课程目标

1、通过分析 Spring 源码，深刻掌握核心原理和设计思想。

2、通过本课的学习，完全掌握 Spring IOC 的重要细节。

3、手绘 Spring DI 运行时序图。

内容定位

1、Spring 使用不熟练者不适合学习本章内容。

2、先掌握执行流程，再理解设计思想，这个过程至少要花 1 个月。

3、Spring 源码非常经典，体系也非常庞大，看一遍是远远不够的。

基于 Annotation 的 IOC 初始化

Annotation 的前世今生

从 Spring2.0 以后的版本中，Spring 也引入了基于注解(Annotation)方式的配置，注解(Annotation)是 JDK1.5 中引入的一个新特性，用于简化 Bean 的配置，可以取代 XML 配置文件。开发人员对注解(Annotation)的态度也是萝卜青菜各有所爱，个人认为注解可以大大简化配置，提高开发速度，但也给后期维护增加了难度。目前来说 XML 方式发展的相对成熟，方便于统一管理。随着 Spring Boot 的兴起，基于注解的开发甚至实现了零配置。但作为个人的习惯而言，还是倾向于 XML 配置文件和注解(Annotation)相互配合使用。Spring IOC 容器对于类级别的注解和类内部的注解分以下两种处理策略：

1)、类级别的注解：如@Component、@Repository、@Controller、@Service 以及 JavaEE6 的@ManagedBean 和@Named 注解，都是添加在类上面的类级别注解，Spring 容器根据注解的过滤规则扫描读取注解 Bean 定义类，并将其注册到 Spring IOC 容器中。

2)、类内部的注解：如@Autowire、@Value、@Resource 以及 EJB 和 WebService 相关的注解等，都是添加在类内部的字段或者方法上的类内部注解，SpringIOC 容器通过 Bean 后置注解处理器解析 Bean 内部的注解。下面将根据这两种处理策略，分别分析 Spring 处理注解相关的源码。

定位 Bean 扫描路径

在 Spring 中管理注解 Bean 定义的容器有两个：AnnotationConfigApplicationContext 和 AnnotationConfigWebApplicationContex。这两个类是专门处理 Spring 注解方式配置的容器，直接依赖于注解作为容器配置信息来源的 IOC 容器。AnnotationConfigWebApplicationContext 是 AnnotationConfigApplicationContext 的 Web 版本，两者的用法以及对注解的处理方式几乎没有差别。现在我们以 AnnotationConfigApplicationContext 为例看看它的源码：

```
public class AnnotationConfigApplicationContext extends GenericApplicationContext implements
AnnotationConfigRegistry {

   //保存一个读取注解的 Bean 定义读取器，并将其设置到容器中
   private final AnnotatedBeanDefinitionReader reader;

   //保存一个扫描指定类路径中注解 Bean 定义的扫描器，并将其设置到容器中
   private final ClassPathBeanDefinitionScanner scanner;

   //默认构造函数，初始化一个空容器，容器不包含任何 Bean 信息，需要在稍后通过调用其 register()
   //方法注册配置类，并调用 refresh()方法刷新容器，触发容器对注解 Bean 的载入、解析和注册过程
   public AnnotationConfigApplicationContext() {
      this.reader = new AnnotatedBeanDefinitionReader(this);
      this.scanner = new ClassPathBeanDefinitionScanner(this);
   }

   public AnnotationConfigApplicationContext(DefaultListableBeanFactory beanFactory) {
      super(beanFactory);
      this.reader = new AnnotatedBeanDefinitionReader(this);
      this.scanner = new ClassPathBeanDefinitionScanner(this);
```

```
}

//最常用的构造函数，通过将涉及到的配置类传递给该构造函数，以实现将相应配置类中的 Bean 自动注册到容器中
public AnnotationConfigApplicationContext(Class<?>... annotatedClasses) {
   this();
   register(annotatedClasses);
   refresh();
}

//该构造函数会自动扫描以给定的包及其子包下的所有类，并自动识别所有的 Spring Bean，将其注册到容器中
public AnnotationConfigApplicationContext(String... basePackages) {
   this();
   scan(basePackages);
   refresh();
}

@Override
public void setEnvironment(ConfigurableEnvironment environment) {
   super.setEnvironment(environment);
   this.reader.setEnvironment(environment);
   this.scanner.setEnvironment(environment);
}

//为容器的注解 Bean 读取器和注解 Bean 扫描器设置 Bean 名称产生器
public void setBeanNameGenerator(BeanNameGenerator beanNameGenerator) {
   this.reader.setBeanNameGenerator(beanNameGenerator);
   this.scanner.setBeanNameGenerator(beanNameGenerator);
   getBeanFactory().registerSingleton(
         AnnotationConfigUtils.CONFIGURATION_BEAN_NAME_GENERATOR, beanNameGenerator);
}

//为容器的注解 Bean 读取器和注解 Bean 扫描器设置作用范围元信息解析器
public void setScopeMetadataResolver(ScopeMetadataResolver scopeMetadataResolver) {
   this.reader.setScopeMetadataResolver(scopeMetadataResolver);
   this.scanner.setScopeMetadataResolver(scopeMetadataResolver);
}

//为容器注册一个要被处理的注解 Bean，新注册的 Bean，必须手动调用容器的
//refresh()方法刷新容器，触发容器对新注册的 Bean 的处理
public void register(Class<?>... annotatedClasses) {
   Assert.notEmpty(annotatedClasses, "At least one annotated class must be specified");
   this.reader.register(annotatedClasses);
}
```

```
    //扫描指定包路径及其子包下的注解类，为了使新添加的类被处理，必须手动调用
    //refresh()方法刷新容器
    public void scan(String... basePackages) {
        Assert.notEmpty(basePackages, "At least one base package must be specified");
        this.scanner.scan(basePackages);
    }

    ...

}
```

通过上面的源码分析，我们可以看啊到 Spring 对注解的处理分为两种方式：

1)、直接将注解 Bean 注册到容器中

可以在初始化容器时注册；也可以在容器创建之后手动调用注册方法向容器注册，然后通过手动刷新容器，使得容器对注册的注解 Bean 进行处理。

2)、通过扫描指定的包及其子包下的所有类

在初始化注解容器时指定要自动扫描的路径，如果容器创建以后向给定路径动态添加了注解 Bean，则需要手动调用容器扫描的方法，然后手动刷新容器，使得容器对所注册的 Bean 进行处理。

