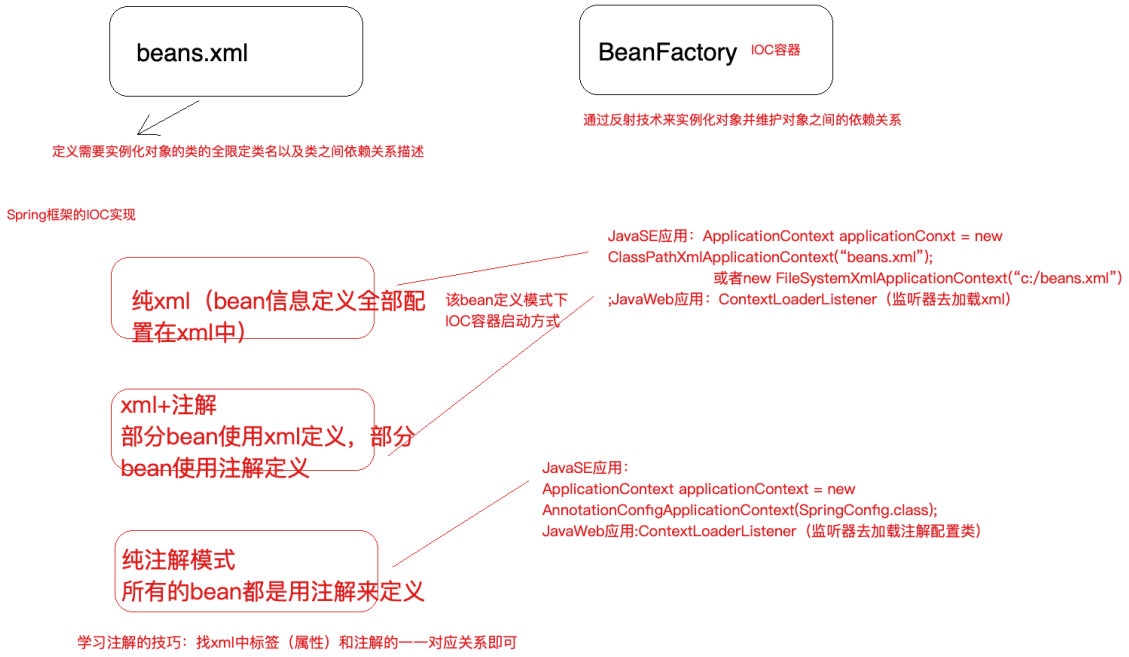
Fix Unresolved Plugin under Maven projects:

https://gist.github.com/kdevo/05542ea81905d12199019699962171de

第四部分 Spring IoC 应用

1. 基础

1.1 Spring 框架的IoC实现有3种模式



Spring 框架的IOC实现有3种模式：

1. **纯xml** （所有bean定义在xml中）
2. **xml + 注解** （部分bean用xml定义，部分bean用注解定义）
3. **纯注解** （所有bean用注解定义）

模式1和模式2有相同的IoC容器启动方式，只是针对JavaSE 应用和JavaWeb应用有所不同：

1. JavaSE 应用，可从2个地方获得xml文件:
    1. 类路径下：`ApplicationContext appcon = new ClassPathXmlApplicationContext("beans.xml");`
    2. 文件系统中：`ApplicationContext appcon = new FileSystemXmlApplicationContext("c:/beans.xml");`

2. JavaWeb应用：通过监听器(ContextLoaderListener)去加载xml

而模式3有着另一种IoC容器启动方式，且针对JavaSE 应用和JavaWeb应用：

1. JavaSE 应用：`ApplicationContext appcon = new AnnotationConfigApplicationContext(SpringConfig.class);`
2. JavaWeb 应用：通过监听器(ContextLoaderListener)去加载注解配置class

1.2 BeanFactory vs. ApplicationContext

BeanFactory是Spring框架中IoC容器的顶层接口,它只是用来定义一些基础功能,定义一些基础规范,而ApplicationContext是它的一个子接口，所以ApplicationContext是具备BeanFactory提供的全部功能的。

通常，我们称BeanFactory为SpringIOC的基础容器，ApplicationContext是容器的高级接口，比BeanFactory要拥有更多的功能，比如说国际化支持（显示不同语言）和资源访问(读取xml，java配置类)等等。

1.3 纯xml模式

Code - `lagou-transfer-ioc-xml`

Step 1: pom.xml文件中引入Spring IoC容器功能的dependency

```
    <!--引入Spring IoC容器功能-->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>5.1.12.RELEASE</version>
    </dependency>
```

Step 2: 把"beans.xml" 改名为 "applicationContext.xml"，并添加文件头

```
<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
        https://www.springframework.org/schema/beans/spring-beans.xsd">
```

Step 3: Test IoC Initiation - SE应用

```
@Test
    public void testIoCInitiation() {

        // xml模式 SE应用
        // 通过读取classpath下的xml文件来启动容器(推荐)
        ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("classpath:applicationContext.xml");
        // 不推荐使用,因为代码迁移时会受到限制
        //ApplicationContext applicationContext1 = new
FileSystemXmlApplicationContext("文件系统的绝对路径");

        AccountDao accountDao = (AccountDao)
applicationContext.getBean("accountDao");
        System.out.println(accountDao);
    }
```

Step 4: 若针对的是Web应用，需通过监听器(ContextLoaderListener)去加载xml

首先，pom.xml 引入Spring WEB的dependency：

```
<!--引入spring web功能-->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-web</artifactId>
      <version>5.1.12.RELEASE</version>
    </dependency>
```

然后，在 webapp/WEB-INF/web.xml 中添加：

```
  <!--配置Spring ioc容器的配置文件供监听器使用，param-value是路径-->
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:applicationContext.xml</param-value>
  </context-param>

  <!--由于监听器是在Spring Web dependency里，所以要先去pom.xml引入Spring Web -->
  <!--然后，获取监听器来启动Spring的IOC容器-->
  <listener>
    <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>
```

Step 5：让Servlet能够获取并使用Spring 的IoC容器

为了让Servlet能够使用Spring 的IoC容器而不是之前我们自定义的BeanFactory，先删除自己写的BeanFactory，然后在TransferServlet.java中Override init方法：

```
public class TransferServlet extends HttpServlet {

    private TransferService transferService = null;

    @Override
    public void init() throws ServletException {
        WebApplicationContext webApplicationContext =
WebApplicationContextUtils.getWebApplicationContext(this.getServletContext());
        // 为了有事务处理，还是要先拿proxy，再通过proxy去拿transfer Service
        ProxyFactory proxyFactory = (ProxyFactory)
webApplicationContext.getBean("proxyFactory");
```

```
        transferService = (TransferService)
proxyFactory.getJDKProxy(webApplicationContext.getBean("transferService"));

        // 【注意！】不能直接去get transfer Service 的Bean，因为这样就会没有事务处理。
        // transferService = (TransferService)
webApplicationContext.getBean("transferService");
    }
```

1.4 纯xml模式 之 细节探讨

1.4.1 实例化Bean的3种方式

这里以生成ConnectionUtils对象来举例。

- 方式一：在xml中，使用无参Constructor（默认，推荐）

```
   <bean id="connectionUtils" class="com.lagou.edu.utils.ConnectionUtils">
</bean>
```

- 方式二：静态方法
  - 在code中，create一个新类 CreateBeanFactory：

```
public class CreateBeanFactory {

    public static ConnectionUtils getInstanceStatic() {
        return new ConnectionUtils();
    }
}
```

  - 在xml中，调用其静态方法来生成ConnectionUtils对象：

```
<bean id="connectionUtils"
class="com.lagou.edu.factory.CreateBeanFactory" factory-
method="getInstanceStatic"/>
```

- 方式三：实例化类后调用其方法
  - 在code中，添加普通方法：

```
public class CreateBeanFactory {

    public ConnectionUtils getInstance() {
        return new ConnectionUtils();
    }
}
```

- 在xml中，先实例化CreateBeanFactory，然后再调用其getInstance方法：

```
<bean id="createBeanFactory"
class="com.lagou.edu.factory.CreateBeanFactory"></bean>
<bean id="connectionUtils" factory-bean="createBeanFactory" factory-
method="getInstance"/>
```

1.4.2 Bean的作用范围

标签中，有个scope属性可以来定义bean的作用范围：

- **singleton**: 单例（默认值），IoC容器中只有一个该类对象 （对象唯一）

  单例bean的生命周期与容器相同：

  - 对象出生：当创建容器时，对象就被创建了
  - 对象活着：只要容器在，对象一直活着
  - 对象死亡：当销毁容器时，对象就被销毁了
- **prototype**: 多例，每次使用该类对象（getBean），都返回一个新的对象（对象不再唯一）

  Spring只负责创建多例bean的对象，不负责管理对象

  - 对象出生：当使用对象时，创建新的对象实例
  - 对象活着：只要对象在使用中，就一直活着
  - 对象死亡：当对象长时间不用时，被java的垃圾回收器回收了
- request： 【适用于WEB应用，但很少用】每一个请求对应于一个对象，新的请求会给新的对象
- session： 【适用于WEB应用，但很少用】每一个session中，对象唯一，新的session会给新的对象
- application 【适用于WEB应用，但很少用】
- websocket 【适用于WEB应用，但很少用】

