Spring Framework 开发参考手册.chm

课程概要：

1. IOC核心知识点回顾
2. IOC 设计原理

一、IOC核心理论回顾

知识点：

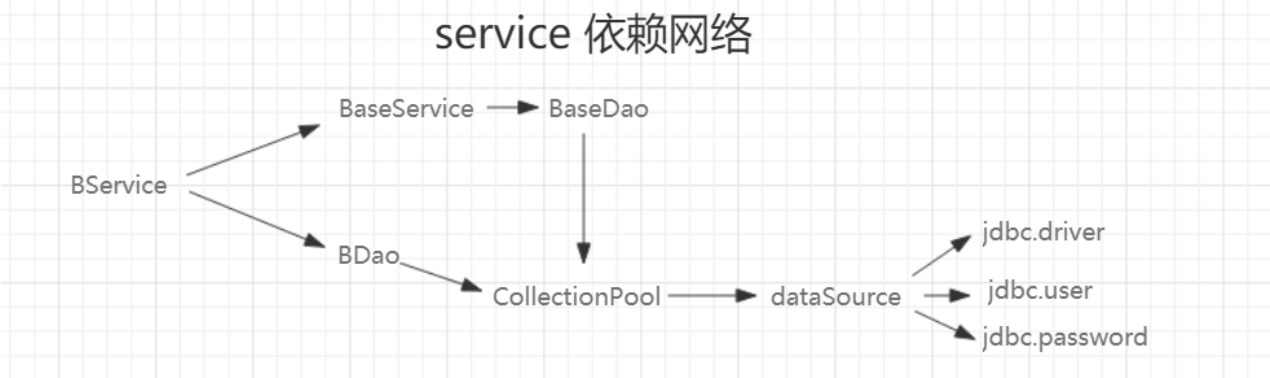
1. Ioc理念概要
2. 实体Bean的创建
3. Bean的基本特性
4. 依赖注入
5. set方法注入
6. 构造方法注入
7. 自动注入(byName、byType)
8. 依赖检测

1、Ioc理论概要

在JAVA的世界中，一个对象A怎么才能调用对象B？通常有以下几种方法。

类别	描述	时间点
外部传入	构造方法传入	
	属性设置传入	设置对象状态时
	运行时做为参数传入	调用时
内部创建	属性中直接创建	创建引用对象时
	初始化方法创建	创建引用对象时
	运行时动态创建	调用时

上表可以看到， 引用一个对象可以在不同地点（其它引用者）、不同时间由不同的方法完成。如果B只是一个非常简单的对象 如直接new B()，怎样都不会觉得复杂，比如你从来不会觉得创建一个String 是一个件复杂的事情。但如果B 是一个有着复杂依赖的Service对象，这时在不同时机引用B将会变得很复杂。



无时无刻都要维护B的复杂依赖关系，试想B对象如果项目中有上百过，系统复杂度将会成陪数增加。IOC容器 的出现正是为解决这一问题，其可以将对象的构建方式统一，并且自动维护对象的依赖关系，从而降低系统的实现成本。前提是需要提前对目标对象基于XML进行声明。

2、实体Bean的构建

1. 基于Class构建
2. 构造方法构建
3. 静态工厂方法创建
4. FactoryBean创建

1、基于ClassName构建

```
<bean class="com.duo.spring.HelloSpring"></bean>
```

这是最常规的方法，其原理是在spring底层会基于class 属性 通过反射进行构建。

2、构造方法构建

```
<bean class="com.duo.spring.HelloSpring">
    <constructor-arg name="name" type="java.lang.String" value="luban"/>
    <constructor-arg index="1" type="java.lang.String" value="sex" />
</bean>
```

如果需要基于参数进行构建，就采用构造方法构建，其对应属性如下：
name:构造方法参数变量名称
type:参数类型
index:参数索引，从0开始
value:参数值，spring 会自动转换成参数实际类型值
ref:引用容串的其它对象

3、静态工厂方法创建

```
<bean class="com.duo.spring.HelloSpring" factory-method="build">
    <constructor-arg name="type" type="java.lang.String" value="B"/>
</bean>
```

