**Servlet 3.0 新特性概述**

Servlet 3.0 作为 Java EE 6 规范体系中一员，随着 Java EE 6 规范一起发布。该版本在前一版本（Servlet 2.5）的基础上提供了若干新特性用于简化 Web 应用的开发和部署。其中有几项特性的引入让开发者感到非常兴奋，同时也获得了 Java 社区的一片赞誉之声：

1. 异步处理支持：有了该特性，Servlet 线程不再需要一直阻塞，直到业务处理完毕才能再输出响应，最后才结束该 Servlet 线程。在接收到请求之后，Servlet 线程可以将耗时的操作委派给另一个线程来完成，自己在不生成响应的情况下返回至容器。针对业务处理较耗时的情况，这将大大减少服务器资源的占用，并且提高并发处理速度。
2. 新增的注解支持：该版本新增了若干注解，用于简化 Servlet、过滤器（Filter）和监听器（Listener）的声明，这使得 web.xml 部署描述文件从该版本开始不再是必选的了。
3. 可插性支持：熟悉 Struts2 的开发者一定会对其通过插件的方式与包括 Spring 在内的各种常用框架的整合特性记忆犹新。将相应的插件封装成 JAR 包并放在类路径下，Struts2 运行时便能自动加载这些插件。现在 Servlet 3.0 提供了类似的特性，开发者可以通过插件的方式很方便的扩充已有 Web 应用的功能，而不需要修改原有的应用。

下面我们将逐一讲解这些新特性，通过下面的学习，读者将能够明晰了解 Servlet 3.0 的变化，并能够顺利使用它进行日常的开发工作。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-lo-servlet30/#ibm-pcon)

**异步处理支持**

Servlet 3.0 之前，一个普通 Servlet 的主要工作流程大致如下：首先，Servlet 接收到请求之后，可能需要对请求携带的数据进行一些预处理；接着，调用业务接口的某些方法，以完成业务处理；最后，根据处理的结果提交响应，Servlet 线程结束。其中第二步的业务处理通常是最耗时的，这主要体现在数据库操作，以及其它的跨网络调用等，在此过程中，Servlet 线程一直处于阻塞状态，直到业务方法执行完毕。在处理业务的过程中，Servlet 资源一直被占用而得不到释放，对于并发较大的应用，这有可能造成性能的瓶颈。对此，在以前通常是采用私有解决方案来提前结束 Servlet 线程，并及时释放资源。

Servlet 3.0 针对这个问题做了开创性的工作，现在通过使用 Servlet 3.0 的异步处理支持，之前的 Servlet 处理流程可以调整为如下的过程：首先，Servlet 接收到请求之后，可能首先需要对请求携带的数据进行一些预处理；接着，Servlet 线程将请求转交给一个异步线程来执行业务处理，线程本身返回至容器，此时 Servlet 还没有生成响应数据，异步线程处理完业务以后，可以直接生成响应数据（异步线程拥有 ServletRequest 和 ServletResponse 对象的引用），或者将请求继续转发给其它 Servlet。如此一来， Servlet 线程不再是一直处于阻塞状态以等待业务逻辑的处理，而是启动异步线程之后可以立即返回。

异步处理特性可以应用于 Servlet 和过滤器两种组件，由于异步处理的工作模式和普通工作模式在实现上有着本质的区别，因此默认情况下，Servlet 和过滤器并没有开启异步处理特性，如果希望使用该特性，则必须按照如下的方式启用：

1. 对于使用传统的部署描述文件 (web.xml) 配置 Servlet 和过滤器的情况，Servlet 3.0 为 <servlet> 和 <filter> 标签增加了 <async-supported> 子标签，该标签的默认取值为 false，要启用异步处理支持，则将其设为 true 即可。以 Servlet 为例，其配置方式如下所示：
2. <servlet>
3. <servlet-name>DemoServlet</servlet-name>
4. <servlet-class>footmark.servlet.Demo Servlet</servlet-class>
5. <async-supported>true</async-supported>

</servlet>

1. 对于使用 Servlet 3.0 提供的 @WebServlet 和 @WebFilter 进行 Servlet 或过滤器配置的情况，这两个注解都提供了 asyncSupported 属性，默认该属性的取值为 false，要启用异步处理支持，只需将该属性设置为 true 即可。以 @WebFilter 为例，其配置方式如下所示：

