什么是servlet

servlet是一门用于开发动态web资源的技术，可以运行在Web服务器中的小型Java程序，有时也叫做服务器端的小应用程序。servlet可以通过 HTTP协议接收和响应来自 Web 客户端的请求。

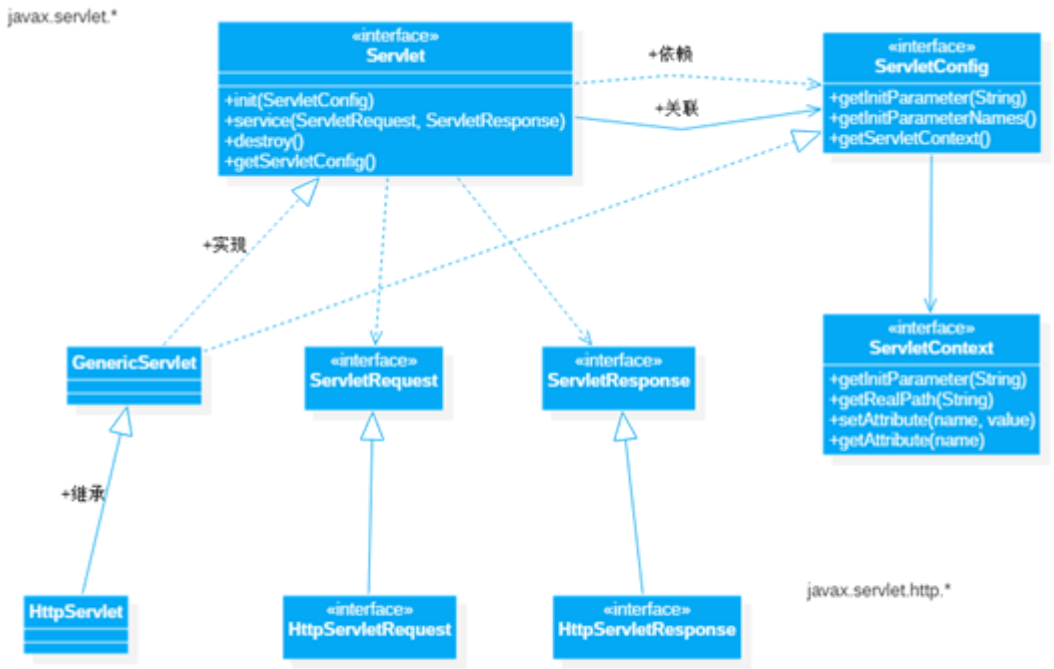
Servlet生命周期**

初始化阶段------调用**init()**方法

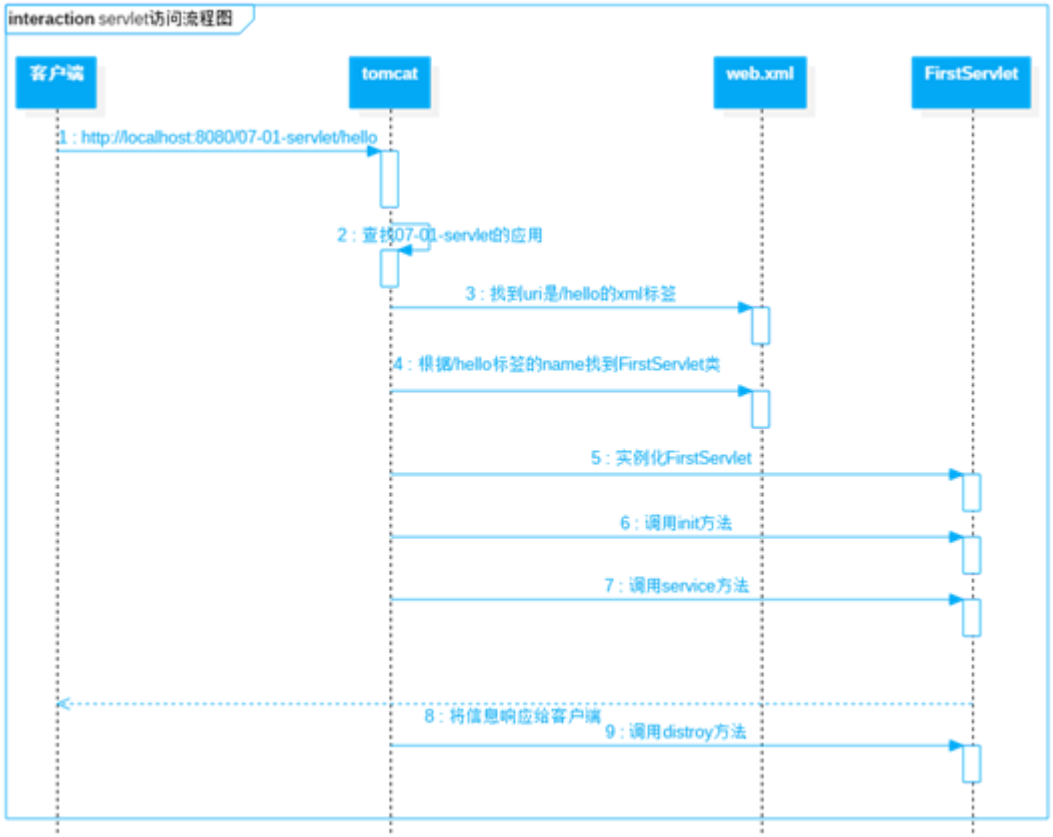
响应客户请求阶段------调用**service()**方法

终止阶段------调用**destroy()**方法

Servlet类**



Servlet工作流程**



Servlet的特征**

Servlet是单例多线程的，只创建一个servlet对象，但是每次请求都会起一个线程并在自己线程栈内存中执行service方法。为了保证其线程安全性，一般情况下是**不建议在 Servlet类中定义可修改的成员变量**，因为每个线程均可修改这个成员变量，会出现线程安全问题。

一个 Servlet 实例只会执行一次无参构造器与 init()方法，并且是在第一次访问时执行。用户每提交一次对当前 Servlet 的请求，就会执行一次 service()方法。一个Servlet 实例只会执行一次destroy()方法，在应用停止时执行。

默认情况下，Servlet在Web容器启动时是不会被实例化的。

Servlet主要负责接收用户请求HttpServletRequest,在doGet(),doPost()中做相应的处理，并将回应HttpServletResponse反馈给用户。Servlet可以设置初始化参数，供Servlet内部使用。

servlet的需要部署在tomcat中才能运行，有时tomcat也被称为是servlet的容器。**

Servlet与线程安全**

Servlet不是线程安全的，多线程并发的读写会导致数据不同步的问题。解决的办法是尽量不要定义成员变量，而是要把成员变量分别定义在doGet()和doPost()方法内。虽然使用synchronized语句块可以解决问题，但是会造成线程的等待，不是很科学的办法。 注意：多线程的并发的读写Servlet类属性会导致数据不同步。但是如果只是并发地读取属性而不写入，则不存在数据不同步的问题。因此**Servlet里的只读属性最好定义为final类型的。**

