@dugu9sword 2016-05-24T05:46:41.000000Z 字数 9364 阅读 31735

tiny-spring 分析

> 前言
>
> 在阅读 Spring 的源代码（依赖注入部分和面向切面编程部分）时遇到不少困惑，庞大的类文件结构、纷繁复杂的方法调用、波诡云谲的多态实现，让自己深陷其中、一头雾水。
> 后来注意到 code4craft 的 tiny-spring 项目，实现了一个微型的 Spring，提供对 IoC 和 AOP 的最基本支持，麻雀虽小，五脏俱全，对 Spring 的认知清晰了不少。这个微型框架的结构包括文件名、方法名都是参照 Spring 来实现的，对于初读 Spring 的学习者，作为研究 Spring 的辅助工具应该能够受益匪浅。
> 在研究 tiny-spring 的时候，收获颇多，把对这个微型框架的一些分析写了下来，行文可能有点紊乱。
>
> 本文结构
>
> 1. 第一部分 **IoC** 容器的实现 对应了 tiny-spring 的 step-1 到 step-5 部分，这 5 个 step 实现了基本的 IoC 容器，支持singleton类型的bean，包括初始化、属性注入、以及依赖 Bean 注入，可从 XML 中读取配置，XML 读取方式没有具体深入。
> 2. 第二部分 **AOP** 容器的实现 对应了 tiny-spring 的 step-6 到 step-9 部分。step-10 中对 cglib 的支持没有分析。这 4 个 step 可以使用 AspectJ 的语法进行 AOP 编写，支持接口代理。考虑到 AspectJ 语法仅用于实现 `execution("***")` 部分的解析，不是主要内容，也可以使用 Java 的正则表达式粗略地完成，因此没有关注这些细节。
>
> 参考书目
> 《Spring 实战》《Spring 技术内幕》

---

tiny-spring 分析
IoC 容器的实现
文件结构
Resource

        - BeanDefinition
        - BeanFactory
        - ApplicationContext
    - 设计模式
        - 模板方法模式
        - 代理模式
- AOP 的实现
    - 重新分析 IoC 容器
        - BeanFactory 的构造与执行
        - ApplicationContext 的构造和执行
        - IoC 实现的一些思考与分析
            - 分析 1：AOP 可以在何处被嵌入到 IoC 容器中去？
            - 分析 2：BeanFactory 和 ApplicationContext 设计上的耦合
            - 分析 3：tiny-spring 总体流程的分析
    - JDK 对动态代理的支持
    - AOP 的植入与实现细节
        - 在 Bean 初始化过程中完成 AOP 的植入
        - AOP 中动态代理的实现步骤
            - 动态代理的内容
            - 动态代理的步骤
    - 设计模式
        - 代理模式
        - 策略模式
    - 为 tiny-spring 添加拦截器链

为什么 GitHub 不支持 TOC

IoC 容器的实现

文件结构

Resource

以 `Resource` 接口为核心发散出的几个类，都是用于解决 IoC 容器中的内容从哪里来的问题

，也就是 配置文件从哪里读取、配置文件如何读取 的问题。

类名	说明
`Resource`	接口，标识一个外部资源。通过 `getInputStream()` 方法 获取资源的输入流 。
`UrlResource`	实现 `Resource` 接口的资源类，通过 URL 获取资源。
`ResourceLoader`	资源加载类。通过 `getResource(String)` 方法获取一个 `Resouce` 对象，是 获取 **`Resouce`** 的主要途径 。

注： 这里在设计上有一定的问题，`ResourceLoader` 直接返回了一个 `UrlResource`，更好的方法是声明一个 `ResourceLoader` 接口，再实现一个 `UrlResourceLoader` 类用于加载 `UrlResource`。

BeanDefinition

以 `BeanDefinition` 类为核心发散出的几个类，都是用于解决 `Bean` 的具体定义问题，包括 `Bean` 的名字是什么、它的类型是什么，它的属性赋予了哪些值或者引用，也就是 如何在 **IoC** 容器中定义一个 **`Bean`**，使得 **IoC** 容器可以根据这个定义来生成实例 的问题。

