# 1.xml

## 1.1概述【理解】

- 万维网联盟(W3C)

  万维网联盟(W3C)创建于1994年，又称W3C理事会。1994年10月在麻省理工学院计算机科学实验室成立。 建立者： Tim Berners-Lee (蒂姆·伯纳斯·李)。 是Web技术领域最具权威和影响力的国际中立性技术标准机构。到目前为止，W3C已发布了200多项影响深远的Web技术标准及实施指南，

  - 如广为业界采用的超文本标记语言HTML（标准通用标记语言下的一个应用）、
  - 可扩展标记语言XML（标准通用标记语言下的一个子集）
  - 以及帮助残障人士有效获得Web信息的无障碍指南（WCAG）等

- xml概述

  XML的全称为(EXtensible Markup Language)，是一种可扩展的标记语言 标记语言: 通过标签来描述数据的一门语言(标签有时我们也将其称之为元素) 可扩展：标签的名字是可以自定义的,XML文件是由很多标签组成的,而标签名是可以自定义的

- 作用
  - 用于进行存储数据和传输数据
  - 作为软件的配置文件

- 作为配置文件的优势
  - 可读性好
  - 可维护性高

## 1.2标签的规则【应用】

- 标签由一对尖括号和合法标识符组成

  ```
  <student>
  ```

- 标签必须成对出现

```
<student> </student>
前边的是开始标签，后边的是结束标签
```

- 特殊的标签可以不成对,但是必须有结束标记

```
<address/>
```

- 标签中可以定义属性,属性和标签名空格隔开,属性值必须用引号引起来

```
<student id="1"> </student>
```

- 标签需要正确的嵌套

```
这是正确的: <student id="1"> <name>张三</name> </student>
这是错误的: <student id="1"><name>张三</student></name>
```

## 1.3语法规则【应用】

- 语法规则
  - XML文件的后缀名为：xml
  - 文档声明必须是第一行第一列

    <?xml version="1.0" encoding="UTF-8" standalone="yes"?> version：该属性是必须存在的 encoding：该属性不是必须的

    打开当前xml文件的时候应该是使用什么字符编码表(一般取值都是UTF-8)

    standalone: 该属性不是必须的，描述XML文件是否依赖其他的xml文件，取值为yes/no
  - 必须存在一个根标签，有且只能有一个
  - XML文件中可以定义注释信息
  - XML文件中可以存在以下特殊字符

    ```
    < < 小于
    > > 大于
    &amp; & 和号
    &apos; ' 单引号
    &quot; " 引号
    ```
  - XML文件中可以存在CDATA区

    <![CDATA[ …内容… ]]>
- 示例代码

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--注释的内容-->
<!--本xml文件用来描述多个学生信息-->
<students>

    <!--第一个学生信息-->
```

```
    <student id="1">
        <name>张三</name>
        <age>23</age>
        <info>学生&lt; &gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;的信息</info>
        <message> <![CDATA[内容 <<<<<< >>>>>> ]]></message>
    </student>

    <!--第二个学生信息-->
    <student id="2">
        <name>李四</name>
        <age>24</age>
    </student>

</students>
```

## 1.4xml解析【应用】

- 概述

  xml解析就是从xml中获取到数据

- 常见的解析思想

  DOM(Document Object Model)文档对象模型:就是把文档的各个组成部分看做成对应的对象。 会把xml文件全部加载到内存,在内存中形成一个树形结构,再获取对应的值

Document对象：整个xml文档
Element对象：所有标签
Attribute对象：所有属性
Text对象：所有文本内容
Node对象

DOM解析思想

