
Java EE7 Web

非同步處理

鄭安翔

ansel_cheng@hotmail.com

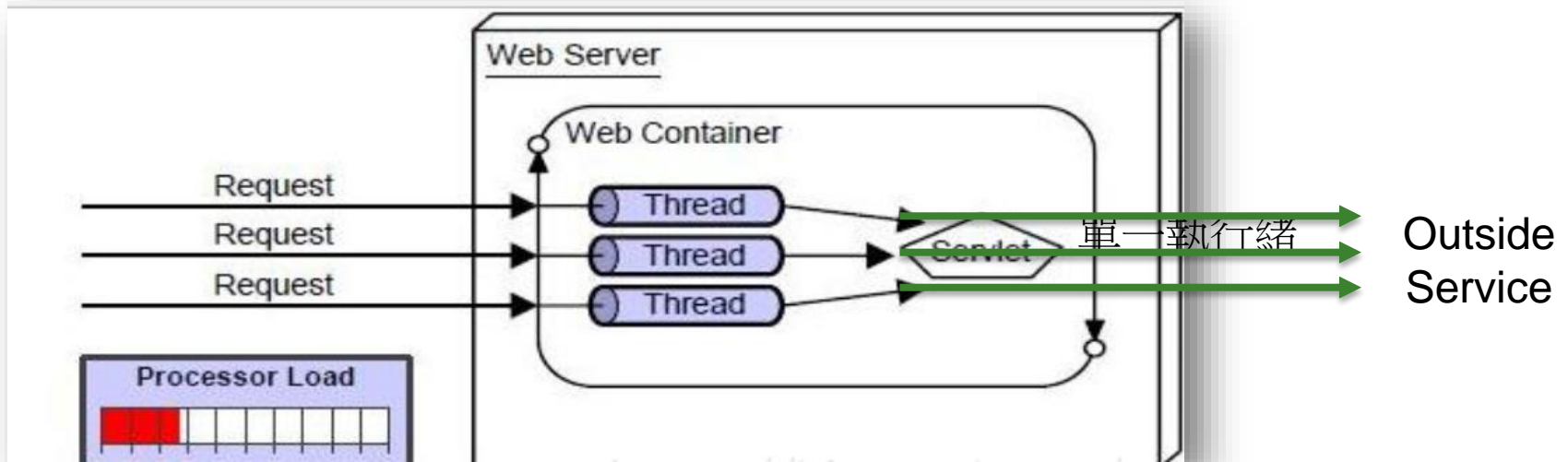
課程大綱

- 1) 非同步**Servlet**的機制
- 2) 非同步**JavaScript** 客戶端程式
- 3) 服務器推送 **Server-Push**

Servlet 多工處理架構

■ Servlet 多工處理架構

- ❑ 相同請求共用容器中同一個 **Servlet** 物件
- ❑ 每個請求由一個執行緒(**Thread**)負責處理
- ❑ 當處理程序時間過長時，後面的相同請求需要等待執行完成之後，釋放**Servlet**物件，才能執行



同步執行長時間Servlet

```
LongTaskServlet.java
Source History
1 package com.gjun;
2
3 import ...8 lines
11
12 @WebServlet("/LongTask")
13 public class LongTaskServlet extends HttpServlet {
14
15     @Override
16     protected void doGet(HttpServletRequest req, HttpServletResponse resp)
17         throws ServletException, IOException {
18         resp.setContentType("text/html;charset=UTF-8");
19         PrintWriter out = resp.getWriter();
20         out.print("1.進入Servlet的時間： " + new Date() + " <br> <br>");
21
22         // 商業邏輯！
23         longTask(out);
24
25         out.print("3.結束Servlet的時間： " + new Date() + " <br> <br>");
26     }
27
28     private void longTask(PrintWriter out) {
29         try {
30             Thread.sleep(10000);
31             out.print("2.任務處理完畢的時間： " + new Date() + " <br> <br>");
32         } catch (Exception e) {
33         }
34     }
35
36 }
```

localhost:8080/AsyncDemo/Lo x +

localhost:8080/Asyn... Q ☆ S :

- 1.進入Servlet的時間： Thu Dec 24 21:56:54 CST 2020
- 2.任務處理完畢的時間： Thu Dec 24 21:57:04 CST 2020
- 3.結束Servlet的時間： Thu Dec 24 21:57:04 CST 2020

localhost:8080/AsyncDemo/Lo x +

localhost:8080/Asyn... Q ☆ S :

- 1.進入Servlet的時間： Thu Dec 24 21:57:04 CST 2020
- 2.任務處理完畢的時間： Thu Dec 24 21:57:14 CST 2020
- 3.結束Servlet的時間： Thu Dec 24 21:57:14 CST 2020

localhost:8080/AsyncDemo/Lo x +

localhost:8080/Asyn... Q ☆ S :

