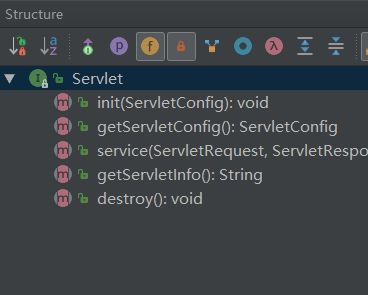
Java web知识

1. **servlet**
2. **servlet概念**

服务器端Java程序，servlet需要交给服务器来运行。与javax.servlet.Servlet接口有关的java程序。其主要功能在于交互式的浏览和修改数据，生成动态web内容。事实上，servlet就是一个Java接口。Servlet只有5个方法。



servlet接口定义的是一套处理网络请求的规范，所有实现servlet的类，都需要实现它那五个方法，其中最主要的是两个生命周期方法 init()和destroy()，还有一个处理请求的service()，也就是说，所有实现servlet接口的类，或者说，所有想要处理网络请求的类，都需要回答这三个问题：你初始化时要做什么你销毁时要做什么你接受到请求时要做什么这是Java给的一种规范！就像阿西莫夫的机器人三大定律、行尸走肉里Rick的那三个问题一样，规范！

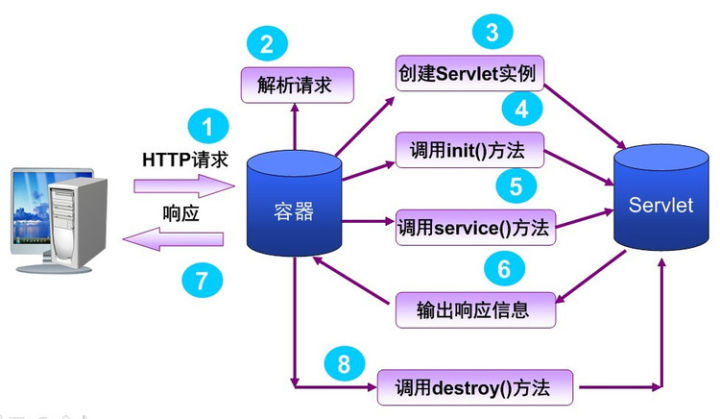
servlet是一个规范，那实现了servlet的类，就能处理请求了吗？

答案是不能

你可以随便谷歌一个servlet的hello world教程，里面都会让你写一个servlet，相信我，你从来不会在servlet中写什么监听8080端口的代码，servlet不会直接和客户端打交道！

那请求怎么来到servlet呢？

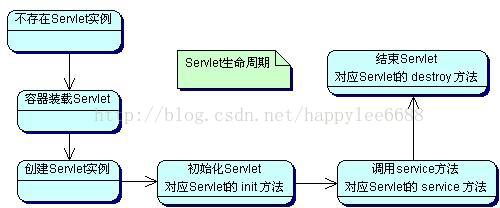
答案是servlet容器，比如我们最常用的tomcat，同样，你可以随便谷歌一个servlet的hello world教程，里面肯定会让你把servlet部署到一个容器中，不然你的servlet压根不会起作用。tomcat才是与客户端直接打交道的家伙，他监听了端口，请求过来后，根据url等信息，确定要将请求交给哪个servlet去处理，然后调用那个servlet的service方法，service方法返回一个response对象，tomcat再把这个response返回给客户端。



**注：**更常用的是GenericServlet和HttpServlet这个类,其中请求网页抛出的各种错误以及提示都是HttpServlet这个类抛出来的，servlet是单例的，由Tomcat进行维护，多个用户请求同一个servlet地址的时候，其实访问的servlet是同一个，在内存中只会有一个servlet。

我们每次访问Tomcat的时候，Tomcat都会为我们分配开启一个线程，但是Tomcat内部会有一个线程池对线程进行管理，也就是我们每次访问都会分配一个线程，但是不见得会开辟一个新线程

1. **Servlet的生命周期**



/\*\*

\* servlet的生命周期：init(ServletConfig)、service(ServletRequest,ServletResponse)、destroy()

\* \* 服务器

\* \* 第一次请求

\* \* Class clazz = Class.forName("cn.itcast.HelloWorld2");

\* \* Object obj = clazz.newInstance();

\* \* 获得init

\* \* 调用init，服务器创建ServletConfig，作为实参传递

\* \* 获得service

\* \* 调用service，服务器创建两个参与，作为实参

\* \* 第二次请求 。。。

\* \* 服务器缓存了clazz（也就不用再实例化servlet）

\* \* 获得service

\* \* 调用

\* \* 服务器关闭

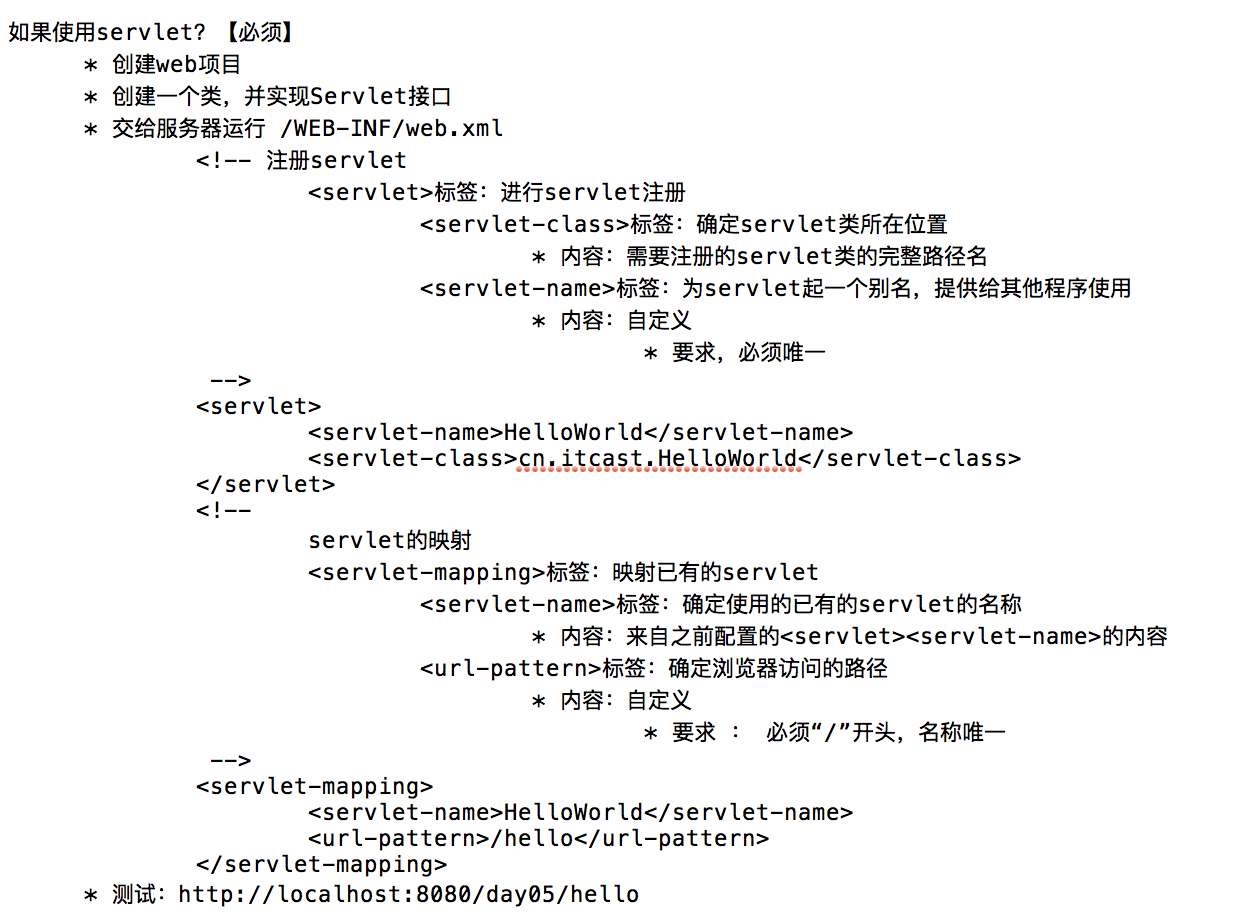
\* \* 获得clazz

\* \* 获得destroy方法

\* \* 调用

\*

\*/

****

****

1. **GenericServlet类**

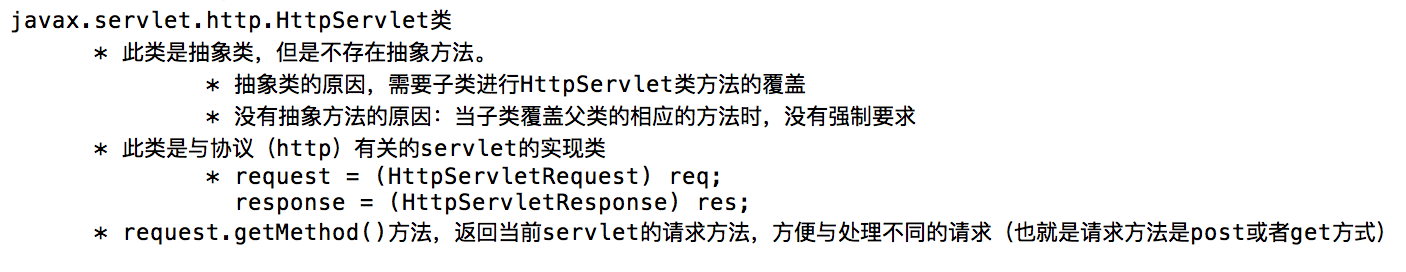
因为原生servlet接口中包含好几个方法，在使用时，子类都要继承这几个方法，有些方法子类根本用不到，因此我们使用GenericServlet类继承servlet接口，然后只暴露用户使用的最多的几个方法，这样我们在使用时，只继承GenericServlet类，可以简化代码开发。

****

1. **HttpServlet类**

HttpServlet类进一步封装了servlet接口，继承自GenericServlet抽象类，用于接收http协议请求

主要方法是doGet(),doPost()方法，用于接收get和post请求方式的请求。此处最好是看源码，继承自servlet类只有service方法。

****

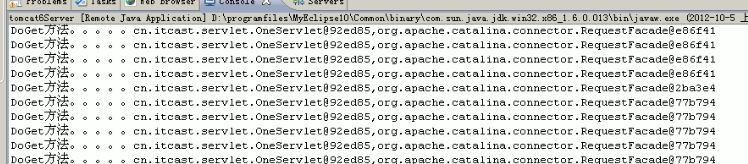
****

如果想**初始化**则直接实现无参init()方法

重点：



当我们在浏览器开启一个页面访问该servlet的时候，发现req的地址是同一个，刷新多次也是同一个，如果我们再重启开启一个浏览器，再次访问该servlet的时候，发现req的内存地址变了，如果我们多次重复开启浏览器去访问该servlet的操作，发现有些内存会被重复使用，也就是说，req的内存地址在每次请求过后，不会销毁，而是会重复利用，等到下次请求的时候再利用，如果此时没有足够的req的内存可用时，才会重新开辟新的内存地址，这是Tomcat的内存管理机制，resp的内存管理机制也是一样的。（备注：每次申请开辟新的内存地址的时候比较消耗时间和资源，因此Tomcat才会复用内存地址）



1. **Servlet的示例**

**步骤1：**新建一个类，并且让这个类继承servlet接口，实现接口中的某些方法。

**import** java.io.IOException;

**import** javax.servlet.Servlet;

**import** javax.servlet.ServletConfig;

**import** javax.servlet.ServletException;

**import** javax.servlet.ServletRequest;

**import** javax.servlet.ServletResponse;

**public** **class** HelloWorld **implements** Servlet{

@Override

**public** **void** destroy() {

//tomcat关闭的时候，输出一次

System.*out*.println("destroy");

}

@Override

**public** ServletConfig getServletConfig() {

// **TODO** Auto-generated method stub

**return** **null**;

}

@Override

**public** String getServletInfo() {

// **TODO** Auto-generated method stub

**return** **null**;

}

/\*\*

\* servlet的初始化方法

\* \* 执行次数：1次

\*/

@Override

**public** **void** init(ServletConfig config) **throws** ServletException {

//输出一次

System.*out*.println("init");

// 1、返回当前servlet的名称：<servlet><servlet-name>配置

System.*out*.println(config.getServletName());

// 2、返回指定初始化参数名称的值

String str = config.getInitParameter("gf");

System.*out*.println(str);

}

@Override

**public** **void** service(ServletRequest arg0, ServletResponse arg1)

**throws** ServletException, IOException {

//输出n次

System.*out*.println("service");

}

}

**步骤2：**在web-inf文件夹下的web.xml文件中对servlet进行配置

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

<web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

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

xmlns:web=*"http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"*

xsi:schemaLocation=*"http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd"*

metadata-complete=*"true"* version=*"3.0"*>

<servlet>

<servlet-name>hello</servlet-name>

<servlet-class>com.gh.servlet.HelloWorld</servlet-class>

<init-param>

<!-- 初始化参数的名称 -->

<param-name>gf</param-name>

<!-- 初始化参数的值 -->

<param-value>凤姐</param-value>

</init-param>

</servlet>

<!-- servlet的映射 <servlet-mapping>标签：映射已有的servlet

<servlet-name>标签：确定使用的已有的servlet的名称 \* 内容：来自之前配置的<servlet><servlet-name>的内容

<url-pattern>标签：确定浏览器访问的路径 \* 内容：自定义 \* 要求 ： 必须“/”开头，名称唯一 -->

<servlet-mapping>

<servlet-name>hello</servlet-name>

<url-pattern>/hello</url-pattern>

</servlet-mapping>

</web-app>

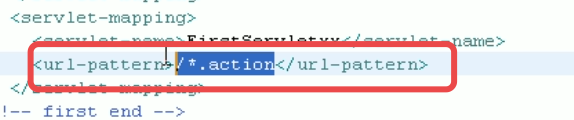
**注意**：<servlet-mapping>配置标签必须放到<servlet>标签之后。

1. **servlet的虚拟路径**

也就是在配置servlet-mapping的时候，它的配置规则。

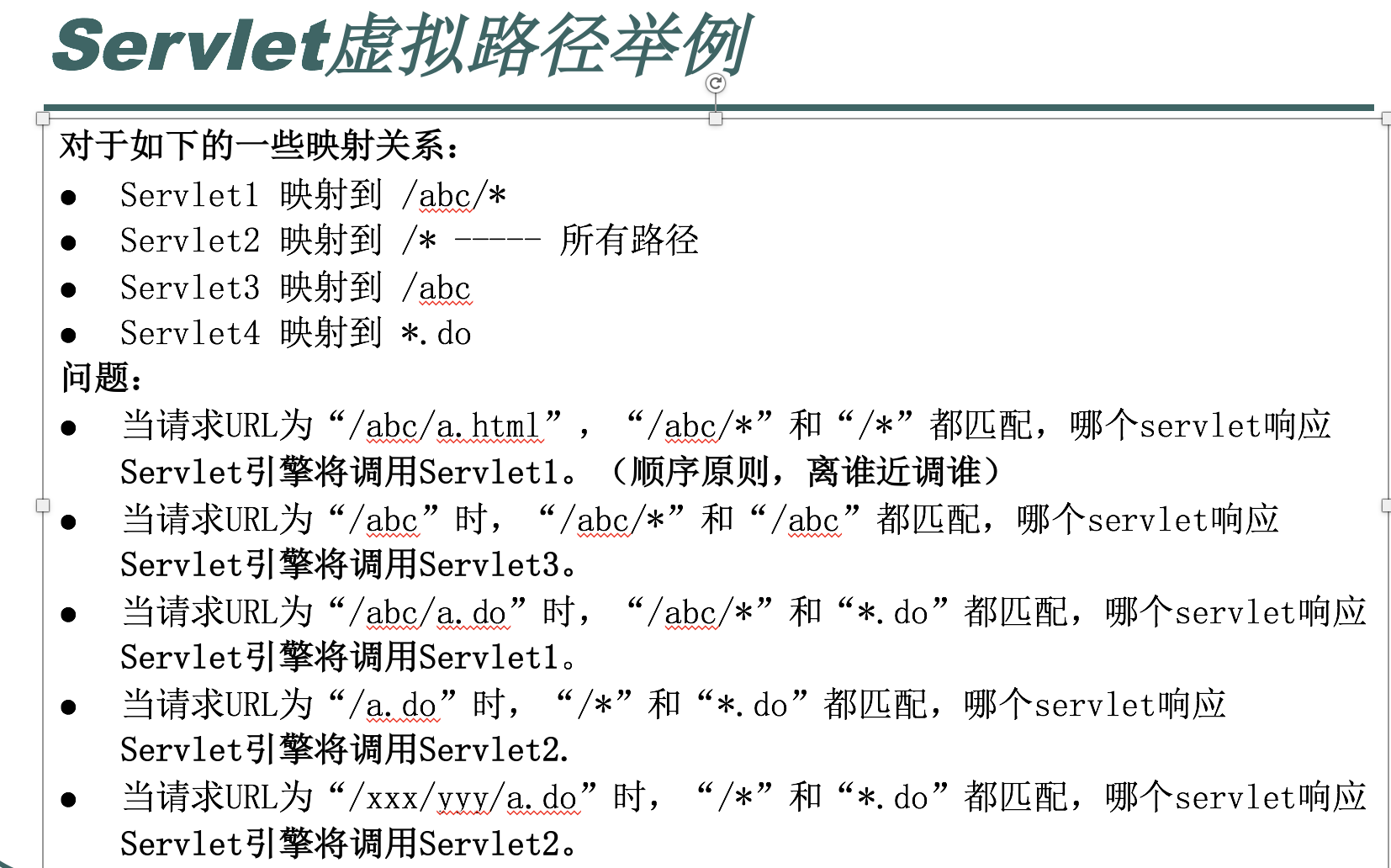
1、只能使用 / 或 \* 开头

2、/ 和 \*. 不能同时存在，如下这种情况就不能存在，启动会报错



3、/ 或 /\* 表示任意

4、只有\*.才表示通配符，比如：\*.do



1. **缺省servlet**

如果某个Servlet的映射路径仅仅为一个正斜杠（/），那么这个Servlet就成为当前Web应用程序的缺省Servlet。

凡是在web.xml文件中找不到匹配的<servlet-mapping>元素的URL，它们的访问请求都将交给缺省Servlet处理，也就是说，缺省Servlet用于处理所有其他Servlet都不处理的访问请求。

在<tomcat的安装目录>\conf\web.xml文件中，注册了一个名称为org.apache.catalina.servlets.DefaultServlet的Servlet，并将这个Servlet设置为了缺省Servlet。





如果我们将web.xml配置文件改成上面的配置，如果我们的服务器没有index.html欢迎页面，服务器将返回所有静态文件的列表。这样操作是很消耗资源的，服务器响应会变慢，因此不建议将其设置为true。

Tomcat是servlet容器，也就是只能处理servlet请求，不能处理html静态文件，如果想让Tomcat处理静态文件，就需要使用缺省servlet。

注：当访问Tomcat服务器中的某个静态HTML文件和图片时，实际上是在访问这个缺省Servlet。

1. **服务器启动时servlet初始化init**

**描述：** 我们每次发送请求的时候，服务器才去初始化servlet，也就是调用servlet的init函数进行初始化，如果初始化的步骤比较复杂，那么用户的请求响应会相当慢，因此我们最好是在服务器启动的时候就去初始化servlet，这样在用户请求过来的时候，就直接响应用户请求，不用再去初始化，这样的响应速度更快。

如下配置以后，tomcat在启动的时候就会去初始化servlet的init()函数，在web.xml配置文件里面进行配置。

<!-- 要求当前servlet，在服务器启动时进行初始化工作 -->

<servlet>

<servlet-name>FirstServletxx</servlet-name>

<servlet-class>cn.itcast.servlet.FirstServlet</servlet-class>

<!-- 配置当前servlet在服务器启动时加载顺序 取值：0-6 ,值越小，优先加载 -->

<load-on-startup>2</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>FirstServletxx</servlet-name>

<url-pattern>/firstServlet</url-pattern>

</servlet-mapping>

1. **BaseServlet**

通常我们在使用httpServlet的时候，还是有很多不便的地方，比如通过get请求获取前端参数的时候，中文通常会出现乱码，如果我们每次通过获取到参数值，然后再次转码会增加太多的重复代码。因此我们通过BaseServlet对httpServlet进行再次包装，我们使用servlet的时候就继承BaseServlet就行，这样可以避免很多代码。

|  |
| --- |
| **public** **class** BaseServlet **extends** HttpServlet {  @Override  **public** **void** service(HttpServletRequest req, HttpServletResponse resp)  **throws** ServletException, IOException {  req.setCharacterEncoding("UTF-8");  String methodName = req.getParameter("cmd");  **try**{  Method mm = **this**.getClass().getMethod(methodName,HttpServletRequest.**class**,HttpServletResponse.**class**);  //声明包装类  MyRequest mr = **new** MyRequest(req);  mm.invoke(**this**,mr,resp);  }**catch**(Exception e){  e.printStackTrace();  }  }  }  //包装request  **class** MyRequest **extends** HttpServletRequestWrapper{  **private** HttpServletRequest req;  **public** MyRequest(HttpServletRequest request) {  **super**(request);  **this**.req=request;  }  //修改getparameter方法  @Override  **public** String getParameter(String name) {  String value = req.getParameter(name);  **if**(req.getMethod().equals("GET")){  System.***err***.println("转码");  **try**{  value = **new** String(value.getBytes("ISO-8859-1"),"UTF-8");  }**catch**(Exception e){  }  }  **return** value;  }    @Override  **public** String[] getParameterValues(String name) {  String[] vals = req.getParameterValues(name);  **if**(req.getMethod().equals("GET")){  **for**(**int** i=0;i<vals.length;i++){  **try** {  vals[i] = **new** String(vals[i].getBytes("ISO-8859-1"),req.getCharacterEncoding());  } **catch** (UnsupportedEncodingException e) {  e.printStackTrace();  }  }  }  **return** vals;  }  @Override  **public** Map getParameterMap() {  Map<String,String[]> mm = req.getParameterMap();  **if**(req.getMethod().equals("GET")){  Iterator<String[]> it= mm.values().iterator();  **while**(it.hasNext()){  String[] ss = it.next();  **for**(**int** i=0;i<ss.length;i++){  **try** {  ss[i] = **new** String(ss[i].getBytes("ISO-8859-1"),req.getCharacterEncoding());  } **catch** (UnsupportedEncodingException e) {  e.printStackTrace();  }  }  }  }  **return** mm;  }  } |

1. **Jsp与servlet**

Jsp本身也是servlet，因此在web.xml配置文件中如何配置servlet，就可以在web.xml中如何配置jsp

|  |
| --- |
| <servlet>  <servlet-name>c</servlet-name>  <jsp-file>/WEB-INF/secu/c.jsp</jsp-file>  </servlet> |

1. **ServletContext**
2. **概念**

对当前web项目上下文的描述(对当前web项目所有内容的描述)，tomcat为每一个web项目单独创建一个区域，用来管理整个项目。此区域成为ServletContext。tomcat在启动时创建，tomcat关闭时销毁。

获取：servlet -- > init(ServletConfig) --> config.getServletContext();

①获得实际路径

ServletContext sc = **this**.getServletContext();

System.*out*.println(sc.getRealPath("WEB-INF/web.xml"));

结果：

C:\ProgramFiles\ApacheSoftwareFoundation\Tomcat7.0\webapps\servlet\WEB-INF\web.xml

②获得URL

ServletContext sc = **this**.getServletContext();

System.*out*.println(sc.getResource("WEB-INF/web.xml"));

结果：

/localhost/servlet/WEB-INF/web.xml

③获得流【\*\*】

ServletContext sc = **this**.getServletContext();

System.*out*.println(sc.getResourceAsStream("WEB-INF/web.xml"));

结果：

[java.io.ByteArrayInputStream@691177](mailto:java.io.ByteArrayInputStream@691177)

1. **ServletContext共享数据**

ServletContext对象对所有的servlet共享数据,也就是说使用 setAttribute设置的内容，对于所有该web项目的访问者都可以获取到该内容，他是共享的。

这个内容就好比我们对web项目进行设置的配置内容一样，都是共享的，因此setAttribute能设置的内容也可以在web.xml文件中进行配置。

得到是getAttribute、删除是removeAttribute

配置内容：

<context-param>

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

<param-value>root</param-value>

</context-param>

将其直接放进web.xml配置文件父标签就是web-app

获取值的方式：

String name =getServletContext(). getInitParameter(“username”);

1. **多个servlet间共享数据**

**方法1：**用servletContext

**方法2：**用session

1. **Web的相对路径**

前提：相对当前的页面，比如当前页面是1.html

分类：

1）abc:与当前页面同级的目录或servlet的名称

当前页面：http://localhost:8080/day06\_web/b/b.html

目标页面：http://localhost:8080/day06\_web/b/c/c.html

\* <a href="c/c.html">c.html</a><br>

2）/abc:相对于web站点，%tomcat%/webapps/

\* <a href="/day06\_web/b/c/c.html">c.html</a><br>

3）./abc:当前目录，与第一种情况相同

\* <a href="./c/c.html">c.html</a><br>

4）../abc:上一次目录

\* <a href="../b/c/c.html">c.html</a><br>

1. **HttpServletResponse响应**

服务器对浏览器做出的响应，将需要发送给浏览器的所有数据全部存放在此对象上。发送数据，使用流操作，将所需要的数据，存放在指定的流中，数据将显示到浏览器中。

流有两种，字节流和字符流，两者不能同时使用，只能二选一。

**发送数据**

字节流：一般在程序中使用具有拷贝功能等，比如图片，视频等

**package** com.gh.servlet;

**import** java.io.IOException;

**import** java.util.Date;

**import** javax.servlet.ServletException;

**import** javax.servlet.ServletOutputStream;

**import** javax.servlet.http.HttpServlet;

**import** javax.servlet.http.HttpServletRequest;

**import** javax.servlet.http.HttpServletResponse;

**public** **class** Hello **extends** HttpServlet {

**public** **void** doGet(HttpServletRequest request, HttpServletResponse response)

**throws** ServletException, IOException {

//使用字符流，将数据发送到浏览器

// \* 获得字节流 OutputStream InputStream

ServletOutputStream out = response.getOutputStream();

// \* 准备数据

**long** data = **new** Date().getTime() ; //"Hello World !!!";

// \* 将数据写入到获得的字符流中

out.print(data);

}

**public** **void** doPost(HttpServletRequest request, HttpServletResponse response)

**throws** ServletException, IOException {

}

}

字符流：一般在程序中发送数据内容

//使用 字符流 发送数据 Writer Reader

// \* 获得

PrintWriter out = response.getWriter();

// \* 数据

String date = **new** Date().toString();

// \* 发送数据到浏览器

out.println(date);

**处理中文**

**字符流：**与字节流相比，字符流不需要设置数据的编码，只需要通知浏览器用什么编码解析数据就行

//设置响应浏览器的字符编码 : 通知tomcat和浏览器，当前servlet发送的数据的编码

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

//发送中文数据 --“中文”

String data = "中文";

//字节流

PrintWriter out = response.getWriter();

//out.print(data); 不能发送中文数据 java.io.CharConversionException Not an ISO 8859-1 character: 中

out.print(data);