创建servlet的三种方式

定一个类实现javax.servlet.Servlet接口

```
public void destroy() {}

public ServletConfig getServletConfig() {
    return null;
}

public String getServletInfo() {
    return null;
}

public void init(ServletConfig arg0) throws ServletException {}

@Override
public void service(ServletRequest arg0, ServletResponse arg1) throws ServletException, IOException {}
```

定义一个类继承javax.servet.GenericServlet类

```
public void service(ServletRequest req, ServletResponse res) throws ServletException, IOException {}
```

该类是一个抽象类，实现了servlet接口和ServletConfig接口并重写了除了service方法以外的全部方法，这样子类在继承GenericServlet类时，只需重写service方法。

定义一个类继承javax.servlet.http.HttpServlet类

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
}

protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
    doPost(req, resp);
}
```

实际开发中经常使用继承HttpServlet类的方式创建一个servlet

HttpServlet*子类不需要重写service方法，倘若重写了该方法后可能会导致所编写的Servlet无法正常工作。**

GenericServlet*类为什么有两个init方法**

如果在子类中想要调用init方法的话，需要重写init方法，但这个重写的init(ServletConfig)方法必须要调用父类的init(ServletConfig) 方法，即在第一句必须写上super.init(config)；否则将无法获取到ServletConfig对象。若ServletConfig对象未获取，程序在运行时就有可能会出现空指针异常。为了避免这个问题的出现，在GenericServlet类中自己定义了一个没有参数的init方法，该方法就是让子类去重写的，子类重写该方法时，无需编写super.init(config)；为了保证该无参方法在初始化时执行，在init(ServletConfig config)方法中对其进行了调用。

简述web.xml的作用 属于部署描述符，在整个JAVA中只要是容器都会存在部署描述符，此部署描述符可以控制整个WEB中各个组件的运行状态，也可以配置整个窗口的状态。web.xml配置的时候要注意先后顺序。**监听器>过滤器>servlet**

设置欢迎页面

在浏览器地址栏中直接通过项目名称访问时，默认显示的页面就是欢迎页面，可以是.html，.jsp，可以通过welcome-file-list进行设置。**可以为应用设置多个欢迎页面，但只会有一个起作用，系统加载这些欢迎页面的顺序与其代码的顺序相同，即由上到下逐个查找，一旦找到，则马上显示，不会再向下查找。**

如果当前应用没有指定欢迎页面，则系统会从当前项目的根目录下依次查找 index.html、 index.htm

及 index.jsp 文件，如果这些文件不存在的话，浏览器会报出 404 错误。

web.xml*中servlet配置**

```
<!-- 创建一个servlet实例 -->
  <servlet>
      <!-- 给servlet取一个名字，该名字需与servlet-mapping中的servlet-name一致 -->
      <servlet-name>firstServlet</servlet-name>
      <!-- servlet的包名+类名 -->
      <servlet-class>com.monkey1024.servlet.FirstServlet</servlet-class>
  </servlet>

  <!-- 给servlet一个可以访问的URI地址 -->
  <servlet-mapping>
      <!-- servlet的名字，与 servlet中的servlet-name一致-->
      <servlet-name>firstServlet</servlet-name>
      <!-- URI地址:http://locahost:8080/07-01-servlet/hello -->
      <url-pattern>/hello</url-pattern>
  </servlet-mapping>
```

load-on-startup

```
<servlet>
  <servlet-name>test</servlet-name>
  <servlet-class>com.first.test</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

添加load-on-startup的作用是，**标记是否在Tomcat启动时创建并初始化这个 Servlet实例。**它的值必须是一个整数，表示servlet应该被载入的顺序。

如果该元素不存在或者这个数为负时，则容器会当该Servlet被请求时，再加载。当值大于等于 0 时，表示容器在启动时就加载并初始化这个 Servlet，数值越小，该 Servlet的优先级就越高，其被创建的也就越早；当值相同时，容器会自己选择创建顺序。

url-pattern**的设置****

url-pattern标签用于对请求进行筛选匹配，对当前注册的 Servlet 所要处理的请求类型进行筛选。对于url-pattern中路径的写法，有多种不同模式，表示不同的意义，一个Servlet可以对应多个url-pattern.

精确路径模式

请求路径必须与url-pattern的值完全相同才可被当前 Servlet 处理。

通配符路径模式

该模式中的路径由两部分组成：精确路径部分与通配符部分。请求路径中只有携带了url-pattern值中指定的精确路径部分才可被当前 Servlet 处理。

后辍名模式

请求路径最后的资源名称必须携带中指定的后辍名，其请求才可被当前Servlet 处理

全路径模式

提交的所有请求全部可被当前的 Servlet 处理。其值可以指定为/*，也可指定为/。

```
<url-pattern>/test/first</url-pattern><!-- 精确路径模式 -->
<url-pattern>/test/*</url-pattern><!-- 通配符路径模式 -->
<url-pattern>*.do</url-pattern><!-- 后缀名模式 -->
<url-pattern>/*</url-pattern><!-- 全路径模式 -->
<url-pattern>/</url-pattern><!-- 全路径模式 -->
```

/**与/的区别**

/与/表示所有请求均会被当前 Servlet 所处理。如果一个servlet的url-pattern是/或/，则该servlet表示默认映射，**当一个请求找不到相应的url的servlet时，系统会调用这个默认映射的servlet。**

这两个路径的不同之处是

使用/*

表示当前的 Servlet 会拦截用户对于静态资源（.css、.js、.html、.jpg、.png.....）与动态资源（.jsp）的请求。即用户不会直接获取到这些资源文件，而是将请求交给当前 Servlet来处理了。

使用/

表示当前的 Servlet 会拦截用户对于静态资源（.css、.js、.html、.jpg、.png.....），但**对于动态资源的请求，是不进行拦截的。**即用户请求的静态资源文件是不能直接获取到的。

对于 Servlet 的url-pattern的设置，我们一般是不会将其指定为/*或/的。一旦有一个 Servlet 的url-patter被设置为了/*或/，则整个应用的静态资源将可能无法正常显示。

url-pattern**路径优先级**

路径优先后辍匹配原则

精确路径优先匹配原则

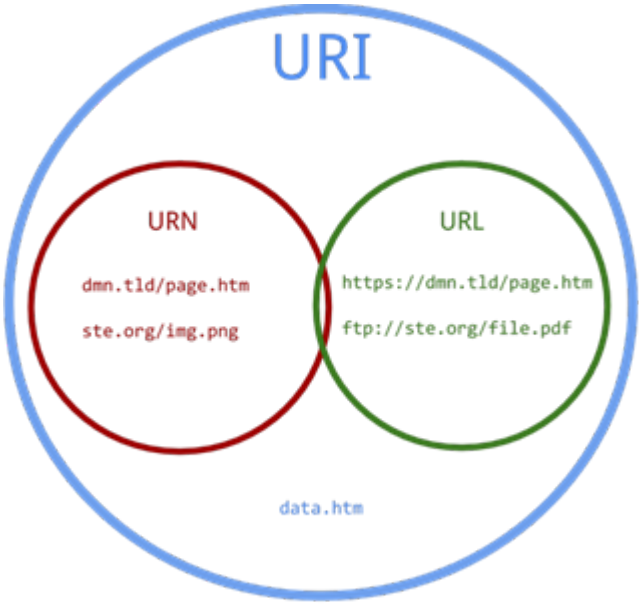
最长路径优先匹配原则

```
<url-pattern>/test/*.do</url-pattern><!-- 路径优先后辍匹配原则 -->
<url-pattern>/test/*</url-pattern>

<url-pattern>/test/first</url-pattern><!-- 精确路径优先匹配原则 -->
<url-pattern>/test/*</url-pattern>

<url-pattern>/test/first/*</url-pattern><!-- 最长路径优先匹配原则 -->
<url-pattern>/test/*</url-pattern>
```

