day14 【XML和Dom4j、正则表达式】

今日内容

- XML
  - **定义XML----组成成员--------->必须掌握**
  - 解析XML----Dom4j,XPath
- **正则表达式--------->必须掌握**
  - 可以书写简单的正则表达式
  - 可以看懂正则表达式

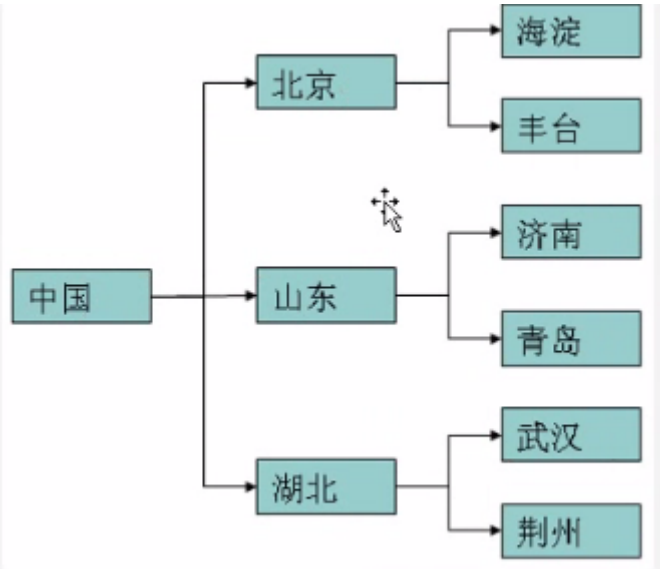
第一章 XML

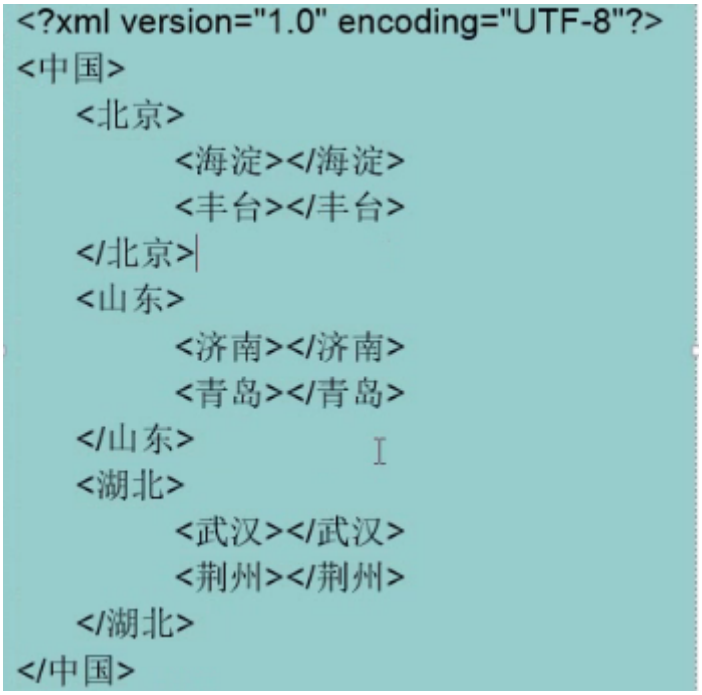
1.1 XML介绍

1.1 什么是XML

- XML 指可扩展标记语言（**EXtensible Markup Language**）
- XML是用来存储数据, 传输数据的，不是用来显示数据的。之后学习另外一个HTML是用来显示数据的。
- XML 标签没有被预定义。您需要自行定义标签。
- XML 是 W3C 的推荐标准

  W3C在1988年2月发布1.0版本，2004年2月又发布1.1版本，但因为1.1版本不能向下兼容1.0版本，所以1.1没有人用。同时，在2004年2月W3C又发布了1.0版本的第三版。我们要**学习的还是1.0版本。**





1.2 XML 与 HTML 的主要差异

- html语法松散,**xml语法严格,区分大小写**
- html做页面展示,xml传输数据,存储数据
- html所有标签都是预定义的,**xml所有标签都是自定义的**

1.3 xml的作用

- ==**作为配置文件。**== javaee框架 ssm大部分都会使用xml作为配置文件
- XML可以存储数据 , 作为数据交换的载体(使用XML格式进行数据的传输)。

1.2 XML组成元素

一个标准XML文件一般由以下几部分组成:**文档声明、元素、属性、注释、转义字符、字符区。**

文档声明

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--注释   快捷键: ctrl+/ -->
<!--
    文档声明:
        1.文档声明可以有也可以没有
        2.文档声明是以<?xml开头,以?>结尾
        3.如果有文档声明,必须放在第1行第1列
        4.文档声明中有2个常见属性:
            version:表示当前xml的版本,必须属性,一般写1.0
            encoding:表示当前xml的编码,可选属性,一般写utf-8,默认是utf-8
-->
<books>

</books>
```

注释

```
<!--注释内容-->
```

- XML的注释，既以`<!--`开始，`-->`结束。
- 注释不能嵌套
- idea上快捷键：`ctrl + /`

元素\标签

```
<!--
    元素\标签:
        1.元素是xml的重要组成部分,也叫做标签
        2.标签可以分为开始标签和结束标签,开始标签:<标签名>,结束标签: </标签名>
        3.开始标签和结束标签之间的内容叫做标签内容,标签内容可以是文本,也可以是其他标签
        4.标签是可以嵌套的,但不能乱嵌套
        5.开始标签和结束标签之间也可以没有内容,叫做空标签,一般定义为: <标签名/>
        6.一个xml文件只能有一个根标签,而且必须有一个根标签
        7.标签名一定要遵守命名规则和规范,但不要以xml,Xml,XML...这种命名
-->
<books>
    <book>
        <name>斗罗大陆</name>
        <author>唐家三少</author>
        <price>99.8</price>
    </book>

    <!--空标签-->
    <book></book>
    <book/>

</books>
```

属性

```
<!--
    属性:
        1.属性是标签的重要组成部分,必须写在开始标签,不能定义在结束标签
        2.属性的格式: 属性名=属性值,注意属性值必须用引号引起来(单引号,双引号)
        3.一个标签中可以定义0-N个属性,但属性名不能相同
        4.属性名必须遵守命名规则和规范(不能以数字开头,不能以美元符号开头,不能以特殊字符开头)
-->
<book name="斗破苍穹" author="天蚕土豆" price="99.8"></book>
```

转义字符

因为有些特殊的字符在XML中是不会被识别的，所以在元素体或属性值中想使用这些符号就必须使用转义字符（也叫实体字符），例如：">"、"<"、"'"、"""、"&"。

&lt;	<	小于
&gt;	>	大于
&amp;	&	和号
&apos;	'	省略号
&quot;	"	引号

注意：严格地讲，**在 XML 中仅有字符 "<"和"&" 是非法的。**省略号、引号和大于号是合法的，但是把它们替换为实体引用是个好的习惯。

转义字符应用示例：