//字符流不需要设置数据的编码

// out.write(data.getBytes("UTF-8"));

**字节流：**

//设置响应浏览器的字符编码 : 通知tomcat和浏览器，当前servlet发送的数据的编码

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

//发送中文数据 --“中文”

String data = "中文";

//字节流

ServletOutputStream out = response.getOutputStream();

//out.print(data); 不能发送中文数据 java.io.CharConversionException Not an ISO 8859-1 character: 中

out.write(data.getBytes("UTF-8"));

out.write(data.getBytes());

**注：**response.setContentType("text/html;charset=UTF-8")函数是告诉浏览器对网页数据用utf-8编码去解析，而不是对网页内容进行编码；data.getBytes("UTF-8")是对服务器响应浏览器的数据内容进行utf-8编码。

编码流程：

首先服务器响应浏览器的数据，服务器不对数据进行编码的话，那么就是服务器系统默认的编码,然后数据通过http协议进行传输，http协议会将数据进行iso-8859-1编码，然后在网络上进行传输，浏览器接收到数据，浏览器首先用iso-8859-1进行解析数据，然后使用操作系统默认的编码进行解析，比如中国的window系统默认就是gbk编码，这样会用gbk编码进行解析。最后如果其中响应头里面有说用什么编码进行解析网页，那么则用服务器告诉的编码进行解析，如果没有，则不继续进行解析。

1. **HttpServletResquest请求**

浏览器向服务器的请求（浏览器将数据发送给服务器时，数据存放的地方），请求方式：GET和POST

GET：发送的数据，追加在请求的URL之上

POST：发送的数据在HTTP请求体中

**浏览器发送数据**

\* 表单form，

\* method属性：指定的请求方式

\* action属性：接收数据的程序路径

**服务器获得浏览器发送的数据**

\* 获得单个数据

\* request.getParameter("username");

\* 获得一组数据，返回的是数组

\* request.getParameterValues("love");

**处理中文**

前提是页面也使用utf-8编码

**页面使用post请求：**

//需要通知Tomcat，用户提交的数据的字符集是utf-8

request.setCharacterEncoding("utf-8");

**页面使用get请求：**

方法1：

//获取所有的请求数据字段

String name = request.getQueryString();

name = URLDecoder.decode(name,"utf-8");

方法2：

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

//或者下面这种写法,由于http协议将get传来的参数用ISO-8859-1进行编码，那么现在就用ISO-8859-1进行解码成字节数组，然后再将字节数组按照utf-8的编码进行解析

username = **new** String(username.getBytes("ISO-8859-1"),"utf-8");

**注：一般在有表单进行中文请求数据时不建议使用get方式，建议使用post方式，这样避免繁杂的中文编码问题**

1. **重定向、转发、跳转**

**①跳转** 跳转的过程跟重定向基本上是一致的，在此不多加说明

//跳转

//必须设置状态值，要不然是不会跳转的

response.setStatus(302);

//设置跳转到目标页面，当用户输入http://localhost:8080/servlet/hello时，首先浏览器会请求到hello页面然后再自动跳转到hello.jsp页面

response.setHeader("location", "http://localhost:8080/servlet/hello.jsp");

**②重定向** 整个操作都是浏览器完成，浏览器发送两次请求。http状态码302

重定向：重新确定请求方向，也就是浏览器发送请求过来，服务器告诉浏览器这个请求不对，然后服务器会告诉浏览器一个正确的请求地址，浏览器拿到该请求地址，按照该请求地址再次发送请求。

注：可以指定当前的web项目，也可以指定其他的web资源，重定向不能设置请求的参数，原因是两次请求，tomcat创建了两个request

//重定向，路径必须是绝对路径

response.sendRedirect("http://localhost:8080/day07/1.html");



我们在浏览器输入<http://localhost:8080/day03/hello>的时候，浏览器发送两次请求，重定向到后面这个页面，注意此时的浏览器地址栏的地址变了。

**③转发**  服务器从一个servlet转发到另外一个servlet，浏览器只发送一次请求，但是执行了两个servlet服务

注：只能指定当前的web项目，转发的时候可以设置请求的参数，在转发的页面可以获取到该参数，因为tomcat只创建一个request。

因此需要在servlet间传递参数的时候，需要用到转发，而不是重定向

**hello servlet:**

**public** **void** doGet(HttpServletRequest request, HttpServletResponse response)

**throws** ServletException, IOException {

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

//转发 -- 当前请求未完成

// \* 获得请求调度器

/\*

\* 参数:

\* 当前页面：http://localhost:8080/servlet/hello

\* 目标页面：http://localhost:8080/servlet/other

\* / -- context root --> WebRoot --> tomcat/webapps/servlet

\*/

RequestDispatcher dispatcher = request.getRequestDispatcher("/other");

//需要将数据发送的浏览器

PrintWriter out = response.getWriter();

System.*out*.println("转发前");

out.print("页面转发前<br/>"); //没有发送到浏览器

// \* 转发操作 -- 调用服务器端的一个方法

dispatcher.forward(request, response); //显示最后一个输出的内容

// dispatcher.include(request, response); //合并当前所有的执行页面的输出内容

out.print("页面转发后<br/>"); //没有发送到浏览器

System.*out*.println("转发后");

}

**Other servlet:**

**package** com.gh.servlet;

**import** java.io.IOException;

**import** javax.servlet.ServletException;

**import** javax.servlet.http.HttpServlet;

**import** javax.servlet.http.HttpServletRequest;

**import** javax.servlet.http.HttpServletResponse;

**public** **class** Other **extends** HttpServlet {

@Override

**protected** **void** doGet(HttpServletRequest req, HttpServletResponse resp)

**throws** ServletException, IOException {

resp.getWriter().print("转发页面");

System.*out*.println("other");

}

@Override

**protected** **void** doPost(HttpServletRequest req, HttpServletResponse resp)

**throws** ServletException, IOException {

}

}

\* hello servlet里面的request对象 --> org.apache.catalina.connector.RequestFacade -- HttpServletRequest –ServletRequest

\* other servlet里面的request对象 --> org.apache.catalina.core.ApplicationHttpRequest

\* class ApplicationHttpRequest extends HttpServletRequestWrapper {

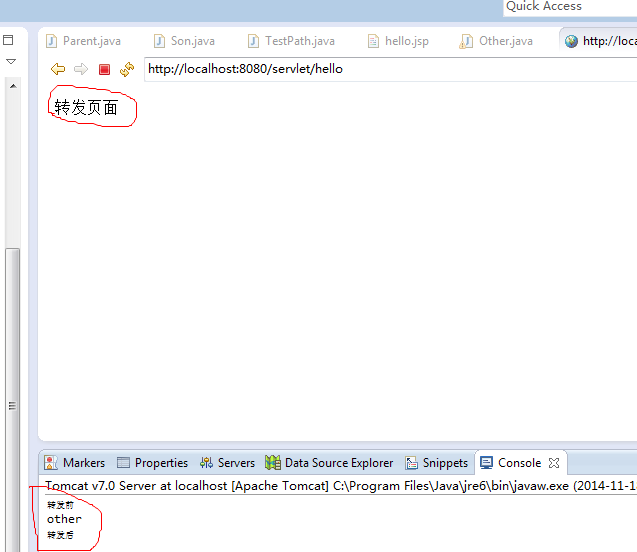
\* public class HttpServletRequestWrapper implements HttpServletRequest {

\* ApplicationHttpRequest -- HttpServletRequestWrapper -- HttpServletRequest

此处将第一个request（hello servlet）中的内容，拷贝给第二个request(other servlet)。值相同，但是对象已经不同了。

结果：此处页面只有转发后的页面的输出，没有之前servlet的输出。如果我们想要之前的servlet的输出内容展示到页面，我们可以使用include来代替forward方法。

forward:当调度器，协调多个servlet时，此方法返回最后一个servlet的页面输出内容。



1. **网页中生成验证码以及验证正确性**

源代码见百度云盘：技术资料/web源码/验证码

1. **URLConnection**

//访问 http://localhost:8080/day08\_web/loginServlet

//URL类对web资源的定位

URL url = **new** URL("http://localhost:8080/day08\_web/loginServlet");

//获得连接 ，做准备

URLConnection conn = url.openConnection();

//必须确定当前请求，是否可以读写

conn.setDoInput(**true**); //确定是否可以读，默认是true

conn.setDoOutput(**true**);//确定是否可以写，默认是false

//准备发送的数据 -- request填充

OutputStream out = conn.getOutputStream();

out.write("user=abcd1234".getBytes()); //此方法将内容写到http请求体中

//连接

//conn.connect();

//建立链接之后，获得web资源

InputStream is = conn.getInputStream(); //会自动链接,只有调用方法，整个请求才生效

// byte[] buf = new byte[1024];

// int len = -1;

// while( (len = is.read(buf)) > -1 ){

// String str = new String(buf,0,len);

// System.out.println(str);

// }

Scanner scanner = **new** Scanner(is);

**while**(scanner.hasNext()){

System.*out*.println("url -->" + scanner.nextLine());

}

is.close();

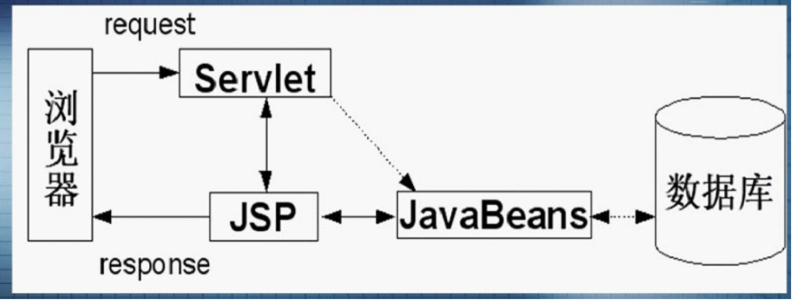
1. **中文Unicode编解码**

|  |
| --- |
| String data = "中文";  String encode = URLEncoder.encode(data,"utf-8");  System.out.println(encode);  String decode = URLDecoder.decode(encode,"utf-8");  System.out.println(decode); |

1. **MVC**
2. **MVC**

Servlet+JSP+JavaBean(MVC)模式适合开发复杂的web应用，在这种模式下，servlet负责处理用户请求，jsp负责数据显示，javabean负责封装数据。 Servlet+JSP、JavaBean模式程序各个模块之间层次清晰，web开发推荐采用此种模式。

* Servlet 控制显示哪个JSP页面给用户
* JSP 响应用户请求，把结果数据展现给用户
* JavaBean 对数据库的数据的存取，复杂的业务功能和逻辑处理



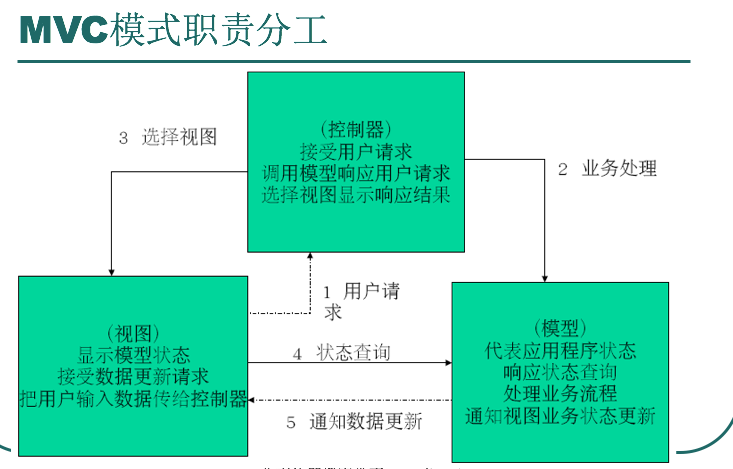
MVC是Model-View-Controller的简称

设计模式：

把应用程序分成三个核心模块，模型、视图、控制器

* 模型（Model）部分负责管理程序的业务数据
* 视图（View）部分负责显示界面
* 控制器（Controller）部分负责与用户进行交互（接受请求和选择响应视图）

MVC职责分工：



1. **JavaBean**

JavaBean是一个遵循特定写法的Java类，它通常具有如下特点：

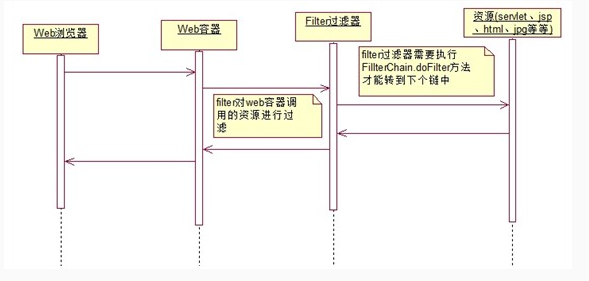
* 这个Java类必须具有一个无参的构造函数
* 字段必须私有化。
* 私有化的字段必须通过public类型的方法暴露给其它程序，并且方法的命名也必须遵守一定的命名规范。

JavaBean在J2EE开发中，通常用于封装数据，对于遵循以上写法的JavaBean组件，其它程序可以通过反射技术实例化JavaBean对象，并且通过反射那些遵守命名规范的方法，从而获知JavaBean的属性，进而调用其属性保存数据。

**Javabean的属性特点：**

* JavaBean的属性可以是任意类型，并且一个JavaBean可以有多个属性。每个属性通常都需要具有相应的setter、 getter方法，setter方法称为属性修改器，getter方法称为属性访问器。
* 属性修改器必须以小写的set前缀开始，后跟属性名，且属性名的第一个字母要改为大写，例如，name属性的修改器名称为setName，password属性的修改器名称为setPassword。
* 属性访问器通常以小写的get前缀开始，后跟属性名，且属性名的第一个字母也要改为大写，例如，name属性的访问器名称为getName，password属性的访问器名称为getPassword。
* 一个JavaBean的某个属性也可以只有set方法或get方法，这样的属性通常也称之为只写、只读属性。

1. **过滤器Filter**
2. **概念**



在web中过滤器就是一个类javax.servlet.Filter

**过滤器特点**：

1）、过滤器只对URL（路径）进行过滤

2）、过滤器开发人员来实现，开发人员必须实现filter，它的定义只是一个接口，web服务器不对他进行任何的实质性的操作和功能

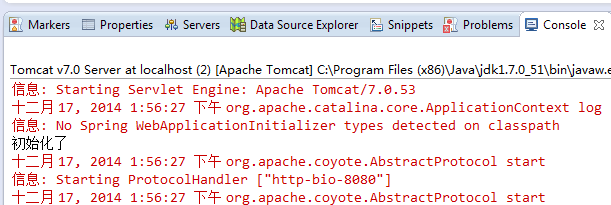
3）、过滤器由于是web的核心组，所以这个filter的实现者也必须要配置到web.xml中

4）、三个生命周期方法init，destroy，doFilter（执行过滤任务）。用户的每次请求，都会执行doFilter方法，而init，destroy只会执行一次。

Init方法执行的时间：在项目启动时，直接初始化Filter的对象，所以会在tomcat启动时执行init方法。

**作用：**通过Filter技术，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、统一WEB编码等一些高级功能。

**注：**在tomcat或者其他服务器启动的时候，就会启动过滤器，过滤器就会执行初始化函数init()。



**应用场景：**

①通过控制对chain.doFilter的方法的调用，来决定是否需要访问目标资源。

比如，可以在用户权限验证等等。判断用户是否有访问某些资源的权限，有权限放行，没权限不执行chain.doFilter方法。

②通过在调用chain.doFilter方法之前，做些处理来达到某些目的。

比如，解决中文乱码的问题等等。可以在doFilter方法前，执行设置请求编码与响应的编码。甚至可以对request接口进行封装装饰来处理get请求方式的中文乱码问题(重写相应的request.getParameter方法)。

③通过在调用chain.doFilter方法之后，做些处理来达到某些目的。

比如对整个web网站进行压缩。在调用chain.doFilter方法之前用类A对response对象进行封装装饰，重写getOutputStream和重写getWriter方法。在类A内部中，将输出内容缓存进ByteArrayOutputStream流中，然后在chain.doFilter方法执行后，获取类A中ByteArrayOutputStream流缓存数据，用GZIPOutputStream流进行压缩下。

1. **过滤器能做什么？**

1）、过滤或是设置字符集。最最基本的。对post有效。

2）、过滤关键字。

3）、在拦截的过程中修改request(包装、代理)以实现程序要求。如通过get方式可以获取中文字符。

4）、设置对某些页面缓存或是对某些页面不缓存。

5）、验证用户是否是有效的用户。即已经登录了的用户。

6）、自动登录。

7）、对输出的数据进行压缩。或是对全站进行压缩。

8）、用过虑器来管理事务,OSIV模式,与使用代理管理事务对比与分析。

9）、实现权限过虑器。－综合示例。相对比较复杂。

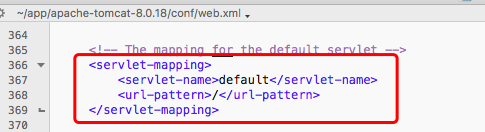
1. **filter的url－pattern设置**

url-pattern的设置：

/\*：对所有url都请求这个servlet。| 都被这个filter过虑.

/：只可以给servlet。 － 所有其他的servlet不处理的url都由这个url所指定的servlet处理。默认servlet。

Tomcat安装路径下web.xml配置文件内容：



\*.jspx：所有以jspx结束都请求到这个Servlet或是被某个Filter拦截。如<http://local:8080/proj/abc.jspx>

/jsps/\*：所有路径中以/jsps/开头的都请求到这个servlet或是被某个filter拦截到。

/jsps/\*.jspx 错误的：SUN规定\*两边不可以同时出现字符。

1. **对转发进行拦截过滤**

如果有jsps/abc.jsp这个页面，则显示这个页面有两种方式：

1：在地址栏请求：http://local:8080/project/jsps/abc.jsp -- 可以被拦截到的，因为：路径符合/jsp/abc.jsp

2：req.getRequestDispathcer(“/jsps/abc.jsp”).forward(req,resp); --这是转发，默认情况下，过滤器不会对转发的内容进行拦截。

可以通过设置如下属性：

Dipatcher属性:

Request – 默认值，只拦截用户新的请求。

Forward – 对转发进行拦截。

Include – 对include拦截。

Error - 对错误进行拦截。

具体如下：



1. **配置过虑器的初始化参数**

在filter类中有一个方法：init(FilterConfig config); 而filtetrConfig中包含了从web.xml中读取初始化参数值。

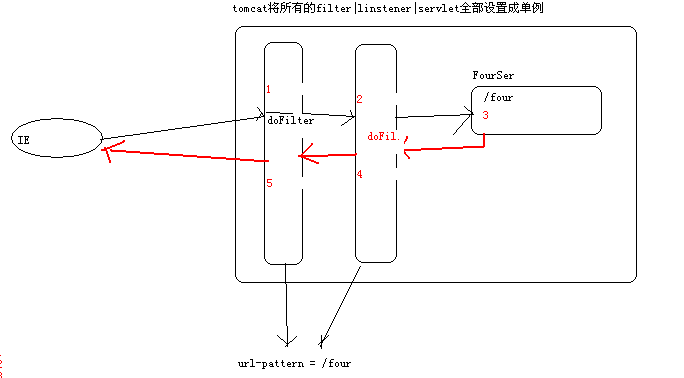
|  |
| --- |
| <filter>  <filter-name>one</filter-name>  <filter-class>cn.itcast.filter.OneFilter</filter-class>  <init-param>  <param-name>name</param-name>  <param-value>张三Jack</param-value>  </init-param>  <init-param>  <param-name>age</param-name>  <param-value>88</param-value>  </init-param>  </filter> |

在filter类中获取值：

|  |
| --- |
| **private** FilterConfig conf;  //此方法只会在tomcat在启动时执行一次  **public** **void** init(FilterConfig conf) **throws** ServletException {  **this**.conf=conf;  System.*err*.println("过虑器初始化了:"+**this**);  //获取name的值：  String name = conf.getInitParameter("name");  System.*err*.println("name is:"+name);  System.*err*.println("----------------------------");  Enumeration<String> en= conf.getInitParameterNames();  **while**(en.hasMoreElements()){  String paramName = en.nextElement();  String val = conf.getInitParameter(paramName);  System.*err*.println(paramName+"="+val);  }  } |

1. **过虑器链FilterChain**

当一个web项目中有多个连接器，且这些过虑器对同个url进行过虑。这时就形成了过虑器链。



在filter的配置当中，有两个配置:filter,filter-mapping。Filter-mapping在前，则先执行，在后则后执行。

配置如下：

|  |
| --- |
| <filter>  <filter-name>first</filter-name>  <filter-class>cn.itcast.filter.FirstFilter</filter-class>  </filter>  <filter>  <filter-name>second</filter-name>  <filter-class>cn.itcast.filter.SecondFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>second</filter-name>  <url-pattern>/four</url-pattern>  </filter-mapping>  <filter-mapping>  <filter-name>first</filter-name>  <url-pattern>/four</url-pattern>  </filter-mapping> |

此处省略了两个servlet的具体实现。

1. **示例**

步骤1：新建一个filter过滤器，并且必须实现filter接口

|  |
| --- |
| **public** **class** FilterTest **implements** Filter{  **private** FilterConfig filterConfig;  /\*\*  \* 销毁函数，在tomcat正常关闭的时候执行，在tomcat异常关闭的时候可能不会执行  \*/  @Override  **public** **void** destroy() {  System.*out*.println("销毁了");  }  /\*\*  \* 当有用户请求过来的时候执行，该函数主要进行过滤功能  \*/  @Override  **public** **void** doFilter(ServletRequest req, ServletResponse resp,  FilterChain chain) **throws** IOException, ServletException {  System.*out*.println("Demo1过滤前");  //获取请求传来的参数  System.*out*.println(filterConfig.getInitParameter("param1"));  //执行下一个过滤器器，如果没有下一个过滤器则执行目标的servlet  chain.doFilter(req, resp);//放行。让其走到下个链或目标资源中  System.*out*.println("Demo1过滤后");  }  /\*\*  \* 在tomcat启动的时候执行  \*/  @Override  **public** **void** init(FilterConfig config) **throws** ServletException {  System.*out*.println("初始化了");  **this**.filterConfig = config;  }  } |

步骤2：在web.xml文件中配置filter过滤器

|  |
| --- |
| <filter>  <filter-name>testFilter</filter-name>  <filter-class>com.gh.filter.FilterTest</filter-class>  <init-param>  <param-name>param1</param-name>  <param-value>value在这里呢</param-value>  </init-param>  </filter>  <filter-mapping>  <filter-name>testFilter</filter-name>  <!-- 表示过滤所有请求，也可以设置过滤某个包下的请求，比如page/\* -->  <url-pattern>/\*</url-pattern>  <!-- 过滤特定的servlet -->  <!-- <servlet-name>test</servlet-name> -->  <dispatcher>REQUEST</dispatcher> <!-- 没有配置dispatcher就是默认request方式的 -->  <dispatcher>FORWARD</dispatcher>  <dispatcher>ERROR</dispatcher>  <dispatcher>INCLUDE</dispatcher>  </filter-mapping>  <servlet>  <servlet-name>test</servlet-name>  <servlet-class>com.gh.encoding.MgrDataHandler</servlet-class>  </servlet>  <servlet-mapping>  <servlet-name>test</servlet-name>  <url-pattern>/test</url-pattern>  </servlet-mapping> |

1. **设置request编码（post方式）**

写一个过虑器，对所有url全部过虑，/\*.在doFilter方法中，设置request的编码为utf-8。一般情况下，这个过虑器永远是第一个要执行的过虑器。

最好是通过配置设置编码。<filter><init-param>

此方式只对post请求参数有效，get请求方式无效。

第一步：实现Filter接口，在doFIlter中接收初始化参数，设置编码

|  |
| --- |
| **public** **class** CharFilter **implements** Filter {  //声明编码的成员变量  **private** String encoding;  **public** **void** init(FilterConfig config) **throws** ServletException {  encoding = config.getInitParameter("bm");  }  **public** **void** doFilter(ServletRequest request, ServletResponse response,  FilterChain chain) **throws** IOException, ServletException {  request.setCharacterEncoding(encoding);  //放行,必须要放行。  chain.doFilter(request, response);  }  **public** **void** destroy() {  }  } |

第二步：将上面的类配置到web.xml

|  |
| --- |
| <filter>  <filter-name>char</filter-name>  <filter-class>cn.itcast.filter.CharFilter</filter-class>  <init-param>  <!-- 为了便于配置，在配置文件中设置编码 -->  <param-name>bm</param-name>  <param-value>UTF-8</param-value>  </init-param>  </filter>  <filter-mapping>  <filter-name>char</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping> |

1. **设置request编码（post和get）**

get请求和post请求不一样，参数是跟在url后面，该参数首先会进行Unicode编码，然后转为ISO-8859-1，通过网络传输到web服务器，web服务器收到请求后，才请求取出参数，此时参数是ISO-8859-1的，因此通过ISO-8859-1转为字节数组，然后再将字节数组转为UTF-8，这样才能正确解析参数。

此处我们使用包装类来处理get的请求参数的编码

在CharFilter中对reuqest进行包装。目的：修改增强getParameter方法

第一步：声明包装类

|  |
| --- |
| // 声明包装类  **class** MyRequest **extends** HttpServletRequestWrapper {  **public** MyRequest(HttpServletRequest request) {  **super**(request);  }  // 增强getParamter  @Override  **public** String getParameter(String name) {  String val = **super**.getParameter(name);  **try** {  val = **new** String(val.getBytes("ISO-8859-1"),  **super**.getCharacterEncoding());  } **catch** (UnsupportedEncodingException e) {  e.printStackTrace();  }  **return** val;  }  } |

第二步：在doFilter方法中，声明包装类的实例

|  |
| --- |
| **public** **class** CharFilter **implements** Filter {  // 声明编码的成员变量  **private** String encoding;  **public** **void** init(FilterConfig config) **throws** ServletException {  encoding = config.getInitParameter("bm");  }  **public** **void** doFilter(ServletRequest request, ServletResponse response,  FilterChain chain) **throws** IOException, ServletException {  request.setCharacterEncoding(encoding);//对get没有效  response.setContentType("text/html;charset=" + encoding);  // 判断是否需要包装  HttpServletRequest req = (HttpServletRequest) request;  **if** (req.getMethod().equals("GET")) {  request = **new** MyRequest(req);  }  // 声明包装类的实例  // 放行  chain.doFilter(request, response);  }  **public** **void** destroy() {  }  } |

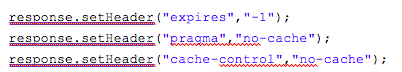
声明包装类

|  |
| --- |
| **class** MyRequest **extends** HttpServletRequestWrapper {  **public** MyRequest(HttpServletRequest request) {  **super**(request);  }  // 增强getParamter  @Override  **public** String getParameter(String name) {  String val = **super**.getParameter(name);  **try** {  val = **new** String(val.getBytes("ISO-8859-1"),  **super**.getCharacterEncoding());  } **catch** (UnsupportedEncodingException e) {  e.printStackTrace();  }  **return** val;  }  } |

