Spring 的 BeanFactory

作者：thas
原文链接：https://ld246.com/article/1592753524049
来源网站：链滴

基本模型

简单工厂模式 + 策略模式 + Scope

简单工厂模式管理 Bean 的创建, Scope 管理 Bean 的作用域(在哪些场景下无需重复创建).

工厂模式

1. 简单工厂模式: 定义一个工厂类，根据传入的参数不同返回不同的实例，被创建的实例具有共同的类或接口。

2. 工厂方法模式: 定义一个用于创建对象的工厂接口，让子类决定将哪一个类实例化。工厂方法模式一个类的实例化延迟到其子类。

3. 抽象工厂模式: 提供一个创建一系列相关或相互依赖对象的接口，而无须指定它们具体的类。（在象工厂模式中，每一个具体工厂都提供了多个工厂方法用于产生多种不同类型的对象）

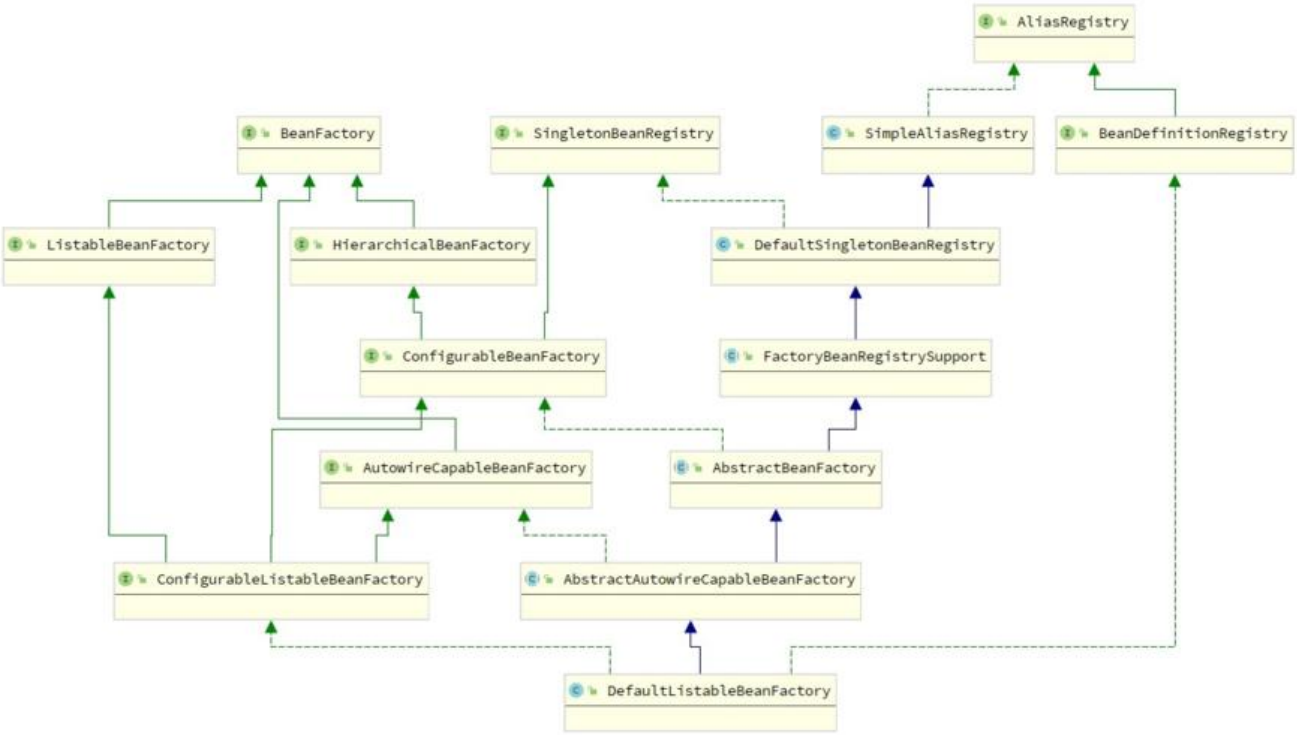
其中 3 是 2 的扩展, 增加了产品族的概念. 与建造者模式类似, 其侧重点在生产对象上.

