> 作者：千锋-索尔
>
> 版本：QF1.0
>
> 版权：千锋Java教研院

基于Java的容器配置——JavaConfig

本节介绍如何在Java代码中使用注解来配置Spring容器。 它包括以下主题：：

- 基本概念: `@Bean` 和 `@Configuration`
- 使用`AnnotationConfigApplicationContext`实例化Spring容器
- 使用`@Bean`注解
- 使用`@Configuration`注解
- 编写基于Java的配置
- 定义Bean配置文件
- `PropertySource` 抽象
- 使用 `@PropertySource`
- 声明中的占位符

1.1. 基本概念: `@Bean` 和 `@Configuration`

Spring新的基于Java配置的核心内容是`@Configuration`注解的类和`@Bean`注解的方法。

`@Bean`注解用于表明方法的实例化，、配置和初始化都是由Spring IoC容器管理的新对象，对于那些熟悉Spring的`<beans/>`XML配置的人来说， `@Bean`注解扮演的角色与`<bean />`元素相同。开发者可以在任意的Spring `@Component`中使用`@Bean`注解方法 ，但大多数情况下， `@Bean`是配合`@Configuration`使用的。

使用`@Configuration`注解类时，这个类的目的就是作为bean定义的地方。此外， `@Configuration`类允许通过调用同一个类中的其他`@Bean`方法来定义bean间依赖关系。 最简单的`@Configuration`类如下所示：

```
@Configuration
public class AppConfig {

    @Bean
    public MyService myService() {
        return new MyServiceImpl();
    }
}
```

前面的`AppConfig`类等效于以下Spring `<beans/>`XML：

```
<beans>
    <bean id="myService" class="com.acme.services.MyServiceImpl"/>
</beans>
```

完整的@Configuration模式对比“lite”模式的@Bean?

当`@Bean`方法在没有用`@Configuration`注解的类中声明时，它们将会被称为“lite”的模式处理。例如，`@Component`中声明的bean方法或者一个普通的旧类中的bean方法将被视为 “lite”的。包含类的主要目的不同，而`@Bean`方法在这里是一种额外的好处。例如，服务组件可以通过在每个适用的组件类上使用额外的`@Bean`方法将管理视图公开给容器。 在这种情况下，`@Bean`方法是一种通用的工厂方法机制。

与完整的`@Configuration`不同，lite的`@Bean`方法不能声明bean之间的依赖关系。 相反，它们对其包含组件的内部状态进行操作，并且可以有选择的对它们可能声明的参数进行操作。因此，这样的`@Bean`注解的方法不应该调用其他`@Bean`注解的方法。 每个这样的方法实际上只是特定bean引用的工厂方法，没有任何特殊的运行时语义。不经过CGLIB处理，所以在类设计方面没有限制（即，包含类可能是最终的）。

在常见的场景中，`@Bean`方法将在`@Configuration`类中声明，确保始终使用“full”模式，这将防止相同的`@Bean`方法被意外地多次调用，这有助于减少在 “lite”模式下操作时难以跟踪的细微错误。

`@Bean`和`@Configuration`注解将在下面的章节深入讨论，首先，我们将介绍使用基于Java代码的配置来创建Spring容器的各种方法。

1.2. 使用`AnnotationConfigApplicationContext`初始化Spring容器

以下部分介绍了Spring的`AnnotationConfigApplicationContext`，它是在Spring 3.0中引入的。这是一个强大的(versatile)`ApplicationContext`实现,它不仅能解析`@Configuration`注解类 ,也能解析`@Component`注解的类和使用JSR-330注解的类.

当使用`@Configuration`类作为输入时,`@Configuration`类本身被注册为一个bean定义,类中所有声明的`@Bean`方法也被注册为bean定义.

当提供`@Component`和JSR-330类时，它们被注册为bean定义，并且假定在必要时在这些类中使用DI元数据，例如`@Autowired`或`@Inject`。

简单结构

与实例化`ClassPathXmlApplicationContext`时Spring XML文件用作输入的方式大致相同， 在实例化`AnnotationConfigApplicationContext`时可以使用`@Configuration`类作为输入。 这允许完全无XML使用Spring容器，如以下示例所示：

```
public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);
    MyService myService = ctx.getBean(MyService.class);
    myService.doStuff();
}
```

如前所述，`AnnotationConfigApplicationContext`不仅限于使用`@Configuration`类。 任何`@Component`或JSR-330带注解的类都可以作为输入提供给构造函数，如以下示例所示：

```
public static void main(String[] args) {
    ApplicationContext ctx = new
AnnotationConfigApplicationContext(MyServiceImpl.class, Dependency1.class,
Dependency2.class);
    MyService myService = ctx.getBean(MyService.class);
    myService.doStuff();
}
```

上面假设`MyServiceImpl`, `Dependency1`, 和 `Dependency2`使用Spring依赖注入注解，例如`@Autowired`。

