链滴

# 自己动手实现一个简单的 IOC

作者：stateis0

原文链接：https://ld246.com/article/1512475912190

来源网站：链滴

再上一篇文章中，楼主和大家一起分析spring的 IOC 实现，剖析了Spring的源码，看的出来，源码常复杂，这是因为Spring的设计者需要考虑到框架的扩展性，健壮性，性能等待元素，因此设计的很杂。楼主在最后也说要实现一个简单的 IOC，让我们更加深刻的理解IOC，因此，有了这篇文章。

当然我们是仿照Spring 的 IOC，因此代码命名和设计基本是仿照spring的。

我们将分为几步来编写简易 IOC，首先设计组件，再设计接口，然后关注实现。

# 1. 设计组件。

我们还记得Spring中最重要的有哪些组件吗？BeanFactory 容器，BeanDefinition Bean的基本数据构，当然还需要加载Bean的资源加载器。大概最后最重要的就是这几个组件。容器用来存放初始化好Bean，BeanDefinition 就是Bean的基本数据结构，比如Bean的名称，Bean的属性 PropertyValue Bean的方法，是否延迟加载，依赖关系等。资源加载器就简单了，就是一个读取XML配置文件的类读取每个标签并解析。

# 2. 设计接口

首先肯定需要一个BeanFactory，就是Bean容器，容器接口至少有2个最简单的方法，一个是获取Bea，一个注册Bean.

```
/**
 * 需要一个beanFactory 定义ioc 容器的一些行为 比如根据名称获取bean， 比如注册bean，参数为ean的名称，bean的定义
 *
 * @author stateis0
 * @version 1.0.0
 * @Date 2017/11/30
 */
public interface BeanFactory {

  /**
   * 根据bean的名称从容器中获取bean对象
   *
   * @param name bean 名称
   * @return bean实例
   * @throws Exception 异常
   */
  Object getBean(String name) throws Exception;

  /**
   * 将bean注册到容器中
   *
   * @param name bean 名称
   * @param bean bean实例
   * @throws Exception 异常
   */
  void registerBeanDefinition(String name, BeanDefinition bean) throws Exception;
}
```

根据Bean的名字获取Bean对象，注册参数有2个，一个是Bean的名字，一个是 BeanDefinition 对象。

定义完了Bean最基本的容器，还需要一个最简单 BeanDefinition 接口，我们为了方便，但因为我们个不必考虑扩展，因此可以直接设计为类，BeanDefinition 需要哪些元素和方法呢？ 需要一个 Bean 对象，一个Class对象，一个ClassName字符串，还需要一个元素集合 PropertyValues。这些就能组一个最基本的 BeanDefinition 类了。那么需要哪些方法呢？其实就是这些属性的get set 方法。 我看看该类的详细：

```
package cn.thinkinjava.myspring;

/**
 * bean 的定义
 *
 * @author stateis0
 */
public class BeanDefinition {

  /**
   * bean
   */
  private Object bean;

  /**
   * bean 的 CLass 对象
   */
  private Class beanClass;

  /**
   * bean 的类全限定名称
   */
  private String ClassName;

  /**
   * 类的属性集合
   */
  private PropertyValues propertyValues = new PropertyValues();

  /**
   * 获取bean对象
   */
  public Object getBean() {
    return this.bean;
  }

  /**
   * 设置bean的对象
   */
  public void setBean(Object bean) {
    this.bean = bean;
  }

  /**
   * 获取bean的Class对象
   */
  public Class getBeanclass() {
    return this.beanClass;
```

```
  }

  /**
   * 通过设置类名称反射生成Class对象
   */
  public void setClassname(String name) {
    this.ClassName = name;
    try {
      this.beanClass = Class.forName(name);
    } catch (ClassNotFoundException e) {
      e.printStackTrace();
    }
  }

  /**
   * 获取bean的属性集合
   */
  public PropertyValues getPropertyValues() {
    return this.propertyValues;
  }

  /**
   * 设置bean的属性
   */
  public void setPropertyValues(PropertyValues pv) {
    this.propertyValues = pv;
  }

}
```