URL**和URI**



URI（Uniform Resource Identifier）**是一个紧凑的字符串用来标示抽象或物理资源。”

URL (Uniform Resource Locator) 是URI的子集, 除了确定一个资源,还提供一种定位该资源的主要访问机制(如其网络“位置”)。

URN（Uniform Resource Name）**是唯一标识的一部分，就是一个特殊的名字。

URI可以分为URL,URN或同时具备locators 和names特性的一个东西。URN作用就好像一个人的名字，URL就像一个人的地址。换句话说：**URN确定了东西的身份，URL提供了找到它的方式。**

让URI能成为URL的当然就是那个“访问机制”，“网络位置”。例如：http:// or ftp://。

URI强调的是给资源标记命名，URL强调的是给资源定位，URL显然比URI包含信息更多，**大多数情况下大家觉得给一个网络资源分别命名和给出地址太麻烦，干脆就用地址既当地址用，又当标记名用，所以，**URL也充当了WWW万维网里面URI的角色**

ServletConfig接口简介**

在 Servlet 接口的 init()方法中有一个参数 ServletConfig，这个参数类型是个接口，里面是一些 在 web.xml 中对当前 Servlet类的配置信息。Servlet 规范将Servlet 的配置信息全部封装到了 ServletConfig 接口对象中。在tomcat调用 init()方法时，首先会将 web.xml 中当前 Servlet 类的配置信息封装为一个对象。这个对象的类型实现了 ServletConfig 接口， Web 容器会将这个对象传递给init()方法中的 ServletConfig 参数。

ServletConfig的特点**

每一个servlet都对应一个ServletConfig用于封装各自的配置信息，即有几个servlet就会产生几个ServletConfig对象。

设置初始化参数

```xml
<servlet>
    <servlet-name>configServlet01</servlet-name>
    <servlet-class>com.monkey1024.servlet.ConfigTest01</servlet-class>
    <init-param>
        <param-name>userName</param-name>
        <param-value>monkey1024</param-value>
    </init-param>
    <init-param>
        <param-name>password</param-name>
        <param-value>123456</param-value>
    </init-param>
</servlet>
```

ServletConfig中的方法****

String getInitParameter(String name)：获取指定名称的初始化参数值。

Enumeration getInitParameterNames()：获取当前 Servlet 所有的初始化参数名称。其返回值为枚举类型 Enumeration。

String getServletName()：获取当前 Servlet 的中指定的 Servlet名称。

ServletContext getServletContext()：获取到当前Servlet的上下文对象ServletContext。

遍历Enumeration

```java
Enumeration<String> param = config.getInitParameterNames();


while(param.hasMoreElements()){
    String name = param.nextElement();
    String value = config.getInitParameter(name);
    System.out.println(name + "=" +  value);
}
```

ServletContext接口简介****

WEB容器在启动时，它会为每个WEB应用程序都创建一个对应的ServletContext对象，ServletContext对象包含Web应用中所有 Servlet 在 Web 容器中的一些数据信息。**ServletContext随着Web应用的启动而创建，随着Web应用的关闭而销毁。一个Web应用只有一个ServletContext 对象。**

ServletContext中不仅包含了 web.xml 文件中的配置信息，还包含了当前应用中所有Servlet可以共享的数据。**可以这么说， ServeltContext 可以代表整个应用，所以ServletContext有另外一个名称：application。

ServletConfig对象中维护了ServletContext对象的引用，开发人员在编写servlet时，可以通过ServletConfig.getServletContext()方法获得ServletContext对象。****

设置初始化参数

```
<!-- 初始化参数 -->
<context-param>
    <param-name>MySQLDriver</param-name>
    <param-value>com.mysql.jdbc.Driver</param-value>
</context-param>
<context-param>
    <param-name>dbURL</param-name>
    <param-value>jdbc:mysql:</param-value>
</context-param>
```

ServletContext中常用方法****

String getInitParameter ()：获取 web.xml文件的中指定名称的上下文参数值 。

Enumeration getInitParameterNames()：获取 web.xml文件的中的所有的上下文参数名称。其返回值为枚举类型 Enumeration。

void setAttribute(String name, Object object)：在 ServletContext 的公共数据空间中，也称为域属性空间，放入数据。这些数据对于 Web应用来说，是全局性的，与整个应用的生命周期相同。当然，放入其中的数据是有名称的，通过名称来访问该数据。

Object getAttribute(String name)：从 ServletContext 的域属性空间中获取指定名称的数据。

void removeAttribute(String name)：从 ServletContext 的域属性空间中删除指定名称的数据。

String getRealPath(String path)：获取当前 Web 应用中指定文件或目录在本地文件系统中的路径。

String getContextPath()：获取当前应用在 Web 容器中的名称。

HttpServletRequest简介****

Web服务器收到客户端的http请求，会针对每一次请求，创建一个用于代表请求的HttpServletRequest类型的request对象，并将"HTTP请求协议"的完整内容封装到该对象中。开发者获拿到request对象后就可以获取客户端发送给服务器的请求数据了。

HttpServletRequest的生命周期****

当客户端浏览器向服务器发送请求后，服务器会根据HTTP请求协议的格式对请求进行解析。同时，服务器会创建HttpServletRequest类型的对象，即请求对象，然后将解析出的数据封装到该请求对象中。此时HttpServletRequest实例就创建并初始化完毕了，也就是说，请求对象是由服务器创建。**当服务器向客户端发送响应结束后，HttpServletRequest 实例对象被服务器销毁**，HttpServletRequest对象的生命周期很短暂。

一次请求对应一个请求对象， 另外一次请求对应另外一个请求对象，即每次请求都会创建一个HttpServletRequest类型的对象，这些对象之间没有关系。

HttpServletRequest中常用的方法****

获得客户机请求头

getHeader(string name)：返回String

getHeaders(String name)：返回Enumeration

getHeaderNames()：返回Enumeration

获得客户机请求参数(客户端提交的数据)

Map getParameterMap()：编写框架时常用。获取包含所有请求参数及值的Map对象。需要注意，该Map的value为String[]，即一个参数所对应的值为一个数组。说明一个参数可以对应多个值。

Enumeration getParameterNames()：不常用，获取请求参数Map的所有key,即获取所有请求参数名。

String[] getParameterValues(String name)：常用，根据指定的请求参数名称，获取其对应的所有值。这个方法一般用于获取复选框(checkbox)数据。

String getParameter(String name)：常用，根据指定的请求参数名称，获取其对应的值。若该参数名称对应的是多个值，则该方法获取到的是第一个值。这个方法是最常用的方法。