```
<!--转义字符-->
<price>
    价格 > 1000 &amp;&amp; 价格 &lt;  2000
</price>
```

字符区(了解)

- CDATA 内部的所有东西都会被解析器忽略,当做文本

```
 <!-- 字符区,快捷键: CD -->
<![CDATA[
            价格 > 1000 &amp;&amp; 价格 &lt;  2000
            价格 > 1000 && 价格 < 2000
]]>
```

综合案例

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--注释   快捷键: ctrl+/ 注释不能嵌套-->
<!--
    文档声明:
        1.文档声明可以有也可以没有
        2.文档声明是以<?xml开头,以?>结尾
        3.如果有文档声明,必须放在第1行第1列
        4.文档声明中有2个常见属性:
            version:表示当前xml的版本,必须属性,一般写1.0
            encoding:表示当前xml的编码,可选属性,一般写utf-8,默认是utf-8
-->
<!--
    元素\标签:
        1.元素是xml的重要组成部分,也叫做标签
        2.标签可以分为开始标签和结束标签,开始标签:<标签名>,结束标签: </标签名>
        3.开始标签和结束标签之间的内容叫做标签内容,标签内容可以是文本,也可以是其他标签
        4.标签是可以嵌套的,但不能乱嵌套
        5.开始标签和结束标签之间也可以没有内容,叫做空标签,一般定义为: <标签名/>
        6.一个xml文件只能有一个根标签,而且必须有一个根标签
        7.标签名一定要遵守命名规则和规范,但不要以xml,Xml,XML...这种命名
-->
```

```
<!--
    属性:
        1.属性是标签的重要组成部分,必须写在开始标签,不能定义在结束标签
        2.属性的格式: 属性名=属性值,注意属性值必须用引号引起来(单引号,双引号)
        3.一个标签中可以定义0-N个属性,但属性名不能相同
        4.属性名必须遵守命名规则和规范(不能以数字开头,不能以美元符号开头,不能以特殊字符开头)
-->
<!--
    转义字符:
        概述:有些特殊的字符在XML中是不会被识别的，所以在元素体或属性值中想使用这些符号就必须使用转义字符
        注意：严格地讲，在 XML 中仅有字符 "<"和"&" 是非法的。省略号、引号和大于号是合法的，
        转义字符:
            <       &lt;
            &       &amp;
-->
<!--
    字符区: CDATA 内部的所有东西都会被解析器忽略,当做文本
    快捷键: CD
-->
<books>
    <book>
        <name>斗罗大陆</name>
        <author>唐家三少</author>
        <price>99.8</price>
    </book>

    <!--空标签-->
    <book name="斗破苍穹" author="天蚕土豆" price="99.8"></book>
    <book/>

    <!--转义字符-->
    <price>
        价格 > 1000 &amp;&amp; 价格 &lt;  2000
    </price>

    <!--字符区-->
    <![CDATA[
            价格 > 1000 &amp;&amp; 价格 &lt;  2000
            价格 > 1000 && 价格 < 2000
    ]]>
</books>
```

1.3 XML文件的约束-DTD约束(了解)

xml约束概述

- 在XML技术里，可以编写一个文档来约束一个XML文档的书写规范，这称之为XML约束。
- 两种约束：DTD约束(文件后缀为dtd)，Schema约束(文件后缀为xsd)
- 约束文档定义了在XML中允许出现的元素(标签)名称、属性及元素(标签)出现的顺序等等。
- **注意：约束不是我们要写的东西，我们的工作是根据约束去写XML**

根据DTD约束写XML

- DTD约束文档

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--
    引入方式:
    1.方式一: 内部DTD，在XML文档内部嵌入DTD，只对当前XML有效。
        <?xml version="1.0" encoding="UTF-8"?>
        <!DOCTYPE 根元素 [元素声明]>
    2.方式二: 外部DTD-本地DTD，DTD文档在本地系统上，企业内部自己项目使用。
        <!DOCTYPE 根元素 SYSTEM "文件名">
    3.方式三:外部DTD-公共DTD，DTD文档在网络上，一般都有框架提供 ， 也是我们使用最多的.
        <!DOCTYPE 根元素 PUBLIC "DTD名称" "DTD文档的URL">
-->
<!--
    标签层级关系的约束:  <!ELEMENT 父标签 (子标签的约束...)>
    数量词:
          *     表示元素可以出现0到多个          大于等于0
          +     表示元素可以出现至少1个          大于等于1
          ?     表示元素可以是0或1个
          ,     表示元素需要按照顺序显示
          |     表示元素需要选择其中的某一个
-->
<!--
    标签类型的约束: <!ELEMENT 标签名 标签类型>
    标签类型: (#PCDATA):文本类型 EMPTY:空标签    ANY: 任意类型

-->
<!--
    对属性的约束:
        <!ATTLIST 标签名
             属性名 属性类型 [属性约束]
             属性名 属性类型 [属性约束]
             ...
             属性名 属性类型 [属性约束]
             >
   属性类型:
        - CDATA :表示文本字符串
        - ID:表示属性值唯一,不能以数字开头
        - 枚举类型(枚举值|枚举值|...): 使用的使用只能从枚举列表中任选其一

   属性约束:
        - REQUIRED: 表示该属性必须出现
        - IMPLIED: 表示该属性可有可无
        - FIXED:表示属性的取值为一个固定值。语法: #FIXED "固定值"

-->
<!--根标签是书架,根标签下至少有1个书标签-->
<!ELEMENT 书架 (书*)>
<!--书标签下有书名,作者,售价子标签,并且这些子标签必须按照顺序出现: 书名,作者,售价-->
<!ELEMENT 书 (书名,作者,售价)>
<!--书名,作者,售价这三个标签的类型都是文本类型-->
<!ELEMENT 书名 (#PCDATA)>
<!ELEMENT 作者 (#PCDATA)>
<!ELEMENT 售价 (#PCDATA)>
```

