

儒 释 道 · 随缘

之 Web

目录

第一部分 XML.....	3
第二部分 HTTP.....	10
第三部分 TOMCAT.....	31
第三部分 SERVLET.....	34
第四部分 JSP	67
第五部分 SQL.....	78
第六部分 JDBC.....	88

第一部分 XML

一、XML 概述

- 英文全称为 Extensible Markup Language，翻译过来为可扩展标记语言。XML 技术是 W3C 组织发布的，目前遵循的是 W3C 组织于 2000 发布的 XML1.0 规范。
- 现实生活中存在着大量的数据，在这些数据之间往往存在一定的关系，我们希望能在计算机中保存和处理这些数据的同时能够保存和处理他们之间的关系。
- XML 就是为了解决这样的需求而产生数据存储格式。

1、XML 是什么

XML 是一种数据存储格式，这种数据存储格式在存储数据内容的同时，还能够利用标签之间的嵌套关系来保存数据之间的关系。

2、XML 应用场景

- 1) 利用 XML 跨平台的特性，用来在不同的操作系统不同的开发语言之间传输数据。如果说 java 是一门跨平台的语言，那 XML 就是跨平台的数据。
- 2) 利用 XML 可以保存具有关系的数据的特性，还常常被用来做为配置文件使用。

3、XML 文件

把 XML 格式的数据保存到文件中，这样的文件通常起后缀名为.XML，这样的文件就叫做 XML 文件，XML 文件是 XML 数据最常见的存在形式，但是，这不是 XML 的唯一存在形式（在内存中或在网络中也可以存在），不要把 XML 狭隘的理解成 XML 文件。

4、XML 校验

浏览器除了内置 html 解析引擎外还内置了 XML 解析器，利用浏览器打开 XML 格式的数据，就可以进行 XML 校验。

二、xml 语法

1、文档声明

一个格式良好的 XML 必须包含也只能包含一个文档声明，并且文档声明必须出现在 XML 文档第一行，其前不能有其他任何内容。

1. 最简单的写法：

<?XML version="1.0" ?>其中的 version 代表当前 XML 所遵循的规范版本；

2. 使用 encoding 属性指定文档所使用的字符集编码：

<?XML version="1.0" encoding="gb2312" ?>

注意：encoding 属性指定的编码集和 XML 真正使用的编码应该一致，如果不一致就会有乱码问题。encoding 属性默认值为老外喜欢的 iso8859-1

3. 使用 standalone 属性指定当前 XML 文档是否是一个独立文档：

<?XML version="1.0" standalone="no" ?>

standalone 属性用来指明当前 xml 是否是一个独立的 xml，默认值是 yes 表明当前文档不需要依赖于其他文档，如果当前文档依赖其他文档而存在则需要将此值设置为 no

注意：很多的解析器会忽略这个属性，但是学习知识要按标准去学，所以这个属性也要掌握。

2、元素

元素分为开始标签和结束标签，在开始标签和结束标签之间的文本称为标签体，如果一个标签即不含标签体也不包含其他标签，那这样的标签可以把开始标签和结束标签进行合并，这样的标签叫自闭标签。

<a>xxxxx <a/>

一个元素也可以包含若干子元素，但是要注意所有的标签都要进行合理嵌套。

一个格式良好的 XML 文档应该具有并且只能有一个跟标签，其他标签都应该在这个跟标签的子孙标签。

元素的命名规范：

1. 区分大小写，例如，`<P>`和`<p>`是两个不同的标记。
2. 不能以数字或标点符号或“_”开头。
3. 不能以 XML(或 XML、或 Xml 等)开头。
4. 不能包含空格。
5. 名称中间不能包含冒号(：)

3、属性

一个元素可以包含多个属性，属性的值要用单引号或双引号括起来。如果属性的之中包含双引号，就要用单引号了。

每个属性都有它自己的名称和取值，例如：

```
<china capital="beijing"/>
```

属性的名在定义时要遵循和 xml 元素相同的命名规则

属性的值需要用单引号或双引号括起来

4、注释

```
<!-- 注释内容 -->
```

注释可以出现在 xml 文档的任意位置除了整个文档的最前面.不能出现在文档声明之前

注释不能嵌套注释

实验：把注释写到文档声明之前，用 ie 打开是没问题，但是用 chrome 打开是报错的。这就看出来了解析器有不同的处理，我们学习的时候还是按标准去学。

5、CDATA 区/转义字符

1. `<![CDATA[转义的内容]]>`

当 XML 中一段内容不希望被解析器解析时可以使用 CDATA 区将其包住

当解析器遇到 CDATA 区时会将其内容当作文本对待，不会进行解析

语法：`<![CDATA[内容]]>`

2. 转义字符

`&` --> `&`;

`<` --> `<`;

`>` --> `>`;

`"` --> `"`;

`'` --> `'`;

3. CDATA 区和转义字符的区别

- 1) CDATA 区可以成段的进行转义，而转义字符一次只能转义一个字符
- 2) CDATA 区转义的字符可以保存数据本来的格式只是通知解析器按文本去处理。转义字符改变了数据本身的内容，利用其他字符替代了转义字符。请思考，如果要转义的内容是一段 js 程序的话，如果用转义字符合适不合适？

6、处理指令

处理指令，简称 PI (processing instruction)。处理指令用来指挥解析引擎如何解析 XML 文档内容。

```
<?xml:stylesheet type="text/css" href="1.css"?>
```

三、XML 约束

1、什么是约束

在 xml 技术里，可以编写一个文档来约束一个 xml 文档的写法，这称之为 XML 约束。约束 xml 文档的写法，对 xml 进行校验。

2、DTD 约束

DTD 是一门 XML 约束技术，用来约束 XML 写法。

如何在 XML 中引入一个 DTD:

1. 外部引入

本地文件引入:

如果写的是 SYSTEM 表明当前引入的 dtd 在当前文件系统中，后面制定的文件位置是当前硬盘中的位置

<!DOCTYPE 根元素的名称 SYSTEM "文件所在的路径">

公共位置文件引入:

如果写的是 PUBLIC 表明当前引入的 dtd 在网络公共位置中,后面要指明 dtd 的名字和 dtd 所在网络位置 URL 地址

<!DOCTYPE 根元素的名称 PUBLIC "dtd 名称" "dtd 所在的 URL">

2. 内部引入

直接在 xml 中书写 dtd

<!DOCTYPE 根元素名称[

dtd 约束....

]>

dtd 语法:

1.元素

<!ELEMENT 元素名称 元素约束>

(1)存放类型

ANY:当前声明的元素可以包含任意子元素

EMPTY:当前声明的元素不能包含任何元素

(2)存放内容: 利用小括号括起来的元素的名称, 用来表示该元素中可以存放哪些内容

<!ELEMENT "元素名" (可以包含的元素的名称)>

小括号中的内容, 可以有多个子元素的名称

如果用 “,” 分割这些子元素就表明这些子元素必须按指定的顺序出现

如果用 “|” 分割这些内容就表明这些子元素只能出现其中之一

使用 “+” 来表明内容可以出现一次或多次

使用 “*” 来表明内容可以出现零次或多次

使用 “?” 来表明内容可以出现零次或一次

#PCDATA 表明该元素可以包含标签体

可以利用()进行组操作:

<!ELEMENT MYFILE ((TITLE*, AUTHOR?, EMAIL)* | COMMENT)>

2.属性

<!ATTLIST 元素名

属性名 属性类型 属性约束

属性名 2 属性类型 属性约束

.....

>

属性类型

(1) CDATA: 表明该属性的值是一个普通的文本值。

(2) ENUMERATED: 表明该属性的值只能取指定范围内的其中之一

(3) ID:表明该属性值在整个文档中必须唯一, 注意 ID 类型的属性的值必须以字母下划线开头, 并且不能以数字开头, 不能包含空白字符

属性约束

- (1) #REQUIRED 来表明当前这个属性是必须存在的属性
- (2) #IMPLIED 来表明当前这个属性是可选的属性
- (3) #FIXED "固定值" 来表明当前这个属性具有一个默认的值, 可以不明确指定该属性, 解析器会帮你加上, 如果你硬是指定了一个其他的值, 会出错。
- (4) "默认值" 来表明当前属性具有一个默认的值, 如果给这个属性指定一个值就用指定的值, 如果不指定呢, 就使用默认值。

3. 实体

可以理解为对一段内容的引用, 如果有一段内容到处在被使用, 可以将其设计为一个实体

引用实体: 用在 XML 中的实体

声明实体: <!ENTITY 实体名称 "实体内容">

引用引用实体: &实体名称;

参数实体: 用在 DTD 文件中的实体, 参数实体被 DTD 文件自身使用

声明实体:<!ENTITY % 实体名称 "实体内容">

引用参数实体: %实体名称;

举例 1:

```
<!ENTITY % TAG_NAMES "姓名 | EMAIL | 电话 | 地址">
```

```
<!ELEMENT 个人信息 (%TAG_NAMES; | 生日)>
```

```
<!ELEMENT 客户信息 (%TAG_NAMES; | 公司名)>
```

举例 2:

```
<!ENTITY % common.attributes
```

```
" id ID #IMPLIED
```

```
account CDATA #REQUIRED "
```

```
>
```

```
...
```

```
<!ATTLIST purchaseOrder %common.attributes;>
```

```
<!ATTLIST item %common.attributes;>
```

3、Schema 约束

需要掌握名称空间的概念, 会读简单的 Schema 就可以了, 不需要大家自己会写

Schema 是 xml 的约束技术, 出现的目的是为了替代 dtd

本身也是一个 xml, 非常方便使用 xml 的解析引擎进行解析

对名称空间有非常好的支持

支持更多的数据类型, 并且支持用户自定义数据类型

可以进行语义级别的限定, 限定能力大大强于 dtd

相对于 dtd 不支持实体

相对于 dtd 复杂的多, 学习成本比较高

4、Schema 与 DTD 的比较

XML Schema 符合 XML 语法结构。

DOM、SAX 等 XML API 很容易解析出 XML Schema 文档中的内容。

XML Schema 对名称空间支持得非常好。

XML Schema 比 XML DTD 支持更多的数据类型, 并支持用户自定义新的数据类型。

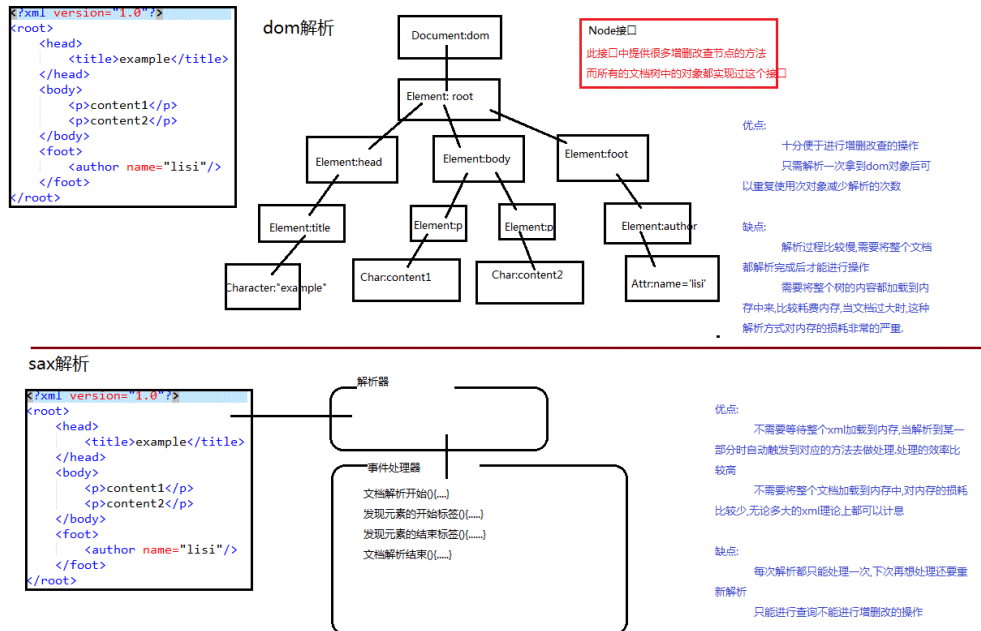
XML Schema 定义约束的能力非常强大，可以对 XML 实例文档作出细致的语义限制。

XML Schema 不能像 DTD 一样定义实体，比 DTD 更复杂，但 Xml Schema 现在已是 w3c 组织的标准，它正逐步取代 DTD。

四、XML 解析

1、XML 编程

利用 java 程序去增删改查(CRUD)xml 中的数据



2、解析思想

基于这两种解析思想市面上就有了很多的解析 api

sun jaxp 既有 dom 方式也有 sax 方式,并且这套解析 api 已经加入到 j2se 的规范中,意味这不需要导入任何第三方开发包就可以直接使用这种解析方式,但是这种解析方式效率低下,没什么人用。

dom4j 可以使用 dom 方式高效的解析 xml.

pull android 里的 api

dom4j 导入开发包,通常只需要导入核心包就可以了,如果在使用的过程中提示少什么包到 lib 目录下导入缺少的包即可

3、Demo4j 解析

```
public class Demo4jDemo2 {

    public void attr() throws Exception{

        SAXReader reader = new SAXReader();

        Document dom = reader.read("book.xml");

        Element root = dom.getRootElement();

        Element bookEle = root.element("书");
        bookEle.addAttribute("出版社", "传智出版社");
        String str = bookEle.attributeValue("出版社");
        System.out.println(str);

        Attribute attr = bookEle.attribute("出版社");

        attr.getParent().remove(attr);

        XMLWriter writer = new XMLWriter(new FileOutputStream("book.xml"), OutputFormat.createPrettyPrint());

        writer.write(dom);

        writer.close();

    }

    @Test

    public void del() throws Exception{

    }
```

```
SAXReader reader = new SAXReader();

Document dom = reader.read("book.xml");

Element root = dom.getRootElement();

Element price2Ele = root.element("书").element("特价");

price2Ele.getParent().remove(price2Ele);

XMLWriter writer = new XMLWriter(new FileOutputStream ("book.xml"),OutputFormat.createPrettyPrint());

writer.write(dom);

writer.close();

}

@Test

public void update()throws Exception{

    SAXReader reader = new SAXReader();

    Document dom = reader.read("book.xml");

    Element root = dom.getRootElement();

    root.element("书").element("特价").setText("4.0 元");

    XMLWriter writer = new XMLWriter(new FileOutputStream ("book.xml"),OutputFormat.createPrettyPrint());

    writer.write(dom);

    writer.close();

}

@Test

public void add()throws Exception{

    SAXReader reader = new SAXReader();

    Document dom = reader.read("book.xml");

    Element root = dom.getRootElement();//凭空创建<特价>节点, 设置标签体

    Element price2Ele = DocumentHelper.createElement(" 特价");

    price2Ele.setText("40.0 元");//获取父标签<书>将特价节点挂载上去

    Element bookEle = root.element(" 书");

    bookEle.add(price2Ele);//将内存中的 dom 树会写到 xml 文件中,从而使 xml 中的数据进行更新//FileWriter writer = new FileWriter("book.xml");

    //dom.write(writer);//writer.flush();// writer.close();

    XMLWriter writer = new XMLWriter(new FileOutputStream ("book.xml"),OutputFormat.createPrettyPrint());

    writer.write(dom);

    writer.close();

}

@Test

public void find() throws Exception{

    SAXReader reader = new SAXReader();

    Document dom = reader.read("book.xml");

    Element root = dom.getRootElement();

    List<Element> list = root.elements();

    Element book2Ele = list.get(1);

    System.out.println(book2Ele.element("书名").getText());

}

}
```


4、Sax 解析

```
public class SaxDemo1 {

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

        SAXParserFactory factory = SAXParserFactory.newInstance();//1. 获取解析器工厂

        SAXParser parser = factory.newSAXParser();//2. 通过工厂获取 sax 解析器

        XMLReader reader = parser.getXMLReader();//3. 获取读取器

        reader.setContentHandler(new MyContentHandler() ); //4. 注册事件处理器

        reader.parse("book.xml");//5. 解析 xml

    }

}

class MyContentHandler2 extends DefaultHandler// 适配器设计模式

    private String eleName = null;

    private int count = 0;

    @Override

    public void startElement(String uri, String localName, String name,Attributes attributes) throws SAXException {

        this.eleName = name;

    }

    @Override

    public void characters(char[] ch, int start, int length)throws SAXException {

        if("书名".equals(eleName) && ++count==2){

            System.out.println(new String(ch,start,length));

        }

    }

    @Override

    public void endElement(String uri, String localName, String name)throws SAXException {

        eleName = null;

    }

}

class MyContentHandler implements ContentHandler{

    public void startDocument() throws SAXException {

        System.out.println("文档解析开始了.....");

    }

    public void startElement(String uri, String localName, String name,Attributes atts) throws SAXException {

        System.out.println("发现了开始标签,"+name);

    }

    public void characters(char[] ch, int start, int length)throws SAXException {

        System.out.println(new String(ch,start,length));

    }

    public void endElement(String uri, String localName, String name)throws SAXException {

        System.out.println("发现结束标签,"+name);

    }

    public void endDocument() throws SAXException {

        System.out.println("文档解析结束了.....");

    }

}
```

```
public void endPrefixMapping(String prefix) throws SAXException { // TODO Auto-generated method stub
}

public void ignorableWhitespace(char[] ch, int start, int length) throws SAXException { // TODO Auto-generated method stub
}

public void processingInstruction(String target, String data) throws SAXException { // TODO Auto-generated method stub
}

public void setDocumentLocator(Locator locator) { // TODO Auto-generated method stub
}

public void skippedEntity(String name) throws SAXException { // TODO Auto-generated method stub
}

public void startPrefixMapping(String prefix, String uri) throws SAXException { // TODO Auto-generated method stub
}
}
```

第二部分 HTTP

一、Http 概念

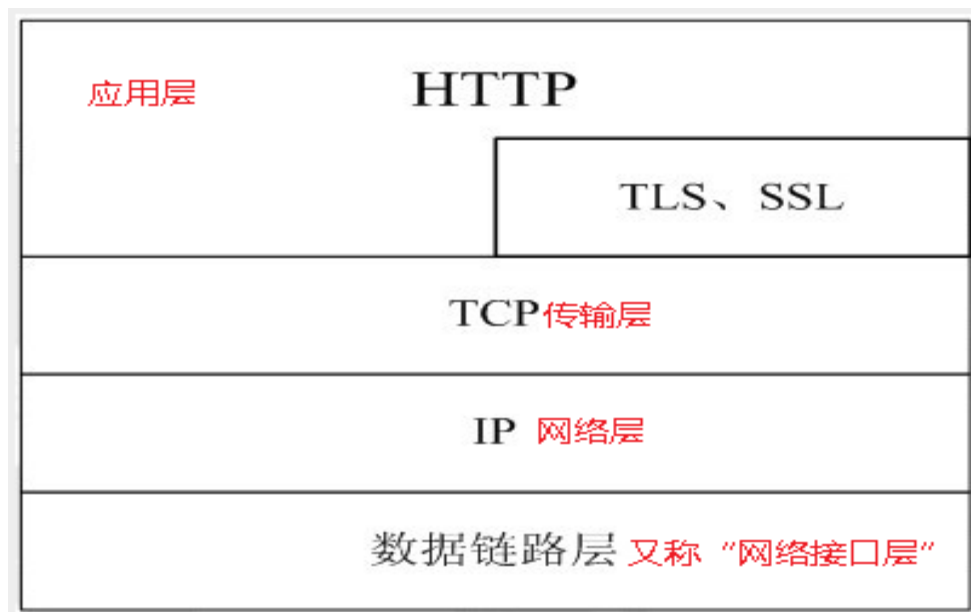
协议是指计算机通信网络中两台计算机之间进行通信所必须共同遵守的规定或规则，超文本传输协议(HTTP)是一种通信协议，它允许将超文本标记语言(HTML)文档从 Web 服务器传送到客户端的浏览器。

HTTP 协议，即超文本传输协议(Hypertext transfer protocol)。是一种详细规定了浏览器和万维网(WWW = World Wide Web)服务器之间互相通信的规则，通过因特网传送万维网文档的数据传送协议。

HTTP 协议是用于从 WWW 服务器传输超文本到本地浏览器的传送协议。它可以使浏览器更加高效，使网络传输减少。它不仅保证计算机正确快速地传输超文本文档，还确定传输文档中的哪一部分，以及哪部分内容首先显示(如文本先于图形)等。

HTTP 是一个应用层协议，由请求和响应构成，是一个标准的客户端服务器模型。HTTP 是一个无状态的协议。

在 Internet 中所有的传输都是通过 TCP/IP 进行的。HTTP 协议作为 TCP/IP 模型中应用层的协议也不例外。HTTP 协议通常承载于 TCP 协议之上，有时也承载于 TLS 或 SSL 协议层之上，这个时候，就成了我们常说的 HTTPS。如下图所示：



HTTP 默认的端口号为 80，HTTPS 的端口号为 443。

浏览网页是 HTTP 的主要应用，但是这并不代表 HTTP 就只能应用于网页的浏览。HTTP 是一种协议，只要通信的双方都遵守这个协议，HTTP 就能有用武之地。比如咱们常用的 QQ，迅雷这些软件，都会使用 HTTP 协议(还包括其他的协议)。

二、HTTP 简史

它的发展是万维网协会（World Wide Web Consortium）和 Internet 工作小组 IETF（Internet Engineering Task Force）合作的结果，（他们）最终发布了一系列的 RFC，RFC 1945 定义了 HTTP/1.0 版本。其中最著名的就是 RFC 2616。RFC 2616 定义了今天普遍使用的一个版本——HTTP 1.1。

三、特点

HTTP 协议永远都是客户端发起请求，服务器回送响应。这样就限制了使用 HTTP 协议，无法实现在客户端没有发起请求的时候，服务器将消息推送给客户端。

HTTP 协议的主要特点可概括如下：

- 1、支持客户/服务器模式。支持基本认证和安全认证。
- 2、简单快速：客户向服务器请求服务时，只需传送请求方法和路径。请求方法常用的有 GET、HEAD、POST。每种方法规定了客户与服务器联系的类型不同。由于 HTTP 协议简单，使得 HTTP 服务器的程序规模小，因而通信速度很快。
- 3、灵活：HTTP 允许传输任意类型的数据对象。正在传输的类型由 Content-Type 加以标记。
- 4、HTTP 0.9 和 1.0 使用非持续连接：限制每次连接只处理一个请求，服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间。HTTP 1.1 使用持续连接：不必为每个 web 对象创建一个新的连接，一个连接可以传送多个对象。
- 5、无状态：HTTP 协议是无状态协议。无状态是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。

无状态协议：

协议的状态是指下一次传输可以“记住”这次传输信息的能力。

http 是不会为了下一次连接而维护这次连接所传输的信息,为了保证服务器内存。

比如客户获得一张网页之后关闭浏览器，然后再一次启动浏览器，再登陆该网站，但是服务器并不知道客户关闭了一次浏览器。由于 Web 服务器要面对很多浏览器的并发访问，为了提高 Web 服务器对并发访问的处理能力，在设计 HTTP 协议时规定 Web 服务器发送 HTTP 应答报文和文档时，不保存发出请求的 Web 浏览器进程的任何状态信息。这有可能出现一个浏览器在短短几秒之内两次访问同一对象时，服务器进程不会因为已经给它发过应答报文而不接受第二期服务请求。由于 Web 服务器不保存发送请求的 Web 浏览器进程的任何信息，因此 HTTP 协议属于无状态协议（Stateless Protocol）。

HTTP 协议是无状态的和 Connection: keep-alive 的区别：

无状态是指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。从另一方面讲，打开一个服务器上的网页和你之前打开这个服务器上的网页之间没有任何联系。

HTTP 是一个无状态的面向连接的协议，无状态不代表 HTTP 不能保持 TCP 连接，更不能代表 HTTP 使用的是 UDP 协议（无连接）。从 HTTP/1.1 起，默认都开启了 Keep-Alive，保持连接特性，简单地说，当一个网页打开完成后，客户端和服务器之间用于传输 HTTP 数据的 TCP 连接不会关闭，如果客户端再次访问这个服务器上的网页，会继续使用这一条已经建立的连接。

Keep-Alive 不会永久保持连接，它有一个保持时间，可以在不同的服务器软件（如 Apache）中设定这个时间。

四、工作流程

一次 HTTP 操作称为一个事务，其工作过程可分为四步：

- 1) 首先客户机与服务器需要建立连接。只要单击某个超级链接，HTTP 的工作开始。
- 2) 建立连接后，客户机发送一个请求给服务器，请求方式的格式为：

统一资源标识符（URL）、协议版本号，后边是 MIME 信息包括请求修饰符、客户机信息和可能的内容。

- 3) 服务器接到请求后，给予相应的响应信息，其格式为一个状态行，包括信息的协议版本号、一个成功或错误的代码，后边是 MIME 信息包括服务器信息、实体信息和可能的内容。

- 4) 客户端接收服务器所返回的信息通过浏览器显示在用户的显示屏上，然后客户机与服务器断开连接。

如果在以上过程中的某一步出现错误，那么产生错误的信息将返回到客户端，由显示屏输出。对于用户来说，这些过程是由 HTTP 自己完成的，用户只要用鼠标点击，等待信息显示就可以了。

HTTP 是基于传输层的 TCP 协议，而 TCP 是一个端到端的面向连接的协议。所谓的端到端可以理解为进程到进程之间的通信。所以 HTTP 在开始传输之前，首先需要建立 TCP 连接，而 TCP 连接的过程需要所谓的“三次握手”。

在 TCP 三次握手之后，建立了 TCP 连接，此时 HTTP 就可以进行传输了。一个重要的概念是面向连接，既 HTTP 在传输完成之间，并不断开 TCP 连接。在 HTTP1.1 中(通过 Connection 头设置)这是默认行为。

五、HTTP 头域

每个头域由一个域名，冒号（:）和域值三部分组成。域名是大小写无关的，域值前可以添加任何数量的空格符，头域可以被扩展为多行，在每行开始处，使用至少一个空格或制表符。

1、请求信息

发出的请求信息格式如下：

- 请求行，例如 GET /images/logo.gif HTTP/1.1，表示从/images 目录下请求 logo.gif 这个文件。
- （请求）消息报头，例如 Accept-Language: en
- 空行
- 可选的消息体（请求正文）

请求行和标题必须以<CR><LF>作为结尾（也就是，回车然后换行）。空行内必须只有<CR><LF>而无其他空格。在 HTTP/1.1 协议中，所有的请求头，除 post 外，都是可选的。

METHOD /path - to - resource HTTP/Version-number
Header-Name-1: value
Header-Name-2: value
Optional request body

三个部分分别是：请求行、消息报头、请求正文。

2、请求方法

HTTP/1.1 协议中共定义了八种方法（有时也叫“动作”）来表明 Request-URI 指定的资源的不同操作方式：

OPTIONS - 返回服务器针对特定资源所支持的 HTTP 请求方法。也可以利用向 Web 服务器发送'*'的请求来测试服务器的功能性。

HEAD- 向服务器索要一个与 GET 请求相一致的响应，只不过响应体将不会被返回。这一方法可以在不必传输整个响应内容的情况下，就可以获取包含在响应消息头中的元信息。该方法常用于测试超链接的有效性，是否可以访问，以及最近是否更新。

GET - 向特定的资源发出请求。注意：GET 方法不应当被用于产生“副作用”的操作中，例如在 web app 中。其中一个原因是 GET 可能会被网络蜘蛛等随意访问。

POST - 向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST 请求可能会导致新的资源的建立和/或已有资源的修改。

PUT - 向指定资源位置上传其最新内容。

DELETE - 请求服务器删除 Request-URI 所标识的资源。

TRACE- 回显服务器收到的请求，主要用于测试或诊断。

CONNECT - HTTP/1.1 协议中预留给能够将连接改为管道方式的代理服务器。

PATCH - 用来将局部修改应用于某一资源，添加于规范 RFC5789。

方法名称是区分大小写的。当某个请求所针对的资源不支持对应的请求方法的时候，服务器应当返回状态码 405 (Method Not Allowed)；当服务器不认识或者不支持对应的请求方法的时候，应当返回状态码 501 (Not Implemented)。

HTTP 服务器至少应该实现 GET 和 HEAD 方法，其他方法都是可选的。此外，除了上述方法，特定的 HTTP 服务器还能够扩展自定义的方法。

GET 和 POST 的区别：

- 1、GET 提交的数据会放在 URL 之后，以?分割 URL 和传输数据，参数之间以&相连，如

EditPosts.aspx?name=test1&id=123456.

POST 方法是把提交的数据放在 HTTP 包的 Body 中。

- 2、GET 提交的数据大小有限制，最多只能有 1024 字节（因为浏览器对 URL 的长度有限制），而 POST 方法提交的数据没有限制。
- 3、GET 方式需要使用 Request.QueryString 来取得变量的值，而 POST 方式通过 Request.Form 来获取变量的值。
- 4、GET 方式提交数据，会带来安全问题，比如一个登录页面，通过 GET 方式提交数据时，用户名和密码将出现在 URL 上，如果页面可以被缓存或者其他人可以访问这台机器，就可以从历史记录获得该用户的账号和密码。

3、响应消息

客户端向服务器发送一个请求，服务器以一个状态行作为响应，响应的内容包括：消息协议的版本、成功或者错误编码、服务器信息、实体元信息以及必要的实体内容。根据响应类别的类别，服务器响应里可以含实体内容，但不是所有的响应都有实体内容。

返回的响应信息格式如下：

- 状态行：HTTP-Version 空格 Status-Code 空格 Reason-Phrase CRLF
- 消息报头
- 空行
- 响应正文

Http/version-number	status code	message
Header-Name-1: value		
Header-Name-2: value		
Optional Response body		

状态行，格式如下（下图中红线标出的那行）。

HTTP-Version 表示 HTTP 版本，例如为 HTTP/1.1。

Status-Code 是结果代码，用三个数字表示。

Status-Code 用于机器自动识别，Status-Code 的第一个数字代表响应类别，可能取 5 个不同的值。后两个数字没有分类作用。Status-Code 的第一个数字代表响应的类别，后续两位描述在该类响应下发生的具体状况。

Reason-Phrase 是个简单的文本描述，解释 Status-Code 的具体原因。

Reason-Phrase 用于人工理解。

▼ Response Headers view parsed

HTTP/1.1 200 OK

Date: Fri, 11 Jul 2014 03:49:33 GMT

Server: Apache/2.2.22 (Unix) mod_ssl/2.2.22 OpenSSL/1.0.1e-fips PHP/5.3.17

X-Powered-By: PHP/5.3.17

Expires: Thu, 19 Nov 1981 08:52:00 GMT

Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0

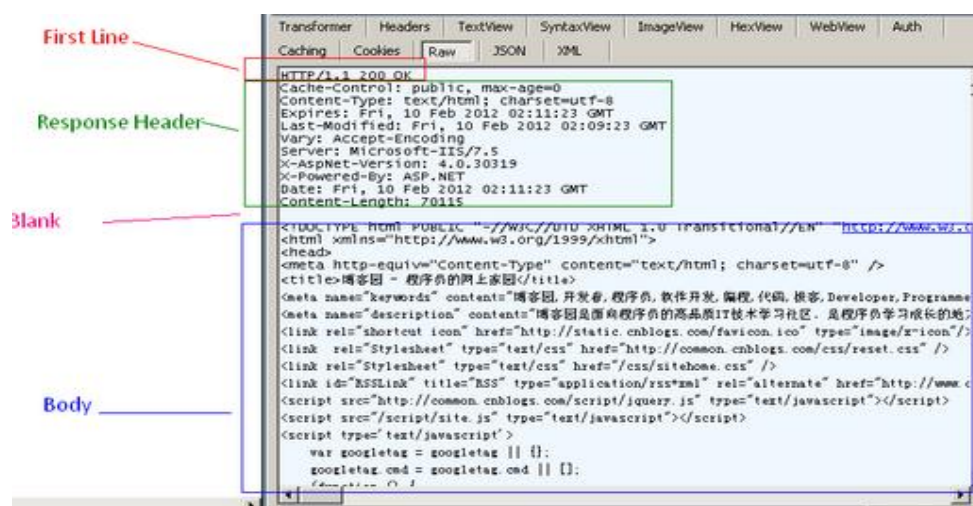
Pragma: no-cache

Keep-Alive: timeout=5, max=776

Connection: Keep-Alive

Transfer-Encoding: chunked

Content-Type: text/html



三个部分分别是：状态行、消息报头、响应正文。

无论你何时浏览一个网页，你的电脑都会通过一个使用 HTTP 协议的服务器来获取所请求的数据。在你请求的网页显示在浏览器之前，支配网页的网站服务器会返回一个包含有状态码的 HTTP 头文件。这个状态码提供了有关所请求网页的相关条件信息。如果一切正常，一个标准网页会收到一条诸如 200 的状态码。当然我们的目的不是去研究 200 响应码，而是去探讨那些代表出现错误信息的服务器头文件响应码，例如表示“未找到指定网页”的 404 码。

4、响应头域

服务器需要传递许多附加信息，这些信息不能全放在状态行里。因此，需要另行定义响应头域，用来描述这些附加信息。响应头域主要描述服务器的信息和 Request-URI 的信息。

```
HTTP/1.1 200 OK\r\n
[Expert Info (Chat/Sequence): HTTP/1.1 200 OK\r\n]
Request Version: HTTP/1.1
Status Code: 200
Response Phrase: OK
Server: JSP2/1.0.27\r\n
Date: Sat, 12 Jul 2014 06:58:25 GMT\r\n
Content-Type: image/gif\r\n
Connection: keep-alive\r\n
Content-Length: 228\r\n
ETag: "1691437"\r\n
Accept-Ranges: bytes\r\n
Last-Modified: Thu, 10 Jul 2014 08:19:47 GMT\r\n
Expires: Fri, 18 Jul 2014 05:44:46 GMT\r\n
Cache-Control: max-age=604800\r\n
\r\n
```

5、HTTP 常见的请求头

(在 HTTP/1.1 协议中，所有的请求头，除 Host 外，都是可选的)

If-Modified-Since: 把浏览器端缓存页面的最后修改时间发送到服务器去，服务器会把这个时间与服务器上实际文件的最后修改时间进行对比。如果时间一致，那么返回 304，客户端就直接使用本地缓存文件。如果时间不一致，就会返回 200 和新的文件内容。客户端接到之后，会丢弃旧文件，把新文件缓存起来，并显示在浏览器中。

例如：If-Modified-Since: Thu, 09 Feb 2012 09:07:57 GMT

The screenshot shows the Fiddler interface with a list of web sessions on the left and detailed headers on the right. The session list includes:

#	Result	Protocol	Host	URL
1	200	HTTP	www.cnblogs.com	/
2	304	HTTP	www.cnblogs.com	/css/sitehome.css
3	304	HTTP	common.cnblogs.com	/css/reset.css
4	304	HTTP	www.cnblogs.com	/script/site.js
5	304	HTTP	common.cnblogs.com	/script/jquery.js
6	304	HTTP	common.cnblogs.com	/script/gpt.js
7	304	HTTP	www.cnblogs.com	/images/logo_small.gif
8	200	HTTP	www.cnblogs.com	/ws/PublicUserService
9	304	HTTP	static.cnblogs.com	/images/icon_rss.gif
10	200	HTTP	www.cnblogs.com	/ws/SiteHomeService
11	200	HTTP	pic.cnblogs.com	/face/u313549.jpg?ic
12	304	HTTP	www.cnblogs.com	/ws/PublicUserService
13	304	HTTP	pic.cnblogs.com	/face/u313549.jpg?ic
14	304	HTTP	pic.cnblogs.com	/face/u313549.jpg?ic
15	304	HTTP	pic.cnblogs.com	/face/u313549.jpg?ic
16	304	HTTP	pic.cnblogs.com	/face/u313549.jpg?ic
17	304	HTTP	pic.cnblogs.com	/face/u313549.jpg?ic
18	304	HTTP	pic.cnblogs.com	/face/u313549.jpg?ic
19	200	HTTP	www.cnblogs.com	/ws/SiteHomeService
20	200	HTTP	pic.cnblogs.com	/face/u323522.jpg?ic
21	304	HTTP	pic.cnblogs.com	/face/u106968.gif
22	304	HTTP	pic.cnblogs.com	/face/u342353.jpg?ic
23	200	HTTP	pic.cnblogs.com	/face/u220660.jpg?ic
24	200	HTTP	pic.cnblogs.com	/face/u159329.jpg
25	200	HTTP	www.cnblogs.com	/ws/PublicUserService
26	200	HTTP	pic.cnblogs.com	/face/u510.jpg

Red handwritten text on the screenshot reads: "本地文件的修改时间和服务器上的文件修改时间一样。说明文件没有更新过。HTTP服务器将返回304，告诉客户端使用本地缓存文件" (The modification time of the local file is the same as the modification time of the file on the server. This indicates that the file has not been updated. The HTTP server will return 304, telling the client to use the local cached file).