获取客户端信息的方法（全部返回string）

getRequestURL方法返回客户端发出请求时的完整URL。

getRequestURI方法返回请求行中的资源名部分。

getQueryString 方法返回请求行中的参数部分。

getRemoteAddr方法返回发出请求的客户机的IP地址

getRemoteHost方法返回发出请求的客户机的完整主机名

getRemotePort方法返回客户机所使用的网络端口号

getLocalAddr方法返回WEB服务器的IP地址。

getLocalName方法返回WEB服务器的主机名

getMethod得到客户机请求方式

数据空间范围对比

在 JavaWeb 编程的 API 中，存在三个可以存放数据的空间范围对象，这三个对象中所

存储的数据作用范围，由大到小分别为：

ServletContext—>HttpSession—>HttpServletRequest

ServletContext，即application，置入其中的数据是整个web应用范围的，可以完成跨会话

共享数据。

HttpSession，置入其中的数据是会话范围的，可以完成跨请求共享数据。

HttpServletRequest，置入其中的数据是请求范围的，可以完成跨 Servlet 共享数据。

但这些 Servlet 必须在同一请求中。

对于这三个域属性空间对象的使用原则是，在可以保证功能需求的前提下，优先使用小

范围的。这样不仅可以节省服务器内存，还可以保证数据的安全性。

乱码的产生原因

当用户通过浏览器提交一个包含UTF-8编码格式的两个中文请求时，浏览器会将这两个中文字符变为六个字节（一般一个UTF-8汉字占用三个字节），即形成六个类似%8E的字节表示形式，并将这六个字节上传至 Tomcat 服务器。

Tomcat 服务器在接收到这六个字节后，并不知道它们原始采用的是什么字符编码。**而Tomcat默认的编码格式为ISO-8859-1。**所以会将这六个字节按照ISO-8859-1的格式进行解码，解码后在控制台显示，所以在控制台会显示乱码。

乱码的解决

针对 POST 提交乱码的解决方式

在接收请求参数之前先通过request的**setCharacterEncoding()**方法，指定请求体的字符编码格式。这样的话，在接收到请求中的参数后，就可按照指定的字符编码进行解码。

注意，request的 setCharacterEncoding()方法只能解决POST提交方式中的乱码问题，对于GET提交方式的不起作用。因为该方法设置的是请求体中的字符编码，**GET提交中的参数不出现在请求体中，而出现在请求行。**

针对get提交乱码的解决方式

可以通过修改Tomcat默认字符编码的方式来解决GET提交方式中携带中文的乱码问题。在 Tomcat 安装目录的conf/server.xml中，找到端口号为8080的标签，在其中添加URIEncoding=”UTF-8"的设置，即可将Tomcat默认字符编码修改为UTF-8。

万能解决方案

```
//根据html中的name的名字获取用户在input中填写的值
String username = request.getParameter("username");
//将数据按照ISO8859-1编码后放到字节数组中
byte[] bytes = username.getBytes("ISO8859-1");
//将字节数组按照UTF-8解码为字符串
username = new String(bytes,"UTF-8");
```

代码量较大，开发中使用较少。

HttpServletResponse**简介****

Web服务器收到客户端的http请求，会针对每一次请求，分别创建一个用于代表响应的HttpServletResponse类型的response对象，开发者可以将要向客户端返回的数据封装到response对象中。这个对象中封装了向客户端发送数据、发送响应头，发送响应状态码的方法。

HttpServletResponse**向客户端发送数据****

```
// 不能同时使用
OutputStream op = response.getOutputStream();
byte[] b = str1.getBytes("utf-8");
op.write(b);

PrintWriter out = response.getWriter();
out.write(str2);
```

使用PrintWriter流处理字节数据，会导致数据丢失，因此在编写下载文件功能时，要使用OutputStream流，避免使用PrintWriter流，因为OutputStream流是字节流，可以处理任意类型的数据，而PrintWriter流是字符流，只能处理字符数据，如果用字符流处理字节数据，会导致数据丢失。

Servlet程序向ServletOutputStream或PrintWriter对象中写入的数据将被Servlet引擎从response里面获取，Servlet引擎将这些数据当作响应消息的正文，然后再与响应状态行和各响应头组合后输出到客户端。

Serlvet的service方法结束后，Servlet引擎将检查getWriter或getOutputStream方法返回的输出流对象是否已经调用过close方法，如果没有，Servlet引擎将调用close方法关闭该输出流对象。**

文件下载

```
// 1.获取要下载的文件的绝对路径
String realPath = this.getServletContext().getRealPath("/download/佳哥.JPG");
// 2.获取要下载的文件名
String fileName = realPath.substring(realPath.lastIndexOf("\\") + 1);
// 3.设置content-disposition响应头控制浏览器以下载的形式打开文件
// response.setHeader("content-disposition", "attachment;filename="+fileName);
response.setHeader("content-disposition", "attachment;filename=" + URLEncoder.encode(fileName, "UTF-8"));
// 4.获取要下载的文件输入流
InputStream in = new FileInputStream(realPath);
int len = 0;
// 5.创建数据缓冲区
byte[] buffer = new byte[1024];
// 6.通过response对象获取OutputStream流
OutputStream out = response.getOutputStream();
// 7.将FileInputStream流写入到buffer缓冲区
while ((len = in.read(buffer)) > 0) {
    // 8.使用OutputStream将缓冲区的数据输出到客户端浏览器
    out.write(buffer, 0, len);
}
in.close();
```

中文文件名要使用URLEncoder.encode方法进行编码(URLEncoder.encode(fileName, "字符编码"))，否则会出现文件名乱码。

HttpServletResponse响应乱码的解决方案**

响应时会产生乱码的原因是在 HTTP 协议中规定，默认响应体的字符编码为ISO-8859-1。所以，若要解决乱码问题，就需要修改响应体的默认编码。一般情况下，有两种方式可以修改：

方法一：HttpServletResponse的**setCharacterEncoding(“utf-8”)**方法，将编码修改为utf-8，然后再通过**setHead(“Content-type”,”text/html;charset=UTF-8");**方法告诉客户端浏览器的编码方式。

方法二：为了简便操作，开发者可以直接使用HttpServletResponse 的**setContentType(“text/html;charset=utf-8”)**方法，告诉浏览器的编码方式，该方法相当于方法一种的两条代码。