接下来，将会对两种处理方式详细分析其实现过程。

读取 Annotation 元数据

当创建注解处理容器时，如果传入的初始参数是具体的注解 Bean 定义类时，注解容器读取并注册。

1)、AnnotationConfigApplicationContext 通过调用注解 Bean 定义读取器

AnnotatedBeanDefinitionReader 的 register()方法向容器注册指定的注解 Bean，注解 Bean 定义读取器向容器注册注解 Bean 的源码如下：

```
//注册多个注解 Bean 定义类
public void register(Class<?>... annotatedClasses) {
    for (Class<?> annotatedClass : annotatedClasses) {
        registerBean(annotatedClass);
    }
}
```

```
//注册一个注解 Bean 定义类
public void registerBean(Class<?> annotatedClass) {
   doRegisterBean(annotatedClass, null, null, null);
}

public <T> void registerBean(Class<T> annotatedClass, @Nullable Supplier<T> instanceSupplier) {
   doRegisterBean(annotatedClass, instanceSupplier, null, null);
}

public <T> void registerBean(Class<T> annotatedClass, String name, @Nullable Supplier<T> instanceSupplier) {
   doRegisterBean(annotatedClass, instanceSupplier, name, null);
}

//Bean 定义读取器注册注解 Bean 定义的入口方法
@SuppressWarnings("unchecked")
public void registerBean(Class<?> annotatedClass, Class<? extends Annotation>... qualifiers) {
   doRegisterBean(annotatedClass, null, null, qualifiers);
}

//Bean 定义读取器向容器注册注解 Bean 定义类
@SuppressWarnings("unchecked")
public void registerBean(Class<?> annotatedClass, String name, Class<? extends Annotation>... qualifiers) {
   doRegisterBean(annotatedClass, null, name, qualifiers);
}

//Bean 定义读取器向容器注册注解 Bean 定义类
<T> void doRegisterBean(Class<T> annotatedClass, @Nullable Supplier<T> instanceSupplier, @Nullable String name,
      @Nullable Class<? extends Annotation>[] qualifiers, BeanDefinitionCustomizer... definitionCustomizers) {

   //根据指定的注解 Bean 定义类，创建 Spring 容器中对注解 Bean 的封装的数据结构
   AnnotatedGenericBeanDefinition abd = new AnnotatedGenericBeanDefinition(annotatedClass);
   if (this.conditionEvaluator.shouldSkip(abd.getMetadata())) {
      return;
   }

   abd.setInstanceSupplier(instanceSupplier);
   //解析注解 Bean 定义的作用域，若@Scope("prototype")，则 Bean 为原型类型；
   //若@Scope("singleton")，则 Bean 为单态类型
   ScopeMetadata scopeMetadata = this.scopeMetadataResolver.resolveScopeMetadata(abd);
   //为注解 Bean 定义设置作用域
   abd.setScope(scopeMetadata.getScopeName());
   //为注解 Bean 定义生成 Bean 名称
   String beanName = (name != null ? name : this.beanNameGenerator.generateBeanName(abd, this.registry));
```

```
    //处理注解 Bean 定义中的通用注解
    AnnotationConfigUtils.processCommonDefinitionAnnotations(abd);
    //如果在向容器注册注解 Bean 定义时，使用了额外的限定符注解，则解析限定符注解。
    //主要是配置的关于 autowiring 自动依赖注入装配的限定条件，即@Qualifier 注解
    //Spring 自动依赖注入装配默认是按类型装配，如果使用@Qualifier 则按名称
    if (qualifiers != null) {
       for (Class<? extends Annotation> qualifier : qualifiers) {
          //如果配置了@Primary 注解，设置该 Bean 为 autowiring 自动依赖注入装//配时的首选
          if (Primary.class == qualifier) {
             abd.setPrimary(true);
          }
          //如果配置了@Lazy 注解，则设置该 Bean 为非延迟初始化，如果没有配置，
          //则该 Bean 为预实例化
          else if (Lazy.class == qualifier) {
             abd.setLazyInit(true);
          }
          //如果使用了除@Primary 和@Lazy 以外的其他注解，则为该 Bean 添加一
          //个 autowiring 自动依赖注入装配限定符，该 Bean 在进 autowiring
          //自动依赖注入装配时，根据名称装配限定符指定的 Bean
          else {
             abd.addQualifier(new AutowireCandidateQualifier(qualifier));
          }
       }
    }
    for (BeanDefinitionCustomizer customizer : definitionCustomizers) {
       customizer.customize(abd);
    }

    //创建一个指定 Bean 名称的 Bean 定义对象，封装注解 Bean 定义类数据
    BeanDefinitionHolder definitionHolder = new BeanDefinitionHolder(abd, beanName);
    //根据注解 Bean 定义类中配置的作用域，创建相应的代理对象
    definitionHolder = AnnotationConfigUtils.applyScopedProxyMode(scopeMetadata, definitionHolder,
this.registry);
    //向 IOC 容器注册注解 Bean 类定义对象
    BeanDefinitionReaderUtils.registerBeanDefinition(definitionHolder, this.registry);
}
```

从上面的源码我们可以看出，注册注解 Bean 定义类的基本步骤：

a、需要使用注解元数据解析器解析注解 Bean 中关于作用域的配置。

b、使用 AnnotationConfigUtils 的 processCommonDefinitionAnnotations()方法处理注解 Bean 定义类中通用的注解。

c、使用 AnnotationConfigUtils 的 applyScopedProxyMode()方法创建对于作用域的代理对象。

d、通过 BeanDefinitionReaderUtils 向容器注册 Bean。

下面我们继续分析这 4 步的具体实现过程

2)、AnnotationScopeMetadataResolver 解析作用域元数据

AnnotationScopeMetadataResolver 通过 resolveScopeMetadata()方法解析注解 Bean 定义类的作用域元信息，即判断注册的 Bean 是原生类型(prototype)还是单态(singleton)类型，其源码如下：

```
//解析注解 Bean 定义类中的作用域元信息
@Override
public ScopeMetadata resolveScopeMetadata(BeanDefinition definition) {
   ScopeMetadata metadata = new ScopeMetadata();
   if (definition instanceof AnnotatedBeanDefinition) {
      AnnotatedBeanDefinition annDef = (AnnotatedBeanDefinition) definition;
      //从注解 Bean 定义类的属性中查找属性为”Scope”的值，即@Scope 注解的值
      //annDef.getMetadata().getAnnotationAttributes 方法将 Bean
      //中所有的注解和注解的值存放在一个 map 集合中
      AnnotationAttributes attributes = AnnotationConfigUtils.attributesFor(
            annDef.getMetadata(), this.scopeAnnotationType);
      //将获取到的@Scope 注解的值设置到要返回的对象中
      if (attributes != null) {
         metadata.setScopeName(attributes.getString("value"));
         //获取@Scope 注解中的 proxyMode 属性值，在创建代理对象时会用到
         ScopedProxyMode proxyMode = attributes.getEnum("proxyMode");
         //如果@Scope 的 proxyMode 属性为 DEFAULT 或者 NO
         if (proxyMode == ScopedProxyMode.DEFAULT) {
            //设置 proxyMode 为 NO
            proxyMode = this.defaultProxyMode;
         }
         //为返回的元数据设置 proxyMode
         metadata.setScopedProxyMode(proxyMode);
      }
   }
   //返回解析的作用域元信息对象
   return metadata;
}
```

上述代码中的 annDef.getMetadata().getAnnotationAttributes()方法就是获取对象中指定类型的注解的值。

3)、AnnotationConfigUtils 处理注解 Bean 定义类中的通用注解

AnnotationConfigUtils 类的 processCommonDefinitionAnnotations()在向容器注册 Bean 之前，首先对注解 Bean 定义类中的通用 Spring 注解进行处理，源码如下：

```
//处理 Bean 定义中通用注解
static void processCommonDefinitionAnnotations(AnnotatedBeanDefinition abd, AnnotatedTypeMetadata metadata) {
   AnnotationAttributes lazy = attributesFor(metadata, Lazy.class);
   //如果 Bean 定义中有@Lazy 注解，则将该 Bean 预实例化属性设置为@lazy 注解的值
   if (lazy != null) {
      abd.setLazyInit(lazy.getBoolean("value"));
   }

   else if (abd.getMetadata() != metadata) {
      lazy = attributesFor(abd.getMetadata(), Lazy.class);
      if (lazy != null) {
         abd.setLazyInit(lazy.getBoolean("value"));
      }
   }
   //如果 Bean 定义中有@Primary 注解，则为该 Bean 设置为 autowiring 自动依赖注入装配的首选对象
   if (metadata.isAnnotated(Primary.class.getName())) {
      abd.setPrimary(true);
   }
   //如果 Bean 定义中有@ DependsOn 注解，则为该 Bean 设置所依赖的 Bean 名称，
   //容器将确保在实例化该 Bean 之前首先实例化所依赖的 Bean
   AnnotationAttributes dependsOn = attributesFor(metadata, DependsOn.class);
   if (dependsOn != null) {
      abd.setDependsOn(dependsOn.getStringArray("value"));
   }

   if (abd instanceof AbstractBeanDefinition) {
      AbstractBeanDefinition absBd = (AbstractBeanDefinition) abd;
      AnnotationAttributes role = attributesFor(metadata, Role.class);
      if (role != null) {
         absBd.setRole(role.getNumber("value").intValue());
      }
      AnnotationAttributes description = attributesFor(metadata, Description.class);
      if (description != null) {
         absBd.setDescription(description.getString("value"));
      }
   }
}
```

4)、AnnotationConfigUtils 根据注解 Bean 定义类中配置的作用域为其应用相应的代理策略

AnnotationConfigUtils 类的 applyScopedProxyMode()方法根据注解 Bean 定义类中配置的作用域@Scope 注解的值，为 Bean 定义应用相应的代理模式，主要是在 Spring 面向切面编程(AOP)中使用。源码如下：

```
//根据作用域为 Bean 应用引用的代码模式
static BeanDefinitionHolder applyScopedProxyMode(
      ScopeMetadata metadata, BeanDefinitionHolder definition, BeanDefinitionRegistry registry) {

   //获取注解 Bean 定义类中@Scope 注解的 proxyMode 属性值
   ScopedProxyMode scopedProxyMode = metadata.getScopedProxyMode();
   //如果配置的@Scope 注解的 proxyMode 属性值为 NO，则不应用代理模式
   if (scopedProxyMode.equals(ScopedProxyMode.NO)) {
      return definition;
   }
   //获取配置的@Scope 注解的 proxyMode 属性值，如果为 TARGET_CLASS
   //则返回 true，如果为 INTERFACES，则返回 false
   boolean proxyTargetClass = scopedProxyMode.equals(ScopedProxyMode.TARGET_CLASS);
   //为注册的 Bean 创建相应模式的代理对象
   return ScopedProxyCreator.createScopedProxy(definition, registry, proxyTargetClass);
}
```

这段为 Bean 引用创建相应模式的代理，这里不做深入的分析。

5)、BeanDefinitionReaderUtils 向容器注册 Bean

BeanDefinitionReaderUtils 主要是校验 BeanDefinition 信息，然后将 Bean 添加到容器中一个管理 BeanDefinition 的 HashMap 中。

