来监视或者管理服务器中注册的 MBeans 。

再说说 JMX 如何管理新实例的问题，我们知道， JMX 管理的是资源。何谓资源，资源一般代表真实存在的事物。比如说上面例子中的热水器，我们家只有一个热水器，所以一个 MBean 的实例足矣，不必使用 new 来创建另一个实例。但是，您可能会问，我们家假如有两个热水器怎么办？在这种情况下，我们需要两个 MBean 的实例，把它们分别命名为 "MyHome:name=centralheater_1" 和 "MyHome:name=centralheater_2" ，并注册到 MBeanServer ，这两个热水器之间没有任何关系，就和 Java 中同一个类的两个实例类似。当然，同一个类的两个实例之间可以通过 static 属性共享资源。一般说来， JMX 中的 MBean 对应的是真实世界里存在的事物，或者服务器中独一无二的对象，这些对象往往长期驻留在内存中，所以需要管理。如果您新建一个实例，等您的方法退出之后，垃圾回收器马上将这个对象清理掉，您也没有必要使用 JMX 来管理这种昙花一现的对象，对吗？

1.7 JMX在Tomcat中的应用（三）

发表时间: 2009-11-06

三、 Tomcat 中的 JMX

通过上面 JMX 的简单介绍和举例，我们对 JMX 有了一个整体概念。现在我们就来查看一下 JMX 在 Tomcat 中的应用。首先，我们使用 JConsole 查看一下 Tomcat 中有哪些 MBeans 。

3.1 首先，请在 Eclipse 中启动 Tomcat ，在虚拟机参数中，设置下面几个参数：

-Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.port=9999
-Dcom.sun.management.jmxremote.ssl="false"
-Dcom.sun.management.jmxremote.authenticate="false"

运行界面如下所示：

3.2 Tomcat 运行后，请打开 JConsole.exe ，我们可以远程连接到 localhost:9999 ，我们可以看到 Tomcat 中的 MBeans 如下图所示：

我们从上图可以看出， Tomcat 中的 MBeans 位于 Catalina 和 Users 两个 domain 中， Catalina 域名中包含我们所关心的一些 Tomcat 关键组件，比如说 Server 、 Service 、 Realm 、 Engine 和 Connector 等等关键组件，这些 MBean 分别对应我们前面在 Tomcat 架构中讲述的那些组件资源。那么，这些组件是如何注册到 MBeanServer 的呢，注册的流程又是如何，这个问题需要解读 Tomcat 源代码，此处暂不细表，且看下回分解。

3.3 下面，我们做一个非常有趣的实验，体验一下 JMX 管理的乐趣。首先请在浏览器中打开 http://localhost:8080/examples/jsp/jsp2/el/basic-arithmetic.jsp 页面，然后再上图所示的 JConsole 中的 Manager 节点下的 /examples-->localhost 中找到操作中的 “ listSessionIds ” 方法，然后点击调用该方法，您会在弹出的对话框中看到您刚才访问 basic-arithmetic.jsp 页面的 sessionId 值，它是一个 16 位的字符串，我的机器上显示为 “ 4998AB8A07480360BC24A9E9C11A39CA ” ；接下来，请在 Manager 节点下的 /examples-->localhost 中找到属性中的 sessionIdLength 属性，把它的值从 16 改为 22 ，请关闭浏览器，然后重新打开浏览器，再访问一下上面的页面，这时，您再调用 “ listSessionIds” 方法查看一下 sessionId 的列表，会发现新产生的 sessionId 的

位数是 22 位，在我的机器上返回“ 4998AB8A07480360BC24A9E9C11A39CA 04348EFDE953D0B56A206BF11A13E1A5CBB14F316B4F” 两个 sessionId 值。当然，您也可以输入 sessionId 值，调用 expireSession 方法来让某个 session 过期。

Tomcat 中 MBean 的管理方式很多，例如，您可以通过下面的方法打印、查找或者管理 Tomcat 中的 MBean ，该方法的优点是不用打开 RMI 端口，所有操作都是通过 Servlet 转发给 MBeanServer 完成的，具体步骤如下：

a) 首先打开 conf 目录下的 tomcat-users.xml 文件，在 <tomcat-users> 和 </tomcat-users> 标签之间加上下面两行，然后保存该文件。

```
<role rolename="manager"/>
<user username="admin" password="admin" roles="manager"/>
```

这表示我们要添加一个新用户，用户名为 admin ，密码也是 admin ，用户具有 manager 权限。

b) 重启 Tomcat ，然后在浏览器中打开下面的 URL ， http://localhost:8080/manager/jmxproxy/ ，请输入用户名密码 admin/admin ，您将看到 Tomcat 中所有的 MBeans 。在我的机器上，显示 108 个 MBeans 的详细信息。如果您访问 http://localhost:8080/manager/jmxproxy/?qry=*%3Aj2eeType=Servlet%2c* ，您将会看到所有已经加载的 Servlet 的信息，该 qry 是查找 j2eeType=Servlet 的所有 MBeans 。如果您有兴趣，您还可以通过这个 jmxproxy 来动态设置一些 Tomcat 中组件运行时的值。

1.8 JMX在Tomcat中的应用（四）

发表时间: 2009-11-06

四、 Tomcat 中最简单的 MBean

下面我们打开 Tomcat 源代码，看看 Tomcat 中最简单的一个 MBean 。在 Tomcat 的启动引导类 Bootstrap.java 的 172 到 187 行，我们可以看到如下代码：

```
ClassLoader classLoader = ClassLoaderFactory.createClassLoader
(locations, types, parent);

// Retrieving MBean server
MBeanServer mBeanServer = null;
if (MBeanServerFactory.findMBeanServer(null).size() > 0) {
mBeanServer =
  (MBeanServer) MBeanServerFactory.findMBeanServer(null).get(0);
} else {
mBeanServer = MBeanServerFactory.createMBeanServer();
}

// Register the server classloader
ObjectName objectName =
new ObjectName("Catalina:type=ServerClassLoader,name=" + name);
mBeanServer.registerMBean(classLoader, objectName);
```

4.1 上面的代码首先使用 ClassLoaderFactory 工厂类创建一个 ClassLoader ；

4.2 然后在 MBeanServerFactory 这个工厂类中查找 MBeanServer ，如果没有发现，就使用这个工厂类创建一个 MBeanServer ；

4.3 第三步是给刚才创建的 ClassLoader 这个 MBean 取个名字 “ Name: Catalina:type=ServerClassLoader,name=common ” ，然后注册到 MBeanServer 。

您如果在 JConsole 中观察这个 MBean ，会发现这个 MBean 没有向管理应用程序暴露任何属性和方法，并且 Classloader 似乎也符合 JMX 命名规范，它也不是一个 DynamicBean ，这是为什么呢？首先，这个 Classloader

其实一个 StandardClassloader ，而不是 JDK 中缺省的 Classloader ，您如果打开 ClassLoaderFactory 的 createClassLoader 方法，马上就可以看到这一点；另外，看看 StanderClassloader 的类签名，我们会发现该类实现了 StandardClassLoaderMBean 接口，这是符合 JMX 命名规范的；请打开 StandardClassLoaderMBean 接口的源代码，您会发现这是一个空接口，这意味着实现这个接口的 MBean 不会向管理程序暴露任何属性和方法。

