**Java Servlet Technology**

Java Servlet技术使用请求 - 响应编程模型在Web应用程序中提供动态的面向用户的内容。

# 什么是Servlet?

servlet是一种Java编程语言类，用于扩展托管应用程序的服务器的功能，这些应用程序使用请求 - 响应编程模型来访问。虽然servlet可以响应任何类型的请求，但它们通常用于扩展由Web服务器托管的应用程序。对于这样的应用程序，Java Servlet技术定义了HTTP特定的servlet类。

javax.servlet和javax.servlet.http包提供用于编写​​servlet的接口和类。所有servlet都必须实现Servlet接口，它定义了生命周期方法。实现通用服务时，可以使用或扩展Java Servlet API提供的GenericServlet类。HttpServlet类提供了诸如doGet和doPost等方法来处理特定于HTTP的服务。

# Servlet生命周期

servlet的生命周期由其部署在的容器来控制。当请求映射到servlet时，容器将执行以下步骤。

1. 如果servlet的实例不存在，则Web容器：
   1. 加载servlet类
   2. 创建servlet类实例
   3. 通过调用init方法来初始化servlet实例（初始化包含在Creating and Initializing a Servlet中）
2. 容器调用service方法，传递request和response对象。service方法在Writing Service Methods中讨论。

如果需要删除servlet，容器通过调用servlet的destroy方法来最终回收servlet。有关更多信息，请参阅Finalizing a Servlet。

## 处理Servlet生命周期事件

你可以通过定义监听器对象来监听和响应servlet的生命周期事件，该监听器对象的方法会在servlet生命周期事件发生时被调用。要使用这些监听器对象，必须定义并指定监听器类。

### 定义监听器类

你定义了一个监听器类作为listener接口的实现。下表列举了可以被监听的事件，以及必须实现的对应接口。当调用一个监听器方法时，它传递会一个包含与事件相关的信息的事件。例如，一个HttpSessionEvent会被传递给在HttpSessionListener接口中的方法，HttpSessionEvent包含了一个HttpSession

Servlet生命周期事件

| **Object** | **Event** | **Listener Interface and Event Class** |
| --- | --- | --- |
| Web context | Initialization and destruction | javax.servlet.ServletContextListener and ServletContextEvent |
| Web context | Attribute added, removed, or replaced | javax.servlet.ServletContextAttributeListener and ServletContextAttributeEvent |
| Session | Creation, invalidation, activation, passivation, and timeout | javax.servlet.http.HttpSessionListener, javax.servlet.http.HttpSessionActivationListener, and HttpSessionEvent |
| Session | Attribute added, removed, or replaced | javax.servlet.http.HttpSessionAttributeListenerand HttpSessionBindingEvent |
| Request | A servlet request has started being processed by web components | javax.servlet.ServletRequestListener and ServletRequestEvent |
| Request | Attribute added, removed, or replaced | javax.servlet.ServletRequestAttributeListener and ServletRequestAttributeEvent |

使用@WebListener注解来定义一个监听器，来获取特定Web应用程序中context。用@WebListener注解的类必须实现以下接口之一：

javax.servlet.ServletContextListener

javax.servlet.ServletContextAttributeListener

javax.servlet.ServletRequestListener

javax.servlet.ServletRequestAttributeListener

javax.servlet..http.HttpSessionListener

javax.servlet..http.HttpSessionAttributeListener

例如，以下代码片段定义了一个实现其中两个接口的监听器：

import javax.servlet.ServletContextAttributeListener;

import javax.servlet.ServletContextListener;

import javax.servlet.annotation.WebListener;

@WebListener()

public class SimpleServletListener implements ServletContextListener,

ServletContextAttributeListener {

...

## 处理Servlet错误

当servlet执行时，可能会发生任何一个异常。发生异常时，Web容器生成一个包含以下消息的默认页面：

A Servlet Exception Has Occurred

但是您也可以指定容器应该返回给定异常的特定错误页面。

# 共享信息

Web组件与大多数对象一样，通常可以与其他对象一起工作来完成任务。 Web组件可以通过执行以下操作来实现。

* 使用私有帮助对象（例如，JavaBeans组件）。
* 共享作为公共域属性的对象。
* 使用数据库
* 调用其他web资源。Java Servlet 技术机制允许一个web组件调用其他的web资源。（在Invoking Other Web Resources有讲述）

## 使用域对象

协作的web组件通过作为四个作用域对象属性的对象来共享信息。你可以使用表示域的类的getAttribute和setAttribute方法来访问这些属性。下表列举了这些域对象。

| **Scope Object** | **Class** | **Accessible From** |
| --- | --- | --- |
| Web context | javax.servlet.ServletContext | Web components within a web context. See [Accessing the Web Context](https://docs.oracle.com/javaee/7/tutorial/servlets008.htm#BNAGL). |
| Session | javax.servlet.http.HttpSession | Web components handling a request that belongs to the session. See [Maintaining Client State](https://docs.oracle.com/javaee/7/tutorial/servlets009.htm#BNAGM). |
| Request | Subtype of javax.servlet.ServletRequest | Web components handling the request. |
| Page | javax.servlet.jsp.JspContext | The JSP page that creates the object. |