扫描指定包并解析为 BeanDefinition

当创建注解处理容器时，如果传入的初始参数是注解 Bean 定义类所在的包时，注解容器将扫描给定的包及其子包，将扫描到的注解 Bean 定义载入并注册。

1)、ClassPathBeanDefinitionScanner 扫描给定的包及其子包

AnnotationConfigApplicationContext 通过调用类路径 Bean 定义扫描器 ClassPathBeanDefinitionScanner 扫描给定包及其子包下的所有类，主要源码如下：

```
public class ClassPathBeanDefinitionScanner extends ClassPathScanningCandidateComponentProvider {
```

```
//创建一个类路径 Bean 定义扫描器
public ClassPathBeanDefinitionScanner(BeanDefinitionRegistry registry) {
   this(registry, true);
}

//为容器创建一个类路径 Bean 定义扫描器，并指定是否使用默认的扫描过滤规则。
//即 Spring 默认扫描配置：@Component、@Repository、@Service、@Controller
//注解的 Bean，同时也支持 JavaEE6 的@ManagedBean 和 JSR-330 的@Named 注解
public ClassPathBeanDefinitionScanner(BeanDefinitionRegistry registry, boolean useDefaultFilters) {
   this(registry, useDefaultFilters, getOrCreateEnvironment(registry));
}

public ClassPathBeanDefinitionScanner(BeanDefinitionRegistry registry, boolean useDefaultFilters,
      Environment environment) {

   this(registry, useDefaultFilters, environment,
         (registry instanceof ResourceLoader ? (ResourceLoader) registry : null));
}

public ClassPathBeanDefinitionScanner(BeanDefinitionRegistry registry, boolean useDefaultFilters,
      Environment environment, @Nullable ResourceLoader resourceLoader) {

   Assert.notNull(registry, "BeanDefinitionRegistry must not be null");
   //为容器设置加载 Bean 定义的注册器
   this.registry = registry;

   if (useDefaultFilters) {
      registerDefaultFilters();
   }
   setEnvironment(environment);
   //为容器设置资源加载器
   setResourceLoader(resourceLoader);
}

//调用类路径 Bean 定义扫描器入口方法
public int scan(String... basePackages) {
   //获取容器中已经注册的 Bean 个数
   int beanCountAtScanStart = this.registry.getBeanDefinitionCount();

   //启动扫描器扫描给定包
   doScan(basePackages);

   // Register annotation config processors, if necessary.
   //注册注解配置(Annotation config)处理器
```

```
        if (this.includeAnnotationConfig) {
            AnnotationConfigUtils.registerAnnotationConfigProcessors(this.registry);
        }

        //返回注册的 Bean 个数
        return (this.registry.getBeanDefinitionCount() - beanCountAtScanStart);
    }

    //类路径 Bean 定义扫描器扫描给定包及其子包
    protected Set<BeanDefinitionHolder> doScan(String... basePackages) {
        Assert.notEmpty(basePackages, "At least one base package must be specified");
        //创建一个集合，存放扫描到 Bean 定义的封装类
        Set<BeanDefinitionHolder> beanDefinitions = new LinkedHashSet<>();
        //遍历扫描所有给定的包
        for (String basePackage : basePackages) {
            //调用父类 ClassPathScanningCandidateComponentProvider 的方法
            //扫描给定类路径，获取符合条件的 Bean 定义
            Set<BeanDefinition> candidates = findCandidateComponents(basePackage);
            //遍历扫描到的 Bean
            for (BeanDefinition candidate : candidates) {
                //获取 Bean 定义类中@Scope 注解的值，即获取 Bean 的作用域
                ScopeMetadata scopeMetadata = this.scopeMetadataResolver.resolveScopeMetadata(candidate);
                //为 Bean 设置注解配置的作用域
                candidate.setScope(scopeMetadata.getScopeName());
                //为 Bean 生成名称
                String beanName = this.beanNameGenerator.generateBeanName(candidate, this.registry);
                //如果扫描到的 Bean 不是 Spring 的注解 Bean，则为 Bean 设置默认值，
                //设置 Bean 的自动依赖注入装配属性等
                if (candidate instanceof AbstractBeanDefinition) {
                    postProcessBeanDefinition((AbstractBeanDefinition) candidate, beanName);
                }
                //如果扫描到的 Bean 是 Spring 的注解 Bean，则处理其通用的 Spring 注解
                if (candidate instanceof AnnotatedBeanDefinition) {
                    //处理注解 Bean 中通用的注解，在分析注解 Bean 定义类读取器时已经分析过
                    AnnotationConfigUtils.processCommonDefinitionAnnotations((AnnotatedBeanDefinition) candidate);
                }
                //根据 Bean 名称检查指定的 Bean 是否需要在容器中注册，或者在容器中冲突
                if (checkCandidate(beanName, candidate)) {
                    BeanDefinitionHolder definitionHolder = new BeanDefinitionHolder(candidate, beanName);
                    //根据注解中配置的作用域，为 Bean 应用相应的代理模式
                    definitionHolder =
                            AnnotationConfigUtils.applyScopedProxyMode(scopeMetadata, definitionHolder,
this.registry);
                    beanDefinitions.add(definitionHolder);
```

```
                //向容器注册扫描到的 Bean
                registerBeanDefinition(definitionHolder, this.registry);
            }
        }
    }
    return beanDefinitions;
}

...

}
```

类路径 Bean 定义扫描器 ClassPathBeanDefinitionScanner 主要通过 findCandidateComponents()方法调用其父类 ClassPathScanningCandidateComponentProvider 类来扫描获取给定包及其子包下的类。

2)、ClassPathScanningCandidateComponentProvider 扫描给定包及其子包的类

ClassPathScanningCandidateComponentProvider 类的 findCandidateComponents()方法具体实现扫描给定类路径包的功能，主要源码如下：

```
public class ClassPathScanningCandidateComponentProvider implements EnvironmentCapable, ResourceLoaderAware {

    //保存过滤规则要包含的注解，即 Spring 默认的@Component、@Repository、@Service、
    //@Controller 注解的 Bean，以及 JavaEE6 的@ManagedBean 和 JSR-330 的@Named 注解
    private final List<TypeFilter> includeFilters = new LinkedList<>();

    //保存过滤规则要排除的注解
    private final List<TypeFilter> excludeFilters = new LinkedList<>();

    //构造方法，该方法在子类 ClassPathBeanDefinitionScanner 的构造方法中被调用
    public ClassPathScanningCandidateComponentProvider(boolean useDefaultFilters) {
        this(useDefaultFilters, new StandardEnvironment());
    }

    public ClassPathScanningCandidateComponentProvider(boolean useDefaultFilters, Environment environment) {
        //如果使用 Spring 默认的过滤规则，则向容器注册过滤规则
        if (useDefaultFilters) {
            registerDefaultFilters();
        }
        setEnvironment(environment);
        setResourceLoader(null);
```

```
    }

    //向容器注册过滤规则
    @SuppressWarnings("unchecked")
    protected void registerDefaultFilters() {
        //向要包含的过滤规则中添加@Component 注解类，注意 Spring 中@Repository
        //@Service 和@Controller 都是 Component，因为这些注解都添加了@Component 注解
        this.includeFilters.add(new AnnotationTypeFilter(Component.class));
        //获取当前类的类加载器
        ClassLoader cl = ClassPathScanningCandidateComponentProvider.class.getClassLoader();
        try {
            //向要包含的过滤规则添加 JavaEE6 的@ManagedBean 注解
            this.includeFilters.add(new AnnotationTypeFilter(
                    ((Class<? extends Annotation>) ClassUtils.forName("javax.annotation.ManagedBean", cl)), false));
            logger.debug("JSR-250 'javax.annotation.ManagedBean' found and supported for component scanning");
        }
        catch (ClassNotFoundException ex) {
            // JSR-250 1.1 API (as included in Java EE 6) not available - simply skip.
        }
        try {
            //向要包含的过滤规则添加@Named 注解
            this.includeFilters.add(new AnnotationTypeFilter(
                    ((Class<? extends Annotation>) ClassUtils.forName("javax.inject.Named", cl)), false));
            logger.debug("JSR-330 'javax.inject.Named' annotation found and supported for component scanning");
        }
        catch (ClassNotFoundException ex) {
            // JSR-330 API not available - simply skip.
        }
    }


    //扫描给定类路径的包
    public Set<BeanDefinition> findCandidateComponents(String basePackage) {
        if (this.componentsIndex != null && indexSupportsIncludeFilters()) {
            return addCandidateComponentsFromIndex(this.componentsIndex, basePackage);
        }
        else {
            return scanCandidateComponents(basePackage);
        }
    }

    private Set<BeanDefinition> addCandidateComponentsFromIndex(CandidateComponentsIndex index, String
basePackage) {
        //创建存储扫描到的类的集合
```

```
    Set<BeanDefinition> candidates = new LinkedHashSet<>();
    try {
       Set<String> types = new HashSet<>();
       for (TypeFilter filter : this.includeFilters) {
          String stereotype = extractStereotype(filter);
          if (stereotype == null) {
             throw new IllegalArgumentException("Failed to extract stereotype from "+ filter);
          }
          types.addAll(index.getCandidateTypes(basePackage, stereotype));
       }
       boolean traceEnabled = logger.isTraceEnabled();
       boolean debugEnabled = logger.isDebugEnabled();
       for (String type : types) {
          //为指定资源获取元数据读取器，元信息读取器通过汇编(ASM)读//取资源元信息
          MetadataReader metadataReader = getMetadataReaderFactory().getMetadataReader(type);
          //如果扫描到的类符合容器配置的过滤规则
          if (isCandidateComponent(metadataReader)) {
             //通过汇编(ASM)读取资源字节码中的 Bean 定义元信息
             AnnotatedGenericBeanDefinition sbd = new AnnotatedGenericBeanDefinition(
                   metadataReader.getAnnotationMetadata());
             if (isCandidateComponent(sbd)) {
                if (debugEnabled) {
                   logger.debug("Using candidate component class from index: " + type);
                }
                candidates.add(sbd);
             }
             else {
                if (debugEnabled) {
                   logger.debug("Ignored because not a concrete top-level class: " + type);
                }
             }
          }
          else {
             if (traceEnabled) {
                logger.trace("Ignored because matching an exclude filter: " + type);
             }
          }
       }
    }
    catch (IOException ex) {
       throw new BeanDefinitionStoreException("I/O failure during classpath scanning", ex);
    }
    return candidates;
 }
```

```
    //判断元信息读取器读取的类是否符合容器定义的注解过滤规则
    protected boolean isCandidateComponent(MetadataReader metadataReader) throws IOException {
        //如果读取的类的注解在排除注解过滤规则中，返回 false
        for (TypeFilter tf : this.excludeFilters) {
            if (tf.match(metadataReader, getMetadataReaderFactory())) {
                return false;
            }
        }
        //如果读取的类的注解在包含的注解的过滤规则中，则返回 ture
        for (TypeFilter tf : this.includeFilters) {
            if (tf.match(metadataReader, getMetadataReaderFactory())) {
                return isConditionMatch(metadataReader);
            }
        }
        //如果读取的类的注解既不在排除规则，也不在包含规则中，则返回 false
        return false;
    }

}
```