最后，希望这篇入门级的简单介绍，能有助于大家理解 JMX 及 JMX 在 Tomcat 中的应用。

附录：针对本文中的热水器小例子，我们给出了一个简单问题，即如何修改我们既有的代码，让这些代码所在的资源能使用 JMX 管理？如果您稍微翻阅一下 Tomcat 的源代码，您会发现， Tomcat 的作者们在 JMX 升级时对已有源代码的改动有点粗暴，勇猛有余，优美不足。 org.apache.catalina.core 包中的关键组件，大部分后加了 preRegister() 、 getObjectName() 等等方法，在 init() 方法中又添加了一堆 Registry.getRegistry.unregisterComponent 或 registerComponent 代码，这些方法其实这些 core 组件没有直接关系，也不是这些 core 组件应该具有的功能，并且这些后添加的代码及其类似。当然，这种情况在我们实际项目中更为多见，主要原因是时间不足，资源有限等等。

如果要比较优美的解决上面的问题，我个人认为，首先，保持现有代码，然后对现有代码进行扩展而不是大刀阔斧的修改已有代码。就拿我们热水器的简单例子来说，我们不要修改 CentralHeaterImpl.java 的现有代码，而是使用 Wrapper 设计模式，设计一个新类，然后将这个 CentralHeaterImpl 类适配成我们需要的 MBean 接口，具体实现如下：

a) CentralHeaterDecoratorMBean.java 源代码

```
package carl.test.jmx;
/**
* @author carlwu
*
*/
public interface CentralHeater Decorator MBean {

/**
* return the heater provider
* @return
*/
public String getHeaterProvider();


/**
* Get current temperature of heater
* @return the temperature of the heater
*/
public int getCurrentTemperature();

/**
* Set the new temperature
* @param newTemperature
*/
public void setCurrentTemperature( int newTemperature);

}
```

CentralHeaterDecorator .java 的源代码：

```
package carl.test.jmx;
public class CentralHeaterDecorator implements CentralHeaterDecoratorMBean {

private CentralHeaterImpl centralHeater ;

public CentralHeaterDecorator(CentralHeaterImpl theCentralHeater){
```

```
centralHeater =theCentralHeater;
}


public int getCurrentTemperature() {
// TODO Auto-generated method stub
return centralHeater .getCurrentTemperature();
}

public void setCurrentTemperature( int newTemperature) {
// TODO Auto-generated method stub
centralHeater .setCurrentTemperature(newTemperature);
}



public String getHeaterProvider() {
// TODO Auto-generated method stub
return centralHeater . HEATER_PROVIDER ;
}


public String printCurrentTemperature() {
// TODO Auto-generated method stub
String returnMsg = "Current temperature is:"
    + centralHeater .getCurrentTemperature();
System. out .println(returnMsg);
return returnMsg;

}

}
```

最后，请把 Agent 代码中的下面一行：

```
CentralHeaterInf centralHeater = new CentralHeaterImpl();
```

改为：

```
CentralHeaterDecoratorMBean centralHeater = new CentralHeaterDecorator(new
CentralHeaterImpl());
```

其运行效果完全一样，但我们完全没有改动既有代码。

1.9 分析 Tomcat catalina.bat 脚本

发表时间: 2009-11-06

Catalina.bat是tomcat所有脚本中最重要的脚本，完成几乎所有的tomcat操作。如启动，关闭等等,都是由catalina.bat脚本来完成的。接下来，我将对Tomcat catalina.bat脚本进行分析。

首先省去catalina.bat开头诸多注解，这些注解主要是讲解各个变量是干什么的。需要的话，自己看下英文就可以了。这里就不翻译了。

```
rem Guess CATALINA_HOME if not defined  查看是否在tomcat目录下，与startup.bat里相同，不解释了。需要的话可以看我的另一篇博客。
set CURRENT_DIR=%cd%
if not "%CATALINA_HOME%" == "" goto gotHome
set CATALINA_HOME=%CURRENT_DIR%
if exist "%CATALINA_HOME%\bin\catalina.bat" goto okHome
cd ..
set CATALINA_HOME=%cd%
cd %CURRENT_DIR%
:gotHome
if exist "%CATALINA_HOME%\bin\catalina.bat" goto okHome
echo The CATALINA_HOME environment variable is not defined correctly
echo This environment variable is needed to run this program
goto end
:okHome

rem Get standard environment variables
if exist "%CATALINA_HOME%\bin\setenv.bat" call "%CATALINA_HOME%\bin\setenv.bat" 如果存在setenv.bat脚本，调用它，我的tomcat 没有这个脚本

rem Get standard Java environment variables
if exist "%CATALINA_HOME%\bin\setclasspath.bat" goto okSetclasspath 查看是否存在setclasspath.bat脚本，如果存在，转到okSetclasspath位置
echo Cannot find %CATALINA_HOME%\bin\setclasspath.bat 否则输出下面两行，并退出
echo This file is needed to run this program
goto end
:okSetclasspath    okSetclasspath位置
```

set BASEDIR=%CATALINA_HOME% 设定BASEDIR变量与CATALINA_HOME变量值相同
call "%CATALINA_HOME%\bin\setclasspath.bat" %1 调用setclasspath.bat脚本并加上参数
if errorlevel 1 goto end 如果存在错误 退出

rem Add on extra jar files to CLASSPATH 设定JSSE_HOME变量，如果存在加入CLASSPATH，不存在跳过

if "%JSSE_HOME%" == "" goto noJsse 检查是否存在JSSE_HOME变量
set
CLASSPATH=%CLASSPATH%;%JSSE_HOME%\lib\jcert.jar;%JSSE_HOME%\lib\jnet.jar;%JSSE_HOME%
如果有加入到CLASSPATH变量后面
:noJsse
set CLASSPATH=%CLASSPATH%;%CATALINA_HOME%\bin\bootstrap.jar 将bootstrap.jar加入到CLASSPATH里

if not "%CATALINA_BASE%" == "" goto gotBase 如果CATALINA_BASE变量不为空，跳过，转到gotBase位置
set CATALINA_BASE=%CATALINA_HOME% 如果为空，将CATALINA_BASE设为CATALINA_HOME变量的值
:gotBase

if not "%CATALINA_TMPDIR%" == "" goto gotTmpdir CATALINA_TMPDIR不为空，跳过，转到gotTmpdir位置
set CATALINA_TMPDIR=%CATALINA_BASE%\temp 如果为空，将 CATALINA_TMPDIR设为%CATALINA_BASE%\temp变量的值（即tomcat\temp）
:gotTmpdir