## 控制共享资源并发访问

在多线程服务器中，可以并发访问共享资源。除了范围对象属性之外，共享资源还包括内存中的数据，例如实例或类变量，以及外部对象，如文件，数据库连接和网络连接。

并发访问可能会在几种情况下出现。

* 多个Web组件访问存储在Web上下文中的对象。
* 多个Web组件访问存储在会话中的对象。
* Web组件中的多个线程访问变量实例。Web容器通常会为每一个请求创建一个线程来处理。为了确保servlet实例一次只处理一个请求，servlet可以实现SingleThreadModel接口。如果一个servlet实现了这个接口，那么在servlet的service方法中并不会同时执行两个线程。Web容器可以通过同步对servlet的单个实例的访问或通过维护Web组件实例池并将每个新请求分派到空闲实例来实现此保证。此接口不会阻止Web组件访问共享资源（如静态类变量或外部对象）导致的同步问题。

当资源可以同时访问时，它们可以以不一致的方式使用。您可以通过使用http://docs.oracle.com/javase/tutorial/essential/concurrency/中的线程课程中描述的同步技术来控制访问来防止这种情况。

# 创建并初始化一个Servlet

在Web应用程序使用@WebServlet注解来定义一个servlet组件。此注解在一个类上指定，并包含有关正在声明的servlet的元数据。注解的servlet必须至少指定一个URL模式。这可以通过在注解上使用urlPatterns或value属性来完成。所有其他属性都是可选的，都有默认设置。当注解唯一的属性是URL时，使用value属性。否则，当使用其他属性时，使用urlPatterns属性。

使用@WebServlet注解的类必须继承javax.servlet.http.HttpServlet类。例如，以下代码片段使用URL模式/report定义了一个servlet：

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

@WebServlet("/report")

public class MoodServlet extends HttpServlet {

...

Web容器在加载和实例化servlet类之后以及从客户端递送请求之前初始化一个servlet。要自定义此进程以允许servlet读取持久配置数据，初始化资源并执行任何其他一次性活动，您可以覆盖Servlet接口的init方法，也可以指定@WebServlet注解的initParams属性。initParams属性包含@WebInitParam注解。如果无法完成初始化过程，则servlet会抛出一个UnavailableException。

使用初始化参数提供特定servlet所需的数据。相比之下，上下文参数提供了可用于Web应用程序的所有组件的数据。

# 编写Service方法

Servlet提供的服务是在GenericServlet中的service方法，在一个HttpServlet对象的doMethod方法（Method可以是Get，Delete，Options，Post，Put或Trace中的一个），或者由实现Servlet接口的类定义的任何其他协议特定方法中来实现。Service方法是servlet中用来为任何方法来给一个客户端提供服务。service方法的一般模式是从请求中提取信息，访问外部资源，然后根据该信息填充响应。对于HTTP servlet，填写响应的正确过程是执行以下操作：

1. 从response中获取一个输出流
2. 填写响应头
3. 在输出流中写入主体内容

在提交响应之前，必须始终设置响应头。Web容器将在响应提交后忽略任何设置或添加头信息的尝试。接下来的两节介绍如何从请求中获取信息并生成响应。

## 从请求中获取信息

一个request包含在客户端和servlet之间传递的数据。所有的request都实现ServletRequest接口。该接口定义了访问以下信息的方法：

* 参数，通常用于在客户端和servlet之间传递信息
* 对象值属性，通常用于在Web容器和servlet之间或协作servlet之间传递信息
* 关于用于通信请求的协议以及涉及到客户端和服务器的请求的信息
* 与本地化有关的信息

您还可以从request中检索输入流，并手动解析数据。要想读取字符数据，使用由request的getReader方法返回的BufferedReader对象。要想读取二进制数据，使用getInputStream方法返回的ServletInputStream对象。

HTTP servlet被传递一个HTTP request对象，HttpServletRequest，它包含了请求的URL，HTTP头，查询字符串，等等。一个HTTP 请求URL包好一下部分。

http://[*host*]:[*port*][*request-path*]?[*query-string*]

请求路径（request-path）进一步由以下元素组成。

* **Context Path**：由一个正斜杠（/）与servlet的web程序的根contenxt连接
* **Servlet path**：这个路径部分与这次请求激活的组件别名相对应。此路径以正斜杠（/）开头。
* **Path Info：这个**请求路径的一部分不是context路径或servlet路径的一部分。

您可以使用HttpServletRequest接口的getContextPath，getServletPath和getPathInfo方法来访问此信息。除了request URI和路径部分之间的URL编码差异外，request URI始终由context路径加上servlet路径加上路径信息组成。

查询字符串由一组参数和值组成。通过使用getParameter方法从请求中检索各个参数。这有两种生成查询字符串的方式。

* 查询字符串可以显式地显示在网页中。
* 当提交具有GET HTTP方法的表单时，将查询字符串附加到URL。

## 构建响应

一个response包含在服务器和客户端之间传递的数据。所有response都实现了ServletResponse接口。此接口定义了允许您执行以下操作的方法。

* 检索用于向客户端发送数据的输出流。要发送字符数据，使用response的getWriter方法返回的PrinteWriter对象。要在多用途因特网邮件扩充（MIME）响应中发送二进制数据，使用getOutputStream方法返回的ServletOutputStream对象。要混合二进制和文本数据，如在多部分响应中，使用ServletOutputStream并手动管理字符部分。
* 使用setContentType(String)方法指定response返回的内容类型（如，text/html）。这个方法必须在响应提交前调用。内容类型名称的注册表由互联网号码分配机构（IANA）保存，网址为<http://www.iana.org/assignments/media-types/>。
* 指示是否使用setBufferSize（int）方法缓冲输出。默认情况下，写入输出流的任何内容都将立即发送到客户端。缓冲允许在将任何内容发回客户端之前写入内容，从而为servlet提供更多的时间来设置适当的状态代码和响应头，或转发到另一个Web资源。这个方法必须在任何内容被写入或者在响应提交前。
* 设置本地化信息，如区域设置和字符编码。有关详细信息，请参见第20章“国际化和本地化Web应用程序”。