```
<?xml version="1.0" encoding="UTF-8" ?>
<students>
    <!--第一个学生信息-->
    <student id="1">
        <name>张三</name>
        <age>23</age>
    </student>
    <!--第二个学生信息-->
    <student id="2">
        <name>李四</name>
        <age>24</age>
    </student>
</students>
```

Document
students
student
student
id
name
age
id
name
age
1
张三
23
2
李四
24

- 常见的解析工具
  - JAXP: SUN公司提供的一套XML的解析的API
  - JDOM: 开源组织提供了一套XML的解析的API-jdom
  - DOM4J: 开源组织提供了一套XML的解析的API-dom4j,全称：Dom For Java
  - pull: 主要应用在Android手机端解析XML

- 解析的准备工作
  1. 我们可以通过网站：https://dom4j.github.io/ 去下载dom4j

今天的资料中已经提供,我们不用再单独下载了,直接使用即可

2. 将提供好的dom4j-1.6.1.zip解压,找到里面的dom4j-1.6.1.jar
3. 在idea中当前模块下新建一个libs文件夹,将jar包复制到文件夹中
4. 选中jar包 -> 右键 -> 选择add as library即可

- 需求
  - 解析提供好的xml文件
  - 将解析到的数据封装到学生对象中
  - 并将学生对象存储到ArrayList集合中
  - 遍历集合
- 代码实现

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--注释的内容-->
<!--本xml文件用来描述多个学生信息-->
<students>

    <!--第一个学生信息-->
    <student id="1">
        <name>张三</name>
        <age>23</age>
    </student>

    <!--第二个学生信息-->
    <student id="2">
        <name>李四</name>
        <age>24</age>
    </student>

</students>

// 上边是已经准备好的student.xml文件
public class Student {
    private String id;
    private String name;
    private int age;

    public Student() {
    }

    public Student(String id, String name, int age) {
        this.id = id;
        this.name = name;
        this.age = age;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
```

```java
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "Student{" +
                "id='" + id + '\'' +
                ", name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}

/**
 * 利用dom4j解析xml文件
 */
public class XmlParse {
    public static void main(String[] args) throws DocumentException {
        //1.获取一个解析器对象
        SAXReader saxReader = new SAXReader();
        //2.利用解析器把xml文件加载到内存中,并返回一个文档对象
        Document document = saxReader.read(new File("myxml\\xml\\student.xml"));
        //3.获取到根标签
        Element rootElement = document.getRootElement();
        //4.通过根标签来获取student标签
        //elements():可以获取调用者所有的子标签.会把这些子标签放到一个集合中返回.
        //elements("标签名"):可以获取调用者所有的指定的子标签,会把这些子标签放到一个集合中并返回
        //List list = rootElement.elements();
        List<Element> studentElements = rootElement.elements("student");
        //System.out.println(list.size());

        //用来装学生对象
        ArrayList<Student> list = new ArrayList<>();


        //5.遍历集合,得到每一个student标签

        for (Element element : studentElements) {
```

```java
            //element依次表示每一个student标签

            //获取id这个属性
            Attribute attribute = element.attribute("id");
            //获取id的属性值
            String id = attribute.getValue();


            //获取name标签
            //element("标签名"):获取调用者指定的子标签
            Element nameElement = element.element("name");
            //获取这个标签的标签体内容
            String name = nameElement.getText();


            //获取age标签
            Element ageElement = element.element("age");
            //获取age标签的标签体内容
            String age = ageElement.getText();

//            System.out.println(id);
//            System.out.println(name);
//            System.out.println(age);

            Student s = new Student(id,name,Integer.parseInt(age));
            list.add(s);
        }
        //遍历操作
        for (Student student : list) {
            System.out.println(student);
        }
    }
}
```

## 1.5DTD约束【理解】

- 什么是约束

  用来限定xml文件中可使用的标签以及属性

- 约束的分类

  - DTD
  - schema

- 编写DTD约束

  - 步骤

    1. 创建一个文件，这个文件的后缀名为.dtd

    2. 看xml文件中使用了哪些元素

       <!ELEMENT> 可以定义元素

    3. 判断元素是简单元素还是复杂元素

       简单元素：没有子元素。 复杂元素：有子元素的元素；

- 代码实现

  ```
  <!ELEMENT persons (person)>
  <!ELEMENT person (name,age)>
  <!ELEMENT name (#PCDATA)>
  <!ELEMENT age (#PCDATA)>
  ```

- 引入DTD约束
  - 引入DTD约束的三种方法
    - 引入本地dtd

      <!DOCTYPE 根元素名称 SYSTEM 'DTD文件的路径'>
    - 在xml文件内部引入

      <!DOCTYPE 根元素名称 [ dtd文件内容 ]>
    - 引入网络dtd

      <!DOCTYPE 根元素的名称 PUBLIC "DTD文件名称" "DTD文档的URL">
  - 代码实现
    - 引入本地DTD约束

      ```
      // 这是persondtd.dtd文件中的内容,已经提前写好
      <!ELEMENT persons (person)>
      <!ELEMENT person (name,age)>
      <!ELEMENT name (#PCDATA)>
      <!ELEMENT age (#PCDATA)>

      // 在person1.xml文件中引入persondtd.dtd约束
      <?xml version="1.0" encoding="UTF-8" ?>
      <!DOCTYPE persons SYSTEM 'persondtd.dtd'>

      <persons>
          <person>
              <name>张三</name>
              <age>23</age>
          </person>

      </persons>
      ```

    - 在xml文件内部引入

      ```
      <?xml version="1.0" encoding="UTF-8" ?>
      <!DOCTYPE persons [
              <!ELEMENT persons (person)>
              <!ELEMENT person (name,age)>
              <!ELEMENT name (#PCDATA)>
              <!ELEMENT age (#PCDATA)>
              ]>


      <persons>
      ```

```
    <person>
        <name>张三</name>
        <age>23</age>
    </person>

</persons>
```

- 引入网络dtd

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE persons PUBLIC "dtd文件的名称" "dtd文档的URL">