此处省略了配置文件web.xml内容

1. **设置页面缓存或不缓存**

要想让所有浏览器不缓存页面（IE、火狐、Chrome）：



分析：通过一个过虑器，对所有\*.jsp过虑，设置三个头，全部不缓存。

第一步：实现过虑器接口

|  |
| --- |
| **public** **void** doFilter(ServletRequest request, ServletResponse response,  FilterChain chain) **throws** IOException, ServletException {  //转换  HttpServletResponse resp = (HttpServletResponse) response;  resp.setHeader("expires","-1");  resp.setHeader("pragma","no-cache");  resp.setHeader("cache-control","no-cache");  //都放行  chain.doFilter(request, response);  } |

第二步：配置过虑器，url-pattern=\*.jsp

|  |
| --- |
| <!-- 配置控制缓存的filter -->  <filter>  <filter-name>cache</filter-name>  <filter-class>cn.itcast.filter.CacheFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>cache</filter-name>  <url-pattern>\*.jsp</url-pattern>  </filter-mapping> |

设置对某些页面缓存N天

resp.setDateHeader("expires", time);

time:是毫秒数

1. **包装模式**
2. **使用过滤器对数据进行压缩**

**功能描述：**用户发来一个请求，服务器读取磁盘的一个文件，将文件数据压缩后返回给用户并显示。

**用到的技术：**读取文件用File类，数据压缩用GZIPOutputStream类

实现：过滤器

|  |
| --- |
| **public** **class** FilterTest **implements** Filter{  **private** FilterConfig filterConfig;  /\*\*  \* 销毁函数，在tomcat正常关闭的时候执行，在tomcat异常关闭的时候可能不会执行  \*/  @Override  **public** **void** destroy() {  System.*out*.println("销毁了");  }  /\*\*  \* 当有用户请求过来的时候执行，该函数主要进行过滤功能  \*/  @Override  **public** **void** doFilter(ServletRequest req, ServletResponse res,  FilterChain chain) **throws** IOException, ServletException {  System.*out*.println("Demo1过滤前");  //声明MyResponse包装类  // MyResponseWrapper resp = new MyResponseWrapper((HttpServletResponse) res);  //放行时，必须要传递自己包装过的类  chain.doFilter(req, res);  //目标类调用完成了,返回 以后读取a.txt  File file = **new** File("e:/a/a.txt");  //声明一个缓存的字符串对象  StringBuilder sb = **new** StringBuilder();  //读取文件  InputStream in = **new** FileInputStream(file);  **byte**[] b = **new** **byte**[1024];  **int** len = 0;  **while**((len=in.read(b))!=-1){  sb.append(**new** String(b,0,len,"gbk"));  }  in.close();  //转成字节开始压缩,gbk转成utf-8  **byte**[] src = sb.toString().getBytes("UTF-8");  //声明缓存容器  ByteArrayOutputStream destBytes = **new** ByteArrayOutputStream();  //声明压缩流  GZIPOutputStream gzip = **new** GZIPOutputStream(destBytes);  //压缩数据  gzip.write(src);  gzip.close();  //获取压缩以后数据  **byte**[] dest = destBytes.toByteArray();  System.*err*.println("压缩之前："+src.length);  System.*err*.println("压缩以后："+dest.length);  //输出  //必须要使用原生 的response  HttpServletResponse response = (HttpServletResponse) res;  res.setContentType("text/html;charset=UTf-8");  OutputStream out = res.getOutputStream();  response.setHeader("Content-encoding", "gzip");//必须  response.setContentLength(dest.length);  out.write(dest);  System.*out*.println("Demo1过滤后");  }  /\*\*  \* 在tomcat启动的时候执行  \*/  @Override  **public** **void** init(FilterConfig config) **throws** ServletException {  System.*out*.println("初始化了");  **this**.filterConfig = config;  }  } |

1. **对jsp文件进行压缩**

使用过滤器对jsp进行压缩

思想：

因为jsp所有的数据都是通过response.getWriter()获取的，然后out.write()方法输出内容到jsp，因此我们可以将out.write的数据统一压缩，再返回到jsp页面。

首先我们新建一个response的实现类，也是response的一个包装类，将out.write()的数据缓存到一个容器中或者一个文件中，然后声明一个过滤器，在chain.doFilter()方法执行后，将out.write()缓存的数据进行压缩，然后调用原生response的write()方法将压缩后的数据写入到jsp页面。

新建servlet

|  |
| --- |
| /\*\*   \* 此文件正常输出信息?times=9   \*/  **public** **class** OneServlet **extends** HttpServlet {  **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  String times = request.getParameter("times");  **int** time = Integer.*parseInt*(times);  response.setContentType("text/html;charset=UTF-8");  OutputStream out = response.getOutputStream();  **for**(**int** i=0;i<time;i++){  out.write("你好Hello".getBytes("UTF-8"));  }  }  } |

新建过滤器，使用了包装类

|  |
| --- |
| **public** **class** GzipFilter2 **implements** Filter {  **public** **void** init(FilterConfig filterConfig) **throws** ServletException {  }  **public** **void** doFilter(ServletRequest request, ServletResponse response,  FilterChain chain) **throws** IOException, ServletException {  HttpServletResponse resp= (HttpServletResponse) response;  //声明包装类对象  MyResponse2 myresp = **new** MyResponse2(resp);  //放行,调用oneServlet.doGet  chain.doFilter(request, myresp);    //第二步：从myresp2中读取原生的数据（未压缩的数据）  **byte**[] src = myresp.getSrc();    //第三步：开始压缩  ByteArrayOutputStream destBytes = **new** ByteArrayOutputStream();  GZIPOutputStream zip = **new** GZIPOutputStream(destBytes);  zip.write(src);  zip.close();    //第三步：输出-使用原生的response  resp.setContentType("text/html;charset=UTF-8");  //获压缩以后数据  **byte**[] dest = destBytes.toByteArray();  System.***err***.println("压缩之前："+src.length);  System.***err***.println("压缩以后："+dest.length);  //设置头  resp.setContentLength(dest.length);  resp.setHeader("Content-Encoding","gzip");  //输出  OutputStream out = resp.getOutputStream();  out.write(dest);  }  **public** **void** destroy() {  }  }  //第一步：声明response的包装类  **class** MyResponse2 **extends** HttpServletResponseWrapper{  //将这个容器/a.txt,声明成员变量  **private** ByteArrayOutputStream srcByte;  **private** PrintWriter out;  **public** MyResponse2(HttpServletResponse response) {  **super**(response);  }  //修改增强getWtier方法  @Override  **public** PrintWriter getWriter() **throws** IOException {  srcByte = **new** ByteArrayOutputStream();  out = **new** PrintWriter(  **new** OutputStreamWriter(srcByte, "UTF-8"));  **return** out;  }  //覆盖  @Override  **public** ServletOutputStream getOutputStream() **throws** IOException {  srcByte = **new** ByteArrayOutputStream();  ServletOutputStream out = **new** ServletOutputStream() {  //所有IO最终都是一个个字节写出信息  @Override  **public** **void** write(**int** b) **throws** IOException {  System.***err***.println(">>>:"+b);  srcByte.write(b);//写到自己的缓存中去-相当于StringBuffer.append(""+b);  }  @Override  **public** **boolean** isReady() {  **return** **false**;  }  @Override  **public** **void** setWriteListener(WriteListener arg0) {  }  };  **return** out;  }  //提供一个方法获取原生的数据  **public** **byte**[] getSrc(){  **if**(out!=**null**){  out.close();  }  **return** srcByte.toByteArray();  }  } |

配置web.xml文件

|  |
| --- |
| <filter-mapping>    <filter-name>zip2</filter-name>    <servlet-name>OneServlet</servlet-name>  <url-pattern>\*.jsp</url-pattern>    </filter-mapping>    <servlet>      <servlet-name>OneServlet</servlet-name>      <servlet-class>cn.itcast.servlet.OneServlet</servlet-class>    </servlet> |

HttpServletResponseWrapper类是对HttpServletResponse接口的实现，我们实现HttpServletResponseWrapper类比用HttpServletResponse类更方便。

1. **监听器Listener**
2. **概念**

监听器就是一个实现待定接口的普通Java程序，此程序专门用于监听另一个类的方法调用。监听器都是使用观察者设计模式。

监听器存在以下对象：

监听者：XxxxxListener － 所有的监听者是一个接口。

被监听者 ：任意对象都可以成为被监听者 － 早已经存在。

监听到的事件：XxxxEvent－ 永远是一个具体类，用来放监听到的数据，里面都有一个方法叫getSource() – 返回的是监听到对象。

在Javaweb中存在三个被监听对象:

HttpServletRequest

HttpSessoin

ServletContext

活化：化石变动物

钝化：动物变化石



1. **应用场景**

分类：

①**ContextListener应用场景：**

记录一个网站的刷新量。

当服务器关闭时，必须要保存到文件中或是数据库中去。

当服务器启动时，先从文件中读取并放到ServletContext。

在Filter中记录访问量，每次访问都加1。

好处：信息不是太重要，没有必要每次用户访问都访问数据库或是操作文件。在为不影响用户的速度感受，应该开始一个新的线程去操作数据。这样即使在后台使用同步技术，用户也不会感觉到速度很慢。

②**HttpSession的监听器：**

监听HttpSession的创建和销毁。

③**Session的属性设置监听事件**

主要作用是：记录当前在登录人数。注意，登录是指注册成为合法用户并成功登录的人。

只要有人访问本站点，成功创建一个Session就是一个会话，一个会话就是一个在线人数。

用HttpSSessionListener来监听Session的创建。

④**ServletRequestListener：**经常用来记录访问量。

1. **观察者模式**

|  |
| --- |
| **public** **class** TestObersver {  **public** **static** **void** main(String[] args) {  Person person = **new** Person();//声明被观察者  System.***err***.println("pp:"+person);  person.addPersonListener(**new** PersonListener() {  **public** **void** running(PersonEvent pe) {  System.***err***.println("你正在跑....."+pe.getSource());  **throw** **new** RuntimeException("他跑了。。。");  }  });  person.run();  }  }  **class** Person{  **private** PersonListener pl;  **public** **void** addPersonListener(PersonListener pl){  **this**.pl = pl;  }  **public** **void** run(){  **if**(pl!=**null**){  pl.running(**new** PersonEvent(**this**));  }  System.***err***.println("我正在跑步......");  }  }  **interface** PersonListener{  **void** running(PersonEvent pe);  }  **class** PersonEvent{  **private** Object src;  **public** PersonEvent(Object obj) {  **this**.src=obj;  }  **public** Object getSource(){  **return** src;  }  } |

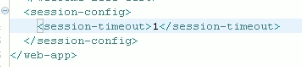
1. **统计在线人数**

只要有人访问本站点，成功创建一个Session就是一个会话，一个会话就是一个在线人数。

用HttpSSessionListener来监听Session的创建。

|  |
| --- |
| **public** **class** MySessionListener **implements** HttpSessionListener {  **private** Integer online=1;  //request.getSession();  **public** **void** sessionCreated(HttpSessionEvent se) {  System.*err*.println("有人访问本网点了");  HttpSession hs1 = (HttpSession) se.getSource();  HttpSession hs2 = se.getSession();  System.*err*.println("hs1:"+hs1.getId()+","+hs2.getId()+",ip:");  //获取整个域的对象  ServletContext sc= se.getSession().getServletContext();  sc.setAttribute("online",online++);  //将所有session放到servletContext  //先从application获取所有已经维护的sesison  List<HttpSession> list = (List<HttpSession>) sc.getAttribute("sessions");  **if**(list==**null**){//第一个访问的人  list = **new** ArrayList<HttpSession>();  sc.setAttribute("sessions", list);  }  list.add(hs2);  }  //过期(30),s.invalidate();  **public** **void** sessionDestroyed(HttpSessionEvent se) {  System.*err*.println("有人退出了..."+se.getSession().getId());  ServletContext sc= se.getSession().getServletContext();  sc.setAttribute("online",online--);  }  } |

在web.xml设置session过期时间



1. **简单javaweb中的监听器demo**

实现：

**package** com.gh.listener;

**import** javax.servlet.ServletContextEvent;

**import** javax.servlet.ServletContextListener;

**public** **class** ListenerTest **implements** ServletContextListener {

**private** Thread thread = **new** Thread(**new** Runnable() {

**private** **int** count = 0;

@Override

**public** **void** run() {

**for** (**int** i = 0; i < 10000000; i++) {

**try** {

Thread.*sleep*(2000);

System.*out*.println(count++);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

});

@Override

**public** **void** contextDestroyed(ServletContextEvent arg0) {

System.*out*.println("监听器销毁 了");

thread.~~stop~~();

}

@Override

**public** **void** contextInitialized(ServletContextEvent arg0) {

System.*out*.println("监听器创建了");

thread.start();

}

}

在web.xml中注册

<listener>

<listener-class>com.gh.listener.ListenerTest</listener-class>

</listener>

说明：

1:配置一个Listener只要求提供类名就可以了。

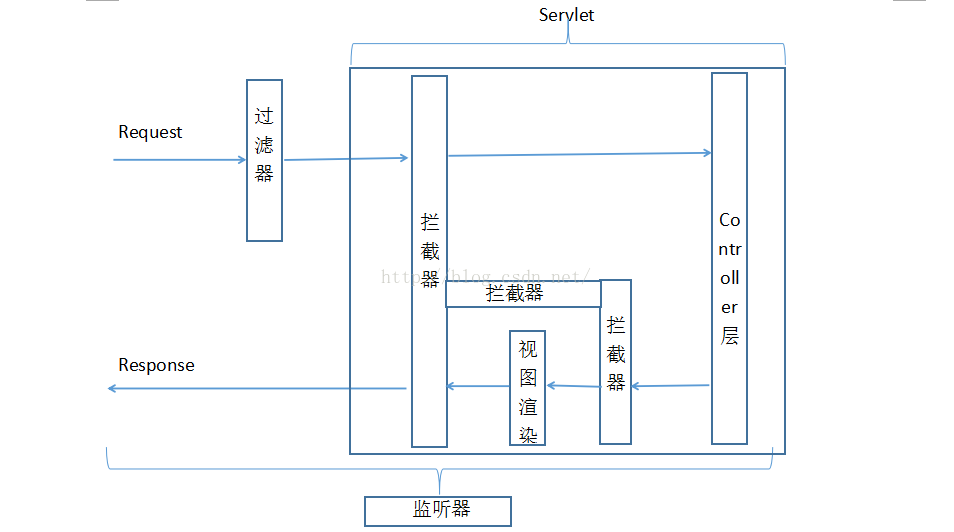
2：在tomcat启动时，会自动的初始化这个监听器类。

3：tomcat创建的Listner,Serlvet,Filter都是单例的.

1. **监听器、拦截器、过滤器**

**概念：**

从整个项目中看，一个servlet请求的执行过程就变成了这样context-param-->listener-->filter-->servlet-->interceptor(指的是拦截器)，为什么拦截器是在servlet执行之后，因为拦截器本身就是在servlet内部的。



context-param：就是一些需要初始化的配置，放入context-param中，从而被监听器（这里特指org.springframework.web.context.ContextLoaderListener）监听，然后加载；

监听器(listener)：就是对项目起到监听的作用，它能感知到包括request(请求域)，session(会话域)和applicaiton(应用程序)的初始化和属性的变化；

过滤器(filter)：就是对请求起到过滤的作用，它在监听器之后，作用在servlet之前，对请求进行过滤；

servlet：就是对request和response进行处理的容器，它在filter之后执行，servlet其中的一部分就是controller层（标记为servlet\_2），还包括渲染视图层(标记为servlet\_3)和进入controller之前系统的一些处理部分(servlet\_1)，另外我们把servlet开始的时刻标记为servlet\_0，servlet结束的时刻标记为servlet\_4。

拦截器(interceptor)：就是对请求和返回进行拦截，它作用在servlet的内部，具体来说有三个地方：

1）servlet\_1和servlet\_2之间，即请求还没有到controller层

2）servlet\_2和servlet\_3之间，即请求走出controller层次，还没有到渲染时图层

3）servlet\_3和servlet\_4之间，即结束视图渲染，但是还没有到servlet的结束

**使用原则：**

对整个流程清楚之后，然后就是各自的使用，在使用之前应该有一个使用规则，为什么这个说，因为有些功能比如判断用户是否登录，既可以用过滤器，也可以用拦截器，用哪一个才是合理的呢？那么如果有一个原则，使用起来就会更加合理。实际上这个原则是有的：

把整个项目的流程比作一条河，那么监听器的作用就是能够听到河流里的所有声音，过滤器就是能够过滤出其中的鱼，而拦截器则是拦截其中的部分鱼，并且作标记。所以当需要监听到项目中的一些信息，并且不需要对流程做更改时，用监听器；当需要过滤掉其中的部分信息，只留一部分时，就用过滤器；当需要对其流程进行更改，做相关的记录时用拦截器。下面是具体的使用案例。

监听器：

1. 获取当前在线人数

过滤器：

（1）进行编码设置、压缩、访问日志记录、对未登陆用户的判断

拦截器：

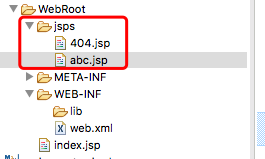
（1）全局日志

1. **配置错误和异常页面**

在web项目中的web.xml配置文件添加如下配置



页面结构:



1. **Base64编码**

**原理：**把3个字符变成4个字符。每76个字符加一个换行符。

**例:1：**将3个字符转为4个字符，前面的两位用00代替

转换前 11111111, 11111111, 11111111 （二进制）

转换后 00111111, 00111111, 00111111, 00111111 （二进制）

**例子2：**

转换前 10101101,10111010,01110110

转换后 00101011, 00011011 ,00101001 ,00110110

十进制 43 27 41 54

对应码表中的值 r b p 2

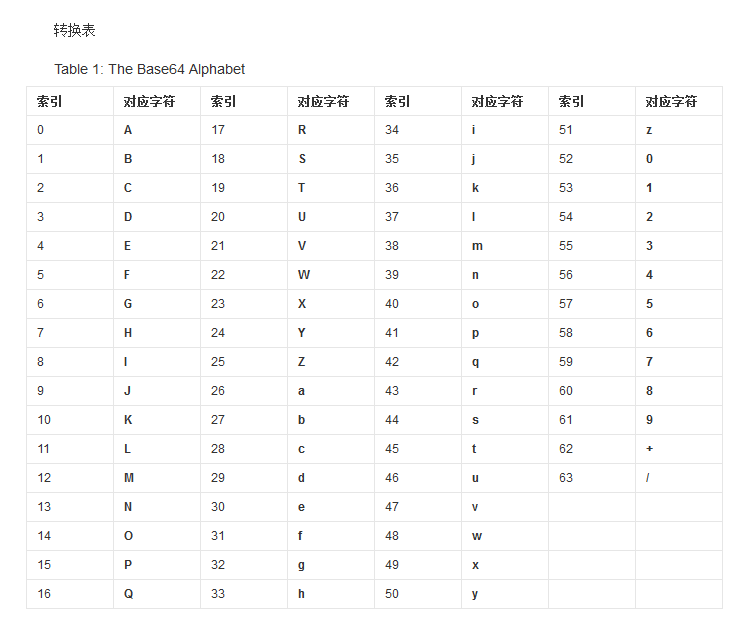
所以上面的24位编码，编码后的Base64值为 rbp2

解码同理，把 rbq2 的二进制位连接上再重组得到三个8位值，得出原码。

**作用场景：**Base64常用于在通常处理文本数据的场合，表示、传输、存储一些二进制数据。包括MIME的email、在XML中存储复杂数据。

在MIME格式的电子邮件中，base64可以用来将binary的字节序列数据编码成ASCII字符序列构成的文本。使用时，在传输编码方式中指定base64。使用的字符包括大小写字母各26个，加上10个数字，和加号“+”，斜杠“/”，一共64个字符，等号“=”用来作为后缀用途。

完整的base64定义可见RFC 1421和RFC 2045。编码后的数据比原始数据略长，为原来的43。在电子邮件中，根据RFC 822规定，每76个字符，还需要加上一个回车换行。可以估算编码后数据长度大约为原长的135.1%。



**Java实例：**

对用户名和密码进行base64编码



String name = "one";

String pwd = "1234";

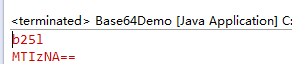
BASE64Encoder en = **new** BASE64Encoder();

name = en.encode(name.getBytes());

pwd = en.encode(pwd.getBytes());

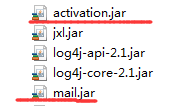
System.*err*.println(name);

System.*err*.println(pwd);



1. **发送简单邮件Mail**

**依赖的包：**



**import** java.util.Date;

**import** java.util.Properties;

**import** javax.mail.Address;

**import** javax.mail.Authenticator;

**import** javax.mail.Message.RecipientType;

**import** javax.mail.PasswordAuthentication;

**import** javax.mail.Session;

**import** javax.mail.Transport;

**import** javax.mail.internet.InternetAddress;

**import** javax.mail.internet.MimeMessage;

**import** org.junit.Test;

**public** **class** SendMailDemo {

//发一封最简单的邮件

@Test

**public** **void** sendMail() **throws** Exception{

//第一步：声明properties对象放信息

Properties prop = **new** Properties();

//设置连接哪一台服务器，发送邮件的服务器，在这里使用的是163 的邮件发送服务器

prop.setProperty("mail.host","smtp.163.com");

//设置是否验证

prop.setProperty("mail.smtp.auth", "true");

//第二步：声明用户名和密码

Authenticator auth = **new** Authenticator() {

//此访求返回用户和密码的对象

**public** PasswordAuthentication getPasswordAuthentication() {

//发件人的用户名以及密码

PasswordAuthentication pa = **new** PasswordAuthentication("wyyg6868", "wyyg666666");

**return** pa;

}

};

////第二步：获取Session对象

Session session =

Session.*getDefaultInstance*(prop,auth);

//设置session的调试模式

session.setDebug(**true**);

//第三步：声明信息

MimeMessage mm1 =

**new** MimeMessage(session);

//第四步：设置发件人email

Address from = **new** InternetAddress("wyyg6868@163.com");

mm1.setFrom(from);

//第五步：设置收件人

mm1.setRecipient(RecipientType.***TO***,**new** InternetAddress("982452937@qq.com"));

//第六步：设置主题

mm1.setSubject("这是用Java发One.....的邮件sfasdf3");

//设置发送日期

mm1.setSentDate(**new** Date(1000000000));

//设置邮件内容

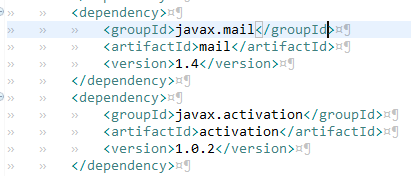
mm1.setContent("你好，这是用java发的邮件111", "text/html;charset=UTF-8");

//第七步：

Transport.*send*(mm1);

}

}





注：发送的邮件很多会被归入垃圾邮箱

1. **发送带图片等文件的邮件mail**

**具体源码见：百度云盘-技术资料-web源码-发送邮件**

**import** java.io.File;

**import** java.util.Properties;

**import** javax.activation.DataHandler;

**import** javax.activation.DataSource;

**import** javax.activation.FileDataSource;

**import** javax.mail.Authenticator;

**import** javax.mail.Message.RecipientType;

**import** javax.mail.PasswordAuthentication;

**import** javax.mail.Session;

**import** javax.mail.Transport;

**import** javax.mail.internet.InternetAddress;

**import** javax.mail.internet.MimeBodyPart;

**import** javax.mail.internet.MimeMessage;

**import** javax.mail.internet.MimeMultipart;

**import** javax.mail.internet.MimeUtility;

**import** org.junit.Test;

**public** **class** SendAttechmentDemo {

@Test

**public** **void** sendFile() **throws** Exception{

Properties p = **new** Properties();

p.setProperty("mail.host","smtp.163.com");

p.setProperty("mail.smtp.auth","true");

Session s = Session.*getDefaultInstance*(p,**new** Authenticator() {

@Override

**public** PasswordAuthentication getPasswordAuthentication() {

**return** **new** PasswordAuthentication("wj\_leaf12345", "1qaz2wsx");

}

});

s.setDebug(**true**);

//声明MimeMessage

MimeMessage msg = **new** MimeMessage(s);

msg.setFrom(**new** InternetAddress("wj\_leaf12345@163.com"));

msg.setRecipient(RecipientType.*TO*, **new** InternetAddress("wyyg666666@163.com"));

msg.setSubject("图片的");

//第一步：声明多处理的Part

MimeMultipart mm = **new** MimeMultipart();

//第二步：声明

MimeBodyPart body1 = **new** MimeBodyPart();

//第三步：设置符件

DataSource ds = **new** FileDataSource(**new** File("./img/a.jpg"));

DataHandler dh = **new** DataHandler(ds);

body1.setDataHandler(dh);

//必须要设置名称

body1.setFileName(MimeUtility.*encodeText*("美女.jpg"));

MimeBodyPart body2 = **new** MimeBodyPart();

//第三步：设置符件

DataSource ds2 = **new** FileDataSource(**new** File("./img/b.jpg"));

DataHandler dh2 = **new** DataHandler(ds2);

body2.setDataHandler(dh2);

//必须要设置名称

body2.setFileName(MimeUtility.*encodeText*("美女2.jpg"));

MimeBodyPart body3 = **new** MimeBodyPart();

//第三步：设置符件

DataSource ds3 = **new** FileDataSource(**new** File("./img/m.mp3"));

DataHandler dh3 = **new** DataHandler(ds3);

body3.setDataHandler(dh3);

//必须要设置名称

body3.setFileName(MimeUtility.*encodeText*("世纪末.mp3"));

//将body1添加到mm

mm.addBodyPart(body1);

mm.addBodyPart(body2);

mm.addBodyPart(body3);

msg.setContent(mm);

//发送

Transport.*send*(msg);

}

}

1. **poi**
2. **java对象数据导出为excel表格**

依赖的jar包：



第一步：新建表格对应的对象，省略get和set方法

**import** java.util.Date;

**public** **class** Student {

**private** **int** id;

**private** String name;

**private** **int** age;

**private** Date birth;

**public** Student()

{

}

**public** Student(**int** id, String name, **int** age, Date birth)

{

**this**.id = id;

**this**.name = name;

**this**.age = age;

**this**.birth = birth;

}

}

第二步：

**import** java.io.FileOutputStream;

**import** java.text.SimpleDateFormat;

**import** java.util.ArrayList;

**import** java.util.List;

**import** org.apache.poi.hssf.usermodel.HSSFCell;

**import** org.apache.poi.hssf.usermodel.HSSFCellStyle;

**import** org.apache.poi.hssf.usermodel.HSSFRow;

**import** org.apache.poi.hssf.usermodel.HSSFSheet;

**import** org.apache.poi.hssf.usermodel.HSSFWorkbook;

**import** com.gh.bean.Student;

**public** **class** ExportExcel {

/\*\*

\* **@功能**：手工构建一个简单格式的Excel

\*/

**private** **static** List<Student> getStudent() **throws** Exception

{

List list = **new** ArrayList();

SimpleDateFormat df = **new** SimpleDateFormat("yyyy-mm-dd");

Student user1 = **new** Student(1, "张三", 16, df.parse("1997-03-12"));

Student user2 = **new** Student(2, "gh", 17, df.parse("1996-08-12"));

Student user3 = **new** Student(3, "王五", 26, df.parse("1985-11-12"));

list.add(user1);

list.add(user2);

list.add(user3);