HTTP response对象javax.servlet.http.HttpServletResponse具有表示HTTP标头的字段，如下所示。

* 状态码，用于指示请求不满足或请求被重定向的原因。
* Cookies，用于在客户端存储应用程序特定的信息。有时，cookies用于维护用于跟踪用户会话的标识符（请参阅“会话跟踪”）。

# 过滤请求和响应

一个过滤器是一个可以转换请求或响应头和内容（或两者）的对象。过滤器不同于Web组件，因为过滤器通常不会创建响应。相反，过滤器提供可以“附加”到任何类型的Web资源的功能。因此，过滤器不应具有与作为过滤器作用的Web资源的任何依赖关系; 这样，它可以由多种类型的网络资源组成。

过滤器可以执行的主要任务如下。

* 查询请求并相应地执行。
* 阻止请求/响应对进一步传递。
* 修改请求头和数据。您可以通过提供自定义版本的请求来执行此操作。
* 修改响应头和数据。您可以通过提供自定义版本的响应来实现。
* 与外部资源交互。

过滤器的应用包括验证，日志，图像转换，数据压缩，加密，分解字符串，XML转换等。

你可以配置一个web资源，由特定顺序的零个，一个或者多个过滤器组成的过滤链来过滤。当包含组件的Web应用程序被部署时，该过滤链被指定，并且低昂web容器加载组件时，它被实例化。

## 过滤器编程

过滤API由javax.servlet包中的Filter，FilterChain和FilterConfig接口定义。您可以通过实现Filter接口定义一个过滤器。

使用@WebFilter注解来定义Web应用程序中的过滤器。此注解在类上指定，并包含有关正在声明的过滤器的元数据。注解的过滤器必须至少指定一个URL模式。这可以通过在注解上使用urlPatterns或value属性来完成。所有其他属性都是可选的，都有一个默认设置。当注解上唯一的属性是URL模式时，使用value属性;当其他属性也被使用时，使用urlPatterns属性。

使用@WebFilter注解注解的类必须实现javax.servlet.Filter接口。

要向过滤器添加配置数据，请指定使用@WebFilter注解的initParams属性。该initParams在属性包含一个@WebInitParam注解。以下代码片段定义了一个过滤器，指定了一个初始化参数：

import javax.servlet.Filter;

import javax.servlet.annotation.WebFilter;

import javax.servlet.annotation.WebInitParam;

@WebFilter(filterName = "TimeOfDayFilter",

urlPatterns = {"/\*"},

initParams = {

@WebInitParam(name = "mood", value = "awake")})

public class TimeOfDayFilter implements Filter {

...

Filter接口中最重要的方法是doFilter，它有三个参数，分别是request，response和filter chain对象。此方法可以执行以下操作。

* 检查请求头
* 自定义请求对象，如果过滤器希望修改请求头或数据。
* 自定义响应对象，如果过滤器希望修改响应头或数据。
* 调用过滤器链中的下一个实体。如果当前的过滤器是以目标为Web组件或静态资源结束的链中的最后一个过滤器，那么下一个实体是链末尾的资源; 否则，它是在WAR中配置的下一个过滤器。过滤器通过调用链对象上的doFilter方法来调用下一个实体，并传入被调用的请求和响应或它可能创建的包装版本为参数。或者过滤器可以通过不调用下一个实体来阻止请求。在后一种情况下，过滤器负责填写响应。
* 在链中调用下一个过滤器后，检查响应头。
* 抛出异常以指示处理中的错误。

除了doFilter，你还必须实现init和destroy方法。当过滤器被实例化时，容器调用init方法。如果你想将初始化参数传递给过滤器，则可以从传递给init的FilterConfig对象中检索它们。

## 自定义请求和响应编程

过滤器有许多方法来修改请求或响应。例如，过滤器可以向请求添加属性，或者可以在响应中插入数据。

过滤器要修改一个响应，那么它通常必须在响应返回客户端前前捕获它。为此，您将一个备用流传递给生成响应的servlet。备用流阻止servlet在完成时关闭原始响应流，并允许过滤器修改servlet的响应。

要将这个备用流传递给servlet，过滤器将创建一个响应包装器，它重写getWriter或getOutputStream方法以返回此备用流。包装器被传递给过滤器链的doFilter方法。包装方法默认调用到包装的请求或响应对象。

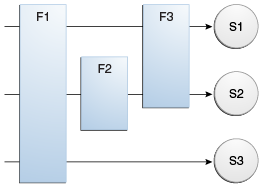
要重写请求方法，将请求包装在一个继承ServletRequestWrapper或HttpServletRequestWrapper的对象中。要重写响应方法，将响应包装在一个继承了ServletResponseWrapper 或者HttpServletResponseWrapper的对象中.

