45-工厂模式（下）：如何设计实现一个DependencyInjection框架?

在上一节课我们讲到，当创建对象是一个“大工程”的时候，我们一般会选择使用工厂模式，来封装对象复杂的创建过程，将对象的创建和使用分离，让代码更加清晰。那何为“大工程”呢？上一节课中我们讲了两种情况，一种是创建过程涉及复杂的if-else分支判断，另一种是对象创建需要组装多个其他类对象或者需要复杂的初始化过程。

今天，我们再来讲一个创建对象的“大工程”，依赖注入框架，或者叫依赖注入容器（Dependency Injection Container），简称DI容器。在今天的讲解中，我会带你一块搞清楚这样几个问题：DI容器跟我们讲的工厂模式又有何区别和联系？DI容器的核心功能有哪些，以及如何实现一个简单的DI容器？

话不多说，让我们正式开始今天的学习吧！

工厂模式和DI容器有何区别?

实际上，DI容器底层最基本的设计思路就是基于工厂模式的。DI容器相当于一个大的工厂类，负责在程序启动的时候，根据配置（要创建哪些类对象，每个类对象的创建需要依赖哪些其他类对象）事先创建好对象。当应用程序需要使用某个类对象的时候，直接从容器中获取即可。正是因为它持有一堆对象，所以这个框架才被称为“容器”。

DI容器相对于我们上节课讲的工厂模式的例子来说，它处理的是更大的对象创建工程。上节课讲的工厂模式中，一个工厂类只负责某个类对象或者某一组相关类对象（继承自同一抽象类或者接口的子类）的创建，而DI容器负责的是整个应用中所有类对象的创建。

除此之外，DI容器负责的事情要比单纯的工厂模式要多。比如，它还包括配置的解析、对象生命周期的管理。接下来，我们就详细讲讲，一个简单的ID容器应该包含哪些核心功能。

DI容器的核心功能有哪些?

总结一下，一个简单的DI容器的核心功能一般有三个：配置解析、对象创建和对象生命周期管理。

首先，我们来看配置解析。

在上节课讲的工厂模式中，工厂类要创建哪个类对象是事先确定好的，并且是写死在工厂类代码中的。作为一个通用的框架来说，框架代码跟应用代码应该是高度解耦的，DI容器事先并不知道应用会创建哪些对象，不可能把某个应用要创建的对象写死在框架代码中。所以，我们需要通过一种形式，让应用告知DI容器要创建哪些对象。这种形式就是我们要讲的配置。

我们将需要由DI容器来创建的类对象和创建类对象的必要信息（使用哪个构造函数以及对应的构造函数参数都是什么等等），放到配置文件中。容器读取配置文件，根据配置文件提供的信息来创建对象。

下面是一个典型的Spring容器的配置文件。Spring容器读取这个配置文件，解析出要创建的两个对象：rateLimiter和redisCounter，并且得到两者的依赖关系：rateLimiter依赖redisCounter。

```
public class RateLimiter {
  private RedisCounter redisCounter;
  public RateLimiter(RedisCounter redisCounter) {
```

```
    this.redisCounter = redisCounter;
  }
  public void test() {
    System.out.println("Hello World!");
  }
  //...
}

public class RedisCounter {
  private String ipAddress;
  private int port;
  public RedisCounter(String ipAddress, int port) {
    this.ipAddress = ipAddress;
    this.port = port;
  }
  //...
}

配置文件beans.xml：
<beans>
   <bean id="rateLimiter" class="com.xzg.RateLimiter">
      <constructor-arg ref="redisCounter"/>
   </bean>

   <bean id="redisCounter" class="com.xzg.redisCounter">
     <constructor-arg type="String" value="127.0.0.1">
     <constructor-arg type="int" value=1234>
   </bean>
</beans>
```

其次，我们再来看对象创建。

在DI容器中，如果我们给每个类都对应创建一个工厂类，那项目中类的个数会成倍增加，这会增加代码的维护成本。要解决这个问题并不难。我们只需要将所有类对象的创建都放到一个工厂类中完成就可以了，比如BeansFactory。

