**Servlet3.0中Servlet的使用**

**目录**

1.注解配置

2.异步调用

3.文件上传

       相对于之前的版本，Servlet3.0中的Servlet有以下改进：

  支持注解配置。

  支持异步调用。

  直接有对文件上传的支持。

       在这篇文章中我将主要讲这三方面的应用示例。

**1.注解配置**

       在以往我们的Servlet都需要在web.xml文件中进行配置（Servlet3.0同样支持），但是在Servlet3.0中引入了注解，我们只需要在对应的Servlet类上使用@WebServlet注解进行标记，我们的应用启动之后就可以访问到该Servlet。对于一个@WebServlet而言，有一个属性是必须要的，那就是它的访问路径。@WebServlet中有两个属性可以用来表示Servlet的访问路径，分别是value和urlPatterns。value和urlPatterns都是数组形式，表示我们可以把一个Servlet映射到多个访问路径，但是value和urlPatterns不能同时使用。如果同时使用了value和urlPatterns，我们的Servlet是无法访问到的。下面是一个使用@WebServlet的简单Servlet示例。

**Java代码  [收藏代码](javascript:void())**

2. **import** java.io.IOException;
4. **import** javax.servlet.ServletException;
5. **import** javax.servlet.annotation.WebServlet;
6. **import** javax.servlet.http.HttpServlet;
7. **import** javax.servlet.http.HttpServletRequest;
8. **import** javax.servlet.http.HttpServletResponse;
10. /\*\*
11. \*
12. \* Servlet3.0支持使用注解配置Servlet。我们只需在Servlet对应的类上使用@WebServlet进行标注，
13. \* 我们就可以访问到该Servlet了，而不需要再在web.xml文件中进行配置。@WebServlet的urlPatterns
14. \* 和value属性都可以用来表示Servlet的部署路径，它们都是对应的一个数组。
15. \*/
16. @WebServlet(name="exampleServlet", urlPatterns="/servlet/example")
17. **public** **class** ExampleServlet **extends** HttpServlet {
19. **private** **static** **final** **long** serialVersionUID = 1L;
21. @Override
22. **protected** **void** doGet(HttpServletRequest request,
23. HttpServletResponse response) **throws** ServletException, IOException {
24. **this**.doPost(request, response);
25. }
27. @Override
28. **protected** **void** doPost(HttpServletRequest request,
29. HttpServletResponse response) **throws** ServletException, IOException {
30. response.getWriter().write("Hello User.");
31. }
33. }

**初始化参数**

       使用@WebServlet时也可以配置初始化参数，它是通过@WebServlet的initParams参数来指定的。initParams是一个@WebInitParam数组，每一个@WebInitParam代表一个初始化参数。

**Java代码  [收藏代码](javascript:void())**

2. **import** java.io.IOException;
3. **import** java.util.Enumeration;
5. **import** javax.servlet.ServletException;
6. **import** javax.servlet.annotation.WebInitParam;
7. **import** javax.servlet.annotation.WebServlet;
8. **import** javax.servlet.http.HttpServlet;
9. **import** javax.servlet.http.HttpServletRequest;
10. **import** javax.servlet.http.HttpServletResponse;
12. /\*\*
13. \* 带初始化参数的Servlet
14. \* WebServlet的属性initParams可以用来指定当前Servlet的初始化参数，它是一个数组，
15. \* 里面每一个@WebInitParam表示一个参数。
16. \*/
17. @WebServlet(value="/servlet/init-param", initParams={@WebInitParam(name="param1", value="value1")})
18. **public** **class** WebInitParamServlet **extends** HttpServlet {
20. /\*\*
21. \*
22. \*/
23. **private** **static** **final** **long** serialVersionUID = 1L;
25. @Override
26. **protected** **void** doGet(HttpServletRequest req, HttpServletResponse resp)
27. **throws** ServletException, IOException {
28. **this**.doPost(req, resp);
29. }
31. @Override
32. **protected** **void** doPost(HttpServletRequest req, HttpServletResponse resp)
33. **throws** ServletException, IOException {
34. Enumeration<String> paramNames = **this**.getServletConfig().getInitParameterNames();
35. String paramName;
36. **while** (paramNames.hasMoreElements()) {
37. paramName = paramNames.nextElement();
38. resp.getWriter().append(paramName + " = " + **this**.getServletConfig().getInitParameter(paramName));
39. }
40. resp.getWriter().close();
41. }
43. }