if not exist "%CATALINA_HOME%\bin\tomcat-juli.jar" goto noJuli 如果不存在tomcat-juli.jar这个类，转到noJuli位置
set JAVA_OPTS=%JAVA_OPTS% -
Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager -
Djava.util.logging.config.file="%CATALINA_BASE%\conf \logging.properties" 如果存在，将变量加入到JAVA_OPTS里
:noJuli

```
set JAVA_OPTS=%JAVA_OPTS% -Xms128m -Xmx512m -Dfile.encoding=UTF8 -
Duser.timezone=GMT -Djava.security.auth.login.config=%CATALINA_HOME%/conf/jaas.config
设定JAVA_OPTS变量

echo Using CATALINA_BASE:   %CATALINA_BASE%     输出CATALINA_BASE变量值
echo Using CATALINA_HOME:   %CATALINA_HOME%     输出CATALINA_HOME变量值
echo Using CATALINA_TMPDIR: %CATALINA_TMPDIR% 输出CATALINA_TMPDIR变量值
if ""%1"" == ""debug"" goto use_jdk        如果变量%1里存在debug ，转到use_jdk位置
echo Using JRE_HOME:        %JRE_HOME%     输出JRE_HOME变量值
goto java_dir_displayed      转到java_dir_displayed
:use_jdk
echo Using JAVA_HOME:       %JAVA_HOME%    输出JAVA_HOME变量值
:java_dir_displayed
                    下面几行设定相应变量
set _EXECJAVA=%_RUNJAVA%
set MAINCLASS=org.apache.catalina.startup.Bootstrap
set ACTION=start
set SECURITY_POLICY_FILE=
set DEBUG_OPTS=
set JPDA=

if not ""%1"" == ""jpda"" goto noJpda
set JPDA=jpda
if not "%JPDA_TRANSPORT%" == "" goto gotJpdaTransport
set JPDA_TRANSPORT=dt_shmem
:gotJpdaTransport
if not "%JPDA_ADDRESS%" == "" goto gotJpdaAddress
set JPDA_ADDRESS=jdbconn
:gotJpdaAddress
if not "%JPDA_SUSPEND%" == "" goto gotJpdaSuspend
set JPDA_SUSPEND=n
:gotJpdaSuspend
if not "%JPDA_OPTS%" == "" goto gotJpdaOpts
set JPDA_OPTS=-Xdebug -
Xrunjdwp:transport=%JPDA_TRANSPORT%,address=%JPDA_ADDRESS%,server=y,suspend=%JPDA_
:gotJpdaOpts
shift
```

```
:noJpda

if ""%1"" == ""debug"" goto doDebug    如果%1为debug，转到doDebug，运行debug模式
if ""%1"" == ""run"" goto doRun        如果%1为run，转到doRun，运行正常模式
if ""%1"" == ""start"" goto doStart    如果%1为start，转到doStart，启动tomcat
if ""%1"" == ""stop"" goto doStop      如果%1为stop，转到doStop，关闭tocmat
if ""%1"" == ""version"" goto doVersion 如果%1为version，转到doVersion，显示tomcat的版本号

echo Usage:  catalina ( commands ... ) 如果%1没有上述内容，输出下面几行，并结束
echo commands:
echo   debug             Start Catalina in a debugger
echo   debug -security   Debug Catalina with a security manager
echo   jpda start        Start Catalina under JPDA debugger
echo   run               Start Catalina in the current window
echo   run -security     Start in the current window with security manager
echo   start             Start Catalina in a separate window
echo   start -security   Start in a separate window with security manager
echo   stop              Stop Catalina
echo   version           What version of tomcat are you running?
goto end

:doDebug
shift                    将%2里的值转到%1
set _EXECJAVA=%_RUNJDB%  将变量 _EXECJAVA设为_RUNJDB变量的值
set DEBUG_OPTS=-sourcepath "%CATALINA_HOME%\..\..\jakarta-tomcat-
catalina\catalina\src\share"
设定DEBUG_OPTS变量

if not ""%1"" == ""-security"" goto execCmd
如果%1不为-security，转到execCmd位置

shift       将%2里的值转到%1
echo Using Security Manager       输出该行
set SECURITY_POLICY_FILE=%CATALINA_BASE%\conf\catalina.policy
设定SECURITY_POLICY_FILE变量的值

goto execCmd     转到execCmd位置
```

```
:doRun
shift        将%2里的值转到%1
if not ""%1"" == ""-security"" goto execCmd
```

1.10 编写批处理文件

发表时间: 2009-11-06

嘿嘿，批处理的介绍。不光可以提高自己动手能力还能学到很多知识，转帖一份，欢迎大家把优秀的批处

批处理的介绍

扩展名是bat(在nt/2000/xp/2003下也可以是cmd)的文件就是批处理文件。

首先批处理文件是一个文本文件，这个文件的每一行都是一条DOS命令（大部分时候就好象我们在DOS提

其次，批处理文件是一种简单的程序，可以通过条件语句(if)和流程控制语句(goto)来控制命令运行的

第三，每个编写好的批处理文件都相当于一个DOS的外部命令，你可以把它所在的目录放到你的DOS搜索

第四，在DOS和Win9x/Me系统下，C:盘根目录下的AUTOEXEC.BAT批处理文件是自动运行批处理文件，每

```
@ECHO OFF
PATH C:\WINDOWS;C:\WINDOWS\COMMAND;C:\UCDOS;C:\DOSTools;C:\SYSTOOLS;C:\WINTOOLS;C:\
LH SMARTDRV.EXE /X
LH DOSKEY.COM /INSERT
LH CTMOUSE.EXE
SET TEMP=D:\TEMP
SET TMP=D:\TEMP
```

批处理的作用
简单的说，批处理的作用就是自动的连续执行多条命令。

这里先讲一个最简单的应用：在启动wps软件时，每次都必须执行（>前面内容表示DOS提示符）：

```
C:\>cd wps
C:\WPS>spdos
C:\WPS>py
C:\WPS>wbx
C:\WPS>wps
```

如果每次用WPS之前都这样执行一遍，您是不是觉得很麻烦呢？

好了，用批处理，就可以实现将这些麻烦的操作简单化，首先我们编写一个runwps.bat批处理文件，内

```
@echo off
c:
cd\wps
spdos
py
wbx
wps
cd\
```

以后，我们每次进入wps，只需要运行runwps这个批处理文件即可。

常用命令

echo、@、call、pause、rem(小技巧：用::代替rem)是批处理文件最常用的几个命令，我们就从他们开

echo 表示显示此命令后的字符
echo off 表示在此语句后所有运行的命令都不显示命令行本身
@与echo off相象，但它是加在每个命令行的最前面，表示运行时不显示这一行的命令行（只能影响当前
call 调用另一个批处理文件（如果不用call而直接调用别的批处理文件，那么执行完那个批处理文件后
pause 运行此句会暂停批处理的执行并在屏幕上显示Press any key to continue...的提示，等待用
rem 表示此命令后的字符为解释行（注释），不执行，只是给自己今后参考用的（相当于程序中的注释

例1：用edit编辑a.bat文件，输入下列内容后存盘为c:\a.bat，执行该批处理文件后可实现：将根目录

批处理文件的内容为: 命令注释：

```
@echo off 不显示后续命令行及当前命令行
dir c:\*.* >a.txt 将c盘文件列表写入a.txt
call c:\ucdos\ucdos.bat 调用ucdos
echo 你好 显示"你好"
pause 暂停,等待按键继续
rem 准备运行wps 注释：准备运行wps
cd ucdos 进入ucdos目录
wps 运行wps
```