@WebFilter(urlPatterns = "/demo",**asyncSupported = true**)

public class DemoFilter implements Filter{...}

一个简单的模拟异步处理的 Servlet 示例如下：

@WebServlet(urlPatterns = "/demo", asyncSupported = true)

public class AsyncDemoServlet extends HttpServlet {

@Override

public void doGet(HttpServletRequest req, HttpServletResponse resp)

throws IOException, ServletException {

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

PrintWriter out = resp.getWriter();

out.println("进入Servlet的时间：" + new Date() + ".");

out.flush();

//在子线程中执行业务调用，并由其负责输出响应，主线程退出

AsyncContext ctx = req.startAsync();

new Thread(new Executor(ctx)).start();

out.println("结束Servlet的时间：" + new Date() + ".");

out.flush();

}

}

public class Executor implements Runnable {

private AsyncContext ctx = null;

public Executor(AsyncContext ctx){

this.ctx = ctx;

}

public void run(){

try {

//等待十秒钟，以模拟业务方法的执行

Thread.sleep(10000);

PrintWriter out = ctx.getResponse().getWriter();

out.println("业务处理完毕的时间：" + new Date() + ".");

out.flush();

ctx.complete();

} catch (Exception e) {

e.printStackTrace();

}

}

}

除此之外，Servlet 3.0 还为异步处理提供了一个监听器，使用 AsyncListener 接口表示。它可以监控如下四种事件：

1. 异步线程开始时，调用 AsyncListener 的 onStartAsync(AsyncEvent event) 方法；
2. 异步线程出错时，调用 AsyncListener 的 onError(AsyncEvent event) 方法；
3. 异步线程执行超时，则调用 AsyncListener 的 onTimeout(AsyncEvent event) 方法；
4. 异步执行完毕时，调用 AsyncListener 的 onComplete(AsyncEvent event) 方法；

要注册一个 AsyncListener，只需将准备好的 AsyncListener 对象传递给 AsyncContext 对象的 addListener() 方法即可，如下所示：

AsyncContext ctx = req.startAsync();

ctx.addListener(new AsyncListener() {

public void onComplete(AsyncEvent asyncEvent) throws IOException {

// 做一些清理工作或者其他

}

...

});

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-lo-servlet30/#ibm-pcon)

**新增的注解支持**

Servlet 3.0 的部署描述文件 web.xml 的顶层标签 <web-app> 有一个 metadata-complete 属性，该属性指定当前的部署描述文件是否是完全的。如果设置为 true，则容器在部署时将只依赖部署描述文件，忽略所有的注解（同时也会跳过 web-fragment.xml 的扫描，亦即禁用可插性支持，具体请看后文关于 [可插性支持](http://www.ibm.com/developerworks/cn/java/j-lo-servlet30/#major4)的讲解）；如果不配置该属性，或者将其设置为 false，则表示启用注解支持（和可插性支持）。

**@WebServlet**

@WebServlet 用于将一个类声明为 Servlet，该注解将会在部署时被容器处理，容器将根据具体的属性配置将相应的类部署为 Servlet。该注解具有下表给出的一些常用属性（以下所有属性均为可选属性，但是 vlaue 或者 urlPatterns 通常是必需的，且二者不能共存，如果同时指定，通常是忽略 value 的取值）：

**表 1. @WebServlet 主要属性列表**

| **属性名** | **类型** | **描述** |
| --- | --- | --- |
| name | String | 指定 Servlet 的 name 属性，等价于 <servlet-name>。如果没有显式指定，则该 Servlet 的取值即为类的全限定名。 |
| value | String[] | 该属性等价于 urlPatterns 属性。两个属性不能同时使用。 |
| urlPatterns | String[] | 指定一组 Servlet 的 URL 匹配模式。等价于 <url-pattern> 标签。 |
| loadOnStartup | int | 指定 Servlet 的加载顺序，等价于 <load-on-startup> 标签。 |
| initParams | WebInitParam[] | 指定一组 Servlet 初始化参数，等价于 <init-param> 标签。 |
| asyncSupported | boolean | 声明 Servlet 是否支持异步操作模式，等价于 <async-supported> 标签。 |
| description | String | 该 Servlet 的描述信息，等价于 <description> 标签。 |
| displayName | String | 该 Servlet 的显示名，通常配合工具使用，等价于 <display-name> 标签。 |

下面是一个简单的示例：

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

loadOnStartup = -1, name = "SimpleServlet", displayName = "ss",

initParams = {@WebInitParam(name = "username", value = "tom")}

)

public class SimpleServlet extends HttpServlet{ … }

如此配置之后，就可以不必在 web.xml 中配置相应的 <servlet> 和 <servlet-mapping> 元素了，容器会在部署时根据指定的属性将该类发布为 Servlet。它的等价的 web.xml 配置形式如下：

