### XML解析方式？

Dom4j、Sax、Pull

### Dom4j与Sax区别

 dom4j不适合大文件的解析，因为它是一下子将文件加载到内存中，所以有可能出现内存溢出，sax是基于事件来对xml进行解析的，所以他可以解析大文件的xml，也正是因为如此，所以dom4j可以对xml进行灵活的增删改查和导航，而sax没有这么强的灵活性，所以sax经常是用来解析大型xml文件，而要对xml文件进行一些灵活（crud）操作就用dom4j。

### 使用dom4j解析xml

解析XML过程是通过获取Document对象，然后继续获取各个节点以及属性等操作，因此获取Document对象是第一步，大体说来，有三种方式：

1.自己创建Document对象

|  |
| --- |
| **Document document = DocumentHelper.*createDocument*();**  **Element root = document.addElement("students");** |

其中students是根节点，可以继续添加其他节点等操作。

2.自己创建Document对象

|  |
| --- |
| **// 创建SAXReader对象**  **SAXReader reader = new SAXReader();**  **// 读取文件 转换成Document**  **Document document = reader.read(new File("XXXX.xml"));** |

3.读取XML文本内容获取Document对象

|  |
| --- |
| **String xmlStr = "<students>......</students>";**  **Document document = DocumentHelper.*parseText*(xmlStr);** |

### 解析xml代码

Xml配置:

|  |
| --- |
| **<?xml version="1.0" encoding="UTF-8"?>**  **<students>**  **<student1 id="001">**  **<微信公众号>每特学院</微信公众号>**  **<学号>20140101</学号>**  **<地址>北京海淀区</地址>**  **<座右铭>要么强大，要么听话</座右铭>**  **</student1>**  **<student2 id="002">**  **<新浪微博>蚂蚁课堂</新浪微博>**  **<学号>20140102</学号>**  **<地址>北京朝阳区</地址>**  **<座右铭>在哭泣中学会坚强</座右铭>**  **</student2>**  **</students>** |

Java代码

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** SAXException, DocumentException {  XmlUtils xmlUtils = **new** XmlUtils();  xmlUtils.test001();  }  **public** **void** test001() **throws** DocumentException {  SAXReader saxReader = **new** SAXReader();  Document read = saxReader.read(getClassPath("student.xml"));  // 获取根节点  Element rootElement = read.getRootElement();  *getNodes*(rootElement);  }  **public** InputStream getClassPath(String xmlPath) {  InputStream resourceAsStream = getClass().getClassLoader().getResourceAsStream(xmlPath);  **return** resourceAsStream;  }  **public** **static** **void** getNodes(Element rootElement) {  System.***out***.println("获取当前名称:" + rootElement.getName());  // 获取属性信息  List<Attribute> attributes = rootElement.attributes();  **for** (Attribute attribute : attributes) {  System.***out***.println("属性:" + attribute.getName() + "---" + attribute.getText());  }  // 获取属性value  String value = rootElement.getTextTrim();  **if** (!StringUtils.*isEmpty*(value)) {  System.***out***.println("value:" + value);  }  // 使用迭代器遍历,继续遍历子节点  Iterator<Element> elementIterator = rootElement.elementIterator();  **while** (elementIterator.hasNext()) {  Element next = elementIterator.next();  *getNodes*(next);  } |

注意:

this.getClass().getClassLoader().getResourceAsStream(xmlPath) 获取当前项目路径xmlfsfs

## XML与JSON区别

Xml是重量级数据交换格式，占宽带比较大。

JSON是轻量级交换格式，xml占宽带小。

所有很多互联网公司都会使用json作为数据交换格式

很多银行项目，大多数还是在使用xml。

## 什么是SpringIOC

spring ioc指的是控制反转，IOC容器负责实例化、定位、配置应用程序中的对象及建立这些对象间的依赖。交由Spring来管理这些，实现解耦

## SpringIOC原理

使用反射机制+XML技术

### 手写SpringIOCXML版本

|  |
| --- |
| /\*\*  \* 手写Spring专题 XML方式注入bean  \*  \* **@author** 作者:余胜军  \*  \*/  **public** **class** ClassPathXmlApplicationContext {  // xml路径地址  **private** String xmlPath;  **public** ClassPathXmlApplicationContext(String xmlPath) {  **this**.xmlPath = xmlPath;  }  **public** Object getBean(String beanId) **throws** Exception {  // 1. 读取配置文件  List<Element> elements = readerXml();  **if** (elements == **null**) {  **throw** **new** Exception("该配置文件没有子元素");  }  // 2. 使用beanId查找对应的class地址  String beanClass = findXmlByIDClass(elements, beanId);  **if** (StringUtils.*isEmpty*(beanClass)) {  **throw** **new** Exception("未找到对应的class地址");  }  // 3. 使用反射机制初始化，对象  Class<?> forName = Class.*forName*(beanClass);  **return** forName.newInstance();  }  // 读取配置文件信息  **public** List<Element> readerXml() **throws** DocumentException {  SAXReader saxReader = **new** SAXReader();  **if** (StringUtils.*isEmpty*(xmlPath)) {  **new** Exception("xml路径为空...");  }  Document read = saxReader.read(getClassXmlInputStream(xmlPath));  // 获取根节点信息  Element rootElement = read.getRootElement();  // 获取子节点  List<Element> elements = rootElement.elements();  **if** (elements == **null** || elements.isEmpty()) {  **return** **null**;  }  **return** elements;  }  // 使用beanid查找该Class地址  **public** String findXmlByIDClass(List<Element> elements, String beanId) **throws** Exception {  **for** (Element element : elements) {  // 读取节点上是否有value  String beanIdValue = element.attributeValue("id");  **if** (beanIdValue == **null**) {  **throw** **new** Exception("使用该beanId为查找到元素");  }  **if** (!beanIdValue.equals(beanId)) {  **continue**;  }  // 获取Class地址属性  String classPath = element.attributeValue("class");  **if** (!StringUtils.*isEmpty*(classPath)) {  **return** classPath;  }  }  **return** **null**;  }  // 读取xml配置文件  **public** InputStream getClassXmlInputStream(String xmlPath) {  InputStream resourceAsStream = getClass().getClassLoader().getResourceAsStream(xmlPath);  **return** resourceAsStream;  }  } |

### 手写SpringIOC注解版本

|  |
| --- |
| /\*\*  \* 手写Spring专题 注解版本注入bean  \*  \* **@author** 作者:余胜军  \*  \*/  @SuppressWarnings({ "rawtypes", "unchecked" })  **public** **class** ClassPathXmlApplicationContext {  // 扫包范围  **private** String packageName;  ConcurrentHashMap<String, Object> initBean = **null**;  **public** ClassPathXmlApplicationContext(String packageName) {  **this**.packageName = packageName;  }  // 使用beanID查找对象  **public** Object getBean(String beanId) **throws** Exception {  // 1.使用反射机制获取该包下所有的类已经存在bean的注解类  List<Class> listClassesAnnotation = findClassExisService();  **if** (listClassesAnnotation == **null** || listClassesAnnotation.isEmpty()) {  **throw** **new** Exception("没有需要初始化的bean");  }  // 2.使用Java反射机制初始化对象  initBean = initBean(listClassesAnnotation);  **if** (initBean == **null** || initBean.isEmpty()) {  **throw** **new** Exception("初始化bean为空!");  }  // 3.使用beanID查找查找对应bean对象  Object object = initBean.get(beanId);  // 4.使用反射读取类的属性,赋值信息  attriAssign(object);  **return** object;  }  // 使用反射读取类的属性,赋值信息  **public** **void** attriAssign(Object object) **throws** IllegalArgumentException, IllegalAccessException {  // 1.获取类的属性是否存在 获取bean注解  Class<? **extends** Object> classInfo = object.getClass();  Field[] declaredFields = classInfo.getDeclaredFields();  **for** (Field field : declaredFields) {  // 属性名称  String name = field.getName();  // 2.使用属性名称查找bean容器赋值  Object bean = initBean.get(name);  **if** (bean != **null**) {  // 私有访问允许访问  field.setAccessible(**true**);  // 给属性赋值  field.set(object, bean);  **continue**;  }  }  }  // 使用反射机制获取该包下所有的类已经存在bean的注解类  **public** List<Class> findClassExisService() **throws** Exception {  // 1.使用反射机制获取该包下所有的类  **if** (StringUtils.*isEmpty*(packageName)) {  **throw** **new** Exception("扫包地址不能为空!");  }  // 2.使用反射技术获取当前包下所有的类  List<Class<?>> classesByPackageName = ClassUtil.*getClasses*(packageName);  // 3.存放类上有bean注入注解  List<Class> exisClassesAnnotation = **new** ArrayList<Class>();  // 4.判断该类上属否存在注解  **for** (Class classInfo : classesByPackageName) {  ExtService extService = (ExtService) classInfo.getDeclaredAnnotation(ExtService.**class**);  **if** (extService != **null**) {  exisClassesAnnotation.add(classInfo);  **continue**;  }  }  **return** exisClassesAnnotation;  }  // 初始化bean对象  **public** ConcurrentHashMap<String, Object> initBean(List<Class> listClassesAnnotation)  **throws** InstantiationException, IllegalAccessException {  ConcurrentHashMap concurrentHashMap = **new** ConcurrentHashMap<String, Object>();  **for** (Class classInfo : listClassesAnnotation) {  // 初始化对象  Object newInstance = classInfo.newInstance();  // 获取父类名称  String beanId = *toLowerCaseFirstOne*(classInfo.getSimpleName());  concurrentHashMap.put(beanId, newInstance);  }  **return** concurrentHashMap;  }  // 首字母转小写  **public** **static** String toLowerCaseFirstOne(String s) {  **if** (Character.*isLowerCase*(s.charAt(0)))  **return** s;  **else**  **return** (**new** StringBuilder()).append(Character.*toLowerCase*(s.charAt(0))).append(s.substring(1)).toString();  }  } |