如果你正在对一个对象进行A/B测试，就可以采用静态工厂方法的方式创建，其于策略创建不同的对像或填充不同的属性。
该模式下必须创建一个静态工厂方法，并且方法返回该实例，spring 会调用该静态方法创建对象。

```
public static HelloSpring build(String type) {
    if (type.equals("A")) {
        return new HelloSpring("luban", "man");
    } else if (type.equals("B")) {
        return new HelloSpring("diaocan", "woman");
    } else {
        throw new IllegalArgumentException("type must A or B");
    }
}
```

4、FactoryBean创建

```
<bean class="com.duo.spring.LubanFactoryBean" id="helloSpring123"></bean>
```

指定一个Bean工厂来创建对象，对象构建初始化 完全交给该工厂来实现。配置Bean时指定该工厂类的类名。

```
public class LubanFactoryBean implements FactoryBean {
    @Override
    public Object getObject() throws Exception {
        return new HelloSpring();
    }
    @Override
    public Class<?> getObjectType() {
        return HelloSpring.class;
    }
    @Override
    public boolean isSingleton() {
        return false;
    }
}
```

3、bean的基本特性

- 作用范围
- 生命周期
- 装载机制

a、作用范围
很多时候Bean对象是无状态的，而有些又是有状态的 无状态的对象我们采用单例即可，而有状态则必须是多例的模式，通过scope 即可创建
scope=“prototype”
scope=“singleton”

```
scope=“prototype
<bean class="com.duo.spring.HelloSpring" scope="prototype">
</bean>
```

如果一个Bean设置成 prototype 我们可以 通过BeanFactoryAware 获取 BeanFactory 对象即可每次获取的都是新对像。

b、生命周期
Bean对象的创建、初始化、销毁即是Bean的生命周期。通过 init-method、destroy-method 属性可以分别指定期构建方法与初始方法。

```
<bean class="com.duo.spring.HelloSpring" init-method="init" destroy-
method="destroy"></bean>
```

如果觉得麻烦，可以让Bean去实现 InitializingBean.afterPropertiesSet()、DisposableBean.destroy() 方法。分别对应 初始和销毁方法

c、加载机制
指示Bean在何时进行加载。设置lazy-init 即可，其值如下：
true: 懒加载，即延迟加载
false:非懒加载，容器启动时即创建对象
default:默认，采用default-lazy-init 中指定值，如果default-lazy-init 没指定就是false

```
<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"
default-lazy-init="true">
```

什么时候使用懒加载?
懒加载会容器启动的更快，而非懒加载可以容器启动时更快的发现程序当中的错误 ，选择哪一个就看追求的是启动速度，还是希望更早的发现错误，一般我们会选 择后者。

4、依赖注入

试想IOC中如果没有依赖注入，那这个框架就只能帮助我们构建一些简单的Bean，而之前所说的复杂Bean的构建问题将无法解决，spring这个框架不可能会像现在这样成功。 spring 中 ioc 如何依赖注入呢。有以下几种方式：

1. set方法注入
2. 构造方法注入
3. 自动注入(byName、byType)
4. 方法注入(lookup-method)

2、set方法注入

```
<bean class="com.duo.spring.HelloSpring">
    <property name="fine" ref="fineSpring"/>
</bean>
```

3、构造方法注入

```
<bean class="com.duo.spring.HelloSpring">
    <constructor-arg name="fine">
        <bean class="com.duo.spring.FineSpring"/>
    </constructor-arg>
</bean>
```

4、自动注入（byName\byType\constructor)

```
<bean id="helloSpringAutowireConstructor" class="com.duo.spring.HelloSpring"
autowire="byName">
</bean>
```

byName：基于变量名与bean 名称相同作为依据插入
byType：基于变量类别与bean 名称作
constructor：基于IOC中bean 与构造方法进行匹配（语义模糊，不推荐）