**2.异步调用**

       在Servlet3.0中，在Servlet内部支持异步处理。它的逻辑是当我们请求一个Servlet时，我们的Servlet可以先返回一部分内容给客户端。然后在Servlet内部异步处理另外一段逻辑，等到异步处理完成之后，再把异步处理的结果返回给客户端。这意味着当我们的Servlet在处理一段比较费时的业务逻辑时，我们可以先返回一部分信息给客户端，然后异步处理费时的业务，而不必让客户端一直等待所有的业务逻辑处理完。等到异步处理完之后，再把对应的处理结果返回给客户端。

       异步调用是通过当前HttpServletRequest的startAsync()方法开始的，它返回一个AsyncContext。之后我们可以调用AsyncContext的start()方法来新起一个线程进行异步调用。在新线程内部程序的最后我们最好是调用一下当前AsyncContext的complete()方法，否则异步调用的结果需要等到设置的超时时间过后才会返回到客户端。另外当异步调用超时以后会接着调用异步任务，即新起的线程。

**Java代码  [收藏代码](javascript:void())**

2. **import** java.io.IOException;
3. **import** java.io.PrintWriter;
5. **import** javax.servlet.AsyncContext;
6. **import** javax.servlet.ServletException;
7. **import** javax.servlet.annotation.WebServlet;
8. **import** javax.servlet.http.HttpServlet;
9. **import** javax.servlet.http.HttpServletRequest;
10. **import** javax.servlet.http.HttpServletResponse;
12. /\*\*
13. \* 支持异步返回的Servlet
14. \* 对于Servlet的异步返回，首先我们必须指定@WebServlet的asyncSupported属性为true（默认是false），同时在它之前的Filter
15. \* 的asyncSupported属性也必须是true，否则传递过来的request就是不支持异步调用的。
16. \*
17. \*/
18. @WebServlet(value="/servlet/async", asyncSupported=**true**)
19. **public** **class** AsyncServlet **extends** HttpServlet {
21. /\*\*
22. \*
23. \*/
24. **private** **static** **final** **long** serialVersionUID = 1L;
26. @Override
27. **protected** **void** doGet(HttpServletRequest req, HttpServletResponse resp)
28. **throws** ServletException, IOException {
29. **this**.doPost(req, resp);
30. }
32. @Override
33. **protected** **void** doPost(HttpServletRequest req, HttpServletResponse resp)
34. **throws** ServletException, IOException {
35. resp.setContentType("text/plain;charset=UTF-8");
36. **final** PrintWriter writer = resp.getWriter();
37. writer.println("异步之前输出的内容。");
38. writer.flush();
39. //开始异步调用，获取对应的AsyncContext。
40. **final** AsyncContext asyncContext = req.startAsync();
41. //设置超时时间，当超时之后程序会尝试重新执行异步任务，即我们新起的线程。
42. asyncContext.setTimeout(10\*1000L);
43. //新起线程开始异步调用，start方法不是阻塞式的，它会新起一个线程来启动Runnable接口，之后主程序会继续执行
44. asyncContext.start(**new** Runnable() {
46. @Override
47. **public** **void** run() {
48. **try** {
49. Thread.sleep(5\*1000L);
50. writer.println("异步调用之后输出的内容。");
51. writer.flush();
52. //异步调用完成，如果异步调用完成后不调用complete()方法的话，异步调用的结果需要等到设置的超时
53. //时间过了之后才能返回到客户端。
54. asyncContext.complete();
55. } **catch** (Exception e) {
56. e.printStackTrace();
57. }
58. }
60. });
61. writer.println("可能在异步调用前输出，也可能在异步调用之后输出，因为异步调用会新起一个线程。");
62. writer.flush();
63. }
65. }