1.4.3 Bean的标签属性

- **id**属性: 用于给bean提供一个唯一标识。在一个标签内部，标识必须唯一。
- **class**属性:用于指定创建Bean对象的全限定类名。
- **name**属性:用于给bean提供一个或多个名称。多个名称用空格分隔。
- **factory-bean**属性:用于指定创建当前bean对象的工厂bean的唯一标识。当指定了此属性之后，

class属性失效。

- **factory-method**属性:用于指定创建当前bean对象的工厂方法，如配合factory-bean属性使用， 则class属性失效。如配合class属性使用，则方法必须是static的。
- **scope**属性:用于指定bean对象的作用范围。通常情况下就是singleton。当要用到多例模式时， 可以配置为prototype。
- **init-method**属性:用于指定bean对象的初始化方法，此方法会在bean对象装配后调用。必须是 一个无参方法。
- **destroy-method**属性:用于指定bean对象的销毁方法，此方法会在bean对象销毁前执行。它只 能为scope是singleton时起作用。

1.4.4 DI依赖注入的xml配置

即往一个对象中传值。

```
<!-- DI依赖注入的xml配置，方式一：set注入  -->
    <bean id="myDIClassSetter" class="com.lagou.edu.pojo.MyDIClassSetter">
        <property name="name" value="Jenny"/>
        <property name="sex" value="1"/>
        <property name="balance" value="100.3"/>
    </bean>


<!-- DI依赖注入的xml配置，方式二：带参constructor注入  -->
    <bean id="myDIClassConstructor"
class="com.lagou.edu.pojo.MyDIClassConstructor">
        <!-- 使用index不方便，因为要人工对应顺序  -->
<!--        <constructor-arg index="0" value="Lisa"/>-->
<!--        <constructor-arg index="1" value="2"/>-->
<!--        <constructor-arg index="2" value="88.2"/>-->
<!--        <constructor-arg index="3" ref="connectionUtils"/>-->

        <!-- 使用name可以不用考虑顺序  -->
        <constructor-arg name="balance" value="88.2"/>
        <constructor-arg name="connectionUtils" ref="connectionUtils"/>
        <constructor-arg name="name" value="Lisa"/>
        <constructor-arg name="sex" value="2"/>

    </bean>
```

用set形式来注入复杂的数据类型：

```
<!-- DI依赖注入的xml配置，方式一：set注入  -->
  <bean id="myDIClassSetter" class="com.lagou.edu.pojo.MyDIClassSetter">
        <property name="name" value="Jenny"/>
```

```
        <property name="sex" value="1"/>
        <property name="balance" value="100.3"/>

        <!--set注入注入复杂数据类型。
        由于<array>和<set>都是一元存储，所以内部标签可以互换(见下)；同理<map>和<props>也
是。
            <property name="myArray">
              <set>
                <value>array1</value>
                <value>array2</value>
              </set>
            </property>
        -->
        <property name="myArray">
            <array>
                <value>array1</value>
                <value>array2</value>
                <value>array3</value>
            </array>
        </property>

        <property name="myMap">
            <map>
                <entry key="key1" value="value1"/>
                <entry key="key2" value="value2"/>
            </map>
        </property>

        <property name="mySet">
            <set>
                <value>set1</value>
                <value>set2</value>
            </set>
        </property>

        <property name="myProperties">
            <props>
                <prop key="prop1">value1</prop>
                <prop key="prop2">value2</prop>
            </props>
        </property>

    </bean>
```

1.5 xml + 注解模式

Code - `lagou-transfer-ioc-xmlannotation`

> **【注意】**
>
> 1. 实际企业开发中，纯xml模式使用已经很少了
> 2. 引入注解功能，不需要引入额外的jar
> 3. xml+注解结合模式，xml文件依然存在，所以，spring IoC容器的启动**仍然从加载xml开始**
> 4. 哪些bean的定义写在xml中，哪些bean的定义使用注解?
>   1. 第三方jar中的bean定义在xml，比如Druid数据库连接池
>   2. 自己开发的bean定义使用注解

1.5.1 第三方jar中的bean定义在xml

Druid数据库来自于第三方Alibaba，所以在xml中定义它并将其set进依赖它的主类中：

```
 <!-- 第三方jar中的bean定义在xml  -->
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://localhost:3306/xiaoyudb?
serverTimezone=UTC"/>
        <property name="username" value="root"/>
        <property name="password" value="1Q2w3e4r!"/>
    </bean>

    <bean id="connectionUtils" class="com.lagou.edu.utils.ConnectionUtils">
        <!-- Set datasource into 主class   -->
        <property name="dataSource" ref="dataSource"/>
    </bean>
```

主类里添加private属性和set方法：

```
public class ConnectionUtils {

    private DataSource dataSource;

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
  ...
```

1.5.2 自己开发的bean使用注解定义

Step 1: 根据现在的xml，在各个Class上加上@Repository，@Service或者@Component

Step 2: 在各个已有注解的类中，以DI的思想引入其依赖：

- **@Autowired**: 自动装载，按照类型注入。加在属性上，则该属性无需再有set方法。
  - 如果按照类型无法唯一锁定对象（例如AccountDao有很多实现类且都加装到了IoC中），可以用@Qualifier("具体的某个实现类的名字") 来指定。
- @Resource: 默认按照 ByName 自动注入

```
public class TransferService {
    @Resource
    private AccountDao accountDao;
    @Resource(name="studentDao")
    private StudentDao studentDao;
    @Resource(type="TeacherDao")
    private TeacherDao teacherDao;
    @Resource(name="manDao",type="ManDao")
    private ManDao manDao;
}
```

注意：

@Resource 在 Jdk 11中已经移除，如果要使用，需要单独引入jar包：

```
<dependency>
    <groupId>javax.annotation</groupId>
  <artifactId>javax.annotation-api</artifactId>
    <version>1.3.2</version>
</dependency>
```

Step 3: 创建jdbc.properties文件来改造DataSource的引入方式

1.6 纯注解模式

改造xml+注解模式，将xml中遗留的内容全部以注解的形式迁移出去，**最终删除xml，从Java配置类**启动。

对应注解：

- @Configuration 注解，表名当前类是一个配置类
- @ComponentScan 注解，替代 context:component-scan
- @PropertySource，引入外部属性配置文件
- @Import 引入其他配置类
- @Value 对变量赋值，可以直接赋值，也可以使用 ${} 读取资源配置文件中的信息

- **@Bean** 将方法返回对象加入 SpringIOC 容器

1.6.1 创建SpringConfig配置类

从而完全替代掉applicationContext.xml配置文件：

```
package com.lagou.edu;

import com.alibaba.druid.pool.DruidDataSource;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;

import javax.sql.DataSource;

/*
* @Configuration 注解表明当前类是一个配置类
* @ComponentScan({pkg1, pkg2, pkg3})
* @PropertySource({file1, file2})
* */
@Configuration
@ComponentScan({"com.lagou.edu"})
@PropertySource({"classpath:jdbc.properties"})
public class SpringConfig {

    // @Value(值的来源) 可以对变量直接赋值，也可以使用 ${} 读取资源配置文件中的信息 来赋值
    @Value("${jdbc.driver}")
    private String driverClassName;

    @Value("${jdbc.url}")
    private String url;

    @Value("${jdbc.username}")
    private String username;

    @Value("${jdbc.password}")
    private String pw;

    // @Bean(id的名字) 将方法返回对象（Druid DS）加入 Spring IoC 容器中
    @Bean("dataSource")
    public DataSource createDataSource() {
        // 创建一个Druid DS 并给其属性赋值
```

```
        DruidDataSource druidDataSource = new DruidDataSource();
        druidDataSource.setDriverClassName(driverClassName);
        druidDataSource.setUrl(url);
        druidDataSource.setUsername(username);
        druidDataSource.setPassword(pw);

        return druidDataSource;
    }
}
```

1.6.2 让Java SE应用使用SpringConfig配置类来启动IoC容器

在 MyIoCTest 中：

```
import com.lagou.edu.SpringConfig;
import com.lagou.edu.dao.AccountDao;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MyIoCTest {

    @Test
    public void testIoCInitiation() {

        // SE应用 纯注解模式
        // AnnotationConfigApplicationContext(可传入多个配置类)
        ApplicationContext applicationContext = new
AnnotationConfigApplicationContext(SpringConfig.class);

        AccountDao accountDao = (AccountDao)
applicationContext.getBean("accountDao");
        System.out.println(accountDao);
    }
}
```

1.6.3 让Java WEB应用使用SpringConfig配置类来启动IoC容器

在webapp/WEB-INF/web.xml中，添加：

```
<!--告诉ContextloaderListener我们是使用注解的方式启动IoC容器-->
<context-param>
  <param-name>contextClass</param-name>
  <param-
value>org.springframework.web.context.support.AnnotationConfigWebApplicationConte
xt</param-value>
</context-param>

<!--配置Spring Config，param-value是配置类的全类名-->
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>com.lagou.edu.SpringConfig</param-value>
</context-param>
```

当然，这时就可以直接删除掉applicationContext.xml文件了。

2. 高级特性

2.1 lazy-Init 延迟加载

ApplicationContext默认行为：在Spring启动时，将所有 **singleton bean** 提前进行实例化。当getBean时，是从缓存里拿到的bean。

如果设置lazy-init为true，则该bean将不会在 ApplicationContext 启动时提前被实例化，而是第一次向容器

通过 getBean 索取 bean 时才实例化的。

```
<bean id="testBean" calss="cn.lagou.LazyBean" lazy-init="true" />
```

