how tomcat works

中文版

2011-5-26

曹旭东

目录

简介

概览

本书所讲述内容适用于 tomcat 版本 *4·1·12* 至 *5·0·18*。

适合读者

- jsp/servlet 开发人员，想了解 tomcat 内部机制的 coder；
- 想加入 tomcat 开发团队的 coder；
- web 开发人员，但对软件开发很有兴趣的 coder；
- 想要对 tomcat 进行定制的 coder。

在阅读之前，希望你已经对 java 中的面向对象和 servlet 开发有所了解。

servlet 容器是如何工作的

servlet 容器是一个挺复杂的系统。但是，基本上，针对一个 servlet 的 request 请求，servlet 需要做一下三件事：

- 创建一个实现了 javax·servlet·ServletRequest 接口或 javax·servlet·http·ServletRequest 接口的 Request 对象，并用请求参数、请求头（headers）、cookies、查询字符串、uri 等信息填充该 Request 对象；
- 创建一个实现了 javax·servlet·ServletResponse 接口或 javax·servlet·http·ServletResponse 接口的 Response 对象；
- 调用相应的 servlet 的服务方法，将先前创建的 request 对象和 response 对象作为参数传入。接收请求的 servlet 从 request 对象中读取信息，并将返回值写入到 response 对象。

catalina 结构图

catalina 本身是一个成熟的软件，设计开发结构十分优雅，功能结构模块化。从 servlet 容器的功能角度看，catalian 可以划分为两大模块：connector 模块和 container 模块。

图表 1　Catalina 功能总体划分图

这里 connector 的功能是将用户请求与 container 连接。connector 的任务的是为每个接收到的 HTTP 请求建立 request 对象和 response 对象。然后，将处理过程交给 container 模块。container 模块从 connector 模块中接收到 request 对象和 response 对象，并负责调用相应的 servlet 的服务方法。

当然，上面只是对这个处理过程的简化描述。在处理过程中，container 还要做很多其他的事。例如，在调用 servlet 的服务方法前，它必须载入该 servlet，对用户身份进行认证（需要的话），更新该用户的 session 对象等。

tomcat 的版本 4 和版本 5

区别如下：

- tomcat5 支持 servlet2.4 和 jsp2.0 规范，tomcat4 支持 servlet2.3 和 jsp1.2 规范；
- tomcat5 默认的 connector 比 tomcat4 默认的 connector 执行效率更高；
- 在 tomcat 后台处理上，tomcat5 是共享线程的，而 tomcat4 的组件都使用各自的线程，从这方面讲，tomcat5 所消耗的资源更少；
- tomcat5 不需要映射组件来查找子组件，因此，代码量更少，更简单。

章节简介

本书共 20 章，前两章是简介内容。

第 1 章介绍了 HTTP 服务器是如何工作的，第 2 章介绍了一个简单的 servlet 容器。接下来两章着重于 connector 的说明，从第 5 章到第 20 章着个介绍 container 中的各个组件（component）。

第 1 章　一个简单的 Web 服务器

本章介绍了 java web 服务器是如何运行的，简要介绍 HTTP 协议。一个 java web 服务器会使用两个很重要的类，java.net.Socket 和 java.net.ServerSocket，并通过 HTTP 消息与客户端进行通信。

1.1 The Hypertext Transfer Protocol (HTTP)

HTTP 协议是基于请求-响应的协议，客户端请求一个文件，服务器对该请求进行响应。HTTP 使用 TCP 协议，默认使用 80 端口。最初的 HTTP 协议版本是 HTTP/0.9，后被 HTTP/1.0 替代。目前使用的版本是 HTTP/1.1，该协议定义于 Request for Comments (RFC) 2616。

在 HTTP 协议中，总是由主动建立连接、发送 HTTP 请求的客户端来初始化一个事务。服务器不负责连接客户端，或创建一个到客户端的回调连接（callback connection）。

1.2 HTTP Request

一个 HTTP 请求包含以下三部分：

- Method—Uniform Resource Identifier (URI)—Protocol/Version
- Request headers
- Entity body

举例如下（注意三部分之间要有空行）：

```
POST /examples/default.jsp HTTP/1.1
Accept: text/plain; text/html
Accept-Language: en-gb
Connection: Keep-Alive
Host: localhost
User-Agent: Mozilla/4.0 (compatible; MSIE 4.01; Windows 98)
Content-Length: 33
```

Content-Type：application/x-www-form-urlencoded

Accept-Encoding：gzip，deflate

lastName=Franks&firstName=Michael

每个 HTTP 请求都会有一个请求方法，HTTP1.1 中支持的方法包括，GET、POST、HEAD、OPTIONS、PUT、DELETE 和 TRACE。互联网应用中最常用的是 GET 和 POST。

URI 指明了请求资源的地址，通常是从网站更目录开始计算的一个相对路径，因此它总是以斜线“/”开头的。URL 实际上是 URI 的一种类型（参见 http://www.ietf.org/rfc/rfc2396.txt）。

请求头（header）中包含了一些关于客户端环境和请求实体（entity）的有用的信息。例如，客户端浏览器所使用的语言，请求实体信息的长度等。每个请求头使用 CRLF（回车换行符，“\r\n”）分隔。注意请求头的格式：

请求头名+英文空格+请求头值

请求头和请求实体之间有一个空白行（CRLF）。这是 HTTP 协议规定的格式。HTTP 服务器，以此确定请求实体是从哪里开始的。上面的例子中，请求实体是：

lastName=Franks&firstName=Michael

1.3 HTTP Response

与 HTTP Request 类似，HTTP Response 也由三部分组成：

- Protocol—Status code—Description
- Response headers
- Entity body

举例如下：

HTTP/1.1 200 OK

Server：Microsoft-IIS/4.0

Date：Mon，5 Jan 2004 13:13:33 GMT

Content-Type：text/html

```
Last-Modified: Mon, 5 Jan 2004 13:13:12 GMT
Content-Length: 112

<html>
<head>
<title>HTTP Response Example</title>
</head>
<body>
Welcome to Brainy Software
</body>
</html>
```

注意响应实体（entity）与响应头（header）之间有一个空白行（CRLF）。

1.4 Socket 类

socket 通信的实例代码如下：

```
Socket socket = new Socket("127.0.0.1", "8080");      //建立连接
OutputStream os = socket.getOutputStream();   //获取输出流
boolean autoflush = true;
PrintWriter out = new PrintWriter( socket.getOutputStream(), autoflush);  //设置自动 flush
BufferedReader in = new BufferedReader( new InputStreamReader( socket.getInputstream() ));

// send an HTTP request to the web server
out.println("GET /index.jsp HTTP/1.1");          //拼装 HTTP 请求信息
```

```
out·println("Host: localhost:8080");
out·println("Connection: Close");
out·println();

// read the response
boolean loop = true;
StringBuffer sb = new StringBuffer(8096);
while (loop) {
  if ( in·ready() ) {
    int i=0;
     while (i!=-1) {
       i = in·read();
       sb·append((char) i);
    }
    loop = false;
  }
  Thread·currentThread()·sleep(50);    //由于是阻塞写入，暂停 50ms，保证可以写入。
}

// display the response to the out console
System·out·println(sb·toString());
socket·close();
```

1.5 ServerSocket 类

Socket 类表示一个客户端 socket，相应的 ServerSocket 类表示了一个服务器端应用。服务器端 socket 需要等待来自客户端的连接请求。一旦 ServerSocket 接收到来自客户端的连接请求，它会实例化一个 Socket 类的对象来处理与客户端的通信。

1.6 应用举例

该程序包括三个部分，HttpServer、Request 和 Response。该程序只能发送静态资源，如 HTML 文件，图片文件，但不会发送响应头信息。

代码文件如下：

Response.java　HttpServer.java　Request.java

第 2 章　一个简单的 servlet 容器

2.1　简述

本章通过两个小程序说明如何开发一个自己的 servlet 容器。第一个程序的设计非常简单，仅仅用于说明 servlet 容器是如何运行的。第二个稍微复杂一点点，会调用第一个程序。这两个 servlet 容器都能处理简单的 servlet 和静态资源。PrimitiveServlet 类可用于测试 servlet 容器。 PrimitiveServlet.java

注意，每一章内容中都会调用前一章的应用程序，直到第 17 章完成一个完整的 servlet 容器。

2.2　javax.servlet.Servlet 接口

Servlet 接口需要实现下面的 5 个方法：

- public void init(ServletConfig config) throws ServletException
- public void service(ServletRequest request, ServletResponse response) throws ServletException, java·io·IOException
- public void destroy()
- public ServletConfig getServletConfig()
- public java·lang·String getServletInfo()

在某个 servlet 类被实例化之后，init 方法由 servlet 容器调用。servlet 容器只调用该方法一次，调用后则可以执行服务方法了。在 servlet 接收任何请求之前，必须是经过正确初始化的。

当一个客户端请求到达后，servlet 容器就调用相应的 servlet 的 service 方法，并将 Request 和 Response 对象作为参数传入。在 servlet 实例的生命周期内，service 方法会被多次调用。

在将 servlet 实例从服务中移除前，会调用 servlet 实例的 destroy 方法。一般情况下，在服务器关闭前，会发生上述情况，servlet 容器会释放内存。只有当 servlet 实例的 service 方法中所有的线程都退出或执行超时后，才会调用 destroy 方法。当容器调用了 destroy 方法精辟，就不会再调用 service 方法了。

2.3　Application 1

下面从 servlet 容器的角度观察 servlet 的开发。在一个全功能 servlet 容器中，对 servlet 的每个 HTTP 请求

来说，容器要做下面几件事：

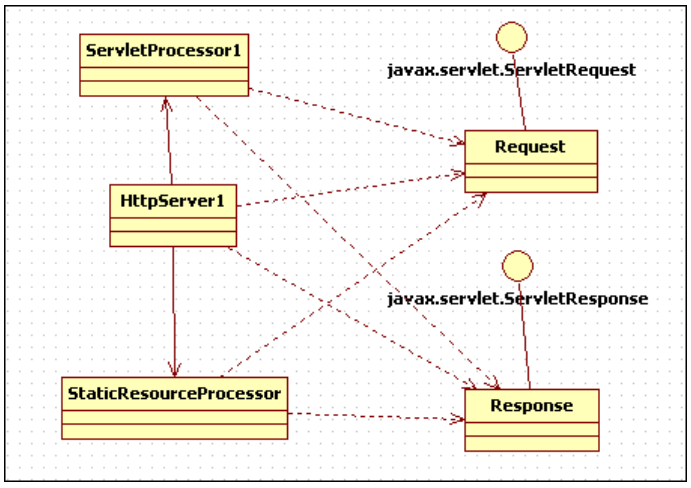
- 当第一次调用 servlet 时，要载入 servlet 类，调用 init 方法（仅此一次）；
- 针对每个 request 请求，创建一个 Request 对象和一个 Resposne 对象；
- 调用相应的 servlet 的 service 方法，将 Request 对象和 Response 对象作为参数传入；
- 当关闭 servlet 时，调用 destroy 方法，并卸载该 servlet 类。

这里建立的 servlet 容器是一个很小的容器，没有实现所有的功能。因此，它仅能运行非常简单的 servlet 类，无法调用 servlet 的 init 和 destroy 方法。它能执行功能如下所示：

- 等待 HTTP 请求；
- 创建 Request 和 Response 对象；
- 若请求的是一个静态资源，则调用 StaticResourceProcessor 对象的 process 方法，传入 request 和 response 对象；
- 若请求的是 servlet，则载入相应的 servlet 类，调用 service 方法，传入 request 对象和 response 对象。

注意，在这个 servlet 中，每次请求 servlet 都会载入 servlet 类。

该程序包括 *6* 个类：HttpServer*1*、Request、Response、StaticResourceProcessor、ServletProcessor*1*、Constants。



图表 2 简单 servlet 容器 1 的 UML 图

该程序的入口点（静态 main 方法）在类 HttpServer*1* 中。main 方法中创建 HttpServer*1* 的实例，饭后调用其 await 方法。await 方法等待 HTTP 请求，为接收到的每个请求创建 request 和 response 对象，将它们分发到一个 StaticResourceProcessor 类或 ServletProcessor 类的实例。

2·3·1 HttpServer*1* 类

代码清单如下：

HttpServer1.java

该类与第一章的 HttpServer 类类似，只是完善了对静态资源和动态资源的处理。

2·3·2 Request 类

代码清单如下：

Request.java

该类实现了 javax·servlet·ServletRequest 接口，但并不返回实际内容。

2·3·3 Response 类

实现了 javax·servlet·ServletResponse 接口，大部分方法都返回一个空值，除了 getWriter 方法以外。

Response.java

在 getWriter 方法中，PrintWriter 类的构造函数的第二个参数表示是否启用 autoFlush。因此，若是设置为 false，则如果是 servlet 的 service 方法的最后一行调用打印方法，则该打印内容不会被发送到客户端。这个 bug 会在后续的版本中修改。

2·3·4 StaticResourceProcessor 类

该类用于处理对静态资源的请求。

StaticResourceProcessor.java

2·3·5 ServletProcessor1 类

该类用于处理对 servlet 资源的请求。

ServletProcessor1.java

该类很简单，只有一个 process 方法。载入 servlet 时使用的是 UrlClassLoader 类，它是 ClassLoader 类的直接子类，有三种构造方法。

- public URLClassLoader(URL[] urls);

  参数为一个 Url 对象的数组，每个 url 指明了从哪里查找 servlet 类。若某个 Url 是以“/”结尾的，则认为它是一个目录；否则，认为它是一个 jar 文件，必要时会将它下载并解压。
  注：在 servlet 容器中，查找 servlet 类的位置称为 repository。
  在我们的应用程序中，servlet 容器只需要查找一个 repository，在工作目录的 webroot 路径下。

- public URL(URL context, java·lang·String spec, URLStreamHandler hander) throws MalformedURLException

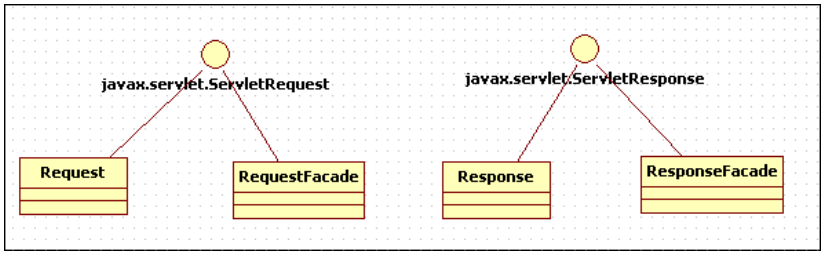
- public URL(java·lang·String protocol, java·lang·String host, java·lang·String file) throws

MalformedURLException

2.4 Application 2

在之前的程序中，有一个严重的问题，必须将 ex02·pyrmont·Request 和 ex02·pyrmont·Response 分别转型为 javax·servlet·ServletRequest 和 javax·servlet·ServletResponse，再作为参数传递给具体的 servlet 的 service 方法。这样并不安全，熟知 servlet 容器的人可以将 ServletRequest 和 ServletResponse 类向下转型为 Request 和 Response 类，并执行 parse 和 sendStaticResource 方法。

一种解决方案是将这两个方法的访问修饰符改为默认的（即，default），这样就可以避免包外访问。另一种更好的方案是使用外观设计模式。uml 图如下：



图表 3 Facade 模式

在第二个应用程序中，添加了两个 façade 类，RequestFacade 和 ResponseFacade。RequestFacade 类实现了 ServletRequest 接口，通过在其构造方法中传入一个 ServletRequest 类型引用的 Request 对象来实例化。ServletRequest 接口中每个方法的实现都会调用 Request 对象的相应方法。但是，ServletRequest 对象本身是 private 类型，这样就不能从类的外部进行访问。这里也不再将 Request 对象向上转型为 ServletRequest 对象，而是创建一个 RequestFacade 对象，并把它传给 service 方法。这样，就算是将在 servlet 中获取了 ServletRequest 对象，并向下转型为 RequestFacade 对象，也不能再访问 ServletRequest 接口中的方法了，就可以避免前面所说的安全问题。

RequestFacade·java 代码如下：

RequestFacad.java

注意它的构造函数，接收一个 Request 对象，然后向上转型为 ServletRequest 对象，赋给其 private 成员变量 request。该类的其他方法中，都是调用 request 的相应方法实现的，这样就将 ServletRequest 完整的封装得 RequestFacade 中了。

同理，ResponseFacade 类也是这样的。

Application 2 中的类包括，HttpServer2、Request、Response、StaticResourceProcessor、ServletProcessor2、Constants。

第 3 章　连接器（Connector）

3.1　概述

在简介一章里说明了，tomcat 由两大模块组成：连接器（connector）和容器（container）。本章将使用连接器来增强 application 2 的功能。一个支持 servlet2·3 和 2·4 规范的连接器必须要负责创建 javax·servlet·http·HttpServletRequest 和 javax·servlet·http·HttpServletResponse 实例，并将它们作为参数传递给要调用的某个的 servlet 的 service 方法。在第 2 章中的 servlet 容器仅仅能运行实现了 javax·servlet·Servlet 接口，并想 service 方法中传入了 javax·servlet·ServletRequest 和 javax·servlet·ServletResponse 实例的 servlet。由于连接器并不知道 servlet 的具体类型（例如，该 servlet 是否 javax·servlet·Servlet 接口，还是继承自 javax·servlet·GenericServlet 类，或继承自 javax·servlet·http·HttpServlet 类），因此连接器总是传入 HttpServletRequest 和 HttpServletResponse 的实例对象。

本章中所要建立的 connector 实际上是 tomcat4 中的默认连接器（将在第 4 章讨论）的简化版。本章中，connector 和 container 将分离开。

在开始说明本章的程序之前，先花点时间介绍下 org·apache·catalina·util·StringManager 类，它被 tomcat 用来处理不同模块内错误信息的国际化。在本章中，也是这样用的。

3.2　StringManager 类

tomcat 将错误信息写在一个 properties 文件中，这样便于读取和编辑。但若是将所有类的错误信息都写在一个 properties 文件，优惠导致文件太大，不便于读写。为避免这种情况，tomcat 将 properties 文件按照不同的包进行划分，每个包下都有自己的 properties 文件。例如，org·apache·catalina·connector 包下的 properties 文件包含了该包下所有的类中可能抛出的错误信息。每个 properties 文件都由一个 org·apache·catalina·util·StringManager 实例来处理。在 tomcat 运行时，会建立很多 StringManager 类的实例，每个实例对应一个 properties 文件。

当包内的某个类要查找错误信息时，会先获取对应的 StringManager 实例。StringManager 被设计为在包内是共享的一个单例，功过 hashtable 实现。如下面的代码所示：

```
private static Hashtable managers = new Hashtable();
public synchronized static StringManager
    getManager(String packageName) {
```

```
        StringManager mgr = (StringManager)managers.get(packageName);      if (mgr == null) {
          mgr = new StringManager(packageName);
          managers.put(packageName, mgr);
        }
        return mgr;
    }
    StringManager sm = StringManager.getManager("ex03.pyrmont.connector.http");
```

3.3 Application

从本章开始，每章的应用程序都会按照模块进行划分。本章的应用程序可分为 3 个模块：connector、startup、core。

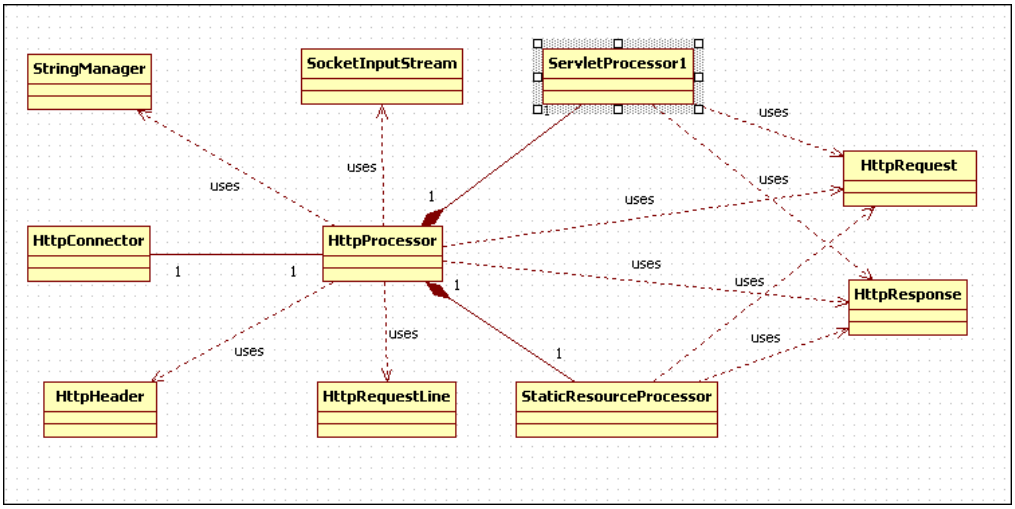
startup 模块仅包括一个 StartUp 类，负责启动应用程序。

connector 模块的类可分为以下 5 个部分：

- 连接器及其支持类（HttpConnector 和 HttpProcessor）；
- 表示 http 请求的类（HttpRequest）及其支持类；
- 表示 http 响应的类（HttpResponse）及其支持类；
- 外观装饰类（HttpRequestFacade 和 HttpResponseFacade）；
- 常量类。

core 模块包括 ServletProcessor 类和 StaticResourceProcessor 类。

下面是程序的 uml 图：



图表 4　application 的 uml 图

相比于第 2 章中的程序，HttpServer 在本章中被分成了 HttpConnector 和 HttpProcessor 两个类。Request 和 Response 分别被 HttpRequest 和 HttpResponse 代替。此外，本章的应用程序中还使用了一些其他的类。

在第 2 章中，HttpServer 负责等待 http 请求，并创建 request 和 response 对象。本章中，等待 http 请求的工作由 HttpConnector 完成，创建 request 和 response 对象的工作由 HttpProcessor 完成。

本章中，http 请求用 HttpRequest 对象表示，该类实现了 javax·servlet·http·HttpServletRequest 接口。一个 HttpRequest 对象在传给 servlet 的 service 方法前，会被转型为 HttpServletRequest 对象。因此，需要正确设置每个 HttpRequest 对象的成员变量，方便 servlet 使用。需要设置的值包括，uri，请求字符串，参数，cookie 和其他一些请求头信息等。由于连接器并不知道 servlet 中会使用那些变量，素以它会将从 http 请求中获取的变量都设置到 HttpRequest 对象中。但是，处理一个 http 请求会设计到一些比较耗时的操作，如字符串处理等。因此，若是 connector 仅仅传入 servlet 需要用到的值就会节省很多时间。tomcat 的默认 connector 对这些值的处理是等到 servlet 真正用到的时候才处理的。

tomcat 的默认 connector 和本程序的 connector 通过 SocketInputStream 类来读取字节流，可通过 socket 的 getInputStream 方法来获取该对象。它有两个重要的方法 readRequestLine 和 readHeader。readRequestLine 方法返回一个 http 请求的第一行，包括 uri，请求方法和 http 协议版本。从 socket 的 inputStream 中处理字节流意味着要从头读到尾（即不能返回来再读前面的内容），因此，readRequestLine 方法一定要在 readHeader 方法前调用。readRequestLine 方法返回的是 HttpRequestLine 对象，readHeader 方法返回的是 HttpHeader 对象（key-value 形式）。获取 HttpHeader 对象时，应重复调用 readHeader 方法，直到再也无法获取到。

HttpProcessor 对象负责创建 HttpRequest 对象，并填充它的成员变量。在其 parse 方法中，将请求行（request line）和请求头（request header）信息填充到 HttpRequest 对象中，但并不会填充请求体（request body）和查询字符串（query string）。

3·3·1 启动

在 Bootstrap 类的 main 方法内实例化一个 HttpConnector 类的对象，并调用其 start 方法就可以启动应用程序。

Bootstrap.java

3·3·2 connector

HttpConnector 类实现了 java·lang·Runnable 接口，这样它可以专注于自己的线程。启动应用程序时，会创建一个 HttpConnector 对象，其 run 方法会被调用。其 run 方法中是一个循环体，执行以下三件事：

- 等待 http 请求；
- 为每个请求创建一个 HttpPorcessor 对象；
- 调用 HttpProcessor 对象的 process 方法。

HttpProcessor 类的 process 方法从 http 请求中获取 socket。对每个 http 请求，它要做一下三件事：

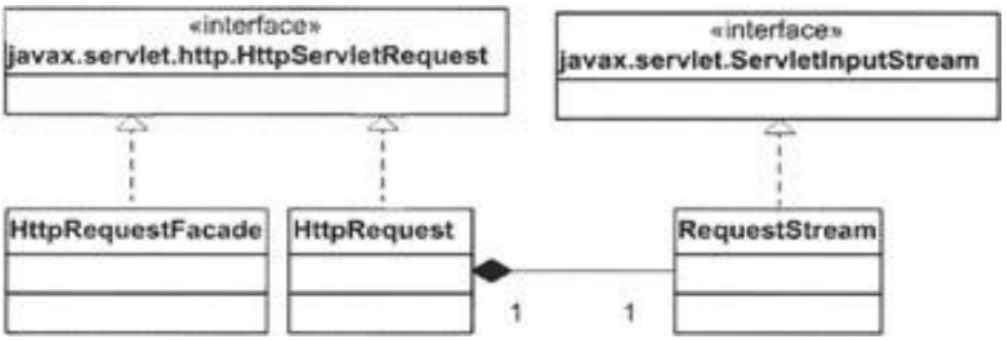
- 创建一个 HttpRequest 对象和一个 HttpResponse 对象；
- 处理请求行（request line）和请求头（request headers），填充 HttpRequest 对象；
- 将 HttpRequest 对象和 HttpResponse 对象传给 ServletProcessor 或 StaticResourceProcessor 的

process 方法。

HttpConnector.java　　HttpProcessor.java

3.3.3 创建 HttpRequest 对象

HttpRequest 类实现了 javax.servlet.http.HttpServletRequest 接口。其伴随的外观类是 HttpRequestFacade。日 uml 图如下所示：



图表 5　HttpRequest 类的 uml 图

其中 HttpRequest 的很多方法都是空方法，但已经可以从 hhtp 请求中获取 headers，cookies 和参数信息了。这三种数据分别以 HashMap、ArrayList 和 ParameterMap（后面介绍）存储。

3.3.3.1 SocketInputStream 类

本章的应用程序中，使用的 SocketInputStream 就是 org.apache.catalina.connector.http.SocketInputStream。该类提供了获取请求行（request line）和请求头（request header）的方法。通过传入一个 InputStream 对象和一个代表缓冲区大小的整数值来创建 SocketInputStream 对象。

3.3.3.2 解析请求行（request line）

HttpProcessor 的 process 调用其私有方法 parseRequest 来解析请求行（request line，即 http 请求的第一行）。下面是一个请求行（request line）的例子：

GET /myApp/ModernServlet?userName=tarzan&password=pwd HTTP/1.1

注意：“GET”后面和“HTTP”前面各有一个空格。

请求行的第 2 部分是 uri 加上查询字符串。在上面的例子中，uri 是：

/myApp/ModernServlet

问号后面的都是查询字符串，这里是：

userName=tarzan&password=pwd

在 servlet/jsp 编程中，参数 jsessionid 通常是嵌入到 cookie 中的，也可以将其嵌入到查询字符串中。parseRequest 方法的具体内容参见代码。

HttpProcessor.java

3.3.3.3 解析请求头（request header）

请求头（request header）由 HttpHeader 对象表示。可以通过 HttpHeader 的无参构造方法建立对象，并将其作为参数传给 SocketInputStream 的 readHeader 方法，该方法会自动填充 HttpHeader 对象。parseHeader 方法内有一个循环体，不断的从 SocketInputStream 中读取 header 信息，直到读完。获取 header 的 name 和 value 值可使用下米娜的语句：

```
String name = new String(header.name, 0, header.nameEnd);
String value = new String(header.value, 0, header.valueEnd);
```

获取到 header 的 name 和 value 后，要将其填充到 HttpRequest 的 header 属性（hashMap 类型）中：

```
request.addHeader(name, value);
```

其中某些 header 要设置到 request 对象的属性中，如 contentLength 等。

3.3.3.4 解析 cookie

cookie 是由浏览器作为请求头的一部分发送的，这样的请求头的名字是 cookie，它的值是一个 key-value 对。举例如下：

```
Cookie: userName=budi; password=pwd;
```

对 cookie 的解析是通过 org.apache.catalina.util.RequestUtil 类的 parseCookieHeader 方法完成的。该方法接受一个 cookie 头字符串，返回一个 javax.servlet.http.Cookie 类型的数组。方法实现如下：

```
public static Cookie[] parseCookieHeader(String header) {
    if ((header == null) || (header.length 0 < 1) )
        return (new Cookie[0]);
    ArrayList cookies = new ArrayList();
    while (header.length() > 0) {
```

```
            int semicolon = header.indexOf(';');
            if (semicolon < 0)
                semicolon = header.length();
            if (semicolon == 0)
                break;
            String token = header.substring(0, semicolon);
            if (semicolon < header.length())
                header = header.substring(semicolon + 1);
            else
                header = "";
            try {
                int equals = token.indexOf('=');
                if (equals > 0) {
                    String name = token.substring(0, equals).trim();
                    String value = token.substring(equals+1).trim();
                    cookies.add(new Cookie(name, value));
                }
            } catch (Throwable e) { ; }
        }
        return ((Cookie[]) cookies.toArray (new Cookie [cookies.size ()]));
    }
```

3.3.3.5 获取参数

在调用 javax.servlet.http.HttpServletRequest 的 getParameter、getParameterMap、getParameterNames

或 getParameterValues 方法之前，都不会涉及到对查询字符串或 http 请求体的解析。因此，这四个方法的实现都是先调用 parseParameter 方法。

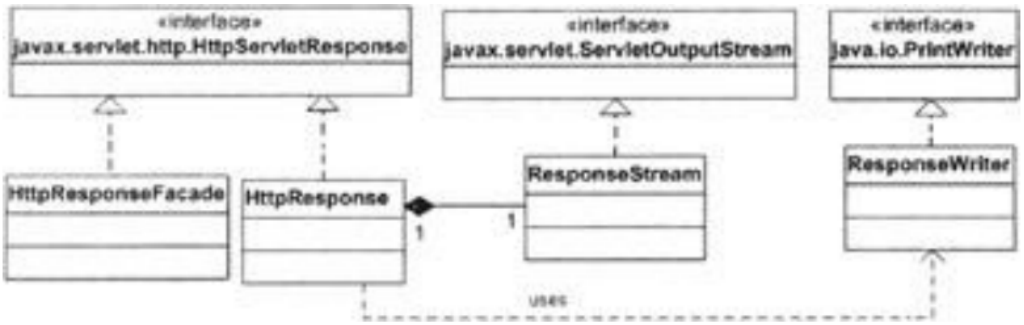
参数只会被解析一次，因为，HttpRequest 类会设置一个标志位表明是否已经完成参数解析了。参数可以出现在查询字符串或请求体中。若用户使用的 GET 方法，则所有的参数都会在查询字符串中；若是使用的 POST 方法，则请求体中也可能会有参数。所有的 key-value 的参数对都会存储在 HashMap 中，其中的值是不可修改的。tomcat 中使用的是一个特殊的 hashmap 类，org·apache·catalina·util·ParameterMap。

ParameterMap 类继承自 java·util·HashMap，使用一个标志位来表示锁定。如果该标志位为 false，则可以对其中的 key-value 进行添加、修改、删除操作，否则，执行这些操作时，会抛出 IllegalStateException 异常。代码如下：

ParameterMap.java

3.3.3.6 创建 HttpResponse 对象

HttpResponse 类继承自 javax·servlet·http·HttpServletResponse，其相应的外观类是 HttpResponseFacade。其 uml 图如下所示：



图表 6 HttpResponse 及其外观类的 uml 图示

在第 2 章中，HttpResponse 的功能有限，例如，它的 getWriter 方法返回的 java·io·PrintWriter 对象执行了 print 方法时，并不会自动 flush。本章的程序将解决此问题。在此之前，先说明一下什么是 Writer。

在 servlet 中，可以使用 PrintWriter 对象想输出流中写字符。可以使用任意编码格式，但在发送的时候，实际上都是字节流。

在本章中，将要使用的是 ex03·pyrmont·connector·ResponseStream 类作为 PrintWriter 的输出流。该类直接继承自 java·io·OutputStream 类。

类 ex03·pyrmont·connector·ResponseWriter 继承自 PrintWriter，重写了其 print 和 println 方法，实现自动 flush。因此，本章适用 ResponseWriter 作为输出对象。

示例代码如下：

```
public PrintWriter getWriter() throws IOException {
```

```
    ResponseStream newStream = new ResponseStream(this);
    newStream.setCommit(false);
    OutputStreamWriter osr =
      new OutputStreamWriter(newStream, getCharacterEncoding());
    writer = new ResponseWriter(osr);
    return writer;
  }
```

3.3.3.7 静态资源处理器和 servlet 处理器

本章的 servlet 处理器和第 2 章的 servlet 处理器类似，都只有一个 process 方法。但个，本章中，process 方法接收的参数类型为 HttpRequest 和 HttpResponse。方法签名如下：

```
public void process(HttpRequest request, HttpResponse response);
```

此外，process 使用了 request 和 response 的外观类，并在调用了 servlet 的 service 方法后，再调用 HttpResponse 的 finishResponse 方法。示例代码如下：

```
servlet = (Servlet) myClass.newInstance();
HttpRequestFacade requestPacade = new HttpRequestFacade(request);
HttpResponseFacade responseFacade = new  HttpResponseFacade(response);
servlet.service(requestFacade, responseFacade);
((HttpResponse) response).finishResponse();
```

第 4 章　tomcat 的默认连接器

4.1　简介

第三章的连接器只是一个学习版，是为了介绍 tomcat 的默认连接器而写。第四章会深入讨论下 tomcat 的默认连接器（这里指的是 tomcat4 的默认连接器，现在该连接器已经不推荐使用，而是被 Coyote 取代）。

tomcat 的连接器是一个独立的模块，可被插入到 servlet 容器中。目前已经有很多连接器的实现，包括 Coyote，mod_jk，mod_jk2，mod_webapp 等。tomcat 的连接器需要满足以下要求：

（1）实现 org·apache·catalina·Connector 接口；

（2）负责创建实现了 org·apache·catalina·Request 接口的 request 对象；

（3）负责创建实现了 org·apache·catalina·Response 接口的 response 对象。

tomcat4 的连接器与第三章实现的连接器类似，等待 http 请求，创建 request 和 response 对象，调用 org·apache·catalina·Container 的 invoke 方法将 request 对象和 response 对象传入 container。在 invoke 方法中，container 负责载入 servlet 类，调用其 call 方法，管理 session，记录日志等工作

tomcat 的默认连接器中有一些优化操作没有在 chap3 的连接器中实现。首先是提供了一个对象池，避免频繁创建一些创佳代价高昂的对象。其次，默认连接器中很多地方使用了字符数组而非字符串。

本章的程序是实现一个使用默认连接器的 container。但，本章的重点不在于 container，而是 connector。另一个需要注意的是，默认的 connector 实现了 HTTP1·1，也可以服务 HTTP1·0 和 HTTP0·9 的客户端。

本章以 HTTP1·1 的 3 个新特性开始，这对于理解默认 connector 的工作机理很重要。然后，要介绍 org·apache·catalina·Connector 接口。

4.2　HTTP1.1 的新特性

4·2·1　持久化连接

在 http1·1 之前，当服务器端将请求的资源返回后，就会断开与客户端的连接。但是，网页上会包含一些其他资源，如图片，applet 等。因此，客户端请求资源后，浏览器还需要下载页面引用的资源。如果页面和资源使用是通过不同的连接下载的，那么整个处理过程会很慢。因此，HTTP1·1 引入了持久化连接。

使用持久化连接，当客户端下载页面后，服务器并不会立刻关闭连接，而是等待浏览器请求页面要引用的页

面资源。这样，页面和资源使用同一个连接下载，这样就节省很多的工作和时间。

HTTP1.1 中默认使用持久化连接，而客户端也可以主动使用。方法是在请求头中加入下面信息：

connection: keep-alive

4.2.2 编码

建立了持久化连接后，服务器可以使用该连接发送多个资源，而客户端也可以使用该连接发送多个请求。发送方在发送消息时就要附带上发送内容的长度，这样，接收方才能知道如何解释这些字节。但，通常的情况是，发送方并不知道要发送多少字节。例如，container 可以在接收到一些字节后就向客户端返回一些信息，而不必等所有的字节都接收后再返回响应。因此，必须有某种方法告诉接收方如何解释字节流。

其实，即使没有发出多个请求，服务器或客户端也不需要知道有多少字节要发送。在 HTTP1.0 中，服务器可以不管 content-length 头信息，尽管往连接中写响应内容就行。这种情况下，客户端就一直读内容，直到读方法返回-1，此时表示已经没有更多信息了。

在 HTTP1.1 中，使用了一个特殊的头信息，transfer-encoding，表明字节流按照块发送。每个块的长度以 16 进制表示，后跟 CRLF，然后是发送的内容。每次事务以一个 0 长度的块为结束标识。

例如，你想发送两个块，一个 29 字节，一个 9 字节。发送格式如下：

1D\r\n

I'm as helpless as a kitten u

9\r\n

p a tree.

