Servlet2.3规范

第一章:

servlet2.3规范用到了一下的一些规范：J2EE、JSP1.1、JNDI
在14章中讲述了规范中的所有的classes类或接口（改文中不讲述）。对开发者而言以下的有些相关的协议：URI、URL、HTTP/1.0、MIME、HTCPCP/1.0、XML

1.1 什么是servlet?

servlet是一个基于java技术的web组件，该组件被容器管理，能被编译成字节码被web服务调用；容器也被称之为引擎，是支持servlet功能的web服务的扩展。servlet之间的通信是通过客户端请求被引擎执行成request/response对象进行的。

1.2 什么是servlet引擎?

servlet引擎是web服务器或应用服务器的一部分，服务器能够支持网络的请求/响应，基于请求解析MIME，基于响应格式化MIME。servlet引擎是一个servlet容器，也掌管着servlet的生命周期。

所有的servlet引擎都必须支持HTTP的请求/响应模式，但HTTPS的请求/响应模式也是被支持的。HTTP的版本最小要HTTP/1.0，最好是HTTP/1.1。servlet引擎也具有安全和权限的一些特性，这些特性其服务器应提供。

1.3 例子

一个典型的事件执行的顺序是:

1) 客户端向web服务器发起一个HTTP请求
2) HTTP请求被web服务器接受，并移交给servlet引擎，servlet引擎可以在主机的同一个进程、不同的进程或其他的web服务主机的进程中启动。
3) servlet引擎根据servlet的配置档确定调用的servlet，并把request对象、response对象传给它。
4) 4．servlet通过request对象知道客户端的使用者是谁，客户的请求信息是什么和其他的一些信息。servlet处理完请求后把要返回的信息放入response对象返回到客户端

5） 一旦servlet完成了请求的处理，servlet引擎就会刷新response，把控制权返回给web服务器

1.4与其它技术的比较

与其它服务相比servlet有以下的一些优点

1) 运行速度上比CGI快，因为使用了多线程
2) servlet使用了标准的api，可被许多web服务支持
3) 与系统无关性，一次编译多次使用

第二章

servlet接口是servlet api核心部分，所有的servlet都是直接或间接的实现了这些接口。两个最重要的servlet api 接口是GenericServlete 和 HttpServlet，更多的开发者都继承HttpServlet去实现他们的servlet

2.1 Request 包含的方法

一个基本的servlet接口应该定义一个方法包含客户端的信息，每次servlet引擎把一个request发送到一个servlet事例，这个方法都要被调用。

对于并发的请求，web应用需要设计者设计的servlet引擎能分配多个线程执行这个方法。

2.1.1 HTTP 请求处理的方法

HttpServlet是实现了Servlet接口的抽象类，增加了一些新的方法，这些方法在处理HTTP请求时会被service方法自动调用，这些方法是:

doGet 接受 HTTP 的GET请求
doPost 接受 HTTP 的POST请求
doPut 接受 HTTP的PUT请求
doDelete 接受 HTTP的DELETE请求
doHead 接受 接受 HTTP的HEAD请求
doOptions 接受 HTTP的OPTIONS请求
doTrace 接受 HTTP的TRACE请求

一个开发者只会涉及到doGet和doPost方法，其它的方法是为非常熟悉HTTP的设计师准备的

2.1.2

HTTP/1.0只定义了doGet，doHead，doPost方法，没有定义PUT，DELETE，OPTIOONS和TRACE方法

2.1.3

HttpServlet接口定义了getLastModified方法

2.2 实例数

2.2.1

在分布式环境中servlet引擎为每个servlet只能声明一个实例，当一个servlet实现了SingleThreadModel接口时，servlet引擎可以声明多个实例去处理请求，servlet在应用服务的部署描述中定义发布.

2.2.2单线程servlet

SingleThreadModel接口保证了在同一时刻一个servlet实例的service方法只会被一个线程执行。这对于每个请求发送给每个实例是很重要的。引擎可以从对象池中选择，对象池可以在同一时刻保持多个实例，如HttpSession可以被多个servlet在任何时候调用包括实现了SingleThreadModel接口的servlet

2.3 servlet的生命周期

一个好的生命周期的定义应该是servlet怎么被引入了，怎么实例化的，怎么初始化的？当请求从客户端来的时候，是怎么从服务器中取出来的，这些在javax.servlet.Servlet的接口的init，service，destroy方法中都有明确的定义。

所有的servlet都必须实现GenericServlet或HttpServlet抽象类

2.3.1 servlet的引入和实例化

servlet引擎会可靠的引入和实例化servlet，当servlet引擎被启动时servlet就被引入和实例化了，或者当一个servlet被请求时被引擎引入和实例化。

servlet引擎启动时，需要装载的类通过java的装载类进行装载，被装载的类可以在本地文件系统、远程文件系统或网络服务中。在装载完后，引擎就实例化它们。

2.3.2 初始化

在servlet对象实例化后，引擎必须在这个servlet接受客户段请求之前初始化，在初始化中servlet可以读取固定的配置信息，一些昂贵的资源如数据库连接和一次性激活的资源，引擎通过调用servlet接口的init方法初始化。每个serlet对象都实现了Servlet接口和ServletConfig接口，ServletConfig接口允许servlet接受web应用配置档中配置的参数，还向servlet中传入了一个描述servlelt运行环境的类（ServletContext）

2.3.2.1 在初始化时发生错误

在初始化过程中，servlet实例能抛出UnavailableException 或ServletException异常。在这样的情况下servlet不能被放入服务中，必须被引擎释放，destroy方法没有被调用。在初始化失败后

引擎可以在UnavailableException异常规定的最短无效时间后实例化新的一个实例，再初始化。

2.3.3 request

当servlet初始化完成后，引擎可以使用它去处理客户端的请求了。请求被封装在Servletrequest类型的对象中，响应信息被封装在ServletResponse类型的对象中，这两个对象以参数的形式传给Servlet接口中的service方法。

2.3.3.1 多线程问题

servlet引擎可以发送并发的请求给servlet的service方法，servlet开发者必须提供足够的线程来运行service方法。

对开发者来说一个可以选择的方法是实现SingleThreadModel接口，以确保在同一时刻只有一个请求在service方法中。一个引擎要确保请求能够在servlet中持续化，或维持在一个servlet实例池中，如果servlet是web应用服务的一部分，引擎可以在一个虚拟机中拥有一个servlet实例化的池。

对于没有实现SingleThreadModel接口的servlet，如果service方法（或 doGet，doPost）被声明为synchronized，引擎将不能用实例池的途径，而必须持续化请求，强力建议开发者不能声明synchronize service方法，这样会严重影响系统的性能。

2.3.3.2 request中的异常

在处理请求时servlet可以抛出ServletException或UnavailableException异常，一个ServletException异常会在处理一个请求出现错误时抛出，引擎将清除这个异常。当servlet一时或永久地不能获得一个请求时就会抛出UnavailableException异常,如果是一个永久性异常时引擎将调用它的destroy方法从服务器中消除servlet实例，如果是暂时性异常时引擎在异常期间不发送请求给servlet。如果返回SERVICE_UNAVAILABLE(503)响应，在这期间引擎将不接受任何请求，直到header中出现Retry-After。引擎可以不区分永久性还是暂时性的异常把所有的UnavailableException作为永久性处理。

2.3.3.3 线程安全

执行request和response对象不能保证是线程安全的，意思是说他们只能在请求的线程中使用，不能被其它线程中的对象使用。

2.3.4 结尾

servlet实例可能被引擎保存几毫秒或和引擎一样的生命时间或在这两者之间。当引擎决定结束一个servlet时，调用它的destroy方法，在destroy中释放任何长久固定的资源。

在引擎调用desroy方法之前，必须保证运行在service方法中的线程都完成处理，或者超过了服务定义的执行时间。

一旦servlet实例的destroy方法被调用，引擎不在发送任何请求给这个实例。如果引擎再次使用这个servlet就必须再建一个这个servlet的实例。

在destroy方法执行完成后，引擎将释放这个servlet实例，于是就符合垃圾回收机制的条件了。

3.1 介绍ServletContext接口

ServletContext接口定义了servlet运行环境的信息。引擎提供商有义务在servlet引擎中提供一个实现了ServletContext接口的对象。通过这个对象servlet能够获得log事件，资源的URL，设置或存储servlet之间通信的变量。ServletContext在web服务中确定了一个所有请求开始的路径，是ServletContext的上下文路径。

3.2 ServletContext 接口的作用范围

每个web应用配置到容器中都会产生一个实现了ServvletContext接口的实例。如果是分布式的，将会在每个java虚拟机上产生一个ServletContext实例。容器中默认固有的web应用（不是发布上来的web应用）有一个默认的ServletContext，在分布式中这个默认的ServletContext只存在于一个虚拟机中，是不可分配的。

3.3 初始化参数

ServletContext接口的getInitParameter,getInitParameterNames方法接受部署描述文件中的初始化参数，这些参数可以是的安装信息，或网站管理员的mail或名字或对系统的评论。

3.4 上下文属性

servlet可以通过一个名称把对象邦定到servletContext中，帮定到ServletContext中的对象都能被同一个web服务中的其它对象引用。ServletContext中的属性方法有:

setAttribute
getAttribute
getAttributeNames
removeAttribute

3.4.1 在分布式系统中 上下文的属性

上下文属性是在本地的虚拟机中保存的，这防止了ServletContext属性存在于分布式的内存中。当信息需要在一个分布式环境中共享的时候，信息应该被放在session中，或存在数据库中，或存在一个实体bean中。

3.5 资源

ServletContext接口提供了获取web服务中的静态资源的方法:

getResource
getResourceAsStream

这些方法以一个"/"作为上下文的根目录，后跟着资源路径的路径为参数。这些资源可以在本地服务系统中也可以在另个web应用中，或在一个远程的文件系统中。

这些方法不能用于去获得一个动态的资源，如要调用一个jsp页面，getResource("/index.jsp")将返回index.jsp的原代码，不能web显示index.jsp。

要获得资源列表可以用getResourcePaths(String path)方法

3.6 多主机 和 Servlet 上下文

web服务中可能在一个IP上有多个逻辑主机的情况。在这种情况下每个逻辑主机必须有自己单独的servlet上下文，或者设置多个servlet 上下文，但在逻辑主机中不能共享这些servlet 上下文，一个主机只能单独使用一个。