```
<!--书标签中有id,编号,出版社,type属性-->
<!--id属性的类型是ID类型(唯一,不能以数字开头),必须出现-->
<!--编号属性的类型是文本字符串类型,可有可无-->
<!--出版社属性的类型是枚举类型,默认值是传智播客-->
<!--type属性的类型是文本字符串类型,固定值为IT-->
<!ATTLIST 书
        id ID #REQUIRED
        编号 CDATA #IMPLIED
        出版社 (清华|北大|传智播客) "传智播客"
        type CDATA #FIXED "IT"
        >
```

- XML

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

<!DOCTYPE 书架 SYSTEM "bookdtd.dtd">
<书架>
    <书 id="heima001" 编号="001" 出版社="清华" type="IT">
        <书名>斗罗大陆</书名>
        <作者>唐家三少</作者>
        <售价>99.8</售价>
    </书>
    <书 id="heima002" 编号="001" 出版社="清华" type="IT">
        <书名>斗罗大陆</书名>
        <作者>唐家三少</作者>
        <售价>99.8</售价>
    </书>
</书架>
```

```
<?xml version = "1.0" encoding="GB2312" standalone="yes"?>
<!DOCTYPE 购物篮 [
        <!ELEMENT 购物篮 (肉+)>
        <!ELEMENT 肉 EMPTY>
        <!ATTLIST 肉 品种 ( 鸡肉 | 牛肉 | 猪肉 | 鱼肉 ) "鸡肉">
        ]>
<购物篮>
    <肉 品种="猪肉"></肉>
    <肉></肉>
    <肉></肉>
    <肉/>
</购物篮>
```

语法(了解)

引入约束到xml文件中

1. 内部DTD，在XML文档内部嵌入DTD，只对当前XML有效。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE 根元素 [元素声明]>><!--内部DTD-->
```

2. 外部DTD—本地DTD，DTD文档在本地系统上，企业内部自己项目使用。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE 根元素 SYSTEM "文件名"><!--外部本地DTD-->
```

3. 外部DTD—公共DTD，DTD文档在网络上，一般都有框架提供，也是我们使用最多的.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE 根元素 PUBLIC "DTD名称" "DTD文档的URL">

例如: <!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
```

元素声明(了解)

1. 约束元素的嵌套层级

   语法

```
<!ELEMENT 父标签 （子标签…）>
例如:
<!ELEMENT books (book+)> <!--约束根元素是"books"，"books"子元素为"book"，“+”为数量词-->
<!ELEMENT book (name,author,price)><!--约束"book"子元素依次为“name”、“author”、“price”，-->
```

2. 约束元素体里面的数据

   语法

```
<!ELEMENT 标签名字 标签类型>
例如 <!ELEMENT name (#PCDATA)>
```

   标签类型: EMPTY(即空元素，例如<hr/>) ANY(任意类型) (#PCDATA) 字符串数据

   代码

```
<!ELEMENT name (#PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT price (#PCDATA)>
```

3. **数量词(掌握)**

数量词符号	含义
*	表示元素可以出现0到多个
+	表示元素可以出现至少1个
?	表示元素可以是0或1个
,	表示元素需要按照顺序显示
|	表示元素需要选择其中的某一个

属性声明(了解)

语法

```
<!ATTLIST 标签名称
        属性名称1 属性类型1 属性说明1
        属性名称2 属性类型2 属性说明2
        …
>
例如
<!ATTLIST book bid ID #REQUIRED>
```

属性类型

- CDATA :表示文本字符串
- ID:表示属性值唯一,不能以数字开头
- ENUMERATED (DTD没有此关键字)：表示枚举，只能从枚举列表中任选其一，如(鸡肉|牛肉|猪肉|鱼肉)

属性说明:

- REQUIRED：表示该属性必须出现
- IMPLIED：表示该属性可有可无
- FIXED:表示属性的取值为一个固定值。语法：#FIXED "固定值"

属性说明

代码

```
<!ATTLIST 书                                    <!--设置"书"元素的的属性列表-->
        id ID #REQUIRED                         <!--"id"属性值为必须有-->
        编号 CDATA #IMPLIED                      <!--"编号"属性可有可无-->
        出版社 (清华|北大|传智播客) "传智播客"    <!--"出版社"属性值是枚举值，默认为“传智播客”-->
        type CDATA #FIXED "IT"                  <!--"type"属性为文本字符串并且固定值为"IT"-->
>
```

案例

```
<?xml version = "1.0" encoding="GB2312" standalone="yes"?>
<!DOCTYPE 购物篮 [
        <!ELEMENT 购物篮 (肉+)>
        <!ELEMENT 肉 EMPTY>
        <!ATTLIST 肉 品种 ( 鸡肉 | 牛肉 | 猪肉 | 鱼肉 ) "鸡肉">
        ]>
<购物篮>
    <肉 品种="牛肉"></肉>
    <肉/>
</购物篮>
```

1.4 schema约束(了解)

概念

schema和DTD一样, 也是一种XML文件的约束.

Schema 语言也可作为 XSD（XML Schema Definition）。

Schema约束的文件的后缀名.xsd

Schema 功能更强大，数据类型约束更完善。

根据schema约束写出xml文档

- Schema约束文档:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--    传智播客教学实例文档.将注释中的以下内容复制到要编写的xml的声明下面
复制内容如下到XML文件中:
<书架 xmlns="http://www.itcast.cn"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.itcast.cn bookSchema.xsd" >
-->
<xs:schema
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.itcast.cn"
        elementFormDefault="qualified">

    <!--element:元素\标签-->
    <!--根标签的名称为书架-->
    <xs:element name='书架'>

        <!--complexType: 复杂类型-->
        <!--书架标签是一个复杂标签(标签下面有子标签)-->
        <xs:complexType>
            <!--sequence:顺序-->
            <!--子标签需要按照顺序出现-->
            <!--maxOccurs:最多出现多少次-->
            <!--minOccurs:最少出现多少次-->
            <!--unbounded:没有边界,无限次数-->
            <!--书架的子标签最少出现1次,最多出现2次-->
            <xs:sequence maxOccurs='2' minOccurs="1">
                <!--书架的子标签名为书-->
                <xs:element name='书'>
                    <!--书标签是一个复杂类型的标签(标签下面有子标签)-->
                    <xs:complexType>
                        <!--书标签的子标签需要按照顺序出现-->
                        <xs:sequence>
                           <!--顺序: 书名,作者,售价-->
                            <!--书名标签的类型是string-->
                            <!--作者标签的类型是string-->
                            <!--售价标签的类型是double-->
                            <xs:element name='书名' type='xs:string'/>
                            <xs:element name='作者' type='xs:string'/>
                            <xs:element name='售价' type='xs:double'/>
                        </xs:sequence>
                        <!--attribute:属性-->
                        <!--optional:可选-->
                        <!--required:可选-->
                        <!--书标签的属性名为bid,属性的类型为int,属性的约束为可选-->
```

```
                        <xs:attribute name="bid" type="xs:int"
use="required"/>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

- 根据上面的Schema约束编写XML