批处理文件的参数

批处理文件还可以像C语言的函数一样使用参数（相当于DOS命令的命令行参数），这需要用到一个参数

%[1-9]表示参数，参数是指在运行批处理文件时在文件名后加的以空格（或者Tab）分隔的字符串。变量

例2：C:根目录下有一批处理文件名为f.bat，内容为：

```
@echo off
format %1
```

如果执行C:\>f a:
那么在执行f.bat时，%1就表示a:，这样format %1就相当于format a:，于是上面的命令运行时实际执

例3：C:根目录下一批处理文件名为t.bat，内容为:

```
@echo off
type %1
type %2
```

那么运行C:\>t a.txt b.txt
%1 ： 表示a.txt
%2 ： 表示b.txt
于是上面的命令将顺序地显示a.txt和b.txt文件的内容。

特殊命令

if goto choice for是批处理文件中比较高级的命令，如果这几个你用得很熟练，你就是批处理文件的

一、if 是条件语句，用来判断是否符合规定的条件，从而决定执行不同的命令。 有三种格式:

1、if [not] "参数" == "字符串" 待执行的命令

参数如果等于(not表示不等，下同)指定的字符串，则条件成立，运行命令，否则运行下一句。

例：if "%1"=="a" format a:

2、if [not] exist [路径\]文件名 待执行的命令
如果有指定的文件，则条件成立，运行命令，否则运行下一句。

如: if exist c:\config.sys type c:\config.sys
表示如果存在c:\config.sys文件，则显示它的内容。

3、if errorlevel <数字> 待执行的命令

很多DOS程序在运行结束后会返回一个数字值用来表示程序运行的结果(或者状态)，通过if errorleve

如if errorlevel 2 goto x2

二、goto 批处理文件运行到这里将跳到goto所指定的标号(标号即label，标号用:后跟标准字符串来定

如:

```
goto end

:end
echo this is the end
```

标号用“:字符串”来定义，标号所在行不被执行。

三、choice 使用此命令可以让用户输入一个字符（用于选择），从而根据用户的选择返回不同的error

注意：choice命令为DOS或者Windows系统提供的外部命令，不同版本的choice命令语法会稍有不同，

choice的命令语法（该语法为Windows 2003中choice命令的语法，其它版本的choice的命令语法与此

```
CHOICE [/C choices] [/N] [/CS] [/T timeout /D choice] [/M text]
```

描述:
该工具允许用户从选择列表选择一个项目并返回所选项目的索引。

参数列表:
/C choices 指定要创建的选项列表。默认列表是 "YN"。

/N 在提示符中隐藏选项列表。提示前面的消息得到显示，
选项依旧处于启用状态。

/CS 允许选择分大小写的选项。在默认情况下，这个工具
是不分大小写的。

/T timeout 做出默认选择之前，暂停的秒数。可接受的值是从 0
到 9999。如果指定了 0，就不会有暂停，默认选项
会得到选择。

/D choice 在 nnnn 秒之后指定默认选项。字符必须在用 /C 选
项指定的一组选择中；同时，必须用 /T 指定 nnnn。

/M text 指定提示之前要显示的消息。如果没有指定，工具只
显示提示。

/? 显示帮助消息。

注意:
ERRORLEVEL 环境变量被设置为从选择集选择的键索引。列出的第一个选
择返回 1，第二个选择返回 2，等等。如果用户按的键不是有效的选择，
该工具会发出警告响声。如果该工具检测到错误状态，它会返回 255 的
ERRORLEVEL 值。如果用户按 Ctrl+Break 或 Ctrl+C 键，该工具会返回 0
的 ERRORLEVEL 值。在一个批程序中使用 ERRORLEVEL 参数时，将参数降
序排列。

示例:
```
CHOICE /?
CHOICE /C YNC /M "确认请按 Y，否请按 N，或者取消请按 C。"
CHOICE /T 10 /C ync /CS /D y
CHOICE /C ab /M "选项 1 请选择 a，选项 2 请选择 b。"
CHOICE /C ab /N /M "选项 1 请选择 a，选项 2 请选择 b。"
```

如果我运行命令：CHOICE /C YNC /M "确认请按 Y，否请按 N，或者取消请按 C。"
屏幕上会显示：

确认请按 Y，否请按 N，或者取消请按 C。 [Y,N,C]?

例：test.bat的内容如下（注意，用if errorlevel判断返回值时，要按返回值从高到低排列）：

```
@echo off
choice /C dme /M "defrag,mem,end"
if errorlevel 3 goto end
if errorlevel 2 goto mem
if errotlevel 1 goto defrag

:defrag
c:\dos\defrag
goto end

:mem
mem
goto end

:end
echo good bye
```

此批处理运行后，将显示“defrag,mem,end[D,M,E]?” ，用户可选择d m e ，然后if语句根据用户的

四、for 循环命令，只要条件符合，它将多次执行同一命令。

语法：
对一组文件中的每一个文件执行某个特定命令。

```
FOR %%variable IN (set) DO command [command-parameters]
```

%%variable 指定一个单一字母可替换的参数。
(set) 指定一个或一组文件。可以使用通配符。
command 指定对每个文件执行的命令。
command-parameters
为特定命令指定参数或命令行开关。

例如一个批处理文件中有一行:

```
for %%c in (*.bat *.txt) do type %%c
```

则该命令行会显示当前目录下所有以bat和txt为扩展名的文件的内容。

批处理示例

1. IF-EXIST

1)

首先用记事本在C:\建立一个test1.bat批处理文件，文件内容如下：

```
@echo off
IF EXIST \AUTOEXEC.BAT TYPE \AUTOEXEC.BAT
IF NOT EXIST \AUTOEXEC.BAT ECHO \AUTOEXEC.BAT does not exist
```

然后运行它：

```
C:\>TEST1.BAT
```

如果C:\存在AUTOEXEC.BAT文件，那么它的内容就会被显示出来，如果不存在，批处理就会提示你该文

2)

接着再建立一个test2.bat文件，内容如下：

```
@ECHO OFF
IF EXIST \%1 TYPE \%1
IF NOT EXIST \%1 ECHO \%1 does not exist
```

执行:

```
C:\>TEST2 AUTOEXEC.BAT
```

该命令运行结果同上。

说明：

(1) IF EXIST 是用来测试文件是否存在的，格式为

IF EXIST [路径+文件名] 命令

(2) test2.bat文件中的%1是参数，DOS允许传递9个批参数信息给批处理文件，分别为%1~%9(%0表示t

3) 更进一步的，建立一个名为TEST3.BAT的文件，内容如下：

```
@echo off
IF "%1" == "A" ECHO XIAO
IF "%2" == "B" ECHO TIAN
IF "%3" == "C" ECHO XIN
```

如果运行：

```
C:\>TEST3 A B C
```

屏幕上会显示:

```
XIAO
TIAN
XIN
```

如果运行：

```
C:\>TEST3 A B
```

屏幕上会显示

```
XIAO
TIAN
```

在这个命令执行过程中，DOS会将一个空字符串指定给参数%3。

2、IF-ERRORLEVEL

建立TEST4.BAT，内容如下：

```
@ECHO OFF
XCOPY C:\AUTOEXEC.BAT D:IF ERRORLEVEL 1 ECHO 文件拷贝失败
IF ERRORLEVEL 0 ECHO 成功拷贝文件
```