3.7

引擎提供类重新装载机制是必须的，必须确认应用中所有的类和接口都可以在单类装载器中装载；在session绑定监听事件中引擎会终止正在装在的类。以前版本的装载器，引擎创建一个新的装载器装载一个servlet或class和类装载器装载其他的servlet或类截然不同；这可能装载一个未知的类或对象，产生不可预知的行为。这是新的类装载器中是应该注意预防的问题。

3.7.1 临时工作目录

Servlet 上下文需要一个临时存储的目录。servlet引擎必须为每个servlet 上下文提供一个私有临时目录，通过javax.servlet.context.tempdir的context属性使目录有效。与该属性关联的对象必须是java.io.File类型。

在许多servlet引擎实现中提供请求可以识别的通用的机制。

当servlet引擎重起始不必维护临时目录中的内容，但要确保临时目录中的该上下文内容对于运行在该servlet引擎中的其它web应用中的servlet 上下文是不可见的。

4 requeset

request对象包含了客户端的所有请求信息。在HTTP协议中，客户端发送到服务端的信息都包含在请求的HTTP 头和消息体中

4.1 HTTP 协议的参数

客户端发送给servlet引擎的参数是包含在请求中的，引擎从客户端请求的URI字符串中或POST数据中解析出请求的参数。参数以name-value的形式存储的。任何一个name可以对应多个value。在ServletRequest接口的方法中：
getParameter
getParameterNames
getParameterValues
getParameterValues方法返回关联到一个name上的一个String对象的数组。getParameter返回name对应的values数组中的第一个value。URI和POST体中的参数都会放入请求参数的set对象中。URI中的参数会在POST体之前被引入，如URI中的参数"a=hello"post体中的参数是a=goodbye&a=world,则参数set中的内容是a=(hello,goodbye,world).以HTTP GET请求的参数是不隐蔽的，参数必须通过getRequestURI或getPathInfo方法获得参数字串。

4.1.1 参数什么时候有效

在post form中的数据参数被放入参数set之前的情况是这样的：

1）请求是一个HTTP或一个HTTPS
2）HTTP方法是POST
3）内容的类型是application/x-www-form-urlencoded
4）初始化过的servlet从request对象中调用getParameter方法（或getParameterNames，getParameterValues）。

Post form 中的数据符合条件的就放入参数set中，不符合的就放入request对象的输入流中。

4.2 属性

request的属性是一个对象，引擎可以把API不能表达的信息放入属性中，一个servlet也可以设置一个属性信息用于servlet之间的通信。request对象中的属性方法有：
getAttribute
getAttributeNames
setAttribute
一个属性名称只能关联一个value。属性名以"java."或"javax."为前缀的是规范保留的，类似的"sun."和"com.sun"是sun公司的保留字，这些保留的前缀是不能使用的。name建议使用统一的包命名规范名称。

4.3 头

servlet通过HttpServletRequest接口的方法获得HTTP的包头信息，这些方法是：
getHeader
getHeaders
getHeaderNames
getHeader 方法返回头的名称。一个名称可以关联多个头信息，如果在这种情况下，getHeader方法返回第一个头信息。
getHeaders返回与一个名称关联的所有头信息存放在Enumeration对象中。HttpServletRequest提供了一些提取头信息的类型转换方法，如：
getIntHeader 把头信息中的数据转换成int型，如果转换失败会报NumberFormatException错误。
getDateHeader 把头信息中日期的数据转换成date型，如果转换失败会报IllealArgumentException错误

4.4 请求路径

context路径：这路径是和ServletContext对象关联的，在web服务中默认的上下文路径是空的字

符串，如果上下文路径不是web服务的根目录，则路径以'/'字符开始，但不能以'/'结束。
Servlet 路径：与该请求匹配的servlet的路径。该路径以'/'字符开头，或以'/*'开头但后面为空字串。
路径信息：是请求路径的一部分，但不是context路径的一部分，也不是Servlet路径的一部分，它既不为null也不是以'/'开头的字符串。
上一路径在HttpServletRequest接口中对应的方法是：
getContextPath
getServletPath
getPathInfo
requestURI = contextPath + servletPath + pathInfo
上下文配置的例子：
Conteext Path /catalog
Servlet Mapping Pattern:/lawn/*
Servlet:LawnServlet
Servlet Mapping Pattern:/garden/*
Servlet:GardenServlet
Servlet Mapping Pattern:*.jsp
Servlet:JSPServlet
观察下面的路径
Request path path Elements
/catalog/lawn/index.htm ContextPath:/catalog
ServletPath:/lawn
PathInfo:/index.html
/catalog/garden/implements/
ContextPath:/catalog
ServletPath:/garden
PathInfo:/implements/
/catalog/help/feedback.jsp
ContextPath:/catalog
ServletPath:/help/feedback.jsp
PathInfo:null

4.5 路径转换

在API中有两个简单的方法允许开发者获得文件系统的路径：
ServletContext.getRealPath
HttpServletRequet.getPathTranslated
getRealPath(String aPath)方法返回本地文件系统的绝对路径。getPathTranslated方法计算出请求pathInfo中的绝对路径。
以上的两个方法，servlet引擎不能辨认文件的路径是否有效，当web应用调用一个不确定远程文件系统，或数据库路径中的文件时，会返回null

4.6 Cookies

HttpServletRequest接口中提供了getCookies方法返回请求中的cookies数组，在每次客户端请求时cookies数据就从客户端发送给服务。客户端返还的部分cookie信息是cookie的name和cookie的value。当cookie被送入浏览器时，cookie的其它信息就可以设置了。

4.7 SSL 属性

如果一个请求被转给一个安全的协议，如HTTPS，这些信息必须暴露给ServletRequest接口的

isSecure方法。web引擎必须把下面的信息暴露给servlet开发者:

Attribute	Attribute Name	javaType
Cipher suite	javax.servlet.request.cipher_suite	String
bit size of the algo-rithm	javax.servlet.request.key_size	Integer

如果一个SSL证书伴随着一个请求，servlet引擎必须把它作为一个数组对象暴露给servlet开发者，该数组中有
java.security.cert.X509Certificate对象和放在ServletRequest属性中的javax.servlet.request.X509Certificate对象。
数组排列的顺序是升序，在链中的证书的顺序就是客户端设置的顺序。

4.8 国际化

ServletRequest接口的方法中提供了的方法:
getLocale
getLocales
getLocale方法将返回客户端将从中获得内容的首选的locale。要想知道更多的关于Accept-Language header 怎么解释客户端首选的语言的，请看14.4章
getLocales方法返回一个Locale objects的Enumeration,从首选的locale开始递减。
如果客户端没有制定首选的locale，servlet引擎一定要提供一个默认的locale供getLocale方法返回，getLocales方法必须包含一个默认的locale的locale element

4.9 Request 数据的编码

有许多web浏览器不能发送一个编码的头内容，所以把编码留给解读HTTP请求的Read去做。对于默认的请求编码，引擎通常创建一个reader用"ISO-8859-1"去解析POST的数据，如果客户端没有指明编码，或者客户端发送失败，getCharacterEncoding方法就返回null。
如果客户端没有设置编码，而请求需被另外一种编码，可用ServletRequest接口中的setCharacterEncoding(String enc) 方法。必须在解析post数据或读取请求流之前调用这些方法。

4.10 Request对象的生命周期

每个request对象仅在servlet的service方法或filter中的doFilter方法中有效，引擎重用request对象是为了降低创建request对象的性能消耗。
开发者必须清楚request对象在给定的范围外的一些不确定的行为。

第五章

response对象封装着服务端送给客户端的信息，从服务端传回的信息可以包含在请求的头和消息体重。

5.1 缓存

servlet引擎支持应答缓存，典型的servlet会默认的执行缓存，servlet可以指定缓存参数。
设置缓存信息的方法在ServletResponse接口中的方法有:

getBufferSize
setBufferSize
isCommitted
Reset
resetBuffer
flushBuffer

这些方法只有在servlet调用ServletOutputStream 或Writer之前有效。
getBufferSize返回缓存的大小，如果没有缓存，该方法返回0。
setBufferSize可以设置缓存的大小，但不是必须的。servlet会根据请求放置适当的缓存大小。这方

法必须在servlet调用ServletOutputStream 或Writer方法之前被调用。如果在之后调用就会抛出IllegalStateException错误。
isCommitted返回一个boolen值，标志是否有任何一个字节的数据被返回给客户端了。flushBuffer方法强制把缓存中的信息写到客户端。
当response没有提交缓存内容时调用reset方法就会清除缓存中的信息，包括头信息和状态码。
resetBuffer方法会清除缓存中的信息，但不会清除头和状态码。在commit之后调用reset或resetBuffer都会抛出IllegalStateException错误，缓存中的内容不受影响。
使用缓存时，当缓存满时response就必须立刻刷新把缓存中的内容发送给客户端；只要第一个字节到了客户端，commit的状态就为true。

5.2 Headers

servlet能够通过HttpServletResponse的一些方法设置HTTP响应的头信息，这些方法有:
setHeader
addHeader
setHeader方法会把给定的一个name-values放到头信息中，头信息中只能有一个name-values，后面setHeader会覆盖前面setHeader方法中的内容，一个name可以有多个value。
addHeader方法可以增加一个value到已有的name上，如果name不同就会新建一个name-values
头可以包含一些信息，如日期或数字对象。
以下的一些方法用适当的数据类型设置头信息:
setIntHeader
setDateHeader
addIntHeader
addDateHeader
在响应被发送到客户端之前，头信息是必须被设置的，如果没有设置头信息，servlet引擎将不会发送该请求到客户端。HTTP1.1规范没有规定必须设置响应的头信息。当程序员没有设置响应体的Content-Type时，servlet引擎也不需要设置一个默认的类型。

5.3 其他一些方法

HttpServletResonse接口中还有其他的一些方法:
sendRedirect
sendError
sendRedirect方法将设置合适的头和体信息，用于重定向客户端到另一个URL。如果sendRedirect参数是个相对路径，则在底层servlet引擎中会把这相对路径转换成绝对路径返回给客户端的。
如果相对路径不能被引擎转换成绝对路径就会抛出IllegalArgumentException错误。
sendError方法会把一条错误信息作为头和体信息发送给客户端。
如果在调用sendRedirect或sendError之前设置了头和体信息，再调用sendRedirect或sendError时，之前的头和体中的数据信息都将没用，不会被发送到客户端。如果使用了缓存，在调用sendRedirect或sendError时，之前的信息都将被清除。如果在commit之后调用sendRedirect或sendError就会抛出IllegalStateException错误。