```
方式一:不取别名
<?xml version="1.0" encoding="UTF-8" ?>
<!--引入约束-->
<书架 xmlns="http://www.itcast.cn"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.itcast.cn bookSchema.xsd" >
    <书 bid="1">
        <书名>斗罗大陆</书名>
        <作者>唐家三少</作者>
        <售价>99.8</售价>
    </书>
</书架>

方式二: 取别名
<?xml version="1.0" encoding="UTF-8" ?>
<!--引入约束-->
<a:书架 xmlns:a="http://www.itcast.cn"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.itcast.cn bookSchema.xsd" >
    <a:书 bid="1">
        <a:书名>斗罗大陆</a:书名>
        <a:作者>唐家三少</a:作者>
        <a:售价>99.8</a:售价>
    </a:书>
</a:书架>
```

第二章 Dom4j

2.1 XML解析

解析方式

- 开发中比较常见的解析方式有三种，如下：
    1. DOM：要求解析器把整个XML文档装载到内存，并解析成一个Document对象
        a）优点：元素与元素之间保留结构关系，故可以进行增删改查操作。
        b）缺点：XML文档过大，可能出现内存溢出
    2. SAX：是一种速度更快，更有效的方法。她逐行扫描文档，一边扫描一边解析。并以事件驱动的方式进行具体解析，每执行一行，都触发对应的事件。（了解）
        a）优点：不会出现内存问题，可以处理大文件
        b）缺点：只能读，不能回写。

3. PULL：Android内置的XML解析方式，类似SAX。（了解）

- 解析器，就是根据不同的解析方式提供具体实现。有的解析器操作过于繁琐，为了方便开发人员，有提供易于操作的解析开发包

解析包

- JAXP：sun公司提供支持DOM和SAX开发包
- **Dom4j：比较简单的的解析开发包(常用)**
- JDom：与Dom4j类似
- Jsoup：功能强大DOM方式的XML解析开发包，尤其对HTML解析更加方便(项目中讲解)

2.2 Dom4j的基本使用 重点掌握

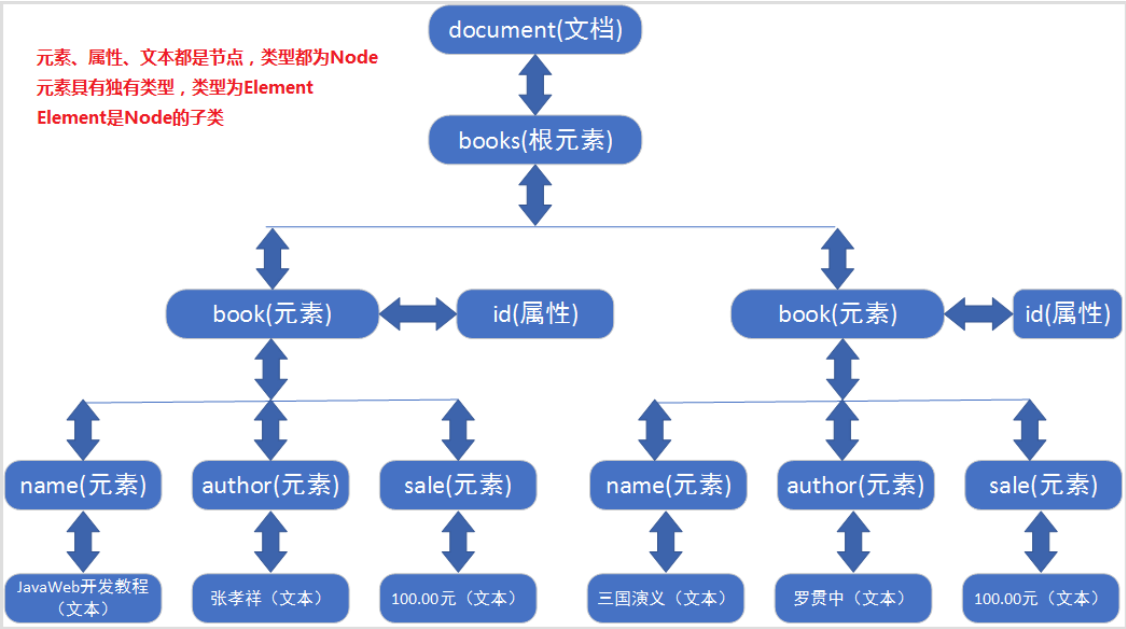
2.2.1 DOM解析原理及结构模型

- 解析原理

  XML DOM 和 HTML DOM一样，**XML DOM 将整个XML文档加载到内存，生成一个DOM树，并获得一个Document对象，通过Document对象就可以对DOM进行操作。**以下面books.xml文档为例。

```
<?xml version="1.0" encoding="UTF-8"?>
<books>
    <book id="0001">
        <name>JavaWeb开发教程</name>
        <author>张孝祥</author>
        <sale>100.00元</sale>
    </book>
    <book id="0002">
        <name>三国演义</name>
        <author>罗贯中</author>
        <sale>100.00元</sale>
    </book>
</books>
```

- 结构模型



DOM中的核心概念就是节点，在XML文档中的元素、属性、文本，在DOM中都是节点！所有的节点都封装到了Document对象中。

2.2.2 使用步骤

1. 导入jar包 dom4j-1.6.1j.jar
2. 创建解析器对象
3. 使用解析器对象读取xml文件,生成Document对象
4. 根据Document对象获得根元素
5. 根据根元素获取对于的子元素或者属性
6. .....

2.2.3 常用的方法