       对于一个Servlet如果要支持异步调用的话我们必须指定其asyncSupported属性为true（默认是false）。使用@WebServlet注解标注的Servlet我们可以直接指定其asyncSupported属性的值为true，如：

@WebServlet(value=”/servlet/async”, asyncSupported=true)。而对于在web.xml文件中进行配置的Servlet来说，我们需要在配置的时候指定其asyncSupported属性为true。

**Xml代码  [收藏代码](javascript:void())**

1. **<servlet>**
2. **<servlet-name>**xxx**</servlet-name>**
3. **<servlet-class>**xxx**</servlet-class>**
4. **<async-supported>**true**</async-supported>**
5. **</servlet>**
6. **<servlet-mapping>**
7. **<servlet-name>**xxx**</servlet-name>**
8. **<url-pattern>**xxx**</url-pattern>**
9. **</servlet-mapping>**

       Servlet的异步调用程序的关键是要调用当前HttpServletRequest的startAsync()方法。至于利用返回的AsyncContext来新起一个线程进行异步处理就不是那么的必须了，因为在HttpServletRequest startAsync()之后，我们可以自己新起线程进行异步处理。

**Java代码  [收藏代码](javascript:void())**

1. @WebServlet(value="/servlet/async", asyncSupported=**true**)
2. **public** **class** AsyncServlet **extends** HttpServlet {
4. /\*\*
5. \*
6. \*/
7. **private** **static** **final** **long** serialVersionUID = 1L;
9. @Override
10. **protected** **void** doGet(HttpServletRequest req, HttpServletResponse resp)
11. **throws** ServletException, IOException {
12. **this**.doPost(req, resp);
13. }
15. @Override
16. **protected** **void** doPost(HttpServletRequest req, HttpServletResponse resp)
17. **throws** ServletException, IOException {
18. resp.setContentType("text/plain;charset=UTF-8");
19. **final** PrintWriter writer = resp.getWriter();
20. writer.println("异步之前输出的内容。");
21. writer.flush();
22. //开始异步调用，获取对应的AsyncContext。
23. **final** AsyncContext asyncContext = req.startAsync();
24. //设置超时时间，当超时之后程序会尝试重新执行异步任务，即我们新起的线程。
25. asyncContext.setTimeout(10\*1000L);
26. Runnable r = **new** Runnable() {
27. @Override
28. **public** **void** run() {
29. **try** {
30. Thread.sleep(5\*1000L);
31. writer.println("异步调用之后输出的内容。");
32. writer.flush();
33. //异步调用完成
34. asyncContext.complete();
35. } **catch** (Exception e) {
36. e.printStackTrace();
37. }
38. }
39. };
40. Thread t = **new** Thread(r);
41. //开启自己的线程进行异步处理
42. t.start();
43. writer.println("可能在异步调用前输出，也可能在异步调用之后输出，因为异步调用会新起一个线程。");
44. writer.flush();
45. }
47. }

**异步调用监听器**

       当我们需要对异步调用做一个详细的监听的时候，比如监听它是否超时，我们可以通过给AsyncContext设置对应的监听器AsyncListener来实现这一功能。AsyncListener是一个接口，里面定义了四个方法，分别是针对于异步调用开始、结束、出错和超时的。