<persons>
    <person>
        <name>张三</name>
        <age>23</age>
    </person>

</persons>
```

- DTD语法
  - 定义元素

    定义一个元素的格式为：<!ELEMENT 元素名 元素类型> 简单元素：

    EMPTY: 表示标签体为空

    ANY: 表示标签体可以为空也可以不为空

    PCDATA: 表示该元素的内容部分为字符串

    复杂元素： 直接写子元素名称. 多个子元素可以使用","或者"|"隔开； ","表示定义子元素的顺序 ; "|": 表示子元素只能出现任意一个 "?"零次或一次, "+"一次或多次, "*"零次或多次;如果不写则表示出现一次

    定义一个元素的格式为：<!ELEMENT 元素名 元素类型>

    **简单元素:**

    EMPTY: 表示标签体为空

    ANY: 表示标签体可以为空也可以不为空

    PCDATA: 表示该元素的内容部分为字符串

    **复杂元素:**

    直接写子元素名称。

    多个子元素可以使用","或者"|"隔开；

    ","表示定义子元素的顺序；

    "|": 表示子元素只能出现任意一个

    "?"零次或一次,

    "+"一次或多次,

    "*"零次或多次;

    如果不写则表示出现一次

```
<?xml version="1.0" encoding="UTF-8" ?>
<persons>
    <person>
        <name>张三</name>
        <age>23</age>
    </person>
</persons>
```

```
<!ELEMENT persons (person+) >
<!ELEMENT person (name , age)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT age (#PCDATA) >
```

  - 定义属性

    格式

    定义一个属性的格式为：<!ATTLIST 元素名称 属性名称 属性的类型 属性的约束>

    属性的类型：
    CDATA类型：普通的字符串

    属性的约束:

    // #REQUIRED： 必须的 // #IMPLIED： 属性不是必需的 // #FIXED value：属性值是固定的

- 代码实现

```
<!ELEMENT persons (person+)>
<!ELEMENT person (name,age)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT age (#PCDATA)>
<!ATTLIST person id CDATA #REQUIRED>


<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE persons SYSTEM 'persondtd.dtd'>

<persons>
    <person id="001">
        <name>张三</name>
        <age>23</age>
    </person>

    <person id = "002">
        <name>张三</name>
        <age>23</age>
    </person>

</persons>
```

## 1.6schema约束【理解】

- schema和dtd的区别
  1. schema约束文件也是一个xml文件，符合xml的语法，这个文件的后缀名.xsd
  2. 一个xml中可以引用多个schema约束文件，多个schema使用名称空间区分（名称空间类似于java包名）
  3. dtd里面元素类型的取值比较单一常见的是PCDATA类型，但是在schema里面可以支持很多个数据类型
  4. schema 语法更加的复杂

Schema文件用来约束一个xml文件
同时也被别的文件约束着

- 编写schema约束
  - 步骤

    1，创建一个文件，这个文件的后缀名为.xsd。 2，定义文档声明 3，schema文件的根标签为： 4，在中定义属性： xmlns=http://www.w3.org/2001/XMLSchema 5，在中定义属性 ： targetNamespace =唯一的url地址，指定当前这个schema文件的名称空间。 6，在中定义属性 ： elementFormDefault="qualified"，表示当前schema文件是一个质量良好的文件。 7，通过element定义元素 8，判断当前元素是简单元素还是复杂元素

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<persons>
    <person>
        <name>张三</name>
        <age>23</age>
    </person>
</persons>
```

person.xsd

```
<?xml version="1.0" encoding="UTF-8" ?>
<schema
xmlns= “本文件是约束别人的"
targetNamespace= “自己的名称空间”
elementFormDefault=“本文件是质量良好的">

  <element name=“根标签名">

     <complexType>   复杂的元素
         <sequence>  里面的子元素必须要按照顺序定义


         </sequence>
     </complexType>

  </element>

</schema>
```

1，创建一个文件，这个文件的后缀名为.xsd。

2，定义文档声明

3，schema文件的根标签为：<schema>

4，在<schema>中定义属性：

xmlns=http://www.w3.org/2001/XMLSchema

5，在<schema>中定义属性：

targetNamespace =唯一的url地址。

指定当前这个schema文件的名称空间。

6，在<schema>中定义属性：

elementFormDefault="qualified “

表示当前schema文件是一个质量良好的文件。

7，通过element定义元素

8，**判断当前元素是简单元素还是复杂元素**

- 代码实现

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<schema
    xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.itheima.cn/javase"
    elementFormDefault="qualified"
>

    <!--定义persons复杂元素-->
    <element name="persons">
        <complexType>
            <sequence>
                <!--定义person复杂元素-->
                <element name = "person">
                    <complexType>
                        <sequence>
                            <!--定义name和age简单元素-->
                            <element name = "name" type = "string"></element>
                            <element name = "age" type = "string"></element>
                        </sequence>