The right pane shows the Request Headers for the selected session:

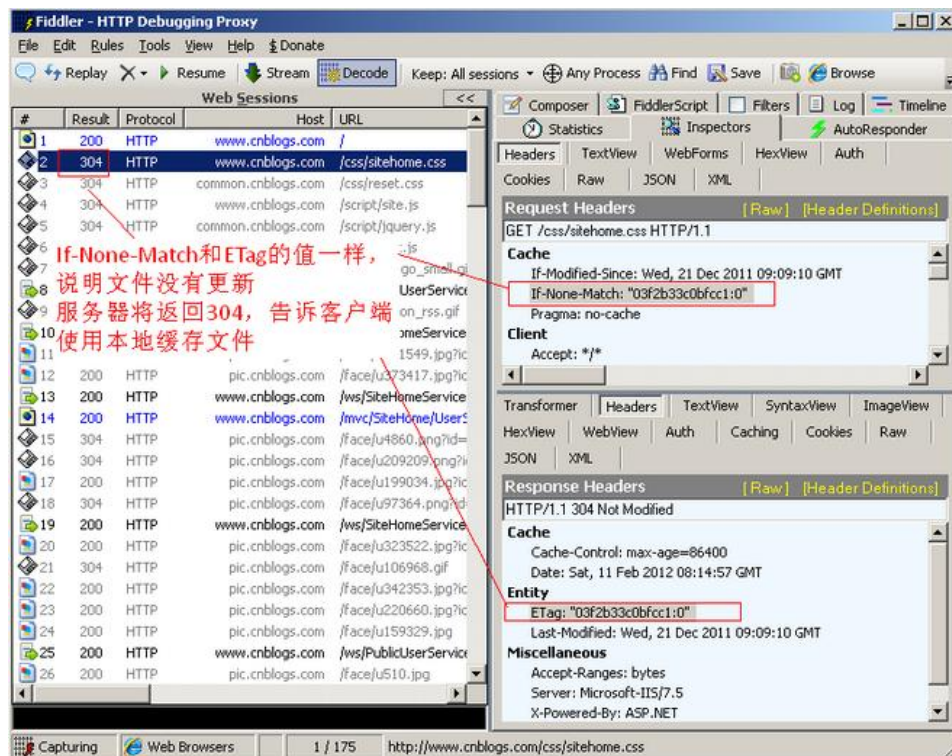
```
Request Headers
GET /css/sitehome.css HTTP/1.1
Cache
If-Modified-Since: Wed, 21 Dec 2011 09:09:10 GMT
If-None-Match: "03f2b33c0bfcc1:0"
Pragma: no-cache
Client
Accept: */*
```

The Response Headers for the selected session are:

```
Response Headers
HTTP/1.1 304 Not Modified
Cache
Cache-Control: max-age=86400
Date: Sat, 11 Feb 2012 08:14:57 GMT
Entity
ETag: "03f2b33c0bfcc1:0"
Last-Modified: Wed, 21 Dec 2011 09:09:10 GMT
Miscellaneous
Accept-Ranges: bytes
Server: Microsoft-IIS/7.5
X-Powered-By: ASP.NET
```

If-None-Match: If-None-Match 和 ETag 一起工作，工作原理是在 HTTP Response 中添加 ETag 信息。当用户再次请求该资源时，将在 HTTP Request 中加入 If-None-Match 信息(ETag 的值)。如果服务器验证资源的 ETag 没有改变（该资源没有更新），将返回一个 304 状态告诉客户端使用本地缓存文件。否则将返回 200 状态和新的资源和 Etag。使用这样的机制将提高网站的性能。

例如: If-None-Match: "03f2b33c0bfcc1:0"。



Pragma: 指定“no-cache”值表示服务器必须返回一个刷新后的文档，即使它是代理服务器而且已经有了页面的本地拷贝；在 HTTP/1.1 版本中，它和 Cache-Control:no-cache 作用一模一样。Pragma 只有一个用法，例如： Pragma: no-cache

注意：在 HTTP/1.0 版本中，只实现了 Pragma:no-cache, 没有实现 Cache-Control

Cache-Control: 指定请求和响应遵循的缓存机制。缓存指令是单向的（响应中出现的缓存指令在请求中未必会出现），且是独立的（在请求消息或响应消息中设置 Cache-Control 并不会修改另一个消息处理过程中的缓存处理过程）。请求时的缓存指令包括 no-cache、no-store、max-age、max-stale、min-fresh、only-if-cached，响应消息中的指令包括 public、private、no-cache、no-store、no-transform、must-revalidate、proxy-revalidate、max-age、s-maxage。

Cache-Control:Public 可以被任何缓存所缓存

Cache-Control:Private 内容只缓存到私有缓存中

Cache-Control:no-cache 所有内容都不会被缓存

Cache-Control:no-store 用于防止重要的信息被无意的发布。在请求消息中发送将使得请求和响应消息都不使用缓存。

Cache-Control:max-age 指示客户机可以接收生存期不大于指定时间（以秒为单位）的响应。

Cache-Control:min-fresh 指示客户机可以接收响应时间小于当前时间加上指定时间的响应。

Cache-Control:max-stale 指示客户机可以接收超出超时期间的响应消息。如果指定 max-stale 消息的值，那么客户机可以接收超出超时期指定值之内的响应消息。

Accept: 浏览器端可以接受的 MIME 类型。例如：Accept: text/html 代表浏览器可以接受服务器回发的类型为 text/html 也就是我们常说的 html 文档，如果服务器无法返回 text/html 类型的数据，服务器应该返回一个 406 错误(non acceptable)。通配符 * 代表任意类型，例如 Accept: /* 代表浏览器可以处理所有类型，（一般浏览器发给服务器都是发这个）。

Accept-Encoding: 浏览器申明自己可接收的编码方法，通常指定压缩方法，是否支持压缩，支持什么压缩方法（gzip, deflate）;Servlet 能够向支持 gzip 的浏览器返回经 gzip 编码的 HTML 页面。许多情形下这可以减少 5 到 10 倍的下载时间。例如：Accept-Encoding: gzip, deflate。如果请求消息中没有设置这个域，服务器假定客户端对各种内容编码都可以接受。

Accept-Language: 浏览器申明自己接收的语言。语言跟字符集的区别：中文是语言，中文有多种字符集，比如 big5, gb2312, gbk 等等；例如：Accept-Language: en-us。如果请求消息中没有设置这个报头域，服务器假定客户端对各种语言都可以接受。

Accept-Charset: 浏览器可接受的字符集。如果在请求消息中没有设置这个域，缺省表示任何字符集都可以接受。

User-Agent: 告诉 HTTP 服务器，客户端使用的操作系统和浏览器的名称和版本。

例如：User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; CIBA; .NET CLR 2.0.50727; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729; .NET4.0C; InfoPath.2; .NET4.0E)。

Content-Type: 例如：Content-Type: application/x-www-form-urlencoded。

Referer: 包含一个 URL，用户从该 URL 代表的页面出发访问当前请求的页面。提供了 Request 的上下文信息的服务器，告诉服务器我是从哪个链接过来的，比如从我主页上链接到一个朋友那里，他的服务器就能够从 HTTP Referer 中统计出每天有多少用户点击我主页上的链接访问他的网站。

例如：Referer: http://translate.google.cn/?hl=zh-cn&tab=wT

Connection:

例如：Connection: keep-alive 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。HTTP 1.1 默认进行持久连接。利用持久连接的优点，当页面包含多个元素时（例如 Applet，图片），显著地减少下载所需要的时间。要实现这一点，Servlet 需要在应答中发送一个 Content-Length 头，最简单的实现方法是：先把内容写入 ByteArrayOutputStream，然后在正式写出内容之前计算它的大小。

Connection: close 代表一个 Request 完成后，客户端和服务器之间用于传输 HTTP 数据的 TCP 连接会关闭，当客户端再次发送 Request，需要重新建立 TCP 连接。

Host:（发送请求时，该头域是必需的）主要用于指定被请求资源的 Internet 主机和端口号，它通常从 HTTP URL 中提取出来的。HTTP/1.1 请求必须包含主机头域，否则系统会以 400 状态码返回。

例如：我们在浏览器中输入：http://www.guet.edu.cn/index.html，浏览器发送的请求消息中，就会包含 Host 请求头域：Host: http://www.guet.edu.cn，此处使用缺省端口号 80，若指定了端口号，则变成：Host: 指定端口号。

Cookie: 最重要的请求头之一，将 cookie 的值发送给 HTTP 服务器。

Content-Length: 表示请求消息正文的长度。例如：Content-Length: 38。

Authorization: 授权信息，通常出现在对服务器发送的 WWW-Authenticate 头的应答中。主要用于证明客户端有权查看某个资源。当浏览器访问一个页面时，如果收到服务器的响应代码为 401（未授权），可以发送一个包含 Authorization 请求报头域的请求，要求服务器对其进行验证。

UA-Pixels, UA-Color, UA-OS, UA-CPU: 由某些版本的 IE 浏览器所发送的非标准的请求头，表示屏幕大小、颜色深度、操作系统和 CPU 类型。

From: 请求发送者的 email 地址，由一些特殊的 Web 客户程序使用，浏览器不会用到它。

Range: 可以请求实体的一个或者多个子范围。例如，

表示头 500 个字节：bytes=0-499

表示第二个 500 字节：bytes=500-999

表示最后 500 个字节：bytes=-500

表示 500 字节以后的范围：bytes=500-

第一个和最后一个字节：bytes=0-0,-1

同时指定几个范围：bytes=500-600,601-999

但是服务器可以忽略此请求头，如果无条件 GET 包含 Range 请求头，响应会以状态码 206（PartialContent）返回而不是以 200（OK）。

6、HTTP 常见的响应头

Allow: 服务器支持哪些请求方法（如 GET、POST 等）。

Date: 表示消息发送的时间，时间的描述格式由 rfc822 定义。例如，Date: Mon, 31 Dec 2001 04:25:57 GMT。Date 描述的时间表示世界标准时，换算成本地时间，需要知道用户所在的时区。你可以用 setDateHeader 来设置这个头以避免转换时间格式的麻烦

Expires: 指明应该在什么时候认为文档已经过期，从而不再缓存它，重新从服务器获取，会更新缓存。过期之前使用本地缓存。HTTP 1.1 的客户端和缓存会将非法的日期格式（包括 0）看作已经过期。eg：为了让浏览器不要缓存页面，我们也可以将 Expires 实体报头域，设置为 0。

例如：Expires: Tue, 08 Feb 2022 11:35:14 GMT

P3P: 用于跨域设置 Cookie, 这样可以解决 iframe 跨域访问 cookie 的问题

例如: P3P: CP=CURa ADMa DeVa PSo PSDo OUR BUS UNI PUR INT DEM STA PRE COM NAV OTC NOI DSP COR

Set-Cookie: 非常重要的 header, 用于把 cookie 发送到客户端浏览器, 每一个写入 cookie 都会生成一个 Set-Cookie。

例如: Set-Cookie: sc=4c31523a; path=/; domain=.acookie.taobao.com

ETag: 和 If-None-Match 配合使用。

Last-Modified: 用于指示资源的最后修改日期和时间。Last-Modified 也可用 setDateHeader 方法来设置。

Content-Type: WEB 服务器告诉浏览器自己响应的对象的类型和字符集。Servlet 默认为 text/plain, 但通常需要显式地指定为 text/html。由于经常要设置 Content-Type, 因此 HttpServletResponse 提供了一个专用的方法 setContentType。可在 web.xml 文件中配置扩展名和 MIME 类型的对应关系。

例如: Content-Type: text/html; charset=utf-8

Content-Type: text/html; charset=GB2312

Content-Type: image/jpeg

媒体类型的格式为: 大类/小类, 比如 text/html。

IANA(The Internet Assigned Numbers Authority, 互联网数字分配机构)定义了 8 个大类的媒体类型, 分别是:

application— (比如: application/vnd.ms-excel.)

audio (比如: audio/mpeg.)

image (比如: image/png.)

message (比如: .message/http.)

model(比如: model/vrml.)

multipart (比如: multipart/form-data.)

text(比如: text/html.)

video(比如: video/quicktime.)

Content-Range: 用于指定整个实体中的一部分的插入位置, 他也指示了整个实体的长度。在服务器向客户返回一个部分响应, 它必须描述响应覆盖的范围和整个实体长度。一般格式: Content-Range: bytes-unitSPfirst-byte-pos-last-byte-pos/entity-length。例如, 传送头 500 个字节次字段的形式: Content-Range: bytes0-499/1234 如果一个 http 消息包含此节(例如, 对范围请求的响应或对一系列范围的重叠请求), Content-Range 表示传送的范围。

Content-Length: 指明实体正文的长度, 以字节方式存储的十进制数字来表示。在数据下行的过程中, Content-Length 的方式要预先在服务器中缓存所有数据, 然后所有数据再一股脑儿地发给客户端。只有当浏览器使用持久 HTTP 连接时才需要这个数据。如果你想要利用持久连接的优势, 可以把输出文档写入 ByteArrayOutputStream, 完成后查看其大小, 然后把该值放入 Content-Length 头, 最后通过 byteArrayStream.writeTo(response.getOutputStream())发送内容。

例如: Content-Length: 19847

Content-Encoding: WEB 服务器表明自己使用了什么压缩方法(gzip, deflate)压缩响应中的对象。只有在解码之后才可以得到 Content-Type 头指定的内容类型。利用 gzip 压缩文档能够显著地减少 HTML 文档的下载时间。Java 的 GZIPOutputStream 可以很方便地进行 gzip 压缩, 但只有 Unix 上的 Netscape 和 Windows 上的 IE 4、IE 5 才支持它。因此, Servlet 应该通过查看 Accept-Encoding 头(即 request.getHeader("Accept-Encoding"))检查浏览器是否支持 gzip, 为支持 gzip 的浏览器返回经 gzip 压缩的 HTML 页面, 为其他浏览器返回普通页面。

例如: Content-Encoding: gzip

Content-Language: WEB 服务器告诉浏览器自己响应的对象所用的自然语言。例如: Content-Language: da。没有设置该域则认为实体内容将提供给所有的语言阅读。

Server: 指明 HTTP 服务器用来处理请求的软件信息。例如: Server: Microsoft-IIS/7.5、Server: Apache-Coyote/1.1。此域能包含多个产品标识和注释, 产品标识一般按照重要性排序。

X-AspNet-Version: 如果网站是用 ASP.NET 开发的, 这个 header 用来表示 ASP.NET 的版本。

例如: X-AspNet-Version: 4.0.30319

X-Powered-By: 表示网站是用什么技术开发的。

例如: X-Powered-By: ASP.NET

Connection:

例如: Connection: keep-alive 当一个网页打开完成后,客户端和服务端之间用于传输 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。

Connection: close 代表一个 Request 完成后,客户端和服务端之间用于传输 HTTP 数据的 TCP 连接会关闭,当客户端再次发送 Request,需要重新建立 TCP 连接。

Location: 用于重定向到一个新的位置,包含新的 URL 地址。表示客户端应当到哪里去提取文档。Location 通常不是直接设置的,而是通过 HttpServletResponse 的 sendRedirect 方法,该方法同时设置状态代码为 302。Location 响应报头域常用在更换域名的时候。

Refresh: 表示浏览器应该在多少时间之后刷新文档,以秒计。除了刷新当前文档之外,你还可以通过 setHeader("Refresh", "5; URL=http://host/path") 让浏览器读取指定的页面。注意这种功能通常是通过设置 HTML 页面 HEAD 区的 <META HTTP-EQUIV="Refresh" CONTENT="5;URL=http://host/path"> 实现,这是因为,自动刷新或重定向对于那些不能使用 CGI 或 Servlet 的 HTML 编写者十分重要。但是,对于 Servlet 来说,直接设置 Refresh 头更加方便。注意 Refresh 的意义是“N 秒之后刷新本页面或访问指定页面”,而不是“每隔 N 秒刷新本页面或访问指定页面”。因此,连续刷新要求每次都发送一个 Refresh 头,而发送 204 状态代码则可以阻止浏览器继续刷新,不管是使用 Refresh 头还是 <META HTTP-EQUIV="Refresh" ...>。注意 Refresh 头不属于 HTTP 1.1 正式规范的一部分,而是一个扩展,但 Netscape 和 IE 都支持它。

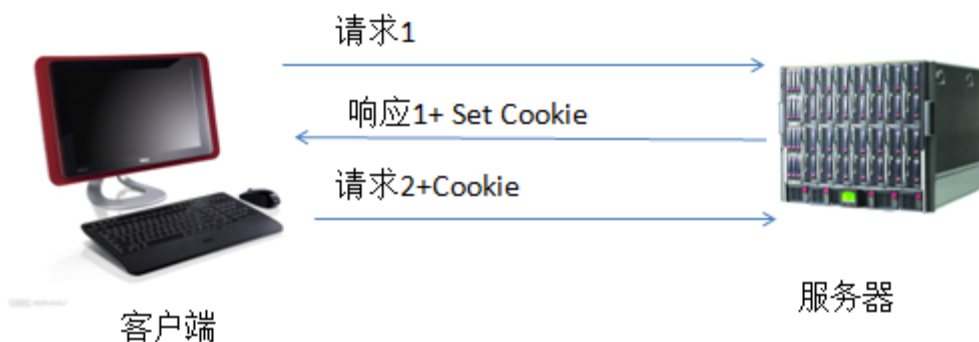
WWW-Authenticate: 该响应报头域必须被包含在 401 (未授权的) 响应消息中,客户端收到 401 响应消息时候,并发送 Authorization 报头域请求服务器对其进行验证时,服务端响应报头就包含该报头域。

eg: WWW-Authenticate: Basic realm="Basic Auth Test!" //可以看出服务器对请求资源采用的是基本验证机制。

六、解决 HTTP 无状态的问题

1、通过 Cookies 保存状态信息

通过 Cookies,服务器就可以清楚的知道请求 2 和请求 1 来自同一个客户端。



2、通过 Session 保存状态信息

Session 机制是一种服务器端的机制,服务器使用一种类似于散列表的结构(也可能就是使用散列表)来保存信息。

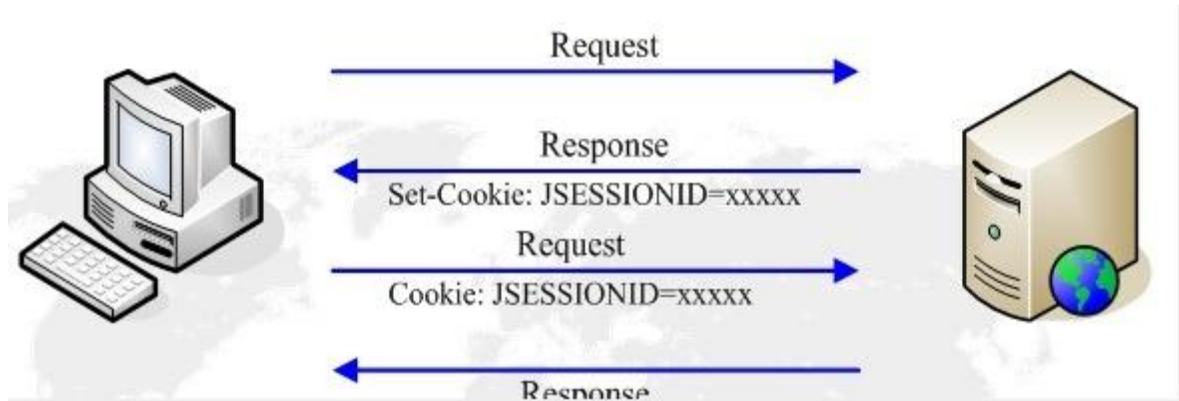
当程序需要为某个客户端的请求创建一个 session 的时候,服务器首先检查这个客户端的请求里是否已包含了一个 session 标识 - 称为 session id,如果已包含一个 session id 则说明以前已经为此客户端创建过 session,服务器就按照 session id 把这个 session 检索出来使用(如果检索不到,可能会新建一个),如果客户端请求不包含 session id,则为此客户端创建一个 session 并且生成一个与此 session 相关联的 session id,session id 的值应该是一个既不会重复,又不容易被找到规律以伪造的字符串,这个 session id 将被在本次响应中返回给客户端保存。

Session 的实现方式:

1、使用 Cookie 来实现

服务器给每个 Session 分配一个唯一的 JSESSIONID，并通过 Cookie 发送给客户端。

当客户端发起新的请求的时候，将在 Cookie 头中携带这个 JSESSIONID。这样服务器能够找到这个客户端对应的 Session。



2、使用 URL 回写来实现

URL 回写是指服务器在发送给浏览器页面的所有链接中都携带 JSESSIONID 的参数，这样客户端点击任何一个链接都会把 JSESSIONID 带会服务器。如果直接在浏览器输入服务端资源的 url 来请求该资源，那么 Session 是匹配不到的。

Tomcat 对 Session 的实现，是一开始同时使用 Cookie 和 URL 回写机制，如果发现客户端支持 Cookie，就继续使用 Cookie，停止使用 URL 回写。如果发现 Cookie 被禁用，就一直使用 URL 回写。jsp 开发处理到 Session 的时候，对页面中的链接记得使用 response.encodeURL()。

Cookie 和 Session 有以下明显的不同点：

- 1) Cookie 将状态保存在客户端，Session 将状态保存在服务器端；
- 2) Cookies 是服务器在本地机器上存储的小段文本并随每一个请求发送至同一个服务器。Cookie 最早在 RFC2109 中实现，后续 RFC2965 做了增强。网络服务器用 HTTP 头向客户端发送 cookies，在客户端，浏览器解析这些 cookies 并将它们保存为一个本地文件，它会自动将同一服务器的任何请求绑上这些 cookies。Session 并没有在 HTTP 的协议中定义；
- 3) Session 是针对每一个用户的，变量的值保存在服务器上，用一个 sessionId 来区分是哪个用户 session 变量，这个值是通过用户的浏览器在访问的时候返回给服务器，当客户禁用 cookie 时，这个值也可能设置为由 get 来返回给服务器；
- 4) 就安全性来说：当你访问一个使用 session 的站点，同时在自己机子上建立一个 cookie，建议在服务器端的 SESSION 机制更安全些。因为它不会任意读取客户存储的信息。

3、通过表单变量保持状态

除了 Cookies 之外，还可以使用表单变量来保持状态，比如 Asp.net 就通过一个叫 ViewState 的 Input="hidden" 的框来保持状态，比如：

```
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="/wEPDwUKMjA0OTM4MTAwNGRkXUfhIDv1Cs7/qhBlyZROCzlvf5U=" />
```

这个原理和 Cookies 大同小异，只是每次请求和响应所附带的信息变成了表单变量。

4、通过 QueryString 保持状态

QueryString 通过将信息保存在所请求地址的末尾来向服务器传送信息，通常和表单结合使用，一个典型的 QueryString 比如：
www.xxx.com/xxx.aspx?var1=value&var2=value2

七、使用 telnet 进行 http 测试

在 Windows 下，可使用命令窗口进行 http 简单测试。输入 cmd 进入命令窗口，在命令行键入如下命令后按回车：

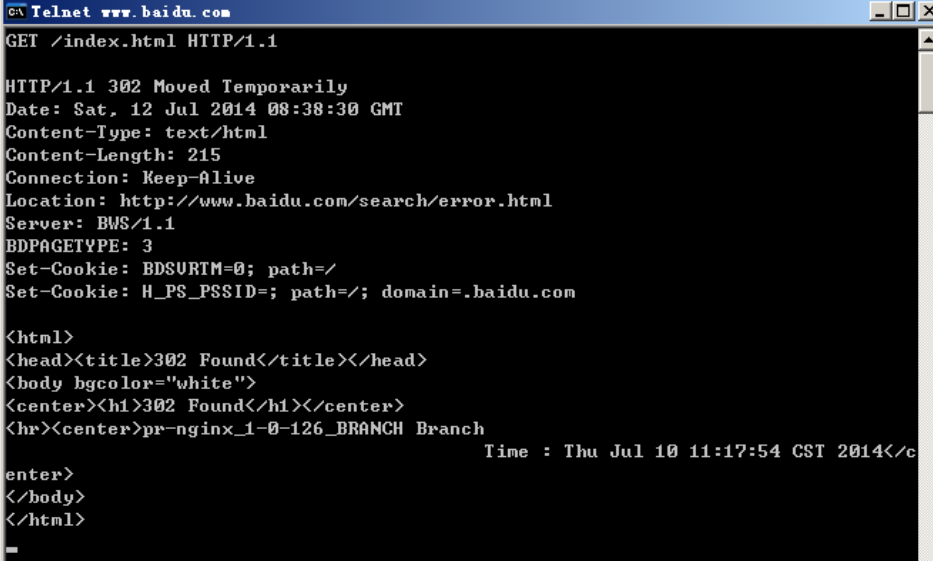
```
telnet www.baidu.com 80
```

而后在窗口中按下"Ctrl+J"后按回车可让返回结果回显。

接着开始发请求消息，例如发送如下请求消息请求 baidu 的首页消息，使用的 HTTP 协议为 HTTP/1.1:

```
GET /index.html HTTP/1.1
```

注意: copy 如上的消息到命令窗口后需要按两个回车换行才能得到响应的消息，第一个回车换行是在命令后键入回车换行，是 HTTP 协议要求的。第二个是确认输入，发送请求。



```
GV Telnet www.baidu.com
GET /index.html HTTP/1.1

HTTP/1.1 302 Moved Temporarily
Date: Sat, 12 Jul 2014 08:38:30 GMT
Content-Type: text/html
Content-Length: 215
Connection: Keep-Alive
Location: http://www.baidu.com/search/error.html
Server: BWS/1.1
BDPAGETYPE: 3
Set-Cookie: BDSURTM=0; path=/
Set-Cookie: H_PS_PSSID=; path=/; domain=.baidu.com

<html>
<head><title>302 Found</title></head>
<body bgcolor="white">
<center><h1>302 Found</h1></center>
<hr><center>pre-nginx_1-0-126_BRANCH Branch
Time : Thu Jul 10 11:17:54 CST 2014</c
enter>
</body>
</html>
```

可看到，当采用 HTTP/1.1 时，连接不是在请求结束后就断开的。若采用 HTTP1.0，在命令窗口键入:

```
GET /index.html HTTP/1.0
```

此时可以看到请求结束之后马上断开。

读者还可以尝试在使用 GET 或 POST 等时，带上头域信息，例如键入如下信息:

```
GET /index.html HTTP/1.1
```

```
connection: close
```

```
Host: www.baidu.com
```

八、URL 详解

URL(Uniform Resource Locator) 地址用于描述一个网络上的资源，基本格式如下

schema://host[:port#]/path/.../[;url-params][?query-string][#anchor]

scheme 指定低层使用的协议(例如: http, https, ftp)

host HTTP 服务器的 IP 地址或者域名

port# HTTP 服务器的默认端口是 80，这种情况下端口号可以省略。如果使用了别的端口，必须指明，例如

http://www.cnblogs.com:8080/

path 访问资源的路径

url-params

query-string 发送给 http 服务器的数据

anchor- 锚

URL 的一个例子:

http://www.mywebsite.com/sj/test?id=8079?name=sviergn&x=true#stuff

Schema: http

host: www.mywebsite.com

path: /sj/test

URL params: id=8079

Query String: name=sviergn&x=true

Anchor: stuff

九、缓存的实现原理

WEB 缓存(cache)位于 Web 服务器和客户端之间。

缓存会根据请求保存输出内容的副本，例如 html 页面，图片，文件，当下一个请求来到的时候：如果是相同的 URL，缓存直接使用副本响应访问请求，而不是向源服务器再次发送请求。

HTTP 协议定义了相关的消息头来使 WEB 缓存尽可能好的工作。

1、缓存的优点

减少相应延迟：因为请求从缓存服务器（离客户端更近）而不是源服务器被相应，这个过程耗时更少，让 web 服务器看上去相应更快。

减少网络带宽消耗：当副本被重用时会减低客户端的带宽消耗；客户可以节省带宽费用，控制带宽的需求的增长并更易于管理。

2、客户端缓存生效的常见流程

服务器收到请求时，会在 200OK 中回送该资源的 Last-Modified 和 ETag 头，客户端将该资源保存在 cache 中，并记录这两个属性。当客户端需要发送相同的请求时，会在请求中携带 If-Modified-Since 和 If-None-Match 两个头。两个头的值分别是响应中 Last-Modified 和 ETag 头的值。服务器通过这两个头判断本地资源未发生变化，客户端不需要重新下载，返回 304 响应。

3、Web 缓存机制

HTTP/1.1 中缓存的目的是为了在很多情况下减少发送请求，同时许多情况下可以不需要发送完整响应。前者减少了网络回路的数量；HTTP 利用一个“过期（expiration）”机制来为此目的。后者减少了网络应用的带宽；HTTP 用“验证（validation）”机制来为此目的。

HTTP 定义了 3 种缓存机制：

1) Freshness：允许一个回应消息可以在源服务器不被重新检查，并且可以由服务器和客户端来控制。例如，Expires 回应头给了一个文档不可用的时间。Cache-Control 中的 max-age 标识指明了缓存的最长时间；