5、依赖方法注入(lookup-method)
当一个单例的Bean，依赖于一个多例的Bean，用常规方法只会被注入一次，如果每次都想要获取一个全新实例就可以采用lookup-method 方法来实现。

```
#编写一个抽像类
public abstract class MethodInject {
    public void handlerRequest() {
      // 通过对该抽像方法的调用获取最新实例
        getFine();
    }
    # 编写一个抽像方法
    public abstract FineSpring getFine();
}
// 设定抽像方法实现
<bean id="MethodInject" class="com.duo.spring.MethodInject">
    <lookup-method name="getFine" bean="fine"></lookup-method>
</bean>
```

该操作的原理是基于动态代理技术，重新生成一个继承至目标类，然后重写抽像方法到达注入目的。
前面说所单例Bean依赖多例Bean这种情况也可以通过实现 ApplicationContextAware 、BeanFactoryAware 接口来获取BeanFactory 实例，从而可以直接调用getBean方法获取新实例，推荐使用该方法，相比lookup-method语义逻辑更清楚一些。

二、IOC 设计原理与实现

知识点：

1、源码学习的目标
2、Bean的构建过程
3、BeanFactory与ApplicationContext区别

1、源码学习目标：

不要为了读书而读书，同样不要为了阅读源码而读源码。没有目的一头扎进源码的黑森林当中很快就迷路了。到时就不是我们读源码了，而是源码'毒'我们。毕竟一个框架是由专业团队，历经N次版本迭代的产物，我们不能指望像读一本书的方式去阅读它。 所以必须在读源码之前找到目标。是什么呢?
大家会想，读源码的目标不就是为了学习吗？这种目标太过抽像，目标无法验证。通常我们会设定两类型目标：一种是对源码进行改造，比如添加修改某些功能，在实现这种目标的过程当中自然就会慢慢熟悉了解该项目。但然这个难度较大，耗费的成本也大。另一个做法是 自己提出一些问题，阅读源码就是为这些问题寻找答案。以下就是我们要一起在源码中寻找答案的问题：

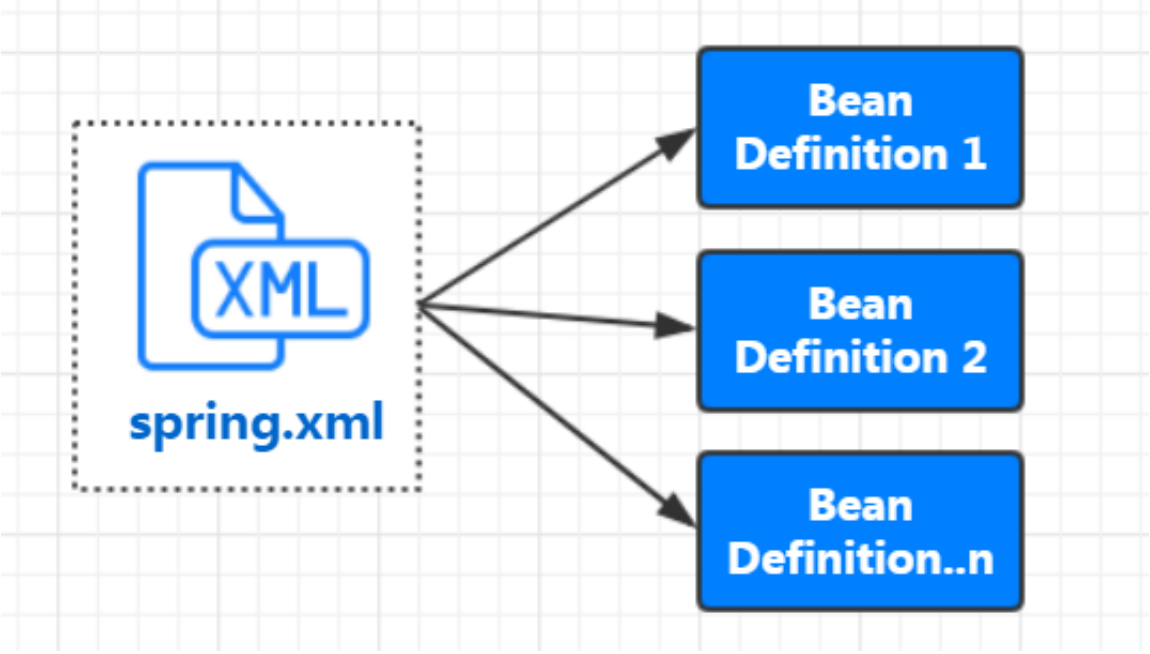
1. Bean工厂是如何生产Bean的?
2. Bean的依赖关系是由谁解来决的?
3. Bean工厂和应用上文的区别?