类名	说明
`BeanDefinition`	该类保存了 `Bean` 定义。包括 `Bean` 的 名字 `String beanClassName`、类型 `Class beanClass`、属性 `PropertyValues propertyValues`。根据其 类型 可以生成一个类实例，然后可以把 属性 注入进去。`propertyValues` 里面包含了一个个 `PropertyValue` 条目，每个条目都是键值对 `String - Object`，分别对应要生成实例的属性的名字与类型。在 Spring 的 XML 中的 `property` 中，键是 `key`，值是 `value` 或者 `ref`。对于 `value` 只要直接注入属性就行了，但是 `ref` 要先进行解析。`Object` 如果是 `BeanReference` 类型，则说明其是一个引用，其中保存了引用的名字，需要用先进行解析，转化为对应的实际 `Object`。
`BeanDefinitionReader`	解析 `BeanDefinition` 的接口。通过 `loadBeanDefinitions(String)` 来从一个地址加载类定义。

类名	说明
`AbstractBeanDefinitionReader`	实现 `BeanDefinitionReader` 接口的抽象类（未具体实现 `loadBeanDefinitions`，而是规范了 `BeanDefinitionReader` 的基本结构）。内置一个 `HashMap rigistry`，用于保存 `String`-`beanDefinition` 的键值对。内置一个 `ResourceLoader resourceLoader`，用于保存类加载器。用意在于，使用时，只需要向其 `loadBeanDefinitions()` 传入一个资源地址，就可以自动调用其类加载器，并把解析到的 `BeanDefinition` 保存到 `registry` 中去。
`XmlBeanDefinitionReader`	具体实现了 `loadBeanDefinitions()` 方法，从 XML 文件中读取类定义。

BeanFactory

以 `BeanFactory` 接口为核心发散出的几个类，都是用于解决 IoC 容器在 已经获取 **`Bean`** 的定义的情况下，如何装配、获取 **`Bean`** 实例 的问题。

类名	说明
`BeanFactory`	接口，标识一个 IoC 容器。通过 `getBean(String)` 方法来 获取一个对象
`AbstractBeanFactory`	`BeanFactory` 的一种抽象类实现，规范了 IoC 容器的基本结构，但是把生成 `Bean` 的具体实现方式留给子类实现。IoC 容器的结构：`AbstractBeanFactory` 维护一个 `beanDefinitionMap` 哈希表用于保存类的定义信息（`BeanDefinition`）。获取 `Bean` 时，如果 `Bean` 已经存在于容器中，则返回之，否则则调用 `doCreateBean` 方法装配一个 `Bean`。（所谓存在于容器中，是指容器可以通过 `beanDefinitionMap` 获取 `BeanDefinition` 进而通过其 `getBean()` 方法获取 `Bean`。）
`AutowireCapableBeanFactory`	可以实现自动装配的 `BeanFactory`。在这个工厂中，实现了 `doCreateBean` 方法，该方法分三步：1，通过 `BeanDefinition` 中保存的类信息实例化一个对象；2，把对象保存在 `BeanDefinition` 中，以备下次获取；3，为其装配属性。装配属性时，通过 `BeanDefinition` 中维护的 `PropertyValues` 集合类

类名	说明
	，把 `String` - `Value` 键值对注入到 `Bean` 的属性中去。如果 `Value` 的类型是 `BeanReference` 则说明其是一个引用（对应于 XML 中的 `ref` ），通过 `getBean` 对其进行获取，然后注入到属性中。

ApplicationContext

以 `ApplicationContext` 接口为核心发散出的几个类，主要是对前面 `Resouce` 、`BeanFactory`、 `BeanDefinition` 进行了功能的封装，解决 根据地址获取 **IoC** 容器并使用的问题。

类名	说明
`ApplicationContext`	标记接口，继承了 `BeanFactory` 。通常，要实现一个 IoC 容器时，需要先通过 `ResourceLoader` 获取一个 `Resource` ，其中包括了容器的配置、`Bean` 的定义信息。接着，使用 `BeanDefinitionReader` 读取该 `Resource` 中的 `BeanDefinition` 信息。最后，把 `BeanDefinition` 保存在 `BeanFactory` 中，容器配置完毕可以使用。注意到 `BeanFactory` 只实现了 `Bean` 的 装配、获取，并未说明 `Bean` 的 来源 也就是 `BeanDefinition` 是如何 加载 的。该接口把 `BeanFactory` 和 `BeanDefinitionReader` 结合在了一起。
`AbstractApplicationContext`	`ApplicationContext` 的抽象实现，内部包含一个 `BeanFactory` 类。主要方法有 `getBean()` 和 `refresh()` 方法。`getBean()` 直接调用了内置 `BeanFactory` 的 `getBean()` 方法，`refresh()` 则用于实现 `BeanFactory` 的刷新，也就是告诉 `BeanFactory` 该使用哪个资源（`Resource` ）加载类定义（ `BeanDefinition` ）信息，该方法留给子类实现 ，用以实现 从不同来源的不同类型的资源加载类定义 的效果。
ClassPathXmlApplicationContext	从类路径加载资源的具体实现类。内部通过 `XmlBeanDefinitionReader` 解析 `UrlResourceLoader` 读取到的 `Resource` ，获取 `BeanDefinition` 信息，然后将其保存到内置的 `BeanFactory` 中。