有了基本的 BeanDefinition 数据结构，还需要一个从XML中读取并解析为 BeanDefinition 的操作，首先我们定义一个 BeanDefinitionReader 接口，该接口只是一个标识，具体由抽象类去实现一个本方法和定义一些基本属性，比如一个读取时需要存放的注册容器，还需要一个委托一个资源加载器esourceLoader， 用于加载XML文件，并且我们需要设置该构造器必须含有资源加载器，当然还有些get set 方法。

```
package cn.thinkinjava.myspring;

import cn.thinkinjava.myspring.io.ResourceLoader;
import java.util.HashMap;
import java.util.Map;

/**
 * 抽象的bean定义读取类
 *
 * @author stateis0
 */
public abstract class AbstractBeanDefinitionReader implements BeanDefinitionReader {

  /**
   * 注册bean容器
   */
  private Map<String, BeanDefinition> registry;
```

```
    /**
     * 资源加载器
     */
    private ResourceLoader resourceLoader;

    /**
     * 构造器器必须有一个资源加载器， 默认插件创建一个map容器
     *
     * @param resourceLoader 资源加载器
     */
    protected AbstractBeanDefinitionReader(ResourceLoader resourceLoader) {
        this.registry = new HashMap<>();
        this.resourceLoader = resourceLoader;
    }

    /**
     * 获取容器
     */
    public Map<String, BeanDefinition> getRegistry() {
        return registry;
    }

    /**
     * 获取资源加载器
     */
    public ResourceLoader getResourceLoader() {
        return resourceLoader;
    }

}
```

有了这几个抽象类和接口，我们基本能形成一个雏形，BeanDefinitionReader 用于从XML中读取配文件，生成 BeanDefinition 实例，存放在 BeanFactory 容器中，初始化之后，就可以调用 getBean 方法获取初始化成功的Bean。形成一个完美的闭环。

# 3. 如何实现

刚刚我们说了具体的流程：从XML中读取配置文件， 解析成 BeanDefinition，最终放进容器。说白就3步。那么我们就先来设计第一步。

## 1. 从XML中读取配置文件， 解析成 BeanDefinition

我们刚刚设计了一个读取BeanDefinition 的接口 BeanDefinitionReader 和一个实现它的抽象类 AbstactBeanDefinitionReader，抽象了定义了一些简单的方法，其中由一个委托类-----ResourceLoade， 我们还没有创建， 该类是资源加载器，根据给定的路径来加载资源。我们可以使用Java 默认的类库 java.net.URL 来实现，定义两个类，一个是包装了URL的类 ResourceUrl， 一个是依赖 ResourceUrl 的资源加载类。

**ResourceUrl 代码实现**

```
/**
```

```
 * 资源URL
 */
public class ResourceUrl implements Resource {

  /**
   * 类库URL
   */
  private final URL url;

  /**
   * 需要一个类库URL
   */
  public ResourceUrl(URL url) {
    this.url = url;
  }

  /**
   * 从URL中获取输入流
   */
  @Override
  public InputStream getInputstream() throws Exception {
    URLConnection urlConnection = url.openConnection();
    urlConnection.connect();
    return urlConnection.getInputStream();

  }

}
```

**ResourceLoader 实现**

```
/**
 * 资源URL
 */
public class ResourceUrl implements Resource {

  /**
   * 类库URL
   */
  private final URL url;

  /**
   * 需要一个类库URL
   */
  public ResourceUrl(URL url) {
    this.url = url;
  }

  /**
   * 从URL中获取输入流
   */
  @Override
  public InputStream getInputstream() throws Exception {
```

```
    URLConnection urlConnection = url.openConnection();
    urlConnection.connect();
    return urlConnection.getInputStream();
  }
}
```

当然还需要一个接口，只定义了一个抽象方法

```
package cn.thinkinjava.myspring.io;

import java.io.InputStream;

/**
 * 资源定义
 *
 * @author stateis0
 */
public interface Resource {

  /**
   * 获取输入流
   */
  InputStream getInputstream() throws Exception;
}
```