情景一：立即加载的 bean1引用了一个延迟加载的 bean2

那么 bean1 在容器启动时被实例 化，而 bean2 由于被 bean1 引用，所以也被实例化，这种情况也符合延时加载的 bean 在第一次调用 时才被实例化的规则。

情景二：全局的bean都延迟加载

在容器层次中通过在 元素上使用 "default-lazy-init" 属性来控制延时初始化。如下面配置:

```
 <beans default-lazy-init="true">
    <!-- no beans will be eagerly pre-instantiated... -->
</beans>
```

标签的priority高于全局，自己的scope的priority更高：如果一个 bean 的 scope 属性为 scope="pototype" 时，即使设置了 lazy-init="false"，容器启动时也不 会实例化bean，而是调用 getBean 方法实例化的。

应用场景

- 开启延迟加载一定程度提高容器启动和运转性能
- 对于不常使用的 Bean 设置延迟加载，这样偶尔使用的时候再加载，不必要从一开始该 Bean 就占 用资源。

2.2 FactoryBean 和 BeanFactory

2.2.1 FactoryBean 简介

BeanFactory接口是容器的顶级接口，定义了容器的一些基础行为，负责生产和管理Bean的一个工厂，

具体使用它下面的子接口类型，比如ApplicationContext。

Spring中Bean有两种，一种是普通Bean，一种是工厂Bean(FactoryBean)，FactoryBean可以生成

某一个类型的Bean实例(返回给我们)，也就是说我们可以借助于它自定义Bean的创建过程。

FactoryBean Interface 见下：

```
package org.springframework.beans.factory;

import org.springframework.lang.Nullable;

// 可以让我们自定义Bean的创建过程(完成复杂Bean的定义)
public interface FactoryBean<T> {

    @Nullable
    T getObject() throws Exception;  // 返回FactoryBean创建的Bean实例，如果
isSingleton返回true，则该实例会放到Spring容器 的单例对象缓存池中

    @Nullable
    Class<?> getObjectType();   // 返回FactoryBean创建的Bean类型
```

```
    default boolean isSingleton() {
        return true;
    }
```

2.2.2 FactoryBean 应用

Step 1: 创建Company Class

Step 2：创建CompanyFactoryBean Class

```
package com.lagou.edu.factory;

import com.lagou.edu.pojo.Company;
import org.springframework.beans.factory.FactoryBean;

public class CompanyFactoryBean implements FactoryBean<Company> {

    private String companyInfo; // 信息包括：公司名称，地址，规模
    public void setCompanyInfo(String companyInfo) {
        this.companyInfo = companyInfo;
    }

    @Override
    public boolean isSingleton() {
        return true;
    }

    @Override
    public Company getObject() throws Exception {
        // 创建复杂对象Company
        Company company = new Company();
        String[] strings = companyInfo.split(",");
        company.setName(strings[0]);
        company.setAddress(strings[1]);
        company.setScale(Integer.parseInt(strings[2]));

        return company;
    }

    @Override
    public Class<?> getObjectType() {
        return Company.class;
```

```
    }
}
```

Step 3: 创建Test

获取到的bean是Company 类：

```
@Test
    public void testCompanyFactoryBean() {
        ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("classpath:applicationContext.xml");

        Object result = applicationContext.getBean("companyBean");
        System.out.println(result); // result 是 Company 类，而不是
CompanyFactoryBean 类
    }
```

获取到的bean是CompanyFactoryBean 类：

```
@Test
    public void testCompanyFactoryBean() {
        ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("classpath:applicationContext.xml");

        Object result = applicationContext.getBean("&companyBean");
        System.out.println(result);  // result 是 CompanyFactoryBean 类
    }
```

2.3 BeanPostProcessor 后置处理器

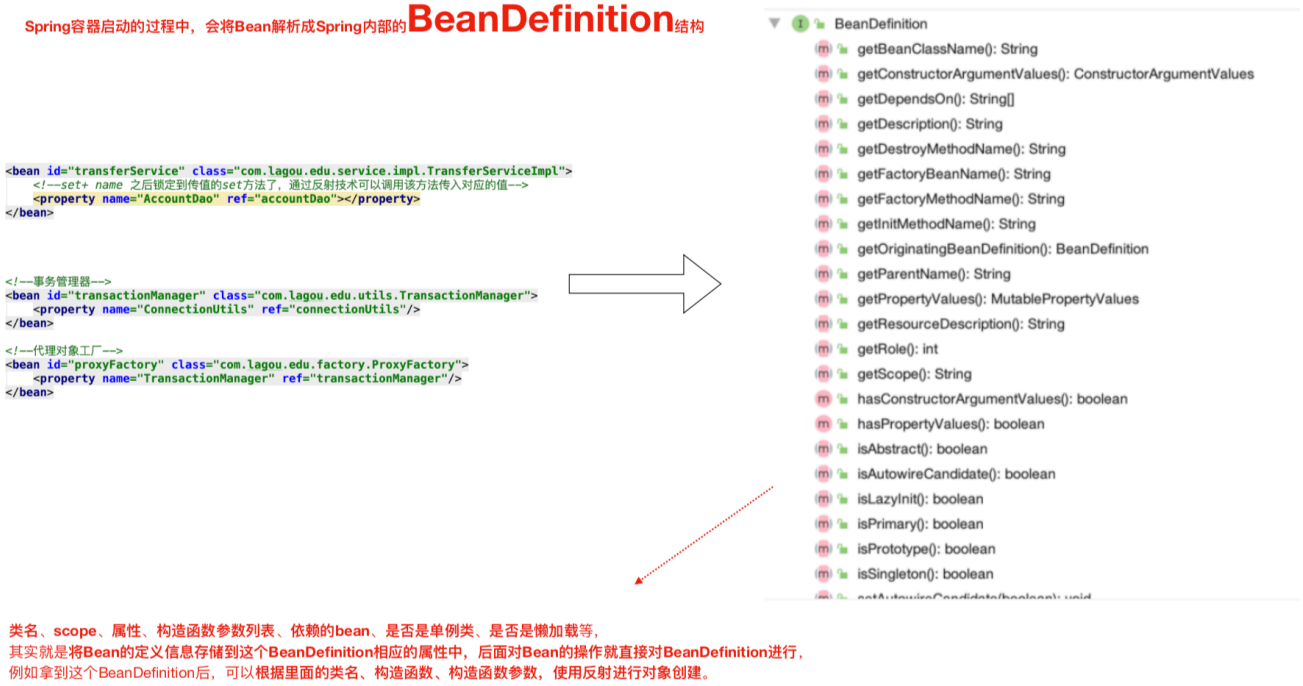
2.3.1 后置处理器简介

Spring提供了两种后处理bean的扩展接口：

- BeanPostProcessor：针对Bean。在Bean对象实例化(并不是Bean的整个生命周期完成)之后可以使用BeanPostProcessor 做一些事情
- BeanFactoryPostProcessor：针对BeanFactory，BeanFactory会生成Bean，在BeanFactory初始化之后可以使用BeanFactoryPostProcessor 做一些事情

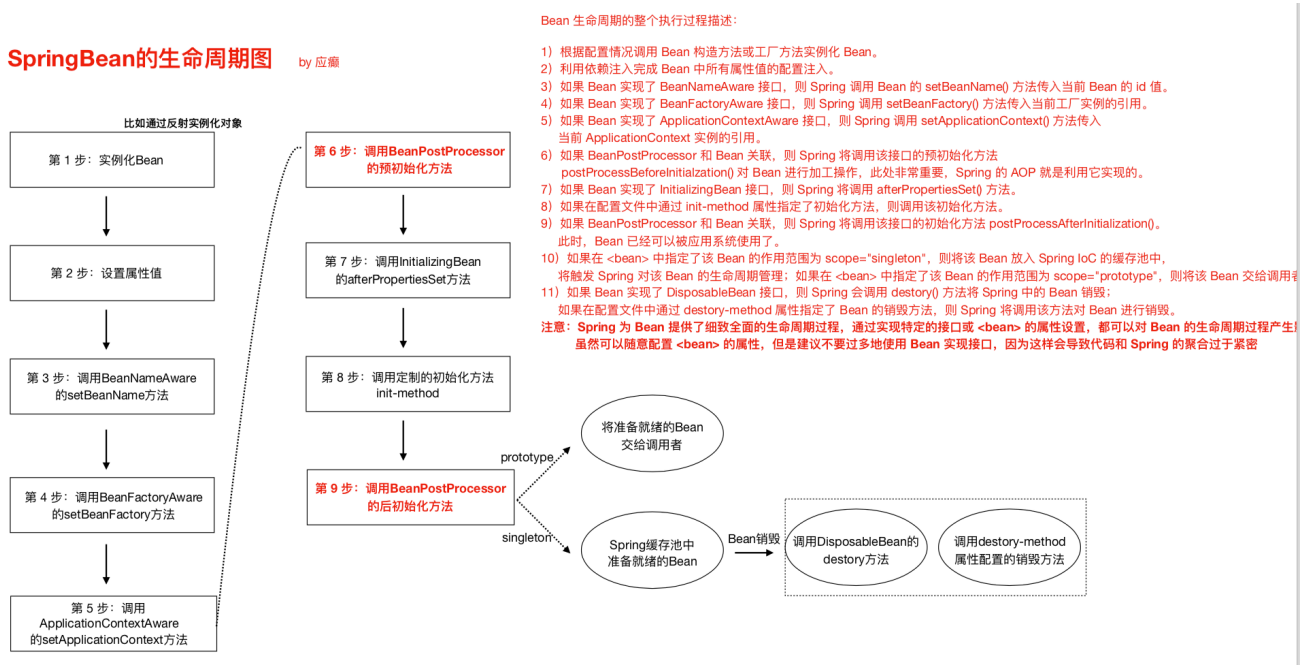
2.3.2 Spring将xml封装进Beans Definition

把xml配置信息读取加载进来之后，Spring会将xml中的一个Bean解析并封装成Spring内部的一个Bean Definition结构对象:



2.3.3 Spring Bean的lifycycle（生命周期）

Vedio ”Task 2-第四部分-高级特性之后置处理器“ 4:00



第1步

第2步

第3步：实现BeanFactoryAware Interface，然后Override setBeanName 方法:

```
public class Result implements BeanNameAware {
  ...

  @Override
    public void setBeanName(String name) {
        System.out.println("3. 注册我成为bean时，定义的id为：" + name);
    }
}
```

第4步：BeanFactoryAware是一个Interface：

```
public class Result implements BeanNameAware, BeanFactoryAware {
  ...

@Override
    public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
        System.out.println("4. 管理我的BeanFactory是这个：" + beanFactory);
    }
}
```

第5步：实现ApplicationContextAware Interface：

```
public class Result implements BeanNameAware, BeanFactoryAware,
ApplicationContextAware {
  ...

   @Override
    public void setApplicationContext(ApplicationContext applicationContext)
throws BeansException {
        System.out.println("5. 拿到高级容器Interface ApplicationContext：" +
applicationContext);
    }
}
```

第6步：实现BeanPostProcessor Interface，该接口提供了两个方法，分别在Bean的**初始化方法前(Before)**和**初始化方法后(After)**执行（具体这个初始化方法指的是 什么方法，类似我们在定义bean时，定义了init-method所指定的方法）：

```
public interface BeanPostProcessor {
      @Nullable
      default Object postProcessBeforeInitialization(Object bean, String
beanName) throws BeansException {
          return bean;
      }

      @Nullable
      default Object postProcessAfterInitialization(Object bean, String beanName)
throws BeansException {
          return bean;
      }
  }
```

创建自定义的后置处理器：

```
/*
* 第6步：
* 拦截实例化之后的对象（现在才走到第6步，实例化+属性注入了，但还不是一个完整的Spring Bean）
* */
public class MyBeanPostProcessor implements BeanPostProcessor {

    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName)
throws BeansException {
        // 只针对 lazyResult 这个bean进行处理
        if ("lazyResult".equalsIgnoreCase(beanName)) {
            System.out.println("6.1 MyBeanPostProcessor的 Before方法 拦截处理
lazyResult bean");
        }

        // 处理完之后要将bean返回回去
        return bean;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName)
throws BeansException {
        // 只针对 lazyResult 这个bean进行处理
        if ("lazyResult".equalsIgnoreCase(beanName)) {
```

```
            System.out.println("9.=6.2 MyBeanPostProcessor的 After方法 拦截处理
lazyResult bean");
        }

        // 处理完之后要将bean返回回去
        return bean;
    }
}
```

第 6之后 步：@PostConstruct 发生在init method之前

```
/*
 * https://www.baeldung.com/spring-postconstruct-predestroy
 *
 * Note that both @PostConstruct and @PreDestroy annotations is part of Java EE.
 * And since Java EE has been deprecated in Java 9 and removed in Java 11
 * we have to add an additional dependency, Java EE, to use these annotations.
 * */
@PostConstruct
public void postConstruct() {
    System.out.println("6之后, This is post construct method...");
}
```

第7步：实现InitializingBean Interface:

```
public class Result implements BeanNameAware, BeanFactoryAware,
ApplicationContextAware, InitializingBean {
  ...

    @Override
    public void afterPropertiesSet() throws Exception {
        System.out.println("7. afterPropertiesSet...");
    }
}
```

第8步：调用自定义的init-method：

```
public class Result implements ... {
  ...

  public void myInitMethod() {
        System.out.println("8. 这时才开始 init-method ... ");
    }
}
```

记得在xml文件中加上属性"init-method":

```
<bean id="lazyResult" class="com.lagou.edu.pojo.Result" lazy-init="false" init-
method="myInitMethod"/>
```

第9步：=6.2, 即调用BeanPostProcessor的 AfterInitialization 方法。

Test:

```
@Test
public void testSpringBeanLifeCycle() {
    ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("classpath:applicationContext.xml");

    Object result = applicationContext.getBean("lazyResult");
    System.out.println();
    System.out.println("Finally create the bean: " + result);
}
```

Test结果:

```
3. 注册我成为bean时，定义的id为：lazyResult
4. 管理我的BeanFactory是这个：
org.springframework.beans.factory.support.DefaultListableBeanFactory@71ba6d4e:
defining beans
[accountDao,proxyFactory,myBeanPostProcessor,transferService,connectionUtils,tran
sactionManager,org.springframework.context.annotation.internalConfigurationAnnota
tionProcessor,org.springframework.context.annotation.internalAutowiredAnnotationP
rocessor,org.springframework.context.annotation.internalCommonAnnotationProcessor
,org.springframework.context.event.internalEventListenerProcessor,org.springframe
work.context.event.internalEventListenerFactory,org.springframework.context.suppo
rt.PropertySourcesPlaceholderConfigurer#0,dataSource,lazyResult,companyBean];
root of factory hierarchy
5. 拿到高级容器Interface ApplicationContext：
org.springframework.context.support.ClassPathXmlApplicationContext@62bd765,
started on Fri May 01 15:35:41 PDT 2020
6.1 MyBeanPostProcessor的 Before方法 拦截处理lazyResult bean
6之后，This is post construct method...
7. afterPropertiesSet...
8. 这时才开始 init-method ...
9.=6.2 MyBeanPostProcessor的 After方法 拦截处理lazyResult bean

Finally create the bean: Result{status='null', message='null'}
```

2.3.4 Spring Singleton Bean的销毁

在Result class中，实现DisposableBean Interface并Override destroy方法，在destroy之前，可以用annotation @PostDestroy 加入一个销毁前的操作：

```
public class Result implements BeanNameAware, ..., DisposableBean {
...
  @PreDestroy
  public void myPreDestroy() {
      System.out.println("Singleton Bean's Destroy: Before destroying, do this
PreDestroy   method...");
  }

  @Override
  public void destroy() throws Exception {
      System.out.println("Singleton Bean's Destroy: destroy the bean. ");
  }
}
```

Test:

```
@Test
public void testSpringSingletonBeanDestroy() {
    ClassPathXmlApplicationContext applicationContext = new
ClassPathXmlApplicationContext("classpath:applicationContext.xml");

    Object result = applicationContext.getBean("lazyResult");
    System.out.println();
    System.out.println("Finally create the bean: " + result);

    // 关闭IoC容器，由于lazyResult bean是singleton的，所以会被IoC容器管理到其后续的
destroy过程
    applicationContext.close();
}
```

Test结果：

```
...
7. afterPropertiesSet...
8. 这时才开始 init-method ...
9.=6.2 MyBeanPostProcessor的 After方法 拦截处理lazyResult bean

Finally create the bean: Result{status='null', message='null'}
Singleton Bean's Destroy: Before destroying, do this PreDestroy method...
Singleton Bean's Destroy: destroy the bean.
```

2.4 BeanFactoryPostProcessor 后置处理器

BeanFactory产生之后，开始生产Bean对象之前（即2.3.3之前, 2.3.2之后，此时 **bean 刚被解析成BeanDefinition对象**），可以用BeanFactoryPostProcessor做些事情：

```
package org.springframework.beans.factory.config;

import org.springframework.beans.BeansException;

@FunctionalInterface
public interface BeanFactoryPostProcessor {
    void postProcessBeanFactory(ConfigurableListableBeanFactory var1) throws
BeansException;
}
```

此接口只提供了一个方法，方法参数为ConfigurableListableBeanFactory，该参数类型定义了一些方法：

```
public interface ConfigurableListableBeanFactory extends ListableBeanFactory,
AutowireCapableBeanFactory, ConfigurableBeanFactory {
    void ignoreDependencyType(Class<?> var1);

    void ignoreDependencyInterface(Class<?> var1);

    void registerResolvableDependency(Class<?> var1, @Nullable Object var2);

    boolean isAutowireCandidate(String var1, DependencyDescriptor var2) throws
NoSuchBeanDefinitionException;

    // BeanFactory 可以拿到 BeanDifinition，然后我们可以对定义的属性进行修改
    BeanDefinition getBeanDefinition(String var1) throws
NoSuchBeanDefinitionException;

    Iterator<String> getBeanNamesIterator();

    void clearMetadataCache();

    void freezeConfiguration();

    boolean isConfigurationFrozen();

    void preInstantiateSingletons() throws BeansException;
}
```

关于如何修改BeanDifition中的属性：

Step 1：先通过BeanFactoryPostProcessor拿到BeanDefinition

Step 2：然后通过BeanDefinition的方法，去手动修改bean标签中所定义的属性值

典型应用：有很多个关于jdbc的配置文件，用最新的去覆盖老的配置。

第五部分 Spring IoC 源码深度剖析

- 好处：提高培养代码架构思维、深入理解框架
- 读源码时需要坚持的原则：

    - 定焦原则：抓主线
    - 宏观原则：站在上帝视角，关注源码结构和业务流程（淡化具体某行代码的编写细节）读源码的方法和技巧
- 读源码的方法和技巧：
    - 断点（观察调用栈）
    - Find Usages
    - 经验（Spring框架中的 `doXXXX` 方法就是**真正**在做具体处理的地方）