0\r\n

4.2.3 状态码 100 的使用

在 HTTP1.1 中，客户端在发送请求体之前，可能会先向服务器端发送这样的头信息：

Expect: 100-continue

然后等待服务器端的确认。

当客户端准备发送一个较长的请求体，而不确定服务端是否会接收时，就可能会发送上面的头信息。而服务器若是可以接受，则可以对此头信息进行响应，返回：

HTTP/1.1 100 Continue

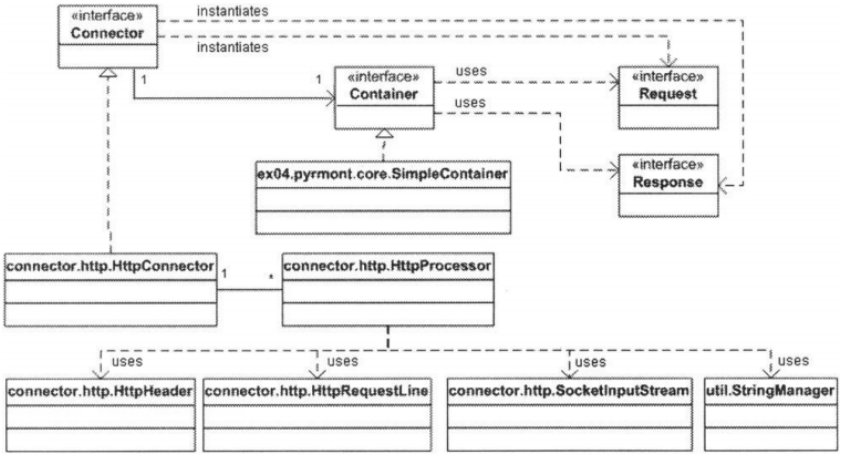
注意，返回内容后面要加上 CRLF。

4.3 Connector 接口

tomcat 的 connector 必须实现 org.apache.catalina.Connector 接口。该接口有很多方法，最重要的是 getContainer，setContainer，createRequest 和 createResponse。

setContainer 方法用于将 connector 和 container 联系起来，getContainer 则可以返回响应的 container，createRequest 和 createResponse 则分别负责创建 request 和 response 对象。

org.apache.catalina.connector.http.HttpConnector 类是 Connector 接口的一个实现，将在下一章讨论。响应的 uml 图如下所示：



图表 7 默认连接器的 uml 示意图

注意，connector 和 container 是一对一的关系，而 connector 和 processor 是一对多的关系。

4.4 HttpConnector 类

在第三章中，已经实现了一个与 org.apache.catalina.connector.http.HttpConnector 类似的简化版 connector。它实现了 org.apache.catalina.Connector 接口，java.lang.Runnable 接口（确保在自己的线程中运行）和 org.apache.catalina.Lifecycle 接口。Lifecycle 接口用于维护每个实现了该接口的 tomcat 的组件的生命周期。

Lifecycle 具体内容将在第六章介绍。实现了 Lifecycle 接口后，当创建一个 HttpConnector 实例后，就应该调用其 initialize 方法和 start 方法。在组件的整个生命周期内，这两个方法只应该被调用一次。下面要介绍一些与第三章不同的功能：创建 ServerSocket，维护 HttpProcessor 池，提供 Http 请求服务。

4.4.1 创建 ServerSocket

HttpConnector 的 initialize 方法会调用一个私有方法 open，返回一个 java.net.ServerSocket 实例，赋值给

成员变量 serverSocket。这里并没有直接调用 ServerSocket 的构造方法，而是用过 open 方法调用 ServerSocket 的一个工厂方法来实现。具体的实现方式可参考 ServerSocketFactory 类和 DefaultServerSocketFactory 类（都在 org·apache·catalina·net 包内）。

4·4·2 维护 HttpProcessor 对象池

在第三章的程序中，每次使用 HttpProcessor 时，都会创建一个实例。而在 tomcat 的默认 connector 中，使用了一个 HttpProcessor 的对象池， 其中的每个对象都在其自己的线程中使用。因此，connector 可同时处理多个 http 请求。

HttpConnector 维护了一个 HttpProcessor 的对象池，避免了频繁的创建 HttpProcessor 对象。该对象池使用 java·io·Stack 实现。

在 HttpConnector 中，创建的 HttpProcessor 数目由两个变量决定：minProcessors 和 maxProcessors。

```
protected int minProcessors = 5;

private int maxProcessors = 20;
```

默认情况下，minProcessors=5，maxProcessors=20，可通过其 setter 方法修改。

初始化的时候，HttpConnector 会创建 minProcessors 个 HttpProcessor 对象。若不够用就继续创建，直到到达 maxProcessors 个。此时，若还不够，则后达到的 http 请求将被忽略。若是不希望对 maxProcessors 进行限制，可以将其置为负数。此外，变量 curProcessors 表示当前已有的 HttpProcessor 实例数目。

下面是 start 方法中初始化 HttpProcessor 对象的代码：

```
while (curProcessors < minProcessors) {

  if ((maxProcessors > 0) && (curProcessors >= maxProcessors))

    break;

  HttpProcessor processor = newProcessor();

  recycle(processor);

}
```

其中 newProcessor 方法负责创建 HttpProcessor 实例，并将 curProcessors 加 1。recycle 方法将新创建的 HttpProcessor 对象入栈。

每个 HttpProcessor 对象负责解析请求行和请求头，填充 request 对象。因此，每个 HttpProcessor 对象都关联一个 request 对象和 response 对象。HttpProcessor 的构造函数会调用 HttpConnector 的 createRequest 方法和 createResponse 方法。

4.4.3 提供 Http 请求服务

HttpConnector 类的主要业务逻辑在其 run 方法中（例如第三章的程序中那样）。run 方法中维持一个循环体，该循环体内，服务器等待 http 请求，直到 HttpConnector 对象回收。

```
while (!stopped) {

 Socket socket = null;

 try {

   socket = serverSocket.accept();

 …
```

对于每个 http 请求，通过调用其私有方法 createProcessor 获得一个 HttpProcessor 对象。这里，实际上是从 HttpProcessor 的对象池中拿一个对象。

注意，若是此时对象池中已经没有空闲的 HttpProcessor 实例可用，则 createProcessor 返回 null。此时，服务器会直接关闭该连接，忽略该请求。如代码所示：

```
if (processor == null) {

        try {

          log(sm.getString("httpConnector.noProcessor"));

          socket.close();

        }

        …

        continue;
```

若是 createProcessor 方法返回不为空，则调用该 HttpProcessor 实例的 assign 方法，并将客户端 socket 对象作为参数传入：

```
processor.assign(socket);
```

这时，HttpProcessor 实例开始读取 socket 的输入流，解析 http 请求。这里有一个重点，assign 方法必须立刻返回，不能等待 HttpProcessor 实例完成解析再返回，这样才能处理后续的 http 请求。由于每个 HttpProcessor 都可以使用它自己的线程进行处理，所以这并不难实现。

4.5 HttpProcessor 类

HttpProcessor 类与第三章中的实现相类似。本章讨论下它的 assign 方法是如何实现异步功能的（即可同时处理多个 http 请求）。

在第三章中，HttpProcessor 类运行在其自己的线程中。在处理下一个请求之前，它必须等待当前请求的处理完成。下面是第三章中的 HttpConnector 类的 run 方法的部分代码：

```
public void run() {
  ...
 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);
 }
}
```

可以 process 方法是同步的。但是在 tomcat 的默认连接器中，HttpProcessor 实现了 java.lang.Runnable 接口，每个 HttpProcessor 的实例都可以在其自己的线程中运行，成为“处理器线程”（“processor thread”）。HttpConnector 创建每个 HttpProcessor 实例时，都会调用其 start 方法，启动其处理器线程。下面的代码显示了 tomcat 默认 connector 的 HttpProcessor 实例的 run 方法：

```
public void run() {
   // Process requests until we receive a shutdown signal
```

```
        while (!stopped) {
            // Wait for the next socket to be assigned
            Socket socket = await();
            if (socket == null)
                continue;
            // Process the request from this socket
            try {
                process(socket);
            } catch (Throwable t) {
                log("process.invoke", t);
            }
            // Finish up this request
            connector.recycle(this);
        }
        // Tell threadStop() we have shut ourselves down successfully
        synchronized (threadSync) {
            threadSync.notifyAll();
        }
    }
```

这个循环体做的事是：获取 socket，处理它，调用 connector 的 recycle 方法将当前的 HttpProcessor 入栈。recycle 方法的实现是：

```
    void recycle(HttpProcessor processor) {
        processors.push(processor);
    }
```

注意，循环体在执行到 await 方法时会暂停当前处理器线程的控制流，直到获取到一个新的 socket。换句话说，在 HttpConnector 调用 HttpProcessor 实例的 assign 方法前，程序会一直等下去。但是，assign 方法并不是在当前线程中执行的，而是在 HttpConnector 的 run 方法中被调用的。这里称 HttpConnector 实例所在的线程为连接器线程（connector thread）。那么，assign 方法是如何通知 await 方法它已经被调用了呢？方法是使用一个成为 available 的 boolean 变量和 java·lang·Object 的 wait 和 notifyAll 方法。

注意，wait 方法会暂停本对象所在的当前线程，使其处于等待状态，直到另一线程调用了该对象的 notify 或 notifyAll 方法。

下面是 HttpProcessor 的 assign 方法和 await 方法的实现代码：

```
synchronized void assign(Socket socket) {

    // Wait for the processor to get the previous socket

    while (available) {

      try {

        wait();

      }

      catch (InterruptedException e) {

      }

    }

    // Store the newly available Socket and notify our thread

    this·socket = socket;

    available = true;

    notifyAll();

    …

}

private synchronized Socket await() {

    // Wait for the Connector to provide a new Socket

     while (!available) {

      try {
```

```
            wait();
          }
          catch (InterruptedException e) {
          }
        }

        // Notify the Connector that we have received this Socket
        Socket socket = this.socket;
        available = false;
        notifyAll();
        if ((debug >= 1) && (socket != null))
          log(" The incoming request has been awaited");
        return (socket);
    }
```

当处理器线程刚刚启动时，available 值为 false，线程在循环体内 wait，直到任意一个线程调用了 notify 或 notifyAll 方法。也就是说，调用 wait 方法会使线程暂定，直到连接器线程调用 HttpProcessor 实例的 notify 或 notifyAll 方法。

当一个新 socket 被设置后，连接器线程调用 HttpProcessor 的 assign 方法。此时 available 变量的值为 false，会跳过循环体，该 socket 对象被设置到 HttpProcessor 实例的 socket 变量中。然后连接器变量设置了 available 为 true，调用 notifyAll 方法，唤醒处理器线程。此时 available 的值为 true，跳出循环体，将 socket 对象赋值给局部变量，将 available 设置为 false，调用 notifyAll 方法，并将给 socket 返回。

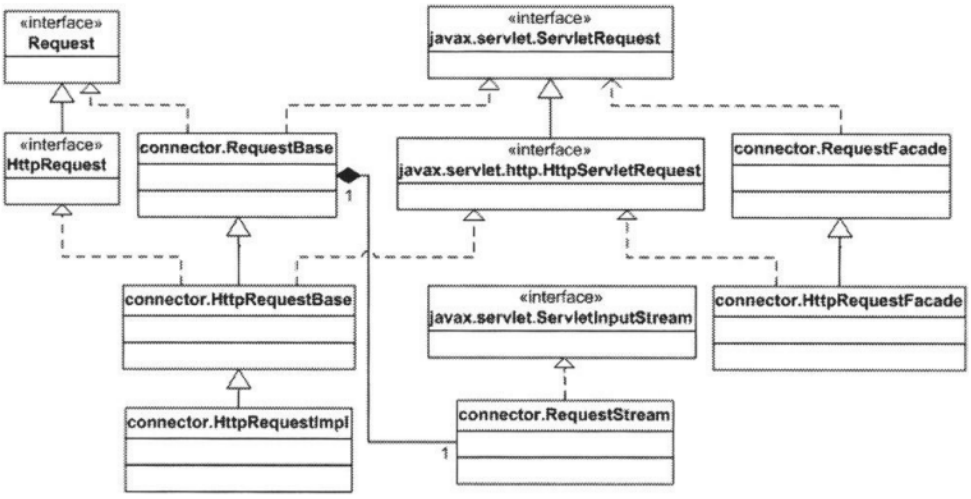
为什么 await 方法要使用一个局部变量保存 socket 对象的引用，而不返回实例的 socket 变量呢？是因为在当前 socket 被处理完之前，可能会有新的 http 请求过来，产生新的 socket 对象将其覆盖。

为什么 await 方法要调用 notifyAll 方法？考虑这种情况，当 available 变量的值还是 true 时，有一个新的 socket 达到。在这种情况下，连接器线程会在 assign 方法的循环体中暂停，直到处理器线程调用 notifyAll 方法。

4.6 request 对象

默认连接器中的 http request 对象由 org.apache.catalina.Request 接口表示。该接口直接集成自 RequestBase 类，RequestBase 是 HttpRequest 的父类。最终的实现类是 HttpRequestImpl，继承自 HttpRequest。

与第三章的类关系类似，本章中也有外观类，RequestFacade 和 HttpRequestFacade。响应的 uml 示意图如下：

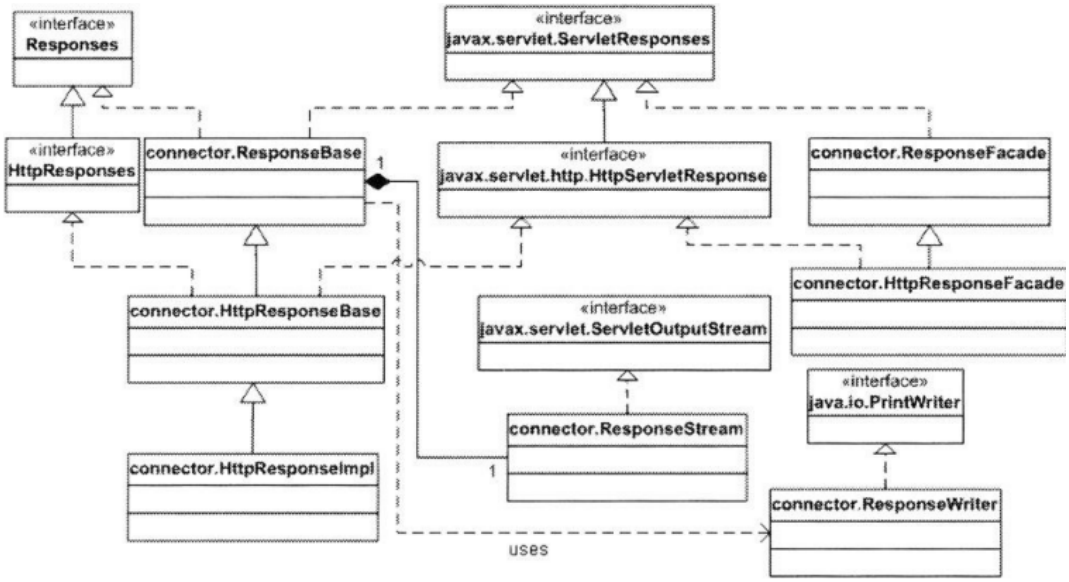


图表 8　Request 接口及相关类的 UML 示意图

注意，图中报名包括 javax·servlet 和 javax·servlet·http，其余类的报名均是 org·apache·catalina，只是被省略掉了。

4.7　response 对象

uml 示意图如下：



图表 9　Response 接口及其相关类 UML 示意图

4.8　处理 request 对象

在这一点上，你已经知道了 request 对象和 response 对象，以及 HttpConnector 是如何创建它们的。本节中，将着重讨论 HttpProcessor 类的 process 方法。process 方法做了三件事，解析连接，解析请求，解析请求头。

process 使用一个 boolean 变量 ok 来表示在处理过程中是否有错误发生，以及一个 boolean 变量

finishResponse 来表示是否应该调用 Response 接口的 finishResponse 方法。

此外，process 还是了实例的其他一些 boolean 变量，如 keepAlive，stopped 和 http*11*。keepAlive 表明该连接是否是持久化连接，stopped 表明 HttpProcessor 实例是否被 connector 终止，这样的话 processor 也应该停止。http*11* 表明客户端发来的请求是否支持 HTTP*/1·1*

与第三章类似，SocketInputStream 实例用于包装 socket 输入流。注意，SocketInputStream 的构造函数也接受缓冲区的大小为参数，该参数来自 connector，而不是 HttpProcessor 的一个局部变量。因为对默认 connector 的使用者来说，HttpProcessor 是不可见的。如下面的代码所示：

```
SocketInputStream input = null;

OutputStream output = null;

// Construct and initialize the objects we will need

try {

    input = new SocketInputStream(socket.getInputstream(),

    connector.getBufferSize());

}

catch (Exception e) {

    ok = false;

}
```

然后是一个 while 循环，不断的读取输入流内容，直到 HttpProcessor 实例被终止，或处理过程报异常，或连接被断开。代码如下：

```
keepAlive = true;

while (!stopped && ok && keepAlive) {

    …

}
```

在循环体内，process 方法现将 finishResponse 设置为 true，获取输出流，执行一些 request 和 response 对象的初始化操作。

```
    finishResponse = true;
```

```
try {
  request.setStream(input);
  request.setResponse(response);
  output = socket.getOutputStream();
  response.setStream(output);
  response.setRequest(request);
  ((HttpServletResponse) response.getResponse()).setHeader
    ("Server", SERVER_INFO);
}
catch (Exception e) {
  log("process.create", e); //logging is discussed in Chapter 7
  ok = false;
}
```

然后，process 方法调用 parseConnection，parseRequest 和 parseHeaders 方法开始解析 http 请求。

```
try {
  if (ok) {
    parseConnection(socket);
    parseRequest(input, output);
    if (!request.getRequest().getProtocol()
      .startsWith("HTTP/0"))
      parseHeaders(input);
```

parseConnection 方法获取请求所使用的协议，其值可以是 HTTP 0.9， HTTP 1.0 或 HTTP 1.1。若值为

HTTP 1.0，则将 keepAlive 置为 false，因此 HTTP 1.0 不支持持久化连接。若是在 http 请求头中找到发现 "Expect: 100-continue"，则 parseHeaders 设置 sendAck 为 true。若请求协议为 HTTP 1.1，会对调用 ackRequest 方法对“Expect: 100-continue”请求头响应。此外，还会检查是否允许分块。

```
if (http11) {
    // Sending a request acknowledge back to the client if
    // requested.
    ackRequest(output);
    // If the protocol is HTTP/1.1, chunking is allowed.
    if (connector.isChunkingAllowed())
        response.setAllowChunking(true);
}
```

ackRequest 方法检查 sendAck 的值，若其值为 true，则发送下面格式的字符串：

HTTP/1.1 100 Continue\r\n

在解析 http 请求的过程中，有可能会抛出很多种异常。发生任何一个异常都会将变量 ok 或 finishResponse 设置为 false。完成解析后，process 方法将 request 和 response 对象作为参数传入 container 的 invoke 方法。

```
try {
    ((HttpServletResponse) response).setHeader
        ("Date", FastHttpDateFormat.getCurrentDate());
    if (ok) {
        connector.getContainer().invoke(request, response);
    }
}
```

然后，若变量 finishResponse 的值为 true，则调用 response 对象的 finishResponse 方法和 request 对象的 finishRequest，再将输出 flush 掉。

```
if (finishResponse) {
    …
    response.finishResponse();
    …
    request.finishRequest();
    …
    output.flush();
```

while 循环的最后一部分是检查 response 的头信息“Connection”是否被设为了“close”，或者协议是否是 HTTP1.0。若这两种情况为真，则将 keepAlive 置为 false。最后将 request 和 response 对象回收。

```
if ( "close".equals(response.getHeader("Connection")) ) {
    keepAlive = false;
 }
 // End of request processing
 status = Constants.PROCESSOR_IDLE;
 // Recycling the request and the response objects
 request.recycle();
 response.recycle();
}
```

若 keepAlive 为 true，或在解析和 container 的 invoke 中没有发生错误，或 HttpProcessor 对象没有被回收，则 while 循环则继续运行。否则调用 shutdownInput 方法，并关闭 socket。

```
try {
 shutdownInput(input);
```

```
        socket.close();
    }
    ...
```

注意，shutdownInput 会检查是否有为读完的字节，若有，则跳过这些字节。

4.8.1 解析连接

parseConnection 从 socket 接收 internet 地址，将其赋值给 HttpRequestImpl 对象。此外，还要检查是否使用了代理，将 socket 对象赋值给 request 对象。代码如下：

```
private void parseConnection(Socket socket) throws IOException, ServletException {
    if (debug >= 2)
        log(" parseConnection: address=" + socket.getInetAddress() + ", port=" +
    connector.getPort());
    ((HttpRequestImpl) request).setInet(socket.getInetAddress());
    if (proxyPort != 0)
        request.setServerPort(proxyPort);
    else
        request.setServerPort(serverPort);
    request.setSocket(socket);
}
```

4.8.2 解析 request

与第三章的程序类似。

4.8.3 解析请求头

默认 connector 的 parseHeaders 方法是用了 org.apache.catalina.connector.http 包内的 HttpHeader 类和 DefaultHeader 类。HttpHeader 类表示一个 http 请求中的请求头。这里与第三章不同的是，这里并没有使用字符串，而是使用了字符数组来避免代价高昂的字符串操作。DefaultHeaders 类是一个 final 类，包含了字符数组形式的标准 http 请求头：

```
static final char[] AUTHORIZATION_NAME = "authorization".toCharArray();

static final char[] ACCEPT_LANGUAGE_NAME = "accept-language".toCharArray();

static final char[] COOKIE_NAME = "cookie".toCharArray();

...
```

parseHeaders 方法使用 while 循环读取所有的请求头信息。while 循环以调用 request 对象的 allocateHeader 方法获取一个内容为空的 HttpHeader 实例开始。然后，该 HttpHeader 实例被传入 SocketInputStream 的 readHeader 方法中。

```
HttpHeader header = request.allocateHeader();

 // Read the next header

 input.readHeader(header);
```

若所有的请求头都已经读取过了，则 readHeader 方法不会再给 HttpHeader 对象设置 name 属性了。这时就可退出 parseHeaders 方法了。

```
if (header.nameEnd == 0) {

    if (header.valueEnd == 0) {

       return;

    }

    else {

       throw new ServletException

          (sm.getString("httpProcessor.parseHeaders.colon"));

    }
```

```
}
```

若是一个 HttpHeader 有 name，那么肯定也会有 value。

```
String value = new String(header.value, 0, header.valueEnd);
```

接下来，与第三章类似，parseHeaders 方法将读取到的请求头的 name 与 DefaultHeaders 中 header 的 name 比较。注意，这里的比较是字符数组的比较，而不是字符串的比较。

```
if (header.equals(DefaultHeaders.AUTHORIZATION_NAME)) {
    request.setAuthorization(value);
}
else if (header.equals(DefaultHeaders.ACCEPT_LANGUAGE_NAME)) {
    parseAcceptLanguage(value);
}
else if (header.equals(DefaultHeaders.COOKIE_NAME)) {
    // parse cookie
}
else if (header.equals(DefaultHeaders.CONTENT_LENGTH_NAME)) {
    // get content length
}
else if (header.equals(DefaultHeaders.CONTENT_TYPE_NAME)) {
    request.setContentType(value);
}
else if (header.equals(DefaultHeaders.HOST_NAME)) {
    // get host name
}
```

```
    else if (header.equals(DefaultHeaders.CONNECTION_NAME)) {
      if (header.valueEquals(DefaultHeaders.CONNECTION_CLOSE_VALUE)) {
        keepAlive = false;
        response.setHeader("Connection", "close");
      }
    } else if (header.equals(DefaultHeaders.EXPECT_NAME)) {
      if (header.valueEquals(DefaultHeaders.EXPECT_100_VALUE))
        sendAck = true;
      else
        throw new ServletException(sm.getstring
          ("httpProcessor.parseHeaders.unknownExpectation"));
    }
     else if (header.equals(DefaultHeaders.TRANSFER_ENCODING_NAME)) {
      //request.setTransferEncoding(header);
    }

    request.nextHeader();
```

4.9 简单的 container 程序

这里重在展示如何使用默认的 connector。程序包括两个类：ex04.pyrmont.core.SimpleContainer 类和 ex04 pyrmont.startup.Bootstrap 类。SimpleContainer 类继承自 org.apache.catalina.Container，这样就可以金额默认的 connector 进行关联。Bootstrap 用于启动程序。

SimpleContainer.java

这里仅仅给出了 invoke 方法的实现。invoke 方法会创建一个 class loader，载入 servlet 类，调用 servlet 的 service 方法。与第三章中的 ServletProcessor 类的 process 方法类似。

Bootstrap.java

Boorstrap 类的 main 方法创建 org·apache·catalina·connector·http·HttpConnector 类和 SimpleContainer 类的实例，然后调用 connector 的 setContainer 方法将 connector 和 container 关联。接下来调用 connector 的 initialize 和 start 方法。

第 5 章 container

container 用于处理对 servlet 的请求，并未客户端填充 resposne 对象。container 由 org·apache·catalina·Container 接口表示。共有四种类型的 container：engine，host，context 和 wrapper。本章涉及到的是 context 和 wrapper。engine 和 host 留在第 *13* 章讨论。本章以对 Container 接口的讨论开始，随后介绍 container 中的流水线（pipelining）机制。然后对 Wrapper 和 Context 接口进行介绍。

5.1 Container 接口

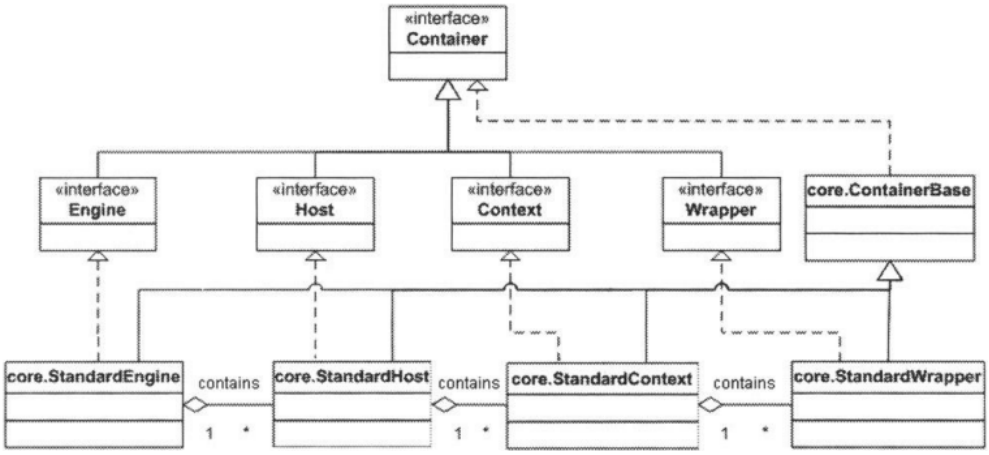
container 必须实现 org·apache·catalina·Container 接口。然后将 container 实例设置到 connector 的 setContainer 方法中。这样，connector 就可以调用 container 的 invoke 方法了。示例代码如下：

```
HttpConnector connector = new HttpConnector();

SimpleContainer container = new SimpleContainer();

connector·setContainer(container);
```

首先要注意的是，在 tomcat 中，共有四种类型的 container，分别由不同的概念层次：

- engine：表示 tomcat 的整个 servlet 引擎；
- host：表示包含有一个或多个 context 的虚拟机；
- context：表示一个 web 应用。一个 context 中可以有多个 wrapper；
- wrapper：表示一个独立的 servlet。

上述的每个概念层级由 org·apache·catalina 包内的一个接口表示，分别是 Engine，Host，Context 和 Wrapper，它们都继承自 Container 接口。这四个接口的标准实现分别是 StandardEngine，StandardHost，StandardContext 和 StandardWrapper，都在 org·apache·catalina·core 包内。下面的 uml 图展示了各类的关系：



图表 10　Container 接口及其实现类的 UML 图示

注意，所有的实现类都继承自 ContainerBase 类。

catalina 中的部署并不是必须将所有四种 container 都包括在内。例如，本章中第一个应用的 container 模块仅仅用到了 wrapper。

一个 container 可以有 *0* 个或多个低层级的子 container。例如，一般情况下，一个 context 中会有一个或多个 wrapper，一个 host 中会有 *0* 个或多个 context。但是，wrapper 处理层级的最底层，因此，它无法再包含子 container 了。可以通过 Container 接口的 addChild 方法添加子 container，removeChild 删除一个子 container，此外，Container 接口也支持对子 container 的查找，方法是 findChild 和 findChildren。

container 中可以包含一些支持的组件，如 Loader，logger，Manger，Realm 和 Resources 等，提供了 getter 和 setter 方法进行访问。这些组件将在后续的章节中讨论。

有趣的是，Container 接口被设计成，在部署应用是，tomcat 管理员可以通过编辑选项文件（server·xml）来决定使用哪种 container。这是通过引入流水线（pipeline）和 container 中阀（valve）的集合实现的。

5.2　pipeline 任务

本节说明在 connector 调用了 container 的 invoke 方法后发生了什么事。按照相关的四个接口分为四个小节讨论，这四个接口都在 org·apache·catalina 包中，分别是 Pipeline，Valve，ValveContext 和 Contained。

pipeline 包括了 container 中要执行的任务。一个阀（valve）表示一个指定的任务。在 pipeline 中有一个基础 valve（basic valve），但是使用者可根据自己的需要添加其他的 valve。注意，vavle 的数量被定义为额外添加的 valve 的数量，不包括 basic valve。有趣的是，valve 可以通过修改 server·xml 来动态添加。

如果对 servlet 的过滤器有所了解的话，就不难理解 pipeline 和 valve 是如何工作的。pipeline 就像是 filter 链，每个 valve 就像是一个过滤器。实际上，valve 与过滤器类似，它可以控制传递给它的 request 和 response 对象。当一个 valve 处理结束后，它就调用 pipeline 中的下一个 valve 进行处理。basic valve 总是最后被调用的。

container 中有一个 pipeline。当调用了 container 的 invoke 方法后，container 将处理过程交给它的 pipeline，而 pipeline 会调用它的第一个 valve，valve 执行完后会调用后续的 valve，知道所有的 valve 都调用结束。下面是伪代码：

```
// invoke each valve added to the pipeline for (int n=0; n<valves·length; n++) {
```

```
    valve[n].invoke( ... );
}
// then, invoke the basic valve
basicValve.invoke( ... );
```

但是，tomcat 的设计者选择了另一种实现方法，通过引入接口 org.apache.catalina.ValveContext 来实现。当 connector 调用 container 的 invoke 方法后，container 要做的事并没有硬编码写在 invoke 方法中，而是调用 pipeline 的 invoke 方法，该方法的签名与 container 的 invoke 方法相同：

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

下面是 org.apache.catalina.core.ContainerBase 的 invoke 实现：

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

其中 pipeline 是 Pipeline 接口的实例。

pipeline 必须保证添加到其中的所有 valve 和 basic valve 都被调用一次。pipeline 是通过创建一个 ValveContext 实例来实现的。ValveContext 是作为 pipeline 的一个内部类实现的，这样，ValveContext 实例就可以访问 pipeline 的所有成员了。ValveContext 接口中最重要的方法是 invokeNext：

```
public void invokeNext(Request request, Response response) throws IOException, ServletException;
```

在所有的 container 中，org.apache.catalina.core.StandardPipeline 类实现了 Pipeline 接口，在该类中有一个内部类 StandardPipelineValveContext 实现了 ValveContext 接口。StandardPipelineValveContext 的代码如下所示：

```
protected class StandardPipelineValveContext implements ValveContext {
    protected int stage = 0;
    public String getInfo() {
      return info;
```

```
    }
    public void invokeNext(Request request, Response response) throws IOException, ServletException
{
      int subscript = stage;
      stage = stage + 1;
      // Invoke the requested Valve for the current request thread
      if (subscript < valves.length) {
        valves[subscript].invoke(request, response, this);
      }
      else if ((subscript == valves.length) && (basic != null)) {
        basic.invoke(request, response, this);
      }
      else {
        throw new ServletException
          (sm.getString("standardPipeline.noValve"));
      }
    }
  }
```

invokeNext 方法是用变量 subscript 和 stage 表明要调用哪个 valve。当第一个 valve 在 pipeline 的 invoke 方法中被调用时，subscript 的值是 *0*，stage 的值为 *1*。因此，第一个 valve 被调用。pipeline 中的第一个 valve 接收 ValveContext 的实例，调用它的 invokeNext 方法。这时 subscript 的值为 *1*，可以调用第二个 valve。当 invokeNext 方法被最后一个 valve 调用时，subscript 等于 valve 的数量，然后，调用 basic valve。

tomcat*5* 中从 StandardPipeline 取消了 StandardPipelineValveContext，取而代之的是 org.apache.catalina.core.StandardValveContext。代码如下：

```
package org.apache.catalina.core;

import java.io.IOException;
import javax.servlet.ServletException;
import org.apache.catalina.Request;
import org.apache.catalina.Response;
import org.apache.catalina.Valve;
import org.apache.catalina.ValveContext;
import org.apache.catalina.util.StringManager;

public final class StandardValveContext implements ValveContext {
    protected static StringManager sm =
      StringManager.getManager(Constants.Package);
    protected String info =
      "org.apache.catalina.core.StandardValveContext/1.0";
    protected int stage = 0;
    protected Valve basic = null;
    protected Valve valves[] = null;
    public String getInfo() {
      return info;

    }
```

```
    public final void invokeNext(Request request, Response response)
      throws IOException, ServletException {
      int subscript = stage;
      stage = stage + 1;
      // Invoke the requested Valve for the current request thread
      if (subscript < valves.length) {          valves[subscript].invoke(request, response, this);
      }
      else if ((subscript == valves.length) && (basic != null)) {
        basic.invoke(request, response, this);
      }
      else {
        throw new ServletException
          (sm.getString("standardPipeline.noValve"));
      }
    }


    void set(Valve basic, Valve valves[]) {
      stage = 0;
      this.basic = basic;
      this.valves = valves;
    }
}
```

5·2·1 Pipeline

container 调用 pipeline 的 invoke 方法开始对 valve 进行逐个调用。使用 Pipeline 接口的 addValve 方法可以添加新的 valve，或调用 removeValve 方法删除对某个 valve 的调用。使用 setBasicValve 和 getBasicValve 可以对 basic valve 进行设置。

Pipeline 接口定义如下：

```
package org·apache·catalina;

import java·io·IOException;



import javax·servlet·ServletException;

public interface Pipeline {

    public Valve getBasic();

    public void setBasic(Valve valve);

    public void addValve(Valve valve);

    public Valve[] getValves();

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

    public void removeValve(Valve valve);

}
```

5·2·2 Valve 接口

Valve 接口用于处理一个请求，包含了两个方法，invoke 和 getInfo。getInfo 方法返回 valve 的实现信息 Valve 接口的定义如下：

```
package org·apache·catalina;

import java·io·IOException;

import javax·servlet·ServletException;
```

```
public interface Valve {
    public String getInfo();
    public void invoke(Request request, Response response,
      ValveContext context) throws IOException, ServletException;
}
```

5.2.3 ValveContext 接口

该接口有两个方法，invokeNext 和 getInfo。接口定义如下：

```
package org.apache.catalina;
import java.io.IOException;
import javax.servlet.ServletException;

public interface ValveContext {
    public String getInfo();
    public void invokeNext(Request request, Response response)
      throws IOException, ServletException;

}
```

5.2.4 Contained 接口

一个 valve 也可以实现 org.apache.catalina.Contained 接口，该接口将 valve 与某个 container 绑定。接口定义如下：

```
package org.apache.catalina;
```

```
public interface Contained {

public Container getContainer();

public void setContainer(Container container);
```

5.3 Wrapper 应用程序

org·apache·catalina·Wrapper 接口表示一个 wrapper 层级的 container，表示一个独立的 servlet 定义。Wrapper 继承自 Container 接口，又添加了一些额外的方法。Wrapper 接口的实现要负责管理 servlet 的生民周期，例如，调用 init，service，destroy 等方法。若是向 Wrapper 的 addChild 方法被调用，则抛出 IllegalArgumantException 异常。

Wrapper 接口中比较重要的方法是 load 和 allocate 方法。allocate 方法会为 wrapper 分配一个 servlet 实例，而且，allocate 方法还要考虑下 servlet 类是否实现了 javax·servlet·SingleThreadModel 接口（在 *11* 章讨论）。load 方法载入并初始化 servlet 类。方法签名如下：

```
public javax·servlet·Servlet allocate() throws javax·servlet·ServletException;

public void load() throws javax·servlet·ServletException;
```

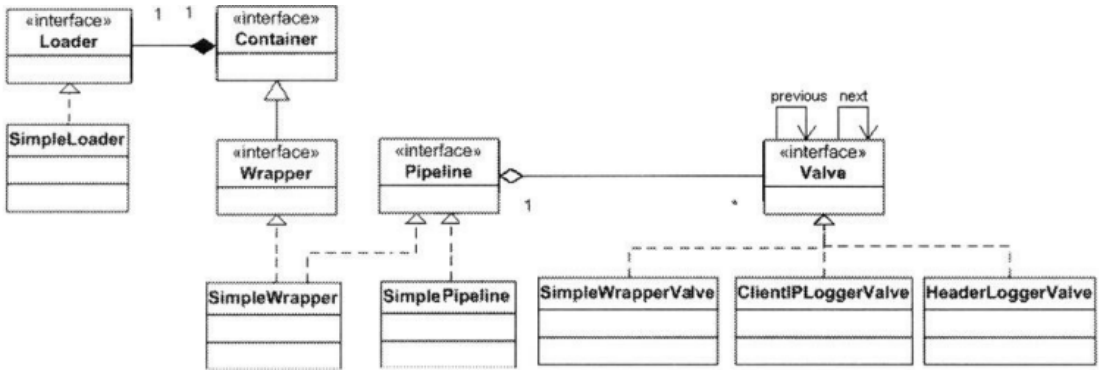
5.4 Context 接口

Context 表示了一个 web 应用，一个 context 中可以有一个或多个 wrapper。Context 接口比较重要的方法是 addWrapper 和 createWrapper（在第十二章讨论）。

5.5 Wrapper 程序实例

该程序展示了如何写一个最小的 container 模块。核心类是 ex*05*·pyrmont·core·SimpleWrapper，实现了 Wrapper 接口。SimpleWrapper 类中包含了一个 Pipeline（由 ex*05*·pyrmont·core·SimplePipeline 实现），使用一个 Loader（由 ex*05*·pyrmont·core·SimpeLoader 实现）载入 servlet。pipeline 中有一个 basic valve（ex*05*·pyrmont·core·SimpleWrapperValve），和两个额外的 valve（ex*05*·pyrmont·core·ClientIPLoggerValve

和 ex05·pyrmont·core·HeaderLoggerValve）。程序的 uml 图如下所示：



图表 11　Wrapper 程序 UML 示意图

注意：程序使用 tomcat4 默认的 connector。

5·5·1　ex05·pyrmont·core·SimpleLoader

该类用于在 container 中载入 servlet 类。该类通过变量 WEB_ROOT 指明要在哪里查找 servlet 类。使用 ClassLoader 和 Container，分别指明类载入器和 container。

SimpleLoader 类的构造函数会初始化一个 class loader，以便 SimpleWrapper 实例使用。代码如下：

```
public SimpleLoader() {
 try {
   URL[] urls = new URL[1];
   URLStreamHandler streamHandler = null;
   File classPath = new File(WEB_ROOT);
   String repository = (new URL("file", null,
     classPath.getCanonicalPath() + File.separator)).toString() ;
   urls[0] = new URL(null, repository, streamHandler);
   classLoader = new URLClassLoader(urls);
 }
 catch (IOException e) {
   System.out.println(e.toString() );
```

```
    }
  }
```

5.5.2 ex05.pyrmont.core.SimplePipeline

SimplePipeline 类实现了 org.apache.catalina.Pipeline 接口，最重要的是 invoke 方法，该方法中包含了一个内部类，SimplePipelineValveContext，该类实现了 org.apache.catalina.ValveContext 接口。

5.5.3 ex05.pyrmont.core.SimpleWrapper

该类实现了 org.apache.catalina.Wrapper 接口，提供了 allocate 和 load 方法的实现，声明了两个变量：

```
private Loader loader;
protected Container parent = null;
```

loader 变量指明了载入 servlet 要使用的 loader 实例。parent 变量指明该 wrapper 的父 container。注意其中的 getLoader 方法，代码如下：

```
public Loader getLoader() {
    if (loader != null)
        return (loader);
    if (parent != null)
        return (parent.getLoader());
    return (null);
}
```

该方法返回一个用于 servlet 类的 loader，若 wrapper 已经关联了一个 loader，则直接将其返回，否则从父 container 处获取并返回，若还是没有，返回 null。

SimpleWrapper 有一个 pipeline，需要通过 setBasic 方法为其设置一个 basic valve。

5.5.4 ex05.pyrmont.core.SimpleWrapperValve

SimpleWrapperValve 类实现了 org.apache.catalina.Valve 接口和 org.apache.catalina.Contained 接口，是一个 basic valve，专用于为 SimpleWrapper 处理请求。其最重要的方法 invoke 如下所示：

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

    SimpleWrapper wrapper = (SimpleWrapper) getContainer();
    ServletRequest sreq = request.getRequest();
    ServletResponse sres = response.getResponse();
    Servlet servlet = null;
    HttpServletRequest hreq = null;
    if (sreq instanceof HttpServletRequest)
        hreq = (HttpServletRequest) sreq;
    HttpServletResponse hres = null;
    if (sres instanceof HttpServletResponse)
        hres = (HttpServletResponse) sres;
    //Allocate a servlet instance to process this request
    try {
        servlet = wrapper.allocate();
        if (hres!=null && hreq!=null) {
            servlet.service(hreq, hres);
        } else {
            servlet.service(sreq, sres);
        }
```