好了， AbstractBeanDefinitionReader 需要的元素已经有了， 但是，很明显该方法不能实现读取 BeanDefinition 的任务。那么我们需要一个类去继承抽象类，去实现具体的方法， 既然我们是XML 配置件读取，那么我们就定义一个 XmlBeanDefinitionReader 继承 AbstractBeanDefinitionReader ，现一些我们需要的方法， 比如读取XML 的readrXML， 比如将解析出来的元素注册到 registry 的 Ma 中， 一些解析的细节。我们还是看代码吧。

**XmlBeanDefinitionReader 实现读取配置文件并解析成Bean**

```
package cn.thinkinjava.myspring.xml;

import cn.thinkinjava.myspring.AbstractBeanDefinitionReader;
import cn.thinkinjava.myspring.BeanDefinition;
import cn.thinkinjava.myspring.BeanReference;
import cn.thinkinjava.myspring.PropertyValue;
import cn.thinkinjava.myspring.io.ResourceLoader;
import java.io.InputStream;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

/**
 * 解析XML文件
 *
 * @author stateis0
 */
public class XmlBeanDefinitionReader extends AbstractBeanDefinitionReader {
```

```
/**
 * 构造器，必须包含一个资源加载器
 *
 * @param resourceLoader 资源加载器
 */
public XmlBeanDefinitionReader(ResourceLoader resourceLoader) {
  super(resourceLoader);
}

public void readerXML(String location) throws Exception {
  // 创建一个资源加载器
  ResourceLoader resourceloader = new ResourceLoader();
  // 从资源加载器中获取输入流
  InputStream inputstream = resourceloader.getResource(location).getInputstream();
  // 获取文档建造者工厂实例
  DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
  // 工厂创建文档建造者
  DocumentBuilder docBuilder = factory.newDocumentBuilder();
  // 文档建造者解析流 返回文档对象
  Document doc = docBuilder.parse(inputstream);
  // 根据给定的文档对象进行解析，并注册到bean容器中
  registerBeanDefinitions(doc);
  // 关闭流
  inputstream.close();
}

/**
 * 根据给定的文档对象进行解析，并注册到bean容器中
 *
 * @param doc 文档对象
 */
private void registerBeanDefinitions(Document doc) {
  // 读取文档的根元素
  Element root = doc.getDocumentElement();
  // 解析元素的根节点及根节点下的所有子节点并添加进注册容器
  parseBeanDefinitions(root);
}

/**
 * 解析元素的根节点及根节点下的所有子节点并添加进注册容器
 *
 * @param root XML 文件根节点
 */
private void parseBeanDefinitions(Element root) {
  // 读取根元素的所有子元素
  NodeList nl = root.getChildNodes();
  // 遍历子元素
  for (int i = 0; i < nl.getLength(); i++) {
    // 获取根元素的给定位置的节点
    Node node = nl.item(i);
    // 类型判断
    if (node instanceof Element) {
      // 强转为父类型元素
```

```
            Element ele = (Element) node;
            // 解析给给定的节点，包括name，class，property， name， value，ref
            processBeanDefinition(ele);
        }
    }
}

/**
 * 解析给给定的节点，包括name，class，property， name， value，ref
 *
 * @param ele XML 解析元素
 */
private void processBeanDefinition(Element ele) {
    // 获取给定元素的 name 属性
    String name = ele.getAttribute("name");
    // 获取给定元素的 class 属性
    String className = ele.getAttribute("class");
    // 创建一个bean定义对象
    BeanDefinition beanDefinition = new BeanDefinition();
    // 设置bean 定义对象的 全限定类名
    beanDefinition.setClassname(className);
    // 向 bean 注入配置文件中的成员变量
    addPropertyValues(ele, beanDefinition);
    // 向注册容器 添加bean名称和bean定义
    getRegistry().put(name, beanDefinition);
}

/**
 * 添加配置文件中的属性元素到bean定义实例中
 *
 * @param ele 元素
 * @param beandefinition bean定义 对象
 */
private void addPropertyValues(Element ele, BeanDefinition beandefinition) {
    // 获取给定元素的 property 属性集合
    NodeList propertyNode = ele.getElementsByTagName("property");
    // 循环集合
    for (int i = 0; i < propertyNode.getLength(); i++) {
        // 获取集合中某个给定位置的节点
        Node node = propertyNode.item(i);
        // 类型判断
        if (node instanceof Element) {
            // 将节点向下强转为子元素
            Element propertyEle = (Element) node;
            // 元素对象获取 name 属性
            String name = propertyEle.getAttribute("name");
            // 元素对象获取 value 属性值
            String value = propertyEle.getAttribute("value");
            // 判断value不为空
            if (value != null && value.length() > 0) {
                // 向给定的 “bean定义” 实例中添加该成员变量
                beandefinition.getPropertyValues().addPropertyValue(new PropertyValue(name, value));
            } else {
                // 如果为空，则获取属性ref
```

```
            String ref = propertyEle.getAttribute("ref");
            if (ref == null || ref.length() == 0) {
              // 如果属性ref为空，则抛出异常
              throw new IllegalArgumentException(
                  "Configuration problem: <property> element for property '"
                      + name + "' must specify a ref or value");
            }
            // 如果不为空，测创建一个 “bean的引用” 实例，构造参数为名称，实例暂时为空
            BeanReference beanRef = new BeanReference(name);
            // 向给定的 “bean定义” 中添加成员变量
            beandefinition.getPropertyValues().addPropertyValue(new PropertyValue(name, beanR
f));
          }
        }
      }
    }

}
```