<servlet>

<display-name>ss</display-name>

<servlet-name>SimpleServlet</servlet-name>

<servlet-class>footmark.servlet.SimpleServlet</servlet-class>

<load-on-startup>-1</load-on-startup>

<async-supported>true</async-supported>

<init-param>

<param-name>username</param-name>

<param-value>tom</param-value>

</init-param>

</servlet>

<servlet-mapping>

<servlet-name>SimpleServlet</servlet-name>

<url-pattern>/simple</url-pattern>

</servlet-mapping>

**@WebInitParam**

该注解通常不单独使用，而是配合 @WebServlet 或者 @WebFilter 使用。它的作用是为 Servlet 或者过滤器指定初始化参数，这等价于 web.xml 中 <servlet> 和 <filter> 的 <init-param> 子标签。@WebInitParam 具有下表给出的一些常用属性：

**表 2. @WebInitParam 的常用属性**

| **属性名** | **类型** | **是否可选** | **描述** |
| --- | --- | --- | --- |
| name | String | 否 | 指定参数的名字，等价于 <param-name>。 |
| value | String | 否 | 指定参数的值，等价于 <param-value>。 |
| description | String | 是 | 关于参数的描述，等价于 <description>。 |

**@WebFilter**

@WebFilter 用于将一个类声明为过滤器，该注解将会在部署时被容器处理，容器将根据具体的属性配置将相应的类部署为过滤器。该注解具有下表给出的一些常用属性 ( 以下所有属性均为可选属性，但是 value、urlPatterns、servletNames 三者必需至少包含一个，且 value 和 urlPatterns 不能共存，如果同时指定，通常忽略 value 的取值 )：

**表 3. @WebFilter 的常用属性**

| **属性名** | **类型** | **描述** |
| --- | --- | --- |
| filterName | String | 指定过滤器的 name 属性，等价于 <filter-name> |
| value | String[] | 该属性等价于 urlPatterns 属性。但是两者不应该同时使用。 |
| urlPatterns | String[] | 指定一组过滤器的 URL 匹配模式。等价于 <url-pattern> 标签。 |
| servletNames | String[] | 指定过滤器将应用于哪些 Servlet。取值是 @WebServlet 中的 name 属性的取值，或者是 web.xml 中 <servlet-name> 的取值。 |
| dispatcherTypes | DispatcherType | 指定过滤器的转发模式。具体取值包括： ASYNC、ERROR、FORWARD、INCLUDE、REQUEST。 |
| initParams | WebInitParam[] | 指定一组过滤器初始化参数，等价于 <init-param> 标签。 |
| asyncSupported | boolean | 声明过滤器是否支持异步操作模式，等价于 <async-supported> 标签。 |
| description | String | 该过滤器的描述信息，等价于 <description> 标签。 |
| displayName | String | 该过滤器的显示名，通常配合工具使用，等价于 <display-name> 标签。 |

下面是一个简单的示例：

@WebFilter(servletNames = {"SimpleServlet"},filterName="SimpleFilter")

public class LessThanSixFilter implements Filter{...}

如此配置之后，就可以不必在 web.xml 中配置相应的 <filter> 和 <filter-mapping> 元素了，容器会在部署时根据指定的属性将该类发布为过滤器。它等价的 web.xml 中的配置形式为：

<filter>

<filter-name>SimpleFilter</filter-name>

<filter-class>xxx</filter-class>

</filter>

<filter-mapping>

<filter-name>SimpleFilter</filter-name>

<servlet-name>SimpleServlet</servlet-name>

</filter-mapping>

**@WebListener**

该注解用于将类声明为监听器，被 @WebListener 标注的类必须实现以下至少一个接口：

* ServletContextListener
* ServletContextAttributeListener
* ServletRequestListener
* ServletRequestAttributeListener
* HttpSessionListener
* HttpSessionAttributeListener

该注解使用非常简单，其属性如下：

**表 4. @WebListener 的常用属性**

| **属性名** | **类型** | **是否可选** | **描述** |
| --- | --- | --- | --- |
| value | String | 是 | 该监听器的描述信息。 |

一个简单示例如下：

@WebListener("This is only a demo listener")

public class SimpleListener implements ServletContextListener{...}

如此，则不需要在 web.xml 中配置 <listener> 标签了。它等价的 web.xml 中的配置形式如下：

<listener>

<listener-class>footmark.servlet.SimpleListener</listener-class>

</listener>

**@MultipartConfig**

该注解主要是为了辅助 Servlet 3.0 中 HttpServletRequest 提供的对上传文件的支持。该注解标注在 Servlet 上面，以表示该 Servlet 希望处理的请求的 MIME 类型是 multipart/form-data。另外，它还提供了若干属性用于简化对上传文件的处理。具体如下：