```
        } catch (ServletException e) { }
    }
```

SimpleWrapperValve 类作为 basic valve 使用，因此，其 invoke 方法不需要调用 valveContext 的 invokeNext 方法，它调用 servlet 的 service 方法，而不是 wrapper 类的。

5.5.5 ex05.pyrmont.valves.ClientIPLoggerValve

该类打印用户的 ip 信息。代码如下：

ClientIpLoggerValve.java

注意其 invoke 方法，它先调用 valveContext 的 invokeNext 方法，然后再打印 ip。

5.5.6 ex05.pyrmont.valves.HeaderLoggerValve

该类与 ClientIPLoggerValve 类似，打印请求头信息。代码如下：

HeaderLoggerValve.java

5.5.7 ex05.pyrmont.startup.Bootstrap1

该类用于启动应用程序，代码如下：

Bootstrap1.java

5.6 Context 程序实例

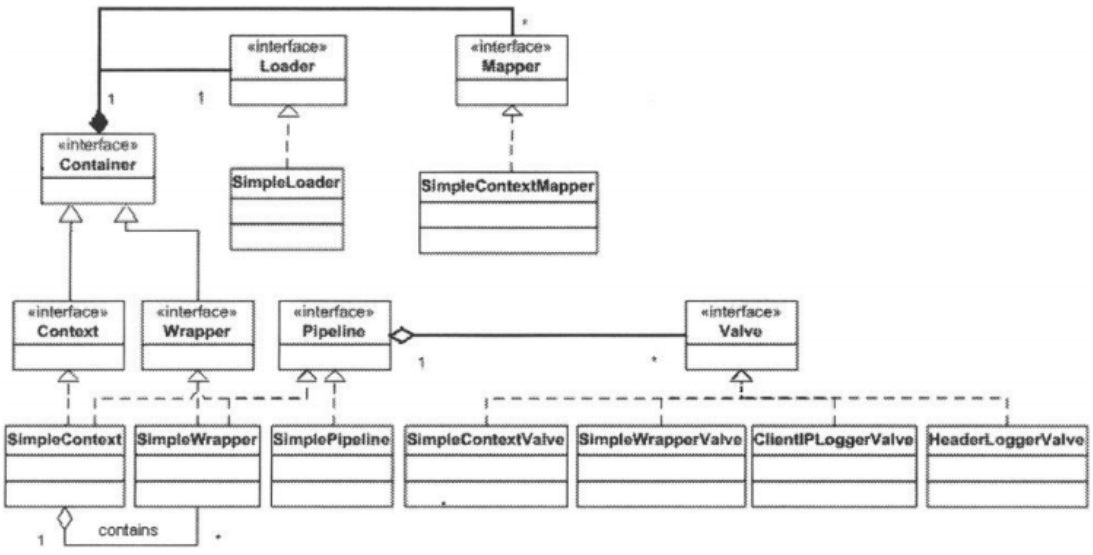
本节的程序展示了如何使用 context 和 wrapper。在程序中是了一个 mapper（一个组件）来帮助 context 选择某个 wrapper 来处理特殊的请求。

注意：mapper 组件仅在 tomcat4 中，tomcat5 使用了其他的方法。

本例中，mapper 是 ex05·pyrmont·core·SimpleContextMapper 类的实例，该类实现了 org·apache·catalina·Mapper 接口。container 中可以包含有多个 mapper 来支持不同的请求协议。例如，一个 mapper 处理 HTTP 协议请求，另一个 mapper 处理 HTTPS 协议的请求。org·apache·catalina·Mapper 接口定义如下所示：

```
package org·apache·catalina;

public interface Mapper {

    public Container getContainer();

    public void setContainer(Container container);

    public String getProtocol();

    public void setProtocol(String protocol);

    public Container map(Request request, boolean update);

}
```

其中 setProtocol 和 getProtocol 指明了该 mapper 负责处理哪种协议，map 方法返回使用哪个子 container 处理特殊的请求。相关类的 UML 图如下所示：



图表 12　Context 应用程序相关类 UML 示意图

SimpleContext 类表示 context，SimpleContextMapper 作为其 mapper，SimpleContextValve 作为其 basic valve。context 中有两个 valve，ClientIPLoggerValve 和 HeaderLoggerValve，还有两个 wrapper，都是 SimpleWrapper，这两个 wrapper 都使用 SimpleWrapperValve 作为其 basic valve，且不再添加其他 valve。

在 Context 应用程序中，使用了相同的 loader 和 valve，但是它们之间是通过 context，而不是 wrapper 关

联的，这样，所有的 wrapper 就都可以使用一个 loader 了。context 作为 connector 的 container 使用，因此，当 connector 接收到一个 http 请求时，会调用 context 的 invoke 方法。剩下的步骤如下所示：

（1）container 中包含一个 pipeline，container 的方法会调用 pipeline 的 invoke 方法；

（2）pipeline 调用所有添加到其中的 valve，最后调用 basic valve；

（3）在 wrapper 中，basic valve 负责载入 servlet 类，并相应 http 请求；

（4）在包含子 container 的 context 中，basic valve 使用 mapper 找到某个子 container 负责处理 http 请求；若找到了这样的子 container，则调用其 invoke 方法，调用第一步。

5.6.1 ex05.pyrmont.core.SimpleContextValve

该类中最重要的方法是 invoke 方法，具体实现如下：

```
public void invoke(Request request, Response response,      ValveContext valveContext)
  throws IOException, ServletException {
  // Validate the request and response object types
  if (!(request.getRequest() instanceof HttpServletRequest) ||
    !(response.getResponse() instanceof HttpServletResponse)) {
    return;
  }

  // Disallow any direct access to resources under WEB-INF or META-INF
  HttpServletRequest hreq = (HttpServletRequest) request.getRequest();
  String contextPath = hreq.getContextPath();
  String requestURI = ((HttpRequest) request).getDecodedRequestURI();
  String relativeURI =
    requestURI.substring(contextPath.length()).toUpperCase();
```

```
    Context context = (Context) getContainer();
    // Select the Wrapper to be used for this Request

   Wrapper wrapper = null;
    try {
      wrapper = (Wrapper) context.map(request, true);
    }
    catch (IllegalArgumentException e) {
      badRequest(requestURI, (HttpServletResponse)
        response.getResponse());
      return;
    }
    if (wrapper == null) {
      notFound(requestURI, (HttpServletResponse) response.getResponse());
      return;
    }
    // Ask this Wrapper to process this Request
    response.setContext(context);
    wrapper.invoke(request, response);
}
```

5·6·2 ex05·pyrmont·core·SimpleContextMapper

SimpleContextMapper 类实现了 org·apache·catalina·Mapper 接口。代码如下：

SimpleContextMapper.java

5·6·3 ex05·pyrmont·core·SimpleContext

SimpleContext 类是本程序的 context 实现，是分配给 connector 的主 container，但是对每个独立 servlet 的处理是由 wrapper 完成的。本程序中有两个 servlet，两个 wrapper，它们都有对应的名字，PrimitiveServlet 对应的 wrapper 的名字是 Primitive，ModernServlet 对应的 wrapper 的名字是 Modern。SimpleContext 是通过 url 映射来决定调用哪个 wrapper 的。本程序有两个 url 可以使用，其中“/Primitive”会调用 Primitive，“/Modern”会调用 Modern。你也可以添加自己的映射。

Container 和 Context 接口中有很多方法，在本程序的实现中，大部分都是空方法，但与映射有关的方法都实现了，这些方法是：

（1）addServletMapping，添加一个 url 和 wrapper 的映射；

（2）findServletMapping，通过 url 查找对应的 wrapper；

（3）addMapper，在 context 中添加一个 mapper。SimpleContext 类中有两个变量 mapper 和 mappers。mapper 表示程序使用的默认 mapper，mappers 包含了 SimpleContext 实例中所有的 mapper。第一个被添加到 mappers 中的 mapper 成为默认 mapper；

（4）findMapper，找到正确的 mapper，在 SimpleContext 中，它返回默认 mapper；

（5）map，返回负责处理当前请求的 wrapper。

5·6·4 ex05·pyrmont·startup·Bootstrap2

实现代码如下：

Bootstrap2.java

第 6 章　生命周期（Lifecycle）

6.1　概述

catalina 包含有很多组件，随 catalina 一起启动/关闭。例如，当 container 关闭时，它必须调用已经载入的 servlet 的 destroy 方法。tomcat 中的实现机制是通过实现 org·apache·catalina·Lifecycle 接口来管理。

实现了 org·apache·catalina·Lifecycle 接口的组件会触发下面的事件：

- BEFORE_START_EVENT；
- START_EVENT；
- AFTER_START_EVENT；
- BEFORE_STOP_EVENT；
- STOP_EVENT；
- AFTER_STOP_EVENT。

当组件启动是会触发前三个事件（BEFORE_START_EVENT，START_EVENT，AFTER_START_EVENT），关闭组件时会触发后三个事件（BEFORE_STOP_EVENT，STOP_EVENT，AFTER_STOP_EVENT）。而相应的监听器由 org·apache·catalina·LifecycleListener 接口表示。

本章将讨论三种类型，Lifecycle，LifecycleEvent 和 LifecycleListener。

6.2　Lifecycle 接口

catalina 在设计上允许一个组件包含其他组件，如 container 中可以包含 loader，manger 等组件。父组件负责启动/关闭其包含的子组件。所有的组件都可以通过其父组件来启动/关闭，这种单一启动/关闭机制是通过 Lifecycle 接口实现的。Lifecycle 接口定义如下：

```
package org·apache·catalina; 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;

}
```

其中，最重要的方法时 start 和 stop 方法。父组件通过这两个方法来启动/关闭该组件。addLifecycleListener，findLifecycleListeners，removeLifecycleListener 三个方法用于向组件注册/查找/删除监听器。当事件发生时，会触发监听器。接口中还定义了相关事件。

6.3 LifecycleEvent 类

org·apache·catalina·LifecycleEvent 类表示生命周期中的某个事件。定义如下：

LifecycleEvent.java

6.4 LifecycleListener 接口

接口定义如下：

```
package org·apache·catalina;

import java·util·EventObject;

public interface LifecycleListener {

    public void lifecycleEvent(LifecycleEvent event);

}
```

其中，当监听的事件发生时，触发 lifecycleEvent 方法。

6.5 LifecycleSupport 类

实现了 Lifecycle 接口的组件可以向监听器注册感兴趣的事件，catalina 提供一个工具类来管理对组件注册的监听器，org·apache·catalina·util·LifecycleSupport。代码如下：

LifecycleSupport.java

LifecycleSupport 类中用数组类型的变量 listeners 存储了所有监听器。当添加一个新的监听器时，是创建一个新数组，存储全部的监听器。删除一个监听器的时候，也是返回一个新的数组对象。

fireLifecycleEvent 方法会触发已经注册的各个监听器。在实现上，有如下代码要注意：

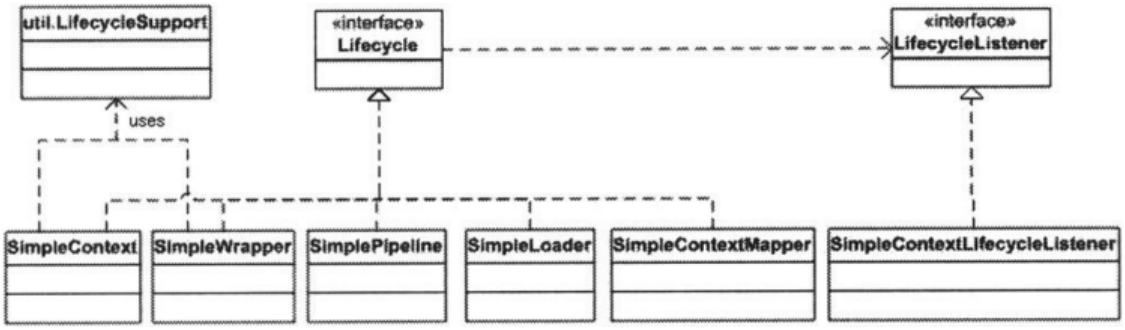
```
synchronized (listeners) {
    interested = (LifecycleListener[]) listeners·clone();
}
```

之所以要克隆一份新的监听器数组，是为了保证在触发已经注册的监听器时不会受到临时新注册的监听器的影响。

实现了 Lifecycle 接口的组件可以使用 LifecycleSupport 类，对监听器进行添加、删除、触发等操作。

6.6 应用程序

本章的应用程序建立于第五章的应用程序基础之上，重在说明 Lifecycle 接口及相关类的使用方法。程序中包含一个 context，两个 wrapper，一个 loader 和一个 mapper。程序中的组件实现了 Lifecycle，在 context 中使用了一个拦截器。第五章中的两个 valve，在这里没有继续使用。下面是相关的 UML 示意图：



图表 13　应用程序相关类示意图

6·6·1 ex06·pyrmont·core·SimpleContext

SimpleContext 类与第五章的 SimpleContext 相似，只是多实现了 Lifecycle，使用下面的变量 lifecycle 引用 LifecycleSupport 的实例：

```
protected LifecycleSupport lifecycle = new LifecycleSupport(this);
```

此外，使用 boolean 变量 started 表明 SimpleContext 实例是否已经启动运行。部分实现代码如下所示：

SimpleContext.java

请注意 start 方法是如何启动所有的子 container 和相关联的组件的，以及是如何关闭它们的。实现机制是，要启动 container 中所有的组件，只需要启动从层次上看最高层的组件（本程序中是 SimpleContext 的实例），关闭的时候，也只需要关闭同一个组件。

SimpleContext 的 start 方法会检查之前是否已经启动，若是，则抛出 LifecycleException 异常，否则触发 BEFORE_START_EVENT 事件，然后，SimpleContext 中注册的监听器都会接收到事件。然后，start 方法设置 started 变量，并启动其组件和子 container。之后，start 方法还会触发 START_EVENT 和 AFTER_START_EVENT 事件。stop 方法执行 start 相反的操作。

6·6·2 ex06·pyrmont·core·SimpleContextLifecycleListener

该类表示了 SimpleContext 实例中的一个监听器。代码如下：

SimpleContextLifecycleListener.java

这个类比较简单，只是打印出了触发的事件。

6·6·3 ex06·pyrmont·core·SimpleLoader

该类与第五章相似，只是实现了 Lifecycle 接口，而具体的实现方法只是打印了一个字符串。最重要的是，通过实现 Lifecycle 接口，该类可以由其关联的 container 启动。 代码如下：

SimpleLoader.java

6.6.4 ex06.pyrmont.core.SimplePipeline

除了实现 Pipeline 接口外，该类还实现了 Lifecycle 接口。实现 Lifecycle 的接口方法仍然留空。重点仍然是可以从 container 启动该类。

6·6·5 ex06·pyrmont·core·SimpleWrapper

与第五章相似。这里还实现了 Lifecycle 接口。Lifecycle 接口的实现方法仍然留空。代码如下：

SimpleWrapper.java

其 start 和 stop 方法与 context 中的机制相同。

第 7 章 Logger

7.1 概述

Logger 是 catalina 中用于记录消息的组件。在 catalina 中，logger 是与 container 相关联的，与其他组件相比，稍简单一些。在 org·apache·catalina·logger 包中，tomcat 提供了不同类型的 logger。

7.2 Logger

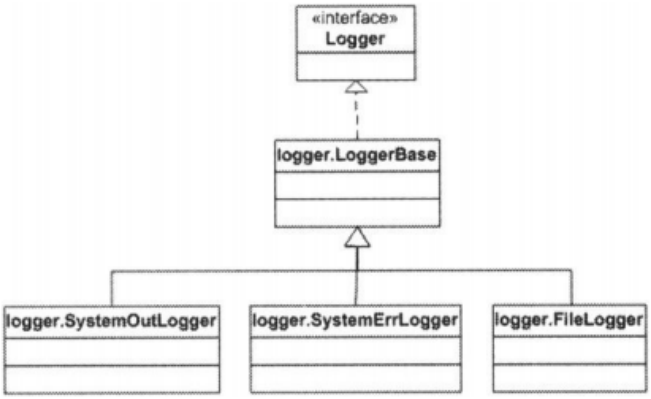
catalina 中的 logger 都要实现 Logger 接口。Logger 接口定义如下：

Logger.java

其中最后两个 log 方法可以传入日志级别，若是传入级别比 logger 实例中设置级别低，则会被记录，否就忽略。共有五的日志级别，FATAL，ERROR，WARNING，INFORMATION 和 DEBUG。此外，Logger 接口可以通过 getter 和 setter 方法设置相关联的 container，以及设置 PropertyChangeListener 监听器。

7.3 Tomcat's Loggers

tomcat 提供了三个 logger，分别是 FileLogger，SystemErrorLogger 和 SystemOutLogger。他们都位于 org·apache·catalina·logger 包下，均继承自 org·apache·catalina·logger·LoggerBase 类。在 tomcat4 中 LoggerBase 实现了 org·apache·catalina·Logger 接口，在 tomcat5 中，它还实现了 Lifecycle 和 MBeanRegistration 接口。相关类的 UML 图如下：



图表 14 tomcat's logger

7.3.1 LoggerBase 类

在 tomcat5 中，LoggerBase 类会更复杂一些，因为它与 Mbeans 进行了合并。因此，这里只讨论 tomcat4 中的 LoggerBase 类。在 tomcat4 中，LoggerBase 是一个虚类，它实现了 Logger 接口中除 log(String msg) 方法外的的全部方法。需要子类实现的方法签名如下：

```
public abstract void log(String msg);
```

该类的 verbosity 变量被设置为 protected，默认值为 ERROR，可通过 getter/setter 方法访问/修改。在使用 setter 方法进行修改时，会检查设置级别。若设置的级别不在预定义的级别范围内，则放弃设置。

有两个 log 方法需要传入 verbosity 参数，在这些重载的方法中，只有当传入的 verbosity 级别比当前 logger 实例的 verbosity 级别低时，才会调用 log(String message)的重载方法。

7.3.2 SystemOutLogger 类

该类继承自 LoggerBase，实现了 log(String message)方法。代码如下所示：

```
package org.apache.catalina.logger;

public class SystemOutLogger extends LoggerBase {


    protected static final String info =
       "org.apache.catalina.logger.SystemOutLogger/1.0";


    public void log(String msg) {
       System.out.println(msg);
    }
}
```

7.3.3 SystemErrLogger 类

与 SystemOutLogger 类似，区别在于该类将 msg 打印到标准错误上。代码如下：

```
package org.apache.catalina.logger;

public class SystemErrLogger extends LoggerBase {


    protected static final String info =
       "org.apache.catalina.logger.SystemErrLogger/1.0";
    public void log(String msg) {        System.err.println(msg);
    }
}
```

7.3.4 FileLogger 类

该类将从 container 中接收到的 msg 写到文件中，可以选择是否添加时间戳。当该类首次被实例化时，会创建一个日志文件，文件名中包含当日的日期信息。若日期变化了，则创建一个新文件。在 tomcat4 中，FileLogger 类实现了 Lifecycle 接口，可以像其他实现了 Lifecycle 接口的组件一样，由其关联的 container 启动关闭。在 tomcat5 中，改为由 LoggerBase 实现 Lifecycle 接口。

在 tomcat4 中，FileLogger 类的启动关闭仅仅是触发了监听器的事件。实现代码如下：

```
public void start() throws LifecycleException {

    // Validate and update our current component state
    if (started)
      throw new LifecycleException
        (sm.getString("fileLogger.alreadyStarted"));
    lifecycle.fireLifecycleEvent(START_EVENT, null);
    started = true;
```

```
}

public void stop() throws LifecycleException {

    // Validate and update our current component state

    if (!started)

      throw new LifecycleException

       (sm.getString("fileLogger.notStarted"));

    lifecycle.fireLifecycleEvent(STOP_EVENT, null);

    started = false;      close ();

}
```

其中，stop 方法还调用了私有方法 close 来关闭日志文件。

FileLogger 类中最重要的是 log 方法，其实现如下：

```
public void log(String msg) {

    // Construct the timestamp we will use, if reguested

    Timestamp ts = new Timestamp(System.currentTimeMillis());

    String tsString = ts.toString().substring(0, 19);

    String tsDate = tsString.substring(0, 10);



     // If the date has changed, switch log files

    if (!date.equals(tsDate)) {

      synchronized (this) {

        if (!date.equals(tsDate)) {

          close ();

          date = tsDate;

          open ();
```

```
        }
      }
    }

    // Log this message, timestamped if necessary
    if (writer != null) {
      if (timestamp) {
        writer.println(tsString + " " + msg);
      }
      else {
        writer.println(msg);
      }
    }
  }
```

服务器一般会启动很久，因此，在 FileLogger 实例的生命周期内，可能会打开/关闭多个日志文件。

7.3.4.1 open 方法

该方法在指定的目录中创建新的日志文件。代码如下：

```
private void open() {
    // Create the directory if necessary
    File dir = new File(directory);
    if (!dir.isAbsolute())
```

```
            dir = new File(System.getProperty("catalina.base"), directory);
        dir.mkdirs();

        // Open the current log file
        try {
            String pathname = dir.getAbsolutePath() + File.separator +
                prefix + date + suffix;
            writer = new PrintWriter(new FileWriter(pathname, true), true);
        }
        catch (IOException e) {
            writer = null;
        }
    }
```

该方法会检查要创建的文件是否已经存在，所需要以来的目录路径是否存在。然后创建一个 PrintWriter 实例，赋值给 FileLogger 的成员变量。

7.3.4.2 close 方法

该方法负责 flush 已经写如的内容，关闭 PrintWriter 实例，将成员变量 writer 置为 null，date 置为空字符串（为“”）。实现代码如下：

```
    private void close() {
        if (writer == null)
            return;
        writer.flush();
        writer.close();
```

```
        writer = null;
        date = "";
    }
```

7.3.4.3 log 方法

log 方法会先创建一个 java·sql·Timestamp 类的实例（该类是 java·util·Date 类的包装），方便获取日期。代码如下：

```
    Timestamp ts = new Timestamp(System·currentTimeMillis());
    String tsString = ts·toString()·substring(0, 19);
    String tsDate = tsString·substring(0, 10);
```

然后，将 tsDate 与成员变量 date（初始化为空字符串）进行比较，若不相同，则关闭当前日志文件，将 tsDate 赋值给 date，并以此创建新的日志文件。

最后，想日志文件中写日志。

```
    // Log this message, timestamped if necessary
    if (writer != null) {
      if (timestamp) {
        writer·println(tsString + " " + msg);
      }
      else {
        writer·println(msg);
      }
    }
```

7.4 应用程序

本章的应用程序与第六章类似，只是多了一个 FileLogger 组件。代码如下：

Bootstrap.java

第 8 章 Loader

8.1 概述

前面的章节中简要介绍了 loader 的使用，本章将阐述 web 应用中 loader 是如何工作的。container 中需要一个自定义的 loader，不能简单的直接使用系统的 loader，因为所要运行的 servlet 是不可信任的。假如像使用之前章节一样使用系统中的 loader，则该 loader 载入的 servlet 和其他类就能访问当前 jvm 实例中 CLASSPATH 环境变量下所有的类了，这非常不安全。servlet 应该仅仅被允许从 WEB-INF/classes 及其子目录，和 WEB-INF/lib 下部署额 lib 中载入类。因此，servlet 容器需要一个自定义的 loader。每一个 web 一样（对应 context 容器）都有一个自定义的 loader。在 catalina 中，loader 组件要实现 org·apache·catalina·Loader 接口。

使用自定义 loader 的另一个原因是，可以支持 class 的自动重载。tomcat 中的 loader 实现使用了另一个线程来检查 servlet 和相关类的时间戳，若是发生变化，则重新载入。为了支持 class 的重载，loader 还要实现 org·apache·catalina·loader·Reloader 接口。

本章中有两个名词会大规模使用，repository 和 resources。repository 表示 loader 要在哪里搜索要载入的类。resources 表示 loader 中 DirContext 对象的引用。

8.2 java 本身的 loader

每次创建 java 对象时，jvm 都会使用 loader 载入该类。一般情况下，jvm 只会在 java 核心库的 CLASSPATH 路径下寻找指定类。若找不到，则抛出 java·lang·ClassNotFoundException 异常。

从 j2se 1·2 开始，jvm 使用三种 loader：引导类载入器（bootstrap class loader），扩展类载入器（extension class loader）和系统类载入器（system class loader）。这三种有着父子继承关系（引导类载入位于最高层）。

引导类载入器（bootstrap class loader）用于引导 jvm。当使用 java 命令时，引导类载入器开始工作。引导类载入是使用本地方法实现的，因为它要负责载入启动 jvm 的类。此外，它还要负责载入 java 核心类，例如 java·io 和 java·lang 包下的类，它的搜索路径包括 rt·jar 和 i18n·jar 等包，具体查找哪些包依赖于 jvm 和操作系统的版本。

扩展类载入器（extension class loader）负责载入标准扩展目录下的类。这有利与程序开发，因为程序员只需要将 jar 包拷贝到扩展目录中，扩展类载入器会从这些 jar 包中查找需要的类。扩展目录依赖于 jvm 的具体实现。sun 的 jvm 实现中标准扩展目录是“/jdk/jre/lib/ext”。

系统类载入器（system class loader）是默认的类载入器，从 CLASSPATH 中搜索需要的类。

那么，jvm 到底使用哪个 loader 呢？为了尽可能的安全，jvm 在这个问题上使用代理模型。每次需要载入一个新类时，首先使用系统类载入器。但是，该类并不会被立刻载入。相反，系统类载入器将这个任务代理给扩展类载入器执行，而扩展类载入器又将该任务代理给引导类载入器执行。因此，引导类载入器实际上是先载入这个

类的。当引导类载入器无法载入这个类时，扩展类载入器尝试载入这个类，若扩展类载入器也载入失败，则系统类载入器尝试载入，若还是失败，抛出 java·lang·ClassNotFoundException 异常。

为什么要走这个过程呢？

关键原因在于安全性的考虑。例如，你可以使用 security manager 来限制对某个目录的访问。现在，有人自己写了一个也叫 java·lang·Object 的类，企图访问硬盘资源。由于 jvm 自身相信名为 java·lang·Object 类，就不会去监控该类的行为，而 security manager 也就不会起作用了。

还好，这种情况不会发生。原因如下：当程序中调用了名为 java·lang·Object 类时，系统类载入器会将这个请求先交给扩展类载入器实行，而扩展类载入器则会将这个任务先转交给引导类载入器执行。引导类载入器搜索核心类库，找到标准 java·lang·Object 类。结果，自定义的恶意 java·lang·Object 类就不会被调用。

java 的类载入机制的一个好处就是，你可以通过扩展 java·lang·ClassLoader 类的方法自定义类载入器。tomcat 中使用自定义类载入器的原因如下：

（1）指定明确的类载入规则

（2）缓存已经载入的类

（3）预载入某个类，更方便使用。

8.3 Loader 接口

在载入 web 应用中需要的 servlet 和相关类时有一些明确的规则要遵守。web 应用中的 servlet 必须配部署到 WEB-INF/classes 目录或其子目录下。但是，servlet 不能访问其他的类（这里指的是在 web 应用中自定义的类），即使这些类被包含在运行着当前 tomcat 的 jvm 的 CLASSPATH 中。servlet 只能访问 WEB-INF/lib 目录下的类库。

tomcat 的 loader 指的是 web 应用 loader，而不是类 loader。loader 组件必须实现 org·apache·catalina·Loader 接口。loader 组件的使用 org·apache·catalina·loader·WebappClassLoader 类型的自定义类载入器。在 web 应用 loader 实例内部，可以使用 Loader 接口的 getClassLoader 方法获得 ClassLoader 的实例。

此外，Loader 接口中定义了一些与 repository 相关的方法。web 应用中的 WEB-INF/classes 目录和 WEB-INF/lib 目录被默认添加到 repository 中。Loader 接口的 addRepository 的方法用于添加一个新的 repository，而 findRepositories 方法则返回包含所有 repository 的数组。

tomcat 的 loader 组件通常使用 getContainer 方法和 setContainer 方法与某个 context 容器相关联。若是 context 中的某个类修改了之后，loader 可以进行重载。为了实现自动重载，Loader 接口中有一个 modified 方

法，该方法返回所有的 repository 中是否有类被修改了。但是，loader 并不负责重新载入类，相反，它会调用 Context 接口的 reload 接口。Context 接口的 setReloadable 方法和 getReloadable 方法用于决定 Loader 是否可以自动重新载入类。默认情况下，Context 接口的标准实现是不启用类自动重载功能的。因此，要启用该功能需要在 server·xml 中添加如下内容：

```
<Context path="/myApp" docBase="myApp" debug="0" reloadable="true"/>
```

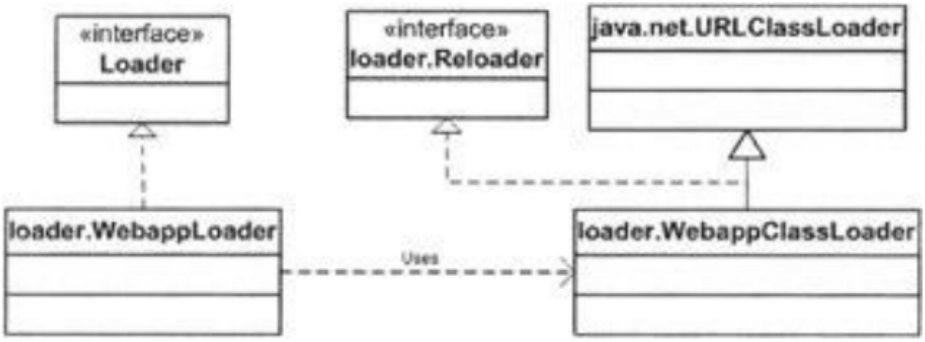
Loader 接口提供了 getDelegate 方法和 setDelegate 方法可以获知是否代理父类载入器。
Loader 接口定义如下：

Loader.java

catalina 中 org·apache·catalina·loader·WebappLoader 实现了 Loader 接口。WebappLoader 的实例中包含了一个 org·apache·catalina·loader·WebappClassLoader 实例用于载入类，WebappClassLoader 扩展了 java·net·URLClassLoader 类。

注意，当关联有 loader 的 container 需要载入一个类时，container 首先会调用其 loader 的 getClassLoader 方法获取类载入器。然后，container 调用类载入器的 loadClass 方法载入需要的类。

下图是 toncat 中 Loader 接口及其相关实现类的 UML 示意图：



图表 15　Loader 接口及其实现类的 UML 示意图

8.4　Reloader 接口

为了支持类的自动重载功能，类载入器需要实现 org·apache·catalina·loader·Reloader 接口。该接口定义如下：

```
package org·apache·catalina·loader;

public interface Reloader {
```

```
    public void addRepository(String repository);

    public String[] findRepositories ();


    public boolean modified();

}
```

其中最重要的方法是 modified 方法，该方法返回 web 应用中类是否被修改了。

8.5 WebappLoader 类

org·apache·catalina·loader·WebappLoader 类实现了 Loader 接口，表示一个 web 应用中的 loader，负责为 web 应用载入需要的类。WebappLoader 会创建一个 org·apache·catalina·loader·WebappClassLoader 的实例作为其类载入器。像其他 catalina 组件一样，WebappLoader 类要实现 org·apache·catalina·Lifecycle 接口，由其 container 负责启动/关闭。WebappLoader 类还是先了 java·lang·Runnable 接口，这样就可以使用另一个线程来重复调用其类载入器的 modified 方法来检查是否有类被修改了。若 modified 方法返回 true，WebappLoader 实例会通知其 container 来重新载入类（注意，不是 WebappLoader 执行类的重载）。

调用 WebappLoader 类的 start 方法后要执行的重要任务包括：

（1）创建一个类载入器

（2）设置 repository

（3）设置类路径

（4）设置访问权限

（5）为类的自动重载打开一个新线程

8·5·1 创建类载入器

为了重新载入类，WebappLoader 类使用了一个内部类载入器。Loader 接口中只提供了 getClassLoader 方法，而没有 setClassLoader 方法。因此，你无法为 WebappLoader 指定一个类载入器，只能使用默认的类载入器。

这样是不是不够灵活呢？

当然不是。WebappLoader 类提供了 getClassLoader 和 setClassLoader 方法来获取/修改其私有的类载入器。函数的变量是一个字符串，指明了类载入器的名字。默认情况下，loaderClass 变量的值是 org·apache·catalina·loader·WebappClassLoader。你可以通过 WebappClassLoader 来创建自己的类载入器，然后调用 setLoaderClass 方法来强制 WebappLoader 使用你自定义的类载入器。否则，启动的时候，WebappLoader 会调用其私有方法 createClassLoader 来创建一个 WebappClassLoader 的实例。createClassLoader 方法代码如下：

```
private WebappClassLoader createClassLoader() throws Exception {

    Class clazz = Class·forName(loaderClass);

   WebappClassLoader classLoader = null;

    if (parentClassLoader == null) {

      // Will cause a ClassCast if the class does not extend

      // WebappClassLoader, but this is on purpose (the exception will be

       // caught and rethrown)

      classLoader = (WebappClassLoader) clazz·newInstance();

      // in Tomcat 5, this if block is replaced by the following:

      // if (parentClassLoader == null) {

      //    parentClassLoader =

      //       Thread·currentThread()·getContextClassLoader();

      // }



   }

    else {

      Class[] argTypes = { ClassLoader·class };

      Object[] args = { parentClassLoader };

      Constructor constr = clazz·getConstructor(argTypes);
```

```
            classLoader = (WebappClassLoader) constr.newInstance(args);
        }
        return classLoader;
    }
```

注意，你自定义的类载入器要继承自 WebappClassLoader 类，否则 createClassLoader 方法在进行类型转换时会报异常。

8.5.2 设置 repository

WebappLoader 类的 start 方法会调用 setRepositories 方法将 repository 添加到其类载入器中。WEB-INF/classes 目录通过类载入的 addRepository 方法传入到类载入器中，而设置 WEB-INF/lib 目录使用的是 setJarPath 方法。

8.5.3 设置类路径

该任务是通过在 start 方法中调用 setClassPath 方法完成的。setClassPath 方法会在 servlet context 中设置一个字符串形式的属性来指明类路径信息。

8.5.4 设置访问权限

若是在 tomcat 运行时，使用了 security manager，则 setPermissions 方法会为类载入器设置访问权限，如只能访问 WEB-INF/classes 和 WEB-INF/lib 和目录。若是 security manager 没有运行，则 setPermissions 方法什么也不做。

8.5.5 开启新线程执行类的重新载入

WebappLoader 类使用一个线程不断的检查资源的时间戳，该项检查每隔一段时间执行一次，默认是每 个 15 秒执行一次。该值存储在 checkInterval 变量中，可通过 getter/setter 方法访问/修改。

在 tomcat4 中，WebappLoader 实现了 java·lang·Runnable 接口，从而支持类的自动重载。其中 run 方法实现如下：

```
public void run() {
    if (debug >= 1)
      log("BACKGROUND THREAD Starting");

    // Loop until the termination semaphore is set
    while (!threadDone) {
      // Wait for our check interval
      threadSleep();
      if (!started)
        break;
      try {
        // Perform our modification check
        if (!classLoader·modified())
          continue;
      }
      catch (Exception e) {
        log(sm·getString("webappLoader·failModifiedCheck"), e);
        continue;
      }
      // Handle a need for reloading
      notifyContext();
      break;
```

```
    }

    if (debug >= 1)
      log("BACKGROUND THREAD Stopping");
  }
```

（注意，在 tomcat5 中，检查类是否被修改的任务该有由 org·apache·catalina·core·StandardContext 类的 backgroundProcess 方法完成。这个方法被 org·apache·catalina·core·ContainerBase 实例的一个线程周期性的调用。）

在 run 方法中，使用一个 while 循环。若没有类被修改，则不停的循环。否则执行重载。while 循环的任务：

（1）sleep 一段时间，时长 checkInterval 秒；

（2）调用 WebappLoader 实例的类载入器的 modified 方法检查是否有类被修改。若没有类被修改，则重新循环；

（3）若类被修改了，调用私有方法 notifyContext，通知 WebappLoader 实例关联的 Context 重载相关类。

notifyContext 方法的实现代码如下：

```
  private void notifyContext() {
    WebappContextNotifier notifier = new WebappContextNotifier();
    (new Thread(notifier))·start();
  }
```

notifyContext 方法并不直接调用 Context 接口的 reload 方法，而是实例化一个内部类 WebappContextNotifier，然后由该对象去完成重载类的任务。WebappContextNotifier 类实现了 Runnable 接口，这样，重载类的工作就可以在另一个线程完成。WebappContextNotifier 类的定义如下：

```
  protected class WebappContextNotifier implements Runnable {
    public void run() {
      ((Context) container)·reload();
    }
```

```
}
```

8.6 WebappClassLoader 类

org·apache·catalina·loader·WebappClassLoader 类表示了一个 web 应用内的类载入器。WebappClassLoader 继承自 java·net·URLClassLoader 类（之前的章节中用该 loader 载入所需的类）。

WebappClassLoader 的设计中考虑了优化和安全两方面。例如，它会缓存之前已经载入的类来提升效率。此外，还会缓存载入失败的类，这样以后再次载入这个类时，就直接抛出 ClassNotFoundException 异常。WebappClassLoader 在 repository 指定的位置和指定的 jar 中查找需要载入的类。考虑到安全性，有一些类是 WebappClassLoader 不会载入的，这些类的名字被存储字符串数组变量 triggers 中。此外，有一些包内的都不允许载入，这些包名存储在字符串数组变量 packageTriggers 中。示例如下：

```
private static final String[] triggers = {
    "javax·servlet·Servlet"                    // Servlet API
};
private static final String[] packageTriggers = {
    "javax",                                   // Java extensions
    "org·xml·sax",                             // SAX 1 & 2
    "org·w3c·dom",                             // DOM 1 & 2
    "org·apache·xerces",                       // Xerces 1 & 2
    "org·apache·xalan"                         // Xalan
};
```

8·6·1 类缓存

为了更好的性能，已经被载入的类会被缓存起来，这样，下次再使用该类时，会从缓存中获取。缓存可以在本地执行，WebappClassLoader 实例可以管理以它所加载并缓存的类。此外，java·lang·ClassLoader maintains 为了了一个向量（vector），保存已经载入的类，防止这些类在不使用是被 gc 掉。

每个被 WebappClassLoader 载入的类（无论是在 WEB-INF/classes 目录下还是在某个 jar 内），都被视为“资源”而添加了引用。“资源”用 org·apache·catalina·loader·ResourceEntry 类表示。ResourceEntry 实例保存一些与载入类相关的信息，如修改日期，Manifest 等。该类定义如下：

```
package org·apache·catalina·loader;

import java·net·URL;

import java·security·cert·Certificate; import java·util·jar·Manifest;


public class ResourceEntry {

    public long lastModifled = -1;

    // Binary content of the resource·

    public byte[] binaryContent = null;

    public Class loadedClass = null;

    // URL source from where the object was loaded·

    public URL source = null;

    // URL of the codebase from where the object was loaded·

    public URL CodeBase = null;

    public Manifest manifest = null;

    public Certificate[] certificates = null;

}
```

所有已经载入的类会存储在一个名为 resourceEntries 的 HashMap 类型的变量中，那些载入失败的类被存储到为名 notFoundResources 的 HashMap 类型的变量中。

8·6·2 载入类

载入类时，WebappClassLoader 类要遵守如下规则：

（1）因为所有已经载入的类都会缓存起来，所以载入类时要先检查本地缓存；

（2）若本地缓存中没有，检查上一层缓存，如调用 java·lang·ClassLoader 类的 findLoadedClass 方法；

（3）若两个缓存中都没有，则使用系统类载入器进行加载，防止 web 应用中的类覆盖 J2EE 的类（注：不能直接使用 WebappClassLoader 类在 repository 中查找类，要先将加载的任务代理给系统类载入器，保证 java 核心类，CLASSPATH 中的类先载入）；

（4）若启用了 SecurityManager，检查该类是否被允许载入。若该类是黑名单中的一员，抛出 ClassNotFoundException 异常；

（5）若标志位 delegate 为 true（可以代理其 parent 类载入器），或者要载入的类所在的包名在 packageTrigger 中存在，则使用 parent 类载入器。若 parent 类载入器为 null，使用系统类载入器；

（6）从当前 repository 中载入类；

（7）若当前 repository 中没有需要的类，且标志位 delegate 为 false，使用 parent 类载入器。若 parent 类载入器为 null，使用系统类载入器；

（8）若仍未找到需要的类，抛出 ClassNotFoundException 异常。

8·6·3 应用程序

本章的程序用于展示如何使用 WebappLoader 类。tomcat 中 Context 接口的标准实现是 org·apache·catalina·core·StandardContext 类。本章程序中使用也是该类。你需要注意个是 StandardContext 类是如何工作的，如何监听事件的。监听器必须实现 org·apache·catalina·lifecycle·LifecycleListener 接口，调用 StandardContext 类的 setConfigured 方法。本章的应用程序中监听器由 ex08·pyrmont·core·SimpleContextConfig 类表示，该类定义如下：