**return** list;

}

**public** **static** **void** main(String[] args) **throws** Exception

{

// 第一步，创建一个webbook，对应一个Excel文件

HSSFWorkbook wb = **new** HSSFWorkbook();

// 第二步，在webbook中添加一个sheet,对应Excel文件中的sheet

HSSFSheet sheet = wb.createSheet("学生表一");

// 第三步，在sheet中添加表头第0行,注意老版本poi对Excel的行数列数有限制short

HSSFRow row = sheet.createRow((**int**) 0);

// 第四步，创建单元格，并设置值表头 设置表头居中

HSSFCellStyle style = wb.createCellStyle();

style.setAlignment(HSSFCellStyle.*ALIGN\_CENTER*); // 创建一个居中格式

HSSFCell cell = row.createCell((**short**) 0);

cell.~~setCellValue~~("学号");

cell.setCellStyle(style);

cell = row.createCell((**short**) 1);

cell.~~setCellValue~~("姓名");

cell.setCellStyle(style);

cell = row.createCell((**short**) 2);

cell.~~setCellValue~~("年龄");

cell.setCellStyle(style);

cell = row.createCell((**short**) 3);

cell.~~setCellValue~~("生日");

cell.setCellStyle(style);

// 第五步，写入实体数据 实际应用中这些数据从数据库得到，

List list = ExportExcel.*getStudent*();

**for** (**int** i = 0; i < list.size(); i++)

{

row = sheet.createRow((**int**) i + 1);

Student stu = (Student) list.get(i);

// 第四步，创建单元格，并设置值

row.createCell((**short**) 0).setCellValue((**double**) stu.getId());

row.createCell((**short**) 1).~~setCellValue~~(stu.getName());

row.createCell((**short**) 2).setCellValue((**double**) stu.getAge());

cell = row.createCell((**short**) 3);

cell.~~setCellValue~~(**new** SimpleDateFormat("yyyy-mm-dd").format(stu

.getBirth()));

}

**try**

{ // 第六步，将文件存到指定位置 ,保存到e盘

FileOutputStream fout = **new** FileOutputStream("E:/students.xls");

wb.write(fout);

fout.close();

}

**catch** (Exception e)

{

e.printStackTrace();

}

}

}

1. **导出数据库所有表数据到Excel**

|  |
| --- |
| @Test  **public** **void** export() **throws** Exception{  //声明需要导出的数据库  String dbName = "focus";  //声明book  HSSFWorkbook book = **new** HSSFWorkbook();  //获取Connection,获取db的元数据  Connection con = DataSourceUtils.*getConn*();  //声明statemen  Statement st = con.createStatement();  //st.execute("use "+dbName);  DatabaseMetaData dmd = con.getMetaData();  //获取数据库有多少表  ResultSet rs = dmd.getTables(dbName,dbName,**null**,**new** String[]{"TABLE"});  //获取所有表名　－　就是一个sheet  List<String> tables = **new** ArrayList<String>();  **while**(rs.next()){  String tableName = rs.getString("TABLE\_NAME");  tables.add(tableName);  }  **for**(String tableName:tables){  HSSFSheet sheet = book.createSheet(tableName);  //声明sql  String sql = "select \* from "+dbName+"."+tableName;  //查询数据  rs = st.executeQuery(sql);  //根据查询的结果，分析结果集的元数据  ResultSetMetaData rsmd = rs.getMetaData();  //获取这个查询有多少行  **int** cols = rsmd.getColumnCount();  //获取所有列名  //创建第一行  HSSFRow row = sheet.createRow(0);  **for**(**int** i=0;i<cols;i++){  String colName = rsmd.getColumnName(i+1);  //创建一个新的列  HSSFCell cell = row.createCell(i);  //写入列名  cell.setCellValue(colName);  }  //遍历数据  **int** index = 1;  **while**(rs.next()){  row = sheet.createRow(index++);  //声明列  **for**(**int** i=0;i<cols;i++){  String val = rs.getString(i+1);  //声明列  HSSFCell cel = row.createCell(i);  //放数据  cel.setCellValue(val);  }  }  }  con.close();  book.write(**new** FileOutputStream("d:/a/"+dbName+".xls"));  } |

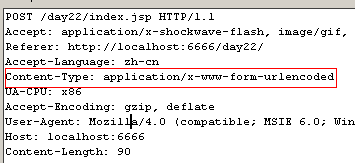
1. **文件上传和下载**
2. **向服务器上传文件**

**核心**

1)、用<input type=”file”/> 来声明一个文件域。File:\_\_\_\_\_ <浏览>.

2）、必须要使用post方式的表单。

3）、必须设置表单的类型（content-type）为multipart/form-data.是设置这个表单传递的不是key=value值。传递的是字节码。

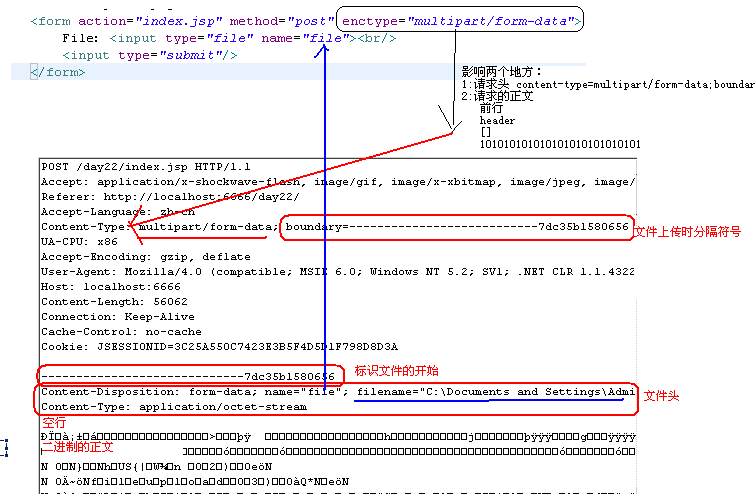


对于一个普通的表单来说只要它是post类型。默认就是

Content-type：application/x-www-from-urlencoded

如果要实现文件上传，必须设置Content-type=“multipart/form-data”

**表单与请求的对应关系：**



如果不设置enctype的值，就会取默application/x-www-form-urlencoded，

这里传递的是键值对key-value，enctype属性只有两个值可选。

**上传文件**

|  |
| --- |
| /\*\*  \* 如果一个表单的类型是post且enctype为multipart/form-date  \* 则所有数据都是以二进制的方式向服务器上传递。  \* 所以req.getParameter("xxx")永远为null。  \* 只可以通过req.getInputStream()来获取数据,获取正文的数据  \*/  **public** **class** UpServlet **extends** HttpServlet {  **public** **void** doPost(HttpServletRequest req, HttpServletResponse resp)  **throws** ServletException, IOException {  req.setCharacterEncoding("UTF-8");  String txt = req.getParameter("txt");//返回的是null  System.*err*.println("txt is :"+txt);  System.*err*.println("=========================================");  InputStream in = req.getInputStream();  // byte[] b= new byte[1024];  // int len = 0;  // while((len=in.read(b))!=-1){  // String s = new String(b,0,len);  // System.err.print(s);  // }  BufferedReader br = **new** BufferedReader(**new** InputStreamReader(in));  String firstLine = br.readLine();//读取第一行，且第一行是分隔符号  String fileName = br.readLine();  fileName = fileName.substring(fileName.lastIndexOf("\\")+1);// bafasd.txt"  fileName = fileName.substring(0,fileName.length()-1);    br.readLine();  br.readLine();  String data = **null**;  //获取当前项目的运行路径  String projectPath = getServletContext().getRealPath("/up");  PrintWriter out = **new** PrintWriter(projectPath+"/"+fileName);  **while**((data=br.readLine())!=**null**){  **if**(data.equals(firstLine+"--")){  **break**;  }  out.println(data);  }  out.close();  }  } |

**使用apache-Fileupload处理文件上传**

**核心类：**

DiskFileItemFactory – 设置磁盘空间，保存临时文件。只是一个具类。

ServletFileUpload － 文件上传的核心类，此类接收request,并解析reqeust。

servletfileUpload.parseRequest(requdest) - List<FileItem>

一个FileItem就是一个标识的开始：---------243243242342 到 ------------------245243523452—就是一个FileItem

**依赖的jar包：**



**实现步骤：**

设置enctype属性的值为multipart/form-data，其次method必须为post

<form action=*"http://localhost:8080/extend/file"* enctype=*"multipart/form-data"* method=*"post"*>

File:<input type=*"file"* name=*"file"*/>

<input type=*"submit"* value=*"提交"*/>

</form>

添加servlet接收上传的文件

/\*\*

\* 如果一个表单的类型是post且enctype为multipart/form-date

\* 则所有数据都是以二进制的方式向服务器上传递。

\* 所以req.getParameter("xxx")永远为null。

\* 只可以通过req.getInputStream()来获取数据,获取正文的数据

\*

\*/

**public** **class** FileServlet **extends** HttpServlet{

**protected** **void** doPost(HttpServletRequest req, HttpServletResponse resp)

**throws** ServletException, IOException {

req.setCharacterEncoding("UTf-8");

System.*out*.println("上传文件");

//获取项目的路径

String path = getServletContext().getRealPath("/");

/\*\*

\* DiskFileItemFactory构造的两个参数

\* 第一个参数：sizeThreadHold - 设置缓存(内存)保存多少字节数据，默认为10K

\* 如果一个文件没有大于10K，则直接使用内存直接保存成文件就可以了。

\* 如果一个文件大于10K，就需要将文件先保存到临时目录中去。

\* 第二个参数 File 是指临时目录位置

\*

\*/

//第一步声明diskfileitemfactory工厂类，用于在指的磁盘上设置一个临时目录

DiskFileItemFactory disk =

**new** DiskFileItemFactory(1024\*10,**new** File("d:/a"));

//第二步：声明ServletFileUpoload，接收上面的临时目录

ServletFileUpload up = **new** ServletFileUpload(disk);

//第三步：解析request

**try** {

**boolean** isMultipart = up.*isMultipartContent*(req);

**if** (isMultipart) {

**try** {

List<FileItem> list = up.parseRequest(req);

//如果就一个文件

FileItem file = list.get(0);

//获取文件名,带路径

String fileName = file.getName();

fileName = fileName.substring(fileName.lastIndexOf("\\")+1);

//获取文件的类型

String fileType = file.getContentType();

//获取文件的字节码

InputStream in = file.getInputStream();

//声明输出字节流

OutputStream out = **new** FileOutputStream(path+"/"+fileName);

//文件copy

**byte**[] b = **new** **byte**[1024];

**int** len = 0;

**while**((len=in.read(b))!=-1){

out.write(b,0,len);

}

out.close();

**long** size = file.getInputStream().available();

//删除上传的临时文件

file.delete();

//显示数据

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

PrintWriter op = resp.getWriter();

op.print("文件上传成功<br/>文件名:"+fileName);

op.print("<br/>文件类型:"+fileType);

op.print("<br/>文件大小（bytes）"+size);

} **catch** (FileUploadException e1) {

e1.printStackTrace();

}

}

} **catch** (Exception e) {

e.printStackTrace();

}

}

@Override

**protected** **void** doGet(HttpServletRequest req, HttpServletResponse resp)

**throws** ServletException, IOException {

**this**.doPost(req, resp);

}

}

配置web.xml文件

<servlet>

<servlet-name>file</servlet-name>

<servlet-class>com.gh.file.FileServlet</servlet-class>

</servlet>

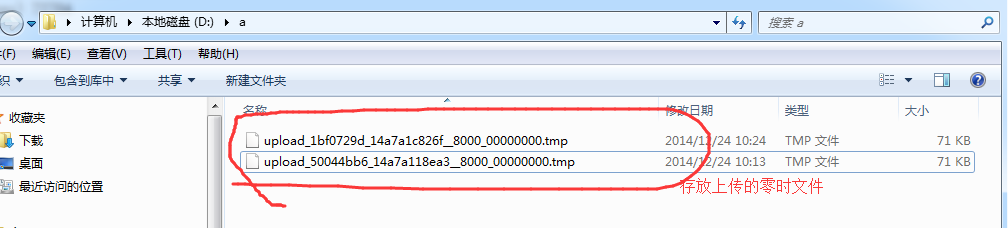
<servlet-mapping>

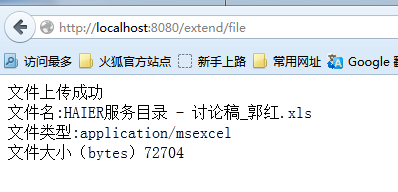
<servlet-name>file</servlet-name>

<url-pattern>/file</url-pattern>

</servlet-mapping>

效果：





**文件重命名**

如果上传带中文名的文件，此时服务器拿到的名称是中文Unicode编码后的值，而不是中文，因此，解决该问题我们可以对拿到的名称进行Unicode解码，但是一般最好是重命名文件，按我们自己的规则去命名文件。

此处省略了很多代码，只贴出关键部分代码



**用COS实现文件上传**

/\*\*

\* 在Cos中就一个类，

\* MultipartRequest它是request的包装类。

\*/

**public** **class** CosServlet **extends** HttpServlet {

**public** **void** doPost(HttpServletRequest request, HttpServletResponse resp)

**throws** ServletException, IOException {

//第一步：声明文件的保存目录

String path = getServletContext().getRealPath("/up");

//第二步：文件传

//声明文件重新取名的策略

FileRenamePolicy rename = **new** DefaultFileRenamePolicy();

MultipartRequest req =

**new** MultipartRequest(request,path,1024\*1024\*100,"UTF-8",**new** MyRename());

// //第三步：显示信息，

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

PrintWriter out = resp.getWriter();

out.print("文件名称1:"+req.getOriginalFileName("img1"));

out.print("<br/>新名称:"+req.getFilesystemName("img1"));

out.print("<br/>类型1:"+req.getContentType("img1"));

out.print("<br/>大小1："+req.getFile("img1").length());

out.print("<br/>说明:"+req.getParameter("desc1"));

**if**(req.getContentType("img1").contains("image/")){

out.print("<img src='"+request.getContextPath()+"/up/"+req.getFilesystemName("img1")+"'></img>");

}

// out.print("<hr/>");

// out.print("文件名称2:"+req.getOriginalFileName("img2"));

// out.print("<br/>类型2:"+req.getContentType("img2"));

// out.print("<br/>大小2："+req.getFile("img2").length());

// out.print("<br/>说明2:"+req.getParameter("desc2"));

//

//

// out.print("<hr/>");

// out.print("文件名称3:"+req.getOriginalFileName("img3"));

// out.print("<br/>类型3:"+req.getContentType("img3"));

// out.print("<br/>大小3："+req.getFile("img3").length());

// out.print("<br/>说明3:"+req.getParameter("desc3"));

}

}

**class** MyRename **implements** FileRenamePolicy{

**public** File rename(File file) {

String fileName = file.getName();

String extName = fileName.substring(fileName.lastIndexOf("."));

String uuid = UUID.*randomUUID*().toString().replace("-","");

String newName = uuid+extName;//abc.jpg

file = **new** File(file.getParent(),newName);

**return** file;

}

}

1. **向服务器下载文件**

**写下载的servlet服务：**

**public** **class** FileDownServlet **extends** HttpServlet{

**public** **void** doPost(HttpServletRequest req, HttpServletResponse resp)

**throws** ServletException, IOException {

req.setCharacterEncoding("UTF-8");

String name = req.getParameter("name");

//第一步：设置响应的类型

resp.setContentType("application/force-download");

//第二读取文件

String path = getServletContext().getRealPath("/"+name);

InputStream in = **new** FileInputStream(path);

//设置响应头

//对文件名进行url编码

name = URLEncoder.*encode*(name, "UTF-8");

resp.setHeader("Content-Disposition","attachment;filename="+name);

resp.setContentLength(in.available());

//第三步：开始文件copy

OutputStream out = resp.getOutputStream();

**byte**[] b = **new** **byte**[1024];

**int** len = 0;

**while**((len=in.read(b))!=-1){

out.write(b,0,len);

}

out.close();

in.close();

}

@Override

**protected** **void** doGet(HttpServletRequest req, HttpServletResponse resp)

**throws** ServletException, IOException {

**this**.doPost(req, resp);

}

}

配置web.xml

<servlet>

<servlet-name>fileDown</servlet-name>

<servlet-class>com.gh.file.FileDownServlet</servlet-class>

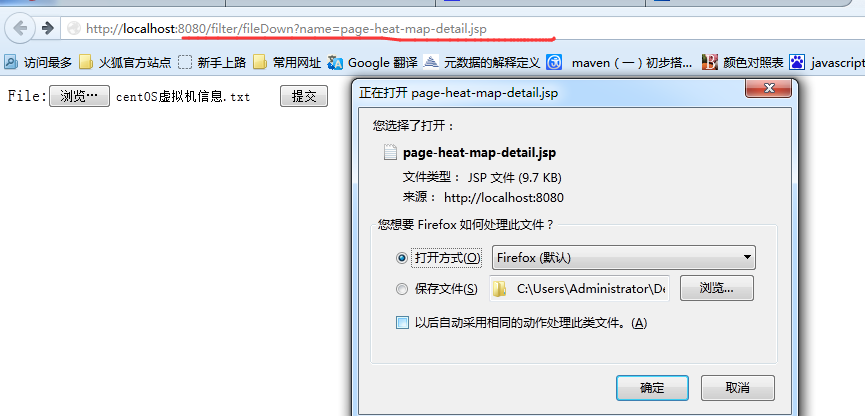
</servlet>

<servlet-mapping>

<servlet-name>fileDown</servlet-name>

<url-pattern>/fileDown</url-pattern>

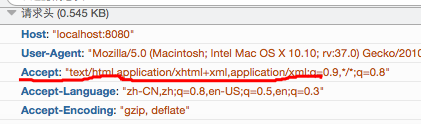
</servlet-mapping>



1. **获取图片验证码，火狐浏览器无端请求两次**

**描述：**在地址栏输入获取验证码的URL，获取验证码，在谷歌和Safari中都只请求一次，但是在火狐浏览器却无端请求两次。

**原因：**在地址栏输入URL获取验证码，请求获取接受的类型默认为text文本，而返回的是图片，谷歌和Safari浏览器则获取到什么就显示什么，而火狐则不同，它会重新再发送一次请求，去请求图片，因此会产生两次请求。



1. **Jsoup和fastjson解析文本数据**

**问题描述：**fastjson是阿里巴巴提供的一个开源json解析框架，jsoup是Java网络连接的一个开源框架，jsoup通过网络获取到文本，然后fastjson对文本进行解析。Jsoup里面提供了 get()方法，此方法得到的是Document对象，会使文本中的转译字符变成他应有的形式，如果转译解析后，再用fastjson进行解析就会出现问题，比如字符串中包含双引号的转译，解析后肯定会报错，这个时候应该不要对字符串进行解析。

解决方法：使用如下jsoup方法，这种方法就不会对转译字符进行处理

****

1. **Web服务器的演化**

我们在做一个访问量不大的项目的时候,一台服务器部署上一个应用+数据库也就够了.

那么访问量稍微大一点之后呢,为了解决用户反馈的卡,反应慢的情况,我们就上集群.架设nginx,部署多个服务,由nginx负责把请求转发到其他服务上,这样就解决了用户说的卡慢问题.

过了一段时间之后呢,我们发现数据库已经扛不住了,应用服务完好,数据库有时候宕机. 那这个时候呢,我们就上数据库读写分离,再架设几台数据库服务器,做主从,做分库分表. 然后数据库也不宕机了,服务又恢复了流畅.

又过了一段时间,公司事业增增日上,服务访问量越来越高,且大部分都是查询, 吸取之前宕机且为了办证数据库的健壮性,我们这个时候又加上了缓存, 把用户高频次访问的数据放到缓存里.

后来,项目功能越来越多,整个项目也愈发庞大,修改一个类就需要全盘上传,切换nginx重启,这样的发布流程越来越长,越来越繁杂.然后我们开始把模块拆分,用户信息分个项目,订单系统分一个项目.这样就达到了,用户模块代码修改的时候,只需要更新用户信息服务就好了.但是还是需要切换顶层的nginx.把要重启的服务的流量切到可用服务上. 这个时候我们就想到了RPC

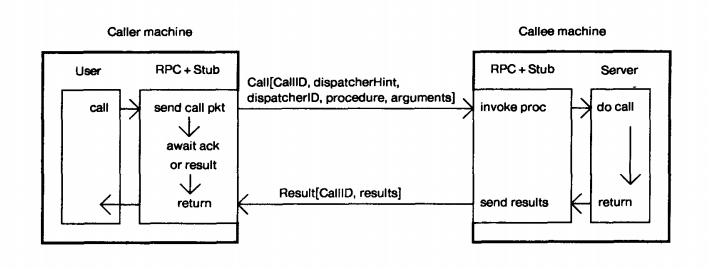
1. **RPC**

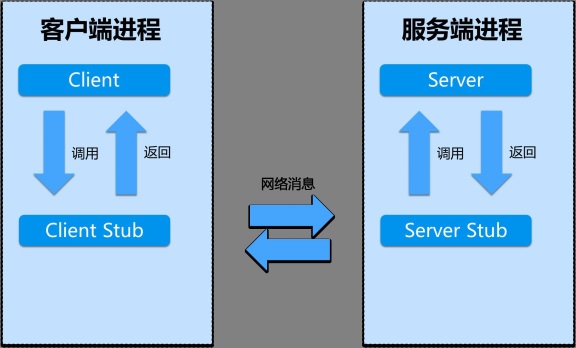
**背景：**早期单机时代，一台电脑上运行多个进程，大家各干各的，老死不相往来。假如A进程需要一个画图的功能，B进程也需要一个画图的功能，程序员就必须为两个进程都写一个画图的功能。这不是整人么？于是就出现了IPC（Inter-process communication，单机中运行的进程之间的相互通信）。OK，现在A既然有了画图的功能，B就调用A进程上的画图功能好了，程序员终于可以偷下懒了。

到了网络时代，大家的电脑都连起来了。以前程序只能调用自己电脑上的进程，能不能调用其他机器上的进程呢？于是就程序员就把IPC扩展到网络上，这就是RPC（远程过程调用）了。现在不仅单机上的进程可以相互通信，多机器中的进程也可以相互通信了。

要知道实现RPC很麻烦呀，什么多线程、什么Socket、什么I/O，都是让咱们普通程序员很头疼的事情。于是就有牛人开发出RPC框架（比如，CORBA、RMI、Web Services、RESTful Web Services等等）。

**概念：**RPC 是远程过程调用（Remote Procedure Call）的缩写形式，RPC 是指计算机 A 上的进程，调用另外一台计算机 B 上的进程，其中 A 上的调用进程被挂起，而 B 上的被调用进程开始执行，当值返回给 A 时，A 进程继续执行。调用方可以通过使用参数将信息传送给被调用方，而后可以通过传回的结果得到信息。而这一过程，对于开发人员来说是透明的。





远程过程调用采用客户机/服务器(C/S)模式。请求程序就是一个客户机，而服务提供程序就是一台服务器。和常规或本地过程调用一样，远程过程调用是同步操作，在远程过程结果返回之前，需要暂时中止请求程序。使用相同地址空间的低权进程或低权线程允许同时运行多个远程过程调用。

**协议：**rpc支持多种协议，主要还是看实现的rpc框架支持什么协议，目前主要支持tcp/ip协议

**框架：**目前流行的开源RPC框架还是比较多的。下面重点介绍三种：

gRPC是Google最近公布的开源软件，基于最新的HTTP2.0协议，并支持常见的众多编程语言。 我们知道HTTP2.0是基于二进制的HTTP协议升级版本，目前各大浏览器都在快马加鞭的加以支持。 这个RPC框架是基于HTTP协议实现的，底层使用到了Netty框架的支持。

Thrift是Facebook的一个开源项目，主要是一个跨语言的服务开发框架。它有一个代码生成器来对它所定义的IDL定义文件自动生成服务代码框架。用户只要在其之前进行二次开发就行，对于底层的RPC通讯等都是透明的。不过这个对于用户来说的话需要学习特定领域语言这个特性，还是有一定成本的。

Dubbo是阿里集团开源的一个极为出名的RPC框架，在很多互联网公司和企业应用中广泛使用。协议和序列化框架都可以插拔是及其鲜明的特色。同样 的远程接口是基于Java Interface，并且依托于spring框架方便开发。可以方便的打包成单一文件，独立进程运行，和现在的微服务概念一致。

偷偷告诉你集团内部已经不怎么使用dubbo啦，现在用的比较多的叫HSF，又名“好舒服”。后面有可能会开源，大家拭目以待。

**主要特质**：

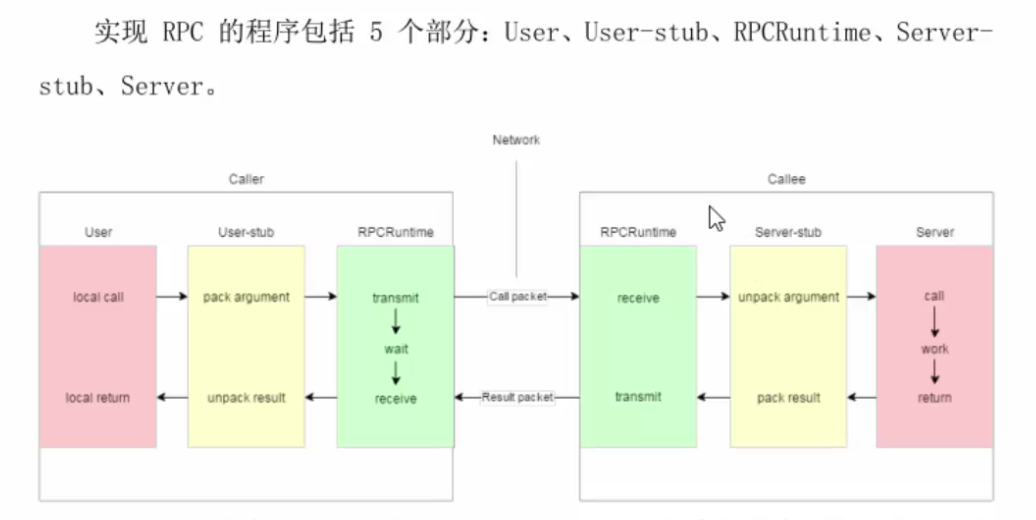
（1）rpc是协议：协议意味着规范。目前典型的rpc实现包括：Dubbo、Thrift、Hetty等。但这些实现往往都会附加其他主要功能，例如Dubbo还包括了服务管理、访问权限管理等功能。

（2）网络协议和网络IO模型对其透明：既然rpc的客户端认为自己是在调用本地对象，那么传输层使用的是TCP/UDP还是http协议，又或者是一些其他的网络协议它就不需要关心了。既然网络协议对其透明，那么调用过程中，使用的是哪一种网络IO模型调用者也不需要关心。

（3）信息格式对其透明：远程调用过程中，需要传递一些参数，并且会返回一个调用结果。至于这些参数会以某种信息格式传递给网络上的另外一台计算机，这个信息格式是怎样构成的，调用方是不需要关心的。

（4）跨语言能力：对应调用方来说，不知道也无需知道远程的程序使用的是什么语言运行的，无论服务器方使用什么语言，本次调用都应该成功，并且返回值也应该按照调用方程序语言所能理解的形式进行描述。

**实现rpc**：



user就是发起rpc调用的client，当user想发起一个远程调用时，它实际是通过本地调用user-stub。user-stub负责将调用的接口、方法和参数通过约定的协议规范进行编码并通过本地的RPCRuntime实例传输到远端的实例。远端RPCRuntime实例收到请求后交给server-stub进行解码后发起本地端调用，调用结果再返回给user端。

stub：为屏蔽客户调用远程主机上的对象，必须提供某种方式来模拟本地对象，这种本地对象称为存根（stub），存根负责接收本地方法调用，并将它们委派给各自的具体实现对象。

1. **restful风格**

REST是英文representational state transfer(表象性状态转变)或者表述性状态转移;Rest是web服务的一种架构风格;使用HTTP,URI,XML,JSON,HTML等广泛流行的标准和协议;轻量级,跨平台,跨语言的架构设计;它是一种设计风格,不是一种标准,是一种思想。

Rest架构的主要原则：

网络上的所有事物都被抽象为资源

每个资源都有一个唯一的资源标识符

同一个资源具有多种表现形式(xml,json等)

对资源的各种操作不会改变资源标识符

所有的操作都是无状态的

符合REST原则的架构方式即可称为RESTful

什么是Restful？

对应的中文是rest式的;Restful web service是一种常见的rest的应用,是遵守了rest风格的web服务;rest式的web服务是一种ROA(The Resource-Oriented Architecture)(面向资源的架构).