有许多地方说工厂方法模式是简单工厂的深化, 说是为了解耦, 我认为这是非常不准确的, 没有必要为记住这三个模式刻意将它们串联起来.

简单工厂模式侧重的是主程序与子系统的解耦, 是依赖倒置原则的体现, 它可以看做是最简单 IOC 的现: 主程序依赖简单工厂, 将创建对象的权利交给工厂, 主程序只需要依赖子系统的接口.

但是, 简单工厂模式有个弊端, 就是简单工厂自身会耦合业务子系统, 违反了开闭原则. 比较简单的修改式是通过反射的方式传入类名加载实例类.

Spring 的 BeanFactory 在简单工厂模式的基础上引入了策略模式, 向 BeanFactory 中注册 BeanNa e 和 BeanDefinition, 通过 BeanName 获得想要的 Bean. 实现该功能的接口就是 最基本的 BeanFact ry + BeanDefinitionRegistry.



BeanFactory - 简单工厂模式

BeanFactory 其实就是简单工厂模式, 它只负责提供查找 Bean 实例和 Bean 的部分信息 的方法.

```
public interface BeanFactory {

    String FACTORY_BEAN_PREFIX = "&";

    // 查找 Bean
    Object getBean(String name) throws BeansException;

    Object getBean(String name, Object... args) throws BeansException;

    <T> T getBean(Class<T> requiredType) throws BeansException;

    <T> T getBean(Class<T> requiredType, Object... args) throws BeansException;

    // 查找 Bean 延迟加载
    <T> ObjectProvider<T> getBeanProvider(Class<T> requiredType);

    <T> ObjectProvider<T> getBeanProvider(ResolvableType requiredType);

    // 是否存在 Bean
    boolean containsBean(String name);

    // 查找Bean的其他信息 是否单例 类型 别名
    boolean isSingleton(String name) throws NoSuchBeanDefinitionException;

    boolean isPrototype(String name) throws NoSuchBeanDefinitionException;

    boolean isTypeMatch(String name, ResolvableType typeToMatch) throws NoSuchBeanDefin
```

原文链接：Spring 的 BeanFactory

```
tionException;

    boolean isTypeMatch(String name, Class<?> typeToMatch) throws NoSuchBeanDefinitionE
ception;

    Class<?> getType(String name) throws NoSuchBeanDefinitionException;

    Class<?> getType(String name, boolean allowFactoryBeanInit) throws NoSuchBeanDefiniti
nException;

    String[] getAliases(String name);

}
```

BeanFactory 和寻常的工厂没什么两样, 就是生产对象.

BeanDefinitionRegistry - 策略模式

BeanFactory 是个通用的工厂, 本身并不清楚它可以生产哪些对象, 如何创建这些对象. 这就用到了 BenDefinition, 通过 BeanDefinition 来定义 Bean 的创建方法, 注册 BeanName 和 BeanDefinition 告诉 BeanFactory 可以生产哪些对象.

```
public interface BeanDefinitionRegistry extends AliasRegistry {

    // 注册 BeanDefinition
    void registerBeanDefinition(String beanName, BeanDefinition beanDefinition)
            throws BeanDefinitionStoreException;

    void removeBeanDefinition(String beanName) throws NoSuchBeanDefinitionException;

    BeanDefinition getBeanDefinition(String beanName) throws NoSuchBeanDefinitionExcepti
n;

    boolean containsBeanDefinition(String beanName);

    String[] getBeanDefinitionNames();

    int getBeanDefinitionCount();

    boolean isBeanNameInUse(String beanName);

}
```