```
package ex08·pyrmont·core;

import org·apache·catalina·Context;

import org·apache·catalina·Lifecycle;

import org·apache·catalina·LifecycleEvent;
```

```
import org.apache.catalina.LifecycleListener;


public class SimpleContextConfig implements LifecycleListener {
    public void lifecycleEvent(LifecycleEvent event) {
      if (Lifecycle.START_EVENT.equals(event.getType())) {
        Context context = (Context) event.getLifecycle();
        context.setConfigured(true);
      }
    }
}
```

你只需要将 StandardContext 类和 SimpleContextConfig 类实例化，并调用 org.apache.catalina.Lifecycle 接口的 addLifecycleListener 方法，将 SimpleContextConfig 实例注册到 StandardContext 实例中。

此外，应用程序保留了前面章节的借各类 SimplePipeline，SimpleWrapper 和 SimpleWrapperValve。该程序可使用 PrimitiveServlet 和 ModernServlet 进行测试。这次使用了 StandardContext，则 servlet 必须位于应用程序目录的 WEB-INF/classes 目录下。

为了告诉 StandardContext 实例 web 应用的位置，你需要设置一个系统属性（不是操作系统），名为“catalina.base”，值为“user.dir”属性的值。如下所示：

```
System.setProperty("catalina.base", System.getProperty("user.dir"));
```

然后，在 main 方法中实例化默认 connector：

```
Connector connector = new HttpConnector();
```

然后，创建两个 wrapper，并实例化：

```
Wrapper wrapper1 = new SimpleWrapper();
wrapper1.setName("Primitive");
wrapper1.setServletClass("PrimitiveServlet");
```

```
Wrapper wrapper2 = new SimpleWrapper();
wrapper2.setName("Modern");
wrapper2.setServletClass("ModernServlet");
```

再创建一个 StandardContext 实例，设置路径：

```
Context context = new StandardContext();
// StandardContext's start method adds a default mapper
context.setPath("/myApp");
context.setDocBase("myApp");
```

上面的代码的功能等同于下面 server.xml 的配置：

```
<Context path="/myApp" docBase="myApp"/>
```

将两个 wrapper 添加到 context 中，为它们设置 mapping：

```
context.addChild(wrapper1);
context.addChild(wrapper2);
context.addServletMapping("/Primitive", "Primitive");
context.addServletMapping("/Modern", "Modern");
```

下一步，实例化一个 listener，并注册到 context 中：

```
LifecycleListener listener = new SimpleContextConfig();
((Lifecycle) context).addLifecycleListener(listener);
```

接着，实例化 loader（WebappLoader），并关联到 context 中：

```
Loader loader = new WebappLoader();
```

```
context·setLoader(loader);
```

然后，将 context 与 connector 关联，调用 connector 的 initialize 和 start 方法，再调用 context 的 start 方法：

```
connector·setContainer(context);
try {
    connector·initialize();
    ((Lifecycle) connector)·start();
    ((Lifecycle) context)·start();
```

接下来的工作仅仅是展示 resources 的 docBase 和类载入器中所有的 repository：

```
// now we want to know some details about WebappLoader
 WebappClassLoader classLoader = (WebappClassLoader)
    loader·getClassLoader();
  System·out·println("Resources' docBase: " +
   ((ProxyDirContext)classLoader·getResources ())·getDocBase());
  String[] repositories = classLoader·findRepositories();
  for (int i=0; i<repositorles·length; i++) {
    System·out·println("  repository: " + repositories[i]);
  }
```

最后，程序在用户输入 Enter 键后退出：

```
// make the application wait until we press a key·
System·in·read();
((Lifecycle) context)·stop();
```

第 9 章 session 管理

9.1 概述

catalina 通过一个成为 manager 的组件管理 session，该组件要实现 org·apache·catalina·Manager 接口，且必须与一个 context 关联。相比于其他组件，manager 负责创建、更新、销毁 session 对象。

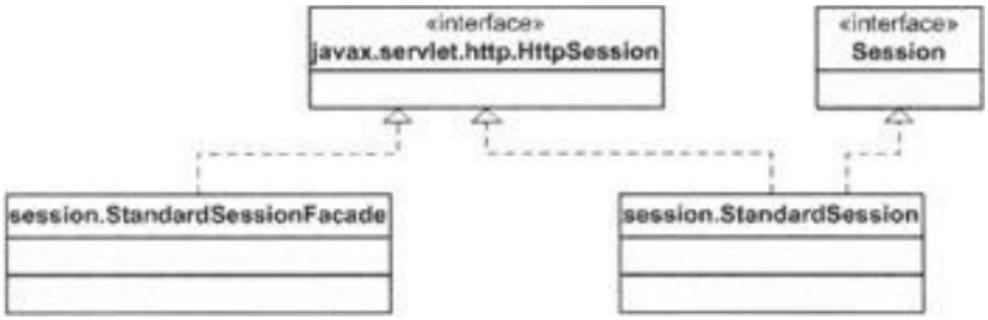
servlet 可以通过 javax·servlet·http·HttpServletRequest 接口的 getSession 方法获取 session 对象，在 catalina 的默认 connector 中，使用 org·apache·catalina·connector·HttpRequestBase 类表示 HttpServletRequest。下面是 HttpRequestBase 类的一些方法：

HttpRequestBase.java

默认情况下，manager 将 session 对象存储在内存中。但 tomcat 中，也可以将 session 对象持久化到文件或数据库中。catalina 的 org·apache·catalina·session 包内有 session 管理的相关类。

9.2 Sessions

在 servlet 编程方面，session 对象由 javax·servlet·http·HttpSession 接口表示。catalina 中 Session 接口的标准实现是 StandardSession 类，位于 org·apache·catalina·session 包内。但是，为了安全起见，manager 并不直接将 session 对象传给 servlet，而是使用了 session 的外观类 StandardSessionFacade（位于 org·apache·catalina·session 包内）。在内部实现上，manager 使用了另一个外观类，org·apache·catalina·Session 接口。相关类的 UML 示意图如下所示：



图表 16 session 管理相关类的 UML 图

注意：Session 类，StandardSession 类和 StandardSessionFacade 类的前缀 org·apache·catalina 被省略掉了。

9·2·1 Session 接口

Session 接口是作为 catalina 的外观类使用的。Session 对象的标准实现 StandardSession 类也实现了 javax·servlet·http·HttpSession 接口。Session 接口定义如下：

Session.java

session 对象总是存在于 manager 组件中，getManager/setManager 方法用于设置 session 和 manager 的关联。对某个 session 实例来说，在与其关联的 manager 内，该 session 有唯一标识，该标识可通过 setId/getId 方法来访问。manager 调用 getLastAccessedTime 方法来决定某个 session 对象的有效性。manager 调用 setValid 方法将某个 session 对象标识为有效。每次访问 session 对象时，它的调用方法都会修改该 session 对象的最后访问时间。最后，manager 可以通过调用 expire 方法将某个 session 对象标识为过期，也可以通过 getSession 方法获取一个经过 session 外观类包装过的 HttpSession 对象。

9·2·2 StandardSession 类

StandardSession 是 catalina 中 Session 接口的标准实现。除了实现 javax·servlet·http·HttpSession 接口和 org·apache·catalina·Session 接口外，StandardSession 类还实现了 java·lang·Serializable 接口。

StandardSession 的构造函数接收一个 Manager 类的实例，迫使一个 session 实例必须属于一个 manager 实例：

```
public StandardSession(Manager manager);
```

下面是 StandardSession 实例的一些比较重要的变量，用于保存该 StandardSession 实例的一些状态。注意，带有 transient 修饰符的变量无法被序列化。

```
// session attributes

private HashMap attributes = new HashMap();
```

```
// the authentication type used to authenticate our cached Principal, if any
private transient String authType = null;
private long creationTime = 0L;
private transient boolean expiring = false;
private transient StandardSessionFacade facade = null;
private String id = null;
private long lastAccessedTime = creationTime;
// The session event listeners for this Session·
private transient ArrayList listeners = new ArrayList();
private Manager manager = null;
private int maxInactiveInterval = -1;
// Flag indicating whether this session is new or not·
private boolean isNew = false;
private boolean isValid = false;
private long thisAccessedTime = creationTime;
```

注意，在 tomcat5 中，上面的变量的访问修饰符已经改为 protected。

getSession 方法会返回一个 StandardSessionFacade 类的实例：

```
public HttpSession getSession() {
    if (facade == null)
        facade = new StandardSessionFacade(this); //传入该 session 实例
    return (facade);
}
```

当某个 session 对象在超过一定时间没有访问后，会通过调用 Session 接口的 expire 方法将该 session 对象标识为过期，这个时间是由变量 maxInactiveInterval 表示的。下面的代码是该方法在 StandardSession 中的实现：

```
public void expire(boolean notify) {

    // Mark this session as "being expired" if needed
    if (expiring)
        return;
    expiring = true;
    setValid(false);
    // Remove this session from our manager's active sessions
    if (manager != null)
        manager.remove(this);
    // Unbind any objects associated with this session
    String keys [] = keys();
    for (int i = 0; i < keys.length; i++)
        removeAttribute(keys[i], notify);
    // Notify interested session event listeners
    if (notify) {
        fireSessionEvent(Session.SESSION_DESTROYED_EVENT, null);
    }


    // Notify interested application event listeners
    // FIXME - Assumes we call listeners in reverse order
    Context context = (Context) manager.getContainer();
    Object listeners[] = context.getApplicationListeners();
```

```
if (notify && (listeners != null)) {
  HttpSessionEvent event = new HttpSessionEvent(getSession());

  for (int i = 0; i < listeners.length; i++) {
    int j = (listeners.length - 1) - i;
    if (!(listeners[j] instanceof HttpSessionListener))
      continue;
    HttpSessionListener listener =
     (HttpSessionListener) listeners[j];
    try {
      fireContainerEvent(context, "beforeSessionDestroyed",
        listener);
      listener.sessionDestroyed(event);
      fireContainerEvent(context, "afterSessionDestroyed", listener);
    }
    catch (Throwable t) {
      try {
        fireContainerEvent(context, "afterSessionDestroyed",
          listener);
      }
      catch (Exception e) {
        ;
      }
```

```
            // FIXME - should we do anything besides log these?
            log(sm.getString("standardSession.sessionEvent"), t);
          }
        }    }

    // We have completed expire of this session
    expiring = false;
    if ((manager != null) && (manager instanceof ManagerBase)) {
      recycle();
    }
  }
```

9.2.3 StandardSessionFacade 类

为了传递一个 session 对象给 servlet，catalina 会实例化一个 Session 对象，填充 session 对象内容，然后再传给 servlet。但是，实际上，catalina 传递的是 session 的外观类 StandardSessionFacade 的实例，该类仅仅实现了 javax.servlet.http.HttpSession 接口。这就，servlet 开发人员就不能将 HttpSession 对象向下转换为 StandardSessionFacade 类型，也就不会暴露出 StandardSessionFacade 所有的公共接口了。

9.3 Manager

manager 组件负责管理 session 对象，例如创建和销毁 session 对象。catalina 中 manager 组件必须要实现 org.apache.catalina.Manager 接口。org.apache.catalina.session 包内的 ManagerBase 是 Manager 接口的基本实现，它有两个直接子类：StandardManager 类和 PersistentManagerBase 类。

当运行 tomcat 时，StandardManager 实例负责在内存中管理 session，但当服务器关闭时，会将当前内存中的 session 写入到文件中，等服务器再次启动时，会重新载入这些 session。

PersistentManagerBase 类是 manager 组件的基类，将 session 对象存储在二级存储设备中。它有两个直

接子类，PersistentManager 类和 DistributedManager 类（DistributedManager 类仅存在于 tomcat4 中）。相关类的 UML 示意图如下：

图表 17　Manager 接口及其相关类的 UML 示意图

9.3.1　Manager 接口

Manager 接口定义如下：

Manager.java

Manager 接口通过 getContainer/setContainer 来与某个 container 相关联。使用 createSession 方法来创建一个 session 对象。使用 add 方法将某个 session 对象添加到 session 池中，或使用 remove 方法将某个 session 删除。getMaxInactiveInterval/setMaxInactiveInterval 来访问/设置 maxInactiveInterval 变量，单位为秒。

使用 unload 方法可以将 session 对象持久化到二级存储设备中，load 方法则可以将其载入到内存中。

9.3.2　ManagerBase 类

ManagerBase 类是一个抽象类，实现了 Manager 接口。该类提供了一些基本的功能。ManagerBase 的 createSession 方法会创建一个新的 session 对象。其 protected 方法 generateSessionId 方法会返回一个 session 的唯一标识符。

活动 session 对象存储在一个名为 sessions 的 HashMap 类型的变量中：

```
protected HashMap sessions = new HashMap();
```

下面是 add 和 remove 方法的实现，注意同步的使用：

```
public void add(Session session) {
    synchronized (sessions) {
        sessions.put(session.getId(), session);
    }
}
public void remove(Session session) {
    synchronized (sessions) {
        sessions.remove(session.getId());
    }
}
```

findSessions 方法（无参数）以数组形式返回所有活动的 session 对象，findSession 方法则返回某个指定 id 的 session 对象。

```
public Session[] findSessions() {
    Session results[] = null;
    synchronized (sessions) {
        results = new Session[sessions.size()];
        results = (Session[]) sessions.values().toArray(results);
    }
    return (results);
}
```

```
    public Session findSession(String id) throws IOException {
        if (id == null)
            return (null);
        synchronized (sessions) {

            Session session = (Session) sessions.get(id);
            return (session);
        }
    }
```

9.3.3 StandardManager 类

StandardManager 类是 Manager 接口的标准实现，该类将 session 对存储与内存中。StandardManager 类还是先了 Lifecycle 接口，可由其 container 负责启动/关闭。其中，stop 方法会调用 unload 方法将 session 对象序列化到一个名为 Session.ser 的文件中，每个 context 一个文件。SESSIONS.ser 文件位于 CATALINA_HOME 目录下的 work 目录中。当 StandardManager 再次启动时，会调用 load 方法从文件再次读取 session 对象到内存中。

当 session 无效时，manager 组件要负责销毁 session 对象。在 tomcat4 中，StandardManager 类使用另一个线程完成此任务。因此，StandardManager 类还要实现 java.lang.Runnable 接口。下面是 run 方法的实现：

```
    public void run() {
        // Loop until the termination semaphore is set
        while (!threadDone) {
            threadSleep();
            processExpires();
        }
```

```
}
```

threadSleep 方法会使线程休息一段时间，时间长度由变量 checkInterval 指定，单位为秒，默认为 60，可通过 setCheckInterval 方法进行设置。

processExpire 方法变量 manager 管理的所有的 session，比较当前时间与 session 对象的 lastAccessedTime 属性值。若两个时间差值大于 maxInactiveInterval，则调用 session 对象的 expire 方法将其标识为过期的。变量 maxInactiveInternal 的值可通过 setMaxInactiveInterval 方法进行设置，时间单位是秒。但是，不要傻乎乎认为 tomcat 部署时使用的值。org·apache·catalina·core·ContainerBase 类的 setManager 方法会 setContainer 方法，，而 setContainer 方法会修改 maxInactiveInterval 的值。

在 tomcat5 中，StandardManager 类没有实现 java·lang·Runnable 接口，processExpires 方法会直接被 backgroundProcess 方法调用，tomcat4 中并不包含 backgroundProcess 方法：

```
public void backgroundProcess() {
    processExpires();
}
```

StandardManager 实例的 backgroundProcess 方法会被 org·apache·catalina·core·StandardContext 实例的 backgroundProcess 方法调用。而 StandardContext 类会周期性的调用 backgroundProcess 方法。

9·3·4 PersistentManagerBase 类

PersistentManagerBase 类是所有持久化 manager 的父类。StandardManager 类和持久化 manager 的区别在于后者的存储方式（store）。store 表示了管理 session 对的二级存储设备。PersistentManagerBase 使用私有变量 store 保存对二级存储设备的引用：

```
private Store store = null;
```

在持久化 manager 中，session 对象可以备份也可换出。备份时，session 对象被拷贝到 store 中。当内存中 session 对象超过一定数量或某个 session 长时间未被访问时，该 session 会被换出，它会被写到 store 中。换出 session 是为了节省内存。

在 tomcat4 中，PersistentManagerBase 实现了 java·lang·Runnable 接口，使用另一个线程周期性的备份

和换出 session。下面的 run 方法的实现：

```
public void run() {

    // Loop until the termination semaphore is set

    while (!threadDone) {

      threadSleep();

      processExpires();

      processPersistenceChecks();

    }

}
```

像 StandardManager 类中一样，processExpired 方法检查 session 对象是否过期。processPersistenceChecks 方法会调用其他三个方法：

```
public void processPersistenceChecks() {

    processMaxIdleSwaps();

    processMaxActiveSwaps();

    processMaxIdleBackups();

}
```

在 tomcat5 中，PersistentManagerBase 类不再实现 java·lang·Runnable 接口。备份和换出 session 是在 backgroundProcess 方法中完成的，该方法被与该 manager 关联的 StandardContext 实例周期性的调用。

9.3.4.1 swap out（换出）

PersistentManagerBase 类在换出 session 对象时要遵守一定的规则。只有当活动 session 的数量超过一定限制，或某个 session 已经过期时，才能将其换出。

当有过多的 session 对象时，PersistentManagerBase 类的实例只是很简单的将任意 session 对象换出，直到 session 对象的数量等于 maxActiveSessions。

当某个 session 对象长时间没有访问时，PersistentManagerBase 类使用两个变量来判断 session 对象是否要被换出：minIdleSwap 和 maxIdleSwap。若是 session 对象的 lastAccessedTime 大于 minIdleSwap 和 maxIdleSwap 时，则要换出该 session。若希望所有的 session 都不被换出，可将 maxIdleSwap 的值置为负数。

由于活动 session 对象可以被换出，所以，某个活动的 session 可能在内存中，也可能在 store 中。因此，调用 findSession(String id)时，会先在内存中查找，若内存中没有，则从 store 中查找。下面是 PersistentManagerBase 类中 findSession 方法的实现：

```
public Session findSession(String id) throws IOException {

    Session session = super.findSession(id);

    if (session != null)

      return (session);

    // not found in memory, see if the Session is in the Store

    session = swapIn(id);

    // swapIn returns an active session in the Store

    return (session);

}
```

9.3.4.2 back up（备份）

不是所有的 session 都会备份，PersistentManagerBase 类只会备份那些空闲时间超过 maxIdleBackup 的 session。该任务由 processMaxIdleBackups 方法完成。

9.3.5 PersistentManager 类

PersistentManager 类继承自 PersistentManagerBase 类，并没有添加其他的方法，只是多了两个属性：

```
package org.apache.catalina.session;

public final class PersistentManager extends PersistentManagerBase {

    // The descriptive information about this implementation.
```

```
    private static final String info = "PersistentManager/1.0";

    // The descriptive name of this Manager implementation (for logging).
    protected static String name = "PersistentManager";

    public String getInfo() {
      return (this.info);
    }

    public String getName() {
      return (name);
    }
  }
```

9.3.6 DistributedManager 类

tomcat4 中提供了 DistributedManager 类，该类继承自 PersistentManagerBase 类。该类用于两个或多于两个节点的集群环境。一个节点表示了一个 tomcat 的部署。集群中的节点可以在同一台物理机器，也可以在不同的物理机器。在集群环境中，每个节点必须使用 DistributedManager 实例作为其 manager，才能支持 session 的复制，这也是 DistributedManager 类的主要功能。

为了复制 session，当创建或销毁 session 对象时，DistributedManager 实例会向其他节点发送消息。此外，集群中的节点也必须能够接收其他节点发送的消息。

为了发送和接收消息，在 org.apache.catalina.cluster 包内有一些可供使用的工具类。其中，ClusterSender 类用于发送消息，ClusterReceiver 用于接收消息。

DistrbutedManager 实例的 createSession 方法要创建一个 session 对象存储在当前 DistrbutedManager 实例中，并使用 ClusterSender 实例向其他节点发送消息。createSession 方法实现如下：

```
public Session createSession() {
    Session session = super.createSession();
    ObjectOutputStream oos = null;
    ByteArrayOutputStream bos = null;
```

```
        ByteArrayInputStream bis = null;
         try {
           bos = new ByteArrayOutputStream();
           oos = new ObjectOutputStream(new BufferedOutputStream(bos));

           ((StandardSession)session).writeObjectData(oos);
           oos.close();
           byte[] obs = bos.toByteArray();
           clusterSender.send(obs);
           if(debug > 0)
              log("Replicating Session: "+session.getId());
        }
        catch (IOException e) {
           log("An error occurred when replicating Session: " +
              session.getId());
        }
        retun (session);
    }
```

其中，createSession 方法先调用父类的 createSession 方法创建一个 session 对象，然后以字节数组的形式将 session 对象发送到其他节点。

DistribubedManager 类还实现了 java.lang.Runnable 接口，使用另一个线程来将某个 session 标识为过期，或接收其他节点发送的消息。run 方法实现如下：

```
    public void run() {
        // Loop until the termination semaphore is set
        while (!threadDone) {
```

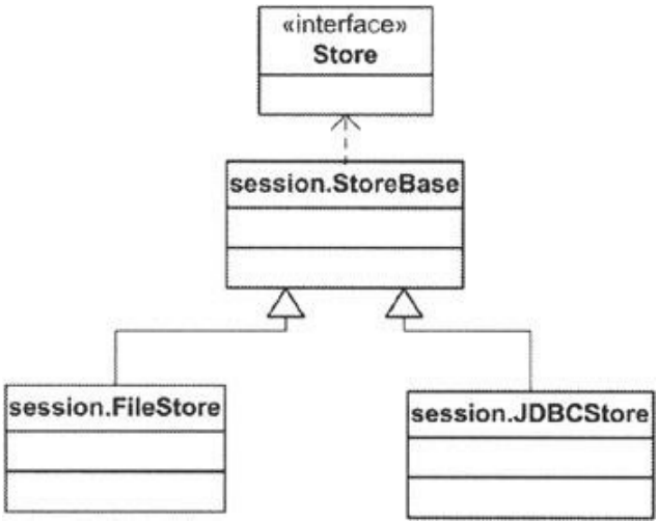
```
            threadSleep();
            processClusterReceiver();
            processExpires();
            processPersistenceChecks();
        }
    }
```

9.4 Stores

store 由 org·apache·catalina·Store 接口表示。manager 组件使用 store 作为持久化 session 的存储设备。Store 接口定义为：

Store.java

其中 save 和 load 是两个比较重要的方法。save 方法将指定的 session 对象持久化到 store 中。load 方法则根据参数 id 从 store 中找到对象的 session，载入到内存中。keys 方法返回所有 session 的 id 数组。相关类的 UML 示意图如下所示：



图表 18　Store 接口及相关类的 UML 示意图

9.4.1 StoreBase 类

StoreBase 类是一个抽象类，提供了 store 的基本功能。该类有两个直接子类：FileStore 类和 JDBCStore 类。StoreBase 类并没有实现 Store 接口的 save 方法和 load 方法，因为，这两个方法依赖于具体的持久化设备。

tomcat4 中，StoreBase 类实现了 java.lang.Runnable 接口，使用另一个线程周期性的检查 session，从活动 session 的集合中删除过期的 session。下面是 run 方法的实现：

```
public void run() {

  // Loop until the termination semaphore is set

  while (!threadDone) {

    threadSleep();

    processExpires();

  }

}
```

processExpires 方法获取所有的活动 session 对象，检查每个 session 的 lastAccessedTime 值，删除那些长时间不活动的 session 对象。下面是 processExpires 方法的实现：

StoreBase.java

在 tomcat5 中，不再使用专用的线程调用 processExpires 方法，相反，由与该 store 相关联的 PersistentManagerBase 实例的 backgroundProcess 方法周期性的调用 processExpires 方法。

9.4.2 FileStore 类

FileStore 类将 session 对象持久化到某个文件中。文件名的格式为“sessionid+‘.session’”。文件位于临时的工作目录下，可以调用 FileStore 类的 setDirectory 方法修改临时目录的位置。

save 方法使用 java.io.ObjectOutputStream 类将 session 对象进行序列化。因此，session 实例中存储的所有对象都要实现 java.lang.Serializable 接口。load 方法使用 java.io.ObjectInputStream 接口实现 session 的反序列化。

9.4.3 JDBCStore 类

JDBCStore 类将 session 对象通过 jdbc 存入数据库中。因此，为了使用 JDBCStore，你需要调用 setDriverName 方法和 setConnectionURL 方法来设置数据库驱动和连接 url。

9.5 应用程序

本章的应用程序与第 8 章类似，使用默认的 connector，使用一个 context 作为容器，容器中有一个 wrapper。区别一，context 中有一个 StandardManager 类的实例，用于管理 session 对象。

9.5.1 Bootstrap 类

本章的 Bootstrap 类与前几章的 Bootstrap 类的区别在于，本章的 Bootstrap 类会创建一个 org.apache.catalina.session.StandardManager 类的实例，并将之与 Context 相关联。

main 方法会先设置“catalina.base”系统属性，并实例化 connector：

```
System.setProperty("catalina.base", System.getProperty("user.dir"));
Connector connector = new HttpConnector();
```

创建 wrapper：

```
Wrapper wrapper1 = new SimpleWrapper();
wrapper1.setName("Session");
wrapper1.setServletclass("SessionServlet");
```

再创建 StandardContext 对象，设置 path 和 docBase 属性，并将 wrapper 添加到 context 中：

```
Context context = new StandardContext();
context.setPath("/myApp");
```

```
context.setDocBase("myApp");

context.addChild(wrapper1);
```

接下来设置 servlet 的映射：

```
context.addServletMapping("/myApp/Session", "Session");
```

为 context 创建监听器和 loader：

```
LifecycleListener listener = new SimpleContextConfig();

((Lifecycle) context).addLifecycleListener(listener);

// here is our loader

Loader loader = new WebappLoader();

// associate the loader with the Context

context.setLoader(loader);

connector.setContainer (context);
```

接下来是本章的重点内容，manager：

```
Manager manager = new StandardManager();

context.setManager(manager);
```

最后，初始化并启动 connector 和 context：

```
connector.initialize();

((Lifecycle) connector).start();

((Lifecycle) context).start();
```

9·5·2 SimpleWrapperValve 类

回忆一下本章开头的内容，当 servlet 调用 javax·servlet·http·HttpServletRequest 接口的 getSession 方法获取 session 实例。当调用 getSession 方法时，request 对象必须访问与 context 相关联的 manager。manager 组件要么创建一个新的 session，要么返回一个已经存在的 session。request 对象为了能够访问 manager，它必须能够访问 context。为了达到此目的，在 SimpleWrapperValve 类的 invoke 方法中，可以调用 org·apache·catalina·Request 接口的 setContext 方法，传入 Context 实例。因此，必须在 servlet 的 service 方法调用前设置好 context。下面是 SimpleWrapperValve 类的 invoke 方法的实现：

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


    SimpleWrapper wrapper = (SimpleWrapper) getContainer();
    ServletRequest sreq = request·getRequest();
    ServletResponse sres = response·getResponse();
    Servlet servlet = null;
    HttpServletRequest hreq = null;
    if (sreq instanceof HttpServletRequest)
      hreq = (HttpServletRequest) sreq;
    HttpServletResponse hres = null;
    if (sres instanceof HttpServletResponse)
      hres = (HttpServletResponse) sres;


    // pass the Context to the Request object so that
    // the Request object can call the Manager
    Context context = (context) wrapper·getparent();
    request·setcontext(context);
```

```
    // Allocate a servlet instance to process this request
    try {
      servlet = wrapper.allocate();
      if (hres!=null && hreq!=null) {
        servlet.service(hreq, hres);
      }
      else {
        servlet.service(sreq, sres);
      }
    }
    catch (ServletException e) {
    }
}
```

第 10 章 安全性

10.1 概述

一些 web 应用的内容是受限的，只有有特定权限的用户才能访问。本章将介绍 container 如何支持安全性控制。

servlet 容器通过一个称为 authenticator 的 valve 来支持安全认证。当 container 启动时，authenticator 被添加到 container 的 pipeline 中。在 wrapper 被调用之前，会先调用 authenticator，用来对用户进行认证。若用户输入了正确的用户名密码，则 authenticator 会调用下一个 valve，否则会直接返回，不再继续执行剩余的 valve。

authenticator 调用 context 的 realm 的 authenticate 方法对用户身份进行认证。

本章会先介绍一些 servlet 编程中与安全相关的对象（realm，role，principal 等），然后用一个应用程序来展示如何为 servlet 添加基本的认证功能。

10.2 Realm（领域）

realm 是通过用户名密码对用户身份进行认真的组件，通常附属于某个 context，而一个 container 也只能有一个 realm。可以调用 container 的 setRealm 方法将 realm 和 container 相关联。

realm 如何对用户进行认证呢？答案是使用存储在 realm 实例的信息或从 store 中查找需要的信息，这取决于 realm 的具体实现。在 tomcat 中，用户信息被存储在 tomcat-users·xml 文件中。

在 catalina 中，realm 由 org·apache·catalina·Realm 接口表示，该接口中最重要的方法是，经常使用的是第一个重载方法：

```
public Principal authenticate(String username, String credentials);

public Principal authenticate(String username, byte[] credentials);

public Principal authenticate(String username, String digest, String nonce, String nc, String cnonce,

String qop, String realm, String md5a2);

public Principal authenticate(X509Certificate certs[]);
```

Realm 接口中还有一个 hasRole 方法：

```
public boolean hasRole(Principal principal, String role);
```

此外，getContainer／setContainer 用来与某个 container 相关联。

tomcat 中 Realm 接口的基本实现是 org·apache·catalina·realm·RealmBase 类。org·apache·catalina·realm 包内，还提供了一些 RealmBase 类的继承类，如 JDBCRealm，JNDIRealm，MemoryRealm 和 UserDatabaseRealm。默认情况下，使用的是 MemoryRealm 类。当 MemoryRealm 类启动时，它会读取 tomcat-users·xml 文件中的内容。本章的应用程序中，将会建立一个自定义的 realm。

10.3 GenericPrincipal

principal 由 java·security·Principal 接口表示。在 catalina 中的实现是 org·apache·catalina·realm·GenericPrincipal 类。一个 GenericPrincipal 实例必须关联一个 realm，如下面的构造方法所示：

```
public GenericPrincipal(Realm realm, String name, String password) {
  this(realm, name, password, null);
}
public GenericPrincipal(Realm realm, String name, String password,
  List roles) {
  super();
  this·realm = realm;
  this·name = name;
  this·password = password;
  if (roles != null) {
    this·roles = new String[roles·size()];
    this·roles = (String[]) roles·toArray(this·roles);
    if (this·roles·length > 0)
```

```
            Arrays.sort(this.roles);
        }
    }
```

GenericPrincipal 实例必须有用户名和密码，但角色列表 roles 是可选的。可以通过调用 hasRole 方法来检查该 principal 是否有指定的 role。下面是 tomcat4 中 hasRole 的实现：

```
public boolean hasRole(String role) {
    if (role == null)
        return (false);
    return (Arrays.binarySearch(roles, role) >= 0);
}
```

tomcat5 实现了 servlet2.4 标准，因此需要能够对通配符"*"（表示任意角色）进行识别：

```
public boolean hasRole(String role) {
    if ("*".equals(role)) // Special 2.4 role meaning everyone
        return true;
    if (role == null)
        return (false);
    return (Arrays.binarySearch(roles, role) >= 0);
}
```

10.4 LoginConfig

登录配置包含了一个 realm 名，由 org.apache.catalina.deploy.LoginConfig 类表示，该类是一个 final 类。

LoginConfig 类将 realm 名和相应的认证方法进行了封装。调用 LoginConfig 的 getRealmName 方法可以获得 realm 的名字，调用 getAuthName 方法获取认证方法。其中认证方法必须是下面的可选项之一：BASIC，DIGEST，FORM 或 CLIENT-CERT。若使用了基于表单的认证方式，则 LoginConfig 实例也可以使用属性 loginPage 和 errorPage 分别表示登录 url 和错误页面。

在部署后，启动 tocmat 时，它会读取 web·xml 文件。若 web·xml 文件包含了 login-config 元素，tomcat 会创建一个 LoginConfig 实例，并设置其属性。authenticator 会调用 LoginConfig 的 getRealmName 方法获取 realm 名，然后将其发送给浏览器显示在登录对话框中。若 realm 名为空，则会将服务名和端口发送给浏览器。如下图所示：

图表 19　基本认证对话框

10.5 Authenticator

org·apache·catalina·Authenticator 接口表示了一个 authenticator 组件。它没有任何方法，只是起到一个标志位的作用，这样，其他的组件就可以检测某个组件是否是一个 authenticator。

catalina 提供了一个 Authenticator 接口的基本实现，org·apache·catalina·authenticator·AuthenticatorBase 类。除了实现 Authenticator 接口外，AuthenticatorBase 类还扩展了 org·apache·catalina·valves·ValveBase 类，也就是说，AuthenticatorBase 类也是一个 valve。org·apache·catalina·authenticator 包下有一些 Authenticator 接口的是实现类，例如 BasicAuthenticator（支持基本认证），FormAuthenticator（基于表单的认证），DigestAuthentication（支持摘要的认证），SSLAuthenticator（支持 SSL 的认证）。此外，若 tomcat 用户没有为 auth-method 指定值时，会使用 NonLoginAuthenticator 类进行认真。NonLoginAuthenticator 类仅仅检查安全性，

并不涉及用户身份认证。

下图是 org·apache·catalina·authenticator 包中相关类的 UML 示意图：

图表 20　认证相关类的 UML 示意图

10.6　安装 Authenticator

login-config 元素仅仅在部署描述符中出现一次，login-config 中包含了 auth-method 元素来指定认证方法。也就是说，一个 context 中只能有一个 LoginConfig 实例，和一个 authentication 的实例。

下面是 auth-method 与 authenticator 的对应表：

auth-method	authenticator
BASIC	BasicAuthenticator
FORM	FormAuthenticator
DIGEST	DigestAuthenticator
CLIENT-CERT	SSLAuthenticator

若是没有使用 auth-method 元素，则默认设置为 NONE，这时使用 NonLoginAuthenticator 进行安全认证。

由于 authenticator 是在运行时才确定了，因此相应类也是动态载入的。StandardContext 类使用 org·apache·catalina·startup·ContextConfig 类来 StandardContext 的实例进行属性配置，其中就包括 authenticator 的安装，以及将 authenticator 与 context 进行关联。本章的应用程序使用了 ContextConfig 的简单实现来动态的载入 BasicAuthenticator 类。

10.7 应用程序

本章的应用程序会使用一些 catalina 中与安全相关的类。还要使用 SimplePipeline 类和 SimpleWrapper 类，而 SimpleWrapperValve 类和 SimpleContextConfig 类则与第 9 章类似，但 SimpleContextConfig 类的 authenticatorConfig 方法会将一个 BasicAuthenticator 实例添加到 StandardContext 中。

第 1 个应用程序使用两个类，ex10·pyrmont·startup·Bootstrap1 和 ex10·pyrmont·realm·SimpleRealm。第 2 个应用程序使用另两个类，ex10·pyrmont·startup·Bootstrap2 和 ex10·pyrmont·realm·SimpleUserDatabaseRealm。

10·7·1 ex10·pyrmont·core·SimpleContextConfig 类

SimpleContextConfig 类与第 9 章类似，org·apache·catalina·core·StandardContext 实例将该实例的 configured 属性设置为 true。但是本章程序中，SimpleContextConfig 添加了 authenticatorConfig 方法（由 lifeCycleEvent 方法调用）。authenticatorConfig 方法实例化一个 BasicAuthenticator 对象，并将其添加到 StandardContext 的 pipeline 中。SimpleContextConfig 类的实现如下：

SimpleContextConfig.java

authenticatorConfig 方法会检查关联的 container 中是否有安全限制。若没有，则该方法直接返回。若有，则 authenticatorConfig 会检查 context 中是否有 LoginConfig 对象。若没有，则实例化一个。然后，会见 StandardContext 对象的 pipeline 中的 basic valve 或附加的 vavle 是否是一个 authenticator。因为，context 中只能有一个 authenticator，所以只要找到一个 authenticator，则 authenticatorConfig 方法就直接返回。然后，会检查 context 中是否关联有 realm。若没有，就不需要安装 authenticator，返回即可。否则，就动态的载入 BasicAuthenticator 类，实例化一个对象，并将其作为一个 valve 添加到 StandardContext 实例中。

10·7·2 ex10·pyrmont·realm·SimpleRealm 类

SimpleRealm 类主要用来说明 realm 是如何工作的。代码如下：

SimpleRealm.java

SimpleRealm 类实现了 Realm 接口。在其构造函数中，调用 createUserDatabase 方法创建两个用户。每个用户用一个内部类 User 表示，并为每个用户设置了角色，最后将两个用户添加到 ArrayList 类型的 users 变量中。SimpleRealm 类提供了 authenticate 方法的实现，代码如下：

```
public Principal authenticate(String username, String credentials) {
    System.out.println("SimpleRealm.authenticate()");
    if (username==null || credentials==null)
      return null;
    User user = getUser(username, credentials);
    if (user==null)


      return null;
    return new GenericPrincipal(this, user.username,
      user.password, user.getRoles());
  }
```

该方法被 authenticator 调用，若用户信息无效，则返回 null。否则，返回一个表示 user 的 Principal 对象。

10.7.3 ex10.pyrmont.realm.SimpleUserDatabaseRealm

SimpleUserDatabaseRealm 类表示了一个复杂一点的 realm，该类并不直接将用户信息编码在方法中，而是读取 tomcat-users.xml 文件，通过文件信息对用户身份进行认证。 代码如下：

SimpleUserDatabaseRealm.java

在初始化了 SimpleUserDatabaseRealm 类后，必须调用其 createDatabase 方法来读取用户信息。

10·7·4 ex10·pyrmont·startup·Bootstrap1 类

代码如下所示：

Bootstrap1.java

Bootstrap1 类的 main 方法创建了两个 SimpleWrapper，分别设置他们的名字和类。然后，main 方法创建一个 StandardContext 对象，设置它的 path 和 docBase 属性，添加一个 SimpleContextConfig 类型的监听器，后者将在 StandardContext 中创建一个 BasicAuthenticator。接下来，会为 StandardContext 添加一个 loader 和两个 servlet 映射。

main 方法创建一个 SecurityCollection 对象，并调用其 addPattern 和 addMethod 方法设置属性。addPattern 方法要对哪个 UML 进行安全检查。addMethod 方法指定了安全检查使用的方法。接下来，main 方法实例化 SecurityConstraint 对象，并将其添加到 collection 中。它还设置哪种角色可以访问受限资源。

接下来，main 方法创建一个 LoginConfig 对象和一个 SimpleRealm 对象，然后将 realm，constraint 和 loginConfig 对象与 StandardContext 相关类。最后，启动 context。