```
1.创建解析器对象: SAXReader sr = new SAXReader();
2.读取xml文件进行解析,生成Document对象---使用SAXReader方法
    Document read(String fileName);
3.使用Document对象获取根元素
    Element getRootElement();
4.使用元素获取子元素--->使用Element方法
    public List elements()                    : 获取当前元素的所有子元素
    public String getName()                   : 获取元素的元素名
    public String getText()                   : 获取当前元素的文本值
    public String attributeValue(String name)    : 获取当前元素下某个属性的值,传入属性名
    public Element element(String name)       : 根据元素名获取指定子元素(如果有多个就获取到第一个)
    public String elementText(String name)    : 获取指定子元素的文本值,参数是子元素名称
```

2.2.4 方法演示

- xml

```xml
<?xml version="1.0" encoding="utf-8" ?>
<books>
    <book id="0001">
        <name>JavaWeb开发教程</name>
        <author>张孝祥</author>
        <sale>100.00元</sale>
    </book>
    <book id="0002">
        <name>三国演义</name>
        <author>罗贯中</author>
        <sale>100.00元</sale>
    </book>
</books>
```

- 解析

```java
public class Test {
    public static void main(String[] args) throws Exception {
      // 1.导入Dom4j的jar包
        // 2.创建解析器对象
        SAXReader sr = new SAXReader();

        // 3.使用解析器对象读xml文件,生成Document对象
        Document document = sr.read("day14\\books.xml");

        // 4.使用Document对象获取根元素
        Element rootE = document.getRootElement();
        System.out.println("根元素的名称:" + rootE.getName());// books

        // 5.使用根元素获取根元素的所有子标签
        List<Element> list1 = rootE.elements();// 2个元素: book,book

        // 6.循环遍历根标签下的所有子标签
        for (Element e1 : list1) {
            System.out.println("元素名称:" + e1.getName() + ",book标签id属性的
值" + e1.attributeValue("id"));
            // 获取e1标签下的所有子标签
            List<Element> list2 = e1.elements();
            // 循环遍历e1标签下的所有子标签
            for (Element e2 : list2) {
                System.out.println("元素名称:" + e2.getName() + ",元素的文本:"
+ e2.getText());
            }
            System.out.println("---------");
        }
        System.out.println("--------------");
        // 获取根标签下的book标签
        Element bookE = rootE.element("book");
        System.out.println("book标签的名称:" + bookE.getName() + ",book标签的id
属性值:" + bookE.attributeValue("id"));

        // 获取book标签下author子标签的文本
        String text = bookE.elementText("author");
        System.out.println("book标签下author子标签的文本:" + text);// 张孝祥
    }
}
```

2.3 Dom4J结合XPath解析XML

2.3.1 介绍

XPath 使用**路径表达式**来选取HTML\XML 文档中的元素节点或属性节点。节点是通过沿着路径 (path) 来选取的。XPath在解析HTML\XML文档方面提供了独树一帜的路径思想。

说白了就是用来表示xml文件中标签或者属性的路径

2.3.2 XPath使用步骤

步骤1：导入jar包(dom4j和jaxen-1.1-beta-6.jar)

步骤2：创建解析器对象

步骤3: 通过dom4j的SaxReader解析器对象,读xml文件,生成Document对象

步骤4： 利用Xpath提供的api,根据XPath路径直接解析对应的标签或者属性.

document常用的api

- public Node selectSingleNode(String xpath); 获得一个节点(标签\元素)
- public List selectNodes(String xpath); 获得多个节点(标签\元素)

2.3.3 XPath语法(了解)

- XPath表达式，就是用于选取HTML文档中节点的表达式字符串。

  获取XML文档节点元素一共有如下4种XPath语法方式：

  1. 绝对路径表达式方式 例如: /根元素/子元素/子子元素...
  2. 相对路径表达式方式 例如: 子元素/子子元素.. 或者 ./子元素/子子元素..
  3. 全文搜索路径表达式方式 例如: //子元素//子子元素
  4. 谓语（条件筛选）方式 例如: //元素[@attr1=value]
  5. .....

2.3.3.1 绝对路径表达式(了解)

- 以/开头的路径叫做是绝对路径，绝对路径要从根元素开始写,是一个完整的路径

```
public class Test1_绝对路径 {
    public static void main(String[] args) throws Exception {
        // 绝对路径: 以/开头的路径叫做是绝对路径，绝对路径要从根元素开始写,是一个完整的路径
        // 1.创建解析器对象
        SAXReader sr = new SAXReader();

        // 2.读取xml文件,生成Document对象
        Document document = sr.read("day14\\tianqi.xml");

        // 3.需求:获取深圳的最低温度
        Node node = document.selectSingleNode("/天气预报/深圳/温度/最低温度");
        System.out.println("深圳的最低温度:" + node.getText());// 24
```

```
    }
}
```

2.3.3.2 相对路径表达式(了解)

- 相对路径就是**相对当前节点元素位置**继续查找节点，**不以/开头**, ../ 表示上一个元素, ./表示当前元素

```
public class Test2_相对路径 {
    public static void main(String[] args) throws Exception{
        // 相对路径:相对路径就是相对当前节点元素位置继续查找节点，不以/开头， ../ 表示上一个元素， ./表示当前元素
        // 1.创建解析器对象
        SAXReader sr = new SAXReader();

        // 2.读取xml文件,生成Document对象
        Document document = sr.read("day14\\tianqi.xml");

        // 3.需求:获取深圳的最低温度
        Node node1 = document.selectSingleNode("/天气预报/深圳/温度/最低温度");
        System.out.println("深圳的最低温度:" + node1.getText());// 24

        // 4.需求:根据node1获取广州黄浦区的最高温度
        Node node2 = node1.selectSingleNode("../../../广州/黄浦区/温度/最高温度");
        System.out.println("广州黄浦区的最高温度:"+node2.getText());// 31

    }
}
```

2.3.3.3 全文搜索路径表达式(了解)

- 代表不论中间有多少层,直接获取所有子元素中满足条件的元素,需要使用//

```
public class Test3_全文搜索路径 {
    public static void main(String[] args) throws Exception{
        // 全文搜索路径:代表不论中间有多少层,直接获取所有子元素中满足条件的元素,需要使用//
        // 1.创建解析器对象
        SAXReader sr = new SAXReader();

        // 2.读取xml文件,生成Document对象
        Document document = sr.read("day14\\tianqi.xml");

        // 3.需求:获取所有的湿度
        List<Element> list = document.selectNodes("//湿度");
        for (Element e : list) {
            System.out.println("湿度:"+e.getText());
        }