为什么会出现Restful？

在Restful之前的操作：

http://127.0.0.1/user/query/1 GET 根据用户id查询用户数据

http://127.0.0.1/user/save POST 新增用户

http://127.0.0.1/user/update POST 修改用户信息

http://127.0.0.1/user/delete GET/POST 删除用户信息

RESTful用法：

http://127.0.0.1/user/1 GET 根据用户id查询用户数据

http://127.0.0.1/user POST 新增用户

http://127.0.0.1/user PUT 修改用户信息

http://127.0.0.1/user DELETE 删除用户信息

之前的操作是没有问题的,大神认为是有问题的,有什么问题呢?你每次请求的接口或者地址,都在做描述,例如查询的时候用了query,新增的时候用了save,其实完全没有这个必要,我使用了get请求,就是查询.使用post请求,就是新增的请求,我的意图很明显,完全没有必要做描述,这就是为什么有了restful。

如何使用：



SpringMVC实现restful服务:

SpringMVC原生态的支持了REST风格的架构设计

所涉及到的注解:

--@RequestMapping

---@PathVariable

---@ResponseBody

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.bind.annotation.RequestParam;

import org.springframework.web.bind.annotation.ResponseBody;

import cn.itcast.mybatis.pojo.User;

import cn.itcast.mybatis.service.NewUserService;

@RequestMapping("restful/user")

@Controller

public class RestUserController {

    @Autowired

    private NewUserService newUserService;

    /\*\*

     \* 根据用户id查询用户数据

     \*

     \* @param id

     \* @return

     \*/

    @RequestMapping(value = "{id}", method = RequestMethod.GET)

    @ResponseBody

    public ResponseEntity<User> queryUserById(@PathVariable("id") Long id) {

        try {

            User user = this.newUserService.queryUserById(id);

            if (null == user) {

                // 资源不存在，响应404

                return ResponseEntity.status(HttpStatus.NOT\_FOUND).body(null);

            }

            // 200

            // return ResponseEntity.status(HttpStatus.OK).body(user);

            return ResponseEntity.ok(user);

        } catch (Exception e) {

            e.printStackTrace();

        }

        // 500

        return ResponseEntity.status(HttpStatus.INTERNAL\_SERVER\_ERROR).body(null);

    }

    /\*\*

     \* 新增用户

     \*

     \* @param user

     \* @return

     \*/

    @RequestMapping(method = RequestMethod.POST)

    public ResponseEntity<Void> saveUser(User user) {

        try {

            this.newUserService.saveUser(user);

            return ResponseEntity.status(HttpStatus.CREATED).build();

        } catch (Exception e) {

            // TODO Auto-generated catch block

            e.printStackTrace();

        }

        // 500

        return ResponseEntity.status(HttpStatus.INTERNAL\_SERVER\_ERROR).body(null);

    }

    /\*\*

     \* 更新用户资源

     \*

     \* @param user

     \* @return

     \*/

    @RequestMapping(method = RequestMethod.PUT)

    public ResponseEntity<Void> updateUser(User user) {

        try {

            this.newUserService.updateUser(user);

            return ResponseEntity.status(HttpStatus.NO\_CONTENT).build();

        } catch (Exception e) {

            e.printStackTrace();

        }

        // 500

        return ResponseEntity.status(HttpStatus.INTERNAL\_SERVER\_ERROR).body(null);

    }

    /\*\*

     \* 删除用户资源

     \*

     \* @param user

     \* @return

     \*/

    @RequestMapping(method = RequestMethod.DELETE)

    public ResponseEntity<Void> deleteUser(@RequestParam(value = "id", defaultValue = "0") Long id) {

        try {

            if (id.intValue() == 0) {

                // 请求参数有误

                return ResponseEntity.status(HttpStatus.BAD\_REQUEST).build();

            }

            this.newUserService.deleteUserById(id);

            // 204

            return ResponseEntity.status(HttpStatus.NO\_CONTENT).build();

        } catch (Exception e) {

            e.printStackTrace();

        }

        // 500

        return ResponseEntity.status(HttpStatus.INTERNAL\_SERVER\_ERROR).body(null);

    }

}

1. **cookie和Session**

1、session 在服务器端，cookie 在客户端（浏览器）

2、session 默认被存在在服务器的一个文件里（不是内存）

3、session 的运行依赖 sessionid，而 sessionid 是存在 cookie 中的，也就是说，如果浏览器禁用了 cookie ，同时 session 也会失效（但是可以通过其它方式实现，比如在 url 中传递 session\_id）

4、session 可以放在文件、数据库、或内存中都可以。

5、用户验证这种场合一般会用 session

详细描述：

1. 由于HTTP协议是无状态的协议，所以服务端需要记录用户的状态时，就需要用某种机制来识具体的用户，这个机制就是Session.典型的场景比如购物车，当你点击下单按钮时，由于HTTP协议无状态，所以并不知道是哪个用户操作的，所以服务端要为特定的用户创建了特定的Session，用于标识这个用户，并且跟踪用户，这样才知道购物车里面有几本书。这个Session是保存在服务端的，有一个唯一标识。在服务端保存Session的方法很多，内存、数据库、文件都有。集群的时候也要考虑Session的转移，在大型的网站，一般会有专门的Session服务器集群，用来保存用户会话，这个时候 Session 信息都是放在内存的，使用一些缓存服务比如Memcached之类的来放 Session。

2. 思考一下服务端如何识别特定的客户？这个时候Cookie就登场了。每次HTTP请求的时候，客户端都会发送相应的Cookie信息到服务端。实际上大多数的应用都是用 Cookie 来实现Session跟踪的，第一次创建Session的时候，服务端会在HTTP协议中告诉客户端，需要在 Cookie 里面记录一个Session ID，以后每次请求把这个会话ID发送到服务器，我就知道你是谁了。有人问，如果客户端的浏览器禁用了 Cookie 怎么办？一般这种情况下，会使用一种叫做URL重写的技术来进行会话跟踪，即每次HTTP交互，URL后面都会被附加上一个诸如 sid=xxxxx 这样的参数，服务端据此来识别用户。

3. Cookie其实还可以用在一些方便用户的场景下，设想你某次登陆过一个网站，下次登录的时候不想再次输入账号了，怎么办？这个信息可以写到Cookie里面，访问网站的时候，网站页面的脚本可以读取这个信息，就自动帮你把用户名给填了，能够方便一下用户。这也是Cookie名称的由来，给用户的一点甜头。所以，总结一下：Session是在服务端保存的一个数据结构，用来跟踪用户的状态，这个数据可以保存在集群、数据库、文件中；Cookie是客户端保存用户信息的一种机制，用来记录用户的一些信息，也是实现Session的一种方式。

**会话**：当访问web资源之后，通过连接继续请求当前站点的资源，然后直到关闭浏览器的整个过程就是会话。会话（Session）跟踪是Web程序中常用的技术，用来跟踪用户的整个会话。常用的会话跟踪技术是Cookie与Session。Cookie通过在客户端记录信息确定用户身份，Session通过在服务器端记录信息确定用户身份。

1. **Cookie**

将服务器的数据保存到浏览器的技术。

IE：将cookie保存到指定的目录下，以文件的形式存储

Firefox和Chrome：将cookie保存到指定的文件中

**Cookie机制**：

在程序中，会话跟踪是很重要的事情。理论上，一个用户的所有请求操作都应该属于同一个会话，而另一个用户的所有请求操作则应该属于另一个会话，二者不能混淆。例如，用户A在超市购买的任何商品都应该放在A的购物车内，不论是用户A什么时间购买的，这都是属于同一个会话的，不能放入用户B或用户C的购物车内，这不属于同一个会话。

而Web应用程序是使用HTTP协议传输数据的。HTTP协议是无状态的协议。一旦数据交换完毕，客户端与服务器端的连接就会关闭，再次交换数据需要建立新的连接。这就意味着服务器无法从连接上跟踪会话。即用户A购买了一件商品放入购物车内，当再次购买商品时服务器已经无法判断该购买行为是属于用户A的会话还是用户B的会话了。要跟踪该会话，必须引入一种机制。

Cookie就是这样的一种机制。它可以弥补HTTP协议无状态的不足。在Session出现之前，基本上所有的网站都采用Cookie来跟踪会话

**Cookie流程：**

Cookie实际上是一小段的文本信息。客户端请求服务器，如果服务器需要记录该用户状态，就使用response向客 户端浏览器颁发一个Cookie。客户端浏览器会把Cookie保存起来。当浏览器再请求该网站时，浏览器把请求的网址连同该Cookie一同提交给服务 器。服务器检查该Cookie，以此来辨认用户状态。服务器还可以根据需要修改Cookie的内容。

Cookie三要素：name-value(键值对),path（路径）,domain（域名）。只要其中之一变化，则每次请求的时候都会产生新的cookie。

**Cookie持久化**

服务器设置cookie，如果cookie不存在则新增cookie，如果已经存在，则覆盖。如果我们在服务器设置cookie的时候，不设置持久化的时间（setMaxAge），那么当整个会话结束后，cookie信息将被删除（cookie在内存中）。如果想要在浏览器关闭重启后，还能保留cookie，则需要进行持久化。将cookie保存到硬盘中，通过setMaxAge方法进行设置，单位为秒，如果设置为0，则进行删除操作。

**服务器写cookie：**

**注意的是写cookie的路径如果是全局变量最好写到根目录下，要不然读取cookie的时候容易出错，读不到，原因是路径不对。**

**在获取cookie的时候默认从根路径获取**

**public** **static** **void** setCookies(HttpServletResponse response, String name,

String value, **int** expiry,String path) {

Cookie cookie = **new** Cookie(name, value);

//cookie.setHttpOnly(true);

//设置cookie的存活时间

cookie.setMaxAge(expiry);

cookie.setPath(path);

response.addCookie(cookie);

}

使用的时候：

setCookies(response, AccountStr.LOGIN\_STATUS, "true", 60\*60\*30,"/");

**服务器获取cookie：**

获取cookie

/\*\*

\* 取得某个Cookie值

\*

\* **@param** request

\* **@param** key

\* **@return**

\*/

**private** String getCookie(HttpServletRequest request, String key) {

Cookie[] ca = request.getCookies();

**if** (ca == **null** || ca.length < 1)

**return** **null**;

**for** (Cookie c : ca) {

**if** (c.getName().equals(key)) {

**return** c.getValue();

}

}

**return** **null**;

}



**cookie路径问题：**

request.getCookies()是获取当前浏览器的cookie，而不能获取其他浏览器的cookie。此方法是获取当前servlet所在目录，以及之上所有目录下的cookie信息。

比如我们设置cookie的请求A是http://localhost/day02/setCookie,此请求会将cookie设置到day02目录下，然后同一个服务器，另外一个设置cookie的请求B是<http://localhost/day02/cookie/setCookie>，此请求会将cookie的值设置到/day02/cookie目录下。此时通过请求<http://localhost/day02/getCookie>，会发现只能得到的是请求A设置进的cookie值，通过请求<http://localhost/day02/cookie/getCookie>得到的是/day02/cookie下的cookie信息以及cookie目录以上的目录的cookie信息，也就是day02目录下的cookie值也能获取到，请求A和B设置的cookie信息都能得到。我们还可以在新增cookie的时候指定cookie的路径，通过setPath方法。

**Cookie处理中文**

我们不能在cookie中直接存放中文，必须经过一次转码，我们可以在设置cookie的时候使用Unicode对中文进行编码，然后在获取cookie信息的时候使用Unicode对中文进行解码。

\* 编码：String returnData = URLEncoder.encode(data, "UTF-8");

\* 解码：String value = URLDecoder.decode(c.getValue(), "UTF-8");

**cookie的value大小限制**

cookie的value值不是无限的，它允许的最大值是4kb，如果超过该值，则设置cookie不成功，不会报错，但是也没有cookie文件。

**删除cookie**

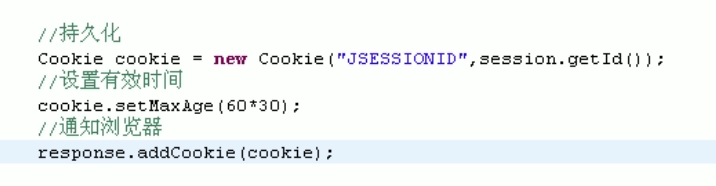
|  |
| --- |
| //删除名称为name的cookie  Cookie c = **new** Cookie("name","");  //默认值-1,在浏览器中存在。 0：删除文件中的cookie和浏览器中的cookie  c.setMaxAge(0);  c.setPath(request.getContextPath());  response.addCookie(c); |

1. **Session**

Session是另一种记录客户状态的机制，不同的是Cookie保存在客户端浏览器中，而Session保存在服务器上。客户端浏览器访问服务器的时候，服务器把客户端信息以某种形式记录在服务器上。这就是Session。客户端浏览器再次访问时只需要从该Session中查找该客户的状态就可以了。

如果说Cookie机制是通过检查客户身上的“通行证”来确定客户身份的话，那么Session机制就是通过检查服务器上的“客户明细表”来确认客户身份。Session相当于程序在服务器上建立的一份客户档案，客户来访的时候只需要查询客户档案表就可以了。

注意：session必须是建立在cookie的基础上，必须通过cookie进行关联，也就是要使用Session的前提是必须使用cookie。如果cookie被删除，再次发送请求会重新创建Session。在tomcat中我们创建session的时候tomcat会自动为我们创建一个会话cookie，因此不需要我们创建，不过该cookie会在浏览器关闭的时候失效。如果想长期访问的话，我们需要持久化cookie。如下代码就是在服务器持久化cookie



**获取Session：**

//获得session

HttpSession session = request.getSession(); //没有创建，有返回

当我们创建Session的时候，服务器会产生一个Sessionid，然后将该Sessionid的值设置进cookie，然后后面的请求的cookie里面都会带上Sessionid。Session是由Tomcat创建的，Session的有效期是30分钟，也就是不使用该Session，30分钟后会自动销毁。该配置在Tomcat的web.xml配置文件中可以修改。另外，通过调用Session的invalidate()方法可以使Session失效。

**生命周期：**

Session保存在服务器端。为了获得更高的存取速度，服务器一般把Session放在内存里。每个用户都会有一个独立的Session。如果Session内容过于复杂，当大量客户访问服务器时可能会导致内存溢出。因此，Session里的信息应该尽量精简。

Session在用户第一次访问服务器的时候自动创建。需要注意只有访问JSP、Servlet等程序时才会创建Session，只访问HTML、IMAGE等静态资源并不会创建Session。如果尚未生成Session，也可以使用request.getSession(true)强制生成Session。

Session生成后，只要用户继续访问，服务器就会更新Session的最后访问时间，并维护该Session。用户每访问服务器一次，无论是否读写Session，服务器都认为该用户的Session“活跃（active）”了一次。

**url重写：**

问题描述：当我们浏览器禁用cookie的时候，我们在同一个会话下发送第二次请求的时候，请求是获取不到cookie的，因此他还会重新向服务器发送请求cookie里面的数据(比如Sessionid)。此时我们使用URL重写技术，将cookie里面的数据加在URL后面，当做参数传递回去。

我们可以使用url重写技术

解决方法1：使用 encodeURL方法

HttpSession session = req.getSession();

System.out.println(session.getId());

String url = "urlServlet1";

//url重写

url = resp.encodeURL(url);

resp.getWriter().print("<a href='"+url+"'>url</a>");

源码：

public String encodeRedirectURL(String url) {

if (isEncodeable(toAbsolute(url))) {

return (toEncoded(url, request.getSessionInternal().getIdInternal()));

} else {

return (url);

}

}

注意：这里的禁用cookie在IE8及以下是生效的，不过在IE9及以后已经失效（意思就是即便禁用了cookie，还是能获取到session，而不会产生新的session）,IE9以后对localhost这种ip段不能禁用cookie，127.0.0.1就行。

方法2：使用encodeRedirectURL()方法

源码：

public String encodeRedirectURL(String url) {

if (isEncodeable(toAbsolute(url))) {

return (toEncoded(url, request.getSessionInternal().getIdInternal()));

} else {

return (url);

}

}

两个方法区别在于，如果参数为空字符串，返回结果不同。一般我们都使用encodeURL方法

1. **http协议**

**http：**

客户端连上web服务器后，若想获得web服务器中的某个web资源，需遵守一定的通讯格式，http协议用于定义客户端与web服务器通讯的格式。

http协议:hypertext Transfer protocol超文本传输协议，它是TCP/IP协议的一个应用层协议，用于定义WEB浏览器与WEB服务器之间交换数据的过程。

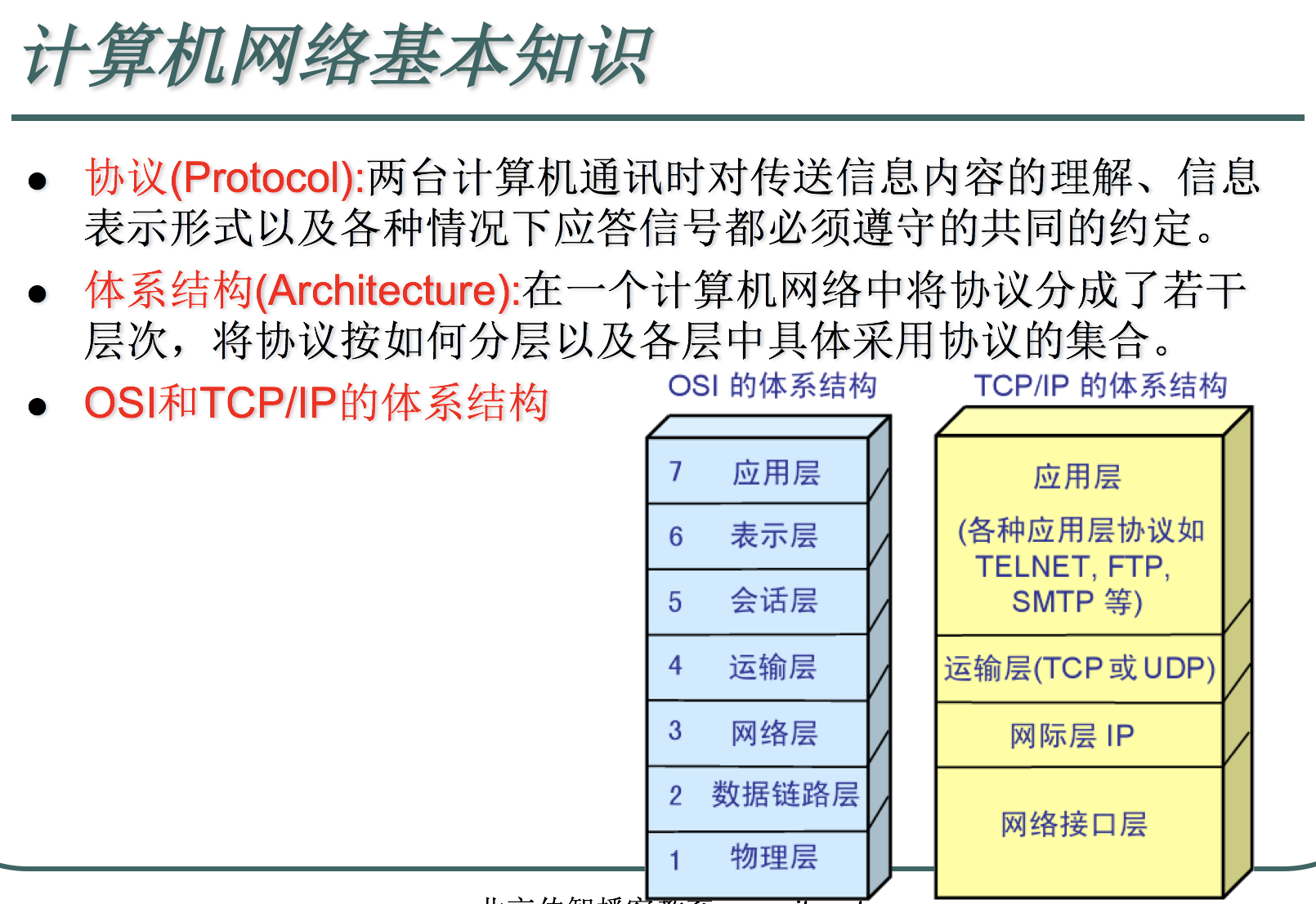
（1）它是基于tcp连接的传输协议

（2）默认端口号80

（3）基于请求-响应模式的协议

（4）无状态，是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快。

（5）支持B/S模式，也就是浏览器/服务器模式



**http协议版本：**

在HTTP1.0协议中，客户端与web服务器建立连接后，只能获得一个web资源，也就是只能发起一次HTTP请求，连接就会关闭。

在HTTP1.1协议中，允许客户端与web服务器建立连接后，在一个连接上获取多个web资源。也就是可以发起多次HTTP请求，用的都是同一个连接。

**http请求：**

分类：请求行，请求头，请求体

**请求行：**描述请求的资源

请求格式： 请求方法 资源路径 协议



请求方法：

get:请求内容的参数追加到URL之后，比如<http://www.baidu.com/login?user=1122&name=gh>

get请求的URL长度有限制（1K）

post:请求内容放置在请求体中，他对大小没有任何限制。因此参数比较多的时候建议使用post请求。

**请求头：**

Accept：浏览器通知服务器，浏览器可以接收的数据类型。

Accept-charset:浏览器通知服务器，浏览器支持的字符集。

Accept-Encoding：浏览器通知服务器，浏览器支持的压缩格式

Accept-language：浏览器通知服务器，浏览器支持的语言种类

Host:浏览器通知服务器，浏览器访问服务器的主机名

If-Modified-since:浏览器询问服务器，浏览器本地是否修改过（因为这样如果浏览器没有修改，则使用缓存，有就从服务器重新获取）

Referer：浏览器通知服务器，浏览器当前页来自哪个页面（防止盗链）

user-agent：浏览器通知服务器，浏览器端的基本信息（浏览器内核，系统版本等）

connection：浏览器保持服务器，是否保持连接状态

cookie：

Date：浏览器通知服务器，浏览器端的时间

**请求体：**

请求体与请求头之间存在一个空行

**http响应：**

分类：响应行 响应头 响应体

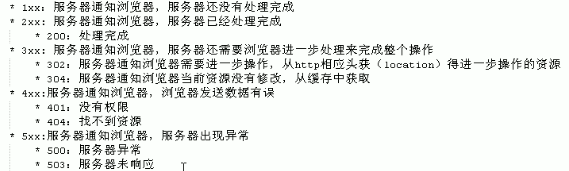
**响应行：**

协议 状态码 状态信息

http1.1 200 ok

**状态码：**

服务器通知浏览器的行为，执行过程



**响应头：**

location：设置重定向跳转

content-encoding:值可以为gzip，服务器告诉浏览器数据的压缩格式

content-Language：zh-cn，设置语言，国际化的时候会用到

content-type:text/html;charset=utf-8，数据编码

refresh:2（单位是秒），刷新的时间,设置后，浏览器会自动在2秒间隔内再重新发起请求获取内容。还可以作为倒计时跳转，比如支付成功后倒计时3秒再跳转到某个页面。

Content-Disposition:attachment;filename=2.jpg。提示浏览器下载文件，而不是解析数据。其中2.jpg是随便取的。

**响应体：**

响应头和响应体之间存在一个空行

响应体，也就是服务器响应浏览器所有的内容

**tcp协议：**

TCP/IP不是一个协议，而是一个协议族的统称。里面包括了IP协议，IMCP协议，TCP协议，以及我们更加熟悉的http、ftp、pop3协议等等。

**背景：**

TCP协议对应于传输层，而HTTP协议对应于应用层，从本质上来说，二者没有可比性。Http协议是建立在TCP协议基础之上的，当浏览器需要从服务器获取网页数据的时候，会发出一次Http请求。Http会通过TCP建立起一个到服务器的连接通道，当本次请求需要的数据完毕后，Http会立即将TCP连接断开，这个过程是很短的。所以Http连接是一种短连接，是一种无状态的连接。所谓的无状态，是指浏览器每次向服务器发起请求的时候，不是通过一个连接，而是每次都建立一个新的连接。如果是一个连接的话，服务器进程中就能保持住这个连接并且在内存中记住一些信息状态。而每次请求结束后，连接就关闭，相关的内容就释放了，所以记不住任何状态，成为无状态连接。

随着时间的推移，html页面变得复杂了，里面可能嵌入了很多图片，这时候每次访问图片都需要建立一次tcp连接就显得低效了。因此Keep-Alive被提出用来解决效率低的问题。从HTTP/1.1起，默认都开启了Keep-Alive，保持连接特性，简单地说，当一个网页打开完成后，客户端和服务器之间用于传输HTTP数据的TCP连接不会关闭，如果客户端再次访问这个服务器上的网页，会继续使用这一条已经建立的连接Keep-Alive不会永久保持连接，它有一个保持时间，可以在不同的服务器软件（如Apache）中设定这个时间。虽然这里使用TCP连接保持了一段时间，但是这个时间是有限范围的，到了时间点依然是会关闭的，所以我们还把其看做是每次连接完成后就会关闭。后来，通过Session, Cookie等相关技术，也能保持一些用户的状态。但是还是每次都使用一个连接，依然是无状态连接。

以前有个概念很容易搞不清楚。就是为什么Http是无状态的短连接，而TCP是有状态的长连接？Http不是建立在TCP的基础上吗，为什么还能是短连接？现在明白了，Http就是在每次请求完成后就把TCP连接关了，所以是短连接。而我们直接通过Socket编程使用TCP协议的时候，因为我们自己可以通过代码区控制什么时候打开连接什么时候关闭连接，只要我们不通过代码把连接关闭，这个连接就会在客户端和服务端的进程中一直存在，相关状态数据会一直保存着。

**概念：**

TCP 用于应用程序之间的通信。当应用程序希望通过 TCP 与另一个应用程序通信时，它会发送一个通信请求。这个请求必须被送到一个确切的地址。在双方“握手”之后，TCP 将在两个应用程序之间建立一个全双工 (full-duplex) 的通信。