10·7·5 ex10·pyrmont·startup·Bootstrap2 类

Bootstrap2 类用来启动第二个应用程序，与第一个类似，只是使用了 SimpleUserDatabase 类作为 realm。代码如下所示：

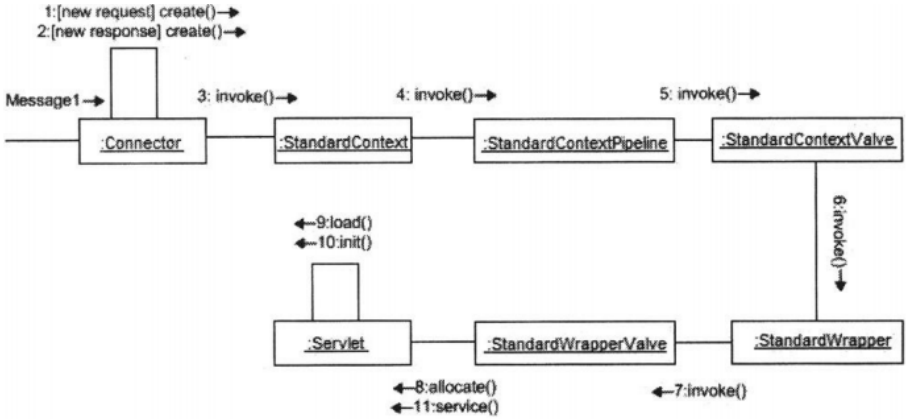
Bootstrap2.java

第 11 章　StandardWrapper

第 5 章中已经说明了 tomcat 中有 4 种类型的 container：engine， host， context， wrapper。在前面的章节中已经说明了如何构建自己的 context 和 wrapper。一般情况下，context 中包含一个或多个 wrapper，每个 wrapper 表示一个 servlet 定义。本章将要对 catalina 中 Wrapper 的标准实现进行说明。

11.1　方法调用序列

对与每个接收到的 http 请求，connector 调用与其关联的 container 的 invoke 方法，然后，container 会调用其子 container 的 invoke 方法。例如，若 connector 与一个 StandardContext 相关联，那么 connector 会调用 StandardContext 实例的 invoke 方法，而 StandardContext 实例会调用其子 container 的 invoke 方法（本例中就是调用 StandardWrapper 的 invoke 方法）。下如说明了处理 http 请求的方法调用序列：



图表 21　处理 HTTP 请求的方法调用序列

具体过程如下：

（1）connector 创建 request 和 response 对象；

（2）connector 调用 StandardContext 实例的 invoke 方法；

（3）StandardContext 接着调用其 pipeline 的 invoke 方法，StandardContext 中 pipeline 的 basic valve 是 StandardContextValve，因此，StandardContext 的 pipeline 会调用 StandardContextValve 的 invoke 方法；

（4）StandardContextValve 的 invoke 方法获取 wrapper 处理请求，调用 wrapper 的 invoke 方法；

（5）StandardWrapper 是 Wrapper 接口的标准实现，StandardWrapper 实例的 invoke 方法会调用其 pipeline 的 invoke 方法；

（6）StandardWrapper 的 pipeline 中的 basic valve 是 StandardWrapperValve，因此，会调用其 invoke

方法，StandardWrapperValve 调用 wrapper 的 allocate 方法获取 servlet 实例；

（7）allocate 方法调用 load 方法载入 servlet 类，若已经载入，则无需重复载入；

（8）load 方法调用 servlet 的 init 方法；

（9）StandardWrapperValve 调用 servlet 的 service 方法。

注意：StandardContext 类的构造方法会设置一个 StandardContextValve 类的实例作为其 basic valve，StandardWrapper 于此类似：

```
publicStandardContext() {


  super();

  pipeline·setBasic(new StandardContextValve());

  namingResources·setContainer(this);

}
```

本章将详细说明，servlet 是如何调用的。

11.2 SingleThreadModel

servlet 类可以实现 javax·servlet·SingleThreadModel 接口，这样的 servlet 也称为 SingleThreadModel (STM)servlet。根据 servlet 规范，实现此接口的目的是保证 servlet 一次只处理一个请求。下面的 servlet2·4 的规范说明（servlet2·3 于此类似）：

If a servlet implements this interface, you are guaranteed that no two threads will execute concurrently in a servlet's service method· The servlet container can guarantee this by synchronizing access to a single instance of the servlet, or by maintaining a pool of servlet instances and dispatching each new request to a free servlet· This interface does not prevent synchronization problems that result from servlets accessing shared resources such as static class variables or classes outside the scope of the servlet·

（翻译：若 servlet 类实现了 SingleThreadModel 接口，则可以保证绝不会有两个线程同时执行某个 servlet 的 service 方法。servlet 容器是通过控制对单一 servlet 实例的同步访问实现，或者维护一个 servlet 对象池，然后将每个新请求分发到一个新的 servlet 实例上。该接口并不能防止 servlet 访问共享资源造成的同步问题，例如访问类的静态变量或访问 servlet 作用域之外的类。）

很多程序员并没有仔细阅读这段话，想当然的认为，实现了该接口的 servlet 就是线程安全的。请再读一遍规范。

事实上，实现了 SingleThreadModel 接口的 servlet 只保证在同一时刻，只有一个线程在执行 servlet 的 service 方法。但是，为了提高效率，servlet 容器会创建多个 STM servlet 实例。也就是说，STM servlet 的 service 方法会在多个 STM servlet 实例中并发执行。这就有可能引起同步问题。

在 servlet2·4 中，SingleThreadModel 已经被标识为 deprecated，因为它会是程序员误以为实现了该接口的 servlet 就是线程安全的。但，servlt2·3 和 servlet2·4 都还对该接口提供了支持。

11.3 StandardWrapper

StandardWrapper 对象的主要任务是载入它所表示的 servlet，并进行实例化。但是，StandardWrapper 类并不调用 servlet 的 service 方法。该任务由 StandardWrapperValve 对象（StandardWrapper 实例中 pipeline 的 basic valve）完成。StandardWrapperValve 对象通过 allocate 方法从 StandardWrapper 中获取 servlet 实例，并调用 servlet 的 service 方法。

当第一次使用到某个 servlet 时，StandardWrapper 会动态的载入该 servlet 类，因此，它必须知道 servlet 类的完整限定名。调用 setServletClass 方法指定 servlet 的类名，调用 setName 方法为该 servlet 指定一个名字。

在创建 servlet 对象时，StandardWrapper 还必须考虑到该 servlet 类是否实现了 SingleThreadModel 接口。对于那些没有实现 SingleThreadModel 接口的 servlet，StandardWrapper 只会载入 servlet 类一次，对每个请求都返回同一个 servlet 的引用。StandardWrapper 实例并不会为该 servlet 创建多个实例，因为它假设程序员会处理好同步问题。而对于一个 STM servlet，事情有些不同。StandardWrapper 必须保证不会有两个线程同时执行该 servlet 的 service 方法。如果，StandardWrapper 通过单例模式使用 STM servlet，下面是实现代码：

```
Servlet instance = <get an instance of the servlet>;

if (<servlet implementing SingleThreadModel>) {

  synchronized (instance) {

    instance·service(request, response);

  }

}
```

```
else {
    instance.service(request, response);
}
```

但是，为了更好的性能，StandardWrapper 会维护一个 STM servlet 的对象池。

11.3.1 生成 servlet

正如本节开头所说，StandardWrapperValve 调用 wrapper 的 allocate 方法获取某个指定的 servlet 对象。StandardWrapper 类要负责实现 allocate 方法。该方法签名如下：

```
public javax.servlet.Servlet allocate() throws ServletException;
```

为了支持 STM servlet，allocate 需要变得复杂一些。事实上，allocate 为了处理 STM servlet 和 non-STM servlet 分为两个部分。第一部分框架如下：

```
if (!singleThreadModel) {
    // returns a non-STM servlet instance
}
```

bool 变量 singleThreadModel 用来表明该 servlet 类是否是 STM servlet，初始值为 false，当调用 loadServlet 方法时，会检查该 servlet 类，并修改 singleThreadModel 的值。

当 singleThreadModel 为 true 时，执行 allocate 方法的第二部分，框架如下：

```
synchronized (instancepool) {
    // returns an instance of the servlet from the pool
}
```

对于 non-STM servlet，StandardWrapper 类定义了 javax.servlet.Servlet 类型的变量，用以存储 servlet

的引用：

```
private Servlet instance = null;
```

allocate 方法会检查变量 instance 是否是 null，若是，则调用 loadServlet 方法载入相关的 servlet 类，然后，将 countAllocated 值加 1，并返回 instance。代码如下：

```
if (!singleThreadModel) {
  // Load and initialize our instance if necessary
  if (instance == null) {
    synchronized (this) {
      if (instance == null) {          try {
          instance = loadServlet();
        }
        catch (ServletException e) {
          throw e;
        }
        catch (Throwable e) {
          throw new ServletException
            (sm.getString("standardWrapper.allocate"), e);
        }
      }
    }
  }
  if (!singleThreadModel) {
    if (debug >= 2)
      log("  Returninq non-STM instance");
```

```
            countAllocated++;
            return (instance);
        }
    }
```

若 StandardWrapper 表示的是一个 STM servlet，则 allocate 会试图从池中返回一个 servlet 实例。变量 instancePool 是一个 java·util·Stack 类型的栈， instancePool 在 loadMethod 方法中初始化，用于存储 STM servlet 实例。

```
private Stack instancePool = null;
```

只要 STM servlet 实例的数量不超过一个指定的树木，allocate 方法就会从栈中返回一个 servlet 实例。变量 maxInstances 指定了栈中元素的最大数量，默认是 20。

```
private int maxInstances = 20;
```

变量 nInstances 指明了当前栈中 servlet 实例的数量：

```
private int nInstances = 0;
```

下面是 allocate 方法第二部分的代码：

```
synchronized (instancePool) {
    while (countAllocated >= nInstances) {
        // Allocate a new instance if possible, or else wait
        if (nInstances < maxInstances) {
            try {
                instancePool·push(loadServlet());
                nInstances++;
            } catch (ServletException e) {
```

```
                throw e;
            } catch (Throwable e) {
                throw new ServletException(sm.getString("StandardWrapper.allocate"), e);
            }
        }else {
            try {
                instancePool.wait();
            } catch (InterruptedException e) {
                ;
            }
        }
    }
    if (debug >= 2)
        log("  Returning allocated STM instance");
    countAllocated++;
    return (Servlet) instancePool.pop();
}
```

当已经分配出去的 servlet 的数量大于等于栈中实例的最大值时，循环等待，直到有新的 servlet 实例可供分配。

11.3.2 载入 servlet

StandardWrapper 类实现了 Wrapper 接口的 load 方法，load 方法调用 loadServlet 方法载入 servlet 类，并调用其 init 方法（此时要传入一个 javax.servlet.ServletConfig 实例作为参数）。

loadServlet 方法首先会检查当前的 StandardWrapper 类是不是一个 STM servlet，若不是，且变量 instance 不为 null（表示以前已经载入过这个类），就直接返回该引用：

```
// Nothing to do if we already have an instance or an instance pool

if (!singleThreadModel && (instance != null))

    return instance;
```

若 instance 为 null，或该 servlet 是一个 STM servlet，则执行后续的方法。

首先，获取 System·out 和 System·err，便于使用 javax·servlet·ServletContext 的 log 方法记录日志：

```
PrintStream out = System·out;

SystemLogHandler·startCapture();
```

然后，定义类型为 javax·servlet·Servlet 名为 servlet 的变量，变量 servlet 表示已载入的 servlet 的实例引用（该引用的值由 loadServlet 方法返回）：

```
Servlet servlet = null;
```

loadServlet 方法负责载入该 servlet 类。类名会写入变量字符串 actualClass 中：

```
String actualClass = servletclass;
```

但，由于 catalina 是 jsp 容器，loadServlet 方法必须检查该 servlet 是不是一个 jsp 页面，若是，loadServlet 要从 jsp 页面中获取实际的 servlet 类：

```
if ((actualClass == null) && (jspFile != null)) {

    Wrapper jspWrapper = (Wrapper)

     ((Context) getParent())·findChild(Constants·JSP_SERVLET_NAME);

     if (jspWrapper != null)

       actualClass = jspWrapper·getServletClass();

}
```

若无法获取 jsp 页面的 servlet 名，则使用变量 actualClass 的值。但是，若没有调用 StandardWrapper 类的 setServletClass 方法设置 actualClass 的值，则抛出异常，且后面的方法也不再执行。

```
// Complain if no servlet class has been specified
if (actualClass == null) {
  unavailable(null);
  throw new ServletException
    (sm.getString("StandardWrapper.notClass", getName()));
}
```

这时，servlet 类名已经解析完成，loadServlet 方法会获取 loader，若找不到 loader，则抛出异常，不再执行后续方法：

```
// Acquire an instance of the class loader to be used
Loader loader = getLoader();
if (loader == null) {
  unavailable(null);
  throw new ServletException
  (sm.getString("StandardWrapper.missingLoader", getName()));
}
```

若 loader 不为 null，则调用 loader 的 getClassLoader 方法获取 ClassLoader。

在 org.apache.catalina 包下，catalina 提供了一些用于访问 servlet 容器内部成员的专用 servlet。若 servlet 是专用 servlet，例如，若 isContainerProvidedServlet 返回 true，会将变量 classLoader 赋值为另一个 ClassLoader 实例，这样就可以访问 catalina 内部：

```
// Special case class loader for a container provided servlet
if (isContainerProvidedServlet(actualClass)) {
  ClassLoader = this.getClass().getClassLoader();
  log(sm.getString ("standardWrapper.containerServlet", getName()));
}
```

有了 loader 和 servlet 类名后，loadServlet 方法就可以载入 servlet 类了：

```
// Load the specified servlet class from the appropriate class loader
Class classClass = null;
try {
  if (ClassLoader != null) {
      System.out.println("Using classLoader.loadClass");
      classClass = classLoader.loadClass(actualClass);
  } else {
      System.out.println("Using forName");
      classClass = Class.forName(actualClass);
  }
} catch (ClassNotFoundException e) {
  unavailable(null);
  throw new ServletException (sm.getstring("standardWrapper.missingClass", actualClass), e);
}
if (classClass == null) {
  unavailable(null);
  throw new ServletException (sm.getString("standardWrapper.missingClass", actualClass));
}
```

然后，实例化 servlet：

```
// Instantiate and initialize an instance of the servlet class itself
try {
```

```
    servlet = (Servlet) classClass.newInstance();
} catch (ClassCastException e) {
    unavailable(null);
    // Restore the context ClassLoader
    throw new ServletException (sm.getString("standardWrapper.notServlet", actualClass), e);
} catch (Throwable e) {
    unavailable(null);
    // Restore the context ClassLoader
    throw new ServletException (sm.getstring("standardWrapper.instantiate", actualClass), e);
}
```

在 loadMethod 实例化 servlet 之前，会调用 isServletAllowed 方法检查该 servlet 是否被允许使用：

```
// Check if loading the servlet in this web application should be allowed
if (!isServletAllowed(servlet)) {
    throw new SecurityException
        (sm.getString("standardWrapper.privilegedServlet",
            actualClass));
}
```

若通过了安全检查，会继续检查该 servlet 是否是一个 ContainerServlet 类型的 servlet。实现了 ContainerServlet 接口的 servlet 可以访问 catalina 的内部功能。若该 servlet 是一个 ContainerServlet 的实现类，loadServlet 会调用 ContainerServlet 接口的 setWrapper 方法，传入 StandardWrapper 实例：

```
// Special handling for ContainerServlet instances
if ((servlet instanceof ContainerServlet) &&
    isContainerProvidedServlet(actualClass)) {
        ((ContainerServlet) servlet).setWrapper(this);
```

```
}
```

接下来，loadServlet 方法触发 BEFORE_INIT_EVENT 事件，调用 init 方法：

```
try {
    instanceSupport.fireInstanceEvent(
      InstanceEvent.BEFORE_INIT_EVENT, servlet);
    servlet.init(facade);
```

注意，init 方法传入了一个 javax.servlet.ServletConfig 对象的外观类 façade。

若变量 loadOnStartup 大于 0，而且 servlet 实际上是一个 jsp 页面，调用 servlet 的 service 方法：

```
// Invoke jspInit on JSP pages
if ((loadOnStartup > 0) && (jspFile != null)) {
    // Invoking jspInit
    HttpRequestBase req = new HttpRequestBase();
    HttpResponseBase res = new HttpResponseBase();
    req.setServletPath(jspFile);
    req.setQueryString("jsp_precompile=true");
    servlet.service(req, res);
}
```

接下来，loadServlet 方法触发 AFTER_INIT_EVENT 事件：

```
instanceSupport.fireInstanceEvent (InstanceEvent.AFTER_INIT_EVENT, servlet);
```

若该 servlet 是一个 STM servlet，则将该 servlet 实例添加到 servlet 对象池中。因此会先判断变量 instancePool 是否为 null，若是，为 instancePool 赋一个 Stack 对象：

```
// Register our newly initialized instance
    singleThreadModel = servlet instanceof SingleThreadModel;
    if (singleThreadModel) {
      if (instancePool == null)
        instancePool = new Stack();
    }
    fireContainerEvent("load", this);
  }
```

在 finally 代码块中，loadServlet 方法停止对 System·out 和 System·err 的使用，记录在 servlet 处理过程中产生的日志：

```
  finally {
    String log = SystemLogHandler·stopCapture();
    if (log != null && log·length() > 0) {
      if (getServletContext() != null) {
        getServletContext()·log(log);
      }
      else {
        out·println(log);
      }
    }
  }
```

最后，返回 servlet 实例：

```
return servlet;
```

11.3.3 ServletConfig 对象

StandardWrapper 类的 loadServlet 方法在载入 servlet 后会调用 init 方法。init 方法需要传入一个 javax.servlet.ServletConfig 实例作为参数。StandardWrapper 类实现了 javax.servlet.ServletConfig 接口和 Wrapper 接口。在 ServletConfig 接口有以下几个方法：getServletContext，getServletName，getInitParameter 和 getInitParameterNames。接下来分别说明这几个方法的实现。

注意，StandardWrapper 类并不将自身实例传给 init 方法，而是将自身的包装类 StandardWrapperFacade 的实例传入，这样就很好的将一些 public 方法隐藏起来了，防止被 servlet 程序员调用。

11.3.3.1 getServletContext 方法

该方法的签名如下：

```
public ServletContext getServletContext();
```

StandardWrapper 实例肯定是 StandardContext 实例的子 container，也就是说，StandardWrapper 的父 container 是 StandardContext。可以调用 StandardContext 对象的 getServletContext 方法来获得一个 ServletContext 实例。下面是 getServletContext 方法的实现：

```
public ServletContext getServletContext() {
    if (parent == null)
      return (null);
    else if (!(parent instanceof Context))  //必须是 Context 容器
      return (null);
    else
      return (((Context) parent).getServletContext());
}
```

11.3.3.2 getServletName 方法

该方法返回 servlet 的名字，方法签名如下：

```
public java·lang·String getServletName() ;
```

下面是 StandardWrapper 中 getServletName 方法实现：

```
public String getServletName() {
    return (getName());
}
```

在该方法中，仅仅是简单的调用了 ContainerBase 类（StandardWrapper 的父类）的 getName，ContainerBase 类中 getName 方法的实现如下：

```
public String getName() {
    return (name);
}
```

可以通过 setName 方法设置变量 name 的值。

11.3.3.3 getInitParameter 方法

该方法返回指定初始参数的值。方法签名如下：

```
public java·lang·String getInitParameter(java·lang·String name);
```

在 StandardWrapper 类中，初始化参数存储在一个 HashMap 类型的名为 parameters 的变量中：

```
private HashMap parameters = new HashMap();
```

调用 StandardWrapper 类的 addInitParameter 方法来填充变量 parameters 的值：

```
public void addInitParameter(String name, String value) {
```

```
    synchronized (parameters) {
      parameters.put(name, value);
    }
    fireContainerEvent("addInitParameter", name);
}
```

下面是 StandardWrapper 类中 getInitParameter 方法的实现：

```
public String getInitParameter(String name) {
    return (findInitParameter(name));
}
```

其中 findInitParameter 方法接受一个指明初始化参数名的字符串变量，调用变量 parameters 的 get 方法获取初始化参数的值：

```
public String findInitParameter(String name) {

    synchronized (parameters) {
      return ((String) parameters.get(name));
    }
}
```

11.3.3.4 getInitParameterNames 方法

该方法返回所有初始化参数的名字的集合（实际上是一个枚举类型 java.util.Enumeration），下面是方法签名和实现：

```
public java.util.Enumeration getInitParameterNames();
```

```
public Enumeration getInitParameterNames() {
    synchronized (parameters) {
        return (new Enumerator(parameters.keyset()));
    }
}
```

11.3.4 container 的父子关系

wrapper 代表了一个 servlet，wrapper 中不能再有子容器，不应该再调用其 addChild，否则抛出 java.lang.IllegalStateException 异常。下面是 StandardWrapper 类中 addChild 方法的实现：

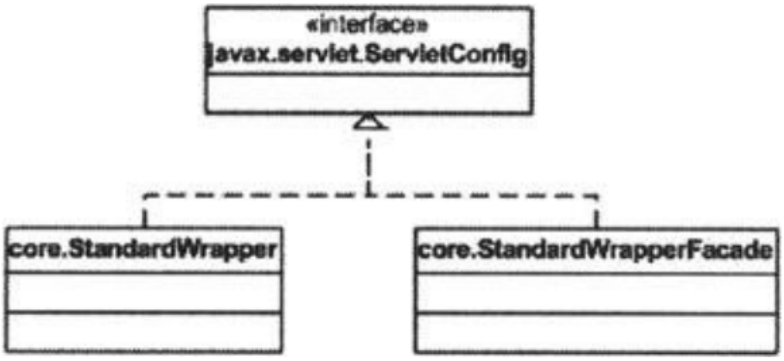
```
public void addChild(Container child) {
    throw new IllegalStateException
        (sm.getString("StandardWrapper.notChild"));
}
```

wrapper 的父容器只能是 context，若是在调用 wrapper 的 setParent 方法时传入了一个非 context 的容器，则会抛出 java.lang.IllegalArgumentException 异常：

```
public void setParent(Container container) {
    if ((container != null) && !(container instanceof Context))
        throw new IllegalArgumentException
            (sm.getString("standardWrapper.notContext"));
    super.setParent(container);
}
```

11.4 StandardWrapperFacade 类

StandardWrapper 实例会调用它所载入的 servlet 的 init 方法，init 方法需要一个 javax·servlet·ServletConfig 实例，StandardWrapper 本身实现了 javax·servlet·ServletConfig 接口，所以，StandardWrapper 可以将自己的实例传入。但是，StandardWrapper 需要将其大部分 public 方法对 servlet 隐藏起来，因此，传入了 StandardWrapper 的外观类 StandardWrapperFacade。下如是相关类的 UML 图：



图表 22　StandardWrapper 类及其相关类的 UML 示意图

下面的代码创建了一个 StandardWrapperFacade 实例，传入的是 StandardWrapper 的实例：

```
private StandardWrapperFacade facade = new StandardWrapperFacade(this);
```

StandardWrapperFacade 类提供了 ServletConfig 类型的变量：

```
private ServletConfig config = null;
```

当在 StandardWrapper 实例内创建 StandardWrapperFacade 实例时，传入 StandardWrapper 实例，并为变量 config 赋值：

```
public StandardWrapperFacade(StandardWrapper config) {
    super();
    this·config = (ServletConfig) config;
}
```

因此，当创建 StandardWrapper 实例调用 servlet 实例的 init 方法时，它会传入一个 StandardWrapperFacade 实例。这样，在 servlet 内调用 ServletConfig 的 getServletName，getInitParameter 和 getInitParameterNames 方法会转换为调用 StandardWrapper 实例的相应方法，还可以对 servlet 隐藏一些不必要的方法：

```
public String getServletName() {
    return config.getServletName();
}
public String getInitParameter(String name) {
    return config.getInitParameter(name);
}
public Enumeration getInitParameterNames() {
    return config.getInitParameterNames();
}
public ServletContext getServletContext() {
    ServletContext theContext = config.getServletContext();
    if ((theContext != null) && (theContext instanceof ApplicationContext))
      theContext = ((ApplicationContext) theContext).getFacade();
    return (theContext);
}
```

11.5 StandardWrapperValve 类

StandardWrapperValve 类是 StandardWrapper 实例的 basic valve，要做两件事：

（1）执行与该 servlet 关联的全部过滤器 ；

（2）调用 servlet 的 service 方法。

为完成上述任务，在 StandardWrapperValve 的 invoke 方法中会做以下几件事：

（1）获取 servlet 实例；

（2）调用私有方法 createFilterChain，创建过滤器链；

（3）调用过滤器链的 doFilter 方法，包括调用 servlet 的 service 方法；

（4）释放过滤器链；

（5）调用 wrapper 的 deallocate 方法；

（6）若该 servlet 再也不会被使用到，调用 wrapper 的 unload 方法。

下面是 invoke 方法的部分代码：

```
// Allocate a servlet instance to process this request
try {
  if (!unavailable) {
    servlet = wrapper.allocate();
  }
}
...
// Acknowlege the request
try {
  response.sendAcknowledgement();
}
...
// Create the filter chain for this request
ApplicationFilterChain filterChain = createFilterChain(request, servlet);
// Call the filter chain for this request
// This also calls the servlet's servicet() method
try {
```

```
    String jspFile = wrapper.getJspFile();
    if (jspFile != null)
      sreq.setAttribute(Globals.JSP_FILE_ATTR, jspFile);
    else
      sreq.removeAttribute(Globals.JSP_FILE_ATTR);
    if ((servlet != null) && (filterChain != null)) {
      filterChain.doFilter(sreq, sres);
    }
    sreq.removeAttribute(Globals.JSP_FILE_ATTR);
  }    ...
  // Release the filter chain (if any) for this request
  try {
    if (filterChain != null)
      filterChain.release();
  }
  ...
  // Deallocate the allocated servlet instance
  try {
    if (servlet != null) {
      wrapper.deallocate(servlet);
    }
  }
  ...
  // If this servlet has been marked permanently unavailable,
```

```
// unload it and release this instance
try {
  if ((servlet != null) && (wrapper.getAvailable() ==
    Long.MAX_VALUE)) {
    wrapper.unload();
  }

}
...
```

其中最重要的是 createFilterChain 方法和 doFilter 方法的调用。createFilterChain 方法创建了 ApplicationFilterChain 实例，并将添加了与 servlet 相关联的全部 filter。

11.6 FilterDef 类

org.apache.catalina.deploy.FilterDef 类表示了一个 filter 定义，该类的定义如下：

FilterDef.java

FilterDef 类中每个属性表示了在定义 filter 元素时声明的子元素。其中 Map 类型的变量 parameters 存储了所有的初始化参数。addInitParameter 方法用于向 parameters 中添加新的 key/value。

11.7 ApplicationFilterConfig 类

org.apache.catalina.core.ApplicationFilterConfig 类实现了 javax.servlet.FilterConfig 接口。

ApplicationFilterConfig 类用于管理 web 应用中的 filter。通过传入一个 org·apache·catalina·Context 实例和一个 FilterDef 实例创建一个 ApplicationFilterConfig 实例：

```
public ApplicationFilterConfig(Context context, FilterDef filterDef)
    throws ClassCastException, ClassNotFoundException,
      IllegalAccessException, InstantiationException, ServletException
```

Context 对象表示了一个 web 应用，FilterDef 对象表示一个 filter，ApplicationFilterConfig 类的 getFilter 方法会返回 javax·servlet·Filter 实例。该方法负责载入并实例化一个 filter：

```
Filter getFilter() throws ClassCastException, ClassNotFoundException,
    IllegalAccessException, InstantiationException, ServletException {

    // Return the existing filter instance, if any
    if (this·filter != null)
      return (this·filter);
     // Identify the class loader we will be using

    String filterClass = filterDef·getFilterClass();
    ClassLoader classLoader = null;
    if (filterClass·startsWith("org·apache·catalina·"))
      classLoader = this·getClass()·getClassLoader();
    else
      classLoader = context·getLoader()·getClassLoader();

    ClassLoader oldCtxClassLoader =
      Thread·currentthread()·getContextClassLoader();
```

```
    // Instantiate a new instance of this filter and return it
    Class clazz = classLoader.loadClass(filterClass);
    this.filter = (Filter) clazz.newInstance();
    filter.init(this);
    return (this.filter);
}
```

11.8 ApplicationFilterChain 类

org.apache.catalina.core.ApplicationFilterChain 类实现了 javax.servlet.FilterChain 接口，StandardWrapperValve 对象的 invoke 方法会创建 ApplicationFilterChain 对象，并调用其 doFilter 方法。ApplicationFilterChain 类的 doFilter 方法会调用过滤器链中第一个过滤器的 doFilter 方法。Filter 的 doFilter 方法的签名是：

```
public void doFilter(ServletRaquest request, ServletResponse response,
    FilterChain chain) throws java.io.IOException, ServletException
```

ApplicationFilterChain 类的 doFIlter 方法会传入 ApplicationFilterChain 自身对象作为第三个参数，filter 可以显式调用过滤器链的 doFilter 方法来调用后续 filter 的 doFilter 方法。示例代码如下：

```
public void doFilter(ServletRequest request, ServletResponse response,
    FilterChain chain) throws IOException, ServletException {
    // do something here
    ...
    chain.doFilter(request, response);
}
```

11.9 应用程序

本章的应用程序包含两个类：ex*11*·pyrmont·core·SimpleContextConfig 和 ex*11*·pyrmont·startup·Bootstrap，其中 SimpleContextConfig 类与之前章节相同，Bootstrap 类的实现代码如下：

Bootstrap.java

第 12 章　StandardContext 类

12.1　概述

正如前面章节所述，context 表示了一个 web 应用，其中包含一个或多个 wrapper，每个 wrapper 表示一个 servlet 定义。context 还需要其他组件的支持，典型的如 loader 和 manager。本章将说明 StandardContext 是如何工作的。

12.2　StandardContext 的配置

创建了 StandardContext 实例后，必须调用调用其 start 方法来准备接收 http 请求。但是，在调用 start 方法时，可能会报错，这时 StandardContext 对象的 available 属性会被置为 false，available 属性表明了 StandardContext 对象是否可用。

若要是 start 方法正确执行，必须正确对 StandardContext 对象正确配置。tomcat 中，配置 StandardContext 对象需要一系列操作。正确设置后，StandardContext 对象才能读取并解析 web·xml 文件。

StandardContext 类的 configured 属性表明 StandardContext 对象是否被正确设置。StandardContext 使用一个事件监听器作为配置器（configurator）。当调用 StandardContext 的 start 方法时，其中要做的一件事是，触发一个生命周期的事件。若设置成功，监听器会将 configured 属性设置为 true。否则，StandardContext 实例会拒绝启动，也就无法接收 http 请求。

在第 11 章中，你已经看到，生周期监听器会被添加到 StandardContext 实例中。它的类型是 ch11·pyrmont·core·SimpleContextConfig，它仅仅是将 StandardContext 实例的 configured 属性设置为 true，不做其他的事。在 tomcat 的实际应用中，对 StandardContext 实例进行配置的监听器的类型是 org·apache·catalina·startup·ContextConfig。

12·2·1　StandardContext 类的构造函数

下面是 StandardContext 类的构造函数的实现：

```
public StandardContext() {

    super();

    pipeline·setBasic(new StandardContextValve());
```

```
        namingResources.setContainer(this);
    }
```

构造函数中最重要的事情是为 pipeline 设置 basic valve，org.apache.catalina.core.StandardContextValve。该 valve 会处理从 connector 中接收到的 http 请求。

12.2.2 启动 StandardContext

start 方法初始化 StandardContext 对象，用监听器配置 StandardContext 实例。配置成功后，监听器会将 configured 属性置为 true。最后，start 方法会将 available 属性，true 表明 StandardContext 对象设置正确，与其相关联的子容器和组件也都设置正确，并正确启动，于是，StandardContext 实例可以准备接收 http 请求了，期间若发生了错误，则 available 的值会被置为 false。

StandardContext 使用一个 bool 型标志位变量 configured（初始化为 false），来表明 StandardContext 对象是否被正确设置。在 start 方法结尾，StandardContext 会检查 configured 变量的值，若 configured 为 true，则 StandardContext 启动成功，否则，调用 stop 方法，关闭所有在在 start 方法已经启动的组件。start 方法的实现代码如下：

StandardContext.java

注意，在 tomcat5 中，start 方法虽然与 tomcat4 中类似，但包含了一些与 jmx 相关的代码（但如果，你不使用 jmx 的话，这些代码是没有作用的）。

下面是 start 方法要做的一些事：

（1）触发 BEFORE_START 事件；

（2）将 availability 和 configured 属性设置为 false；

（3）设置 resources，loader，manager；

（4）初始化字符串映射；

（5）启动与该 context 关联的组件；

（6）启动子 container；

（7）启动 pipeline；

（8）启动 manager；

（9）触发 START 事件，这里监听器（ContextConfig）会执行一些配置操作，若设置成功，ContextConfig 会将 StandardContext 实例的 configured 变量设置为 true；

（10）检查 configured 属性的值，若为 true，调用 postWelcomePages 方法，载入子 wrapper，将 availability 属性设置为 true。若 configured 为 false，则调用 stop 方法；

（11）触发 AFTER_START 事件。

12.2.3 invoke 方法

在 tomcat4 中，StandardContext 的 invoke 方法由相关联的 connector 调用，或者当 StandardContext 是一个子 container 时，由 host 的 invoke 方法调用。StandardContext 的 invoke 方法第一次调用时会检查应用程序是否正在重载过程中，若是，则等待应用程序重载完成。然后，调用其父类（ContainerBase）的 invoke 方法。

```
public void invoke(Request request, Response response) throws IOException, ServletException {
  // Wait if we are reloading
  while (getPaused()) {
    try {
      Thread.sleep(1000);
    }
    catch (InterruptedException e) {
      ;
    }
  }  // Normal request processing
  if (swallowOutput) {
    try {
      SystemLogHandler.startCapture();
```

```
            super.invoke(request, response);
        }
        finally {
            String log = SystemLogHandler.stopCapture();
            if (log != null && log.length() > 0) {
                log(log);
            }
        }
    }
    else {
        super.invoke(request, response);
    }
}
```

其中，getPaused 方法返回 paused 属性的值，当 paused 值为 true 时，表明应用程序正在重载。

在 tomcat5 中，StandardContext 类并没有提供 invoke 方法的实现，因此会调用 ContainerBase 的 invoke 方法，检查应用程序是否在重载的工作被移到了 StandardContextValve 类的 invoke 方法中。

12.3 StandardContextMapper 类

对于每个接收到的 http 请求，都会调用 StandardContext 的 pipeline 的 basic valve。StandardContext 的 basic valve 是 org.apache.catalina.core.StandardContextValve 类实现的。StandardContextValve.invoke 方法要做的第一件事是获取一个 wrapper 对象处理请求。

在 tomcat4 中，StandardContextValve 实例使用 context 的 mapper 找到一个合适的 wrapper 使用。一一旦获得了 wrapper，就会调用 wrapper 的 invoke 方法。下面简要介绍下 mapper 组件。

ContainerBase 类（StandardContext 的父类）定义了 addDefaultMapper 方法添加一个默认的 mapper：

```
protected void addDefaultMapper(String mapperClass) {
    if (mapperClass == null)
        return;
    if (mappers.size() >= 1)
        return;
    // Instantiate and add a default Mapper
    try {
        Class clazz = Class.forName(mapperClass);
        Mapper mapper = (Mapper) clazz.newInstance();
        mapper.setProtocol("http");
        addMapper(mapper);
    }
    catch (Exception e) {
        log(sm.getString("containerBase.addDefaultMapper", mapperClass), e);
    }
}
```

StandardContext 实例在其 start 方法中调用 addDefaultMapper 方法，并传入一个 mapperClass 变量的值：

```
public synchronized void start() throws LifecycleException {
    …
    if (ok) {
        try {
            addDefaultMapper(this.mapperClass);
    …
```

```
}
```

StandardContext 中定义的 mapperClass 变量如下：

```
private String mapperClass = "org.apache.catalina.core.StandardContextMapper";
```

你必须要调用 mapper 的 setContainer 方法将 mapper 和 container 相关联。tomcat 中 org.apache.catalina.Mapper 接口的标准实现类是 org.apache.catalina.core.StandardContextMapper。StandardContextMapper 实例仅仅能与 context 相关联，如代码所示：

```
public void setContainer(Container container) {
    if (!(container instanceof StandardContext))
        throw new IllegalArgumentException (sm.getstring("httpContextMapper.container"));
    context = (StandardContext) container;
}
```

mapper 中最重要的方法是 map 方法，该方法会返回一个子 container 来处理 http 请求，该方法的签名如下：

```
public Container map(Request request, boolean update);
```

在 StandardContextMapper 中，map 方法返回一个 wrapper 处理请求，若是找不到 wrapper，方法返回 null。

现在回到本节开始处的讨论，对每个 http 请求，StandardContextValve 实例调用 context 的 map 方法，传入一个 org.apache.catalina.Request 对象。map 方法（实际上定义在父类 ContainerBase 中）会针对某个特定的协议调用 findMapper 方法返回一个 mapper 对象，然后调用 mapper 对象的 map 方法：

```
// Select the Mapper we will use
Mapper mapper = findMapper(request.getRequest().getProtocol());
if (mapper == null)
    return (null);
// Use this Mapper to perform this mapping
```

```
return (mapper.map(request, update));
```

StandardContextMapper 类的 map 方法会先获取要匹配的 uri：

```
String contextPath = ((HttpServletRequest) request.getRequest()).getContextPath();
String requestURI = ((HttpRequest) request).getDecodedRequestURI();
String relativeURI = requestURI.substring(contextPath.length());
```

然后，试图应用匹配规则找到一个 wrapper：

```
// Apply the standard request URI mapping rules from the specification
Wrapper wrapper = null;
String servletPath = relativeURI;
String pathInfo = null;
String name = null;
// Rule 1 -- Exact Match
if (wrapper == null) {
    if (debug >= 2)
      context.log(" Trying exact match");
    if (!(relativeURI.equals("/")))
      name = context.findServletMapping(relativeURI);
    if (name != null)
      wrapper = (Wrapper) context.findChild(name);
    if (wrapper != null) {
      servletPath = relativeURI;
      pathInfo = null;
```

```
    }
}

// Rule 2 -- Prefix Match
if (wrapper == null) {
    if (debug >= 2)
        context.log(" Trying prefix match");
    servletPath = relativeURI;
    while (true) {
        name = context.findServletMapping(servletPath + "/*");
        if (name != null)
            wrapper = (Wrapper) context.findChild(name);
        if (wrapper != null) {
            pathInfo = relativeURI.substring(servletPath.length());
            if (pathInfo.length() == 0)
                pathInfo = null;
            break;
        }
        int slash = servletPath.lastIndexOf('/');
        if (slash < 0)
            break;
        servletPath = servletPath.substring(0, slash);
    }
```

```
}


// Rule 3 -- Extension Match
if (wrapper == null) {
    if (debug >= 2)
        context.log(" Trying extension match");
    int slash = relativeURI.lastIndexOf('/');
    if (slash >= 0) {
        String last = relativeURI.substring(slash);
        int period = last.lastIndexOf('.');
        if (period >= 0) {
            String pattern = "*" + last.substring(period);
            name = context.findServletMapping(pattern);
            if (name != null)
                wrapper = (Wrapper) context.findChild(name);
            if (wrapper != null) {
                servletPath = relativeURI;
                pathInfo = null;
            }
        }
    }
}
```

```
// Rule 4 -- Default Match
if (wrapper == null) {
    if (debug >= 2)
        context.log(" Trying default match");
    name = context.findServletMapping("/");
    if (name != null)
        wrapper = (Wrapper) context.findChild(name);
    if (wrapper != null) {
        servletPath = relativeURI;
        pathInfo = null;
    }
}
```

也许你会问，context 是如何得到这些信息用来映射 servlet 的呢？下面的代码在 StandardContext 添加了两个 servlet 映射，和两个 wrapper：

```
context.addServletMapping("/Primitive", "Primitive");
context.addServletMapping("/Modern", "Modern");
context.addChild(wrapper1);
context.addChild(wrapper2);
```

在 tomcat5 中，删除 Mapper 接口及其相关类，而是从 request 对象的 getWrapper 方法中获取 wrapper 对象。

12.4 对重载的支持

StandardContext 类中定义了 reloadable 属性来指明该应用是否启用的重载功能。当启用了重载后，当 web·xml 或 WEB-INF/classes 目录下文件被重新编译后，应用程序会重载。