然后执行文件:

```
C:\>TEST4
```

如果文件拷贝成功，屏幕就会显示“成功拷贝文件”，否则就会显示“文件拷贝失败”。

IF ERRORLEVEL 是用来测试它的上一个DOS命令的返回值的，注意只是上一个命令的返回值，而且返回

因此下面的批处理文件是错误的：

```
@ECHO OFF
XCOPY C:\AUTOEXEC.BAT D:\
IF ERRORLEVEL 0 ECHO 成功拷贝文件
IF ERRORLEVEL 1 ECHO 未找到拷贝文件
IF ERRORLEVEL 2 ECHO 用户通过ctrl-c中止拷贝操作
IF ERRORLEVEL 3 ECHO 预置错误阻止文件拷贝操作
IF ERRORLEVEL 4 ECHO 拷贝过程中写盘错误
```

无论拷贝是否成功，后面的：

```
未找到拷贝文件
用户通过ctrl-c中止拷贝操作
预置错误阻止文件拷贝操作
拷贝过程中写盘错误
```

都将显示出来。

以下就是几个常用命令的返回值及其代表的意义：

```
backup
0 备份成功
1 未找到备份文件
2 文件共享冲突阻止备份完成
3 用户用ctrl-c中止备份
4 由于致命的错误使备份操作中止

diskcomp
0 盘比较相同
1 盘比较不同
2 用户通过ctrl-c中止比较操作
3 由于致命的错误使比较操作中止
4 预置错误中止比较

diskcopy
0 盘拷贝操作成功
1 非致命盘读/写错
```

```
2 用户通过ctrl-c结束拷贝操作
3 因致命的处理错误使盘拷贝中止
4 预置错误阻止拷贝操作

format
0 格式化成功
3 用户通过ctrl-c中止格式化处理
4 因致命的处理错误使格式化中止
5 在提示“proceed with format(y/n)?”下用户键入n结束

xcopy
0 成功拷贝文件
1 未找到拷贝文件
2 用户通过ctrl-c中止拷贝操作
4 预置错误阻止文件拷贝操作
5 拷贝过程中写盘错误

3、IF STRING1 == STRING2

建立TEST5.BAT，文件内容如下：
@echo off
IF "%1" == "A" FORMAT A:

执行：
C:\>TEST5 A
屏幕上就出现是否将A:盘格式化的内容。

注意：为了防止参数为空的情况，一般会将字符串用双引号（或者其它符号，注意不能使用保留符号）括
如：if [%1]==[A] 或者 if %1*==A*

5、GOTO

建立TEST6.BAT，文件内容如下：
@ECHO OFF
IF EXIST C:\AUTOEXEC.BAT GOTO _COPY
GOTO _DONE
```

```
:_COPY
COPY C:\AUTOEXEC.BAT D:\
:_DONE
```

注意：
(1) 标号前是ASCII字符的冒号":"，冒号与标号之间不能有空格。
(2) 标号的命名规则与文件名的命名规则相同。
(3) DOS支持最长八位字符的标号，当无法区别两个标号时，将跳转至最近的一个标号。

6、FOR

建立C:\TEST7.BAT，文件内容如下：

```
@ECHO OFF
FOR %%C IN (*.BAT *.TXT *.SYS) DO TYPE %%C
```

运行：

```
C:>TEST7
```

执行以后，屏幕上会将C:盘根目录下所有以BAT、TXT、SYS为扩展名的文件内容显示出来（不包括隐藏文

1.11 《How Tomcat Works》读书笔记（二）:Connector

发表时间: 2009-11-06

Chapter Three：Connector

tomcat的Connector名字叫做Coyote，我之前也写了几篇关于coyote的博客，不过在看了第三章后，才对tomcat的 Connector有了更加深入的认识。需要说明的是，这一章的Connector只是一个简化版，而第四章介绍的也只是“默认”（旧版本）的 Tomcat的Connector，正因为“默认”的Connector性能不佳，才产生了后来的coyote，这是后话。

StringManager

在讲述连接器前，首先介绍一个tomcat内部使用频率非常高的工具类——StringManager，简称sm（O(∩_∩)O有点歧义~）。我 们知道tomcat是一个大项目，里面的package很多，而每个package内的类都需要输出很多信息，包括错误信息、调试信息，等等。为了降低耦 合性，tomcat开发人员专门设计了这个sm类，用来存取相关的输出信息。每个package都有一个 “LocalStrings.properties”文件，就像ini文件那样保存了这些信息。我们只需要像下面这样：

```
StringManager sm = StringManager.getManager("ex03.pyrmont.connector.http");
```

就可以获得一个特定于某个package输出信息的sm，然后直接getString即可。为了便于支持多语言，一般还包括了 LocalStrings_es.properties和LocalStrings_ja.properties，sm会自动根据本地语言设置来选择相应 的语种，这和Struts非常像（兴许Struts就是模仿tomcat的）。可惜，没有LocalStrings_cn.properties

Bootstrap

像我们看到的tomcat6一样，在这一章的小例子中，也专门将Bootstrap类提取出来，用于启动tomcat，当然这里还是非常简单的new了一个Connector，以后的章节会陆续添加功能。在这里，Connector实现了Runnable接口

```
public final class Bootstrap {
  public static void main(String[] args) {
    HttpConnector connector = new HttpConnector();
    connector.start();
  }
}
```

Connector

终于可以一睹Connector的“芳容”了！很遗憾，这一章的Connector还非常简陋，只是把前一章中监听Socket的部分代码copy了过来，略微有点不同的是：

```
// Hand this socket off to an HttpProcessor
        HttpProcessor processor = new HttpProcessor(this);
        processor.process(socket);
```

很明显，这里多了一个HttpProcessor ，其实就相当于“容器”的角色。tomcat发展到现在，其架构也还是：Connector+Container

HttpProcessor

可以说，HttpProcessor 是这一章的重头戏，大部分功能都是通过这个类，直接或间接地实现的。

照旧，在HttpProcessor 的process方法中，首先获取Socket的输入输出流、new 一个Request和Response，然后调用：

```
parseRequest(input, output);
    parseHeaders(input);
```

这两个方法是本类的核心，parseRequest是处理http请求行（就是类似“GET http://xxx.xxx/xxx?name=xxx” ，位于http请求的第一行），parseHeaders则处理请求行之后的一堆 header，比如content-length、cookie等。别看只有两个方法，深入进去其实调用了很多其他类的方法，看来解析一个http请求也 不是那么容易的。

SocketInputStream

这个类转自tomcat源代码，它对Socket的原始inputstream进行了封装，负责将以下字段分离出来：

- http schema：请求行中的GET,POST等等
- URI：例如，http://xxx.xxx/xxx ，这里还要区分是绝对路径还是相对路径
- 查询字符串：就是“?”后面的那些键值对
- header：http 请求的headers

当然，分离出来的字段都是以char数组的形式保存的，因为生成String的开销很大，通常都是“lazy load” ，不到不得已不会随便new string

Populate Request

所谓populate，即是“赋值”的意思，就是调用Request的那些set方法，把之前解析出来的那些字段一个个放进Request对象中，供后续使用。考虑到解析、分割字符串的开销很大，tomcat的原则是把查询字符串（query string）和cookie 的解析工作放到Servlet中，因为这些字段未必一定会用到，要的时候在生成也不迟，从而节省了系统资源。