1.3. 使用`@Bean`注解

@Bean是一个方法级别的注解，它与XML中的`<bean/>`元素类似。注解支持`<bean/>`提供的一些属性，例如init-method、destroy-method、autowiring、name。开发者可以在`@Configuration`类或`@Component`类中使用`@Bean`注解。

声明一个Bean

要声明一个bean，只需使用`@Bean`注解方法即可。使用此方法，将会在`ApplicationContext`内注册一个bean，bean的类型是方法的返回值类型。默认情况下， bean名称将与方法名称相同。以下示例显示了`@Bean`方法声明：

```
@Configuration
public class AppConfig {

    @Bean
    public TransferServiceImpl transferService() {
        return new TransferServiceImpl();
    }
}
```

前面的配置完全等同于以下Spring XML:

```
<beans>
    <bean id="transferService" class="com.acme.TransferServiceImpl"/>
</beans>
```

这两个声明都在`ApplicationContext`中创建一个名为`transferService`的bean，并且绑定了`TransferServiceImpl`的实例。如下图所示：

transferService -> com.acme.TransferServiceImpl

还可以使用接口（或基类）返回类型声明`@Bean`方法，如以下示例所示：

```
@Configuration
public class AppConfig {

    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl();
    }
}
```

但是，这会将预先类型预测的可见性限制为指定的接口类型(`TransferService`),然后在实例化受影响的单一bean时,只知道容器的完整类型(`TransferServiceImpl`）。。非延迟的单例bean根据它们的声明顺序进行实例化，因此开发者可能会看到不同类型的匹配结果，这具体取决于另一个组件尝试按未类型匹配的时间(如`@Autowired TransferServiceImpl`， 一旦`transferService` bean已被实例化,这个问题就被解决了).

如果通过声明的服务接口都是引用类型,那么`@Bean` 返回类型可以安全地加入该设计决策.但是,对于实现多个接口的组件或可能由其实现类型引用的组件, 更安全的方法是声明可能的最具体的返回类型(至少按照注入点所要求的特定你的bean）。

Bean之间的依赖

一个使用`@Bean`注解的方法可以具有任意数量的参数描述构建该bean所需的依赖，例如，如果我们的`TransferService`需要`AccountRepository`， 我们可以使用方法参数来实现该依赖关系，如以下示例所示：

```
@Configuration
public class AppConfig {

    @Bean
    public TransferService transferService(AccountRepository accountRepository) {
        return new TransferServiceImpl(accountRepository);
    }
}
```

接收生命周期回调

使用`@Bean`注解定义的任何类都支持常规的生命周期回调，并且可以使用JSR-的`@PostConstruct`和`@PreDestroy`注解。

`@Bean`注解支持指定任意初始化和销毁回调方法，就像`bean`元素上的Spring XML的`init-method`和`destroy-method`属性一样，如下例所示：

```
public class BeanOne {

    public void init() {
        // initialization logic
    }
}

public class BeanTwo {

    public void cleanup() {
        // destruction logic
    }
}

@Configuration
public class AppConfig {

    @Bean(initMethod = "init")
    public BeanOne beanOne() {
        return new BeanOne();
    }

    @Bean(destroyMethod = "cleanup")
    public BeanTwo beanTwo() {
        return new BeanTwo();
    }
}
```

默认情况下，使用Java Config定义的bean中`close`方法或者`shutdown`方法，会作为销毁回调而自动调用。若bean中有`close`或`shutdown`方法，并且不希望在容器关闭时调用它，则可以将`@Bean(destroyMethod="")`添加到bean定义中以禁用默认`(inferred)`模式。

开发者可能希望对通过JNDI获取的资源执行此操作，因为它的生命周期是在应用程序外部管理的。更进一步，使用`DataSource`时一定要关闭它，不关闭将会出问题。

以下示例说明如何防止`DataSource`的自动销毁回调：

```
@Bean(destroyMethod="")
public DataSource dataSource() throws NamingException {
    return (DataSource) jndiTemplate.lookup("MyDS");
}
```

同样地，使用`@Bean`方法，通常会选择使用程序化的JNDI查找：使用Spring的`JndiTemplate` / `JndiLocatorDelegate`帮助类或直接使用JNDI的`InitialContext`，但是不要使用`JndiObjectFactoryBean`的变体，因为它会强制开发者声明一个返回类型作为`FactoryBean`的类型用于代替实际的目标类型，这会使得交叉引用变得很困难。

对于前面注解中上面示例中的`BeanOne`，在构造期间直接调用`init()`方法同样有效，如下例所示：

```
@Configuration
public class AppConfig {

    @Bean
    public BeanOne beanOne() {
        BeanOne beanOne = new BeanOne();
        beanOne.init();
        return beanOne;
    }

    // ...
}
```