0. Spring 源码构建

- Github 下载源码：https://github.com/spring-projects/spring-framework/tree/5.1.x
- 安装Gradle 5.6.3 (https://gradle.org/install/)
- 其他软件：IntelliJ 2019.3， JDK-11
- 导入：耗费一定时间 (4m 39s)
- 编译工程：选择右边菜单中的 “spring-core / Tasks / other / complieTestJava”
    - 此编译会自行按照顺序： spring-core => oxm => context => beans => aspects => aop

在build好的Spring 源码中：

- 右键点击 `spring-framework-5.1.x` ，点New一个module，这样这个New出来的module就会自动被放在 `spring-framework-5.1.x` folder中。
- 为了让新建的module中的bean也能被IoC容器识别并管理，所以得让它去依赖spring-context工程：
    - 打开新建module的 `build.gradle` 文件，修改成：

```
plugins {
    id 'java'
}

group 'org.springframework'
version '5.1.12.BUILD-SNAPSHOT'

sourceCompatibility = 11
// https://github.com/gradle/gradle/issues/8009
targetCompatibility = 11

repositories {
    mavenCentral()
}
```

```
dependencies {
  // 可以直接调用spring-context的源码
    compile(project(":spring-context"))
  //  compile group: 'org.aspectj', name: 'aspectjweaver', version:
'1.8.6'
testCompile group: 'junit', name: 'junit', version: '4.12'
}
```

> 关于targetCompatibility，由于没法在IntelliJ中进行设置，所以这里直接用代码设置了。

1. Spring IoC容器初始化主体流程

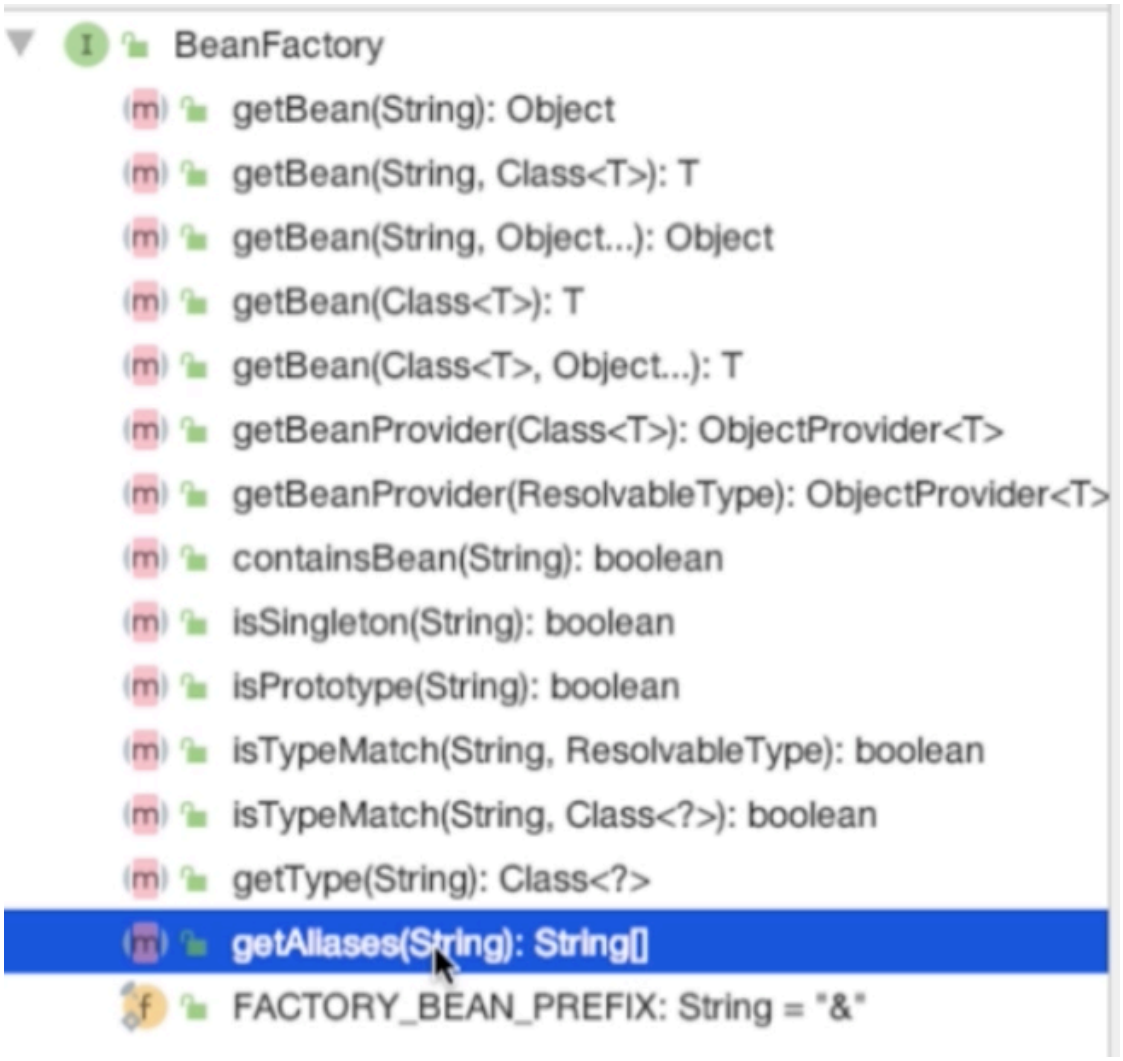
1.1 Spring IoC 容器体系

BeanFacotry 是顶级容器/根容器，规范了/定义了容器的基础行为。

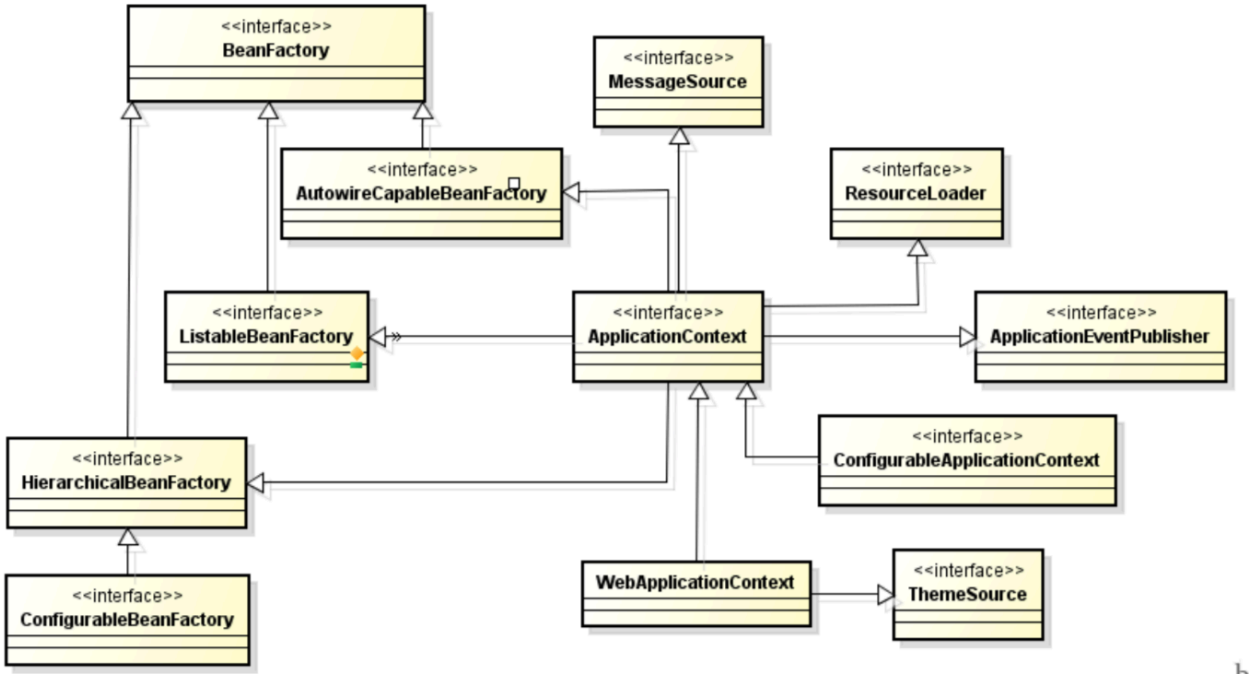
ApplicationContext是容器的高级Interface，

IoC并不仅是个map，准确来说，map是IoC容器的一个成员，叫做单例池, singletonObjects,

容器是：一组组件（包括BeanFactory、单例池、BeanPostProcessor等）以及他们之间的协作过程的集合。



BeanFacotry容器层级划分体系，ApplicationContext需要什么功能，就去implements什么Interface：



例如：

- ApplicationContext Interface implements ListableBeanFactory Interface (ListableBeanFactory 负责批量返回)
- ApplicationContext Interface implements ResourceLoader Interface （ResourceLoader负责加载资源）
- HierarchicalBeanFactory Interface implements BeanFactory (HierarchicalBeanFactory 可以获得parentBeanFactory)