- 1.進入Servlet的時間： Thu Dec 24 21:57:14 CST 2020
- 2.任務處理完畢的時間： Thu Dec 24 21:57:24 CST 2020
- 3.結束Servlet的時間： Thu Dec 24 21:57:24 CST 2020

非同步Servlet的機制

■ Asynchronous 非同步 Servlet的機制

□ Java EE 6 新增

□ 避免需長時間處理的servlet,造成應用程式效能負擔

- 網路容器會為每個請求分配一個執行緒,執行期間(請求資訊擷取、商業邏輯運算、產生回應), **Servlet**物件不會被釋放
- 某些請求的商業邏輯需要長時間處理(ex:長時間運算、等待某個資源), 會長時間佔用**Servlet**物件
- 這類的請求過多時,系統無法回應其他請求

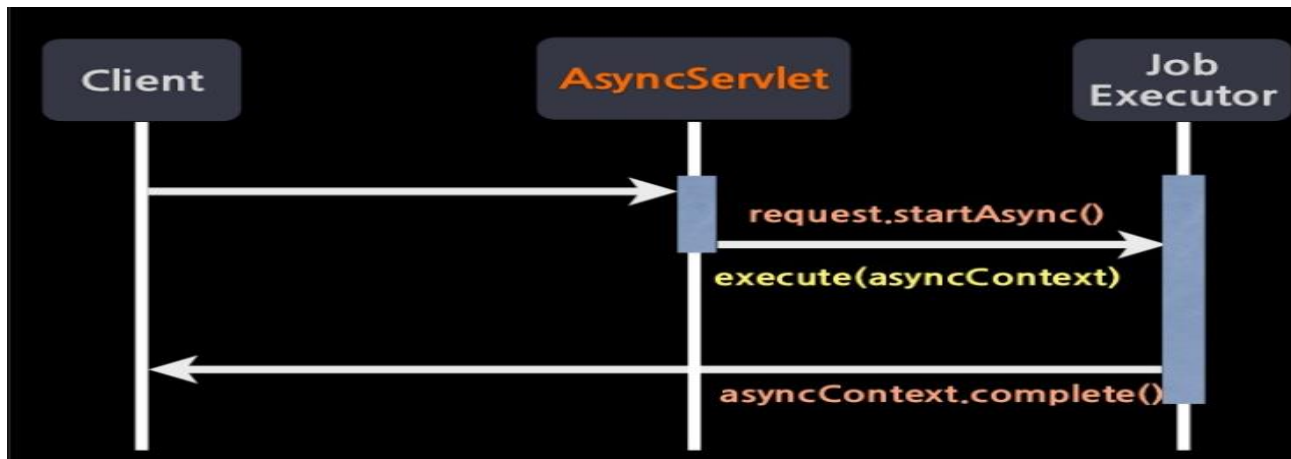
□ 與AJAX技術不同,但可能與其同時使用

- Asynchronous JavaScript And XML (AJAX)

非同步Servlet的機制

- **Asynchronous 非同步 Servlet的運作流程**
 - 資訊擷取、商業邏輯、產生回應由不同執行緒執行
 - 請求資訊擷取後,釋放容器分配的**Servlet**執行緒
 - 由另一執行緒物件處理商業邏輯
 - 以請求及回應物件建立一**AsyncContext**物件,供執行緒使用
 - 商業邏輯運算
 - 建立一個**Runnable**的物件,排入一個執行緒池（**Thread pool**）
 - 通常執行緒池的執行緒數量是固定的,這些須長時間處理的請求，在這些有限數量的執行緒中完成
 - 長時間運算完成或等待資源已獲得,轉送至**JSP view**元件產生回應
 - 使用另外的**JSP**產生回應, 符合**MVC**架構

非同步Servlet的機制



AsyncContext

<<interface>> javax.servlet.ServletRequest

```
getAsyncContext() : AsyncContext
getAttribute(name : String) : Object
getAttributeNames() : Enumeration
getContentTypeLength() : int
getContentType() : String
getInputStream() : ServletInputStream
getLocale() : java.util.Locale
getLocales() : Enumeration
getParameter(name : String) : String
getParameterNames() : Enumeration
getParameterValues(name : String) : String[]
getProtocol() : String
getReader() : java.io.BufferedReader
getServerName() : String
getServerPort() : int
isSecure() : boolean
removeAttribute(name : String)
setAttribute(name : String, Object o)
startAsync() : AsyncContext
startAsync(ServletRequest servletRequest,
ServletResponse servletResponse)
: AsyncContext
```

<<interface>> javax.servlet.AsyncContext

```
ASYNC_CONTEXT_PATH
ASYNC_PATH_INFO
ASYNC_QUERY_STRING
ASYNC_REQUEST_URI
ASYNC_SERVLET_PATH

addListener(AsyncListener listener)
addListener(listener:AsyncListener, request:
    ServletRequest, response:ServletResponse )
complete()
createListener(clazz:Class<T> ) : <T extends
    AsyncListener>

dispatch()
dispatch(context: ServletContext , path:String)
dispatch(path:String)
getRequest() : ServletRequest
getResponse() : ServletResponse
getTimeout() : long
hasOriginalRequestAndResponse() : boolean
start(run : Runnable)
```