                    </complexType>
                </element>
            </sequence>
        </complexType>

    </element>

</schema>
```

- 引入schema约束

- 步骤

  1，在根标签上定义属性xmlns="http://www.w3.org/2001/XMLSchema-instance" 2，通过xmlns引入约束文件的名称空间 3，给某一个xmlns属性添加一个标识，用于区分不同的名称空间 格式为: xmlns:标识="名称空间地址" ,标识可以是任意的，但是一般取值都是xsi 4，通过xsi:schemaLocation指定名称空间所对应的约束文件路径 格式为：xsi:schemaLocation = "名称空间url 文件路径"

- 代码实现

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

<persons
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.itheima.cn/javase"
    xsi:schemaLocation="http://www.itheima.cn/javase person.xsd"
>
    <person>
        <name>张三</name>
        <age>23</age>
    </person>


</persons>
```

- schema约束定义属性
  - 代码示例

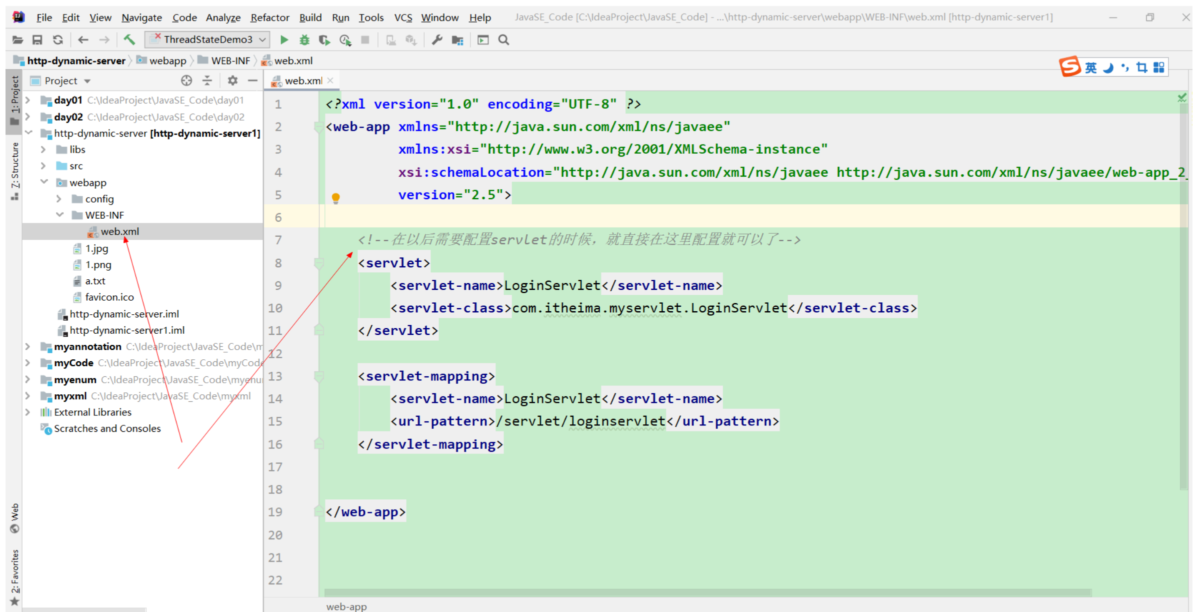
```
<?xml version="1.0" encoding="UTF-8" ?>
<schema
    xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.itheima.cn/javase"
    elementFormDefault="qualified"
>

    <!--定义persons复杂元素-->
    <element name="persons">
        <complexType>
            <sequence>
                <!--定义person复杂元素-->
                <element name = "person">
                    <complexType>
                        <sequence>
                            <!--定义name和age简单元素-->
                            <element name = "name" type = "string"></element>
                            <element name = "age" type = "string"></element>
                        </sequence>

                        <!--定义属性，required( 必须的)/optional( 可选的)-->
                        <attribute name="id" type="string" use="required"></attribute>
                    </complexType>

                </element>
            </sequence>
```

```xml
            </complexType>
        </element>

</schema>


<?xml version="1.0" encoding="UTF-8" ?>
<persons
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.itheima.cn/javase"
    xsi:schemaLocation="http://www.itheima.cn/javase person.xsd"
>
    <person id="001">
        <name>张三</name>
        <age>23</age>
    </person>

</persons>
```

## 1.7服务器改进【应用】

- 准备xml文件

  1. 在当前模块下的webapp目录下新建一个文件夹，名字叫WEB-INF
  2. 新建一个xml文件，名字叫web.xml
  3. 将资料中的web.xml文件中引入约束的代码复制到新建的web.xml文件中
  4. 将要解析的数据配置到xml文件中