## 指定过滤器映射

Web容器使用过滤器映射（filter mappings）来决定如何将过滤器应用于Web资源。一个过滤器映射通过名称匹配一个过滤器到一个web组件，或者通过URL模式到一个web资源。过滤器按照过滤器映射在WAR的过滤器映射列表中显示的顺序来调用。你可以在部署描述文件中为一个WAR指定一个过滤器映射列表，通过使用NetBeans IDE或使用XML编写该列表。

如果要将每个请求记录到Web应用程序，请将命中计数器过滤器映射到URL模式/ \*。

您可以将过滤器映射到一个或多个Web资源，并且可以将多个过滤器映射到一个Web资源。这在图17-1中示出，其中过滤器F1被映射到servlets S1，S2和S3;过滤器F2映射到servlet S2;并且滤波器F3被映射到servlets S1和S2。



回想一下，filter chain是传递给过滤器的doFilter方法的对象之一。该链通过过滤器映射间接形成。链中过滤器的顺序与过滤器映射在Web应用程序部署描述符中显示的顺序相同。

当过滤器映射到servlet S1时，Web容器调用F1的doFilter方法。S1的过滤器链中每个过滤器的doFilter方法通过前一个过滤器的chain.doFilter方法来调用。因为S1的过滤器链包含过滤器F1和F3，所以F1调用chain.doFilter调用过滤器F3的doFilter方法。当F3的doFilter方法完成时，控制返回到F1的doFilter方法。

### 使用NetBeans IDE来指定过滤器映射

1. 在Project选项卡中展开应用程序的项目节点。
2. 展开project下的Web Pages 和 WEB-INF节点。
3. 双击web.xml。
4. 点击编辑器窗口上的Filters。
5. 展开编辑器窗口中的Servelt Filters节点。
6. 点击Add Filter Element通过名称或者URL去映射一个过滤器到一个web资源上
7. 在Add Servlet Filter 对话窗口，在Filter Name字段输入过滤器的名称
8. 点击Browse找到过滤器应用到的servlet类。

您可以包含通配符，以便您可以将过滤器应用于多个servlet。

1. 点击OK
2. 要限制过滤器如何应用于请求，请按照下列步骤操作。
   1. 展开Filter Mappings节点
   2. 从过滤器列表中选择过滤器
   3. 点击Add
   4. 在Add Filter Mapping对话窗口，选择以下dispatcher类型之一：

* REQUEST: 只有当请求直接来自客户端时
* ASYNC：只有异步请求来自客户端
* FORWARD: 只有当请求被转发到组件时（请参阅Transferring Control to Another Web Component）
* INCLUDE：只有当请求被包含的组件处理时（请参阅Including Other Resources in the Response）
* ERROR：只有当使用错误页面机制处理请求时（请参阅Handling Servlet Errors）

您可以通过选择多个dispatcher类型来指示过滤器应用于上述情况的任何组合。如果没有指定类型，默认选项为REQUEST。

# 调用其他Web资源

Web组件可以间接和直接地调用其他Web资源。一个 Web组件间接调用其他的web资源，通过在返回给客户端的内容中嵌入一个指向其他web组件的URL。当它正在执行时，Web组件直接调用另一个资源，通过包含另一个web组件的内容或者转发请求到另一个资源中。

要调用运行Web组件的服务器上可用的资源，必须首先使用getRequestDispatcher（“URL”）方法获取一个RequestDispatcher对象。您可以从一个request或Web context中获取RequestDispatcher对象; 然而，两种方法的行为略有不同。该方法将所请求资源的路径作为参数。Request可以机诶搜一个相对路径（也就是，一个不以/开头的路径），但是web context需要一个绝对路径。如果资源不可用或服务器尚未为该类型的资源实现RequestDispatcher对象，则getRequestDispatcher将返回null。您的servlet应该准备好处理这种情况。

## 在response中包含其他资源

在Web组件返回的响应中，包含其他Web资源（例如横幅内容或版权信息）通常是有用的。要包含另一个资源，调用RequestDispatcher对象的include方法：

include(request, response);

如果资源是静态的，这个include方法启用程序化服务端include。如果资源是一个web组件，这个方法的作用是发送请求到这个包含的web组件中，执行Web组件，然后将在所包含的servlet中的响应的执行结果包含在内。一个包含的Web组件可以访问该请求对象，但是它对该响应对象可以做什么有限。

* 它可以写入响应的正文并提交响应。
* 它不能设置头或调用任何影响响应标头的方法，如setCookie。

## 转交控制权给另一个web组件

在某些应用程序中，您可能希望有一个Web组件对请求进行初步处理，并使另一个组件生成响应。例如，您可能希望部分地处理请求，然后根据请求的性质转移到另一个组件。

要将控制权转发到另一个Web组件，可以调用RequestDispatcher的forward方法。当转发请求时，请求的URL被设置为转发页面的路径。原始URI及其组成部分保存为以下请求属性：

javax.servlet.forward.request\_uri

javax.servlet.forward.context\_path

javax.servlet.forward.servlet\_path

javax.servlet.forward.path\_info

javax.servlet.forward.query\_string

forward方法将回复用户的责任给了另一个资源。如果您已经在servlet中访问了ServletOutputStream或PrintWriter对象，则不能使用此方法;这样做会引发IllegalStateException。

# 访问Web Context

Web组件执行的上下文是一个实现ServletContext接口的对象。您可以使用getServletContext方法检索Web context。 Web context提供了以下访问的方法

* 初始化参数
* 与Web context相关联的资源
* 对象值属性
* 日志功能

计数器的访问方法是同步的，以防止并发运行的servlet执行不兼容的操作。过滤器使用context的getAttribute方法检索计数器对象。计数器的递增值记录在日志中。

# 维持客户端状态

许多应用程序要求来自客户端的一系列的相互联系的请求。例如，Web应用程序可以跨请求保存用户购物车的状态。基于Web的应用程序负责维护这种状态，这被称为会话（session），因为HTTP是无状态的。为了支持一个需要维持状态的应用程序，Java Servlet技术提供了一个用于管理会话的API，并实现了允许会话的几种机制。

## 访问一个会话

会话由HttpSession对象表示。通过调用request对象的getSession方法来访问会话。此方法返回与此request相关联的当前会话;或者，如果request没有会话，则此方法创建一个会话。