1.2 Spring Bean life cycle 关键时机点

首先，添加 `MyBeanFactoryPostProcessor`，`MyBeanPostProcessor`，并在xml文件中将他俩register成bean。

然后，分别在`LagouBean`， `MyBeanFactoryPostProcessor`，`MyBeanPostProcessor` 这三个类中的构造函数的地方加断点，可以发现：

```
/*
 * LagouBean构造器执行、初始化方法执行、MyBeanPostProcessor的before/after方法：
 *        AbstractApplicationContext # refresh # finishBeanFactoryInitialization
 * MyBeanFactoryPostProcessor的constructor、方法执行：
 *        AbstractApplicationContext # refresh # invokeBeanFactoryPostProcessors
 * MyBeanPostProcessor的constructor：
 *        AbstractApplicationContext # refresh # registerBeanPostProcessors
 *
 * 可见，AbstractApplicationContext中的refresh方法是很重要的方法。
 */
```

1.3 refresh方法中的主流程

加断点在：

```
ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("classpath:applicationContext.xml");
```

> Step Into ClassPathXmlApplicationContext中，会进入一个static的模块中，此时点 Step Out，然后再点Step Into，就可以看到ClassPathXmlApplicationContext的构造器。

AbstractApplicationContext中refresh方法中的主流程，主要分为2个子流程：

1. BeanFactory的创建
2. Bean的创建

```
  @Override
  public void refresh() throws BeansException, IllegalStateException {
    /*
    * 在refresh的时候加了对象锁。在close的时候也加了这个对象锁，目的是为了
    * 让容器在启动的时候不能进行close，让容器close的时候不能进行refresh和启动。
    * */
    synchronized (this.startupShutdownMonitor) {
      /*
      * Prepare this context for refreshing.
      * 做真正refresh之前，做的一些准备工作：
      *    设置Spring容器的启动时间，
      *    开启活跃状态，撤销关闭状态，
      *    如果有拓展的属性也去initiate一下，
      *    验证环境信息里一些必须存在的属性等
```

```
        * */
        prepareRefresh();

        /*
        * 【重点 - BeanFactory的创建】Tell the subclass to refresh the internal bean
factory.
        * （1）获取BeanFactory，默认实现的是 DefaultListableBeanFactory，
        * （2）把xml中的Bean信息读取并把每个bean封装成一个BeanDefinition 并注册进
BeanDefinitionRegistry
        * */
        ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

        /*
        * Prepare the bean factory for use in this context.
        * BeanFactory的准备工作（即前置），例如对其进行一些设置（context的类加载器等）
        * */
        prepareBeanFactory(beanFactory);

        try {
            /*
            * Allows post-processing of the bean factory in context subclasses.
            * BeanFactory 准备工作完成后，进行的后置处理工作。没有实现，留给扩展使用。
            * */
            postProcessBeanFactory(beanFactory);

            /*
            * Invoke factory processors registered as beans in the context.
            * 实例化BeanFactoryPostProcessor Interface的Bean，并调用其Override的方法
            * */
            invokeBeanFactoryPostProcessors(beanFactory);

            // ====== 以上已经将BeanFactory的事情准备得差不多了，下面是对Bean的准备 =======

            /*
            * Register bean processors that intercept bean creation.
            * 注册BeanPostProcessors，之后好用
            * */
            registerBeanPostProcessors(beanFactory);

            /*
            * Initialize message source for this context.
            * 初始化MessageSource组件（可做国际化功能，消息绑定，消息解析等）
            * */
            initMessageSource();

            // Initialize event multicaster for this context.
```

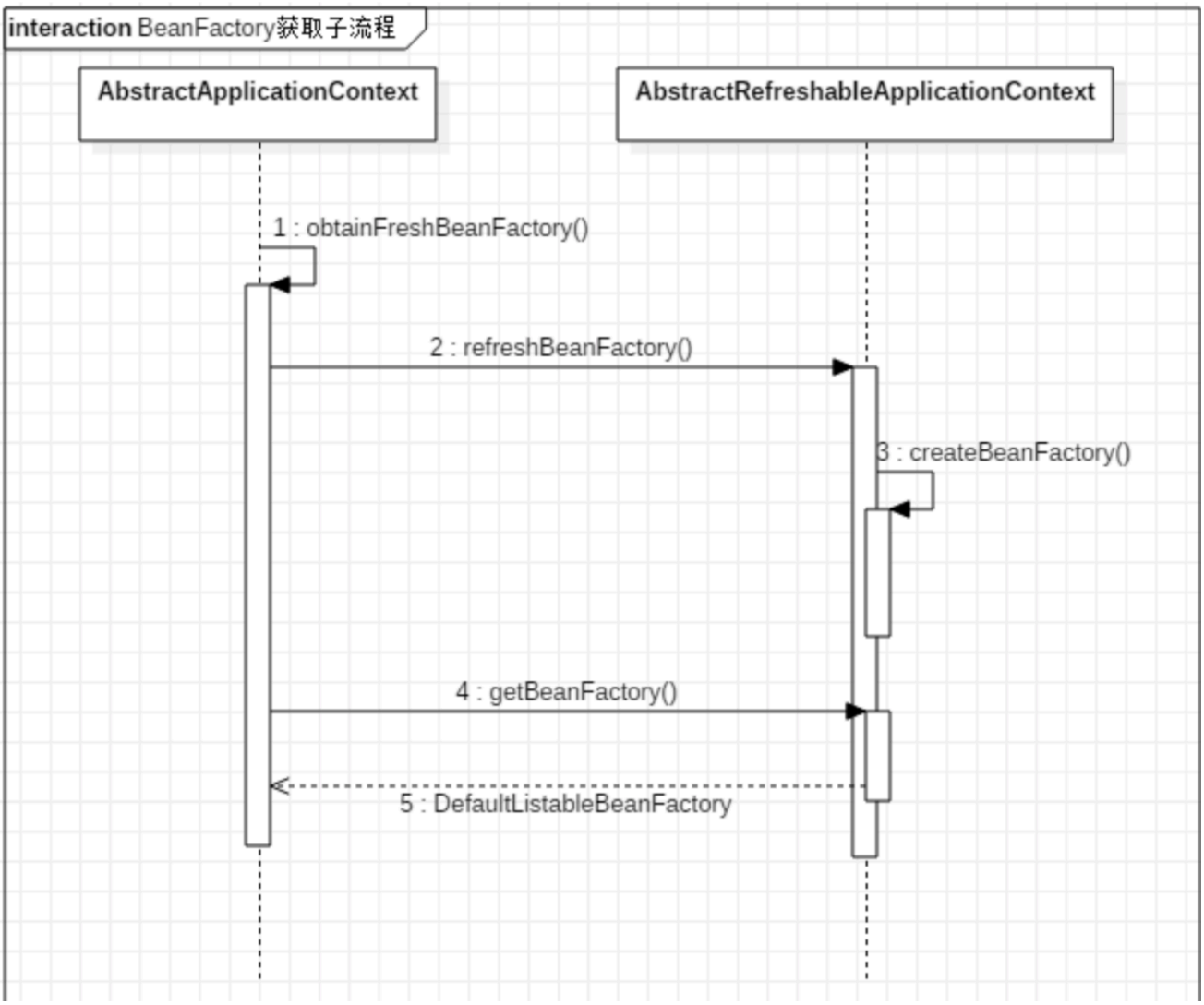
```
            initApplicationEventMulticaster();

            // Initialize other special beans in specific context subclasses.
            onRefresh();

            // Check for listener beans and register them.
            registerListeners();

            /*
            * 【重点】Instantiate all remaining (non-lazy-init) singletons.
            * 初始化创建所有剩下的非懒加载的单例Bean,
            * 填充Bean中属性,
            * 初始化方法调用（例如 afterPropertiesSet方法，init-method方法）
            * 调用BeanPostProcessor的after方法对实例bean进行后置处理
            * */
            finishBeanFactoryInitialization(beanFactory);

            /*
            * Last step: publish corresponding event.
            * 完成context的刷新。
            * 主要调用LifecycleProcessor的 onRefresh 方法。
            * */
            finishRefresh();
        }

        catch (BeansException ex) {
            ...
        }

        finally {
            ...
        }
    }
}
```

2. 子流程 之 BeanFactory的创建

画时序图的软件们有这些 - link

2.1 通过obtainFreshBeanFactory来获取BeanFactory

时序图：



2.2 BeanDefinition 加载xml并注册进BeanDefinitionRegistry

调用栈：

org.springframework.context.support.AbstractRefreshableApplicationContext#loadBeanDefinitions

org.springframework.context.support.AbstractXmlApplicationContext#loadBeanDefinitions(org.springframework.beans.factory.support.DefaultListableBeanFactory)

org.springframework.beans.factory.support.AbstractBeanDefinitionReader#**loadBeanDefinitions**(java.lang.String...)

org.springframework.beans.factory.xml.XmlBeanDefinitionReader#**loadBeanDefinitions**(org.springframework.core.io.Resource)

org.springframework.beans.factory.xml.XmlBeanDefinitionReader#**doLoadBeanDefinitions**

读取xml完成，解析成了Document类。接下来开始注册：

org.springframework.beans.factory.xml.XmlBeanDefinitionReader#**registerBeanDefinitions**

org.springframework.beans.factory.xml.DefaultBeanDefinitionDocumentReader#**doRegisterBeanDefinitions**

org.springframework.beans.factory.xml.DefaultBeanDefinitionDocumentReader#**parseBeanDefinitions**