StandardContext 类是通过其 loader 实现应用程序重载的。在 tomcat4 中，StandardContext 对象中的 WebappLoader 类实现了 Loader 接口，并使用另一个线程检查 WEB-INF 目录中的类和 jar 的时间戳。只需要调用 setContainer 方法将 WebappLoader 对象与 StandardContext 对象相关联就可以启动该检查线程。下面是 WebappLoader 类的 setContainer 的实现代码：

```
public void setContainer(Container container) {

    // Deregister from the old Container (if any)

    if ((this·container != null) && (this·container instanceof Context))

      ((Context) this·container)·removePropertyChangeListener(this);

    // Process this property change

    Container oldContainer = this·container;

    this·container = container;

    support·firePropertyChange("container", oldContainer, this·container);

    // Register with the new Container (if any)

    if ((this·container!=null) && (this·container instanceof Context)) {

      setReloadable( ((Context) this·container)·getReloadable() );

      ((Context) this·container)·addPropertyChangeListener(this);

    }

}
```

注意这段代码的末尾，若 container 是一个 context，会调用 setReloadable 方法，也就是说，WebappLoader 实例的 reloadable 属性和 StandardContext 实例的 reloadable 实例是一样的。下面是 WebappLoader 类的 setReloadable 方法的实现：

```
public void setReloadable(boolean reloadable) {
```

```
        // Process this property change
        boolean oldReloadable = this.reloadable;
        this.reloadable = reloadable;
        support.firePropertyChange("reloadable", new Boolean(oldReloadable), new
    Boolean(this.reloadable));
        // Start or stop our background thread if required
        if (!started)
            return;
        if (!oldReloadable && this.reloadable)
            threadStart();
        else if (oldReloadable && !this.reloadable)
            threadStop();
    }
```

若是 reloadable 属性从 false 被修改为 true，会调用 threadStart 方法；若是从 true 修改为 false，调用 threadStop 方法。threadStart 方法会开启一个专用的线程来不断的检查 WEB-INF 目录下的类和 jar 的时间戳，而 threadStop 方法会终止该线程。

在 tomcat5 中，检查类的时间戳的工作改为由 backgroundProcess 方法执行。

12.5 backgroundProcess 方法

context 的运行需要其他组件的支持，例如 loader 和 manager。通常来说，这些组件需要使用各自的线程执行一些后台处理任务。例如，为了支持自动重载，loader 组件需要使用一个线程周期性的检查类和 jar 的时间戳; manager 组件使用一个线程检查它所管理的 session 对象的过期时间。在 tomat4 中，这些组件都开启了各自的

线程完成工作。

在 tomcat5 中，使用了不同的方法。所有的后台处理共享同一的线程。若某个组件或 container 需要周期性的执行一个操作，只需要将代码写到 backgroundProcess 方法中即可。

这个共享线程由 ContainerBase 对象创建，并在其 start 方法中调用 threadStart 方法：

```
protected void threadStart() {

    if (thread != null)

      return;

    if (backgroundProcessorDelay <= 0)

      return;

    threadDone = false;

    String threadName = "ContainerBackgroundProcessor[" + toString() + "]";

    thread = new Thread(new ContainerBackgroundProcessor(), threadName);

    thread.setDaemon(true);

    thread.start();

}
```

threadStart 方法通过传入一个实现了 java.lang.Runnable 接口的 ContainerBackgroundProcessor 实例构造了一个新线程。ContainerBackgroundProcessor 类定义如下：

ContainerBackgroundProcessor.java

ContainerBackgroundProcessor 类实际上是 ContainerBase 类的内部类。在其 run 方法中是一个 while 循环，周期性的调用 processChildren 方法。而 processChildren 方法会调用自身对象的 backgroundProcess，和子 container 的 processChildren 方法。通过实现 backgroundProcess 方法，ContainerBase 的子类可以使用一个专用线程来执行周期性任务，代码如下：

```
public void backgroundProcess() {

    if (!started)

      return;
```

```
count = (count + 1) % managerChecksFrequency;
if ((getManager() != null) && (count == 0)) {
  try {
    getManager()·backgroundProcess();
  }
  catch ( Exception x ) {
    log·warn("Unable to perform background process on manager", x);
  }
}
if (getloader() != null) {
  if (reloadable && (getLoader()·modified())) {
    try {
      Thread·currentThread()·setContextClassloader
        (StandardContext·class·getClassLoader());
      reload();          }
    finally {
      if (getLoader() != null) {
        Thread·currentThread()·setContextClassLoader
          (getLoader()·getClassLoader());
      }
    }
  }
  if (getLoader() instanceof WebappLoader) {
```

```
            ((WebappLoader) getLoader()).closeJARs(false);
        }
    }
}
```

第 13 章　Host 和 Engine

13.1　概述

本章重点介绍两个 container，host 和 engine。理论上，当你只有一个 context 时，不需要使用 host。在 org·apache·catalina·Context 接口的说明中有如下一段话：

The parent Container attached to a Context is generally a Host, but may be some other implementation, or may be omitted if it is not necessary·

（翻译：Context 的父 container 通常是 host，也有可能是其他实现，或者不使用父 container。）

但是在实际应用中，tomcat 总是会使用一个 host。

engine 表示了整个 catalina 的 servlet 引擎。engine 是处于最顶层的 container。被添加到 engine 的子 container 通常是 org·apache·catalina·Host 或 org·apache·catalina·Context。tomcat 默认是使用 engine 的。

本章会介绍与 Host 和 Engine 接口相关的类。首先介绍 Host 接口及其相关实现类 StandardHost，StandardHostMapper 和 StandardHostValve。接下来使用一个应用程序来说明如何使用 host。然后介绍 Engine 接口与其相关实现类 StandardEngine 和 StandardEngineValve，并使用另一个程序进行使用说明。

13.2　Host 接口

Host 接口的定义如下：

Host.java

其中比较重要的是 map 方法，该方法返回一个 context 处理接收到的 http 请求。

13.3　StandardHost 类

catalina 中 org·apache·catalina·core·StandardHost 类是 Host 接口的标准实现。该类继承自 the

org·apache·catalina·core·ContainerBase 类，实现了 Host 和 Deployer 接口。

与 StandardContext 和 StandardWrapper 类似， StandardHost 的构造函数会将 basic valve（org·apache·catalina·core·StandardHostValve）添加到其 pipeline 中：

```
public StandardHost() {
    super();
    pipeline·setBasic(new StandardHostValve());
}
```

当调用其 start 方法（tomcat 启动）时， StandardHost 会添加两个 valve： ErrorReportValve 和 ErrorDispatcherValve（均位于 org·apache·catalina·valves 包下）。tomcat4 中。StandardHost 的 start 方法如下：

```
public synchronized void start() throws LifecycleException {
    // Set error report valve
    if ((errorReportValveClass != null) && (!errorReportValveClass·equals(""))) {
        try {
            Valve valve = (Valve) Class·forName(errorReportValveClass)·newInstance();
            addValve(valve);
        } catch (Throwable t) {
            log(sm·getString ("StandardHost·invalidErrorReportValveClass",
errorReportValveClass));
        }
    }
    // Set dispatcher valve
    addValve(new ErrorDispatcherValve());
    super·start();
```

```
}
```

变量 errorReportValveClass 的值定义在 StandardHost 类中：

```
private String errorReportValveClass = "org.apache.catalina.valves.ErrorReportValve";
```

每当接收到一个 http 请求，会调用 host 的 invoke 方法。在 StandardHost 类并没有提供 invoke 方法的实现，因此，它会调用其父类 ContainerBase 的 invoke 方法。而会 ContainerBase 的 invoke 方法调用 StandardHostValve 的 invoke 方法。此外，StandardHostValve 的 invoke 方法会调用 StandardHost 的 map 方法来获取一个 context 处理 http 请求。StandardHost 中 map 方法的实现如下：

```
public Context map(String uri) {
    if (debug > 0)
        log("Mapping request URI '" + uri + "'");
    if (uri == null)
        return (null);
    // Match on the longest possible context path prefix
    if (debug > 1)
        log("  Trying the longest context path prefix");
    Context context = null;
    String mapuri = uri;
    while (true) {
        context = (Context) findChild(mapuri);
        if (context != null)
            break;
        int slash = mapuri.lastIndexOf('/');
        if (slash < 0)
            break;
```

```
            mapuri = mapuri.substring(0, slash);
        }
        // If no Context matches, select the default Context
        if (context == null) {
            if (debug > 1)
                log("  Trying the default context");
            context = (Context) findChild("");
        }
        // Complain if no Context has been selected
        if (context == null) {
            log(sm.getString("standardHost.mappingError", uri));
            return (null);
        }
        // Return the mapped Context (if any)
        if (debug > 0)
            log(" Mapped to context " + context.getPath() + "");
        return (context);
    }
```

注意，tomcat4 中，ContainerBase 类中也定义了一个 map 方法，方法签名如下：

```
public Container map(Request request, boolean update);
```

tomcat5 中，mapper 组件已经被去掉，context 是通过 request 对象获取的。

13.4 StandardHostMapper 类

在 tomcat4 中，ContainerBase 类（StandardHost 的父类）会调用 addDefaultMapper 方法创建一个默认 mapper，使用 mapperClass 属性加以引用。下面是 ContainerBase 类的 addDefaultMapper 方法的实现：

```
protected void addDefaultMapper(String mapperClass) {
    // Do we need a default Mapper?
    if (mapperClass == null)
      return;
    if (mappers.size() >= 1)
      return;
    // Instantiate and add a default Mapper
    try {
      Class clazz = Class.forName(mapperClass);
      Mapper mapper = (Mapper) clazz.newInstance();
      mapper.setProtocol("http");
      addMapper(mapper);
    } catch (Exception e) {
      log(sm.getString("containerBase.addDefaultMapper", mapperClass), e);
    }
}
```

变量 mapperClass 的值定义与 StandardHost 类中：

```
private String mapperClass = "org.apache.catalina.core.StandardHostMapper";
```

下面是 StandardHostMapper 类中最重要的方法 map 的实现：

```
public Container map(Request request, boolean update) {
    // Has this request already been mapped?
    if (update && (request.getContext() != null))
      return (request.getContext());
    // Perform mapping on our request URI
    String uri = ((HttpRequest) request).getDecodedRequestURI();
    Context context = host.map(uri);
    // Update the request (if requested) and return the selected Context
    if (update) {
      request.setContext(context);
      if (context != null)
        ((HttpRequest) request).setContextPath(context.getPath());
      else
        ((HttpRequest) request).setContextPath(null);
    }
    return (context);
  }
```

注意，这里 map 方法仅仅是简单的调用了 host 的 map 方法。

13.5 StandardHostValve 类

org.apache.catalina.core.StandardHostValve 类是 StandardHost 的 basic valve。其 invoke 方法实现如下：

StanardHostValve.java

在 tomcat4 中，invoke 方法通过调用 StandardHost 的 map 方法来获取合适的 context：

```
// Select the Context to be used for this Request
StandardHost host = (StandardHost) getContainer();
Context context = (Context) host·map(request, true);
```

invoke 方法会通过 request 获取 session 对象，并调用其 access 方法。access 会修改 session 的最后访问时间。下面是 org·apache·catalina·session·StandardSession 类的 access 方法的实现：

```
public void access() {
    this·isNew = false;
    this·lastAccessedTime = this·thisAccessedTime;
    this·thisAccessedTime = System·currentTimeMillis();
}
```

最后，invoke 方法调用 context 的 invoke 来处理 http 请求。

13.6 为什么必须要有一个 host

在 tomcat4 和 5 中，若一个 web 应用使用 ContextConfig 对象进行设置，就必须使用一个 host 对象。原因如下。

ContextConfig 对象需要锁定 web·xml 文件的位置。在其 applicationConfig 方法会试图打开 web·xml 文件，下面是 applicationConfig 方法的片段：

```
synchronized (webDigester) {
    try {
```

```
URL url = servletContext.getResource(Constants.ApplicationWebXml);
InputSource is = new InputSource(url.toExternalForm());
is.setByteStream(stream);
...
webDigester.parse(is);
...
```

其中，Constants.ApplicationWebXml 的值为“/WEB-INF/web.xml”，servletContext 是一个 org.apache.catalina.core.ApplicationContext 类型（实现了 javax.servlet.ServletContext 接口）的对象。

下面是 ApplicationContext 的 getResource 方法的部分实现代码：

```
public URL getResource(String path) throws MalformedURLException {
    DirContext resources = context.getResources();
    if (resources != null) {
        String fullPath = context.getName() + path;
        // this is the problem. Host must not be null
        String hostName = context.getParent().getName();
```

其中，最后一行清楚的表明了，若是使用 ContextConfig 对象进行配置的的话，context 必须有一个父 container，也就是 host。

13.7 应用程序 1

本节的应用程序重在说明 host 的使用方法，程序包含两个类 ex13.pyrmont.core.SimpleContextConfig 类和 ex13.pyrmont.startup.Bootstrap1 类。其中 SimpleContextConfig 类与第 11 章相同，下面是 Bootstrap1 的实现代码：

Bootstrap1.java

13.8 Engine 接口

org·apache·catalina·Engine 接口表示了一个 engine 对象，也就是 tomcat 整个的 servlet 引擎。当部署 tomcat 时要使用多个虚拟机的话，就需要使用 engine。事实上，部署的 tomcat 都会是使用一个 engine。Engine 接口定义如下：

Engine.java

13.9 StandardEngine 类

org·apache·catalina·core·StandardEngine 类是 Engine 接口的标准实现。相比与 StandardContext 类和 StandardHost 类，StandardEngine 类相对小一些。在实例化的时候，StandardEngine 会添加一个 basic valve：

```
public StandardEngine() {
    super();
    pipeline·setBasic(new StandardEngineValve());
}
```

作为一个顶层 container，engine 的子 container 只能是 host，所以，若是给它设置以了一个非 host 的 container，会抛出异常。下面是 StandardEngine 类的 addChild 方法的实现代码：

```
public void addChild(Container child) {
    if (!(child instanceof Host))
        throw new IllegalArgumentException (sm·getString("StandardEngine·notHost"));
    super·addChild(child);
}
```

此外，engine 也不会有父 container 了，因此，调用其 setContainer 方法会抛出异常：

```
public void setParent(Container container) {
    throw new IllegalArgumentException
     (sm.getString("standardEngine.notParent"));
}
```

13.10 StandardEngineValve 类

org.apache.catalina.core.StandardEngineValve 类是 StandardEngine 的 basic valve。下面是 StandardEngineValve 类的 invoke 方法的实现代码：

StandardEngineValve.java

在验证了 request 和 response 对象的类型后，invoke 方法会获取 host 对象，用来处理请求。

13.11 应用程序 2

Bootstrap2.java

第 14 章　Server 与 Service

14.1　概述

在前面的章节中，已经说明了 connector 和 container 是如何工作的·在 8080 端口上，只能有一个 connector 服务于 http 请求。此外，在前面章节的应用程序中缺少了启动/停止 container 的机制。本章将对此说明，并介绍两个组件 server 和 service。

14.2　server

org·apache·catalina·Server 接口表示了整个 catalina 的 servlet 引擎，囊括了所有的组件。server 使用一种优雅的方法来启动/停止整个系统，不需要对 connector 和 container 分别启动/关闭。

当启动 server 时，server 会负责启动其所有的组件，然后就等待关闭命令。server 使用 service 来获取组件，如 connector 和 container。Server 接口定义如下：

Server.java

shutdown 属性保存了发送给 server 的关闭命令。port 属性保存了 server 会从哪个端口获取关闭命令。addService 方法用于为 server 添加 service，removeService 方法则可以删除某个 service。initialize 方法会在启动之前执行。

14.3　StandardServer

org·apache·catalina·core·StandardServer 类是 Server 接口的标准实现。其中的 shutdown 方法是最重要的，用于关闭 server。该类中的许多方法用于将 server 配置信息保存为一个新的 server·xml 文件。

server 中可以有 0 个或多个 service。StandardServer 提供了 addService，removeService 和 findServices 方法用于对 service 进行操作。与 StandardServer 的生命周期有关的方法有：initialize，start，stop 和 await。与其他组件类似，initialize 用于初始化，start 用于启动，然后调用 await 等待关闭命令，最后，调用 stop 关闭

server。调用 await 方法后，server 会被阻塞，直到总 8085 端口（或其他端口，自定）收到了关闭命令。当 await 命令返回后，stop 方法会关闭所有的子组件。

14·3·1 initialize 方法

initialize 方法用于对 server 进行初始化添加到其中的 service，实现代码如下：

```
public void initialize() throws LifecycleException {
    if (initialized)
      throw new LifecycleException ( sm·getString("StandardServer·initialize·initialized"));
    initialized = true;
    // Initialize our defined Services
    for (int i = 0; i < services·length; i++) {
      services[i]·initialize();
    }
}
```

注意，当 initialize 方法调用过一次后，标志位 initialized 会被置为 true，再次调用该方法会抛出异常。

14·3·2 start 方法

start 方法用于启动 server，该方法会启动其所有的 service，继而启动所有的组件，如 connector 和 container。start 方法实现如下：

```
public void start() throws LifecycleException {
    // Validate and update our current component state
    if (started)
      throw new LifecycleException (sm·getString("standardServer·start·started"));
```

```
    // Notify our interested LifecycleListeners
    lifecycle.fireLifecycleEvent(BEFORE_START_EVENT, null);
    lifecycle.fireLifecycleEvent(START_EVENT, null);
    started = true;
    // Start our defined Services
    synchronized (services) {
      for (int i = 0; i < services.length; i++) {
        if (services[i] instanceof Lifecycle)
          ((Lifecycle) services[i]).start();
      }
    // Notify our interested LifecycleListeners
    lifecycle.fireLifecycleEvent(AFTER_START_EVENT, null);
}
```

注意，start 方法只能调用一次，再次调用会抛出异常。

14.3.3 stop 方法

stop 方法用于关闭 server，实现代码如下：

```
public void stop() throws LifecycleException {
    // Validate and update our current component state
    if (!started)
      throw new LifecycleException(sm.getString("standardServer.stop.notStarted"));
    // Notify our interested LifecycleListeners
```

```
    lifecycle.fireLifecycleEvent(BEFORE_STOP_EVENT, null);
    lifecycle.fireLifecycleEvent(STOP_EVENT, null);
    started = false;
    // Stop our defined Services
    for (int i = 0; i < services.length; i++) {
      if (services[i] instanceof Lifecycle)
        ((Lifecycle) services[i]).stop();
    }
    // Notify our interested LifecycleListeners
    lifecycle.fireLifecycleEvent(AFTER_STOP_EVENT, null);
  }
```

同样，该方法只能调用一次。

14.3.4 await 方法

await 方法负责等待关闭整个 tomcat 系统的命令。实现代码如下：

StandardServer.java

await 方法创建一个 ServerSocket 对象，监听 8085 端口，当接收到关闭命令时，就会执行关闭命令。

14.4 Service 接口

org.apache.catalina.Service 接口表示了一个 service。一个 service 可以持有一个 container 和多个

connector，所有的 connector 都会与这个 container 相关联。Service 接口定义如下：

Service.java

14.5 StandardService 类

org·apache·catalina·core·StandardService 类是 Service 接口的标准实现。StandardService 的 initialize 方法用于初始化添加到其中的 connector。StandardService 类还实现 Lifecycle 接口，因此，它也可以启动 connector 和 container。

14·5·1 connector 和 container

StandardService 对象中有两种组件：connector 和 container。其中 container 只有一个，connector 可以有多个。多个 connector 使用 tomcat 可以多种不同的请求协议。例如，一个处理 http 请求，一个处理 https 请求。connector 和 container 组件使用下面的变量进行引用：

```
private Container container = null;

private Connector connectors[] = new Connector[0];
```

使用 setContainer 方法将 container 和 service 相关联，该方法实现如下：

```
public void setContainer(Container container) {

    Container oldContainer = this·container;

    //先将原先的 container 置空

    if ((oldContainer != null) && (oldContainer instanceof Engine))

      ((Engine) oldContainer)·setService(null);

    //设置新的 container

    this·container = container;

    //进行关联操作
```

```
if ((this.container != null) && (this.container instanceof Engine))
    ((Engine) this.container).setService(this);
//启动 container
if (started && (this.container != null) && (this.container instanceof Lifecycle)) {
   try {
      ((Lifecycle) this.container).start();
   } catch (LifecycleException e) {
      ;
   }
}
  //将 connector 和 container 相关联
synchronized (connectors) {
   for (int i = 0; i < connectors.length; i++)
      connectors[i].setContainer(this.container);
}
//在新的 service 启动后，关闭原先的 service
if (started && (oldContainer != null) &&
   (oldContainer instanceof Lifecycle)) {
   try {
      ((Lifecycle) oldContainer).stop();
   }
   catch (LifecycleException e) {
      ;
```

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

addConnector 方法和 removeConenctor 方法分别用于添加/删除 connector 到 service 中。实现代码如下：

StandardService.java

14.5.2 与生命周期有关的方法

与生命周期有关的方法包括从 Lifecycle 接口中实现的 start/stop 方法，在加上 initialize 方法。其中 initialize 方法会调用该 service 中所有 connector 的 initialize 方法。initilize 方法实现代码如下：

```
public void initialize() throws LifecycleException {
    if (initialized)
      throw new LifecycleException ( sm.getString("StandardService.initialize.initialized"));
    initialized = true;
    // Initialize our defined Connectors
    synchronized (connectors) {
      for (int i = 0; i < connectors.length; i++) {
        connectors[i].initialize();
      }
    }
```

```
}
```

start 方法负责启动该 service 持有的 connector 和 container，实现代码如下：

```
public void start() throws LifecycleException {
    // Validate and update our current component state
    if (started) {
      throw new LifecycleException (sm.getString("standardService.start.started"));
    }
    // Notify our interested LifecycleListeners
    lifecycle.fireLifecycleEvent(BEFORE_START_EVENT, null);
    System.out.println (sm.getString("standardService.start.name", this.name));
    lifecycle.fireLifecycleEvent(START_EVENT, null);
    started = true;
    // Start our defined Container first
    if (container != null) {
      synchronized (container) {
        if (container instanceof Lifecycle) {
          ((Lifecycle) container).start();
        }
      }
    }

    // Start our defined Connectors second
    synchronized (connectors) {
```

```
        for (int i = 0; i < connectors.length; i++) {
            if (connectors[i] instanceof Lifecycle)
                ((Lifecycle) connectors[i]).start();
        }
    }
    // Notify our interested LifecycleListeners
    lifecycle.fireLifecycleEvent(AFTER_START_EVENT, null);
}
```

stops 方法用于关闭 connector 和 container，实现代码如下：

```
public void stop() throws LifecycleException {
    // Validate and update our current component state
    if (!started) {
        throw new LifecycleException (sm.getString("standardService.stop.notStarted"));
    }
    // Notify our interested LifecycleListeners
    lifecycle.fireLifecycleEvent(BEFORE_STOP_EVENT, null);
    lifecycle.fireLifecycleEvent(STOP_EVENT, null);
    System.out.println (sm.getString("standardService.stop.name", this.name));
    started = false;
    // Stop our defined Connectors first
    synchronized (connectors) {
        for (int i = 0; i < connectors.length; i++) {
            if (connectors[i] instanceof Lifecycle)
```

```
            ((Lifecycle) connectors[i]).stop();
          }
        }
        // Stop our defined Container second
        if (container != null) {
          synchronized (container) {
            if (container instanceof Lifecycle) {
              ((Lifecycle) container).stop();
            }
          }
        }
        // Notify our interested LifecycleListeners
        lifecycle.fireLifecycleEvent(AFTER_STOP_EVENT, null);
    }
```

注意，调用一次 start 方法后，在调用 stop 方法之前，是不能重复调用 start 方法的，否则会抛出异常。同理，对 stop 方法也是如此。

14.6 应用程序

本章的应用程序重在说明如何使用 server 和 service，主要使用三个类，SimpleContextConfig（与第 *13* 章相同），Bootstrap（用于启动程序）和 Stopper（用于关闭程序）。

14.6.1 Bootstrap 类

实现代码如下：

Bootstrap.java

Bootstrap 类的 main 方法的开始部分与第 13 章类似，它会创建一个 engine，一个 host，一个 context，两个 wrapper 和一个 connector。然后，将它们进行关联。但，需要注意的是，connector 并没有与顶层 container（engine）相关联。相反，main 方法创建了一个 service 对象，和一个 server 对象，将 service 添加到 server 中：

```
Service service = new StandardService();
service.setName("Stand-alone Service");
Server server = new StandardServer();
server.addService(service);
```

然后，main 方法将 connector 和 engine 添加到 service 中：

```
service.addConnector(connector);
service.setContainer(engine);
```

这样，connector 就和 container 关联起来了。然后，main 方法调用 server 的 initialize 和 start 方法，初始化 connector，并将 connector 和 container 启动：

```
if (server instanceof Lifecycle) {
  try {
    server.initialize();
    ((Lifecycle) server).start();
```

接下来，main 方法调用 server 的 await 方法，进入循环等待，监听 8085 端口。注意，此时 connector 已经其中，但监听的是另一个端口 8080。当 await 返回时，说明接收到了关闭命令，此时，main 方法调用 server

的 stop 方法，关闭其所有组件。

14·6·2 Stopper 类

实现代码如下：

Stopper.java

Stopper 类通过 socket 向 server 发送一个关闭命令，从而关闭 server。

第 15 章 Digester

15.1 概述

正如前几章所述，我们使用 Bootstrap 类来实例化 connector，context，wrapper 和其他组件，然后调用各种 set 方法将它们关联起来，例如：

```
Connector connector = new HttpConnector();

Context context = new StandardContext();

connector.setContainer(context);
```

然后，再调用各种 set 方法设置属性，例如：

```
context.setPath("/myApp") ;

context.setDocBase("myApp");
```

此外，你还要实例化各种组件，并将它们添加到 context 中，例如：

```
LifecycleListener listener = new SimpleContextConfig();

((Lifecycle) context).addLifecycleListener(listener);

Loader loader = new WebappLoader();

context.setLoader(loader);
```

这些都完成后，就可以启动系统了：

```
connector.initialize();

((Lifecycle) connector).start ();

((Lifecycle) context).start();
```

上面的方法并不好，将所有的配置都写到代码里很不利于使用和扩展。 tomcat 中使用 xml 文件（如 server.xml）来记录系统配置。例如，server.xml 中，一个 Context 标签表示一个 context 容器。设置 path 和 docPath

属性可以使用下面的配置：

```
<context docBase="myApp" path="/myApp"/>
```

tomcat 使用了开源库 Digester 将 xml 文件中的元素转换为 java 对象。

15.2 Digester

Digester 是 apache commons 下的一个子项目，包含下面三个包：

（1）org.apache.commons.digester：该包可用于处理 xml 文件；

（2）org.apache.commons.digester.rss：可用于解析 rss；

（3）org.apache.commons.digester.xmlrules：该包提供了基于 xml 的定义。

15.2.1 Digester 类

org.apache.commons.digester.Digester 类是 Digester 库的主类。该类可用于解析 xml 文件，对与 xml 文档中的每个元素，Digester 对象都会检查是否要做事先预定义的事件。在进行 xml 解析之前，开发人员需要设计匹配模式。以下面的 example.xml 文件为例：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<employee firstName="Brian" lastName="May">
    <office>
      <address streeName="Wellington Street" streetNumber="110"/>
    </office>
</employee>
```

该 xml 文件中根元素是 employee，employee 元素的有一个模式 employee。office 是 employee 的子元素，因此，office 的模式是 employee/office。以此类推，address 元素的模式是 employee/office/address。

规则指明了当 Digester 找到了某个模式时需要做的一个或多个动作。规则使用 org·apache·commons·digester·Rule 类表示。Digester 对象中包含 0 个或多个 Rule 对象。在 Digeser 对象中，规则和模式使用使用 org·apache·commons·digester·Rules 对象进行存储的。每次添加一个 rule 到 digester 中时，rule 实际上都被添加到 rules 中。

此外，rule 对象有 begin 和 end 方法。当开始解析 xml 文件时，每当遇到一个符合某个模式的开始标签时，digester 会调用 rule 的 begin 方法。而当遇到结束标签时，digester 会调用 rule 的 end 方法。

下面是在解析 example·xml 文件时，digester 会做的几件事：

（1）digester 会先遇到 employee 的开始标签，因此，它会检查是否有 rule 和模式 employee 相关联。若有，调用该 rule 的 begin 方法；

（2）然后，它会遇到 office 的开始标签，检查是否有 rule 与模式 employee/office 相关联，若有，调用该 rule 的 begin 方法；

（3）接下来遇到 address 的开始标签，执行过程与前面相同；

（4）然后，遇到前面已经遇到的标签的结束标签，调用各自 rule 的 end 方法。

15.2.1.1 创建对象

若想要 digester 根据找到的模式创建相应的对象，则可以调用 addObjectCreate 方法。该方法有四个重载版本，比较有用的是下面两个：

public void addObjectCreate(java·lang·String pattern，java·lang·Class clazz)

public void addObjectCreate(java·lang·String pattern，java·lang·String className)

当你想要创建 Employee 对象时，就可以像这样调用：

digester·addObjectCreate("employee"，ex15·pyrmont·digestertest·Employee·class);

addObjectCreate 方法的另两个重载函数允许你创建定义在 xml 文件中类，无需通过参数进行。这个特点很有用，可以在运行时决定类的名称。方法签名如下：

public void addObjectCreate(java·lang·String pattern，java·lang·String className，java·lang·String attributeName)

public void addObjectCreate(java·lang·String pattern，java·lang·String attributeName，java·lang·Class

clazz)

在这两个方法中，参数 attributeName 指明了 xml 文件中元素的属性名，该属性的值就是要被实例化的类的名字。例如，可以通过下面的代码和 xml 文件创建 Employee 对象：

<employee firstName="Brian" lastName="May" className="ex15.pyrmont.digestertest.Employee">

digester.addObjectCreate("employee", null, "className");

或者在 addObjectCreate 方法指定类的默认名，当找不到属性，这使用默认的类名：

digester.addObjectCreate("employee", "ex15.pyrmont.digestertest.Employee", "className");

创建的对象会被存储在一个栈中，可以用 pop，push 等方法对栈进行操作。

15.2.1.2 设置属性

另一个比较有用的方法是 addSetProperties，使用该方法可以为创建的对象设置属性。其中的一个重载方法的签名是：

public void addSetProperties(java.lang.String pattern)

使用时，传入一个模式字符串，像下面这样：

digester.addObjectCreate("employee", "ex15.pyrmont.digestertest.Employee");

digester.addSetProperties("employee");

上面的 digester 对象有两个 rule，创建对象和设置属性，都是通过“employee”模式触发的。rule 按照其添加到 digester 中的顺序执行。对于下面的元素，

<employee firstName="Brian" lastName="May">

Digester 对象会首先创建 ex15.pyrmont.digestertest.Employee 对象，然后调用 setFirstName 和 setLastName 方法设置属性。

15.2.1.3 调用方法

Digester 类允许通过添加一个 rule 的方式来调用栈顶对象的方法。该方法签名如下：

```
public void addCallMethod (java.lang.String pattern, java.lang.String methodName)
```

15.2.1.4 创建对象之间的关系

Digester 对象中包含有一个内部栈，用于临时存储创建的对象。当使用 addObjectCreate 方法创建一个对象时，生成的对象会存储在这个栈中。

addSetNext 方法用于创建栈中两个对象之间的关系，实际上，这两个对象是 xml 文件中两个具有父子关系的标签所对应的对象。方法签名如下：

```
public void addSetNext(java.lang.String pattern, java.lang.String methodName)
```

其中，参数 pattern 是子元素对应的模式，参数 methodName 是父元素添加子元素时使用的方法名。考虑到 Digester 创建对象时会先创建父元素对应的对象，所以相比于子元素对象，父元素对象会更靠近栈底。实例代码如下：

```
digester.addObjectCreate("employee", "ex15.pyrmont.digestertest.Employee");

digester.addObjectCreate("employee/office", "ex15.pyrmont.digestertest.Office");

digester.addSetNext("employee/office", "addOffice");
```

15.2.1.5 验证 xml 文档

Digester 对象解析的 xml 文档的有效性可通过 schema 进行验证，然后将结果记录与 Digester 对象的 validating 属性中，该属性默认为 false。

setValidating 方法用于设置是否要对 xml 文件进行验证，方法签名如下：

```
public void setValidating(boolean validating)
```

15.2.2 Digester 示例 1

本程序展示了如何使用 Digester 动态的创建对象并设置属性。其中示例类 Employee.java 和对应的 xml 代码如下：

Employee.java employee1.xml

Employee 类有三个属性，firstName， lastName 和 office，各自有 getter/setter 方法，还有一个方法，printName。应用程序使用一个 Digester 对象，添加 rules 来创建 Employee 对象并设置属性。代码如下：

Test01.java

15.2.3 Digester 示例 2

该应用程序展示了如何创建对象及对象间的关系。其中 Office.java，Address.java 和 xml 代码如下：

Office.java Address.java employee2.xml

Test02.java 代码如下：

Test02.java

15.2.4 Rule 类

Rule 类中包含了一些方法，其中最重要的是 begin 和 end 方法。当 Digester 对象遇到 xml 文档的开始标签时，会调用匹配规则的 begin 方法；当遇到结束标签时，会调用 end 方法。Rule 类的 begin 和 end 方法的签名如下：

```
public void begin(org.xml.sax.Attributes attributes) throws java.lang.Exception;
```

```
public void end() throws java.lang.Exception
```

Digester 对象是如何完成这些工作的呢？当调用 digester 对象的 addObjectCreate，addCallMethod，addSetNext 或其他方法时，你可以直接调用 digester 的 addRule 方法，该方法会将一个 rule 对象和它所匹配的模式添加到 digester 对象中。addRule 方法签名如下：

```
public void addRule(java.lang.String pattern, Rule rule);
```

Digester 类的 addRule 方法的实现如下：

```
public void addRule(String pattern, Rule rule) {
    rule.setDigester(this);
    getRules().add(pattern, rule);
}
```

查看 Digester 类的 addObjectCreate 方法的重载实现如下：

```
public void addObjectCreate(String pattern, String className) {
    addRule(pattern, new ObjectCreateRule(className));
}
public void addObjectCreate(String pattern, Class clazz) {
    addRule(pattern, new ObjectCreateRule(clazz));
}
public void addObjectCreate(String pattern, String className,
    String attributeName) {
    addRule(pattern, new ObjectCreateRule(className, attributeName));
}
public void addObjectCreate(String pattern,
    String attributeName, Class clazz) {
```

```
        addRule(pattern, new ObjectCreateRule(attributeName, clazz));
    }
```

这四个重载的方法都调用了 addRule 方法，ObjectCreateRule 类是 Rule 的子类，该类的实例可作为 addRule 方法的第二个参数使用。下面是 ObjectCreateRule 类中 begin 和 end 方法的实现：

ObjectCreateRule.java

begin 方法的最后三行创建了一个对象，然后放到 digester 的内部栈中。end 方法会从栈中获取对象。

15.2.5 Digester 示例 3：使用 UsingSet

向 digester 对象中添加 rule 还可以调用 addRuleSet 方法，方法签名如下：

```
public void addRuleSet(RuleSet ruleSet);
```

org.apache.commons.digester.RuleSet 类表示了 rule 对象的集合。该接口定义了两个方法 addRuleInstance 和 getNamespaceURI，方法签名如下：

```
public void addRuleInstance(Digester digester);

public java.lang.String getNamespaceURI();
```

addRuleInstance 方法会添加 rule 集合到 digester 对象中，getNamespaceURI 方法返回命名空间 uri，该 uri 会匹配到 RuleSet 中所有的 rule 对象。因此，创建了 digester 对象后，可以创建一个 RuleSet 对象，然后将 RuleSet 对象传给 digester 的 addRuleSet 方法。

RuleSetBase 类实现了 RuleSet 接口，RuleSetBase 是一个虚类，提供了 getNamespaceURI 方法的实现，使用者只需要提供 addRuleInstance 方法的实现即可。

下面要改造一下 Test02 类，引入 EmployeeRuleSet 类。EmployeeRuleSet 实现如下：

EmployeeRuleSet.java

注意 EmployeeRuleSet 类中 addRuleInstances 方法的实现，它添加的规则与 Test02 类中相同。Test03 类创建了 EmployeeRuleSet 实例，并将它 digester 对象中。Test03 实现代码如下：

Test03.java

当程序运行时，Test03 会产生与 Test02 相同的输出。之所以 Test03 的代码稍短一些是因为有一些代码被隐藏到 EmployeeRuleSet 类中。

接下来，你会看到，catalina 使用了 RuleSetBase 的子类来初始化 server 和其他组件。

15.3 ContextConfig

与其他组件不同，StandardContext 必须有一个监听器，这个监听器会负责配置 StandardContext 实例属性，设置成功后会将 StandardContext 的变量 configured 置为 true。在前面的章节中，我们使用 SimpleContextConfig 类作为 StandardContext 的监听器，这个类非常简单，唯一的用途就是设置 configured 变量。

在 tomcat 的实际应用中，StandardContext 的标准监听器是 org·apache·catalina·startup·ContextConfig 类。与简单的 SimpleContextConfig 不同，ContextConfig 会执行很多对 StandardContext 来说必不可少的任务。例如，当 StandardContext 安装 authenticator 到其 pipeline 中时，ContextConfig 就会参与其中。此外，它还添加了一个 certificate valve（org·apache·catalina·valves·CertificateValve）到 pipeline 中。

但是，最重要的是，ContextConfig 对象还要读取 catalina 默认的 web·xml 和应用程序自定义的 web·xml，并将 xml 元素转换为 java 对象。默认的 web·xml 位于$CATALINE_HOME/conf 下，其中定义了很多 servlet，并对文件类型做了映射，指定了 session 的过期时间。应用程序自定义的 web·xml 位于 WEB-INF 下。其实这两个都不是必须的。

在解析 xml 文件时，每当遇到一个 servlet 定义，ContextConfig 对象就创建一个 StandardWrapper 对象，这样就省去了手动创建 wrapper 的工作，只需要创建 ContextConfig 对象，并调用 StandardContext 的 addLifecycleListener 方法将 ContextConfig 对象添加到 StandardContext 之中就可以了：

```
LifecycleListener listener = new ContextConfig();
((Lifecycle) context)·addLifecycleListener(listener);
```

当启动时，StandardContext 会触发三个事件，BEFORE_START_EVENT，START_EVENT，AFTER_START_EVENT，而在关闭时会触发另外三个关闭事件，BEFORE_STOP_EVENT，STOP_EVENT，

AFTER_STOP_EVENT。ContextConfig 对象会对其中的 START_EVENT 和 STOP_EVENT 作出响应。其中，ContextConfig 的响应函数 lifecycleEvent 的实现如下：

```
public void lifecycleEvent(LifecycleEvent event) {
    // Identify the context we are associated with
    try {
      context = (Context) event.getLifecycle();
      if (context instanceof StandardContext) {
        int contextDebug = ((StandardContext) context).getDebug();
        if (contextDebug > this.debug)
          this.debug = contextDebug;
      }
    }
    catch (ClassCastException e) {
      log(sm.getString("contextConfig.cce", event.getLifecycle()), e);
      return;
    }
    // Process the event that has occurred
    if (event.getType().equals(Lifecycle.START_EVENT))
      start();
    else if (event.getType().equals(Lifecycle.STOP_EVENT))
      stop();
  }
```

其中 ContextConfig 的 start 方法实现代码如下：

ContextConfig.java

15.3.1 defaultConfig 方法

defaultConfig 方法负责读取并解析位于$CATALINA_HOME/conf 目录下的默认 web.xml 文件，实现代码如下：

ContextConfig.java

defaultConfig 方法先创建一个指向默认 web.xml 文件的引用对象 file：

```
File file = new File(Constants.DefaultWebXml);
```

其中 DefaultWebXML 定义在 org.apache.catalina.startup.Constants 类中：

```
public static final String DefaultWebXml = "conf/web.xml";
```

然后 defaultConfig 方法处理该 xml 文件，它会先锁定 webDigester 对象，然后解析文件：

