# JavaWeb

# 三层模型 MVC

* 1、MVC设计模式
* MVC设计模式
* MVC模式（Model-View-Controller)是软件工程中的一种软件架构模式，把软件系统分为三个基本部分：模型（Model)、视图（View)和控制器（Controller)。
* MVC模式最早为Trygve Reenskaug提出，为施乐帕罗奥多研究中心（Xerox PARC)的Smalltalk语言发明的一种软件设计模式。
* MVC可对程序的后期维护和扩展提供了方便，并且使程序某些部分的重用提供了方便。而且MVC也使程序简化，更加直观。  
  - 控制器Controller：对请求进行处理，负责请求转发；  
  - 视图View：界面设计人员进行图形界面设计；  
  - 模型Model：程序编写程序应用的功能（实现算法等等)、数据库管理；
* 注意，MVC不是Java的东西，几乎现在所有B/S结构的软件都采用了MVC设计模式。但是要注意，MVC在B/S结构软件并没有完全实现，例如在我们今后的B/S软件中并不会有事件驱动！
* 2、JavaWeb与MVC
* JavaWeb的经历了JSP Model1、JSP Model1二代、JSP Model2三个时期。
* 2.1 JSP Model1第一代
* JSP Model1是JavaWeb早期的模型，它适合小型Web项目，开发成本低！Model1第一代时期，服务器端只有JSP页面，所有的操作都在JSP页面中，连访问数据库的API也在JSP页面中完成。也就是说，所有的东西都耦合在一起，对后期的维护和扩展极为不利。
* 2.2 JSP Model1第二代
* JSP Model1第二代有所改进，把业务逻辑的内容放到了JavaBean中，而JSP页面负责显示以及请求调度的工作。虽然第二代比第一代好了些，但还让JSP做了过多的工作，JSP中把视图工作和请求调度（控制器)的工作耦合在一起了。
* 2.3 JSP Model2
* JSP Model2模式已经可以清晰的看到MVC完整的结构了。  
  - JSP：视图层，用来与用户打交道。负责接收用来的数据，以及显示数据给用户；  
  - Servlet：控制层，负责找到合适的模型对象来处理业务逻辑，转发到合适的视图；  
  - JavaBean：模型层，完成具体的业务工作，例如：开启、转账等。
* JSP Model2适合多人合作开发大型的Web项目，各司其职，互不干涉，有利于开发中的分工，有利于组件的重用。但是，Web项目的开发难度加大，同时对开发人员的技术要求也提高了。

# 5.1 JavaWeb经典三层框架

* 我们常说的三层框架是由JavaWeb提出的，也就是说这是JavaWeb独有的！
* 所谓三层是表述层（WEB层)、业务逻辑层（Business Logic)，以及数据访问层（Data Access)。  
  - WEB层：包含JSP和Servlet等与WEB相关的内容；  
  - 业务层：业务层中不包含JavaWeb API，它只关心业务逻辑；  
  - 数据层：封装了对数据库的访问细节；
* 注意，在业务层中不能出现JavaWeb API，例如request、response等。也就是说，业务层代码是可重用的，甚至可以应用到非Web环境中。业务层的每个方法可以理解成一个万能，例如转账业务方法。业务层依赖数据层，而Web层依赖业务层！

# Web服务器

* Web服务器的作用是接收客户端的请求，给客户端作出响应。
* 对于JavaWeb程序而已，还需要有Servlet容器，Servlet容器的基本功能是把动态资源转换成静态资源。
* 我们需要使用的是Web服务器和Servlet容器，通常这两者会集于一身。下面是对JavaWeb服务器：  
  - Tomcat（Apache)：当前应用最广的JavaWeb服务器；  
  - JBoss（Redhat红帽)：支持JavaEE，应用比较广；EJB容器  
  - GlassFish（Orcale)：Oracle开发JavaWeb服务器，应用不是很广；  
  - Resin（Caucho)：支持JavaEE，应用越来越广；  
  - Weblogic（Oracle)：收费；支持JavaEE，适合大型项目；  
  - Websphere（IBM)：收费；支持JavaEE，适合大型项目；
* 支持JavaEE是EJB容器

# Servlet

# 5.2 定义

* Servlet是JavaWeb的三大组件之一，它属于动态资源。Servlet的作用是处理请求，服务器会把接收到的请求交给Servlet来处理，在Servlet中通常需要：  
  - 接收请求数据；  
  - 处理请求；  
  - 完成响应。
* 例如客户端发出登录请求，或者输出注册请求，这些请求都应该由Servlet来完成处理！Servlet需要我们自己来编写，每个Servlet必须实现javax.servlet.Servlet接口。

# 5.3 生命周期

* 生命周期方法：  
  - void init(ServletConfig)：出生之后（1次)；  
  - void service(ServletRequest request, ServletResponse response)：每次处理请求时都会被调用；  
  - void destroy()：临死之前（1次)；
* 特性：  
  - 单例，一个类只有一个对象；当然可能存在多个Servlet类！  
  - 多线程的，所以它的效率是高的！（不是线程安全的)
* Servlet类由我们来写，但对象由服务器来创建，并且由服务器来调用相应的方法。

## Servlet的出生

* 服务器会在Servlet第一次被访问时创建Servlet，或者是在服务器启动时创建Servlet。如果服务器启动时就创建Servlet，那么还需要在web.xml文件中配置。也就是说默认情况下，Servlet是在第一次被访问时由服务器创建的。
* 而且一个Servlet类型，服务器只创建一个实例对象，例如在我们首次访问http://localhost:8080/helloservlet/helloworld时，服务器通过“/helloworld”找到了绑定的Servlet名称为cn.itcast.servlet.HelloServlet，然后服务器查看这个类型的Servlet是否已经创建过，如果没有创建过，那么服务器才会通过反射来创建HelloServlet的实例。当我们再次访问http://localhost:8080/helloservlet/helloworld时，服务器就不会再次创建HelloServlet实例了，而是直接使用上次创建的实例。
* 在Servlet被创建后，服务器会马上调用Servlet的void init(ServletConfig)方法。请记住， Servlet出生后马上就会调用init()方法，而且一个Servlet的一生。这个方法只会被调用一次。
* 我们可以把一些对Servlet的初始化工作放到init方法中！

## Servlet服务

* 当服务器每次接收到请求时，都会去调用Servlet的service()方法来处理请求。服务器接收到一次请求，就会调用service() 方法一次，所以service()方法是会被调用多次的。正因为如此，所以我们才需要把处理请求的代码写到service()方法中！

## Servlet的销毁

* Servlet是不会轻易离去的，通常都是在服务器关闭时Servlet才会离去！在服务器被关闭时，服务器会去销毁Servlet，在销毁Servlet之前服务器会先去调用Servlet的destroy()方法，我们可以把Servlet的临终遗言放到destroy()方法中，例如对某些资源的释放等代码放到destroy()方法中。
* Servlet接口相关类型
* 在Servlet接口中还存在三个我们不熟悉的类型：  
  - ServletRequest：service() 方法的参数，它表示请求对象，它封装了所有与请求相关的数据，它是由服务器创建的；  
  - ServletResponse：service()方法的参数，它表示响应对象，在service()方法中完成对客户端的响应需要使用这个对象；  
  - ServletConfig：init()方法的参数，它表示Servlet配置对象，它对应Servlet的配置
* ServletRequest和ServletResponse
* ServletRequest和ServletResponse是Servlet#service() 方法的两个参数，一个是请求对象，一个是响应对象，可以从ServletRequest对象中获取请求数据，可以使用ServletResponse对象完成响应。
* ServletRequest和ServletResponse的实例由服务器创建，然后传递给service()方法。如果在service() 方法中希望使用HTTP相关的功能，那么可以把ServletRequest和ServletResponse强转成HttpServletRequest和HttpServletResponse。这也说明我们经常需要在service()方法中对ServletRequest和ServletResponse进行强转，这是很心烦的事情。不过后面会有一个类来帮我们解决这一问题的。
* HttpServletRequest方法：  
  - String getParameter(String paramName)：获取指定请求参数的值；  
  - String getMethod()：获取请求方法，例如GET或POST；  
  - String getHeader(String name)：获取指定请求头的值；  
  - void setCharacterEncoding(String encoding)：设置请求体的编码！
* 因为GET请求没有请求体，所以这个方法只对POST请求有效。当调用
* request.setCharacterEncoding(“utf-8”)之后，再通过getParameter()方法获取参数值时，那么参数值都已经通过了转码，即转换成了UTF-8编码。所以，这个方法必须在调用getParameter()方法之前调用！
* HttpServletResponse方法：  
  - PrintWriter getWriter()：获取字符响应流，使用该流可以向客户端输出响应信息。例如response.getWriter().print(“Hello JavaWeb!”)；  
  - ServletOutputStream getOutputStream()：获取字节响应流，当需要向客户端响应字节数据时，需要使用这个流，例如要向客户端响应图片；  
  - void setCharacterEncoding(String encoding)：用来设置字符响应流的编码，例如在调用setCharacterEncoding(“utf-8”);之后，再response.getWriter()获取字符响应流对象，这时的响应流的编码为utf-8，使用response.getWriter()输出的中文都会转换成utf-8编码后发送给客户端；  
  - void setHeader(String name, String value)：向客户端添加响应头信息，
* 例如setHeader(“Refresh”, “3;url=http://www.itcast.cn”)，表示3秒后自动刷新到http://www.itcast.cn；  
  - void setContentType(String contentType)：该方法是setHeader(“content-type”, “xxx”)的简便方法，即用来添加名为content-type响应头的方法。content-type响应头用来设置响应数据的MIME类型，例如要向客户端响应jpg的图片，那么可以setContentType(“image/jepg”)，如果响应数据为文本类型，那么还要同时设置编码，例如setContentType(“text/html;chartset=utf-8”)表示响应数据类型为文本类型中的html类型，并且该方法会调用setCharacterEncoding(“utf-8”)方法；  
  - void sendError(int code, String errorMsg)：向客户端发送状态码，以及错误消息。例如给客户端发送404：response(404, “您要查找的资源不存在！”)。
* ServletConfig
* Servlet的配置信息，即web.xml文件中的元素。
* 一个ServletConfig对象对应着一个servlet元素的配置信息（servlet-name，servlet-class)
* getServletName获取的是的功能
* getServletContext获取的是Servlet上下文对象
* ServletConfig对象对应web.xml文件中的元素。例如你想获取当前Servlet在web.xml文件中的配置名，那么可以使用servletConfig.getServletName()方法获取！
* ServletConfig对象是由服务器创建的，然后传递给Servlet的init()方法，你可以在init()方法中使用它！  
  - String getServletName()：获取Servlet在web.xml文件中的配置名称，即指定的名称；  
  - ServletContext getServletContext()：用来获取ServletContext对象；  
  - String getInitParameter(String name)：用来获取在web.xml中配置的初始化参数，通过参数名来获取参数值；  
  - Enumeration getInitParameterNames()：用来获取在web.xml中配置的所有初始化参数名称；
* 在元素中还可以配置初始化参数：  
    
   One  
   cn.itcast.servlet.OneServlet  
     
   paramName1  
   paramValue1  
     
     
   paramName2  
   paramValue2
* 在OneServlet中，可以使用ServletConfig对象的getInitParameter()方法来获取初始化参数，例如：
* String value1 = servletConfig.getInitParameter(“paramName1”);//获取到paramValue1

# 5.4 实现Servlet的方式

* 实现Servlet有三种方式：  
  - 实现javax.servlet.Servlet接口；  
  - 继承javax.servlet.GenericServlet类；  
  - 继承javax.servlet.http.HttpServlet类；
* 通常我们会去继承HttpServlet类来完成我们的Servlet
* GenericServlet
* GenericServlet概述
* GenericServlet是Servlet接口的实现类，我们可以通过继承GenericServlet来编写自己的Servlet。下面是GenericServlet类的源代码：
* GenericServlet.java  
  public abstract class GenericServlet implements Servlet, ServletConfig,  
   java.io.Serializable {  
   private static final long serialVersionUID = 1L;  
   private transient ServletConfig config;  
   public GenericServlet() {}  
   @Override  
   public void destroy() {}  
   @Override  
   public String getInitParameter(String name) {  
   return getServletConfig().getInitParameter(name);  
   }  
   @Override  
   public Enumeration getInitParameterNames() {  
   return getServletConfig().getInitParameterNames();  
   }  
   @Override  
   public ServletConfig getServletConfig() {  
   return config;  
   }  
   @Override  
   public ServletContext getServletContext() {  
   return getServletConfig().getServletContext();  
   }  
   @Override  
   public String getServletInfo() {  
   return "";  
   }  
   @Override  
   public void init(ServletConfig config) throws ServletException {  
   this.config = config;  
   this.init();  
   }  
  public void init() throws ServletException {}  
  这个init方法是为了拓展init而设置的，子类可以重写这个无参数的init方法  
   public void log(String msg) {  
   getServletContext().log(getServletName() + ": " + msg);  
   }  
   public void log(String message, Throwable t) {  
   getServletContext().log(getServletName() + ": " + message, t);  
   }  
   @Override  
   public abstract void service(ServletRequest req, ServletResponse res)  
   throws ServletException, IOException;  
   @Override  
   public String getServletName() {  
   return config.getServletName();  
   }  
  }
* GenericServlet的init()方法
* 在GenericServlet中，定义了一个ServletConfig实例变量，并在init(ServletConfig)方法中把参数ServletConfig赋给了实例变量。然后在该类的很多方法中使用了实例变量config。
* 如果子类覆盖了GenericServlet的init(StringConfig)方法，那么this.config=config这一条语句就会被覆盖了，也就是说GenericServlet的实例变量config的值为null，那么所有依赖config的方法都不能使用了。如果真的希望完成一些初始化操作，那么去覆盖GenericServlet提供的init()方法，它是没有参数的init()方法，它会在init(ServletConfig)方法中被调用。
* 实现了ServletConfig接口
* GenericServlet还实现了ServletConfig接口，所以可以直接调用getInitParameter()、getServletContext()等ServletConfig的方法。
* HttpServlet
* HttpServlet概述
* HttpServlet类是GenericServlet的子类，它提供了对HTTP请求的特殊支持，所以通常我们都会通过继承HttpServlet来完成自定义的Servlet。
* HttpServlet覆盖了service()方法
* HttpServlet类中提供了service(HttpServletRequest,HttpServletResponse)方法，这个方法是HttpServlet自己的方法，不是从Servlet继承来的。在HttpServlet的service(ServletRequest,ServletResponse)方法中会把ServletRequest和ServletResponse强转成HttpServletRequest和HttpServletResponse，然后调用
* service(HttpServletRequest,HttpServletResponse)方法，这说明子类可以去覆盖service(HttpServletRequest,HttpServletResponse)方法即可，这就不用自己去强转请求和响应对象了。
* 其实子类也不用去覆盖service(HttpServletRequest,HttpServletResponse)方法，因为HttpServlet还要做另一步简化操作，下面会介绍。
* HttpServlet.java  
  public abstract class HttpServlet extends GenericServlet {  
   protected void service(HttpServletRequest req, HttpServletResponse resp)  
   throws ServletException, IOException {  
   ……  
  }  
   @Override  
   public void service(ServletRequest req, ServletResponse res)  
   throws ServletException, IOException {
* HttpServletRequest request;  
   HttpServletResponse response;  
    
   try {  
   request = (HttpServletRequest) req;  
   response = (HttpServletResponse) res;  
   } catch (ClassCastException e) {  
   throw new ServletException("non-HTTP request or response");  
   }  
   service(request, response);
* }  
  ……  
  }
* doGet()和doPost()
* 在HttpServlet的service(HttpServletRequest,HttpServletResponse)方法会去判断当前请求是GET还是POST，如果是GET请求，那么会去调用本类的doGet()方法，如果是POST请求会去调用doPost()方法，这说明我们在子类中去覆盖doGet()或doPost()方法即可。  
  public class AServlet extends HttpServlet {  
  public void doGet(HttpServletRequest request, HttpServletResponse response)  
   throws ServletException, IOException {  
   System.out.println("hello doGet()...");  
  }  
  }  
  public class BServlet extends HttpServlet {  
  public void doPost(HttpServletRequest request, HttpServletResponse response)  
   throws ServletException, IOException {  
   System.out.println("hello doPost()...");  
  }  
  }
* Servlet细节  
  - 不要在Servlet中创建成员！创建局部变量即可！  
  - 可以创建无状态成员！  
  - 可以创建有状态的成员，但状态必须为只读的！（只有get，没有set)
* Servlet与线程安全
* 因为一个类型的Servlet只有一个实例对象，那么就有可能会现时出一个Servlet同时处理多个请求，那么Servlet是否为线程安全的呢？答案是：“不是线程安全的”。这说明Servlet的工作效率很高，但也存在线程安全问题！
* 所以我们不应该在Servlet中创建成员变量，因为可能会存在一个线程对这个成员变量进行写操作，另一个线程对这个成员变量进行读操作。
* 让服务器在启动时就创建Servlet（很少这么做)
* 默认情况下，服务器会在某个Servlet第一次收到请求时创建它。也可以在web.xml中对Servlet进行配置，使服务器启动时就创建Servlet。  
    
   hello1  
   cn.itcast.servlet.Hello1Servlet  
   0  
    
    
   hello1  
   /hello1  
    
    
   hello2  
   cn.itcast.servlet.Hello2Servlet  
   1  
    
    
   hello2  
   /hello2  
    
    
   hello3  
   cn.itcast.servlet.Hello3Servlet  
   2  
    
    
   hello3  
   /hello3
* 在元素中配置元素可以让服务器在启动时就创建该Servlet，其中元素的值必须是大于等于的整数，它的使用是服务器启动时创建Servlet的顺序。上例中，根据的值可以得知服务器创建Servlet的顺序为Hello1Servlet、Hello2Servlet、Hello3Servlet。
* 是的子元素，用来指定Servlet的访问路径，即URL。
* 它必须是以“/”开头！  
  1)- 可以在中给出多个，例如：  
    
   AServlet  
   /AServlet  
   /BServlet
* 那么这说明一个Servlet绑定了两个URL，无论访问/AServlet还是/BServlet，访问的都是AServlet。

2)- 还可以在中使用通配符，所谓通配符就是星号“\*”，星号可以匹配任何URL前缀或后缀，使用通配符可以命名一个Servlet绑定一组URL，例如：  


<url-pattern>/servlet/\*<url-patter>：/servlet/a、/servlet/b，都匹配/servlet/\*；

- *.do：/abc/def/ghi.do、/a.do，都匹配*.do；  


<url-pattern>/\*<url-pattern>：匹配所有URL；

请注意，通配符要么为前缀，要么为后缀，不能出现在URL中间位置，也不能只有通配符。例如：/\*.do就是错误的，因为星号出现在URL的中间位置上了。\*.\*也是不对的，因为一个URL中最多只能出现一个通配符。

* 注意，通配符是一种模糊匹配URL的方式，如果存在更具体的，那么访问路径会去匹配具体的。例如：  
    
   hello1  
   cn.itcast.servlet.Hello1Servlet  
    
    
   hello1  
   /servlet/hello1  
    
    
   hello2  
   cn.itcast.servlet.Hello2Servlet  
    
    
   hello2  
   /servlet/\*
* 当访问路径为http://localhost:8080/hello/servlet/hello1时，因为访问路径即匹配hello1的，又匹配hello2的，但因为hello1的中没有通配符，所以优先匹配，即设置hello1。
* web.xml文件的继承
* 在${CATALINA\_HOME}\conf\web.xml中的内容，相当于写到了每个项目的web.xml中，它是所有web.xml的父文件。
* 每个完整的JavaWeb应用中都需要有web.xml，但我们不知道所有的web.xml文件都有一个共同的父文件，它在Tomcat的conf/web.xml路径。
* conf/web.xml

default  
 org.apache.catalina.servlets.DefaultServlet  
   
 debug  
 0  
   
   
 listings  
 false  
   
 1

<servlet>  
 <servlet-name>jsp</servlet-name>  
 <servlet-class>org.apache.jasper.servlet.JspServlet</servlet-class>  
 <init-param>  
 <param-name>fork</param-name>  
 <param-value>false</param-value>  
 </init-param>  
 <init-param>  
 <param-name>xpoweredBy</param-name>  
 <param-value>false</param-value>  
 </init-param>  
 <load-on-startup>3</load-on-startup>  
</servlet>  
  
<servlet-mapping>  
 <servlet-name>default</servlet-name>  
 <url-pattern>/</url-pattern>  
</servlet-mapping>  
  
<servlet-mapping>  
 <servlet-name>jsp</servlet-name>  
 <url-pattern>\*.jsp</url-pattern>  
 <url-pattern>\*.jspx</url-pattern>  
</servlet-mapping>  
  
<session-config>  
 <session-timeout>30</session-timeout>  
</session-config>  
  
<!-- 这里省略了大概4000多行的MIME类型的定义,这里只给出两种MIME类型的定义 -->  
<mime-mapping>  
 <extension>bmp</extension>  
 <mime-type>image/bmp</mime-type>  
</mime-mapping>  
<mime-mapping>  
 <extension>htm</extension>  
 <mime-type>text/html</mime-type>  
</mime-mapping>  
  
<welcome-file-list>  
 <welcome-file>index.html</welcome-file>  
 <welcome-file>index.htm</welcome-file>  
 <welcome-file>index.jsp</welcome-file>  
</welcome-file-list>

* ServletContext（存取数据，获取资源)
* 一个项目只有一个ServletContext对象！
* 我们可以在N多个Servlet中来获取这个唯一的对象，使用它可以给多个Servlet传递数据！
* 这个对象在Tomcat启动时就创建，在Tomcat关闭时才会死去！
* ServletContext概述
* 服务器会为每个应用创建一个ServletContext对象：  
  - ServletContext对象的创建是在服务器启动时完成的；  
  - ServletContext对象的销毁是在服务器关闭时完成的。
* ServletContext对象的作用是在整个Web应用的动态资源之间共享数据！例如在AServlet中向ServletContext对象中保存一个值，然后在BServlet中就可以获取这个值，这就是共享数据了。
* 获取ServletContext  
  - ServletConfig#getServletContext()；  
  - GenericServlet#getServletContext();  
  - HttpSession#getServletContext()  
  - ServletContextEvent#getServletContext()
* 在Servlet中获取ServletContext对象：  
  - 在void init(ServletConfig config)中：ServletContext context = config.getServletContext();，ServletConfig类的getServletContext()方法可以用来获取ServletContext对象；
* 在GenericeServlet或HttpServlet中获取ServletContext对象：  
  - GenericServlet类有getServletContext()方法，所以可以直接使用
* this.getServletContext()来获取；
* public class MyServlet implements Servlet {  
  public void init(ServletConfig config) {  
  ServletContext context = config.getServletContext();  
  }  
  …  
  }  
  public class MyServlet extends HttpServlet {  
  public void doGet(HttpServletRequest request, HttpServletResponse response) {  
  ServletContext context = this.getServletContext();  
  }  
  }
* 域对象的功能
* ServletContext是JavaWeb四大域对象之一：  
  - PageContext；  
  - ServletRequest；  
  - HttpSession；  
  - ServletContext；
* 所有域对象都有存取数据的功能，因为域对象内部有一个Map，用来存储数据，下面是ServletContext对象用来操作数据的方法：  
  - void setAttribute(String name, Object value)：用来存储一个对象，也可以称之为存储一个域属性，例如：servletContext.setAttribute(“xxx”, “XXX”)，在ServletContext中保存了一个域属性，域属性名称为xxx，域属性的值为XXX。请注意，如果多次调用该方法，并且使用相同的name，那么会覆盖上一次的值，这一特性与Map相同；  
  - Object getAttribute(String name)：用来获取ServletContext中的数据，当前在获取之前需要先去存储才行，例如：String value = (String)servletContext.getAttribute(“xxx”);，获取名为xxx的域属性；  
  - void removeAttribute(String name)：用来移除ServletContext中的域属性，如果参数name指定的域属性不存在，那么本方法什么都不做；  
  - Enumeration getAttributeNames()：获取所有域属性的名称；
* 获取应用初始化参数  
  - Servlet也可以获取初始化参数，但它是局部的参数；也就是说，一个Servlet只能获取自己的初始化参数，不能获取别人的，即初始化参数只为一个Servlet准备！  
  - 可以配置公共的初始化参数，为所有Servlet而用！这需要使用ServletContext才能使用！
* 还可以使用ServletContext来获取在web.xml文件中配置的应用初始化参数！注意，应用初始化参数与Servlet初始化参数不同：
* web.xml  
  <web-app ...>  
  ...  
  （为ServletContext设置的公共初始化参数)  
  paramName1  
  paramValue1   
    
    
  paramName2  
  paramValue2   
    
    
   ServletContext context = this.getServletContext();  
   String value1 = context.getInitParameter("paramName1");  
   String value2 = context.getInitParameter("paramName2");  
   System.out.println(value1 + ", " + value2);
* Enumeration names = context.getInitParameterNames();  
  while(names.hasMoreElements()) {  
   System.out.println(names.nextElement());  
  }