2、Bean的构建过程

spring.xml 文件中保存了我们对Bean的描述配置，BeanFactory 会读取这些配置然后生成对应的Bean。这是我们对ioc 原理的一般理解。但在深入一些我们会有更多的问题?

1. 配置信息最后是谁JAVA中哪个对象承载的?
2. 这些承载对象是谁业读取XML文件并装载的?
3. 这些承载对象又是保存在哪里?

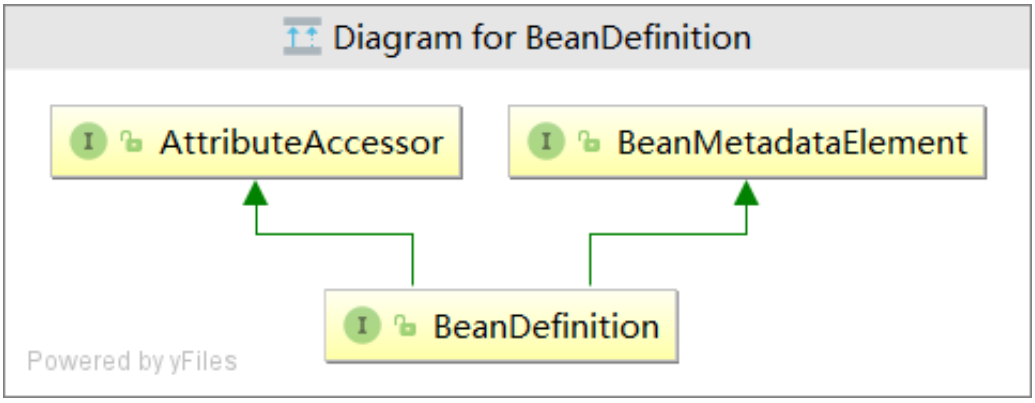
BeanDefinition （Bean定义）
ioc 实现中 我们在xml 中描述的Bean信息最后 都将保存至BeanDefinition （定义）对象中，其中xml bean 与BeanDefinition 程一对一的关系。



由此可见，xml bean中设置的属性最后都会体现在BeanDefinition中。如:

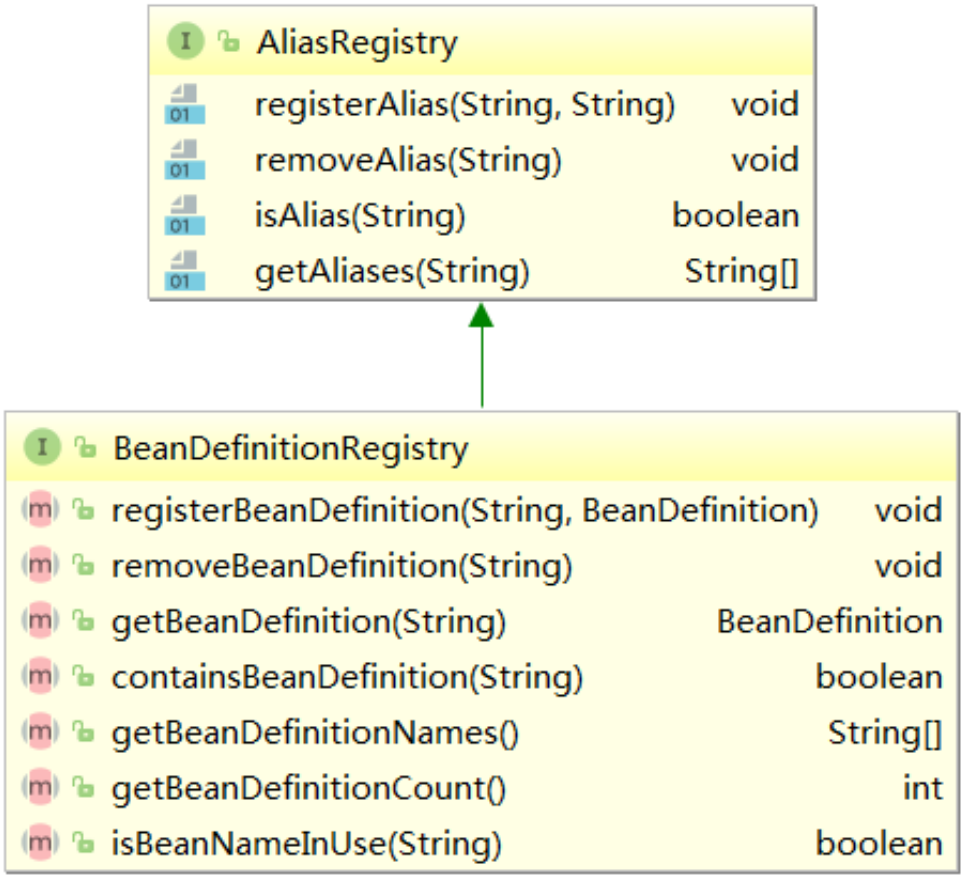
XML-bean	BeanDefinition
class	beanClassName
scope	scope
lazy-init	lazyInit
constructor-arg	ConstructorArgument
property	MutablePropertyValues
factory-method	factoryMethodName
destroy-method	AbstractBeanDefinition.destroyMethodName
init-method	AbstractBeanDefinition.initMethodName
autowire	AbstractBeanDefinition.autowireMode
id	
name	