5.4 国际化

当客户端用一特殊的语言（或客户端设置了语言）发出请求时，servlet会设置相应的响应语言信息，ServletResponse接口中设置响应语言的方法是setLocale。这个方法会设置一个合适的Content-Language到头信息中。最好是开发者在调用getWriter方法之前调用setLocale方法，确保返回的PrintWriter已经被设置好了语言信息。如果在调用setLocale之后又调用了setContentType，setLocale中的内容将被setContentType中的字符集覆盖。
response默认的编码方式是"ISO-8859-1"。

5.5 response对象的关闭

当response被关闭时，引擎必须刷新该response缓存中的所有内容到客户端。关闭的顺序是:

1）关闭servlet的service方法
2）response 中setContentLength方法设置的指定数量的信息被写入response
3）调用sendError方法
4）调用sendRedirect方法

5.6 response对象的生命周期

每个response对象仅在servlet的serrvice方法或filter的doFilter的方法中有效。引擎重复使用reponse对象，是为了降低创建response对象的开销。开发者必须注意response对象在指定范围外可能出现的一些意外的行为。

6 Filtering

Fileter是servlet2.3新增的部分。这一章介绍Filter类和方法，以及在web工程中的配置。

6.1什么是Fileter

Filter是重复使用的，用于变换HTTP请求和响应以及头信息中的内容。Filter不能像servlet那样创建response响应，但可以修改请求和响应的内容。

6.1.1例举一些Filter

验证Filter
登陆，审核Filter
图像处理Filter
数据压缩Filter
加密Filter
XSL/T Filter
MIME Filter

6.2 主要观念

开发者通过创建实现javax.servlet.Filter接口的类，并提供一个没有参数的构造函数来创建一个Filter。
在描述文件中Fileter用filter表示，调用方法用filter-mapping进行配置。

6.3 Filter生命周期

在web工程发布后，在请求使引擎访问一个web资源之前，引擎必须定位Filter列表；引擎必须确保为列表中的每一个Filter建立了一个实例，并调用了他们的init(FilterConfig config)方法。在这过程中可以抛出异常。
部署描述文件中定义的所有filter，仅会在引擎中产生一个实例。
引擎为filter提供了一个FilterConfig类，该类附有ServletContext和一个带有初始化参数的set。
当引擎接受一个请求时，引擎就会调用filter列表中第一个filter的doFilter方法，把ServletRequest，ServletResponse和FilterChain作为参数传给它。
filter中doFilter方法典型的处理步骤是:

1）检查请求头信息
2）开发者创建一个实现了ServletRequest或HttpServletRequest的类，去包装request对象，以便修改请求的头信息或体数据。
3）开发者创建一个实现了ServletReqponse或HttpServletResponse的类，去包装response对象，以便修改请求的头信息或体数据。
4)filter可以调用链中的下一个实体，下一个实体是另一个filter，如果该filter是列表中最后的一个，则它的下一个实体就是一个目标web资源。如果要调用下一个filter的doFilter方法，把request，和response对象传给FilterChain对象的doFilter方法中就可以了。
Filter chain 的doFilter方法是由引擎提供的，引擎在该方法中会定位filter列表中的下一个filter，

调用它的doFilter方法，把传来的request和response对象传给它。

5）在调用chain.doFilter之后，filter可以检测响应的头信息

6）在这些过程中，filter可以抛出异常。当在调用doFilter过程中抛出UnavailableException异常时，引擎重复尝试处理下面的filter chain的方法，如过时后还没请求到filter chain 就会关闭对filter chain的请求。

当filter是列表中最后一个filter时，它的下一个实体是描述配置文件中filter后面的servlet或其它资源。

在引擎删除一个Filter之前，引擎必须调用Filter的destroy方法，来释放资源。

6.3.1 包装Requests 和Responsees

过滤的中心观念是对request或response的包装，在这种模式下，开发者不仅可以改写存在的方法，还可以创建自己的新方法，用于特殊的过滤任务，例如：开发者希望扩展response对象，希望有个更高层次的输出流对象（writer）。

为了支持包装模式，引擎不许保证在整个过滤链中，传递的request和response对象都是同一个对象。

6.3.2 Filter的环境

Filter的初始参数可以在描述配置文件中用init-params元素来配置，在运行时中，用FilterConfig的getInitParameter和getInitParamesterNames方法得到配置参数。

6.3.3 Filter在web工程中的配置

在部署描述文件中:

filter-name:filter名称

filter-class:filter类路径

init-params:用于初始化参数

如果开发者在部署描述中为一个filter类描述了两个定义，则引擎会创建这个filter类的两个实例。

下面是个配置的例子:

```
<filter>
        <filter-name>Image Filter</filter-name>
        <filter-class>com.acme.ImageServlet</fiflter-class>
</filter>
```

一旦filter在部署描述中定义，filter-mapping就可以被定义了，filter-mapping在web应用中是定义关联filter的servlet和静态资源的。

如:

```
<filter-mapping>
        <filter-name>Image Filter</filter-name>
        <servlet-name>ImageServlet</servlet-name>
</filter-mapping>
```

Image Filter 的 Filter就和ImageServlet 的Servlet建立了关联。

Filter 可以和一群servlet和静态资源关联，用url-pattern。如:

```
<filter-mapping>
        <filter-name>Loging Filter</filter-name>
        <url-pattern>/*</url-pattern>
<filter-mapping>
```

引擎建立特殊请求URI的Filter链的顺序是:

1)url-pattern映射fiter-mapping的顺序和描述文件中定义的顺序是一样的。

2）servlet-name映射filter-mapping的顺序和描述文件中定义的顺序是一样的。

这种需求要求引擎在接受请求时:

.识别符合SRV.11.2规则的web资源。

.如果一些filter是servlet和有servlet-name的web资源匹配的，引擎就会创建一个和描述文件中映射servlet-name的顺序一样的filter链。
.如果一些filter是rul-pattern关联的，引擎就会创建一个和描述文件中映射url-pattern的顺序一样的efilter链。
一个高性能的web容器将会缓存filter链。

第七章 **Sessions**

超文本传输协议（HTTP）是无状态的协议。要建立一个有效的web应用，客户端之间的通信是需要的。有很多会话跟踪的策略，
但是直接使用这些技术都很难使用。servlet规范中提供了一个简单的HttpSession接口，不需要开发者关心会话跟踪的具体细节。

7.1 会话跟踪机制

下面描述了几种会话的跟踪机制

7.1.1 Cookies

HTTP cookies是最常用的会话跟踪机制，所有的servlet引擎都应该支持这种方法。
引擎发送一个cookie到客户端，客户端就会在以后的请求中把这个cookie返回给服务器。用户会话跟踪的cookie的名字必须是JSESSIONID

7.1.2 SSL Sessions

在安全套接字层，加密技术用在了HTTPS协议，从一个客户端来的多个请求允许用一个含糊的标识，servlet引擎就用这个数据定义一个Session。

7.1.3 URL重写

URL重写是最低性能的通用会话跟踪方法。当一个客户端不能接受cookie时，URL重写就会作为基本的会话跟踪方法；URL重写包括一个附加的数据，一个session id，这样的URL会被引擎解析和一个session相关联。一个session id是作为URL的一个被编码的参数传输的，这个参数名字必须是jsessionid.如下面的例子：

http://www.myserver.com/catalog/index.html;jsessionid=1234

7.1.4 会话的完整性

一个web容器必须支持HTTP 会话。而当cookies方法不被支持时，通常使用URL重写方法。

7.2 创建一个会话

servlet设计者必须考虑到一个客户端不能加入session的情况。

7.3 会话范围

HttpSession对象只在应用程序级有效，通常用于session的cookie可以服务于不同的上下文，但一个HttpSession实例只服务于一个会话。举个例子：如一个servlet A用RequestDispatcher去调用另一个web应用中的另一个servlet B，用于A和B的会话一定是两个不同的会话。

7.4 Session属性的邦定

一个servlet可以通过一个name邦定一个对象到HttpSession实例中；只要获得包含同一个会话的请求对象，任何邦定到会话中的对象在同一个ServletContext中对于其它的servlet都是可用的。
当把一个对象放入session或从session删除时可能要通知其它对象，这些信息能够被实现了HttpSessionBindingListener接口的对象获得，这个接口定义了一下的一些方法。
valueBound
valueUnbound
valueBound方法在HttpSession接口调用getAttribute方法获得一个有效的对象之前调用。valueUnbound方法在HttpSession接口调用getAttribute方法获得一个不再有效的对象后调用。

7.5 会话超时

在HTTP协议中，当客户端不再有效时，没有清楚的定义终止信号。这就意味着通常只能采用时间超时来表明客户端不再有效。
默认的超时时间是servlet引擎定义的，通过HttpSession的getMaxInactiveInterval方法可以得到超时的时间；开发者可用用setMaxInactiveInterval方法来设置超时的时间，以秒定义的。如果一个session的超时时间被设置为-1，则这个session将永远有效。

7.6 最后访问时间

在当前的请求中用HttpSession接口的getLastAccessedTime可以获得最后一次访问session的时间。

7.7 重要session

7.7.1 线程问题

在一个可以配置的应用中，所有的请求都是一个会话的一部分，引擎一定能够取出通过setAttribute或putValue放入HttpSession对象中的对象。注意以下的情况：
.引擎一定能够访问实现了Serializable接口的对象。
.引擎可以选择存储HttpSession对象中指定的对象，如EJB组件和事务。
.引擎能够监听到会话的变动。
如果放入session中的对象没有被Seializable或没有效，servlet可以抛出IllegalArgumentException；如果引擎不支持Session存储对象的机制，引擎一定会抛出IllegalArgumentException。
这些限制意味着，在一个分布式引擎中，不会有额外的并发问题。
如果引擎为了service的品质持续化或迁移session，使用本地持续化的HttpSession或它的属性是不受限制的，开发者要想确保放入session中的属性对象能够可用，最好对象实现Serializable接口。
在迁移一个session时引擎必须通知session中实现了HttpSessinActivationListener的属性对象，必须通知在序列化前钝化的或序列化后激活的session的监听器。
开发分布式的开发者应该清楚的是，一旦引擎运行在超过一个JVM的时候，就不能用static 表明变量来存储应用状态，应该用EJB或数据库赖存储。

7.7.3客户端

因为cookies或SSL证书都是被web浏览器访问过程控制的，与任何特殊的window浏览器是没有关系的。所以从所有window客户端到一个servlet引擎的请求是同一个会话的一部分。最好是开发者总是设想所有的window客户端是一起参与同一个会话的。