**Java代码  [收藏代码](javascript:void())**

1. **import** java.io.IOException;
2. **import** java.io.PrintWriter;
4. **import** javax.servlet.AsyncContext;
5. **import** javax.servlet.AsyncEvent;
6. **import** javax.servlet.AsyncListener;
7. **import** javax.servlet.ServletException;
8. **import** javax.servlet.annotation.WebServlet;
9. **import** javax.servlet.http.HttpServlet;
10. **import** javax.servlet.http.HttpServletRequest;
11. **import** javax.servlet.http.HttpServletResponse;
13. /\*\*
14. \* 支持异步返回的Servlet
15. \* 对于Servlet的异步返回，首先我们必须指定@WebServlet的asyncSupported属性为true（默认是false），同时在它之前的Filter
16. \* 的asyncSupported属性也必须是true，否则传递过来的request就是不支持异步调用的。
17. \*
18. \*/
19. @WebServlet(value="/servlet/async2", asyncSupported=**true**)
20. **public** **class** AsyncServlet2 **extends** HttpServlet {
22. /\*\*
23. \*
24. \*/
25. **private** **static** **final** **long** serialVersionUID = 1L;
27. @Override
28. **protected** **void** doGet(HttpServletRequest req, HttpServletResponse resp)
29. **throws** ServletException, IOException {
30. **this**.doPost(req, resp);
31. }
33. @Override
34. **protected** **void** doPost(HttpServletRequest req, HttpServletResponse resp)
35. **throws** ServletException, IOException {
36. resp.setContentType("text/plain;charset=UTF-8");
37. **final** PrintWriter writer = resp.getWriter();
38. writer.println("异步之前输出的内容。");
39. writer.flush();
40. //开始异步调用，获取对应的AsyncContext。
41. **final** AsyncContext asyncContext = req.startAsync();
42. //设置当前异步调用对应的监听器
43. asyncContext.addListener(**new** MyAsyncListener());
44. //设置超时时间，当超时之后程序会尝试重新执行异步任务，即我们新起的线程。
45. asyncContext.setTimeout(10\*1000L);
46. //新起线程开始异步调用，start方法不是阻塞式的，它会新起一个线程来启动Runnable接口，之后主程序会继续执行
47. asyncContext.start(**new** Runnable() {
49. @Override
50. **public** **void** run() {
51. **try** {
52. Thread.sleep(5\*1000L);
53. writer.println("异步调用之后输出的内容。");
54. writer.flush();
55. //异步调用完成
56. asyncContext.complete();
57. } **catch** (Exception e) {
58. e.printStackTrace();
59. }
60. }
62. });
63. writer.println("可能在异步调用前输出，也可能在异步调用之后输出，因为异步调用会新起一个线程。");
64. writer.flush();
65. }
67. /\*\*
68. \* 异步调用对应的监听器
69. \* @author Yeelim
70. \* @date 2014-2-8
71. \* @mail yeelim-zhang@todaytech.com.cn
72. \*/
73. **private** **class** MyAsyncListener **implements** AsyncListener {
75. @Override
76. **public** **void** onComplete(AsyncEvent event) **throws** IOException {
77. System.out.println("异步调用完成……");
78. event.getSuppliedResponse().getWriter().println("异步调用完成……");
79. }
81. @Override
82. **public** **void** onError(AsyncEvent event) **throws** IOException {
83. System.out.println("异步调用出错……");
84. event.getSuppliedResponse().getWriter().println("异步调用出错……");
85. }
87. @Override
88. **public** **void** onStartAsync(AsyncEvent event) **throws** IOException {
89. System.out.println("异步调用开始……");
90. event.getSuppliedResponse().getWriter().println("异步调用开始……");
91. }
93. @Override
94. **public** **void** onTimeout(AsyncEvent event) **throws** IOException {
95. System.out.println("异步调用超时……");
96. event.getSuppliedResponse().getWriter().println("异步调用超时……");
97. }
99. }
101. }

注：

       对于正常执行的异步调用而言上述代码中开始是没有监听到的，只有在异步调用超时，重新执行异步任务的时候才有监听到异步调用的开始。不过如果需要监听异步第一次开始的话，我们可以在异步调用开始的时候做相应的监听器监听到异步调用开始时需要做的内容。

**3.文件上传**

       在Servlet3.0中上传文件变得非常简单。我们只需通过request的getPart(String partName)获取到上传的对应文件对应的Part或者通过getParts()方法获取到所有上传文件对应的Part。之后我们就可以通过part的write(String fileName)方法把对应文件写入到磁盘。或者通过part的getInputStream()方法获取文件对应的输入流，然后再对该输入流进行操作。要使用request的getPart()或getParts()方法对上传的文件进行操作的话，有两个要注意的地方。首先，用于上传文件的form表单的enctype必须为multipart/form-data；其次，对于使用注解声明的Servlet，我们必须在其对应类上使用@MultipartConfig进行标注，而对于在web.xml文件进行配置的Servlet我们也需要指定其multipart-config属性，如：