- 需求

  把uri和servlet信息放到一个concurrentHashMap集合当中 当浏览器请求一个动态资源时，我们会获取uri对应的servlet来处理当前业务

- 实现步骤

  1. 导入dom4j的jar包
  2. 定义一个XmlParseServletConfig类实现ParseServletConfig接口

3. 在parse方法里面就可以解析xml文件了

- 代码实现

```
// web.xml配置文件中配置的信息
<?xml version="1.0" encoding="UTF-8" ?>
<web-app 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"
         version="2.5">

    <!--在以后需要配置servlet的时候，就直接在这里配置就可以了-->
    <servlet>
        <servlet-name>LoginServlet</servlet-name>
        <servlet-class>com.itheima.myservlet.LoginServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>LoginServlet</servlet-name>
        <url-pattern>/servlet/loginservlet</url-pattern>
    </servlet-mapping>

</web-app>

// 定义一个XmlParseServletConfig类实现ParseServletConfig接口
public class XMLParseServletConfig implements ParseServletConfig {
    //定义web.xml文件的路径
    private static final String WEB_XML_PATH = "http-dynamic-server/webapp/WEB-
INF/web.xml";

    //在parse方法里面就可以解析xml文件了
    @Override
    public void parse() {
        try {
            //1.创建一个解析器对象(注意:如果解析器对象等不能使用,请检查一下jar包是否导入)
            SAXReader saxReader = new SAXReader();

            //2.利用解析器把xml文件读取到内存中
            Document document = saxReader.read(new File(WEB_XML_PATH));

            //3.获取根节点元素对象
            Element rootElement = document.getRootElement();

            //创建一个Map集合，用来存储servlet的配置信息
            HashMap<String,String> servletInfoHashMap = new HashMap<>();

            //4.获取根元素对象下所有的servlet元素的对象
            List<Element> servletInfos = rootElement.elements("servlet");

            //5.遍历集合，依次获取到每一个servlet标签对象
            for (Element servletInfo : servletInfos) {
                //servletInfo依次表示每一个servlet标签对象
```

```
            //获取到servlet下的servlet-name元素对象，并且获取标签体内容
            String servletName = servletInfo.element("servlet-name").getText();
            //获取到servlet下的servlet-class元素对象，并且获取标签体内容
            String servletClass = servletInfo.element("servlet-class").getText();

            servletInfoHashMap.put(servletName,servletClass);
        }

        //--------------------servlet-mapping-------------------------------------
        //获取到所有的servlet-mapping标签
        List<Element> servletMappingInfos = rootElement.elements("servlet-mapping");
        //遍历集合依次得到每一个servlet-mapping标签
        for (Element servletMappingInfo : servletMappingInfos) {
            //servletMappingInfo依次表示每一个servlet-mapping标签

            //获取servlet-mapping标签标签中的servlet-name标签的标签体内容
            String servletName = servletMappingInfo.element("servlet-name").getText();

            //获取servlet-mapping标签标签中的url-pattern标签的标签体内容
            String urlPattern = servletMappingInfo.element("url-pattern").getText();

            //通过servletName来获取到servlet的全类名
            String servletClassName = servletInfoHashMap.get(servletName);

            //通过反射来创建这个servlet对象
            Class clazz = Class.forName(servletClassName);

            //获取该类所实现的所有的接口信息,得到的是一个数组
            Class[] interfaces = clazz.getInterfaces();

            //定义一个boolean类型的变量
            boolean flag =  false;
            //遍历数组
            for (Class clazzInfo : interfaces) {
                //判断当前所遍历的接口的字节码对象是否和HttpServlet的字节码文件对象相同
                if(clazzInfo == HttpServlet.class){

                    //如果相同,就需要更改flag值.结束循环
                    flag = true;
                    break;
                }
            }

            if(flag){
                //true就表示当前的类已经实现了HttpServlet接口
                HttpServlet httpServlet = (HttpServlet) clazz.newInstance();
                //4.将uri和httpServlet添加到map集合中
                ServletConcurrentHashMap.map.put(urlPattern,httpServlet);
            }else{
                //false就表示当前的类还没有实现HttpServlet接口
                throw new NotImplementsHttpServletException(clazz.getName() + "Not
Implements HttpServlet");

            }
```

```java
            }
        } catch (NotImplementsHttpServletException e) {
            e.printStackTrace();
        }catch (Exception e) {
            e.printStackTrace();
        }
    }
}

public class LoaderResourceRunnable implements  Runnable {
    @Override
    public void run() {
//        //执行parse方法
//        ParseServletConfig parseServletConfig = new PropertiesParseServletConfig();
//        parseServletConfig.parse();

        ParseServletConfig parseServletConfig = new XMLParseServletConfig();
        parseServletConfig.parse();

    }
}
```