**表 5. @MultipartConfig 的常用属性**

| **属性名** | **类型** | **是否可选** | **描述** |
| --- | --- | --- | --- |
| fileSizeThreshold | int | 是 | 当数据量大于该值时，内容将被写入文件。 |
| location | String | 是 | 存放生成的文件地址。 |
| maxFileSize | long | 是 | 允许上传的文件最大值。默认值为 -1，表示没有限制。 |
| maxRequestSize | long | 是 | 针对该 multipart/form-data 请求的最大数量，默认值为 -1，表示没有限制。 |

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-lo-servlet30/#ibm-pcon)

**可插性支持**

如果说 3.0 版本新增的注解支持是为了简化 Servlet/ 过滤器 / 监听器的声明，从而使得 web.xml 变为可选配置， 那么新增的可插性 (pluggability) 支持则将 Servlet 配置的灵活性提升到了新的高度。熟悉 Struts2 的开发者都知道，Struts2 通过插件的形式提供了对包括 Spring 在内的各种开发框架的支持，开发者甚至可以自己为 Struts2 开发插件，而 Servlet 的可插性支持正是基于这样的理念而产生的。使用该特性，现在我们可以在不修改已有 Web 应用的前提下，只需将按照一定格式打成的 JAR 包放到 WEB-INF/lib 目录下，即可实现新功能的扩充，不需要额外的配置。

Servlet 3.0 引入了称之为“Web 模块部署描述符片段”的 web-fragment.xml 部署描述文件，该文件必须存放在 JAR 文件的 META-INF 目录下，该部署描述文件可以包含一切可以在 web.xml 中定义的内容。JAR 包通常放在 WEB-INF/lib 目录下，除此之外，所有该模块使用的资源，包括 class 文件、配置文件等，只需要能够被容器的类加载器链加载的路径上，比如 classes 目录等。

现在，为一个 Web 应用增加一个 Servlet 配置有如下三种方式 ( 过滤器、监听器与 Servlet 三者的配置都是等价的，故在此以 Servlet 配置为例进行讲述，过滤器和监听器具有与之非常类似的特性 )：

* 编写一个类继承自 HttpServlet，将该类放在 classes 目录下的对应包结构中，修改 web.xml，在其中增加一个 Servlet 声明。这是最原始的方式；
* 编写一个类继承自 HttpServlet，并且在该类上使用 @WebServlet 注解将该类声明为 Servlet，将该类放在 classes 目录下的对应包结构中，无需修改 web.xml 文件。
* 编写一个类继承自 HttpServlet，将该类打成 JAR 包，并且在 JAR 包的 META-INF 目录下放置一个 web-fragment.xml 文件，该文件中声明了相应的 Servlet 配置。web-fragment.xml 文件示例如下：

<?xml version="1.0" encoding="UTF-8"?>

<web-fragment

xmlns=http://java.sun.com/xml/ns/javaee

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="3.0"

xsi:schemaLocation="http://java.sun.com/xml/ns/javaee

http://java.sun.com/xml/ns/javaee/web-fragment\_3\_0.xsd"

metadata-complete="true">

<servlet>

<servlet-name>fragment</servlet-name>

<servlet-class>footmark.servlet.FragmentServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>fragment</servlet-name>

<url-pattern>/fragment</url-pattern>

</servlet-mapping>

</web-fragment>

从上面的示例可以看出，web-fragment.xml 与 web.xml 除了在头部声明的 XSD 引用不同之外，其主体配置与 web.xml 是完全一致的。

由于一个 Web 应用中可以出现多个 web-fragment.xml 声明文件，加上一个 web.xml 文件，加载顺序问题便成了不得不面对的问题。Servlet 规范的专家组在设计的时候已经考虑到了这个问题，并定义了加载顺序的规则。

web-fragment.xml 包含了两个可选的顶层标签，<name> 和 <ordering>，如果希望为当前的文件指定明确的加载顺序，通常需要使用这两个标签，<name> 主要用于标识当前的文件，而 <ordering> 则用于指定先后顺序。一个简单的示例如下：

<web-fragment...>

<name>FragmentA</name>

<ordering>

<after>

<name>FragmentB</name>

<name>FragmentC</name>

</after>

<before>

<others/>

</before>

</ordering>

...