### 常用反射工具类

|  |
| --- |
| **public** **class** ClassUtil {  /\*\*  \* 取得某个接口下所有实现这个接口的类  \*/  **public** **static** List<Class> getAllClassByInterface(Class c) {  List<Class> returnClassList = **null**;  **if** (c.isInterface()) {  // 获取当前的包名  String packageName = c.getPackage().getName();  // 获取当前包下以及子包下所以的类  List<Class<?>> allClass = *getClasses*(packageName);  **if** (allClass != **null**) {  returnClassList = **new** ArrayList<Class>();  **for** (Class classes : allClass) {  // 判断是否是同一个接口  **if** (c.isAssignableFrom(classes)) {  // 本身不加入进去  **if** (!c.equals(classes)) {  returnClassList.add(classes);  }  }  }  }  }  **return** returnClassList;  }  /\*  \* 取得某一类所在包的所有类名 不含迭代  \*/  **public** **static** String[] getPackageAllClassName(String classLocation, String packageName) {  // 将packageName分解  String[] packagePathSplit = packageName.split("[.]");  String realClassLocation = classLocation;  **int** packageLength = packagePathSplit.length;  **for** (**int** i = 0; i < packageLength; i++) {  realClassLocation = realClassLocation + File.***separator*** + packagePathSplit[i];  }  File packeageDir = **new** File(realClassLocation);  **if** (packeageDir.isDirectory()) {  String[] allClassName = packeageDir.list();  **return** allClassName;  }  **return** **null**;  }  /\*\*  \* 从包package中获取所有的Class  \*  \* **@param** pack  \* **@return**  \*/  **public** **static** List<Class<?>> getClasses(String packageName) {  // 第一个class类的集合  List<Class<?>> classes = **new** ArrayList<Class<?>>();  // 是否循环迭代  **boolean** recursive = **true**;  // 获取包的名字 并进行替换  String packageDirName = packageName.replace('.', '/');  // 定义一个枚举的集合 并进行循环来处理这个目录下的things  Enumeration<URL> dirs;  **try** {  dirs = Thread.*currentThread*().getContextClassLoader().getResources(packageDirName);  // 循环迭代下去  **while** (dirs.hasMoreElements()) {  // 获取下一个元素  URL url = dirs.nextElement();  // 得到协议的名称  String protocol = url.getProtocol();  // 如果是以文件的形式保存在服务器上  **if** ("file".equals(protocol)) {  // 获取包的物理路径  String filePath = URLDecoder.*decode*(url.getFile(), "UTF-8");  // 以文件的方式扫描整个包下的文件 并添加到集合中  *findAndAddClassesInPackageByFile*(packageName, filePath, recursive, classes);  } **else** **if** ("jar".equals(protocol)) {  // 如果是jar包文件  // 定义一个JarFile  JarFile jar;  **try** {  // 获取jar  jar = ((JarURLConnection) url.openConnection()).getJarFile();  // 从此jar包 得到一个枚举类  Enumeration<JarEntry> entries = jar.entries();  // 同样的进行循环迭代  **while** (entries.hasMoreElements()) {  // 获取jar里的一个实体 可以是目录 和一些jar包里的其他文件 如META-INF等文件  JarEntry entry = entries.nextElement();  String name = entry.getName();  // 如果是以/开头的  **if** (name.charAt(0) == '/') {  // 获取后面的字符串  name = name.substring(1);  }  // 如果前半部分和定义的包名相同  **if** (name.startsWith(packageDirName)) {  **int** idx = name.lastIndexOf('/');  // 如果以"/"结尾 是一个包  **if** (idx != -1) {  // 获取包名 把"/"替换成"."  packageName = name.substring(0, idx).replace('/', '.');  }  // 如果可以迭代下去 并且是一个包  **if** ((idx != -1) || recursive) {  // 如果是一个.class文件 而且不是目录  **if** (name.endsWith(".class") && !entry.isDirectory()) {  // 去掉后面的".class" 获取真正的类名  String className = name.substring(packageName.length() + 1, name.length() - 6);  **try** {  // 添加到classes  classes.add(Class.*forName*(packageName + '.' + className));  } **catch** (ClassNotFoundException e) {  e.printStackTrace();  }  }  }  }  }  } **catch** (IOException e) {  e.printStackTrace();  }  }  }  } **catch** (IOException e) {  e.printStackTrace();  }  **return** classes;  }  /\*\*  \* 以文件的形式来获取包下的所有Class  \*  \* **@param** packageName  \* **@param** packagePath  \* **@param** recursive  \* **@param** classes  \*/  **public** **static** **void** findAndAddClassesInPackageByFile(String packageName, String packagePath, **final** **boolean** recursive,  List<Class<?>> classes) {  // 获取此包的目录 建立一个File  File dir = **new** File(packagePath);  // 如果不存在或者 也不是目录就直接返回  **if** (!dir.exists() || !dir.isDirectory()) {  **return**;  }  // 如果存在 就获取包下的所有文件 包括目录  File[] dirfiles = dir.listFiles(**new** FileFilter() {  // 自定义过滤规则 如果可以循环(包含子目录) 或则是以.class结尾的文件(编译好的java类文件)  **public** **boolean** accept(File file) {  **return** (recursive && file.isDirectory()) || (file.getName().endsWith(".class"));  }  });  // 循环所有文件  **for** (File file : dirfiles) {  // 如果是目录 则继续扫描  **if** (file.isDirectory()) {  *findAndAddClassesInPackageByFile*(packageName + "." + file.getName(), file.getAbsolutePath(), recursive,  classes);  } **else** {  // 如果是java类文件 去掉后面的.class 只留下类名  String className = file.getName().substring(0, file.getName().length() - 6);  **try** {  // 添加到集合中去  classes.add(Class.*forName*(packageName + '.' + className));  } **catch** (ClassNotFoundException e) {  e.printStackTrace();  }  }  }  }  } |