注意：设置响应编码时必须在 PrintWriter 对象产生之前先设置，否则将不起作用。

request.getAttribute()和 request.getParameter()有何区别**

getParameter 返回的是String,用于读取提交的表单中的值;（获取之后会根据实际需要转换为自己需要的相应类型，比如整型，日期类型啊等等）

getAttribute 返回的是Object，需进行转换,可用setAttribute 设置成任意对象，使用很灵活，可随时用

get*和post请求的区别**

	GET	POST
浏览器后退/刷新	无害	数据会被重新提交
书签	可收藏为书签	不可收藏为书签
缓存	能被缓存	不能缓存
编码类型	url编码	多种编码
历史	参数保留在浏览器历史中。	参数不会保存在浏览器历史中。
对数据长度的限制	当发送数据时，GET 方法向 URL 添加数据；URL 的长度是受限制的（URL 的最大长度是 2048 个字符）。	无限制，上传文件通常要使用post方式
对数据类型的限制	只允许 ASCII 字符。	没有限制。也允许二进制数据。
安全性	与 POST 相比，GET 的安全性较差，因为所发送的数据是 URL 的一部分。在发送密码或其他敏感信息时绝不要使用 GET ！	POST 比 GET 更安全，因为参数不会被保存在浏览器历史或 web 服务器日志中。
可见性	数据在 URL 中对所有人都是可见的。	数据不会显示在 URL 中。
传递数据	get将表单中数据按照name=value的形式，添加到action 所指向的URL 后面，并且两者使用"?"连接，而各个变量之间使用"&"连接	post是将表单中的数据放在HTTP协议的请求头或消息体中，传递到action所指向URL

转发（Forword）

转发是指浏览器发送请求到servlet1之后，servlet1需要访问servlet2，因此在服务器内部跳转到的servlet2，转发有时也称为服务器内跳转。整个过程浏览器只发出一次请求，服务器只发出一次响应。所以，无论是servlet1还是servlet2，整个过程中，只存在一次请求，即用户所提交的请求。因此servlet1和servlet2均可从这个请求中获取到用户提交请求时所携带的相关数据。

重定向（Redirect）

重定向是浏览器发送请求到servlet1之后，servlet1需要访问servlet2，但并未在服务器内直接访问，而是由服务器自动向浏览器发送一个响应，浏览器再自动提交一个新的请求，这个请求就是对servlet2 的请求。对于servlet2的访问，是先由服务器响应客户端浏览器，再由客户端浏览器向服务器发送对servlet2的请求，所以重定向有时又称为服务器外跳转。

整个过程中，浏览器共提交了两次请求，服务器共发送了两次响应。只不过，第一次响

应与第二次请求，对于用户来说是透明的，是感知不到的。用户认为，自己只提交了一次请

求，且只收到了一次响应。

转发和重定向的区别

转发是服务器行为，重定向是客户端行为。

转发通过RequestDispatcher对象的forward（HttpServletRequest request,HttpServletResponse response）方法实现的。RequestDispatcher可以通过HttpServletRequest 的getRequestDispatcher()方法获得。

request.getRequestDispatcher("/test.jsp").forward(request, response);

重定向是利用服务器返回的状态吗来实现的。客户端浏览器请求服务器的时候，服务器会返回一个状态码。服务器通过HttpServletRequestResponse的setStatus(int status)方法设置状态码。如果服务器返回301或者302，则浏览器会到新的网址重新请求该资源。sendRedirect内部的实现原理：使用response设置302状态码和设置location响应头实现重定向

response.sendRedirect("/test.jsp");

转发只能跳转到当前应用的资源中，重定向不仅可以跳转到当前应用的其它资源，也可以跳转到到其它应用中资源

请求响应

forword：浏览器只发出一次请求，收到一次响应

redirect：浏览器发出两次请求，接收到两次响应

从地址栏显示来说

forward是服务器请求资源,服务器直接访问目标地址的URL,把那个URL的响应内容读取过来,然后把这些内容再发给浏览器.浏览器根本不知道服务器发送的内容从哪里来的,所以它的地址栏还是原来的地址. redirect是服务端根据逻辑,发送一个状态码,告诉浏览器重新去请求那个地址.所以地址栏显示的是新的URL.

从数据共享来说

forward:转发页面和转发到的页面可以共享request里面的数据. redirect:不能共享数据.

从运用地方来说

forward:一般用于用户登陆的时候,根据角色转发到相应的模块. redirect:一般用于用户注销登陆时返回主页面和跳转到其它的网站等

从效率来说

forward:高. redirect:低.

转发不会执行转发后的代码;重定向会执行重定向之后的代码

请求转发与重定向的选择

若需要跳转到其它应用，则使用重定向。

若是处理表单数据的Servlet1要跳转到另外的Servlet2上，则需要选择重定向。为了防止表单重复提交。

若对某一请求进行处理的 Servlet 的执行需要消耗大量的服务器资源（CPU、内存），此时这个Servlet 执行完毕后，也需要重定向。

其它情况，一般使用请求转发。

自动刷新(Refresh)

自动刷新不仅可以实现一段时间之后自动跳转到另一个页面，还可以实现一段时间之后自动刷新本页面。Servlet中通过HttpServletResponse对象设置Header属性实现自动刷新例如：

Response.setHeader("Refresh","5;URL=http://localhost:8080/servlet/example.htm");

其中5为时间，单位为秒。URL指定就是要跳转的页面（如果设置自己的路径，就会实现每过一秒自动刷新本页面一次）

Cookie

Cookie是客户端技术，程序把每个用户的数据以cookie的形式写给用户各自的浏览器。当用户使用浏览器再去访问服务器中的web资源时，就会带着各自的数据去。这样，web资源处理的就是用户各自的数据了。

一个WEB站点可以给一个WEB浏览器发送多个Cookie，一个WEB浏览器也可以存储多个WEB站点提供的Cookie。不同的web服务器在客户端所生成的cookie之间是不能相互访问和共享的。

浏览器一般只允许存放300个Cookie，每个站点最多存放20个Cookie，每个Cookie的大小限制为4KB。

Cookie**设置绑定路径****

Cookie cookie = new Cookie("username","1234");

cookie.setPath(request.getContextPath() + "/test");

获取cookie

```
Cookie[] cookie = request.getCookies();
for(Cookie c : cookie){
    System.out.println("name="+c.getName());
    System.out.println("value="+c.getValue());
}
```

设置cookie的有效时长

默认情况下，Cookie是保存在浏览器的缓存中的，关闭浏览器后Cookie也就消失了。开发者可以通过设置Cookie的有效时长，将Cookie写入到客户端硬盘文件中。

可以通过下面的方法设置有效时长：public void setMaxAge(int expiry)，其中expiry的单位为秒，整型。

大于 0，则表示要将 Cookie 写入到硬盘文件中；

小于 0，则表示 Cookie 存放在浏览器缓存中，与不设置时长等效；

等于 0，则表示 Cookie产生后直接失效。

Session

Session是服务器端技术，利用这个技术，服务器在运行时可以为每一个用户的浏览器创建一个其独享的session对象，在实际应用当中，服务器程序可以把一些敏感数据写到用户浏览器独占的session中可以提高安全性，当用户使用浏览器访问其它程序时，其它程序可以从用户的session中取出该用户的数据，为用户服务。

Session**的失效****

若某个Session 在指定的时间范围内一直未被访问，那么 Session 将超时，即将失效。在 web.xml 中可以通过标签设置 Session 的超时时间，单位为分钟。默认 Session 的超时时间为 30 分钟。这个时间并不是从 Session 被创建开始计时的生命周期时长，而是从**最后一次被访问开始计时，在指定的时长内一直未被访问的时长。**

```
<!-- 设置失效时间为60分钟 -->
<session-config>
     <session-timeout>60</session-timeout>
</session-config>
```

可以在servlet中调用session中的invalidate()方法使session失效，invalidate()是指清空session对象里的东西，并不指清除这个session对象本身，setMaxInactiveInterval(int interval) 可以设置超时时间

JavaEE**中的cookie****

在javax.servlet.http包下有个名为Cookie的类，通过该类就可以向客户端设置cookie数据了。

JavaEE**中的session****

在javax.servlet.http包下有个HttpSession类，通过该类就可以操作session。

获取Session对象的方式：通过调用request对象中的getSession()方法就可以获取Session对象了，不需要手动new创建。

session**工作原理****

服务器会为每个浏览器分配一个session，每个浏览器只能访问自己的session对象，可http协议是无状态的，那服务器是如何识别这些浏览器的呢?