## 将对象与会话相关联

您可以按名称将对象值属性与会话相关联。这些属性可以由属于相同Web context的任何Web组件访问，并且可以由相同会话的任一个request来处理。

回想一下，您的应用程序可以通知Web context和session的监听器对象servlet生命周期事件（请看处理Servlet生命周期事件）。您还可以通知对象与某个会话关联的某些事件，例如以下情况。

* 对象在一个session中被添加或者删除时。要接收此通知，您的对象必须实现javax.servlet.http.HttpSessionBindingListener接口。
* 当连接对象的session将被钝化或激活时。一个session会被钝化或激活，当它在虚拟机中被移动或者被保存并且从持久化存储中被恢复时。要接收此通知，您的对象必须实现javax.servlet.http.HttpSessionActivationListener接口。

## 会话管理

因为HTTP客户端无法表明它不再需要会话，所以每个会话都有一个关联的超时时间，以便它的资源可以被回收。可以通过使用会话的getMaxInactiveInterval和setMaxInactiveInterval方法来访问超时时间。

* 为了确保活动会话没有超时，您应该通过使用service方法定期访问会话，因为这会重置会话的生存时间计数器。
* 特定客户端交互完成后，您可以使用会话的invalidate方法来使服务器端的会话无效，并删除任何会话数据。

### 使用NetBeans IDE设置会话超时时间

要使用NetBeans IDE在部署描述符中设置超时时间，请按照下列步骤操作。

1. 打开项目如果你还没有打开的话
2. 在“Projects”选项卡中展开项目的节点。
3. 展开项目节点下的“Web Pages” 和 “WEB-INF”节点。
4. 双击web.xml
5. 点击编辑器上部的General
6. 在Session Timeout字段，输入一个整数值

整数值表示会话超时之前必须通过的不活动分钟数。

## 会话追踪

为了将会话与用户相关联，Web容器可以使用多种方法，所有这些方法都涉及在客户端和服务器之间传递标识符。该标识符可以作为cookie保存在客户端上，或者Web组件可以在返回给客户端的每个URL中包括标识符。

如果您的应用程序使用会话对象，则必须确保通过在客户端关闭Cookie时使应用程序重写URL来启用会话跟踪。您可以通过在servlet返回的所有URL上调用response的encodeURL（URL）方法来执行此操作。该方法仅在Cookie被禁用时才在URL中包含的session ID;否则，该方法返回URL不变。

# 终结一个Servlet

Web容器可能会决定servlet应该从服务中删除（比如，当一个容器想要回收内存资源或者当它准备关闭的时候）。在这种情况下，容器调用Servlet接口的destroy方法。在此方法中，您可以释放servlet正在使用的任何资源，并保存任何持久状态。destroy方法释放在init方法中创建的数据库对象。

一个servlet的service方法应该在servlet移除的时候全部结束。服务器尝试通过仅在当所有的服务请求已经返回，或者在服务器指定宽限期之后，才调用destrou方法来确保这点。如果你的servlet操作可能运行的比服务器宽限时间更久的话，那么在调用destroy方法时，操作仍然可以运行。您必须确保任何线程仍处理客户端请求完成。

本节的其余部分将介绍如何执行以下操作。

* 跟踪当前运行service方法的线程数。
* 通过使用destroy方法提醒长时间运行的线程，关闭操作在等待它们的完成，以此提供一个干净的关机。
* 长时间运行的方法定期检查关机，如有必要，停止工作，清理并返回。

## 跟踪服务请求

要跟踪服务请求，请在servlet类中包含一个计算正在运行的service方法数的字段。该字段应该具有同步访问方法来递增，递减并返回其值：

public class ShutdownExample extends HttpServlet {

private int serviceCounter = 0;

...

// Access methods for serviceCounter

protected synchronized void enteringServiceMethod() {

serviceCounter++;

}

protected synchronized void leavingServiceMethod() {

serviceCounter--;

}

protected synchronized int numServices() {

return serviceCounter;

}

}

Service方法应当每次在进入方法的时候递增service counter，并且应当怎每次方法返回的时候递减counter。这是几种您的HttpServlet子类应该重写service方法的情景之一。新方法应该调用super.service来保留原始service方法的功能：

protected void service(HttpServletRequest req,

HttpServletResponse resp)

throws ServletException,IOException {

enteringServiceMethod();

try {

super.service(req, resp);

} finally {

leavingServiceMethod();

}

}

## 提醒方法关闭

为了确保清洁关机，你的destroy方法不应该释放任何共享资源，直到所有的服务请求全部完成。这样做的一部分就是检查service counter。另一部分是去提醒长时间运行的方法，现在是关闭的时间了。对于此通知，需要另一个字段。该字段应具有通常的访问方式：

public class ShutdownExample extends HttpServlet {

private boolean shuttingDown;

...

//Access methods for shuttingDown

protected synchronized void setShuttingDown(boolean flag) {

shuttingDown = flag;

}

protected synchronized boolean isShuttingDown() {

return shuttingDown;

}

}

以下是使用这些字段提供清洁关机的destroy方法的示例：

public void destroy() {

/\* Check to see whether there are still service methods /\*

/\* running, and if there are, tell them to stop. \*/

if (numServices()> 0) {

setShuttingDown(true);

}

/\* Wait for the service methods to stop. \*/

while (numServices()> 0) {

try {

Thread.sleep(interval);

} catch (InterruptedException e) {

}

}

}

## 创建礼让长时间运行的方法

提供干净关闭的最后一步是使任何长时间运行的方法表现得礼貌。可能运行很长时间的方法应该检查通知他们关闭的字段的值，并且如果需要的话应该中断他们的工作：

public void doPost(...) {

...

for(i = 0; ((i < lotsOfStuffToDo) &&

!isShuttingDown()); i++) {

try {

partOfLongRunningOperation(i);

} catch (InterruptedException e) {

...