    }
}
```

2.3.3.4 谓语（条件筛选 了解）

- 介绍

  谓语，又称为条件筛选方式，就是根据条件过滤判断进行选取节点

  格式：

  String xpath1="//元素[@attr1=value]";//获取元素属性attr1=value的元素

  String xpath2="//元素[@attr1>value]/@attr1"//获取元素属性attr1>value的d的所有attr1的值

  String xpath3="//元素[@attr1=value]/text()";//获取符合条件元素体的自有文本数据

```
String xpath4="//元素[@attr1=value]/html()";//获取符合条件元素体的自有html代码数据。
```

String xpath3="//元素[@attr1=value]/allText()";//获取符合条件元素体的所有文本数据（包含子元素里面的文本）

```
public class Test4_条件筛选 {
    public static void main(String[] args) throws Exception {
        // 1.创建解析器对象
        SAXReader sr = new SAXReader();

        // 2.读取xml文件,生成Document对象
        Document document = sr.read("day14\\tianqi.xml");

        // 3.需求:获取所有等级为C的最高温度
        List<Element> list = document.selectNodes("//最高温度[@level='C']");
        for (Element e : list) {
            System.out.println("等级为C的最高温度:" + e.getText());
        }

    }
}
```

第三章 正则表达式

3.1 正则表达式的概念及演示

- **概述**: 正则表达式其实就是一个**匹配规则,用来替换之前复杂的if结构判断**
- 在Java中，我们经常需要验证一些字符串，是否符合规则, 例如：校验qq号码是否正确,手机号码是否正确,邮箱是否正确等等。那么如果使用if就会很麻烦, 而正则表达式就是用来验证各种字符串的规则。它内部描述了一些规则，我们可以验证用户输入的字符串是否匹配这个规则。
- 先看一个不使用正则表达式验证的例子：下面的程序让用户输入一个QQ号码，我们要验证：
  - QQ号码必须是5--15位长度
  - 而且必须全部是数字
  - 而且首位不能为0
- 使用if判断方式验证:

```
/**
```

```
     *  校验qq号码是否符合规则
     * @param qq
     * @return
     */
    public static boolean checkQQ1(String qq){
        //- QQ号码必须是5--15位长度
        if (qq.length() < 5 || qq.length() > 15){
            return false;
        }

        //- 而且必须全部是数字
        for (int i = 0; i < qq.length(); i++) {
            char c = qq.charAt(i);
            // 判断每一个字符,如果有一个字符不是数字,就直接返回false
            if (c < '0' || c > '9'){
                return false;
            }
        }

        //- 而且首位不能为0
        if (qq.charAt(0) == '0'){
            return false;
        }

        // 说明符合规则
        return true;
    }
```

- 使用正则表达式验证：
  - `public boolean matches(String regex);` 判断此字符串是否匹配给定的正则表达式

```
public static boolean checkQQ2(String qq){
    return qq.matches("[1-9]\\d{4,14}");
}
```

3.2 正则表达式的基本使用

3.2.1 正则表达式-字符类

- 语法示例：**[] 表示匹配单个字符　^ 取反　- 范围**

1. [abc]：代表a或者b，或者c字符中的一个。
2. [^abc]：代表除a,b,c以外的任何单个字符。
3. [a-z]：代表a-z的所有小写字符中的一个。 左右包含
4. [A-Z]：代表A-Z的所有大写字符中的一个。
5. [0-9]：代表0-9之间的某一个数字字符。
6. [a-zA-Z0-9]：代表a-z或者A-Z或者0-9之间的任意一个字符。
7. [a-dm-p]：a 到 d 或 m 到 p之间的任意一个字符。
8. ....

- 代码示例：

```
public class Test1_字符类 {
    public static void main(String[] args) {
          /*
            正则表达式-字符类
                - 语法示例：[] 表示匹配单个字符    ^ 取反     - 范围
                  1. [abc]：代表a或者b，或者c字符中的一个。
                  2. [^abc]：代表除a,b,c以外的任何字符。
                  3. [a-z]：代表a-z的所有小写字符中的一个。 左右包含
                  4. [A-Z]：代表A-Z的所有大写字符中的一个。
                  5. [0-9]：代表0-9之间的某一个数字字符。
                  6. [a-zA-Z0-9]：代表a-z或者A-Z或者0-9之间的任意一个字符。
                  7. [a-dm-p]：a 到 d 或 m 到 p之间的任意一个字符。
         */
        // 需求:
        // 1.验证字符串是否以h开头，以d结尾，中间是a,e,i,o,u中某个字符
        // 2.验证字符串是否以h开头，以d结尾，中间不是a,e,i,o,u中的某个字符
        // 3.验证字符串是否a-z的任何一个小写字符开头，后跟ad
        // 4.验证字符串是否以a-d或者m-p之间某个字符开头，后跟ad
           // 需求:
        // 1.验证字符串是否以h开头，以d结尾，中间是a,e,i,o,u中某个字符
        System.out.println("had".matches("h[aeiou]d"));// true
        System.out.println("haed".matches("h[aeiou]d"));// false
        System.out.println("hbd".matches("h[aeiou]d"));// false
        System.out.println("h1d".matches("h[aeiou]d"));// false
        System.out.println("---------------");

        // 2.验证字符串是否以h开头，以d结尾，中间不是a,e,i,o,u中的某个字符
        System.out.println("had".matches("h[^aeiou]d"));// false
        System.out.println("haed".matches("h[^aeiou]d"));// false
        System.out.println("hbd".matches("h[^aeiou]d"));// true
        System.out.println("h1d".matches("h[^aeiou]d"));// true
        System.out.println("---------------");

        // 3.验证字符串是否a-z的任何一个小写字符开头，后跟ad
        System.out.println("had".matches("[a-z]ad"));// true
        System.out.println("abad".matches("[a-z]ad"));// false
        System.out.println("Aad".matches("[a-z]ad"));// false
        System.out.println("1ad".matches("[a-z]ad"));// false
        System.out.println("---------------");

        // 4.验证字符串是否以a-d或者m-p之间某个字符开头，后跟ad
        System.out.println("had".matches("[a-dm-p]ad"));// false
        System.out.println("aad".matches("[a-dm-p]ad"));// true
        System.out.println("mad".matches("[a-dm-p]ad"));// true
        System.out.println("Aad".matches("[a-dm-p]ad"));// false
    }
}
```

3.2.2 正则表达式-逻辑运算符

- 语法示例：
  1. &&：并且
  2. |　：或者
- 代码示例：