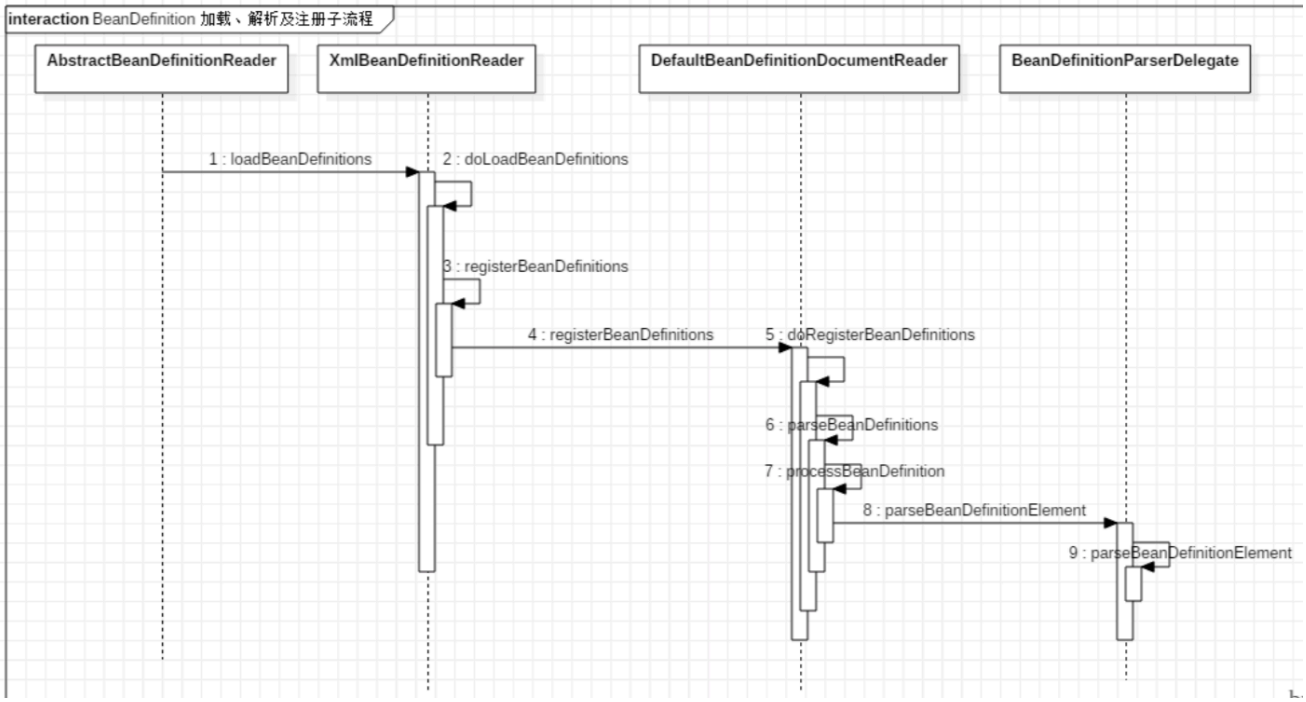
org.springframework.beans.factory.xml.DefaultBeanDefinitionDocumentReader#parseDefaultElement

org.springframework.beans.factory.xml.DefaultBeanDefinitionDocumentReader#**processBeanDefinition**

org.springframework.beans.factory.support.BeanDefinitionReaderUtils#**registerBeanDefinition**

org.springframework.beans.factory.support.DefaultListableBeanFactory#**registerBeanDefinition**

根据调用栈，省略了一些啰嗦的步骤，最后画出时序图：



3. 子流程 之 Bean的创建

子流程入口在AbstractApplicationContext Class中的refresh方法中：

```
        /*
         * 【重点】Instantiate all remaining (non-lazy-init) singletons.
         * 初始化创建所有剩下的非懒加载的单例Bean,
         * 填充Bean中属性,
         * 初始化方法调用（例如 afterPropertiesSet方法，init-method方法）
         * 调用BeanPostProcessor的after方法对实例bean进行后置处理
         * */
        finishBeanFactoryInitialization(beanFactory);
```

调用栈：

org.springframework.context.support.AbstractApplicationContext#**finishBeanFactoryInitialization**

org.springframework.beans.factory.support.DefaultListableBeanFactory#**preInstantiateSingletons**

org.springframework.beans.factory.support.AbstractBeanFactory#getBean(java.lang.String)

org.springframework.beans.factory.support.AbstractBeanFactory#**doGetBean**

org.springframework.beans.factory.support.DefaultSingletonBeanRegistry#getSingleton(java.lang.String, org.springframework.beans.factory.ObjectFactory<?>)

org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#createBean(java.lang.String, org.springframework.beans.factory.support.RootBeanDefinition, java.lang.Object[])

真真正正地开始创建bean

org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#**doCreateBean**

第1步

org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#createBeanInstance

第2步

org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#populateBean

org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#**initializeBean**(java.lang.String, java.lang.Object, org.springframework.beans.factory.support.RootBeanDefinition)

org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#**invokeAwareMethods**

- Step 3：BeanNameAware （BeanClassLoaderAware）
- Step 4：BeanFactoryAware
- Step 5: ？ 没找到 【直播提问】

```
public interface ApplicationContextAware extends Aware {

  /**
   * Set the ApplicationContext that this object runs in.
   * Normally this call will be used to initialize the object.
   * <p>Invoked after population of normal bean properties but before an init
callback such
   * as {@link
org.springframework.beans.factory.InitializingBean#afterPropertiesSet()}
   * or a custom init-method. Invoked after {@link
ResourceLoaderAware#setResourceLoader},
   * {@link ApplicationEventPublisherAware#setApplicationEventPublisher} and
   * {@link MessageSourceAware}, if applicable.
   * @param applicationContext the ApplicationContext object to be used by
this object
   * @throws ApplicationContextException in case of context initialization
errors
   * @throws BeansException if thrown by application context methods
   * @see org.springframework.beans.factory.BeanInitializationException
   */
  void setApplicationContext(ApplicationContext applicationContext) throws
BeansException;

}
```

initializeBean 方法：

```
protected Object initializeBean(final String beanName, final Object bean,
@Nullable RootBeanDefinition mbd) {
    if (System.getSecurityManager() != null) {
      AccessController.doPrivileged((PrivilegedAction<Object>) () -> {
        invokeAwareMethods(beanName, bean);
        return null;
      }, getAccessControlContext());
    }
    else {
      // 见SpringBean lifecycle图的第3步、第4步
      invokeAwareMethods(beanName, bean);
    }

    // 没找到 第5步？？？ 【直播提问】

    Object wrappedBean = bean;
    if (mbd == null || !mbd.isSynthetic()) {
      // 第6步：BeanPostProcessor的 Before 方法
```

```
      wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean,
beanName);
    }

    try {
      // 第7步 调用InitializingBean的afterPropertiesSet方法
      // + 第8步 调用定制的init-method
      invokeInitMethods(beanName, wrappedBean, mbd);
    }
    catch (Throwable ex) {
      throw new BeanCreationException(
          (mbd != null ? mbd.getResourceDescription() : null),
          beanName, "Invocation of init method failed", ex);
    }
    if (mbd == null || !mbd.isSynthetic()) {
      // 第9步 BeanPostProcessor的 After 方法
      wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean,
beanName);
    }

    return wrappedBean;
  }
```

Step 6:

applyBeanPostProcessorsBeforeInitialization

Step 7:

org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#invokeInitMethods

Step 8：

org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#invokeCustomInitMethod

Step 9:

applyBeanPostProcessorsAfterInitialization

4. lazy-init 延迟加载机制原理

在Bean的创建 子流程里，在实例化所有剩下的非单例bean时：

org.springframework.beans.factory.support.DefaultListableBeanFactory#preInstantiateSingletons

```
@Override
public void preInstantiateSingletons() throws BeansException {
   if (logger.isTraceEnabled()) {
      logger.trace("Pre-instantiating singletons in " + this);
   }

   // Iterate over a copy to allow for init methods which in turn register new
bean definitions.
   // While this may not be part of the regular factory bootstrap, it does
otherwise work fine.
   List<String> beanNames = new ArrayList<>(this.beanDefinitionNames);  // 放的
bean的names，即bean的id

   // Trigger initialization of all non-lazy singleton beans...
   for (String beanName : beanNames) {
      // 往当前子bean合并来自于父beandefinition的信息
      RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);
      //  【看这里！！！】只有当 非抽象 且 单例 且 非懒加载，才会继续进行实例化
      if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {
         if (isFactoryBean(beanName)) {   // 看是否是FactoryBean本身
            Object bean = getBean(FACTORY_BEAN_PREFIX + beanName);  //
FACTORY_BEAN_PREFIX 即 &
            if (bean instanceof FactoryBean) {
               final FactoryBean<?> factory = (FactoryBean<?>) bean;
               boolean isEagerInit;
               if (System.getSecurityManager() != null && factory instanceof
SmartFactoryBean) {
                  isEagerInit =
AccessController.doPrivileged((PrivilegedAction<Boolean>)
                              ((SmartFactoryBean<?>) factory)::isEagerInit,
                        getAccessControlContext());
               }
               else {
                  isEagerInit = (factory instanceof SmartFactoryBean &&
                        ((SmartFactoryBean<?>) factory).isEagerInit());
               }
               if (isEagerInit) {
                  getBean(beanName);
               }
            }
         }
         else {
            //  【看这里！！！】实例化当前bean
            getBean(beanName);
```

```
        }
    }
}

// Trigger post-initialization callback for all applicable beans...
for (String beanName : beanNames) {
    ...
```