# 2.枚举

## 2.1概述【理解】

为了间接的表示一些固定的值，Java就给我们提供了枚举 是指将变量的值一一列出来,变量的值只限于列举出来的值的范围内

## 2.2定义格式【应用】

- 格式

```java
public enum s {
    枚举项1,枚举项2,枚举项3;
}
注意: 定义枚举类要用关键字enum
```

- 示例代码

```java
// 定义一个枚举类，用来表示春，夏，秋，冬这四个固定值
public enum Season {
    SPRING,SUMMER,AUTUMN,WINTER;
}
```

## 2.3枚举的特点【理解】

- 特点
    - 所有枚举类都是Enum的子类
    - 我们可以通过"枚举类名.枚举项名称"去访问指定的枚举项

  - 每一个枚举项其实就是该枚举的一个对象
  - 枚举也是一个类，也可以去定义成员变量
  - 枚举类的第一行上必须是枚举项，最后一个枚举项后的分号是可以省略的，但是如果枚举类有其他的东西，这个分号就不能省略。建议不要省略
  - 枚举类可以有构造器，但必须是private的，它默认的也是private的。

    枚举项的用法比较特殊：枚举("");
  - 枚举类也可以有抽象方法，但是枚举项必须重写该方法
- 示例代码

```
public enum Season {

    SPRING("春"){

        //如果枚举类中有抽象方法
        //那么在枚举项中必须要全部重写
        @Override
        public void show() {
            System.out.println(this.name);
        }

    },

    SUMMER("夏"){
        @Override
        public void show() {
            System.out.println(this.name);
        }
    },

    AUTUMN("秋"){
        @Override
        public void show() {
            System.out.println(this.name);
        }
    },

    WINTER("冬"){
        @Override
        public void show() {
            System.out.println(this.name);
        }
    };


    public String name;


    //空参构造
    //private Season(){}
```

```
    //有参构造
    private Season(String name){
        this.name = name;
    }

    //抽象方法
    public abstract void show();
}

public class EnumDemo {
    public static void main(String[] args) {
        /*
        1.所有枚举类都是Enum的子类
        2.我们可以通过"枚举类名.枚举项名称"去访问指定的枚举项
        3.每一个枚举项其实就是该枚举的一个对象
        4.枚举也是一个类，也可以去定义成员变量
        5.枚举类的第一行上必须是枚举项，最后一个枚举项后的分号是可以省略的，
          但是如果枚举类有其他的东西，这个分号就不能省略。建议不要省略
        6.枚举类可以有构造器，但必须是private的，它默认的也是private的。
          枚举项的用法比较特殊：枚举("");
        7.枚举类也可以有抽象方法，但是枚举项必须重写该方法
    */

        //第二个特点的演示
        //我们可以通过"枚举类名.枚举项名称"去访问指定的枚举项
        System.out.println(Season.SPRING);
        System.out.println(Season.SUMMER);
        System.out.println(Season.AUTUMN);
        System.out.println(Season.WINTER);

        //第三个特点的演示
        //每一个枚举项其实就是该枚举的一个对象
        Season spring = Season.SPRING;
    }
}
```

## 2.4枚举的方法【应用】

- 方法介绍

| 方法名 | 说明 |
| --- | --- |
| String name() | 获取枚举项的名称 |
| int ordinal() | 返回枚举项在枚举类中的索引值 |
| int compareTo(E o) | 比较两个枚举项，返回的是索引值的差值 |
| String toString() | 返回枚举常量的名称 |
| static T valueOf(Class type,String name) | 获取指定枚举类中的指定名称的枚举值 |
| values() | 获得所有的枚举项 |

- 示例代码

```java
public enum Season {
    SPRING,SUMMER,AUTUMN,WINTER;
}

public class EnumDemo {
    public static void main(String[] args) {
//        String name() 获取枚举项的名称
        String name = Season.SPRING.name();
        System.out.println(name);
        System.out.println("-----------------------------");

//        int ordinal() 返回枚举项在枚举类中的索引值
        int index1 = Season.SPRING.ordinal();
        int index2 = Season.SUMMER.ordinal();
        int index3 = Season.AUTUMN.ordinal();
        int index4 = Season.WINTER.ordinal();
        System.out.println(index1);
        System.out.println(index2);
        System.out.println(index3);
        System.out.println(index4);
        System.out.println("-----------------------------");

//        int compareTo(E o) 比较两个枚举项，返回的是索引值的差值
        int result = Season.SPRING.compareTo(Season.WINTER);
        System.out.println(result);//-3
        System.out.println("-----------------------------");

//        String toString()   返回枚举常量的名称
        String s = Season.SPRING.toString();
        System.out.println(s);
        System.out.println("-----------------------------");

//        static <T> T valueOf(Class<T> type,String name)
//        获取指定枚举类中的指定名称的枚举值
        Season spring = Enum.valueOf(Season.class, "SPRING");
        System.out.println(spring);
        System.out.println(Season.SPRING == spring);
        System.out.println("-----------------------------");

//        values()       获得所有的枚举项
        Season[] values = Season.values();
        for (Season value : values) {
            System.out.println(value);
        }


    }
}
```