- [ ] 演示查看 BeanDefinition 属性结构



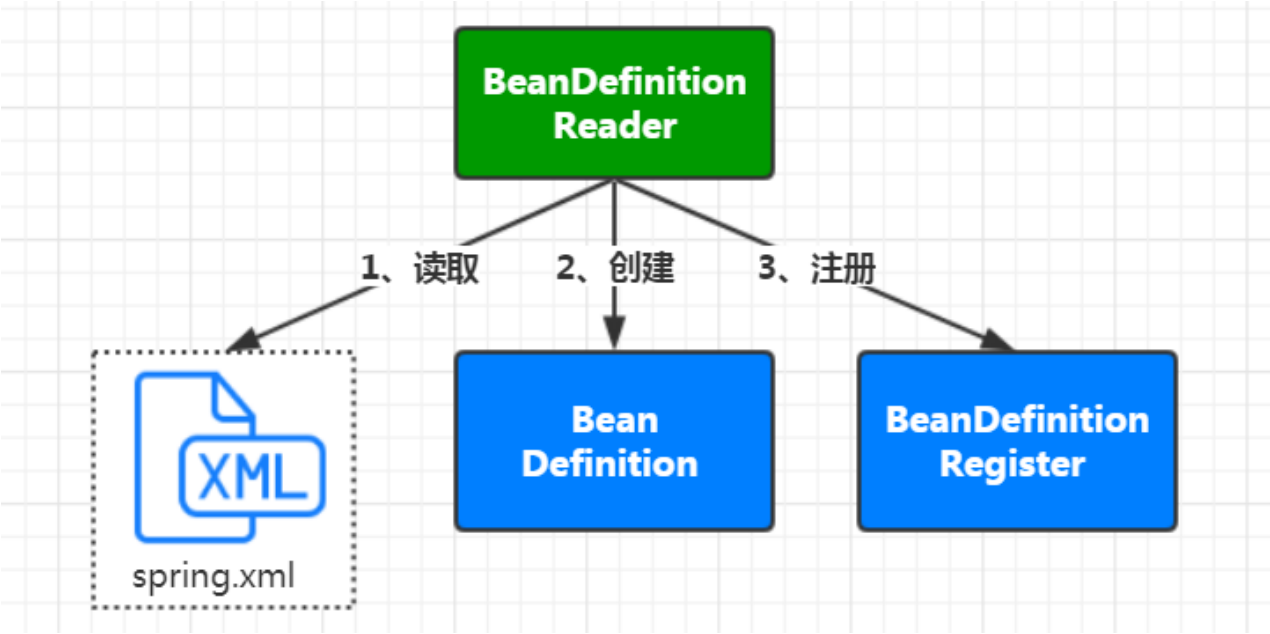
BeanDefinitionRegistry（Bean注册器）
在上表中我们并没有看到 xml bean 中的 id 和name属性没有体现在定义中，原因是ID 其作为当前Bean的存储key注册到了BeanDefinitionRegistry 注册器中。name 作为别名key 注册到了AliasRegistry 注册中心。其最后都是指向其对应的BeanDefinition。