服务器对Session对象是以Map的形式进行管理的，每创建一个session对象，服务器都会向该Map中的key放入一个32位长度的随机串，这个随机串称为JSessionID，之后将该session对象的引用放入到map的value中。

session放入到Map之后，服务器还会自动将"JSESSIONID"作为name，32位长度的随机串作为value，放到cookie中并发送到客户端。该cookie会默认放到浏览器的缓存中，只要浏览器不关闭就一直存在。当浏览器第二次向服务器发送请求时会携带该cookie，服务器接收到之后会根据JSessionID从Map中找到与之对应的session对象。

cookie 和session 的区别

数据存放位置不同:

cookie数据存放在客户的浏览器上，session数据放在服务器上。

安全程度不同:

cookie不是很安全，别人可以分析存放在本地的COOKIE并进行COOKIE欺骗,考虑到安全应当使用session。

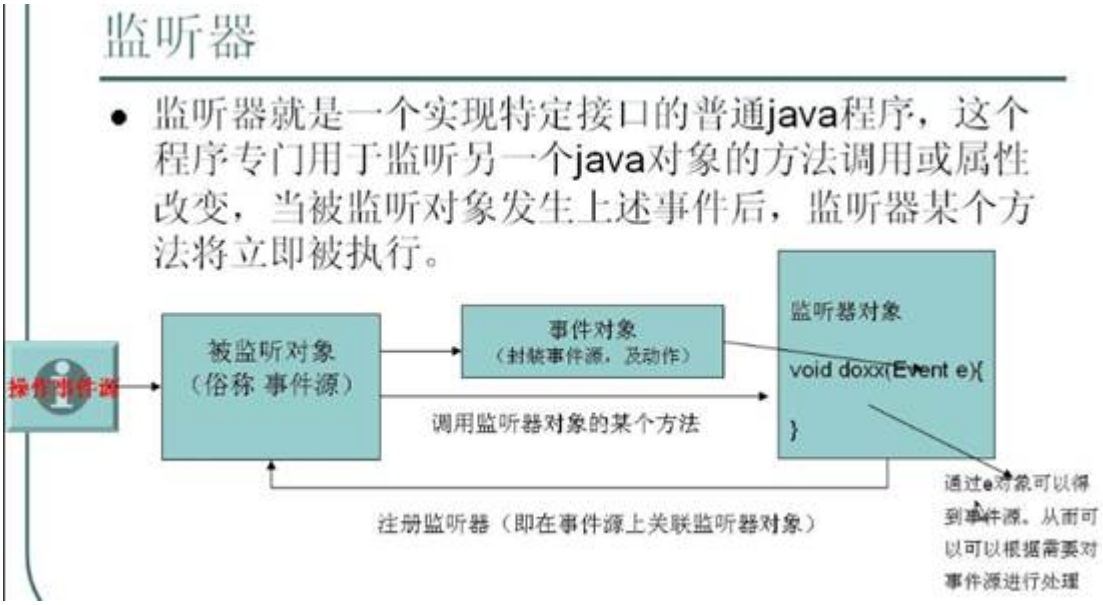
性能使用程度不同:

session会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能,考虑到减轻服务器性能方面，应当使用cookie。

数据存储大小不同:

单个cookie保存的数据不能超过4K，很多浏览器都限制一个站点最多保存20个cookie，而session则存储与服务端，浏览器对其没有限制。

监听器



java的事件监听机制**

事件监听涉及到三个组件：事件源、事件对象、事件监听器

当事件源上发生某一个动作时，它会调用事件监听器的一个方法，并在调用该方法时把事件对象传递进去，开发人员在监听器中通过事件对象，就可以拿到事件源，从而对事件源进行操作。

servlet中的监听器**

servlet中的监听器是用于监听web常见对象HttpServletRequest,HttpSession,ServletContext。主要有下面三个作用：

1.监听web对象创建与销毁。

2.监听web对象的属性变化，添加、删除、修改。

3.监听session绑定javaBean操作，活化（从硬盘读取到内存）与钝化（从内存持久化到硬盘）操作。

当监听器发现被监听的对象发生变化时，可以做一些操作。监听器其实就是一个实现特定接口的普通java程序，这个程序专门用于监听另一个java对象的方法调用或属性改变，当被监听对象发生上述事件后，监听器某个方法立即被执行。

创建监听器的步骤

创建一个类，实现指定的监听器接口

重写接口中的方法

在web.xml文件中配置监听器

监听器的配置

```
<listener>
    <listener-class>com.listener.FirstListener</listener-class>
</listener>
```

在servlet中一共有8个监听器，按照监听器的作用分类如下：

- 监听web对象创建与销毁的监听器
  - ServletContextListener
  - HttpSessionListener
  - ServletRequestListener
- 监听web对象属性变化的监听器
  - ServletContextAttributeListener
  - HttpSessionAttributeListener
  - ServletRequestAttributeListener
- 监听session绑定javaBean操作的监听器
  - HttpSessionBindingListener
  - HttpSessionActivationListener

监听web对象属性变化的监听器

三个监听器接口分别是**ServletContextAttributeListener, HttpSessionAttributeListener 和 ServletRequestAttributeListener**，这三个接口中都定义了三个方法来处理被监听对象中的属性的增加，删除和替换的事件，同一个事件在这三个接口中对应的方法名称完全相同，只是接受的参数类型不同。

attributeAdded方法****

```
1 public void attributeAdded(ServletContextAttributeEvent scae)
2 public void attributeReplaced(HttpSessionBindingEvent  hsbe)
3 public void attributeRmoved(ServletRequestAttributeEvent srae)
```

attributeRemoved方法****

```
1 public void attributeRemoved(ServletContextAttributeEvent scae)
2 public void attributeRemoved (HttpSessionBindingEvent  hsbe)
3 public void attributeRemoved (ServletRequestAttributeEvent srae)
```

attributeReplaced方法****

```
1 public void attributeReplaced(ServletContextAttributeEvent scae)
2 public void attributeReplaced (HttpSessionBindingEvent  hsbe)
3 public void attributeReplaced (ServletRequestAttributeEvent srae)
```

感知Session绑定的事件监听器

保存在Session域中的对象可以有多种状态：绑定(session.setAttribute("bean",Object))到Session中；从Session域中解除(session.removeAttribute("bean"))绑定；随Session对象持久化到一个存储设备中；随Session对象从一个存储设备中恢复

Servlet 规范中定义了两个特殊的监听器接口HttpSessionBindingListener和HttpSessionActivationListener来帮助JavaBean对象了解自己在Session域中的这些状态，**实现这两个接口的类不需要 web.xml 文件中进行注册。**