第八章 **Dispatching Requests**

当建立一个web应用时，把一个请求传给另一个servlet或在response中包含另一个servlet的输出是经常使用的。RequestDispatcher接口就提供了一些方法。

8.1 获得RequestDispatcher

实现了接口RequesetDispatcher接口的对象可以在ServletContext的getRequestDispatcher或getNamedDispatchcer方法得到。
getRequestDispatcher的参数是一个以根目录'/'开始的一个路径，该方法会查找路径下的servlet，并把它封装成RequestDispatchcer对象返回。
getNamedDispatcher方法把一个ServletContext知道的servlet名字作为参数，如果找到servlet，该servlet就被封装成RequestDispatcher对象返回，如果没有找到则返回null。
RequestDispatcher对象中使用相对路径也是可以的。在ServletRequest中提供了getRequestDispatcher方法；这个方法和ServletContext中同名的方法功能类似。servlet引擎会用request的信息把相对路径转化成完整路径的。如ServleItRequest.getRequestDispatcher("header.html")和ServletConext.getRequestDispatcher("/garden/headere.html")是等效的。

8.1.1 在Request Dispatcher 路径中附加字符串

在ServletContext和ServletRequest创建RequestDispatcher方法中参数都可以带字符串如:
Context.getRequestDispatcher("/raisons.jsp?orderno=5");

8.2 Request Dispatcher的使用

对于使用Request Dispatcher 而言就是一个servlet调用include或forward方法，这些方法的参数是Servlet接口传来的request和response对象实例。引擎必须确保调用Request Dispatcher的处理过程是在同一个JVM的同一个线程中。

8.3 include 方法

RequestDispatcher接口的include方法可以在任何时候被调用；目标servlet可以包含所有外的request对象，不过response对象的使用是有限制的:
response只能写信息到ServletOutputStream 或者Writer中，调用response对象的flushBuffer方法进行提交。不能够设置头信息，任何方法都不会影响到response的头信息。

8.3.1 被包含的request参数

除了可以用getNamedDispatcher方法包含一个servlet外，以下的属性可以被设置:

- Java.servlet.include.request_uri
- Java.servlet.include.context_path
- Java.servlet.include.servlet_path
- Java.servlet.include.path_info
- Java.servlet.include.query_string

用request对象的getAttribute方法可以获取被包含servlet的以上属性。
如果被包含的servlet能后通过getNamedDispatcher方法找到就不必设置以上属性了。

8.4 Forward 方法

RequestDispatcher接口中的forward方法，只有servlet没有提交响应到客户端时才可用；如果响应buffer中有数据还没有提交，当调用forward方法中目标servlet的service方法前，buffer中的内容会被清空；如果buffer中的数据提交了，则发生IllegalStateException错误。
request对象的路径必须放映获取RequestDispatcher对象的路径。
有个例外，如果RequestDispatcher是通过getNamedDispatcher方法得到的，request对象必须反映原始request的路径。
在RequestDispatcher接口方法forward返回前，response的内容必须被提交，并由引擎关闭该servlet。

8.4.1 query String

在Request Dispatcher中创建的路径是可以带参数的。

8.5 错误

如果request Dispatcher的目标servlet抛出运行时错误或ServletException 或IOException，错误就会被传给调用的servlet；在上传之到调用的servlet之前，所有其他的exception都应该包装成ServletExceptions。

第九章 **web** 应用

一个web应用是一堆servlet，html页面，类和其他资源的集合。web应用可以被发布运行在很多服务提供商的多种引擎下。

9.1 web服务器

在web服务中一个web应用的根目录是一个特殊的路径，例如：一个网站目录可以以http://www.mycorp.com/登录，所有的请求都将以这个作为前缀发送到以这个前缀描述的servletContext环境中。
在任何时候一个web应用的实例只能运行在一个JVM中。

9.2 和servletContext的关系

servlet引擎会强迫web应用和ServletContext的通信，一个ServletContext对象提供了一个servlet使得该

应用可见。

9.3 web应用中的元素

一个web应用包含以下的元素:

.Servlets
.JSP
.Utility Classes
.Static documents(html,images,sounds,etc)
.Client side Java applets,beans,and classes
.Descriptive meta informateion

9.4 部署层次

这个协议定义了一个层次结构，用于部署和打包，这个结构存在于一个文件中。

9.5 目录结构

一个web应用存在一个目录层次结构。文件根目录是应用的一部分。例如：一个web应用的上下文路径是/catalog，web应用的index.html文件就能被/catalog/index.html请求访问。URL和上下文路径的匹配规则将在11章讨论。servlet引擎必须拒绝一个具有现在冲突的上下文路径的web应用，这种情况是有的，如：两个web应用发布在同一个上下文路径中，或一个web应用的上下文路径是另一个web应用上下文路径的子路径。

有一个特殊的目录（"WEB-INF"）在应用中存在，这个目录包含所有与应用相关，又不在根目录中的事物。可以直接被引擎提供给客户端的文件不放在WEB-INF中，但WEB-INF目录对于调用ServletContext的getResource和getResourceAsStream方法的servlet 代码是有效的。如果开发者想用servlet代码调用一个不希望暴露给客户端的一个配置信息，就可以把这个配置信息放在WEB-INF目录下。请求都是和资源相匹配的；敏感的匹配如客户端的请求是"/WEB-INF/foo"和"/Web-iNf/foo"，但不应该把定位于/WEB-INF下的内容作为结果返回。

WEB-INF目录下的内容有:

./WEB-INF/web.xml 部署描述文件
./WEB-INF/classes/ 存放servlet class
./WEB-INF/lib/*.jar 是jar包的目录

应用的classloader先load WEB-INF/classes目录下的class后load WEB-INF/lib目录下的jar包

9.5.1 目录结构的一个例子

一个简单web应用的目录结构:

/index.html
/howto.jsp
/images/banner.gif
/images/jumping.gif
/WEB-INF/web.xml
/WEB-INF/lib/jspbean.jar
/WEB-INF/classes/com/mycorp/servlets/MyServlet.class
/WEB-INF/classes/com/util/MyUtils.class

9.6 web应用的存档文件

一个web应用可以被java打包工具打包成war文件，当被打包后包中就会有一个额外META-INF目录，该目录下存放了打包工具的一些信息。

9.7 web应用部署描述

下面是web应用部署描述中的配置类型:

.ServletContext Init Parameters
.Session Configuration
.Servlet / JSP Definitions
.Servlet / JSP Mappings
.MIME Type Mappings
.Welcome File list
.Error Pages
.Security

9.7.1 可靠的扩展

web容器须提供一种机制使得web应用知道jar文件中包含的有用资源或代码。
引擎因该提供编辑、配置库文件的程序。
在WAR中提供一个MANIFEST.MF文件，描述扩展名列表是比较好的。标准的JAR是应该有的，这个文件描述的扩展名应该遵循Http://java.sun.com/j2se/1.3/docs/guide/extensions/versioning.html中的规定。web容器应该能够识别WEB-INF/lib文件夹中的任何文件的扩展名，如果不能够识别就应该拒绝该应用程序，并报出错误。

9.7.2 web应用的classloader

引擎用于装载war中的servlet的装载器必须能够让开发者装载jar库中的任何资源。但装载的资源禁止覆盖j2se或servlet API中的类；通常建议的做法是装载器不允许war中的servlet去访问web引擎中的类。
还有一个被推荐的做法是实现应用类装载器，war中被装载的类或资源就会被放到container-wide JAR库的特定类或资源中。

9.8 替换web应用

一个服务器可能会在不重新启动引擎的情况下用一个新版本的应用替换原有的应用。当一个应用被替换时，引擎应提供一个robust方法去保存该应用中的session

9.9 错误句柄

9.9.1 request Attributes

web应用必须列出在使用中资源发生的错误，这些资源在部署描述中都有定义。
如果错误在一个servlet或一个jsp页面中发生，则在第9.1章中的如下的请求属性就会被设置:

Request Attributes	Type
Javax.servlet.error.status_code	java.lang.Integer
Javax.servlet.error.exception_type	java.lang.Class
Javax.servlet.error.message	java.lang.String
Javax.servlet.error.exception	java.lang.Throwable
Javax.servlet.error.request_uri	java.lang.String
Javax.servlet.error.servlet_name	java.lang.String

这些属性允许这个servlet根据这些状态码、错误类型、错误信息、被抛出的错误对象、错误产生的servlet被访问的URI（可以用getRequestURI得到）、或错误产生的servlet的逻辑名称产生特殊的内容。
在2.3版本中错误类型和错误信息属性是多余的，他们被保留只是为了向下兼容以前的版本。

9.9.2 错误页面

当一个servlet产生错误时，开发者可以订制错误内容返回给客户端。部署描述文件定义了一个错误页面列表。servlet在response中设置错误状态码或产生的异常或错误被引擎支持时，引擎就会从部署描述文件中调用相应的配置的错误资源。
如果一个错误码被设置在了response中，引擎在部署描述文件的错误页面列表中用status-code方式匹配对应的资源，如果找到就调用本地的资源。

在一个请求被处理的过程中servlet可以抛出以下的异常:
.runtime exceptions or errors
.ServletExceptions or subclasses thereof
.IOException or subclasses thereof
web应用可以用exception-type元素来描述错误页面，在这种情况下引擎会通过比较用exception-type元素定义的error-page列表中的异常来匹配产生的异常。匹配的结果是返回定义的与错误匹配的本地资源。在继承类中，最近的类将被调用。
如果没有一个error-page包含的exception-type与class-heirarchy相匹配。抛出的ServletException或其子类异常，被引擎通过ServletException.getRootCause方法获得，获得后用这个异常再去配置的error page列表中去匹配。
在部署描述文件中用exception-type元素定义的Error-page中exception-type的类名必须是唯一的。
当错误发生在servlet调用的RequestDispatcher中时error page机制是不能够干预到的；这样的情况如:一个servlet用RequestDispatcher去调用另一个有错误的servlet。
如果一个servlet产生的错误没有被描述的错误页面机制所抓到，引擎必须设置response的状态码为500

9.10 Welcome Files

web应用可以在部署描述文件中定义一个welcome files调用的URI列表，这个机制的目的是允许开发者定义自己的访问首页。
如果没有在部署描述中配置welcome 文件,引擎将把局部请求（没有指明具体访问资源，如www.cacolg.com/index.html,请求访问时用www.cacolg.com/访问的）发送到适当的资源中，如：一个可能默认的servlet，或列出该目录下的文件列表，或返回404响应错误。
一个例子:
1）在部署描述中定义index.html和default.jsp为welcome files
2）定义一个servlet的mapping路径为/foo/
WAR中有的文件如下:
/foo/index.html
/foo/default.html
/foo/orderform.html
/foo/home.gif
/catalog/default.jsp
/catalog/products/shop.jsp
/catalog/products/register.jsp