</web-fragment>

如上所示， <name> 标签的取值通常是被其它 web-fragment.xml 文件在定义先后顺序时引用的，在当前文件中一般用不着，它起着标识当前文件的作用。

在 <ordering> 标签内部，我们可以定义当前 web-fragment.xml 文件与其他文件的相对位置关系，这主要通过 <ordering> 的 <after> 和 <before> 子标签来实现的。在这两个子标签内部可以通过 <name> 标签来指定相对应的文件。比如：

<after>

<name>FragmentB</name>

<name>FragmentC</name>

</after>

以上片段则表示当前文件必须在 FragmentB 和 FragmentC 之后解析。<before> 的使用于此相同，它所表示的是当前文件必须早于 <before> 标签里所列出的 web-fragment.xml 文件。

除了将所比较的文件通过 <name> 在 <after> 和 <begin> 中列出之外，Servlet 还提供了一个简化的标签 <others/>。它表示除了当前文件之外的其他所有的 web-fragment.xml 文件。该标签的优先级要低于使用 <name> 明确指定的相对位置关系。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-lo-servlet30/#ibm-pcon)

**ServletContext 的性能增强**

除了以上的新特性之外，ServletContext 对象的功能在新版本中也得到了增强。现在，该对象支持在运行时动态部署 Servlet、过滤器、监听器，以及为 Servlet 和过滤器增加 URL 映射等。以 Servlet 为例，过滤器与监听器与之类似。ServletContext 为动态配置 Servlet 增加了如下方法：

* ServletRegistration.Dynamic addServlet(String servletName,Class<? extends Servlet> servletClass)
* ServletRegistration.Dynamic addServlet(String servletName, Servlet servlet)
* ServletRegistration.Dynamic addServlet(String servletName, String className)
* <T extends Servlet> T createServlet(Class<T> clazz)
* ServletRegistration getServletRegistration(String servletName)
* Map<String,? extends ServletRegistration> getServletRegistrations()

其中前三个方法的作用是相同的，只是参数类型不同而已；通过 createServlet() 方法创建的 Servlet，通常需要做一些自定义的配置，然后使用 addServlet() 方法来将其动态注册为一个可以用于服务的 Servlet。两个 getServletRegistration() 方法主要用于动态为 Servlet 增加映射信息，这等价于在 web.xml( 抑或 web-fragment.xml) 中使用 <servlet-mapping> 标签为存在的 Servlet 增加映射信息。

以上 ServletContext 新增的方法要么是在 ServletContextListener 的 contexInitialized 方法中调用，要么是在 ServletContainerInitializer 的 onStartup() 方法中调用。

ServletContainerInitializer 也是 Servlet 3.0 新增的一个接口，容器在启动时使用 JAR 服务 API(JAR Service API) 来发现 ServletContainerInitializer 的实现类，并且容器将 WEB-INF/lib 目录下 JAR 包中的类都交给该类的 onStartup() 方法处理，我们通常需要在该实现类上使用 @HandlesTypes 注解来指定希望被处理的类，过滤掉不希望给 onStartup() 处理的类。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-lo-servlet30/#ibm-pcon)

**HttpServletRequest 对文件上传的支持**

此前，对于处理上传文件的操作一直是让开发者头疼的问题，因为 Servlet 本身没有对此提供直接的支持，需要使用第三方框架来实现，而且使用起来也不够简单。如今这都成为了历史，Servlet 3.0 已经提供了这个功能，而且使用也非常简单。为此，HttpServletRequest 提供了两个方法用于从请求中解析出上传的文件：

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

前者用于获取请求中给定 name 的文件，后者用于获取所有的文件。每一个文件用一个 javax.servlet.http.Part 对象来表示。该接口提供了处理文件的简易方法，比如 write()、delete() 等。至此，结合 HttpServletRequest 和 Part 来保存上传的文件变得非常简单，如下所示：

Part photo = request.getPart("photo");

photo.write("/tmp/photo.jpg");

// 可以将两行代码简化为 request.getPart("photo").write("/tmp/photo.jpg") 一行。

另外，开发者可以配合前面提到的 @MultipartConfig 注解来对上传操作进行一些自定义的配置，比如限制上传文件的大小，以及保存文件的路径等。其用法非常简单，故不在此赘述了。

需要注意的是，如果请求的 MIME 类型不是 multipart/form-data，则不能使用上面的两个方法，否则将抛异常。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-lo-servlet30/#ibm-pcon)

**总结**

Servlet 3.0 的众多新特性使得 Servlet 开发变得更加简单，尤其是异步处理特性和可插性支持的出现，必将对现有的 MVC 框架产生深远影响。虽然我们通常不会自己去用 Servlet 编写控制层代码，但是也许在下一个版本的 Struts 中，您就能切实感受到这些新特性带来的实质性改变。