你可能会说，如果要创建的类对象非常多，BeansFactory中的代码会不会线性膨胀（代码量跟创建对象的个数成正比）呢？实际上并不会。待会讲到DI容器的具体实现的时候，我们会讲“反射”这种机制，它能在程序运行的过程中，动态地加载类、创建对象，不需要事先在代码中写死要创建哪些对象。所以，不管是创建一个对象还是十个对象，BeansFactory工厂类代码都是一样的。

最后，我们来看对象的生命周期管理。

上一节课我们讲到，简单工厂模式有两种实现方式，一种是每次都返回新创建的对象，另一种是每次都返回同一个事先创建好的对象，也就是所谓的单例对象。在Spring框架中，我们可以通过配置scope属性，来区分这两种不同类型的对象。scope=prototype表示返回新创建的对象，scope=singleton表示返回单例对象。

除此之外，我们还可以配置对象是否支持懒加载。如果lazy-init=true，对象在真正被使用到的时候（比如：BeansFactory.getBean(“userService”)）才被被创建；如果lazy-init=false，对象在应用启动的时候就事先创建好。

不仅如此，我们还可以配置对象的init-method和destroy-method方法，比如init-

method=loadProperties()，destroy-method=updateConfigFile()。DI容器在创建好对象之后，会主动调用init-method属性指定的方法来初始化对象。在对象被最终销毁之前，DI容器会主动调用destroy-method属性指定的方法来做一些清理工作，比如释放数据库连接池、关闭文件。

如何实现一个简单的DI容器?

实际上，用Java语言来实现一个简单的DI容器，核心逻辑只需要包括这样两个部分：配置文件解析、根据配置文件通过“反射”语法来创建对象。

1.最小原型设计

因为我们主要是讲解设计模式，所以，在今天的讲解中，我们只实现一个DI容器的最小原型。像Spring框架这样的DI容器，它支持的配置格式非常灵活和复杂。为了简化代码实现，重点讲解原理，在最小原型中，我们只支持下面配置文件中涉及的配置语法。

```
配置文件beans.xml
<beans>
   <bean id="rateLimiter" class="com.xzg.RateLimiter">
      <constructor-arg ref="redisCounter"/>
   </bean>

   <bean id="redisCounter" class="com.xzg.redisCounter" scope="singleton" lazy-init="true">
     <constructor-arg type="String" value="127.0.0.1">
     <constructor-arg type="int" value=1234>
   </bean>
</bean
```

最小原型的使用方式跟Spring框架非常类似，示例代码如下所示：

```
public class Demo {
  public static void main(String[] args) {
    ApplicationContext applicationContext = new ClassPathXmlApplicationContext(
            "beans.xml");
    RateLimiter rateLimiter = (RateLimiter) applicationContext.getBean("rateLimiter");
    rateLimiter.test();
    //...
  }
}
```

2.提供执行入口

前面我们讲到，面向对象设计的最后一步是：组装类并提供执行入口。在这里，执行入口就是一组暴露给外部使用的接口和类。

通过刚刚的最小原型使用示例代码，我们可以看出，执行入口主要包含两部分：ApplicationContext和ClassPathXmlApplicationContext。其中，ApplicationContext是接口，ClassPathXmlApplicationContext是接口的实现类。两个类具体实现如下所示：

```
public interface ApplicationContext {
  Object getBean(String beanId);
}

public class ClassPathXmlApplicationContext implements ApplicationContext {
  private BeansFactory beansFactory;
  private BeanConfigParser beanConfigParser;

  public ClassPathXmlApplicationContext(String configLocation) {
    this.beansFactory = new BeansFactory();
    this.beanConfigParser = new XmlBeanConfigParser();
    loadBeanDefinitions(configLocation);
  }

  private void loadBeanDefinitions(String configLocation) {
    InputStream in = null;
    try {
      in = this.getClass().getResourceAsStream("/" + configLocation);
      if (in == null) {
        throw new RuntimeException("Can not find config file: " + configLocation);
      }
      List<BeanDefinition> beanDefinitions = beanConfigParser.parse(in);
      beansFactory.addBeanDefinitions(beanDefinitions);
    } finally {
      if (in != null) {
        try {
          in.close();
        } catch (IOException e) {
          // TODO: log error
        }
      }
    }
  }

  @Override
  public Object getBean(String beanId) {
    return beansFactory.getBean(beanId);
  }
}
```