注册注解 BeanDefinition

AnnotationConfigWebApplicationContext 是 AnnotationConfigApplicationContext 的 Web 版，它们对于注解 Bean 的注册和扫描是基本相同的，但是 AnnotationConfigWebApplicationContext 对注解 Bean 定义的载入稍有不同，AnnotationConfigWebApplicationContext 注入注解 Bean 定义源码如下：

```
//载入注解 Bean 定义资源
@Override
protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory) {
    //为容器设置注解 Bean 定义读取器
    AnnotatedBeanDefinitionReader reader = getAnnotatedBeanDefinitionReader(beanFactory);
    //为容器设置类路径 Bean 定义扫描器
    ClassPathBeanDefinitionScanner scanner = getClassPathBeanDefinitionScanner(beanFactory);

    //获取容器的 Bean 名称生成器
    BeanNameGenerator beanNameGenerator = getBeanNameGenerator();
    //为注解 Bean 定义读取器和类路径扫描器设置 Bean 名称生成器
    if (beanNameGenerator != null) {
        reader.setBeanNameGenerator(beanNameGenerator);
        scanner.setBeanNameGenerator(beanNameGenerator);
```

```
      beanFactory.registerSingleton(AnnotationConfigUtils.CONFIGURATION_BEAN_NAME_GENERATOR,
beanNameGenerator);
   }

   //获取容器的作用域元信息解析器
   ScopeMetadataResolver scopeMetadataResolver = getScopeMetadataResolver();
   //为注解 Bean 定义读取器和类路径扫描器设置作用域元信息解析器
   if (scopeMetadataResolver != null) {
      reader.setScopeMetadataResolver(scopeMetadataResolver);
      scanner.setScopeMetadataResolver(scopeMetadataResolver);
   }

   if (!this.annotatedClasses.isEmpty()) {
      if (logger.isInfoEnabled()) {
         logger.info("Registering annotated classes: [" +
               StringUtils.collectionToCommaDelimitedString(this.annotatedClasses) + "]");
      }
      reader.register(this.annotatedClasses.toArray(new Class<?>[this.annotatedClasses.size()]));
   }

   if (!this.basePackages.isEmpty()) {
      if (logger.isInfoEnabled()) {
         logger.info("Scanning base packages: [" +
               StringUtils.collectionToCommaDelimitedString(this.basePackages) + "]");
      }
      scanner.scan(this.basePackages.toArray(new String[this.basePackages.size()]));
   }

   //获取容器定义的 Bean 定义资源路径
   String[] configLocations = getConfigLocations();
   //如果定位的 Bean 定义资源路径不为空
   if (configLocations != null) {
      for (String configLocation : configLocations) {
         try {
            //使用当前容器的类加载器加载定位路径的字节码类文件
            Class<?> clazz = ClassUtils.forName(configLocation, getClassLoader());
            if (logger.isInfoEnabled()) {
               logger.info("Successfully resolved class for [" + configLocation + "]");
            }
            reader.register(clazz);
         }
         catch (ClassNotFoundException ex) {
            if (logger.isDebugEnabled()) {
               logger.debug("Could not load class for config location [" + configLocation +
```

```
                    "] - trying package scan. " + ex);
            }
            //如果容器类加载器加载定义路径的 Bean 定义资源失败
            //则启用容器类路径扫描器扫描给定路径包及其子包中的类
            int count = scanner.scan(configLocation);
            if (logger.isInfoEnabled()) {
               if (count == 0) {
                  logger.info("No annotated classes found for specified class/package [" + configLocation + "]");
               }
               else {
                  logger.info("Found " + count + " annotated classes in package [" + configLocation + "]");
               }
            }
         }
      }
   }
}
```

以上就是解析和注入注解配置资源的全过程分析。

Spring 自动装配之依赖注入

依赖注入发生的时间

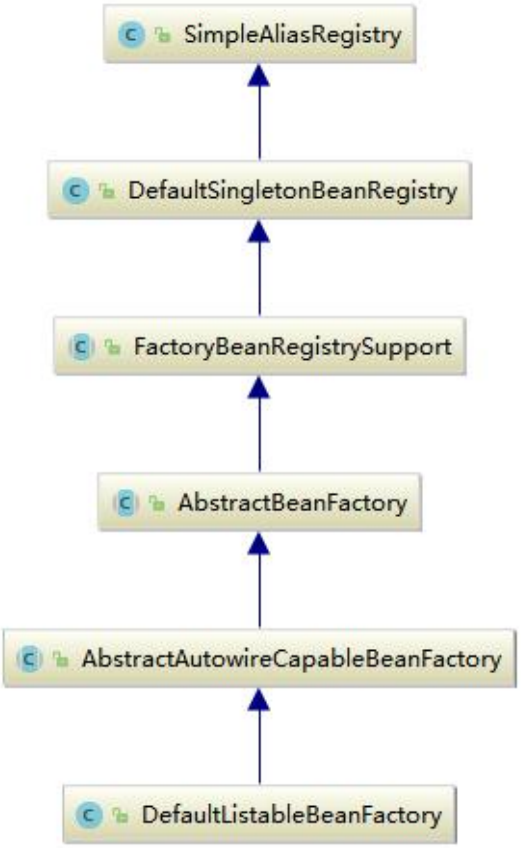
当 Spring IOC 容器完成了 Bean 定义资源的定位、载入和解析注册以后，IOC 容器中已经管理类 Bean 定义的相关数据，但是此时 IOC 容器还没有对所管理的 Bean 进行依赖注入，依赖注入在以下两种情况发生：

1)、用户第一次调用 getBean()方法时，IOC 容器触发依赖注入。

2)、当用户在配置文件中将<bean>元素配置了 lazy-init=false 属性，即让容器在解析注册 Bean 定义时进行预实例化，触发依赖注入。

BeanFactory 接口定义了 Spring IOC 容器的基本功能规范，是 Spring IOC 容器所应遵守的最底层和最基本的编程规范。BeanFactory 接口中定义了几个 getBean()方法，就是用户向 IOC 容器索取管理的