**Xml代码  [收藏代码](javascript:void())**

1. **<servlet>**
2. **<servlet-name>**xxx**</servlet-name>**
3. **<servlet-class>**xxx.xxx**</servlet-class>**
4. **<multipart-config></multipart-config>**
5. **</servlet>**
6. **<servlet-mapping>**
7. **<servlet-name>**xxx**</servlet-name>**
8. **<url-pattern>**/servlet/xxx**</url-pattern>**
9. **</servlet-mapping>**

       不管是基于注解的@MultipartConfig，还是基于web.xml文件配置的multipart-config，我们都可以给它们设置几个属性。

  file-size-threshold：数字类型，当文件大小超过指定的大小后将写入到硬盘上。默认是0，表示所有大小的文件上传后都会作为一个临时文件写入到硬盘上。

  location：指定上传文件存放的目录。当我们指定了location后，我们在调用Part的write(String fileName)方法把文件写入到硬盘的时候可以，文件名称可以不用带路径，但是如果fileName带了绝对路径，那将以fileName所带路径为准把文件写入磁盘。

  max-file-size：数值类型，表示单个文件的最大大小。默认为-1，表示不限制。当有单个文件的大小超过了max-file-size指定的值时将抛出IllegalStateException异常。

  max-request-size：数值类型，表示一次上传文件的最大大小。默认为-1，表示不限制。当上传时所有文件的大小超过了max-request-size时也将抛出IllegalStateException异常。

上面的属性是针对于web.xml中配置Servlet而言的，其中的每一个属性都对应了multipart-config元素下的一个子元素。对于基于注解配置的Servlet而言，@MultipartConfig的属性是类型的，我们只需把上述对应属性中间的杠去掉，然后把对应字母大写即可，如maxFileSize。

       下面给出Servlet3.0中文件上传的一个示例。

**Html：**

**Html代码  [收藏代码](javascript:void())**

1. **<form** method="post" action="servlet/upload" enctype="multipart/form-data"**>**
2. **<input** type="file" name="upload"**/>**
3. **<input** type="submit" value="upload"**/>**
4. **</form>**

**对应Servlet：**

**Java代码  [收藏代码](javascript:void())**

1. @WebServlet("/servlet/upload")
2. @MultipartConfig
3. **public** **class** FileUploadServlet **extends** HttpServlet {
5. /\*\*
6. \*
7. \*/
8. **private** **static** **final** **long** serialVersionUID = 1L;
10. @Override
11. **protected** **void** doPost(HttpServletRequest req, HttpServletResponse resp)
12. **throws** ServletException, IOException {
13. req.setCharacterEncoding("UTF-8");
14. Part part = req.getPart("upload");
15. //格式如：form-data; name="upload"; filename="YNote.exe"
16. String disposition = part.getHeader("content-disposition");
17. System.out.println(disposition);
18. String fileName = disposition.substring(disposition.lastIndexOf("=")+2, disposition.length()-1);
19. String fileType = part.getContentType();
20. **long** fileSize = part.getSize();
21. System.out.println("fileName: " + fileName);
22. System.out.println("fileType: " + fileType);
23. System.out.println("fileSize: " + fileSize);
24. String uploadPath = req.getServletContext().getRealPath("/upload");
25. System.out.println("uploadPath" + uploadPath);
26. part.write(uploadPath + File.separator +fileName);
27. }
29. }

       对于Servlet3.0中的文件上传还有一个需要注意的地方，当我们把Part写入到硬盘以后，我们原先的Part（也就是之前的临时文件）可能已经删了，这个时候如果我们再次去访问Part的内容的话，那它就是空的，系统会抛出异常说找不到对应的文件。