Spring如何根据BeanDefinition实例化为bean，并管理这些bean?

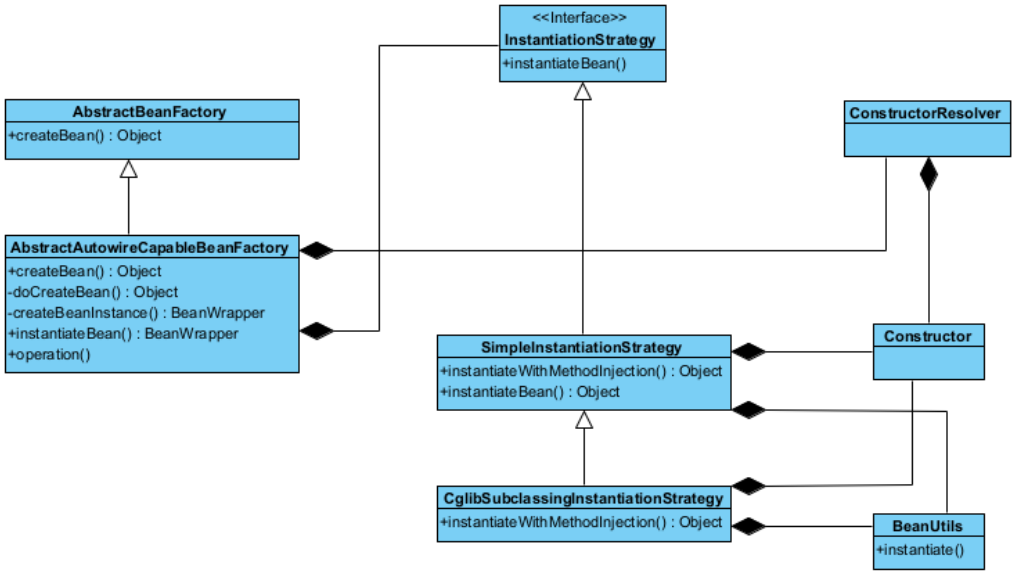
一、实例化bean

下面是XML配置的Car，Spring是如何生成Car实例的呢?

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="car" class="**.Car" scope="prototype"/>

