Spring源码阅读笔记

源码环境搭建

- 安装git/jdk17/
- idea中clone https://github.com/spring-projects/spring-framework到本地
- 等待Gradle安装完依赖
- 用Gradle编译Spring-oxm模块中的Tasks中的other中的complieTestJava, 如果没问题则源码编译成功了
- 新建一个子模块
- build.gradle中添加以下代码

```
dependencies {
    implementation(project(":spring-beans"))
    implementation(project(":spring-context"))
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.6.0'
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine'
}
```

- 创建一个测试类和一个配置类

```
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;


public class KukahaMain {


   public static void main(String[] args) {
      AnnotationConfigApplicationContext annotationConfigApplicationContext =
new               AnnotationConfigApplicationContext(SysConfig.class);
      System.out.println(annotationConfigApplicationContext);
   }

}
```

```
import org.springframework.context.annotation.Configuration;


@Configuration
public class SysConfig {
}
```

源码阅读---IOC

源码基础---工厂模式

- 工厂模式所实现的最重要的一件事就是解耦, 所谓解耦, 解的是依赖, 让本来在编译期的依赖报错转移到运行时依赖
- 例如下面代码, 本来需要我们手动new出来一个对象, 这样的话在编译期间如果编译器发现你写的代码有这个依赖没有导入, 则会报错,
- 使用工厂模式, 将对象在运行时反射创建, 编译器在编译的时候将不会检查依赖, 而在实际运行时才会去根据全限定类名去反射出来所依赖的对象(运行时才去根据全限定类名去查找依赖)

```
/**
 * 自定义一个BeanFactory
 */
public class BeanFactory {
   // 配置属性
   private static Properties properties;
   // 类加载的时候通过本类的类加载器来读取出来配置文件, 注入properties属性
   static {
      properties = new Properties();
      InputStream resourceAsStream = BeanFactory.class.getClassLoader()
            .getResourceAsStream("bean.properties");
      try {
         properties.load(resourceAsStream);
      } catch (IOException e) {
         throw new ExceptionInInitializerError("初始化properties失败");
      }

   }
   // 向外提供一个获取Bean的方法, 通过传入名字来查找全限定类名反射创建对象交给调用者
   public static Object getBean(String beanName) throws ClassNotFoundException,
NoSuchMethodException, IllegalAccessException, InvocationTargetException,
InstantiationException {
      String beanPath =  properties.getProperty(beanName);
      Object ob= Class.forName(beanPath).getDeclaredConstructor().newInstance();
      return ob;
   }


}
```

```
public interface IDoService{
   public void exec();
}
```

```
public class BeanService implements IDoService {
   @Override
   public void exec() {
      System.out.println("执行业务");
   }
}
```

```
public class Client {
   public static void main(String[] args) throws ClassNotFoundException,
NoSuchMethodException, InvocationTargetException, InstantiationException,
IllegalAccessException {
      //调用工厂, 通过名字来交给工厂创建Bean
      IDoService beanService = (IDoService) BeanFactory.getBean("beanService");
      beanService.exec();
   }
}
```

在以上代码中, 自定义了一个工厂类, 提供了读取静态配置文件的静态方法, 向外暴露了一个获取Bean对象的方法, 在Client中, 我们就可以使用工厂类去反射获取对象了, 而不需要自己手动new

但是这种形式有一点小问题, 就是创建出来的对象是多例的情况, 因为每次创建出来的对象都是不一样的, 但是在Web开发中, service等组件不会存在可以改变的类成员, 不会存在多线程问题, 所有可以使用单例对象, 这就可以引出spring的核心---容器

- 实现方法: 工厂类创建时候读取配置文件, 一次性创建出来所有对象, 使用map容器来把这些对象名称和实例保存起来, 当client需要使用这些对象的时候, 从容器中拿出来使用即可.