从上面的代码中，我们可以看出，ClassPathXmlApplicationContext负责组装BeansFactory和BeanConfigParser两个类，串联执行流程：从classpath中加载XML格式的配置文件，通过BeanConfigParser解析为统一的BeanDefinition格式，然后，BeansFactory根据BeanDefinition来创建对象。

3.配置文件解析

配置文件解析主要包含BeanConfigParser接口和XmlBeanConfigParser实现类，负责将配置文件解析为BeanDefinition结构，以便BeansFactory根据这个结构来创建对象。

配置文件的解析比较繁琐，不涉及我们专栏要讲的理论知识，不是我们讲解的重点，所以这里我只给出两个类的大致设计思路，并未给出具体的实现代码。如果感兴趣的话，你可以自行补充完整。具体的代码框架如下所示：

```
public interface BeanConfigParser {
  List<BeanDefinition> parse(InputStream inputStream);
  List<BeanDefinition> parse(String configContent);
}

public class XmlBeanConfigParser implements BeanConfigParser {

  @Override
  public List<BeanDefinition> parse(InputStream inputStream) {
    String content = null;
    // TODO:...
    return parse(content);
  }

  @Override
  public List<BeanDefinition> parse(String configContent) {
    List<BeanDefinition> beanDefinitions = new ArrayList<>();
    // TODO:...
    return beanDefinitions;
  }

}

public class BeanDefinition {
  private String id;
  private String className;
  private List<ConstructorArg> constructorArgs = new ArrayList<>();
  private Scope scope = Scope.SINGLETON;
  private boolean lazyInit = false;
  // 省略必要的getter/setter/constructors

  public boolean isSingleton() {
    return scope.equals(Scope.SINGLETON);
  }


  public static enum Scope {
    SINGLETON,
    PROTOTYPE
  }

  public static class ConstructorArg {
    private boolean isRef;
    private Class type;
    private Object arg;
    // 省略必要的getter/setter/constructors
  }
}
```

4.核心工厂类设计

最后，我们来看，BeansFactory是如何设计和实现的。这也是我们这个DI容器最核心的一个类了。它负责根据从配置文件解析得到的BeanDefinition来创建对象。

如果对象的scope属性是singleton，那对象创建之后会缓存在singletonObjects这样一个map中，下次再请求此对象的时候，直接从map中取出返回，不需要重新创建。如果对象的scope属性是prototype，那每次请求对象，BeansFactory都会创建一个新的对象返回。

实际上，BeansFactory创建对象用到的主要技术点就是Java中的反射语法：一种动态加载类和创建对象的机制。我们知道，JVM在启动的时候会根据代码自动地加载类、创建对象。至于都要加载哪些类、创建哪些对象，这些都是在代码中写死的，或者说提前写好的。但是，如果某个对象的创建并不是写死在代码中，而是放到配置文件中，我们需要在程序运行期间，动态地根据配置文件来加载类、创建对象，那这部分工作就没法让JVM帮我们自动完成了，我们需要利用Java提供的反射语法自己去编写代码。