```
public class Test2_逻辑运算符 {
    public static void main(String[] args) {
          /*
            1. &&：并且
            2. | ：或者
         */
        // 需求:
        //1.要求字符串是小写辅音字符开头，后跟ad  除了a,e,i,o,u之外,其他的都是辅音字母
        //2.要求字符串是aeiou中的某个字符开头，后跟ad
            //1.要求字符串是小写辅音字符开头，后跟ad  除了a,e,i,o,u之外,其他的都是辅音字母
        System.out.println("had".matches("[a-z&&[^aeiou]]ad"));// true
        System.out.println("aad".matches("[a-z&&[^aeiou]]ad"));// false
        System.out.println("Aad".matches("[a-z&&[^aeiou]]ad"));// false
        System.out.println("1ad".matches("[a-z&&[^aeiou]]ad"));// false
        System.out.println("-----");

        //2.要求字符串是aeiou中的某个字符开头，后跟ad
        System.out.println("had".matches("[a|e|i|o|u]ad"));// false
        System.out.println("aad".matches("[a|e|i|o|u]ad"));// true
        System.out.println("Aad".matches("[a|e|i|o|u]ad"));// false
        System.out.println("1ad".matches("[a|e|i|o|u]ad"));// false
    }
}
```

3.2.3 正则表达式-预定义字符

- 语法示例：
    1. "." ： 匹配任何字符。如果要表示一个字符点,那么就得使用`\\.`
    2. "\d"：任何数字[0-9]的简写；
    3. "\D"：任何非数字[^0-9]的简写；
    4. "\s"： 空白字符：[\t\n\x0B\f\r] 的简写
    5. "\S"： 非空白字符：[^\s] 的简写
    6. "\w"：单词字符：[a-zA-Z_0-9]的简写
    7. "\W"：非单词字符：[^\w]
- 代码示例：

```
public class Test3_预定义字符 {
    public static void main(String[] args) {
          // 需求:
        // 1.验证字符串是否是一个3位的整数
        System.out.println("123".matches("[1-9]\\d\\d"));// true
        System.out.println("1234".matches("[1-9]\\d\\d"));// false
        System.out.println("023".matches("[1-9]\\d\\d"));// false
        System.out.println("a23".matches("[1-9]\\d\\d"));// false
        System.out.println("--------");

        // 2.验证手机号：1开头，第二位：3/5/8，剩下9位都是0-9的数字

 System.out.println("13412345678".matches("1[358]\\d\\d\\d\\d\\d\\d\\d\\d\\d\\d"));
// true

 System.out.println("134123456789".matches("1[358]\\d\\d\\d\\d\\d\\d\\d\\d\\d\\d"))
;// false
```

```
 System.out.println("17412345678".matches("1[358]\\d\\d\\d\\d\\d\\d\\d\\d\\d\\d"));
// false

 System.out.println("23412345678".matches("1[358]\\d\\d\\d\\d\\d\\d\\d\\d\\d\\d"));
// false

 System.out.println("134a2345678".matches("1[358]\\d\\d\\d\\d\\d\\d\\d\\d\\d\\d"));
// false
        System.out.println("--------");

        // 3.验证字符串是否以h开头，以d结尾，中间是任何一个字符
        System.out.println("had".matches("h.d"));// true
        System.out.println("h.d".matches("h.d"));// true
        System.out.println("h%d".matches("h.d"));// true
        System.out.println("h%&d".matches("h.d"));// false
        System.out.println("a%d".matches("h.d"));// false
        System.out.println("--------");

        // 4.验证str是否是：h.d
        System.out.println("had".matches("h\\.d"));// false
        System.out.println("h.d".matches("h\\.d"));// true
        System.out.println("h%d".matches("h\\.d"));// false
        System.out.println("h%&d".matches("h\\.d"));// false
        System.out.println("a%d".matches("h\\.d"));// false

    }
}
```

3.2.4 正则表达式-数量词

- 语法示例：
    1. X? : 0次或1次
    2. X* : 0次到多次
    3. X+ : 1次或多次
    4. X{n} : 恰好n次
    5. X{n,} : 至少n次,包含n
    6. X{n,m}: n到m次(n和m都是包含的)
- 代码示例：

```
public class Test4_数量词 {
    public static void main(String[] args) {
         /*
            正则表达式-数量词
            - 语法示例：
              1. X? : 0次或1次
              2. X* : 0次到多次
              3. X+ : 1次或多次
              4. X{n} : 恰好n次
              5. X{n,} : 至少n次
              6. X{n,m}: n到m次(n和m都是包含的)
         */
        // 1..验证字符串是否是一个3位的整数
        System.out.println("123".matches("[1-9]\\d{2}"));// true
        System.out.println("1234".matches("[1-9]\\d{2}"));// false
```

```
        System.out.println("023".matches("[1-9]\\d{2}"));// false
        System.out.println("a23".matches("[1-9]\\d{2}"));// false
        System.out.println("--------");

        // 2.验证str是否是多位数字
        System.out.println("1".matches("\\d+"));// true
        System.out.println("12".matches("\\d+"));// true
        System.out.println("123".matches("\\d+"));// true
        System.out.println("1234".matches("\\d+"));// true
        System.out.println("023".matches("\\d+"));// true
        System.out.println("a23".matches("\\d+"));// false
        System.out.println("--------");

        // 3.验证手机号：1开头，第二位：3/5/8，剩下9位都是0-9的数字
        System.out.println("13412345678".matches("1[358]\\d{9}"));// true
        System.out.println("134123456789".matches("1[358]\\d{9}"));// false
        System.out.println("17412345678".matches("1[358]\\d{9}"));// false
        System.out.println("23412345678".matches("1[358]\\d{9}"));// false
        System.out.println("134a2345678".matches("1[358]\\d{9}"));// false
        System.out.println("--------");

        // 4.验证qq号码：1).5--15位；2).全部是数字;3).第一位不是0
        System.out.println("123456".matches("[1-9]\\d{4,14}"));// true
        System.out.println("1234".matches("[1-9]\\d{4,14}"));// false
        System.out.println("1234a56".matches("[1-9]\\d{4,14}"));// false
        System.out.println("023456".matches("[1-9]\\d{4,14}"));// false

    }
}
```

3.2.5 正则表达式-分组括号( )