spring核心---IOC容器(手写)

```
/**
 * 自定义一个BeanFactory
 */
public class BeanFactory {
   // 配置属性
   private static Properties properties;
   // 定义一个map来存放我们要创建的对象, 即容器
   private static Map<String, Object> beans;

   // 类加载的时候通过本类的类加载器来读取出来配置文件, 注入properties属性
   static {
      properties = new Properties();
      InputStream resourceAsStream = BeanFactory.class.getClassLoader()
            .getResourceAsStream("bean.properties");
      beans = new HashMap<>();
      try {
         // 加载配置
         properties.load(resourceAsStream);
         // 取出配置文件的的keys
         Enumeration<Object> keys = properties.keys();
         // 遍历枚举
         while (keys.hasMoreElements()) {
            // 取出每个key
            String key = keys.nextElement().toString();
            // 根据key获取全限定类名
            String beanPath = properties.getProperty(key);
            // 反射创建对象
            Object value =
Class.forName(beanPath).getDeclaredConstructor().newInstance();
            beans.put(key, value);
         }
      } catch (IOException | ClassNotFoundException | NoSuchMethodException |
InstantiationException | IllegalAccessException | InvocationTargetException e) {
         throw new ExceptionInInitializerError("初始化properties失败");
```

```
        }

    }

    // 向外提供一个获取Bean的方法, 通过传入名字来查找容器中的对象交给调用者
    public static Object getBean(String beanName) throws ClassNotFoundException,
NoSuchMethodException, IllegalAccessException, InvocationTargetException,
InstantiationException {
        return beans.get(beanName);
    }


}
```

```
public class Client {
    public static void main(String[] args) throws ClassNotFoundException,
NoSuchMethodException, InvocationTargetException, InstantiationException,
IllegalAccessException {
//      //调用工厂, 通过名字来交给工厂创建Bean
//      IDoService beanService = (IDoService) BeanFactory.getBean("beanService");
//      beanService.exec();

        // 使用spring的核心容器
        ApplicationContext ac=new ClassPathXmlApplicationContext("bean.xml");
        BeanService beanService = ac.getBean("beanService", BeanService.class);
        beanService.exec();
    }
}
```

仅仅改动BeanFactory代码和client调用代码

- 通过创建了一个静态map容器, 在BeanFactory创建的时候就加载properties文件反射一次性创建出来所有对象放入map容器中, 当client需要调用依赖的时候, 直接从map中取即可
- 这就是IOC控制反转的原理, 通过工厂模式, 将我们手动创建的对象反转给工厂去创建, 我们需要用的时候问工厂要就行了

正式开始源码阅读

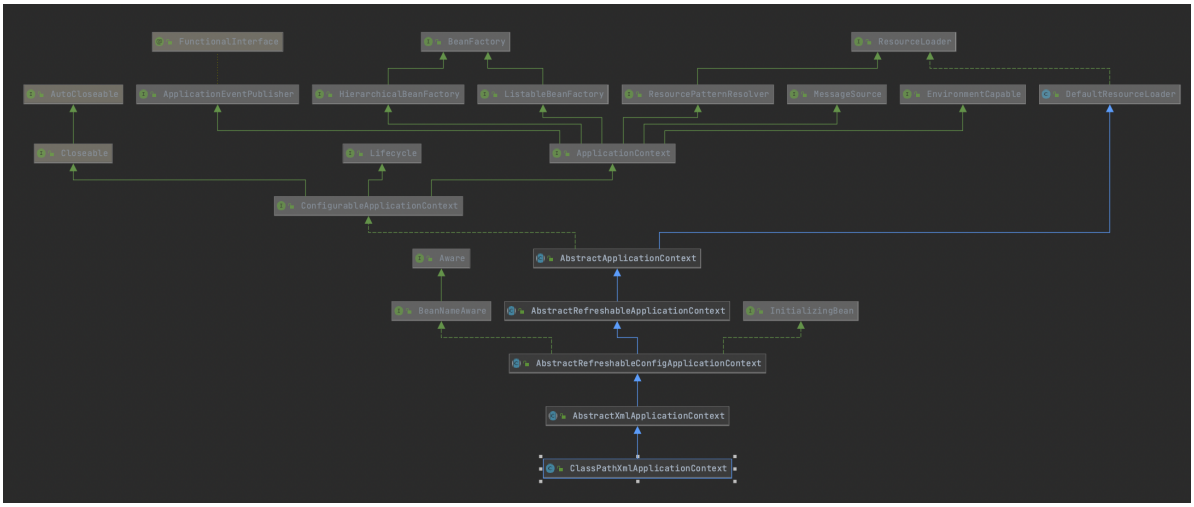
以上面的Client入口文件作为锚点, 作为分析的起点

分析之前, 首先提出几个问题

1. IOC的核心容器是什么数据结构, 是何时被创建的, 是被谁创建的?
2. 我们自己编写的对象是何时被创建的, 是被谁创建的, 是如何创建的?
3. 被创建的bean对象是何时放入IOC容器中, 被谁放入的?

针对以上三点问题, 我们一点点深入源码去查阅

首先来一张ClassPathXmlApplicationContext类的依赖图



针对继承图, 可以得知, ClassPathXmlApplicationContext继承了AbstractXmlApplicationContext继承了AbstractRefreshableConfigApplicationContext继承了AbstractRefreshableApplicationContext继承了AbstractApplicationContext这四个主要的继承关系