# 5.5 请求&响应

* 服务器每次收到请求时，都会为这个请求开辟一个新的线程。
* Response
* 1、概述
* response是Servlet#service方法的一个参数，类型为
* javax.servlet.http.HttpServletResponse。在客户端发出每个请求时，服务器都会创建一个response对象，并传入给Servlet.service()方法。response对象是用来对客户端进行响应的，这说明在service()方法中使用response对象可以完成对客户端的响应工作。
* response对象的功能分为以下四种：  
  - 设置响应头信息；  
  - 发送状态码；  
  - 设置响应正文；  
  - 重定向。
* 2、响应正文
* response是响应对象，向客户端输出响应正文（响应体)可以使用response的响应流，repsonse一共提供了两个响应流对象：  
  - PrintWriter out = response.getWriter()：获取字符流；  
  - ServletOutputStream out = response.getOutputStream()：获取字节流；
* 当然，如果响应正文内容为字符（html)，那么使用response.getWriter()，如果响应内容是字节（图片等)，例如下载时，那么可以使用response.getOutputStream()
* (ServletOutputStream)。
* 注意，在一个请求中，不能同时使用这两个流！也就是说，要么你使用repsonse.getWriter()，要么使用response.getOutputStream()，但不能同时使用这两个流。不然会抛出IllegalStateException异常。
* 字符响应流  
  - 字符编码
* 在使用response.getWriter()时需要注意默认字符编码为ISO-8859-1，如果希望设置字符流的字符编码为utf-8，可以使用response.setCharaceterEncoding(“utf-8”)来设置。这样可以保证输出给客户端的字符都是使用UTF-8编码的！
* 但客户端浏览器并不知道响应数据是什么编码的！如果希望通知客户端使用UTF-8来解读响应数据，那么还是使用response.setContentType("text/html;charset=utf-8")方法比较好，因为这个方法不只会调用response.setCharaceterEncoding(“utf-8”)，还会设置content-type响应头，客户端浏览器会使用content-type头来解读响应数据。  
  - 缓冲区
* response.getWriter()是PrintWriter类型，所以它有缓冲区，缓冲区的默认大小为8KB。也就是说，在响应数据没有输出8KB之前，数据都是存放在缓冲区中，而不会立刻发送到客户端。当Servlet执行结束后，服务器才会去刷新流，使缓冲区中的数据发送到客户端。
* 如果希望响应数据马上发送给客户端：  
  - 向流中写入大于8KB的数据；  
  - 调用response.flushBuffer()方法来手动刷新缓冲区；
* 字节响应流
* 将一张图片转为字节流写入到response中
* 读取图片，使用commons-io包的方法。
* byte[] image = IOUtils.toByteArray(new FileInputStream(this.getServletContext().getRealPath("/images/cat.jpeg")));
* response.getOutputStream().write(image);
* 3、设置响应头信息
* 响应头是键值对
* 可以使用response对象的setHeader()方法来设置响应头！使用该方法设置的响应头最终会发送给客户端浏览器！  
  - response.setHeader(“content-type”, “text/html;charset=utf-8”)：设置content-type响应头，该头的作用是告诉浏览器响应内容为html类型，编码为utf-8。而且同时会设置response的字符流编码为utf-8，即response.setCharaceterEncoding(“utf-8”)；  
  - response.setHeader("Refresh","5; URL=http://www.itcast.cn")：5秒后自动跳转到传智主页。
* 4、设置状态码及其他方法  
  - response.setContentType("text/html;charset=utf-8")：等同\于调用
* response.setHeader(“content-type”, “text/html;charset=utf-8”)；  
  - response.setCharacterEncoding(“utf-8”)：设置字符响应流的字符编码为utf-8；   
  - response.setStatus(200)：设置状态码；  
  - response.sendError(404, “您要查找的资源不存在”)：当发送错误状态码时，Tomcat会跳转到固定的错误页面去，但可以显示错误信息。
* response.sendError(状态码)
* 5、重定向
  + 1)什么是重定向
* 当你访问http://www.sun.com时，你会发现浏览器地址栏中的URL会变成http://www.oracle.com/us/sun/index.htm，这就是重定向了。
* 重定向是服务器通知浏览器去访问另一个地址，即再发出另一个请求。
  + 2)完成重定向
* 响应码为200表示响应成功，而响应码为302表示重定向。所以完成重定向的第一步就是设置响应码为302。
* 因为重定向是通知浏览器发出第二个请求，所以浏览器需要知道第二个请求的URL，所以完成重定向的第二步是设置Location头，指定第二个请求的URL地址。

public class AServlet extends HttpServlet {  
 public void doGet(HttpServletRequest request, HttpServletResponse response)  
 throws ServletException, IOException {  
 response.setStatus(302);  
 response.setHeader("Location", "http://www.itcast.cn");  
 }  
}

* 上面代码的作用是：当访问AServlet后，会通知浏览器重定向到传智主页。客户端浏览器解析到响应码为302后，就知道服务器让它重定向，所以它会马上获取响应头Location，然发出第二个请求。
  + 3)便捷的重定向方式sendRedirect  
    public class AServlet extends HttpServlet {  
    public void doGet(HttpServletRequest request, HttpServletResponse response)  
     throws ServletException, IOException {  
    response.sendRedirect("http://www.itcast.cn");  
    }  
    }
* response.sendRedirect()方法会设置响应头为302，以设置Location响应头。
* 如果要重定向的URL是在同一个服务器内，那么可以使用相对路径，例如：  
  public class AServlet extends HttpServlet {  
  public void doGet(HttpServletRequest request, HttpServletResponse response)  
   throws ServletException, IOException {  
   response.sendRedirect("/hello/BServlet");  
   注意是/项目名/路径（请求URI)  
  }  
  }
* 重定向的URL地址为：http://localhost:8080/hello/BServlet
  + 4)重定向小结  
    - 重定向是两次请求；  
    - 重定向的URL可以是其他应用，不局限于当前应用；  
    - 重定向的响应头为302，并且必须要有Location响应头；  
    - 重定向就不要再使用response.getWriter()或response.getOutputStream()输出数据，不然可能会出现异常；
* Request
* 1、概述
* request是Servlet.service()方法的一个参数，类型为javax.servlet.http.HttpServletRequest。在客户端发出每个请求时，服务器都会创建一个request对象，并把请求数据封装到request中，然后在调用Servlet.service()方法时传递给service()方法，这说明在service()方法中可以通过request对象来获取请求数据。
* request的功能可以分为以下几种：  
  - 封装了请求头数据；  
  - 封装了请求正文数据，如果是GET请求，那么就没有正文；  
  - request是一个域对象，可以把它当成Map来添加获取数据；  
  - request提供了请求转发和请求包含功能。
* 2、域方法
* request是域对象！在JavaWeb中一共四个域对象，其中ServletContext就是域对象，它在整个应用中只创建一个ServletContext对象。request其中一个，request可以在一个请求中共享数据。
* 一个请求会创建一个request对象，如果在一个请求中经历了多个Servlet，那么多个Servlet就可以使用request来共享数据。
* 下面是request的域方法：  
  - void setAttribute(String name, Object value)：用来存储一个对象，也可以称之为存储一个域属性，例如：servletContext.setAttribute(“xxx”, “XXX”)，在request中保存了一个域属性，域属性名称为xxx，域属性的值为XXX。请注意，如果多次调用该方法，并且使用相同的name，那么会覆盖上一次的值，这一特性与Map相同；  
  - Object getAttribute(String name)：用来获取request中的数据，当前在获取之前需要先去存储才行，例如：String value = (String)request.getAttribute(“xxx”);，获取名为xxx的域属性；  
  - void removeAttribute(String name)：用来移除request中的域属性，如果参数name指定的域属性不存在，那么本方法什么都不做；  
  - Enumeration getAttributeNames()：获取所有域属性的名称；
* 3、获取请求头数据
* request与请求头相关的方法有：  
  - String getHeader(String name)：获取指定名称的请求头；  
  - Enumeration getHeaderNames()：获取多值头；  
  - int getIntHeader(String name)：获取值为int类型的请求头。  
  - long getDateHeader(String name)：获取值为long类型的请求头
* 4、获取请求相关的其它方法
* request中还提供了与请求相关的其他方法，有些方法是为了我们更加便捷的方法请求头数据而设计，有些是与请求URL相关的方法。  
  - int getContentLength()：获取请求体的字节数，GET请求没有请求体，没有请求体返回-1；  
  - String getContentType()：获取请求类型，如果请求是GET，那么这个方法返回null；如果是POST请求，那么默认为application/x-www-form-urlencoded，表示请求体内容使用了URL编码；  
  - String getMethod()：返回请求方法，例如：GET/POST  
  - Locale getLocale()：返回当前客户端浏览器的Locale。java.util.Locale表示国家和言语，这个东西在国际化中很有用；  
  - String getCharacterEncoding()：获取请求编码，如果没有setCharacterEncoding()，那么返回null，表示使用ISO-8859-1编码；  
  - void setCharacterEncoding(String code)：设置请求编码，只对请求体有效！注意，对于GET而言，没有请求体！！！所以此方法只能对POST请求中的参数有效！  
  - String getContextPath()：返回上下文路径（/项目名)，例如：/hello  
  - String getQueryString()：返回请求URL中的参数，例如：name=zhangSan  
  - String getRequestURI()：返回请求URI路径，例如：/hello/oneServlet  
  - StringBuffer getRequestURL()：返回请求URL路径，例如：
* http://localhost/hello/oneServlet，即返回除了参数以外的路径信息；  
  - String getServletPath()：返回Servlet路径，例如：/oneServlet  
  - String getRemoteAddr()：返回当前客户端的IP地址；  
  - String getRemoteHost()：返回当前客户端的主机名，但这个方法的实现还是获取IP地址；  
  - String getScheme()：返回请求协议，例如：http；  
  - String getServerName()：返回主机名，例如：localhost  
  - int getServerPort()：返回服务器端口号，例如：8080
* System.out.println("IP:"+request.getRemoteAddr());
* System.out.println("请求方式:"+request.getMethod());
* System.out.println("User-Agent请求头:"+request.getHeader("User-Agent"));
* System.out.println("协议名:"+request.getScheme());
* System.out.println("主机名:"+request.getServerName());
* System.out.println("端口号:"+request.getServerPort());
* System.out.println("项目:"+request.getContextPath());
* System.out.println("Servlet路径:"+request.getServletPath());
* System.out.println("请求参数:"+request.getQueryString());
* System.out.println("URI:"+request.getRequestURI());
* System.out.println("URL:"+request.getRequestURL());
* =========================================================================
* http://localhost:8080/Request&Response/RequestHeaderServlet?username=sxj&password=sxj
* ==========================================================================
* IP:127.0.0.1
* 请求方式:GET
  + User-Agent请求头:Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/55.0.2883.87 Safari/537.36
* 协议名:http
* 主机名:localhost
* 端口号:8080
* 项目:/Request&Response
* Servlet路径:/RequestHeaderServlet
* 请求参数: username=sxj&password=sxj
* URI:/Request&Response/RequestHeaderServlet（RequestURI)
* URL:http://localhost:8080/Request&Response/RequestHeaderServlet
* 整个访问的路径就等于URL+QueryString
* URL=协议名+主机名+端口号+URI
* 案例：request.getRemoteAddr()：封IP
* 可以使用request.getRemoteAddr()方法获取客户端的IP地址，然后判断IP是否为禁用IP。  
  String ip = request.getRemoteAddr();  
   System.out.println(ip);  
   if(ip.equals("127.0.0.1")) {  
   response.getWriter().print("您的IP已被禁止！");  
   } else {  
   response.getWriter().print("Hello!");  
   }
* 案例：防盗链
* 可以使用request.getAttribute(“Referer”)如果不是当前页面，那么属于盗链，则跳转到当前页面
* 如果是从地址栏直接输入URL，那么Referee返回的是null
* String referer = request.getHeader("Referer");
* System.out.println(referer);
* if(referer == null || !referer.contains("localhost")){
* response.sendRedirect("http://www.baidu.com");
* }else{
* response.getWriter().print("hello");//如果是从指定页面跳转来的
* }
* 5、获取请求参数
* 最为常见的客户端传递参数方式有两种：  
  - 浏览器地址栏直接输入：一定是GET请求；  
  - 超链接：一定是GET请求；  
  - 表单：可以是GET，也可以是POST，这取决与的method属性值；
* GET请求和POST请求的区别：  
  - GET请求：  
  - 请求参数会在浏览器的地址栏中显示，所以不安全；  
  - 请求参数长度限制长度在1K之内；  
  - GET请求没有请求体，无法通过request.setCharacterEncoding()来设置参数的编码；  
  - POST请求：  
  - 请求参数不会显示浏览器的地址栏，相对安全；  
  - 请求参数长度没有限制；
* 无论是GET|POST请求，都可以使用相同的API来获取请求参数。
* 请求参数有一个key一个value的，也有一个key多个value的。
* 超链接
* 下面是使用request获取请求参数的API：  
  - String getParameter(String name)：通过指定名称获取参数值；
* public void doGet(HttpServletRequest request, HttpServletResponse response)  
   throws ServletException, IOException {  
   String v1 = request.getParameter("p1");  
   String v2 = request.getParameter("p2");  
   System.out.println("p1=" + v1);  
   System.out.println("p2=" + v2);  
  }
* public void doPost(HttpServletRequest request, HttpServletResponse response)  
   throws ServletException, IOException {  
   String v1 = request.getParameter("p1");  
   String v2 = request.getParameter("p2");  
   System.out.println("p1=" + v1);  
   System.out.println("p2=" + v2);   
  }

- String[] getParameterValues(String name)：当多个参数名称相同时，可以使用方法来获取；  
超链接  
 public void doGet(HttpServletRequest request, HttpServletResponse response)  
 throws ServletException, IOException {  
 String[] names = request.getParameterValues("name");  
 System.out.println(Arrays.toString(names));  
 }  
- Enumeration getParameterNames()：获取所有参数的名字；  
   
 参数1：<br/>  
 参数2：<br/>  
   
   
 public void doPost(HttpServletRequest request, HttpServletResponse response)  
 throws ServletException, IOException {  
 Enumeration names = request.getParameterNames();  
 while(names.hasMoreElements()) {  
 System.out.println(names.nextElement());  
 }  
 }

- Map getParameterMap()：获取所有参数封装到Map中，其中key为参数名，value为参数值，因为一个参数名称可能有多个值，所以参数值是String[]，而不是String。  
超链接  
 Map<String,String[]> paramMap = request.getParameterMap();  
 for(String name : paramMap.keySet()) {  
 String[] values = paramMap.get(name);  
 System.out.println(name + ": " + Arrays.toString(values));  
 }  
p2: [v2, vv2]  
p1: [v1, vv1]

* 示例：
* 点击这里，测试GET请求<br>
* 用户名：<br>
* 密 码：<br>
* 爱 好：吃饭<br>
* 睡觉<br>
* 编程<br>
* //遍历Map

public class RequestParameterServlet extends HttpServlet {

public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

* Enumeration names = request.getParameterNames();
* while(names.hasMoreElements()){
* String name = names.nextElement();
* System.out.println(name+"="+request.getParameter(name));
* }
* }

public void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

* Map<String,String[]> map = request.getParameterMap();
* for(Entry<String,String[]> entry:map.entrySet()){
* System.out.println(entry.getKey()+"="+Arrays.toString(entry.getValue()));
* }
* }
* }
* username=sxj
* password=sxj
* username=[23]
* password=[123]
* hobby=[eat, sleep]
* String username = request.getParameter("username");
* String password = request.getParameter("password");
* String [] hobbies = request.getParameterValues("hobby");
* System.out.println("username:"+username);
* System.out.println("password:"+password);
* System.out.println("hobbies:"+Arrays.toString(hobbies));
* 6、请求转发和请求包含
* 无论是请求转发还是请求包含，都表示由多个Servlet共同来处理一个请求。例如Servlet1来处理请求，然后Servlet1又转发给Servlet2来继续处理这个请求。
  + 1)请求转发（常用)
* 在AServlet中，把请求转发到BServlet：
* 参数是Servlet路径（servlet-mapping中的url-pattern)
* 相当于/项目名  
  public class AServlet extends HttpServlet {  
  public void doGet(HttpServletRequest request, HttpServletResponse response)  
   throws ServletException, IOException {  
   System.out.println("AServlet");  
   RequestDispatcher rd = request.getRequestDispatcher("/BServlet");  
   rd.forward(request, response);  
  }  
  }  
  public class BServlet extends HttpServlet {  
  public void doGet(HttpServletRequest request, HttpServletResponse response)  
   throws ServletException, IOException {  
   System.out.println("BServlet");  
  }  
  }  
  Aservlet  
  BServlet
* 如果转发了，那么在原本请求的Servlet中设置的响应体是失效的。
* 如果转发前，在原本请求的Servlet中设置的响应体超过24K，那么响应体仍有效（缓冲区溢出)。
  + 2)请求包含（不常用)
* 在AServlet中，把请求包含到BServlet：  
  public class AServlet extends HttpServlet {  
  public void doGet(HttpServletRequest request, HttpServletResponse response)  
   throws ServletException, IOException {  
   System.out.println("AServlet");  
   RequestDispatcher rd = request.getRequestDispatcher("/BServlet");  
   rd.include(request, response);  
  }  
  }  
  public class BServlet extends HttpServlet {  
  public void doGet(HttpServletRequest request, HttpServletResponse response)  
   throws ServletException, IOException {  
   System.out.println("BServlet");  
  }  
  }  
  Aservlet  
  BServlet
  + 3)请求转发与请求包含比较  
    - 如果在AServlet中请求转发到BServlet，那么在AServlet中就不允许再输出响应体，即不能再使用response.getWriter()和response.getOutputStream()向客户端输出，这一工作应该由BServlet来完成；如果是使用请求包含，那么没有这个限制；  
    - 请求转发虽然不能输出响应体，但还是可以设置响应头的，例如：response.setContentType(”text/html;charset=utf-8”);  
    - 请求包含大多是应用在JSP页面中，完成多页面的合并；  
    - 请求转发大多是应用在Servlet中，转发目标大多是JSP页面；
  + 4)请求转发与重定向比较  
     - 1)请求转发是一个请求，而重定向是两个请求；  
    - 请求转发后浏览器地址栏不会有变化，而重定向会有变化，因为重定向是两个请求；  
     - 2)请求转发的目标只能是本应用中的资源（给出Servlet路径)，重定向的目标可以是其他应用（请求URI或者其他路径)；  
     - 3)请求转发对AServlet和BServlet的请求方法是相同的，即要么都是GET，要么都是POST，因为请求转发是一个请求；  
    - 重定向的第二个请求一定是GET；  
     - 4)请求转发效率更高；当需要地址栏发生变化时，需要使用重定向；需要在下一个Servlet中获取之前Servlet设置的域，需要使用转发

# Filter

* 什么是过滤器
* 过滤器JavaWeb三大组件之一，它与Servlet很相似！不过过滤器是用来拦截请求的，而不是处理请求的。
* 当用户请求某个Servlet时，会先执行部署在这个请求上的Filter，如果Filter“放行”，那么会继承执行用户请求的Servlet；如果Filter不“放行”，那么就不会执行用户请求的Servlet。
* 其实可以这样理解，当用户请求某个Servlet时，Tomcat会去执行注册在这个请求上的Filter，然后是否“放行”由Filter来决定。可以理解为，Filter来决定是否调用Servlet！当执行完成Servlet的代码后，还会执行Filter后面的代码。
* 过滤器之hello world
* 其实过滤器与Servlet很相似，我们回忆一下如果写的第一个Servlet应用！写一个类，实现Servlet接口！没错，写过滤器就是写一个类，实现Filter接口。  
  public class HelloFilter implements Filter {  
  public void init(FilterConfig filterConfig) throws ServletException {}  
  public void doFilter(ServletRequest request, ServletResponse response,  
   FilterChain chain) throws IOException, ServletException {  
   System.out.println("Hello Filter");  
  }  
  public void destroy() {}  
  }
* 第二步也与Servlet一样，在web.xml文件中部署Filter：  
    
   helloFilter  
   cn.itcast.filter.HelloFilter  
    
    
   helloFilter  
   /*(/*表示拦截所有的访问请求)
* 当用户访问index.jsp页面时，会执行HelloFilter的doFilter()方法！在我们的示例中，index.jsp页面是不会被执行的，如果想执行index.jsp页面，那么我们需要放行！  
  public void doFilter(ServletRequest request, ServletResponse response,  
   FilterChain chain) throws IOException, ServletException {  
   System.out.println("filter start...");  
   chain.doFilter(request, response);  
   System.out.println("filter end...");  
  }
* 有很多同学总是错误的认为，一个请求在给客户端输出之后就算是结束了，这是不对的！其实很多事情都需要在给客户端响应之后才能完成！
* 过滤器的生命周期  
  - init(FilterConfig)：在服务器启动时会创建Filter实例，并且每个类型的Filter只创建一个实例，从此不再创建！在创建完Filter实例后，会马上调用init()方法完成初始化工作，这个方法只会被执行一次；  
  - doFilter(ServletRequest req,ServletResponse res,FilterChain chain)：这个方法会在用户每次访问“目标资源（pattern>index.jsp)”时执行，如果需要“放行”，那么需要调用FilterChain的doFilter(ServletRequest,ServletResponse)方法，如果不调用FilterChain的doFilter()方法，那么目标资源将无法执行；  
  - destroy()：服务器会在创建Filter对象之后，把Filter放到缓存中一直使用，通常不会销毁它。一般会在服务器关闭时销毁Filter对象，在销毁Filter对象之前，服务器会调用Filter对象的destory()方法。  
  - 单例，同Servlet一致
* FilterConfig
* Filter接口中的init()方法的参数类型为FilterConfig类型。它的功能与ServletConfig相似，与web.xml文件中的配置信息对应。下面是FilterConfig的功能介绍：  
  - ServletContext getServletContext()：获取ServletContext的方法；  
  - String getFilterName()：获取Filter的配置名称；与元素对应；  
  - String getInitParameter(String name)：获取Filter的初始化配置，与元素对应；  
  - Enumeration getInitParameterNames()：获取所有初始化参数的名称。
* FilterChain
* doFilter()方法的参数中有一个类型为FilterChain的参数，它只有一个方法：doFilter(ServletRequest,ServletResponse)。
* 前面我们说doFilter()方法的放行，让请求流访问目标资源！但这么说不严谨，其实调用该方法的意思是，“我（当前Filter)”放行了，但不代表其他人（其他过滤器)也放行。
* 如果当前过滤器是最后一个过滤器，那么调用chain.doFilter()方法表示执行目标资源，而不是最后一个过滤器，那么chain.doFilter()表示执行下一个过滤器的doFilter()方法。
* 多个过滤器执行顺序
* 一个目标资源可以指定多个过滤器，过滤器的执行顺序是在web.xml文件中的部署顺序：  
    
   myFilter1  
   cn.itcast.filter.MyFilter1  
    
    
   myFilter1  
   /index.jsp  
    
    
   myFilter2  
   cn.itcast.filter.MyFilter2  
    
    
   myFilter2  
   /index.jsp  
    
  public class MyFilter1 extends HttpFilter {  
  public void doFilter(HttpServletRequest request, HttpServletResponse response,  
   FilterChain chain) throws IOException, ServletException {  
   System.out.println("filter1 start...");  
   chain.doFilter(request, response);//放行，执行MyFilter2的doFilter()方法  
   System.out.println("filter1 end...");  
  }  
  }  
  public class MyFilter2 extends HttpFilter {  
  public void doFilter(HttpServletRequest request, HttpServletResponse response,  
   FilterChain chain) throws IOException, ServletException {  
   System.out.println("filter2 start...");  
   chain.doFilter(request, response);//放行，执行目标资源  
   System.out.println("filter2 end...");  
  }  
  }  
    
   This is my JSP page. <br>
* <%System.out.println("index.jsp"); %>
* 当有用户访问index.jsp页面时，输出结果如下：  
  filter1 start...  
  filter2 start...  
  index.jsp  
  filter2 end...  
  filter1 end...
* AFilter B Filter CFilter
* 有点像是递归，调用filterchain的doFilter方法会放行给下一个过滤器，这个doFilter前面的代码的执行顺序是A-B-C，而后面的代码的执行顺序是C-B-A
* 四种拦截方式
* 我们来做个测试，写一个过滤器，指定过滤的资源为b.jsp，然后我们在浏览器中直接访问b.jsp，你会发现过滤器执行了！
* 但是，当我们在a.jsp中request.getRequestDispathcer(“/b.jsp”).forward(request,response)时，就不会再执行过滤器了！也就是说，默认情况下，只能直接访问目标资源才会执行过滤器，而forward执行目标资源，不会执行过滤器！  
  public class MyFilter extends HttpFilter {  
  public void doFilter(HttpServletRequest request,  
   HttpServletResponse response, FilterChain chain)  
   throws IOException, ServletException {  
   System.out.println("myfilter...");  
   chain.doFilter(request, response);  
  }  
  }   
    
   myfilter  
   cn.itcast.filter.MyFilter  
    
    
   myfilter  
   /b.jsp
* <%  
   request.getRequestDispatcher("/b.jsp").forward(request, response);  
   %>
* http://localhost:8080/filtertest/b.jsp -->直接访问b.jsp时，会执行过滤器内容；
* http://localhost:8080/filtertest/a.jsp --> 访问a.jsp，但a.jsp会forward到b.jsp，这时就不会执行过滤器！
* 其实过滤器有四种拦截方式！分别是：REQUEST、FORWARD、INCLUDE、ERROR。  
  - REQUEST：直接访问目标资源时执行过滤器。包括：在地址栏中直接访问、表单提交、超链接、重定向，只要在地址栏中可以看到目标资源的路径，就是REQUEST；  
  - FORWARD：转发访问执行过滤器。包括RequestDispatcher#forward()方法、<jsp:forward>标签都是转发访问；  
  - INCLUDE：包含访问执行过滤器。包括RequestDispatcher#include()方法、<jsp:include>标签都是包含访问；  
  - ERROR：当目标资源在web.xml中配置为中时，并且真的出现了异常，转发到目标资源时，会执行过滤器。
* 可以在中添加0~n个子元素，来说明当前访问的拦截方式。
* 如果没有写dispatcher，默认为REQUEST  
    
   myfilter  
   /b.jsp  
   REQUEST  
   FORWARD  
    
    
   myfilter  
   /b.jsp  
    
    
   myfilter  
   /b.jsp  
   FORWARD
* 其实最为常用的就是REQUEST和FORWARD两种拦截方式，而INCLUDE和ERROR都比较少用！下面给出ERROR拦截方式的例子：  
    
   myfilter  
   /b.jsp  
   ERROR  
    
    
   500  
   /b.jsp
* <%  
   if(true)  
   throw new RuntimeException("嘻嘻~");  
   %>
* 过滤器的应用场景
* 过滤器的应用场景：  
  - 执行目标资源之前做预处理工作，例如设置编码，这种通常都会放行，只是在目标资源执行之前做一些准备工作；
* 几乎是所有的Sevlet中都需要写request.setCharacterEndoing() 可以把它放入到一个Filter中  
  - 通过条件判断是否放行，例如校验当前用户是否已经登录，或者用户IP是否已经被禁用；  
  - 在目标资源执行后，做一些后续的特殊处理工作，例如把目标资源输出的数据进行处理；
* 设置目标资源
* 在web.xml文件中部署Filter时，可以通过“*”来执行目标资源：*  
    