这个全双工的通信将占用两个计算机之间的通信线路，直到它被一方或双方关闭为止。

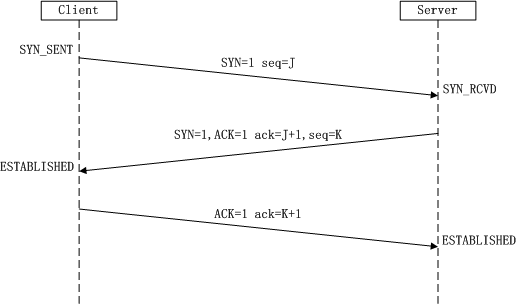
UDP 和 TCP 很相似，但是更简单，同时可靠性低于 TCP。

一个TCP连接由一个4元组构成，它们分别是发送端ip地址，发送端tcp端口，接收端ip地址，接收端tcp端口。也就是说，tcp连接由一对套接字构成。

TCP连接通常分为3个阶段：启动、数据传输（连接已建立）、退出（连接关闭）。

**三次握手：**

建立连接三次握手过程如下图：

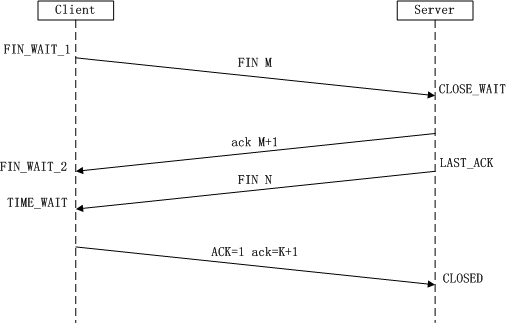


**SYN攻击：**

在三次握手过程中，Server发送SYN-ACK之后，收到Client的ACK之前的TCP连接称为半连接（half-open connect），此时Server处于SYN\_RCVD状态，当收到ACK后，Server转入ESTABLISHED状态。SYN攻击就是Client在短时间内伪造大量不存在的IP地址，并向Server不断地发送SYN包，Server回复确认包，并等待Client的确认，由于源地址是不存在的，因此，Server需要不断重发直至超时，这些伪造的SYN包将产时间占用未连接队列，导致正常的SYN请求因为队列满而被丢弃，从而引起网络堵塞甚至系统瘫痪。SYN攻击时一种典型的DDOS攻击，检测SYN攻击的方式非常简单，即当Server上有大量半连接状态且源IP地址是随机的，则可以断定遭到SYN攻击了，使用如下命令可以让之现行：netstat -nap | grep SYN\_RECV

**TCP四次挥手：**

所谓四次挥手（Four-Way Wavehand）即终止TCP连接，就是指断开一个TCP连接时，需要客户端和服务端总共发送4个包以确认连接的断开。在socket编程中，这一过程由客户端或服务端任一方执行close来触发，整个流程如下图所示：



由于TCP连接时全双工的，因此，每个方向都必须要单独进行关闭，这一原则是当一方完成数据发送任务后，发送一个FIN来终止这一方向的连接，收到一个FIN只是意味着这一方向上没有数据流动了，即不会再收到数据了，但是在这个TCP连接上仍然能够发送数据，直到这一方向也发送了FIN。首先进行关闭的一方将执行主动关闭，而另一方则执行被动关闭，上图描述的即是如此。

（1）第一次挥手：Client发送一个FIN，用来关闭Client到Server的数据传送，Client进入FIN\_WAIT\_1状态。

（2）第二次挥手：Server收到FIN后，发送一个ACK给Client，确认序号为收到序号+1（与SYN相同，一个FIN占用一个序号），Server进入CLOSE\_WAIT状态。

（3）第三次挥手：Server发送一个FIN，用来关闭Server到Client的数据传送，Server进入LAST\_ACK状态。

（4）第四次挥手：Client收到FIN后，Client进入TIME\_WAIT状态，接着发送一个ACK给Server，确认序号为收到序号+1，Server进入CLOSED状态，完成四次挥手。

**为什么建立连接是三次握手，而关闭连接却是四次挥手呢？**

这是因为服务端在LISTEN状态下，收到建立连接请求的SYN报文后，把ACK和SYN放在一个报文里发送给客户端。而关闭连接时，当收到对方的FIN报文时，仅仅表示对方不再发送数据了但是还能接收数据，己方也未必全部数据都发送给对方了，所以己方可以立即close，也可以发送一些数据给对方后，再发送FIN报文给对方来表示同意现在关闭连接，因此，己方ACK和FIN一般都会分开发送。

**TCP的优点：** 可靠，稳定 TCP的可靠体现在TCP在传递数据之前，会有三次握手来建立连接，而且在数据传递时，有确认、窗口、重传、拥塞控制机制，在数据传完后，还会断开连接用来节约系统资源。

**TCP的缺点：** 慢，效率低，占用系统资源高，易被攻击 TCP在传递数据之前，要先建连接，这会消耗时间，而且在数据传递时，确认机制、重传机制、拥塞控制机制等都会消耗大量的时间，而且要在每台设备上维护所有的传输连接，事实上，每个连接都会占用系统的CPU、内存等硬件资源。 而且，因为TCP有确认机制、三次握手机制，这些也导致TCP容易被人利用，实现DOS、DDOS、CC等攻击。

**套接字：**

源IP地址和目的IP地址以及源端口号和目的端口号的组合称为套接字。其用于标识客户端请求的服务器和服务。Socket就叫套接字

**套接字工作原理**

要通过互联网进行通信，你至少需要一对套接字，其中一个运行于客户机端，我们称之为

ClientSocket，另一个运行于服务器端，我们称之为ServerSocket。

根据连接启动的方式以及本地套接字要连接的目标，套接字之间的连接过程可以分为三个

步骤：服务器监听，客户端请求，连接确认。

所谓服务器监听，是服务器端套接字并不定位具体的客户端套接字，而是处于等待连接的状态，实时监控网络状态。

所谓客户端请求，是指由客户端的套接字提出连接请求，要连接的目标是服务器端的套接字。为此，客户端的套接字必须首先描述它要连接的服务器的套接字，指出服务器端套接字的地址和端口号，然后就向服务器端套接字提出连接请求。

所谓连接确认，是指当服务器端套接字监听到或者说接收到客户端套接字的连接请求，它就响应客户端套接字的请求，建立一个新的线程，把服务器端套接字的描述发给客户端，一旦

客户端确认了此描述，连接就建立好了。而服务器端套接字继续处于监听状态，继续接收其他

客户端套接字的连接请求。

**UDP协议：**

UDP(User Datagram Protocol)传输与IP传输非常类似。你可以将UDP协议看作IP协议暴露在传输层的一个接口。UDP协议同样以数据包(datagram)的方式传输，它的传输方式也是"Best Effort"的，所以UDP协议也是不可靠的(unreliable)。那么，我们为什么不直接使用IP协议而要额外增加一个UDP协议呢？ 一个重要的原因是IP协议中并没有端口(port)的概念。IP协议进行的是IP地址到IP地址的传输，这意味者两台计算机之间的对话。但每台计算机中需要有多个通信通道，并将多个通信通道分配给不同的进程使用(关于进程，可以参考Linux进程基础)。一个端口就代表了这样的一个通信通道。正如我们在邮局和邮差中提到的收信人的概念一样。UDP协议实现了端口，从而让数据包可以在送到IP地址的基础上，进一步可以送到某个端口。

UDP是面向非连接的协议，它不与对方建立连接，而是直接就把数据包发送过去。

**UDP的优点：** 快，比TCP稍安全 UDP没有TCP的握手、确认、窗口、重传、拥塞控制等机制，UDP是一个无状态的传输协议，所以它在传递数据时非常快。没有TCP的这些机制，UDP较TCP被攻击者利用的漏洞就要少一些。但UDP也是无法避免攻击的，比如：UDP Flood攻击……

**UDP的缺点：** 不可靠，不稳定 因为UDP没有TCP那些可靠的机制，在数据传递时，如果网络质量不好，就会很容易丢包。

什么时候应该使用TCP： 当对网络通讯质量有要求的时候，比如：整个数据要准确无误的传递给对方，这往往用于一些要求可靠的应用，比如HTTP、HTTPS、FTP等传输文件的协议，POP、SMTP等邮件传输的协议。 在日常生活中，常见使用TCP协议的应用如下： 浏览器，用的HTTP FlashFXP，用的FTP Outlook，用的POP、SMTP Putty，用的Telnet、SSH QQ文件传输 ………… 什么时候应该使用UDP： 当对网络通讯质量要求不高的时候，要求网络通讯速度能尽量的快，这时就可以使用UDP。 比如，日常生活中，常见使用UDP协议的应用如下： QQ语音 QQ视频 TFTP 。

**单工、全双工和半双工通信：**

单工就是指A只能发信号，而B只能接收信号，通信是单向的，就象灯塔之于航船——灯塔发出光信号而航船只能接收信号以确保自己行驶在正确的航线上。

半双工指一个时间段内只有一个动作发生，举个简单例子，一条窄窄的马路，同时只能有一辆车通过，当目前有两量车对开，这种情况下就只能一辆先过，等到头儿后另一辆再开，这个例子就形象的说明了半双工的原理。早期的对讲机、以及早期集线器等设备都是实行半双工的产品。随着技术的不断进步，半双工会逐渐退出历史舞台若使用**同一根传输线**既作接收又作发送,虽然数据可以在两个方向上传送,但通信双方不能同时收发数据,这样的传送方式就是半双工制.采用半双工方式时,通信系统每一端的发送器和接收器,通过收 / 发开关转接到通信线上,进行方向的切换,因此,会产生时间延迟.收 / 发开关实际上是由软件控制的电子开关。

全双工Full－Duplex Transmissions指交换机在发送数据的同时也能够接收数据，两者同步进行，这好像我们平时打电话一样，说话的同时也能够听到对方的声音。目前的交换机都支持全双工。当数据的发送和接收分流,分别由两根不同的传输线传送时,通信双方都能在同一时刻进行发送和接收操作,这样的传送方式就是全双工制.在全双工方式下,通信系统的每一端都设置了发送器和接收器,因此,能控制数据同时在两个方向上传送.全双工方式无需进行方向的切换,因此,没有切换操作所产生的时间延迟,这对那些不能有时间延误的交互式应用(例如远程监测和控制系统)十分有利.这种方式要求通讯双方均有发送器和接收器,同时,需要 2 根数据线传送数据信号.(可能还需要控制线和状态线,以及地线).

**URL：**

比如<http://localhost:8080/login>就叫URL。浏览器会根据此地址去具体服务器获取相关的资源（资源可以是文件，网页，图片等）。浏览器首先拿到URL，然后dns服务将域名解析为具体IP，然后根据IP访问到具体服务器，然后根据端口号，访问到具体的服务，由此服务找到具体的资源。最后服务器将资源返回到浏览器，浏览器将资源进行解析。

协议：http是协议中的一种

域名：也就是与IP进行一一对应的，方便用户记忆的。IP地址一般用户都记不住，因此相当于是给IP地址取了一个名称。

本地域名：localhost

IP地址：描述每一台计算机，也就是每台计算机的唯一编号。IP地址不好记，所以才有了域名。本地IP：127.0.0.1

上网我们需要使用外网IP，也就是运营商动态分配的外网IP，这个IP可能会动态变动。

局域网IP：也就是内网的IP

DNS：域名解析服务，也就是将域名解析为IP地址

8080：也就是端口号，每个服务都有自己的端口号（类似于现实中的房间号），80端口是http协议的默认端口，因此在输入URL时可以不写80端口也能访问。

8080是Tomcat的默认端口

3306：MySQL的默认端口

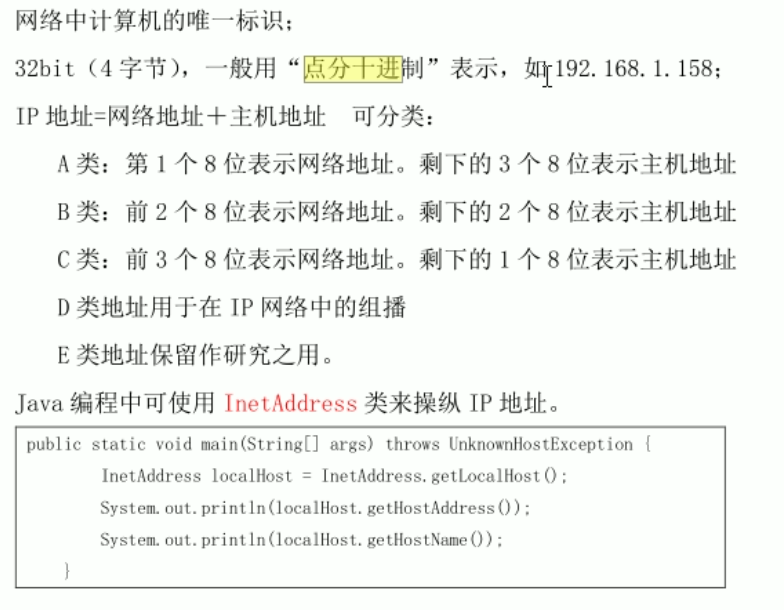
6379：Redis的默认端口

/login：资源路径

1. **网络通信**

**ip:**

也就是在网络通信，对每个设备的唯一设备编号，相当于地址。可以通过ip找到具体设备

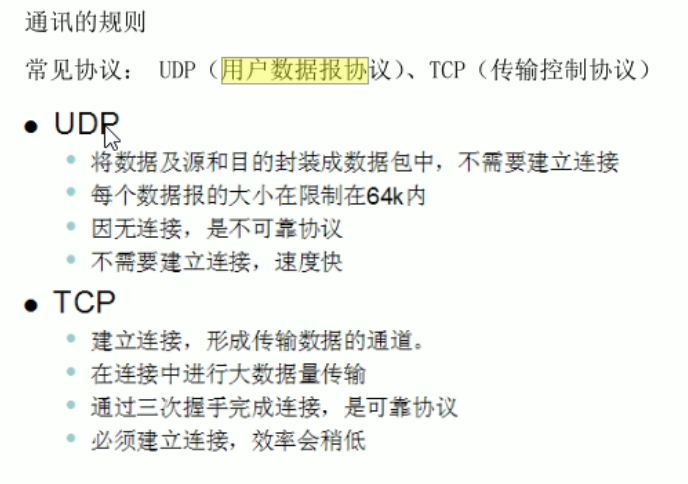


**端口号：**

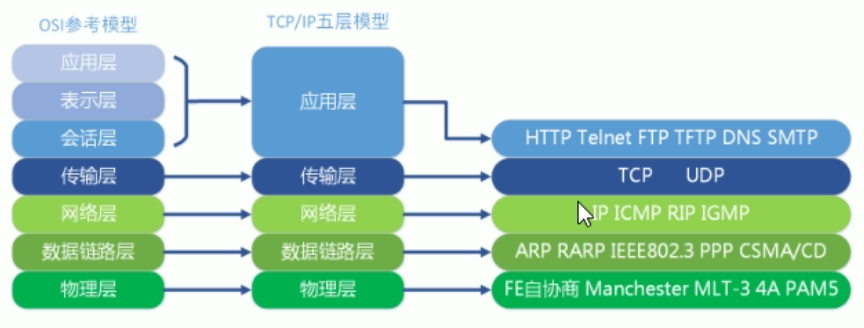
用于标识进程的逻辑地址，不同进程的标识不一样。

有效端口范围：0-65535，其中0-1024系统使用或保留端口

**协议：**



**OSI模型：**



OSI模型每层含义：

（1）物理层：主要定义物理设备标准，如网线的接口类型、光纤的接口类型、各种传输介质的传输速率等。

主要作用是将数据最终编码为0、1标识的比特流，通过物理介质传输

这一层的数据叫做比特。

（2）数据链路层：主要将接收到的数据进行MAC地址的封装与解封装。常把这一层的数据叫做帧，这一层常工作的设备是交换机。

（3）网络层：主要将接收到的数据进行IP地址的封装与解封装。常把这一层的数据叫数据包，这一层设备是路由器

（4）传输层：定义了一些数据传输的协议和端口号，主要将接收的数据进行分段和传输，到达目的地址后再进行重组。常把这一层的数据叫段

（5）会话层：通过传输层建立数据传输的通路，主要在系统之间发起会话或者接收会话请求。

（6）表示层：主要进行对接收数据的解释、加密与解密、压缩与解压缩。确保一个系统的应用层发送的数据能被另外一个系统的应用层识别

（7）应用层：主要是为一些终端应用程序提供服务，直接面对着用户。

1. **socket机制**

概念：socket，又称为套接字，用于描述IP地址和端口。应用程序通常通过socket向网络发出请求或者应答网络请求。Socket就是为网络编程提供的一种机制。

特点：

（1）通信两端都必须有socket

（2）网络通信其实就是socket之间的通信

（3）数据在两个socket之间通过IO传输

（4）网络编程也称为socket编程，套接字编程

（5）socket通信是client/server模型

**基于UDP的socket通信：**

示例：



接收端：也就是服务端

public static void main(String[] args) {

try {

//创建服务端socket对象

DatagramSocket server = new DatagramSocket(12321);

//接收数据，创建接收数据的数据包

byte[] bys = new byte[1024];

DatagramPacket packet = new DatagramPacket(bys, bys.length);

server.receive(packet);

//针对数据包进行解析

String host = packet.getAddress().getHostAddress();

byte[] data = packet.getData();

System.out.println("接收来自"+host+"信息，"+new String(data));

//释放资源

server.close();

} catch (Exception e) {

e.printStackTrace();

}

}

**基于tcp的socket通信：**

客户端代码：

private void tcpSocket() throws Exception {

/\*

\* 1.创建客户端socket服务

\* 2.指定数据的IP和端口

\* 3.通过socket建立的通道输出数据

\* 4.释放资源

\*/

Socket client = new Socket("127.0.0.1", 12321);

//取得输出流

OutputStream output = client.getOutputStream();

output.write("hello ,this is tcp".getBytes());

client.close();

}

服务端：

private void tcpSocketServer() throws Exception {

/\*

\* 1.创建服务端socket服务

\* 2.监听连接

\* 3.获取输入流，读取数据

\* 4.释放资源

\*/

ServerSocket server = new ServerSocket(12321);

//监听连接，没有请求则阻塞，有请求则获取请求，并处理

Socket s = server.accept();

//取得输出流

InputStream input = s.getInputStream();

byte[] bys = new byte[1024];

int len = input.read(bys);

System.out.println(new String(bys,0,len));

s.close();

//如果关闭server，则server服务端就不再运行，不能继续监听，一般情况是不关闭

server.close();

}

1. **IO通信模型**

网络通信的本质是网络间的数据IO，只要有IO，就会有阻塞或非阻塞的问题，无论这个IO是网络的，还是硬盘的。原因在于程序是运行在操作系统之上的，任何形式的IO操作发起都需要操作系统的支持。

**BIO阻塞模式：**

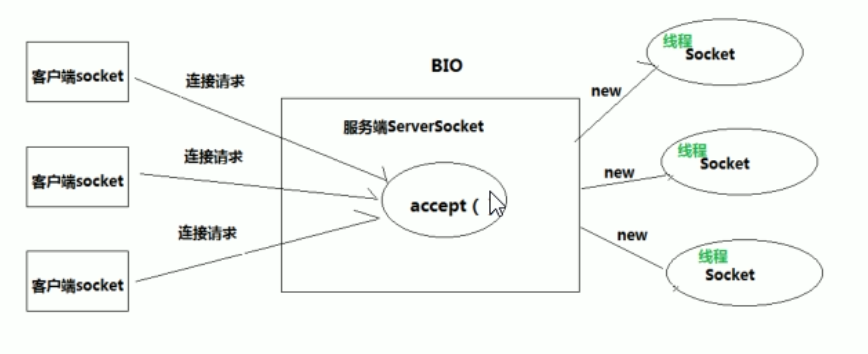
BIO即blocking IO，是一种阻塞式的IO。

Jdk1.4版本之前socket即BIO模式

BIO的问题在于accept()、read()的操作点都是阻塞的。

服务器线程发起一个accept动作，询问操作系统是否有新的socket信息从端口X发送过来。注意是询问操作系统。如果操作系统没有发现有socket从指定端口X来，那么操作系统就会等待。这样serverSocket.accept()方法就会一直等待。这就是为什么accept()方法为什么会阻塞。

如果想让bio同时处理多个客户端请求，就必须使用多线程，即每次accept阻塞等待来自客户端请求，一旦收到连接请求就建立通信，同时开启一个新的线程来处理这个套接字的数据读写请求，然后立刻又继续accept等待其他客户端连接请求，即为每一个客户端连接请求都创建一个线程来单独处理。



**NIO非阻塞模式：**

NIO即non-blocking IO,是一种非阻塞式的IO，jdk1.4之后提供。

NIO在发起读请求以后，不会立即拿到结果

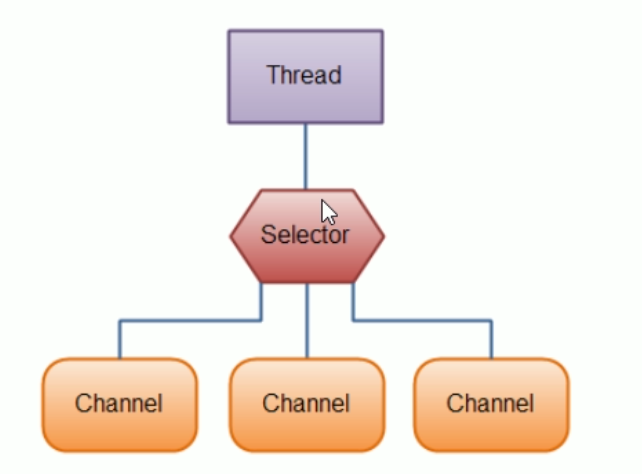
它的发起请求这一步，不会阻塞请求者的线程。会通过不断的轮询，来实现获取已经准备好内核I/O操作的流。

NIO三大核心部分：channel通道、buffer缓冲区、selector选择器

Buffer：容器对象，包含一些写入或者读出的数据。在NIO库，所有数据都是用缓冲区处理的。在读取数据时，它是直接读到缓冲区中的，在写入数据时，也是写入到缓冲区中。任何时候访问NIO中的数据，都是通过缓冲区进行操作。

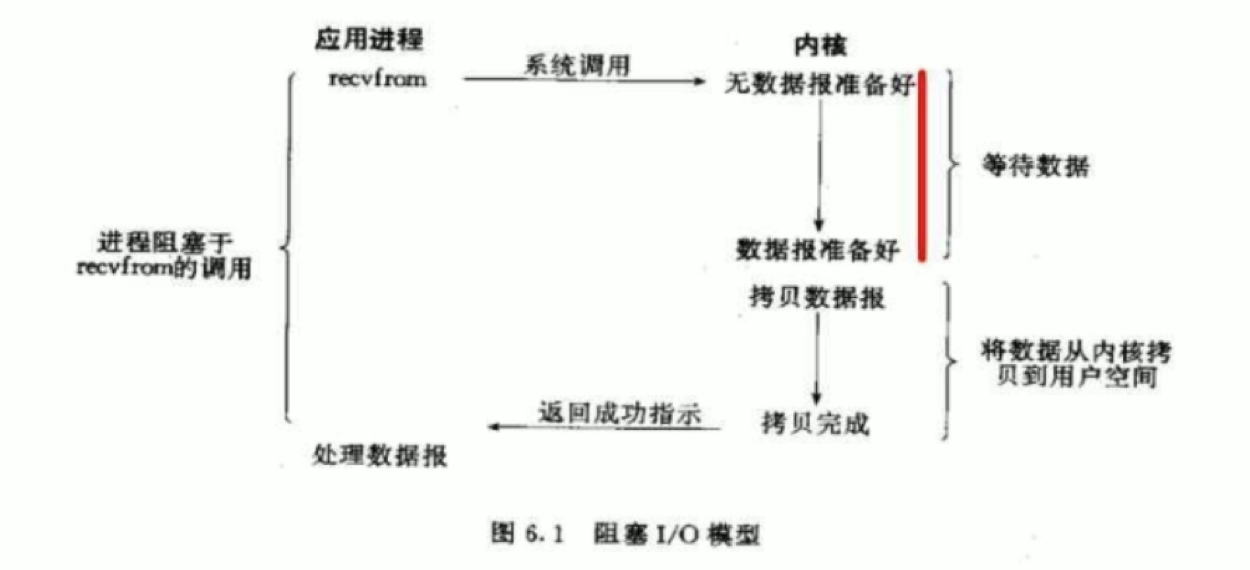
Channel：通道对象，对数据的读取和写入要通过channel，它就像水管一样。通道不同于流的地方就是通道是双向的，可以用于读、写和同时读写操作。Channel不会直接处理字节数据，而是通过buffer对象来处理数据。

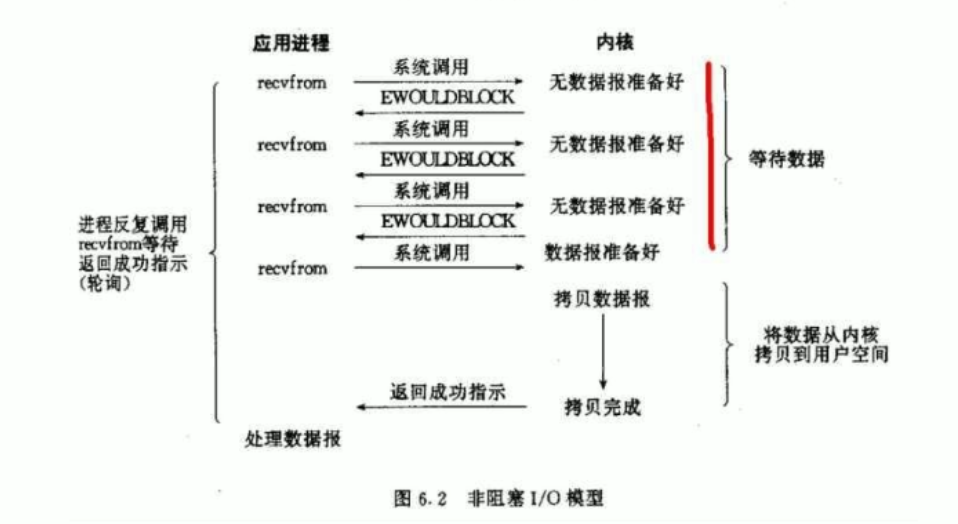
Selector：多路复用器，选择器。提供选择已经就绪的任务的能力。Selector会不断轮询注册在其上的channel，如果某个channel上面发生读或者写事件，这个channel就处于就绪状态，会被selector轮询出来，进行后续的I/O操作。这样服务器只需要一两个线程就可以进行多客户端通信。



**阻塞/非阻塞、同步/非同步：**

**阻塞IO和非阻塞IO**这两个概念是程序级别的。主要描述的是程序请求操作系统IO操作后，如果IO资源没有准备好，那么程序该如何处理问题：前者等待，后者继续执行（并且使用线程一直轮询，直到有IO资源准备好了。）





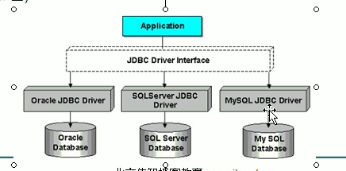
**同步IO和非同步IO**，这两个概念是操作系统级别的。主要描述的是操作系统在收到程序请求IO操作后，如果IO资源没有准备好，该如何响应程序的问题：前者不响应，直到IO 资源准备好以后；后者返回一个标记（好让程序和自己知道以后的数据往哪里通知），当IO资源准备好以后，再用事件机制返回给程序。