Bean 的方法，我们通过分析其子类的具体实现，理解 Spring IOC 容器在用户索取 Bean 时如何完成依赖注入。



在 BeanFactory 中我们可以看到 getBean(String...)方法，但它具体实现在 AbstractBeanFactory 中。

寻找获取 Bean 的入口

AbstractBeanFactory 的 getBean()相关方法的源码如下：

```
//获取 IOC 容器中指定名称的 Bean
@Override
public Object getBean(String name) throws BeansException {
   //doGetBean 才是真正向 IOC 容器获取被管理 Bean 的过程
   return doGetBean(name, null, null, false);
}
```

```
//获取 IOC 容器中指定名称和类型的 Bean
@Override
public <T> T getBean(String name, @Nullable Class<T> requiredType) throws BeansException {
   //doGetBean 才是真正向 IOC 容器获取被管理 Bean 的过程
   return doGetBean(name, requiredType, null, false);
}

//获取 IOC 容器中指定名称和参数的 Bean
@Override
public Object getBean(String name, Object... args) throws BeansException {
   //doGetBean 才是真正向 IOC 容器获取被管理 Bean 的过程
   return doGetBean(name, null, args, false);
}

//获取 IOC 容器中指定名称、类型和参数的 Bean
public <T> T getBean(String name, @Nullable Class<T> requiredType, @Nullable Object... args)
      throws BeansException {
   //doGetBean 才是真正向 IOC 容器获取被管理 Bean 的过程
   return doGetBean(name, requiredType, args, false);
}

@SuppressWarnings("unchecked")
//真正实现向 IOC 容器获取 Bean 的功能，也是触发依赖注入功能的地方
protected <T> T doGetBean(final String name, @Nullable final Class<T> requiredType,
      @Nullable final Object[] args, boolean typeCheckOnly) throws BeansException {

   //根据指定的名称获取被管理 Bean 的名称，剥离指定名称中对容器的相关依赖
   //如果指定的是别名，将别名转换为规范的 Bean 名称
   final String beanName = transformedBeanName(name);
   Object bean;

   //先从缓存中取是否已经有被创建过的单态类型的 Bean
   //对于单例模式的 Bean 整个 IOC 容器中只创建一次，不需要重复创建
   Object sharedInstance = getSingleton(beanName);
   //IOC 容器创建单例模式 Bean 实例对象
   if (sharedInstance != null && args == null) {
      if (logger.isDebugEnabled()) {
         //如果指定名称的 Bean 在容器中已有单例模式的 Bean 被创建
         //直接返回已经创建的 Bean
         if (isSingletonCurrentlyInCreation(beanName)) {
            logger.debug("Returning eagerly cached instance of singleton bean '" + beanName +
                  "' that is not fully initialized yet - a consequence of a circular reference");
         }
         else {
```

```
            logger.debug("Returning cached instance of singleton bean '" + beanName + "'");
        }
    }
    //获取给定 Bean 的实例对象，主要是完成 FactoryBean 的相关处理
    //注意：BeanFactory 是管理容器中 Bean 的工厂，而 FactoryBean 是
    //创建创建对象的工厂 Bean，两者之间有区别
    bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);
}

else {
    //缓存没有正在创建的单例模式 Bean
    //缓存中已经有已经创建的原型模式 Bean
    //但是由于循环引用的问题导致实例化对象失败
    if (isPrototypeCurrentlyInCreation(beanName)) {
        throw new BeanCurrentlyInCreationException(beanName);
    }

    //对 IOC 容器中是否存在指定名称的 BeanDefinition 进行检查，首先检查是否
    //能在当前的 BeanFactory 中获取的所需要的 Bean，如果不能则委托当前容器
    //的父级容器去查找，如果还是找不到则沿着容器的继承体系向父级容器查找
    BeanFactory parentBeanFactory = getParentBeanFactory();
    //当前容器的父级容器存在，且当前容器中不存在指定名称的 Bean
    if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {
        //解析指定 Bean 名称的原始名称
        String nameToLookup = originalBeanName(name);
        if (parentBeanFactory instanceof AbstractBeanFactory) {
            return ((AbstractBeanFactory) parentBeanFactory).doGetBean(
                    nameToLookup, requiredType, args, typeCheckOnly);
        }
        else if (args != null) {
            //委派父级容器根据指定名称和显式的参数查找
            return (T) parentBeanFactory.getBean(nameToLookup, args);
        }
        else {
            //委派父级容器根据指定名称和类型查找
            return parentBeanFactory.getBean(nameToLookup, requiredType);
        }
    }

    //创建的 Bean 是否需要进行类型验证，一般不需要
    if (!typeCheckOnly) {
        //向容器标记指定的 Bean 已经被创建
        markBeanAsCreated(beanName);
    }
```

```
try {
   //根据指定 Bean 名称获取其父级的 Bean 定义
   //主要解决 Bean 继承时子类合并父类公共属性问题
   final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);
   checkMergedBeanDefinition(mbd, beanName, args);

   //获取当前 Bean 所有依赖 Bean 的名称
   String[] dependsOn = mbd.getDependsOn();
   //如果当前 Bean 有依赖 Bean
   if (dependsOn != null) {
      for (String dep : dependsOn) {
         if (isDependent(beanName, dep)) {
            throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                  "Circular depends-on relationship between '" + beanName + "' and '" + dep + "'");
         }
         //递归调用 getBean 方法，获取当前 Bean 的依赖 Bean
         registerDependentBean(dep, beanName);
         //把被依赖 Bean 注册给当前依赖的 Bean
         getBean(dep);
      }
   }

   //创建单例模式 Bean 的实例对象
   if (mbd.isSingleton()) {
      //这里使用了一个匿名内部类，创建 Bean 实例对象，并且注册给所依赖的对象
      sharedInstance = getSingleton(beanName, () -> {
         try {
            //创建一个指定 Bean 实例对象，如果有父级继承，则合并子类和父类的定义
            return createBean(beanName, mbd, args);
         }
         catch (BeansException ex) {
            //显式地从容器单例模式 Bean 缓存中清除实例对象
            destroySingleton(beanName);
            throw ex;
         }
      });
      //获取给定 Bean 的实例对象
      bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);
   }

   //IOC 容器创建原型模式 Bean 实例对象
   else if (mbd.isPrototype()) {
      //原型模式(Prototype)是每次都会创建一个新的对象
```

```
            Object prototypeInstance = null;
            try {
                //回调 beforePrototypeCreation 方法，默认的功能是注册当前创建的原型对象
                beforePrototypeCreation(beanName);
                //创建指定 Bean 对象实例
                prototypeInstance = createBean(beanName, mbd, args);
            }
            finally {
                //回调 afterPrototypeCreation 方法，默认的功能告诉 IOC 容器指定 Bean 的原型对象不再创建
                afterPrototypeCreation(beanName);
            }
            //获取给定 Bean 的实例对象
            bean = getObjectForBeanInstance(prototypeInstance, name, beanName, mbd);
        }

        //要创建的 Bean 既不是单例模式，也不是原型模式，则根据 Bean 定义资源中
        //配置的生命周期范围，选择实例化 Bean 的合适方法，这种在 Web 应用程序中
        //比较常用，如：request、session、application 等生命周期
        else {
            String scopeName = mbd.getScope();
            final Scope scope = this.scopes.get(scopeName);
            //Bean 定义资源中没有配置生命周期范围，则 Bean 定义不合法
            if (scope == null) {
                throw new IllegalStateException("No Scope registered for scope name '" + scopeName + "'");
            }
            try {
                //这里又使用了一个匿名内部类，获取一个指定生命周期范围的实例
                Object scopedInstance = scope.get(beanName, () -> {
                    beforePrototypeCreation(beanName);
                    try {
                        return createBean(beanName, mbd, args);
                    }
                    finally {
                        afterPrototypeCreation(beanName);
                    }
                });
                //获取给定 Bean 的实例对象
                bean = getObjectForBeanInstance(scopedInstance, name, beanName, mbd);
            }
            catch (IllegalStateException ex) {
                throw new BeanCreationException(beanName,
                        "Scope '" + scopeName + "' is not active for the current thread; consider " +
                        "defining a scoped proxy for this bean if you intend to refer to it from a singleton",
                        ex);
```

```
            }
        }
    }
    catch (BeansException ex) {
        cleanupAfterBeanCreationFailure(beanName);
        throw ex;
    }
}

//对创建的 Bean 实例对象进行类型检查
if (requiredType != null && !requiredType.isInstance(bean)) {
    try {
        T convertedBean = getTypeConverter().convertIfNecessary(bean, requiredType);
        if (convertedBean == null) {
            throw new BeanNotOfRequiredTypeException(name, requiredType, bean.getClass());
        }
        return convertedBean;
    }
    catch (TypeMismatchException ex) {
        if (logger.isDebugEnabled()) {
            logger.debug("Failed to convert bean '" + name + "' to required type '" +
                    ClassUtils.getQualifiedName(requiredType) + "'", ex);
        }
        throw new BeanNotOfRequiredTypeException(name, requiredType, bean.getClass());
    }
}
return (T) bean;
}
```

通过上面对向 IOC 容器获取 Bean 方法的分析，我们可以看到在 Spring 中，如果 Bean 定义的单例模式(Singleton)，则容器在创建之前先从缓存中查找，以确保整个容器中只存在一个实例对象。如果 Bean 定义的是原型模式(Prototype)，则容器每次都会创建一个新的实例对象。除此之外，Bean 定义还可以扩展为指定其生命周期范围。

上面的源码只是定义了根据 Bean 定义的模式，采取的不同创建 Bean 实例对象的策略，具体的 Bean 实例对象的创建过程由实现了 ObjectFactory 接口的匿名内部类的 createBean()方法完成，ObjectFactory 使用委派模式，具体的 Bean 实例创建过程交由其实现类