注 1：在 Spring 的实现中，对 `ApplicatinoContext` 的分层更为细致。`AbstractApplicationContext` 中为了实现 不同来源 的 不同类型 的资源加载类定义，把这两步分层实现。以“从类路径读取 XML 定义”为例，首先使用 `AbstractXmlApplicationContext` 来实现 不同类型 的资源解析，接着，通过 `ClassPathXmlApplicationContext` 来实现 不同来源 的资源解析。
注 2：在 tiny-spring 的实现中，先用 `BeanDefinitionReader` 读取 `BeanDefiniton` 后，保存在内置的 `registry` （键值对为 `String` - `BeanDefinition` 的哈希表，通过 `getRigistry()` 获取）中，然后由 `ApplicationContext` 把 `BeanDefinitionReader` 中 `registry` 的键值对一个个赋值给 `BeanFactory` 中保存的 `beanDefinitionMap`。而在 Spring 的实现中，`BeanDefinitionReader` 直接操作 `BeanDefinition` ，它的 `getRegistry()` 获取的不是内置的 `registry` ，而是 `BeanFactory` 的实例。如何实现呢？以 `DefaultListableBeanFactory` 为例，它实现了一个 `BeanDefinitonRigistry` 接口，该接口把 `BeanDefinition` 的 注册 、获取 等方法都暴露了出来，这样，`BeanDefinitionReader` 可以直接通过这些方法把 `BeanDefiniton` 直接加载到 `BeanFactory` 中去。

设计模式

注：此处的设计模式分析不限于 *tiny-spring*，也包括 *Spring* 本身的内容

模板方法模式

该模式大量使用，例如在 `BeanFactory` 中，把 `getBean()` 交给子类实现，不同的子类 `**BeanFactory` 对其可以采取不同的实现。

代理模式

在 tiny-spring 中（Spring 中也有类似但不完全相同的实现方式），`ApplicationContext` 继承了 `BeanFactory` 接口，具备了 `getBean()` 功能，但是又内置了一个 `BeanFactory` 实例，`getBean()` 直接调用 `BeanFactory` 的 `getBean()` 。但是`ApplicationContext` 加强了 `BeanFactory`，它把类定义的加载也包含进去了。

AOP 的实现

重新分析 IoC 容器

注：以下所说的 *BeanFactory* 和 *ApplicationContext* 不是指的那几个最基本的接口类（例如 *BeanFactory* 接口，它除了 *getBean* 空方法之外，什么都没有，无法用来分析。），而是指这一类对象总体的表现，比如 *ClasspathXmlApplicationContext*、*FileSystemXmlApplicationContext* 都算是 *ApplicationContext*。

BeanFactory 的构造与执行

`BeanFactory` 的核心方法是 `getBean(String)` 方法，用于从工厂中取出所需要的 `Bean`。`AbstractBeanFactory` 规定了基本的构造和执行流程。

`getBean` 的流程：包括实例化和初始化，也就是生成 *Bean*，再执行一些初始化操作。

1. `doCreateBean` ：实例化 `Bean`。
   a. `createInstance` ：生成一个新的实例。
   b. `applyProperties` ：注入属性，包括依赖注入的过程。在依赖注入的过程中，如果 `Bean` 实现了 `BeanFactoryAware` 接口，则将容器的引用传入到 `Bean` 中去，这样，`Bean` 将获取对容器操作的权限，也就允许了 编写扩展 **IoC** 容器的功能的 **Bean**。
2. `initializeBean(bean)` ： 初始化 `Bean`。
   a. 从 `BeanPostProcessor` 列表中，依次取出 `BeanPostProcessor` 执行 `bean = postProcessBeforeInitialization(bean,beanName)` 。（为什么调用 *BeanPostProceesor* 中提供方法时，不是直接 *post...(bean,beanName)* 而是 *bean = post...(bean,beanName)* 呢？见分析1 。另外，*BeanPostProcessor* 列表的获取有问题，见分析2。）
   b. 初始化方法（tiny-spring 未实现对初始化方法的支持）。
   c. 从 `BeanPostProcessor` 列表中， 依次取出 `BeanPostProcessor` 执行其 `bean = postProcessAfterInitialization(bean,beanName)`。