- [ ] 演示查看 BeanDefinitionRegistry属性结构



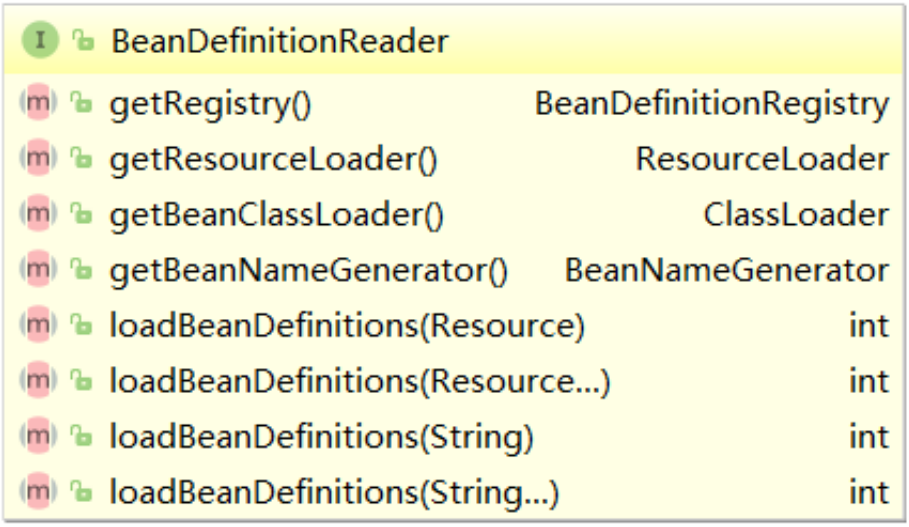
BeanDefinitionReader（Bean定义读取）

至此我们学习了 BeanDefinition 中存储了Xml Bean信息，而BeanDefinitionRegister 基于ID和name保存了Bean的定义。接下要学习的是从xml Bean到BeanDefinition 然后在注册至BeanDefinitionRegister 整个过程。



上图中可以看出Bean的定义是由BeanDefinitionReader 从xml 中读取配置并构建出BeanDefinitionReader,然后在基于别名注册到BeanDefinitionRegister中。

- [ ] 查看BeanDefinitionReader结构



方法说明：

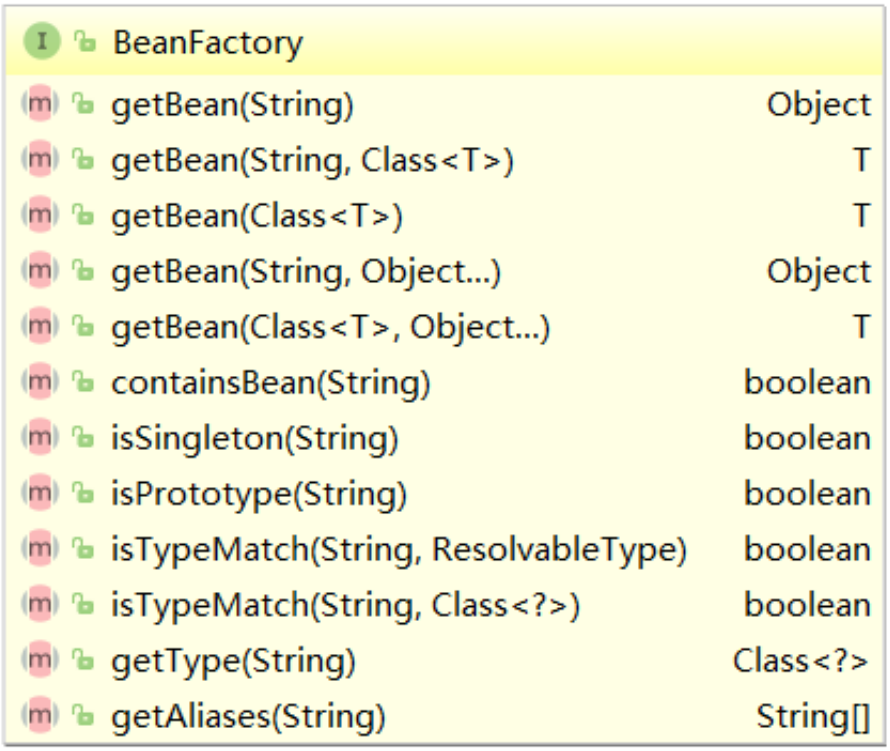
- **loadBeanDefinitions(Resource resource)**
  - 基于资源装载Bean定义并注册至注册器
- **int loadBeanDefinitions(String location)**
  - 基于资源路径装载Bean定义并注册至注册器
- **BeanDefinitionRegistry getRegistry()**
  - 获取注册器
- **ResourceLoader getResourceLoader()**
  - 获取资源装载器