}

}

}

# 使用Java Servlet技术上传文件

支持文件上传是许多Web应用程序的一个非常基本和通用的要求。在以前版本的Servlet规范中，实现文件上传需要使用外部库或复杂的输入处理。Java Servlet规范现在帮助以通用和便捷的方式为这个问题提供一个可行的解决方案。Java Servlet技术现在支持文件上传开箱即用，因此实现此规范的任何Web容器都可以解析多部分请求，并通过HttpServletRequest对象使mime附件可用。

一个新的注解，javax.servlet.annotation.MultipartConfig，用于指示声明它的servlet的期望使用multipart / form-data的MIME类型的请求。一个使用@MultipartConfig注解的Servlet可以获取给定multipart / form-data请求的Part组件，通过调用request.getPart(String name) 或者request.getParts() 方法.

## @MultipartConfig 注解

@MultipartConfig注解支持以下可选属性。

* Location：一个文件系统上目录的绝对路径。location属性不支持相对于应用程序上下文的路径。此路径用于临时存储文件，当part被处理时或者文件大小超过指定的fileSizeThreshold设置时。默认的location是“”。
* fileSizeThreshold：文件暂时存储在磁盘上时的大小，以字节为单位。默认大小是0字节
* MaxFileSize：允许上传的文件的最大大小，以字节为单位。如果任何上传的文件大小超过此设置，web容器会抛出一个异常（IllegalStateException）。默认大小是无限制。
* maxRequestSize：允许multipart/form-data请求的最大大小，以字节为单位。如果所有上传的文件的总体大小超过此阈值，Web容器将会抛出一个异常。默认大小是无限制。

例如，@MultipartConfig注解可以以如下构造：

@MultipartConfig(location="/tmp", fileSizeThreshold=1024\*1024,

maxFileSize=1024\*1024\*5, maxRequestSize=1024\*1024\*5\*5)

您可以在web.xml文件中添加以下作为servlet配置元素的子元素，而不是使用@MultipartConfig注解来对文件上传servlet中的这些属性进行硬编码：

<multipart-config>

<location>/tmp</location>

<max-file-size>20848820</max-file-size>

<max-request-size>418018841</max-request-size>

<file-size-threshold>1048576</file-size-threshold>

</multipart-config>

getParts和getPart方法

Servlet规范支持两种额外的HttpServletRequest方法：

* Collection<Part> getParts()
* Part getPart(String name)

request.getParts（）方法返回所有Part对象的集合。如果您有多个类型文件的输入，则返回多个Part对象。因为Part对象是有名称的，所以getPart（String name）方法可用于访问特定的Part。或者，可以使用返回Iterable <Part>的getParts（）方法来获取所有Part对象上的迭代器。

javax.servlet.http.Part接口是一个简单的接口，它提供了允许对每个Part进行内省的方法。方法如下：

* 获取Part的名称，大小，和内容类型
* 查询Part提交的头部信息
* 删除Part
* 将一个Part写入磁盘

例如，Part接口提供一个write（String filename）方法使用指定的名称来写入文件。然后，该文件可以保存在使用@MultipartConfig注释的location属性指定的目录中，或者，在fileupload示例的情况下，在由表单中的目标字段指定的位置。。

**异步处理**

在应用程序服务器中的web容器通常为每个客户端请求使用一个服务线程。在重负载条件下，容器需要大量线程来满足所有的客户端请求。可扩展性限制包括内存不足或容器线程池的耗尽。要创建可扩展的Web应用程序，您必须确保与请求相关联的线程没有空闲，这样容器可以使用它们来处理新的请求。

下面有两种与请求相关的线程被闲置的情况

* 在创建响应之前，线程需要等待资源变为可用或者正在处理数据。例如，应用程序可能需要在生成响应之前查询数据库或从远程Web服务访问数据。
* 线程需要等待事件才能生成响应。例如，应用程序可能必须在生成响应之前等待JMS消息，来自其他客户端的新信息或队列中可用的新数据。

这些场景表示了阻塞式操作，它们会限制web应用程序的可拓展性。异步处理是指将这些阻塞操作分配给新线程，并将与请求相关联的线程立即重新调用到容器。

**Servlet中的异步处理**

Java EE为servlet和过滤器提供异步处理支持。如果servlet或过滤器在处理请求时遇到潜在的阻塞操作，则可以将操作分配给异步执行上下文，并将与请求相关联的线程立即返回到容器而不产生响应。阻塞操作在异步执行上下文中在不同的线程中完成，可以生成响应或将请求发送到另一个servlet。

要在servlet上启用异步处理，请将@WebServlet注释上的参数asyncSupported设置为true，如下所示：

@WebServlet(urlPatterns={"/asyncservlet"}, asyncSupported=true)

public class AsyncServlet extends HttpServlet { ... }

javax.servlet.AsyncContext类提供了在service方法中执行异步处理所需的功能。要获取AsyncContext的实例，请在service方法的请求对象上调用startAsync（）方法;例如：

public void doGet(HttpServletRequest req, HttpServletResponse resp) {

...

AsyncContext acontext = req.startAsync();

...