   *myfilter*  
   */*

这一特性与Servlet完全相同！通过这一特性，我们可以在用户访问敏感资源时，执行过滤器，例如：<url-pattern>/admin/\*<url-pattern>，可以把所有管理员才能访问的资源放到/admin路径下，这时可以通过过滤器来校验用户身份。

* 还可以为指定目标资源为某个Servlet，例如：  
    
   myservlet  
   cn.itcast.servlet.MyServlet  
    
    
   myservlet  
   /abc  
    
    
   myfilter  
   cn.itcast.filter.MyFilter  
    
    
   myfilter  
   myservlet（可以有多个)
* 当用户访问http://localhost:8080/filtertest/abc时，会执行名字为myservlet的Servlet，这时会执行过滤器。
* Filter小结
* Filter的三个方法：  
  - void init(FilterConfig)：在Tomcat启动时被调用；  
  - void destroy()：在Tomcat关闭时被调用；  
  - void doFilter(ServletRequest,ServletResponse,FilterChain)：每次有请求时都调用该方法；
* FilterConfig类：与ServletConfig相似，用来获取Filter的初始化参数  
  - ServletContext getServletContext()：获取ServletContext的方法；  
  - String getFilterName()：获取Filter的配置名称；   
  - String getInitParameter(String name)：获取Filter的初始化配置，与元素对应；  
  - Enumeration getInitParameterNames()：获取所有初始化参数的名称。
* FilterChain类：  
  - void doFilter(ServletRequest,ServletResponse)：放行！表示执行下一个过滤器，或者执行目标资源。可以在调用FilterChain的doFilter()方法的前后添加语句，在FilterChain的doFilter()方法之前的语句会在目标资源执行之前执行，在FilterChain的doFilter()方法之后的语句会在目标资源执行之后执行。
* 四种拦截方式：REQUEST、FORWARD、INCLUDE、ERROR，默认是REQUEST方式。  
  - REQUEST：拦截直接请求方式；  
  - FORWARD：拦截请求转发方式；  
  - INCLUDE：拦截请求包含方式；  
  - ERROR：拦截错误转发方式。

# Listener

# 5.6 1、监听器概述

* 在JavaWeb被监听的事件源为：ServletContext、HttpSession、ServletRequest，即三大域对象。  
  - 监听域对象“创建”与“销毁”的监听器；  
  - 监听域对象“操作域属性”的监听器；  
  - 监听HttpSession的监听器。
* 2、创建与销毁监听器
* 创建与销毁监听器一共有三个：  
  - ServletContextListener：Tomcat启动和关闭时调用下面两个方法  
  

public void contextInitialized(ServletContextEvent evt)：ServletContext对象被创建后调用；



public void contextDestroyed(ServletContextEvent evt)：ServletContext对象被销毁前调用；

- HttpSessionListener：开始会话和结束会话时调用下面两个方法  


public void sessionCreated(HttpSessionEvent evt)：HttpSession对象被创建后调用；



public void sessionDestroyed(HttpSessionEvent evt)：HttpSession对象被销毁前调用；

- ServletRequestListener：开始请求和结束请求时调用下面两个方法  


public void requestInitiallized(ServletRequestEvent evt)：ServletRequest对象被创建后调用；



public void requestDestroyed(ServletRequestEvent evt)：ServletRequest对象被销毁前调用。

* 事件对象  
  - ServletContextEvent：ServletContext getServletContext()；  
  - HttpSessionEvent：HttpSession getSession()；  
  - ServletRequestEvent：  
  - ServletRequest getServletRequest()  
  - ServletContext getServletContext()
* 编写测试例子：  
  - 编写MyServletContextListener类，实现ServletContextListener接口；  
  - 在web.xml文件中部署监听器；  
  - 为了看到session销毁的效果，在web.xml文件中设置session失效时间为1分钟；

/\*

* ServletContextListener实现类
* contextDestroyed() -- 在ServletContext对象被销毁前调用
* contextInitialized() -- -- 在ServletContext对象被创建后调用
* ServletContextEvent -- 事件类对象
* 该类有getServletContext()，用来获取ServletContext对象，即获取事件源对象  
   \*/  
  public class MyServletContextListener implements ServletContextListener {  
  public void contextDestroyed(ServletContextEvent evt) {  
   System.out.println("销毁ServletContext对象");  
  }
* public void contextInitialized(ServletContextEvent evt) {  
   System.out.println("创建ServletContext对象");  
  }  
  }  
  /\*
* HttpSessionListener实现类
* sessionCreated() -- 在HttpSession对象被创建后被调用
* sessionDestroyed() -- -- 在HttpSession对象被销毁前调用
* HttpSessionEvent -- 事件类对象
* 该类有getSession()，用来获取当前HttpSession对象，即获取事件源对象  
   \*/  
  public class MyHttpSessionListener implements HttpSessionListener {  
  public void sessionCreated(HttpSessionEvent evt) {  
   System.out.println("创建session对象");  
  }
* public void sessionDestroyed(HttpSessionEvent evt) {  
   System.out.println("销毁session对象");  
  }  
  }

/\*

* ServletRequestListener实现类
* requestDestroyed() -- 在ServletRequest对象被销毁前调用
* requestInitialized() -- 在ServletRequest对象被创建后调用
* ServletRequestEvent -- 事件类对象
* 该类有getServletContext()，用来获取ServletContext对象
* 该类有getServletRequest()，用来获取当前ServletRequest对象，即事件源对象  
   \*/  
  public class MyServletRequestListener implements ServletRequestListener {  
  public void requestDestroyed(ServletRequestEvent evt) {  
   System.out.println("销毁request对象");  
  }
* public void requestInitialized(ServletRequestEvent evt) {  
   System.out.println("创建request对象");  
  }  
  }

cn.itcast.listener.MyServletContextListener  
  
  
cn.itcast.listener.MyHttpSessionListener  
  
  
cn.itcast.listener.MyServletRequestListener  
  
  
 1

* 3、操作域属性的监听器
* 当对域属性进行增、删、改时，执行的监听器一共有三个：  
  - ServletContextAttributeListener：在ServletContext域进行增、删、改属性时调用下面方法。  
  

public void attributeAdded(ServletContextAttributeEvent evt)



public void attributeRemoved(ServletContextAttributeEvent evt)



public void attributeReplaced(ServletContextAttributeEvent evt)

- HttpSessionAttributeListener：在HttpSession域进行增、删、改属性时调用下面方法  


public void attributeAdded(HttpSessionBindingEvent evt)



public void attributeRemoved (HttpSessionBindingEvent evt)



public void attributeReplaced (HttpSessionBindingEvent evt)

- ServletRequestAttributeListener：在ServletRequest域进行增、删、改属性时调用下面方法  


public void attributeAdded(ServletRequestAttributeEvent evt)



public void attributeRemoved (ServletRequestAttributeEvent evt)



public void attributeReplaced (ServletRequestAttributeEvent evt)

* 下面对这三个监听器的事件对象功能进行介绍：  
  - ServletContextAttributeEvent  
  - String getName()：获取当前操作的属性名；  
  - Object getValue()：获取当前操作的属性值；  
  - ServletContext getServletContext()：获取ServletContext对象。  
  - HttpSessionBindingEvent  
  - String getName()：获取当前操作的属性名；  
  - Object getValue()：获取当前操作的属性值；  
  - HttpSession getSession()：获取当前操作的session对象。  
  - ServletRequestAttributeEvent  
  - String getName()：获取当前操作的属性名；  
  - Object getValue()：获取当前操作的属性值；  
  - ServletContext getServletContext()：获取ServletContext对象；  
  - ServletRequest getServletRequest()：获取当前操作的ServletRequest对象。
* 如果是替换，那么getValue返回的是原值；如果想拿到新值，那么去对应的域对象中取。
* 4、HttpSession的监听器
* 还有两个与HttpSession相关的特殊的监听器，这两个监听器的特点如下：  
  - 不用在web.xml文件中部署；  
  - 这两个监听器不是给session添加，而是给Bean添加。即让Bean类实现监听器接口，然后再把Bean对象添加到session域中。
* 下面对这两个监听器介绍一下：  
  - HttpSessionBindingListener：当某个类实现了该接口后，可以感知本类对象添加到session中，以及感知从session中移除。例如让Person类实现HttpSessionBindingListener接口，那么当把Person对象添加到session中，或者把Person对象从session中移除时会调用下面两个方法：  
  

public void valueBound(HttpSessionBindingEvent event)：当把监听器对象添加到session中会调用监听器对象的本方法；



public void valueUnbound(HttpSessionBindingEvent event)：当把监听器对象从session中移除时会调用监听器对象的本方法；

* 这里要注意，HttpSessionBindingListener监听器的使用与前面介绍的都不相同，当该监听器对象添加到session中，或把该监听器对象从session移除时会调用监听器中的方法。并且无需在web.xml文件中部署这个监听器。
* 示例步骤：  
  - 编写Person类，让其实现HttpSessionBindingListener监听器接口；  
  - 编写Servlet类，一个方法向session中添加Person对象，另一个从session中移除Person对象；  
  - 在index.jsp中给出两个超链接，分别访问Servlet中的两个方法。
* Pseron.java  
  public class Person implements HttpSessionBindingListener {  
  private String name;  
  private int age;  
  private String sex;
* public Person(String name, int age, String sex) {  
   super();  
   this.name = name;  
   this.age = age;  
   this.sex = sex;  
  }
* public Person() {  
   super();  
  }
* public String toString() {  
   return "Person [name=" + name + ", age=" + age + ", sex=" + sex + "]";  
  }
* public String getName() {  
   return name;  
  }
* public void setName(String name) {  
   this.name = name;  
  }
* public int getAge() {  
   return age;  
  }
* public void setAge(int age) {  
   this.age = age;  
  }
* public String getSex() {  
   return sex;  
  }
* public void setSex(String sex) {  
   this.sex = sex;  
  }
* public void valueBound(HttpSessionBindingEvent evt) {  
   System.out.println("把Person对象存放到session中：" + evt.getValue());  
  }
* public void valueUnbound(HttpSessionBindingEvent evt) {  
   System.out.println("从session中移除Pseron对象：" + evt.getValue());  
  }  
  }
* ListenerServlet.java  
  public class ListenerServlet extends BaseServlet {  
  public String addPerson(HttpServletRequest request, HttpServletResponse response)  
   throws ServletException, IOException {  
   Person p = new Person("zhangSan", 23, "male");  
   request.getSession().setAttribute("person", p);  
   return "/index.jsp";  
  }
* public String removePerson(HttpServletRequest request, HttpServletResponse response)  
   throws ServletException, IOException {  
   request.getSession().removeAttribute("person");  
   return "/index.jsp";  
  }
* index.jsp  
    
   addPerson  
   <br/>  
   removePerson  
   <br/>
* Session有一种重生的性质。
* 在服务器开启时保存了session，然后当服务器被关闭时，会在硬盘上保存一个session属性的序列化文件。
* 启动服务器时会重新将该序列化文件读入内存，并删除该文件。
* 在conf/context.xml中进行一些配置可以取消保存session序列化文件的操作。  
  - HttpSessionActivationListener：Tomcat会在session不被使用时钝化session对象，所谓钝化session，就是把session通过序列化的方式保存到硬盘文件中。当用户再使用session时，Tomcat还会把钝化的对象再活化session，所谓活化就是把硬盘文件中的session在反序列化回内存。当session被Tomcat钝化时，session中存储的对象也被钝化，当session被活化时，也会把session中存储的对象活化。如果某个类实现了HttpSessionActiveationListener接口后，当对象随着session被钝化和活化时，下面两个方法就会被调用：  
  

public void sessionWillPassivate(HttpSessionEvent se)：当对象感知被活化时调用本方法；



public void sessionDidActivate(HttpSessionEvent se)：当对象感知被钝化时调用本方法；

* HttpSessionActivationListener监听器与HttpSessionBindingListener监听器相似，都是感知型的监听器，例如让Person类实现了HttpSessionActivationListener监听器接口，并把Person对象添加到了session中后，当Tomcat钝化session时，同时也会钝化session中的Person对象，这时Person对象就会感知到自己被钝化了，其实就是调用Person对象的sessionWillPassivate()方法。当用户再次使用session时，Tomcat会活化session，这时Person会感知到自己被活化，其实就是调用Person对象的sessionDidActivate()方法。
* 注意JavaBean同时需要实现Serializable接口。
* 注意，因为钝化和活化session，其实就是使用序列化和反序列化技术把session从内存保存到硬盘，和把session从硬盘加载到内存。这说明如果Person类没有实现Serializable接口，那么当session钝化时就不会钝化Person，而是把Person从session中移除再钝化！这也说明session活化后，session中就不再有Person对象了。

# Cookie(服务器创建，客户端保存)

* Cookie如果我不想js或http读，应该设置什么属性？里面有个属性httpOnly，这个属性你了解吗？

# 5.7 1、Cookie概述

* 1.1 什么叫Cookie
* Cookie翻译成中文是小甜点，小饼干的意思。在HTTP中它表示服务器送给客户端浏览器的小甜点。由HTTP协议制定。其实Cookie就是一个键和一个值构成的，随着服务器端的响应发送给客户端浏览器。然后客户端浏览器会把Cookie保存起来，当下一次再访问服务器时把Cookie再发送给服务器。
* Cookie是由服务器创建，然后通过响应发送给客户端的一个键值对。客户端会保存Cookie，并会标注出Cookie的来源（哪个服务器的Cookie)。当客户端向服务器发出请求时会把所有这个服务器Cookie包含在请求中发送给服务器，这样服务器就可以识别客户端了！
* 1.2 Cookie规范  
  - Cookie大小上限为4KB；  
  - 一个服务器最多在客户端浏览器上保存20个Cookie；  
  - 一个浏览器最多保存300个Cookie；
* 上面的数据只是HTTP的Cookie规范，但在浏览器大战的今天，一些浏览器为了打败对手，为了展现自己的能力起见，可能对Cookie规范“扩展”了一些，例如每个Cookie的大小为8KB，最多可保存500个Cookie等！但也不会出现把你硬盘占满的可能！
* 注意，不同浏览器之间是不共享Cookie的。也就是说在你使用IE访问服务器时，服务器会把Cookie发给IE，然后由IE保存起来，当你在使用FireFox访问服务器时，不可能把IE保存的Cookie发送给服务器。
* 1.3 Cookie与HTTP头
* Cookie是通过HTTP请求头和响应头在客户端和服务器端传递的：  
  - Cookie：请求头，客户端发送给服务器端；  
  - 格式：Cookie: a=A; b=B; c=C。即多个Cookie用分号分开；
* 一个Cookie对应多个键值对  
  - Set-Cookie：响应头，服务器端发送给客户端；  
  - 一个Cookie对象一个Set-Cookie，一个键值对
* Set-Cookie: a=A
* Set-Cookie: b=B
* Set-Cookie: c=C
* 比如可以用response.addHeader(“Set-Cookie”,”a=A”);发送一个Cookie
* 便捷的使用Cookie方式：
* HttpServletRequest:
* 没有返回null
* HttpServletResponse:
* 1.4 Cookie的覆盖
* 如果服务器端发送重复的Cookie那么会覆盖原有的Cookie，例如客户端的第一个请求服务器端发送的Cookie是：Set-Cookie: a=A；第二请求服务器端发送的是：Set-Cookie: a=AA，那么客户端只留下一个Cookie，即：a=AA。
* 2、Cookie的生命周期
* Cookie不只是有name和value，Cookie还是生命。所谓生命就是Cookie在客户端的有效时间，可以通过setMaxAge(int)来设置Cookie的有效时间。  
   - cookie.setMaxAge(-1)：cookie的maxAge属性的默认值就是-1，表示只在浏览器内存中存活。一旦关闭浏览器窗口，那么cookie就会消失。  
  - cookie.setMaxAge(60\*60)：表示cookie对象可存活1小时。当生命大于0时，浏览器会把Cookie保存到硬盘上，就算关闭浏览器，就算重启客户端电脑，cookie也会存活1小时；  
  - cookie.setMaxAge(0)：cookie生命等于0是一个特殊的值，它表示cookie被作废！也就是说，如果原来浏览器已经保存了这个Cookie，那么可以通过Cookie的setMaxAge(0)来删除这个Cookie。无论是在浏览器内存中，还是在客户端硬盘上都会删除这个Cookie。
* 3、Cookie的path（服务器的路径)
* 3.1 什么是Cookie的路径
* 现在有WEB应用A，向客户端发送了10个Cookie，这就说明客户端无论访问应用A的哪个Servlet都会把这10个Cookie包含在请求中！但是也许只有AServlet需要读取请求中的Cookie，而其他Servlet根本就不会获取请求中的Cookie。这说明客户端浏览器有时发送这些Cookie是多余的！
* 可以通过设置Cookie的path来指定浏览器，在访问什么样的路径时，包含什么样的Cookie。
* 3.2 Cookie路径与请求路径的关系
* 下面我们来看看Cookie路径的作用：
* 下面是客户端浏览器保存的3个Cookie的路径：
* a: /cookietest；
* b: /cookietest/servlet；
* c: /cookietest/jsp；
* 下面是浏览器请求的URL：
* A: http://localhost:8080/cookietest/AServlet；
* B: http://localhost:8080/cookietest/servlet/BServlet；
* C: http://localhost:8080/cookietest/jsp/a.jsp；

- 请求A时，会在请求中包含a；  
- 请求B时，会在请求中包含a、b；  
- 请求C时，会在请求中包含a、c；

* 请求路径如果包含了Cookie路径，那么会在请求中包含这个Cookie。
* A请求的URL包含了“/cookietest”，所以会在请求中包含路径为“/cookietest”的Cookie；  
  - B请求的URL包含了“/cookietest”，以及“/cookietest/servlet”，所以请求中包含路径为“/cookietest”和“/cookietest/servlet”两个Cookie；  
  - C请求的URL包含了“/cookietest”，以及“/cookietest/jsp”，所以请求中包含路径为“/cookietest”和“/cookietest/jsp”两个Cookie；
* 3.3 设置Cookie的路径
* 设置Cookie的路径需要使用setPath()方法，例如：
* cookie.setPath(“/cookietest/servlet”);
* 默认路径是访问资源的上一级路径
* 如果没有设置Cookie的路径，那么Cookie路径的默认值当前访问资源所在路径（上一级)，例如：  
  - 访问http://localhost:8080/cookietest/AServlet时添加的Cookie默认路径为/cookietest；  
  - 访问http://localhost:8080/cookietest/servlet/BServlet时添加的Cookie默认路径为/cookietest/servlet；  
  - 访问http://localhost:8080/cookietest/jsp/BServlet时添加的Cookie默认路径为/cookietest/jsp；
* 4、Cookie的domain
* Cookie的domain属性可以让网站中二级域共享Cookie，次要！
* 百度你是了解的对吧！
* http://www.baidu.com
* http://zhidao.baidu.com
* http://news.baidu.com
* http://tieba.baidu.com
* 现在我希望在这些主机之间共享Cookie（例如在www.baidu.com中响应的cookie，可以在news.baidu.com请求中包含)。很明显，现在不是路径的问题了，而是主机的问题，即域名的问题。处理这一问题其实很简单，只需要下面两步：  
  - 设置Cookie的path为“/”：c.setPath(“/”)；  
  - 设置Cookie的domain为“.baidu.com”：c.setDomain(“.baidu.com”)。
* 5、Cookie中保存中文
* Cookie的name和value都不能使用中文，如果希望在Cookie中使用中文，那么需要先对中文进行URL编码，然后把编码后的字符串放到Cookie中。
* 向客户端响应中添加Cookie  
  String name = URLEncoder.encode("姓名", "UTF-8");  
   String value = URLEncoder.encode("张三", "UTF-8");  
   Cookie c = new Cookie(name, value);  
   c.setMaxAge(3600);  
   response.addCookie(c);
* 从客户端请求中获取Cookie  
  response.setContentType("text/html;charset=utf-8");  
   Cookie[] cs = request.getCookies();  
   if(cs != null) {  
   for(Cookie c : cs) {  
   String name = URLDecoder.decode(c.getName(), "UTF-8");  
   String value = URLDecoder.decode(c.getValue(), "UTF-8");  
   String s = name + ": " + value + "<br/>";  
   response.getWriter().print(s);  
   }  
   }

# 5.8 6、Cookie中的属性

## HttpOnly

* 如果Cookie中设置了HttpOnly属性，那么通过js脚本将无法读取到cookie信息，这样能有效的防止XSS攻击，窃取cookie内容，这样就增加了cookie的安全性，即便是这样，也不要将重要信息存入cookie。
* Servlet3.0支持setHttpOnly(boolean httpOnly)。
* 非servlet3.0的JAVAEE项目也可以通过设置Header进行设置：
* response.setHeader("Set-Cookie", "cookiename=value; Path=/;Domain=domainvalue;Max-Age=seconds;HTTPOnly");

## max-age

* expires/Max-Age 字段为此cookie超时时间。若设置其值为一个时间，那么当到达此时间后，此cookie失效。不设置的话默认值是Session，意思是cookie会和session一起失效。当浏览器关闭(不是浏览器标签页，而是整个浏览器) 后，此cookie失效。

## secure

* 设置是否只能通过https来传递此条cookie
* Session(服务器创建，服务器保存)
* session默认过期时间，过长会怎么样：30min，过长占用内存
* 如何防止session被别人伪造cookie得到：Cookie的值后面跟上一段防篡改的验证串。防篡改的验证串可以由DES(cookie的内容+盐值)，也可以用 MD5（cookie内容+密钥)，。也可以是SHA1（cookie内容+密钥)，这里的密钥只有站点本身知道，如果这个都泄漏了那就真完蛋了。这个值在服务器接收到Cookie以后，就可以用Cookie的内容+密钥重新计算一次验证串，和提交上来的做比对，如果是一致的，我们就认为cookie没有被篡改，反之，cookie肯定被篡改过，我们就不要相信这一次提交。如果所有的Cookie都经过防篡改验证，那么也就不用担心SessionID被冒名顶替的事情发生了。
* 如果服务器的负载量很高内存负荷不住要怎么办：尽量规避使用，优先以Cookie（包括加密Cookie)来解决；Session要自己实现分布式储存，比如使用 redis 等做储存。
* 1、HttpSession概述（属于JavaWeb)
* 1.1 什么是HttpSesssion
* javax.servlet.http.HttpSession接口表示一个会话，我们可以把一个会话内需要共享的数据保存到HttSession对象中！
* 会话：会话范围是某个用户从首次访问服务器开始，到该用户关闭浏览器结束。
* 1.2 获取HttpSession对象  
  - HttpSession request.getSesssion()：如果当前会话已经有了session对象那么直接返回，如果当前会话还不存在会话，那么创建session并返回；  
  - HttpSession request.getSession(boolean)：当参数为true时，与requeset.getSession()相同。如果参数为false，那么如果当前会话中存在session则返回，不存在返回null；  
  - JSP中session是一个内置对象，可以直接使用
* 1.3 HttpSession是域对象
* 我们已经学习过HttpServletRequest、ServletContext，它们都是域对象，现在我们又学习了一个HttpSession，它也是域对象。它们三个是Servlet中可以使用的域对象，
* 而JSP中可以多使用一个域对象PageContext。
* HttpServletRequest：一个请求创建一个request对象，所以在同一个请求中可以共享request，例如一个请求从AServlet转发到BServlet，那么AServlet和BServlet可以共享request域中的数据；  
  - ServletContext：一个应用只创建一个ServletContext对象，所以在ServletContext中的数据可以在整个应用中共享，只要不关闭服务器，那么ServletContext中的数据就可以共享；  
  - HttpSession：一个会话创建一个HttpSession对象，同一会话中的多个请求中可以共享session中的数据；
* 下面是session的域方法：  
  - void setAttribute(String name, Object value)：用来存储一个对象，也可以称之为存储一个域属性，例如：session.setAttribute(“xxx”, “XXX”)，在session中保存了一个域属性，域属性名称为xxx，域属性的值为XXX。请注意，如果多次调用该方法，并且使用相同的name，那么会覆盖上一次的值，这一特性与Map相同；  
  - Object getAttribute(String name)：用来获取session中的数据，当前在获取之前需要先去存储才行，例如：String value = (String) session.getAttribute(“xxx”);，获取名为xxx的域属性；  
  - void removeAttribute(String name)：用来移除HttpSession中的域属性，如果参数name指定的域属性不存在，那么本方法什么都不做；  
  - Enumeration getAttributeNames()：获取所有域属性的名称；
* 2、登录案例
* 需要的页面：  
  - login.jsp：登录页面，提供登录表单；  
  - succ1.jsp：主页，显示当前用户名称，如果没有登录，显示您还没登录；  
  - succ2.jsp：同上
* Servlet：  
  - LoginServlet：在login.jsp页面提交表单时，请求本Servlet。在本Servlet中获取用户名、密码进行校验，如果用户名、密码错误，显示“用户名或密码错误”，如果正确保存用户名session中，然后重定向到succ1.jsp；
* 保存用户名到session的目的是保存登录状态
* 当用户没有登录时访问succ1.jsp或succ2.jsp，保存错误信息“您还没有登录”，并转发给login.jsp。如果用户在login.jsp登录成功后到达succ1.jsp页面会显示当前用户名，而且不用再次登录去访问succ2.jsp也会显示用户名。因为多次请求在一个会话范围，succ1.jsp和succ2.jsp都会到session中获取用户名，session对象在一个会话中是相同的，所以都可以获取到用户名！
* 用户名的自动填充：如果曾经登录过（session中保存有username)，则将该username保存到Cookie中，发送给客户端。如果再次访问login.jsp，则将username自动填充。
* 注意要设置Cookie的生命周期
* 代码：
* login.jsp
* <%
* String username = "";
* Cookie[] cookies = request.getCookies();
* if (cookies != null) {
* for (Cookie c : cookies) {
* if (c.getName().equals("username")) {
* username = c.getValue();
* }
* }
* }
* %>
* username:<br>
* password:<br> <input
* type="submit" value="login">
* <%
* String status = (String) request.getAttribute("status");
* if (status != null) {
* out.print(status);
* }
* %>
* LoginServlet

public class LoginServlet extends HttpServlet {

public void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

* request.setCharacterEncoding("utf-8");
* String username = request.getParameter("username");
* String password = request.getParameter("password");
* if (username.equals("admin") && password.equals("admin")) {
* // 使用session保存信息，可以使用重定向，因为处于同一会话
* // 避免Servlet的路径出现在地址栏中
* //保存一个session的域
* request.getSession().setAttribute("username", username);
* //保存一个cookie
* Cookie cookie = new Cookie("username", username);
  + cookie.setMaxAge(60*60*24);//一天
* response.addCookie(cookie);
* response.sendRedirect("/Session/jsp/succ1.jsp");
* } else {
* // 使用request保存信息，只能使用转发，因为处于同一请求
* request.setAttribute("status", "用户名或密码错误");
* request.getRequestDispatcher("/jsp/login.jsp").forward(request, response);
* }
* }
* }
* succ1.jsp
* <%
* String username = (String)session.getAttribute("username");
* if(username != null){
* out.print("欢迎您,"+username+"!");
* }else{
* request.setAttribute("status", "您尚未登录");
* request.getRequestDispatcher("/jsp/login.jsp").forward(request, response);
* }
* %>
* 3、session的实现原理
* session底层是依赖Cookie的！我们来理解一下session的原理吧！
* 当首次使用session时，服务器端要创建session，session是保存在服务器端，而给客户端的session的id（一个cookie中保存了sessionId)。客户端带走的是sessionId，而数据是保存在session中。
* 当客户端再次访问服务器时，在请求中会带上sessionId，而服务器会通过sessionId找到对应的session，而无需再创建新的session。
* 如果Servlet中没有创建session，那么响应头中就不会有sessionid的cookie。但是所有的JSP页面中，session可以直接使用，也就是说每个JSP页面都会自动获取session，无论是否使用，访问JSP页面一定会带回来一个sessionid。
* 4、session与浏览器
* session保存在服务器，而sessionId通过Cookie发送给客户端，但这个Cookie的生命不是-1，即只在浏览器内存中存在，也就是说如果用户关闭了浏览器，那么这个Cookie就丢失了。
* 当用户再次打开浏览器访问服务器时，就不会有sessionId发送给服务器，那么服务器会认为你没有session，所以服务器会创建一个session，并在响应中把sessionId中到Cookie中发送给客户端。
* 你可能会说，那原来的session对象会怎样？当一个session长时间没人使用的话，服务器会把session删除了！这个时长在Tomcat中配置是30分钟，可以在${CATALANA}/conf/web.xml找到这个配置，当然你也可以在自己的web.xml中覆盖这个配置！
* web.xml  
    