## 3.注解

## 3.1概述【理解】

- 概述

  对我们的程序进行标注和解释

- 注解和注释的区别

  - 注释: 给程序员看的
  - 注解: 给编译器看的

- 使用注解进行配置配置的优势

  代码更加简洁,方便

## 3.2自定义注解【理解】

- 格式

  public @interface 注解名称 {

  public 属性类型 属性名() default 默认值 ;

  }

- 属性类型

  - 基本数据类型
  - String
  - Class
  - 注解
  - 枚举
  - 以上类型的一维数组

- 代码演示

```java
public @interface Anno2 {
}

public enum Season {
    SPRING,SUMMER,AUTUMN,WINTER;
}

public @interface Anno1 {

    //定义一个基本类型的属性
    int a () default 23;

    //定义一个String类型的属性
    public String name() default "itheima";

    //定义一个Class类型的属性
    public Class clazz() default Anno2.class;

    //定义一个注解类型的属性
    public Anno2 anno() default @Anno2;


    //定义一个枚举类型的属性
```

```java
    public Season season() default Season.SPRING;

    //以上类型的一维数组
    //int数组
    public int[] arr() default {1,2,3,4,5};

    //枚举数组
    public Season[] seasons() default {Season.SPRING,Season.SUMMER};


    //value。后期我们在使用注解的时候，如果我们只需要给注解的value属性赋值。
    //那么value就可以省略
    public String value();

}

//在使用注解的时候如果注解里面的属性没有指定默认值。
//那么我们就需要手动给出注解属性的设置值。
//@Anno1(name = "itheima")
@Anno1("abc")
public class AnnoDemo {
}
```

- 注意

  如果只有一个属性需要赋值，并且属性的名称是value，则value可以省略，直接定义值即可

- 自定义注解案例

  - 需求

    自定义一个注解@Test,用于指定类的方法上,如果某一个类的方法上使用了该注解,就执行该方法

  - 实现步骤

    1. 自定义一个注解Test,并在类中的某几个方法上加上注解
    2. 在测试类中,获取注解所在的类的Class对象
    3. 获取类中所有的方法对象
    4. 遍历每一个方法对象,判断是否有对应的注解

  - 代码实现

    ```java
    //表示Test这个注解的存活时间
    @Retention(value = RetentionPolicy.RUNTIME)
    public @interface Test {
    }

    public class UseTest {

        //没有使用Test注解
        public void show(){
            System.out.println("UseTest....show....");
        }

        //使用Test注解

        @Test
    ```

```java
    public void method(){
        System.out.println("UseTest....method....");
    }


    //没有使用Test注解
    @Test
    public void function(){
        System.out.println("UseTest....function....");
    }
}

public class AnnoDemo {
    public static void main(String[] args) throws ClassNotFoundException,
IllegalAccessException, InstantiationException, InvocationTargetException {
        //1.通过反射获取UseTest类的字节码文件对象
        Class clazz = Class.forName("com.itheima.myanno3.UseTest");

        //创建对象
        UseTest useTest = (UseTest) clazz.newInstance();


        //2.通过反射获取这个类里面所有的方法对象
        Method[] methods = clazz.getDeclaredMethods();


        //3.遍历数组，得到每一个方法对象
        for (Method method : methods) {
            //method依次表示每一个方法对象。
            //isAnnotationPresent(Class<? extends Annotation> annotationClass)
            //判断当前方法上是否有指定的注解。
            //参数：注解的字节码文件对象
            //返回值：布尔结果。  true 存在  false 不存在
            if(method.isAnnotationPresent(Test.class)){
                method.invoke(useTest);
            }
        }
    }
}
```

## 3.3元注解【理解】

- 概述

  元注解就是描述注解的注解

- 元注解介绍

| 元注解名 | 说明 |
| --- | --- |
| @Target | 指定了注解能在哪里使用 |
| @Retention | 可以理解为保留时间(生命周期) |
| @Inherited | 表示修饰的自定义注解可以被子类继承 |
| @Documented | 表示该自定义注解，会出现在API文档里面。 |

- 示例代码

```
@Target({ElementType.FIELD,ElementType.TYPE,ElementType.METHOD})  //指定注解使用的位置（成员变量，类，方法）
@Retention(RetentionPolicy.RUNTIME) //指定该注解的存活时间
//@Inherited //指定该注解可以被继承
public @interface Anno {
}

@Anno
public class Person {
}

public class Student extends Person {
    public void show(){
        System.out.println("student.......show.........");
    }
}

public class StudentDemo {
    public static void main(String[] args) throws ClassNotFoundException {
        //获取到Student类的字节码文件对象
        Class clazz = Class.forName("com.itheima.myanno4.Student");

        //获取注解。
        boolean result = clazz.isAnnotationPresent(Anno.class);
        System.out.println(result);
    }
}
```