}

此调用将请求置于异步模式，并确保退出service方法后未提交响应。你必须在异步上下文中等待阻塞式操作完成后或者将请求转发到另一个servlet后，生成一个响应。

下表介绍了AsyncContext类提供的基本功能。

| **Method Signature** | **Description** |
| --- | --- |
| void start(Runnable run) | 容器提供了可以处理阻塞操作的不同线程。  你为阻塞式操作提供代码，作为一个实现了Runable接口的类。可以提供这个类作为内部类，当调用start方法，或者使用使用其他机制将AsyncContext实例传递给您的类。 |
| ServletRequest getRequest() | 返回用于初始化此异步上下文的request。在上面的例子中，request与service方法中的相同。  您可以在异步上下文中使用此方法从request中获取参数。 |
| ServletResponse getResponse() | 返回用于初始化此异步上下文的response。在上面的例子中，response与service方法中的相同。  您可以在异步上下文中使用此方法，将阻塞操作的结果写入响应。 |
| void complete() | 完成异步操作并关闭与此异步上下文相关联的响应。  在写入异步上下文中的响应对象之后调用此方法。 |
| void dispatch(String path) | 将请求和响应对象调度到给定的路径。.  在阻止操作完成后，您可以使用此方法让另一个servlet写入响应。 |

**等待资源**

本节演示如何在下面示例中使用AsyncContext提供的功能：

1. 一个servlet接收一个来自GET请求中的参数。
2. servlet使用资源（如数据库或Web服务）根据参数的值检索信息。资源有时会很慢，所以这可能是一个阻塞操作。
3. servlet使用资源的结果生成响应

以下代码显示了不使用异步处理的基本servlet：

@WebServlet(urlPatterns={"/syncservlet"})

public class SyncServlet extends HttpServlet {

private MyRemoteResource resource;

@Override

public void init(ServletConfig config) {

resource = MyRemoteResource.create("config1=x,config2=y");

}

@Override

public void doGet(HttpServletRequest request,

HttpServletResponse response) {

response.setContentType("text/html;charset=UTF-8");

String param = request.getParameter("param");

String result = resource.process(param);

/\* ... print to the response ... \*/

}

}

以下代码显示了使用异步处理的同一servlet：

@WebServlet(urlPatterns={"/asyncservlet"}, asyncSupported=true)

public class AsyncServlet extends HttpServlet {

/\* ... Same variables and init method as in SyncServlet ... \*/

@Override

public void doGet(HttpServletRequest request,

HttpServletResponse response) {

response.setContentType("text/html;charset=UTF-8");

final AsyncContext acontext = request.startAsync();

acontext.start(new Runnable() {

public void run() {

String param = acontext.getRequest().getParameter("param");

String result = resource.process(param);

HttpServletResponse response = acontext.getResponse();

/\* ... print to the response ... \*/

acontext.complete();

}

}

AsyncServlet将asyncSupported = true添加到@WebServlet注解。其余的差异在服务方法之内。

* request.startAsync（）导致请求被异步处理;该响应不会在服务方法结束时发送给客户端。
* acontext.start（new Runnable（）{...}）从容器中获取一个新的线程。
* 内部类的run（）方法中的代码在新线程中执行。内部类可以访问异步上下文以从请求读取参数并写入响应。调用异步上下文的complete（）方法提交响应并将其发送给客户端。

AsyncServlet的服务方法立即返回，而请求在异步上下文中处理。

# 非阻塞式I/O

在应用程序服务器中的web容器通常为每个客户端请求使用一个服务线程。要开发可扩展的Web应用程序，您必须确保与客户端请求相关联的线程从不闲置需要等待阻塞操作完成。“异步处理”提供了一种在新线程中执行应用程序特定阻塞操作，并将与请求相关联的线程立即返回到容器中的机制。即使您在服务方法中使用异步处理来处理所有特定于应用程序的阻塞操作，基于与input/output会阻塞的考虑，与客户端请求相关联的线程仍然可能暂时闲置。

例如，如果客户端通过缓慢的网络连接提交大型HTTP POST请求，相比客户端提供的请求速度，服务器可以更快的处理请求。使用传统的I / O，与此请求关联的容器线程有时会空闲等待请求的剩余部分完成。

在异步模式下处理请求时，Java EE为servlet和过滤器提供非阻塞I / O支持。以下步骤总结了如何使用非阻塞I / O来处理请求并在服务方法中写入响应。

1. 将request置入如“异步处理”中所述的异步模式。
2. 在service方法中从request和response对象中获取一个输入流或输出流。
3. 分配一个读取监听器给输入流或者一个写入监听器给输出流。
4. 在监听器的回调方法内处理请求和响应。

表17-4和表17-5描述了用于非阻塞I / O支持的servlet输入和输出流中可用的方法。表17-6描述了读取监听器和写入监听器的接口。

***Table 17-4 Nonblocking I/O Support in javax.servlet.ServletInputStream***

| **Method** | **Description** |
| --- | --- |
| void setReadListener(ReadListener rl) | 将此输入流与一个包含回调方法的监听器对象关联起来，以用来异步读取数据。你以匿名类形式提供这个监听器对象，或者使用其他机制将输入流传递给这个读取监听器对象。 |
| boolean isReady() | 如果数据可以读取数据而不阻塞时，则返回true。 |
| boolean isFinished() | 当所有的数据读取完毕后返回true。 |

***Table 17-5 Nonblocking I/O Support in javax.servlet.ServletOutputStream***

| **Method** | **Description** |
| --- | --- |
| void setWriteListener(WriteListener wl) | 将此输出流与一个包含回调方法的监听器对象关联起来，以用来异步写入数据。你以匿名类形式提供这个写入监听器对象，或者使用其他机制将输出流传递给这个写入监听器对象。 |
| boolean isReady() | 如果数据可以写入而不阻塞时，则返回true。 |