3）请求的URI为	处理后的URI
/foo 或 /foo/	/foo/index.html
/catalog/	/catalog/default.jsp
/catalog/index.html	404 not found
/catalog/products/	404 not found 也可能返回shop.jsp and /or register.jsp 列表。

9.11 web应用环境

j2EE定义的命名环境能够使得应用在不需要知道外部信息怎么命名的情况下比较方便的访问资源或外部信息。
servlet作为j2EE完整的一部分，使web应用部署描述文件提供了一个servlet可以访问资源和EJB，这些部署描述元素有:
.env-entry
.ejb-ref
.ejb-local-ref

.resource-ref
.resource-env-ref
开发者使用这些元素描述web应用中需要用到的对象，这些对象都要在web容器运行时注册到JNDI命名空间。
在J2EE1.3版本j2EE的环境需求中，servlet引擎不是J2EE技术的一部分，web环境要提供的功能在J2EE规范中有描述。如果没有实现支持环境所要提供的功能，在发布应用时，web容器就会抛出警告。
实现servlet引擎在J2EE中是需要的，应该被纳入J2EE1.3中。J2EE1.3应该提供更多的内容。
servlet引擎必须支持对象的lookup方法，查找对象并在引擎控制的线程中实例化。
servlet引擎应该支持开发者创建的线程，因为应用创建的线程不是很轻便，开开发者不得不依赖于这些功能有限的线程。这些需求将被加入到下一个版本的servlet规范中。

第十章 应用周期事件

10.1 介绍

事件是servlet2.3种新添的内容。应用事件使得web开发者能够控制ServletContext和HttpSession对象的信息交互，使得管理web使用的资源更有效，方便。

10.2 事件监听器

事件监听器是实现了servlet事件监听接口的类。在web发布是这些监听类就被实例化和注册在web容器中。
servlet事件监听器提供了在ServletContext和HttpSerssion对象状态发生改变时触发的事件。Servlet cotext监听器用于管理应用的资源或虚拟机的状态。HTTP session监听器管理与会话关联的资源。
可以有多个监听器监听每一个事件类型。开发者可以指定引擎调用监听类的顺序。

10.2.1 事件类型和监听接口

Event Type	ListenerInterface	说明
Lifecycle	javax.servlet.ServletContextListener	当servlet context被创建并有效的接受第一个请求 或servlet context销毁前
Changes to attributees	javax.servlet.ServletContextAttributesListener	当servlet context中的属性发生added,removed,replaced
Lifecycle	javax.servlet.http.HttpSessionListener	当HttpSession被创建，无效或超时
Changes to attributes	javax.servlet.HttpSessionAttributesListener	当属性added,removed或replaced时

10.2.2 一个使用监听的例子

一个简单的web应用中有servlet要访问数据库，开发者提供一个context 监听类管理数据库连接。
1）web应用启动时，监听类被装载，登陆数据库，在servlet context中保存数据库连接。
2）servlet访问数据库连接
3）当web服务销毁时，或应用从web服务中删除时，关闭数据库连接。

10.3 监听类的配置

10.3.1 对监听类的规定

web开发者提供实现了以上监听接口的类，每个类应该有一个没有参数的构造器函数。监听类放在WEB-INF/classes下或以一个jar文件放在WEB-INF/lib下都可以。

10.3.2 部署描述

web容器对每个监听类只会创建一个实例，在第一个请求到来之前实例化并注册。web容器注册监听类

的顺序根据他们实现的接口和在部署描述文件中定义的顺序。web应用调用监听实例的顺序按照他们注册的顺序。

10.3.4 在销毁时的事件

当应用销毁时监听事件的执行顺序按部署描述中的顺序，先执行session中的监听事件再执行context中的监听事件。session的无效事件必须在context的销毁事件之前被调用。

10.4 部署描述的例子

下面给出注册两个servlet cocntext lifecycle监听器和一个HttpSession监听器的例子。
Com.acme.MyconnectionManager和com.acme.MyLoggingMoudule都实现了javax.servletServletContextListener接口,com.acme.MyloggingModule另外还实现了javax.servlet.HttpSessionListener接口。开发者希望com.acme.MyConnectionManager在com.acme.MyLoggingModule之前管理者servlet context 的生命周期事件。部署描述文件如下:

```
<web-app>
 <display-name>MyListeningApplication</display-name>
 <listener>
    <listener-class>com.acme.MyConnectionManager</listener-class>
 </listenrer>
 <listenrer>
    <listenrer-class>com.acme.MyLoggingModele</listener-class>
 </listener>
 <sevlet>
    <display-name>RegistrationServlet</display-name>
    ..etc
 </servlet>
</web-app>
```

10.5 监听器的实例和线程

在第一个请求被web容器接受之前实例化并注册好监听器类是必须的。监听器在整个web应用生命周期中都要使用。
ServletContext和HttpSession对象属性的改变可能会同时产生，引擎不需要同步这些属性类的事件。

10.6 分布式容器组

在分布式web容器组中，HttpSession和ServletContext实例只活动与它们本地的JVM中。在分布式web容器中，监听实例会在每一个web容器中创建实例。

10.7 session事件

监听器使得开发者可以跟踪web应用中的session。知道session是否变得无效是经常被用到的，因为session超时时引擎会使session变得无效，或应用会调用invalidate方法。

第十一章 请求到**servlet**的映射

11.1 URI的使用

web容器根据客户端的请求决定要调用的资源。
URL路径映射规则是第一个匹配成功就不再匹配了。
1）引擎将尽力为每一个请求一个servlet的路径匹配一个servlet
2）引擎将递归的匹配最长的路径前缀（在一个目录树中）
3）如果在URL路径中的最后一节有扩展名（例如：.jsp），则servlet引擎将会匹配一个适当的servlet获取请求对象
4）如果没有一个servlet能够匹配请求，引擎将用一个适当的资源来处理该请求。如：在应用中配置了

默认的servlet，就会被用来处理匹配不到资源的请求。

11.2 匹配规则

在web应用描述文件中，匹配的定义如下：

.以'/'开始，以'/*'为结尾的字符串,用作路径匹配

.以'*.'开始的字符串，用作扩展名匹配

.包含'/'字符串，定义一个默认的servlet。如匹配的servlet路径是请求URI路径的最小上下文路径，路经的info为空。

.其它的字符串用作精确的匹配。

11.2.1 绝对匹配

servlet引擎能够匹配任何精确的资源，如后缀为*.jsp的匹配，引擎中有JSP引擎的话，就会把所有的JSP页面和与之对象的资源相匹配。

11.2.2 匹配的例子：

path pattern	servlet
/foo/bar/*	servlet1
/baz/*	servlet2
/catalog	servlet3
*.bop	servlet4

incoming path	servlet handling request
/foo/bar/index.html	servlet1
/foo/bar/index.bop	servlet1
/baz	servlet2
/baz/index.html	servlet2
/catalog	servlet3
/catalog/racecar.bop	servlet4
/index.bop	servlet4

第十二章 安全

12.1介绍

web应用的资源能够被很多的用户访问，这些资源经常没有保护的暴露在网络中，因此一个稳定的web应用需要一个安全的环境。

提高安全性有以下的几个方面：

.认证：访问一个实例前需要一个特殊的ID进行授权认证后才能访问该实例

.资源的访问控制：一些机密资源或局部资源只限制给某些用户或程序使用。

.数据完整性：在传输过程中数据不能被意外的改变。

.机密性：确定信息只能被授权过的用户使用。

12.2 公共安全

安全性声明是指表明应用是有安全结构的；包括权限、访问控制、认证。在web应用中部署描述是安全声明的主要工具。

开发者应该为应用运行时配一个逻辑安全策略，在运行时中，servlet引擎用这个安全策略去验证授权请求。

安全模块应该适用web应用的静态内容，当servlet用RequestDispatcher调用一个静态资源或servlet用forward或include,时安全模块不适用。

12.3 程序级安全

当应用的安全模块不能充分的表明安全时程序安全就可以被使用。
程序安全有以下部分组成:
HttpServletRequest接口:
.getRemoteUser
.isUserInRole
.getUserPrincipal
getRemoteUser方法返回客户端用户的名称，用于授权。
isUserInRole方法判断远程用户是否在一个安全的角色内。
getUserPrincipal方法返回一个java.security.Principal对象，表明当前用户的主要名称。这个API允许servlet根据这个信息处理一些业务逻辑。
如果用户没有授权，getRemoteUser返回null，isUserInRole返回false，getUserPrincipal返回null。
isUserInRole以一个role-name为参数。一个security-role-ref元素为在部署描述文件中定义，role-name子元素包含角色名称。Security-role包含一个子元素role-link,role-link的value值是客户端用户将匹配的安全角色。当用isUserInRole时引擎将调用security-role-ref到security-role的匹配。
举个例子:

```
<security-role-ref>
 <role-name>FOO</role-name>
 <role-link>manager</role-link>
</security-role-ref>
```

当一个用户属于"manager"角色是调用isUsesrInRole("FOO")返回true

如果匹配一个security-role元素的security-role-ref没有被定义，引擎必须找出一个与security-role列表不同的roel-name元素作为web应用默认的安全角色。但默认的安全角色限制了变换角色名称不需要重新编译的机动性。

12.4 角色

一个安全角色是一组用户的逻辑名称。当应用发布时，角色就部署在web应用的运行时环境中了。在基于安全属性的请求到来时，servlet引擎就会执行公共安全或程序级安全。安全请求在以下的一些情况中会产生:
1）开发者给用户群配置了一个安全角色。
2）开发者把一个安全角色配置给一个安全域中的一个主要名称。

12.5 验证

web客户端可以用以下的一些机制验证用户:
.HTTP Basic Authentication
.HTTP Digest Authentication
.HTTPS Client Authentication
.Form Based Authentication

12.5.1 HTTP Basic Authentication

HTTP Basic Authentication是基于用户名和密码的验证机制，是在HTTP/1.0规范中定义的。web服务要求客户端验证用户，web服务用一个realm字符串作为请求的一部分，用户通过这个realm字符串被验证。realm字符串不会和任何安全域相关联。客户端获得用户名称和密码发送到服务端。然后服务端用一个特殊的realm去验证该用户。
Basic Authentication 不是安全的验证协议。用户的密码是用base64编码的，服务端是不能够识别的。一些附加的安全措施可以被使用，这些协议有：HTTPS安全传输协议或网络安全标准（如