AsyncContext

ServletRequest 方法	回傳型態	用途說明
getAsyncContext()	AsyncContext	取得由最近的startAsync()方法建立的AsyncContext物件
startAsync()	AsyncContext	啟動非同步模式, 以當前的ServletRequest及ServletResponse物件建立AsyncContext
startAsync(request:ServletRequest, response:ServletResponse)	AsyncContext	啟動非同步模式, 以指定的ServletRequest及ServletResponse物件建立AsyncContext

AsyncContext 方法	回傳型態	用途說明
start(run : Runnable)	void	容器啟動一個執行緒執行指定的Runnable
getRequest() :	ServletRequest	取得請求物件
getResponse()	ServletResponse	取得回應物件
complete()	void	非同步操作回應完成,關閉當前AsyncContext的請求及回應物件
dispatch()	void	非同步操作完成,派送AsyncContext的請求和回應物件至servlet容器
dispatch(path:String)	void	非同步操作完成,派送AsyncContext的請求和回應物件至指定路徑

非同步Servlet的設定/標註

- 使用Annotation標註非同步Servlet

`@WebServlet(urlPatterns = "/xxx", asyncSupported = true)`

- 使用web.xml設定非同步Servlet

- `<async-supported>`標籤為true

```
<servlet>
  <servlet-name>AsyncServlet</servlet-name>
  <servlet-class>controller.AsyncServlet</servlet-class>
  <async-supported>true</async-supported>
</servlet>
```

Forwarding and Filtering

- 非同步處理完成後,可轉送至其他Servlet / JSP
 - 派送目標不一定要是非同步Servlet
 - 回應頁面可以顯示同步或非同步Servlet的回應
- 非同步處理也可以驅動過濾器
 - 使用Annotation標註設定
 - @WebFilter(urlPatterns = "/xxx", asyncSupported = true)
 - 使用web.xml設定

```
<filter>
  <filter-name>AsyncFilter</filter-name>
  <filter-class>web.AsyncFilter</filter-class>
  <async-supported>true</async-supported>
</filter>
```

非同步執行

localhost:8080/AsyncDemo/As...

1.進入Servlet的時間：Thu Dec 24 22:05:44 CST 2020

3.結束Servlet的時間：Thu Dec 24 22:05:44 CST 2020

2.任務處理完畢的時間：Thu Dec 24 22:05:54 CST 2020

localhost:8080/AsyncDemo/As...

1.進入Servlet的時間：Thu Dec 24 22:05:46 CST 2020

3.結束Servlet的時間：Thu Dec 24 22:05:46 CST 2020

2.任務處理完畢的時間：Thu Dec 24 22:05:56 CST 2020

localhost:8080/AsyncDemo/As...