***Table 17-6 Listener Interfaces for Nonblocking I/O Support***

| **Interface** | **Methods** | **Description** |
| --- | --- | --- |
| ReadListener | void onDataAvailable()  void onAllDataRead()  void onError(Throwable t) | 当有数据可用来读取、所有的数据被读取完毕或者当有错误发生时，一个ServletInputStream实例在它的监听器上调用这些方法。 |
| WriteListener | void onWritePossible()  void onError(Throwable t) | 当有可以写入数据时或者当有错误发生时，一个ServletOutputStream实例在它的监听器上调用这些方法。 |

## 使用非阻塞I/O 读取一个大的HTTP POST 请求

本节中的代码显示了如何通过将请求置于异步模式（如“异步处理”中所述）并使用表17-4和表17-6中的非阻塞I / O功能来读取servlet内的大型HTTP POST请求。

@WebServlet(urlPatterns={"/asyncioservlet"}, asyncSupported=true)

public class AsyncIOServlet extends HttpServlet {

@Override

public void doPost(HttpServletRequest request,

HttpServletResponse response)

throws IOException {

final AsyncContext acontext = request.startAsync();

final ServletInputStream input = request.getInputStream();

input.setReadListener(new ReadListener() {

byte buffer[] = new byte[4\*1024];

StringBuilder sbuilder = new StringBuilder();

@Override

public void onDataAvailable() {

try {

do {

int length = input.read(buffer);

sbuilder.append(new String(buffer, 0, length));

} while(input.isReady());

} catch (IOException ex) { ... }

}

@Override

public void onAllDataRead() {

try {

acontext.getResponse().getWriter()

.write("...the response...");

} catch (IOException ex) { ... }

acontext.complete();

}

@Override

public void onError(Throwable t) { ... }

});

}

}

此示例使用@WebServlet注解参数asyncSupported = true声明这个web servlet带有异步支持。服务方法首先通过调用request对象的startAsync（）方法将请求置于异步模式，这是为了使用非阻塞I / O所必需的。然后，服务方法获得与请求相关联的输入流，并分配一个定义为内部类的读取监听器。监听器读取请求的一部分，当它可用时，然后在完成读取请求后向客户端写入一些响应。

# 协议升级处理

在HTTP / 1.1中，客户端可以通过使用Upgrade头字段来请求切换在当前连接上的不同协议。如果服务器接受客户端指示的切换协议请求，它会使用状态吗101（切换协议）产生一个HTTP响应。在此交换之后，客户端和服务器使用新协议进行通信。

例如，一个客户端可以使一个HTTP请求切换到XYZP协议：

GET /xyzpresource HTTP/1.1

Host: localhost:8080

Accept: text/html

Upgrade: XYZP

Connection: Upgrade

OtherHeaderA: Value

客户端可以使用HTTP头为新协议指定一个参数。服务器可以接受这个请求并且产生如下的一个响应：

HTTP/1.1 101 Switching Protocols

Upgrade: XYZP

Connection: Upgrade

OtherHeaderB: Value

(XYZP data)

Java EE支持Servlet中的HTTP协议升级功能，如表17-7所示。

***Table 17-7 Protocol Upgrade Support***

| **Class or Interface** | **Method** |
| --- | --- |
| HttpServletRequest | HttpUpgradeHandler upgrade(Class handler)  Upgrade方法开始协议升级处理。这个方法实例化一个实现了 HttpUpgradeHandler 接口的类，并且将连接委托给它。  你在接受来自一个客户端切换协议的请求时，在service里调用upgrade方法。 |
| HttpUpgradeHandler | void init(WebConnection wc)  当servlet接受切换协议请求后，调用init方法。你可以实现这个方法，并且从Webconnection对象中获取输入输出流 来实现新的协议. |
| HttpUpgradeHandler | void destroy()  当客户端断开连接后调用destroy方法。你可以实现这个方法并且释放任何与处理新协议相关联的资源。. |
| WebConnection | ServletInputStream getInputStream()  getInputStream方法提供对连接的输入流的访问。你可以在返回的流中使用非阻塞I/O来实现新协议 |
| WebConnection | ServletOutputStream getOutputStream()  getOutputStream方法提供对连接的输出流的访问。你可以在返回的流中使用非阻塞I/O来实现新协议 |

以下代码演示如何接受来自客户端的HTTP协议升级请求：

@WebServlet(urlPatterns={"/xyzpresource"})

public class XYZPUpgradeServlet extends HttpServlet {

@Override

public void doGet(HttpServletRequest request,

HttpServletResponse response) {

if ("XYZP".equals(request.getHeader("Upgrade"))) {

/\* Accept upgrade request \*/

response.setStatus(101);

response.setHeader("Upgrade", "XYZP");

response.setHeader("Connection", "Upgrade");

response.setHeader("OtherHeaderB", "Value");

/\* Delegate the connection to the upgrade handler \*/

XYZPUpgradeHandler = request.upgrade(XYZPUpgradeHandler.class);

/\* (the service method returns immedately) \*/

} else {

/\* ... write error response ... \*/

}

}

}

XYZPUpgradeHandler类处理这个连接：

public class XYZPUpgradeHandler implements HttpUpgradeHandler {

@Override

public void init(WebConnection wc) {

ServletInputStream input = wc.getInputStream();

ServletOutputStream output = wc.getOutputStream();

/\* ... implement XYZP using these streams (protocol-specific) ... \*/

}

@Override

public void destroy() { ... }

}

实现HttpUpgradeHandler的类使用当前连接的流与客户端使用新协议进行通信。有关HTTP协议升级支持的详细信息，请参阅http://jcp.org/en/jsr/detail?id=340上的Servlet 3.1规范。