IPSEC协议、VPN策略）。

12.5.2 HTTP Digest Authentication

和HTTP Basic Authentication一样，HTTP Digest Authentication 也是验证用户名和密码的。然而这种验证是把密码加密传输的，要比Basic Authentication的base64编码要安全的多。Digest Authentication 没有被广泛的运用，建议servlet引擎支持这种验证，但不是必须的。

12.5.3 Form Based Authentication

web应用部署描述文件包含登陆表单和错误页面。登陆表单必须包含用户名和用户密码字段，这两个字段必须以j_username和j_password命名。当一个用户要访问一个被保护的资源时，引擎就会验证该用户信息，如果该用户验证通过就会调用保护的资源，如果没有验证通过，以下的步骤就会发生：

1）登陆表单被送到客户端，启动这个验证的URL路径将被引擎保存。
2）用户被要求填写用户名和密码
3）客户端重新post表单到服务端
4）引擎尝试着重新去验证该用户的信息
5）如果验证有失败，响应被设置为401的错误页面将被返回
6）如果验证成功，如果该访问的资源在一个授权角色中，将进一步验证。
7）如果用户是授权用户，客户端将用保存的URL路径重新定向到访问的资源。

发送到验证失败的用户的错误页面包含了失败的信息。

Form Based Authentication 和 Basic Authentication 有一样的弊端，密码是用简单的文本传输的，不能够被服务端验证。附加的一些协议可以增强这部分功能。如HTTPS传输协议，或网路安全标准（如IPSEC协议或VPN策略）。

12.5.3.1 登陆表单的一些注意事项

登陆的表单和跟踪session的URL实现上是有限制的。
登陆表单只能在cookies或SSL跟踪session的方式下才能使用。
登陆表单要进行验证的话，表单的action就必须为j_security_check。这个约束使得登陆表单访问的资源没有问题，也避免了把表单外的字段提交进请求中。
在HTML页面中登陆表单的一个例子如下：

```
<form method="POST" action="j_security_check" >
 <input type="text" name="j_username">
 <input type="password" name="j_password">
</form>
```

当登陆表单因为HTTP请求被调用时，原始的请求参数必须被引擎存储，在验证成功后重新定向请求的资源。
如果用登陆表单的用户被验证通过，创建了一个session,当session超时或调用了失效方法，使得登陆者推出了。后续来的请求就必须重新对用户进行验证。

12.5.4 HTTPS Client Authentication

用HTTPS验证用户是很强的验证机制。这个验证需要用户拥有一个公共钥匙（PKC）。servlet引擎不需要支持HTTPS协议。

12.6 验证信息的跟踪

基本的身份验证实在运行时环境中进行的：

1）确认一个登陆验证机制或策略已经配置在web应用中。
2）把需验证的信息发往一个容器中的所有应用。

3）当一个安全域被删除时，用户的请求就需要重新验证。

第十三章 部署描述文件

13.1 部署描述元素

部署描述文件中的所有元素都要被所有的servlet引擎支持。配置类型有下面的几种：

.ServletContext Init Parameteres
.Session Configuration
.Servlet Declaration
.Servlet Mapping
.Application Lifecycle Listener classes
.Filter Dfinitions and Filter Mappings
.MIME Type Mappings
.Error Pages

当servlet 引擎是实现了J2EE规范的一部分的时候，安全信息才有必要在部署描述文件中定义。部署描述文件不光光只支持servlet规范，在部署描述文件为了适应web应用的需求增加了其他的部属描述元素。如：

.taglib
.用于查找JNDI对象的一些元素（env-entry,ejb-ref,ejb-local-ref,resource-ref,resource-env-ref）

13.2 处理部属描述文件的规则

本节主要讲一下web容器访问部属描述文件的几种规则。

.web容器应该忽略部属描述文件中数据的开始空字符和最后的空字符。
.web容器应该有一个广泛的选项用户检查web应用的有效性。如检查web应用是否包含部署描述文件，部属描述文件结构是否正确。部署描述中应该有语义检查，如：安全规则有同样的名字，应该报错等等。
.部署描述中的URI是假定在URL解码表单中的
.引擎必须正确解释部属文件中的路径，如：路径'/a/../b'必须解释为'/b'，因为路径是以'..'开始的路径在部署描述中是无效的。
.调用资源的URI与WAR的根目录关联，以'/'开头
.在一个元素中，如果它的显示值是一个列表，则这个列表将可能会报错

13.2.1 DOCTYPE

所有2.3版本的部属描述文件的DOCTYPE必须是：

```
<!DOCTYPE web-app PUBLIC "-//sun Microsystems, inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
```

13.3 DTD

web应用的部署描述文件DTD如下：

```
<!--
 web-app 元素是部署描述的根元素
-->
<!ELEMENT web-app(icon?,display-name?,description?,distributable?,context-param*,filter*,
filter-mapping*,listener*,servlet*,servlet-mapping*，session-config?,mime-mapping*,welcome-
file-list?,error-page*,taglib*,resource-env-ref*,resource-ref*,security-constraint*,login-config?,
```

security-role*,env-entry*,ejb-ref*,ejb-local-ref*)>
<!--
Auth-constraint 元素定义了可以访问资源列表的用户角色。Role-name可以作为security-role元素的子元素出现也可以以role-name"*"出现表示所有的角色，如果*和role-name都出现了，引擎将解释这个资源可以被所有角色使用。如果没有定义角色，用户就不能够访问定义的资源。
Used in :security-constraint
-->
<!ELEMENT auth-constraint (description?,role-name*)>
<!--
Auth-method 元素是配置web应用的安全机制的，是访问被保护资源的先决条件。用户必须用这个验证机制验证。这个元素Value值有"BASIC","DIGEST","FORM"或"CLIENT-CERT"
Userd in : login-config
-->
<!ELEMENT auth-method(#PCDATA)>
<!--
Context-param元素包含了servlet context初始化的参数
Used in :web-app
-->
<!ELEMENT context-param(param-name,param-value,description?)>
<!--
description元素用于对父元素进行说明，description元素可以包含任何说明信息。当父元素被工具访问时这些说明就会显示。
Used in :auth-contraint,context-param,ejb-local-ref,ejb-ref,env-entry,filter,init-param,resource-env-ref,resource-ref,ren-as,security-role,security-role-ref,servlet,user-data-constraint,web-app,web-resource-collection
-->
<!ELEMENT description(#PCDATA)>
<!--
Display-name元素是一个简称，被调用的工具显示，这个简称不必是唯一的
Used in: filter,security-constraint,sevlet,web-app
例如:
<display-name>Employee self Service </display-name>
-->
<!ELEMENT display-name(#PCDATA)>
<!--
Distributable 元素,这个元素出现在部署描述中，说明该应用可以部署在分布式servlet引擎中。
Used in :web-app
-->
<!ELEMENT distributable EMPTY>
<!--
Ejb-link 元素用在ejb-ref 或ejb-local-ref元素中，去指定EJB关联的一个enterprise bean.
Ejb-link中的名字是一个关联着enterprise bean的路径 或者是目标bean+一个以"#"开头的路径。
一个ejb-name可以对应多个enterprise beans。

Used in : ejb-local-ref,ejb-ref
例如:
<ejb-link>EmployeeRecord</ejb-link>
<ejb-link>../products/product.jar#ProductEJB</ejb-link>
-->
<!ELEMENT ejb-link (#PCDAATA)>
<!--
ejb-local-ref 元素用于描述本地enterprise bean的home接口，描述由下面的部分组成:
一个可选的描述
与enterprise bean 相关的EJB名称
Enterprise bean 的类型
Enterprise bean 的本地接口
可选的ejb-link信息
Used in： web-app
-->
<!ELEMENT ejb-local-ref (description？，ejb-ref-name,ejb-ref-type,local-home,local,ejb-link?)>
<!--
Ejb-ref 元素用于描述enterprise bean的home接口。描述由下面组成:
一个可选描述
与enterprise bean 相关的EJB名称
Enterprise bean 的类型
Enterprise bean 的本地接口
可选的ejb-link信息
Used in：web-app
-->
<!ELEMENT ejb-ref(description？，ejb-ref-name,ejb-ref,type,home,remote,ejb-link?)>
<!--
Ejb-ref-name 元素包含一个EJB的名字，这个名字必须是唯一的；这个EJB是web应用环境和关联的java:comp/env context的入口。建议名字以"ejb/"开头。
Used in : ejb-local-ref,ejb-ref
例如:
<ejb-ref-name>ejb/Payroll</ejb-ref-name>
-->
<!ELEMENT ejb-ref-name(#PCDATA)>
<!--
Ejb-ref-type 元素包含enterprise bean的类型。Ejb-ref-type元素必须是下面的一种:
<ejb-ref-type>Entity</ejb-ref-type>
<ejb-ref-type>Session</ejb-ref-type>
Used in :ejb-local-ref,ejb-ref
-->
<!ELEMENT ejb-ref-type(#PCDATA)>
<!--

Env-entry 元素包含了web应用的环境入口的描述。描述包含一个可选的描述，一个环境入口的名称，一个可选的value，如果value没有被指定，在部署中必须提供。
-->
<!ELEMENT env-entry (description?,env-entry-name,env-entry-value?,env-entry-type)>
<!--
env-entry-name元素包含了web应用环境入口的名称，这个名称是一个java:comp/env context关联的JNDI名称。名字必须是唯一的。
例如:
<env-entry-name>minAmount</env-entry-name>
Used in: env-entry
-->
<!ELEMENT env-entry-name(#PCDATA)>
<!--
Env-entry-type 元素包含环境入口值的fully-qualified java类型，这是web应用程序期望有的。
Env-enry-type合法的类型如下:
Java.lang.Boolean
Java.lang.Byte
Java.lang.Character
Java.lang.String
Java.lang.Short
Java.lang.Integer
Java.lang.Long
Java.lang.Float
Java.lang.Double
Used in : env-entry
-->
<!ELEMENT env-entry-type(#PCDATA)>
<!--
Env-entry-value元素包含了web应用环境入口的值，值必须是一个字符串。
例如:
<env-entry-value>100.00</env-entry-value>
Used in : env-entry
<!ELEMENT env-entry-value(#PCDATA)>