```
synchronized (webDigester) {
 try {
    InputSource is = new InputSource("file://" + file.getAbsolutePath());
    stream = new FileInputStream(file);
    is.setByteStream(stream);
    webDigester.setDebug(getDebug());
    if (context instanceof StandardContext)
      ((StandardContext) context).setReplaceWelcomeFiles(true);
```

```
webDigester.clear();
webDigester.push(context);
webDigester.parse(is);
```

15.3.2 applicationConfig 方法

applicationConfig 与 defaultConfig 方法类似，只不过它处理的是应用程序自定义的 web.xml 文件。实现代码如下：

ContextConfig.java

15.3.3 创建 digester

ContextConfig 类中，使用 Digester 类型的变量 webDigester 来引用一个 digester 对象：

```
private static Digester webDigester = createWebDigester();
```

使用 createWebDigester 方法创建了 digester 对象后，然后使用该 digester 对象解析 web.xml 文件，createWebDigester 方法的实现代码如下：

```
private static Digester createWebDigester() {
    URL url = null;
    Digester webDigester = new Digester();
    webDigester.setValidating(true);
    url = ContextConfig.class.getResource( Constants.WebDtdResourcePath_22);
    webDigester.register(Constants.WebDtdPublicId_22, url.toString());
    url = ContextConfig.class.getResource( Constants.WebDtdResourcePath_23);
```

```
        webDigester.register(Constants.WebDtdPublicId_23, url.toString());
        webDigester.addRuleSet(new WebRuleSet());
        return (webDigester);
    }
```

createWebDigester 方法调用了 webDigester 的 addRuleSet 方法，传入了一个 org.apache.catalina.startup.WebRuleSet 类型的参数。WebRuleSet 类是 org.apache.commons.digester.RuleSetBase 的子类。WebRuleSet 类的定义如下（这里并不是完整的类定义）：

WebRuleSet.java

15.4 应用程序

本章的应用程序重在说明如何使用 ContextConfig 对象，只包含 Bootstrap 一个类。Bootstrap 类的实现如下：

Bootstrap.java

第 16 章 Shutdown Hook

16.1 概述

在很多实际应用环境中，当管理员关闭了你的应用程序时，你需要做一些善后清理工作。但问题，管理员才不管你那一套，很有可能不给你做清理工作。例如，在 tomcat 的部署应用中，通过实例化一个 server 对象来启动 servlet 容器，调用其 start 方法，然后逐个调用组件的 start 方法。正常情况下，为了关闭这些已经启动的组件，你应该发送关闭命令（如 14 章所述）。

tomcat 中提供了 hook，可以在关闭过程中运行一些代码，执行清理的工作。在 java 中，有两种事件会关闭虚拟机：

（1）当调用 System·exit()方法或程序的最后一条代码执行完毕时，程序正常退出；

（2）用户键入 ctrl+c 或在未关闭 java 程序的情况下，退出系统，会导致 jvm 强制关闭。

在关闭 jvm 时，虚拟机会经过以下两个阶段：

（1）虚拟机启动所有已经注册的 shutdown hook。shutdown hook 是先前已经注册到 Runtime 中的线程，所有的 shutdown hook 会并发执行，直到完成任务；

（2）虚拟机调用所有没有被调用过多析构方法。

本章重在说明第一个阶段，因为该阶段允许程序员在关闭 jvm 前执行清理工作。实际上，shutdown hook 是一个 java·lang·Thread 子类的实例。创建 shutdown hook 的方法如下：

（1）创建一个 Thread 类的子类；

（2）实现 run 方法，当应用程序关闭时，会调用此方法；

（3）在你的应用程序中，实例化 shutdown hook；

（4）使用 addShutdownHook 方法将 shutdown hook 注册到当前 Runtime 中。

shutdown hook 会由 jvm 启动，不用你自己启动。下面的代码展示了如何使用 shutdown hook：

ShutdownHookDemo.java

16.2 tomcat 中的 shutdown hook

org·apache·catalina·startup·Catalina 类中，你可以找到 shutdown hook。CatalinaShutdownHook 类继承自 java·lang·Thread，实现了 run 方法，该方法中会调用 server 的 stop 方法。CatalinaShutdownHook 类定义如下：

```
protected class CatalinaShutdownHook extends Thread {

  public void run() {

    if (server != null) {

      try {

        ((Lifecycle) server)·stop();          }

      catch (LifecycleException e) {

        System·out·println("Catalina·stop: " + e);

        e·printStackTrace(System·out);

        if (e·getThrowable() != null) {

          System·out·println("----- Root Cause -----");

          e·getThrowable()·printStackTrace(System·out);

        }

      }

    }

  }

}
```

第 17 章　启动 tomcat

17.1　概述

本章重在说明 tomcat 启动的过程，使用了两个类，Catalina 和 Bootstrap，均在 org·apache·catalina·startup 包下。Catalina 类用于启动关闭 server 对象，并对 server·xml 文件进行解析。Bootstrap 类创建 Catalina 的实例，并调用其 process 方法。理论上，这两个类可以合并为一个。为了支持 tomcat 的多种运行模式，提供了多种 Bootstrap 类。例如，上述 Bootstrap 类就是为了将 tomcat 作为一个独立应用而准备的。此外，org·apache·catalina·startup·BootstrapService 类，用在 Windows NT 上运行 tomcat。为了使用方便，tomcat 提供了 bat 和 sh 文件来启动/关闭。

17.2　Catalina 类

org·apache·catalina·startup·Catalina 类是启动类。它包含了一个 Digester 对象用于解析 server·xml 文件。Catalina 类中还有一个 server 对象（server 中包含有 service），正如 15 章所述，service 对象中有一个 container 和一个或多个的 connector。

你可以通过实例化 Catalina 类，并调用其 process 方法来启动 tomcat。其中要传入合适的参数。第一个参数是 start（用于启动）或 stop（用于关闭）。其他可选的参数是-help，-config，-debug 和 -nonaming。

一般情况下，即使 Catalina 类本身提供了 main 方法可以启动 tomcat，你还是需要使用 Bootstrap 对象来实例化 Catalina 对象，并调用其 process 方法启动 tomcat。

tomcat4 中 Catalina 类的 process 方法的实现如下：

```
public void process(String args[]) {
    setCatalinaHome();
    setCatalinaBase();
    try {
      if (arguments(args))
        execute();
```

```
    } catch (Exception e) {
        e.printStackTrace(System.out);
    }
}
```

其中，process 方法设置了两个系统属性，catalina.home 和 catalina.base.catalina.home，默认值是 user.dir 属性的值。catalina.base 属性的值与 catalina.home 相同（注意，user.dir 属性值是用户的工作目录，例如，调用 java 命令的目录）。然后，process 调用 arguments 方法，传入命令行参数。arguments 方法实现如下：

```
protected boolean arguments(String args[]) {
    boolean isConfig = false;
    if (args.length < 1) {
        usage();
        return (false);
    }
    for (int i = 0; i < args.length; i++) {
        if (isConfig) {
            configFile = args[i];
            isConfig = false;
        } else if (args[i].equals("-config")) {
            isConfig = true;
        } else if (args[i].equals("-debug")) {
            debug = true;
        } else if (args[i].equals("-nonaming")) {
            useNaming = false;
```

```
        } else if (args[i].equals("-help")) {
            usage();
            return (false);
        } else if (args[i].equals("start")) {
            starting = true;
        } else if (args[i].equals("stop")) {
            stopping = true;
        } else {
            usage();
            return (false);
        }
      }
      return (true);
  }
```

process 方法检查 arguments 方法的返回值，若是返回 true，则继续调用 execute 方法。execute 方法的实现如下：

```
  protected void execute() throws Exception {
      if (starting)
        start();
      else if (stopping)
        stop();
  }
```

17·2·1 start 方法

start 方法会创建一个 Digester 对象来解析 server·xml 文件。在解析 server·xml 文件之前，start 方法会调用 Digester 对象的 push 方法，传入当前的 Catalina 对象为参数。这样，Catalina 对象就成了 Digester 对象内部对象栈的第一个对象。解析 server·xml 文件后，会将变量 server 指向一个 Server 对象（默认是 org·apache·catalina·core·StandardServer 类型的对象）。然后，start 方法会调用 server 的 initialize 和 start 方法。Catalina 对象的 start 方法会调用 Server 对象的 await 方法，server 对象会使用一个专用的线程来等待关闭命令。await 方法会循环等待，知道接收到正确的关闭命令。当 await 方法返回时，Catalina 对象的 start 方法会调用 server 对象的 stop 方法，从而关闭 server 对象和其他的组件。此外，start 方法还会注册 shutdown hook，确保服务器关闭时会执行 Server 对象的 stop 方法。start 方法的实现如下：

Catalina.java

17·2·2 stop 方法

Catalina 对象的 stop 方法会关闭 Server 对象，其实现如下：

Catalina.java

注意，stop 方法通过调用 createStopDigester 方法创建一个 Digester 对象，然后将 Catalina 对象 push 到 Digester 对象的内部对象栈中。

17·2·3 启动 Digester

createStartDigester 方法创建了一个 Digester 对象，然后将规则添加到其中，解析 server·xml 文件。添加到 Digester 对象中的规则是理解 tomcat 配置的关键。createStartDigester 方法的实现如下：

Catalina.java

17.2.4 关闭 Digester

createStopDigester 方法返回一个 Digester 对象来关闭 Server 对象。createStopDigester 方法实现如下：

```
protected Digester createStopDigester() {
    // Initialize the digester
    Digester digester = new Digester();
    if (debug)
        digester.setDebug(999);
    // Configure the rules we need for shutting down
    digester.addObjectCreate("Server", "org.apache.catalina.core.StandardServer", "className");
    digester.addSetProperties("Server");
    digester.addSetNext("Server", "setServer", "org.apache.catalina.Server");
    return (digester);
}
```

17.3 Bootstrap 类

org.apache.catalina.startup.Bootstrap 类提供了启动 tomcat 的切入点（还有一些其他的类也有此功能）。当使用 bat 或 sh 启动 tomcat 时，实际上会调用该类的 main 方法。在 main 方法中会创建三个 loader，并实例化 Catalina 对象，然后调用 Catalina 对象的 process 方法。Bootstrap 类的定义如下所示：

Bootstrap.java

Bootstrap 类定义了四个静态方法，其中 getCatalinaHome 返回 catalina.home 属性的值，若没有，则返回 user.dir 属性的值。getCatalinaBase 方法与 getCatalinaHome 方法类似。Bootstrap 类的 main 方法构造了三个 loader，之所以如此做是为了防止　运行 WEB-INF/classes 和 WEB-INF/lib 目录外的类。其中 commonLoader

允许从%CATALINA_HOME%/common/classes，%CATALINA_HOME%/common/endorsed 和%CATALINA_HOME%/common/lib 目录下载入类。catalinaLoader 负责载入 servlet 容器需要使用的类，它只会从%CATALINA_HOME%/server/classes 和%CATALINA_HOME%/server/lib 目录下查找。sharedLoader 会从%CATALINA_HOME%/shared/classes 和%CATALJNA_HOME%/shared/lib 目录，以及对 commonLoader 可用的目录下查找需要的类。然后，将 sharedLoader 设置为每个 web 应用的类载入器的父类载入器。

注意，sharedLoader 并不访问 catalina 的内部类，或 CLASSPATH 中的类。有关于类载入的机制，参见第 8 章。

在创建了三个 loader 之后，main 方法会载入 Catalina 类，实例化，并将之赋值给 startupInstance 变量。然后调用 setParentClassLoader 方法。最后，main 方法调用 Catalina 对象的 process 对象。

第 18 章　部署器

18.1　概述

要使用一个 web 应用，首先要将其部署到一个 host 中。在 tomcat 中，context 可以是 war 形式或将整个 web 应用拷贝到%CATALINA_HOME%/webapp 下。对于你所部署的每个 web 应用，可以包含一个描述文件（该文件是可选的），该文件中包含了对 context 的配置选项，是 xml 格式的文件。

注意，tomcat4 和 5 使用两个应用程序来管理 tomcat 及其应用的部署，分别是 manager 应用和 admin 应用。这里两个应用位于%CATALINA_HOME%/server/webapps 目录下，各自有一个描述文件，分别是 manager·xml 和 admin·xml。

本章将讨论使用一个部署器来部署 web 应用，部署器由 org·apache·catalina·Deployer 接口表示。部署器需要与一个 host 相关联，用于部署 context。部署一个 context 到 host，意思是创建一个 StandardContext 实例，并将该 context 实例添加到 host 中。该子 context 随父 host 而启动（因为 container 在启动时总是会调用其子 container 的 start 方法，除非该该 container 是一个 wrapper）。

本章会先说明 tomcat 部署器如何部署一个 web 应用，说明 Deployer 接口及其标准实现 org·apache·catalina·core·StandardHostDeployer 类。

18.2　部署一个 web 应用

在 15 章中，使用如下代码实例化一个 StandardHost 类，并添加一个 context 到其中。

```
Context context = new StandardContext();
context·setPath("/app1");
context·setDocBase("app1"); LifecycleListener listener = new ContextConfig();
((Lifecycle) context)·addLifecycleListener(listener);
Host host = new StandardHost();
host·addChild(context);
```

这是之前部署应用程序的方法。但是，这中硬编码的方法并不好。tomcat 中在 StandardHost 中使用了一个生命周期监听器（lifecycle listener）org·apache·catalina·startup·HostConfig 来部署应用。

当调用 StandardHost 的 start 方法时，会触发 START 事件，HostConfig 会响应该事件，调用其 start 方法，在该方法中会部署指定目录中的 web 应用。

回忆 *15* 章，当时说明 Digester 如何解析 xml 文件，但并没有讨论所有的规则，其中就跳过了对部署的讨论。

org·apache·catalina·startup·Catalina 类是启动类，使用 Digester 将 xml 转换为 java 对象。Catalina 类中定义了 createStartDigester 方法来添加规则到 Digester 中：

```
digester·addRuleSet(new HostRuleSet("Server/Service/Engine/"));
```

The org·apache·catalina·startup·HostRuleSet 类继承自 org·apache·commons·digester·RuleSetBase 类，作为 RuleSetBase 的子类，HostRuleSet 提供了 addRuleInstances 方法实现，该方法定义了 RuleSet 中的规则（Rule）。下面是 HostRuleSet 类的 addRuleInstances 方法的实现片段：

```
public void addRuleInstances(Digester digester) {
    digester·addObjectCreate(prefix + "Host", "org·apache·catalina·core·StandardHost", "className");
    digester·addSetProperties(prefix + "Host");
    digester·addRule(prefix + "Host", new CopyParentClassLoaderRule(digester));
    digester·addRule(prefix + "Host",
        new LifecycleListenerRule (digester, "org·apache·catalina·startup·HostConfig",
"hostConfigClass"));
```

如代码中所说，当出现模式 Server/Service/Engine/Host 时，会创建一个 org·apache·catalina·startup·HostConfig 实例，并被添加到 host，作为一个生命周期监听器。换句话说，HostConfig 对象会处理 StandardHost 对象的 start 和 stop 方法触发的事件。下面的代码是 HostConfig 的 lifecycleEvent 方法实现：

```
public void lifecycleEvent(LifecycleEvent event) {
    // Identify the host we are associated with
```

```
    try {
      host = (Host) event.getLifecycle();
      if (host instanceof StandardHost) {
        int hostDebug = ((StandardHost) host).getDebug();
        if (hostDebug > this.debug) {
          this.debug = hostDebug;
        }
        setDeployXML(((StandardHost) host).isDeployXML());
        setLiveDeploy(((StandardHost) host).getLiveDeploy());
        setUnpackWARs(((StandardHost) host).isUnpackWARs());
      }
    } catch (ClassCastException e) {
      log(sm.getString("hostConfig.cce", event.getLifecycle()), e);
      return;
    }

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

若 host 是一个 org·apache·catalina·core·StandardHost 实例，会调用 setDeployXML，setLiveDeploy 和 setUnpackWARs 方法：

```
setDeployXML(((StandardHost) host)·isDeployXML());

setLiveDeploy(((StandardHost) host)·getLiveDeploy());

setUnpackWARs(((StandardHost) host)·isUnpackWARs());
```

StandardHost 类的 isDeployXML 方法指明 host 是否要部署一个描述文件，默认为 true。liveDeploy 属性指明 host 是否要周期性的检查是否有新的应用部署。unpackWARs 属性指明 host 是否要解压缩 war 文件。

接收到 START 事件后，HostConfig 的 lifecycleEvent 方法会调用 start 方法来部署 web 应用：

```
protected void start() {

    if (debug >= 1)

        log(sm·getString("hostConfig·start"));

    if (host·getAutoDeploy()) {

        deployApps();

    }

    if (isLiveDeploy ()) {

        threadStart();

    }

}
```

若 autoDeploy 属性为 true（默认为 true），则 start 方法会调用 deployApps 方法。此外，若 liveDeploy 属性为 true（默认为 true），则该方法会开一个新线程调用 threadStart 方法。

deployApps 方法从 host 中获取 appBase 属性的值，该值定义于 server·xml 文件中。部署程序会从 %CATALINE_HOME%/webapps 目录下查找 web 应用进行部署。此外，该目录下找到的 war 文件和描述文件也会被部署。deployApps 方法实现如下：

```
protected void deployApps() {

    if (!(host instanceof Deployer))
```

```
        return;
    if (debug >= 1)
        log(sm.getString("hostConfig.deploying"));
    File appBase = appBase();
    if (!appBase.exists() || !appBase.isDirectory())
        return;
    String files[] = appBase.list();
    deployDescriptors(appBase, files);
    deployWARs(appBase, files);
    deployDirectories(appBase, files);
}
```

deployApps 方法会调用其他三个方法，deployDescriptors，deployWARs 和 deployDirectories。对于其他方法，deployApps 方法会传入 appBase 对象和 appBase 下所有的文件的名称。一个 context 是通过其路径来标识的，所有的 context 必须由其唯一路径。已经被部署的 context 会被添加到 HostConfig 对的部署列表中。因此，在部署一个 context 之前，deployDescriptors，deployWARs 和 deployDirectories 方法必须确保部署列表中的 context 不能有重复的路径。

注意，deployDescriptors，deployWARs 和 deployDirectories 三个方法的调用顺序是固定的，具体原因下回分解。

18.2.1 部署一个描述符

你可以编写一个 xml 文件来描述 context 对象，例如，在 tomcat4 和 5 中的 admin 和 manager 应用中就有如下两个描述文件：

```
<Context path="/admin" docBase="../server/webapps/admin" debug="0" privileged="true">
    <!-- Uncomment this Valve to limit access to the Admin app to
     localhost for obvious security reasons. Allow may be a comma-
```

```
        separated list of hosts (or even regular expressions).
      <Valve className="org.apache.catalina.valves.RemoteAddrValve"
        allow="127.0.0.1"/>
      -->
       <Logger      className="org.apache.catalina.logger.FileLogger"      prefix="localhost_admin_log."
suffix=".txt" timestamp="true"/>
</Context>
<Context path="/manager" docBase="../server/webapps/manager" debug="0" privileged="true">
    <!-- Link to the user database we will get roles from -->
    <ResourceLink name="users" global="UserDatabase"
        type="org.apache.catalina.UserDatabase"/>
</Context>
```

注意，这两个文件中都有 Context 标签，该标签中有 docBase 属性和 path 属性。HostConfig 类使用 deployDescriptors 方法部署在 %CATALINA_HOME%/webapps（tomcat4）目录或 %CATALINA_HOME%/server/webapps/（tomcat5）目录下找到的所有的 xml 文件。deployDescriptors 方法实现如下：

```
protected void deployDescriptors(File appBase, String[] files) {
    if (!deployXML)
      return;
    for (int i = 0; i < files.length; i++) {
      if (files[i].equalsIgnoreCase("META-INF"))
        continue;
      if (files[i].equalsIgnoreCase("WEB-INF"))
```

```
    continue;
if (deployed.contains(files[i]))
    continue;
File dir = new File(appBase, files[i]);
if (files[i].toLowerCase().endsWith(".xml")) {
    deployed.add(files[i]);
    // Calculate the context path and make sure it is unique
    String file = files[i].substring(0, files[i].length() - 4);
    String contextPath = "/" + file;
    if (file.equals("ROOT")) {
        contextPath = "";
    }
    if (host.findChild(contextPath) != null) {
        continue;
    }
    // Assume this is a configuration descriptor and deploy it
    log(sm.getString("hostConfig.deployDescriptor", files[i]));
    try {
        URL config = new URL("file", null, dir.getCanonicalPath());
        ((Deployer) host).install(config, null);
    } catch (Throwable t) {
        log(sm.getString("hostConfig.deployDescriptor.error", files[i]), t);
    }
```

```
        }
      }
    }
```

18.2.2 部署一个 war

可以将 web 应用以一个 jar 文件来部署。HostConfig 类使用 deployWARs 方法，将位于%CATALINA_HOME%/webapps 目录下的 war 文件进行部署。deployWARs 方法实现如下：

```
protected void deployWARs(File appBase, String[] files) {
    for (int i = 0; i < files.length; i++) {
      if (files[i].equalsIgnoreCase("META-INF"))
        continue;
      if (files[i].equalsIgnoreCase("WEB-INF"))
        continue;
      if (deployed.contains(files [i]))
        continue;
      File dir = new File(appBase, files [i]);
      if (files[i].toLowerCase().endsWith(".war")) {
        deployed.add(files [i]);
        // Calculate the context path and make sure it is unique
        String contextPath = "/" + files[i];
        int period = contextPath.lastIndexOf(".");
        if (period >= 0)
        contextPath = contextPath.substring(0, period);
```

```
if (contextPath.equals("/ROOT"))
    contextPath = "";
if (host.findChild(contextPath) != null)
  continue;
if (isUnpackWARs()) {
  // Expand and deploy this application as a directory
  log(sm.getString("hostConfig.expand", files[i]));
  try {
      URL url = new URL("jar:file:" + dir.getCanonicalPath() + "!/");
      String path = expand(url);
      url = new URL("file:" + path);
      ((Deployer) host).install(contextPath, url);
  } catch (Throwable t) {
      log(sm.getString("hostConfig.expand.error", files[i]), t);
  }
} else {
  // Deploy the application in this WAR file
  log(sm.getString("hostConfig.deployJar", files[i]));
  try {
    URL url = new URL("file", null, dir.getCanonicalPath());
    url = new URL("jar:" + url.toString() + "!/");
    ((Deployer) host).install(contextPath, url);
  } catch (Throwable t) {
```

```
                log(sm.getString("hostConfig.deployJar.error", files[i]), t);
            }
          }
        }
      }
    }
```

18.2.3 部署一个目录

你也可以直接将 web 应用的整个目录拷贝到%CATALINA_HOME%/webapps 目录下。HostConfig 类使用 deployDirectories 方法将该 web 应用目录进行部署。deployDirectories 方法实现如下：

```
protected void deployDirectories (File appBase, string[] files){
    for (int i = 0; i < files.length; i++) {
      if (files[i].equalsIgnoreCase("META-INF"))
        continue;
      if (files[i].equalsIgnoreCase("WEB-INF"))
        continue;
      if (deployed.contains(files[i]))
        continue;
      File dir = new File(appBase, files[i]);
      if (dir.isDirectory()) {
        deployed.add(files[i]);
        // Make sure there is an application configuration directory
        // This is needed if the Context appBase is the same as the
```

```
        // web server document root to make sure only web applications
        // are deployed and not directories for web space·
        File webInf = new File(dir, "/WEB-INF");
        if (!webInf·exists() || !webInf·isDirectory() || !webInf·canRead())
            continue;
        // Calculate the context path and make sure it is unique
        String contextPath = "/" + files[i];
        if (files[i]·equals("ROOT"))
            contextPath = "";
        if (host·findChild(contextPath) != null)
            continue;
        // Deploy the application in this directory
        log(sm·getString("hostConfig·deployDir", files[i]));
        try {
            URL url = new URL("file", null, dir·getCanonicalPath());
            ((Deployer) host)·install(contextPath, url);
        } catch (Throwable t) {
            log(sm·getString("hostConfig·deployDir·error", files[i]), t);
        }
    }
  }
}
```

18·2·4 动态部署

正如前面提到的，StandardHost 类使用 HostConfig 对象作为一个生命周期监听器。当 StandardHost 对象启动时，它的 start 方法会触发一个 START 事件，而 HostConfig 对象会对该事件进行响应，调用其 start 方法。在 tomcat4 中，start 方法的最后一行会调用 threadStart 方法（当 liveDeploy 属性为 true 时，该值默认为 true）：

```
if (isLiveDeploy()) {
  threadStart();
}
```

threadStart 方法会启动一个新线程并调用其 run 方法。run 方法会定期检查是否有新应用要部署，或已经部署的应用是否有修改。run 方法实现如下：

```
/**
   * The background thread that checks for web application autoDeploy and changes to the web·xml
config·
   */
public void run() {
    if (debug >= 1)
      log("BACKGROUND THREAD Starting");
    // Loop until the termination semaphore is set
    while (!threadDone) {
      // Wait for our check interval
      threadSleep();
      // Deploy apps if the Host allows auto deploying
      deployApps();
      // Check for web·xml modification
      checkWebXmlLastModified();
```

```
    }
    if (debug >= 1)
      log("BACKGROUND THREAD Stopping");
  }
```

threadSleep 方法会是该线程沉睡一段时间，该时间由 checkInterval 值指定，默认为 15 秒，即每个 15 秒检查一次。

在 tomcat5 中，就没有再使用专用线程检查，而是由 StandardHost 的 backgroundProcess 方法周期性的触发一个"check"事件：

```
public void backgroundProcess() {
    lifecycle.fireLifecycleEvent("check", null);
}
```

接收到 check 事件后，监听器（HostConfig 对象）会调用其 check 方法执行部署应用的检查：

```
public void lifecycleEvent(LifecycleEvent event) {
    if (event.getType().equals("check"))
      check();
    …
```

check 方法的实现如下：

```
protected void check() {
    if (host.getAutoDeploy()) {
      // Deploy apps if the Host allows auto deploying
      deployApps();
      // Check for web.xml modification
```

```
            checkContextLastModified();
        }
    }
```

如代码所述，check 方法会调用 deployApps 方法，正如前面小节所述，deployApps 方法会调用 deployDescriptors，deployWARs 和 deployDirectories 方法。

在 tomcat5 中，check 方法会调用 checkContextLastModified 方法，列出所有已经部署的 web 应用，逐个检查其 web·xml 的时间戳及其 WEB-INF 目录，若有某个文件被修改了，会重启该应用。此外，checkContextLastModified 方法还会检查 war 文件的时间戳，重新部署那些被修改了的 war。

在 tomcat4 中，后台线程的 run 方法会调用 checkContextLastModified 方法，执行的任务与 tomcat5 中相同。

18.3 Deploy 接口

部署器由 org·apache·catalina·Deployer 接口表示，StandardHost 类实现了该接口，因此 StandardHost 对象也是一个部署器，web 应用可以部署到其中。Deployer 接口定义如下：

Deployer.java

StandardHost 类使用一个辅助类（org·apache·catalina·core·StandardHostDeployer）来完成部署应用的相关任务。下面的代码片段展示了 StandardHost 如何将部署任务代理给 StandardHostDeployer：

```
/**
 * The <code>Deployer</code> to whom we delegate application
 * deployment requests·
 */
private Deployer deployer = new StandardHostDeployer(this);
public void install(String contextPath, URL war) throws IOException {
```

```
    deployer.install(contextPath, war);
}
public synchronized void install(URL config, URL war) throws IOException {
    deployer.install(config, war);
}
public Context findDeployedApp(String contextPath) {
    return (deployer.findDeployedApp(contextPath));
}
public String[] findDeployedApps() {
    return (deployer.findDeployedApps());
}
public void remove(String contextPath) throws IOException {
    deployer.remove(contextPath);
}
public void start(String contextPath) throws IOException {
    deployer.start(contextPath);
}
public void stop(String contextPath) throws IOException {
    deployer.stop(contextPath);
}
```

18.4 StandardHostDeployer 类

org·apache·catalina·core·StandardHostDeployer 类是实际完成部署任务的类，由 StandardHost 对象使用：

```
public StandardHostDeployer(StandardHost host) {
    super();
    this·host = host;
}
```

18·4·1 安装一个描述符文件

StandardHostDeployer 类有两个 install 方法，本节和下一节分别来讨论。本节讨论的 install 方法用于安装一个描述符文件。StandardHost 对象在其 install 方法中调用 StandardHostDeployer 对象的 install 方法，StandardHostDeployer 对象的 install 方法实现如下：

```
public synchronized void install(URL config, URL war) throws IOException {
    // Validate the format and state of our arguments
    if (config == null)
        throw new IllegalArgumentException(sm·getString("StandardHost·configRequired"));
    if (!host·isDeployXML())
        throw new IllegalArgumentException (sm·getString("StandardHost·configNotAllowed"));
    // Calculate the document base for the new web application (if needed)
    String docBase = null; // Optional override for value in config file
    if (war != null) {
        String url = war·toString();
        host·log(sm·getString("StandardHost·installingWAR", url));
        // Calculate the WAR file absolute pathname
```

```
    if (url.startsWith("jar:")) {
        url = url.substring(4, url.length() - 2);
    }
    if (url.startsWith("file://"))
        docBase = url.substring(7);
    else if (url.startsWith("file:"))
        docBase = url.substring(5);
    else
        throw new IllegalArgumentException(sm.getString("standardHost.warURL", url));
}
// Install the new web application
this.context = null;
this.overrideDocBase = docBase;
InputStream stream = null;
try {
    stream = config.openStream();
    Digester digester = createDigester();
    digester.setDebug(host.getDebug());
    digester.clear();
    digester.push(this);
    digester.parse(stream);
    stream.close();
    stream = null;
} catch (Exception e) {
```

```
            host.log(sm.getString("standardHost.installError", docBase), e);
            throw new IOException(e.toString());
        } finally {
            if (stream != null) {
                try {
                    stream.close();
                } catch (Throwable t) {
                    ;
                }
            }
        }
    }
```

18.4.2 安装一个 war 文件或目录

第二个 install 接受一个字符串表示 context 路径，和一个 URL 表示 war 文件，该 install 方法实现如下：

```
public synchronized void install(String contextPath, URL war) throws IOException {
    // Validate the format and state of our arguments
    if (contextPath == null)
        throw new IllegalArgumentException(sm.getString("standardHost.pathRequired"));
    if (!contextPath.equals("") && !contextPath.startsWith("/"))
        throw new IllegalArgumentException(sm.getString("standardHost.pathFormat", contextPath));
    if (findDeployedApp(contextPath) != null)
        throw new IllegalStateException(sm.getString("standardHost.pathUsed", contextPath));
```

```
if (war == null)
   throw new IllegalArgumentException (sm.getString("standardHost.warRequired"));
// Calculate the document base for the new web application
host.log(sm.getString("standardHost.installing",
contextPath, war.toString()));
String url = war.toString();
String docBase = null;
if (url.startsWith("jar:")) {
    url = url.substring(4, url.length() - 2);
}
if (url.startsWith("file://"))
   docBase = url.substring(7);
else if (url.startsWith("file:"))
   docBase = url.substring(5);
else
   throw new IllegalArgumentException (sm.getString("standardHost.warURL", url));
// Install the new web application
try {
   Class clazz = Class.forName(host.getContextClass());
   Context context = (Context) clazz.newInstance();
   context.setPath(contextPath);
   context.setDocBase(docBase);
   if (context instanceof Lifecycle) {
      clazz = Class.forName(host.getConfigClass());
```

```
            LifecycleListener listener = (LifecycleListener) clazz.newInstance();
            ((Lifecycle) context).addLifecycleListener(listener);
        }
        host.fireContainerEvent(PRE_INSTALL_EVENT, context);
        host.addChild(context);
        host.fireContainerEvent(INSTALL_EVENT, context);
    } catch (Exception e) {
        host.log(sm.getString("standardHost.installError", contextPath), e);
        throw new IOException(e.toString());
    }
}
```

18.4.3 启动 context

StandardHostDeployer 类中的 start 方法用于启动 context，代码实现如下：

```
public void start(String contextPath) throws IOException {
    // Validate the format and state of our arguments
    if (contextPath == null)
        throw new IllegalArgumentException (sm.getString("standardHost.pathRequired"));
    if (!contextPath.equals("") && !contextPath.startsWith("/"))
        throw new IllegalArgumentException (sm.getString("standardHost.pathFormat", contextPath));
    Context context = findDeployedApp(contextPath);
    if (context == null)
        throw new IllegalArgumentException (sm.getstring("standardHost.pathMissing", contextPath));
```

```
    host.log("standardHost.start " + contextPath);
    try {
      ((Lifecycle) context).start();
    } catch (LifecycleException e) {
      host.log("standardHost.start " + contextPath + ": ", e);
      throw new IllegalStateException
      ("standardHost.start " + contextPath + ": " + e);
    }
  }
```

18.4.4 停止一个 context

StandardHostDeployer 类的 stop 方法可以用来停止 context：

```
public void stop(String contextPath) throws IOException {
    // Validate the format and state of our arguments
    if (contextPath == null)
      throw new IllegalArgumentException (sm.getstring("standardHost.pathRequired"));
    if (!contextPath.equals("") && !contextPath.startsWith("/"))
      throw new IllegalArgumentException (sm.getstring("standardHost.pathFormat", contextPath));
    Context context = findDeployedApp(contextPath);
    if (context == null)
      throw new IllegalArgumentException (sm.getstring("standardHost.pathMissing", contextPath));
    host.log("standardHost.stop " + contextPath);
    try {
```

```
        ((Lifecycle) context).stop();
    }
    catch (LifecycleException e) {
        host.log("standardHost.stop " + contextPath + ": ", e);
        throw new IllegalStateException ("standardHost.stop " + contextPath + ": " + e);
    }
}
```

第 19 章 Manager Servlet

19.1 概述

tomcat4 和 5 中使用 manager 应用可以方便的部署 web 应用程序。但是，manager 应用并不在%CATALINA_HOME%/webapps 目录下，而在%CATALINA_HOME%/server/webapps 目录下。当 tomcat 启动时，会安装 manager 应用，因为 manager 应用有一个描述符文件，manager·xml。

本章重在说明 manager 应用。

19.2 使用 Manager 应用

在 tomcat4 和 5 中，manager 应用位于%CATALINA_HOME%/server/webapps/manager 目录下。该引用中的主 servlet 是 ManagerServlet。在 tomcat4 中，该类位于 org·apache·catalina·servlets 包下。而在 tomcat5 中，该类位于 org·apache·catalina·manager 下，是作为 WEB-INF/lib 目录下的一个 jar 部署的。

注：相比于 tomcat5，tomcat4 中的 manager 更简单些，因此，本章以 tomcat4 中的 manager 为例进行讨论。

下面是 tomcat4 中部署描述符中 servlet 元素的定义：

```
<servlet>
  <servlet-name>Manager</servlet-name>
  <servlet-class>
    org·apache·catalina·servlets·ManagerServlet
  </servlet-class>
  <init-param>
    <param-name>debug</param-name>
    <param-value>2</param-value>
```

```
    </init-param>
  </servlet>
  <servlet>
    <servlet-name>HTMLManager</servlet-name>
    <servlet-class>
      org.apache.catalina.servlets.HTMLManagerServlet
    </servlet-class>
    <init-param>
      <param-name>debug</param-name>
      <param-value>2</param-value>
    </init-param>
  </servlet>
```

manager 应用的描述符文件 manager.xml 说明了该引用的 context 路径：

```
<Context path="/manager" docBase="../server/webapps/manager"
    debug="0" privileged="true">
    <!-- Link to the user database we will get roles from -->
    <ResourceLink name="users" global="UserDatabase" type="org.apache.catalina.UserDatabase"/>
</Context>
```

也就是说，下面的 url 可以访问 manager 应用：

http://localhost:8080/manager/

但是，指得注意的是在该描述符文件中，还有关于安全的限制：

```
<security-constraint>
    <web-resource-collection>
      <web-resource-name>Entire Application</web-resource-name>
      <url-pattern>/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <!-- NOTE:  This role is not present in the default users file -—>
      <role-name>manager</role-name>
    </auth-constraint>
  </security-constraint>
```

也就是说，整个应用被限制于只有拥有 manager 角色的用户才能访问。auth-login 元素指明了只有当用于输入正确的用户名密码时才能访问受限资源：

```
<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>Tomcat Manager Application</realm-name>
</login-config>
```

在 tomcat 中，用户和角色列表存储于 tomcat-users·xml 文件中。因此，要访问 Manager 应用，你必须添加一个 manager 角色，和一个拥有该角色的用户，例如：

```
<?xml version='1·0' encoding='utf-8'?>
<tomcat-users>
    <role rolename="manager"/>
    <user username="tomcat" password="tomcat" roles="manager "/>
```

```
</tomcat-users>
```

19.3 ContainerServlet 接口

实现了 org·apache·catalina·ContainerServlet 接口的 servlet 可以访问表示该 servlet 的 StandardWrapper 对象。通过访问 wrapper，它也就可以访问表示当前应用的 context，此外还可访问 deployer 等对象。ContainerServlet 接口定义如下：

```
package org·apache·catalina;



public interface ContainerServlet {

    public Wrapper getWrapper();

    public void setWrapper(Wrapper wrapper);

}
```

catalina 调用实现了 ContainerServlet 接口的 servlet 的 setWrapper 方法，将表示该 servlet 的 wrapper 传入。

19.4 初始化 ManagerServlet

通常来说，servlet 由一个 org·apache·catalina·core·StandardWrapper 实例表示。第一次调用 servlet 时，会先调用 StandardWrapper 的 loadServlet 方法，然后调用 init 方法。而对于 ManagerServlet 来说，在 loadServlet 方法中有如下代码段：

```
…

// Special handling for ContainerServlet instances

if ((servlet instanceof ContainerServlet) && isContainerProvidedServlet(actualClass)) {

    ((ContainerServlet) servlet)·setWrapper(this);
```

```
    }
    // Call the initialization method of this servlet
    try {
        instanceSupport.fireInstanceEvent(InstanceEvent.BEFORE_INIT_EVENT, servlet);
        servlet.init(facade); …
```

在这段代码中，表示该 servlet 的 wrapper 会被设置到 servlet 中。下面是 isContainerProvidedServlet 方法的实现代码：

```
    private boolean isContainerProvidedServlet(String classname) {
        if (classname.startsWith("org.apache.catalina.")) {
            return (true);
        }
        try {
            Class clazz = this.getClass().getClassLoader().loadClass(classname);
            return (ContainerServlet.class.isAssignableFrom(clazz));
        } catch (Throwable t) {
            return (false);
        }
    }
```

其中，参数 classname 是 ManagerServlet 类的全名，即 org.apache.catalina.servlets.ManagerServlet，这样，方法会返回 true。此外，当某个类是 ContainerServlet 的子类时，也会返回 true。

注意，java.lang.Class 类的 isAssignableFrom 方法只有当该类（接口）与参数指定的类（接口）相同或者是参数指定的类（接口）的父类（父接口）时，才会返回 true。

因此，标识 ManagerServlet 的类 StandardWrapper 的 loadServlet 方法会调用 ManagerServlet 类的 setWrapper 方法。下面是 ManagerServlet 类中 setWrapper 方法的实现代码：

```
public void setWrapper(Wrapper wrapper) {
    this.wrapper = wrapper;
    if (wrapper == null) {
        context = null;
        deployer = null;
    } else {
        context = (Context) wrapper.getParent();
        deployer = (Deployer) context.getParent();
    }
}
```

由于参数 wrapper 不为 null，这样执行 else 块后，就可以访问 context 和 deployer 了。

19.5 列出已经部署的 web 应用

你可以通过访问如下 url 来列出已经部署的 web 应用：

http://localhost:8080/manager/list

返回的结果如下所示：

```
OK - Listed applications for virtual host localhost
/admin:stopped:0:../server/webapps/admin
/app1:running:0:C:\123data\JavaProjects\Pyrmont\webapps\app1
/manager:running:0:../server/webapps/manager
```