### Maven相关依赖

|  |
| --- |
| <dependencies>  <!-- 引入Spring-AOP等相关Jar -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-core</artifactId>  <version>3.0.6.RELEASE</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context</artifactId>  <version>3.0.6.RELEASE</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-aop</artifactId>  <version>3.0.6.RELEASE</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-orm</artifactId>  <version>3.0.6.RELEASE</version>  </dependency>  <dependency>  <groupId>org.aspectj</groupId>  <artifactId>aspectjrt</artifactId>  <version>1.6.1</version>  </dependency>  <dependency>  <groupId>aspectj</groupId>  <artifactId>aspectjweaver</artifactId>  <version>1.5.3</version>  </dependency>  <dependency>  <groupId>cglib</groupId>  <artifactId>cglib</artifactId>  <version>2.1\_2</version>  </dependency>  <!-- https://mvnrepository.com/artifact/com.mchange/c3p0 -->  <dependency>  <groupId>com.mchange</groupId>  <artifactId>c3p0</artifactId>  <version>0.9.5.2</version>  </dependency>  <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>5.1.37</version>  </dependency>  <!-- https://mvnrepository.com/artifact/dom4j/dom4j -->  <dependency>  <groupId>dom4j</groupId>  <artifactId>dom4j</artifactId>  <version>1.6.1</version>  </dependency>  <!-- https://mvnrepository.com/artifact/commons-lang/commons-lang -->  <dependency>  <groupId>commons-lang</groupId>  <artifactId>commons-lang</artifactId>  <version>2.6</version>  </dependency>  </dependencies> |