2) Validation：用来检查以一个缓存的回应是否仍然可用。例如，如果一个回应有一个 Last-Modified 回应头，缓存能够使用 If-Modified-Since 来判断是否已改变，以便判断根据情况发送请求；

3) Invalidation：在另一个请求通过缓存的时候，常常有一个副作用。例如，如果一个 URL 关联到一个缓存回应，但是其后跟着 POST、PUT 和 DELETE 的请求的话，缓存就会过期。

十、HTTP 应用

1、断点续传的实现原理

HTTP 协议的 GET 方法，支持只请求某个资源的某一部分；

206 Partial Content 部分内容响应；

Range 请求的资源范围；

Content-Range 响应的资源范围；

在连接断开重连时，客户端只请求该资源未下载的部分，而不是重新请求整个资源，来实现断点续传。

分块请求资源实例：

Eg1: Range: bytes=306302- : 请求这个资源从 306302 个字节到末尾的部分；

Eg2: Content-Range: bytes 306302-604047/604048: 响应中指示携带的是该资源的第 306302-604047 的字节，该资源共 604048 个字节；

客户端通过并发的请求相同资源的不同片段，来实现对某个资源的并发分块下载。从而达到快速下载的目的。目前流行的 FlashGet 和迅雷基本都是这个原理。

2、多线程下载的原理

下载工具开启多个发出 HTTP 请求的线程：

每个 http 请求只请求资源文件的一部分：Content-Range: bytes 20000-40000/47000；

合并每个线程下载的文件。

3、http 代理

http 代理服务器

代理服务器英文全称是 Proxy Server，其功能就是代理网络用户去取得网络信息。形象的说：它是网络信息的中转站。

代理服务器是介于浏览器和 Web 服务器之间的一台服务器，有了它之后，浏览器不是直接到 Web 服务器去取回网页而是向代理服务器发出请求，Request 信号会先送到代理服务器，由代理服务器来取回浏览器所需要的信息并传送给你的浏览器。

而且，大部分代理服务器都具有缓冲的功能，就好像一个大的 Cache，它有很大的存储空间，它不断将新取得数据储存在它本机的存储器上，如果浏览器所请求的数据在它本机的存储器上已经存在而且是最新的，那么它就不重新从 Web 服务器取数据，而直接将存储器上的数据传送给用户的浏览器，这样就能显著提高浏览速度和效率。更重要的是：Proxy Server(代理服务器)是 Internet 链路级网关所提供的一种重要的安全功能，它的工作主要在开放系统互联(OSI)模型的对话层。

http 代理服务器的主要功能：

1) 突破自身 IP 访问限制，访问国外站点。如：教育网、169 网等网络用户可以通过代理访问国外网站；

2) 访问一些单位或团体内部资源，如某大学 FTP(前提是该代理地址在该资源的允许访问范围之内)，使用教育网内地址段免费代理服务器，就可以用于对教育网开放的各类 FTP 下载上传，以及各类资料查询共享等服务；

3) 突破中国电信的 IP 封锁：中国电信用户有很多网站是被限制访问的，这种限制是人为的，不同 Serve 对地址的封锁是不同的。所以不能访问时可以换一个国外的代理服务器试试；

4) 提高访问速度：通常代理服务器都设置一个较大的硬盘缓冲区，当有外界的信息通过时，同时也将其保存到缓冲区中，当其他用户再访问相同的信息时，则直接由缓冲区中取出信息，传给用户，以提高访问速度；

5) 隐藏真实 IP：上网者也可以通过这种方法隐藏自己的 IP，免受攻击。

对于客户端浏览器而言，http 代理服务器相当于服务器。

而对于 Web 服务器而言，http 代理服务器又担当了客户端的角色。

4、虚拟主机

虚拟主机：是在网络服务器上划分出一定的磁盘空间供用户放置站点、应用组件等，提供必要的站点功能与数据存放、传输功能。

所谓虚拟主机，也叫“网站空间”就是把一台运行在互联网上的服务器划分成多个“虚拟”的服务器，每一个虚拟主机都具有独立的域名和完整的 Internet 服务器（支持 WWW、FTP、E-mail 等）功能。一台服务器上的不同虚拟主机是各自独立的，并由用户自行管理。但一台服务器主机只能支持一定数量的虚拟主机，当超过这个数量时，用户将会感到性能急剧下降。

虚拟主机的实现原理

虚拟主机是用同一个 WEB 服务器，为不同域名网站提供服务的技术。Apache、Tomcat 等均可通过配置实现这个功能。

相关的 HTTP 消息头：Host。

例如：Host: www.baidu.com

客户端发送 HTTP 请求的时候，会携带 Host 头，Host 头记录的是客户端输入的域名。这样服务器可以根据 Host 头确认客户要访问的是哪一个域名。

十一、HTTP 认证方式

HTTP 请求报头：Authorization

HTTP 响应报头：WWW-Authenticate

HTTP 认证是基于质询/回应(challenge/response)的认证模式。

1 基本认证

basic authentication (HTTP1.0 提出的认证方法)

基本认证是一种用来允许 Web 浏览器或其他客户端程序在请求时提供用户名和口令形式的身份凭证的一种登录验证方式。

把 "用户名+冒号+密码"用 BASE64 算法加密后的字符串放在 http request 中的 header Authorization 中发送给服务端。

客户端对于每一个 realm, 通过提供用户名和密码来进行认证的方式。

包含密码的明文传递。

基本认证步骤:

1、客户端访问一个受 http 基本认证保护的资源。

2、服务器返回 401 状态, 要求客户端提供用户名和密码进行认证。(验证失败的时候, 响应头会加上 WWW-Authenticate: Basic realm="请求域"。)

401 Unauthorized

WWW-Authenticate: Basic realm="WallyWorld"

3、客户端将输入的用户名密码用 Base64 进行编码后, 采用非加密的明文方式传送给服务器。

Authorization: Basic xxxxxxxxxx.

4、服务器将 Authorization 头中的用户名密码解码并取出, 进行验证, 如果认证成功, 则返回相应的资源。如果认证失败, 则仍返回 401 状态, 要求重新进行认证。

特记事项:

1、Http 是无状态的, 同一个客户端对同一个 realm 内资源的每一个访问会被要求进行认证。

2、客户端通常会缓存用户名和密码, 并和 authentication realm 一起保存, 所以, 一般不需要你重新输入用户名和密码。

3、以非加密的明文方式传输, 虽然转换成了不易被人直接识别的字符串, 但是无法防止用户名密码被恶意盗用。虽然用肉眼看不出来, 但用程序很容易解密。

优点:

基本认证的一个优点是基本上所有流行的网页浏览器都支持基本认证。基本认证很少在可公开访问的互联网网站上使用, 有时候会在小的私有系统中使用(如路由器

网页管理接口)。后来的机制 HTTP 摘要认证是为替代基本认证而开发的, 允许密钥以相对安全的方式在不安全的通道上传输。

程序员和系统管理员有时会在可信网络环境中使用基本认证, 使用 Telnet 或其他明文网络协议工具手动地测试 Web 服务器。这是一个麻烦的过程, 但是网络上传输的

内容是人可读的, 以便进行诊断。

缺点:

虽然基本认证非常容易实现, 但该方案建立在以下的假设的基础上, 即: 客户端和服务器主机之间的连接是安全可信的。特别是, 如果没有使用 SSL/TLS 这样的传输

层安全的协议, 那么以明文传输的密钥和口令很容易被拦截。该方案也同样没有对服务器返回的信息提供保护。

现存的浏览器保存认证信息直到标签页或浏览器被关闭, 或者用户清除历史记录。HTTP 没有为服务器提供一种方法指示客户端丢弃这些被缓存的密钥。这意味着服务

器端在用户不关闭浏览器的情况下, 并没有一种有效的方法来让用户登出。

一个例子:

这一个典型的 HTTP 客户端和 HTTP 服务器的对话, 服务器安装在同一台计算机上 (localhost), 包含以下步骤:

客户端请求一个需要身份认证的页面, 但是没有提供用户名和口令。这通常是用户在地址栏输入一个 URL, 或是打开了一个指向该页面的链接。服务端响应一个 401 应

答码, 并提供一个认证域。接到应答后, 客户端显示该认证域 (通常是所访问的计算机或系统的描述) 给用户并提示输入用户名和口令。此时用户可以选择确定或取

消。用户输入了用户名和口令后, 客户端软件会在原先的请求上增加认证消息头 (值是 base64encode(username+":"+password)), 然后重新发送再次尝试。

在本例中, 服务器接受了该认证屏幕并返回了页面。如果用户凭据非法或无效, 服务器可能再次返回 401 应答码, 客户端可以

再次提示用户输入口令。

注意:客户端有可能不需要用户交互, 在第一次请求中就发送认证消息头。

客户端请求 (没有认证信息):

GET /private/index.html HTTP/1.0

Host: localhost

(跟随一个换行, 以回车 (CR) 加换行 (LF) 的形式)

服务端应答:

HTTP/1.0 401 Authorization Required

Server: HTTPd/1.0

Date: Sat, 27 Nov 2004 10:18:15 GMT

WWW-Authenticate: Basic realm="Secure Area"

Content-Type: text/html

Content-Length: 311

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/1999/REC-html401-19991224/loose.dtd">
<HTML>
  <HEAD>
    <TITLE>Error</TITLE>
    <META HTTP-EQUIV="Content-Type" CONTENT="text/html;
charset=ISO-8859-1">
  </HEAD>
  <BODY><H1>401 Unauthorized.</H1></BODY>
</HTML>
```

客户端的请求 (用户名 "Aladdin", 口令, password "open sesame"):

GET /private/index.html HTTP/1.0

Host: localhost

Authorization: Basic QWxhZGRpbjpvYVUHNlc2FtZQ==

(跟随一个空行, 如上所述)

服务端的应答:

HTTP/1.0 200 OK

Server: HTTPd/1.0

Date: Sat, 27 Nov 2004 10:19:07 GMT

Content-Type: text/html

Content-Length: 10476

(跟随一个空行, 随后是需凭据页的 HTML 文本)。

HTTP OAuth 认证

OAuth 对于 Http 来说, 就是放在 Authorization header 中的不是用户名密码, 而是一个 token。微软的 Skydrive 就是使用这样的方式。

2、摘要认证

digest authentication (HTTP1.1 提出的基本认证的替代方法)

这个认证可以看做是基本认证的增强版本, 不包含密码的明文传递。

引入了一系列安全增强的选项: “保护质量”(qop)、随机数计数器由客户端增加、以及客户生成的随机数。

$$HA1 = MD5(A1) = MD5(\text{username} : \text{realm} : \text{password})$$

如果 qop 值为 “auth” 或未指定, 那么 HA2 为

$$HA2 = MD5(A2) = MD5(\text{method} : \text{digestURI})$$

如果 qop 值为 “auth-int”, 那么 HA2 为

$$HA2 = MD5(A2) = MD5(\text{method} : \text{digestURI} : MD5(\text{entityBody}))$$

如果 qop 值为 “auth” 或 “auth-int”, 那么如下计算 response:

$$\text{response} = MD5(HA1 : \text{nonce} : \text{nonceCount} : \text{clientNonce} : \text{qop} : HA2)$$

如果 qop 未指定, 那么如下计算 response:

$$\text{response} = MD5(HA1 : \text{nonce} : HA2)$$

在 HTTP 摘要认证中使用 MD5 加密是为了达成“不可逆的”, 也就是说, 当输出已知的时候, 确定原始的输入应该是相当困难的。如果密码本身太过简单, 也许可以

通过尝试所有可能的输入来找到对应的输出 (穷举攻击), 甚至可以通过字典或者适当的查找表加快查找速度。

示例及说明

下面的例子仅仅涵盖了“auth”保护质量的代码, 因为在撰写期间, 所知道的只有 Opera 和 Konqueror 网页浏览器支持“auth-int”(带完整性保护的认证)。

典型的认证过程包括如下步骤:

客户端请求一个需要认证的页面, 但是不提供用户名和密码。通常这是由于用户简单的输入了一个地址或者在页面中点击了某个超链接。

服务器返回 401 "Unauthorized" 响应代码, 并提供认证域(realm), 以及一个随机生成的、只使用一次的数值, 称为密码随机数 nonce。

此时, 浏览器会向用户提示认证域(realm) (通常是所访问的计算机或系统的描述), 并且提示用户名和密码。用户此时可以选择取消。

一旦提供了用户名和密码, 客户端会重新发送同样的请求, 但是添加了一个认证头包括了响应代码。

注意: 客户端可能已经拥有了用户名和密码, 因此不需要提示用户, 比如以前存储在浏览器里的。

客户端请求 (无认证):

GET /dir/index.html HTTP/1.0

Host: localhost

(跟随一个新行, 形式为一个回车再跟一个换行)

服务器响应:

HTTP/1.0 401 Unauthorized

Server: HTTPd/0.9

Date: Sun, 10 Apr 2005 20:26:47 GMT

WWW-Authenticate: Digest realm="testrealm@host.com", //认证域

qop="auth,auth-int", //保护质量

nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093", //服务器密码随机数

opaque="5ccc069c403ebaf9f0171e9517f40e41"

Content-Type: text/html

Content-Length: 311

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/1999/REC-html401-19991224/loose.dtd">
<HTML>
  <HEAD>
    <TITLE>Error</TITLE>
    <META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=ISO-8859-1">
  </HEAD>
  <BODY><H1>401 Unauthorized.</H1></BODY>
</HTML>
```

客户端请求 (用户名 "Mufasa", 密码 "Circle Of Life"):

GET /dir/index.html HTTP/1.0

Host: localhost

Authorization: Digest username="Mufasa",
 realm="testrealm@host.com",
 nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",
 uri="/dir/index.html",
 qop=auth,
 nc=00000001, //请求计数
 cnonce="0a4f113b", //客户端密码随机数
 response="6629fae49393a05397450978507c4ef1",
 opaque="5ccc069c403ebaf9f0171e9517f40e41"

(跟随一个新行, 形式如前所述)。

服务器响应:

HTTP/1.0 200 OK

Server: HTTPd/0.9

Date: Sun, 10 Apr 2005 20:27:03 GMT

Content-Type: text/html

Content-Length: 7984

(随后是一个空行, 然后是所请求受限制的 HTML 页面)

response 值由三步计算而成。当多个数值合并的时候, 使用冒号作为分割符:

- 1、对用户名、认证域(realm)以及密码的合并值计算 MD5 哈希值, 结果称为 HA1。
- 2、对 HTTP 方法以及 URI 的摘要的合并值计算 MD5 哈希值, 例如, "GET" 和 "/dir/index.html", 结果称为 HA2。
- 3、对 HA1、服务器密码随机数(nonce)、请求计数(nc)、客户端密码随机数(cnonce)、保护质量(qop)以及 HA2 的合并值计算 MD5 哈希值。结果即为客户端提供的 **response** 值。

因为服务器拥有与客户端同样的信息, 因此服务器可以进行同样的计算, 以验证客户端提交的 **response** 值的正确性。在上面给出的例子中, 结果是如下计算的。

(MD5()表示用于计算 MD5 哈希值的函数; “\n”表示接下一行; 引号并不参与计算)

HA1 = MD5("Mufasa:testrealm@host.com:Circle Of Life")

```
= 939e7578ed9e3c518a452acee763bce9
HA2 = MD5( "GET:/dir/index.html" )
= 39aff3a2bab6126f332b942af96d3366
Response = MD5( "939e7578ed9e3c518a452acee763bce9:\
dcd98b7102dd2f0e8b11d0f600bfb0c093:\
00000001:0a4f113b:auth:\
39aff3a2bab6126f332b942af96d3366" )
= 6629fae49393a05397450978507c4ef1
```

此时客户端可以提交一个新的请求，重复使用服务器密码随机数(nonce)（服务器仅在每次“401”响应后发行新的 nonce），但是提供新的客户端密码随机数(cnonce)。在后续的请求中，十六进制请求计数器(nc)必须比前一次使用的时候要大，否则攻击者可以简单的使用同样的认证信息重放老的请求。由服务器来确保在每个发出的密码随机数 nonce 时，计数器是在增加的，并拒绝掉任何错误的请求。显然，改变 HTTP 方法和/或计数器数值都会导致不同的 response 值。

服务器应当记住最近所生成的服务器密码随机数 nonce 的值。也可以在发行每一个密码随机数 nonce 后，记住过一段时间让它们过期。如果客户端使用了一个过期的值，服务器应该响应“401”状态号，并且在认证头中添加 stale=TRUE，表明客户端应当使用新提供的服务器密码随机数 nonce 重发请求，而不必提示用户其它用户名和口令。

服务器不需要保存任何过期的密码随机数，它可以简单的认为所有不认识的数值都是过期的。服务器也可以只允许每一个服务器密码随机数 nonce 使用一次，当然，这样就会迫使客户端在发送每个请求的时候重复认证过程。需要注意的是，在生成后立刻过期服务器密码随机数 nonce 是不行的，因为客户端将没有任何机会来使用这个 nonce。

PS：以上只介绍了两种比较基础的，还有其他的一些认证方式就不在这里一一说明了。

十二、HTTPS 传输协议原理

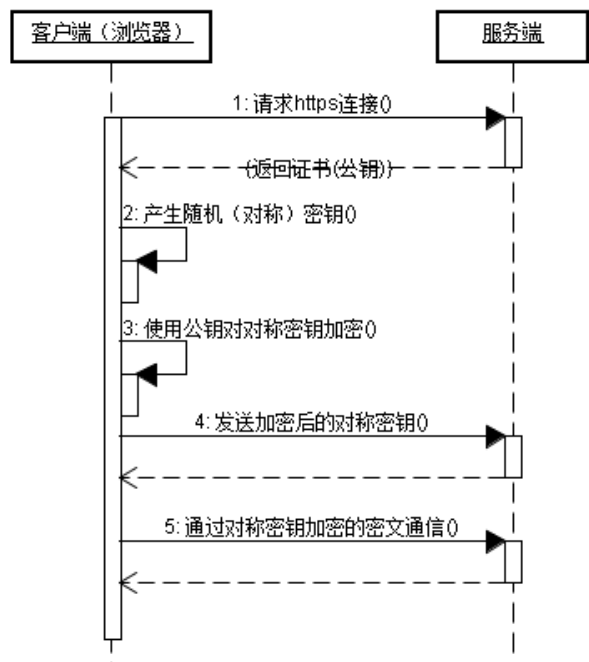
HTTPS（全称：Hypertext Transfer Protocol over Secure Socket Layer），是以安全为目标的 HTTP 通道，简单讲是 HTTP 的安全版。即 HTTP 下加入 SSL 层，HTTPS 的安全基础是 SSL，因此加密的详细内容请看 SSL。

1、两种基本的加解密算法类型

对称加密：密钥只有一个，加密解密为同一个密码，且加解密速度快，典型的对称加密算法有 DES、AES 等。

非对称加密：密钥成对出现（且根据公钥无法推知私钥，根据私钥也无法推知公钥），加密解密使用不同密钥（公钥加密需要私钥解密，私钥加密需要公钥解密），相对对称加密速度较慢，典型的非对称加密算法有 RSA、DSA 等。

2、HTTPS 通信过程



3、HTTPS 通信的优点

客户端产生的密钥只有客户端和服务端能得到；加密的数据只有客户端和服务端才能得到明文；客户端到服务端的通信是安全的。

十三、http 的状态响应码

1**(信息类): 表示接收到请求并且继续处理
100——客户必须继续发出请求
101——客户要求服务器根据请求转换 HTTP 协议版本
2**(响应成功): 表示动作被成功接收、理解和接受
200——表明该请求被成功地完成，所请求的资源发送回客户端
201——提示知道新文件的 URL
202——接受和处理、但处理未完成
203——返回信息不确定或不完整
204——请求收到，但返回信息为空
205——服务器完成了请求，用户代理必须复位当前已经浏览过的文件
206——服务器已经完成了部分用户的 GET 请求
3**(重定向类): 为了完成指定的动作，必须接受进一步处理
300——请求的资源可在多处得到
301——本网页被永久性转移到另一个 URL
302——请求的网页被转移到一个新的地址，但客户访问仍继续通过原始 URL 地址，重定向，新的 URL 会在 response 中的 Location 中返回，浏览器将会使用
303——建议客户访问其他 URL 或访问方式
304——自从上次请求后，请求的网页未修改过，服务器返回此响应时，不会返回网页内容，代表上次的文档已经被缓存了，还可以继续使用
305——请求的资源必须从服务器指定的地址得到
306——前一版本 HTTP 中使用的代码，现行版本中不再使用
307——申明请求的资源临时性删除

4(客户端错误类):** 请求包含错误语法或不能正确执行

400——客户端请求有语法错误，不能被服务器所理解

401——请求未经授权，这个状态代码必须和 WWW-Authenticate 报头域一起使用

HTTP 401.1 - 未授权：登录失败

HTTP 401.2 - 未授权：服务器配置问题导致登录失败

HTTP 401.3 - ACL 禁止访问资源

HTTP 401.4 - 未授权：授权被筛选器拒绝

HTTP 401.5 - 未授权：ISAPI 或 CGI 授权失败

402——保留有效 ChargeTo 头响应

403——禁止访问，服务器收到请求，但是拒绝提供服务

HTTP 403.1 禁止访问：禁止可执行访问

HTTP 403.2 - 禁止访问：禁止读访问

HTTP 403.3 - 禁止访问：禁止写访问

HTTP 403.4 - 禁止访问：要求 SSL

HTTP 403.5 - 禁止访问：要求 SSL 128

HTTP 403.6 - 禁止访问：IP 地址被拒绝

HTTP 403.7 - 禁止访问：要求客户证书

HTTP 403.8 - 禁止访问：禁止站点访问

HTTP 403.9 - 禁止访问：连接的用户过多

HTTP 403.10 - 禁止访问：配置无效

HTTP 403.11 - 禁止访问：密码更改

HTTP 403.12 - 禁止访问：映射器拒绝访问

HTTP 403.13 - 禁止访问：客户证书已被吊销

HTTP 403.15 - 禁止访问：客户访问许可过多

HTTP 403.16 - 禁止访问：客户证书不可信或者无效

HTTP 403.17 - 禁止访问：客户证书已经到期或者尚未生效

404——一个 404 错误表明可连接服务器，但服务器无法取得所请求的网页，请求资源不存在。eg：输入了错误的 URL

405——用户在 Request-Line 字段定义的方法不允许

406——根据用户发送的 Accept 拖，请求资源不可访问

407——类似 401，用户必须首先在代理服务器上得到授权

408——客户端没有在用户指定的饿时间内完成请求

409——对当前资源状态，请求不能完成

410——服务器上不再有此资源且无进一步的参考地址

411——服务器拒绝用户定义的 Content-Length 属性请求

412——一个或多个请求头字段在当前请求中错误

413——请求的资源大于服务器允许的大小

414——请求的资源 URL 长于服务器允许的长度

415——请求资源不支持请求项目格式

416——请求中包含 Range 请求头字段，在当前请求资源范围内没有 range 指示值，请求也不包含 If-Range 请求头字段

417——服务器不满足请求 Expect 头字段指定的期望值，如果是代理服务器，可能是下一级服务器不能满足请求长。

5(服务端错误类):** 服务器不能正确执行一个正确的请求

HTTP 500 - 服务器遇到错误，无法完成请求

HTTP 500.100 - 内部服务器错误 - ASP 错误

HTTP 500-11 服务器关闭

HTTP 500-12 应用程序重新启动

HTTP 500-13 - 服务器太忙

HTTP 500-14 - 应用程序无效

HTTP 500-15 - 不允许请求 global.asa

Error 501 - 未实现

HTTP 502 - 网关错误

HTTP 503: 由于超载或停机维护, 服务器目前无法使用, 一段时间后可能恢复正常

结束语：其他协议还有：文件传输协议 FTP、电子邮件传输协议 SMTP、域名系统服务 DNS、网络新闻传输协议 NNTP 和 HTTP 协议等。

第三部分 TOMCAT

一、下载与安装

安装目录不能包含中文和空格

JAVA_HOME 环境变量指定 Tomcat 运行时所要用的 jdk 所在的位置, 注意, 配到目录就行了, 不用指定到 bin

端口占用问题: netstat -ano 命令查看端口占用信息

Catalina_Home 环境变量: startup.bat 启动哪个 tomcat 由此环境变量指定,

如果不配置则启动当前 tomcat, 推荐不要配置此环境变量

二、目录结构

| - bin : 用于存放可执行命令(catalina.sh), 存放 tomcat 启动关闭所用的批处理文件

| - conf : 用于存放 tomcat 需要的配置文件, 最重要的是 server.xml

*实验:修改 servlet.xml,更改 tomcat 运行所在的端口号, 从 8080 改为 80

| - lib : 存放 tomcat 在运行时要用到的类文件(jsp-api.jar、 servlet-api.jar、 ...).

| - webapps : 最重要的一个目录, 其下放置各种 Servlet 文件、网页文件(JSP HTML ...)、配置文件以及资源文件.此目录的结构 :

| - 应用目录(比如是一个学生管理网站, 就在 webapps 文件夹下建一个 StudentManage 目录)

| - WEB-INF 目录

| - classes 目录, 放置所有的 Servlet 文件或其他类文件

| - lib 目录, 放置本应用所要用到的类文件(jar 包)

| - web.xml 配置文件

| - 资源文件(比如图片), 网页文件(JSP HTML ...)

| - logs : 日志文件 .

| - work : 内部临时文件.

| - temp : 临时文件

```
netstat -ano|findstr "80"
```

```
tasklist|findstr "2436"
```

1. IIS 占用 80 端口

用如下方法可以解决 System 进程占用 80 端口的问题:

打开 RegEdit:

找到 HKEY_LOCAL_MACHINE/SYSTEM/CurrentControlSet/Services/HTTP

找到一个 DWORD 值 **Start**, 将其改为 0

重启电脑, **System** 进程将不会占用 80 端口

2. 路由和远程访问服务占用 80

如果配置了 NAT 80 端口映射, 也是会出现 **System (PID4)** 占用 80 端口的情况。

此时, 停止路由和远程服务服务即可。

三、搭建服务器

1、配置虚拟主机

一个 **tomcat** 可以认为是一台真实主机. 在一台真实主机中可以配置多个站点, 这些站点在访问者看来访问他们就像在访问各自独立的主机一样, 所以我们可以认为这些站点都运行在 **tomcat** 这台真实主机当中的各自的虚拟主机当中. 一个网站就可以认为是一个虚拟主机

在 **conf/server.xml** 中 **<Engine>** 标签下配置 **<Host>** 标签就可以为 **tomcat** 增加一台虚拟主机了

name -- 指定虚拟主机的名称, 浏览器通过这个名称访问虚拟主机

appBase -- 虚拟主机管理的目录, 放置在这个目录下的 **web** 应用当前虚拟主机可以自动加载

* 由于浏览器访问地址时, 需要将地址翻译成对应的 **ip** 才能找到服务器, 这其中翻译的过程是由 **dns** 服务器来实现的.

我们在做实验的时候没有办法去修改 **dns** 服务器, 此时可以使用 **hosts** 文件模拟 **dns** 的功能, 从而完成实验.

* 缺省虚拟主机: 如果来访者是通过 **ip** 来访问, 这个时候服务器无法辨别当前要访问的是哪台虚拟主机中的资源, 此时访问缺省虚拟主机. 缺省虚拟主机可以在 **server.xml** 中 **engin** 标签上通过 **defaultHost** 属性进行配置.

eg:

```
<Engine name="Catalina" defaultHost="localhost">
    <Host name="localhost" appBase="webapps"
        unpackWARs="true" autoDeploy="true"
        xmlValidation="false" xmlNamespaceAware="false">
        <!--<Context path="" docBase="D:\Indeterminate\mycat"/>-->
    </Host>
    <Host name="www.mycat.com" appBase="D:\Indeterminate\mycat">
        <!--<Context path="" docBase="D:\Indeterminate\mycat"/>-->
    </Host>
</Engine>
```

2、为虚拟主机配置 web 应用

三种方式:

(1) 在 **Server.xml** 的 **<Host>** 标签中, 配置 **<Context>** 标签, 就可以为该虚拟主机配置一个 **web** 应用了

如果将 **path** 设置为空则这个 **web** 应用为缺省 **web** 应用

```
<Context path="" docBase="D:\Indeterminate\mycat"/>
```

这种配置方式需要重启服务器不推荐

(2) 在 **tomcat/conf/[Engine]/[Host]/** 在这个目录下写一个 **xml** 文件, 其中 **xml** 文件的名字就是虚拟路径, 在这个 **xml** 中可以配置 **<Context>** 标签, 其中配置真实路径.

```
<Context docBase="D:\Indeterminate\mycat"/>
```

如果所配置的虚拟路径中有/由于文件名中不允许包含/需要用#替代.