   30
* session失效时间也说明一个问题！如果你打开网站的一个页面开始长时间不动，超出了30分钟后，再去点击链接或提交表单时你会发现，你的session已经丢失了！
* 5、session其他常用API  
  - String getId()：获取sessionId；32位长，16进制字符串，使用UUID生成随机字符串；  
  - int getMaxInactiveInterval()：获取session可以的最大不活动时间（秒)，默认为30分钟。当session在30分钟内没有使用，那么Tomcat会在session池中移除这个session；  
  - void setMaxInactiveInterval(int interval)：设置session允许的最大不活动时间（秒)，如果设置为1秒，那么只要session在1秒内不被使用，那么session就会被移除；  
  - long getCreationTime()：返回session的创建时间，返回值为当前时间的毫秒值；  
  - long getLastAccessedTime()：返回session的最后活动时间，返回值为当前时间的毫秒值；  
  - void invalidate()：让session失效！调用这个方法会被session失效，当session失效后，客户端再次请求，服务器会给客户端创建一个新的session，并在响应中给客户端新session的sessionId；（比如退出按钮，退出时让session失效)  
  - boolean isNew()：查看session是否为新。当客户端第一次请求时，服务器为客户端创建session，但这时服务器还没有响应客户端，也就是还没有把sessionId响应给客户端时，这时session的状态为新。请求中没有sessionid的Cookie
* request.getSession().isNew()可以判断这个Session是新的还是旧的。
* 6、URL重写
* 我们知道session依赖Cookie，那么session为什么依赖Cookie呢？因为服务器需要在每次请求中获取sessionId，然后找到客户端的session对象。那么如果客户端浏览器关闭了Cookie呢？那么session是不是就会不存在了呢？
* 其实还有一种方法让服务器收到的每个请求中都带有sessioinId，那就是URL重写！在每个页面中的每个链接和表单中都添加名为jSessionId的参数，值为当前sessionid。当用户点击链接或提交表单时也服务器可以通过获取jSessionId这个参数来得到客户端的sessionId，找到sessoin对象。
* index.jsp

主页

* 也可以使用response.encodeURL()对每个请求的URL处理，这个方法会自动追加jsessionid参数，与上面我们手动添加是一样的效果。  
  主页
* 使用response.encodeURL()更加“智能”，它会判断客户端浏览器是否禁用了Cookie，如果禁用了，那么这个方法在URL后面追加jsessionid，否则不会追加。

# AsyncServlet&Reactor

# 5.9 API

* Servlet异步处理就是让Servlet在处理费时的请求时不要阻塞，而是一部分一部分的显示。
* 也就是说，在使用Servlet异步处理之后，页面可以一部分一部分的显示数据，而不是一直卡，等到请求响应结束后一起显示。
* 在使用异步处理之前，一定要在@WebServlet注解中给出asyncSupported=true，不然默认Servlet是不支持异步处理的。如果存在过滤器，也要设置@WebFilter的asyncSupportedt=true。  
  @WebServlet(urlPatterns = {"/MyServlet"}, asyncSupported=true)  
  public class MyServlet extends HttpServlet {…}
* 注意，响应类型必须是text/html，所以：response.setContentType(“text/html;charset=utf-8”);
* 使用异步处理大致可以分为两步：  
  - Servlet正常响应数据；  
  - Servlet异常响应数据。
* 在Servlet正常响应数据时，没什么可说的，可通知response.getWriter().print()来向客户端输出，但输出后要使用response.getWriter().flush()刷新，不然数据只是在缓冲区中，不能向客户端发送数据的。
* 异步响应数据需要使用request.startAsync()方法获取AsyncContext对象。然后调用AsyncContext对象的start()方法启动异步响应，start()方法需要一个Runnable类型的参数。在Runnable的run()方法中给出异步响应的代码。  
  AsyncContext ac = request.startAsyncContext(request, response);  
  ac.start(new Runnable() {…});
* 注意在异步处理线程中使用response做响应后，要使用response.getWriter().flush()来刷新流，不然数据是不能响应到客户端浏览器的。
* asyncContext.start(new Runnable() {  
   public void run() {  
   for(char i = 'a'; i <= 'z'; i++) {  
   try {  
   Thread.sleep(100);  
   asyncContext.getResponse().getWriter().print(i + " ");  
   asyncContext.getResponse().getWriter().flush();  
   } catch (Exception e) {  
   e.printStackTrace();  
   }  
   }  
   asyncContext.complete();  
   }  
   });
* Tomcat需要知道异步响应是否结束，如果响应不结束，虽然客户端浏览器会看到响应的数据，但是鼠标上只是有个圈圈的不行的转啊转的，表示还没有结束响应。Tomcat会等待到超时为止，这个超时的时间可以通过AsyncContext类的getTimeout()方法获取，Tomcat默认为20000毫秒。当然也可以通过setTimeOut()方法设置，以毫秒为单位。ac.setTimeout(1000\*10)。
* 如果异步线程已经结束了响应，那么可以在异步线程中调用AsyncContext.complete()方法，这样Tomcat就知道异步线程已经完成了工作了。  
  @WebServlet(urlPatterns = {"/AServlet"}, asyncSupported=true)  
  public class AServlet extends HttpServlet {  
  public void doGet(HttpServletRequest request, HttpServletResponse response)  
   throws ServletException, IOException {  
   response.setContentType("text/html;charset=utf-8");  
   PrintWriter out = response.getWriter();  
   out.println("Servlet begin <br>");  
   out.println("Servlet begin <br>");  
   out.println("Servlet begin <br>");  
   out.println("Servlet begin <br>");  
   out.println("Servlet begin <br>");  
   out.println("Servlet begin <br>");  
   out.println("Servlet begin <br>");  
   out.println("Servlet begin <br>");  
   out.println("Servlet begin <br>");  
   out.println("Servlet begin <br>");  
   out.println("Servlet begin <br>");  
   out.println("Servlet begin <br>");  
   out.println("Servlet begin <br>");  
   out.println("Servlet begin <br>");  
   out.println("Servlet begin <br>");
* out.flush();  
  final AsyncContext asyncContext = request.startAsync(request, response);  
  asyncContext.setTimeout(1000 \* 20);  
  asyncContext.start(new Runnable() {  
   public void run() {  
   try {  
   Thread.sleep(1000);  
   asyncContext.getResponse().getWriter().print("马上开始" + "<br/>");  
   asyncContext.getResponse().getWriter().flush();  
   Thread.sleep(2000);  
   } catch (Exception e1) {  
   }  
   for(char i = 'a'; i <= 'z'; i++) {  
   try {  
   Thread.sleep(100);  
   asyncContext.getResponse().getWriter().print(i + "&nbsp;");  
   asyncContext.getResponse().getWriter().flush();  
   } catch (Exception e) {  
   e.printStackTrace();  
   }  
   }  
   asyncContext.complete();  
   }  
  });  
  // asyncContext.start(businessHandleThread);  
  // 也可以用这种方法启动异步线程  
  out.println("Servlet end <br>");
* }  
  }
* 模型
* 在引入Servlet3之前我们的线程模型是如下样子的：
* 整个请求解析、业务处理、生成响应都是由Tomcat线程池进行处理，而且都是在一个线程中处理；不能分离线程处理；比如接收到请求后交给其他线程处理，这样不能灵活定义业务处理模型。
* 引入Servlet3之后，我们的线程模型可以改造为如下样子：
* 此处可以看到请求解析使用Tomcat单线程；而解析完成后会扔到业务队列中，由业务线程池进行处理；这种处理方式可以得到如下好处：
* 1、根据业务重要性对业务进行分级，然后根据分级定义线程池；
* 2、可以拿到业务线程池，可以进行很多的操作，比如监控、降级等。
* 分级线程池
* 好处
* 更高的并发能力；
* 请求解析和业务处理线程池分离；
* 根据业务重要性对业务分级，并分级线程池；
* 对业务线程池进行监控、运维、降级等处理。
* 总结
* 通过异步化我们不会获得更快的响应时间，但是我们获得了整体吞吐量和我们需要的灵活性：请求解析和业务处理线程池分离；根据业务重要性对业务分级，并分级线程池；对业务线程池进行监控、运维、降级等处理。

# 5.10 与SpringWebFlux

* Servlet3.1 规范其中一个新特性是异步处理支持。
* 异步处理支持：Servlet 线程不需一直阻塞，即不需要到业务处理完毕再输出响应，然后结束 Servlet线程。异步处理的作用是在接收到请求之后，Servlet 线程可以将耗时的操作委派给另一个线程来完成，在不生成响应的情况下返回至容器。主要应用场景是针对业务处理较耗时的情况，可以减少服务器资源的占用，并且提高并发处理速度。
* 所以 WebFlux 支持的容器有 Tomcat、Jetty（Non-Blocking IO API) ，也可以像 Netty 和 Undertow 的本身就支持异步容器。在容器中 Spring WebFlux 会将输入流适配成 Mono 或者 Flux 格式进行统一处理。

# 5.11 Reactor

* Reactive Streams（以下简称为RS)是“一种规范，它为基于非阻塞回压的异步流处理提供了标准”。它是一组包含了TCK工具套件和四个简单接口（Publisher、Subscriber、Subscription和Processor)的规范，这些接口将被集成到Java 9.
* Reactor不同于其它框架的最关键一点就是RS。Flux和Mono这两者都是RS的Publisher实现，它们都具备了响应式回压的特点。
* Reactive Streams的主流实现有RxJava和Reactor，Spring WebFlux默认集成的是Reactor。

## Flux 和 Mono

* Flux 和 Mono 是 Reactor 中的两个基本概念。Flux 表示的是包含 0 到 N 个元素的异步序列。在该序列中可以包含三种不同类型的消息通知：正常的包含元素的消息、序列结束的消息和序列出错的消息。当消息通知产生时，订阅者中对应的方法 onNext(), onComplete()和 onError()会被调用。Mono 表示的是包含 0 或者 1 个元素的异步序列。该序列中同样可以包含与 Flux 相同的三种类型的消息通知。Flux 和 Mono 之间可以进行转换。对一个 Flux 序列进行计数操作，得到的结果是一个 Mono对象。把两个 Mono 序列合并在一起，得到的是一个 Flux 对象。

# JDBC

* JDBC的桥接设计模式 jdbc的详细链接过程；底层源码；在Java中调用存储过程的方法
* 为什么要使用JDBC
* JDBC可以跨数据库平台
* 系统可以 小规模时使用MySQL 规模变大时使用Oracle
* 如果不需要改动API函数，就需要分层。
* ODBC统一与数据库的接口，ADO是.NET统一与数据库的接口
* JDBC是java与数据库统一的接口
* 但是注意JDBC还是要求将SQL语句插入到代码中，而SQL语句各数据库有所不同，所以不能完全实现移植。如果要实现完全的移植，需要使用Hibernate技术
* Hibernate将不同数据库微小的区别也屏蔽掉了
* EJB也实现了屏蔽数据库间微小的区别
* JDBC对于java的接口是一致，但对于不同数据库JDBC所用的类库是不同的，对于程序猿而言是透明的。
* JDBC编程步骤
* 先需要找JDBC的类库 java.sql
* Driver 驱动，用来提供给JDBC连接到数据库
* java并不知道使用的是哪种数据库，而java有一个管理数据库的管家DriverManager，它来管理使用哪种数据库的连接。
* 连接到某个数据库，需要先向DriverManager注册。然后通过DriverManager连接到数据库。
* 代码示例：
* import java.sql.\*;

public class TestJDBC {

public static void main(String []args){

* Connection conn = null;
* Statement stmt = null;
* ResultSet rs = null;
* try {
* Class.forName("com.mysql.jdbc.Driver");
* conn = DriverManager.getConnection("jdbc:mysql://localhost/mydata?user=root&password=130119");
* stmt = conn.createStatement();
* rs = stmt.executeQuery("select \* from emp");
* while(rs.next()){
* System.out.println(rs.getString("deptno"));
* }
* } catch (SQLException | InstantiationException | IllegalAccessException | ClassNotFoundException e) {
* e.printStackTrace();
* }finally{
* try{
* if(rs != null){
* rs.close();
* rs = null;
* }
* if(stmt != null ){
* stmt.close();
* stmt= null;
* }
* if(conn != null){
* conn.close();
* conn = null;
* }
* }catch(SQLException e){
* e.printStackTrace();
* }
* }
* }
* }
* 总结：JDBC连接流程  
  1.- 注册：驱动实例化 Class forName()  
  2.- 连接：获取连接 Connection DriverManager getConnection()  
  3.- 执行：通过连接执行SQL语句 Statement createStatement() executeQuery()  
  4.- 接收并输出：循环遍历取出数据 ResultSet next() getString()  
  5.- 关闭：close 先打开的后关闭 close()
* 除此之外，还需要抓Exception，并将关闭的代码放到finally里面
* 关闭前先判断是否为null，如果为null那么关闭时会出错
* 在close中也会抛Exception，也要写try catch

# JDBCUtils

* import java.io.IOException;
* import java.util.Properties;
* import java.sql.\*;

public final class JDBCUtils {

private static String driver;

private static String url;

private static String username;

private static String password;

private JDBCUtils() {

* }
* static {
* Properties props = new Properties();
* try {
* props.load(JDBCUtils.class.getClassLoader().getResourceAsStream(
* "dbinfo.properties"));
* driver = props.getProperty("driver");
* url = props.getProperty("url");
* username = props.getProperty("username");
* password = props.getProperty("password");
* } catch (IOException e) {
* throw new ExceptionInInitializerError(e);
* }
* try {
* Class.forName(driver);
* } catch (ClassNotFoundException e) {
* throw new ExceptionInInitializerError(e);
* }
* }

public static Connection getConn() {

* Connection conn = null;
* try {
* conn = DriverManager.getConnection(url, username, password);
* } catch (SQLException e) {
* e.printStackTrace();
* }
* return conn;
* }

public static void free(ResultSet rs, Statement stmt, Connection conn) {

* try {
* if (rs != null) {
* rs.close();
* rs = null;
* }
* } catch (SQLException e) {
* e.printStackTrace();
* } finally {
* try {
* if (stmt != null) {
* stmt.close();
* stmt = null;
* }
* } catch (SQLException e) {
* e.printStackTrace();
* } finally {
* try {
* if (conn != null) {
* conn.close();
* conn = null;
* }
* } catch (SQLException e) {
* e.printStackTrace();
* }
* }
* }
* }
* }
* 实际调用的代码：
* import java.sql.\*;

public class TestTemplate {

public static void main(String[] args) {

* read();
* }

public static void read(){

* Connection c = null;
* Statement stmt = null;
* ResultSet rs = null;
* try{
* c = JDBCUtils.getConn();
* stmt = c.createStatement();
* rs = stmt.executeQuery("select \* from dept");
* while(rs.next()){
* System.out.println(rs.getString("deptno"));
* }
* }catch(SQLException e){
* e.printStackTrace();
* }finally{
* JDBCUtils.free(rs, stmt, c);
* }
* }
* }
* 以下可以忽略
* 工具类可以采用单例模式

类中有一个静态私有的实例，有一个公开的静态的获取该实例的方法，其他方法是public但不是静态的

* 当其他的类要调用该工具类的方法时，先获取该类的实例，再调用其方法
* 还可以采用延迟加载，当第一次使用该类时new出实例；一开始是没有实例的成员变量的
* 以后使用该类就不会再new第二个实例了。
* 构造实例的成本很高时多采用该方法
* 还可以再完善，并发（多线程)控制时将new实例的部分加锁，避免出现两个实例
* 永远保持单个实例
* 这是双重检查
* DML增删改
* 注意sql语句不建议用\*，并且get获取值时不应该写1,2 ；写为字段名可读性更高
* 最好将sql语句中列出所需要的字段，全部取出效率较低
* 这是为了降低维护成本
* executeUpdate 返回值是int类型，是执行成功的影响行数
* 注意关闭执行DML语句的资源也可以使用封装的free方法，因为会判断是否为空，空即不关闭
* 注意与之前步骤不同，不需要ResultSet这个类，因为不需要接收结果集
* 示例模板（使用了JDBCUtils工具类)：
* import java.sql.\*;

public class TestTemplate {

public static void main(String []args){

* add();
* }

private static void add() {

* Connection c = null;
* Statement stmt = null;
* ResultSet rs = null;
* try{
* c = JDBCUtils.getConn();
* stmt = c.createStatement();
* String sql = "insert into dept2 values(14,'lala','china')";
* int result = stmt.executeUpdate(sql);
* System.out.println(result);
* }catch(SQLException e){
* e.printStackTrace();
* }finally{
* JDBCUtils.free(rs, stmt, c);
* }
* }
* }

# JDBC进阶

* SQL注入攻击指的是通过构建特殊的输入作为参数传入Web应用程序，而这些输入大都是SQL语法里的一些组合，通过执行SQL语句进而执行攻击者所要的操作，其主要原因是程序没有细致地过滤用户输入的数据，致使非法数据侵入系统。
* select \* from emp where ename = '' or 1 = 1;
* PreparedStatement类
* 是Statement接口的子接口
* 表示预编译的 SQL 语句的对象。
* SQL 语句（增改删DML)被预编译并存储在 PreparedStatement 对象中。然后可以使用此对象多次高效地执行该SQL语句。
* 与Statement不同的是通过preparedStatement方法来获得PreparedStatement对象。
* 参数：
* sql - 可能包含一个或多个 '?' IN 参数占位符的 SQL 语句
* 返回：
* 包含预编译 SQL 语句的新的默认 PreparedStatement 对象
* 不同于createStatement不需要参数，这个方法需要一个参数，即预编译的SQL语句，其插入的数据用？占位符表示
* PreparedStatement pstmt = conn.prepareStatement(“insert into dept values(?,?,?)”);
* 之后再使用pstmt来为这三个？赋值
* 使用的方法有：
* 这些方法的第一个参数都是表示第几个占位符
* 1表示第一个？
* 2表示第二个？
* pstmt.setInt(1,deptno);
* pstmt.setString(2,dname);
* pstmt.setString(3,loc);
* 另一个区别是executeUpdate方法不再需要参数传入。
* PreparedStatement类的好处是不必去考虑怎么才能拼凑出格式正确的SQL语句，而是调用方法就可以设置相应的值，十分方便；也易于修改占位符值。尽量使用；可以解决SQL注入问题（过滤掉特殊字符)（最大的优点)；同时效率同Statement相比较高（避免频繁SQL语句，是预编译)
* 可以执行查询和DML操作。
* 示例：
* import java.sql.\*;

public class TestTemplate {

public static void main(String []args){

- if(args.length != 3){

* System.out.println("Input Error!");
  + System.exit(-1);
* }
* int deptno =0 ;
* try{
* deptno = Integer.parseInt(args[0]);
* }catch(NumberFormatException e){
* e.printStackTrace();
* }
* String dname = args[1];
* String loc = args[2];
* Connection c = null;
* PreparedStatement pstmt = null;
* ResultSet rs = null;
* try{
* c = JDBCUtils.getConn();
* pstmt = c.prepareStatement("insert into dept2 values(?,?,?)");
* pstmt.setInt(1, deptno);
* pstmt.setString(2,dname);
* pstmt.setString(3,loc);
* pstmt.executeUpdate();
* }catch(SQLException e){
* e.printStackTrace();
* }finally{
* JDBCUtils.free(rs, pstmt, c);//这里仍可以执行，因为PreparedStatement是Statement的子类，而非超类
* }
* }
* }
* 在java中SQL语句测定时间的一种方法，在执行之前和执行之后的时间打印出来
* JDBC中最耗时间的是建立连接；使用的是Socket连接，三次握手机制
* 发送用户名密码
* 比发送执行SQL语句的时间要长得多
* 注意虽然PreparedStatement是Statement是子类，但是不能调用父类的executeQuery(sql);
* 会出错
* 因为该方法会直接将参数原始的sql语句传到数据库，而不管之前的填充sql语句的步骤
* 示例：
* import java.sql.\*;

public class TestTemplate {

public static void main(String []args){

* read("SMITH");
* }

private static void read(String name) {

* Connection c = null;
* PreparedStatement pstmt = null;
* ResultSet rs = null;
* try{
* String sql = "select \* from emp where ename = ?";
* c = JDBCUtils.getConn();
* pstmt = c.prepareStatement(sql);
* pstmt.setString(1,name);
* rs = pstmt.executeQuery();//如果这里加上sql，那么会报错
* while(rs.next()){
  + System.out.println(rs.getString(1));
* }
* }catch(SQLException e){
* e.printStackTrace();
* }finally{
* JDBCUtils.free(rs, pstmt, c);
* }
* }
* }
* 如何选择？ 一般带有参数的sql都使用PreparedStatement
* 如果没有条件或条件固定的可以使用Statement。 建议全部使用PreparedStatement
* CallableStatement(存储过程)
* 是PreparedStatement的子接口，是Statement的孙子接口。
* 用于执行 SQL 存储过程的接口。JDBC API 提供了一个存储过程 SQL 转义语法，该语法允许对所有 RDBMS 使用标准方式调用存储过程。此转义语法有一个包含结果参数的形式和一个不包含结果参数的形式。如果使用结果参数，则必须将其注册为 OUT 参数。其他参数可用于输入、输出或同时用于二者。参数是根据编号按顺序引用的，第一个参数的编号是 1。
* 继承自PreparedStatement
* Connection
* 可以创建一个CallableStatement对象
* 参数是sql语句
* sql语句是这样写的
* CollableStatement cs = null;
* cs = conn.prepareCall(“{call pro1(?,?)}”); //参数以？表示，之后赋值
* cs.setString(1,”SMITH”);
* cs.setIntFloat(2,456.7f);//f是表示float类型
* 第一个参数是1,2,3等 表示第一个、第二个、第三个？
* 第二个参数是所要设为的值
* 设置完后调用execute方法
* 注意如果在sql 中没有commit提交事务，那么java中无法访问数据库，处于阻塞状态
* 代码：

public static void main(String []args){

* Connection c = null;
* CallableStatement cs = null;
* try{
* c =JDBCUtils.getConn();
* cs = c.prepareCall("{call pro3(?,?)}");
* cs.setString(1, "SMITH");
* cs.setFloat(2, 7800f); //这个也可以是setString
* //注意oracle提供一种自动转换机制，如果该字段是数字类型，那么会自动将字符串转为数字
* cs.execute();
* 可以在SQL工具类中封装一个可以调用存储过程的方法
* 工具类中的成员变量均为静态变量，方法都是静态方法
* SQLHelper:

public static void callProcedure(String sql,String []parameters){

* Connection c= null;
* CallableStatement cs= null;
* ResultSet rs = null;
* //所有含有参数的sql语句都使用PreparedStatement，传入时还包括一个字符串数据
* 然后通过一个循环将参数使用setString将sql语句补充完毕
* 然后调用
* try{
* c =JDBCUtils.getConn();
* cs = c.prepareCall(sql);
* if(parameters != null && "".equals(parameters)){
* for(int i = 0; i< parameters.length ;i++){
* cs.setString(i+1,parameters[i]);
* }
* }
* cs.execute();
* }catch(SQLException e){
* e.printStackTrace();
* }finally{
* JDBCUtils.free(rs,cs,c);
* }
* }
* 调用此方法的代码：
* 带返回值参数的存储过程
* 分页查询
* 存储过程代码：
* create or replace procedure paged\_query
* (v*in*table in varchar2,v*in*record*per*page in number, v*in*now*page in number,v*in\_col varchar2,
* v*out*records out number,v*out*pages out number, v*out*result out p1.my\_cursor) is
* v\_start number;
* v\_end number;
* v\_sql varchar2(2000);
* v*sql*get\_record varchar2(300);
* begin
  + v*start := 1+v*in*record*per*page\*( v*in*now*page-1) ;
* v*end := 1+ v*in*record*per*page \* v*in*now*page;
* v*sql*get*record := 'select count(\*) from '||v*in\_table;
* execute immediate v*sql*get*record into v*out\_records;
* if mod(v*out*records,v*in*record*per*page) = 0 then
* v*out*pages := v*out*records / v*in*record*per*page;
* else
* v*out*pages := v*out*records / v*in*record*per*page + 1;
* end if;
* v\_sql := 'select t2.\* from (select t1.\* ,rownum rn from
* (select \* from '||v*in*table||' order by '||v*in*col||') t1
* where rownum <= '||v\_end||'
* ) t2
* where rn >= '||v\_start;
* open v*out*result for v\_sql;
* end;
* /
* 使用CallableStatement来接收Connection的prepareCall方法，得到statement
* 传入的是含有等同于参数个数的sql语句
* String sql = “{call pro1(?,?)}”;
* 然后set方法设置每个？所对应的值；注意数值类型也可以用setString
* oracle会自动转为相应的数值类型
* 与不含输出参数的过程的第一个不同是还需要调用这个方法registerOutParameter
* 注册输出参数的类型
* 注意不同数据库，参数类型也是不同的
* 第一个参数是给第n个？赋值；第二个参数是对应的返回值的类型
* 对oracle而言是oracle.jdbc.OracleTypes.某个类型
* 最后execute
* 第二个不同是还需要再取出返回值
* get…数据类型(n)
* 比如getString(第几个？的返回值);
* 如果返回的是结果集（游标)，那么需要使用结果集来接收游标变量
* oracle.jdbc.OracleTypes.CURSOR
* 取出返回值的集合
* 方法是getObject 将游标视为一个对象
* 接收的是ResltSet结果集（需要将Object类型强制转为ResultSet)
* ResultSet实际上就是游标
* 之后就可以使用next和get…来获取值了
* java调用代码:
* try {
* c = JDBCUtils.getConn();
* cs = c.prepareCall("{call paged\_query(?,?,?,?,?,?,?)}");
* int page = 2;
* cs.setString(1, "emp");
* cs.setString(2,"5");
* cs.setString(3,""+page);
* cs.setString(4, "sal");
* cs.registerOutParameter(5, oracle.jdbc.OracleTypes.NUMBER);
* //每个返回的变量都需要注册
* cs.registerOutParameter(6, oracle.jdbc.OracleTypes.NUMBER);
* cs.registerOutParameter(7, oracle.jdbc.OracleTypes.CURSOR);
* cs.execute();
  + String recordCount = cs.getString(5);
  + String pageCount = cs.getString(6);
* System.out.println("共有"+recordCount+"条记录");
* //执行完后需要逐个取出返回值
* System.out.println("共有"+pageCount+"页");
* System.out.println("第"+page+"页:");
  + rs = (ResultSet)cs.getObject(7);
* while(rs.next()){
  + System.out.println(rs.getString(1)+","+rs.getString(2));
* }catch ……略
* getGeneratedKeys产生主键
* 用于拿出插入的记录的主键
* getGeneratedKeys(获取主键)
* API介绍：
* Connection:
* prepareStatement
* PreparedStatement prepareStatement(String sql,
* int autoGeneratedKeys)
* throws SQLException
* 创建一个默认 PreparedStatement 对象，该对象能获取自动生成的键。给定常量告知驱动程序是否可以获取自动生成的键。如果 SQL 语句不是一条 INSERT 语句，或者 SQL 语句能够返回自动生成的键（这类语句的列表是特定于供应商的)，则忽略此参数。
* 该SQL语句是插入语句，可以立刻获得插入记录的主键
* 第二个参数是Statement中的常量
* PreparedStatement:
* ResultSet getGeneratedKeys() throws SQLException
* 获取由于执行此 Statement 对象而创建的所有自动生成的键。如果此 Statement 对象没有生成任何键，则返回空的 ResultSet 对象。
* 注：如果未指定表示自动生成键的列，则 JDBC 驱动程序实现将确定最能表示自动生成键的列。
* 返回：包含通过执行此 Statement 对象自动生成的键的 ResultSet 对象
* 执行的是insert语句，返回的是ResultSet
* ResultSet 可能是多个主键（组合主键)，所以需要ResultSet
* 代码：

public static void add() {

* Connection c = null;
* PreparedStatement ps = null;
* ResultSet rs = null;
* String sql = "insert into UserTable(user*name,user*password) values(?,?)";
* try {
* c = JDBCUtils.getConn();
* ps = c.prepareStatement(sql, Statement.RETURN*GENERATED*KEYS);
* ps.setString(1, "wulitaotao");
* ps.setString(2, "666");
* ps.executeUpdate();
* rs = ps.getGeneratedKeys();
* if(rs.next()){
  + System.out.println(“刚插入记录的id为”+rs.getInt(1));
* }
* } catch (SQLException e) {
* e.printStackTrace();
* } finally{
* JDBCUtils.free(rs,ps,c);
* }
* }
* 实际作用是如果数据库的表是自动主键，那么插入之后是不知道id的。只能从数据库中根据其他的唯一键来找到这条记录再取出主键
* 而这种方式可以在插入之后立刻得到主键，然后取出赋给对象，ok
* 优化UserDAOJDBCImpl中的addUser方法
* try {
* c = JDBCUtils.getConn();
* String sql = "insert into UserTable(user*name,user*password,user\_birthday)
* values(?,?,?)";
* pstmt = c.prepareStatement(sql,Statement.RETURN*GENERATED*KEYS);
* pstmt.setString(1, user.getUsername());
* pstmt.setString(2, user.getPassword());
* pstmt.setDate(3, new java.sql.Date(user.getBirthday().getTime()));
* pstmt.executeUpdate();
* rs = pstmt.getGeneratedKeys();
* if(rs.next()){
  + user.setId(rs.getInt(1));
* }
* } catch (SQLException e) {
* throw new DAOException(e.getMessage(),e);
* } finally {
* JDBCUtils.free(rs, pstmt, c);
* }
* 批处理Batch
* 可以一次执行多条SQL语句，调用Statement接口的addBatch方法
* 不必建立多次的连接（建立连接成本很高)，只建立一次连接就可以执行多条语句
* 将给定的 SQL 命令添加到此 Statement 对象的当前命令列表中。通过调用方法 executeBatch 可以批量执行此列表中的命令。
* 参数：
* sql - 通常此参数为 SQL INSERT 或 UPDATE 语句
* 执行语句时调用
* 将一批命令提交给数据库来执行
* 如果全部命令执行成功，则返回更新计数组成的数组。返回数组的 int 元素的排序对应于批中的命令，批中的命令根据被添加到批中的顺序排序。
* 返回：
* 包含批中每个命令的一个元素的更新计数所组成的数组。数组的元素根据将命令添加到批中的顺序排序。
* 批处理适用于Statement，当然适用于其子类PreparedStatement
* 示例1 Statement：
* import java.sql.\*;

public class JDBC {

public static void main(String []args){

* Connection c = null;
* Statement stmt = null;
* ResultSet rs = null;
* try{
* c = JDBCUtils.getConn();
* stmt = c.createStatement();
* stmt.addBatch("insert into dept2 values(88,'aaa','aaa')");
* stmt.addBatch("insert into dept2 values(89,'aaa','aaa')");
* stmt.addBatch("insert into dept2 values(90,'aaa','aaa')");
* stmt.executeBatch();
* }catch(SQLException e){
* e.printStackTrace();
* }finally{
* JDBCUtils.free(rs,stmt,c);
* }
* }
* }
* 示例2 PreparedStatement：
* import java.sql.\*;

public class JDBC {

public static void main(String []args){

* Connection c = null;
* PreparedStatement pstmt = null;
* ResultSet rs = null;
* try{
* c = JDBCUtils.getConn();
* pstmt = c.prepareStatement("insert into
* UserTable(user*name,user*password,user\_birthday) values(?,?,?)");
* for(int i = 0 ; i<100;i++){//一次性插入100条记录
  + pstmt.setString(1, "user"+i+1);
* pstmt.setString(2, "111");
* pstmt.setDate(3, new Date(System.currentTimeMillis()));
* pstmt.addBatch();
* //设置完一条记录添加一次，然后重新设置下一条记录 字段的值
* }
* int []id = pstmt.executeBatch();//返回值全部是1
* for(int i:id){
* System.out.println(i);
* }
* }catch(SQLException e){
* e.printStackTrace();
* }finally{
* JDBCUtils.free(rs,pstmt,c);
* }
* }
* }
* addBatch(sql) 也是存在的，可以将sql语句传入
* 如果记录太多可能会内存溢出
* 事务处理
* 在java程序中将多个DML语句视为一个事务，统一提交和回滚
* 注意java中事务是自动提交的，也就是执行一条DML语句就commit一下
* 需要我们去设置不自动提交事务，由自己来设定事务
* Connection有一个方法setAutoCommit
* 参数为true or false
* 当想提交时，执行Connection的commit方法
* 在执行commit方法之前的DML语句都还未提交，当执行commit方法时将之前的DML语句视为一个事务，整体地执行
* 当事务执行过程中出现异常，可以在catch到exception后回滚
* 可以Connection的rollback方法，有重载的方法，可以无参数，对应sql中的rollback；
* rollback方法可以有参数，是保存点，回滚到这个保存点

public static void testSavepoint() throws Exception{

* Connection c = null;
* Statement stmt = null;
* ResultSet rs = null;
* String sql1 = "update emp2 set sal = sal - 800 where empno = 7369";
* String sql2 = "update emp2 set sal = sal - 800 where empno = 7369";
* try {
* c = JDBCUtils.getConn();
* c.setAutoCommit(false);
* stmt = c.createStatement();
  + stmt.executeUpdate(sql1);
* int i = 8/0; //故意制造出一些错误
  + stmt.executeUpdate(sql2);
* c.commit();
* c.setAutoCommit(true);//恢复现场
* }catch (Exception e) { //捕捉到任何异常，立刻返回到初始状态
* //不能只写SQLException，应该是所有异常，否则ArithmeticException就捕捉不到了，捕捉不到也就无法执行catch块中的代码了
* if(c != null) //如果没有建立连接那么没有必要去rollback
* c.rollback();
* c.setAutoCommit(true);//无论如何都应该恢复现场
* throw e;//交给上一级去处理
* } finally{
* JDBCUtils.free(rs,stmt,c);
* }
* }
* }
* 设置保存点:
* 假如在执行了第一条sql之后设置保存点，如果在执行第二条sql时出错，回滚到第一条sql之后
* Connection:

public static void testSavepoint() throws SQLException {

* Connection c = null;
* Statement stmt = null;
* ResultSet rs = null;
* Savepoint sp = null;
* String sql1 = "update emp2 set sal = sal - 800 where empno = 7369";
* String sql2 = "update emp2 set sal = sal - 800 where empno = 7369";
* String sql3 = "select sal from emp2 where empno = 7369";
* try {
* c = JDBCUtils.getConn();
* c.setAutoCommit(false);
* stmt = c.createStatement();
  + stmt.executeUpdate(sql1);
* sp = c.setSavepoint();
  + rs = stmt.executeQuery(sql3);
* int sal = 0;
* if(rs.next())
* sal = rs.getInt("sal");
* if(sal < 1000)
* throw new RuntimeException("工资过低!");
* //自己new一个异常，以便于与SQLException区分开来
  + stmt.executeUpdate(sql2);
* c.commit();
* c.setAutoCommit(true);//恢复现场
* }catch (RuntimeException e) {
* if(c != null && sp != null){
* c.rollback(sp);
* //如果工资减去800之后小于1000，那么不再减少，保留第一次的结果
* }
* c.setAutoCommit(true);
* throw e;
* } catch (SQLException e) {
* if(c != null){
* c.rollback();
* c.setAutoCommit(true);
* }
* throw e;
* } finally{
* JDBCUtils.free(rs,stmt,c);
* }
* }
* JTA类似指挥官，第一阶段给所有数据库发送一个准备提交的请求，如果有数据库提出要回滚，那么JTA会通知其他数据库，一起回滚
* 如果没有数据库没有提出要回滚，那么第二阶段JTA发送提交的命令

# JDBC桥接模式源码分析（以MySQL为例)

# 5.12 Class.forName

* Class.forName("com.mysql.jdbc.Driver");
* Driver#static{}
* 注册MySQL的Driver

public class Driver extends NonRegisteringDriver implements java.sql.Driver {  
 // ~ Static fields/initializers  
 // ---------------------------------------------  
  
 //  
 // Register ourselves with the DriverManager  
 //  
 static {  
 try {  
 java.sql.DriverManager.registerDriver(new Driver());  
 } catch (SQLException E) {  
 throw new RuntimeException("Can't register driver!");  
 }  
 }

* }
* DriverManager#registerDriver

public static synchronized void registerDriver(java.sql.Driver driver)  
 throws SQLException {  
  
 registerDriver(driver, null);  
}

public static synchronized void registerDriver(java.sql.Driver driver,  
 DriverAction da)  
 throws SQLException {  
  
 /\* Register the driver if it has not already been added to our list \*/  
 if(driver != null) {  
 registeredDrivers.addIfAbsent(new DriverInfo(driver, da));  
 } else {  
 // This is for compatibility with the original DriverManager  
 throw new NullPointerException();  
 }  
  
 println("registerDriver: " + driver);  
  
}

* DriverManager#getConnection

public static Connection getConnection(String url)  
 throws SQLException {  
  
 java.util.Properties info = new java.util.Properties();  
 return (getConnection(url, info, Reflection.getCallerClass()));  
}

private static Connection getConnection(  
 String url, java.util.Properties info, Class<?> caller) throws SQLException {  
 /\*  
 \* When callerCl is null, we should check the application's  
 \* (which is invoking this class indirectly)  
 \* classloader, so that the JDBC driver class outside rt.jar  
 \* can be loaded from here.  
 \*/  
 ClassLoader callerCL = caller != null ? caller.getClassLoader() : null;  
 synchronized(DriverManager.class) {  
 // synchronize loading of the correct classloader.  
 if (callerCL == null) {  
 callerCL = Thread.currentThread().getContextClassLoader();  
 }  
 }  
  
 if(url == null) {  
 throw new SQLException("The url cannot be null", "08001");  
 }  
  
 println("DriverManager.getConnection(\"" + url + "\")");  
  
 // Walk through the loaded registeredDrivers attempting to make a connection.  
 // Remember the first exception that gets raised so we can reraise it.  
 SQLException reason = null;  
  
 for(DriverInfo aDriver : registeredDrivers) {  
 // If the caller does not have permission to load the driver then  
 // skip it.  
 if(isDriverAllowed(aDriver.driver, callerCL)) {  
 try {  
 println(" trying " + aDriver.driver.getClass().getName());  
 Connection con = aDriver.driver.connect(url, info);  
 if (con != null) {  
 // Success!  
 println("getConnection returning " + aDriver.driver.getClass().getName());  
 return (con);  
 }  
 } catch (SQLException ex) {  
 if (reason == null) {  
 reason = ex;  
 }  
 }  
  
 } else {  
 println(" skipping: " + aDriver.getClass().getName());  
 }  
  
 }  
  
 // if we got here nobody could connect.  
 if (reason != null) {  
 println("getConnection failed: " + reason);  
 throw reason;  
 }  
  
 println("getConnection: no suitable driver found for "+ url);  
 throw new SQLException("No suitable driver found for "+ url, "08001");  
}

* Driver#connect

public java.sql.Connection connect(String url, Properties info)  
 throws SQLException {  
 if (url != null) {  
 if (StringUtils.startsWithIgnoreCase(url, LOADBALANCE\_URL\_PREFIX)) {  
 return connectLoadBalanced(url, info);  
 } else if (StringUtils.startsWithIgnoreCase(url,  
 REPLICATION\_URL\_PREFIX)) {  
 return connectReplicationConnection(url, info);  
 }  
 }  
  
 Properties props = null;  
  
 if ((props = parseURL(url, info)) == null) {  
 return null;  
 }  
  
 if (!"1".equals(props.getProperty(NUM\_HOSTS\_PROPERTY\_KEY))) {  
 return connectFailover(url, info);  
 }  
   
 try {  
 Connection newConn = com.mysql.jdbc.ConnectionImpl.getInstance(  
 host(props), port(props), props, database(props), url);  
   
 return newConn;  
 } catch (SQLException sqlEx) {  
 // Don't wrap SQLExceptions, throw  
 // them un-changed.  
 throw sqlEx;  
 } catch (Exception ex) {  
 SQLException sqlEx = SQLError.createSQLException(Messages  
 .getString("NonRegisteringDriver.17") //$NON-NLS-1$  
 + ex.toString()  
 + Messages.getString("NonRegisteringDriver.18"), //$NON-NLS-1$  
 SQLError.SQL\_STATE\_UNABLE\_TO\_CONNECT\_TO\_DATASOURCE, null);  
   
 sqlEx.initCause(ex);  
   
 throw sqlEx;  
 }  
}

# 5.17 总结

* 有了抽象部分——JDBC的API，有了具体实现部分——驱动程序，那么它们如何连接起来呢？就是如何桥接呢？
* 就是前面提到的DriverManager来把它们桥接起来，从某个侧面来看，DriverManager在这里起到了类似于简单工厂的功能，基于JDBC的应用程序需要使用JDBC的API，如何得到呢？就通过DriverManager来获取相应的对象。
* JDBC的这种架构，把抽象和具体分离开来，从而使得抽象和具体部分都可以独立扩展。对于应用程序而言，只要选用不同的驱动，就可以让程序操作不同的数据库，而无需更改应用程序，从而实现在不同的数据库上移植；对于驱动程序而言，为数据库实现不同的驱动程序，并不会影响应用程序。而且，JDBC的这种架构，还合理的划分了应用程序开发人员和驱动程序开发人员的边界。
* Class.forName是用MySql还是Oracle，这个Driver一定会实现接口java.sql.Driver，然后通过DriverManager.registerDriver(new Driver());使DriverManager类持有一个Driver，是否可以把DriverManager当成桥，当成桥连接中的抽象类？然后持有一个接口Driver，至于是MySql还是Oracle，不关心，坐等传参。因为DriverManager持有的是一个Driver接口，你传过来什么，我就得到什么的实例化，然后我再通过getConnection用你的实例，去调用你自己的方法connect，去获得Connection的一个实例。

# 安全

# XSS攻击

# 5.18 原理

* 跨站脚本攻击（Cross-Site Scripting, XSS)：主要是指在用户浏览器内运行了JavaScript 脚本。比如富文本编辑器，如果不过滤用户输入的数据直接显示用户输入的HTML内容的话，就会有可能运行恶意的 JavaScript 脚本，导致页面结构错乱，Cookies 信息被窃取等问题。
* XSS 的原理是恶意攻击者往 Web 页面里插入恶意可执行网页脚本代码，当用户浏览该页之时，嵌入其中 Web 里面的脚本代码会被执行，从而可以达到攻击者盗取用户信息或其他侵犯用户安全隐私的目的。
* 常见的XSS攻击类型有两种，一种是反射型，一种是持久型。

## 反射型

* 攻击者诱使用户点击一个嵌入恶意脚本的链接，达到工具的目的。
* 比如新浪微博中，攻击者发布的微博中含有一个恶意脚本的URL（URL中包含脚本的链接)，用户点击该URL，脚本会自动关注攻击者的新浪微博ID，发布含有恶意脚本URL的微博，攻击就被扩散了。
* 现实中，攻击者可以采用XSS攻击，偷取用户Cookie、密码等重要数据，进而伪造交易、盗窃用户财产、窃取情报。

## 持久型

* 黑客提交含有恶意脚本的请求，保存在被攻击的Web站点的数据库中，用户浏览网页时，恶意脚本被包含在正常页面中，达到攻击的目的。此种攻击经常使用在论坛、博客等Web应用中。

# 5.19 预防

* Web 页面渲染的所有内容或者渲染的数据都必须来自于服务端。
* 后端在入库前应该选择不相信任何前端数据，将所有的字段统一进行转义处理。
* 后端在输出给前端数据统一进行转义处理。
* 前端在渲染页面 DOM 的时候应该选择不相信任何后端数据，任何字段都需要做转义处理。

## 消毒

* XSS攻击者一般都是在请求中嵌入恶意脚本达到攻击的目的，这些脚本是一般在用户输入中不常用的，如果进行过滤和消毒处理，即对某些HTML危险字符转义，如”>”转义为”>”，就可以防止大部分的攻击。为了避免对不必要的内容错误转义，如”3<5”中的”<”需要进行文本匹配后再转义，如”<img src=”这样上下文中的”<”才转义。

## HttpOnly

* 浏览器禁止页面JavaScript访问带有HttpOnly属性的Cookie。

# SQL注入

# 5.20 原理

* 针对 Web 应用使用的数据库，通过运行非法的SQL而产生的攻击。

# 5.21 预防

* 所有的查询语句建议使用数据库提供的参数化查询接口，使用SQL预编译和参数绑定。
* 在应用发布之前建议使用专业的 SQL 注入检测工具进行检测。

# CSRF攻击

# 5.22 原理

* CSRF（Cross-site request forgery)跨站请求伪造，利用跨站请求，在用户不知情的情况下，以用户的身份伪造请求，其核心是利用了浏览器Cookie或者Session，盗取用户身份。
* 下面是CSRF的常见特性：
* 依靠用户标识危害网站
* 利用网站对用户标识的信任，欺骗用户的浏览器发送HTTP请求给目标站点
* 攻击者可以盗用你的登陆信息，以你的身份模拟发送各种请求。攻击者只要借助少许的社会工程学的诡计，例如通过 QQ 等聊天软件发送的链接(有些还伪装成短域名，用户无法分辨)，攻击者就能迫使 Web 应用的用户去执行攻击者预设的操作。例如，当用户登录网络银行去查看其存款余额，在他没有退出时，就点击了一个 QQ 好友发来的链接，那么该用户银行帐户中的资金就有可能被转移到攻击者指定的帐户中。
* 所以遇到 CSRF 攻击时，将对终端用户的数据和操作指令构成严重的威胁。当受攻击的终端用户具有管理员帐户的时候，CSRF 攻击将危及整个 Web 应用程序。

# 5.23 预防

* 防御手段主要是识别请求者身份。
* 1、重要数据交互采用POST进行接收，当然是用POST也不是万能的，伪造一个form表单即可破解
* 2、使用验证码，只要是涉及到数据交互就先进行验证码验证，这个方法可以完全解决CSRF。但是出于用户体验考虑，网站不能给所有的操作都加上验证码。因此验证码只能作为一种辅助手段，不能作为主要解决方案。
* 3、验证HTTP Referer字段，该字段记录了此次HTTP请求的来源地址，最常见的应用是图片防盗链。PHP中可以采用APache URL重写规则进行防御，可参考：http://www.cnblogs.com/phpstudy2015-6/p/6715892.html（但是这个也是可以绕过的)
* 4、为每个表单添加令牌token并验证（推荐)
* （可以使用cookie或者session进行构造。当然这个token仅仅只是针对CSRF攻击，在这前提需要解决好XSS攻击，否则这里也将会是白忙一场【XSS可以偷取客户端的cookie】)
* 可以为每一个表单生成一个随机数秘钥，并在服务器端建立一个拦截器来验证这个token，如果请求中没有token或者token内容不正确，则认为可能是CSRF攻击而拒绝该请求。
  1. 用户访问某个表单页面。
  2. 服务端生成一个Token，放在用户的Session中，或者浏览器的Cookie中。【这里已经不考虑XSS攻击】
  3. 在页面表单附带上Token参数。
  4. 用户提交请求后， 服务端验证表单中的Token是否与用户Session（或Cookies)中的Token一致，一致为合法请求，不是则非法请求。

# DDoS 攻击

# 5.24 原理

* DDoS 又叫分布式拒绝服务，全称 Distributed Denial of Service，其原理就是利用大量的请求造成资源过载，导致服务不可用。
* TCP的半连接

# 5.25 预防

- 网络架构上做好优化，采用负载均衡分流。  
- 确保服务器的系统文件是最新的版本，并及时更新系统补丁。  
- 添加抗 DDos 设备，进行流量清洗。  
- 限制同时打开的 SYN 半连接数目，缩短 SYN 半连接的 Timeout 时间。  
- 限制单 IP 请求频率。  
- 防火墙等防护设置禁止 ICMP 包等。  
- 严格限制对外开放的服务器的向外访问。  
- 运行端口映射程序或端口扫描程序，要认真检查特权端口和非特权端口。  
- 关闭不必要的服务。  
- 认真检查网络设备和主机/服务器系统的日志。只要日志出现漏洞或是时间变更,那这台机器就可能遭到了攻击。  
- 限制在防火墙外与网络文件共享。这样会给黑客截取系统文件的机会，主机的信息暴露给黑客，无疑是给了对方入侵的机会。

# 加密算法

* 什么是对称加密，什么是非对称加密，知道的加密算法有哪些
* BCrypt算法
* 对称加密在加密和解密的过程中，使用相同的密钥；而非对称加密在加密过程中使用公钥进行加密，使用私钥进行解密。

# 5.26 对称加密

* 所谓常规密钥密码体制，即加密密钥与解密密钥是相同的密码体制。这种加密系统又称为对称密钥系统。
* 对称加密算法有：DES、AES等。
* DES 的保密性仅取决于对密钥的保密，而算法是公开的。尽管人们在破译 DES 方面取得了许多进展，但至今仍未能找到比穷举搜索密钥更有效的方法。
* DES 是世界上第一个公认的实用密码算法标准，它曾对密码学的发展做出了重大贡献。
* 目前较为严重的问题是 DES 的密钥的长度。
* 现在已经设计出来搜索 DES 密钥的专用芯片。
* 对称加密的加密和解密需要使用相同的密钥，所以需要解决密钥配送问题。