总结：

- lazy-init 为 true的bean，在Spring IoC容器初始化节点不会被实例化，只有当第一次applicationContext.getBean()时，才会进行初始化并依赖注入
- 如果bean1依赖bean2，bean1的lazy-init为false但bean2的lazy-init为true，那么 bean1 在容器启动时被实例 化，而 bean2 由于被 bean1 引用，所以也被实例化，这种情况也符合延时加载的 bean 在第一次调用 时才被实例化的规则。
- 对于lazy-init为false的bean，applicationContext.getBean() 时会直接从缓存中获得 （因为容器初始化阶段就已经将这些bean创建并缓存了）

5. Spring IoC 循环依赖问题

5.1 什么是循环依赖

循环依赖其实就是循环引用，也就是两个或者两个以上的 Bean 互相持有对方，最终形成闭环。比如对象A依赖于对象B，对象B依赖于对象C，对象C又依赖于对象A。

Spring中循环依赖场景有：

- 构造器的循环依赖（构造器注入）
- Field属性的循环依赖（set注入）

其中，构造器的循环依赖问题无法解决，只能抛出 BeanCurrentlyInCreationException 异常，在解决

属性循环依赖时，spring采用的是提前暴露对象的方法。

5.2 循环依赖处理机制

5.2.1 单例bean 构造器参数循环依赖：无法解决

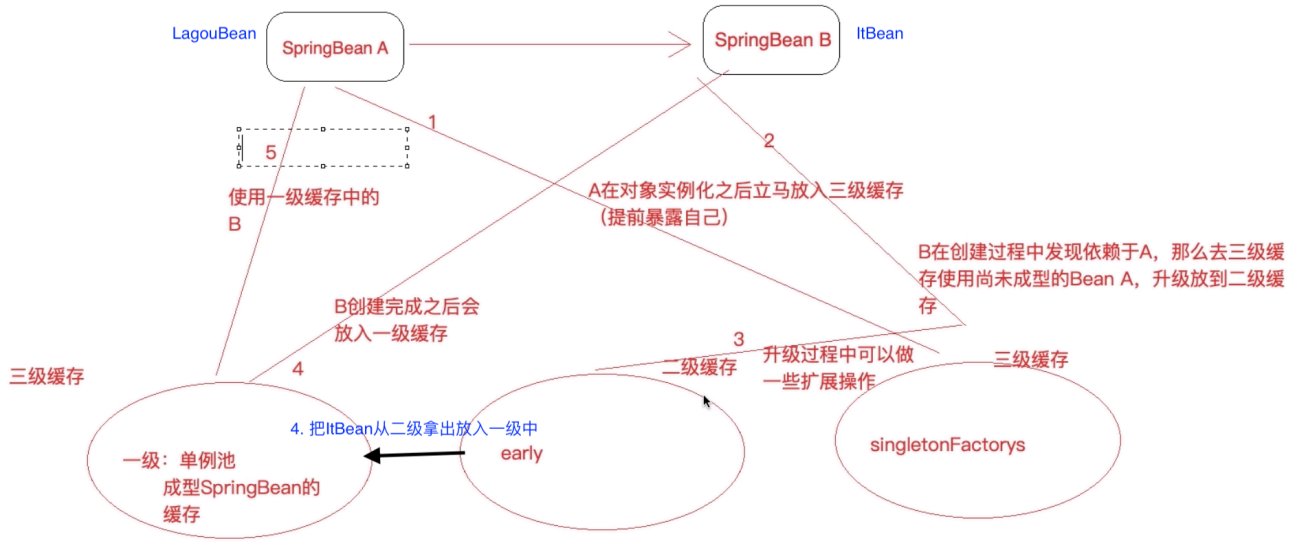
因为无法将依赖或者自身放入各级缓存中。

5.2.2 Prototype原型bean 循环依赖：无法解决

5.2.3 单例bean 通过setter方法或者@Autowired 进行循环依赖：通过3级缓存可以解决

注释见code https://github.com/WildCapriccio/SpringSourceCode-ver5.1.x/tree/circular

流程图：



尚未成型的Bean A被存入三级缓存（因为此时A只做了第1步）；

B在创建过程中发现依赖于A，就去三级缓存拿出尚未成型的Bean A，放到二级缓存中 ==》给A升级了缓存（升级过程中可以做一些扩展操作）

B创建完成。

继续创建A，A可从一级缓存中获取B。

调用栈截图：

```
org.springframework.context.support.AbstractApplicationContext#refresh
    org.springframework.context.support.==AbstractApplicationContext==#finishBeanFactoryInitialization （创建非懒加载的单例bean的入口）
        org.springframework.beans.factory.support.==DefaultListableBeanFactory==#preInstantiateSingletons

            org.springframework.beans.factory.support.AbstractBeanFactory#getBean 【1】，想创建LagouBean
              org.springframework.beans.factory.support.AbstractBeanFactory#doGetBean （尝试去单例池中拿LagouBean，没有，继续往后走）
                org.springframework.beans.factory.support.DefaultSingletonBeanRegistry#getSingleton （开始创建单例LagouBean,验证完要真正开始创建的对象后，先标识该bean正在被创建）

                  org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#createBean
                    org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#doCreateBean （真真正正开始创建LagouBean）
                      [lifecycle]
                      org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#createBeanInstance（Step1. 实例化LagouBean，尚未设置属性）
                      org.springframework.beans.factory.support.DefaultSingletonBeanRegistry#addSingletonFactory （将当前尚未成型的LagouBean放入三级缓存（singletonFactories））
                      org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#populateBean （Step 2. LagouBean属性填充）
                      |
                      |   org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#applyPropertyValues （扫描LagouBean的属性，需要去找有个叫ItBean的property值）
                      |       org.springframework.beans.factory.support.BeanDefinitionValueResolver#resolveValueIfNecessary （去找ItBean的属性值）
                          org.springframework.beans.factory.support.BeanDefinitionValueResolver#resolveReference （继续去找ItBean的属性值，遇到getBean）

                      |               org.springframework.beans.factory.support.AbstractBeanFactory#getBean 同【1】，不过此时是想创建ItBean
                      |                   org.springframework.beans.factory.support.AbstractBeanFactory#doGetBean （尝试去单例池中拿ItBean，没有，继续往后走）
                                            org.springframework.beans.factory.support.DefaultSingletonBeanRegistry#getSingleton
                                            |
                                                org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#createBean
                      |                             org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#doCreateBean
                      |                         |
                                                      [lifecycle]
                                                      org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#createBeanInstance（Step1. 实例化ItBean，尚未设置属性）
                      |                               org.springframework.beans.factory.support.DefaultSingletonBeanRegistry#addSingletonFactory （将当前尚未成型的LagouBean放入三级缓存
                      |                               org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#populateBean （Step 2. ItBean属性填充）
                                                      |   org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#applyPropertyValues （扫描ItBean的属性，需要去找有个叫LagouBean的property值）
                                                      |       org.springframework.beans.factory.support.BeanDefinitionValueResolver#resolveValueIfNecessary （去找LagouBean的属性值）
                                                          org.springframework.beans.factory.support.BeanDefinitionValueResolver#resolveReference （继续去找LagouBean的属性值，遇到getBean）
                      |
                      |
                                                              【关键】
                                                      |           org.springframework.beans.factory.support.AbstractBeanFactory#getBean
                                                              org.springframework.beans.factory.support.AbstractBeanFactory#doGetBean （尝试去单例池中拿LagouBean）
                                                                  org.springframework.beans.factory.support.DefaultSingletonBeanRegistry#getSingleton （尝试从一级缓存、二级缓存和三级缓存中拿LagouBean，最后在三级缓存中找到，从三级缓存中取出，（这里也可以通过org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#getEarlyBeanReference 中的BeanPostProcessor做扩展），最后放入二级缓存，并返回未成形的LagouBean给doGetBean）
                      |                       |           |一路将未成形的LagouBean不停return，这样，就可以顺利完成Step 2 ItBean属性填充
                      |                       |           Step 3 ~ Step 9 常规操作，完成doCreateBean
                      |                       完成ItBean创建，getSingleton 结束。
                      |                       org.springframework.beans.factory.support.DefaultSingletonBeanRegistry#addSingleton （把新创建的ItBean放入一级缓存中，然后从二级和三级缓存中删除）
                      LagouBean属性填充完成
                      Step 3 ~ Step 9 常规操作
```

UML图：

Code [github link]

随堂测试

Spring事务管理核心接口

https://blog.csdn.net/fly910905/article/details/78684772

- PlatformTransactionManager，常用方法：
  - commit(TransactionStatus):void
  - getTransaction(TransactionDefinition):TransactionStatus
  - rollback(TransactionStatus):void
- TransactionDefinition
  - ISOLation_xxxx: 事务隔离级别
  - PROPAGATION_xxxx： 事务的传播行为（不是JDBC中有的，为了解决实际开发问题）
  - getTimeout():int
- TransactionStatus
  - hasSavepoint():boolean
  - isNewTransaction():boolean
  - isCompleted():boolean