```
public class Test5_分组括号 {
    public static void main(String[] args) {
        String str = "AB8JK-REI9O-324FD-LKJFD-656FD-RE7FD";
        System.out.println(str.matches("([A-ZO-9]{5}-)+[A-ZO-9]{5}"));

        // 需求: 高高兴兴,快快乐乐,开开心心,aabb.....
        // \\1 表示第一组   \\2 表示第二组  \\3 表示第三组
        // (.)\\1{1} 表示第一组再来1次   (.)\\2{1} 表示第二组再来1次    注意:如果是再来1
次可以省略{1}
        // (.)\\1{2} 表示第一组再来2次   (.)\\2{2} 表示第二组再来2次    aaabbb
        System.out.println("高高兴兴".matches("(.)\\1(.)\\2"));// true
        System.out.println("快快乐乐".matches("(.)\\1(.)\\2"));// true
        System.out.println("开开心心".matches("(.)\\1(.)\\2"));// true
        System.out.println("aabb".matches("(.)\\1(.)\\2"));// true
        System.out.println("abcd".matches("(.)\\1(.)\\2"));// false
        System.out.println("%^&*".matches("(.)\\1(.)\\2"));// false
        System.out.println("---------");

        // 需求: 高兴高兴,快乐快乐,开心开心,abab,...
        System.out.println("高兴高兴".matches("(..)\\1"));// true
        System.out.println("快乐快乐".matches("(..)\\1"));// true
        System.out.println("开心开心".matches("(..)\\1"));// true
        System.out.println("abab".matches("(..)\\1"));// true
```

```
    }
}
```

3.3 String中正则表达式的使用

3.3.1 String的split方法中使用正则表达式

- String类的split()方法原型：

```
public String[] split(String regex)
//参数regex就是一个正则表达式。可以将当前字符串中匹配regex正则表达式的符号作为"分隔符"来切割字符串。
```

- 代码示例：

```
public class Test1_split {
    public static void main(String[] args) {
        // public String[] split(String regex)

        String str1 = "itheima-itcast-java-php";
        // 对str1进行切割
        String[] arr1 = str1.split("-");// 普通字符串
        for (String s : arr1) {
            System.out.println("s:" + s);
        }

        System.out.println("----------");

        String str2 = "itheima.itcast.java.php";
        // 对str2进行切割
        String[] arr2 = str2.split("\\.");// 正则表达式字符串
        for (String s : arr2) {
            System.out.println("s:" + s);
        }

    }
}
```

3.3.2 String类的replaceAll方法中使用正则表达式

- String类的replaceAll()方法原型：

```
public String replaceAll(String regex,String newStr)
//参数regex就是一个正则表达式。可以将当前字符串中匹配regex正则表达式的字符串替换为newStr。
```

- 代码示例：

```
public class Test2_replaceAll {
```

```
    public static void main(String[] args) {
        // public String replaceAll(String regex,String newStr)
        // 用户输入信息:
        Scanner sc = new Scanner(System.in);
        System.out.println("请输入一个字符串:");
        String msg = sc.nextLine();

        // 过滤非法字符,或者不文明字符
        String message = msg.replaceAll("草|日|他妈的|靠", "***");
        System.out.println("message:" + message);

    }
}
```

总结

必须练习:

1.定义XML文件----XML的组成元素(文档声明,标签,属性,注释,转义字符,字符区)----->1.2
2.使用Dom4j解析XML文件---结合XPATH路径---->2.2,2.3
3.有时间的话分析分析正则表达式,写点简单的正则表达---目的是为了能看懂正则表达式---->3.2
String类中使用正则表达式--->matches,split,replaceAll ---->3.3

- 能够说出XML的作用
  1.作为配置文件=----框架阶段
  2.存储数据,传输数据

- 了解XML的组成元素
  文档声明,标签,属性,注释,转义字符,字符区

- 能够说出有哪些XML约束技术
  dtd,schema

- 能够说出解析XML文档DOM方式原理
  解析器-->读取xml文档-->Dom树--->Document对象

- 能够使用dom4j解析XML文档
  1.导入dom4j的jar包
  2.把jar包添加到classpath路径中
  3.创建解析器对象
  4.使用解析器对象读取xml文件,生成Document对象
  5.使用Document对象获取根标签
  6.使用根标签获取子标签,....

- 能够使用xpath解析XML
  1.导入dom4j和xpath的jar包
  2.把jar包添加到classpath路径中
  3.创建解析器对象
  4.使用解析器对象读取xml文件,生成Document对象
  5.使用Document对象结合xpath路径获取指定标签

- 能够理解正则表达式的作用
  作为匹配规则,替换复杂的if判断操作

- 能够使用正则表达式的字符类

[] 表示匹配单个字符,^表示取反,-表示范围
    1. [abc]：代表a或者b，或者c字符中的一个。
    2. [^abc]：代表除a,b,c以外的任何字符。
    3. [a-z]：代表a-z的所有小写字符中的一个。 左右包含
    4. [A-Z]：代表A-Z的所有大写字符中的一个。
    5. [0-9]：代表0-9之间的某一个数字字符。
    6. [a-zA-Z0-9]：代表a-z或者A-Z或者0-9之间的任意一个字符。
    7. [a-dm-p]：a 到 d 或 m 到 p之间的任意一个字符。
    8. ....
- 能够使用正则表达式的逻辑运算符
  && 并且
  | 或者

- 能够使用正则表达式的预定义字符类
    1. "." ： 匹配任何字符。如果要表示一个字符点,那么就得使用\.
    2. "\d"：任何数字[0-9]的简写；
    3. "\D"：任何非数字[^0-9]的简写；
    4. "\s"： 空白字符：[\t\n\xOB\f\r] 的简写
    5. "\S"： 非空白字符：[^\s] 的简写
    6. "\w"：单词字符：[a-zA-Z_0-9]的简写
    7. "\W"：非单词字符：[^\w]

- 能够使用正则表达式的数量词
    1. X? : 0次或1次
    2. X* : 0次到多次
    3. X+ : 1次或多次
    4. X{n} : 恰好n次
    5. X{n,} : 至少n次
    6. X{n,m}: n到m次(n和m都是包含的)

- 能够使用正则表达式的分组
  ()
- 能够在String的split方法中使用正则表达式
  ```
  String[] split(String regex);
  boolean matches(String regex);
  String replaceAll(String regex,String newStr)
  ```