# 5.27 非对称加密

* 公钥密码体制使用不同的加密密钥与解密密钥，是一种“由已知加密密钥推导出解密密钥在计算上是不可行的”密码体制。
* 公钥密码体制的产生主要是因为两个方面的原因，一是由于常规密钥密码体制的密钥分配问题，另一是由于对数字签名的需求。
* 现有最著名的公钥密码体制是RSA 体制，它基于数论中大数分解问题的体制，由美国三位科学家 Rivest, Shamir 和 Adleman 于 1976 年提出并在 1978 年正式发表。
* 可以用公钥加密，私钥解密；也可以用私钥加密、公钥解密。
* 若密钥能够实现安全交换，那么有可能会考虑仅使用公开密钥加密来通信。但是公开密钥加密与共享密钥加密相比，其处理速度要慢。
* 任何加密方法的安全性取决于密钥的长度，以及攻破密文所需的计算量
* 在这方面，公钥密码体制并不比传统加密体制更加优越
* 由于目前公钥加密算法的开销较大，在可见的将来还不会放弃传统的加密方法
* 公钥需要密钥分配协议，具体的分配过程并不比采用传统加密方法时更简单

# 消息摘要/单向散列

* 单向散列函数也称为消息摘要函数（message digest function)，哈希函数，适用于检查消息完整性的加密技术。
* 单向散列函数有一个输入和一个输出，其中输入称为信息，输出称为散列值。单向散列函数可以根据消息的内容计算出散列值，篡改后的信息的散列值计算结果会不一样，所以散列值可以被用来检查消息的完整性 。
* 常见消息摘要技术：MD5、SHA-1、SHA-256
* CRC、MD5、SHA1都是通过对数据进行计算，来生成一个校验值，该校验值用来校验数据的完整性。
* 不同点：
  1. 算法不同。CRC采用多项式除法，MD5和SHA1使用的是替换、轮转等方法；
  2. 校验值的长度不同。CRC校验位的长度跟其多项式有关系，一般为16位或32位；MD5是16个字节（128位)；SHA1是20个字节（160位)；
  3. 校验值的称呼不同。CRC一般叫做CRC值；MD5和SHA1一般叫做哈希值（Hash)或散列值；
  4. 安全性不同。这里的安全性是指检错的能力，即数据的错误能通过校验位检测出来。CRC的安全性跟多项式有很大关系，相对于MD5和SHA1要弱很多；MD5的安全性很高，不过大概在04年的时候被山东大学的王小云破解了；SHA1的安全性最高（现在SHA-256安全性比较高)。
  5. 效率不同，CRC的计算效率很高；MD5和SHA1比较慢。
  6. 用途不同。CRC一般用作通信数据的校验；MD5和SHA1用于安全（Security)领域，比如文件校验、数字签名等。

# 5.28 密码

* 利用单向散列加密的特性，可以进行密码加密保存，即用户注册时输入的密码不直接保存到数据库，而是对密码进行单向散列加密，将密文存入数据库，用户登录时，进行密码验证，同样计算得到输入密码的密文，并和数据库中的密文比较，如果一致，则密码验证成功。
* 这样保存在数据库中的是用户输入的密码的密文，而且不可逆地计算得到密码的明文，因此即使数据库被拖库（指网站遭到入侵后，黑客窃取其数据库)，也不会泄露用户的密码信息。
* 虽然不能通过算法将单向散列密文反算得到明文，但是由于人们设置密码具有一定的模式吗，因此通过彩虹表（建立一个 源数据与加密数据之间对应的hash表。这样在获得加密数据后通过比较，查询或者一定的运算，可以快速定位源数据)等手段可以进行猜测式破解。
* 为了加强单向散列计算的安全性，还会给散列算法加点盐，salt相当于加密的密钥，增加破解的难度。盐一般都是跟hash一起保存在数据库里，或者作为hash字符串的一部分。salt是由系统随机生成的，并且只有系统知道。这样，即便两个用户使用了同一个密码，由于系统为它们生成的salt值不同，他们的散列值也是不同的。

# 5.29 算法介绍

* sha比md5更安全一些，sha比md5哈希碰撞的概率更小一些。
* 到目前为止，我们已经了解如何为密码生成安全的 Hash 值以及通过利用 salt 来加强它的安全性。但今天的问题是，硬件的速度已经远远超过任何使用字典或彩虹表进行的暴力攻击，并且任何密码都能被破解，只是使用时间多少的问题。
* 为了解决这个问题，主要想法是尽可能降低暴力攻击速度来保证最小化的损失。我们下一个算法同样是基于这个概念。目标是使 Hash 函数足够慢以妨碍攻击，并对用户来说仍然非常快且不会感到有明显的延时。
* 要达到这个目的通常是使用某些 CPU 密集型算法来实现，比如 PBKDF2, Bcrypt 或 Scrypt 。这些算法采用 work factor(也称之为 security factor)或迭代次数作为参数来确定 Hash 函数将变的有多慢，并且随着日后计算能力的提高，可以逐步增大 work factor 来使之与计算能力达到平衡。
* bcrypt是单向的，而且经过salt和cost的处理，使其受rainbow攻击破解的概率大大降低，同时破解的难度也提升不少。
* 因为bcrypt采用了一系列各种不同的Blowfish加密算法，并引入了一个work factor，这个工作因子可以让你决定这个算法的代价有多大。因为这些，这个算法不会因为计算机CPU处理速度变快了，而导致算法的时间会缩短了。因为，你可以增加work factor来把其性能降下来。

# 数字签名

* 别人不能冒充我的签名（不可伪造)，我也不能否认上面的签名是我的（不可抵赖)。
* 数字签名又是靠什么保证不可伪造和不可抵赖两个特性呢？
* 答案是利用公钥加密系统。
* RSA既可以用公钥加密然后私钥解密，也可以用私钥加密然后公钥解密（对称性)。
* 因为RSA中的每一个公钥都有唯一的私钥与之对应，任一公钥只能解开对应私钥加密的内容。换句话说，其它私钥加密的内容，这个公钥是解不开的。
* 这样，如果你生成了一对RSA密钥，你把公钥公布出去，并告诉全世界人这个公钥是你的。之后你只要在发送的消息，比如“123456”，后面加上用私钥加密过的密文，其他人拿公钥解密，看解密得到的内容是不是“123456”就可以知道这个“123456”是不是你发的。
* 其他人因为没有对应的私钥，所以没法生成公钥可以解密的密文，所以是不可伪造的。
* 又因为公钥对应的私钥只有一个，所以只要能成功解密，那么发消息的一定是你，不会是其他人，所以是不可抵赖的。
* 由于直接对原消息进行签名有安全性问题，而且原消息往往比较大，直接使用RSA算法进行签名速度会比较慢，所以我们一般对消息计算其摘要（使用SHA-256等安全的摘要算法)，然后对摘要进行签名。只要使用的摘要算法是安全的（MD5、SHA-1已经不安全了)，那么这种方式的数字签名就是安全的。
* 一个具体的RSA签名过程如下：
* 小明对外发布公钥，并声明对应的私钥在自己手上
* 小明对消息M计算摘要，得到摘要D
* 小明使用私钥对D进行签名，得到签名S
* 将M和S一起发送出去
* 验证过程如下：
* 接收者首先对M使用跟小明一样的摘要算法计算摘要，得到D
* 使用小明公钥对S进行解签，得到D’
* 如果D和D’相同，那么证明M确实是小明发出的，并且没有被篡改过。
* 报文鉴别——接收者能够核实发送者对报文的签名
* 报文的完整性——发送者事后不能抵赖对报文的签名
* 不可否认——接收者不能伪造对报文的签名

# 密钥管理

* 对称密码的密钥、非对称加密的私钥、salt等都需要保证不被泄露。
* 改善密钥安全性的方式有两种：
  + 1)把密钥和算法放在一个单独的服务器上，对外提供加密和解密服务，应用系统通过调用这个服务实现数据的加解密。容易成为应用瓶颈，系统性能开销较高。
  + 2)将加解密算法放到应用系统中，密钥放在独立服务器中。实际存储时，密钥被切分成薯片，加密后分别保存在不同存储介质中，兼顾密钥安全性的同时又改善了性能。

# HTTP

# HTTP概述

* HTTP（hypertext transport protocol)，即超文本传输协议。这个协议详细规定了浏览器和万维网服务器之间互相通信的规则。
* HTTP就是一个通信规则，通信规则规定了客户端发送给服务器的内容格式，也规定了服务器发送给客户端的内容格式。其实我们要学习的就是这个两个格式！客户端发送给服务器的格式叫“请求协议”；服务器发送给客户端的格式叫“响应协议”。

# 请求协议

* 请求协议的格式如下：  
  请求首行； 包括请求类型,要访问的资源以及所使用的HTTP版本.  
  请求头信息；  
  空行； \r\n DOS/Windows:’\r\n’ UNIX/Linux:’\n’ Mac:’\r’  
  请求体。
* 浏览器发送给服务器的内容就这个格式的，如果不是这个格式服务器将无法解读！在HTTP协议中，请求有很多请求方法，其中最为常用的就是GET和POST。不同的请求方法之间的区别，后面会一点一点的介绍。

# 5.30 GET请求

* 打开IE，在访问hello项目的index.jsp之间打开HttpWatch，并点击“Record”按钮。然后访问index.jsp页面。查看请求内容如下：

GET /hello/index.jsp HTTP/1.1  
Host: localhost  
User-Agent: Mozilla/5.0 (Windows NT 5.1; rv:5.0) Gecko/20100101 Firefox/5.0  
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8  
Accept-Language: zh-cn,zh;q=0.5  
Accept-Encoding: gzip, deflate  
Accept-Charset: GB2312,utf-8;q=0.7,\*;q=0.7  
Connection: keep-alive  
Cookie: JSESSIONID=369766FDF6220F7803433C0B2DE36D98

- GET /hello/index.jsp HTTP/1.1：GET请求，请求服务器路径为/hello/index.jsp，协议为1.1；  
- Host:localhost：请求的主机名为localhost；  
- User-Agent: Mozilla/5.0 (Windows NT 5.1; rv:5.0) Gecko/20100101 Firefox/5.0：与浏览器和OS相关的信息。有些网站会显示用户的系统版本和浏览器版本信息，这都是通过获取User-Agent头信息而来的；  


Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8：告诉服务器，当前客户端可以接收的文档类型，其实这里包含了\*/\*，就表示什么都可以接收；

- Accept-Language: zh-cn,zh;q=0.5：当前客户端支持的语言，可以在浏览器的工具选项中找到语言相关信息；  
- Accept-Encoding: gzip, deflate：支持的压缩格式。数据在网络上传递时，可能服务器会把数据压缩后再发送；  
- Accept-Charset: GB2312,utf-8;q=0.7,\*;q=0.7：客户端支持的编码；  
- Connection: keep-alive：客户端支持的链接方式，保持一段时间链接，默认为3000ms；（无状态的)  
- Cookie: JSESSIONID=369766FDF6220F7803433C0B2DE36D98：因为不是第一次访问这个地址，所以会在请求中把上一次服务器响应中发送过来的Cookie在请求中一并发送去过

# 5.31 POST请求

* 为了演示POST请求，我们需要修改index.jsp页面，即添加一个表单：
* 打开HttpWatch，输入hello后点击提交，查看请求内容如下：  
  POST /hello/index.jsp HTTP/1.1  
  Accept: image/gif, image/jpeg, image/pjpeg, image/pjpeg, application/msword, application/vnd.ms-excel, application/vnd.ms-powerpoint, application/x-ms-application, application/x-ms-xbap, application/vnd.ms-xpsdocument, application/xaml+xml, */*  
  Referer: http://localhost:8080/hello/index.jsp  
  Accept-Language: zh-cn,en-US;q=0.5  
  User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0; InfoPath.2; .NET CLR 2.0.50727; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729)  
  Content-Type: application/x-www-form-urlencoded  
  Accept-Encoding: gzip, deflate  
  Host: localhost:8080  
  Content-Length: 13  
  Connection: Keep-Alive  
  Cache-Control: no-cache  
  Cookie: JSESSIONID=E365D980343B9307023A1D271CC48E7D

keyword=hello

* POST请求是可以有体的，而GET请求不能有请求体。  
  - Referer: http://localhost:8080/hello/index.jsp：请求来自哪个页面，例如你在百度上点击链接到了这里，那么Referer:http://www.baidu.com；如果你是在浏览器的地址栏中直接输入的地址，那么就没有Referer这个请求头了；  
  - Content-Type: application/x-www-form-urlencoded：表单的数据类型，说明会使用url格式编码数据；url编码的数据都是以“%”为前缀，后面跟随两位的16进制，例如“传智”这两个字使用UTF-8的url编码用为“%E4%BC%A0%E6%99%BA”；  
  - Content-Length:13：请求体的长度，这里表示13个字节。  
  - keyword=hello：请求体内容！hello是在表单中输入的数据，keyword是表单字段的名字。
* Referer请求头是比较有用的一个请求头，它可以用来做统计工作，也可以用来做防盗链。
* 统计工作：我公司网站在百度上做了广告，但不知道在百度上做广告对我们网站的访问量是否有影响，那么可以对每个请求中的Referer进行分析，如果Referer为百度的很多，那么说明用户都是通过百度找到我们公司网站的。
* 防盗链：我公司网站上有一个下载链接，而其他网站盗链了这个地址，例如在我网站上的index.html页面中有一个链接，点击即可下载JDK7.0，但有某个人的微博中盗链了这个资源，它也有一个链接指向我们网站的JDK7.0，也就是说登录它的微博，点击链接就可以从我网站上下载JDK7.0，这导致我们网站的广告没有看，但下载的却是我网站的资源。这时可以使用Referer进行防盗链，在资源被下载之前，我们对Referer进行判断，如果请求来自本网站，那么允许下载，如果非本网站，先跳转到本网站看广告，然后再允许下载。

# 5.32 GET和POST请求的区别

* 幂等意味着对同一URL的多个请求应该返回同样的结果。
  + 1)前者将请求参数放在URL中，文本格式；后者将请求参数放在请求体中，可以是文本、二进制等格式
  + 2)前者语义上是从服务器获取资源，安全（无副作用)、幂等、可缓存；后者语义上是向服务器提交资源，不安全（有副作用)、不幂等、不可缓存
  + 3)前者的URL是明文传输，会保存在浏览器历史记录中，安全性不足，可能会受到CSRF攻击；后者较为安全（但是如果没有加密的话，都是可以明文获取的)

# 5.33 其他请求方法

* GET（SELECT)：从服务器取出资源（一项或多项)。
* POST（CREATE)：在服务器新建一个资源。
* PUT（UPDATE)：在服务器更新资源（客户端提供改变后的完整资源，数据输入必须与由 GET 接收的数据表示保持一致)。
* PATCH（UPDATE)：在服务器更新资源（客户端提供改变的属性)。
* DELETE（DELETE)：从服务器删除资源。
* HEAD：获取资源的元数据。
* OPTIONS：1、获取服务器支持的HTTP请求方法
* 2、用来检查服务器的性能

# 5.34 响应内容

* 响应协议的格式如下：  
  响应首行；由HTTP协议版本号， 状态码， 状态消息 三部分组成。  
  响应头信息；  
  空行；  
  响应体。
* 响应内容是由服务器发送给浏览器的内容，浏览器会根据响应内容来显示。  
  HTTP/1.1 200 OK  
  Server: Apache-Coyote/1.1  
  Content-Type: text/html;charset=UTF-8  
  Content-Length: 724  
  Set-Cookie: JSESSIONID=C97E2B4C55553EAB46079A4F263435A4; Path=/hello  
  Date: Wed, 25 Sep 2012 04:15:03 GMT

<title>My JSP 'index.jsp' starting page</title>  
<meta http-equiv="pragma" content="no-cache">  
<meta http-equiv="cache-control" content="no-cache">  
<meta http-equiv="expires" content="0">   
<meta http-equiv="keywords" content="keyword1,keyword2,keyword3">  
<meta http-equiv="description" content="This is my page">  
<!--  
<link rel="stylesheet" type="text/css" href="styles.css">  
-->

- HTTP/1.1 200 OK：响应协议为HTTP1.1，状态码为200，表示请求成功，OK是对状态码的解释；  
- Server: Apache-Coyote/1.1：服务器的版本信息；  
- Content-Type: text/html;charset=UTF-8：响应体使用的编码为UTF-8；  
- Content-Length: 724：响应体为724字节；  
- Set-Cookie: JSESSIONID=C97E2B4C55553EAB46079A4F263435A4; Path=/hello：响应给客户端的Cookie；  
- Date: Wed, 25 Sep 2012 04:15:03 GMT：响应的时间，这可能会有8小时的时区差；

# 5.35 响应码

* 响应头对浏览器来说很重要，它说明了响应的真正含义。例如200表示响应成功了，302表示重定向，这说明浏览器需要再发一个新的请求。
* 2xx表示成功，3xx表示重定向，4xx表示客户端出错，5xx表示服务器出错。  
  - 200：请求成功，浏览器会把响应体内容（通常是html)显示在浏览器中；  
  - 404：请求的资源没有找到，说明客户端错误的请求了不存在的资源；  
  - 500：请求资源找到了，但服务器内部出现了错误；  
  - 302：重定向，当响应码为302时，表示服务器要求浏览器重新再发一个请求，服务器会发送一个响应头Location，它指定了新请求的URL地址；  
  - 304：（缓存)

## 301&302

* 301 Move Permanently
* 302 Found
* 301是永久性重定向。当网站需要改版时，多域名指向同一个页面时，为了不让网站被降低和分散权重，就需要使用301重定向来实现，同时在搜索引擎索引库中彻底废弃掉原先的老地址。
* 302是临时性重定向，搜索引擎会抓取新的内容而保留旧的网址。因为服务器返回302代码，搜索引擎认为新的网址只是暂时的，不会传递权重。

# 5.36 其他响应头

* 告诉浏览器不要缓存的响应头：  
  - Expires: -1；（过期时间，-1表示马上过期)  
  - Cache-Control: no-cache；（不缓存)  
  - Pragma: no-cache；（不缓存)
* 自动刷新响应头，浏览器会在3秒之后请求http://www.baidu.com：  
  - Refresh: 3;url=http://www.baidu.com

# 5.37 HTML中指定响应头

* 在HTMl页面中可以使用来指定响应头，例如在index.html页面中给出，表示浏览器只会显示index.html页面3秒，然后自动跳转到http://www.itcast.cn。

# 缓存

# 5.38 强缓存与协商缓存

* 浏览器HTTP缓存可以分为强缓存和协商缓存。强缓存和协商缓存最大也是最根本的区别是：强缓存命中的话不会发请求到服务器（比如chrome中的200 from memory cache)，协商缓存一定会发请求到服务器，通过资源的请求首部字段验证资源是否命中协商缓存，如果协商缓存命中，服务器会将这个请求返回，但是不会返回这个资源的实体，而是通知客户端可以从缓存中加载这个资源（304 not modified)。

# 5.39 控制强缓存的字段按优先级介绍

1.- Pragma Pragma是HTTP/1.1之前版本遗留的通用首部字段，仅作为于HTTP/1.0的向后兼容而使用。虽然它是一个通用首部，但是它在响应报文中时的行为没有规范，依赖于浏览器的实现。RFC中该字段只有no-cache一个可选值，会通知浏览器不直接使用缓存，要求向服务器发请求校验新鲜度。因为它优先级最高，当存在时一定不会命中强缓存。  
2.- Cache-Control Cache-Control是一个通用首部字段，也是HTTP/1.1控制浏览器缓存的主流字段。和浏览器缓存相关的是如下几个响应指令：  
指令 参数 说明  
private 无 表明响应只能被单个用户缓存，不能作为共享缓存（即代理服务器不能缓存它)  
public 可省略 表明响应可以被任何对象（包括：发送请求的客户端，代理服务器，等等)缓存  
no-cache 可省略 缓存前必需确认其有效性  
no-store 无 不缓存请求或响应的任何内容  
max-age=[s] 必需 响应的最大值  
- max-age（单位为s)设置缓存的存在时间，相对于发送请求的时间。只有响应报文首部设置Cache-Control为非0的max-age或者设置了大于请求日期的Expires（下文会讲)才有可能命中强缓存。当满足这个条件，同时响应报文首部中Cache-Control不存在no-cache、no-store且请求报文首部不存在Pragma字段，才会真正命中强缓存。   
- no-cache 表示请求必须先与服务器确认缓存的有效性，如果有效才能使用缓存（协商缓存)，无论是响应报文首部还是请求报文首部出现这个字段均一定不会命中强缓存。Chrome硬性重新加载（Command+shift+R)会在请求的首部加上Pragma：no-cache和Cache-Control：no-cache。  
- no-store 表示禁止浏览器以及所有中间缓存存储任何版本的返回响应，一定不会出现强缓存和协商缓存，适合个人隐私数据或者经济类数据。  


public 表明响应可以被浏览器、CDN等等缓存。



private 响应只作为私有的缓存，不能被CDN等缓存。如果要求HTTP认证，响应会自动设置为private。

3.- Expires Expires是一个响应首部字段，它指定了一个日期/时间，在这个时间/日期之前，HTTP缓存被认为是有效的。无效的日期比如0，表示这个资源已经过期了。如果同时设置了Cache-Control响应首部字段的max-age，则Expires会被忽略。它也是HTTP/1.1之前版本遗留的通用首部字段，仅作为于HTTP/1.0的向后兼容而使用。

# 5.40 控制协商缓存的字段

1.- Last-Modified/If-Modified-Since If-Modified-Since是一个请求首部字段，并且只能用在GET或者HEAD请求中。Last-Modified是一个响应首部字段，包含服务器认定的资源作出修改的日期及时间。当带着If-Modified-Since头访问服务器请求资源时，服务器会检查Last-Modified，如果Last-Modified的时间早于或等于If-Modified-Since则会返回一个不带主体的304响应，否则将重新返回资源。

* If-Modified-Since: , :: GMT Last-Modified: , :: GMT  
  2.- ETag/If-None-Match ETag是一个响应首部字段，它是根据实体内容生成的一段hash字符串，标识资源的状态，由服务端产生。If-None-Match是一个条件式的请求首部。如果请求资源时在请求首部加上这个字段，值为之前服务器端返回的资源上的ETag，则当且仅当服务器上没有任何资源的ETag属性值与这个首部中列出的时候，服务器才会返回带有所请求资源实体的200响应，否则服务器会返回不带实体的304响应。ETag优先级比Last-Modified高，同时存在时会以ETag为准。
* 因为ETag的特性，所以相较于Last-Modified有一些优势：
  1. 某些情况下服务器无法获取资源的最后修改时间
  2. 资源的最后修改时间变了但是内容没变，使用ETag可以正确缓存
  3. 如果资源修改非常频繁，在秒以下的时间进行修改，Last-Modified只能精确到秒

# 5.41 协商缓存细节

* 当用户第一次请求index.html时，服务器会添加一个名为Last-Modified响应头，这个头说明了index.html的最后修改时间，浏览器会把index.html内容，以及最后响应时间缓存下来。当用户第二次请求index.html时，在请求中包含一个名为If-Modified-Since请求头，它的值就是第一次请求时服务器通过Last-Modified响应头发送给浏览器的值，即index.html最后的修改时间，If-Modified-Since请求头就是在告诉服务器，我这里浏览器缓存的index.html最后修改时间是这个，您看看现在的index.html最后修改时间是不是这个，如果还是，那么您就不用再响应这个index.html内容了，我会把缓存的内容直接显示出来。而服务器端会获取If-Modified-Since值，与index.html的当前最后修改时间比对，如果相同，服务器会发响应码304，表示index.html与浏览器上次缓存的相同，无需再次发送，浏览器可以显示自己的缓存页面，如果比对不同，那么说明index.html已经做了修改，服务器会响应200。（只有html等静态资源可以做缓存，动态资源不做缓存)
* 响应头：  
  - Last-Modified：最后的修改时间；
* 请求头：  
  - If-Modified-Since：把上次请求的index.html的最后修改时间还给服务器；
* 状态码：304，比较If-Modified-Since的时间与文件真实的时间一样时，服务器会响应304，而且不会有响应正文，表示浏览器缓存的就是最新版本！

# 幂等性（并非是HTTP的问题，而是服务器API设计问题)

* 幂等性是http层面的问题吗，还是服务器要处理和解决的内容？
* 对HTTP协议的使用实际上存在着两种不同的方式：一种是RESTful的，它把HTTP当成应用层协议，比较忠实地遵守了HTTP协议的各种规定；另一种是SOA的，它并没有完全把HTTP当成应用层协议，而是把HTTP协议作为了传输层协议，然后在HTTP之上建立了自己的应用层协议。这里所讨论的HTTP幂等性主要针对RESTful风格的，但幂等性并不属于特定的协议，它是分布式系统的一种特性；所以，不论是SOA还是RESTful的Web API设计都应该考虑幂等性。下面将介绍HTTP GET、DELETE、PUT、POST四种主要方法的语义和幂等性。
* HTTP GET方法用于获取资源，不应有副作用，所以是幂等的。
* 比如：GET http://www.bank.com/account/123456，不会改变资源的状态，不论调用一次还是N次都没有副作用。请注意，这里强调的是一次和N次具有相同的副作用，而不是每次GET的结果相同。GET http://www.news.com/latest-news这个HTTP请求可能会每次得到不同的结果，但它本身并没有产生任何副作用，因而是满足幂等性的。
* HTTP DELETE方法用于删除资源，有副作用，但它应该满足幂等性。
* 比如：DELETE http://www.forum.com/article/4231，调用一次和N次对系统产生的副作用是相同的，即删掉id为4231的帖子；因此，调用者可以多次调用或刷新页面而不必担心引起错误。
* 比较容易混淆的是HTTP POST和PUT。POST和PUT的区别容易被简单地误认为“POST表示创建资源，PUT表示更新资源”；而实际上，二者均可用于创建资源，更为本质的差别是在幂等性方面。在HTTP规范中对POST和PUT是这样定义的：POST所对应的URI并非创建的资源本身，而是资源的接收者。比如：POST http://www.forum.com/articles的语义是在http://www.forum.com/articles下创建一篇帖子，HTTP响应中应包含帖子的创建状态以及帖子的URI。两次相同的POST请求会在服务器端创建两份资源，它们具有不同的URI；所以，POST方法不具备幂等性。而PUT所对应的URI是要创建或更新的资源本身。比如：PUT http://www.forum/articles/4231的语义是创建或更新ID为4231的帖子。对同一URI进行多次PUT的副作用和一次PUT是相同的；因此，PUT方法具有幂等性。