只要将文件名设置为 **ROOT.xml** 则这个 **xml** 描述的 **web** 应用就成为了缺省 **web** 应用

这种配置方式不需要重启服务器, 推荐

(3)直接将 web 应用放置到虚拟主机管理的目录下,虚拟主机就可以找到这个 web 应用,从而管理这个 web 应用

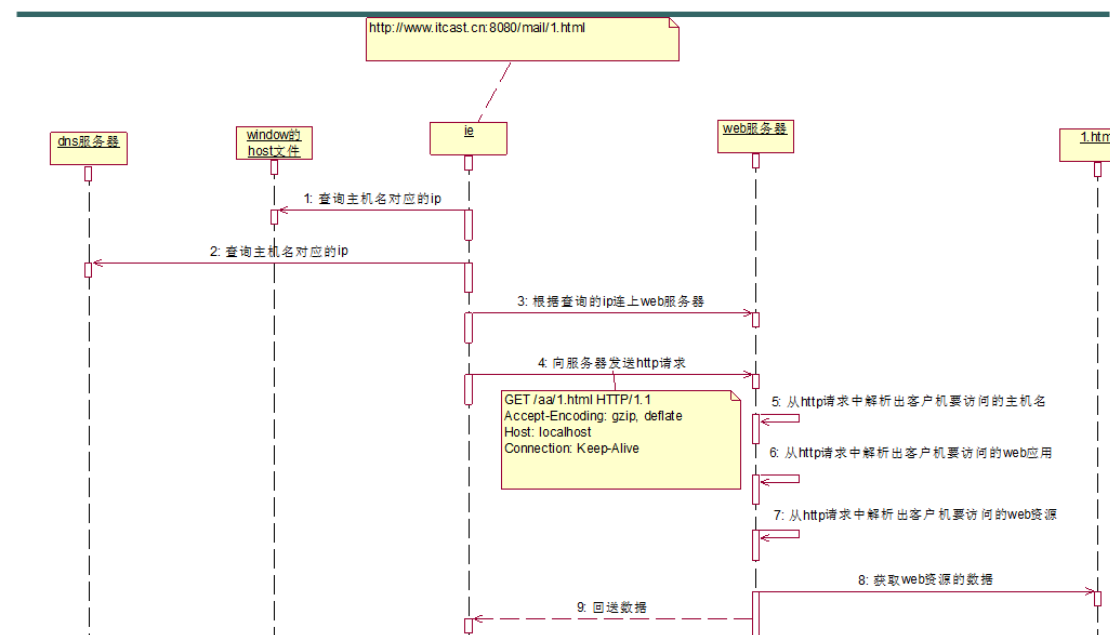
只要将 web 应用文件夹的名称改为 ROOT,这个 web 应用就是缺省 web 应用

*web 应用的虚拟路径映射 -- 就是为 web 应用的真实存在的路径配置一个浏览器访问的虚拟路径

*配置缺省 web 应用 -- 默认的 web 应用,当不写 web 应用名时就访问这个 web 应用

四、IE 与服务器交互图

IE与服务器交互图



五、其他工具

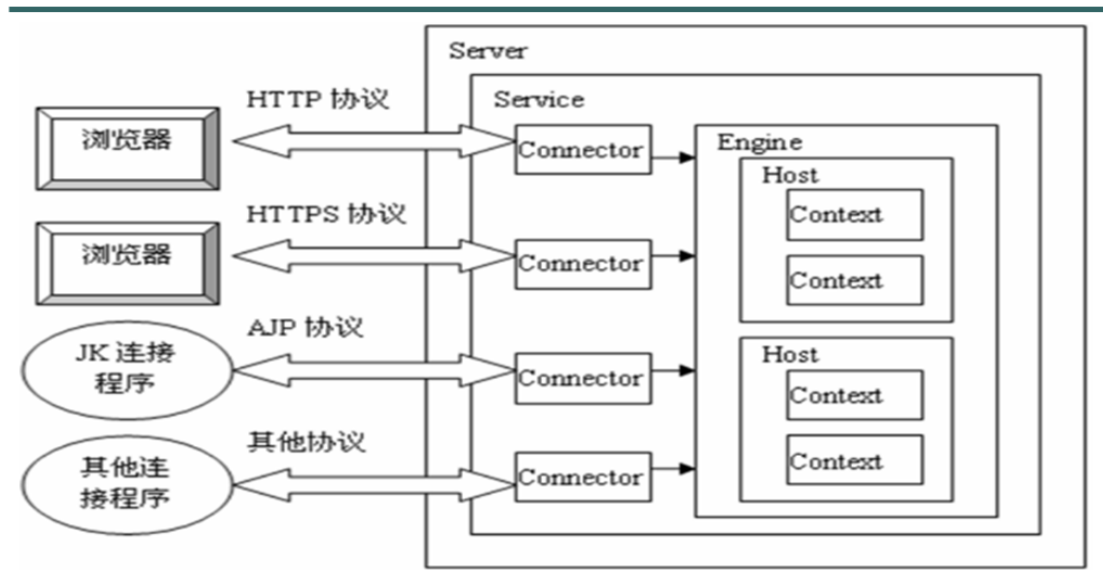
(1)打 war 包: 方式一: `jar -cvf news.war *` 方式二: 直接用压缩工具压缩为 zip 包, 该后缀为 .war

(2)通用 context 和通用 web.xml, 所有的<Context>都继承子 `conf/context.xml`,所有的 `web.xml` 都继承自 `conf/web.xml`

(3)reloadable 让 tomcat 自动加载更新后的 web 应用, 当 java 程序修改后不用重启, 服务器自动从新加载, 开发时设为 true 方便开发, 发布时设为 false, 提高性能

(4)Tomcat 管理平台, 可以在 `conf/tomcat-users.xml` 下配置用户名密码及权限

六、体系架构



七、context 元素常用属性

属 性	描 述
docBase	指定Web应用程序的文档基目录或者WAR文件的路径名。可以指定目录的或WAR文件的绝对路径名，也可以指定相对于Host元素的appBase目录的路径名。该属性是必需的
path	指定Web应用程序的上下文路径。在一个特定的虚拟主机中，所有的上下文路径都必须是唯一的。如果指定一个上下文路径为空字符串（""），则定义了这个虚拟主机的默认Web应用程序，负责处理所有的没有分配给其他Web应用程序的请求
reloadable	如果设置为ture，Tomcat服务器在运行时，会监视WEB-INF/classes和WEB-INF/lib目录下类的改变，如果发现类被更新，Tomcat服务器将自动重新加载该Web应用程序。这个特性在应用程序的开发阶段非常有用，但是它需要额外的运行时开销，所以在产品发布时不建议使用。该属性的默认值是false
unpackWAR	如果为true，Tomcat在运行Web应用程序前将展开所有压缩的Web应用程序。默认值是true

第三部分 SERVLET

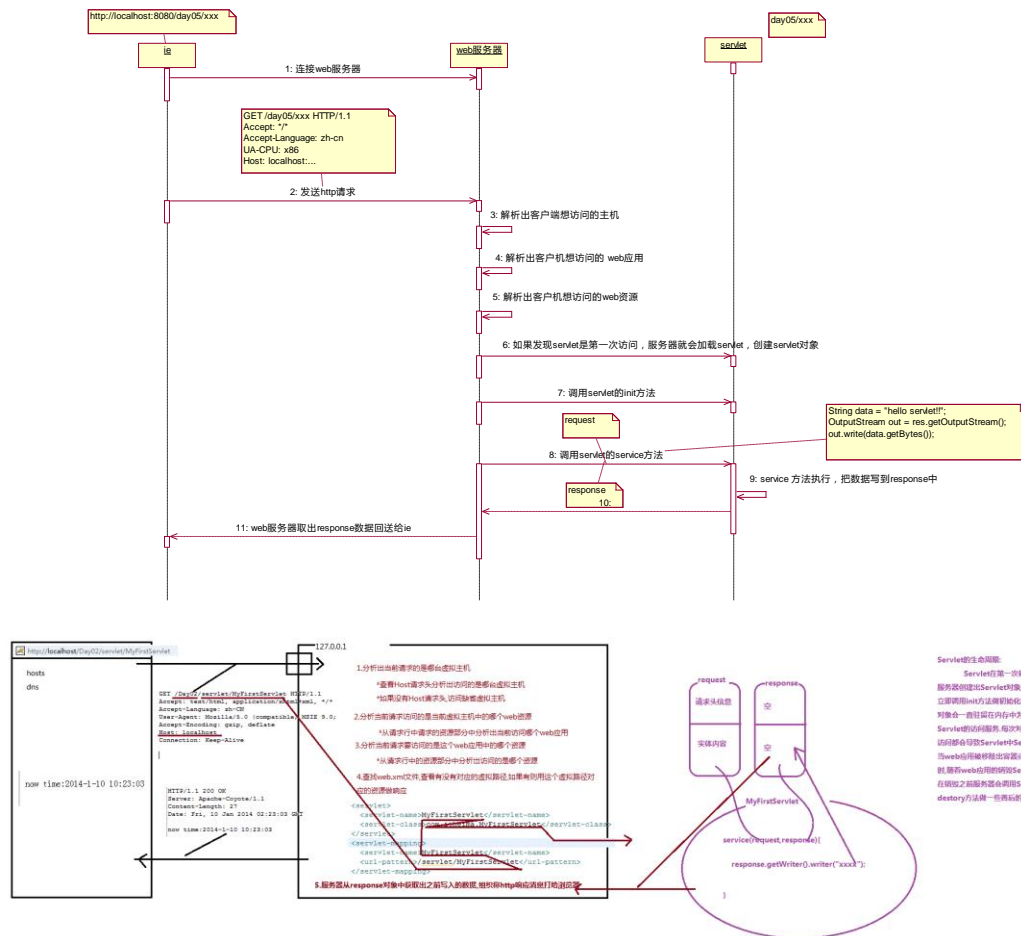
一、概述

sun 提供的一种动态 web 资源开发技术.本质上就是一段 java 小程序.可以将 Servlet 加入到 Servlet 容器中运行.

*Servlet 容器 -- 能够运行 Servlet 的环境就叫做 Servlet 容器. --- tomcat

*web 容器 -- 能够运行 web 应用的环境就叫做 web 容器 --- tomcat

二、Servlet 的调用过程/生命周期



1. 生命周期

一件事物什么时候生，什么时候死，在生存期间必然会做的事情，所有这些放在一起就是该事物的声明周期。

2. Servlet 的生命周期

1) Servlet 在容器中运行，其实例的创建及销毁等都不是由程序员决定的，而是由容器进行控制的。

创建 Servlet 实例有两个时机：1 客户端第一次请求某个 Servlet 时，系统创建该 Servlet 的实例：大部分的 Servlet 都是这种 Servlet。

2) Web 应用启动时立即创建 Servlet 实例，即 `load-on-startup <load-on-startup>1</load-on-startup>`

Servlet 的生命周期通过 `javax.servlet.Servlet` 接口中的 `init()`、`service()` 和 `destroy()` 方法来表示。

每个 Servlet 的运行都遵循如下生命周期。

(1) **加载和实例化**：找到 servlet 类的位置通过类加载器加载 Servlet 类，成功加载后，容器通过 Java 的反射 API 来创建 Servlet

实例，调用的是 Servlet 的默认构造方法（即不带参数的构造方法），

(2) **初始化**：容器将调用 Servlet 的 `init()` 方法初始化这个对象。初始化的目的是为了让 Servlet 对象在处理客户端请求前完成

一些初始化的工作，如建立数据库的连接，获取配置信息等。对于每一个 Servlet 实例，`init()` 方法只被调用一次

(3) **请求处理**：Servlet 容器调用 Servlet 的 `service(req,resp)` 方法对请求进行处理。要注意的是，在 `service()` 方法调用之前，`init()` 方法必须成功执行，对于每一次请求都掉用 `service(req,resp)` 方法处理请求，此时会用 Request 对象封装请求信息，并用 Response 对象（最初是空的）代表响应消息，传入到 `service` 方法里供使用。当 `service` 方法处理完成后，返回服务器服务器根据 Response 中的信息组织称响应消息返回给浏览器。

(4) 服务终止: 响应结束后 `servlet` 并不销毁, 一直驻留在内存中等待下一次请求。直到服务器关闭或 `web` 应用被移除出虚拟主机, `servlet` 对象销毁并在销毁前, 容器就会调用实例的 `destroy()` 方法, 以便让该实例可以释放它所使用的资源

三、Servlet 的继承结构

Servlet 接口 -- 定义了 Servlet 应该具有的基本方法, 所有的 Servlet 都应该直接或间接实现此接口。

|

|--GenericServlet --对 Servlet 接口的默认实现, 通用基本 Servlet 实现, 这是一个抽象类, 其中的大部分不常用的方法都做了默认实现, 只有 `service` 方法是一个抽象方法需要继承者自己实现。

|

|--HttpServlet --在通用 Servlet 的基础上基于 HTTP 协议进行了进一步的强化, 实现了 GenericServlet 中的 `Service` 抽象方法, 判断当前的请求方式, 调用对应到 `doXXX` 方法, 这样一来我们开发 Servlet 的过程中只需继承 `HttpServlet`, 覆盖具体要处理的 `doXXX` 方法就可以根据不同的请求方式实现不同的处理, 一般不要覆盖父类中的 `Service` 方法只要覆盖 `doGet/doPost` 就可以了。

四、Servlet 的细节

利用 `<servlet><servlet-mapping>` 标签注册一个 Servlet

```
<servlet>
```

```
    <servlet-name>FirstServlet</servlet-name>
```

```
    <servlet-class>cn.itheima.FirstServlet</servlet-class>
```

注意: 此处要的是一个 Servlet 的完整类名, 不是包含 `.java` 或 `.class` 扩展的文件路径

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

```
</servlet>
```

```
<servlet-mapping>
```

```
    <servlet-name>FirstServlet</servlet-name>
```

```
    <url-pattern>/FirstServlet</url-pattern>
```

```
</servlet-mapping>
```

(1) 一个 `<servlet>` 可以对应多个 `<servlet-mapping>`, 从而一个 Servlet 可以有多个路径来访问

(2) `url-pattern` 中的路径可以使用 * 匹配符号进行配置, 但是要注意, 只能是 / 开头 / * 结尾或 * 后缀这两种方式

~ 由于 * 的引入, 有可能一个路径被多个 `url-pattern` 匹配, 这是优先级判断条件如下:

哪个最像找哪个

* 后缀永远匹配级最低

(3) `<servlet>` 可以配置 `<load-on-startup>` 指定 `servlet` 随着服务器的启动而加载, 其中配置的数值指定启动的顺序

(4) 缺省 Servlet: 如果有一个 Servlet 的 `url-pattern` 被配置为了一根正斜杠 (/), 这个 Servlet 就变成了缺省 Servlet. 其他 Servlet 都不处理的请求, 由缺省 Servlet 来处理.

在 `conf/web.xml` 中配置了缺省 `servlet`, 对静态资源的访问和错误页面 (400/500) 的输出就是由这个缺省 `servlet` 来处理的. 如果我们自己写一个缺省 `servlet` 把爸爸 `web.xml` 中的缺省 `servlet` 覆盖的话, 会导致静态 `web` 资源无法访问. 所以通常我们不会自己去配置缺省 Servlet.

(5) 线程安全问题

```
int i = 0;
```

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

```
    throws ServletException, IOException {
```

```
    i++;
```

```
    try {
```

```
        Thread.sleep(4*1000);
```

```
    } catch (InterruptedException e) {
```

```
        e.printStackTrace();
    }
    response.getWriter().write(i+"");
}
}
```

终止：容器就会调用实例的 `destroy()` 方法，以便让该实例可以释放它所使用的资源

考点：从始至终只有一个对象，多线程通过线程池访问同一个 `servlet`

`Servlet` 采用多线程来处理多个请求同时访问，`Servlet` 容器维护了一个线程池来服务请求。

线程池实际上是等待执行代码的一组线程叫做工作者线程(`WorkerThread`)，`Servlet` 容器使用一个调度线程来管理工作者线程(`DispatcherThread`)。

当容器收到一个访问 `Servlet` 的请求，调度者线程从线程池中选出一个工作者线程，将请求传递给该线程，然后由该线程来执行 `Servlet` 的 `service` 方法。

当这个线程正在执行的时候，容器收到另外一个请求，调度者线程将从池中选出另外一个工作者线程来服务新的请求，容器并不关心这个请求是否访问的是同一个 `Servlet` 还是另外一个 `Servlet`。

当容器同时收到对同一 `Servlet` 的多个请求，那这个 `Servlet` 的 `service` 方法将在多线程中并发的执行。

如何现实 `servlet` 的单线程模式

```
<%@ page isThreadSafe="false" %>
```

==》由于默认情况下 `Servlet` 在内存中只有一个对象,当多个浏览器并发访问 `Servlet` 时就有可能产生线程安全问题，解决方案：

加锁(`synchronized`)-- 利用同步代码块解决问题。缺陷是，同一时间同步代码块只能处理一个 请求，效率很低下，所以同步代码块中尽量只包含核心的导致线程安全问题的代码。

`SingleThreadModel` 接口 --为该 `servlet` 实现 `SingleThreadModel` 接口，此为一个标记接口，被标记的 `servlet` 将会在内存中保存一个 `servlet` 池，如果一个线程来了而池中没有 `servlet` 对象处理，则创建一个新的。如果池中有空闲的 `servlet` 则直接使用。这并不能真的解决线程安全问题。此接口已经被废弃。

最终解决方案:在 `Servlet` 中尽量少用类变量,如果一定要用类变量则用锁来防止线程安全问题,但是要注意锁住内容应该是造成线程安全问题的核心代码,尽量的少锁主内容,减少等待时间提高 `servlet` 的响应速度

五、ServletConfig 对象

代表当前 `Servlet` 在 `web.xml` 中的配置信息

获取 `ServletConfig` 对象：

```
ServletConfig config = this.getServletConfig();
```

方法：

<code>String getServletName()</code>	-- 获取当前 <code>Servlet</code> 在 <code>web.xml</code> 中配置的名字
<code>String getInitParameter(String name)</code>	-- 获取当前 <code>Servlet</code> 指定名称的初始化参数的值
<code>Enumeration getInitParameterNames()</code>	-- 获取当前 <code>Servlet</code> 所有初始化参数的名字组成的枚举
<code>ServletContext getServletContext()</code>	-- 获取代表当前 <code>web</code> 应用的 <code>ServletContext</code> 对象

这些配置信息没什么大用处，我们还可以在 `ServletConfig` 中保存自己在 `web.xml` 文件中定义的数据

此时的 `web.xml` 文件片段如下：

```
<servlet>
<!-- 自己定义的，要保存在 ServletConfig 对象中的数据 -->
<init-param>
<param-name>jdbc.driver</param-name>
<param-value>oracle.jdbc.driver.OracleDriver</param-value>
</init-param>
<init-param>
```

```
<param-name>jdbc.user</param-name>
<param-value>yinkui</param-value>
</init-param>
...
<servlet-name>query</servlet-name>
<servlet-class>com.kettas.servlet.Query</servlet-class>
</servlet>
```

在 Servlet 中取得这些数据：

```
//getServletConfig 方法继承自父类 GenericServlet
ServletConfig sc = this.getServletConfig();

// 显然， getInitParameter 方法返回的只能是字符串类型数据
String driver = sc.getInitParameter("jdbc.driver");
String user = sc.getInitParameter("jdbc.user");

//通过枚举遍历，得到参数

ServletContext context = this.getServletContext();
Enumeration enumeration = context.getInitParameterNames();
while(enumeration.hasMoreElements()){
    String name = (String) enumeration.nextElement();
    String value = context.getInitParameter(name);
    System.out.println(name+"."+value);
}
```

注意：

1 ServletConfig 对象只能从 web.xml 文件中获取自定义数据(字符串数据)，不存在 setAttribute 方法去存入自定义数据。

2 在 Servlet 中，若要覆盖父类的 init(ServletConfig config)方法，必须这么做：

```
public void init( ServletConfig config ){
    // 覆盖之前调用父类的这个方法，否则 ServletConfig 对象会丢失
    // 此时 this.getServletConfig()返回的是 null，那样我们就不能使用它了
    super.init( config );
    ... }
```

六、ServletContext 对象

用来保存数据的全局唯一对象，一个应用中只有一个 ServletContext 对象，代表当前 web 应用

1、做为域对象可以在整个 web 应用范围内共享数据

域对象：一个域就理解为一个框，这里面可以放置数据，一个域既然称作域，他就有一个可以被看见的范围，这个范围内都可以对这个域中的数据进行操作，那这样的对象就叫做域对象

作用范围：整个 web 应用范围内共享数据，在不同 servlet 之间传递数据

生命周期：当 web 应用被加载进容器时创建代表整个 web 应用的 ServletContext 对象。当服务器关闭或 web 应用被移除出容器时，ServletContext 对象跟着销毁。

```
void setAttribute(String, Object);
Object getAttribute(String);
void removeAttribute(String);

*获取 ServletContext 对象的三种方法(this 指代当前 Servlet)

(1) ServletContext sc = this.getServletContext();
```

```
(2) ServletContext sc = this.getServletConfig().getServletContext();  
(3) ServletContext sc = request.getSession(true).getServletContext();
```

2、用来获取/设置 web 应用的初始化参数

请求参数 parameter --- 浏览器发送过来的请求中的参数信息

初始化参数 initparameter --- 在 web.xml 中为 Servlet 或 ServletContext 配置的初始化时带有的基本参数

域属性 attribute --- 四大作用于中存取的键值对

eg:

```
ServletContext context = this.getServletContext();  
Enumeration enumeration = context.getInitParameterNames();  
while(enumeration.hasMoreElements()){  
    String name = (String) enumeration.nextElement();  
    String value = context.getInitParameter(name);  
    System.out.println(name+"":"+value");  
}  
  
servletContext.setAttribute("name", data); // 两个参数分别为命名属性以及对应的数据  
// 取得 ServletContext 对象中的数据, 参数为命名属性  
// 返回的是 Object 对象, 故要强转
```

web.xml:

```
<?xml version="1.0" encoding="UTF-8"?>  
<web-app version="2.5"  
    xmlns="http://java.sun.com/xml/ns/javaee"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">  
    <context-param>  
        <param-name>username</param-name>  
        <param-value>朴乾</param-value>  
    </context-param>  
    <context-param>  
        <param-name>password</param-name>  
        <param-value>123</param-value>  
    </context-param>  
</web-app>
```

3、实现 Servlet 的转发

重定向 : 302+Location

请求转发 : 服务器内不进行资源流转

***请求转发是一次请求一次响应实现资源流转.请求重定向两次请求两次响应.**

```
this.getServletContext().getRequestDispatcher("/servlet/Demo6Servlet").forward(request, response);
```

方法执行结束, service 就会返回到服务器, 再有服务器去调用目标 servlet. 其中 request 会重新创建, 并将之前的 request 的数据拷贝进去。

4、加载资源文件

ServletContext 对象的一个重要方法 :

```
InputStream is = sc.getResourceAsStream( "fileName" );  
fileName : 使用的是虚拟目录, 不依赖于实际路径/books/ajax.pdf  
最左边一个"/" : web 应用的根目录  
// 获得实际路径 String path = ctx.getRealPath( "/books/ajax.pdf" )
```

在 Servlet 中读取资源文件时:

由于相对路径默认相对的是 java 虚拟机启动的目录, 所以我们直接写相对路径将会是相对于 tomcat/bin 目录 (程序启动的目录--在 web 环境下, 就是 tomcat 启动的目录即 tomcat/bin), 所以是拿不到资源的。如果写成绝对路径, 当项目发布到其他环境时, 绝对路径就错了, 所有找不到资源。

如果写硬盘路径, 可以找到资源, 但是只要一换发布环境, 这个硬盘路径很可能是错误的, 同样不行。

为了解决这样的问题 ServletContext 提供了 getRealPath 方法, 在这个方法中传入一个路径, 这个方法的底层会在传入的路径前拼接当前 web 应用的硬盘路径从而得到当前资源的硬盘路径, 这种方式即使换了发布环境, 方法的底层也能得到正确的 web 应用的路径从而永远都是正确的资源的路径

this.getServletContext().getRealPath("/1.properties"), 给进一个资源的虚拟路径, 将会返回该资源在当前环境下的真实路径。

this.getServletContext().getResourceAsStream("/1.properties"), 给一个资源的虚拟路径返回到该资源真实路径的流。

```
this.getServletContext().getRealPath("config.properties")
```

如果在非 Servlet 环境下要读取资源文件时, 就没有 ServletContext 对象, 可以采用类加载器加载文件的方式读取资源

```
Service.class.getClassLoader().getResource("../../config.properties").getPath()  
classLoader.getResourceAsStream("../../config.properties "), 此方法利用类加载器直接将资源加载到内存中, 有更新延迟的问题, 以及如果文件太大, 占用内存过大。
```

classLoader.getResource("../../config.properties ").getPath(), 直接返回资源的真实路径, 没有更新延迟的问题。

eg: config.properties

```
username=zhang  
password=123
```

```
File file = new File("config.properties");  
System.out.println(file.getAbsolutePath());
```

ServletContext 的环境下, 读取资源

```
Properties prop = new Properties();  
prop.load(new FileReader(this.getServletContext().getRealPath("config.properties")));
```

```
System.out.println(prop.getProperty("username"));
```

```
System.out.println(prop.getProperty("password"));
```

没有 ServletContext 的环境下, 读取资源

```
public void methodo1() throws FileNotFoundException, IOException{  
    //--在没有 ServletContext 的环境下, 如果想要读取资源, 可以使用类加载器以加载类的方式加载资源,  
    //      这里要注意, 类加载器从哪个目录加载类, 就从哪个目录加载资源,  
    //      所以此处的路径一定要给一个相对于类加载目录的路径  
    Properties prop = new Properties();  
    prop.load(new FileReader(Service.class.getClassLoader().getResource("../../config.properties").getPath()));  
    System.out.println(prop.getProperty("username"));  
    System.out.println(prop.getProperty("password"));  
}
```


在 Servlet 中调用

```
Service service = new Service();  
service.methdo1();
```

七、response 对象

1、Resonse 的继承结构

ServletResponse -- 通用的 response 提供了一个响应应该具有最基本的属性和方法

|

|-HttpServletResponse -- 在 ServletResponse 的基础上针对于 HTTP 协议增加了很多强化的属性和方法

2、Response 代表响应

于是响应消息中的 状态码、响应头、实体内容都可以由它进行操作,由此引申出如下实验

1)输出数据

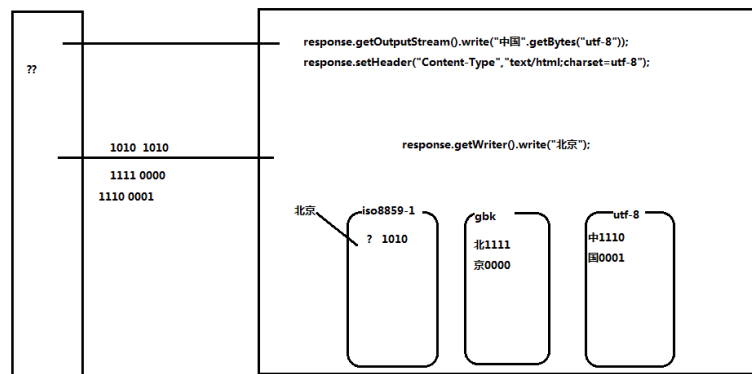
response.getOutputStream().write("中文".getBytes())输出数据,这是一个字节流,是什么字节输出什么字节,而浏览器默认用平台字节码打开服务器发送的数据,如果服务器端使用了非平台码去输出字符的字节数据就需要明确的指定浏览器编码时所用的码表,以防止乱码问题。response.addHeader("Content-type","text/html;charset=gb2312")

response.getWriter().write("中文");输出数据,这是一个字符流, response 会将此字符进行转码操作后输出到浏览器,这个过程默认使用 ISO8859-1 码表,而 ISO8859-1 中没有中文,于是转码过程中用?代替了中文,导致乱码问题。可以指定 response 在转码过程中使用的目标码表,防止乱码。response.setCharacterEncoding("gb2312");

其实 response 还提供了 setContentTyoe("text/html;charset=gb2312")方法,此方法会设置 content-type 响应头,通知浏览器打开的码表,同时设置 response 的转码用码表,从而一行代码解决乱码。

```
response.setContentType("text/html;charset=utf-8");  
response.getOutputStream().write("").getBytes("utf-8");  
  
response.setCharacterEncoding("utf-8");  
response.setContentType("text/html;charset=utf-8");  
response.getWriter().write("");
```

response 输出数据时乱码解决:



2)实现下载

利用 Response 设置 content-disposition 头实现文件下载

设置响应头 content-disposition 为 "attachment;filename=xxx.xxx"

利用流将文件读取进来,再利用 Response 获取响应流输出

如果文件名为中文，一定要进行 URL 编码，编码所用的码表一定要是 utf-8

eg:

```
response.setHeader("Content-Disposition", "attachment;filename="+URLEncoder.encode("XX.jpg", "utf-8"));
InputStream in = new FileInputStream(this.getServletContext().getRealPath("1.jpg"));
OutputStream out = response.getOutputStream();
byte[]bs = new byte[1024];
int i = 0;
while((i=in.read(bs))!=-1){
    out.write(bs,0,i);
}
in.close();
```

3)定时刷新页面

refresh 头控制定时刷新

设置响应头 Refresh 为一个数值，指定多少秒后刷新当前页面

设置响应头 Refresh 为 3;url=/Day05/index.jsp,指定多少秒后刷新到哪个页面

可以用来实现注册后“注册成功，3 秒后跳转到主页”的功能

在 HTML 可以利用<meta http-equiv= "" content="">标签模拟响应头的功能。

eg:

a.

```
<meta http-equiv="Refresh" content="3;url=/Day04/index.jsp">
```

b.

```
response.getWriter().write(new Date().toLocaleString());
response.setHeader("Refresh", "1");
```

c.

```
response.setContentType("text/html;charset=utf-8");
response.getWriter().write("恭喜您注册成功!3 秒后回到主页.....");
response.setHeader("refresh", "3;url=/Day04/index.jsp");
```

4)控制是否缓存资源

利用 response 设置 expires、Cache-Control、Pragma 实现浏览器是否缓存资源，这三个头都可以实现，但是由于历史原因，不同浏览器实现不同，所以一般配合这三个头使用

控制浏览器不要缓存(验证码图片不缓存)设置 expires 为 0 或-1 设置 Cache-Control 为 no-cache、Pragma 为 no-cache

控制浏览器缓存资源。即使不明确指定浏览器也会缓存资源，这种缓存没有截至日期。当在地址栏重新输入地址时会用缓存，但是当刷新或重新开浏览器访问时会重新获得资源。

如果明确指定缓存时间，浏览器缓存是，会有一个截至日期，在截至日期到期之前，当在地址栏重新输入地址或重新开浏览器访问时都会用缓存，而当刷新时会重新获得资源。

eg:

不缓存:

```
response.setIntHeader("Expires", -1);
response.setHeader("Cache-Control", "no-cache");
response.setHeader("Pragma", "no-cache");
response.setContentType("text/html;charset=utf-8");
response.getWriter().write("当前时间是:"+new Date().toLocaleString());
```

缓存:

```
response.setDateHeader("Expires", System.currentTimeMillis()+1000L*3600*24*30);//三个月
```

```
InputStream in = new FileInputStream(this.getServletContext().getRealPath("1.jpg"));
OutputStream out = response.getOutputStream();
byte[] bs = new byte[1024];
int i = 0;
while((i=in.read(bs))!=-1){
    out.write(bs,0,i);
}
in.close();
```

5)请求重定向

古老方法: response.setStatus(302);response.addHeader("Location","URL");

快捷方式: response.sendRedirect("URL");

eg: response.sendRedirect("/Day04/index.jsp");

6) 输出验证码图片

eg:

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    response.setDateHeader("Expires", -1);
    response.setHeader("Cache-Control", "no-cache");
    response.setHeader("Pragma", "no-cache");
    //1.在内存中构建出一张图片
    int height = 30;
    int width = 120;
    int xpyl = 5;
    int ypyl = 22;
    int bang = 20;
    BufferedImage img = new BufferedImage(width,height,BufferedImage.TYPE_INT_RGB);
    //2.获取图像上的画布
    Graphics2D g = (Graphics2D) img.getGraphics();
    //3.设置背景色
    g.setColor(Color.LIGHT_GRAY);
    g.fillRect(0, 0, width, height);
    //4.设置边框
    g.setColor(Color.BLUE);
    g.drawRect(0, 0, width-1, height-1);
    //5.画干扰线
    for(int i = 0;i<5;i++){
        g.setColor(Color.RED);
        g.drawLine(randNum(0, width), randNum(0, height), randNum(0, width), randNum(0, height));
    }
    //6.写字 (base 里是常用中文)
    String base = "略";
    for(int i = 0; i<4;i++){
        g.setColor(new Color(randNum(0,255),randNum(0,255),randNum(0,255)));
        g.setFont(new Font("黑体",Font.BOLD,bang));
```

```
int r = randNum(-45,45);

g.rotate(1.0*r/180*Math.PI, xpyl+(i*30), ypyl);

g.drawString(base.charAt(randNum(0, base.length()-1))+"" , xpyl+(i*30), ypyl);

g.rotate(1.0*-r/180*Math.PI, xpyl+(i*30), ypyl);

}

//将图片输出到浏览器

ImageIO.write(img, "jpg", response.getOutputStream());

}

private Random rand = new Random();

private int randNum(int begin,int end){

    return rand.nextInt(end-begin)+begin;

}
```

3、Response 注意的内容

对于一次请求，Response 的 `getOutputStream` 方法和 `getWriter` 方法是互斥，只能调用其一，特别注意 `forward` 后也不要违反这一规则。

利用 Response 输出数据的时候，并不是直接将数据写给浏览器，而是写到了 Response 的缓冲区中，等到整个 `service` 方法返回后，由服务器拿出 response 中的信息组成 HTTP 响应消息返回给浏览器。

`service` 方法返回后，服务器会自己检查 Response 获取的 `OutputStream` 或者 `Writer` 是否关闭，如果没有关闭，服务器自动帮你关闭，一般情况下不要自己关闭这两个流

八、Request 对象

1. Request 的继承结构

`ServletRequest` -- 通用 request，提供一个 request 应该具有的最基本的方法

|

|--`HttpServletRequest` -- `ServletRequest` 的孩子，针对 http 协议进行了进一步的增强

2、Request 代表请求对象

其中封装了对请求中具有请求行、请求头、实体内容的操作的方法

1) 获取客户机信息

`getRequestURL` 方法返回客户端发出请求完整 URL

`getRequestURI` 方法返回请求行中的资源名部分,在权限控制中常用

`getQueryString` 方法返回请求行中的参数部分

`getRemoteAddr` 方法返回发出请求的客户机的 IP 地址

`getMethod` 得到客户机请求方式

`getContextPath` 获得当前 web 应用虚拟目录名称，特别重要!!!，工程中所有的路径请不要写死，其中的 web 应用名要以此方法去获得

eg:

//1.获取客户端请求的完整 URL

```
String url = request.getRequestURL().toString();
```

```
System.out.println(url);
```

//2.获取客户端请求的资源部分的名称

```
String uri = request.getRequestURI();
```

```
System.out.println(uri);
```

//3.获取请求行中参数部分

```
String qStr = request.getQueryString();
```

```
System.out.println(q$tr);
//4.获取请求客户端的 ip 地址
String ip = request.getRemoteAddr();
System.out.println(ip);
//5.获取客户机的请求方式
String method = request.getMethod();
System.out.println(method);
//6.获取当前 web 应用的名称
String name = request.getContextPath();
System.out.println(name);
response.sendRedirect(request.getContextPath()+"/index.jsp");
```

2)获取请求头信息

getHeader(name)方法 --- String ， 获取指定名称的请求头的值

getHeaders(String name)方法 --- Enumeration<String> ， 获取指定名称的请求头的值的集合，因为可能出现多个重名的请求头

getHeaderNames 方法 --- Enumeration<String> ， 获取所有请求头名称组成的集合

getIntHeader(name)方法 --- int ， 获取 int 类型的请求头的值

getDateHeader(name)方法 --- long(日期对应毫秒) ， 获取一个日期型的请求头的值，返回的是一个 long 值，从 1970 年 1 月 1 日 0 时开始的毫秒值

eg:

```
Enumeration<String> enumeration = request.getHeaderNames();
```

```
while(enumeration.hasMoreElements()){
    String name = enumeration.nextElement();
    String value = request.getHeader(name);
    System.out.println(name+"."+value);
}
```

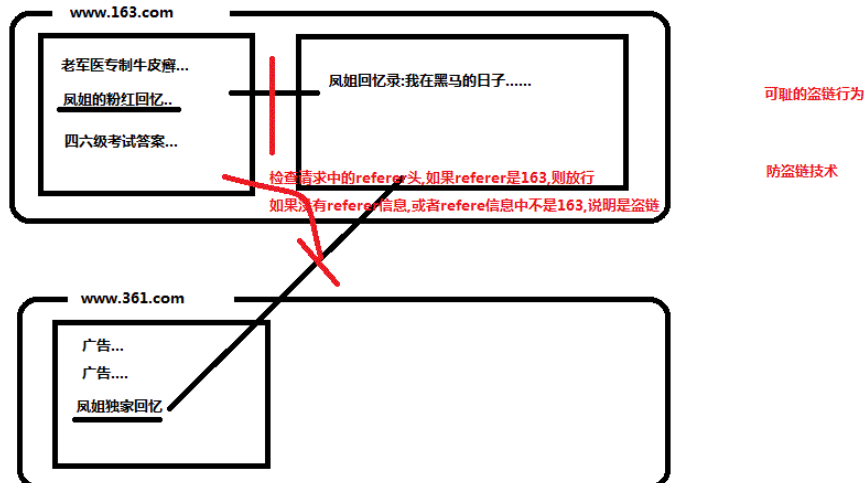
*实验：通过 referer 信息防盗链