具体的解析过程很啰嗦，基本上都是字符串处理，在一堆字节中摸爬滚打，这里就不赘述了。

Create Response

相比Request，Response的工作量就小了一些。但也有不少改进之处。首先，之前的Response.getWriter方法，单纯的返 回一个包装了Socket的OutputStream的PrintWriter，但这是JAVA自带的PrintWriter，功能上不能完全满足 tomcat的需要，例如它的print方法不能自动flush（具体可以参考JavaDoc）。所以，在这里通过两个类：ResponseStream 和ResponseWriter，分别拓展了原始的ServletOutputStream和PrintWriter。下面摘取其中一部分代码：

```
public void write(String s, int off, int len) {
    super.write(s, off, len);
    super.flush();
  }
```

这是ResponseWriter的write方法，可见就是在PrintWriter的方法上多了一个flush而已

其他

搞定Request后，还是像上一章那样，把Request交给ServletProcessor或者StaticResourceProcessor，基本没有大的变动

总结

首先，这一章的服务器架构如下：（图片源自原书）

image

处理流程为：

1. Bootstrap启动HttpConnector
2. HttpConnector监听Socket端口，将得到的Socket对象交给HttpProcessor
3. HttpProcessor通过调用parseRequest和parseHeaders方法，解析底层Socket流中的字节，生成Request对象和Response对象
4. 把Request对象和Response对象交给“容器”处理，即ServletProcessor或者StaticResourceProcessor
5. 载入Servlet对象，利用Facade模式，将Request对象和Response对象传入Servlet的service方法，处理，然后通过Response对象的writer，把响应内容返回给客户端

1.12 《How Tomcat Works》读书笔记（一）

发表时间: 2009-11-06

看了这本书的头三章，写得非常好，可谓深入浅出将tomcat分析的很透彻。虽然书中所讲述的tomcat是“简化版”，但内容也不算少，越到后面代码越多，也越复杂。为了加深印象，遂决定写读书笔记，“好记性不如烂笔头”，说不定还能方便他人。

闲话少说，直入主题：

Chapter One：A Simple Web Server

第一章是一个非常简单的web server，主要目的在于让读者了解Java的web server 编程模式。此外还讲了一下Http协议的一些基础知识，譬如http请求和响应的格式。

基本的web server，就是用java.net.ServerSocket类持续的监听特定的端口，有连接过来时，则返回一个Socket对象，再对Socket对象的输入输出流进行操作。其实这也是大部分服务器的基本思路。

简单贴一些代码

```
ServerSocket serverSocket = null;
int port = 8080;
try {
  serverSocket =  new ServerSocket(port, 1,
    InetAddress.getByName("127.0.0.1"));
}
catch (IOException e) {
  e.printStackTrace();
  System.exit(1);
}
// Loop waiting for a request
while (!shutdown) {
  Socket socket = null;
  InputStream input = null;
  OutputStream output = null;

  try {
    socket = serverSocket.accept();
    input = socket.getInputStream();
```

```
    output = socket.getOutputStream();
    // create Request object and parse
    Request request = new Request(input);
    request.parse();

// create Response object
    Response response = new Response(output);
    response.setRequest(request);
    response.sendStaticResource();
```

其中，Request对象的parse方法，是对输入流的字节进行解析，当然这里只做了最基本的工作，将输入流的内容一字不漏的打印出来。

的确很简单~这里的Request对象实现了Servlet规范中的Request接口，后面我们会看到，如何运用设计模式令Request更加“优雅”。

Chapter Two——A Simple Servlet Container

J2EE流行“容器Container”这一说法，所以这一章开始针对Servlet做一些工作。先说说这个“容器”的架构：

像第一章那样，我们使用一个HttpServer类监听端口，然后将得到的inputStream交给一个Request对象进行解析。然后，根据 HTTP请求行（Request Line）的内容，判断是静态页面还是Servlet页面，再分别交给StaticResoureProcessor类或者 ServletProcessor类进行处理：

```
// check if this is a request for a servlet or
        // a static resource
        // a request for a servlet begins with "/servlet/"
        if (request.getUri().startsWith("/servlet/")) {
          ServletProcessor1 processor = new ServletProcessor1();
          processor.process(request, response);
        }
        else {
          StaticResoureProcessor processor =
            new StaticResourceProcessor();
          processor.process(request, response);
        }
```

StaticResoureProcessor类没什么“技术含量”，只是利用Response类，单纯地将静态页面的内容从html文件中读出来原样返回给客户端（说白了是写到Socket的输出流OutputStream中）。

而Request类在这一章实现了javax.servlet.ServletRequest接口，因此不得不实现接口中的一大堆方法。出于简化考虑，现在大部分方法还是空的，返回一个null而已。

ServletProcessor类

比较出彩的是ServletProcessor类，出彩之处在于通过类加载器从class文件中动态载入Servlet（这里的Servlet的功能只是简单打印字符串），其源代码据说是从tomcat中copy过来的

```
String uri = request.getUri();
String servletName = uri.substring(uri.lastIndexOf("/") + 1
URLClassLoader loader = null;
try {
   // create a URLClassLoader
   URL[] urls = new URL[1];
   URLStreamHandler streamHandler = null;
   File classPath = new File(Constants.WEB_ROOT);
   // the forming of repository is taken from the
   // createClassLoader method in
   // org.apache.catalina.startup.ClassLoaderFactory
   String repository =
    (new URL("file", null, classPath.getCanonicalPath() +
    File.separator)).toString() ;
   // the code for forming the URL is taken from
   // the addRepository method in
   // org.apache.catalina.loader.StandardClassLoader.
   urls[0] = new URL(null, repository, streamHandler);
   loader = new URLClassLoader(urls);
}
```

需要解释一下的是，这里的repository 指的是编译好的类文件的存放路径，也算是tomcat里的一个专门术语了

最后，简单的load进来，调用Servlet的经典方法“service”，把Request和Response传进去，就算大功告成了。

```
myClass = loader.loadClass(servletName);
```

```
servlet = (Servlet) myClass.newInstance();
servlet.service ((ServletRequest) request, (ServletResponse) response);
```

Facade模式

仔细观察上面那行代码，我们将request对象向上转型为ServletRequest，交给service方法。但如果哪个Servlet的开 发者清楚tomcat的内部设计，那么他就可以在service方法中将request向下“还原为”Request对象。Request类提供了许多 tomcat内部使用的功能，譬如解析http流。如果让外人乱用一气，后果很严重。为了避免这个问题，tomcat采用了Facade 设 计模式，即通过一个RequestFacade类（实现了ServletRequest接口），替换Request类传递给service方法。而在 RequestFacade内部，保存我们的request对象，外人只能通过调用RequestFacade实现了的ServletRequest接口 的方法，间接的使用Request的“部分功能”，不能调用Request的一些内部方法，从而优雅的解决了上述问题。

```
public class RequestFacade implements ServletRequest {
  private ServleLRequest request = null;
  public RequestFacade(Request request) {
    this.request = request;
  }
  /* implementation of the ServletRequest*/
  public Object getAttribute(String attribute) {
    return request.getAttribute(attribute);
  }
  public Enumeration getAttributeNames() {
    return request.getAttributeNames();
  }
```