- [ ] 基于示例演示BeanDefinitionReader装载过程

```
//创建一个简单注册器
BeanDefinitionRegistry register = new SimpleBeanDefinitionRegistry();
//创建bean定义读取器
BeanDefinitionReader reader = new XmlBeanDefinitionReader(register);
// 创建资源读取器
DefaultResourceLoader resourceLoader = new DefaultResourceLoader();
// 获取资源
Resource xmlResource = resourceLoader.getResource("spring.xml");
// 装载Bean的定义
reader.loadBeanDefinitions(xmlResource);
// 打印构建的Bean 名称
System.out.println(Arrays.toString(register.getBeanDefinitionNames()));
```

Beanfactory(bean 工厂)

有了Bean的定义就相当于有了产品的配方，接下来就是要把这个配方送到工厂进行生产了。在ioc当中Bean的构建是由BeanFactory 负责的。其结构如下：



方法说明：

- **getBean(String)**
  - 基于ID或name 获取一个Bean
- **T getBean(Class requiredType)**
  - 基于Bean的类别获取一个Bean（如果出现多个该类的实例，将会报错。但可以指定primary="true" 调整优先级来解决该错误 ）
- **Object getBean(String name, Object... args)**
  - 基于名称获取一个Bean，并覆盖默认的构造参数
- **boolean isTypeMatch(String name, Class<?> typeToMatch)**
  - 指定Bean与指定Class 是否匹配

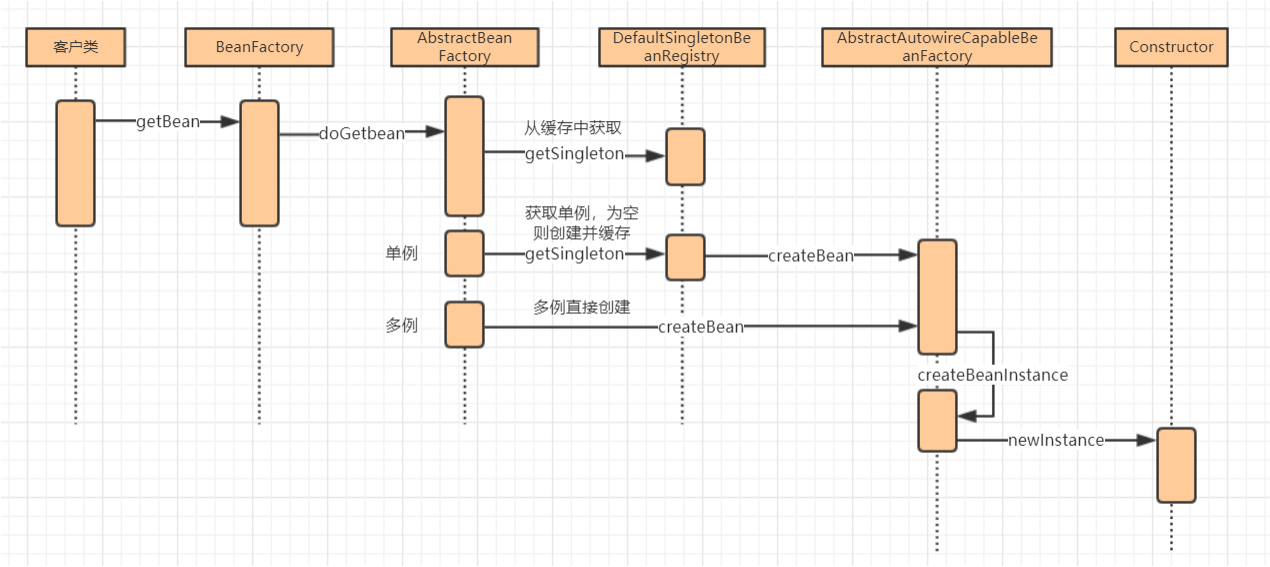
以上方法中重点要关注getBean，当用户调用getBean的时候就会触发 Bean的创建动作，其是如何创建的呢?