AbstractAutowireCapableBeanFactory 完成，我们继续分析 AbstractAutowireCapableBeanFactory 的 createBean()方法的源码，理解其创建 Bean 实例的具体实现过程。

开始实例化

AbstractAutowireCapableBeanFactory 类实现了 ObjectFactory 接口，创建容器指定的 Bean 实例对象，同时还对创建的 Bean 实例对象进行初始化处理。其创建 Bean 实例对象的方法源码如下：

```
//创建 Bean 实例对象
@Override
protected Object createBean(String beanName, RootBeanDefinition mbd, @Nullable Object[] args)
      throws BeanCreationException {

   if (logger.isDebugEnabled()) {
      logger.debug("Creating instance of bean '" + beanName + "'");
   }
   RootBeanDefinition mbdToUse = mbd;

   //判断需要创建的 Bean 是否可以实例化，即是否可以通过当前的类加载器加载
   Class<?> resolvedClass = resolveBeanClass(mbd, beanName);
   if (resolvedClass != null && !mbd.hasBeanClass() && mbd.getBeanClassName() != null) {
      mbdToUse = new RootBeanDefinition(mbd);
      mbdToUse.setBeanClass(resolvedClass);
   }

   //校验和准备 Bean 中的方法覆盖
   try {
      mbdToUse.prepareMethodOverrides();
   }
   catch (BeanDefinitionValidationException ex) {
      throw new BeanDefinitionStoreException(mbdToUse.getResourceDescription(),
            beanName, "Validation of Method overrides failed", ex);
   }

   try {
      //如果 Bean 配置了初始化前和初始化后的处理器，则试图返回一个需要创建 Bean 的代理对象
      Object bean = resolveBeforeInstantiation(beanName, mbdToUse);
      if (bean != null) {
         return bean;
      }
   }
   catch (Throwable ex) {
```

```
            throw new BeanCreationException(mbdToUse.getResourceDescription(), beanName,
                  "BeanPostProcessor before instantiation of bean failed", ex);
         }

         try {
            //创建 Bean 的入口
            Object beanInstance = doCreateBean(beanName, mbdToUse, args);
            if (logger.isDebugEnabled()) {
               logger.debug("Finished creating instance of bean '" + beanName + "'");
            }
            return beanInstance;
         }
         catch (BeanCreationException ex) {
            throw ex;
         }
         catch (ImplicitlyAppearedSingletonException ex) {
            throw ex;
         }
         catch (Throwable ex) {
            throw new BeanCreationException(
                  mbdToUse.getResourceDescription(), beanName, "Unexpected exception during bean creation", ex);
         }
      }

      //真正创建 Bean 的方法
      protected Object doCreateBean(final String beanName, final RootBeanDefinition mbd, final @Nullable Object[] args)
            throws BeanCreationException {

         //封装被创建的 Bean 对象
         BeanWrapper instanceWrapper = null;
         if (mbd.isSingleton()) {
            instanceWrapper = this.factoryBeanInstanceCache.remove(beanName);
         }
         if (instanceWrapper == null) {
            instanceWrapper = createBeanInstance(beanName, mbd, args);
         }
         final Object bean = instanceWrapper.getWrappedInstance();
         //获取实例化对象的类型
         Class<?> beanType = instanceWrapper.getWrappedClass();
         if (beanType != NullBean.class) {
            mbd.resolvedTargetType = beanType;
         }

         //调用 PostProcessor 后置处理器
```

```
synchronized (mbd.postProcessingLock) {
   if (!mbd.postProcessed) {
      try {
         applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);
      }
      catch (Throwable ex) {
         throw new BeanCreationException(mbd.getResourceDescription(), beanName,
               "Post-processing of merged bean definition failed", ex);
      }
      mbd.postProcessed = true;
   }
}

//向容器中缓存单例模式的 Bean 对象，以防循环引用
boolean earlySingletonExposure = (mbd.isSingleton() && this.allowCircularReferences &&
      isSingletonCurrentlyInCreation(beanName));
if (earlySingletonExposure) {
   if (logger.isDebugEnabled()) {
      logger.debug("Eagerly caching bean '" + beanName +
            "' to allow for resolving potential circular references");
   }
   //这里是一个匿名内部类，为了防止循环引用，尽早持有对象的引用
   addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd, bean));
}

//Bean 对象的初始化，依赖注入在此触发
//这个 exposedObject 在初始化完成之后返回作为依赖注入完成后的 Bean
Object exposedObject = bean;
try {
   //将 Bean 实例对象封装，并且 Bean 定义中配置的属性值赋值给实例对象
   populateBean(beanName, mbd, instanceWrapper);
   //初始化 Bean 对象
   exposedObject = initializeBean(beanName, exposedObject, mbd);
}
catch (Throwable ex) {
   if (ex instanceof BeanCreationException && beanName.equals(((BeanCreationException) ex).getBeanName())) {
      throw (BeanCreationException) ex;
   }
   else {
      throw new BeanCreationException(
            mbd.getResourceDescription(), beanName, "Initialization of bean failed", ex);
   }
}
```

```
    if (earlySingletonExposure) {
        //获取指定名称的已注册的单例模式 Bean 对象
        Object earlySingletonReference = getSingleton(beanName, false);
        if (earlySingletonReference != null) {
            //根据名称获取的已注册的 Bean 和正在实例化的 Bean 是同一个
            if (exposedObject == bean) {
                //当前实例化的 Bean 初始化完成
                exposedObject = earlySingletonReference;
            }
            //当前 Bean 依赖其他 Bean，并且当发生循环引用时不允许新创建实例对象
            else if (!this.allowRawInjectionDespiteWrapping && hasDependentBean(beanName)) {
                String[] dependentBeans = getDependentBeans(beanName);
                Set<String> actualDependentBeans = new LinkedHashSet<>(dependentBeans.length);
                //获取当前 Bean 所依赖的其他 Bean
                for (String dependentBean : dependentBeans) {
                    //对依赖 Bean 进行类型检查
                    if (!removeSingletonIfCreatedForTypeCheckOnly(dependentBean)) {
                        actualDependentBeans.add(dependentBean);
                    }
                }
                if (!actualDependentBeans.isEmpty()) {
                    throw new BeanCurrentlyInCreationException(beanName,
                            "Bean with name '" + beanName + "' has been injected into other beans [" +
                            StringUtils.collectionToCommaDelimitedString(actualDependentBeans) +
                            "] in its raw version as part of a circular reference, but has eventually been " +
                            "wrapped. This means that said other beans do not use the final version of the " +
                            "bean. This is often the result of over-eager type matching - consider using " +
                            "'getBeanNamesOfType' with the 'allowEagerInit' flag turned off, for example.");
                }
            }
        }
    }

    //注册完成依赖注入的 Bean
    try {
        registerDisposableBeanIfNecessary(beanName, bean, mbd);
    }
    catch (BeanDefinitionValidationException ex) {
        throw new BeanCreationException(
                mbd.getResourceDescription(), beanName, "Invalid destruction signature", ex);
    }

    return exposedObject;
}
```