当直接使用Java（new对象那种）工作时，可以使用对象执行任何喜欢的操作，并且不必总是依赖于容器生命周期。

指定Bean范围

Spring包含`@Scope`注解，以便可以指定bean的范围。

使用`@Scope`注解

可以使用任意标准的方式为`@Bean`注解的bean指定一个作用域，你可以使用Bean Scopes中的任意标准作用域

默认范围是`singleton`的，但是可以使用`@Scope`注解来覆盖。如下例所示：

```
@Configuration
public class MyConfiguration {

    @Bean
    @Scope("prototype")
    public Encryptor encryptor() {
        // ...
    }
}
```

自定义Bean命名

默认情况下，配置类使用`@Bean`方法的名称作为结果bean的名称。 但是，可以使用`name`属性覆盖此功能，如以下示例所示：

```
@Configuration
public class AppConfig {

    @Bean(name = "myThing")
    public Thing thing() {
        return new Thing();
    }
}
```

bean别名

正如Bean的 命名中所讨论的，有时需要为单个bean提供多个名称，也称为bean别名。 `@Bean`注解的 `name`属性为此接受String数组。 以下示例显示如何为bean设置多个别名：

```
@Configuration
public class AppConfig {

    @Bean(name = { "dataSource", "subsystemA-dataSource", "subsystemB-dataSource" })
    public DataSource dataSource() {
        // instantiate, configure and return DataSource bean...
    }
}
```

1.4. 使用 `@Configuration` 注解

`@Configuration`是一个类级别的注解,表明该类将作为bean定义的元数据配置. `@Configuration`类会将有`@Bean`注解的公开方法声明为bean, .在 `@Configuration`类上调用`@Bean`方法也可以用于定义bean间依赖关系.

注入内部bean依赖

当Bean彼此有依赖关系时,表示依赖关系就像调用另一个bean方法一样简单.如下例所示：

```
@Configuration
public class AppConfig {

    @Bean
    public BeanOne beanOne() {
        return new BeanOne(beanTwo());
    }

    @Bean
    public BeanTwo beanTwo() {
        return new BeanTwo();
    }
}
```

在前面的示例中，`beanOne`通过构造函数注入接收对`beanTwo`的引用。

这种声明bean间依赖关系的方法只有在`@Configuration`类中声明`@Bean`方法时才有效。 不能使用普通的`@Component`类声明bean间依赖关系。

有关基于Java的配置如何在内部工作的更多信息

请考虑以下示例，该示例显示了被调用两次的`@Bean`注解方法:

```
@Configuration
public class AppConfig {

    @Bean
    public ClientService clientService1() {
        ClientServiceImpl clientService = new ClientServiceImpl();
        clientService.setClientDao(clientDao());
        return clientService;
    }

    @Bean
    public ClientService clientService2() {
        ClientServiceImpl clientService = new ClientServiceImpl();
        clientService.setClientDao(clientDao());
        return clientService;
    }

    @Bean
    public ClientDao clientDao() {
        return new ClientDaoImpl();
    }
}
```

`clientDao()`在`clientService1()`中调用一次，在`clientService2()`中调用一次。由于此方法创建了`ClientDaoImpl`的新实例并将其返回，因此通常希望有两个实例（每个服务一个）。 这肯定会有问题：在Spring中，实例化的bean默认具有`singleton`范围。这就是它的神奇之处:所有`@Configuration`类在启动时都使用`CGLIB`进行子类化。 在子类中，子方法在调用父方法并创建新实例之前，首先检查容器是否有任何缓存（作用域）bean。

这种行为可以根据bean的作用域而变化,我们这里只是讨论单例.

从Spring 3.2开始，不再需要将CGLIB添加到类路径中，因为CGLIB类已经在`org.springframework.cglib`下重新打包并直接包含在spring-core JAR中。

由于CGLIB在启动时动态添加功能，因此存在一些限制。 特别是，配置类不能是 final的。 但是，从4.3开始，配置类允许使用任何构造函数，包括使用`@Autowired`或单个非默认构造函数声明进行默认注入。

如果想避免因CGLIB带来的限制,请考虑声明非`@Configuration`类的`@Bean`方法，例如在纯的`@Component`类 .这样在`@Bean`方法之间的交叉方法调用将不会被拦截,此时必须在构造函数或方法级别上进行依赖注入。

1.5. 编写基于Java的配置

Spring的基于Java的配置功能允许撰写注解，这可以降低配置的复杂性。

使用`@Import`注解

就像在Spring XML文件中使用`<import/>`元素来帮助模块化配置一样，`@Import`注解允许从另一个配置类加载`@Bean`定义，如下例所示：

```
@Configuration
public class ConfigA {

    @Bean
    public A a() {
        return new A();
    }
}

@Configuration
@Import(ConfigA.class)
public class ConfigB {

    @Bean
    public B b() {
        return new B();
    }
}
```