</beans>
```

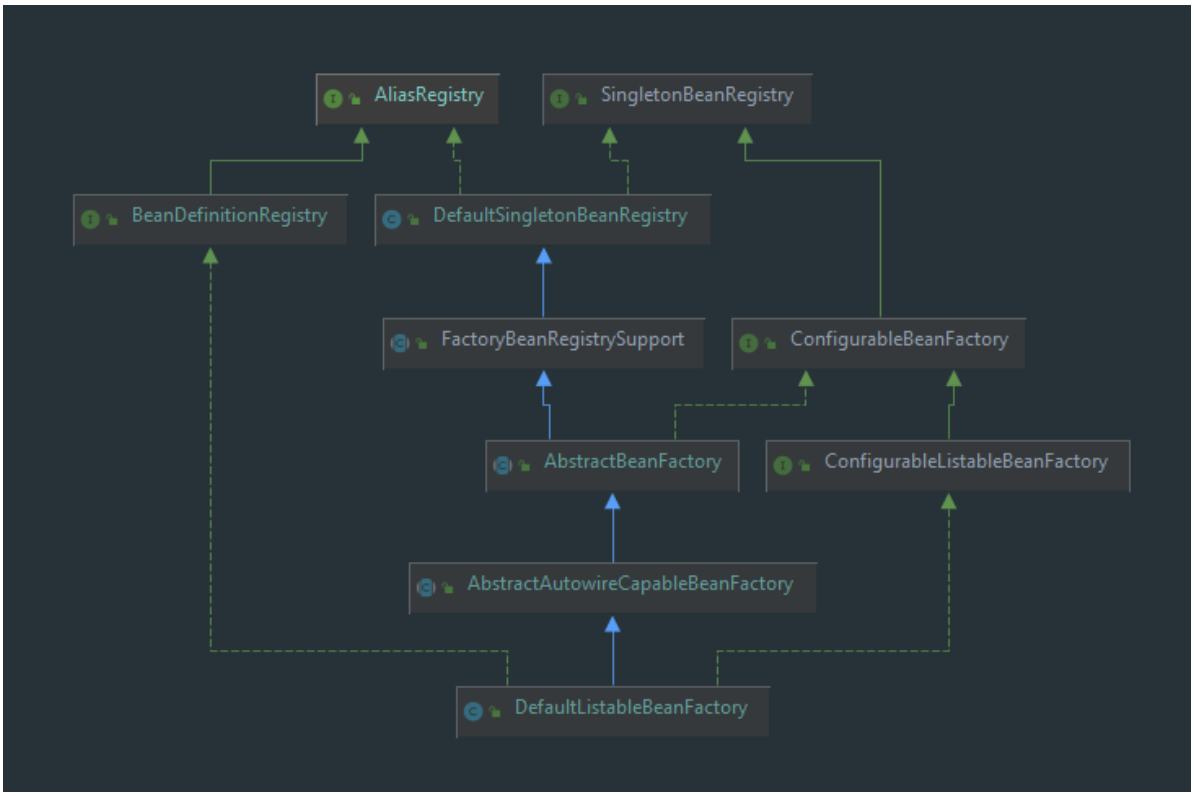
```
public class Car {

    private String carNum;

    public void setCarNum(String carNum) {
        this.carNum = carNum;
    }

    public String getCarNum() {
        return carNum;
    }
}

public class LoadBeanPlay {

    public static void main(String[] args){
        // 获得的容器container已经加载好BeanDefinition
        BeanFactory container = new XmlBeanFactory(new
ClassPathResource("scope.xml"));
        Car car = container.getBean("car", Car.class);
    }

}
```

bean的实例化（不包括其他的管理）的工作由以下类共同完成。



二、管理实例化后的bean

下面的类主要是对实例化后的bean进行管理，如注册、获取、销毁。



其中BeanDefinitionRegistry持有的是BeanDefinition,它是生产bean的原材料供应商。真正产生bean的是SingletonBeanRegistry。真正创建bean实例的步骤在AbstractAutowireCapableBeanFactory中，它会调用AbstractAutoProxyCreator的createProxy方法，来生成一个bean的代理，因为引入了postBeanProcessor机制。最后，AbstractAutowireCapableBeanFactory会把生成的bean保存到DefaultSingletonBeanRegistry的singletonObjects中。

1.SingletonBeanRegistry

```
public interface SingletonBeanRegistry {

    // 把单例以<beanName, bean>的形式注册起来
    void registerSingleton(String beanName, Object singletonObject);
```

```
    @Nullable
    // 通过名字获取实例
    Object getSingleton(String beanName);

    // beanName是否已经被注册
    boolean containsSingleton(String beanName);

    // 获取所有被注册的beanName
    String[] getSingletonNames();

    // 当前被注册的数量
    int getSingletonCount();

    // 返回此注册表使用的单例互斥锁（用于外部协作者）。
    Object getSingletonMutex();

}
```

2.DefaultSingletonBeanRegistry，这个类主要实现了对单例bean实例CRUD，当然了，这个C，是指别人create后，然后交给它来管理的。

```
public class DefaultSingletonBeanRegistry extends SimpleAliasRegistry implements
SingletonBeanRegistry {

    /** Maximum number of suppressed exceptions to preserve. */
    private static final int SUPPRESSED_EXCEPTIONS_LIMIT = 100;


    /** Cache of singleton objects: bean name to bean instance. */
    // 容器内部当前已注册的单例bean
    private final Map<String, Object> singletonObjects = new ConcurrentHashMap<>
(256);

    /** Cache of singleton factories: bean name to ObjectFactory. */
    // 容器内部当前已注册的单例工厂
    private final Map<String, ObjectFactory<?>> singletonFactories = new
HashMap<>(16);

    /** Cache of early singleton objects: bean name to bean instance. */
    //
    private final Map<String, Object> earlySingletonObjects = new
ConcurrentHashMap<>(16);

    /** Set of registered singletons, containing the bean names in registration
order. */
    // 按注册顺序，保存已经注册的单例（bean和bean工厂）名字
    private final Set<String> registeredSingletons = new LinkedHashSet<>(256);

    /** Names of beans that are currently in creation. */
    // 当前正在创建的bean的名称。
    private final Set<String> singletonsCurrentlyInCreation =
            Collections.newSetFromMap(new ConcurrentHashMap<>(16));