BeanDefinition一共有4种方式创建 Bean:

```
直接提供实例类 根据构造方法创建实例
<bean id="bean1" class="cc.thas.spring.Test" />

提供 FactoryBean 自动根据接口中的 getObject 方法创建实例
<bean id="bean2" class="cc.thas.spring.TestFactoryBean" />
```

```
提供自定义的 Factory 类 根据 factory-method 指定的静态方法创建实例
<bean id="bean3" class="cc.thas.spring.TestFacotry" factory-method="staticMethod" />

提供 自定义 Factory 实例 根据 factory-method 指定的实例方法创建实例
<bean id="bean4factory" class="cc.thas.spring.TestFactory"/>
<bean id="bean4" factory-bean="bean4factory" factory-method="nonStaticMethod"/>
```

为了统一 Bean 的这些创建方式, 高层的 BeanFactory 实现提供 ObjectFactory 这样的一个统一接口, 这样需要使用 Bean 的创建方法的地方 (如 Scope) 只要使用 ObjectFactory.getObject() 方法就能创 Bean.

Scope - 作用域/容器

普通的工厂模式, 只提供创建实例的方法, 并不具备容器这个功能, 每次调用都是得到新的对象, 也就相于 Spring 中的原型作用域. Spring 的 BeanFactory 提供了容器功能, 允许在特定的作用域内返回同个, 例如:

1. 单例: 同一个 BeanFactory 内, 只返回唯一的一个 Bean 实例
2. 原型: 没有作用域, 每次查询对象都返回新的对象
3. RequestScope: 依赖 Servlet, 同一个 Http 请求上下文上, 只返回同一个 Bean.
4. SessionScope: 依赖 Servlet, 同一个 HttpSession 中, 只返回同一个 Bean.
5. ApplicationScope: 依赖 Servlet, 同一个 ServletContext中, 只返回同一个 Bean. 与单例作用域非相似.

如上所示, Scope 虽然定义的是 Bean 的作用域, 但是也相当于是管理了容器. 要判断当前作用域, 判断创建新对象, 还是从容器中找出当前作用域的已经创建的对象.

RequestScope 使用 ServletRequest 的 Attribute 做了容器, SessionScope 和 ApplicationScope 理, 所以它们是重度依赖 Servlet 的.

BeanFactory.getBean 的简单实现:

```
String scopeName = mbd.getScope();
final Scope scope = this.scopes.get(scopeName);
if (scope == null) {
  throw new IllegalStateException("No Scope registered for scope name '" + scopeName + "'")

}
Object scopedInstance = scope.get(beanName, () -> {
    beforePrototypeCreation(beanName);
    try {
        return createBean(beanName, mbd, args);
    }
    finally {
        afterPrototypeCreation(beanName);
    }
});
```

BeanFactory 把创建对象的方法包装成 ObjectFactory, 交给 Scope, 把对象的创建权利和时机交给了cope 去做, Scope 根据自己的需要去定义容器.

BeanFactory + BeanDefinitionRegistry + Scope 就实现 Spring 中最基本的 IOC 容器.

深度定制化容器

定制化单例和原型作用域

原型作用域其实根本就没有作用域的概念, 直接代码写死, 每次获取对象都是创建新对象.

开发过程中, 绝大部分的业务对象都是单例的对象, Spring 自己定义 Servlet 相关的那些 Scope, 就像给 Scope 的使用方式打了个 Demo, 没什么人真正用过.

Spring 对这个单例作用域下了狠功夫, 没有使用 Scope 来实现容器, 而是在 BeanFactory 的基础上续扩展, 做特殊定制. 承担这个容器功能的接口就是 SingletonBeanRegistry, 与 Scope 类似, BeanFac ory 将 ObjectFactory 交给 SingletonBeanRegistry, 让它决定创建 Bean 的时机, SingletonBeanReg stry.getSingleton 会在创建完对象, 将实例保存为单例, 放在内部属性 singletonObjects 中.