可以说代码注释写的非常详细，该类方法如下：

1. public void readerXML(String location) 公开的解析XML的方法，给定一个位置的字符串参数即。

2. private void registerBeanDefinitions(Document doc) 给定一个文档对象，并进行解析。

3. private void parseBeanDefinitions(Element root) 给定一个根元素，循环解析根元素下所有子元。

4. private void processBeanDefinition(Element ele) 给定一个子元素，并对元素进行解析，然后拿解析出来的数据创建一个 BeanDefinition 对象。并注册到BeanDefinitionReader 的 Map 容器（该器存放着解析时的所有Bean）中。

5. private void addPropertyValues(Element ele, BeanDefinition beandefinition) 给定一个元素一个 BeanDefinition 对象，解析元素中的 property 元素， 并注入到 BeanDefinition 实例中。

一共5步，完成了解析XML文件的所有操作。 最终的目的是将解析出来的文件放入到 BeanDefinition eader 的 Map 容器中。

好了，到这里，我们已经完成了从XML文件读取并解析的步骤，那么什么时候放进BeanFactory的容呢？ 刚刚我们只是放进了 AbstractBeanDefinitionReader 的注册容器中。因此我们要根据BeanFactry 的设计来实现如何构建成一个真正能用的Bean呢？因为刚才的哪些Bean只是一些Bean的信息。没我们真正业务需要的Bean。

## 2. 初始化我们需要的Bean（不是Bean定义）并且实现依赖注入

我们知道Bean定义是不能干活的，只是一些Bean的信息，就好比一个人，BeanDefinition 就相当你公安局的档案，但是你人不在公安局，可只要公安局拿着你的档案就能找到你。就是这样一个关系。