1. **标签**

1. **JDBC**

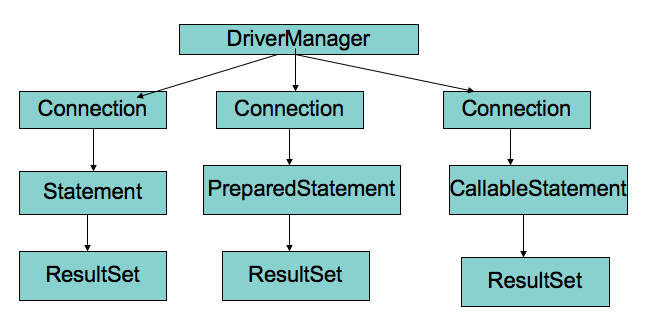
JDBC基础：JDBC（Java database connectivity）是一个独立于特定数据库管理系统，通用的操作数据库的interface（一组API），定义了用来访问数据库的标准Java类库，而接口的实现有各个数据库厂商来完成。

JDBC驱动程序：数据库厂商对JDBC接口的一组实现类（jar包）



1. **连接数据库**

JDBC接口：



连接操作数据库的方法如下：

方法1：使用driver接口的mysql实现类连接mysql数据库

Java.sql.Driver 接口是所有 JDBC 驱动程序需要实现的接口。这个接口是提供给数据库厂商使用的，不同数据库厂商提供不用的实现



方法2:驱动管理器注册

DriverManager.registerDriver(driver) 向 DriverManager 注册给定驱动程序。

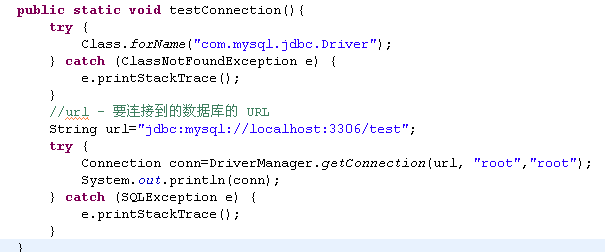


方法3：加载驱动类字节码，该方法才是我们常用的(因为这种方式驱动类不是写死的，我们可以把驱动类写成配置方式，到时我们切换数据库的时候，只需修改配置就行，而不需要修改代码)

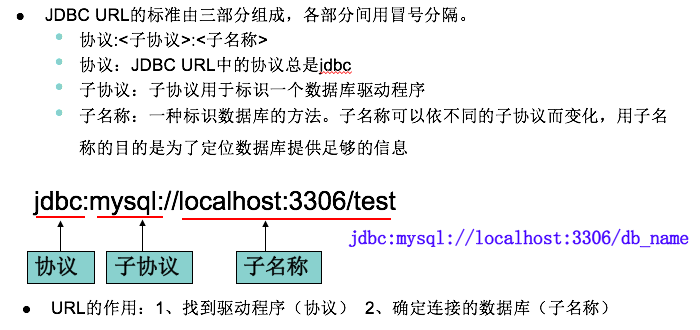
加载 JDBC 驱动需调用 Class 类的静态方法 forName()，向其传递要加载的 JDBC 驱动的类名

DriverManager 类是驱动程序管理器类，负责管理驱动程序

通常不用显式调用 DriverManager 类的 registerDriver() 方法来注册驱动程序类的实例，因为 Driver 接口的实现类都包含了静态代码块，在这个静态代码块中，会调用 DriverManager.registerDriver() 方法来注册自身的一个实例。



连接数据库的URL



1. **操作数据库**

数据库连接被用于向数据库服务器发送命令和 SQL语句，在连接建立后，需要对数据库进行访问，执行sql语句。

在 java.sql 包中有 3 个接口分别定义了对数据库的调用的不同方式：

Statement

PrepatedStatement

CallableStatement

操作数据库的步骤：

1）、 注册驱动class.forName()

2）、得到连接对象connection

3）、得到能执行sql语句的statement对象

4）、执行sql语句

5）、处理返回结果resultSet

6）、关闭资源，先关闭statement再关闭connection

**Statement方式**

通过调用 Connection 对象的 createStatement 方法创建该对象，该对象用于执行静态的 SQL 语句，并且返回执行结果。

Statement 接口中定义了下列方法用于执行 SQL 语句：

* int excuteUpdate(String sql) (针对DDL和DML操作)
* ResultSet excuteQuery(String sql)　（针对DQL操作）

**JDBCUtil工具类**

|  |
| --- |
| package com.gh.util;  import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.ResultSet;  import java.sql.SQLException;  import java.sql.Statement;  public final class JdbcUtil {  private static final String URL ="jdbc:mysql://localhost:3306/dbjdbc";  private static final String USER ="root";  private static final String PWD ="123456";  static{  try {  Class.forName("com.mysql.jdbc.Driver");  } catch (Exception e) {  e.printStackTrace();  }  }  //得到连接  public static Connection getConnection(){  Connection conn = null;  try {  conn = DriverManager.getConnection(URL,USER,PWD);  } catch (Exception e) {  e.printStackTrace();  }  return conn;  }  //关闭指定资源  public static void closeResoure(Connection conn,Statement stat,ResultSet resultSet){  if(resultSet!=null){  try {  resultSet.close();  } catch (SQLException e) {  e.printStackTrace();  }  }  if(stat!=null){  try {  stat.close();  } catch (SQLException e) {  e.printStackTrace();  }  }  if(conn!=null){  try {  conn.close();  } catch (SQLException e) {  e.printStackTrace();  }  }  }  } |

1. **ResultSet**

**简单demo**

通过调用 Statement 对象的 excuteQuery() 方法创建该对象

ResultSet 对象以逻辑表格的形式封装了执行数据库操作的结果集，ResultSet 接口由数据库厂商实现

ResultSet 对象维护了一个指向当前数据行的游标，初始的时候，游标在第一行之前，可以通过 ResultSet 对象的 next() 方法移动到下一行

ResultSet得到的数据是n+2行（第一行和最后一行是默认的，无效数据，用于方便指针移动）

ResultSet 接口的常用方法：

boolean next()

getString()



**可滚动的Result**

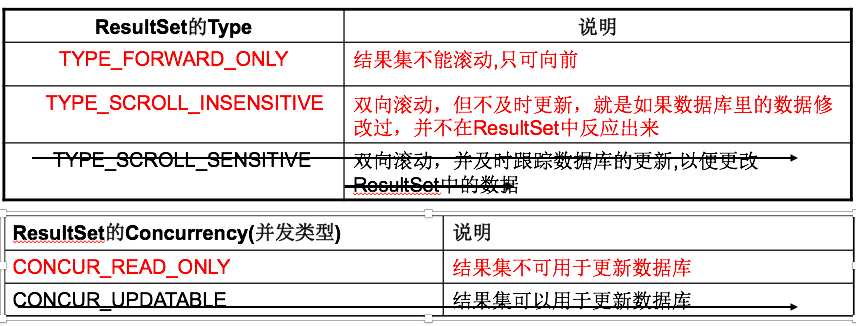
要使用可滚动的结果集，必须使用不同于前面所定义的Statement：

//创建一个 Statement 对象，该对象将生成具有给定类型和并发性的 ResultSet 对象。

Statement stmt = conn.createStatement(type,concurrency);

PreparedStatement stmt =

conn.prepareStatement(sql,type,concurrency)



此处的Type对于MySQL来说，不管我们是否设置了可滚动，MySQL都是可以双向滚动的，也就是此设置对MySQL无效，但是对Oracle有效



1. **PreparedStatement**

可以通过调用 Connection 对象的 preparedStatement() 方法获取 PreparedStatement 对象

PreparedStatement 接口是 Statement 的子接口，它表示一条预编译过的 SQL 语句

PreparedStatement 对象所代表的 SQL 语句中的参数用问号(?)来表示，调用 PreparedStatement 对象的 setXXX() 方法来设置这些参数. setXXX() 方法有两个参数，第一个参数是要设置的 SQL 语句中的参数的索引(从 1 开始)，第二个是设置的 SQL 语句中的参数的值。

**Sql注入**

SQL注入是利用某些系统没有对用户输入的数据进行充分的检查，而在用户输入数据中注入非法的 SQL 语句段或命令，从而利用系统的 SQL引擎完成恶意行为的做法

用 PreparedStatement 取代 Statement 就可以解决。

**PreparedStatement和Statement对比**

代码的可读性和可维护性.

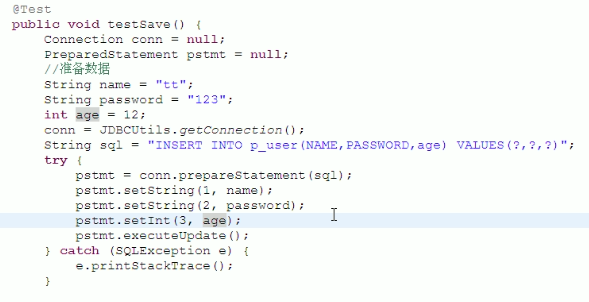
PreparedStatement能保证安全性（解决sql注入问题）

PreparedStatement 能最大可能提高性能：

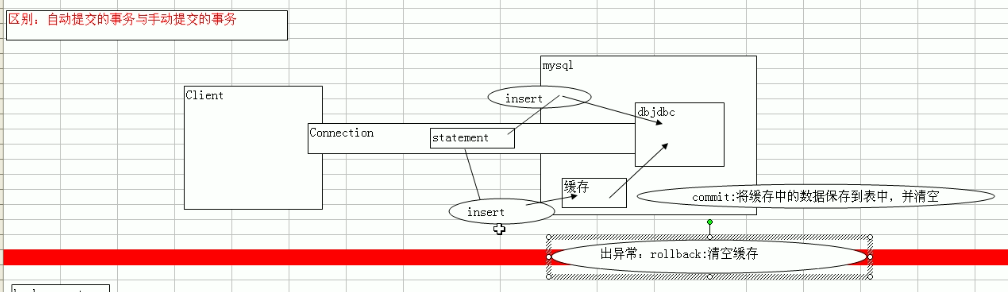
DBServer会对预编译语句提供性能优化。因为预编译语句有可能被重复调用，所以语句在被DBServer的编译器编译后的执行代码被缓存下来，那么下次调用时只要是相同的预编译语句就不需要编译，只要将参数直接传入编译过的语句执行代码中就会得到执行。

在statement语句中,即使是相同操作但因为数据内容不一样,所以整个语句本身不能匹配,没有缓存语句的意义.事实是没有数据库会对普通语句编译后的执行代码缓存.这样每执行一次都要对传入的语句编译一次.

对Mysql来说两者的性能耗时几乎没有区别，但对oracle数据库则非常明显。

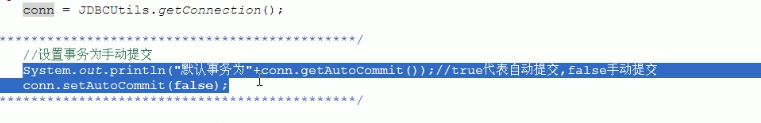


1. **事务操作**



步骤

1）、设置事务为手动提交

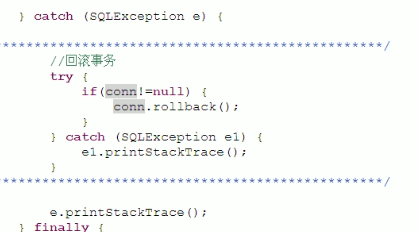


2）、操作数据库

3）、提交事务



4）、出错，事务回滚



1. **批处理**

当需要成批插入或者更新记录时。可以采用Java的批量更新机制，这一机制允许多条语句一次性提交给数据库批量处理。通常情况下比单独提交处理更有效率

JDBC的批量处理语句包括下面两个方法：

* addBatch(String)：添加需要批量处理的SQL语句或是参数；
* executeBatch（）；执行批量处理语句；

通常我们会遇到两种批量执行SQL语句的情况：

* 多条SQL语句的批量处理；
* 一个SQL语句的批量传参；

|  |
| --- |
| /\*\*  \* 用Statement批量保存5000条记录  \* mysql statement batch time=3047  \*/  @Test  public void testStatementBatch() {  Connection conn = null;  Statement stmt = null;  long time = System.currentTimeMillis();  conn = JdbcUtil.getConnection();  try {  stmt = conn.createStatement();  for(int i=0;i<5000;i++) {  String sql = "INSERT INTO b\_user(id,NAME,PASSWORD,age) VALUES(1,'t','t',1)";  stmt.addBatch(sql);  }  stmt.executeBatch();  System.out.println("mysql statement batch time="+(System.currentTimeMillis()-time));  } catch (SQLException e) {  e.printStackTrace();  } finally {  JdbcUtil.closeResoure(conn, stmt, null);  }  } |

1. **数据库连接池**

创建Connection的过程是非常耗时的，为了保证Connection可以重用,加快程序的效率（每次创建连接的时候很耗资源）。应该将Connection进行池管理。

连接池的作用：第一是管理多个连接，第二是回收连接

我们可以使用静态工厂方法管理一个唯一的连接

**静态方法得到连接**

|  |
| --- |
| /\*\*   \* 用静态工厂方法管理一个唯一的可重用的连接   \*/  **public** **class** ConnUtils {  **private** **static** Connection *con*;  //在静态代码块中创建与数据库的连接  **static**{  **try**{  Class.*forName*("com.mysql.jdbc.Driver");  String url = "jdbc:mysql:///db909?characterEncoding=UTf8";  *con* = DriverManager.*getConnection*(url,"root","1234");  }**catch**(Exception e){  **throw** **new** RuntimeException(e.getMessage(),e);  }  }  //使用一个静态方法－静态工厂方法，返回connection实例  **public** **static** Connection getCon(){  **return** *con*;  }  } |

**自己编写连接池**

|  |
| --- |
| **public** **class** ConnUtils2 {  //声明一个容器,放所有声明的连接Connection  **private** **static** List<Connection> *pool* = **new** ArrayList<Connection>();  **static**{  **try**{  Class.*forName*("com.mysql.jdbc.Driver");  String url = "jdbc:mysql:///db909?characterEncoding=UTf8";  **for**(**int** i=0;i<3;i++){  //创建三个连接  Connection con = DriverManager.*getConnection*(url,"root","1234");  //将这个三个连接放到pool中去  *pool*.add(con);  }  System.***err***.println("连接是:"+*pool*);  }**catch**(Exception e){  **throw** **new** RuntimeException(e.getMessage(),e);  }  }  **public** **static** Connection getCon(){  **synchronized** (*pool*) {  Connection con = *pool*.remove(0);  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***err***.println("还有："+*pool*.size());  **return** con;  }  }  //手工的还连接  **public** **static** **void** back(Connection con){  System.***err***.println("还连接："+con);  *pool*.add(con);  }  } |

程序员总是喜欢去调用Connection的close方法，所以为了回收连接，且不让连接对其进行关闭，我们应该重写close方法，对close方法进行增强（方法有3种：子类继承、包装模式、代理模式），程序员调用close方法时不是关连接而是还连接。

**动态代理原理**

动态代理作用：

1）、对某个方法增强

2）、在不污染原来的情况下，修改原类的行为

3）、拦截所有在代理上执行的方法（除了getClass方法）

代理类与被代理类，两个不同的实体，可以修改被代理类的所有方法。

**要求：**

1）、所有被代理类，都必须要拥有一个借口

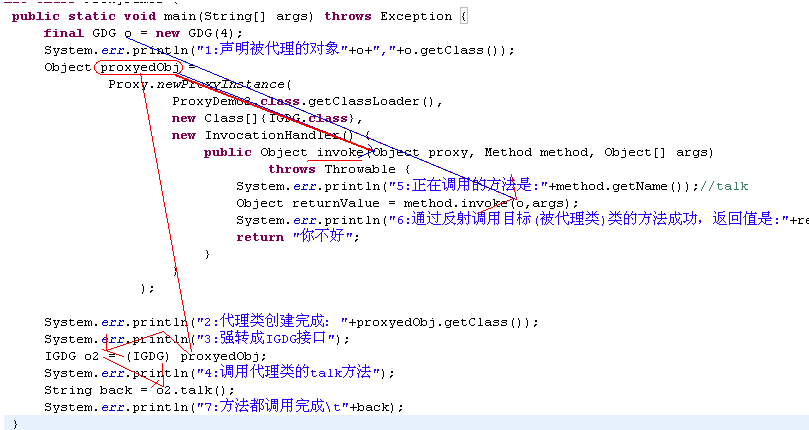
本质上是对方法进行修改，但其实它是通过反射执行的某个方法。

**动态代理的核心类**

1)、Proxy—用于创建给定借口的子类，在内存中动态的创建。$Proxy0，此类只使用一次

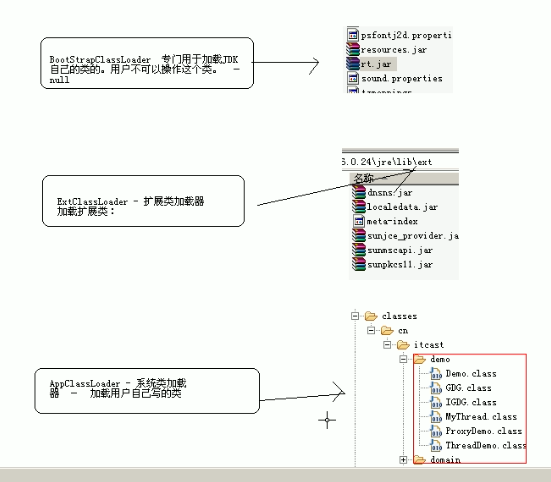
2）、InvocationHandler—执行句柄。在执行时可以获取被代理类的所有反射。用户的每一次调用都会被这个句柄拦截到



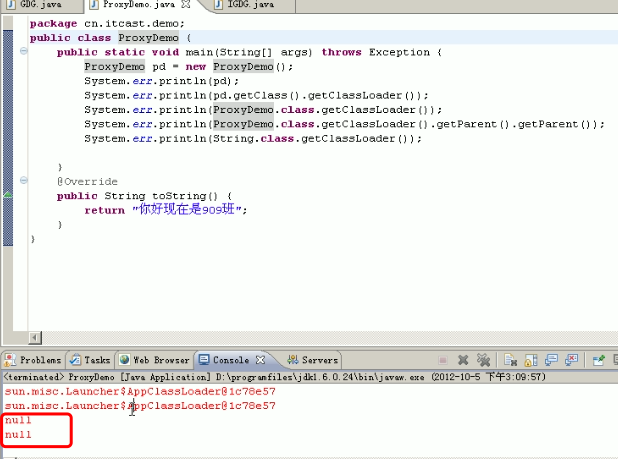


**类加载器**

父类委托机制



查看各个类的加载器情况



如何让自己的类被ExtClassLoader加载？

可以把自己的类打包成jar放到java安装目录下的ext文件夹下

**动态代理实现连接池**

代理目标：原生的Connection

代理的目的：修改close方法，让close方法不可以关闭连接，且主动收回连接。

通过动态代理，和线程通讯：

1）、对Cxonnection进行代理。

　２）、在获取Ｃonnection时，通过同步，如果没有连接时，就让线程进入等待池。

　３）、修改close方法，且在还了连接以后唤醒正在等待的线程。

|  |
| --- |
| **public** **class** ConnUtils3 {  **private** **static** LinkedList<Connection> *pool* = **new** LinkedList<Connection>();  **static**{  **try** {  //声明资源器类　－  Properties prop = **new** Properties();  //获取这个文件的路径  URL url = ConnUtils3.**class**.getClassLoader().getResource("jdbc.properties");  String path = url.getPath();  //为了防止有中文或是空格  path = URLDecoder.*decode*(path,"UTf-8");  File file = **new** File(path);  //加载jdbc.properties这个文件  prop.load(**new** FileInputStream(file));  //获取信息  String driver = prop.getProperty("driver");  Class.*forName*(driver);  String jdbcurl = prop.getProperty("url");  String nm = prop.getProperty("name");  String pwd = prop.getProperty("pwd");  //创建三个原生的连接，都将它们代理  String poolSize = prop.getProperty("poolSize");  **int** size = Integer.*parseInt*(poolSize);  **for**(**int** i=0;i<size;i++){  **final** Connection con = DriverManager.*getConnection*(jdbcurl,nm,pwd);  //对con进行动态代理  Object proxyedObj =  Proxy.*newProxyInstance*(ConnUtils3.**class**.getClassLoader(),  **new** Class[]{Connection.**class**},  **new** InvocationHandler() {  **public** Object invoke(Object proxy, Method method, Object[] args)  **throws** Throwable {  //是否是close方法  **if**(method.getName().equals("close")){  **synchronized** (*pool*) {  *pool*.addLast((Connection) proxy);  *pool*.notify();  }  **return** **null**;//如果调用的是close则不会调用被代理类的方法。  }  **return** method.invoke(con, args);  }  });  //将代理对象放到pool中  *pool*.add((Connection) proxyedObj);  }  } **catch** (Exception e) {  e.printStackTrace();  }  }    **public** **static** Connection getConn(){  **synchronized** (*pool*) {  **if**(*pool*.size()==0){  **try** {  *pool*.wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **return** *getConn*();  }**else**{  Connection con = *pool*.removeFirst();  System.***err***.println("还有几个："+*pool*.size());  **return** con;  }  }  }  } |

1. **DataSource实现连接池**

DataSource 接口由驱动程序供应商实现。共有三种类型的实现：

1. 基本实现：生成标准的 Connection 对象，一个DataSource数据源中，只有一个Connection ,这个不是池管理。

2. 连接池实现：生成自动参与连接池的 Connection 对象。

3. 分布式事务实现：生成一个 Connection 对象，该对象可用于分布式事务，大多数情况下总是参与连接池。JTA.jar – SUN。

**标准的连接池，要求：**

1：实现dataSource接口。

2：声明一个集合类用于管理多个连接。

3：必须要拥有一种能力，回收连接。

4：必须要实现一个方法，getConnection以获取一个连接。

5：实现DataSource接口的类，一般不能拥有static池对象。List.

6：在一个程序中，要求只拥有一个DataSource实例就可以了。

|  |
| --- |
| /\*\*   \* 标准的连接   \*/  **public** **class** DBPool **implements** DataSource {  //声明一个池管理对象  **private** LinkedList<Connection> pool = **new** LinkedList<Connection>();  //在初始化这个DataSourc的子类时在构造方法设置多个连接  **public** DBPool(){  **try**{  Class.*forName*("com.mysql.jdbc.Driver");  String url = "jdbc:mysql:///db909?characterEncoding=UTf8";  **for**(**int** i=0;i<3;i++){  **final** Connection con = DriverManager.*getConnection*(url,"root","1234");  Object proxyedConn =  Proxy.*newProxyInstance*(DBPool.**class**.getClassLoader(),  **new** Class[]{Connection.**class**},  **new** InvocationHandler() {  **public** Object invoke(Object proxyedConnection, Method method, Object[] args)  **throws** Throwable {  **if**(method.getName().equals("close")){  **synchronized** (pool) {  pool.addLast((Connection) proxyedConnection);  pool.notify();  }  **return**  **null**;  }  //目标方法的返回值  Object returnValue=method.invoke(con, args);  **return** returnValue;  }  });  pool.add((Connection) proxyedConn);  }  }**catch**(Exception e){  **throw** **new** RuntimeException(e.getMessage(),e);  }  }  **public** Connection getConnection() **throws** SQLException {  **synchronized** (pool) {  **if**(pool.size()==0){  **try** {  pool.wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **return** getConnection();  }  Connection con = pool.removeFirst();  System.***err***.println("siize:"+pool.size());  **return** con;  }  }  } |

1. **包装Connection**

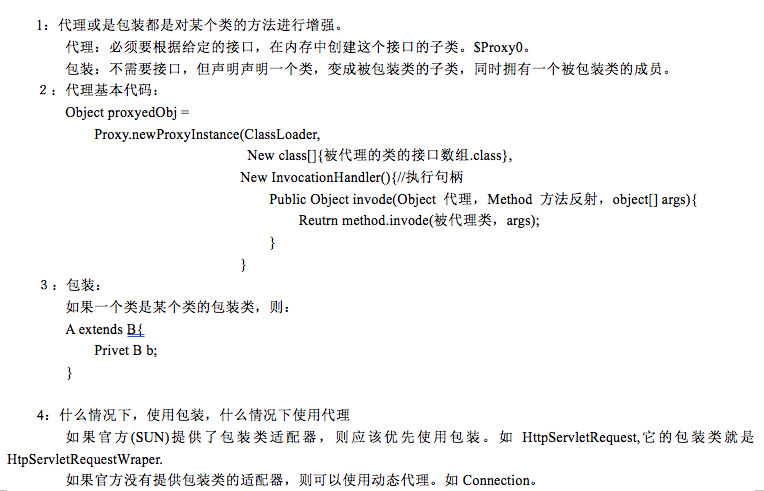
使用包装模式（装饰模式）包装Connection，首先此类成为Connection的子类，声明一个成员且在构造中给conn设置值

|  |
| --- |
| // 以下包装connection  **class** MyConn **implements** Connection {  // 声明被包装类的成员  **private** Connection conn;  // 通过构造接收MySql的connection的对象JDBC4Connection@8888  **public** MyConn(Connection con) {  **this**.conn = con;  }  //关闭连接  **public** **void** close() **throws** SQLException {  **synchronized** (pool) {  //有人调用了关闭方法，不能关  System.***err***.println("有人还连接了。。。。"+**this**);  pool.add(**this**);  pool.notify();  }  }  **public** Statement createStatement() **throws** SQLException {  **return** conn.createStatement();  }  **public** PreparedStatement prepareStatement(String sql)  **throws** SQLException {  **return** conn.prepareStatement(sql);  }  //备注：此处省略了其他需要实现的方法代码。  } |

此处DataSource对Connection的使用代码如下：

|  |
| --- |
| **public** **class** MyDataSource **implements** DataSource {  **private** LinkedList<Connection> pool = **new** LinkedList<Connection>();  **public** MyDataSource() {  **try** {  Class.*forName*("com.mysql.jdbc.Driver");  String url = "jdbc:mysql:///db909?characterEncoding=UTf8";  **for** (**int** i = 0; i < 3; i++) {  //创建原生的连接，// com.mysql.jdbc.JDBC4Connection@8888  Connection con = DriverManager.*getConnection*(url, "root",  "1234");  //声明包装类  MyConn conn = **new** MyConn(con);  pool.add(conn);//将包装类添加到池中去  }  } **catch** (Exception e) {  e.printStackTrace();  }  }  //此方法来自于datasource,用于返回一个连接  **public** Connection getConnection() **throws** SQLException {  **synchronized** (pool) {  **if** (pool.size() == 0) {  **try** {  pool.wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **return** getConnection();  }  Connection con = pool.removeFirst();  System.***err***.println("siize:" + pool.size());  **return** con;  }  }  //备注：此处省略了其他需要实现的方法代码。  } |