至此，一个简单的Servlet容器就造出来了。

1.13 《How Tomcat Works》读书笔记（三）:Tomcat default connector

发表时间: 2009-11-06

Chapter 4: Tomcat default connector

何为default Connector？其实这里指的是tomcat最初设计时使用的Connector，尽管问题多多，现在已经被coyote所取代，但作为教学用例，default Connector仍然不失为一个优秀的组件，值得一学！

这一章的目的是系统的讲述tomcat的Connector，同时为介绍后面的容器作铺垫。

从这一章开始，内容越来越多也逐步深化，因此做笔记也只能摘录一部分，没有基础的人看了也许会觉得不连贯，那么推荐你先去看看原书或者它的翻译。

Http 1.1

default Connector支持Http 1.1标准，相比Http1.0，1.1最大的特色在于增加了对长连接的支持。举个例子，我们浏览网页的时候，一个html页面中，其实包含了很多资源， 如图像、视频、声音，等等。每个资源都有独立的URL地址，在过去1.0时，一次只能针对一个URL获取资源，因此往往打开一个页面的操作其实包含了很多 次的“客户端/服务器”之间的连接过程。反反复复的建立、断开连接是非常耗时的，而且也无疑增加了服务器的负担。在Http1.1中，通过长连接，可以一 次性获取多个资源，无疑是大大节约了网络资源。

客户端可以通过在请求中包含“connection: keep-alive”这一头字段来要求服务器提供长连接

Chunked Encoding（块编码）

有了长连接，客户端应该如何区分在一个连接中传输的多个资源呢？在过去的1.0中，由于每次只传输一个资源，因此不会有这个问题。能否使用 content-length字段？不行。因为服务器并不会等到一整个图像文件都准备好了才传给你，往往是一次传若干字节，这种做法相当于操作系统的“分 时”，有利于提高并发性和用户的使用感受。专家们最后终于想出一个办法，就是在每次传输时加上一些信息表示这次要传多少字节，例如：

```
1D\r\n
I'm as helpless as a kitten u
9\r\n
p a tree.
0\r\n
```

通过“十六进制数字”+“\r\n”，代表一次传输的字节数，最后的“0\r\n”表示一次会话的结束。但书上并没有说一次会话就相当于传一个资源，因此到底如何区分多个资源，还不是很清楚，需要研究Http协议才能知道答案。

Connector接口

二话不说，上图：

image

tomcat里面最基本的两个接口应该就是Connector和container了，接口本身很简单很抽象，但衍生出来的东西却非常复杂。

从图中可以看出，Connector和container是一一对应的，Connector有setContainer方法，但container 没有类似的setConnector，说明只有Connector才知道container的存在，反之是不成立的。Connector是为了把 Request和Response传递给container；而container不必理会谁给他Request，只要能拿到Request、再把响应写 入Response就行了

此外，一个HttpConnector中包含了多个HttpProcessor，目的是为了支持多线程，可以同时处理多个Socket连接。

多线程处理Socket

old version

先来回顾之前的实现：

```
while (!stopped) {
      Socket socket = null;
      try {
        socket = serversocket.accept();
      }       catch (Exception e) {
        continue;
      }
      // Hand this socket off to an Httpprocessor
      HttpProcessor processor = new Httpprocessor(this);
      processor.process(socket);
    }
```

这种做法的后果是，整个处理流程完全是同步的，因为 processor.process 方法是同步的。这样只有一个请求完全处理完毕后，processor.process 才会返回，while循环才能继续，效率之低可想而知，根本应付不了几个并发请求。

new version

既然找到了问题所在，接下来就要提出解决方案。很自然的想法是：线程。

大致思路是这样的：通过一个堆栈，我们保存若干数量的HttpProcessor，这些HttpProcessor都实现了Runnable接口。 HttpConnector和HttpProcessor分别代表了生产者和消费者。一方面，HttpConnector获得一个个的Socket，放入 空闲的HttpProcessor中，HttpProcessor处理完毕后，通过recycle方法将自己“回收”。如果没有空闲的 HttpProcessor，则HttpConnector会将该Socket丢弃，直接关闭。

说到生产者和消费者这种同步模型，那么控制同步的“关键域”是什么呢？此处其实是一个布尔变量available，表示此时 HttpProcessor是否空闲。HttpConnector调用HttpProcessor的同步的assign方法，将Socket交给 HttpProcessor，如果available为假，那么assign方法调用完成直接返回，如果为真，则HttpConnector就wait在 那里直到可用为止。HttpProcessor本身则采取相反的策略。

any question?

仔细推敲一下，会觉得这里的算法有些奇怪。首先，HttpProcessor是放在堆栈中的，凡是“非空闲”的HttpProcessor都不在堆 栈中，它自己忙完了才会recycle回去，所以HttpConnector从堆栈中取出、并调用assign时的HttpProcessor肯定是“空 闲”的；其次，让HttpConnector专门“wait”在那里，等于是整个tomcat都卡在那里，无法再接受新的Socket，不合情理。希望能 有高人指点一下其中的原因，到底是bug，还是有更深层次的考虑。

Request & Response

Request的主要功能基本没有变，Facade依然存在，但是其继承关系比原来复杂了许多，书中也没有一一解释，让人云里雾里，或许后面的章节会有补充。

image

在HttpProcessor取得新的Socket后，依旧是通过process方法对Socket进行处理，流程上还是按照之前的做法，依次解析

1. Connection：检查有没有使用代理
2. Request：类似第三章
3. Headers：类似第三章，但是用char[]取代String作为头字段的名称，例如：static final char[] AUTHORIZATION_NAME = "authorization".toCharArray();

1.14 《How Tomcat Works》读书笔记（四）：容器初探

发表时间: 2009-11-06

第四章：容器初探

接触JAVA EE以来，最初对“容器”一词满头雾水、无比崇拜，后来听到耳朵长茧，一直觉得这个词的定义有点太广了，很多情况下不管沾没沾点关系的都往上靠，力图通过 此术语使自己显得“专业”一些（老实说我写文档也这么做过）。但不论如何，发明这个计算机术语的人还是相当牛的，充分体现了JAVA EE“分层”的思想。

唯一不爽的是，一直以来都处于“容器”的黑盒之外，更加上那些大厂商对自己的JAVA EE“容器”产品的神乎其神的吹嘘宣传，一直没法想象外国那些鬼佬怎么就这么牛能做出这么厉害的东西，我们只有乖乖使用的份？还好有开源，还好有这本 《How Tomcat Works》，可以满足我的好奇心，一窥“容器”的奥秘

Tomcat的容器架构

我们一般都把tomcat、weblogic、websphere app server和JBOSS AS称为“J2EE容器”，是一种广义的说法；而这里的“容器”，指的是tomcat中的两大组件之一的“容器”，属于狭义的说法（另一种组件当然就是 Connector了）。

tomcat的容器架构，一直都没有太大变化，基本元素都是四个接口：

- engine：表示一整个Catalina Servlet引擎
- host：表示一个虚拟主机。什么是虚拟主机可以百度一下“tomcat 虚拟主机配置”
- Context：表示一个web app应用，比如你做的一个网站
- Wrapper：表示单个Servlet