## 3.4改写服务器【理解】

- 需求

  目前项目中Servlet和url对应关系,是配置在xml文件中的,将其改为在Servlet类上通过注解配置实现

- 实现步骤

  1. 定义一个注解(@WebServlet),注解内有一个属性urlPatterns
  2. 在servlet类上去使用该注解,来指定当前Servlet的访问路径
  3. 创建一个注解解析类(AnnoParseServletConfig),该类实现ParseServletConfig接口

4. 实现parse方法

- 代码实现

```
@Target(ElementType.TYPE) //指定该注解可以使用在类上
@Retention(RetentionPolicy.RUNTIME)//指定该注解的存活时间 --- 为运行期
public @interface WebServlet {

    //让用户去指定某一个Servlet在进行访问的时候所对应的请求uri
    public String urlPatterns();
}

// 这里只给出了LoginServlet的配置,其他Servlet同理
@WebServlet(urlPatterns = "/servlet/loginservlet")
public class LoginServlet implements HttpServlet{
    @Override
    public void service(HttpRequest httpRequest, HttpResponse httpResponse) {
       //处理
        System.out.println("LoginServlet处理了登录请求");


       //响应
        httpResponse.setContentTpye("text/html;charset=UTF-8");
        httpResponse.write("登录成功");
    }
}

public class AnnoParseServletConfig implements ParseServletConfig {

    //定义一个servlet路径所对应的常量
    public static final String SERVLET_PATH = "http-dynamic-
server\\src\\com\\itheima\\myservlet";

    //定义包名
    public static final String SERVLET_PACKAGE_NAME = "com.itheima.myservlet.";

    @Override
    public void parse() {
        //获取类名
//   1.获得servlet所在文件夹的路径，并封装成File对象
        File file = new File(SERVLET_PATH);
//   2.调用listFiles方法，获取文件夹下所有的File对象
        File[] servletFiles = file.listFiles();
//   3.遍历数组，获取每一个File对象
        for (File servletFile : servletFiles) {
//   4.获取File对象的名字（后缀名）
            String servletFileName = servletFile.getName().replace(".java", "");
//   5.根据包名 + 类名 得到每一个类的全类名
           String servletFullName = SERVLET_PACKAGE_NAME + servletFileName;
            try {
//   6.通过全类名获取字节码文件对象
                Class servletClazz = Class.forName(servletFullName);
                //   7.判断该类是否有WebServlet注解
                if(servletClazz.isAnnotationPresent(WebServlet.class)){
```

```
                    //  8.判断该Servlet类是否实现HttpServlet接口
                        //获取该类所实现的所有的接口信息,得到的是一个数组
                        Class[] interfaces = servletClazz.getInterfaces();

                        //定义一个boolean类型的变量
                        boolean flag =  false;
                        //遍历数组
                        for (Class clazzInfo : interfaces) {
                            //判断当前所遍历的接口的字节码对象是否和HttpServlet的字节码文件对象相同
                            if(clazzInfo == HttpServlet.class){
                                //如果相同,就需要更改flag值.结束循环
                                flag = true;
                                break;
                            }
                        }

                        if(flag){
                            //  9.如果满足, 则获取注解中的urlPattrens的值,
                            WebServlet annotation = (WebServlet)
servletClazz.getAnnotation(WebServlet.class);
                            String uri = annotation.urlPatterns();

                            //  10.创建当前Servlet类对象存入值位置
                            HttpServlet httpServlet = (HttpServlet) servletClazz.newInstance();
                            //  11.存入集合的键位置
                            ServletConcurrentHashMap.map.put(uri,httpServlet);
                            //
                        }else{
                            //  12.如果不满足, 抛出异常
                            //false就表示当前的类还没有实现HttpServlet接口
                            throw new NotImplementsHttpServletException(servletClazz.getName()
+ "Not Implements HttpServlet");
                        }
                    }
                } catch (NotImplementsHttpServletException e) {
                    e.printStackTrace();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }


        }
    }

public class LoaderResourceRunnable implements  Runnable {
    @Override
    public void run() {
//        //执行parse方法
//        ParseServletConfig parseServletConfig = new PropertiesParseServletConfig();
//        parseServletConfig.parse();


//        ParseServletConfig parseServletConfig = new XMLParseServletConfig();
```

```
//        parseServletConfig.parse();


        ParseServletConfig parseServletConfig = new AnnoParseServletConfig();
        parseServletConfig.parse();

    }
}
```