ApplicationContext 的构造和执行

`ApplicationContext` 的核心方法是 `refresh()` 方法，用于从资源文件加载类定义、扩展容器的功能。

`refresh` 的流程：

1. `loadBeanDefinitions(BeanFactory)` ：加载类定义，并注入到内置的 `BeanFactory` 中，这里的可扩展性在于，未对加载方法进行要求，也就是可以从不同来源的不同类型的资源进行加载。

2. `registerBeanPostProcessors(BeanFactory)` ：获取所有的 `BeanPostProcessor`，并注册到 `BeanFactory` 维护的 `BeanPostProcessor` 列表去。
3. `onRefresh` ：
   a. `preInstantiateSingletons` ：以单例的方式，初始化所有 `Bean`。tiny-spring 只支持 `singleton` 模式。

IoC 实现的一些思考与分析

分析 1：AOP 可以在何处被嵌入到 IoC 容器中去？

在 `Bean` 的初始化过程中，会调用 `BeanPostProcessor` 对其进行一些处理。在它的 `postProcess...Initialization` 方法中返回了一个 `Bean`，这个返回的 `Bean` 可能已经不是原来传入的 `Bean` 了，这为实现 **AOP** 的代理提供了可能！以 JDK 提供的动态代理为例，假设方法要求传入的对象实现了 `IObj` 接口，实际传入的对象是 `Obj`，那么在方法中，通过动态代理，可以 生成一个实现了 **`IObj`** 接口并把 **`Obj`** 作为内置对象的代理类 **`Proxy`** 返回，此时 `Bean` 已经被偷偷换成了它的代理类。

分析 2：BeanFactory 和 ApplicationContext 设计上的耦合

`BeanFactory` 中的 `BeanPostProcessor` 的列表是哪里生成的呢？是在 `ApplicationContext` 中的 `refresh` 方法的第二步，这里设计上应该有些问题，按理说 `ApplicationContext` 是基于 `BeanFactory` 的，`BeanFactory` 的属性的获取，怎么能依赖于 `ApplicationContext` 的调用呢？

分析 3：tiny-spring 总体流程的分析

总体来说，tiny-spring 的 `ApplicaitonContext` 使用流程是这样的：
1. `ApplicationContext` 完成了类定义的读取和加载，并注册到 `BeanFactory` 中去。
2. `ApplicationContext` 从 `BeanFactory` 中寻找 `BeanPostProcessor`，注册到 `BeanFactory`
维护的 `BeanPostProcessor` 列表中去。
3. `ApplicationContext` 以单例的模式，通过主动调用 `getBean` 实例化、注入属性、然后初始化 `BeanFactory` 中所有的 `Bean`。由于所有的 `BeanPostProcessor` 都已经在第 2 步中完成实例化了，因此接下来实例化的是普通 `Bean`，因此普通 `Bean` 的初始化过程可以正常执行。
4. 调用 `getBean` 时，委托给 `BeanFactory`，此时只是简单的返回每个 `Bean` 单例，因为所有的 `Bean` 实例在第三步都已经生成了。

JDK 对动态代理的支持

JDK 中几个关键的类：

类名	说明
`Proxy`	来自 JDK API。提供生成对象的动态代理的功能，通过 `Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)` 方法返回一个代理对象。
`InvocationHandler`	来自 JDK API。通过 `Object invoke(Object proxy, Method method,Object[] args)` 方法实现代理对象中方法的调用和其他处理。

假设以下的情况：

- 对象 `obj` 实现了 `IObj` 接口，接口中有一个方法 `func(Object[] args)`。
- 对象 `handler` 是 `InvocationHandler` 的实例。

那么，通过 `Proxy` 的 `newProxyInstance(obj.getClassLoader(), obj.getClass().getInterfaces(), handler`，可以返回 `obj` 的代理对象 `proxy`。