HttpSessionBindingListener接口****

实现了HttpSessionBindingListener接口的**JavaBean对象**可以感知自己被绑定到Session中和 Session中删除的事件。**当对象被绑定到HttpSession对象中时**，web服务器调用该对象的**void valueBound(HttpSessionBindingEvent event)**方法。**当对象从HttpSession对象中解除绑定时**，web服务器调用该对象的**void valueUnbound(HttpSessionBindingEvent event)**方法

```
public void valueBound(HttpSessionBindingEvent event)  {
}

public void valueUnbound(HttpSessionBindingEvent event)  {
}
```

HttpSessionActivationListener接口****

实现了HttpSessionActivationListener接口的**JavaBean对象**可以感知自己被活化(反序列化)和钝化(序列化)的事件

当绑定到HttpSession对象中的javabean对象将要随HttpSession对象被钝化(序列化)之前，web服务器调用该javabean对象的**void sessionWillPassivate(HttpSessionEvent event)**方法。这样javabean对象就可以知道自己将要和HttpSession对象一起被**序列化(钝化)到硬盘中**.

当绑定到HttpSession对象中的javabean对象将要随HttpSession对象被活化(反序列化)之后，web服务器调用该javabean对象的**void sessionDidActive(HttpSessionEvent event)**方法。这样javabean对象就可以知道自己将要和 HttpSession对象一起被**反序列化(活化)回到内存中**

```
public void sessionDidActivate(HttpSessionEvent arg0) {
}

public void sessionWillPassivate(HttpSessionEvent arg0) {
}
```

监听在Session中存放的指定类型对象的钝化与活化

HttpSessionActivationListener该监听器用于监听在Session中存放的指定类型对象的钝化与活化。**钝化是指将内存中的数据写入到硬盘中，而活化是指将硬盘中的数据恢复到内存。**当用户正在访问的应用或该应用所在的服务器由于种种原因被停掉，然后在短时间内又重启，**此时用户在访问时Session中的数据是不能丢掉的**，在应用关闭之前，需要将数据持久化到硬盘中，在重启后应可以立即重新恢复Session 中的数据。这就称为Session的钝化与活化。

那么Session中的哪些数据能够钝化呢？只有**存放在JVM堆内存中的实现了Serializable接口的对象**能够被钝化。也就是说，对于字符串常量、基本数据类型常量等存放在JVM方法区中常量池中的常量，是无法被钝化的。如果Session中的javabean对象没有实现Serializable接口，那么服务器会先把Session中没有实现Serializable接口的javabean对象移除，然后再把Session序列化(钝化)到硬盘中。对于监听Session中对象数据的钝化与活化，需要注意以下几点：

实体类除了要实现HttpSessionActivationListener接口外，还需要实现Serializable接口。

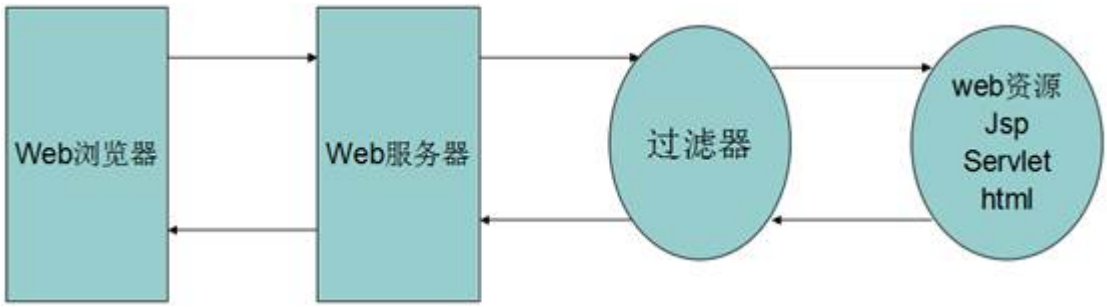
钝化指的是Session中对象数据的钝化，并非是Session的钝化。所以Session中有几个可以钝化的对象，就会发生几次钝化。

Filter

Filter 是 Servlet 规范的三大组件之一，另外两个分别是servlet和listener。Filter也称之为过滤器，可以在请求到达目标资源之前先对请求进行拦截过滤，即对请求进行一些处理；也可以在响应到达客户端之前先对响应进行拦截过滤，即对响应进行一些处理。

WEB开发人员通过Filter技术，可以对web服务器管理的所有web资源：例如Jsp, Servlet, 静态图片文件或静态 html 文件等进行拦截，从而实现一些特殊的功能。例如实现URL级别的权限访问控制、过滤敏感词汇、压缩响应信息、计算系统的响应时间等一些高级功能。

Servlet API中提供了一个Filter接口，开发web应用时，如果编写的Java类实现了这个接口，则把这个java类称之为过滤器Filter。通过Filter技术，开发人员可以实现用户在访问某个目标资源之前，对访问的请求和响应进行拦截



Filter特征**

Filter是单例多线程的。**

Filter是在**应用被加载时创建并初始化**，这是与Servlet不同的地方。Servlet是在该 Servlet被第一次访问时创建。Filter与Servlet的共同点是，其**无参构造器与init()方法只会执行一次。**

用户每提交一次该Filter可以过滤的请求，服务器就会执行一次doFilter()方法，即**doFilter()方法是可以被多次执行的。**

当应用被停止时执行destroy()方法，Filter被销毁，即destroy()方法只会执行一次。

由于Filter是单例多线程的，所以为了保证其线程安全性，一般情况下是**不为Filter类定义可修改的成员变量的。**因为每个线程均可修改这个成员变量，会出现线程安全问题。

Filter接口**

在 Servlet 规范中，有一个 javax.servlet.Filter 接口。实现了该接口的类称为过滤器，接口中有三个方法可以重写：

init()：初始化方法，即Filter被创建后，在后面用到的资源的初始化工作，可以在这里完成。

doFilter()：Filter的核心方法，对于请求与响应的过滤，就是在这个方法中完成的。

destroy()：销毁方法。Filter 被销毁前所调用执行的方法。对于资源的释放工作，可以在这里完成。

```
public void init(FilterConfig fConfig) throws ServletException {
    System.out.println("init");
}

public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
        throws IOException, ServletException {
    System.out.println("do请求");
    chain.doFilter(request, response);
    System.out.println("do响应");
}

public void destroy() {
    System.out.println("destory");
}
```

Filter是如何实现拦截的？**

Filter接口中有一个doFilter方法，当我们编写好Filter，并配置对哪个web资源进行拦截后，WEB服务器每次在调用web资源的service方法之前，都会先调用一下filter的doFilter方法，因此，在该方法内编写代码可达到如下目的：

调用目标资源之前，让一段代码执行。

是否调用目标资源（即是否让用户访问web资源）。

调用目标资源之后，让一段代码执行。

web服务器在调用doFilter方法时，会传递一个filterChain对象进来，filterChain对象是filter接口中最重要的一个对象，它也提供了一个doFilter方法，在Filter的doFilter方法内如果没有执行doFilter(request, response)方法，那么服务器中的资源是不会被访问到的。开发人员可以根据需求决定是否调用此方法，调用该方法，则web服务器就会调用web资源的service方法，即web资源就会被访问，否则web资源不会被访问。