    /** Names of beans currently excluded from in creation checks. */
    // 当前从创建检查中排除的bean的名称
    private final Set<String> inCreationCheckExclusions =
```

```
            Collections.newSetFromMap(new ConcurrentHashMap<>(16));

    /** Collection of suppressed Exceptions, available for associating related
causes. */
    @Nullable
    private Set<Exception> suppressedExceptions;

    /** Flag that indicates whether we're currently within destroySingletons. */
    // 指示我们当前是否在destroySingletons中的标志。
    private boolean singletonsCurrentlyInDestruction = false;

    /** Disposable bean instances: bean name to disposable instance. */
    // 一次性 bean 实例：<beanName : 一次性实例>。
    private final Map<String, Object> disposableBeans = new LinkedHashMap<>();

    /** Map between containing bean names: bean name to Set of bean names that
the bean contains. */
    // 包含 bean 名称之间的映射：bean 名称到 bean 包含的 bean 名称集。
    private final Map<String, Set<String>> containedBeanMap = new
ConcurrentHashMap<>(16);

    /** Map between dependent bean names: bean name to Set of dependent bean
names. */
    // 依赖 bean 名称之间的映射,bean name to 一组依赖 bean 名称。
    // A依赖B和C，D依赖B，那么在dependentBeanMap中保存的是<b,{a,d}>,<c,{a}>
    // 所以它的key是被依赖主体，value是依赖主体的集合，可以看到某个对象被哪些对象依赖
    // 举个例子，可以通过debug来确定。
    private final Map<String, Set<String>> dependentBeanMap = new
ConcurrentHashMap<>(64);

    /** Map between depending bean names: bean name to Set of bean names for the
bean's dependencies. */
    // A依赖B和C，D依赖B，那么在dependenciesForBeanMap中保存的是<a,{b,c}>,<d,{b}>
    // 所以它的key是依赖主体，value是被依赖主体的集合，可以看到某个对象依赖哪些对象。
    // 举个例子，可以通过debug来确定。
    private final Map<String, Set<String>> dependenciesForBeanMap = new
ConcurrentHashMap<>(64);


}
```

小疑惑：ConfigurableBeanFactory接口的destroySingletons()方法，被DefaultSingletonBeanRegistry实现，但为什么DefaultSingletonBeanRegistry的代码中没有implements ConfigurableBeanFactory?
解答：因为AbstractBeanFactory实现了ConfigurableBeanFactory的destroySingletons方法，并且间接继承了DefaultSingletonBeanRegistry类，而AbstractBeanFactory的destroySingletons方法是调用了super.destroySingletons()的，所以destroySingletons的真正实现还是在DefaultSingletonBeanRegistry中，因此可以说DefaultSingletonBeanRegistry实现了ConfigurableBeanFactory接口的某些方法。子类实现一个接口，但是这个接口的真正实现可以在父类中。

FactoryBeanRegistrySupport：这个类在继承DefaultSingletonBeanRegistry基础上，还增加一个功能，就是对那些由FactoryBean产生的单例对象的管理

```
public abstract class FactoryBeanRegistrySupport extends
DefaultSingletonBeanRegistry {
```

```
    // FactoryBeans 创建的单例对象的缓存：FactoryBean名称到对象。
    private final Map<String, Object> factoryBeanObjectCache = new
ConcurrentHashMap<>(16);


    @Override
    protected void removeSingleton(String beanName) {
        synchronized (getSingletonMutex()) {
            // 把父容器的单例bean移除
            super.removeSingleton(beanName);
            // 把自身携带的由FactoryBean产生的单例移除
            this.factoryBeanObjectCache.remove(beanName);
        }
    }


    @Override
    protected void clearSingletonCache() {
        synchronized (getSingletonMutex()) {
            super.clearSingletonCache();
            this.factoryBeanObjectCache.clear();
        }
    }
}
```

AbstractBeanFactory：这个类管理着容器产生的bean，它里面有个抽象方法createBean，当查找父容器和自身后，都没有找到合适的bean实例，此时，就会调用createBean，这个方法由它的子类AbstractAutowireCapableBeanFactory来实现，这一步会和步骤一进行接轨。它们俩共同完成bean的管理。