- [ ] 演示基本BeanFactory获取一个Bean

```
#创建Bean堆栈
// 其反射实例化Bean
java.lang.reflect.Constructor.newInstance(Unknown Source:-1)
BeanUtils.instantiateClass()
//基于实例化策略 实例化Bean
SimpleInstantiationStrategy.instantiate()
AbstractAutowireCapableBeanFactory.instantiateBean()
// 执行Bean的实例化方法
```

```
AbstractAutowireCapableBeanFactory.createBeanInstance()
AbstractAutowireCapableBeanFactory.doCreateBean()
// 执行Bean的创建
AbstractAutowireCapableBeanFactory.createBean()
// 缓存中没有，调用指定Bean工厂创建Bean
AbstractBeanFactory$1.getObject()
// 从单例注册中心获取Bean缓存
DefaultSingletonBeanRegistry.getSingleton()
AbstractBeanFactory.doGetBean()
// 获取Bean
AbstractBeanFactory.getBean()
// 调用的客户类
com.duo.spring.BeanFactoryExample.main()
```

Bean创建时序图：

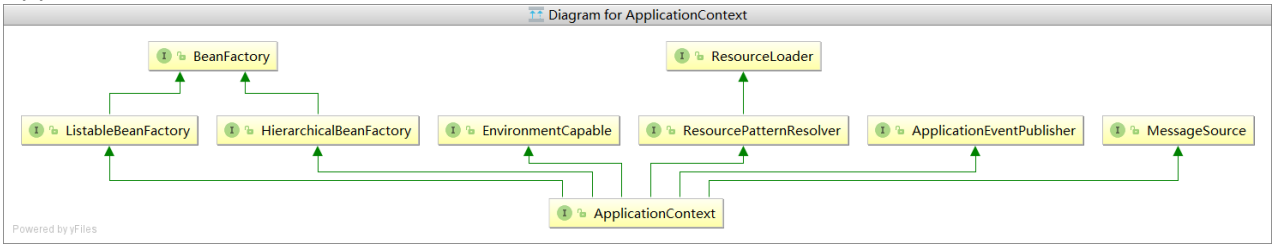


从调用过程可以总结出以下几点：

1. 调用BeanFactory.getBean() 会触发Bean的实例化。
2. DefaultSingletonBeanRegistry 中缓存了单例Bean
3. Bean的创建与初始化是由AbstractAutowireCapableBeanFactory 完成的。

3、BeanFactory 与 ApplicationContext区别

BeanFactory 看下去可以去做IOC当中的大部分事情，为什么还要去定义一个ApplicationContext 呢?
ApplicationContext 结构图



从图中可以看到 ApplicationContext 它由BeanFactory接口派生而来，因而提供了BeanFactory所有的功能。除此之外context包还提供了以下的功能：

1. MessageSource, 提供国际化的消息访问
2. 资源访问，如URL和文件
3. 事件传播，实现了ApplicationListener接口的bean
4. 载入多个（有继承关系）上下文 ，使得每一个上下文都专注于一个特定的层次，比如应用的web层