# 无状态

* 客户端和服务器在某次会话中产生的数据，从而【无状态】就意味着，这些数据不会被保留；协议对于事务处理没有记忆能力，服务器不知道客户端是什么状态。但是通过增加cookie和session机制，现在的网络请求其实是有状态的。在没有状态的http协议下，服务器也一定会保留你每次网络请求对数据的修改，但这跟保留每次访问的数据是不一样的，保留的只是会话产生的结果，而没有保留会话。
* 与之相对的是TCP，TCP是有状态的，因为每一条消息的seq和ack（还有一堆滑动窗口，拥塞的控制参数，等)都和前面消息相关。
* HTTP并不会在内存里保留前次请求相关的任何状态，仅仅以协议逻辑（打包解包)存在，所以是它无状态的。
* 无状态的设计会加强透明度(visibility)，稳定度(reliability)和伸缩度(scalability)。提高透明度是因为系统无需通过请求内容以外的信息判断请求的完整内容；提高稳定度是指在部分失败的情况下，减轻了恢复的难度；提高伸缩度的原因是无需储存请求间的状态使服务器端可以很快释放资源并简化实现。
* 优点在于解放了服务器，每一次请求“点到为止”不会造成不必要连接占用，缺点在于每次请求会传输大量重复的内容信息。

# 跨域 CORS 跨域资源共享

* 之所以会跨域，是因为受到了同源策略的限制，同源策略要求源相同才能正常进行通信，即协议、域名、端口号都完全一致。
* 同源策略具体限制些什么呢？
  1. 不能向工作在不同源的的服务请求数据（client to server)这里有个问题之前也困扰了我很久，就是为什么home.com加载的cdn.home.com/index.js可以向home.com发请求而不会跨域呢？其实home.com加载的JS是工作在home.com的，它的源不是提供JS的cdn，所以这个时候是没有跨域的问题的，并且script标签能够加载非同源的资源，不受同源策略的影响。
  2. 无法获取不同源的document/cookie等BOM和DOM，可以说任何有关另外一个源的信息都无法得到 （client to client)。
* 跨域最常用的方法，应当属CORS，如下图所示：
* 只要浏览器检测到响应头带上了CORS，并且允许的源包括了本网站，那么就不会拦截请求响应。
* CORS把请求分为两种，一种是简单请求，另一种是需要触发预检请求，这两者是相对的，怎样才算“不简单”？只要属于下面的其中一种就不是简单请求：
  + （1)使用了除GET/POST/HEAD之外的请求方式，如PUT/DELETE
  + （2)使用了除Content-Type/Accept等几个常用的http头这个时候就认为需要先发个预检请求

# 5.42 简单请求

* 对于简单请求，浏览器直接发出CORS请求。具体来说，就是在头信息之中，增加一个Origin字段。
* 下面是一个例子，浏览器发现这次跨源AJAX请求是简单请求，就自动在头信息之中，添加一个Origin字段。
* GET /cors HTTP/1.1
* Origin: http://api.bob.com
* Host: api.alice.com
* Accept-Language: en-US
* Connection: keep-alive
* User-Agent: Mozilla/5.0...
* 上面的头信息中，Origin字段用来说明，本次请求来自哪个源（协议 + 域名 + 端口)。服务器根据这个值，决定是否同意这次请求。
* 如果Origin指定的源，不在许可范围内，服务器会返回一个正常的HTTP回应。浏览器发现，这个回应的头信息没有包含Access-Control-Allow-Origin字段（详见下文)，就知道出错了，从而抛出一个错误，被XMLHttpRequest的onerror回调函数捕获。注意，这种错误无法通过状态码识别，因为HTTP回应的状态码有可能是200。
* 如果Origin指定的域名在许可范围内，服务器返回的响应，会多出几个头信息字段。
* Access-Control-Allow-Origin: http://api.bob.com
* Access-Control-Allow-Credentials: true
* Access-Control-Expose-Headers: FooBar
* Content-Type: text/html; charset=utf-8
* 上面的头信息之中，有三个与CORS请求相关的字段，都以Access-Control-开头。
  + （1)Access-Control-Allow-Origin
* 该字段是必须的。它的值要么是请求时Origin字段的值，要么是一个\*，表示接受任意域名的请求。
  + （2)Access-Control-Allow-Credentials
* 该字段可选。它的值是一个布尔值，表示是否允许发送Cookie。默认情况下，Cookie不包括在CORS请求之中。设为true，即表示服务器明确许可，Cookie可以包含在请求中，一起发给服务器。这个值也只能设为true，如果服务器不要浏览器发送Cookie，删除该字段即可。
  + （3)Access-Control-Expose-Headers
* 该字段可选。CORS请求时，XMLHttpRequest对象的getResponseHeader()方法只能拿到6个基本字段：Cache-Control、Content-Language、Content-Type、Expires、Last-Modified、Pragma。如果想拿到其他字段，就必须在Access-Control-Expose-Headers里面指定。上面的例子指定，getResponseHeader('FooBar')可以返回FooBar字段的值。

# 5.43 非简单请求

* 简单请求是那种对服务器有特殊要求的请求，比如请求方法是PUT或DELETE，或者Content-Type字段的类型是application/json。
* 非简单请求的CORS请求，会在正式通信之前，增加一次HTTP查询请求，称为"预检"请求（preflight)。
* 浏览器先询问服务器，当前网页所在的域名是否在服务器的许可名单之中，以及可以使用哪些HTTP动词和头信息字段。只有得到肯定答复，浏览器才会发出正式XMLHttpRequest请求，否则就报错。
* "预检"请求用的请求方法是OPTIONS，表示这个请求是用来询问的。头信息里面，关键字段是Origin，表示请求来自哪个源。
* 除了Origin字段，"预检"请求的头信息包括两个特殊字段。
  + （1)Access-Control-Request-Method
* 该字段是必须的，用来列出浏览器的CORS请求会用到哪些HTTP方法
  + （2)Access-Control-Request-Headers
* 该字段是一个逗号分隔的字符串，指定浏览器CORS请求会额外发送的头信息字段
* 服务器收到"预检"请求以后，检查了Origin、Access-Control-Request-Method和Access-Control-Request-Headers字段以后，确认允许跨源请求，就可以做出回应。
* 如果浏览器否定了"预检"请求，会返回一个正常的HTTP回应，但是没有任何CORS相关的头信息字段。这时，浏览器就会认定，服务器不同意预检请求，因此触发一个错误，被XMLHttpRequest对象的onerror回调函数捕获。
* 一旦服务器通过了"预检"请求，以后每次浏览器正常的CORS请求，就都跟简单请求一样，会有一个Origin头信息字段。服务器的回应，也都会有一个Access-Control-Allow-Origin头信息字段。
* CORS与JSONP的使用目的相同，但是比JSONP更强大。
* JSONP只支持GET请求，CORS支持所有类型的HTTP请求。JSONP的优势在于支持老式浏览器，以及可以向不支持CORS的网站请求数据。

# 长轮询与短轮询

* 短轮询相信大家都不难理解，比如你现在要做一个电商中商品详情的页面，这个详情界面中有一个字段是库存量（相信这个大家都不陌生，随便打开淘宝或者京东都能找到这种页面)。而这个库存量需要实时的变化，保持和服务器里实际的库存一致。
* 这个时候，你会怎么做？
* 最简单的一种方式，就是你用JS写个死循环，不停的去请求服务器中的库存量是多少，然后刷新到这个页面当中，这其实就是所谓的短轮询。
* 这种方式有明显的坏处，那就是你很浪费服务器和客户端的资源。客户端还好点，现在PC机配置高了，你不停的请求还不至于把用户的电脑整死，但是服务器就很蛋疼了。如果有1000个人停留在某个商品详情页面，那就是说会有1000个客户端不停的去请求服务器获取库存量，这显然是不合理的。
* 那怎么办呢？
* 长轮询这个时候就出现了，其实长轮询和短轮询最大的区别是，短轮询去服务端查询的时候，不管库存量有没有变化，服务器就立即返回结果了。而长轮询则不是，在长轮询中，服务器如果检测到库存量没有变化的话，将会把当前请求挂起一段时间（这个时间也叫作超时时间，一般是几十秒,Object.wait)。在这个时间里，服务器会去检测库存量有没有变化，检测到变化就立即返回（Object.notify)，否则就一直等到超时为止。
* 而对于客户端来说，不管是长轮询还是短轮询，客户端的动作都是一样的，就是不停的去请求，不同的是服务端，短轮询情况下服务端每次请求不管有没有变化都会立即返回结果，而长轮询情况下，如果有变化才会立即返回结果，而没有变化的话，则不会再立即给客户端返回结果，直到超时为止。
* 这样一来，客户端的请求次数将会大量减少（这也就意味着节省了网络流量，毕竟每次发请求，都会占用客户端的上传流量和服务端的下载流量)，而且也解决了服务端一直疲于接受请求的窘境。
* 但是长轮询也是有坏处的，因为把请求挂起同样会导致资源的浪费，假设还是1000个人停留在某个商品详情页面，那就很有可能服务器这边挂着1000个线程，在不停检测库存量，这依然是有问题的。
* 因此，从这里可以看出，不管是长轮询还是短轮询，都不太适用于客户端数量太多的情况，因为每个服务器所能承载的TCP连接数是有上限的，这种轮询很容易把连接数顶满。

# 长连接与短连接

* HTTP的短连接和长连接；长连接与短连接的区别（LVS是通过长连接作负载均衡)
* HTTP的长连接和短连接本质上是TCP长连接和短连接。
* 在HTTP/1.0中，默认使用的是短连接。也就是说，浏览器和服务器每进行一次HTTP操作，就建立一次连接，但任务结束就中断连接。
* 但从 HTTP/1.1起，默认使用长连接，用以保持连接特性。使用长连接的HTTP协议，会在响应头有加入这行代码：Connection:keep-alive。
* 在使用长连接的情况下，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。实现长连接要客户端和服务端都支持长连接。
* 长连接可以省去较多的TCP建立和关闭的操作，减少浪费，节约时间。对于频繁请求资源的客户来说，较适用长连接。不过这里存在一个问题，存活功能的探测周期太长，还有就是它只是探测TCP连接的存活，属于比较斯文的做法，遇到恶意的连接时，保活功能就不够使了。在长连接的应用场景下，client端一般不会主动关闭它们之间的连接，Client与server之间的连接如果一直不关闭的话，会存在一个问题，随着客户端连接越来越多，server早晚有扛不住的时候，这时候server端需要采取一些策略，如关闭一些长时间没有读写事件发生的连接，这样可以避免一些恶意连接导致server端服务受损。
* 短连接对于服务器来说管理较为简单，存在的连接都是有用的连接，不需要额外的控制手段。但如果客户请求频繁，将在TCP的建立和关闭操作上浪费时间和带宽。

# URL

* url有最大长度限制，就问长度有限制是get的原因还是url的原因，为什么长度会有限制，是http数据包的头的字段原因还是内容字段的原因
* 是GET的原因，长度受到服务器和客户端的限制。
* URL编解码
* Url的编码格式采用的是ASCII码，而不是Unicode，这也就是说你不能在Url中包含任何非ASCII字符，例如中文。
  + Url中只允许包含英文字母（a-zA-Z)、数字（0-9)、-\_.~4个特殊字符以及所有保留字符；
* RFC3986中指定了以下字符为保留字符：! \* ' ( ) ; : @ & = + $ , / ? # [ ]
* Url编码通常也被称为百分号编码（Url Encoding，also known as percent-encoding)，是因为它的编码方式非常简单，使用%百分号加上两位的十六进制字符。

# URI&URL

* URL（Uniform ResourceLocator)统一资源定位符，是专门为标识网络上的资源位置而设计的一种编址方式。URL一般由3个部分组成：
* 应用层协议
* 主机IP地址或域名
* 资源所在路径/文件名
* 统一资源标识符（Uniform Resource Identifier，或URI)是一个用于标识某一互联网资源名称的字符串。
* URI ：Uniform Resource Identifier，统一资源标识符；
* URL：Uniform Resource Locator，统一资源定位符；
* URN：Uniform ResourceName，统一资源名称。
* 其中，URL,URN是URI的子集。
* URL是一种具体的URI，它不仅唯一标识资源，而且还提供了定位该资源的信息。URI是一种语义上的抽象概念，可以是绝对的，也可以是相对的，而URL则必须提供足够的信息来定位。

# HTTPS

# 5.44 HTTP缺点

- 明文传输，内容可能会被窃听  
- 不验证通信方的身份，因此有可能遭遇伪装  
- 无法证明报文的完整性，所以有可能已遭篡改

* HTTP+加密+认证+完整性保护=HTTPS
* 用SSL将通信的报文主体内容进行加密，使用SSL建立http的安全通信线路，SSL处于http与TCP通信之间，这样的SSL与HTTP组合被称为HTTPS。
* HTTPS 采用对称加密和非对称加密两者并用的混合加密机制
* HTTPS 公钥能用公钥解吗？在客户端抓包，看到的是加密的还是没加密 是没加密的
* https ssl tcp三者关系，其中哪些用到了对称加密，哪些用到了非对称加密，非对称加密密钥是如何实现的
* 加密的私钥和公钥各自如何分配（客户端拿公钥，服务器拿私钥)
* 客户端是如何认证服务器的真实身份，详细说明一下过程，包括公钥如何申请，哪一层加密哪一层解密
* 怎么攻击https
* TLS改进，如果session ticket被偷听到会怎样，如何防止中间人攻击

# 5.45 SSL/TLS

* SSL（Secure Socket Layer安全套接字层)
* TLS（Transport Layer Security)
* SSL发展到3.0版本后改成了TLS。
* TLS主要提供三个基本服务  
  - 加密  
  - 身份验证  
  - 消息完整性校验
* 通常，HTTP 直接和 TCP 通信。当使用 SSL 时，则演变成先和 SSL 通信，再由 SSL 和 TCP 通信了。用 SSL 建立安全通信线路之后，就可以在这条线路上进行 HTTP 通信了。
* SSL 是独立于 HTTP 的协议，所以不光是 HTTP 协议，其他运行在应用层的 SMTP 和 Telnet 等协议均可配合 SSL 协议使用。可以说 SSL 是当今世界上应用最为广泛的网络安全技术。
* 虽然使用 HTTP 协议无法确定通信方，但如果使用 SSL 则可以。SSL 不仅提供加密处理，而且还使用了一种被称为证书的手段，可用于确定双方身份。
* 证书由值得信任的第三方机构颁发，用以证明服务器和客户端是实际存在的。另外，伪造证书从技术角度来说是异常困难的一件事。所以只要能够确认通信方(服务器或客户端)持有的证书，即可判断通信方的真实意图。

# 5.46 中间人攻击

* mim 就是man in the middle，中间人攻击正常情况下浏览器与服务器在TLS连接下内容是加密的，第三方即使可以嗅探到所有的数据，也不能解密。中间人可以与你建立连接，然后中间人再与服务器建立连接，转发你们之间的内容。这时候中间人就获得了明文的信息。
* 有什么危害？你与服务器的通信被第三方解密、查看、修改。如何防范？如果确定是否被攻击？在访问https连接的时候，查看一下服务器提供的证书是不是正确的。除非入侵并取得服务器的证书私钥，否则中间人是不能完全伪装成服务器的样子的。
* 数字证书可以保证服务器发来的公钥是真的来自服务器的

# 5.47 服务器保证其提供的公钥的正确性——数字证书

* 公钥是由数字证书认证机构(CA，Certificate Authority)和其相关机关颁发的公开密钥证书。
* 数字证书认证机构处于客户端与服务器双方都可信赖的第三方机构的立场上。服务器会将这份由数字证书认证机构颁发的公钥证书发送给客户端，以进行公开密钥加密方式通信。公钥证书也可叫做数字证书或直接称为证书。
* 接到证书的客户端可使用数字证书认证机构的公开密钥，对那张证书上的数字签名进行验证，一旦验证通过，客户端便可明确两件事：
* 认证服务器的公开密钥的是真实有效的数字证书认证机构
* 服务器的公开密钥是值得信赖的
* 此处认证机关的公开密钥必须安全地转交给客户端。使用通信方式时，如何安全转交是一件很困难的事，因此，多数浏览器开发商发布版本时，会事先在内部植入常用认证机关的公开密钥。

# 5.48 过程

*  客户端发起HTTPS请求 这个没什么好说的，就是用户在浏览器里输入一个HTTPS网址，然后连接到服务端的443端口。
*  服务端的配置 采用HTTPS协议的服务器必须要有一套数字证书，可以自己制作，也可以向组织申请。区别就是自己颁发的证书需要客户端验证通过，才可以继续访问，而使用受信任的公司申请的证书则不会弹出提示页面。这套证书其实就是一对公钥和私钥。
*  传送证书 这个证书其实就是公钥，只是包含了很多信息，如证书的颁发机构，过期时间等等。
*  客户端解析证书 这部分工作是由客户端的SSL/TLS来完成的，首先会验证公钥是否有效，比如颁发机构，过期时间等等，如果发现异常，则会弹出一个警示框，提示证书存在的问题。如果证书没有问题，那么就生成一个随机值。然后用证书（也就是公钥)对这个随机值进行加密。
*  传送加密信息 这部分传送的是用证书加密后的随机值，目的是让服务端得到这个随机值，以后客户端和服务端的通信就可以通过这个随机值来进行加密解密了。
*  服务端解密信息 服务端用私钥解密后，得到了客户端传过来的随机值，然后把内容通过该随机值（密钥)进行对称加密，将信息和私钥通过某种算法混合在一起，这样除非知道私钥，不然无法获取内容，而正好客户端和服务端都知道这个私钥，所以只要加密算法够复杂，私钥够复杂，数据就够安全。
*  传输加密后的信息 这部分信息就是服务端用私钥加密后的信息，可以在客户端用随机值解密还原。
*  客户端解密信息 客户端用之前生产的私钥解密服务端传过来的信息，于是获取了解密后的内容。整个过程第三方即使监听到了数据，也束手无策。
* 客户端获得服务器的公钥的过程是基于非对称加密实现的（数字证书)
* 而之后客户端和服务器之间的数据交换是基于对称加密实现的。

# 5.49 更具体的过程

*  客户端通过发送 Client Hello 报文开始 SSL 通信。报文中包含客户端支持的 SSL 的指定版本、加密组件(Cipher Suite)列表(所使用的加密算法及密钥长度等)
*  服务器可进行 SSL 通信时，会以 Server Hello 报文作为应答。和客户端一样，在报文中包含 SSL 版本以及加密组件。服务器的加密组件内容是从接收 到的客户端加密组件内筛选出来的。
*  之后服务器发送 Certificate 报文。报文中包含公开密钥证书。
*  最后服务器发送 Server Hello Done 报文通知客户端，最初阶段的 SSL 握手协商部分结束。
*  SSL 第一次握手结束之后，客户端以 Client Key Exchange 报文作为回应。报文中包含通信加密中使用的一种被称为 Pre-master secret 的随机密码串。该 报文已用步骤 3 中的公开密钥进行加密。
*  接着客户端继续发送 Change Cipher Spec 报文。该报文会提示服务器，在此报文之后的通信会采用 Pre-master secret 密钥加密。
*  客户端发送 Finished 报文。该报文包含连接至今全部报文的整体校验值。这次握手协商是否能够成功，要以服务器是否能够正确解密该报文作为判定标准。
*  服务器同样发送 Change Cipher Spec 报文。
*  服务器同样发送 Finished 报文。
*  服务器和客户端的 Finished 报文交换完毕之后，SSL 连接就算建立完成。当然，通信会受到 SSL 的保护。从此处开始进行应用层协议的通信，即发 送 HTTP 请求。
*  应用层协议通信，即发送 HTTP 响应。
*  最后由客户端断开连接。断开连接时，发送 close\_notify 报文。

# WebSocket

* web浏览器和web服务器之间全双工通信标准。
* 优点是，直接发送数据，不用等待客户端请求，一直保持连接状态，且首部信息量少，通信量减少。

# HTTP 2.0

# 5.50 二进制分帧

* 在应用层(HTTP2.0)和传输层(TCP or UDP)之间增加一个二进制分帧层。
* 在二进制分帧层上， HTTP 2.0 会将所有传输的信息分割为更小的消息和帧,并对它们采用二进制格式的编码。Frame 由 Frame Header 和 Frame Payload 两部分组成。不论是原来的 HTTP Header 还是 HTTP Body，在 HTTP/2 中，都将这些数据存储到 Frame Payload，组成一个个 Frame，再发送响应/请求。通过 Frame Header 中的 Type 区分这个 Frame 的类型。由此可见语义并没有太大变化，而是数据的格式变成二进制的 Frame。
* HTTP 2.0 通信都在一个连接上完成，这个连接可以承载任意数量的双向数据流。相应地，每个数据流以消息的形式发送，而消息由一或多个帧组成，这些帧可以乱序发送，然后再根据每个帧首部的流标识符重新组装。

# 5.51 首部压缩

* HTTP 2.0 在客户端和服务器端使用“首部表”来跟踪和存储之前发送的键-值对，对于相同的数据，不再通过每次请求和响应发送;通信期间几乎不会改变的通用键-值对(用户代理、可接受的媒体类型,等等)只需发送一次。事实上,如果请求中不包含首部(例如对同一资源的轮询请求),那么 首部开销就是零字节。此时所有首部都自动使用之前请求发送的首部。
* 如果首部发生变化了，那么只需要发送变化了数据在Headers帧里面，新增或修改的首部帧会被追加到“首部表”。首部表在 HTTP 2.0 的连接存续期内始终存在,由客户端和服务器共同渐进地更新 。
* HTTP/2 使用了专门为首部压缩而设计的 HPACK 算法。

# 5.52 服务器推送

* HTTP 2.0 新增的一个强大的新功能，就是服务器可以对一个客户端请求发送多个响应。换句话说，服务器除了对最初请求的响应外，还可以额外向客户端推送资源，而无需客户端明确地请求。
* 当浏览器请求一个html，服务器其实大概知道你是接下来要请求资源了，而不需要等待浏览器得到html后解析页面再发送资源请求。我们常用的内嵌图片也可以理解为一种强制的服务器推送：我请求html，却内嵌了张图。
* 有了HTTP2.0的服务器推送，HTTP1.x时代的内嵌资源的优化手段也变得没有意义了。而且使用服务器推送的资源的方式更加高效，因为客户端还可以缓存起来，甚至可以由不同的页面共享（依旧遵循同源策略)。当然，浏览器是可以决绝服务器推送的资源的。

# 5.53 多路复用

* 多路复用允许同时通过单一的HTTP/2连接发起多重的请求-响应信息。
* 每个 Frame Header 都有一个 Stream ID 就是被用于实现该特性。每次请求/响应使用不同的 Stream ID。就像同一个 TCP 链接上的数据包通过 IP:PORT来区分出数据包去往哪里一样。通过 Stream ID 标识，所有的请求和响应都可以欢快的同时跑在一条 TCP 链接上了。

# SOA

* Web Service也叫XML Web Service WebService是一种可以接收从Internet或者Intranet上的其它系统中传递过来的请求，轻量级的独立的通讯技术。是通过SOAP在Web上提供的软件服务，使用WSDL文件进行说明，并通过UDDI进行注册。
* SOA是一种架构风格，包括两个方面的内容：
  + 1)抽象出服务，这些服务满足离散、松耦合、可复用、自治、无状态等特征；
  + 2)服务可以灵活地组装和编排，满足流程整合和业务变化的需要
* WebService是SOA的一种实现技术，跨语言，跨平台，提供了标准的服务定义、服务注册、服务接入和访问的方式。使用了XML、SOAP、WSDL、UDDI等技术。

# SOA三角操作模型

- 1)三种角色

* 服务提供者：发布自己的服务，并且对服务请求进行响应
* 服务请求者：利用服务注册中心查找所需要的服务，然后使用该服务
* 服务注册中心：注册已经发布的服务，对其进行分类，并提供搜索服务
  + 2)三个操作：
* 发布：为了使服务可访问，需要发布服务描述以使服务使用者可以发现它
* 查找：服务请求者查询服务注册中心来找到满足其要求的服务
* 绑定：检索到服务描述后，服务请求者继续根据服务描述中的信息调用服务

# XML

* XML：(Extensible Markup Language)扩展型可标记语言。面向短期的临时数据处理、面向万维网络，是Soap的基础。

# SOAP

* SOAP：(Simple Object Access Protocol)简单对象传输协议。是XML Web Service 的通信协议。当用户通过UDDI找到你的WSDL描述文档后，他通过可以SOAP调用你建立的Web服务中的一个或多个操作。SOAP是XML文档形式的调用方法的规范，它可以支持不同的底层接口，像HTTP(S)或者SMTP。
* SOAP=RPC+HTTP+XML：采用HTTP作为底层通讯协议；RPC作为一致性的调用途径，ＸＭＬ作为数据传送的格式，允许服务提供者和服务客户经过防火墙在INTERNET进行通讯交互。
* 简单对象传输协议，是轻量级的、简单的、基于XML的用于交换数据的协议。
* SOAP本质上是一个 XML文档，包含以下元素：
  + 1)Envelope元素：必需元素，根元素，标识此XML文档为一条SOAP消息
* 可以包含命名空间和声明额外的属性
  + 2)Header元素：可选元素，有关SOAP消息的应用程序专用消息
  + 3)Body元素：必需元素，包含所有的请求和响应信息
  + 4)Fault元素：可选元素，提供有关在处理此消息所发生错误的信息
* SOAP处理模型：
  + 1)用XML打包请求
  + 2)将请求发送给服务器
  + 3)服务器接收到请求，解码XML，处理请求，以XML格式返回响应
* SOAP并不假定传输数据的下层协议，因此必须设计为能在各种协议上运行。即使绝大多数SOAP是运行在HTTP上，使用URI标识服务，SOAP也仅仅使用POST方法发送请求，用一个唯一的URI标识服务的入口。
* 使用 HTTP 协议的 SOAP，由于其设计原则上并不像 REST 那样强调与 Web 的工作方式相一致，所以，基于 SOAP 应用很难充分发挥 HTTP 本身的缓存能力。
* HTTP是其通信协议/传输协议，SOAP是其应用协议

# WSDL

* WSDL：(Web Services Description Language) WSDL 文件是一个 XML 文档，用于说明一组 SOAP 消息以及如何交换这些消息。大多数情况下由软件自动生成和使用。
* 网络服务描述语言，是基于XML的，用于描述网络服务、服务定位和服务提供的操作的协议。

# UDDI

* UDDI (Universal Description, Discovery, and Integration) 是一个主要针对Web服务供应商和使用者的新项目。在用户能够调用Web服务之前，必须确定这个服务内包含哪些商务方法，找到被调用的接口定义，还要在服务端来编制软件，UDDI是一种根据描述文档来引导系统查找相应服务的机制。UDDI利用SOAP消息机制（标准的XML/HTTP)来发布，编辑，浏览以及查找注册信息。它采用XML格式来封装各种不同类型的数据，并且发送到注册中心或者由注册中心来返回需要的数据。
* 统一描述、发现、集成协议，提供基于网络服务的注册和发现机制

# REST

* SOAP协议属于复杂的、重量级的协议，当前随着Web2.0的兴起，表述性状态转移（Representational State Transfer，REST)逐步成为一个流行的架构风格。REST是一种轻量级的Web Service架构风格，其实现和操作比SOAP和XML-RPC更为简洁，可以完全通过HTTP协议实现，还可以利用缓存Cache来提高响应速度，性能、效率和易用性上都优于SOAP协议。REST架构对资源的操作包括获取、创建、修改和删除资源的操作正好对应HTTP协议提供的GET、POST、PUT和DELETE方法，这种针对网络应用的设计和开发方式，可以降低开发的复杂性，提高系统的可伸缩性。REST架构尤其适用于完全无状态的CRUD（Create、Read、Update、Delete，创建、读取、更新、删除)操作。
* REST简单而直观，把HTTP协议利用到了极限，在这种思想指导下，它甚至用HTTP请求的头信息来指明资源的表示形式（如果一个资源有多种形式的话，例如人类友善的页面还是机器可读的数据？)，用HTTP的错误机制来返回访问资源的错误。由此带来的直接好处是构建的成本减少了，例如用URI定位每一个资源可以利用通用成熟的技术，而不用再在服务器端开发一套资源访问机制。又如只需简单配置服务器就能规定资源的访问权限，例如通过禁止非GET访问把资源设成只读。
* 1．面向资源的接口设计
* 所有的接口设计都是针对资源来设计的，也就很类似于我们的面向对象和面向过程的设计区别，只不过现在将网络上的操作实体都作为资源来看待，同时URI的设计也是体现了对于资源的定位设计。后面会提到有一些网站的API设计说是REST设计，其实是RPC-REST的混合体，并非是REST的思想。
* 2．抽象操作为基础的CRUD
* 这点很简单，Http中的get,put,post,delete分别对应了read,update,create,delete四种操作，如果仅仅是作为对于资源的操作，抽象成为这四种已经足够了，但是对于现在的一些复杂的业务服务接口设计，可能这样的抽象未必能够满足。其实这也在后面的几个网站的API设计中暴露了这样的问题，如果要完全按照REST的思想来设计，那么适用的环境将会有限制，而非放之四海皆准的。
* 3．Http是应用协议而非传输协议
* 这点在后面各大网站的API分析中有很明显的体现，其实有些网站已经走到了SOAP的老路上，说是REST的理念设计，其实是作了一套私有的SOAP协议，因此称之为REST风格的自定义SOAP协议。
* 4．无状态，自包含
* 这点其实不仅仅是对于REST来说的，作为接口设计都需要能够做到这点，也是作为可扩展和高效性的最基本的保证，就算是使用SOAP的WebService也是一样。

# Git

# git init

* 在本地新建一个repo,进入一个项目目录,执行git init,会初始化一个repo,并在当前文件夹下创建一个.git文件夹.