SingletonBeanRegistry - 定制单例 Bean 容器

其对应的实现是 DefinitionSingletonBeanRegistry, 抛开 BeanFactory 这个接口是可以单独使用的. efinitionSingletonBeanRegistry 与 BeanDefinition 是无关的, 它就是一个保存单例对象的容器, 它关注 BeanName 和 已创建的 Bean 实例.

```
public interface SingletonBeanRegistry {

    // 单例 Bean 的集合
    private final Map<String, Object> singletonObjects = new ConcurrentHashMap<>(256);

    // BeanName 的名称 有顺序
    private final Set<String> registeredSingletons = new LinkedHashSet<>(256);

    // 含销毁方法的 Bean
    private final Map<String, Object> disposableBeans = new LinkedHashMap<>();

    void registerSingleton(String beanName, Object singletonObject);

    Object getSingleton(String beanName);

    boolean containsSingleton(String beanName);

    String[] getSingletonNames();

    int getSingletonCount();

    Object getSingletonMutex();

}
```

AutowireCapableBeanFactory - 依赖注入

BeanFactory 本身支持依赖查找 (DL), 如果 Bean 之间存在依赖关系, 那么所有的依赖都需要自己调用 API 再去 BeanFactory 中查找获得. AutowireCapableBeanFactory 扩展了依赖注入功能 (DI).

属性填充和其他对 Bean 的后置处理也是由这个接口实现的.

```
public interface AutowireCapableBeanFactory extends BeanFactory {

    int AUTOWIRE_NO = 0;

    int AUTOWIRE_BY_NAME = 1;

    int AUTOWIRE_BY_TYPE = 2;

    int AUTOWIRE_CONSTRUCTOR = 3;

    String ORIGINAL_INSTANCE_SUFFIX = ".ORIGINAL";

    <T> T createBean(Class<T> beanClass) throws BeansException;

    void autowireBean(Object existingBean) throws BeansException;

    Object configureBean(Object existingBean, String beanName) throws BeansException;

    Object createBean(Class<?> beanClass, int autowireMode, boolean dependencyCheck) thr
ws BeansException;

    Object autowire(Class<?> beanClass, int autowireMode, boolean dependencyCheck) throw
 BeansException;

    void autowireBeanProperties(Object existingBean, int autowireMode, boolean dependency
heck)
            throws BeansException;

    void applyBeanPropertyValues(Object existingBean, String beanName) throws BeansExcept
on;

    Object initializeBean(Object existingBean, String beanName) throws BeansException;

    Object applyBeanPostProcessorsBeforeInitialization(Object existingBean, String beanName)
            throws BeansException;

    Object applyBeanPostProcessorsAfterInitialization(Object existingBean, String beanName)
            throws BeansException;

    void destroyBean(Object existingBean);

    <T> NamedBeanHolder<T> resolveNamedBean(Class<T> requiredType) throws BeansExc
ption;

    Object resolveBeanByName(String name, DependencyDescriptor descriptor) throws BeansE
ception;

    Object resolveDependency(DependencyDescriptor descriptor, @Nullable String requesting
eanName) throws BeansException;

    Object resolveDependency(DependencyDescriptor descriptor, @Nullable String requesting
eanName,
```

```
        @Nullable Set<String> autowiredBeanNames, @Nullable TypeConverter typeConvert
r) throws BeansException;

}
```

其他功能性增强

AliasRegistry - 别名支持

HierarchicalBeanFactory - 层次性 BeanFactory

允许定义父 BeanFactory.

ConfigurableBeanFactory - 可配置的BeanFactory

通过它修改 BeanFactory

ListableBeanFactory - 列举支持

可以枚举 BeanFactory 元素, 可以依赖查找到集合 (允许查找到多个 Bean 并自动转换成集合类).

ConfigurableListableBeanFactory - 可配置的 ListableBeanFactory.