首先来解答第一个问题

在ClassPathXmlApplicationContext的构造器中, 有一个refresh()的方法, 这个实现方法在AbstractApplicationContext类中, 在这个方法里面, 负责进行bean工厂的创建和bean对象的创建

```
/**
     * Create a new ClassPathXmlApplicationContext with the given parent,
     * loading the definitions from the given XML files.
     * @param configLocations array of resource locations
     * @param refresh whether to automatically refresh the context,
     * loading all bean definitions and creating all singletons.
     * Alternatively, call refresh manually after further configuring the
context.
     * @param parent the parent context
     * @throws BeansException if context creation failed
     * @see #refresh()
     */
    public ClassPathXmlApplicationContext(
            String[] configLocations, boolean refresh, @Nullable
ApplicationContext parent)
            throws BeansException {

        super(parent);
        setConfigLocations(configLocations);
        if (refresh) {
            refresh();
        }
    }
```

AbstractApplicationContext的refresh()方法

```
@Override
public void refresh() throws BeansException, IllegalStateException {
   synchronized (this.startupShutdownMonitor) {
        // .....省略......

      // Tell the subclass to refresh the internal bean factory.
      ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();
```

```
        try {
           // .....省略.....
        }

        catch (BeansException ex) {
              // ...省略...
        }

        finally {
          // ...省略....
        }
    }
}
```

其中核心beanFactory对象由obtainFreshBeanFactory()方法创建, 这个beanFactory对象又被我们的AbstractRefreshableApplicationContext所持有, beanFactory对象对应的类是DefaultListableBeanFactory , 这个类里面有一个属性叫做singleTonObjects的concurrentHashMap的结构, 这个其实就是我们的IOC容器的底层数据结构

其中深入obtainFreshBeanFactory()这个方法, 可以看到DefaultListableBeanFactory的创建细节

```
    /**
     * This implementation performs an actual refresh of this context's
underlying
     * bean factory, shutting down the previous bean factory (if any) and
     * initializing a fresh bean factory for the next phase of the context's
lifecycle.
     */
    @Override
    protected final void refreshBeanFactory() throws BeansException {
        // ...省略...
        try {
            DefaultListableBeanFactory beanFactory = createBeanFactory();
            // ...省略...
            this.beanFactory = beanFactory;
        }
        catch (IOException ex) {
        // ...省略...
        }
    }
```

这个方法是属AbstractRefreshableApplicationContext类的, 故可知, 在这里的继承关系中, 核心IOC容器是AbstractRefreshableApplicationContext所持有的DefaultListableBeanFactory里的singleTonObjects, 是被AbstractRefreshableApplicationContext 所创建出来的,

故可得出总结:

- IOC容器名为singleTonObjects, 底层数据结构是一个concurrentHashMap, 即Map<String, Object>
- IOC容器创建时间为ClassPathXmlApplicationContext类初始化调用构造器的时候创建出来的
- IOC容器在这一堆继承关系中是被AbstractRefreshableApplicationContext的方法创建出来的

然后来回答第二个问题

再次回到AbstractApplication的refresh()方法中

```
@Override
public void refresh() throws BeansException, IllegalStateException {
   synchronized (this.startupShutdownMonitor) {
            // ...省略...

      try {
        // ...省略...

         // Instantiate all remaining (non-lazy-init) singletons.
         finishBeanFactoryInitialization(beanFactory);

      }

      catch (BeansException ex) {
            // ...省略...
      }

      finally {
            // ...省略...
      }
   }
}
```

重点观察finishBeanFactoryInitialization(beanFactory)这个方法, 在这个方法中, 将beanFactory传入, 猜测在这里面对beanFactory的singleTonObjects进行了实例化然后put操作, 直接跟进去

继续跟

```
/**
 * Return an instance, which may be shared or independent, of the specified
bean.
 * @param name the name of the bean to retrieve
 * @param requiredType the required type of the bean to retrieve
 * @param args arguments to use when creating a bean instance using explicit
arguments
 * (only applied when creating a new instance as opposed to retrieving an
existing one)
 * @param typeCheckOnly whether the instance is obtained for a type check,
 * not for actual use
 * @return an instance of the bean
 * @throws BeansException if the bean could not be created
 */
@SuppressWarnings("unchecked")
protected <T> T doGetBean(
            // ...省略...
         RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);

               // ...省略...
        return createBean(beanName, mbd, args);


               // ...省略...
```

```
    return adaptBeanInstance(name, beanInstance, requiredType);
}
```