1.進入Servlet的時間：Thu Dec 24 22:05:47 CST 2020

3.結束Servlet的時間：Thu Dec 24 22:05:47 CST 2020

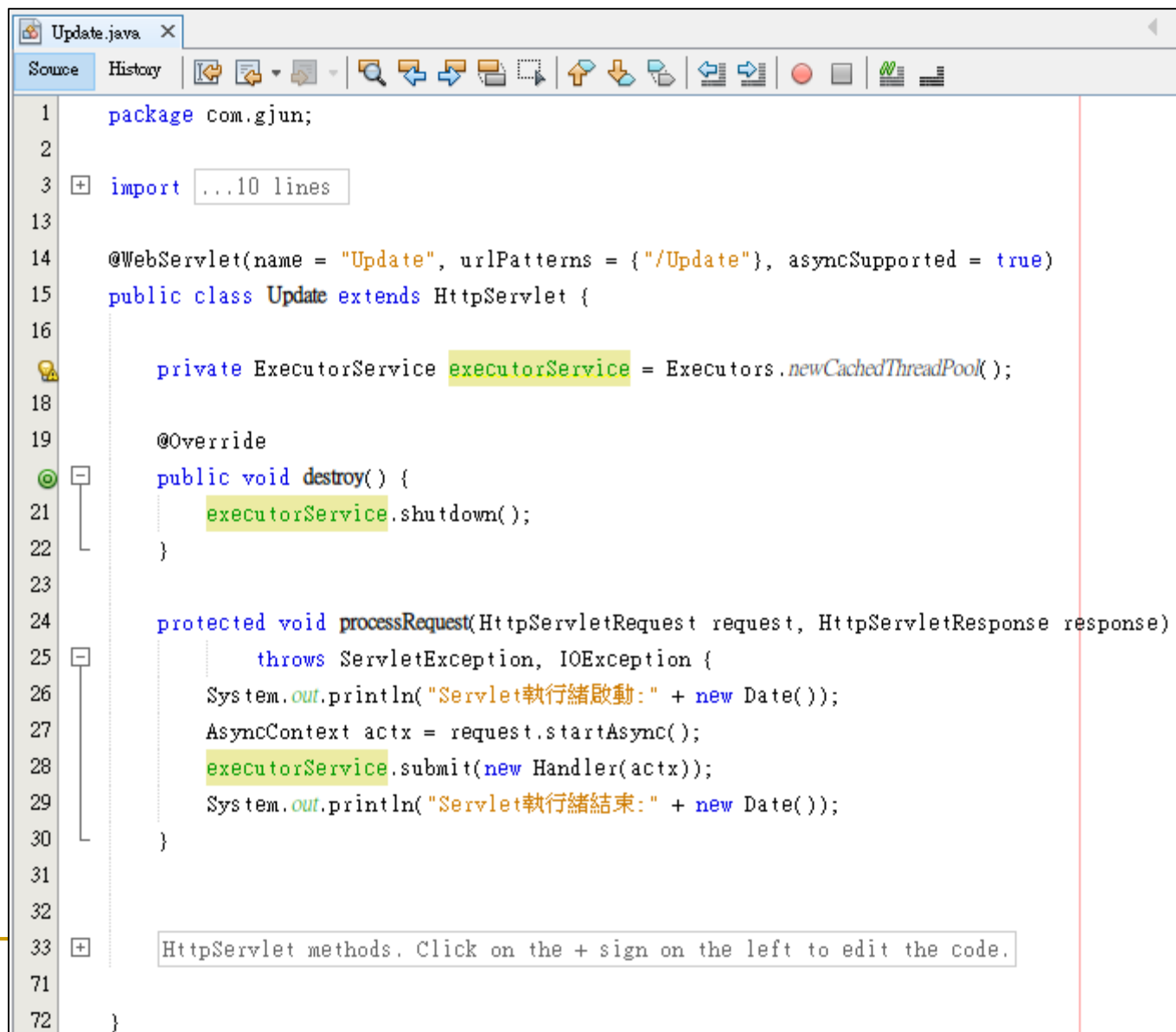
2.任務處理完畢的時間：Thu Dec 24 22:05:57 CST 2020

```
Source History
@WebServlet(urlPatterns = "/AsyncTask", asyncSupported = true)
public class AsyncTaskServlet extends HttpServlet {
    @Override
    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws IOException, ServletException {
        // 1.進入 Servlet
        resp.setContentType("text/html;charset=UTF-8");
        PrintWriter out = resp.getWriter();
        out.println("1.進入Servlet的時間：" + new Date() + "<br><br>");
        // 2.在子執行緒中執行任務調用，並由其負責輸出響應，主執行緒退出
        AsyncContext ctx = req.startAsync();
        new Thread(new AsyncTask(ctx)).start();

        out.println("3.結束Servlet的時間：" + new Date() + "<br><br>");
        out.flush();
        // 3.離開 Servlet 給其他請求連線使用
    }
}

class AsyncTask implements Runnable {
    private AsyncContext ctx = null;
    public AsyncTask(AsyncContext ctx){
        this.ctx = ctx;
    }
    public void run(){
        try {
            // 等待10秒鐘，用來模擬任務所需要的時間
            Thread.sleep(10000);
            PrintWriter out = ctx.getResponse().getWriter();
            out.println("2.任務處理完畢的時間：" + new Date() + "<br><br>");
            ctx.complete(); // 任務完成
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Asynchronous Servlet Example



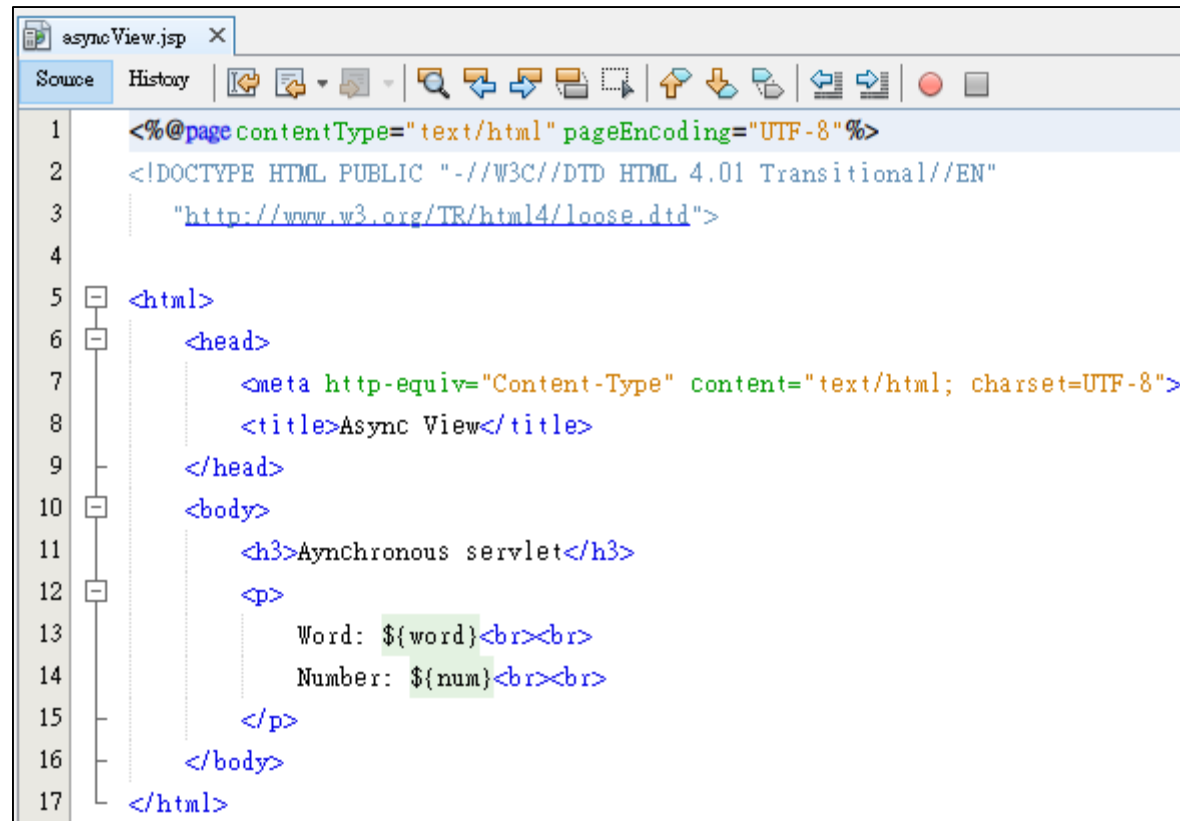
The screenshot shows an IDE window titled "Update.java" with a toolbar and a source code editor. The code is as follows:

```
1 package com.gjun;
2
3 import ...10 lines
13
14 @WebServlet(name = "Update", urlPatterns = {"/Update"}, asyncSupported = true)
15 public class Update extends HttpServlet {
16
17     private ExecutorService executorService = Executors.newCachedThreadPool();
18
19     @Override
20     public void destroy() {
21         executorService.shutdown();
22     }
23
24     protected void processRequest(HttpServletRequest request, HttpServletResponse response)
25         throws ServletException, IOException {
26         System.out.println("Servlet執行緒啟動:" + new Date());
27         AsyncContext actx = request.startAsync();
28         executorService.submit(new Handler(actx));
29         System.out.println("Servlet執行緒結束:" + new Date());
30     }
31
32     HttpServlet methods. Click on the + sign on the left to edit the code.
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72 }
```

Handler Class Implementation

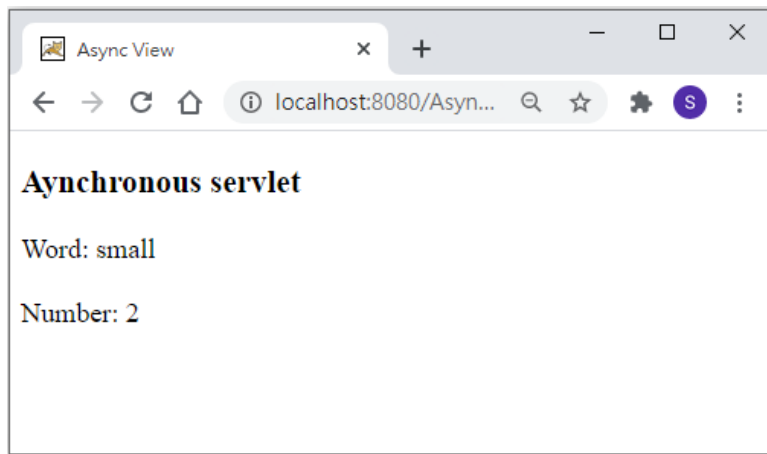
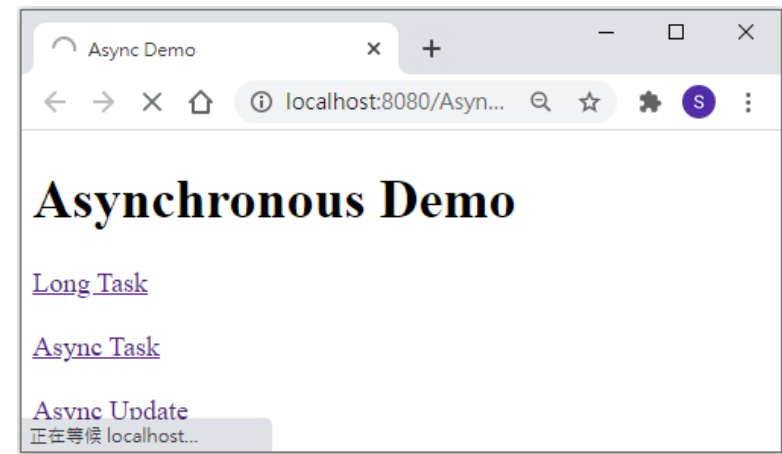
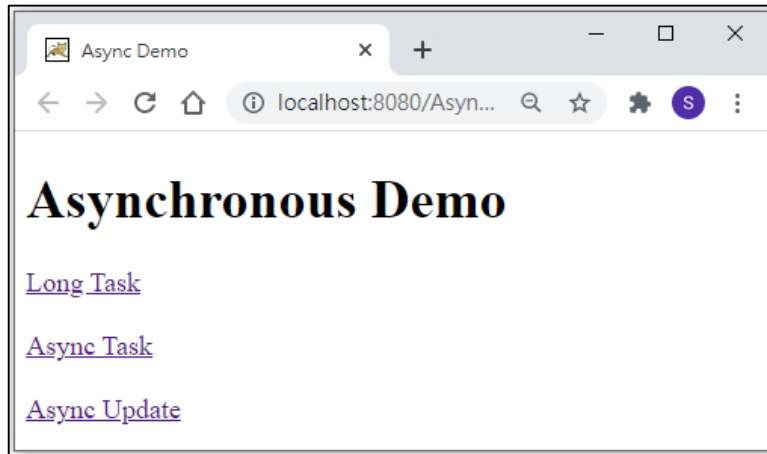
```
Handler.java x
Source History
7 public class Handler implements Runnable {
8
9     private AsyncContext actx;
10    private static final String[] words = {
11        "long", "short", "big", "small", "clever", "foolish", "tidy", "disorganized"
12    };
13
14    public Handler(AsyncContext actx) {
15        this.actx = actx;
16    }
17
18    @Override
19    public void run() {
20        System.out.println("非同步執行緒啟動:" + new Date());
21        try {
22            Thread.sleep(5000 + ((int) (Math.random() * 5000)));
23        } catch (InterruptedException ex) {
24            ex.printStackTrace();
25        }
26        ServletRequest request = actx.getRequest();
27        String word = words[(int) (Math.random() * words.length)];
28        int num = (int)(Math.random()*100);
29        request.setAttribute("word", word);
30        request.setAttribute("num", num);
31        System.out.println("非同步執行緒結束:" + new Date());
32        actx.dispatch("/asyncView.jsp");
33    }
34
35 }
```

Response JSP page