# git clone

* 获取一个url对应的远程Git repo, 创建一个local copy.
* 一般的格式是git clone [url].
* clone下来的repo会以url最后一个斜线后面的名称命名,创建一个文件夹,如果想要指定特定的名称,可以git clone [url] newname指定.

# git status

* 查询repo的状态.
* git status -s: -s表示short, -s的输出标记会有两列,第一列是对staging区域而言,第二列是对working目录而言.

# git log

* show commit history of a branch.
* git log --oneline --number: 每条log只显示一行,显示number条.
* git log --oneline --graph:可以图形化地表示出分支合并历史.
* git log branchname可以显示特定分支的log.
* git log --oneline branch1 ^branch2,可以查看在分支1,却不在分支2中的提交.^表示排除这个分支(Window下可能要给^branch2加上引号).
* git log --decorate会显示出tag信息.
* git log --author=[author name] 可以指定作者的提交历史.
* git log --since --before --until --after 根据提交时间筛选log.
* --no-merges可以将merge的commits排除在外.
* git log --grep 根据commit信息过滤log: git log --grep=keywords
* 默认情况下, git log --grep --author是OR的关系,即满足一条即被返回,如果你想让它们是AND的关系,可以加上--all-match的option.
* git log -S: filter by introduced diff.
* 比如: git log -SmethodName (注意S和后面的词之间没有等号分隔).
* git log -p: show patch introduced at each commit.
* 每一个提交都是一个快照(snapshot),Git会把每次提交的diff计算出来,作为一个patch显示给你看.
* 另一种方法是git show [SHA].
* git log --stat: show diffstat of changes introduced at each commit.
* 同样是用来看改动的相对信息的,--stat比-p的输出更简单一些.

# git add

* 在提交之前,Git有一个暂存区(staging area),可以放入新添加的文件或者加入新的改动. commit时提交的改动是上一次加入到staging area中的改动,而不是我们disk上的改动.
* git add .
* 会递归地添加当前工作目录中的所有文件.

# git diff

* 不加参数的git diff:
* show diff of unstaged changes.
* 此命令比较的是工作目录中当前文件和暂存区域快照之间的差异,也就是修改之后还没有暂存起来的变化内容.
* 若要看已经暂存起来的文件和上次提交时的快照之间的差异,可以用:
* git diff --cached 命令.
* show diff of staged changes.
* (Git 1.6.1 及更高版本还允许使用 git diff --staged，效果是相同的).
* git diff HEAD
* show diff of all staged or unstated changes.
* 也即比较woking directory和上次提交之间所有的改动.
* 如果想看自从某个版本之后都改动了什么,可以用:
* git diff [version tag]
* 跟log命令一样,diff也可以加上--stat参数来简化输出.
* git diff [branchA] [branchB]可以用来比较两个分支.
* 它实际上会返回一个由A到B的patch,不是我们想要的结果.
* 一般我们想要的结果是两个分支分开以后各自的改动都是什么,是由命令:
* git diff [branchA]…[branchB]给出的.
* 实际上它是:git diff $(git merge-base [branchA] [branchB]) [branchB]的结果.

# git commit

* 提交已经被add进来的改动.
* git commit -m “the commit message"
* git commit -a 会先把所有已经track的文件的改动add进来,然后提交(有点像svn的一次提交,不用先暂存). 对于没有track的文件,还是需要git add一下.
* git commit --amend 增补提交. 会使用与当前提交节点相同的父节点进行一次新的提交,旧的提交将会被取消.

# git reset

* undo changes and commits.
* 这里的HEAD关键字指的是当前分支最末梢最新的一个提交.也就是版本库中该分支上的最新版本.
* git reset HEAD: unstage files from index and reset pointer to HEAD
* 这个命令用来把不小心add进去的文件从staged状态取出来,可以单独针对某一个文件操作: git reset HEAD - - filename, 这个- - 也可以不加.
* git reset --soft
* move HEAD to specific commit reference, index and staging are untouched.
* git reset --hard
* unstage files AND undo any changes in the working directory since last commit.
* 使用git reset —hard HEAD进行reset,即上次提交之后,所有staged的改动和工作目录的改动都会消失,还原到上次提交的状态.
* 这里的HEAD可以被写成任何一次提交的SHA-1.
* 不带soft和hard参数的git reset,实际上带的是默认参数mixed.
* 总结:
* git reset --mixed id,是将git的HEAD变了(也就是提交记录变了),但文件并没有改变，(也就是working tree并没有改变). 取消了commit和add的内容.
* git reset --soft id. 实际上，是git reset –mixed id 后,又做了一次git add.即取消了commit的内容.
* git reset --hard id.是将git的HEAD变了,文件也变了.
* 按改动范围排序如下:
* soft (commit) < mixed (commit + add) < hard (commit + add + local working)

# git revert

* 反转撤销提交.只要把出错的提交(commit)的名字(reference)作为参数传给命令就可以了.
* git revert HEAD: 撤销最近的一个提交.
* git revert会创建一个反向的新提交,可以通过参数-n来告诉Git先不要提交.

# git rm

* git rm file: 从staging区移除文件,同时也移除出工作目录.
* git rm --cached: 从staging区移除文件,但留在工作目录中.
* git rm --cached从功能上等同于git reset HEAD,清除了缓存区,但不动工作目录树.

# git clean

* git clean是从工作目录中移除没有track的文件.
* 通常的参数是git clean -df:
* -d表示同时移除目录,-f表示force,因为在git的配置文件中, clean.requireForce=true,如果不加-f,clean将会拒绝执行.

# git mv

* git rm - - cached orig; mv orig new; git add new

# git stash

* 把当前的改动压入一个栈.
* git stash将会把当前目录和index中的所有改动(但不包括未track的文件)压入一个栈,然后留给你一个clean的工作状态,即处于上一次最新提交处.
* git stash list会显示这个栈的list.
* git stash apply:取出stash中的上一个项目(stash@{0}),并且应用于当前的工作目录.
* 也可以指定别的项目,比如git stash apply stash@{1}.
* 如果你在应用stash中项目的同时想要删除它,可以用git stash pop
* 删除stash中的项目:
* git stash drop: 删除上一个,也可指定参数删除指定的一个项目.
* git stash clear: 删除所有项目.

# git branch

* git branch可以用来列出分支,创建分支和删除分支.
* git branch -v可以看见每一个分支的最后一次提交.
* git branch: 列出本地所有分支,当前分支会被星号标示出.
* git branch (branchname): 创建一个新的分支(当你用这种方式创建分支的时候,分支是基于你的上一次提交建立的).
* git branch -d (branchname): 删除一个分支.
* 删除remote的分支:
* git push (remote-name) :(branch-name): delete a remote branch.
* 这个是因为完整的命令形式是:
* git push remote-name local-branch:remote-branch
* 而这里local-branch的部分为空,就意味着删除了remote-branch

# git checkout

* git checkout (branchname)
* 切换到一个分支.
* git checkout -b (branchname): 创建并切换到新的分支.
* 这个命令是将git branch newbranch和git checkout newbranch合在一起的结果.
* checkout还有另一个作用:替换本地改动:
* git checkout --
* 此命令会使用HEAD中的最新内容替换掉你的工作目录中的文件.已添加到暂存区的改动以及新文件都不会受到影响.
* 注意:git checkout filename会删除该文件中所有没有暂存和提交的改动,这个操作是不可逆的.

# git merge

* 把一个分支merge进当前的分支.
* git merge [alias]/[branch]
* 把远程分支merge到当前分支.
* 如果出现冲突,需要手动修改,可以用git mergetool.
* 解决冲突的时候可以用到git diff,解决完之后用git add添加,即表示冲突已经被resolved.

# git tag

* tag a point in history as import.
* 会在一个提交上建立永久性的书签,通常是发布一个release版本或者ship了什么东西之后加tag.
* 比如: git tag v1.0
* git tag -a v1.0, -a参数会允许你添加一些信息,即make an annotated tag.
* 当你运行git tag -a命令的时候,Git会打开一个编辑器让你输入tag信息.
* 我们可以利用commit SHA来给一个过去的提交打tag:
* git tag -a v0.9 XXXX
* push的时候是不包含tag的,如果想包含,可以在push时加上--tags参数.
* fetch的时候,branch HEAD可以reach的tags是自动被fetch下来的, tags that aren’t reachable from branch heads will be skipped.如果想确保所有的tags都被包含进来,需要加上--tags选项.

# git remote

* list, add and delete remote repository aliases.
* 因为不需要每次都用完整的url,所以Git为每一个remote repo的url都建立一个别名,然后用git remote来管理这个list.
* git remote: 列出remote aliases.
* 如果你clone一个project,Git会自动将原来的url添加进来,别名就叫做:origin.
* git remote -v:可以看见每一个别名对应的实际url.
* git remote add [alias] [url]: 添加一个新的remote repo.
* git remote rm [alias]: 删除一个存在的remote alias.
* git remote rename [old-alias] [new-alias]: 重命名.
* git remote set-url [alias] [url]:更新url. 可以加上—push和fetch参数,为同一个别名set不同的存取地址.

# git fetch

* download new branches and data from a remote repository.
* 可以git fetch [alias]取某一个远程repo,也可以git fetch --all取到全部repo
* fetch将会取到所有你本地没有的数据,所有取下来的分支可以被叫做remote branches,它们和本地分支一样(可以看diff,log等,也可以merge到其他分支),但是Git不允许你checkout到它们.

# git pull

* fetch from a remote repo and try to merge into the current branch.
* pull == fetch + merge FETCH\_HEAD
* git pull会首先执行git fetch,然后执行git merge,把取来的分支的head merge到当前分支.这个merge操作会产生一个新的commit.
* 如果使用--rebase参数,它会执行git rebase来取代原来的git merge.

# git rebase

* --rebase不会产生合并的提交,它会将本地的所有提交临时保存为补丁(patch),放在”.git/rebase”目录中,然后将当前分支更新到最新的分支,最后把保存的补丁应用到分支上。本地的所有提交记录会被丢弃。
* rebase的过程中,也许会出现冲突,Git会停止rebase并让你解决冲突,在解决完冲突之后,用git add去更新这些内容,然后无需执行commit,只需要:
* git rebase --continue就会继续打余下的补丁.
* git rebase --abort将会终止rebase,当前分支将会回到rebase之前的状态.

# git push

* push your new branches and data to a remote repository.
* git push [alias] [branch]
* 将会把当前分支merge到alias上的[branch]分支.如果分支已经存在,将会更新,如果不存在,将会添加这个分支.
* 如果有多个人向同一个remote repo push代码, Git会首先在你试图push的分支上运行git log,检查它的历史中是否能看到server上的branch现在的tip,如果本地历史中不能看到server的tip,说明本地的代码不是最新的,Git会拒绝你的push,让你先fetch,merge,之后再push,这样就保证了所有人的改动都会被考虑进来.

# git reflog

* git reflog是对reflog进行管理的命令,reflog是git用来记录引用变化的一种机制,比如记录分支的变化或者是HEAD引用的变化.
* 当git reflog不指定引用的时候,默认列出HEAD的reflog.
* HEAD@{0}代表HEAD当前的值,HEAD@{3}代表HEAD在3次变化之前的值.
* git会将变化记录到HEAD对应的reflog文件中,其路径为.git/logs/HEAD, 分支的reflog文件都放在.git/logs/refs目录下的子目录中.

# AJAX

# 1、AJAX概述

# 5.54 1.1 什么是AJAX

* AJAX（Asynchronous Javascript And XML)翻译成中文就是“异步Javascript和XML”。即使用Javascript语言与服务器进行异步交互，传输的数据为XML（当然，传输的数据不只是XML)。
* AJAX还有一个最大的特点就是，当服务器响应时，不用刷新整个浏览器页面，而是可以局部刷新。这一特点给用户的感受是在不知不觉中完成请求和响应过程。  
  - 与服务器异步交互；  
  - 浏览器页面局部刷新；

# 5.55 1.2. 同步交互与异步交互

- 同步交互：客户端发出一个请求后，需要等待服务器响应结束后，才能发出第二个请求；  
- 异步交互：客户端发出一个请求后，无需等待服务器响应结束，就可以发出第二个请求。

# 5.56 1.3. AJAX常见应用情景

* 当我们在百度中输入一个“传”字后，会马上出现一个下拉列表！列表中显示的是包含“传”字的10个关键字。
* 其实这里就使用了AJAX技术！当文件框发生了输入变化时，浏览器会使用AJAX技术向服务器发送一个请求，查询包含“传”字的前10个关键字，然后服务器会把查询到的结果响应给浏览器，最后浏览器把这10个关键字显示在下拉列表中。  
  - 整个过程中页面没有刷新，只是刷新页面中的局部位置而已！  
  - 当请求发出后，浏览器还可以进行其他操作，无需等待服务器的响应！
* 当输入用户名后，把光标移动到其他表单项上时，浏览器会使用AJAX技术向服务器发出请求，服务器会查询名为zhangSan的用户是否存在，最终服务器返回true表示名为zhangSan的用户已经存在了，浏览器在得到结果后显示“用户名已被注册！”。  
  - 整个过程中页面没有刷新，只是局部刷新了；  
  - 在请求发出后，浏览器不用等待服务器响应结果就可以进行其他操作；

# 5.57 1.4 AJAX的优缺点

* 优点：  
  - AJAX使用Javascript技术向服务器发送异步请求；  
  - AJAX无须刷新整个页面；  
  - 因为服务器响应内容不再是整个页面，而是页面中的局部，所以AJAX性能高；
* 缺点：  
  - AJAX并不适合所有场景，很多时候还是要使用同步交互；  
  - AJAX虽然提高了用户体验，但无形中向服务器发送的请求次数增多了，导致服务器压力增大；  
  - 因为AJAX是在浏览器中使用Javascript技术完成的，所以还需要处理浏览器兼容性问题；

# 2、AJAX技术

# 5.58 2.1 AJAX第一例（发送GET请求)

## 2.1.1 准备工作

* 因为AJAX也需要请求服务器，异步请求也是请求服务器，所以我们需要先写好服务器端代码，即编写一个Servlet！
* 这里，Servlet很简单，只需要输出“Hello AJAX!”。  
  public class AServlet extends HttpServlet {  
  public void doGet(HttpServletRequest request, HttpServletResponse response)  
   throws ServletException, IOException {  
   System.out.println("Hello AJAX!");  
   response.getWriter().print("Hello AJAX!");  
  }  
  }

## 2.1.2 AJAX核心（XMLHttpRequest)

* 其实AJAX就是在Javascript中多添加了一个对象：XMLHttpRequest对象。所有的异步交互都是使用XMLHttpRequest对象完成的。也就是说，我们只需要学习一个Javascript的新对象即可。
* 注意，各个浏览器对XMLHttpRequest的支持也是不同的！大多数浏览器都支持DOM2规范，都可以使用：var xmlHttp = new XMLHttpRequest()来创建对象
* 为了处理浏览器兼容问题，给出下面方法来创建XMLHttpRequest对象：  
  function createXMLHttpRequest() {  
   var xmlHttp;  
   // 适用于大多数浏览器，以及IE7和IE更高版本  
   try{  
   xmlHttp = new XMLHttpRequest();  
   } catch (e) {  
   // 适用于IE6  
   try {  
   xmlHttp = new ActiveXObject("Msxml2.XMLHTTP");  
   } catch (e) {  
   // 适用于IE5.5，以及IE更早版本  
   try{  
   xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");  
   } catch (e){}  
   }  
   }   
   return xmlHttp;  
  }

## 2.1.3 打开与服务器的连接（open方法)

* 当得到XMLHttpRequest对象后，就可以调用该对象的open()方法打开与服务器的连接了。open()方法的参数如下：
* open(method, url, async)：  
  - method：请求方式，通常为GET或POST；  
  - url：请求的服务器地址，例如：/ajaxdemo1/AServlet，若为GET请求，还可以在URL后追加参数；  
  - async：这个参数可以不给，默认值为true，表示异步请求；
* var xmlHttp = createXMLHttpRequest();  
  xmlHttp.open("GET", "/ajaxdemo1/AServlet", true);

## 2.1.4 发送请求

* 当使用open打开连接后，就可以调用XMLHttpRequest对象的send()方法发送请求了。send()方法的参数为POST请求参数，即对应HTTP协议的请求体内容，若是GET请求，需要在URL后连接参数。
* 注意：若没有参数，需要给出null为参数！若不给出null为参数，可能会导致FireFox浏览器不能正常发送请求！  
  xmlHttp.send(null);

## 2.1.5 接收服务器响应

* 当请求发送出去后，服务器端Servlet就开始执行了，但服务器端的响应还没有接收到。接下来我们来接收服务器的响应。
* XMLHttpRequest对象有一个onreadystatechange事件，它会在XMLHttpRequest对象的状态发生变化时被调用。下面介绍一下XMLHttpRequest对象的5种状态：  
  - 0：初始化未完成状态，只是创建了XMLHttpRequest对象，还未调用open()方法；  
  - 1：请求已开始，open()方法已调用，但还没调用send()方法；  
  - 2：请求发送完成状态，send()方法已调用；  
  - 3：开始读取服务器响应；  
  - 4：读取服务器响应结束。
* onreadystatechange事件会在状态为1、2、3、4时引发。
* 下面代码会被执行四次！对应XMLHttpRequest的四种状态！  
   xmlHttp.onreadystatechange = function() {  
   alert('hello');  
   };
* 但通常我们只关心最后一种状态，即读取服务器响应结束时，客户端才会做出改变。我们可以通过XMLHttpRequest对象的readyState属性来得到XMLHttpRequest对象的状态。  
  xmlHttp.onreadystatechange = function() {  
   if(xmlHttp.readyState == 4) {  
   alert('hello');   
   }  
   };
* 其实我们还要关心服务器响应的状态码是否为200，其服务器响应为404，或500，那么就表示请求失败了。我们可以通过XMLHttpRequest对象的status属性得到服务器的状态码。
* 最后，我们还需要获取到服务器响应的内容，可以通过XMLHttpRequest对象的responseText得到服务器响应内容。
* responsXML是xml格式的文本，是document对象  
  xmlHttp.onreadystatechange = function() {  
   if(xmlHttp.readyState == 4 && xmlHttp.status == 200) {  
   alert(xmlHttp.responseText);   
   }  
   };

## 2.1.6 AJAX第一例小结

- 创建XMLHttpRequest对象；  
- 调用open()方法打开与服务器的连接；  
- 调用send()方法发送请求；  
- 为XMLHttpRequest对象指定onreadystatechange事件函数，这个函数会在XMLHttpRequest的1、2、3、4，四种状态时被调用；

* XMLHttpRequest对象的5种状态：  
  - 0：初始化未完成状态，只是创建了XMLHttpRequest对象，还未调用open()方法；  
  - 1：请求已开始，open()方法已调用，但还没调用send()方法；  
  - 2：请求发送完成状态，send()方法已调用；  
  - 3：开始读取服务器响应；  
  - 4：读取服务器响应结束。
* 通常我们只关心4状态。
* XMLHttpRequest对象的status属性表示服务器状态码，它只有在readyState为4时才能获取到。
* XMLHttpRequest对象的responseText属性表示服务器响应内容，它只有在readyState为4时才能获取到！
* var xmlHttp = new XMLHttpRequest();
* xmlHttp.open("GET", "/AJAX/AServlet");
* xmlHttp.send(null);
* xmlHttp.onreadystatechange = function(){
* if(xmlHttp.readyState == 4 && xmlHttp.status == 200){
* var h1 = document.getElementById("h1");
* h1.innerHTML = xmlHttp.responseText;
* }
* }

public class AServlet extends HttpServlet {

public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

* response.setContentType("text/html;charset=utf-8");
* response.getWriter().print("hehe");
* }
* }

# 5.59 2.2 AJAX第二例（发送POST请求)

## 2.2.1 发送POST请求注意事项

* POST请求必须设置ContentType请求头的值为application/x-www.form-encoded。表单的enctype默认值就是为application/x-www.form-encoded！因为默认值就是这个，所以大家可能会忽略这个值！当设置了的enctype=” application/x-www.form-encoded”时，等同与设置了Cotnent-Type请求头。
* 但在使用AJAX发送请求时，就没有默认值了，这需要我们自己来设置请求头：  
  xmlHttp.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
* 当没有设置Content-Type请求头为application/x-www-form-urlencoded时，Web容器会忽略请求体的内容。所以，在使用AJAX发送POST请求时，需要设置这一请求头，然后使用send()方法来设置请求体内容。  
  xmlHttp.send("b=B");
* 这时Servlet就可以获取到这个参数！！！
* AServlet  
  public void doPost(HttpServletRequest request, HttpServletResponse response)  
   throws ServletException, IOException {  
   System.out.println(request.getParameter("b"));  
   System.out.println("Hello AJAX!");  
   response.getWriter().print("Hello AJAX!");  
  }
* ajax2.jsp

function send() {  
 var xmlHttp = createXMLHttpRequest();  
 xmlHttp.onreadystatechange = function() {  
 if(xmlHttp.readyState == 4 && xmlHttp.status == 200) {  
 var div = document.getElementById("div1");  
 div.innerHTML = xmlHttp.responseText;  
 }  
 };  
 xmlHttp.open("POST", "/ajaxdemo1/AServlet?", true);  
 xmlHttp.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");  
 xmlHttp.send("b=B");  
}

测试

# 5.60 2.3 AJAX第三例（用户名是否已被注册)

## 2.3.1 功能介绍

* 在注册表单中，当用户填写了用户名后，把光标移开后，会自动向服务器发送异步请求。服务器返回true或false，返回true表示这个用户名已经被注册过，返回false表示没有注册过。
* 客户端得到服务器返回的结果后，确定是否在用户名文本框后显示“用户名已被注册”的错误信息！

## 2.3.2 案例分析

- regist.jsp页面中给出注册表单；  
- 在username表单字段中添加onblur事件，调用send()方法；  
- send()方法获取username表单字段的内容，向服务器发送异步请求，参数为username；  
- RegistServlet：获取username参数，判断是否为“itcast”，如果是响应true，否则响应false；

## 2.3.3 代码

* regist.jsp
* RegistServlet.java  
  public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  
   String username = request.getParameter("username");  
   System.out.println(username);  
   if(username.equals("admin")){  
   response.getWriter().print(false);  
   }else{  
   response.getWriter().print(true);  
   }  
  }

# 前端

* HTML的DOM对象说几个，Document的对象和方法  
  document.body 返回元素 1  
  document.cookie 返回或设置与当前文档相关的cookie 1  
  document.domain 返回当前文档的服务器域名 1  
  document.referrer 返回连接至当前文档的文档连接 1  
  document.title 返回当前文档的元素 1  
  document.URL 返回当前文档的完整URL 1