可以得出, mbd里面有全限定类名, 而在createBean(beanName,mbd,args)这个方法里面进行了对象的实例化, 继续跟进去

继续点进去

```
@Override
public Object instantiate(RootBeanDefinition bd, @Nullable String beanName,
BeanFactory owner) {


      constructorToUse = clazz.getDeclaredConstructor();
            // ...省略...
      return BeanUtils.instantiateClass(constructorToUse);

}
```

最终在一个名为SimpleInstantiationStrategy的类里面的instantiate()方法里面发现了反射的端倪, 先获取构造器, 然后使用工具类进行实例化, 至此我们可以得出结论回答第二个问题

- 实例对象仍然是这一堆继承关系中的ClassPathXmlApplicationContext构造器构造的时候进行创建的(先创建容器, 再往创建对象)
- 实例对象是由SimpleInstantiationStrategy类所创建的
- 实例对象是由全限定类名获取构造器然后通过反射创建的

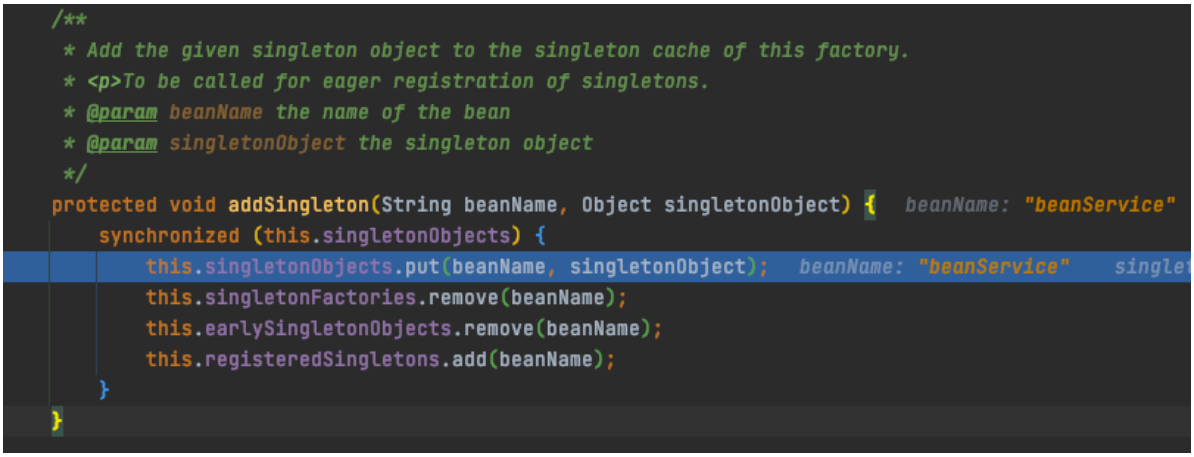
最后来回答第三个问题

经过我们的分析, IOC容器实际为singleTonObjects, 那么如果想要知道是什么时候将实例对象放入容器, 被谁放入容器, 只要知道singleTonObjects这个map在哪里调用put()方法即可

直接在idea上全局搜singletonObjects.put, 给用到的地方打上断点, debugger分析

```
/**
 * Add the given singleton object to the singleton cache of this factory.
 * <p>To be called for eager registration of singletons.
 * @param beanName the name of the bean
 * @param singletonObject the singleton object
 */
protected void addSingleton(String beanName, Object singletonObject) {
   synchronized (this.singletonObjects) {
      this.singletonObjects.put(beanName, singletonObject);
      this.singletonFactories.remove(beanName);
      this.earlySingletonObjects.remove(beanName);
      this.registeredSingletons.add(beanName);
   }
}
```

最终发现是在DefaultSingletonBeanRegistry类中的addSingleton()方法中将创建好的实例对象添加到IOC容器中



由此可得出第三个问题的答案

- 实例对象是在ClassPathXmlApplicationContext构造器构造的时候,(创建容器--->实例化对象--->容器中放入实例化对象) 父类AbstractRefreshableApplicationContext成员属性beanFactory 的父类DefaultSingletonBeanRegistry所放入的
- 是被DefaultSingletonBeanRegistry所放入

源码阅读---Aop