```
1 <%@page contentType="text/html" pageEncoding="UTF-8"%>
2 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
3   "http://www.w3.org/TR/html4/loose.dtd">
4
5 <html>
6   <head>
7     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
8     <title>Async View</title>
9   </head>
10  <body>
11    <h3>Asynchronous servlet</h3>
12    <p>
13      Word: ${word}<br><br>
14      Number: ${num}<br><br>
15    </p>
16  </body>
17 </html>
```

Asynchronous Example



Asynchronous Listeners

■ javax.servlet.AsyncListener

- 監聽AsyncContext的啟始、完成、逾時或錯誤事件

AsyncListener方法	回傳型態	用途說明
onComplete(event: AsyncEvent)	void	當非同步操作完成時呼叫
onError(event: AsyncEvent)	void	當非同步操作發生錯誤時呼叫
onTimeout(event: AsyncEvent)	void	當非同步操作Timeout時呼叫
onStartAsync(event: AsyncEvent)	void	當非同步操作啟動時呼叫

■ 註冊AsyncListener

AsyncContext 方法	回傳型態	用途說明
addListener(AsyncListener)	void	註冊指定AsyncListener監聽非同步AsyncContext

- 執行時依Listener註冊順序

課程大綱

- 1) 非同步Servlet的機制
- 2) 非同步JavaScript 客戶端程式
 - JavaScript 語法
 - AJAX
- 3) 服務器推送Server-Push

Asynchronous JavaScript Clients

- HTTP是基於請求、回應模型
 - 使用者向**Web**伺服器發送一個請求
 - 伺服器接收並處理資訊
 - 送回一個新的網頁
 - 想要獲得伺服端應用程式的最新狀態，使用者必須定期(或不定期)發送請求，浪費了許多頻寬

Asynchronous JavaScript Clients

■ JavaScript

- 主要被作為客戶端腳本語言，在客戶端由瀏覽器上執行
- 不需要伺服器的支援
- 可以在客戶端直接修改頁面顯示內容
 - 網頁中的JavaScript可以發出網路請求, 使用其回應中的資訊更新網頁
- 伺服器和瀏覽器之間交換的資料大量減少
 - 約只有原來的5%
- 降低網路伺服器的負荷