<!--
Error-code元素包含HTTP的一个错误代码，如：404
Used in : error-page
-->
<!ELEMENT error-code(#PCDATA)>
<!--
error-page 元素包含一个错误代码映射或错误类型资源的一个路径
Used in : web-app
-->

```
<!ELEMENT error-page((error-code | exception-type),location)>
<!--
 exception-type 包含一个java exception 类型的类名称
Used in : error-page
-->
<!ELEMENT exception-type(#PCDATA)>
<!--
 extension元素包含一个扩展名。如"txt"
Used in : mine-mapping
-->
<!ELEMENT extension (#PCDATA)>
<!--
 filter被filter-mapping中的一个servlet或一个URL通过filter-name映射。filter在运行时能够通过FilterConfig接口访问初始化的参数。
Used in : web-app
-->
<!ELEMENT filter (icon?,filter-name,display-name?,description?,filter-class,init-param*)>

<!--
 filter-class元素指明filter类.
Used in :filter
-->
<!ELEMENT filtere-class(#PCDATA)>
<!--
 引擎用filter-mapping去匹配请求的URI以及匹配的顺序，引擎在把匹配的URI匹配一个servlet。
Used in : web-app
-->
<!ELEMENT filter-mapping (Filter-name,(url-pattern | servlet-name))>
<!--
 filtere-name元素表明了一个filter的逻辑名称用于匹配用的。逻辑名称必须唯一
Used in : filter,filere-mapping
-->
<!ELEMENT filter-name (#PCDATA)>
<!--
 form-error-page元素定义了当登陆失败是调用的页面。这个路径以"/"开始
Used in: form-login-config
-->
<!ELEMENT form-error-page (#PCDATA)>
<!--
 form-login-config元素指定了登陆的错误页面，如果form不需要验证，这个元素将被忽略。
Used in : login-config
-->
<!ELEMENT form-login-cofig (form-login-page,form-error-page)>
```

```
<!--
 form-login-page 元素定义了登陆的页面。路径以"/"开头
 used in :form-login-config
-->
<!ELEMENT form-login-page (#PCDATA)>
<!--
 home元素包含了enterprise bean的home接口的名称
Used in :ejb-ref
例如:
<home>com.aardvark.payroll.PayrollHome</home>
<!ELEMENT home (#PCDATA)>
<!--
 http-method 包含了HTTP 方法(GET | POST |...)
Used in:web-resource-collection
-->
<!ELEMENT http-method (#PCDATA)>

<!--
 icon 元素包含small-icon 和large-icon元素，指明一个gif或jpeg的图标名称
Used in : filter,servlet,web-app
-->
<!ELEMENT icon (small-icon?,large-icon?)>

<!--
 init-param元素包含了name/value的servlet的初始化参数
Used in : filter,servlet
-->
<!ELEMETN init-param (param-name,param-value,description？)>
<!--
 jsp-file 元素 包含了一个以"/"开头JSP文件的全名。
Used in : servlet
-->
<!ELEMENT jsp-file (#PCDATA)>
<!--
 large-icon 元素包含一个32*32的图标文件名称。图片可以是jpeg或gif。
 used in ： icon
例如:
<large-icon>employee-service-icon32*32.jsp</large-icon>
-->
<!ELEMENT large-icon (#PCDATA)>

<!--
 listener 元素对应着listener bean
```

```
Used in : web-app
-->
<!ELEMENT listener (listener-class)>
<!--
 listener-class 元素，元素值是监听类的类名。
Used in : listener
-->
<!ELEMENT listener-class (#PCDATA)>
<!--
 load-on-startup 元素指明了这个servlet在web启动时是否必须装入（调用servlet的init()方法）。这个内容是可选的，但有值时必须是个整数，如果是负数，引擎可以选择在任何时候装载该servlet，如果是整数或0，引擎就必须在web应用启动时装入该servlet。数字越小越被优先装入。如果值一样，引擎可以自由选择装入的顺序。
Used in : servlet
-->
<!ELEMENT load-on-startup (#PCDATA)>
<!--
 local元素包含了enterprise bean的local接口
Used in : ejb-local-ref
-->
<!ELEMENT local (#PCDATA)>
<!--
 local-home元素包含了enterprise bean的本地home接口
Used in : ejb-local-ref
-->
<!ELEMENT local-home (#PCDATA)>

<!--
 location 元素包含与web应用根目录关联的资源，值必须以'/'开头
Used in : error-page
-->
<!ELEMENT location (#PCDATA)>
<!--
 login-config 元素用于配置验证的方法的。
Used in :web-app
-->
<!ELEMENT login-conifg (auth-method?,realm-name?,form-login-config?)>

<!--
 mime-mapping 元素定义了扩展名和mime类型的映射关系。
Used in :web-app
-->
<!ELEMENT mime-mapping (extension,mime-type)>
```

```
<!--
 mime-type 元素包含了mime类型，如"text/plain"
Used in : mime-mapping
-->
<!ELEMENT mime-type (#PCDATA)>
<!--
 param-name 元素包含参数的名称，参数名必须唯一。
Used in : context-param,init-param
-->
<!ELEMENT param-name (#PCDATA)>

<!--
 param-value元素包含了参数的值
Used in : context-param,init-param
-->
<!ELEMENT param-value (#PCDATA)>

<!--
 realm-name 元素用于HTTP Basic 验证中
Used in : login-config
-->
<!ELEMENT realm-name (#PCDATA)>

<!--
 remote 元素包含了enterprise bean 的remote接口
Used in : ejb-ref
例如:
 <remote>com.wombat.empl.EmployeeService</remote>
-->
<!ELEMENT remote (#PCDATA)>

<!--
 res-auth 元素表明是web应用代码控制资源，还是引擎控制资源。该元素的值只能是以下的一种
    <res-auth>Application</res-auth>
    <res-auth>Container</res-auth>
Used in : resource-ref
-->
<!ELEMENT res-auth (#PCDATA)>

<!--
 res-ref-name 元素指明了资源管理器（连接工厂）的名称，这个名称是和java:comp/env
cocntext关联的JNDI名称。该名称必须是唯一的。
```

```
Used in：resource-ref
-->
<!ELEMENT res-ref-name (#PCDATA)>

<!--
 res-sharing-scope 元素表明了从资源管理器连接工厂获得的连接是否可以被共享。值必须是下面的一种：
 <res-sharing-scope>Shareable</res-sharing-scope>
 <res-sharing-scope>Unshareable</res-sharing-scope>
默认的值是Shareable
Used in : resource-ref
<!ELEMENT ref-sharing-scope (#PCDATA)>

<!--
 res-type 元素描述了资源的数据类型。
Used in : resource-ref
-->
<!ELEMENT res-type (#PCDATA)>

<!--
 resource-env-ref 元素描述了与web应用管理对象相关的资源。包含一个可选的描述，一个资源环境名称，一个环境资源类型。
Used in :web-app
例如：
<resource-env-ref>
 <resource-env-ref-name>jms/StockQueue</resource-env-ref-name>
 <resource-env-type>javax.jms/Queue</resource-env-ref-type>
</resource-env-ref>
-->
<!ELEMENT resource-env-ref (description?,resource-env-ref-name,resource-env-ref-type)>

<!--
 resource-env-ref-name元素一定一个环境资源名称，这个名字是与java:comp/env context关联的JNDI名称，必须以唯一的。
Used in : resource-env-ref
-->
<!ELEMENT resource-env-ref-name (#PCDATA)>

<!--
 resource-env-ref-type元素定义了环境资源的类型，是一个java类或接口的全名。
Used in: resource-env-ref
-->
<!ELEMENT resource-env-ref-type (#PCDATA)>
```

```
<!--
   resource-ref 元素包含web应用涉及的外部资源的描述。它有一个可选的描述，一个资源管理连接工厂的名称，一个资源管理连接工厂的类型id，一个验证类型（Application 或 Container），和一个可选的连接共享的选项（Shareale 或 Unshareable）
Used in : web-app
例如:
        <resource-ref>
        <res-ref-name>jdbc/EmployeeApppDB</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
        <res-sharing-scope>Shareable</res-sharing-scope>
        </resource-ref>
-->
<!ELEMENT resource-ref (description?,res-ref-name,res-type,res-auth,res-sharing-scope?)>

<!--
 role-link 元素是与安全角色相关的。Role-link元素必须包含一个在security-role元素中定义的安全角色的名称。
Used in: security-role-ref
-->
<!ELEMENT role-link (#PCDATA)>

<!--
 role-name 元素包含一个安全角色的名称，名称必须遵守NMTOKEN规则。
Used in : auth-constraint,run-as,security-role,security-role-ref
-->
<!ELEMENT role-name (#PCDATA)>

<!--
 run-as 元素包含一个可选的描述，和一个安全角色的名称。
Used in : servlet
-->
<!ELEMENT run-as (description?,role-name)>

<!--
 security-constraint元素用于安全约束与一个或多个web资源相关联。
Used in :web-app
-->
<!ELEMENT security-constraint (display-name?,web-resource-collection+,auth-constraint?,user-data-constraint?)>

<!--
```

security-role元素包含安全角色的定义，它由一个可选的安全角色的描述，一个安全角色名称组成。
Used in : web-app
例如:

```
        <security-role>
        <description>
          this role includes all employees who are authorized to access the employee service
        application.
        </description>
        <role-name>employee</role-name>
        </security-role>
-->
<!ELEMENT security-role (description?,role-name)>

<!--
 security-role-ref 元素包含一个可选的描述，一个调用代码中的安全角色名称，一个可选的安全角色连接。如果安全角色没有被指定，开发者必须选择一个适当的安全角色。这个role-name元素的值必须是EJBConteext.isCallerInRole(String roleName)或HttpServletRequest.isUserInRole (String role)中的参数。
Used in :servlet
-->
<!ELEMENT security-role-ref (description?,role-name,role-link?)>

<!--
 servlet 元素包含一个servlet的数据描述。如果load-on-startup元素中指定了一个jsp文件，该JSP将被装入。
Used in : web-app
<!ELEMENT servlet (icon?,servlet-name,display-name?,description?,(servlet-class|jsp-file),init-param*,load-on-startup?,run-as?,security-role-ref*)>

<!--
 servlet-class元素包含一个全名的servlet类名称。
Used in : servlet
-->
<!ELEMENT servlet-class (#PCDATA)>

<!--
 servlet-mapping元素定义了一个servlet和url的关联
Used in : web-app
-->
<!ELEMENT servlet-mapping (servlet-name,url-pattern)>

<!--
```

servlet-name元素包含servlet的名称，名称是唯一的。
Used in : filter-mapping,servlet,servlet-mapping
-->
<!ELEMENT servlet-name (#PCDATA)>