Filter***的三种典型应用**

可以在filter中根据条件决定是否调用chain.doFilter(request, response)方法，即是否让目标资源执行

在让目标资源执行之前，可以对request\response作预处理，再让目标资源执行

在目标资源执行之后，可以捕获目标资源的执行结果，从而实现一些特殊的功能

Filter***配置**

```
<filter>
    <filter-name>my</filter-name>
    <filter-class>com.monkey1024.filter.MyFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>my</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

Filter***的全路径匹配只支持/*，不支持/****

Filter***、Listener、Servlet执行顺序**

启动的顺序为listener->Filter->servlet.

执行的顺序不会因为三个标签在配置文件中的先后顺序而改变。

Filter***的生命周期**

当服务器启动，会创建Filter对象，并调用init方法，只调用一次.

当访问资源时，路径与Filter的拦截路径匹配，会执行Filter中的doFilter方法，这个方法是真正拦截操作的方法.

当服务器关闭时，会调用Filter的destroy方法来进行销毁操作.

FilterConfig***接口**

用户在配置filter时，可以使用为filter配置一些初始化参数，当web容器实例化Filter对象，调用其init方法时，会把封装了filter初始化参数的filterConfig对象传递进来。**FilterConfig接口中的方法与ServletConfig接口中的方法，方法名与意义完全相同。**因此开发人员在编写filter时，通过filterConfig对象的方法，就可获得：

String getFilterName()：得到filter的名称。

String getInitParameter(String name)： 返回在部署描述中指定名称的初始化参数的值。如果不存在返回null.

Enumeration getInitParameterNames()：返回过滤器的所有初始化参数的名字的枚举集合。

public ServletContext getServletContext()：返回Servlet上下文对象的引用。

dispatcher 标签

在filter-mapping中还有一个子标签dispatcher，用于设置过滤器所过滤的请求类型。

其有四种取值：**REQUEST、FORWARD、INCLUDE、ERROR**，默认是REQUEST

FORWARD

若请求是由一个Servlet通过RequestDispatcher的forward()方法所转发的，那么这个请求将被值为FORWARD的Filter拦截。即当前Filter**只会拦截由RequestDispatcher的forward()方法所转发的请求。**其它请求均不拦截。

INCLUDE

当前Filter**只会拦截由RequestDispatcher的include()方法所转发的请求。**其它请求均不拦截

ERROR

在web.xml中可以配置错误页面error-page，**当发生指定状态码的错误后，会跳转到指定的页面。**而这个跳转同样是发出的请求。若的值设置为EEROR，则当前过滤器只会拦截转向错误页面的请求，其它请求不会拦截。

```
<error-page>
    <error-code>404</error-code>
    <location>/error.jsp</location>
</error-page>
```

REQUEST

默认值。即不设置dispatcher标签，也相当于指定了其值为REQUEST。**只要请求不是由 RequestDispatcher的forward()方法或include()方法转发的，那么该Filter均会被拦截，即使是向错误页面的跳转请求，同样会被拦截。**

多个Filter的执行过程

若web应用中配置了多个Filter，那么这些Filter的执行过程是**以"链"的方式执行**的。即会将这些与请求相匹配的Filter串成一个可执行的"链"，然后按照这个链中的顺序依次执行。这些Filter在链中的顺序与它们在web.xml中的注册顺序相同，即**web.xml中的注册顺序就是Filter的执行顺序。**

一个Filter的执行完毕，转而执行另一个Filter，这个转向工作是由FilterChain的doFilter()方法完成的。当然，若当前Filter是最后一个Filter，则FilterChain的doFilter()会自动转向最终的请求资源。当请求到达Filter后，Filter可以拦截到请求对象，并对请求进行修改。修改过后，再将该修改过的请求转向下一个资源。

当最终的资源执行完毕，并形成响应对象后，会按照请求访问Filter的倒序，再次访问Filter。此时 Filter可以拦截到响应对象，并对响应进行修改。最终，客户端可以收到已被修改过的响应。

JSP工作原理**

JSP是一种Servlet，但是与HttpServlet的工作方式不太一样。HttpServlet是先由源代码编译为class文件后部署到服务器下，为先编译后部署。而JSP则是先部署后编译。JSP会在客户端第一次请求JSP文件时被编译为HttpJspPage类（接口Servlet的一个子类）。该类会被服务器临时存放在服务器工作目录里面。

由于JSP只会在客户端第一次请求的时候被编译，因此第一次请求JSP时会感觉比较慢，之后就会感觉快很多。如果把服务器保存的class文件删除，服务器也会重新编译JSP。

开发Web程序时经常需要修改JSP。Tomcat能够自动检测到JSP程序的改动。如果检测到JSP源代码发生了改动。Tomcat会在下次客户端请求JSP时重新编译JSP，而不需要重启Tomcat。这种自动检测功能是默认开启的，检测改动会消耗少量的时间，在部署Web应用的时候可以在web.xml中将它关掉。

Jsp*与servlet的区别**

jsp的本质就是servlet，是servlet的一种简化，jsp经编译后就变成了Servlet

jsp更擅长表现于页面显示,servlet更擅长于逻辑控制.

Servlet中没有内置对象，Jsp中的内置对象都是必须通过HttpServletRequest对象，HttpServletResponse对象以及HttpServlet对象得到.

JSP

隐式对象

request：封装客户端的请求，其中包含来自GET或POST请求的参数；

response：封装服务器对客户端的响应；

pageContext：通过该对象可以获取其他对象；

session：封装用户会话的对象；

application：封装服务器运行环境的对象；

out：输出服务器响应的输出流对象；

config：Web应用的配置对象；

page：JSP页面本身（相当于Java程序中的this）；

exception：封装页面抛出异常的对象。

作用域

request 请求对象　作用域 Request

response 响应对象 作用域 Page

pageContext 页面上下文对象 作用域 Page

session 会话对象 作用域 Session

application 应用程序对象 作用域 Application

out 输出对象 作用域 Page

config 配置对象 作用域 Page

page 页面对象 作用域 Page

exception 例外对象 作用域 page

jsp*有四种属性范围：**

application　　在所有应用程序中有效

session　　在当前会话中有效

request　　在当前请求中有效

page　　在当前页面有效

page -> 页面级别，显然只有在一个页面内可用。

page：当前页面从打开到关闭这段时间。

request -> 请求级别 服务器跳转，一次请求之后消失。

request：HTTP请求开始到结束这段时间。

session -> 会话级别 客户端跳转（服务器跳转）

session：HTTP会话开始到结束这段时间。

application = 应用级别，当重启服务器时才会消失

application：服务器启动到停止这段时间。

页面间对象传递的方法

request，session，application，cookie

动态引入和静态引入的区别

静态引入会生成一个java文件，两个jsp文件中可以共享同一个变量，但不能定义重名的变量。

动态引入会生成两个java文件，两个jsp文件中不可以共享同一个变量，可以定义重名的变量。

EL表达式只能从 pageConext、request、session、application 四大域属性空间中获取数据。