当调用 `proxy.func(args)` 时，对象内部将委托给 `handler.invoke(proxy, func, args)` 函数实现。

因此，在 **`handler`** 的 **`invoke`** 中，可以完成对方法拦截的处理。可以先判断是不是要拦截的方法，如果是，进行拦截（比如先做一些操作，再调用原来的方法，对应了 Spring 中的前置通知）；如果不是，则直接调用原来的方法。

AOP 的植入与实现细节

在 `Bean` 初始化过程中完成 AOP 的植入

解决 AOP 的植入问题，首先要解决 在 **IoC** 容器的何处植入 **AOP** 的问题，其次要解决 为哪些对象提供 **AOP** 的植入 的问题。

tiny-spring 中 `AspectJAwareAdvisorAutoProxyCreator` 类（以下简称 `AutoProxyCreator`）是实现 AOP 植入的关键类，它实现了两个接口：

1. `BeanPostProcessor` ：在 `postProcessorAfterInitialization` 方法中，使用动态代理的方式，返回一个对象的代理对象。解决了 在 **IoC** 容器的何处植入 **AOP** 的问题。
2. `BeanFactoryAware` ：这个接口提供了对 `BeanFactory` 的感知，这样，尽管它是容器中的一个 `Bean` ，却可以获取容器的引用，进而获取容器中所有的切点对象，决定对哪些对象的哪些方法进行代理。解决了 为哪些对象提供 **AOP** 的植入 的问题。

AOP 中动态代理的实现步骤

动态代理的内容

首先，要知道动态代理的内容（拦截哪个对象、在哪个方法拦截、拦截具体内容），下面是几个关键的类：

类名	说明
`PointcutAdvisor`	切点通知器，用于提供 对哪个对象的哪个方法进行什么样的拦截 的具体内容。通过它可以获取一个切点对象 `Pointcut` 和一个通知器对象 `Advisor`。
`Pointcut`	切点对象可以获取一个 `ClassFilter` 对象和一个 `MethodMatcher` 对象。前者用于判断是否对某个对象进行拦截（用于 筛选要代理的目标对象），后者用于判断是否对某个方法进行拦截（用于 在代理对象中对不同的方法进行不同的操作）。
`Advisor`	通知器对象可以获取一个通知对象 `Advice` 。就是用于实现 具体的方法拦截，需要使用者编写，也就对应了 Spring 中的前置通知、后置通知、环切通知等。

动态代理的步骤

接着要知道动态代理的步骤：

1. `AutoProxyCreator`（实现了 `BeanPostProcessor` 接口）在实例化所有的 `Bean` 前，最先被实例化。
2. 其他普通 `Bean` 被实例化、初始化，在初始化的过程中，`AutoProxyCreator` 加载 `BeanFactory` 中所有的 `PointcutAdvisor`（这也保证了 `PointcutAdvisor` 的实例化顺序优于普通 `Bean`。），然后依次使用 `PointcutAdvisor` 内置的 `ClassFilter` ，判断当前对象是不是要拦截的类。
3. 如果是，则生成一个 `TargetSource`（要拦截的对象和其类型），并取出 `AutoProxyCreator` 的 `MethodMatcher`（对哪些方法进行拦截）、`Advice`（拦截的具体操作），再，交给 `AopProxy` 去生成代理对象。

4. `AopProxy` 生成一个 `InvocationHandler`，在它的 `invoke` 函数中，首先使用 `MethodMatcher` 判断是不是要拦截的方法，如果是则交给 `Advice` 来执行（`Advice` 由用户来编写，其中也要手动/自动调用原始对象的方法），如果不是，则直接交给 `TargetSource` 的原始对象来执行。

设计模式

代理模式

通过动态代理实现，见分析1中的内容，不再赘述。

策略模式

生成代理对象时，可以使用 JDK 的动态代理和 Cglib 的动态代理，对于不同的需求可以委托给不同的类实现。

为 tiny-spring 添加拦截器链

tiny-spring 不支持拦截器链，可以模仿 Spring 中拦截器链的实现，实现对多拦截器的支持。
tiny-spring 中的 `proceed()` 方法是调用原始对象的方法 `method.invoke(object,args)`。（参见 `ReflectiveMethodInvocation` 类)
为了支持多拦截器，做出以下修改：

- 将 `proceed()` 方法修改为调用代理对象的方法 `method.invoke(proxy,args)`。
- 在代理对象的 `InvocationHandler` 的 `invoke` 函数中，查看拦截器列表，如果有拦截器，则调用第一个拦截器并返回，否则调用原始对象的方法。