以上四个基本元素由上至下逐渐细分，成树状结构，构成了tomcat容器结构的主体，它们都位于org.apache.catalina包

值得一提的是，这四个接口并不是同时必须的，例如，你完全可以做一个只有Wrapper的“迷你版”tomcat，这在一些资源受限的环境中，比如嵌入式系统很有用（说不定将来能放到手机里面跑，O(∩_∩)O哈哈~）

类图如下：

image

一般来说，容器里头都还有session管理、日志等功能，不过这一章暂时还不作讨论。

PipeLine & Valve

熟悉Servlet的人一定接触过Servlet filter，在Servlet处理请求之前，先会由filter“过滤”一下。tomcat内部同样也有类似的东西，那就是Valve——阀门。而所有 的Valve都是装在一个pipeline（管道）里头的，tomcat的开发者估计对水管工之类的活比较感兴趣。这些Valve的功能各异，你也可以自 己开发然后放到tomcat的配置文件里面。

那具体是如何工作的呢？**首先** ，Connector调用容器的invoke方法，把Request给容器，容器再把Request对象给自身的pipeline：

```
public void invoke(Request request, Response response)
  throws IOException, ServletException {
  pipeline.invoke(request, response);

  ...
}
```

然后 ，在pipeline内部有个内部类ValveContext，它来管理所有的Valve。pipeline一般会调用ValveContext的invokeNext

```
public void invokeNext(Request request, Response response)
```

ValveContext又调用第N个Valve的invoke方法（N是一个计数器，记录调用到第几个Valve了），不过参数稍微有点不同

```
public void invoke(Request request, Response response,
  ValveContext ValveContext ) throws IOException, ServletException
```

它把自己作为参数传了进去给Valve，有什么用呢？其实很简单，看看Valve的invoke是怎么实现的

```
public void invoke(Request request, Response response,
  ValveContext valveContext) throws IOException, ServletException {
  // Pass the request and response on to the next valve in our pipeline
  valveContext.invokeNext(request, response);
  // now perform what this valve is supposed to do
    ...
}
```

秘密就在这里！Valve又**回调** 了ValveContext的invokeNext，这样就相当于递归一样，把全部Valve都调用一遍。

仔细推敲会发现，每个Valve都是先调用ValveContext的invokeNext，然后才做自己的工作，所以“第一个”被 pipeline调用的Valve，实际却是最后一个完成自己工作的，有点类似“压栈”操作，第一个Valve最先被压进去，却是最后一个从堆栈中弹出来 的。如果不信，可以做个试验，眼见为实。

修改BasicValve

每个pipeline默认会有一个basicvalve，做一些基本工作，比如把Request传递给下一级子容器，或者把Request交给 Servlet（Wrapper的basicvalve就是做这个的）。从pipeline的源码来看（书中的源码，未必是tomcat的源 码），basicvalve是最后一个被调用的。

```
if (subscript < valves.length) {       valves[subscript].invoke(request, response, this);
  }
  else if ((subscript == valves.length) && (basic != null)) {
    basic.invoke(request, response, this);
  }
```

以上是ValveContext.invokeNext方法的一部分，basic就是指BasicValve，很明显是最后一个被加进去的。

我们在basicvalve的invoke方法第一行增加一个简单的输出，运行之后就会发现，basicvalve的输出在其他Valve的前面，可见上面的推断是正确的！

关于Wrapper的疑问

这一章最后是两个简单的程序：第一个只有Wrapper容器，另一个则由两个Wrapper包含在一个Context容器里组成。Wrapper和 Context接口就不啰嗦了，在后续章节有专门的详细解说。但在这里，每个Wrapper对应一个Servlet，如果是个大项目，那里面的 Servlet起码有几十个，很难想像有那么多的Wrapper在同时跑，会不会导致性能低下呢？也许要看完这本书才能找到答案了

keep moving，坚持每周看一章~！

1.15 《How Tomcat Works》读书笔记（五）：生命周期

发表时间: 2009-11-06

第一次接触到“生命周期”这个词汇，是在软件工程的课程上，“软件的生命周期”，当时也是觉得有点玄，但还算可以理解：软件从需求分析到最后没人用，就像人的一生（人的“需求分析”是什么呢？这是个哲学问题...）

扯远了，回到tomcat。tomcat的“生命周期”非常简单，说白了是一个接口：org.apache.catalina.Lifecycle，内容如下：

```
public interface Lifecycle {
  public static final String START_EVENT = "start";
  public static final String BEFORE_START_EVENT = "before_start";
  public static final String AFTER_START_EVENT = "after_start";
  public static final String STOP_EVENT = "stop";
  public static final String BEFORE_STOP_EVENT = "before_stop";
  public static final String AFTER_STOP_EVENT = "after_stop";


  public void addLifecycleListener(LifecycleListener listener);
  public LifecycleListener[] findLifecycleListeners();
  public void removeLifecycleListener(LifecycleListener listener);
  public void start() throws LifecycleException;
  public void stop() throws LifecycleException;
}
```

看清楚了，其实就是一些常量加上start、stop这些方法。

那么这个接口存在的意义是什么呢？这其实和tomcat的架构有关，因为tomcat是以“容器”的方式来组织的，在前面的几章中也提到了，容器是 以树结构组织的，也就是除了根容器之外，其他容器肯定都有且仅有一个父容器，一层套一层。tomcat的启动一般是通过Bootstrape类来完成的， 这个类会调用根节点容器的启动方法。但是那么多的子容器怎么办呢，Bootstrape不可能一一启动它们；需要关闭tomcat时也是如此。唯一的办法 是定义一个统一的接口，把所有的启动、停止等与“生命周期”有关的方法组织到一块，这个接口就是Lifecycle。就好像人over了一般都要去殡仪 馆，还要做个户籍注销...

因此，一般容器的start方法里面，都会它所包含的子容器和其他实现了生命周期接口的组件的start方法，子容器又以此类推递归调用它们的子容器，整个tomcat就跑起来了~

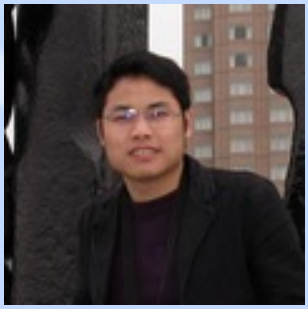
Lifecycle Listener

Lifecycle 当然不会仅仅有start和stop这么简单，它还引入了类似GUI编程的“事件——监听”机制。具体来说就是 org.apache.catalina.LifecycleListener 接口，内容非常简单，就一个 lifecycleEvent(LifecycleEvent event) 方法，用于接收相应的事件。如果你接触过GUI编程，特别是java swing，猜也能猜到tomcat怎么做了——首先注册listener，然后在start或者stop方法中，把相应的“事件”传递给所有已注册的监 听器

至于有哪些“事件”呢？上面的接口定义里面已经告诉我们了，不再啰嗦

总的来说，这一章的内容还是非常好理解的。

Tomcat源码研究



Tomcat源码研究

作者: ss1
http://ss1.javaeye.com