1. **代理和包装总结**





1. **DBCP连接池**

简单的demo

|  |
| --- |
| @Test  **public** **void** testpool1() **throws** Exception{  BasicDataSource ds = **new** BasicDataSource();  //设置driver  ds.setDriverClassName("com.mysql.jdbc.Driver");  //设置url  ds.setUrl("jdbc:mysql:///test?characterEncoding=UTf8");  ds.setPassword("123456");  ds.setUsername("root");  ds.setMaxActive(3);//设置最多有几个连接  ds.setInitialSize(2);//设置在开始时创建几个连接  //..................  ds.setDefaultAutoCommit(**true**);//设置所有连接是否自动提交  ds.setMaxIdle(3000);//设置每个连接最大的空闲时间    Connection c1 = ds.getConnection();  Connection c2 = ds.getConnection();  Connection c3 = ds.getConnection(); System.***err***.println("c1:"+c1.hashCode()+","+c1.getClass());//com.mysql.jdbc.Jdbc4Connection@11111,  System.***err***.println("c2:"+c2.hashCode());//cn.itcast.MyDataSource$MyConn@11111  System.***err***.println("c3:"+c3.hashCode());  c2.close();  Connection c4 = ds.getConnection();  System.***err***.println("c4:"+c4.hashCode());  } |

此处只要我们关闭c2连接，那么再次获取连接的时候，就能拿到c2连接，也就是c4和c2其实拿到的是同一个连接，这就是DBCP连接池

通过配置文件获取连接

|  |
| --- |
| @Test  **public** **void** testPool2() **throws** Exception{  Properties p = **new** Properties();  p.load(DbcpPool.**class**.getResourceAsStream("jdbc.properties"));  DataSource ds = BasicDataSourceFactory.*createDataSource*(p);  Connection c1 = ds.getConnection();  System.***err***.println(c1.hashCode()+","+c1.getClass());  C1.close();  Connection c2 = ds.getConnection();  System.***err***.println("c2:"+c2.hashCode());  } |

jdbc.properties配置文件

|  |
| --- |
| #以下所有属性，都来自于BaiscDataSource的setXxxx  driverClassName=com.mysql.jdbc.Driver  url=jdbc:mysql:///test  username=root  password=123456  maxActive=3 |

通常上面代码都是我们测试的做法，实际开发中我们不可能每次使用的时候都去加载数据库吃配置信息，然后再创建dataSource对象。一个应用应该在启动的时候才会去加载数据库配置信息，然后创建dataSource对象，然后在每次使用的时候通过dataSource对象获取Connection对象，再操作数据库就行了。但是此处我们必须保证dataSource对象是唯一的，不能多次创建，我们使用工厂类去实现该功能。

|  |
| --- |
| **public** **class** DataSourceUtils {  //无法实例化该类  **private** DataSourceUtils(){}  **private** **static** DataSource *ds*;  **static**{  **try**{  Properties p = **new** Properties();  p.load(DataSourceUtils.**class**.getClassLoader().getResourceAsStream("jdbc.properties"));  *ds* = BasicDataSourceFactory.*createDataSource*(p);  }**catch**(Exception e){}  }  //返回一个唯一的连接  **public** **static** Connection getCon(){  Connection con = **null**;  **try** {  con = *ds*.getConnection();  } **catch** (SQLException e) {  e.printStackTrace();  }  **return** con;  }  //提供一个方法返回DataSource  **public** **static** DataSource getDs(){  **return** *ds*;  }  } |

1. **DBUtils**

它是操作数据库的第三方包，DBCP只负责连接数据库的，因此DBUtils依赖DBCP,DBUtils使用的是jdbc的原生事务，需要自己手动操作。

**查询：**

|  |
| --- |
| //封装成唯一的一个bean  @Test  **public** **void** query2() **throws** Exception{  QueryRunner run = **new** QueryRunner(*getDatasSource*());  String sql = "select \* from users where id=?";  User user = run.query(sql,**new** BeanHandler<User>(User.**class**),"U003");  System.***err***.println(user);  } |

Handler主要有BeanHandler、KeyedHandler、MapHandler、MapListHandler、ScalarHandler、BeanListHandler、ResultSetHandler

**增加：**

|  |
| --- |
| @Test  **public** **void** insert2() **throws** Exception{  QueryRunner run = **new** QueryRunner(*getDatasSource*());  String sql = "insert into users values(?,?,?)";  run.update(sql,"U003","王五","7777");  } |

删除:

|  |
| --- |
| @Test  **public** **void** del1() **throws** Exception{  QueryRunner run = **new** QueryRunner(*getDatasSource*());  String sql = "delete from users where name=?";  **int** len = run.update(sql,"李四");  System.***err***.println(len);  } |

更新：

|  |
| --- |
| @Test  **public** **void** udpate1() **throws** Exception{  QueryRunner run = **new** QueryRunner(*getDatasSource*());  String sql = "update users set name=? where id=?";  run.update(sql,"赵'七","U001");  } |

1. **C3P0连接池**

ComboPooledDataSource它是DataSource子类

步骤：

1）、导入c3p0包

2）、在classpath目录下，创建一个c3p0-config.xml

3）、创建工厂类获取这个连接

DataSourceUtils类

|  |
| --- |
| **public** **class** DataSourceUtils {  **private** **static** DataSource *ds*;  **static**{  //默认的读取c3p0-config.xml中默认配置  *ds* = **new** ComboPooledDataSource();  }  **public** **static** DataSource getDatasSource(){  **return** *ds*;  }  **public** **static** Connection getConn(){  Connection con = **null**;  **try**{  con = *ds*.getConnection();//每一次从ds中获取一个新的连接  }**catch**(Exception e){  e.printStackTrace();  }  **return** con;  }  } |

C3p0配置文件c3p0-config.xml内容

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <c3p0-config>  <!-- 默认配置，只可以出现一次 -->  <default-config>  <!-- 连接超时设置30秒 -->  <property name=*"checkoutTimeout"*>30000</property>  <!-- 30秒检查一次connection的空闲 -->  <property name=*"idleConnectionTestPeriod"*>30</property>  <!--初始化的池大小 -->  <property name=*"initialPoolSize"*>2</property>  <!-- 最多的一个connection空闲时间 -->  <property name=*"maxIdleTime"*>30</property>  <!-- 最多可以有多少个连接connection -->  <property name=*"maxPoolSize"*>5</property>  <!-- 最少的池中有几个连接 -->  <property name=*"minPoolSize"*>2</property>  <!-- 批处理的语句的个数 -->  <property name=*"maxStatements"*>50</property>  <!-- 每次增长几个连接 -->  <property name=*"acquireIncrement"*>3</property>  <property name=*"driverClass"*>com.mysql.jdbc.Driver</property>  <property name=*"jdbcUrl"*>  <![CDATA[jdbc:mysql://127.0.0.1:3306/test?useUnicode=true&characterEncoding=UTF-8]]>  </property>  <property name=*"user"*>root</property>  <property name=*"password"*>123456</property>  </default-config>  </c3p0-config> |

测试类

|  |
| --- |
| @Test  **public** **void** c3p0() **throws** Exception{  Connection c1 = DataSourceUtils.*getConn*();  Connection c2 = DataSourceUtils.*getConn*();  Connection c3 = DataSourceUtils.*getConn*();  Connection c4 = DataSourceUtils.*getConn*();  Connection c5 = DataSourceUtils.*getConn*();    System.***err***.println("c1:"+c1);  System.***err***.println("c2:"+c2);  System.***err***.println("c3:"+c3);  System.***err***.println("c4:"+c4);  System.***err***.println("c5:"+c5);  c1.close();  Connection c6 = DataSourceUtils.*getConn*();  System.***err***.println("c6:"+c6);  } |

测试结果：



此处我们发现c6的内存地址和c1的内存地址不一样，但是c6的确是使用的是c1归还的连接。这是为什么呢？

原因是因为ComboPooledDataSource在获取连接的时候（使用的时候），才去包装原生的连接，而不是在构造函数里面去包装原生连接。Dbcp则是在构造函数执行时包装，此处明显c3p0优于DBCP

实现该功能代码如下：

|  |
| --- |
| **public** **class** MyDataSource **implements** DataSource {  **private** LinkedList<Connection> pool = **new** LinkedList<Connection>();  **public** MyDataSource() {  **try** {  Class.*forName*("com.mysql.jdbc.Driver");  String url = "jdbc:mysql:///db909?characterEncoding=UTf8";  **for** (**int** i = 0; i < 3; i++) {  //创建原生的连接，// com.mysql.jdbc.JDBC4Connection@8888  Connection con = DriverManager.*getConnection*(url, "root","1234");  pool.add(con);//将没有经过包装类放到池  }  } **catch** (Exception e) {  e.printStackTrace();  }  }  //此方法来自于datasource,用于返回一个连接  **public** Connection getConnection() **throws** SQLException {  **synchronized** (pool) {  **if** (pool.size() == 0) {  **try** {  pool.wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **return** getConnection();  }  Connection con = pool.removeFirst();  System.***err***.println("siize:" + pool.size());  //在返回之前包装这个类  MyConn mc = **new** MyConn(con);  **return** mc;  }  }  // 以下包装connection  **class** MyConn **implements** Connection {  // 声明被包装类的成员  **private** Connection conn;  // 通过构造接收MySql的connection的对象JDBC4Connection@8888  **public** MyConn(Connection con) {  **this**.conn = con;  }  //关闭连接  **public** **void** close() **throws** SQLException {  **synchronized** (pool) {  //有人调用了关闭方法，不能关  System.***err***.println("有人还连接了。。。。"+**this**);  pool.add(conn);  pool.notify();  }  }  //此处省略其他实现方法  } |

1. **DBUtils的QueryRunner类自己编写**

QueryRunner类

|  |
| --- |
| **public** **class** QueryRunner {  //接收一个ds  **private** DataSource ds;  **public** QueryRunner() {  }  **public** QueryRunner(DataSource ds) {  **this**.ds = ds;  }  /\*\*  \* 只封装成List<Map>  \*/  **public** List<Map<String,Object>> query(String sql){  //封装数据用  List<Map<String,Object>> list = **new** ArrayList<Map<String,Object>>();//声明返回的对象  Connection con = **null**;  **try**{  con = ds.getConnection();  //执行查询  Statement st = con.createStatement();  ResultSet rs = st.executeQuery(sql);  //分析结果集  ResultSetMetaData rsmd = rs.getMetaData();  //获取列数  **int** cols = rsmd.getColumnCount();  //遍历数据  **while**(rs.next()){  //一行数据  Map<String,Object> mm = **new** HashMap<String, Object>();  //遍历列  **for**(**int** i=0;i<cols;i++){  //获取列名  String colName = rsmd.getColumnName(i+1);  //获取数据  Object val     = rs.getObject(i+1);  //封装到map  mm.put(colName, val);  }  //将这个map放到list  list.add(mm);  }    }**catch**(Exception e){  **throw** **new** RuntimeException(e);  }**finally**{  **try** {  con.close();  } **catch** (SQLException e) {  e.printStackTrace();  }  }  **return** list;  }  /\*\*  \* 封装成List<Bean>  \* <T>定义泛型  \* List<T> 使用泛型  \* Class<T> 接收一个T类型的字节码对象  \* Type 是Class父类  \*/  **public** <T> List<T> queryForBean(String sql,Class<T> cls){  //声明返回的对象  List<T> list = **new** ArrayList<T>();  //获取连接  Connection con= **null**;  **try**{  //查询sql,封装数据  con = ds.getConnection();  Statement st = con.createStatement();  //查询  ResultSet rs = st.executeQuery(sql);  //获取列数量  ResultSetMetaData rsmd = rs.getMetaData();  **int** cols = rsmd.getColumnCount();  //遍历数据  **while**(rs.next()){  //每次遍历一行，应该实例化一个Bean  T t = cls.newInstance();//User(id,name,pwd);-select id,name,pwd from users;  //遍历所有列  **for**(**int** i=0;i<cols;i++){  String colName = rsmd.getColumnName(i+1);//id - user.setId();  colName = colName.toLowerCase();  //将colName转成setXxx  String methodName = "set"+colName.substring(0,1).toUpperCase()+colName.substring(1).toLowerCase();  //通过数据库获取数据库对应的Ajava类型QName  //Varchar-java.lang.String,int--java.lang.Integer  String javaType = rsmd.getColumnClassName(i+1);//java.lang.String -  //反射出T的方法  **try**{  Method method = cls.getMethod(methodName,Class.*forName*(javaType));  //执行  //查询数据库  Object value = rs.getObject(colName);  method.invoke(t,value);  }**catch**(Exception e){    }  }  //将t放到  list.add(t);  }  }**catch**(Exception e){  e.printStackTrace();  }**finally**{  **try** {  con.close();  } **catch** (SQLException e) {  e.printStackTrace();  }  }  **return** list;  }  /\*\*  \* 有回调的查询  \*/  **public** <T> T query(String sql,MyHandler<T> mh){  T t = **null**;  //声明conn  Connection con = **null**;  **try**{  con = ds.getConnection();  //执行查询  Statement st = con.createStatement();  ResultSet rs = st.executeQuery(sql);  //让回调去执行数据封装  t = mh.handler(rs);  }**catch**(Exception e){  e.printStackTrace();  }**finally**{  **try** {  con.close();  } **catch** (SQLException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  **return** t;  } |

回调接口MyHandler

|  |
| --- |
| **public** **interface** MyHandler<T> {  T handler(ResultSet rs);  } |

回调类MyBeanListHandler

|  |
| --- |
| **public** **class** MyBeanListHandler<T> **implements** MyHandler<List<T>> {  **private** Class<T> cls;  **public** MyBeanListHandler(Class<T> cls) {  **this**.cls=cls;  }  **public** List<T> handler(ResultSet rs) {  List<T> list = **new** ArrayList<T>();  **try**{  ResultSetMetaData rsmd = rs.getMetaData();  **int** cols = rsmd.getColumnCount();  **while**(rs.next()){  T t = cls.newInstance();  //获取列名  **for**(**int** i=0;i<cols;i++){  String colName = rsmd.getColumnName(i+1);  String methodName =  "set"+colName.substring(0,1)  .toUpperCase()  +colName.substring(1).toLowerCase();  //获取类型  String javaType = rsmd.getColumnClassName(i+1);//="java.lang.String";  **try**{  Method mm = cls.getMethod(methodName, Class.*forName*(javaType));  Object val = rs.getObject(i+1);  mm.invoke(t,val);  }**catch**(Exception e){}  }  list.add(t);  }  }**catch** (Exception e) {  }  **return** list;  }  } |

使用类

|  |
| --- |
| **public** **class** MyDbUtils {  @Test  **public** **void** myutils(){  QueryRunner run = **new** QueryRunner(DataSourceUtils.*getDatasSource*());  String sql = "select \* from uc";  List<Map<String,Object>> list = run.query(sql);  **for**(Map mm:list){  System.***err***.println(mm);  }  }  @Test  **public** **void** myutils2(){  QueryRunner run = **new** QueryRunner(DataSourceUtils.*getDatasSource*());  String sql = "select \* from contacts";  List<Contact> us = run.queryForBean(sql,Contact.**class**);  **for**(Contact u:us){  System.***err***.println(u);  }    }  @Test  **public** **void** myutils3(){  QueryRunner run = **new** QueryRunner(DataSourceUtils.*getDatasSource*());  String sql = "select \* from users";  List<User> cs = run.query(sql,**new** MyBeanListHandler<User>(User.**class**));  **for**(User c:cs){  System.***err***.println(">>:"+c);  }  }  } |

1. **Javaweb中的事务**

执行多个dao的时候，不管调用多少个dao，只要是同一个Connection，且在最后一个dao中提交，或是回滚可以保证事务统一性。

如何保证所有的dao在同一个时间，访问的是同一个Connection？

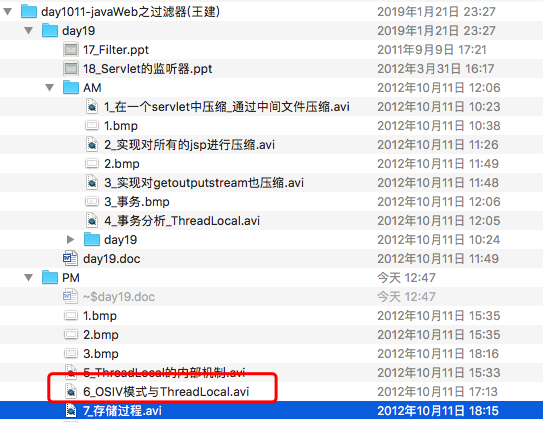
也就是为每一个线程绑定一个Connection，不管该线程调用多少个dao，使用的都是同一个Connection。

ThreadLocal不是线程，它是管理线程的一个map。可以解决上面的问题

Javaweb中，一个用户新的请求过来到服务器，服务器调用controller，controller调service层，service层调用dao层，不管调用多少层，该请求的线程始终是同一个。

1. **OSIV模式**

在过虑器中控制事务（具体实现看如下图），就叫一种模式：OSIV模式



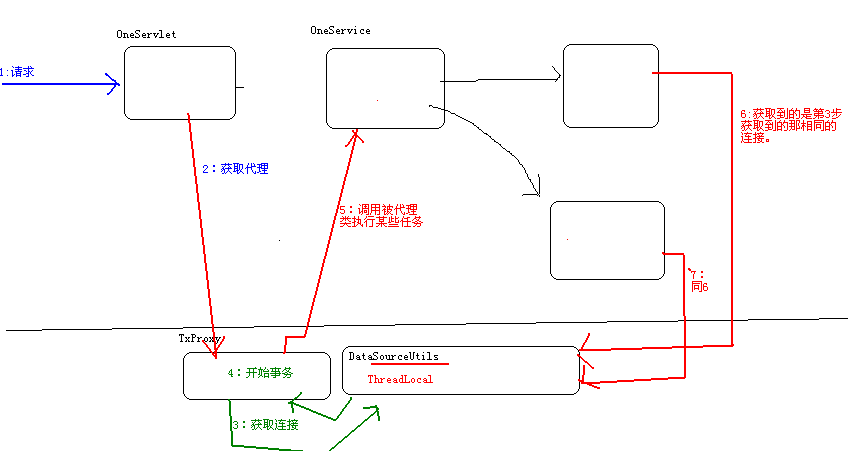
OSIV – Open Session In View =- 打开与数据库的会话在View层。Hibernate现在使用的是AOP模式

这是一个长事务

好处：长事务。可以在View层即找开事务。

不好处：当页面上有错误时也会回滚。解决：判断异常的类型。

1. **代理控制事务**

****

如上图所示，javaweb的事务最好是在service层进行处理，如果dao2出现异常则需要回滚dao1的操作。我们如何统一处理事务呢？此时我们可以对service进行代理，在service代理类里面对事务进行控制，因此此处的核心就是service的代理类。

书写数据库工具类，获取Connection

|  |
| --- |
| **public** **class** DataSourceUtils {  **private** **static** DataSource *ds*;  //声明ThreadLocal容器对象  **private** **static** ThreadLocal<Connection> *tl* = **new** ThreadLocal<Connection>();  **static** {  *ds* = // 默认的读取c3p0-config.xml中默认配置  **new** ComboPooledDataSource();  }  **public** **static** DataSource getDatasSource() {  **return** *ds*;  }  //提供一个返回Connection的方法  **public** **static** Connection getConn(){  Connection con = **null**;  //先从tl中获取数据  con = *tl*.get();  **if**(con==**null**){  **try**{  con = *getDatasSource*().getConnection();  //放到tl  *tl*.set(con);  }**catch**(Exception e){  e.printStackTrace();  }  }  **return** con;  }  } |

步骤2：声明两个dao类，此处省略

步骤3：声明service接口ITxService，具体代码此处省略

步骤4：声明ITxService接口实现类

|  |
| --- |
| **public** **class** TxService **implements** ITxService {  //注入两个dao  **private** Dao1 dao1 = **new** Dao1();  **private** Dao2 dao2 = **new** Dao2();  **public** **void** save(){  dao1.save();  dao2.save();  }  } |

步骤5：声明service的代理类，他可以对所有的service进行代理

|  |
| --- |
| /\*\*   \* 事务的代理类，可以代理任意的Service   \*/  **public** **class** TxProxy **implements** InvocationHandler{  **private** Object src; //声明被代理类对象  **private** TxProxy(Object src){//在私有的构造中给成员设置值  **this**.src=src;  }  /\*\*  \* 提供一个静态的方法返回代理对象  \*/  **public** **static** Object factory(Object src){  Object proxyedObj = //生成被代理类的接口的子类  Proxy.*newProxyInstance*(  TxProxy.**class**.getClassLoader(),  src.getClass().getInterfaces(),  **new** TxProxy(src));  **return** proxyedObj;  }    /\*\*  \* 以下是执行的句柄，当调用代理类的任意方法时都会调用这个方法  \* 在这儿是管理事务的关键  \*/  **public** Object invoke(Object proxy, Method method, Object[] args)  **throws** Throwable {  //第一步：声明连接  Connection con = **null**;  Object returnValue = **null**;  **try**{  //第二步：获取连接  con = DataSourceUtils.*getConn*();  //第三步：设置事务的开始  con.setAutoCommit(**false**);  //第四步：调用目标类(被代理类)的方法  returnValue = method.invoke(src, args);  //第五步：调用如果成功  con.commit();  }**catch**(Exception e){  con.rollback();  **throw** e;  }**finally**{  con.close();  DataSourceUtils.remove();  }  **return** returnValue;  }  } |

步骤7：在servlet中调用service的代理类，此时的代理类就会对service进行代理，控制事务。

|  |
| --- |
| **public** **class** TxServlet **extends** HttpServlet {  /\*\*  \* 注入service的实例,必须要使用接口  \*/  **private** ITxService service = (ITxService) TxProxy.factory(**new** TxService());  **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  service.save();  }  } |

1. **使用注解+代理控制事务**

在**代理控制事务**的基础上进行改进。

在需要事务的方法上添加事务注解。如果不需要事务，则不添加注解。

步骤1：声明事务注解

|  |
| --- |
| @Retention(RetentionPolicy.***RUNTIME***)  @Target(value=ElementType.***METHOD***)  **public** **@interface** Tx {  } |

步骤2：给需要事务的方法添加刚才声明的注解

|  |
| --- |
| **public** **interface** ITxService {  //开启事务  @Tx  **void** save();    **public** **void** query();  } |

步骤3：修改代理类

|  |
| --- |
| /\*\*   \* 事务的代理类，可以代理任意的Service   \*/  **public** **class** TxProxy **implements** InvocationHandler{  **private** Object src; //声明被代理类对象  **private** TxProxy(Object src){//在私有的构造中给成员设置值  **this**.src=src;  }  /\*\*  \* 提供一个静态的方法返回代理对象  \*/  **public** **static** Object factory(Object src){  System.***err***.println("0：返回代理类");  Object proxyedObj = //生成被代理类的接口的子类  Proxy.*newProxyInstance*(  TxProxy.**class**.getClassLoader(),  src.getClass().getInterfaces(),  **new** TxProxy(src));  **return** proxyedObj;  }    /\*\*  \* 以下是执行的句柄，当调用代理类的任意方法时都会调用这个方法  \* 在这儿是管理事务的关键  \*/  **public** Object invoke(Object proxy, Method method, Object[] args)  **throws** Throwable {  //判断是否有事务的注解  **if**(!method.isAnnotationPresent(Tx.**class**)){  System.***err***.println("没有事务的注解，用户不需要事务。直接执行目标方法");  **return** method.invoke(src, args);  }    //第一步：声明连接  Connection con = **null**;  Object returnValue = **null**;  **try**{  //第二步：获取连接  con = DataSourceUtils.*getConn*();  //第三步：设置事务的开始  System.***err***.println("设置事务的开始");  con.setAutoCommit(**false**);  //第四步：调用目标类(被代理类)的方法  returnValue = method.invoke(src, args);  //第五步：调用如果成功提交  System.***err***.println("调用如果成功提交");  con.commit();  }**catch**(Exception e){  System.***err***.println("调用不成功回滚");  con.rollback();  **throw** e;  }**finally**{  con.close();  DataSourceUtils.remove();  }  **return** returnValue;  }  } |

1. **JNDI**
2. **概念**

JNDI – Java Name and Driectory Interface。

JNDI主要用于在容器中配置某些资源，让所有项目可以使用。

JNDI可以提供：

1：数据库连接池。

自定义连接池

第三方连接池

Dbcp

C3p0

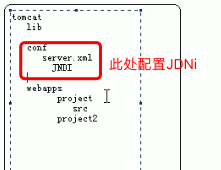
JNDI

2：邮件服务器定义。

3：加载任意的类，让所有项目都可以使用。

1. **Tomcat中使用JDNI**

JNDI:[定义在tomcat容器中，所有项目必须要通过查找的方式获取这个服务。



Tomcat的webapps下面可能有多个project项目，而我们的JDNI是配置在Tomcat上的，因此该Tomcat上的所有project项目都可以使用该JDNI。Project是通过查找的方式获取这个服务。

javax.naming.Context 环境(ENV)上下文。它的一个子类：InitialContext（其中的lookup方法）。

Tomcat支持JNDI，到tomcat的docs查看说明：

配置JNDI:

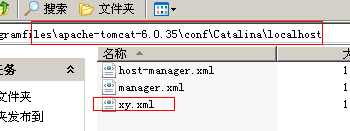
由于 JNDI需要在server.xml中进行配置，所以就不用发布项目

第一步：在tomcat/conf/server.xml中配置：

|  |
| --- |
| <Context path="/aaa" docBase="D:/programfiles/MyEclipse10/wk3/day25/WebRoot">  <Resource name="jdbc/mysql" auth="Container"  type="javax.sql.DataSource" driverClassName="com.mysql.jdbc.Driver"  url="jdbc:mysql:///db909?characterEncoding=UTF8"  username="root" password="1234" maxActive="10" maxIdle="30"  maxWait="-1"/>  </Context> |

auth="Container"：jdni的拥有者

方式2:去Tomcat的如下目录去单独新建一个配置文件，将上面的内容放入该配置文件。



第二步：将mysql-connection.jar放到tomcat/lib目录下

第三步：在java项目中引用，在代码中直接查询这个jndi:

第一种获取的方式：

|  |
| --- |
| **public** **class** TwoServlet **extends** HttpServlet {  **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {    **try** {  //1声明Context上下文查找对象  Context ctx = **new** InitialContext();  //第二步：查找apache环境  Context ctx1 = (Context) ctx.lookup("java:/comp/env");  //第三步：从环境中找jNDI  Object obj = ctx1.lookup("jdbc/mysql");  System.*err*.println(obj);  //第四步：测试连接是否成功  DataSource ds = (DataSource) obj;  Connection con = ds.getConnection();  System.*err*.println(">>"+con);  } **catch** (Exception e) {  e.printStackTrace();  }  }  } |

第二种：

|  |
| --- |
| **try** {  //1声明Context上下文查找对象  Context ctx = **new** InitialContext();  //第二步：查询  Object obj = ctx.lookup("java:/comp/env/jdbc/mysql");  System.*err*.println(obj);  DataSource ds = (DataSource) obj;  Connection con = ds.getConnection();  System.*err*.println(">>"+con);  Statement st = con.createStatement();  ResultSet rs = st.executeQuery("select \* from users");  **while**(rs.next()){  System.*err*.println(rs.getString(1));  }  } **catch** (Exception e) {  e.printStackTrace();  } |

1. **AA**