搞清楚了反射的原理，BeansFactory的代码就不难看懂了。具体代码实现如下所示：

```
public class BeansFactory {
  private ConcurrentHashMap<String, Object> singletonObjects = new ConcurrentHashMap<>();
  private ConcurrentHashMap<String, BeanDefinition> beanDefinitions = new ConcurrentHashMap<>();

  public void addBeanDefinitions(List<BeanDefinition> beanDefinitionList) {
    for (BeanDefinition beanDefinition : beanDefinitionList) {
      this.beanDefinitions.putIfAbsent(beanDefinition.getId(), beanDefinition);
    }

    for (BeanDefinition beanDefinition : beanDefinitionList) {
      if (beanDefinition.isLazyInit() == false && beanDefinition.isSingleton()) {
        createBean(beanDefinition);
      }
    }
  }

  public Object getBean(String beanId) {
    BeanDefinition beanDefinition = beanDefinitions.get(beanId);
    if (beanDefinition == null) {
      throw new NoSuchBeanDefinitionException("Bean is not defined: " + beanId);
    }
    return createBean(beanDefinition);
  }

  @VisibleForTesting
  protected Object createBean(BeanDefinition beanDefinition) {
    if (beanDefinition.isSingleton() && singletonObjects.contains(beanDefinition.getId())) {
      return singletonObjects.get(beanDefinition.getId());
    }

    Object bean = null;
    try {
      Class beanClass = Class.forName(beanDefinition.getClassName());
      List<BeanDefinition.ConstructorArg> args = beanDefinition.getConstructorArgs();
      if (args.isEmpty()) {
        bean = beanClass.newInstance();
      } else {
        Class[] argClasses = new Class[args.size()];
        Object[] argObjects = new Object[args.size()];
        for (int i = 0; i < args.size(); ++i) {
          BeanDefinition.ConstructorArg arg = args.get(i);
          if (!arg.getIsRef()) {
            argClasses[i] = arg.getType();
            argObjects[i] = arg.getArg();
          } else {
            BeanDefinition refBeanDefinition = beanDefinitions.get(arg.getArg());
            if (refBeanDefinition == null) {
              throw new NoSuchBeanDefinitionException("Bean is not defined: " + arg.getArg());
            }
            argClasses[i] = Class.forName(refBeanDefinition.getClassName());
            argObjects[i] = createBean(refBeanDefinition);
          }
```

```
        }
        bean = beanClass.getConstructor(argClasses).newInstance(argObjects);
      }
    } catch (ClassNotFoundException | IllegalAccessException
            | InstantiationException | NoSuchMethodException | InvocationTargetException e) {
      throw new BeanCreationFailureException("", e);
    }

    if (bean != null && beanDefinition.isSingleton()) {
      singletonObjects.putIfAbsent(beanDefinition.getId(), bean);
      return singletonObjects.get(beanDefinition.getId());
    }
    return bean;
  }
}
```

重点回顾

好了，今天的内容到此就讲完了。我们来一块总结回顾一下，你需要重点掌握的内容。

DI容器在一些软件开发中已经成为了标配，比如Spring IOC、Google Guice。但是，大部分人可能只是把它当作一个黑盒子来使用，并未真正去了解它的底层是如何实现的。当然，如果只是做一些简单的小项目，简单会用就足够了，但是，如果我们面对的是非常复杂的系统，当系统出现问题的时候，对底层原理的掌握程度，决定了我们排查问题的能力，直接影响到我们排查问题的效率。

今天，我们讲解了一个简单的DI容器的实现原理，其核心逻辑主要包括：配置文件解析，以及根据配置文件通过“反射”语法来创建对象。其中，创建对象的过程就应用到了我们在学的工厂模式。对象创建、组装、管理完全有DI容器来负责，跟具体业务代码解耦，让程序员聚焦在业务代码的开发上。

课堂讨论

BeansFactory类中的createBean()函数是一个递归函数。当构造函数的参数是ref类型时，会递归地创建ref属性指向的对象。如果我们在配置文件中错误地配置了对象之间的依赖关系，导致存在循环依赖，那BeansFactory的createBean()函数是否会出现堆栈溢出？又该如何解决这个问题呢？

你可以可以在留言区说一说，和同学一起交流和分享。如果有收获，也欢迎你把这篇文章分享给你的朋友。

精选留言：

- 此鱼不得水 2020-02-14 11:15:32
  Spring解决循环依赖的办法是多级缓存。

- Monday 2020-02-14 09:40:25
  思考题：
  1、如果循环依赖的类都是SINGLETON，不会出现堆栈溢出
  2、如果循环依赖的类都是PROTOTYPE，本章的代码来看，的确会出现堆栈溢出；解决办法，可以做递归的深度控制。

- pedro 2020-02-14 09:38:59
  小争哥这个DI框架已经有Spring IOC的雏形了

- 明翼 2020-02-14 08:41:38
  关于递归的判断我的理解是否可以保存一个list，里面保存依赖的class名，如果加入了依赖已经存在了，说明存在递归，则抛出异常

- Geek_3b1096 2020-02-14 04:09:19
  终于解答了我对于DI的疑惑