<!--
session-config 元素定义了session参数
Used in : web-app
-->
<!ELEMENT session-config (session-timeout?)>

<!--
session-timeout 元素定义了一个默认的会话超时的时间，使用于web应用中的所有会话。时间必须是用分钟的数值表示。
如果timeout是0或负数，引擎将确保会话永远不会超时。
Used in : session-config
-->
<!ELEMENT session-timeout (#PCDATA)>

<!--
small-icon 元素包含一个16*16图标文件的名称。
Used in : icon
例如:
<small-icon>employee-service-icon16*16.jpg</small-icon>
-->
<!ELEMENT small-icon (#pCDATA)>

taglib 元素用于描述JSP tag 库。
Used in : web-app
-->
<!ELEMENT taglib (taglib-uri,taglib-location)>

<!--
taglib-location 元素包含一个资源定位，为tag库找到tag描述文件
Used in:taglib
-->
<!ELEMENT taglib-location (#PCDATA)>

<!--
taglib-uri元素描述了一个URI
Used in: taglib
-->
<!ELEMENT taglib-uri (#PCDATA)>

```
<!--
 transport-guarantee元素指定了客户端和服务端的通信关系，有NONE，INTEGRAL，
CONFIDENTIAL。NONE表示着应用不需要任何传输保障。INTEGRAL表示着在数据在客户端到服
务端的过程中不能有任何改变。CONFIDENTIAL表示在传输过程中防止其他传输内容的干扰。在
使用SSI时常用的就INTEGRAL或CONFIDENTIL。
Used in : user-data-constraint
<!ELEMENT transport-guarantee (#PCDATA)>

<!--
 url-pattern 元素包含映射的url。必须符合11.2章中servlet API描述的规则。
Used in:filter-mapping,servlet-mapping,web-resource-collection
-->
<!ELEMENT url-pattern (#PCDATA)>

<!--
 user-data-constraint元素用于表明数据在客户端到服务器端是怎么保护的。
Used in :security-constraint
-->
<!ELEMENT user-data-constraint (description?,transport-guarantee)>

<!--
 web-resource-collection元素用于web应用中安全限制的资源被那些方法使用，如果没有指定，
就可以被web用的所有方法调用。
Used in:security-constraint
-->
<!ELEMENT web-resource-collection (web-resource-name,description?,url-pattern*,http-
method*)>

<!--
 web-resource-name 包含一个web资源集合的名称
Used in:web-resource-collection
-->
<!ELEMENT web-resource-name (#PCDATA)>

<!--
 welcome-file元素包含了web应用中默认的访问文件，如index.html
Used in:welcome-file-list
-->
<!ELEMENT welcome-file (#PCDATA)>

<!--
 welcome-file-list包含welcome-file的列表
```

```
Used in:web-app
-->
<!ELEMENT welcome-file-list (welcome-file+)>
<!--
 ID机制可以增加额外的部署信息，不允许加一个非标准的元素到标准的部署描述中
-->
<!ATTLIST auth-constraint id ID #IMPLIED>
<!ATTLIST auth-method id ID #IMPLIED>
<!ATTLIST context-param id ID #IMPLIED>
<!ATTLIST description id ID #IMPLIED>
<!ATTLIST display-name id ID #IMPLIED>
<!ATTLIST ejb-link id ID #IMPLIED>
<!ATTLIST ejb-local-ref id ID #IMPLIED>
<!ATTLIST ejb-ref  id ID #IMPLIED>
<!ATTLIST ejb-ref-name id ID #IMPLIED>
<!ATTLIST ejb-ref-type id ID #IMPLIED>
<!ATTLIST env-entry id ID #IMPLIED>
<!ATTLIST env-entry-name id ID #IMPLIED>
<!ATTLIST env-entry-type id ID #IMPLIED>
<!ATTLIST env-enry-value id ID #IMPLIED>
<!ATTLIST error-code id ID #IMPLIED>
<!ATTLIST error-page id ID #IMPLIED>
<!ATTLIST exception-type id ID #IMPLIED>
<!ATTLIST extension id ID #IMPLIED>
<!ATTLIST filter id ID #IMPLIED>
<!ATTLIST filter-class id ID #IMPLIED>
<!ATTLIST filtere-mapping id ID #IMPLIED>
<!ATTLIST filter-name id ID #IMPLIED>
<!ATTLIST form-error-page id ID #IMPLIED>
<!ATTLIST form-login-config id ID #IMPLIED>
<!ATTLIST form-login-page id ID #IMPLIED>
<!ATTLIST home id ID #IMPLIED>
<!ATTLIST http-method id ID #IMPLIED>
<!ATTLIST icon id ID #IMPLIED>
<!ATTLIST init-param id ID #IMPLIED>
<!ATTLIST jsp-file id ID #IMPLIED>
<!ATTLIST large-icon id ID #IMPLIED>
<!ATTLIST listener id ID #IMPLIED>
<!ATTLIST listener-class id ID #IMPLIED>
<!ATTLIST load-on-startup id ID #IMPLIED>
<!ATTLIST local id ID #IMPLIED>
<!ATTLIST local-home id ID #IMPLIED>
<!ATTLIST location id ID #IMPLIED>
```

<!ATTLIST login-config id ID #IMPLIED>
<!ATTLIST mime-mapping id ID #IMPLIED>
<!ATTLIST mime-type id ID #IMPLIED>
<!ATTLIST param-name id ID #IMPLIED>
<!ATTLIST param-value id ID #IMPLIED>
<!ATTLIST realm-name id ID #IMPLIED>
<!ATTLIST remote id ID #IMPLIED>
<!ATTLIST res-auth id ID #IMPLIED>
<!ATTLIST res-ref-name id ID #IMPLIED>
<!ATTLIST res-sharing-scope id ID #IMPLIED>
<!ATTLIST res-type id ID #IMPLIED>
<!ATTLIST resource-env-ref id ID #IMPLIED>
<!ATTLIST resource-env-ref-name id ID #IMPLIED>
<!ATTLIST resource-env-ref-type id ID #IMPLIED>
<!ATTLIST resource-ref id ID #IMPLIED>
<!ATTLIST role-link id ID #IMPLIED>
<!ATTLIST role-name id ID #IMPLIED>
<!ATTLIST run-as id ID #IMPLIED>
<!ATTLIST security-constraint id ID #IMPLIED>
<!ATTLIST security-role id ID #IMPLIED>
<!ATTLIST security-role-ref id ID #IMPLIED>
<!ATTLIST sevlet id ID #IMPLIED>
<!ATTLIST servlet-class id ID #IMPLIED>
<!ATTLIST servlet-mapping id ID #IMPLIED>
<!ATTLIST servlet-name id ID #IMPLIED>
<!ATTLIST session-config id ID #IMPLIED>
<!ATTLIST session-timeout id ID #IMPLIED>
<!ATTLIST small-icon id ID #IMPLIED>
<!ATTLIST taglib id ID #IMPLIED>
<!ATTLIST taglib-location id ID #IMPLIED>
<!ATTLIST taglib-uri id ID #IMPLIED>
<!ATTLIST transport-guarantee id ID #IMPLIED>
<!ATTLIST rul-pattern id ID #IMPLIED>
<!ATTLIST user-data-constraint id ID #IMPLIED>
<!ATTLIST web-app id ID #IMPLIED>
<!ATTLIST web-resource-collection id ID #IMPLIED>
<!ATTLIST web-resource-name id ID #IMPLIED>
<!ATTLIST welcome-file id ID #IMPLIED>
<!ATTLIST welcome-file-list id ID #IMPLIED>

13.4 例子

13.4.1 基本的例子

<!DOCTYPE web-app PUBLIC "-//Sun Mmicrosystems, Inc.//DTD Web Application 2.3//EN"

```
"http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
<web-app>
   <display-name>A Simple Application</display-name>
   <context-param>
      <param-name>Webmaster</param-name>
      <param-value>webmaster@mycorp.com</param-value>
  </context-param>
   <servlet>
      <servlet-name>catalog</servlet-name>
      <serrvlet-class>com.mycorp.Catalogservlet</servlet-class>
      <init-param>
         <param-name>catalog</param-name>
         <param-value>Spring</param-value>
      </init-param>
   </servlet>
   <servlet-mapping>
      <servlet-name>catalog</servlet-name>
      <url-pattern>/catalog/*</url-pattern>
   </servlet-mapping>
   <session-config>
      <session-timeout>30</session-timeout>
   </session-config>
   <mime-mapping>
      <extension>pdf</extension>
      <mime-type>application/pdf</mime-type>
   </mime-mapping>
   <welcome-file-list>
      <welcome-file>index.jsp</welcome-file>
      <welcome-file>index.html</welcome-file>
      <welcome-file>index.htm</welcome-file>
   </welcome-file-list>
   <error-page>
      <error-code>404</error-code>
      <location>/404.html</location>
   <error-page>
</web-app>
```

13.4.2 一个安全的例子

```
<!DOCTYPE web-app PUBLIC "-//Sun Mmicrosystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
<web-app>
   <display-name>A Secure Application</display-name>
   <security-role>
```

```
      <role-name>manager</role-name>
   </security-role>
   <servlet>
      <servlet-name>catalog</servlet-name>
      <servlet-class>com.mycorp.CatalogServlet</servlet-class>
      <init-param>
         <param-name>catalog</param-name>
         <param-value>Spring</param-value>
      </init-param>
      <security-role-ref>
          <role-name>MGR</role-name>
          <!-- 在代码中用的角色名称-->
          <role-link>manager</role-link>
      </security-role-ref>
   </servlet>
   <servlet-mapping>
       <servlet-name>catalog</servlet-name>
       <url-pattern>/catalog/*</url-pattern>
   </servlet-mapping>
   <security-constraint>
      <web-resource-collection>
         <web-resource-name>SalesInfo</web-resource-name>
         <url-pattern>/salesinfo/*</url-pattern>
         <http-method>GET</http-method>
         <http-method>POST</http-method>
      </web-resource-collection>
      <auth-constraint>
         <role-name>manager</role-name>
      </auth-constraint>
      <user-data-constraint>
         <transport-guarantee>CONFIDENTIAL</transport-guarantee>
      </user-data-constraint>
   </security-constraint>
</web-app>
```

结尾:
servlet2.3到此就结束了，至于对接口类的解释。这里就不给与了。有兴趣的朋友可以down一个servlet2.3的源码包自己研究研究。