```
String ref = request.getHeader("Referer");
if (ref == null || ref == "" || !ref.startsWith("http://localhost")) {
    response.sendRedirect(request.getContextPath() + "/homePage.html");
} else {
    this.getServletContext().getRequestDispatcher("/WEB-INF/fengjie.html").forward(request, response);
}
```

eg:

```
response.setContentType("text/html;charset=utf-8");
String ref = request.getHeader("Referer");
if(ref==null || "".equals(ref) || !ref.startsWith("http://localhost")){
    response.sendRedirect(request.getContextPath()+"/index.html");
    return;
}
response.getWriter().write("这信息有防盗链.....");
```

防盗链讲解：



3) 获取请求参数

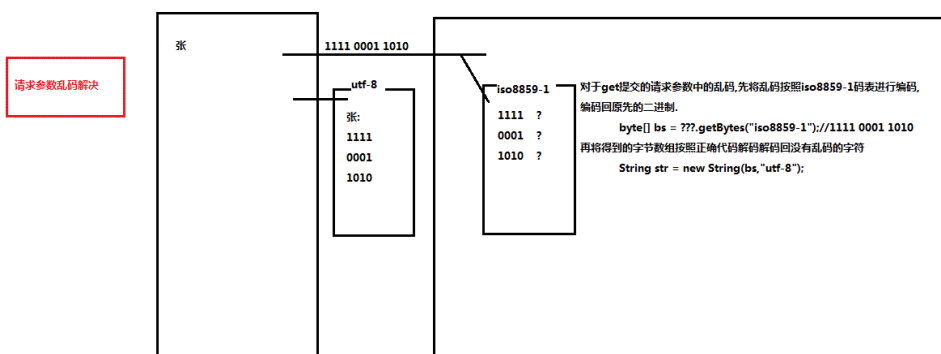
`getParameter(name)` --- String 通过 name 获得值

`getParameterValues(name)` --- String[] 通过 name 获得多值 checkbox

`getParameterNames` --- Enumeration<String> 获得所有请求参数名称组成的枚举

`getParameterMap` --- Map<String,String[]> 获取所有请求参数的组成的 Map 集合, 注意, 其中的键为 String, 值为 String[]

获取请求参数时乱码问题:



浏览器发送的请求参数使用什么编码呢? 当初浏览器打开网页时使用什么编码, 发送就用什么编码。

服务器端获取到发过来的请求参数默认使用 ISO8859-1 进行解码操作, 中文一定有乱码问题

对于 Post 方式提交的数据, 可以设置 `request.setCharacterEncoding("gb2312");` 来明确指定获取请求参数时使用编码。但是此种方式只对 Post 方式提交有效。

对于 Get 方式提交的数据, 就只能手动解决乱码:

```
String username = request.getParameter("username");
username = new String(username.getBytes("iso8859-1"), "utf-8");
```

此种方法对 Post 方式同样有效。

在 tomcat 的 `server.xml` 中可以配置 http 连接器的 `URIEncoding` 可以指定服务器在获取请求参数时默认使用的编码, 从而一劳永逸的解决获取请求参数时的乱码问题。也可以指定 `useBodyEncodingForURI` 参数, 令 `request.setCharacterEncoding` 也对 GET 方式的请求起作用, 但是这俩属性都不推荐使用, 因为发布环境往往不允许修改此属性。

eg:

```
//对于 get 提交只能手动解决请求参数中的乱码
```

```
String username = request.getParameter("username");
username = new String(username.getBytes("iso8859-1"), "utf-8");
System.out.println(username);
// Enumeration<String> enumeration = request.getParameterNames();
// while(enumeration.hasMoreElements()){
//     String name = enumeration.nextElement();
//     String value = request.getParameter(name);
//     System.out.println(name+"-"+value);
// }
```

4)利用 请求域传递对象

生命周期：在 service 方法调用之前由服务器创建，传入 service 方法。整个请求结束，request 生命结束。

作用范围：整个请求链。

作用：在整个请求链中共享数据，最常用的：在 Servlet 中处理好的数据要交给 Jsp 显示，此时参数就可以放置在 Request 域中带过去。

```
setAttribute
getAttribute
removeAttribute
eg:
// request.setAttribute("banana", "yellow banana");
// this.getServletContext().getRequestDispatcher("/servlet/Demo4").forward(request, response);
String result = "yellow banana";
request.setAttribute("result", result);
request.getRequestDispatcher("/show.jsp").forward(request, response);
```

5)request 实现请求转发

ServletContext 可以实现请求转发，request 也可以。

在 forward 之前输入到 response 缓冲区中的数据，如果已经被发送到了客户端，forward 将失败，抛出异常

在 forward 之前输入到 response 缓冲区中的数据，但是还没有发送到客户端，forward 可以执行，但是缓冲区将被清空，之前的数据丢失。注意丢失的只是请求体中的内容，头内容仍然有效。

在一个 Servlet 中进行多次 forward 也是不行的，因为第一次 forward 结束，response 已经被提交了，没有机会再 forward 了

总之，一条原则，一次请求只能有一次响应，响应提交走后，就再也没有机会输出数据给浏览器了。

```
this.getServletContext().getRequestDispatcher("").forward(request,response);
request.getRequestDispatcher("").forward(request,response);
```

请求转发是希望将请求交给另外一个资源执行，所以应该保证只有最后真正要执行的资源才能够输出数据，所以：

*请求转发时，如果已经有数据被写入了 response 的缓冲区，但是这些数据还没有被发送到客户端，则请求转发时，这些数据将会被清空。但是清空的只是响应中的实体内容部分，头信息并不会被清空。

*而请求转发时已经有数据被打给了浏览器，那么再进行请求转发，不能成功，会抛出异常，原因是响应已经结束了，再转发交给其他人没意义了

*在最终输出数据的 Servlet 执行完成后，response 实体内容中的数据将会被设置为已提交的状态，再往里写数据也不会起作用

-----使用以上三条，就保证了最终只有一个 Servlet 能够向浏览器输出数据，所以

*一个 Servlet 里两次请求转发也是不可以的，一次请求交给两人处理自然也是不行。

6)request 实现请求包含

将两个资源的输出进行合并后输出

```
this.getServletContext().getRequestDispatcher("").include(request,response);  
request.getRequestDispatcher("").include(request,response);
```

*被包含的 Servlet 程序不能改变响应消息的状态码和响应头, 如果它里面存在这样的语句, 这些语句的执行结果将被忽略

*常被用来进行页面布局

RequestDispatcher 进行 include 操作

forward 没有办法将多个 servlet 的输出组成一个输出, 因此 RequestDispatcher 提供了 include 方法, 可以将多个 Servlet 的输出组成一个输出返回给浏览器

```
request.getRequestDispatcher("/servlet/Demo17Servlet").include(request, response);  
response.getWriter().write("from Demo16");  
request.getRequestDispatcher("/servlet/Demo18Servlet").include(request, response);
```

常用在页面的固定部分单独写入一个文件, 在多个页面中 include 进来简化代码量。

3、三种资源处理方式的区别

请求重定向

```
response.sendRedirect();
```

请求转发

```
request.getRequestDispatcher().forward();
```

请求包含

```
request.getRequestDispatcher().include();
```

请求重定向和请求转发的区别:

请求重定向地址栏会发生变化.请求转发地址栏不发生变化.

请求重定向两次请求两次响应.请求转发一次请求一次响应.

如果需要在资源跳转时利用 request 域传递域属性则必须使用请求转发

如果希望资源跳转后修改用户的地址栏则使用请求重定向

如果使用请求转发也可以重定向也可以,则优先使用请求转发,减少浏览器对服务器的访问次数减轻服务器的压力.

RequestDispatcher.forward 方法只能将请求转发给同一个 WEB 应用中的组件; 而 HttpServletResponse.sendRedirect 方法还可以重定向到同一个站点上的其他应用程序中的资源, 甚至是使用绝对 URL 重定向到其他站点的资源。

如果传递给 HttpServletResponse.sendRedirect 方法的相对 URL 以 “/” 开头, 它是相对于服务器的根目录; 如果创建 RequestDispatcher 对象时指定的相对 URL 以 “/” 开头, 它是相对于当前 WEB 应用程序的根目录。

调用 HttpServletResponse.sendRedirect 方法重定向的访问过程结束后, 浏览器地址栏中显示的 URL 会发生变化, 由初始的 URL 地址变成重定向的目标 URL; 调用 RequestDispatcher.forward 方法的请求转发过程结束后, 浏览器地址栏保持初始的 URL 地址不变。

HttpServletResponse.sendRedirect 方法对浏览器的请求直接作出响应, 响应的结果就是告诉浏览器去重新发出对另外一个 URL 的访问请求; RequestDispatcher.forward 方法在服务器端内部将请求转发给另外一个资源, 浏览器只知道发出了请求并得到了响应结果, 并不知道在服务器程序内部发生了转发行为。

RequestDispatcher.forward 方法的调用者与被调用者之间共享相同的 request 对象和 response 对象, 它们属于同一个访问请求和响应过程; 而 HttpServletResponse.sendRedirect 方法调用者与被调用者使用各自的 request 对象和 response 对象, 它们属于两个独立的访问请求和响应过程。

2.应用场景（参照图想）

通常情况下都用请求转发, 减少服务器压力

当需要更新地址栏时用请求重定向，如注册成功后跳转到主页。

当需要刷新更新操作时用请求重定向，如购物车付款的操作。

九、URL 编码

1. 由于 HTTP 协议规定 URL 路径中只能存在 ASCII 码中的字符，所以如果 URL 中存在中文或特殊字符需要进行 URL 编码。

2. 编码原理：

将空格转换为加号（+）

对 0-9,a-z,A-Z 之间的字符保持不变

对于所有其他的字符，用这个字符的当前字符集编码在内存中的十六进制格式表示，并在每个字节前加上一个百分号（%）。如字符“+”用%2B 表示，字符“=”用%3D 表示，字符“&”用%26 表示，每个中文字符在内存中占两个字节，字符“中”用%D6%D0 表示，字符“国”用%B9%FA 表示调对于空格也可以使用其十六进制编码方式，即用%20 表示，而不是将它转换成加号（+）

说明：

如果确信 URL 串的特殊字符没有引起使用上的歧义或冲突你也可以对这些字符不进行编码，而是直接传递给服务器。例如，<http://www.it315.org/dealregister.html?name=中国&password=123>

如果 URL 串中的特殊字符可能会产生歧义或冲突，则必须对这些特殊字符进行 URL 编码。例如，服务器会将不编码的“中+国”当作“中国”处理。还例如，当 name 参数值为“中&国”时，如果不对其中的“&”编码，URL 字符串将有如下形式：

<http://www.it315.org/dealregister.html?name=中&国&password=123>，应编码为：

<http://www.it315.org/dealregister.html?name=中%26%20国&password=123>

<http://www.it315.org/example/index.html#section2> 可改写成-->

<http://www.it315.org/example%2Findex.html%23section2>

3. 在 java 中进行 URL 编码和解码

```
URLencoder.encode("xxx", "utf-8");
```

```
URLDecoder.decode(str, "utf-8");
```

十、常用地址的写法

绝对路径:以/开头的路径就叫做绝对路径,绝对路径在相对于的路径上直接拼接得到最终的路径

相对路径:不以/开头的路径就叫做相对路径,相对路径基于当前所在的路径计算的到最终的路径

硬盘路径:以盘符开头的路径就叫做硬盘路径.是哪个路径就是哪个路径.没有相对于谁的问题

虚拟路径: --写虚拟路径时都使用绝对路径

如果路径是给浏览器用的,这个路径相对于虚拟主机,所以需要写上 web 应用的名称

如果路径是个服务器用的,这个路径相对于 web 应用,所以可以省写 web 应用的名称

```
<a href="/Day04/....">
<form action="/Day04/...">

response.setHeader("Location","/Day04/....");
response.setHeader("refresh","3;url=/Day04/...");
response.sendRedirect("/Day04/...");
request.getRequestDispatcher("/index.jsp").forward();
request.getRequestDispatcher("/index.jsp").include();
```

真实路径: --写真实路径时都使用相对路径

根据原理,具体问题具体分析

```
servletContext.getRealPath("config.properties");//--给一个相对于 web 应用目录的路径
```

```
classLoader.getResource(".././config.properties");//--给一个相对于类加载目录的路径
```

```
File file = new File("config.properties");//--相对于程序的启动目录
```

```
new InputStream("config.properties");//--相对于程序的启动目录
```

十一、会话

1、概述

浏览器开始访问网站到访问网站结束期间产生的多次请求响应组合在一起叫做一次会话

会话的过程中会产生会话相关的数据，我们需要将这些数据保存起来。

Cookie：客户端技术

Session：服务器端技术

2、Cookie

Cookie 是基于 set-Cookie 响应头和 Cookie 请求头工作的,服务器可以发送 set-Cookie 请求头命令浏览器保存一个 cookie 信息,浏览器会在访问服务器时以 Cookie 请求头的方式带回之前保存的信息

```
request.getCookies();
```

```
response.addCookie(Cookie c);
```

```
new Cookie(String name,String value)//Cookie 在构造的时候就需要设定好 cookie 的名字和值
```

```
getName(); 获取该 cookie 的名字，注意没有 setName 方法，一个 Cookie 一旦创建出来就不能修改名字了
```

```
getValue();
```

```
setValue();
```

***setMaxAge 与 getMaxAge 方法**

一个 Cookie 如果没有设置过 MaxAge 则这个 Cookie 是一个会话级别的 Cookie,这个 Cookie 信息打给浏览器后浏览器会将它保存在浏览器的内存中,这意味着只要浏览器已关闭随着浏览器内存的销毁 Cookie 信息也就消失了.一个 Cookie 也可以设置 MaxAge,浏览一旦发现收到的 Cookie 被设置了 MaxAge,则会将这个 Cookie 信息以文件的形式保存在浏览器的临时文件夹中,保存到指定的时间到来位置.这样一来即使多次开关浏览器,由于这些浏览器都能在临时文件夹中看到 cookie 文件,所以在 cookie 失效之前 cookie 信息都存在.

想要命令浏览器删除一个 Cookie,发送一个同名同 path 的 cookie,maxage 设置为 0,浏览器以名字+path 识别 cookie,发现同名同 path,cookie 覆盖后立即超时被删除,从而就删除了 cookie.

maxAge = 60 表示：此 cookie 在客户端存在 1 分钟

两个特殊值：

maxAge = -1 表示：此 Cookie 生命周期由保存它的浏览器决定，（浏览器开则生，关则死），默认的

maxAge = 0 表示：删去以前的相应 cookie 存储

***setPath 与 getPath 方法**

-- 用来通知浏览器在访问服务器中的哪个路径及其子路径时带着当前 cookie 信息过来

如果不明确设置,则默认的路径是发送 Cookie 的 Servlet 所在的路径

http://localhost/Day05/servlet/...

setDomain 与 getDomain 方法

-- 用来通知浏览器在访问哪个域名的时候带着当前的 cookie 信息.但是要注意,现代的浏览器一旦发现 cookie 设置过 domain 信息则会拒绝接受这个 Cookie.我们平常不要设置这个方法

eg:

```

response.setContentType("text/html;charset=utf-8");
Cookie [] cs = request.getCookies();
Cookie findC = null;
if(cs!=null){
    for(Cookie c : cs){
        if("lastTime".equals(c.getName())){
            findC = c;
        }
    }
}
if(findC == null){
    response.getWriter().write("您是第一次访问本网站!");
}
}else{
    Long lastTime = Long.parseLong(findC.getValue());
    response.getWriter().write("您上次访问时间是:"+new Date(lastTime).toLocaleString());
}
Date date = new Date();
Cookie c = new Cookie("lastTime",date.getTime()+"");
c.setMaxAge(3600*24*30);
c.setPath(request.getContextPath());
//c.setDomain(".baidu.com");
response.addCookie(c);

```

!案例:曾经看过的书

```

eg:
String ids = "";
Cookie [] cs = request.getCookies();
Cookie findC = null;
if(cs!=null){
    for(Cookie c : cs){
        if("last".equals(c.getName())){
            findC = c;
        }
    }
}
if(findC == null){
    //说明之前没有看过书的记录
    ids += book.getId();
}else{
    //说明之前有历史看过的书的记录,需要根据历史记录算一个新的记录出来
    String [] olds = findC.getValue().split(",");
    StringBuffer buffer = new StringBuffer();
    buffer.append(book.getId()+",");
    for(int i = 0;i<olds.length && buffer.toString().split(",").length<3 ;i++){
        String old = olds[i];
        if(!old.equals(book.getId())){
            buffer.append(old+",");
        }
    }
    ids = buffer.substring(0, buffer.length()-1);
}

```

3、Session

Session 是一个域对象

!!作用范围:当前整个会话范围

!!生命周期:

创建: 当程序第一次调用到 `request.getSession(true)` 方法时说明客户端明确的需要用到 session 此时创建出对应客户端的 Session 对象.

销毁:

a) 当 session 最后使用超过 30 分钟(这个时间是可以可以在 `web.xml` 文件中进行修改的)没有人使用则认为 session 超时销毁这个 session. 可以在 `web.xml` 中配置 `<session-config>` 设置该时间值

b) 程序中明确的调用 `session.invalidate()` 方法可以立即杀死 session.

c) 当服务器被非正常关闭时,随着虚拟机的死亡而死亡.

*如果服务器是正常关闭,还未超时的 session 会被以文件的形式保存在服务器的 `work` 目录下,这个过程叫做 session 的钝化.下次再正常启动服务器时,钝化着的 session 会被恢复到内存中,这个过程叫做 session 的活化.

!!作用:在会话范围内共享数据

获取 `HttpSession` 对象的方法 :

```
// 参数为 true, 表示若存在对应的 HttpSession 对象, 则返回. 若不存在, 则创建一个新的.
```

```
// 若参数为 false, 表示若存在对应的 HttpSession 对象, 则返回. 若不存在, 则返回 null.
```

```
HttpSession session = request.getSession(true);
```

对 `HttpSession` 对象, 进行存取数据的操作

```
// 两个参数, 分别为命名属性和对应的数据
```

```
session.setAttribute("name", data);
```

```
// 一个参数, 命名属性, 注意返回的为 Object 对象, 要强转
```

```
session.getAttribute("name");
```

4、session 的原理

在服务器第一次调用 `request.getSession()` 方法的时候, 会在内存中创建一个 session 对象, 此对象具有一个独一无二的 id 值, 此 id 值将会以 cookie (`JSESSIONID`) 的形式发送给浏览器, 浏览器以后每次访问都会带着此 cookie, 服务器就利用此 cookie 区分浏览器找到对应的 session 空间. cookie (`"JSESSIONID",id`)

`request.getSession()` 方法会检查请求中有没有 `JSESSIONID` cookie, 如果有拿出他的值找到对应的 session 为他服务.

如果没有则检查请求的 URL 后有没有以参数的形式带着 `JSESSIONID` 过来, 如果有则找到对应的 Session 为浏览器服务器

如果还找不到则认为这个浏览器没有对应的 Session, 创建一个 Session 然后再在响应中添加 `JSESSIONID` cookie, 值就是这个 Session 的 id

默认情况下, `JSESSIONID` 的 path 为当前 web 应用的名称, 并且没有设置过 `MaxAge`, 是一个会话级别的 cookie.

这意味着一旦关闭浏览器再新开浏览器时, 由于 `JSESSIONID` 丢失, 会找不到之前的 Session

我们可以手动的发送 `JSESSIONID` cookie, 名字和 path 设置的和自动发送时一样, 但是设置一下 `MaxAge`, 使浏览器除了在内存中保存 `JSESSIONID` 信息以外还在临时文件夹中以文件的形式保存, 从而实现统一电脑中不同浏览器公用一个 `JSESSIONID`, 这样即使重开浏览器仍然可以使用之前的 session

eg:

```
String prod = request.getParameter("prod");
```

```
prod = new String(prod.getBytes("iso8859-1"), "utf-8");
```

```
HttpSession session = request.getSession();
```

```
Cookie jc = new Cookie("JSESSIONID", session.getId());
```

```
jc.setPath(request.getContextPath());
```

```
jc.setMaxAge(1800); // 以文件方式存储 30 分钟
```

```
response.addCookie(jc);
session.setAttribute("prod", prod);
```

5、URL 重写

使禁用 Cookie 的浏览器也可以使用 session: 由于 session 是基于 cookie 运行的, 如果禁用了 cookie 则会导致 session 不可用, 我们可以将提供给这种浏览器的所有的 URL 进行重写, 在所有的 URL 后跟上 JSESSIONID, 从而保证即使禁用了 Cookie 也能以 URL 的形式带回 JSESSIONID, 从而可以使用 session。要重写所有的 URL 是一项成本很高的工作, 一般我们不会这么做。

如果浏览器禁用了 Cookie, 浏览器就没有办法 JSESSIONID cookie, 这样就用不了 Session 了。

我们可以使用 URL 重写的机制, 在所有的超链接后都以参数的形式拼接 JSESSIONID 信息, 从而在点击超链接时可以使用 URL 参数的方式代替 JSESSIONID, 从而使用 Session

将 URL 进行重写拼接上 JSESSIONID 的过程就叫做 URL 重写

request.getSession() -- 在 URL 重写之前一定要先创建出 Session, 才有 Session id, 才能进行重写

response.encodeURL() --- 一般的地址都用这个方法重写

response.encodeRedirectURL() --- 如果地址是用来进行重定向的则使用这个方法

***url 重写的方法一旦发现浏览器带回了任意 cookie 信息, 则认为客户端没有禁用 cookie, 就不会再进行重写操作**

eg:

```
<%
    request.getSession();
    String url1 = request.getContextPath()+"/servlet/BuyServlet?prod=电视机";
    url1 = response.encodeURL(url1);
    String url2 = request.getContextPath()+"/servlet/BuyServlet?prod=冰箱";
    url2 = response.encodeURL(url2);
    String url3 = request.getContextPath()+"/servlet/PayServlet";
    url3 = response.encodeURL(url3);
%>

<a href="<%= url1 %>">电视机</a>
<a href="<%= url2 %>">冰箱</a>
<a href="<%= url3 %>">结账</a>

=====

response.setContentType("text/html;charset=utf-8");
HttpSession session = request.getSession();
String prod = (String) session.getAttribute("prod");
response.getWriter().write("您购买的是"+prod+"价值 9999999999 元");
```

6、cookie 与 session 对比

cookie 是客户端技术

数据保存在客户端, 这个信息可以保存很长时间

数据随时有可能被清空, 所以 cookie 保存的数据是不太靠谱的

数据被保存在了客户端, 随时有可能被人看走, 如果将一些敏感信息比如用户名密码等信息存在 cookie 中, 可能有安全问题

session 是服务器端技术

数据保存在服务器端, 相对来说比较稳定和安全

占用服务器内存, 所以一般存活的时间不会太长, 超过超时时间就会被销毁。我们要根据服务器的压力和 session 的使用情况合理设置 session 的超时时间, 既能保证 session 的存活时间够用, 同时不用的 session 可以及时销毁减少对服务器内存的占用。

Session 案例:

!!用户登录注销

防止表单重复提交

实现一次性验证码

十二、域总结

1、ServletContext、request、session 域的比较

servletContext 的作用域是整个 web 应用，随着服务器启动而创建，如果应用被移除出主机或服务器关闭则销毁。

request 的作用域是整个请求链，每一次请求都会创建一个 request，当请求结束时 request 销毁。

session 的作用域是整个会话，第一次调用 request.getSession 时创建，当一段时间没有使用或服务器关闭或调用 session.invalidate 方法时销毁

什么时候用 ServletContext 什么时候用 request 什么时候用 session?

如果一个数据只是用来显示的话就用 request 域

如果一个数据除了用来显示以外我还会还要用，这时候用 session

如果一个数据除了用来显示以外还要给别人用，这时候用 ServletContext 域

2、web 四大域

pageContext (称之为 page 域)

request (称之为 request 域)

session (称之为 session 域)

servletContext (称之为 application 域)

如果一个数据只在当前 jsp 页面使用,可以使用 pageContext 域存储

如果一个数据,除了在当前 Servlet 中使用,还要在请求转发时带到其他 Servlet 处理或 jsp 中显示,这个时候用 request 域

如果一个数据,除了现在我自己要用,过一会我自己还要用,存在 session 域

如果一个数据,除了现在我自己要用过一会其他人也要用,存在 ServletContext 域中

十三、Servlet 监听器、过滤器

1、介绍

监听器就是一个 java 程序，功能是监听另一个 java 对象变化（方法调用、属性变更）

javaee 提供了八种监听器，分为了三类

2、监听器监听过程

事件源、事件对象、监听器对象、操作事件源

1) 存在被监听对象（事件源）

2) 存在监听器对象

3) 在事件源中注册监听器

4) 操作事件源，使事件源发生改变，产生事件对象

* 事件对象 就是 事件源的改变

5) 事件对象会被传递给监听器，触发监听器相应行为

监听器技术主要应用于图形界面编程 ----- swing 中监听器的使用

3、自定义监听器

监听人的行为（监听器案例）

1) 创建事件源对象 Person

2) 创建监听器对象（通常定义为接口） PersonListener

3) 注册监听器

4) 触发事件源改变 --- 产生事件对象

4、监听器对象

Servlet 提供 8 个监听器，监听事件源主要是三个对象

ServletRequest, HttpSession, ServletContext -----Servlet 的三种数据范围对象

Servlet 的 8 个监听器，分为三类：

第一类：监听三个数据范围对象（request、session、ServletContext）的创建和销毁监听器

第二类：监听三个数据范围对象中属性变更（增加、替换、删除）的监听器

第三类：监听 HttpSession 中对象状态改变（被绑定、解除绑定、钝化、活化）的监听器

5、三个域对象创建和销毁监听器

1) ServletContextListener 监听 ServletContext 对象的创建和销毁事件

void contextInitialized(ServletContextEvent sce) ---- 监听 Context 对象创建

void contextDestroyed(ServletContextEvent sce) ----- 监听 Context 对象的销毁

ServletContext 全局唯一对象，每个工程创建唯一 Context 对象（配置全局初始化参数、保存全局共享数据、读取 web 资源文件）

在服务器启动时创建 ServletContext 对象，在服务器关闭时销毁 ServletContext 对象

6、编写监听器步骤

1) 编写类，实现特定监听器接口

2) Servlet 监听器，不是注册在事件源上，而是注册在 web.xml 中，由容器 tomcat 完成监听器注册

<!-- 注册监听器，tomcat 将监听器注册给事件源，事件源操作后，会自动监听器执行 -->

<!-- 和 Servlet Filter 不同，不需要配置 url -->

<listener>

<listener-class>cn.itcast.web.listener.MyServletContextListener</listener-class>

</listener>

7、监听器应用

1) 保存全局范围对象，因为监听 ServletContext 对象，监听器都可以通过事件对象获得事件源

// 获得被监听事件源对象

ServletContext context = sce.getServletContext();

2) 读取框架配置文件 例如：spring 框架 org.springframework.web.context.ContextLoaderListener

3) 在 ServletContextListener 定义一些定时器程序（任务调度程序）

8、任务调度

最简单 java 中任务调度 ----- 定时器 Timer 和 TimerTask 的使用

Timer 启动定时器任务

void schedule(TimerTask task, Date firstTime, long period) ----- 指定启动任务第一次时间，通过 period 参数指定任务重复执行

void schedule(TimerTask task, long delay, long period) ----- 指定任务距离当前时间 delay 多久开始启动，通过 period 指定任务重复执行

终止定时器任务执行 timer.cancel();

9、HttpSessionListener

监听 HttpSession 对象的创建和销毁

void sessionCreated(HttpSessionEvent se) ----- 监听 Session 对象创建

void sessionDestroyed(HttpSessionEvent se) ---- 监听 Session 对象销毁

Session 对象何时创建： request.getSession(); 第一次执行时 创建 Session 对象

* 访问 JSP 时，因为其内置对象 session，所以一定会创建 Session 对象的

Session 对象何时销毁：1) 不正常关闭服务器 2) Session 对象过期 3) invalidate 方法调用

* 正常关闭服务器时，Session 的数据保存 tomcat/work 目录下 --- 产生 SESSIONS.ser

* session 的过期时间在 web.xml 进行配置

```
<session-config>
    <!-- 单位是分钟，连续 30 分钟没有使用该 Session 对象，就会销毁对象 -->
    <session-timeout>30</session-timeout>
</session-config>
```

10、ServletRequestListener

监听 Request 对象的创建和销毁

void requestDestroyed(ServletRequestEvent sre) ----- 监听 request 对象销毁的

void requestInitialized(ServletRequestEvent sre) ----- 监听 request 对象创建的

每次客户端发起一次新的请求 产生 request 对象，当 response 响应结束后，request 对象进行销毁

forward 不会产生新的 request 对象，sendRedirect 产生新的 request 对象

11、监听器案例

案例一：统计当前在线人数

分析：统计 Session 的个数，存在一个 Session，意味着一个浏览器在访问

案例二：自定义 session 定时扫描器（销毁 session 对象）

编写定时器程序，定时去扫描系统中所有 Session 对象，发现如果一个 Session 1 分钟没有使用了，就销毁该 Session 对象

12、数据变更监听器

ServletRequest、HttpSession、ServletContext 三个数据范围中 数据变更监听器

ServletContextAttributeListener, HttpSessionAttributeListener ServletRequestAttributeListener

这三个接口中都定义了三个方法来处理被监听对象中的属性的增加，删除和替换的事件

1、attributeAdded 属性添加方法

```
public void attributeAdded(ServletContextAttributeEvent scae)
```

```
public void attributeAdded (HttpSessionBindingEvent hsbe)
```

```
public void attributeAdded(ServletRequestAttributeEvent srae)
```

2、attributeRemoved 属性移除方法

```
public void attributeRemoved(ServletContextAttributeEvent scae)
```

```
public void attributeRemoved (HttpSessionBindingEvent hsbe)
```

```
public void attributeRemoved (ServletRequestAttributeEvent srae)
```

3、attributeReplaced 属性替换方法

```
public void attributeReplaced(ServletContextAttributeEvent scae)
```

```
public void attributeReplaced (HttpSessionBindingEvent hsbe)
```

```
public void attributeReplaced (ServletRequestAttributeEvent srae)
```

何时调用 attributeAdded、attributeRemoved、attributeReplaced？

以 HttpSessionAttributeListener 为例

```
// 向 session 中添加了一个属性
session.setAttribute("name","张三"); // 因为 name 属性还不存在 --- attributeAdded

// 向 Session 保存属性 name 的值李四，因为 name 属性已经存在，替换效果
session.setAttribute("name","李四"); // 因为 name 属性以及存在 --- attributeReplaced