 - 很多的處理工作可以在發出請求的客戶端機器上完成

常用JavaScript語法

■ 常用JavaScript語法

- 標籤id屬性定義元素名稱
- document.getElementById(*"name"*)取得頁面中元素
- innerHTML :動態指定/取得網頁上元素內的HTML
- 函式(function)定義重複執行的內容
- 常用事件

事件	事件說明
onChange	使用者改變某一欄的內容
onClick	使用者按下某個按鈕
onLoad	某一頁完全載入
onSubmit	使用者確定送出某表單
onDbIcClick	連續按兩下滑鼠
onKeyDown	按下鍵盤
onKeyPress	按下鍵盤後放開
onKeyUp	放開鍵盤

AJAX 機制

■ AJAX(Asynchronous JavaScript And XML)

- 非同步JavaScript結合XML來交換結構化資料
- 瀏覽器中可使用下列方法建立非同步物件
 - XMLHttpRequest : Internet Explorer 7以後
new XMLHttpRequest();
 - ActiveXObject: Internet Explorer 6以前的版本
new ActiveXObject('Microsoft.XMLHTTP');
- 非同步物件常用方法

方法	回傳值	用途說明
open(string method, string url)	void	開啟對伺服端的連結, Method可為'GET'、'POST'、'HEAD', url為伺服端位址
send(content)	void	對伺服端傳送請求, open()的方法為'GET'時, content為null 方法為'POST'時, content可放字串、XML、JSON格式的 內容在POST本體中發送。

AJAX 機制

- AJAX(Asynchronous JavaScript And XML)
 - 非同步事件
 - `onreadystatechange` : 當狀態變化，呼叫所設置的處理器函式
 - 非同步物件的 `readyState` 數值, 代表處理階段：
 - 0 - 還沒呼叫 `open()`
 - 1 - 已呼叫 `open()`
 - 2 - 已呼叫 `send()`
 - 3 - 正在接收回應
 - 4 - 伺服器端回應結束
 - 非同步物件的 `status` 數值: 表示 HTTP 回應狀態碼
 - AJAX 系統中回應 Response 形式
 - HTML 片段
 - XML
 - JSON (JavaScript Object Notation)

Simple Asynchronous Client Example

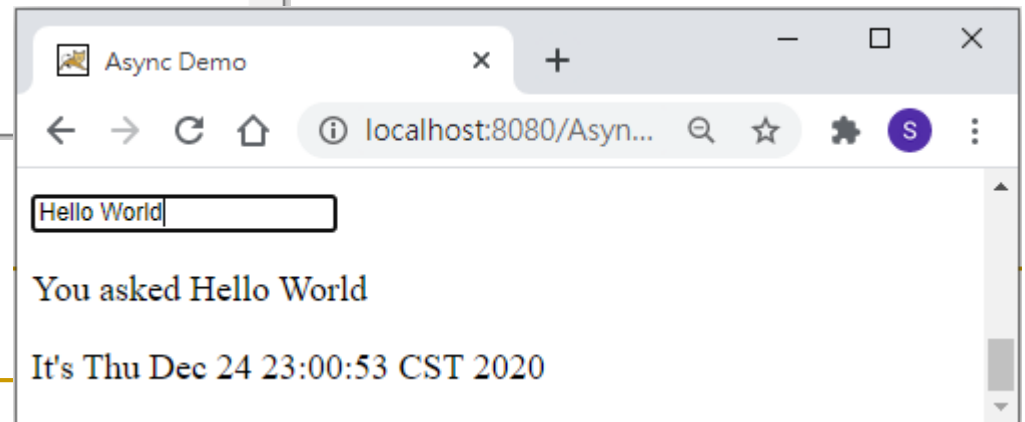
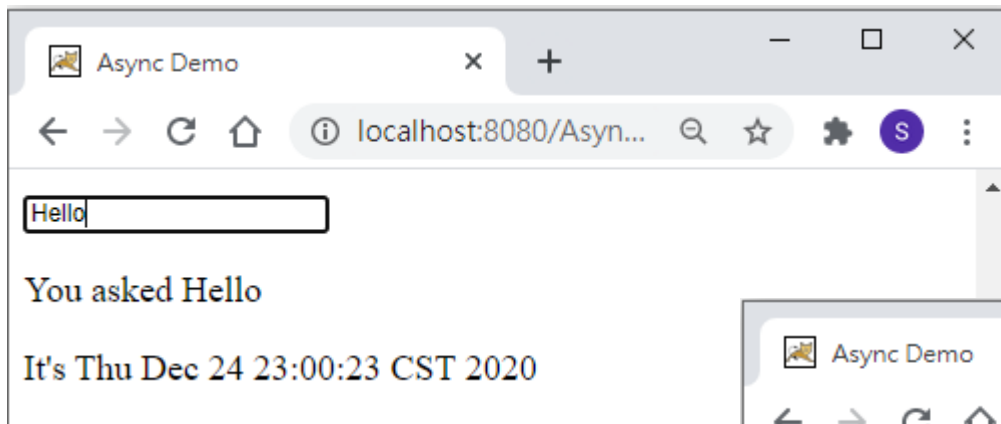
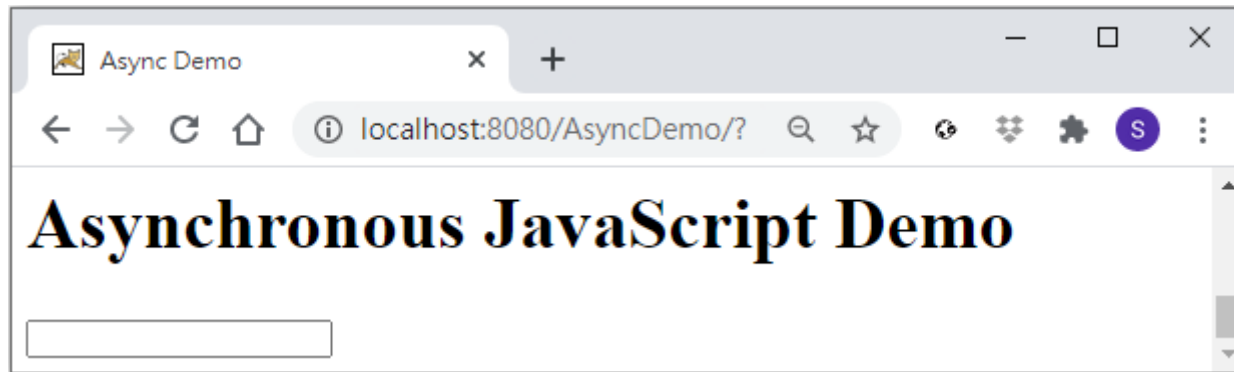
```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
```