通过上面的源码注释，我们看到具体的依赖注入实现其实就在以下两个方法中：

1)、createBeanInstance()方法，生成 Bean 所包含的 java 对象实例。

2)、populateBean()方法，对 Bean 属性的依赖注入进行处理。

下面继续分析这两个方法的代码实现。

选择 Bean 实例化策略

在 createBeanInstance()方法中，根据指定的初始化策略，使用简单工厂、工厂方法或者容器的自动装配特性生成 Java 实例对象，创建对象的源码如下：

```
//创建 Bean 的实例对象
protected BeanWrapper createBeanInstance(String beanName, RootBeanDefinition mbd, @Nullable Object[] args) {
   //检查确认 Bean 是可实例化的
   Class<?> beanClass = resolveBeanClass(mbd, beanName);

   //使用工厂方法对 Bean 进行实例化
   if (beanClass != null && !Modifier.isPublic(beanClass.getModifiers()) && !mbd.isNonPublicAccessAllowed()) {
      throw new BeanCreationException(mbd.getResourceDescription(), beanName,
            "Bean class isn't public, and non-public access not allowed: " + beanClass.getName());
   }

   Supplier<?> instanceSupplier = mbd.getInstanceSupplier();
   if (instanceSupplier != null) {
      return obtainFromSupplier(instanceSupplier, beanName);
   }

   if (mbd.getFactoryMethodName() != null)  {
      //调用工厂方法实例化
      return instantiateUsingFactoryMethod(beanName, mbd, args);
   }

   //使用容器的自动装配方法进行实例化
   boolean resolved = false;
   boolean autowireNecessary = false;
   if (args == null) {
      synchronized (mbd.constructorArgumentLock) {
         if (mbd.resolvedConstructorOrFactoryMethod != null) {
            resolved = true;
            autowireNecessary = mbd.constructorArgumentsResolved;
         }
```

```
      }
   }
   if (resolved) {
      if (autowireNecessary) {
         //配置了自动装配属性，使用容器的自动装配实例化
         //容器的自动装配是根据参数类型匹配 Bean 的构造方法
         return autowireConstructor(beanName, mbd, null, null);
      }
      else {
         //使用默认的无参构造方法实例化
         return instantiateBean(beanName, mbd);
      }
   }

   //使用 Bean 的构造方法进行实例化
   Constructor<?>[] ctors = determineConstructorsFromBeanPostProcessors(beanClass, beanName);
   if (ctors != null ||
         mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_CONSTRUCTOR ||
         mbd.hasConstructorArgumentValues() || !ObjectUtils.isEmpty(args))  {
      //使用容器的自动装配特性，调用匹配的构造方法实例化
      return autowireConstructor(beanName, mbd, ctors, args);
   }

   //使用默认的无参构造方法实例化
   return instantiateBean(beanName, mbd);
}

//使用默认的无参构造方法实例化 Bean 对象
protected BeanWrapper instantiateBean(final String beanName, final RootBeanDefinition mbd) {
   try {
      Object beanInstance;
      final BeanFactory parent = this;
      //获取系统的安全管理接口，JDK 标准的安全管理 API
      if (System.getSecurityManager() != null) {
         //这里是一个匿名内置类，根据实例化策略创建实例对象
         beanInstance = AccessController.doPrivileged((PrivilegedAction<Object>) () ->
               getInstantiationStrategy().instantiate(mbd, beanName, parent),
               getAccessControlContext());
      }
      else {
         //将实例化的对象封装起来
         beanInstance = getInstantiationStrategy().instantiate(mbd, beanName, parent);
      }
      BeanWrapper bw = new BeanWrapperImpl(beanInstance);
```

```
      initBeanWrapper(bw);
      return bw;
   }
   catch (Throwable ex) {
      throw new BeanCreationException(
            mbd.getResourceDescription(), beanName, "Instantiation of bean failed", ex);
   }
}
```

经过对上面的代码分析，我们可以看出，对使用工厂方法和自动装配特性的 Bean 的实例化相当比较清楚，调用相应的工厂方法或者参数匹配的构造方法即可完成实例化对象的工作，但是对于我们最常使用的默认无参构造方法就需要使用相应的初始化策略(JDK 的反射机制或者 CGLib)来进行初始化了，在方法 getInstantiationStrategy().instantiate()中就具体实现类使用初始策略实例化对象。

执行 Bean 实例化

在使用默认的无参构造方法创建 Bean 的实例化对象时，方法 getInstantiationStrategy().instantiate()调用了 SimpleInstantiationStrategy 类中的实例化 Bean 的方法，其源码如下：

```
//使用初始化策略实例化 Bean 对象
@Override
public Object instantiate(RootBeanDefinition bd, @Nullable String beanName, BeanFactory owner) {
   // Don't override the class with CGLib if no overrides.
   //如果 Bean 定义中没有方法覆盖，则就不需要 CGLib 父类类的方法
   if (!bd.hasMethodOverrides()) {
      Constructor<?> constructorToUse;
      synchronized (bd.constructorArgumentLock) {
         //获取对象的构造方法或工厂方法
         constructorToUse = (Constructor<?>) bd.resolvedConstructorOrFactoryMethod;
         //如果没有构造方法且没有工厂方法
         if (constructorToUse == null) {
            //使用 JDK 的反射机制，判断要实例化的 Bean 是否是接口
            final Class<?> clazz = bd.getBeanClass();
            if (clazz.isInterface()) {
               throw new BeanInstantiationException(clazz, "Specified class is an interface");
            }
            try {
               if (System.getSecurityManager() != null) {
                  //这里是一个匿名内置类，使用反射机制获取 Bean 的构造方法
                  constructorToUse = AccessController.doPrivileged(
                        (PrivilegedExceptionAction<Constructor<?>>) () -> clazz.getDeclaredConstructor());
```

```
            }
            else {
               constructorToUse = clazz.getDeclaredConstructor();
            }
            bd.resolvedConstructorOrFactoryMethod = constructorToUse;
         }
         catch (Throwable ex) {
            throw new BeanInstantiationException(clazz, "No default constructor found", ex);
         }
      }
   }
   //使用 BeanUtils 实例化，通过反射机制调用"构造方法.newInstance(arg)"来进行实例化
   return BeanUtils.instantiateClass(constructorToUse);
}
else {
   // Must generate CGLib subclass.
   //使用 CGLib 来实例化对象
   return instantiateWithMethodInjection(bd, beanName, owner);
}
}
```

通过上面的代码分析，我们看到了如果 Bean 有方法被覆盖了，则使用 JDK 的反射机制进行实例化，否则，使用 CGLib 进行实例化。

instantiateWithMethodInjection() 方法调用 SimpleInstantiationStrategy 的子类 CGLibSubclassingInstantiationStrategy 使用 CGLib 来进行初始化，其源码如下：

```
//使用 CGLib 进行 Bean 对象实例化
public Object instantiate(@Nullable Constructor<?> ctor, @Nullable Object... args) {
   //创建代理子类
   Class<?> subclass = createEnhancedSubclass(this.beanDefinition);
   Object instance;
   if (ctor == null) {
      instance = BeanUtils.instantiateClass(subclass);
   }
   else {
      try {
         Constructor<?> enhancedSubclassConstructor = subclass.getConstructor(ctor.getParameterTypes());
         instance = enhancedSubclassConstructor.newInstance(args);
      }
      catch (Exception ex) {
         throw new BeanInstantiationException(this.beanDefinition.getBeanClass(),
               "Failed to invoke constructor for CGLib enhanced subclass [" + subclass.getName() + "]", ex);
```

```
        }
      }

      Factory factory = (Factory) instance;
      factory.setCallbacks(new Callback[] {NoOp.INSTANCE,
            new LookupOverrideMethodInterceptor(this.beanDefinition, this.owner),
            new ReplaceOverrideMethodInterceptor(this.beanDefinition, this.owner)});
      return instance;
   }

   private Class<?> createEnhancedSubclass(RootBeanDefinition beanDefinition) {
      //CGLib 中的类
      Enhancer enhancer = new Enhancer();
      //将 Bean 本身作为其基类
      enhancer.setSuperclass(beanDefinition.getBeanClass());
      enhancer.setNamingPolicy(SpringNamingPolicy.INSTANCE);
      if (this.owner instanceof ConfigurableBeanFactory) {
         ClassLoader cl = ((ConfigurableBeanFactory) this.owner).getBeanClassLoader();
         enhancer.setStrategy(new ClassLoaderAwareGeneratorStrategy(cl));
      }
      enhancer.setCallbackFilter(new MethodOverrideCallbackFilter(beanDefinition));
      enhancer.setCallbackTypes(CALLBACK_TYPES);
      //使用 CGLib 的 createClass 方法生成实例对象
      return enhancer.createClass();
   }
}
```

CGLib 是一个常用的字节码生成器的类库，它提供了一系列 API 实现 Java 字节码的生成和转换功能。

我们在学习 JDK 的动态代理时都知道，JDK 的动态代理只能针对接口，如果一个类没有实现任何接口，要对其进行动态代理只能使用 CGLib。

准备依赖注入

在前面的分析中我们已经了解到 Bean 的依赖注入主要分为两个步骤，首先调用 createBeanInstance()方法生成 Bean 所包含的 Java 对象实例。然后，调用 populateBean()方法，对 Bean 属性的依赖注入进行处理。