// 移除 session 的属性 name
session.removeAttribute("name"); // 移除 name 属性 ---- attributeRemoved
```

***** 如果执行 session.invalidate(); 销毁了 Session 导致 Session 中所有属性被移除

13、状态改变监听器

Session 中对象状态改变 监听器

Session 中对象共有四种状态

- 1、绑定 ----- 对象被添加到 Session 中
- 2、解除绑定 ----- 对象从 Session 中移除
- 3、钝化 ----- Session 中数据被序列化到硬盘上
- 4、活化 ----- Session 序列化数据 从硬盘被加载回内存

使 JavaBean 了解到自己在 Session 中状态的变化

HttpSessionBindingListener 感知到绑定以及解除绑定

HttpSessionActivationListener 感知钝化和活化 状态改变

不需要 web.xml 文件中进行注册 （这两类监听器，不是由容器进行管理的，由 HttpSession 对象管理）

HttpSessionBindingListener

void valueBound(HttpSessionBindingEvent event) 绑定

void valueUnbound(HttpSessionBindingEvent event) 解除绑定

编写 javabean 实现 HttpSessionBindingListener 接口，该 javabean 对象感知到自己被绑定到 Session 或者从 Session 解除绑定

valueBound ， 当对象被加入 session 就会执行

valueUnbound， 当对象从 Session 移除时 就会执行

HttpSessionActivationListener 监听对象被钝化和活化

void sessionDidActivate(HttpSessionEvent se) 活化

void sessionWillPassivate(HttpSessionEvent se) 钝化 （java 对象被序列化到硬盘上）

如果 tomcat 正常关闭，Session 中对象会被序列化到硬盘上 ---- java 对象如果能被序列化必须实现 Serializable 接口
异常：IOException while loading persisted sessions: java.io.WriteAbortedException: writing aborted;
java.io.NotSerializableException: cn.itcast.bean.Bean2

原因：恢复 Session.ser 文件时，因为需要对象无法从文件中加载，发生异常 （删除 Session.ser 就可以了）

* 保存到 Session 中对象，应该被序列化

在 Session 中数据对象，保存了一段时间后没有使用，不想删除对象中数据，（在不关闭服务器情况下）可以让对象数据

进行钝化 ---- 序列化到硬盘上

下次再访问数据时，从钝化的文件中读取序列化数据 ----- 对数据进行活化

* 由 tomcat 完成钝化和活化，配置<Context> 标签

配置 Context 标签有三个常用位置

- 1) tomcat/conf/context.xml 所有虚拟主机、所有 web 应用都可以使用配置
- 2) tomcat/conf/Catalina/localhost/context.xml 当前虚拟主机 所有 web 应用都可以使用配置
- 3) 当前工程/META-INF/context.xml 只对当前工程有效

配置钝化目录在 tomcat/work 下 ---- 和 Session 默认保存位置一样

十三、国际化

1、国际化的概念

一款软件希望不同的国家和地区的使用者都可以使用，这个时候软件中的一些内容和数据需要根据用户地区信息不同而展示成不同的样子。

2、国际化的组成部分

(1)页面中固定文本元素的国际化

ResourceBundle 资源包

<fmt>

(2)对程序动态产生的数据的国际化 -- 日期时间/货币

~1. 日期时间 DateFormat -- SimpleDateFormat

static DateFormat getDateInstance()

获取日期格式器，该格式器具有默认语言环境的默认格式化风格。

static DateFormat getDateInstance(int style)

获取日期格式器，该格式器具有默认语言环境的给定格式化风格。

static DateFormat getDateInstance(int style, Locale aLocale)

获取日期格式器，该格式器具有给定语言环境的给定格式化风格。

static DateFormat getTimeInstance()

获取时间格式器，该格式器具有默认语言环境的默认格式化风格。

static DateFormat getTimeInstance(int style)

获取时间格式器，该格式器具有默认语言环境的给定格式化风格。

static DateFormat getTimeInstance(int style, Locale aLocale)

获取时间格式器，该格式器具有给定语言环境的给定格式化风格。

static DateFormat getDateTimeInstance()

获取日期/时间格式器，该格式器具有默认语言环境的默认格式化风格。

static DateFormat getDateTimeInstance(int dateStyle, int timeStyle)

获取日期/时间格式器，该格式器具有默认语言环境的给定日期和时间格式化风格。

static DateFormat getDateTimeInstance(int dateStyle, int timeStyle, Locale aLocale)

获取日期/时间格式器，该格式器具有给定语言环境的给定格式化风格。

String format(Date date)

~2.货币

static NumberFormat getCurrencyInstance()

返回当前默认语言环境的货币格式。

static NumberFormat getCurrencyInstance(Locale inLocale)

返回指定语言环境的货币格式。

1000 --> \$1000.00 ¥1000.00

~3.消息国际化

十四、过滤器

1、介绍

Servlet 技术规范 描述三种技术：Servlet(服务器小程序)、Filter(过滤器)、Listener(监听器)

Filter 运行在服务器端，对服务器端 web 资源的访问 进行拦截，起到过滤的作用

Servlet API 中 定义接口 Filter，用户只需要编写程序实现 Filter 接口，完成过滤器编写

2、开发 Filter

想要开发一个过滤器需要如下两个步骤：

(1)写一个类实现特定的接口 Filter

生命周期:当服务器启动时,web 应用加载后,立即创建这个 web 应用中的所有的过滤器,过滤器创建出来后立即调用 init 方法执行初始化的操作.

创建出来后一直驻留在内存中为后续的拦截进行服务.每次拦截到请求后都会导致 doFilter 方法执行.

在服务器关闭或 web 应用被移除出容器时,随着 web 应用的销毁过滤器对象销毁.销毁之前调用 destory 方法执行善后工作.

init FilterConfig:代表 web.xml 中对当前过滤器的配置信息

~获取 ServletContext 对象

~获取初始化信息

getInitParameter

getInitParameterNames

doFilter

request

response

FilterChain:

代表过滤器链的对象.

一个资源可能被多个过滤器所拦截到,拦截的顺序和过滤器在 web.xml 中 filter-mapping 的配置顺序相同.

所有对当前资源访问进行拦截的过滤器按照拦截顺序就组成了一个过滤器链.这个过滤器链的最后一个节点是要访问的资源.

Filter 中调用 FilterChain 提供了 doFilter 方法,这个方法一旦被调用就表明当前过滤器没有问题了,请执行过滤器链的下一个节点.如果下一个节点是资源则直接执行了资源

destory

(2)在 web.xml 中注册一下过滤器(配置拦截哪个 web 资源) ----- web.xml

<filter>

<filter-name>Demo1Filter</filter-name> -- 给过滤器起一个名字

```
<filter-class>com.itheima.filter.Demo1Filter</filter-class> -- 过滤器的处理类

<init-param>--可以配置当前过滤器的初始化信息,可以配置多个,在 Filter 中利用 FilterConfig 对象来获取
    <param-name>name1</param-name>
    <param-value>value1</param-value>
</init-param>
</filter>

<filter-mapping> -- 一个 Filter 可以配置多个 filter-mapping
    <filter-name>Demo1Filter</filter-name>
    <url-pattern>/servlet/Demo1Servlet</url-pattern> -- 一个 Filtermapping 中可以配置多个 url-pattern,这个
url-pattern 的写法和 servlet-mapping 中的写法相同
    <url-pattern>/servlet/*</url-pattern>
    <url-pattern>/*</url-pattern>
    <url-pattern>*.do</url-pattern>
    <servlet-name>Demo3Servlet</servlet-name>
    --也可以配置多个 servlet-name,其中填入 servlet 的名字明确的通知要拦截哪个名字的 Servlet
    <dispatcher>REQUEST</dispatcher>
    --配置拦截哪种方式的对资源的访问可以是 REQUEST/FORWARD/INCLUDE/ERROR 四个值之中的一个,可以配置多个 dispatcher,如果一个都不配则默认是 REQUEST
    <dispatcher>FORWARD</dispatcher>
</filter-mapping>
```

FilterConfig 作用和 ServletConfig 类似,用来在 Filter 初始化阶段,将参数传递给过滤器

- 1) 通过 String getInitParameter(String name) 获得过滤器初始化参数
- 2) 通过 ServletContext getServletContext() 获得 ServletContext 对象

* FilterConfig 提供参数,是 Filter 类私有参数,Filter2 的初始化参数,不能在 Filter1 中进行获取

* 配置全局参数,<context-param> 进行配置,通过 ServletContext 获得

<filter-mapping> 过滤器拦截配置

- 1) 如果连接目标资源是一个 Servlet,可以选择 url 和 servlet 名称两种配置方式

```
<!-- 拦截/hello 是 Servlet 路径 -->
<url-pattern>/hello</url-pattern>

<!-- 拦截 Servlet 还可以通过 Servlet 名称进行拦截 -->
<servlet-name>HelloServlet</servlet-name>
```

- 2) url-pattern 和 Servlet 中路径写法一样,有三种: 完全匹配、目录匹配、扩展名匹配
- 3) <dispatcher>指定过滤器所拦截的资源被 Servlet 容器调用的方式

容器调用服务器端资源 有四种方式

REQUEST、FORWARD、INCLUDE、ERROR

3、Filter 应用

应用一: 统一全站字符编码过滤器

案例: 编写 jsp 输入用户名,在 Servlet 中获取用户名,将用户名输出到浏览器上

处理请求 post 乱码代码

```
request.setCharacterEncoding("utf-8");
```

设置响应编码集代码

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

经常会使用，而过滤器可以在目标资源之前执行，将很多程序中处理乱码公共代码，提取到过滤器中，以后程序中不需要处理编码问题了

应用二：禁止浏览器缓存动态页面的过滤器

因为动态页面数据，是由程序生成的，所以如果有缓存，就会发生，客户端查看数据不是最新数据情况，对于动态程序生成页面，设置浏览器端禁止缓存页面内容

```
response.setDateHeader("Expires",-1);
response.setHeader("Cache-Control","no-cache");
response.setHeader("Pragma","no-cache");
```

将禁用缓存代码，提到过滤器中，通过 url 配置，禁用所有 JSP 页面的缓存

应用三：控制浏览器缓存静态 web 资源

Tomcat 缓存策略

对于服务器端经常不变化文件，设置客户端缓存时间，在客户端资源缓存时间到期之前，就不会去访问服务器获取该资源
----- 比 tomcat 内置缓存策略更优手段

- * 减少服务器请求次数，提升性能

设置静态资源缓存时间，需要设置 Expires 过期时间，在客户端资源没有过期之前，不会产生对该资源的请求的

- * 设置 Expires 通常使用 response.setDateHeader 进行设置 设置毫秒值

应用四：自动登陆过滤器

在访问一个站点，登陆时勾选自动登陆（三个月内不用登陆），操作系统后，关闭浏览器；过几天再次访问该站点时，直接进行登陆后状态

在数据库中创建 user 表

```
create table user (
    id int primary key auto_increment,
    username varchar(20),
    password varchar(40),
    role varchar(10)
);

insert into user values(null,'admin','123','admin');
insert into user values(null,'aaa','123','user');
insert into user values(null,'bbb','123','user');
```

自动登陆：未登录、存在自动登陆信息、自动登陆信息正确

在用户完成登陆后，勾选自动登陆复选框，服务器端将用户名和密码以 Cookie 形式，保存在客户端。当用户下次访问该站点，AutoLoginFilter 过滤器从 Cookie 中获取自动登陆信息

- 1) 判断用户是否已经登陆，如果已经登陆，没有自动登陆的必要
- 2) 判断 Cookie 中是否含有自动登陆信息，如果没有，无法完成自动登陆
- 3) 使用 cookie 用户名和密码 完成自动登陆

如果将用户密码保存在 cookie 文件中，非常不安全的，通常情况下密码需要加密后才能保存到客户端

* 使用 md5 算法对密码进行加密

* md5 加密算法是一个单向加密算法，支持明文---密文 不支持密文解密

MySQL 数据库中提供 md5 函数，可以完成 md5 加密

```
mysql> select md5('123');
```

```
+-----+
```

```
| md5('123') |
```

```
+-----+
```

```
| 202cb962ac59075b964b07152d234b70 |
```

```
+-----+
```

解密后结果是 32 位数字 16 进制表示

Java 中提供类 MessageDigest 完成 MD5 加密

将数据表中所有密码 变为密文 update user set password = md5(password);

在 Demo4Servlet 登陆逻辑中，对密码进行 md5 加密

在 AutoLoginFilter 因为从 Cookie 中获得就是加密后密码，所以登陆时无需再次加密

MD5 在 2004 年被王小云破解，md5 算法是多对一加密算法，出现两个加密后相同密文的明文很难发现，王小云并没有研究出 md5 解密算法，研究出一种提高碰撞概率的算法

应用五：过滤器实现 URL 级别权限认证

系统中存在很多资源，将需要进行权限控制的资源，放入特殊路径中，编写过滤器管理访问特殊路径的请求，如果没有相应身份和权限，控制无法访问

认证：who are you？用户身份的识别 ----- 登陆功能

权限：以认证为基础 what can you do？您能做什么？必须先登陆，才有身份，有了身份，才能确定可以执行哪些操作

4、Filter 高级应用

Decorator 模式

- 1) 包装类需要和被包装对象 实现相同接口，或者继承相同父类
- 2) 包装类需要持有 被包装对象的引用

在包装类中定义成员变量，通过包装类构造方法，传入被包装对象

- 3) 在包装类中，可以控制原来那些方法需要加强

不需要加强，调用被包装对象的方法

需要加强，编写增强代码逻辑

ServletRequestWrapper 和 HttpServletRequestWrapper 提供对 request 对象进行包装的方法，但是默认情况下每个方法都是调用原来 request 对象的方法，也就是说包装类并没有对 request 进行增强

在这两个包装类基础上，继承 HttpServletRequestWrapper，覆盖需要增强的方法即可

应用六：完全解决 get 和 post 乱码的过滤器

在 Filter 中，对 request 对象进行包装，增强获得参数的方法

getParameter

getParameterValues

getParameterMap

ServletResponseWrapper 和 HttpServletResponseWrapper 提供了对 response 对象包装，继承 HttpServletResponseWrapper，覆盖需要增强 response 的方法

应用七：增强 Response 对象，对响应数据进行压缩

复习：Tomcat 服务器内，提供对响应压缩 配置实现

在 conf/server.xml 中

```
<Connector port="80" protocol="HTTP/1.1"
```

```
    connectionTimeout="20000"
```

```
    redirectPort="8443"/> 添加 compressableMimeType 和 compression
```

没有压缩：00:00:00.000 0.063 7553 GET 200 text/html http://localhost/

```
<Connector port="80" protocol="HTTP/1.1"
```

```
    connectionTimeout="20000"
```

```
    redirectPort="8443" compressableMimeType="text/html,text/xml,text/plain" compression="on"/>
```

压缩后：00:00:00.000 0.171 2715 GET 200 text/html http://localhost/

Content-Encoding: gzip

Content-Length : 2715

实际开发中，很多情况下，没有权限配置 server.xml，无法通过 tomcat 配置开启 gzip 压缩

编写过滤器对响应数据进行 gzip 压缩

flush 方法

只有没有缓冲区字节流，FileOutputStream 不需要 flush

而字节数组 ByteArrayOutputStream、字节包装流、字符流 需要 flush ---- 这些流在调用 close 方法时都会自动 flush

过滤器

1) 过滤器编写步骤

2) 全局编码、禁用缓存、设置过期时间 过滤器

3) 自动登陆/URL 级别权限控制 --- 必须掌握 *****

4) 通用 get/post 乱码过滤器和响应压缩过滤器 ----- 理解实现过程，保存起来会使用

客户信息系统增删改查必备

掌握分页查询原理

十五、文件上传下载

1、文件上传

1) 提供表单允许用户通过表单选择文件进行上传

```
<input type="file" />
```

注意事项:

- 1) 必须为文件上传 input 提供 name 属性，否则文件上传内容不会被表单提交
- 2) 表单的提交是 post (get 提交数据在 url 地址上显示，有长度限制)
- 3) 设置 enctype=multipart/form-data 使得文件上传编码 ----- MIME 编码格式

POST /Day15/upload.jsp HTTP/1.1

Accept: text/html, application/xhtml+xml, */*

Referer: http://localhost/Day15/upload.jsp

Accept-Language: zh-CN

User-Agent: Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0)

Content-Type: multipart/form-data; boundary=-----7de1e62806e0

Accept-Encoding: gzip, deflate

Host: localhost

Content-Length: 394

Connection: Keep-Alive

Cache-Control: no-cache

Cookie: JSESSIONID=818C14110CA7BFD1FC90610866A220E8

-----7de1e62806e0

Content-Disposition: form-data; name="description1"

xxxx

-----7de1e62806e0

Content-Disposition: form-data; name="description2"

zzzz

-----7de1e62806e0

Content-Disposition: form-data; name="file1"; filename="ip.txt"

Content-Type: text/plain

192

-----7de1e62806e0--

2) 在 Servlet 中将上传的文件保存在服务器的硬盘中

2、常用文件上传 API

1) JSP 独立开发年代 jsp-smartupload ---- JSP Model1

jspSmartUpload 是一个可免费使用的全功能的文件上传下载组件, 适于嵌入执行上传下载操作的 JSP 文件中。

2) JSP+Servlet 开发 web 应用 Apache commons-fileupload ---- JSP Model2

FileUpload 是 Apache commons 下面的一个子项目, 用来实现 Java 环境下面的文件上传功能, 与常见的 SmartUpload 齐名。

3) Servlet3.0 规范中提供对文件上传的支持

Apache commons-fileupload 使用

1) 去 <http://commons.apache.org/fileupload/> 下载 fileupload jar 包

同时下载 commons-fileupload 和 commons-io 两个包 ----- fileupload 依赖 io 包

2) 将 jar 包导入 web 工程 WEB-INF/lib 下

3) 编程实现

步骤一: 获得 DiskFileItemFactory 文件项工厂

步骤二: 通过工厂 获得文件上传请求核心解析类 ServletFileUpload

步骤三: 使用 ServletFileUpload 对 request 进行解析 ---- 获得很多个 FileItem

步骤四: 对每个 FileItem 进行操作 判断 FileItem 是不是普通字段 isFormField

代表普通字段 FileItem

getFieldName(); ---- 获得表单项 name 属性

getString(); ----- 获得表单项 value

代表文件上传 FileItem

getInputStream() --- 获得文件内容输入流

getName() ----- 获得上传文件名称

commons-fileupload 核心 API 分析

DiskFileItemFactory 磁盘文件项工厂类

public DiskFileItemFactory(int sizeThreshold, java.io.File repository)

构造工厂时, 指定内存缓冲区大小和临时文件存放位置

public DiskFileItemFactory()

public void setSizeThreshold(int sizeThreshold) --用来设定内存缓冲区的大小, 默认是 10k

public void setRepository(java.io.File repository)

设置临时文件存放位置, 默认 System.getProperty("java.io.tmpdir").

内存缓冲区: 上传文件时, 上传文件的内容优先保存在内存缓冲区中, 当上传文件大小超过缓冲区大小, 就会在服务器端产生临时文件

临时文件存放位置: 保存超过了内存缓冲区大小上传文件而产生临时文件

* 产生临时文件可以通过 FileItem 的 delete 方法删除

ServletFileUpload 文件上传核心类

boolean isMultipartContent(HttpServletRequest request) 判断上传表单是否为 multipart/form-data 类型

List parseRequest(HttpServletRequest request)

解析 request 对象, 并把表单中的每一个输入项包装成一个 fileitem 对象, 返回 List<FileItem>

setFileSizeMax(long fileSizeMax) 设置单个上传文件的最大值

setSizeMax(long sizeMax) 设置上传文件总量的最大值

设置单个文件上传大小 和 void setSizeMax(long sizeMax) 设置总文件上传大小

setHeaderEncoding(java.lang.String encoding) 设置编码格式, 解决上传文件名乱码问题

setProgressListener(ProgressListener pListener)

设置文件上传监听器 (用来监控文件上传进度), 实时监听文件上传状态

* 上传时间、剩余大小、速度、剩余时间

FileItem 表示文件上传表单中 每个数据部分

boolean isFormField() 判断 FileItem 是一个文件上传对象还是普通表单对象

判断该数据项是否为文件上传项, true 不是文件上传 false 是文件上传

```
if(fileItem.isFormField()){
    // 不是上传项

    java.lang.String getFieldName() 获得普通表单项 name 属性

    java.lang.String getString()/ java.lang.String getString(java.lang.String encoding)

    获得普通表单项 value 属性 传入编码集用来解决输入 value 乱码
}else{
    // 是上传项

    java.lang.String getName() 获得上传文件名 (注意 IE6 存在路径)

    java.io.InputStream getInputStream() 获得上传文件内容输入流

    // 上传文件

    void delete() 删除临时文件 (删除时, 必须要管理输入输出流)
}
```

注意事项: 因为文件上传表单采用编码方式 multipart/form-data 与传统 url 编码不同, 所有 getParameter 方法不能使用 setCharacterEncoding 无法解决输入项乱码问题

如果是一个普通字段项可以调用:

String getFieldName() 获得普通表单对象的 name 属性

String getString(String encoding) 获得普通表单对象的 value 属性, 可以用 encoding 进行编码设置

如果是一个文件上传项:

String getName() 获得上传文件的文件名 (有些浏览器会携带客户端路径)

InputStream getInputStream() 获得上传文件的输入流

delete() 在关闭 FileItem 输入流后, 删除临时文件

3、文件存放应该注意的问题:

- 1.upload 文件夹和 temp 文件夹都要放在 web-inf 目录下保护起来,防止上传入侵和访问其他用户上传资源的问题
- 2.文件名要拼接 uuid 保证唯一
- 3.文件要分目录存储保证同一目录下不要有过多的文件,分目录的算法有很多,介绍了一种根据 hash 值分目录算法

1) 上传文件后, 在服务器端保存位置

第一类存放位置: 直接存放 WebRoot 目录下 和 除 WEB-INF META-INF 的其它子目录下 例如: WebRoot/upload

* 客户端可以直接在浏览器上通过 url 访问位置 (资料无需通过权限控制, 而可以直接访问) --- 对上传资源安全性要求不高、或者资源需要用户直接可见

* 例如: 购物商城商品图片

第二类存放位置: 放入 WEB-INF 及其子目录 或者 不受 tomcat 服务器管理目录 例如: WebRoot/WEB-INF/upload 、

c:\ 、 d:\abc

* 客户端无法通过 URL 直接访问，必须由服务器内部程序才能读取（安全性较高，可以很容易添加权限控制）

* 例如：会员制在线视频

2) 上传文件在同一个目录重名问题

如果文件重名，后上传文件就会覆盖先上传文件

文件名 UUID

```
filename = UUID.randomUUID().toString() + "_" + filename;
```

3) 为了防止同一个目录下上传文件数量过多 ---- 必须采用目录分离算法

1.按照上传时间进行目录分离（周、月）

2. 按照上传用户进行目录分离 ----- 为每个用户建立单独目录

3. 按照固定数量进行目录分离 ----- 假设每个目录只能存放 3000 个文件，每当一个目录存满 3000 个文件后，创建一个新的目录

4. 按照唯一文件名的 hashCode 进行目录分离

```
public static String generateRandomDir(String uuidFileName) {  
    // 获得唯一文件名的 hashCode  
    int hashCode = uuidFileName.hashCode();  
    // 获得一级目录  
    int d1 = hashCode & 0xf;  
    // 获得二级目录  
    int d2 = (hashCode >>> 4) & 0xf;  
  
    return "/" + d2 + "/" + d1; // 共有 256 目录  
}
```

4)乱码问题

普通编写项 value 属性乱码 ----- fileItem.getString(编码集);

上传文件项 文件名乱码 ----- fileupload.setHeaderEncoding(编码集);

4、上传文件的进度监控

ServletFileUpload 类 提供 public void setProgressListener(ProgressListener pListener)

* 为文件上传程序绑定一个监听器对象，通过监听器可以监听文件上传全过程

* 和 AJAX 技术结合，编写文件上传进度条

设置监听器，文件上传程序会自动执行 监听器中 update 方法

```
public void update(long pBytesRead, long pContentLength, int pItems)
```

在方法中可以获得 文件总大小、已经上传大小和 上传第几个元素

能否根据上面三个参数计算：剩余大小、传输速度、已用时间、剩余时间

1) 已用时间 = 当前时间 - 开始时间

2) 速度 = 已经上传大小/已用时间

3) 剩余大小 = 总大小- 已经上传大小

4) 剩余时间 = 剩余大小/速度

5、文件下载

将服务器端文件下载到客户端

```
response.setHeader("Content-Disposition", "attachment;filename="+URLEncoder.encode(filename,"utf-8"));
response.setContentType(this.getServletContext().getMimeType(filename));//MIME 类型
```

6、常见文件下载有两种编写方式

1) 超链接直接指向下载资源

如果文件格式浏览器识别，将直接打开文件，显示在浏览器上， 如果文件格式浏览器不识别，将弹出下载窗口
对于浏览器识别格式的文件，通过另存为进行下载

客户端访问服务器静态资源文件时，静态资源文件是通过 缺省 Servlet 返回的，在 tomcat 配置文件 conf/web.xml 找到 --- org.apache.catalina.servlets.DefaultServlet

2) 编写服务器程序，读取服务器端文件，完成下载

必须设置两个头信息，来自 MIME 协议 Content-Type Content-Disposition

```
response.setContentType(getServletContext().getMimeType(filename));
response.setHeader("Content-Disposition", "attachment;filename=" + filename); //
```

以附件形式打开，不管格式浏览器是否识别

7、综合案例

1.综合案例：网盘

```
create database day15;
```

```
use day15;
```

```
create table netdisk(
    id int primary key auto_increment,
    uuidname varchar(255),
    realname varchar(255),
    savepath varchar(255),
    ip varchar(100),
    uploadtime timestamp,
    description varchar(255)
);
```

index.jsp --- 提供 上传 下载列表

upload.jsp --- 提供上传表单,允许用户选择文件进行上传

UploadServlet --- 保存上传的文件到服务器/在数据库中保存文件相关的信息

DownListServlet --- 查询数据库表找到所有可供下载的资源信息,存入 request 域后带到页面显示

downlist.jsp --- 遍历 request 中所有资源信息,提供下载连接

DownServlet --- 下载指定 id 的资源

2.下载案例

指定一个磁盘目录，通过树形结构遍历，遍历磁盘目录下及其子目录中文体，提供下载

* 遍历一个树形目录结构中所有文件

1) 广度非递归 遍历目录中所有文件

2) 使用 get 方式提交中文时

```
<a href="/day21/downloadList?path=D:\TTPmusic\何晟铭\何晟铭 - 爱的供养.mp3">何晟铭 - 爱的供养.mp3</a><br/>
```

问题: IE6 提交后, 服务器经过 get 乱码处理获得 乱码

原因: IE6 对中文直接进行 get 提交时, 进行 URL 编码 ---- 编码发现问题

解决: 手动对 get 提交中文进行编码 ---- URLEncoder

3) 如果下载文件是中文名, 设置 `response.setHeader("Content-Disposition", "attachment;filename=" + filename);` 出现附件名乱码

不同浏览器处理下载附件名乱码 处理方式不同, 例如 IE 使用 URL 编码、FF 使用 BASE64 编码

通过 USER-AGENT 请求头信息字段, 判断来访者浏览器类型

** 问题: 火狐浏览器 在使用 MimeUtility 进行 Base64 编码 时存在问题, 如果字符串中没有中文, 无法进行编码

解决: 采用手动 BASE64 编码

```
BASE64Encoder base64Encoder = new BASE64Encoder();
```

```
filename = "?utf-8?B?" + base64Encoder.encode(filename.getBytes("utf-8")) + "?=";
```

3.综合案例: 上传下载系统

需求:

1、系统提供一个文件上传功能, 在用户上传文件后, 文件保存在服务器端指定目录, 文件相关信息保存在数据库中

* 没上传一个文件, 数据库中有一条数据记录

2、系统提供一个文件下载功能, 将数据表中所有资源信息, 显示在页面上, 允许用户进行下载

创建数据库环境

```
create database day21;
```

```
create table resources(  
    id int primary key auto_increment,  
    uuidname varchar(100) unique not null,  
    realname varchar(40) not null,  
    savepath varchar(100) not null,  
    uploadtime timestamp,  
    description varchar(255)  
);
```

导入 jar 包、c3p0-config.xml、JDBCUtils 工具类

第四部分 JSP

一、概述

jsp 是 sun 提供动态 web 资源开发技术。为了解决在 Servlet 中拼写 html 内容 css、js 内容十分不方便的问题, sun 提供了这样一门技术。如果说 Servlet 是在 java 中嵌套 HTML, 则 jsp 就是在 HTML 中嵌套 java 代码,从而十分便于组织 html 页面。

jsp 页面在第一次被访问到时会被 jsp 翻译引擎翻译成一个 Servlet,从此对这个 jsp 页面的访问都是由这个 Servlet 执行后进行

输出。

二、JSP 语法

1、JSP 模版元素

jsp 页面中书写的 HTML 内容称作 JSP 的模版元素,在翻译过来的 Servlet 中直接被 `out.write()` 输出到浏览器页面
JSP 模版元素定义了网页的基本骨架,即定义了页面的结构和外观

2、JSP 表达式

`<%= java 表达式 %>` 在翻译过来的 Servlet 中,计算 java 表达式的值后,被 out 输出到浏览器上

举例: 当前时间:`<%= new java.util.Date() %>`

JSP 引擎在翻译脚本表达式时,会将程序数据转成字符串,然后在相应位置用 `out.print(...)` 将数据输给客户端

JSP 脚本表达式中的变量或表达式后面不能有分号 (;)

3、JSP 脚本片断

用于在 JSP 页面中编写多行 Java 代码

`<% 若干 java 语句 %>` 在翻译过来的 Servlet 中,直接被复制粘贴到了对应的位置执行.

在一个 JSP 页面中可以有多个脚本片断,在两个或多个脚本片断之间可以嵌入文本、HTML 标记和其他 JSP 元素

多个脚本片断中的代码可以相互访问,犹如将所有的代码放在一对 `<%%>` 之中的情况

单个脚本片断中的 Java 语句可以是不完整的,但是,多个脚本片断组合后的结果必须是完整的 Java 语句

注意: JSP 脚本片断中只能出现 java 代码,不能出现其它模板元素, JSP 引擎在翻译 JSP 页面中,会将 JSP 脚本片断中的 Java 代码将被原封不动地放到 Servlet 的 `_jspService` 方法中。

JSP 脚本片断中的 Java 代码必须严格遵循 Java 语法,例如,每执行语句后面必须用分号 (;) 结束。

4、JSP 声明

`<%! 若干 java 语句 %>` 在翻译过来的 Servlet 中会被放置到和 `Service` 方法同级的位置,变成了类的一个成员

JSP 页面中编写的所有代码,默认会翻译到 `servlet` 的 `service` 方法中,而 Jsp 声明中的 java 代码被翻译到 `_jspService` 方法的外面

JSP 声明可用于定义 JSP 页面转换成的 Servlet 程序的静态代码块、成员变量和方法

多个静态代码块、变量和函数可以定义在一个 JSP 声明中,也可以分别单独定义在多个 JSP 声明中

JSP 隐式对象的作用范围仅限于 Servlet 的 `_jspService` 方法,所以在 JSP 声明中不能使用这些隐式对象

5、JSP 注释

`<%-- 注释的内容 -->` 被 jsp 注释注释掉的内容,在 jsp 翻译引擎将 jsp 翻译成 Servlet 的过程中会被丢弃,在翻译过来的 Servlet 中没有这些信息

`<%//java 注释%>` java 注释被当作 jsp 脚本片段被翻译到了 Servlet 中,在 java 文件被翻译成 class 文件的时候 注释信息被丢弃

`<!-- HTML 注释 -->` html 注释被当作模版元素输出到了浏览器上,浏览器认识 html 注释不予显示

6、JSP 指令

JSP 指令 (directive) 是为 JSP 引擎而设计的,它们并不直接产生任何可见输出,而只是告诉引擎如何处理 JSP 页面中的其余部分。在 JSP 2.0 规范中共定义了三个指令: `page` 指令、`include` 指令、`taglib` 指令

JSP 指令的基本语法格式:

`<%@ 指令 属性名="值" %>`

举例: `<%@ page contentType="text/html;charset=gb2312"%>`

如果一个指令有多个属性,这多个属性可以写在一个指令中,也可以分开写。

例如:

`<%@ page contentType="text/html;charset=gb2312"%>`

`<%@ page import="java.util.Date"%>`

也可以写作:

```
<%@ page contentType="text/html;charset=gb2312" import="java.util.Date"%>
```

(1) `page` 指令用于定义 JSP 页面的各种属性, 无论 `page` 指令出现在 JSP 页面中的什么地方, 它作用的都是整个 JSP 页面, 为了保持程序的可读性和遵循良好的编程习惯, `page` 指令最好是放在整个 JSP 页面的起始位置

```
<%@ page
    [ language="java" ]
    [ extends="package.class" ]
    [ import="{package.class | package.*}, ..." ]
    [ session="true | false" ]
    [ buffer="none | 8kb | sizekb" ]
    [ autoFlush="true | false" ]
    [ isThreadSafe="true | false" ]
    [ errorPage="relative_url" ]
    [ isErrorPage="true | false" ]
    [ contentType="mimeType [ ;charset=characterSet ]" | "text/html ; charset=ISO-8859-1" ]
    [ pageEncoding="characterSet | ISO-8859-1" ]
    [ isELIgnored="true | false" ]
%>
```

JSP 引擎自动导入下面的包:

- ✓ `java.lang.*`
- ✓ `javax.servlet.*`
- ✓ `javax.servlet.jsp.*`
- ✓ `javax.servlet.http.*`

可以在一条 `page` 指令的 `import` 属性中引入多个类或包, 其中的每个包或类之间使用逗号分隔:

```
<%@ page import="java.util.Date,java.sql.*,java.io.*"%>
```

上面的语句也可以改写为使用多条 `page` 指令的 `import` 属性来分别引入各个包或类:

```
<%@ page import="java.util.Date"%>
<%@ page import="java.sql.*"%>
<%@ page import="java.io.*"%>
```

`errorPage` 属性的设置值必须使用相对路径, 如果以 “/” 开头, 表示相对于当前 WEB 应用程序的根目录 (注意不是站点根目录), 否则, 表示相对于当前页面。

可以在 `web.xml` 文件中使用 `<error-page>` 元素为整个 WEB 应用程序设置错误处理页面, 其中的 `<exception-type>` 子元素指定异常类的完全限定名, `<location>` 元素指定以 “/” 开头的错误处理页面的路径。

如果设置了某个 JSP 页面的 `errorPage` 属性, 那么在 `web.xml` 文件中设置的错误处理将不对该页面起作用。

JSP 引擎会根据 `page` 指令的 `contentType` 属性生成相应的调用 `ServletResponse.setContentType` 方法的语句。

`page` 指令的 `contentType` 属性还具有说明 JSP 源文件的字符编码的作用。

(2) `include` 指令用于引入其它 JSP 页面, 如果使用 `include` 指令引入了其它 JSP 页面, 那么 JSP 引擎将把这两个 JSP 翻译成一个 `servlet`。所以 `include` 指令引入通常也称之为静态引入。

语法:

```
<%@ include file="relativeURL"%>
```

其中的 `file` 属性用于指定被引入文件的路径。路径以 “/” 开头, 表示代表当前 web 应用。

(3) `Taglib` 指令用于在 JSP 页面中导入标签库

7、JSP 内置对象

在翻译过来的 `Servlet` 中 `Service` 方法自动帮我们前置定义的九个对象, 可以在 `jsp` 页面中直接使用。

每个 JSP 页面在第一次被访问时,WEB 容器都会把请求交给 JSP 引擎(即一个 Java 程序)去处理。JSP 引擎先将 JSP 翻译成一个 `_jspServlet`(实质上也是一个 `servlet`),然后按照 `servlet` 的调用方式进行调用。

由于 JSP 第一次访问时会翻译成 `servlet`,所以第一次访问通常会比较慢,但第二次访问,JSP 引擎如果发现 JSP 没有变化,就不再翻译,而是直接调用,所以程序的执行效率不会受到影响。

JSP 引擎在调用 JSP 对应的 `_jspServlet` 时,会传递或创建 9 个与 web 开发相关的对象供 `_jspServlet` 使用。JSP 技术的设计者为便于开发人员在编写 JSP 页面时获得这些 web 对象的引用,特意定义了 9 个相应的变量,开发人员在 JSP 页面中通过这些变量就可以快速获得这 9 大对象的引用。

`request`、`response`、`config`、`application`、`exception`、`Session`、`page`、`out`、`pageContext`

(1) `out` 隐式对象用于向客户端发送文本数据,相当于是 `response.getWriter` 得到 `PrintWriter`。

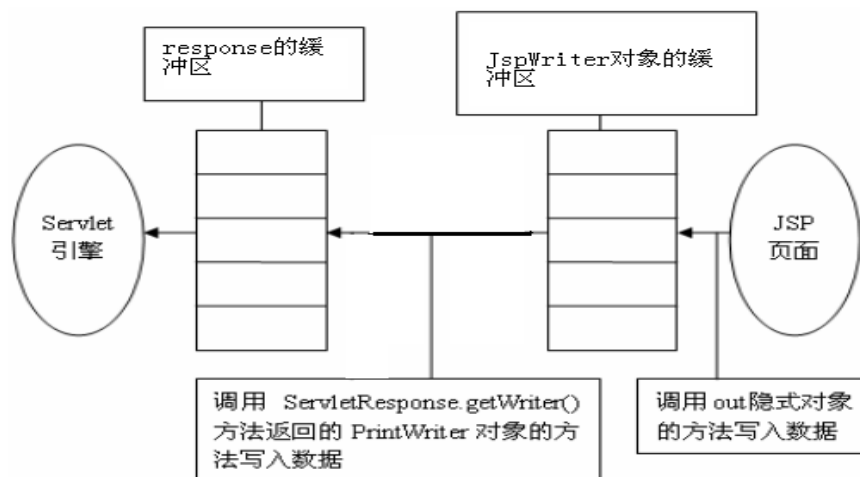
`out` 对象是通过调用 `pageContext` 对象的 `getOut` 方法返回的,其作用和用法与 `ServletResponse.getWriter` 方法返回的 `PrintWriter` 对象非常相似。

JSP 页面中的 `out` 隐式对象的类型为 `JspWriter`,`JspWriter` 相当于一种带缓存功能的 `PrintWriter`,设置 JSP 页面的 `page` 指令的 `buffer` 属性可以调整它的缓存大小,甚至关闭它的缓存。

只有向 `out` 对象中写入了内容,且满足如下任何一个条件时,`out` 对象才去调用 `ServletResponse.getWriter` 方法,并通过该方法返回的 `PrintWriter` 对象将 `out` 对象的缓冲区中的内容真正写入到 `Servlet` 引擎提供的缓冲池中。

`out` 和 `response.getWriter` 获取到的流不同在于,在于这个 `out` 对象本身就具有一个缓冲区。利用 `out` 写出的内容,会先缓冲在 `out` 缓冲区中,直到 `out` 缓冲区满了或者整个页面结束时 `out` 缓冲区中的内容才会被写出到 `response` 缓冲池中,最终可以带到浏览器页面进行展示

(2) 设置 `page` 指令的 `buffer` 属性关闭了 `out` 对象的缓存功能、`out` 对象的缓冲区已满、整个 JSP 页面结束



`[buffer="none | 8kb | sizekb"]` 可以用来禁用 `out` 缓冲区或设置 `out` 缓冲区的大小,默认 8kb

`[autoFlush="true | false"]` 用来设置当 `out` 缓冲区满了以后如果在写入数据时 `out` 如何处理,如果是 `true`,则先将 满了的数据写到 `response` 中后再接受新数据,如果是 `false`,则满了再写入数据直接抛异常

在 `jsp` 页面中需要进行数据输出时,不要自己获取 `response.getWriter`,而是要使用 `out` 进行输出,防止即用 `out` 又用 `response.getWriter` 而导致输出顺序错乱的问题

(3) `pageContext` 对象是 JSP 技术中最重要的一个对象,它代表 JSP 页面的运行环境,这个对象不仅封装了对其它 8 大隐式对象的引用,它自身还是一个域对象,可以用来保存数据。并且,这个对象还封装了 web 开发中经常涉及到的一些常用操作,例如引入和跳转其它资源、检索其它域对象中的属性等

1) 通过 `pageContext` 获得其他对象,可以作为入口对象获取其他八大隐式对象的引用

`getException` 方法返回 `exception` 隐式对象

`getPage` 方法返回 `page` 隐式对象

`getRequest` 方法返回 `request` 隐式对象

`getResponse` 方法返回 `response` 隐式对象
`getServletConfig` 方法返回 `config` 隐式对象
`getServletContext` 方法返回 `application` 隐式对象
`getSession` 方法返回 `session` 隐式对象
`getOut` 方法返回 `out` 隐式对象

2) `pageContext` 作为域对象, 四大作用域的入口, 可以操作四大作用域中的域属性

作用范围: 当前 `jsp` 页面

生命周期: 当对 `jsp` 页面的访问开始时, 创建代表当前 `jsp` 的 `PageContext`, 当对当前 `jsp` 页面访问结束时销毁代

表当前 `jsp` 的 `pageContext`

作用: 在当前 `jsp` 中共享数据

`pageContext` 对象的方法

```
public void setAttribute(java.lang.String name, java.lang.Object value)
public java.lang.Object getAttribute(java.lang.String name)
public void removeAttribute(java.lang.String name)
```

`pageContext` 对象中还封装了访问其它域的方法

```
public java.lang.Object getAttribute(java.lang.String name, int scope)
public void setAttribute(java.lang.String name, java.lang.Object value, int scope)
public void removeAttribute(java.lang.String name, int scope)
```

代表各个域的常量

```
PageContext.APPLICATION_SCOPE
PageContext.SESSION_SCOPE
PageContext.REQUEST_SCOPE
PageContext.PAGE_SCOPE
```

`findAttribute` 方法 (*重点, 查找各个域中的属性) EL 表达式

-- 搜寻四大作用域中的属性, 如果找到则返回该值, 如果四大作用域中都找不到 则返回一个 `null`, 搜寻的顺序是从最小的域开始向最大的域开始寻找

3) 提供了请求转发和请求包含的快捷方法

`PageContext` 类中定义了一个 `forward` 方法和两个 `include` 方法来分别简化和替代 `RequestDispatcher.forward` 方法和 `include` 方法。

```
pageContext.include("/index.jsp");
pageContext.forward("/index.jsp");
```

方法接收的资源如果以 “/” 开头, “/” 代表当前 `web` 应用。

8、JSP 标签

JSP 标签也称之为 Jsp Action(JSP 动作)元素, 它用于在 Jsp 页面中提供业务逻辑功能, 避免在 JSP 页面中直接编写 `java` 代码, 造成 `jsp` 页面难以维护

`sun` 原生提供的标签直接在 `jsp` 页面中就可以使用

`<jsp:include>` -- 实现页面包含, 动态包含

`<jsp:forward>` -- 实现请求转发

`<jsp:param>` -- 配合上面的两个标签使用, 在请求包含和请求转发时用来在路径后拼接一些请求参数

JSP 常用标签

`<jsp:include>` 标签

`<jsp:include>` 标签用于把另外一个资源的输出内容插入进当前 JSP 页面的输出内容之中, 这种在 JSP 页面执行时的引入方式称之为动态引入。

语法:

```
<jsp:include page="relativeURL" | <%=expression%>" flush="true|false" />
```

page 属性用于指定被引入资源的相对路径，它也可以通过执行一个表达式来获得。

flush 属性指定在插入其他资源的输出内容时，是否先将当前 JSP 页面的已输出的内容刷新到客户端。

<jsp:forward>标签

<jsp:forward>标签用于把请求转发给另外一个资源。

语法：

```
<jsp:forward page="relativeURL" | <%=expression%>" />
```

page 属性用于指定请求转发到的资源的相对路径，它也可以通过执行一个表达式来获得

<jsp:param>标签

当使用<jsp:include>和<jsp:forward>标签引入或将请求转发给其它资源时，可以使用<jsp:param>标签向 这个资源传递参数。

语法 1：

```
<jsp:include page="relativeURL" | <%=expression%>">
```

```
<jsp:param name="parameterName" value="parameterValue" | <%= expression %>" />
```

```
</jsp:include>
```

语法 2：

```
<jsp:forward page="relativeURL" | <%=expression%>">
```

```
<jsp:param name="parameterName" value="parameterValue" | <%= expression %>" />
```

```
</jsp:include>
```

<jsp:param>标签的 name 属性用于指定参数名，value 属性用于指定参数值。在<jsp:include>和<jsp:forward>标签中可以使用多个<jsp:param>标签来传递多个参数。

<jsp:include>与 include 指令的比较

<jsp:include>标签是动态引入，<jsp:include>标签涉及到的 2 个 JSP 页面会被翻译成 2 个 servlet，这 2 个 servlet 的内容在执行时进行合并。

而 include 指令是静态引入，涉及到的 2 个 JSP 页面会被翻译成一个 servlet，其内容是在源文件级别进行合并。

不管是<jsp:include>标签，还是 include 指令，它们都会把两个 JSP 页面内容合并输出，所以这两个页面 不要出现重复的 HTML 全局架构标签，否则输出给客户端的内容将会是一个格式混乱的 HTML 文档。

9、JSP 映射

```
<servlet>
    <servlet-name>index</servlet-name>
    <jsp-file>/index.jsp</jsp-file>
</servlet>
<servlet-mapping>
    <servlet-name>index</servlet-name>
    <url-pattern>/jsp/*</url-pattern>
</servlet-mapping>
```

三、EL 表达式

EL 全名为（Expression Language）最初出现的目的是用来取代 jsp 页面中的 jsp 脚本表达式，但是随着 el 的发展 el 的功能已经不限于此了（jsp2.0 之后-- JavaEE1.4 之后-- j2ee4.0 之后）

注意：有些 Tomcat 服务器如不能使用 EL 表达式

（1）升级成 tomcat6

(2) 在 JSP 中加入<%@ page isELIgnored="false" %>

EL 表达式获取数据语法: \${el 表达式}

EL 主要作用:

(1) 获取数据:

EL 表达式语句在执行时, 会调用 `pageContext.findAttribute` 方法, 用标识符为关键字, 分别从 `page`、`request`、`session`、`application` 四个域中查找相应的对象, 找到则返回相应对象, 找不到则返回"" (注意, 不是 `null`, 而是空字符串)

EL 表达式主要用于替换 JSP 页面中的脚本表达式, 以从各种类型的 `web` 域 中检索 `java` 对象、获取数据。(某个 `web` 域中的对象, 访问 `java`bean 的属性、访问 `list` 集合、访问 `map` 集合、访问数组)

使用中括号的地方都可以使用点号替代,除了中括号中是数字或者中括号中包含特殊字符(-)的情况除外

在中括号中如果不用双引号引起来则是变量,先找变量的值再拿变量的值使用.如果用双引号则是常量,直接使用常量的值

~获取常量

字符串/数字/布尔类型,直接写在 el 表达式中,el 直接进行输出

~获取域中的变量

如果 el 中写的是一个变量的名,则 el 会调用 `pageContext` 的 `findAttribute` 方法,在四大作用域中以给定的 名字找对应的属性值,找到后进行输出,如果四个域中都找不到,什么都不输出

~获取数组中的数据

~获取集合中的数据

~获取 Map 中的数据

~获取 `java`bean 的属性

EL 表达式也可以很轻松获取 `JavaBean` 的属性, 或获取数组、`Collection`、`Map` 类型集合的数据, 例如:

`${user.address.city}`

`${user.list[0]}`: 访问有序集合某个位置的元素

`${map.key}` : 获得 `map` 集合中指定 `key` 的值

. 和 [] 区别

结合 `JSTL` 的 `foreach` 标签, 使用 EL 表达式也可以很轻松迭代各种类型的数组或集合, 示例:

迭代数组

迭代 `collection` 类型集合

迭代 `map` 类型集合

(2) 执行运算:

利用 EL 表达式可以在 JSP 页面中执行一些基本的关系运算、逻辑运算和算术运算, 以在 JSP 页面中完成一些简单的逻辑运算。`${user==null}`

关系运算符	说 明	范 例	结 果
= 或 eq	等于	<code>\${ 5 == 5 }</code> 或 <code>\${ 5 eq 5 }</code>	true
!= 或 ne	不等于	<code>\${ 5 != 5 }</code> 或 <code>\${ 5 ne 5 }</code>	false
< 或 lt	小于	<code>\${ 3 < 5 }</code> 或 <code>\${ 3 lt 5 }</code>	true
> 或 gt	大于	<code>\${ 3 > 5 }</code> 或 <code>\${ 3 gt 5 }</code>	false
<= 或 le	小于等于	<code>\${ 3 <= 5 }</code> 或 <code>\${ 3 le 5 }</code>	true
>= 或 ge	大于等于	<code>\${ 3 >= 5 }</code> 或 <code>\${ 3 ge 5 }</code>	false

逻辑运算符	说 明	范 例	结 果
&& 或 and	交集	<code>\${ A && B }</code> 或 <code>\${ A and B }</code>	true / false
或 or	并集	<code>\${ A B }</code> 或 <code>\${ A or B }</code>	true / false
! 或 not	非	<code>\${ !A }</code> 或 <code>\${ not A }</code>	true / false

算数运算 +-* /

逻辑运算

比较运算

三元运算符: `${user!=null?user.name : ""}`

empty 运算符: 检查对象是否为 null 或 “空”

(3) 获取 web 开发常用对象

EL 表达式定义了一些隐式对象，利用这些隐式对象，web 开发人员可以很轻松获得对 web 常用对象的引用，从而获得这些数据。

EL 中内置了 11 个内置对象,这些对象 el 内置的,不需要提前定义就可以直接在 EL 中使用

!pageContext -- 有了它可以很方便的获取 jsp 页面中的 9 大隐式对象

!pageScope -- page 域中属性组成的 Map

!requestScope -- request 域中属性组成的 Map

!sessionScope -- session 域中属性组成的 Map

!applicationScope --application 域中属性组成的 Map

!param -- 所有请求参数组成的 Map<String,String>

paramValues -- 所有请求参数组成的 Map<String,String[]>

header -- 所有请求头组成的 Map<String,String>

headerValues -- 所有请求头组成的 Map<String,String[]>

!cookie -- 所有 cookie 信息组成的 Map<String,Cookie>

initParam -- 所有 web 应用的初始化参数组成 Map

(4) 调用 Java 方法

`${prefix: method(params)}`

需要大家自己会写调用方法的过程,只要会调用别人写好的标签库就可以了 fn 标签库

EL 表达式允许用户开发自定义 EL 函数，以在 JSP 页面中通过 EL 表达式调用 Java 类的方法。

(1)编写一个类，其中应该包含要使用 el 调用的静态方法

(2)编写一个 tld 文件，描述该方法的调用，在创建 tld 文件时应选用 2.0 版本的 jsp 配置，指定名称空间

uri 和缩写 prefix

(3)在 tld 文件中配置方法信息

<function>

<name>encodeURL</name>el 在调用时所使用的方法名

<function-class>cn.itheima.util.EncodeURL</function-class>静态方法所在的类全路径名

<function-signature>

java.lang.String EncodURL(java.lang.String)//对该方法的描述：返回值类型 方法

名(参数类型)

</function-signature>

</function>

(4)在 jsp 中使用<%@ taglib uri="" prefix="ppp"%>引入 tld 文件

(5)在 jsp 中使用`{ppp:encodeURIComponent("xxx")}`调用

四、JSTL 标签库

在 javaee4.0 需要导入 JSTL 相关的 jar 包, 在 javaee5.0 开始, 默认已经包含了此 jar 包。还要需要用`<%@ taglib%>`指令引入标签库

解压缩后将 lib 中的 jstl.jar、standard.jar 复制到 WEB 应用程序的 WEB-INF\lib 下

1、介绍

JavaServer Pages Standard Tag Library

由 JCP (Java Community Process) 指定标准

提供给 Java Web 开发人员一个标准通用的标签函数库和 EL 配合来取代传统直接在页面上嵌入 Java 程序 (Scripting) 的做法, 以提高程序可读性、维护性和方便性

一般我们使用 JSTL1.1 以上的版本, 应为从这个版本开始支持 EL 表达式

JSTL1.0 默认不支持 el 表达式, 不建议使用

2、JSTL 标签库

****核心标签库 (core) --- c

国际化标签 fmt

数据库标签 sql --Servlet

XML 标签 xml

JSTL 函数(EL 函数) el

3、JSTL 核心标签库

`<c:out>` 标签用于输出一段文本内容到 `pageContext` 对象当前保存的 “out” 对象中。

`<c:set>` 标签用于把某一个对象存在指定的域范围内, 或者设置 Web 域中的 `java.util.Map` 类型的属性对象或 `JavaBean` 类型的属性对象的属性。

`<c:remove>` 标签用于删除各种 Web 域中的属性

`<c:catch>` 标签用于捕获嵌套在标签体中的内容抛出的异常, 其语法格式如下: `<c:catch [var="varName"]>nested actions</c:catch>`

`<c:if test= " " >` 标签可以构造简单的 “if then” 结构的条件表达式

`<c:choose>` 标签用于指定多个条件选择的组合边界, 它必须与 `<c:when>` 和 `<c:otherwise>` 标签一起使用。使用 `<c:choose>`, `<c:when>` 和 `<c:otherwise>` 三个标签, 可以构造类似 “if else if else” 的复杂条件判断结构。

`<c:forEach>` 标签用于对一个集合对象中的元素进行循环迭代操作, 或者按指定的次数重复迭代执行标签体中的内容。

`<c:forTokens>` 用来浏览一字符串中所有的成员, 其成员是由定义符号所分隔的

`<c:param>` 标签 在 JSP 页面进行 URL 的相关操作时, 经常要在 URL 地址后面附加一些参数。`<c:param>` 标签可以嵌套在 `<c:import>`、`<c:url>` 或 `<c:redirect>` 标签内, 为这些标签所使用的 URL 地址附加参数。

`<c:import>` 标签, 实现 include 操作

`<c:url>` 标签用于在 JSP 页面中构造一个 URL 地址, 其主要目的是实现 URL 重写。URL 重写就是将会话标识号以参数形式附加在 URL 地址后面

`<c:redirect>` 标签用于实现请求重定向

4、自定义标签

传统标签的继承结构: JspTag

(1) 传统标签

Tag 接口----定义了一个标签处理类应具有的最基本的方法和属性(EVAL_BODY_INCLUDE dostart 方法返回表示执行标签体, SKIP_BODY dostart 方法用, 跳过标签体。 EVAL_PAGE

用在 doendtag 里通知后续页面继续执行, SKIP_PAGE doendtag 里通知后续页面不再执行)

|
|----IterationTag 接口 (提供了 doAfterBody() 在标签体执行过后立即执行, 并提供 EVAL_BODY_AGAIN 供 doafterbody 方法返回表示要重新执行标签体)

|
|----TagSupport 类(提供了对 pageContext 的引用)
|
|----BodyTag 接口 (EVAL_BODY_BUFFERED 在 doStartTag 方法中返回通知标签处理器去缓存标签体 bodyContent)

|
|----BodyTagSupport 类 (getBodyContent() 获取缓存对象 bodyContent)

(2) 简单标签 (简单标签的标签体不能包含脚本内容, 所以 tld 文件中配置 body-content 时不能配置为 JSP)

SimpleTag 接口

|
|----SimpleTagSupport 实现类 (getJspContext() 获取代表 jsp 页面的 pageContext, getJspBody() 代表标签体内内容的 JspFragment 对象)

JspFragment: invoke(Writer out) 此方法一经调用会将标签体输出到对应的流中。如果你直接给一个 null 默认输出到 pageContext.getOut()中

如果 doTag () 方法抛出 SkipPageException 异常, 标签后的 jsp 页面就不再执行。

(3) 自定义标签声明属性: 在标签处理类中提供 setXXX 方法, 并在 tld 文件中通过<attribute>标签声明属性的特性即可。

传统标签:

- (1)写一个类实现 Tag 接口
- (2)写一个 tld 文件,描述写好的类
- (3)在 jsp 页面中引入 tld 文件,就可以在 jsp 页面中使用自定义标签了

分为 doStartTag 和 doEndTag 方法来分别处理发现开始标签和发现结束标签时的代码,在 doStartTag 可以通过返回值来控制标签体是否允许执行,在 doEndTag 方法里可以通过返回值控制标签之后的剩余页面是否允许执行

传统标签的这种开发方式,需要我们分析发现开始标签和发现结束标签时都需要执行什么代码,还需要分析到底要返回什么样的标签体控制程序执行,相对来说相当的繁琐.

简单标签:

- (1)写一个类实现 SimpleTag 接口(继承 SimpleTag 接口的默认实现类 SimpleTagSupport)
- (2)写一个 tld 文件,描述写好的类

(3)在 jsp 页面中引入 tld 文件,就可以在 jsp 页面中使用自定义标签了

当 jsp 在执行的过程中,每当遇到一个简单标签时都会创建一个处理类对象.

调用 `setJspContext` 传入当前 jsp 页面的 `PageContext` 对象.

如果当前标签有父标签则调用 `setParent` 方法将父标签传入,如果没有父标签则这个方法不会被调用.

如果该标签具有属性,调用属性的 `setXXX` 方法将属性的值传入

如果当前标签具有标签体,则会调用 `setJspBody` 将封装了标签体信息的 `JspFragment` 传入,如果没有标签体,这个方法不执行

最后调用 `doTag` 方法,在这个方法里我们可以书写处理标签事件的 java 代码

当自定义标签执行完成后,简单标签对象就销毁掉了.

控制标签体是否执行<c:if>

控制标签之后的内容是否执行

控制标签体重复执行<c:forEach>

修改标签体后输出<c:out>

为自定义标签来增加一个属性:

在标签处理类中增加一个 `javaBean` 属性,这个属性就是要增加的标签的属性,并对外提供 `setXXX` 方法

在 tld 文件中这个标签的描述中描述一下该属性

*想要开发一个简单标签,写一个类继承 `SimpleTagSupport` 覆盖 `doTag` 方法就可以了,可以调用 `getJspContext/getJspBody` 来获取需要的内容

*在 tld 文件中对标签进行描述

```
<tag>
    <name>simpleDemo1</name> -- 标签的名字
    <tag-class>com.itheima.simletag.SimpleDemo1</tag-class> -- 标签的处理类
    <body-content>scriptless</body-content> -- 标签体的类型 JSP(简单标签不能写) Scriptless(任意的 jsp 内容,不包括 java 代码) empty(空标签) tagdependent(标签体是给后台用的,一般不用这种类型)
    <attribute> -- 声明一个属性,可以声明多个属性
        <name>times</name> -- 属性的名字
        <required>true</required> -- 是否为必须存在的属性
        <rtexprvalue>true</rtexprvalue> -- 是否支持 el 表达式
        <type>int</type> -- 属性的 java 类型
    </attribute>
</tag>
```

用户注册登录注销

Servlet+jsp+javaBean+dom4j(XPATH)

javaee 的经典三层架构

com.ithema.web

.service

.dao

.domain


```
.util
.test
.exception
.factory
*junit、dom4j、*jstl、beanutils
*debug 调试模式
users.xml -- 模拟数据库
config.properties -- 主配置文件
```

第五部分 SQL

一、数据库

1、创建数据库

```
create database [if not exists] db_name [character set xxx] [collate xxx]
*创建一个名称为 mydb1 的数据库。
create database mydb1;
*创建一个使用 utf8 字符集的 mydb2 数据库。
create database mydb2 character set utf8;
*创建一个使用 utf8 字符集，并带校对规则的 mydb3 数据库。
create database mydb3 character set utf8 collate utf8_bin ;
```

2、查看数据库

```
显示数据库语句：
SHOW DATABASES
显示数据库创建语句：（查看数据库的创建方式）
SHOW CREATE DATABASE db_name
~查看当前数据库服务器中的所有数据库
show databases;
~查看前面创建的 mydb2 数据库的定义信息
show create database mydb3;
```

3、修改数据库

```
alter database db_name [character set xxx] [collate xxx]
ALTER DATABASE [IF NOT EXISTS] db_name [alter_specification [, alter_specification] ...]
alter_specification:
[DEFAULT] CHARACTER SET charset_name | [DEFAULT] COLLATE collation_name
~查看服务器中的数据库，并把其中 mydb2 字符集修改为 utf8
alter database mydb2 character set utf8;
```

4、删除数据库

```
drop database [if exists] db_name;
DROP DATABASE [IF EXISTS] db_name
~删除前面创建的 mydb1 数据库 drop database mydb1;
drop database mydb1;
```

5、使用数据库

切换数据库 use db_name;

查看当前使用的数据库 select database();

二、表

1、创建表

```
create table tab_name(  
    field1 type,  
    field2 type,  
    ...  
    fieldn type  
)[character set xxx][collate xxx];
```

***java 和 mysql 的数据类型比较

String	-----	char(n) varchar(n) 255 65535
byte short int long float double	-----	tinyint smallint int bigint float double
boolean	-----	bit 0/1
Date	-----	Date、Time、DateTime、timestamp
FileInputStream FileReader	-----	Blob Text

*创建一个员工表 employee

```
create table employee(  
    id int primary key auto_increment ,  
    name varchar(20),  
    gender bit default 1,  
    birthday date,  
    entry_date date,  
    job varchar(20),  
    salary double,  
    resume text  
);
```

约束:

- primary key
- unique
- not null
- auto_increment 主键字段必须是数字类型。
- 外键约束

```
CREATE TABLE table_name  
(  
    field1 datatype,  
    field2 datatype,  
    field3 datatype,
```

) [character set 字符集] [collate 校对规则]

field: 指定列名 datatype: 指定列类型

~创建一个员工表 employee

```
create table employee(  
    id int primary key auto_increment,  
    name varchar(20) unique,  
    gender bit not null,  
    birthday date,  
    entry_date date,  
    job varchar(40),  
    salary double,  
    resume text  
);
```

2、查看表信息

desc tab_name 查看表结构

show tables 查看当前数据库中的所有的表

show create table tab_name 查看当前数据库表建表语句

3、修改表结构

ALTER TABLE table ADD/MODIFY/DROP/CHARACTER SET/CHANGE (column datatype [DEFAULT expr][, column datatype]...);

(1) 增加一列

```
alter table tab_name add [column] 列名 类型;
```

(2) 修改一列类型

```
alter table tab_name modify 列名 类型;
```

(3) 修改列名

```
alter table tab_name change [column] 列名 新列名 类型;
```

(4) 删除一列

```
alter table tab_name drop [column] 列名;
```

(5) 修改表名

```
rename table 表名 to 新表名;
```

(6) 修改表所用的字符集

```
alter table student character set utf8;
```

4、删除表

```
DROP TABLE tab_name;
```

~删除 user 表

```
drop table user;
```

三、表记录

1、增加一条记录

```
INSERT INTO table [(column [, column...])] VALUES (value [, value...]);
```

```
insert into tab_name (field1,field2,.....) values (value1,value2,.....);
```

*插入的数据应与字段的数据类型相同。

*数据的大小应在列的规定范围内，例如：不能将一个长度为 80 的字符串加入到长度为 40 的列中。

*在 values 中列出的数据位置必须与被加入的列的排列位置相对应。

*字符和日期型数据应包含在单引号中' zhang' ' 2013-04-20'

*插入空值：不指定某列的值或 insert into table value(null)，则该列取空值。

*如果要插入所有字段可以省写列列表，直接按表中字段顺序写值列表 insert into tab_name values(value1,value2,.....);

*练习：使用 insert 语句向表中插入三个员工的信息。

```
insert into emp (name,birthday,entry_date, job, salary) values ('张飞','1990-09-09','2000-01-01','打手',999);
```

```
insert into emp (name,birthday,entry_date, job, salary) values ('关羽','1989-08-08','2000-01-01','财神',9999);
```

```
insert into emp (name,birthday,entry_date, job, salary) values ('赵云','1991-07-07','2000-01-02','保安',888);
```

```
insert into emp values (7,'黄忠',1,'1970-05-05','2001-01-01','战神',1000,null);
```

2、修改表记录

```
UPDATE    tbl_name SET col_name1=expr1 [, col_name2=expr2 ...] [WHERE where_definition]
```

```
update tab_name set field1=value1,field2=value2,.....[where 语句]
```

*UPDATE 语法可以用新值更新原有表行中的各列。

*SET 子句指示要修改哪些列和要给予哪些值。

*WHERE 子句指定应更新哪些行。如没有 WHERE 子句，则更新所有的行。

*实验：

将所有员工薪水修改为 5000 元。

```
update emp set salary=5000;
```

将姓名为' zs' 的员工薪水修改为 3000 元。

```
update emp set salary=3000 where name='zs';
```

将姓名为' ls' 的员工薪水修改为 4000 元, job 改为 ccc。

```
update emp set salary=4000, job='ccc' where name='zs';
```

将 wu 的薪水在原有基础上增加 1000 元。

```
update emp set salar=salary+4000 where name='wu';
```

3、删除表操作

```
delete from tab_name [where ....]
```

*如果不跟 where 语句则删除整张表中的数据

*delete 只能用来删除一行记录，不能值删除一行记录中的某一列值（这是 update 操作）。

*delete 语句只能删除表中的内容，不能删除表本身，想要删除表，用 drop

*同 insert 和 update 一样，从一个表中删除记录将引起其它表的参照完整性问题，在修改数据库数据时，头脑中应该始终不要忘记这个潜在的问题。

*TRUNCATE TABLE 也可以删除表中的所有数据，词语句首先摧毁表，再新建表。此种方式删除的数据不能在事务中恢复。

*实验：

删除表中名称为'zs'的记录。

```
delete from emp where name='黄忠';
```

删除表中所有记录。

```
delete from emp;
```

使用 truncate 删除表中记录。

```
truncate table emp;
```

4、select 操作

(1) select [distinct] *|field1, field2, from tab_name 其中 from 指定从哪张表筛选, *表示查找所有列, 也可以指定一个列列表明确指定要查找的列, distinct 用来剔除重复行。

*实验:

~查询表中所有学生的信息。

```
select * from exam;
```

~查询表中所有学生的姓名和对应的英语成绩。

```
select name,english from exam;
```

~过滤表中重复数据

```
select distinct english from exam;
```

~在所有学生分数上加 10 分特长分显示。

```
select name , math+10,english+10,chinese+10 from exam;
```

~统计每个学生的总分。

```
select name ,english+math+chinese from exam;
```

(2) select 也可以使用表达式, 并且可以使用 as 别名

在所有学生分数上加 10 分特长分显示。

```
select name,english+10,chinese+10,math+10 from exam;
```

统计每个学生的总分。

```
select name,english+chinese+math from exam;
```

使用别名表示学生总分。

```
select name,english+chinese+math as 总成绩 from exam;
```

```
select name,english+chinese+math 总成绩 from exam;
```

```
select name english from exam;
```

(3) 使用 where 子句, 进行过滤查询。

*练习:

查询姓名为 XXX 的学生成绩

```
select * from exam where name='张飞';
```

查询英语成绩大于 90 分的同学

```
select name,english from exam where english>90;
```

查询总分大于 200 分的所有同学

```
select name,math+english+chinese as 总成绩 from exam where math+english+chinese>200 ;
```

*where 字句中可以使用:

*比较运算符:

> < >= <= <>

between 10 and 20 值在 10 到 20 之间

in(10, 20, 3) 值是 10 或 20 或 30

like '张 pattern' pattern 可以是%或者_, 如果是%则表示任意多字符, 此例中张三丰 张飞 张 abcd , 如果是_则表示一个字符张_, 张飞符合。张

Is null

*逻辑运算符

在多个条件直接可以使用逻辑运算符 and or not

*实验

~查询姓名为张飞的学生成绩

```
select * from exam where name='张飞';
```

~查询英语成绩大于 90 分的同学

```
select * from exam where english > 90;
```

~查询总分大于 230 分的所有同学

```
select name 姓名, math+english+chinese 总分 from exam where math+english+chinese>230;
```

~查询英语分数在 80—100 之间的同学。

```
select * from exam where english between 80 and 100;
```

~查询数学分数为 75, 76, 77 的同学。

```
select * from exam where math in(75, 76, 77);
```

~查询所有姓张的学生成绩。

```
select * from exam where name like '张%';
```

```
select * from exam where name like '张__';
```

~查询数学分>70, 语文分>80 的同学。

```
select * from exam where math>70 and chinese>80;
```

(4) Order by 指定排序的列, 排序的列即可是表中的列名, 也可以是 select 语句后指定的别名。

Asc 升序、Desc 降序, 其中 asc 为默认值

ORDER BY 子句应位于 SELECT 语句的结尾。

*练习:

对数学成绩排序后输出。

```
select * from exam order by math;
```

对总分排序按从高到低的顺序输出

```
select name, (ifnull(math, 0)+ifnull(chinese, 0)+ifnull(english, 0)) 总成绩 from exam  
order by 总成绩 desc;
```

对姓张的学生成绩排序输出

```
select name, (ifnull(math, 0)+ifnull(chinese, 0)+ifnull(english, 0)) 总成绩 from exam  
where name like '张%' order by 总成绩 desc;
```

(5) 聚合函数: 技巧, 先不要管聚合函数要干嘛, 先把要求的内容查出来再包上聚合函数即可。

count(列名) -- 用来统计符合条件的行的个数

统计一个班级共有多少学生? 先查出所有的学生, 再用 count 包上

```
select count(*) from exam;
```

统计数学成绩大于 70 的学生有多少个?