```
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>Async JavaScript Page</title>
</head>
<body>
  <h1>Asynchronous JavaScript Page</h1>
  <form>
    <input id="inputField" type="text" onkeyup="doUp
  </form>
  <div id="wisdom">
  </div>
```

```
<script>
  wisdomTag = document.getElementById("wisdom");
  inputTag = document.getElementById("inputField");
  function doUpdate() {
    var req;
    if (window.XMLHttpRequest) {
      req = new XMLHttpRequest();
    } else if (window.ActiveXObject) {
      req = new ActiveXObject("Microsoft.XMLHTTP");
    } else {
      alert("AJAX not supported");
    }
    text = inputTag.value;
    req.open("GET", "update.jsp?text=" + text, true);
    req.send(null);
    req.onreadystatechange = function() {
      if (req.readyState == 4 && req.status == 200) {
        wisdomTag.innerHTML = req.responseText;
      }
    }
  }
</script>
</body>
</html>
```

```
<p>You asked ${param["text"]}</p>
<p>It's <%= new java.util.Date() %> </p>
```


Asynchronous Client Example



課程大綱

- 1) 非同步Servlet的機制
- 2) 非同步JavaScript 客戶端程式
- 3) 服務器推送 **Server-Push**

服務器推送(Server-Push)

- **Server-Push 服務器推送機制**
 - 結合非同步Servlet及非同步JavaScript技術
 - Servlet 3.0非同步處理技術解決請求佔用執行緒的問題
 - 瀏覽器端AJAX非同步技術讓畫面在等待期間是可操作的
 - 回應發送之後，由JavaScript進行頁面內容更新
 - 提供類似伺服器端主動通知瀏覽器的機制

Lab

- 建立AsyncTest專案
- 新增Comment.java Servlet
 - 使用標註設定Servlet支援非同步操作
 - 宣告一字符串陣列屬性comment
 - 內容為 {"wonderful", "unexpected", "strange", "elegant"}
 - 修改processRequest()方法
 - 啟動非同步操作,取得一AsyncContext物件
 - 宣告一區域常數參考AsyncContext物件
 - 啟動一個執行緒執行指定的Runnable物件

Lab

- 以匿名類別方式建立Runnale物件, run()內容如下:
 - 由AsyncContext物件取得Request,Response物件
 - 暫停2~15秒
 - 由comment陣列中隨機取得一字串串回
 - 呼叫AsyncContext物件complete(),完成非同步操作回應
- 測試Update非同步Servlet
 - 執行 `http://localhost:8080/AsyncTest/Update`

Lab

- 編輯index.jsp

- HTML body中輸入

<h3>Hello!</h3>

<p>Oh my, that was... let me think...!</p>

- 建立JavaScript區段在</body>之後

- 宣告兩個變數 req 及 toUpdate
 - toUpdate使用document.getElementById()取得‘adjective’標籤值
 - 建構非同步物件req：使用XMLHttpRequest或ActiveXObject

Lab

- 宣告sendRequest(),其中宣告onreadystatechange事件處理
 - 當readystatechange狀態值改變時,檢查readystatechange是否為4, status是否為200
 - 以非同步物件req的回應文字(responseText),作為toUpdate元素('adjective'標籤)的HTML內容
 - 發送一個非同步伺服器請求
 - open()方法 : HTTP方法為Get,URL為Update
 - send()方法
- 呼叫sendRequest()方法
- 測試、執行