上面的 url 会调用 manager 的 list 方法，该方法实现代码如下：

```
protected void list(PrintWriter writer) {
    if (debug >= 1)
      log("list: Listing contexts for virtual host '" + deployer.getName() + "'");
    writer.println(sm.getString("managerServlet.listed", deployer.getName()));
    String contextPaths[] = deployer.findDeployedApps();
    for (int i = 0; i < contextPaths.length; i++) {
      Context context = deployer.findDeployedApp(contextPaths[i]);
      String displayPath = contextPaths[i];
      if( displayPath.equals("") )
        displayPath = "/";
      if (context != null ) {
        if (context.getAvailable()) {
          writer.println(sm.getString("managerServlet.listitem", displayPath, "running", "" +
context.getManager().findSessions().length, context.getDocBase()));
        } else {
          writer.println(sm.getString("managerServlet.listitem", displayPath, "stopped", "0",
context.getDocBase()));
        }
      }
    }
}
```

list 方法会调用 deployer 的 findDeployedApps 方法来获取 catalina 中所有已经部署了的 context 的路径。然后，对路径列表进行迭代，获取每一个 context，然后检查该 context 是否可用。对每个可用的 context，打印出

其 context 信息。

19.6 启动 web 应用

可以使用如下的 url 来启动某个 context：

http://localhost:8080/manager/start?path=/contextPath

其中，contextPath 是该 context 的路径。若该 context 已经启动，则返回错误信息。该 url 实际上会访问 ManagerServlet 的 start 方法，该方法实现如下：

```
protected void start(PrintWriter writer, String path) {
    if (debug >= 1)
      log("start: Starting web application at '" + path + "'");
    if ((path == null) || (!path.startsWith("/") && path.equals(""))) {
      writer.println(sm.getString("managerServlet.invalidPath", path));
      return;
    }
    String displayPath = path;
    if( path.equals("/") )
      path = "";
    try {
      Context context = deployer.findDeployedApp(path);
      if (context == null) {
        writer.println(sm.getString("managerServlet.noContext", displayPath));
        return;
      }
```

```
            deployer.start(path);
            if (context.getAvailable())
                writer.println (sm.getString("managerServlet.started", displayPath));
            else
                writer.println (sm.getString("managerServlet.startFailed", displayPath));
        } catch (Throwable t) {
            getServletContext().log (sm.getString("managerServlet.startFailed", displayPath), t);
            writer.println (sm.getString("managerServlet.startFailed", displayPath));
            writer.println(sm.getString("managerServlet.exception", t.toString()));
        }
    }
```

19.7 关闭 web 应用

可以使用如下的 url 关闭某个 web 应用：

http://localhost:8080/manager/stop?path=/contextPath

其中，contextPath 是该 context 的路径。若该 context 已经关闭，则返回错误信息。该 url 实际上会访问 ManagerServlet 的 stop 方法，该方法实现如下：

```
protected void stop(PrintWriter writer, String path) {
    if (debug >= 1)
        log("stop: Stopping web application at '" + path + "'");
    if ((path == null) || (!path.startsWith("/") && path.equals(""))) {
        writer.println(sm.getString("managerServlet.invalidPath", path));
        return;
```

```
    }
    String displayPath = path;
    if( path.equals("/") )
      path = "";
    try {
      Context context = deployer.findDeployedApp(path);
      if (context == null) {
        writer.println(sm.getString("managerServlet.noContext", displayPath));
        return;
      }
      // It isn't possible for the manager to stop itself
     if (context.getPath().equals(this.context.getPath())) {
        writer.println(sm.getString("managerServlet.noSelf"));
        return;
      }
      deployer.stop(path);
      writer.println(sm.getString("managerServlet.stopped", displayPath));
    } catch (Throwable t) {
      log("ManagerServlet.stop[" + displayPath + "]", t);
      writer.println(sm.getString("managerServlet.exception", t.toString()));
    }
}
```

第 20 章　基于 JMX 的管理

19 章中讨论了 manager 应用，它展示了如何使用实现了 ContainerServlet 接口的 ManagerServlet 类来访问 catalina 的内部类。本章将说明如何使用 jmx 管理 tomcat。本章会先简要介绍 jmx，然后讨论 Common Modeler 库，tomcat 使用该库对 Managed Bean 进行管理。

20.1　jmx 简介

既然 ContainerServlet 已经可以管理 tomcat，为啥要用 jmx？因为 jmx 更灵活。许多基于服务器的应用程序，如 tomcat，jboss，jonas，geromino 等，使用 jmx 管理其资源。

jmx 规范（书中指规范 1·2·1 版本）定义了管理 java 对象的公开标准。例如，tomcat4 和 5 使用 jmx 来启动 servlet 容器中的各种对象（如 server，host，context，valve 等），这样更灵活，更易于管理。tomcat 的开发者也编写了 admin 应用来管理其他 web 应用。

由基于 jmx 的管理应用程序管理的 java 对象成为 jmx 托管资源。事实上，一个 jmx 托管资源也可以是一个应用程序，一个 service 实现，一个设备，一个用户等等。jmx 托管资源由 java 编写，提供了相应的包装。

要想将一个 java 对象转换为 jmx 托管资源，你需要另外创建一个名为 MBean 的对象。org·apache·catalina·mbeans 包下，包含有一些 MBean，例如，ConnectorMBean，StandardEngineMBean，StandardHostMBean，StandardContextMBean。由名字可猜想，ConnectorMBean 用于管理 connector，StandardContextMBean 用于管理 org·apache·catalina·core·StandardContext 实例。你也可以编写自己的 MBean 管理你自己的 java 对象。

MBean 类暴露出了它所管理的 java 对象的属性和方法。管理应用程序本身并不直接访问它所管理的 java 对象，因此，你可以选择性的暴露出需要的属性和方法。

拥有的 MBean 类后，你将它实例化，并将其注册到 MBean 服务器中。MBean 服务器是一个应用中所有 MBean 注册的中心地。管理应用程序通过 MBean 服务器访问 MBean。若是将 jmx 和 servlet 开发相比较的话，管理应用程序则等同于 web 浏览器，MBean 服务器等同于 servlet 容器，它提供了托管资源的访问。MBean 则等同于 servlet/jsp。web 浏览器并不直接访问 servlet/jsp，而是通过 servlet 容器来访问，管理应用程序也是通过 MBean 服务器访问 MBean 的。

有四种类型 MBean：standard，dynamic，open，model。standard Mbean 是四种类型中最容易编写的，但灵活性最差。这里重点讨论 tomcat 所使用的 model MBean。standard MBean 会用来写一个小例子来展示如何使用 jmx。

从结构上，jmx 分为三个层次，表现层，代理层，分布式服务层。MBean 服务器位于代理层，MBean 位于表现层。而分布式服务层将在以后的 jmx 的版本中涉及。

jmx 规范的表现层定义了编写 jmx 托管资源的标准方法，例如定义了如何编写一个 MBean。代理层定义了创建代理的规范。jmx 的代理包括一个 MBean 服务器，用于处理 MBean 服务。一般情况下，代理和 MBean 是位于同一个 jvm 中的。jmx 规范中有一个参考实现，因此，你不必自己实现 MBean 服务器，参考实现给出了实现 MBean 服务器的方法。

jmx 的参考实现可以从这里下载：

http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html

20.2 jmx api

jmx 的参考实现中包含了一个核心 java 库，位于 javax.management 包和其他一些包下（用于 jmx 的其他方方面）。

20.2.1 MBeanServer

javax.management.MBeanServer 接口表示一个 MBean 服务器。要创建一个 MBean 服务器，只需要调用 javax.management.MBeanServerFactory 类的 createMBean 即可。要将一个 MBean 注册到 MBean 服务器中，可以调用 MBeanServer 的 registerMBean 方法。下面是 registerMBean 方法的签名：

```
public ObjectInstance registerMBean(java.lang.Object object,
    ObjectName name) throws InstanceAlreadyExistsException,
  MBeanRegistrationException, NotCompliantMBeanException
```

要注册一个 MBean，需要传入一个 MBean 对象，和一个 ObjectName 对象。一个 ObjectName 实例类似于 HashMap 中的 key，可以唯一标识一个 MBean。registerMBean 方法返回一个 ObjectInstance 实例。javax.management.ObjectInstance 类包装了一个 MBean 和它的类名。

MBeanServer 接口提供了两个方法 queryNames 和 queryMBeans 用于获取某个 MBean 或符合某个匹配模式的一组 MBean。queryNames 方法返回 java.util.Set 对象，包含了匹配某个模式的一组 MBean 的名字。方法签名为：

```
public java.util.Set queryNames(ObjectName name, QueryExp query)
```

参数 query 执行过滤表达式。若参数 name 为 null，或者没有域，而且指定了 key 属性，则会返回所有已经注册的 MBean 的 ObjectName。若 query 为 null，则不会进行过滤。

queryMBeans 方法与 queryNames 方法类似，它返回的也是 java.util.Set，包含了针对所要查找的 MBean

的 ObjectInstance。方法签名为：

public java·util·Set queryMBeans(ObjectName name, QueryExp query)

如果你想要获得 MBean 的对象名称，你可以操作托管资源的属性或者调用 MBean 中暴露出的方法。通过调用 MBeanServer 的 invoke 方法，你可以调用已注册的 MBean 所暴露出的任何方法。MBeanServer 的 getAttribute 和 setAttribute 方法用于对已注册的 MBean 的属性进行操作。

20·2·2 ObjectName

MBean 注册于 MBeanServer，类似于 HashMap 中的 key，每个 MBean 通过一个对象名称进行唯一标识。对象名称由 javax·management·ObjectName 类表示。一个 ObjectName 包含两部分，一个域（domain）和一个 key/value 集合。domain 是一个字符串，可以为空。在一个对象名称中，domain 后面跟着一个冒号和一个或多个 key/value 对。key 是一个非空字符串，且不能包含下列字符：等号，逗号，冒号，问号或星号。一个对象名称中 key 是唯一的。 key 与 value 通过等号连接，key/value 对之间由逗号分隔。例如：

myDomain:type=Car,color=blue

在 MBeanServer 中查找 MBean 时，ObjectName 实例也可以表示属性模式。作为匹配模式的 ObjectName 可以在其 domain 和 key/value 中使用通配符。一个作为匹配模式的 ObjectName 可以有 0 个或多个 key。

20.3 Standard MBeans

standard MBean 是最简单的 MBean，要想通过 standard MBean 管理 java 对象，需要完成一些步骤。这里假设你要管理的 java 对象的类名为 Car。步骤如下：

（1）创建一个接口，该接口名规范为：你的 java 对象类名+MBean。例如，你想要管理的 java 对象的类名为 Car，则需要创建的接口名为 CarMBean；

（2）修改你的 java 类，让其实现 CarMBean 接口；

（3）创建一个代理，该代理中必须包含有 MBeanServer；

（4）为你的 MBean 创建 ObjectName 实例；

（5）实例化 MBeanServer；

（6）将 MBean 注册到 MBeanServer 中。

standard MBean 很容易编写，使用 MBean 需要修改你的 java 类（如需要实现创建的 MBen 接口），在某些工程里，这是不可行的。MBean 的其他类型则不需要这样的限制。

下面代码完成 Car 的 MBean 的创建：

Car.java　　CarMBean.java

你需要在接口中声明，你需要暴露出哪些需要操作的属性和方法。在本例中，CarMBean 接口暴露出了 Car 的所有方法。接下来，需要定义一个代理，StandardAgent，用来创建 MBean，管理 Car 对象。实现代码如下：

StandardAgent.java

StandardAgent 类是代理层，负责实例化 MBeanServer，使用 MBeanServer 对象注册 MBean：

```
public StandardAgent() {
    mBeanServer = MBeanServerFactory·createMBeanServer();
}
```

createMBeanServer 方法返回一个 jmx 参考实现的默认的 MBeanServer 对象。你有兴趣的话，可以自己实现一个 MBeanServer。

createObjectName 方法返回一个 ObjectName 对象。createStandardMBean 方法调用 MBeanServer 类的 createMBean 方法，createMBean 方法接收一个托管资源的类名和一个 ObjectName 实例来创建托管资源的 MBean 对象。createMBean 方法还会将创建的 MBean 注册到 MBeanServer 中。由于 standard MBean 的遵循了固定的命名约定，因此在 createMBean 方法中你不需要提供 MBean 的类型。

StandardAgent 的 main 方法会先创建一个 StandardAgent 对象，调用 getMBeanServer 方法获取一个 MBeanServer 对象引用：

```
StandardAgent agent = new StandardAgent();

MBeanServer mBeanServer = agent·getMBeanServer();
```

然后为 CarMBean 创建 ObejctName，MBeanServer 的默认 domain 被用于 ObejctName 的 domain，ObejctName 的 key/value 使用了 type 作为 key，托管资源的全类名作为 value：

```
String domain = mBeanServer.getDefaultDomain();
String managedResourceClassName = "ex20.pyrmont.standardmbeantest.Car";
ObjectName objectName = agent.createObjectName(domain + ":type=" +
managedResourceClassName);
```

然后，main 方法调用 createStandardBean 方法，传入 ObjectName 对象和托管资源的类名：

```
agent.createStandardBean(objectName, managedResourceClassName);
```

接下来，main 方法通过 CarMBean 来管理 Car 对象。它首先创建了一个 Attribute 对象 colorAttribute 来表示 Car 的 color 属性，并设置值为 blue。然后，它调用 setAttribute 方法，传入 objectName 和 colorAttribute。最后又通过 MBeanServer 的 invoke 方法调用了 Car 的 drive 方法。

```
// manage MBean
try {
  Attribute colorAttribute = new Attribute("Color","blue");
  mBeanServer.setAttribute(objectName, colorAttribute);
  System.out.println(mBeanServer.getAttribute(objectName, "Color"));
  mBeanServer.invoke(objectName,"drive",null,null);
}
```

20.4 Model MBeans

相比于 standard MBean，model MBean 更具灵活性。在编码上，model MBean 难度更大一下，但你也不再需要修改要管理的 java 类了（在 standard MBean 中，你必须修改要管理的 java 类，使其实现 MBean 接口）。

使用 model MBean，你不在需要创建 MBean 接口，相反，javax.management.modelmbean.ModelMBean 接口就是用了表示一个 model MBean，你只需要实现这个接口。jmx 也提供了对这个接口的默认实现，javax.management.modelmbean.RequiredModelMBean。你只需要实例化这个类或其子类。

注意，使用 ModelMBean 接口的其他实现也是可以的。例如，在下面章节讨论的 Commons Modeler 库就

提供了自定义的实现。

使用 model MBean 的难点在于告诉 ModelMBean 托管资源的哪些属性和方法可以暴露给代理层。这里，你需要创建一个 javax·management·modelmbean·ModelMBeanInfo 对象。ModelMBeanInfo 对象描述了要暴露给代理层的构造器，属性，操作和监听器。创建 ModelMBeanInfo 对象的工作单调乏味，但是一旦拥有，别无所求，只要将其与 ModelMBean 相关联即可。

使用 RequiredModelMBean 作为 ModelMBean 的实现，有两个方法将 ModelMBean 和 ModelMBeanInfo 相关联：

（1）在 RequiredModelMBean 的构造函数中传入 ModelMBeanInfo 对象；

（2）调用 RequiredModelMBean 对象的 setModelMBeanInfo 方法，传入 ModelMBeanInfo 对象。

在创建了 ModelMBean 之后，必须要调用其 setManagedResource 方法与托管资源相关联，该方法的签名如下：

```
public void setManagedResource(java·lang·Object managedResource,
    java·lang·String managedResourceType) throws MBeanException,
    RuntimeOperationsException, InstanceNotFoundException,
    InvalidTargetObjectTypeException
```

参数 managedResourceType 的值可以是下面的值之一，ObjectReference，Handle，IOR，EJBHandle 或 RMIReference。当前只支持 ObjectReference。

当然，你还需要创建 ObjectName，并将 MBean 注册到 MBeanServer 中。

20·4·1 MBeanInfo 与 ModelMBeanInfo

javax·management·mbean·ModelMBeanInfo 接口描述了要通过 ModelMBean 暴露给代理层的构造器，属性，操作和监听器。构造器由 javax·management·modelmbean·ModelMBeanConstructorInfo 表示。属性由 javax·management·modelmbean·ModelMBeanAttributeInfo 表示。操作由 javax·management·modelmbean·ModelMBeanOperationInfo 表示，监听器由 javax·management·modelmbean·ModelMBeanNotificationInfo 表示。本章只讨论属性和操作。

jmx 提供了 javax·management·modelmbean·ModelMBeanInfoSupport 的默认实现 ModelMBeanInfo。下面是 ModelMBeanInfoSupport 的构造器：

```
public ModelMBeanInfoSupport(java·lang·String className,
```

java·lang·String description, ModelMBeanAttributeInfo[] attributes,

ModelMBeanConstructorInfo[] constructors,

ModelMBeanOperationInfo[] operations,

ModelMBeanNotificationInfo[] notifications)

ModelMBeanAttributeInfo 的构造器如下：

public ModelMBeanAttributeInfo(java·lang·String name,

java·lang·String type, java·lang·String description,

boolean isReadable, boolean isWritable,

boolean isIs, Descriptor descriptor)

throws RuntimeOperationsException

下面是参数列表说明：

- name，属性名；
- type，属性的类型；
- description，属性描述说明；
- isReadable，若属性有 getter 方法，则为 true，否则为 false；
- isWritable，参考 isReadable；
- isIs，参考 isReadable；
- descriptor，Descriptor 对象，包含了 Attribute 对象的元数据，若置为 null，则会创建默认的 descriptor。

ModelMBeanOperationInfo 的构造器如下：

public ModelMBeanOperationInfo(java·lang·String name,

java·lang·String description, MBeanParameterInfo[] signature,

java·lang·String type, int impact, Descriptor)

throws RuntimeOperationsException

下面是参数列表说明：

- name，方法名；
- description，方法描述；
- signature，MBeanParameterInfo 数组，保存方法的参数；

- type，方法返回值的类型；
- impact，方法的影响（大概是这个意思），可选值如下，INFO，ACTION，ACTION_INFO，UNKNOWN；
- descriptor，Descriptor 对象，包含了 MBeanOperationInfo 对象的元数据，若置为 null，则会创建默认的 descriptor。

20.4.2 ModelMBean 实例

使用 model MBean 管理 Car 对象，你不需要再创建 MBean 接口，只需要实例化 RequiredMBean 类。下面的代码提供了代理层 ModelAgent，用于创建 MBean 来管理 Car 对象。ModelAgent 类实现代码如下：

ModelAgent.java

如代码所示，使用 model MBean，你需要多做一些工作，尤其是要声明所有要暴露出的属性，方法。但灵活性更好。下一节将介绍 Commons Modeler 库，该库可以帮助你更快的编写 model MBean。

20.5 Commons Modeler

Commons Modeler 库是 apache Jakarta 项目的一部分，目的是使编写 model MBeans 更加方便。最打的帮助是你不需要在写代码创建 ModelMBeanInfo 对象了。

回忆前面的章节，创建 RequiredModelMBean 对象时，你需要传入 RequiredModelMBean 的构造函数，创建一个 ModelMBeanInfo 对象：

```
ModelMBeanInfo mBeanInfo = createModelMBeanInfo(objectName, mbeanName);

RequiredModelMBean modelMBean = null;

try {

    modelMBean = new RequiredModelMBean(mBeanInfo);

}

…
```

ModelMBeanInfo 描述了要暴露出的属性和方法，实现 createModelMBeanInfo 方法是枯燥的，因为你不得不列出所有要暴露的属性和方法并将它们传给 ModelMBeanInfo 对象。

而使用 Commons Modeler 库，你就不再需要创建 ModelMBeanInfo 对象，对 MBean 的描述被封装在

org·apache·catalina·modeler·ManagedBean 对象中。你只需要编写一个简单你的描述文件（xml 格式），列出你想要创建的 MBean。对每个 MBean，你要列出他的全限定名，包括 MBean 的名字和托管资源的类名，还要写出要暴露出的属性和方法的名称。然后，使用 org·apache·commons·modeler·Registry 对象读取该 xml 文件，依照配置创建 MBeanServer 和所有的 ManagedBean。

然后，你就可以调用 ManagedBean 的 createMBean 方法创建 MBean 了。之后的事一切照旧。你需要创建一个 ObejctName 对象，将 MBean 注册到 MBeanServer 中。

下面，要说明下 MBean 的描述文件，然后讨论三个比较重要的类 Registry，ManagedBean 和 BaseModelMBean。

注意，tomcat4 中仍然使用了老版本的 Modeler，使用一些 deprecated 的方法。

20·5·1 MBean 描述符

MBean 的描述符是是一个 xml 文件，该文件对有 MBeanServer 管理的 model MBean 进行描述。MBean 描述符以下面的代码开始：

```
<?xml version="1.0"?>

<!DOCTYPE mbeans-descriptors PUBLIC

  "-//Apache Software Foundation//DTD Model MBeans Configuration File"

  "http://jakarta.apache.org/commons/dtds/mbeans-descriptors.dtd">
```

接下来是根元素：

```
<mbeans-descriptors>

…

</mbeans-descriptors>
```

mbeans-descriptors 标签中间部分 mbean 标签，每个 mbean 标签表示一个 model MBean，mbean 元素下的元素表示属性，方法，构造器，监听器。

20.5.1.1 mbean

mbean 元素描述了一个 model MBean，包含了创建 ModelMBeanInfo 对象的信息。mbean 元素有如下的定义：

<!ELEMENT mbean (descriptor?, attribute*, constructor*, notification*, operation*)>

mbean 元素定义了具体的规范，mbean 标签下可以有一个可选的 descriptor 标签，0 个或多个 attribute 标签，constructor 标签，notification 标签，operation 标签。

mbean 标签可以有如下的属性：

- className，ModelMBean 的实现类的全名，若该属性未赋值，则默认使用 org·apache·commons·modeler·BaseModelMBean 类；
- description，该 model MBean 的简单描述；
- domain，MBeanServer 的 domain 值，由 MBeanServer 创建的 ModelMBean 会属于这个 domain；
- group，即“grouping classification”，用来选择 MBean 相似实现的的组；
- name，model MBean 的唯一标识；
- type，托管资源实现类的全名。

20.5.1.2 attribute

使用 attribute 标签描述 MBean 的属性。attribute 标签有一个可选的 descriptor 元素，descriptor 元素有如下可选属性：

- description，该属性的简单描述；
- displayName，该属性的显示名称；
- getMethod，该属性的 getter 方法；
- is，一个 bool 值，指明该 bool 值是否有 getter 方法，默认为 false；
- name，属性名；
- readable，一个 bool 值，指明该属性是否可读，默认为 true；
- setMethod，该属性的 setter 方法；
- type，该属性的 Java 类的全名；
- writeable，一个 bool 值，表明该属性是否有 setter 方法，默认为 true。

20.5.1.3 operation

operation 标签描述了 mdel MBean 中要暴露给管理应用程序的公开方法，可以有 0 个或多个 parameter 元素和如下的属性：

- description，方法的简单描述；
- impact，指明方法的影响力，可选值为，ACTION（write like），ACTION-INFO（write+read like），INFO（read like）或 UNKNOWN；
- name，方法名；
- returnType，方法返回值的 java 类的全名。

20.5.1.4 parameter

- parameter 标签描述了传给构造函数或方法的参数。可以有如下属性：
- description，该参数的简单描述；
- name，参数名；
- type，参数的类型（java 全类名）。

20.5.2 mbean 标签实例

在 catalina 的 mbean-descriptors.xml 文件中声明了一系列 model MBean，该文件位于 org.apache.catalina.mbeans 包下，下面的代码是对托管资源 StandardServer 进行包装的 MBean 的声明：

```
<mbean name="StandardServer" className="org.apache.catalina.mbeans.StandardServerMBean"
    description="Standard Server Component"    domain="Catalina"  group="Server"
    type="org.apache.catalina.core.StandardServer">
    <attribute name="debug"  description="The debugging detail level for this component"  type="int"/>
    <attribute name="managedResource" description="The managed resource this MBean is associated with"
      type="java.lang.Object"/>
    <attribute name="port" description="TCP port for shutdown messages"  type="int"/>
    <attribute name="shutdown" description="Shutdown password" type="java.lang.String"/>
    <operation name="store" description="Save current state to server.xml file"  impact="ACTION"
      returnType="void">
```

```
    </operation>
</mbean>
```

mbean 元素声明了一个 model MBean，其唯一标识是 StandardServer。该 MBean 是一个 org·apache·catalina·mbeans·StandardServerMBean 的对象，负责管理 org·apache·catalina·core·StandardServer 对象。domain 是 Catalina，group 是 Server。

20·5·3 自己编写一个 model MBean

使用 Commons Modeler 库，需要在 mbean 元素的 className 属性中指明自定义的 model MBean 的全类名。默认情况下，Commons Modeler 使用 org·apache·commons·modeler·BaseModelMBean 类。有以下两种情况，你可能需要对其进行扩展：

- 你需要覆盖托管资源的属性；
- 你需要为托管资源添加属性或方法。

20·5·4 注册

org·apache·commons·modeler·Registry 类中有一些方法用于注册 MBean，下面是你可以使用该类做的事情：

- 获取一个 javax·management·MBeanServer 对象（你不在需要调用 javax·management·MBeanServerFactory 的 createMBeanServer 方法）；
- 使用 loadRegistry 方法读取描述符文件；
- 创建一个 ManagedBean 对象，用于创建 model MBean。

20·5·5 ManagedBean

ManagedBean 对象描述了一个 model MBean，该类用于取代 javax·management·MBeanInfo 类。

20.5.6 BaseModelMBean

org.apache.commons.modeler.BaseModelMBean 实现了 javax.management.modelmbean.ModelMBean 接口，使用这个类，就不需要用 javax.management.modelmbean.RequiredModelMBean 类了。

该类用一个比较有用的属性是 resource 属性。resource 表明了该 model MBean 管理的托管资源对象，resource 声明如下：

```
protected java.lang.Object resource;
```

20.5.7 使用 Modeler API

有一个类 Car 是要管理的，Car.java，现在使用 Commons Modeler 库，你不再需要在代码中纠结于 Car 类中众多的属性和方法。你需要将属性和方法写到 xml 文件中，例如下面的文件： 
car-mbean-descriptor.xml，

然后你需要编写一个代理类 ModelAgent，代码如下：ModelAgent2.java。

20.6 Catalian 中的 MBean

正如本章开头所说，Catalina 的 org.apache.catalina.mbeans 包中提供了一系列 MBean，这些 MBean 都直接或间接的继承自 org.apache.commons.modeler.BaseModelMBean。本节会讨论 tomcat4 中几个比较重要的 MBean，ClassNameMBean，StandardServerMBean 和 MBeanFactory。

20.6.1 ClassNameMBean

org.apache.catalina.mbeans.ClassNameMBean 类继承自 org.apache.commons.modeler.BaseModelMBean。它提供了一个只写的属性 className，用于表示托管资源的类名。该类定义如下：

```
package org.apache.catalina.mbeans;
```

```
import javax.management.MBeanException;
import javax.management.RuntimeOperationsException;
import org.apache.commons.modeler.BaseModelMBean;
public class ClassNameMBean extends BaseModelMBean {
    public ClassNameMBean() throws MBeanException, RuntimeOperationsException {
        super();
    }
    public String getClassName() {
        return (this.resource.getClass().getName());
    }
}
```

ClassNameMBean 类是 BaseModelMBean 的子类，其只写属性 className 在托管资源中不可见。mbeans-descriptors.xml 文件中的很多 mbean 元素使用该类作为其 model MBean。

20.6.2 StandardServerMBean

StandardServerMBean 类继承自 org.apache.commons.modeler.BaseModelMBean 类，用于管理 org.apache.catalina.core.StandardServer。StandardServerMBean 类覆盖了托管资源的一些方法，例如 store 方法。当管理应用程序调用 store 方法时，实际上会执行 StandardServerMBean 的 store 方法，而不是 StandardServer 的 store 方法。

```
package org.apache.catalina.mbeans;
import javax.management.InstanceNotFoundException;
import javax.management.MBeanException;
import javax.management.MBeanServer;
```

```
import javax.management.RuntimeOperationsException;
import org.apache.catalina.Server;
import org.apache.catalina.ServerFactory;
import org.apache.catalina.core.StandardServer;
import org.apache.commons.modeler.BaseModelMBean;
public class StandardServerMBean extends BaseModelMBean {
    private static MBeanServer mserver = MBeanUtils.createServer();
    public StandardServerMBean() throws MBeanException, RuntimeOperationsException {
      super();
    }
    public synchronized void stored throws InstanceNotFoundException, MBeanException,
      RuntimeOperationsException {
      Server server = ServerFactory.getServer();
      if (server instanceof StandardServer) {
        try {
          ((StandardServer) server).store();
        } catch (Exception e) {
          throw new MBeanException(e, "Error updating conf/server.xml");
        }
      }
    }
}
```

20.6.3 MBeanFactory

MBeanFactory 类表示一个工程对象，用于创建管理 tomcat 资源的 model Mbean。MBeanFactory 类还提供了删除 MBean 的方法。createStandardContext 方法用于创建托管资源 StandardContext 的 model MBean：

```
public String createStandardContext(String parent, String path, String docBase) throws Exception {

    // Create a new StandardContext instance

    StandardContext context = new StandardContext();

    path = getPathStr(path);

    context.setPath(path);

    context.setDocBase(docBase);

    ContextConfig contextConfig = new ContextConfig();

    context.addLifecycleListener(contextConfig);

    // Add the new instance to its parent component

    ObjectName pname = new ObjectName(parent);

    Server server = ServerFactory.getServer();

    Service service = server.findService(pname.getKeyProperty("service"));

    Engine engine = (Engine) service.getContainer();

    Host host = (Host) engine.findChild(pname.getKeyProperty("host"));

    // Add context to the host

    host.addChild(context);

    // Return the corresponding MBean name

    ManagedBean managed = registry.findManagedBean("StandardContext");

    ObjectName oname = MBeanUtils.createObjectName(managed.getDomain(), context);

    return (oname.toString());

}
```

20.6.4 MBeanUtil

org.apache.catalina.mbeans.MBeanUtil 类是一个工具类，提供了一些静态方法用于创建各种管理 Cataliana 对象的 MBean，还有一些静态方法用于删除 MBean，创建 ObjectName 等。例如，使用 createMBean 方法创建创建一个管理 org.apache.catalina.Server 的 MBean。createMBean 的实现方法如下：

```
public static ModelMBean createMBean(Server server) throws Exception {
    String mname = createManagedName(server);
    ManagedBean managed = registry.findManagedBean(mname);
    if (managed == null) {
      Exception e = new Exception( "ManagedBean is not found with "+mname);
      throw new MBeanException(e);
    }
    String domain = managed.getDomain();
    if (domain == null)
      domain = mserver.getDefaultDomain();
    ModelMBean mbean = managed.createMBean(server);
    ObjectName oname = createObjectName(domain, server);
    mserver.registerMBean(mbean, oname);
    return (mbean);
}
```

20.7 创建 Catalian 的 MBean

前面的章节说明了 catalina 中的一些 model MBean，现在来说明下这些 mbean 的创建。

在 server.xml 中定义了如下的监听器：

```
<Server port="8005" shutdown="SHUTDOWN" debug="0">
    <Listener c1assName="org.apache.catalina.mbeans.ServerLifecycleListener" debug="0"/>
    …
```

这里为 StandardServer 添加了一个监听器，org.apache.catalina.mbeans.ServerLifecycleListener。当 StandardServer 启动时，会触发 START_EVENT 事件：

```
public void start() throws LifecycleException {
    …
    lifecycle.fireLifecycleEvent(START_EVENT, null);
    …
}
```

而当 StandardServer 关闭时，会触发 STOP_EVENT 事件：

```
public void stop() throws LifecycleException {
    …
    lifecycle.fireLifecycleEvent(STOP_EVENT, null);
    …
}
```

这些事件会执行 ServerLifecycleListener 的 lifecycleEvent 方法，下面是 lifecycleEvent 方法的实现：

```
public void lifecycleEvent(LifecycleEvent event) {
    Lifecycle lifecycle = event.getLifecycle();
    if (Lifecycle.START_EVENT.equals(event.getType())) {
      if (lifecycle instanceof Server) {
        // Loading additional MBean descriptors
```

```
            loadMBeanDescriptors();
            createMBeans();
        }
    } else if (Lifecycle.STOP_EVENT.equals(event.getType())) {
        if (lifecycle instanceof Server) {
            destroyMBeans();
        }
    } else if (Context.RELOAD_EVENT.equals(event.getType())) {
        if (lifecycle instanceof StandardContext) {
            StandardContext context = (StandardContext)lifecycle;
            if (context.getPrivileged()) {
                context.getServletContext().setAttribute (Globals.MBEAN_REGISTRY_ATTR,
                    MBeanUtils.createRegistry());
                context.getServletContext().setAttribute (Globals.MBEAN_SERVER_ATTR,
                    MBeanUtils.createServer());
            }
        }
    }
}
```

其中，createMBeans 方法用于创建 MBean，该方法会先创建 MBeanFactory 实例，然后在创建相应的 MBean。ServerLifecycleListener 类中的 createMBeans 方法实现如下：

```
protected void createMBeans() {
    try {
```

```
            MBeanFactory factory = new MBeanFactory();
            createMBeans(factory);
            createMBeans(serverFactory.getserver());
        } catch (MBeanException t) {
            Exception e = t.getTargetException();
            if (e == null)
                e = t;
            log("createMBeans: MBeanException", e);
        } catch (Throwable t) {
            log("createMBeans: Throwable", t);
        }
    }
```

第一个 createMBeans 方法会使用 MBeanUtil 类为 MBeanFactory 创建一个 ObejctName，然后将其注册到 MBeanServer 中。第二个 createMBeans 方法获取一个 org.apache.catalina.Server 对象，为其创建 model MBean。下面的代码展示例如如何为 Server 对象创建一个 MBean：

```
    protected void createMBeans(Server server) throws Exception {
        // Create the MBean for the Server itself
        if (debug >= 2)
            log("Creating MBean for Server " + server);
        MBeanUtils.createMBean(server);
        if (server instanceof StandardServer) {
            ((StandardServer) server).addPropertyChangeListener(this);
        }
```

```
    // Create the MBeans for the global NamingResources (if any)
    NamingResources resources = server.getGlobalNamingResources();
    if (resources != null) {
      createMBeans(resources);
    }
    // Create the MBeans for each child Service
    Service services[] = server.findServices();
    for (int i = 0; i < services.length; i++) {
      // FIXME - Warp object hierarchy not currently supported
      if (services[i].getContainer().getClass().getName().equals
        ("org.apache.catalina.connector.warp.WarpEngine")) {
        if (debug >= 1) {
            log("Skipping MBean for Service " + services[i]);
        }
        continue;
      }
      createMBeans(services[i]);
    }
  }
```

注意，createMBeans 方法调用下面的语句

```
createMBeans(services[i]);
```

遍历所有的 StandardServer 对象中所有的 Service。该方法为每个 service 对象创建 MBean，然后，为每个 service 对象中所有的 connector 和 engine 对象调用 createMBeans 方法。

创建 service mbean 的 createMBeans 方法实现如下：

```
protected void createMBeans(Service service) throws Exception {
    // Create the MBean for the Service itself
    if (debug >= 2)
      log("Creating MBean for Service " + service);
    MBeanUtils.createMBean(service);
    if (service instanceof StandardService) {
      ((StandardService) service).addPropertyChangeListener(this);
    }
    // Create the MBeans for the corresponding Connectors
    Connector connectors[] = service.findConnectors();
    for (int j = 0; j < connectors.length; j++) {
      createMBeans(connectors[j]);
    }
    // Create the MBean for the associated Engine and friends
    Engine engine = (Engine) service.getContainer();
    if (engine != null) {
      createMBeans(engine);
    }
}
```

同样的，对每个 engine 对象，使用 createMBeans 方法为每个 host 创建 mbean：

```
protected void createMBeans(Engine engine) throws Exception {
    // Create the MBean for the Engine itself
    if (debug >= 2) {
```

```
        log("Creating MBean for Engine " + engine);
    }
    MBeanUtils.createMBean(engine);
    …
    Container hosts[] = engine.findChildren();
    for (int j = 0; j < hosts.length; j++) {
        createMBeans((Host) hosts[j]);
    }
    …
}
```

接着为每个 host，使用 createMBeans 方法，为每个 context 创建 mbean：

```
protected void createMBeans(Host host) throws Exception {
    …
    MBeanUtils.createMBean(host);
    …
    Container contexts[] = host.findChildren();
    for (int k = 0; k < contexts.length; k++) {
        createMBeans((Context) contexts[k]);
    }
    …
}
```

以此类推，createMBeans (context)方法实现如下：

```
protected void createMBeans(Context context) throws Exception {
    ...
    MBeanUtils.createMBean(context);
    ...
    context.addContainerListener(this);
    if (context instanceof StandardContext) {
      ((StandardContext) context).addPropertyChangeListener(this);
      ((StandardContext) context).addLifecycleListener(this);
    }
    // If the context is privileged, give a reference to it
    // in a servlet context attribute
    if (context.getPrivileged()) {
      context.getServletContext().setAttribute
        (Globals.MBEAN_REGISTRY_ATTR, MBeanUtils.createRegistry());
      context.getServletContext().setAttribute
        (Globals.MBEAN_SERVER_ATTR, MBeanUtils.createServer());
    }
    ...
}
```

如果 context 的 privileged 属性为 true，则会为 context 添加两个属性，两个属性的 key 分别是 Globals.MBEAN_REGISTRY_ATTR 和 Globals.MBEAN_SERVER_ATTR。下面是 org.apache.catalina.Globals 类的实现片段：

```
/**
```

```
    * The servlet context attribute under which the managed bean Registry
    * will be stored for privileged contexts (if enabled).
    */
public static final String MBEAN_REGISTRY_ATTR = "org.apache.catalina.Registry";
/**
    * The servlet context attribute under which the MBeanServer will be
    * stored for privileged contexts (if enabled).
    */
public static final String MBEAN_SERVER_ATTR = "org.apache.catalina.MBeanServer";
```

MBeanUtils.createRegistry 方法返回一个 Registry 实例，MBeanUtils.createServer 方法返回一个 javax.management.MBeanServer 实例，catalina 的 mbean 就注册于此。换句话说，当 privileged 属性为 true 时，你就可以从一个 web 应用中获取 Registry 和 MBeanServer 对象。

20.8 应用程序

该应用程序用来管理 tomcat，虽然简单，但说明了如何在 tomcat 中使用 mbean。你可以列出 catalina 中所有的 ObjectName 对象，以及当前正在运行的 context，并删除其中的任意一个。

首先要为 web 应用创建描述符，该描述文件需要放到%CATALINA_HOME%/webapps 目录下：

```
<Context path="/myadmin" docBase="../server/webapps/myadmin" debug="8"
    privileged="true" reloadable="true">
</Context>
```

注意，这里 privileged 属性的值必须为 true，docBase 指定了 web 应用的具体路径。该 web 应用中包含了一个 servlet，实现代码如下：

MyAdminServlet.java

下面编写一个 web·xml 文件：

```
<?xml version="1·0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
      PUBLIC "-//Sun Microsystems, Inc·//DTD Web Application 2·3//EN"
       "http://java·sun·com/dtd/web-app_2_3·dtd">
<web-app>
    <servlet>
      <servlet-name>myAdmin</servlet-name>
      <servlet-class>myadmin·MyAdminServlet</servlet-class>
    </servlet>
    <servlet-mapping>
      <servlet-name>myAdmin</servlet-name>
      <url-pattern>/myAdmin</url-pattern>
    </servlet-mapping>
</web-app>
```

执行后，要想列出所有的 ObjectName 对象，可以使用如下的 url：

http://localhost:8080/myadmin/myAdmin?action=listAllMBeans

列出所有的 web 应用可以使用如下的 url：

http://localhost:8080/myadmin/myAdmin?action=listAllContexts