那我们就根据BeanFactory的设计来设计一个抽象类 AbstractBeanFactory。

```
package cn.thinkinjava.myspring.factory;

import cn.thinkinjava.myspring.BeanDefinition;
import java.util.HashMap;
```

```
/**
 * 一个抽象类， 实现了 bean 的方法，包含一个map，用于存储bean 的名字和bean的定义
 *
 * @author stateis0
 */
public abstract class AbstractBeanFactory implements BeanFactory {

  /**
   * 容器
   */
  private HashMap<String, BeanDefinition> map = new HashMap<>();

  /**
   * 根据bean的名称获取bean， 如果没有，则抛出异常 如果有， 则从bean定义对象获取bean实例
   */
  @Override
  public Object getBean(String name) throws Exception {
    BeanDefinition beandefinition = map.get(name);
    if (beandefinition == null) {
      throw new IllegalArgumentException("No bean named " + name + " is defined");
    }
    Object bean = beandefinition.getBean();
    if (bean == null) {
      bean = doCreate(beandefinition);
    }
    return bean;
  }

  /**
   * 注册 bean定义 的抽象方法实现，这是一个模板方法， 调用子类方法doCreate，
   */
  @Override
  public void registerBeanDefinition(String name, BeanDefinition beandefinition) throws Excep
ion {
    Object bean = doCreate(beandefinition);
    beandefinition.setBean(bean);
    map.put(name, beandefinition);
  }

  /**
   * 减少一个bean
   */
  abstract Object doCreate(BeanDefinition beandefinition) throws Exception;
}
```

该类实现了接口的2个基本方法，一个是getBean，一个是 registerBeanDefinition， 我们也设计了个抽象方法供这两个方法调用，将具体逻辑创建逻辑延迟到子类。这是什么设计模式呢？模板模式。要还是看 doCreate 方法，就是创建bean 具体方法，所以我们还是需要一个子类， 叫什么呢? AutowreBeanFactory， 自动注入Bean，这是我们这个标准Bean工厂的工作。看看代码吧?

```
package cn.thinkinjava.myspring.factory;
```

```
import cn.thinkinjava.myspring.BeanDefinition;
import cn.thinkinjava.myspring.PropertyValue;
import cn.thinkinjava.myspring.BeanReference;
import java.lang.reflect.Field;


/**
 * 实现自动注入和递归注入(spring 的标准实现类 DefaultListableBeanFactory 有 1810 行)
 *
 * @author stateis0
 */
public class AutowireBeanFactory extends AbstractBeanFactory {


  /**
   * 根据bean 定义创建实例， 并将实例作为key， bean定义作为value存放，并调用 addPropertyVa
ue 方法 为给定的bean的属性进行注入
   */
  @Override
  protected Object doCreate(BeanDefinition beandefinition) throws Exception {
    Object bean = beandefinition.getBeanclass().newInstance();
    addPropertyValue(bean, beandefinition);
    return bean;
  }

  /**
   * 给定一个bean定义和一个bean实例，为给定的bean中的属性注入实例。
   */
  protected void addPropertyValue(Object bean, BeanDefinition beandefinition) throws Except
on {
    // 循环给定 bean 的属性集合
    for (PropertyValue pv : beandefinition.getPropertyValues().getPropertyValues()) {
      // 根据给定属性名称获取 给定的bean中的属性对象
      Field declaredField = bean.getClass().getDeclaredField(pv.getname());
      // 设置属性的访问权限
      declaredField.setAccessible(true);
      // 获取定义的属性中的对象
      Object value = pv.getvalue();
      // 判断这个对象是否是 BeanReference 对象
      if (value instanceof BeanReference) {
        // 将属性对象转为 BeanReference 对象
        BeanReference beanReference = (BeanReference) value;
        // 调用父类的 AbstractBeanFactory 的 getBean 方法，根据bean引用的名称获取实例，此处
是递归
        value = getBean(beanReference.getName());
      }
      // 反射注入bean的属性
      declaredField.set(bean, value);
    }

  }


}
```

可以看到 doCreate 方法使用了反射创建了一个对象，并且还需要对该对象进行属性注入，如果属性是 ref 类型，那么既是依赖关系，则需要调用 getBean 方法递归的去寻找那个Bean（因为最后一个Bean 的属性肯定是基本类型）。这样就完成了一次获取实例化Bean操作，并且也实现类依赖注入。

# 4. 总结

我们通过这些代码实现了一个简单的 IOC 依赖注入的功能，也更加了解了 IOC， 以后遇到Spring初化的问题再也不会手足无措了。直接看源码就能解决。哈哈

具体代码楼主放在了github上，地址：自己实现的一个简单IOC,包括依赖注入

good luck！！！