```
select count(math) from exam where math>70;
```

统计总分大于 250 的人数有多少?

```
select count(name) from exam where  
(ifnull(math, 0)+ifnull(chinese, 0)+ifnull(english, 0))>250;
```

sum(列名) -- 用来将符合条件的记录的指定列进行求和操作

统计一个班级数学总成绩？先查出所有的数学成绩，再用 sum 包上

```
select sum(math) from exam;
```

统计一个班级语文、英语、数学各科的总成绩

```
select sum(math),sum(english),sum(chinese) from exam;
```

统计一个班级语文、英语、数学的成绩总和

```
select sum(ifnull(math,0)+ifnull(chinese,0)+ifnull(english,0)) as 总成绩 from  
exam;
```

统计一个班级语文成绩平均分

```
select sum(chinese)/count(*) from exam ;
```

注意：sum 仅对数值起作用，否则会报错。

AVG(列名) -- 用来计算符合条件的记录的指定列的值的平均值

求一个班级数学平均分？先查出所有的数学分，然后用 avg 包上。

```
select avg(ifnull(math,0)) from exam;
```

求一个班级总分平均分

```
select avg((ifnull(math,0)+ifnull(chinese,0)+ifnull(english,0))) from exam ;
```

Max、Min -- 用来获取符合条件的所有记录指定列的最大值和最小值

求班级最高分和最低分（数值范围在统计中特别有用）

```
select Max((ifnull(math,0)+ifnull(chinese,0)+ifnull(english,0))) from exam;
```

```
select Min((ifnull(math,0)+ifnull(chinese,0)+ifnull(english,0))) from exam;
```

(6) group by 子句，其后可以接多个列名，也可以跟 having 子句对 group by 的结果进行筛选。

练习：对订单表中商品归类后，显示每一类商品的总价

```
select product, sum(price) from orders group by product;
```

练习：查询购买了几类商品，并且每类总价大于 100 的商品

```
select product, sum(price) from orders group by product having sum(price)>100;
```

!~having 和 where 的差别：

where 子句在分组之前进行过滤（筛选），having 子句在分组之后进行过滤

having 子句中可以使用聚合函数，where 子句中不能使用

很多情况下使用 where 子句的地方可以使用 having 子句进行替代

练习：查询商品列表中除了橘子以外的商品，每一类商品的总价格大于 500 元的商品的名字

```
select product,sum(price) from orders where product<>'桔子' group by  
product having sum(price)>500;
```

(7) 重点：Select from where group by having order by

~~sql 语句书写顺序：

```
select from where groupby having orderby
```

~~sql 语句执行顺序：

```
from where select group by having order by
```

*分析：

```
select math+english+chinese as 总成绩 from exam where 总成绩 >250; ---- 不成功
```

```
select math+english+chinese as 总成绩 from exam having 总成绩 >250; --- 成功
```

```
select math+english+chinese as 总成绩 from exam group by 总成绩 having 总成绩 >250;
```

----成功

```
select math+english+chinese as 总成绩 from exam order by 总成绩;----成功
select * from exam as 成绩 where 成绩.math>85; ---- 成功
```

~~备份恢复数据库

备份: 在 cmd 窗口下 mysqldump -u root -p dbName>c:/1.sql

恢复: 方式 1: 在 cmd 窗口下 mysql -u root -p dbName<c:/1.sql

方式 2: 在 mysql 命令下, source c:/1.sql

要注意恢复数据只能恢复数据本身, 数据库没法恢复, 需要先自己创建出数据后才能进行恢复.

四、约束

1、创建表时指定约束

```
create table tb(
    id int primary key auto_increment,
    name varchar(20) unique not null,
    ref_id int,
    foreign key(ref_id) references tb2(id)
);

create table tb2(
    id int primary key auto_increment
);
```

2、外键约束

表是用来保存显示生活中的数据的, 而现实生活中数据和数据之间往往具有一定的关系, 我们在使用表来存储数据时, 可以明确的声明表和表之前的依赖关系, 命令数据库来帮我们维护这种关系, 向这种约束就叫做外键约束

(1) 增加外键:

可以明确指定外键的名称, 如果不指定外键的名称, mysql 会自动为你创建一个外键名称。

RESTRICT : 只要本表里面有指向主表的数据, 在主表里面就无法删除相关记录。

CASCADE : 如果在 foreign key 所指向的那个表里面删除一条记录, 那么在此表里面的跟那个 key 一样的所有记录都会一同删掉。

```
alter table book add [constraint FK_BOOK] foreign key(pubid) references pub_com(id) [on
delete restrict] [on update restrict];
```

(2) 删除外键

```
alter table 表名 drop foreign key 外键 (区分大小写, 外键名可以 desc 表名查看);
```

3、主键约束

(1) 增加主键 (自动增长, 只有主键可以自动增长)

```
Alter table tb add primary key(id) [auto_increment];
```

(2) 删除主键

```
alter table 表名 drop primary key
```


(3) 增加自动增长

```
Alter table employee modify id int auto_increment;
```

(4) 删除自动增长

```
Alter table tb modify id int;
```

五、多表设计

一对一（311 教室和 20130405 班级，两方都是一）：在任意一方保存另一方的主键

一对多、多对一（班级和学生，其中班级为 1，学生为多）：在多的-方保存另一方的主键

多对多（教师和学生，两方都是多）：使用中间表，保存对应关系

六、多表查询

1、笛卡尔积查询

将两张表的记录进行一个相乘的操作查询出来的结果就是笛卡尔积查询,如果左表有 n 条记录,右表有 m 条记录,笛卡尔积查询出有 $n*m$ 条记录,其中往往包含了很多错误的的数据,所以这种查询方式并不常用

```
select * from ta,tb;
```

```
mysql> select * from ta ,tb;
```

```
+---+-----+-----+---+---+
| id | name | tb_id | id | name |
+---+-----+-----+---+---+
| 1 | aaa | 1 | 1 | xxx |
| 2 | bbb | 2 | 1 | xxx |
| 3 | bbb | 4 | 1 | xxx |
| 1 | aaa | 1 | 2 | yyy |
| 2 | bbb | 2 | 2 | yyy |
| 3 | bbb | 4 | 2 | yyy |
| 1 | aaa | 1 | 3 | yyy |
| 2 | bbb | 2 | 3 | yyy |
| 3 | bbb | 4 | 3 | yyy |
+---+-----+-----+---+---+
```

2、内连接查询

查询的是左边表和右边表都能找到对应记录的记录，查询两张表中都有的关联数据,相当于利用条件从笛卡尔积结果中筛选出了正确的结果。

```
select * from ta,tb where ta.tb_id = tb.id;
```

```
select * from ta inner join tb on ta.tb_id = tb.id;
```

```
mysql> select * from ta inner join tb on ta.tb_id = tb.id;
```

```
+---+-----+-----+---+---+
| id | name | tb_id | id | name |
+---+-----+-----+---+---+
| 1 | aaa | 1 | 1 | xxx |
| 2 | bbb | 2 | 2 | yyy |
+---+-----+-----+---+---+
```

```
+---+-----+-----+---+-----+
```

3、外连接查询

(1) 左外连接查询:在内连接的基础上增加左边表有而右边表没有的记录

```
select * from ta left join tb on ta.tb_id = tb.id;
```

```
mysql> select * from ta left join tb on ta.tb_id = tb.id;
```

```
+---+-----+-----+---+-----+
| id | name | tb_id | id | name |
+---+-----+-----+---+-----+
| 1 | aaa | 1 | 1 | xxx |
| 2 | bbb | 2 | 2 | yyy |
| 3 | bbb | 4 | NULL | NULL |
+---+-----+-----+---+-----+
```

右外连接查询:在内连接的基础上增加右边表有而左边表没有的记录

```
select * from ta right join tb on ta.tb_id = tb.id;
```

```
mysql> select * from ta right join tb on ta.tb_id = tb.id;
```

```
+-----+-----+-----+---+-----+
| id | name | tb_id | id | name |
+-----+-----+-----+---+-----+
| 1 | aaa | 1 | 1 | xxx |
| 2 | bbb | 2 | 2 | yyy |
| NULL | NULL | NULL | 3 | yyy |
+-----+-----+-----+---+-----+
```

全外连接查询:在内连接的基础上增加左边表有而右边表没有的记录和右边表有而左表没有的记录

```
select * from dept full join emp on dept.id=emp.dept_id; -- mysql 不支持全外连接
```

可以使用 union 关键字模拟全外连接:

```
select * from dept left join emp on dept.id = emp.dept_id
```

```
union
```

```
select * from dept right join emp on dept.id = emp.dept_id;
```

```
create table tb (id int primary key,name varchar(20) );
```

```
create table ta (
```

```
id int primary key,
```

```
name varchar(20),
```

```
tb_id int
```

```
);
```

```
insert into tb values(1,'财务部');
```

```
insert into tb values(2,'人事部');
```

```
insert into tb values(3,'科技部');
```

```
insert into ta values (1,'刘备',1);
```

```
insert into ta values (2,'关羽',2);
insert into ta values (3,'张飞',3);
```

```
mysql> select * from ta;
+----+-----+-----+
| id | name | tb_id |
+----+-----+-----+
| 1 | aaa  | 1     |
| 2 | bbb  | 2     |
| 3 | bbb  | 4     |
+----+-----+-----+

mysql> select * from tb;
+----+-----+
| id | name |
+----+-----+
| 1 | xxx  |
| 2 | yyy  |
| 3 | yyy  |
+----+-----+
```

第六部分 JDBC

一、数据库驱动的概念、JDBC

数据库厂商为了方便开发人员从程序中操作数据库而提供的一套 jar 包,通过导入这个 jar 包就可以调用其中的方法操作数据库,这样的 jar 包就叫做数据库驱动,本质上是很多的接口。

sun 定义的一套标准,本质上是一大堆的操作数据库的接口,所有数据库厂商为 java 设计的数据库驱动都实现过这套接口,这样一来同一了不同数据库驱动的方法,开发人员只需要学习 JDBC 就会使用任意数据库驱动了

二、JDBC 快速入门

*在数据库中建立好表

*在程序导入数据库驱动包

1、注册数据库驱动

```
DriverManager.registerDriver(new Driver());
```

缺点一：观察 `mysqlDriver` 源码发现此方法导致了数据库驱动被注册了两次。

缺点二：整个程序域 `mysql` 数据库驱动绑定增加了耦合性

```
Class.forName("com.mysql.jdbc.Driver");
```

2、获取连接

```
DriverManager.getConnection(url, user, password);
```

~url 的写法:

Oracle 写法: `jdbc:oracle:thin:@localhost:1521:sid`

SqlServer—`jdbc:microsoft:sqlserver://localhost:1433; DatabaseName=sid`

```
MySql—jdbc:mysql://localhost:3306/sid
~url 可以接的参数
user、password
useUnicode=true&characterEncoding=UTF-8
```

3、获取传输器

createStatement(): 创建向数据库发送 sql 的 statement 对象。
prepareStatement(sql) : 创建向数据库发送预编译 sql 的 PreparedStatement 对象。

4、利用传输器执行 sql 语句获取结果集

executeQuery(String sql) : 用于向数据库发送查询语句。
executeUpdate(String sql): 用于向数据库发送 insert、update 或 delete 语句
execute(String sql): 用于向数据库发送任意 sql 语句

5、遍历结果集取出结构

ResultSet 以表的样式在内存中保存了查询结果，其中还维护了一个游标，最开始的时候游标在第一行之前，每调用一次 next()方法就试图下移一行，如果移动成功返回 true;

ResultSet 还提供了很多个 Get 方法，用来获取查询结果中的不同类型的数据

除了 next 方法，还有以下方法可以用来遍历结果集:

```
next(): 移动到下一行
Previous(): 移动到前一行
absolute(int row): 移动到指定行
beforeFirst(): 移动 resultSet 的最前面。
afterLast() : 移动到 resultSet 的最后面。
```

6、释放资源

conn 是一个有限的资源，用完立即要释放表

stat 占用内存，所以使用完后也要释放

rs 占用内存，所以使用完后也要释放

释放时后创建的先释放

```
if(rs != null){
    try {
        rs.close();
    } catch (SQLException e) {
        e.printStackTrace();
    } finally{
        rs = null;
    }
}
if(stat != null){
    try {
        stat.close();
    } catch (SQLException e) {
        e.printStackTrace();
    } finally{
        stat = null;
    }
}
```

```
    }  
    if(conn != null){  
        try {  
            conn.close();  
        } catch (SQLException e) {  
            e.printStackTrace();  
        } finally{  
            conn = null;  
        }  
    }  
}
```

7、六个步骤实现 JDBC:

```
//1.注册数据库驱动  
DriverManager.registerDriver(new Driver());  
  
//2.获取数据库连接  
Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/day10", "root", "root");  
  
//3.获取传输器对象  
Statement stat = conn.createStatement();  
  
//4.利用传输器传输 sql 语句到数据库中执行,获取结果集对象  
ResultSet rs = stat.executeQuery("select * from user");  
  
//5.遍历结果集获取查询结果  
while(rs.next()){  
    String name = rs.getString("name");  
    System.out.println(name);  
}  
  
//6.关闭资源  
rs.close();  
stat.close();  
conn.close();
```

三、PreparedStatement

1. Sql 注入: 由于 jdbc 程序在执行的过程中, dao 中执行的 SQL 语句是拼接出来的,其中有一部分内容是由用户从客户端传入,所以当用户传入的数据中包含 sql 关键字时,就有可能通过这些关键字改变 sql 语句的语义,从而执行一些特殊的操作,这样的攻击方式就叫做 sql 注入攻击。

2、PreparedStatement 利用预编译的机制将 sql 语句的主干和参数分别传输给数据库服务器,从而使数据库分辨的出哪些是 sql 语句的主干哪些是参数,这样一来即使参数中带了 sql 的关键字,数据库服务器也仅仅将他当作参数值使用,关键字不会起作用,从而从原理上防止了 sql 注入的问题

3、PreparedStatement 是 Statement 的孩子,不同的是,PreparedStatement 使用预编译机制,在创建 PreparedStatement 对象时就需要将 sql 语句传入,传入的过程中参数要用?替代,这个过程会导致传入的 sql 被进行预编译,然后再调用 PreparedStatement 的 setXXX 将参数设置上去,由于 sql 语句已经经过了预编译,再传入特殊值也不会起作用了。

4、PreparedStatement 主要有如下的三个优点:

- ~1.可以防止 sql 注入
- ~2.由于使用了预编译机制,执行的效率要高于 Statement
- ~3.sql 语句使用?形式替代参数,然后再用方法设置?的值,比起拼接字符串,代码更加优雅.

四、大数据

1、mysql

mysql 数据库也可以直在数据库中保存大文本和大二进制数据，

Text

TINYTEXT(255)、TEXT(64k)、MEDIUMTEXT(16M)和 LONGTEXT(4G)

Blob

TINYBLOB、BLOB、MEDIUMBLOB 和 LONGBLOB

2、JDBC 去操作大文本

~插入大文本：

```
ps = conn.prepareStatement("insert into Demo2Text values(null,?,?)");
```

```
ps.setString(1, "钢铁是怎样练成");
```

```
File file = new File("1.txt");
```

```
ps.setCharacterStream(2, new FileReader(file), (int) file.length());
```

```
//1.Exception in thread "main" java.lang.AbstractMethodError:
```

```
com.mysql.jdbc.PreparedStatement.setCharacterStream(ILjava/io/Reader;J)V
```

```
//ps.setCharacterStream(2, new FileReader(file), file.length());第三个参数是 long 型的是从 1.6 才开始支持的，驱动里还没有开始支持。
```

```
//解决方案：ps.setCharacterStream(2, new FileReader(file), (int)file.length());
```

```
//2.Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
```

```
//文件大小过大，导致 PreparedStatement 中数据多大占用内存，内存溢出
```

```
//-Xms256M-Xmx256M
```

```
//3.com.mysql.jdbc.PacketTooBigException: Packet for query is too large (10886466 > 1048576). You can change this value on the server by setting the max_allowed_packet' variable.
```

```
//数据库连接传输用的包不够大，传输大文本时报此错误
```

```
//在 my.ini 中配置 max_allowed_packet 指定包的大小
```

~查询大文本：

```
Reader rd = rs.getCharacterStream("content");
```

3、JDBC 操作大二进制

~插入：

```
ps = conn.prepareStatement("insert into Demo3Blob values(null,?,?)");
```

```
ps.setString(1, "梦想的力量");
```

```
File file = new File("1.mp3");
```

```
ps.setBinaryStream(2, new FileInputStream(file), (int) file.length());
```

~查询

```
InputStream in = rs.getBinaryStream("content");
```

五、事务

1、事务的概念

事务是指逻辑上的一组操作,这组操作要么同时完成要么同时不完成.

2、事务的管理

默认情况下,数据库会自动管理事务,管理的方式是一条语句就独占一个事务.

如果需要自己控制事务也可以通过如下命令开启/提交/回滚事务

```
start transaction;  
  
commit;  
  
rollback;
```

JDBC 中管理事务:

```
conn.setAutoCommit(false);  
  
conn.commit();  
  
conn.rollback();  
  
SavePoint sp = conn.setSavePoint();  
  
conn.rollback(sp);
```

3、事务的四大特性

一个事务具有的最基本的特性,一个设计良好的数据库可以帮我们保证事务具有这四大特性(ACID)

原子性:原子性是指事务是一个不可分割的工作单位,事务中的操作要么都发生,要么都不发生。

一致性:如果事务执行之前数据库是一个完整性的状态,那么事务结束后,无论事务是否执行成功,数据库仍然是一个完整性状态。

数据库的完整性状态:当一个数据库中的所有数据都符合数据库中所定义的所有的约束,此时可以称数据库是一个完整性状态。

隔离性:事务的隔离性是指多个用户并发访问数据库时,一个用户的事务不能被其它用户的事务所干扰,多个并发事务之间数据要相互隔离。

持久性:持久性是指一个事务一旦被提交,它对数据库中数据的改变就是永久性的,接下来即使数据库发生故障也不应该对其有任何影响。

4、隔离性

将数据库设计成单线程的数据库,可以防止所有的线程安全问题,自然就保证了隔离性.但是如果数据库设计成这样,那么效率就会极其低下。

5、数据库中的锁机制

共享锁:在非 `Serializable` 隔离级别做查询不加任何锁,而在 `Serializable` 隔离级别下做的查询加共享锁,

共享锁的特点:共享锁和共享锁可以共存,但是共享锁和排他锁不能共存

排他锁:在所有隔离级别下进行增删改的操作都会加排他锁,

排他锁的特点:和任意其他锁都不能共存

如果是两个线程并发修改,一定会互相捣乱,这时必须利用锁机制防止多个线程的并发修改

如果两个线程并发查询,没有线程安全问题

如果两个线程一个修改,一个查询.....

在非串行化下,所有的查询都不加锁,所有的修改操作都会加排他锁。

在串行化下,所有的查询都加共享锁,所有的修改都加排他锁。

死锁

6、四大隔离级别

`Read uncommitted` -- 不防止任何隔离性问题,具有脏读/不可重复度/虚读(幻读)问题

`Read committed` -- 可以防止脏读问题,但是不能防止不可重复度/虚读(幻读)问题

`Repeatable read` -- 可以防止脏读/不可重复读问题,但是不能防止虚读(幻读)问题

`Serializable` -- 数据库被设计为单线程数据库,可以防止上述所有问题

从安全性上考虑: `Serializable`>`Repeatable read`>`read committed`>`read uncommitted`

从效率上考虑: `read uncommitted`>`read committed`>`Repeatable read`>`Serializable`

通常来说,一般的应用都会选择 Repeatable read 或 Read committed 作为数据库隔离级别来使用

真正使用数据的时候,根据自己使用数据库的需求,综合分析对安全性和对效率的要求,选择一个隔离级别使数据库运行在这个隔离级别上.

mysql 默认下就是 Repeatable read 隔离级别

oracle 默认下就是 read committed 个隔离级别

查询当前数据库的隔离级别:select @@tx_isolation;

设置隔离级别:set [global/session] transaction isolation level xxxx;其中如果不写默认是 session 指的是修改当前客户端和数据库交互时的隔离级别,而如果使用 global,则修改的是数据库的默认隔离级别

如何查询当前数据库的隔离级别? select @@tx_isolation;

如何设置当前数据库的隔离级别? set [global/session] transaction isolation level ...;

~此种方式设置的隔离级别只对当前连接起作用。

set transaction isolation level read uncommitted;

set session transaction isolation level read uncommitted;

~此种方式设置的隔离级别是设置数据库默认的隔离级别

set global transaction isolation level read uncommitted;

7、脏读

一个事务读取到另一个事务未提交的数据

a 1000

b 1000

a:

start transaction;

update account set money=money-100 where name=a;

update account set money=money+100 where name=b;

b:

start transaction;

select * from account;

a: 900

b: 1100

a:

rollback;

b:

start transaction;

select* from account;

a: 1000

b: 1000

8、不可重复读

在一个事务内读取表中的某一行数据，多次读取结果不同 --- 行级别的问题

a: 1000 1000 1000

b: 银行职员

b:start transaction;

select 活期存款 from account where name='a'; --- 活期存款:1000

select 定期存款 from account where name='a'; --- 定期存款:1000

select 固定资产 from account where name='a'; --- 固定资产:1000

a:

start transaction;

update account set 活期=活期-1000 where name='a';

commit;

select 活期+定期+固定 from account where name='a'; --- 总资产:2000

commit;

9、虚读(幻读)

是指在一个事务内读取到了别的事务插入的数据，导致前后读取不一致 --- 表级别的问题

a: 1000

b: 1000

d: 银行业务人员

d:

start transaction;

select sum(money) from account; --- 2000 元

select count(name) from account; --- 2 个

c:

start transaction;

insert into account values(c,4000);

commit;

select sum(money)/count(name) from account; --- 平均:2000 元/个

commit;

10、更新丢失问题:

如果多个线程操作，基于同一个查询结构对表中的记录进行修改，那么后修改的记录将会覆盖前面修改的记录，前面的修改就丢失掉了，这就叫做更新丢失。

`Serializable` 可以防止更新丢失问题的发生。其他的三个隔离级别都有可能发生更新丢失问题。

`Serializable` 虽然可以防止更新丢失,但是效率太低,通常数据库不会用这个隔离级别,所以我们需要其他的机制来防止更新丢失:

悲观锁:悲观锁悲观的认为每一次操作都会造成更新丢失问题,在每次查询时就加上排他锁

乐观锁:乐观锁会乐观的认为每次查询都不会造成更新丢失.利用一个版本字段进行控制

乐观锁和悲观锁不是数据库中真正存在的锁,只是人们在解决更新丢失时的不同的解决方案,体现的是人们看待事务的态度。

悲观锁:

隔离级别不设置为 `Serializable`,防止效率过低。

在查询时手动加上排他锁。

如果数据库中的数据查询比较多而更新比较少的话,悲观锁将会导致效率低下。

乐观锁:

在表中增加一个 `version` 字段,在更新数据库记录是将 `version` 加一,从而在修改数据时通过检查版本号是否改变判断出当前更新基于的查询是否已经是过时的版本。

如果数据库中数据的修改比较多,更新失败的次数会比较多,程序需要多次重复执行更新操作。

如何使用:

查询非常多,修改非常少,使用乐观锁

修改非常多,查询非常少,使用悲观锁

六、数据库连接池

1、手写连接池

改造 `conn` 的 `close` 方法

继承

装饰

动态代理

2、开源数据源

DBCP:

方式 1:

```
BasicDataSource source = new BasicDataSource();
source.setDriverClassName("com.mysql.jdbc.Driver");
source.setUrl("jdbc:mysql:///day11");
source.setUsername("root");
source.setPassword("root");
```

方式 2:

```
Properties prop = new Properties();
prop.load(new FileReader("dbcp.properties"));
BasicDataSourceFactory factory = new BasicDataSourceFactory();
DataSource source = factory.createDataSource(prop);
```

配置文件中:

```
driverClassName=com.mysql.jdbc.Driver
url=jdbc:mysql:///day11
```

```
username=root

password=root

#<!-- 初始化连接 -->
initialSize=10

#最大连接数量
maxActive=50

#<!-- 最大空闲连接 -->
maxIdle=20

#<!-- 最小空闲连接 -->
minIdle=5

#<!-- 超时等待时间以毫秒为单位 6000 毫秒/1000 等于 60 秒 -->
maxWait=60000
```

3、C3P0 数据源

方式 1:

```
ComboPooledDataSource source = new ComboPooledDataSource();
source.setDriverClass("com.mysql.jdbc.Driver");
source.setJdbcUrl("jdbc:mysql:///day11");
source.setUser("root");
source.setPassword("root");
```

方式 2:

```
ComboPooledDataSource dataSource = new ComboPooledDataSource();
ComboPooledDataSource dataSource = new ComboPooledDataSource("mySorce");
在类加载目录下名称为 c3p0-config.xml 的配置文件中配置:
```

```
<?xml version="1.0"?>
<c3p0-config>
    <default-config>
        <property name="driverClass">com.mysql.jdbc.Driver</property >
        <property name="jdbcUrl">jdbc:mysql:///Day12</property >
        <property name="user">root</property>
        <property name="password">root</property>
    </default-config>
    <!-- This app is massive! -->
    <named-config name="mySorce">
        <property name="driverClass">com.mysql.jdbc.Driver</property >
        <property name="jdbcUrl">jdbc:mysql:///Day12</property >
        <property name="user">root</property>
        <property name="password">root</property>
    </named-config>
```

</c3p0-config>

driverClass、jdbcUrl、user、password

acquireIncrement: 当连接池中已经没有连接时, 连接池自动获取连接时一次获取的连接个数。

initialPoolSize: 连接池初始化时, 获取连接的个数。

maxPoolSize: 连接池可以保有的最大的连接的数量。

maxIdleTime: 当连接空闲多久时释放连接。如果该时间值设置问为 0, 表示从不释放连接。

minPoolSize: 连接池应该保有的最小的连接的数量。

4、tomcat 内置的数据源(DBCP)

1) 如何为 tomcat 配置数据源 (五个位置可以配置)

~tomcat/conf/context.xml 文件中配置<Context>配置在这个位置的信息将会被所有的 web 应用所共享

~tomcat/conf[engin]/[Host]/context.xml 文件中可以配置<Context>标签,这里配置的信息将会被这台虚拟主机中的所有 web 应用所共享

~tomcat/conf/server.xml 文件中的<Host>标签中配置<Context>标签,这是 web 应用的第一种配置方式,在这个标签中配置的信息将只对当前 web 应用起作用

~tomcat/conf[engin]/[Host]/自己创建一个.xml 文件,在这个文件中使用<Context>标签配置一个 web 应用,这是 web 应用第二种配置方式,在这个<Context>标签中配置的信息将只会对当前 web 应用起作用

~web 应用还有第三种配置方式:将 web 应用直接放置到虚拟主机管理的目录,此时可以在 web 应用的 META-INF 文件夹下创建一个 context.xml 文件,在其中可以写<Context>标签进行配置,这种配置信息将只会对当前 web 应用起作用

```
<Resource name="mySource" ---在数据源创建好以后绑定到 jndi 容器中时使用的名字
auth="Container"
type="javax.sql.DataSource" ---当前对象的类型, 默认就是数据源
username="root" --- 数据库用户名
password="root" --- 数据库密码
driverClassName="com.mysql.jdbc.Driver" ---数据库驱动名
url="jdbc:mysql:///day12" --- 数据库连接信息
maxActive="8" --- 最大连接数
maxIdle="4"/> --- 最大空闲连接数
```

2) 如果在程序中获取这个数据源

想要访问 jndi 就必须在 Servlet 中才能执行下列代码:

```
Context initCtx = new InitialContext();
```

```
Context jndi = (Context) initCtx.lookup("java:comp/env");
```

```
DataSource source = jndi.lookup("mySource");
```

必须掌握的内容:

事务的概念

事务的四大特性

事务的 4 个隔离级别是什么以及如何配置

在不同的隔离级别下都分别有什么问题

更新丢失

dbcp 怎么用

c3p0 怎么用

tomcat 数据源怎么用 (JNDI 知道是什么, 在这里会用即可, 不是重点)

七、DbUtils 工具类

1、介绍

DbUtils 提供如关闭连接、装载 JDBC 驱动程序等常规工作的工具类，里面的所有方法都是静态的。

2、QueryRunner -- 两行代码搞定增删改查

该类简化了 SQL 查询，它与 ResultSetHandler 组合在一起使用可以完成大部分的数据库操作，能够大大减少编码量。

(1)QueryRunner() --需要控制事务时,使用这组方法

```
int update(Connection conn, String sql)
```

Execute an SQL INSERT, UPDATE, or DELETE query without replacement parameters.

```
int update(Connection conn, String sql, Object... params)
```

Execute an SQL INSERT, UPDATE, or DELETE query.

```
int update(Connection conn, String sql, Object param)
```

Execute an SQL INSERT, UPDATE, or DELETE query with a single replacement parameter.

```
<T> T query(Connection conn, String sql, ResultSetHandler<T> rsh)
```

Execute an SQL SELECT query without any replacement parameters.

```
<T> T query(Connection conn, String sql, ResultSetHandler<T> rsh, Object... params)
```

Execute an SQL SELECT query with replacement parameters.

(2)QueryRunner(dataSource ds) --不需要控制事务用这组方法

```
int update(String sql)
```

Executes the given INSERT, UPDATE, or DELETE SQL statement without any replacement parameters.

```
int update(String sql, Object... params)
```

Executes the given INSERT, UPDATE, or DELETE SQL statement.

```
int update(String sql, Object param)
```

Executes the given INSERT, UPDATE, or DELETE SQL statement with a single replacement parameter.

```
<T> T query(String sql, ResultSetHandler<T> rsh)
```

Executes the given SELECT SQL without any replacement parameters.

```
<T> T query(String sql, ResultSetHandler<T> rsh, Object... params)
```

Executes the given SELECT SQL query and returns a result object.

3、ResultSetHandler 实现类

ArrayHandler: 把结果集中的第一行数据转成对象数组。

ArrayListHandler: 把结果集中的每一行数据都转成一个对象数组，再存放到 List 中。

!!!!BeanHandler: 将结果集中的第一行数据封装到一个对应的 JavaBean 实例中。

!!!!BeanListHandler: 将结果集中的每一行数据都封装到一个对应的 JavaBean 实例中，存放到 List 里。

MapHandler: 将结果集中的第一行数据封装到一个 Map 里，key 是列名，value 就是对应的值。

MapListHandler: 将结果集中的每一行数据都封装到一个 Map 里，然后再存放到 List

ColumnListHandler: 将结果集中某一列的数据存放到 List 中。

KeyedHandler(name): 将结果集中的每一行数据都封装到一个 Map 里(List<Map>), 再把这些 map 再存到一个 map 里，其 key 为指定的列。

!!!!ScalarHandler: 获取结果集中第一行数据指定列的值,常用来进行单值查询

Dos:
Regedit/services.msc