上面我们已经分析了容器初始化生成 Bean 所包含的 Java 实例对象的过程，现在我们继续分析生成对象后，Spring IOC 容器是如何将 Bean 的属性依赖关系注入 Bean 实例对象中并设置好的，回到 AbstractAutowireCapableBeanFactory 的 populateBean()方法，对属性依赖注入的代码如下：

```
//将 Bean 属性设置到生成的实例对象上
protected void populateBean(String beanName, RootBeanDefinition mbd, @Nullable BeanWrapper bw) {
   if (bw == null) {
      if (mbd.hasPropertyValues()) {
         throw new BeanCreationException(
               mbd.getResourceDescription(), beanName, "Cannot apply property values to null instance");
      }
      else {
         return;
      }
   }

   boolean continueWithPropertyPopulation = true;
   if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {
      for (BeanPostProcessor bp : getBeanPostProcessors()) {
         if (bp instanceof InstantiationAwareBeanPostProcessor) {
            InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;
            if (!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(), beanName)) {
               continueWithPropertyPopulation = false;
               break;
            }
         }
      }
   }

   if (!continueWithPropertyPopulation) {
      return;
   }
   //获取容器在解析 Bean 定义资源时为 BeanDefiniton 中设置的属性值
   PropertyValues pvs = (mbd.hasPropertyValues() ? mbd.getPropertyValues() : null);

   if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_NAME ||
         mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_TYPE) {
      MutablePropertyValues newPvs = new MutablePropertyValues(pvs);

      if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_NAME) {
         autowireByName(beanName, mbd, bw, newPvs);
      }
```

```
      if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_TYPE) {
         autowireByType(beanName, mbd, bw, newPvs);
      }

      pvs = newPvs;
   }

   boolean hasInstAwareBpps = hasInstantiationAwareBeanPostProcessors();
   boolean needsDepCheck = (mbd.getDependencyCheck() != RootBeanDefinition.DEPENDENCY_CHECK_NONE);

   if (hasInstAwareBpps || needsDepCheck) {
      if (pvs == null) {
         pvs = mbd.getPropertyValues();
      }
      PropertyDescriptor[] filteredPds = filterPropertyDescriptorsForDependencyCheck(bw, mbd.allowCaching);
      if (hasInstAwareBpps) {
         for (BeanPostProcessor bp : getBeanPostProcessors()) {
            if (bp instanceof InstantiationAwareBeanPostProcessor) {
               InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;
               pvs = ibp.postProcessPropertyValues(pvs, filteredPds, bw.getWrappedInstance(), beanName);
               if (pvs == null) {
                  return;
               }
            }
         }
      }
      if (needsDepCheck) {
         checkDependencies(beanName, mbd, filteredPds, pvs);
      }
   }

   if (pvs != null) {
      //对属性进行注入
      applyPropertyValues(beanName, mbd, bw, pvs);
   }
}
//解析并注入依赖属性的过程
protected void applyPropertyValues(String beanName, BeanDefinition mbd, BeanWrapper bw, PropertyValues pvs) {
   if (pvs.isEmpty()) {
      return;
   }
   //封装属性值
   MutablePropertyValues mpvs = null;
```

```
List<PropertyValue> original;

if (System.getSecurityManager() != null) {
   if (bw instanceof BeanWrapperImpl) {
      //设置安全上下文，JDK 安全机制
      ((BeanWrapperImpl) bw).setSecurityContext(getAccessControlContext());
   }
}

if (pvs instanceof MutablePropertyValues) {
   mpvs = (MutablePropertyValues) pvs;
   //属性值已经转换
   if (mpvs.isConverted()) {
      try {
         //为实例化对象设置属性值
         bw.setPropertyValues(mpvs);
         return;
      }
      catch (BeansException ex) {
         throw new BeanCreationException(
               mbd.getResourceDescription(), beanName, "Error setting property values", ex);
      }
   }
   //获取属性值对象的原始类型值
   original = mpvs.getPropertyValueList();
}
else {
   original = Arrays.asList(pvs.getPropertyValues());
}

//获取用户自定义的类型转换
TypeConverter converter = getCustomTypeConverter();
if (converter == null) {
   converter = bw;
}
//创建一个 Bean 定义属性值解析器，将 Bean 定义中的属性值解析为 Bean 实例对象的实际值
BeanDefinitionValueResolver valueResolver = new BeanDefinitionValueResolver(this, beanName, mbd, converter);

//为属性的解析值创建一个拷贝，将拷贝的数据注入到实例对象中
List<PropertyValue> deepCopy = new ArrayList<>(original.size());
boolean resolveNecessary = false;
for (PropertyValue pv : original) {
   //属性值不需要转换
   if (pv.isConverted()) {
```

```
            deepCopy.add(pv);
        }
        //属性值需要转换
        else {
            String propertyName = pv.getName();
            //原始的属性值，即转换之前的属性值
            Object originalValue = pv.getValue();
            //转换属性值，例如将引用转换为 IOC 容器中实例化对象引用
            Object resolvedValue = valueResolver.resolveValueIfNecessary(pv, originalValue);
            //转换之后的属性值
            Object convertedValue = resolvedValue;
            //属性值是否可以转换
            boolean convertible = bw.isWritableProperty(propertyName) &&
                  !PropertyAccessorUtils.isNestedOrIndexedProperty(propertyName);
            if (convertible) {
                //使用用户自定义的类型转换器转换属性值
                convertedValue = convertForProperty(resolvedValue, propertyName, bw, converter);
            }
            //存储转换后的属性值，避免每次属性注入时的转换工作
            if (resolvedValue == originalValue) {
                if (convertible) {
                    //设置属性转换之后的值
                    pv.setConvertedValue(convertedValue);
                }
                deepCopy.add(pv);
            }
            //属性是可转换的，且属性原始值是字符串类型，且属性的原始类型值不是
            //动态生成的字符串，且属性的原始值不是集合或者数组类型
            else if (convertible && originalValue instanceof TypedStringValue &&
                  !((TypedStringValue) originalValue).isDynamic() &&
                  !(convertedValue instanceof Collection || ObjectUtils.isArray(convertedValue))) {
                pv.setConvertedValue(convertedValue);
                //重新封装属性的值
                deepCopy.add(pv);
            }
            else {
                resolveNecessary = true;
                deepCopy.add(new PropertyValue(pv, convertedValue));
            }
        }
    }
    if (mpvs != null && !resolveNecessary) {
        //标记属性值已经转换过
        mpvs.setConverted();
```

```
    }

    //进行属性依赖注入
    try {
        bw.setPropertyValues(new MutablePropertyValues(deepCopy));
    }
    catch (BeansException ex) {
        throw new BeanCreationException(
                mbd.getResourceDescription(), beanName, "Error setting property values", ex);
    }
}
```

分析上述代码，我们可以看出，对属性的注入过程分以下两种情况：

1)、属性值类型不需要强制转换时，不需要解析属性值，直接准备进行依赖注入。

2)、属性值需要进行类型强制转换时，如对其他对象的引用等，首先需要解析属性值，然后对解析后的属性值进行依赖注入。

对属性值的解析是在 BeanDefinitionValueResolver 类中的 resolveValueIfNecessary()方法中进行的，对属性值的依赖注入是通过 bw.setPropertyValues()方法实现的，在分析属性值的依赖注入之前，我们先分析一下对属性值的解析过程。

解析属性注入规则

当容器在对属性进行依赖注入时，如果发现属性值需要进行类型转换，如属性值是容器中另一个 Bean 实例对象的引用，则容器首先需要根据属性值解析出所引用的对象，然后才能将该引用对象注入到目标实例对象的属性上去，对属性进行解析的由 resolveValueIfNecessary()方法实现，其源码如下：

```
//解析属性值，对注入类型进行转换
@Nullable
public Object resolveValueIfNecessary(Object argName, @Nullable Object value) {
    //对引用类型的属性进行解析
    if (value instanceof RuntimeBeanReference) {
        RuntimeBeanReference ref = (RuntimeBeanReference) value;
        //调用引用类型属性的解析方法
        return resolveReference(argName, ref);
    }
    //对属性值是引用容器中另一个 Bean 名称的解析
    else if (value instanceof RuntimeBeanNameReference) {
```

```
        String refName = ((RuntimeBeanNameReference) value).getBeanName();
        refName = String.valueOf(doEvaluate(refName));
        //从容器中获取指定名称的 Bean
        if (!this.beanFactory.containsBean(refName)) {
            throw new BeanDefinitionStoreException(
                    "Invalid bean name '" + refName + "' in bean reference for " + argName);
        }
        return refName;
    }
    //对 Bean 类型属性的解析，主要是 Bean 中的内部类
    else if (value instanceof BeanDefinitionHolder) {
        BeanDefinitionHolder bdHolder = (BeanDefinitionHolder) value;
        return resolveInnerBean(argName, bdHolder.getBeanName(), bdHolder.getBeanDefinition());
    }
    else if (value instanceof BeanDefinition) {
        BeanDefinition bd = (BeanDefinition) value;
        String innerBeanName = "(inner bean)" + BeanFactoryUtils.GENERATED_BEAN_NAME_SEPARATOR +
                ObjectUtils.getIdentityHexString(bd);
        return resolveInnerBean(argName, innerBeanName, bd);
    }
    //对集合数组类型的属性解析
    else if (value instanceof ManagedArray) {
        ManagedArray array = (ManagedArray) value;
        //获取数组的类型
        Class<?> elementType = array.resolvedElementType;
        if (elementType == null) {
            //获取数组元素的类型
            String elementTypeName = array.getElementTypeName();
            if (StringUtils.hasText(elementTypeName)) {
                try {
                    //使用反射机制创建指定类型的对象
                    elementType = ClassUtils.forName(elementTypeName, this.beanFactory.getBeanClassLoader());
                    array.resolvedElementType = elementType;
                }
                catch (Throwable ex) {
                    throw new BeanCreationException(
                            this.beanDefinition.getResourceDescription(), this.beanName,
                            "Error resolving array type for " + argName, ex);
                }
            }
            //没有获取到数组的类型，也没有获取到数组元素的类型
            //则直接设置数组的类型为 Object
            else {
                elementType = Object.class;
```

```
        }
    }
    //创建指定类型的数组
    return resolveManagedArray(argName, (List<?>) value, elementType);
}
//解析 list 类型的属性值
else if (value instanceof ManagedList) {
    return resolveManagedList(argName, (List<?>) value);
}
//解析 set 类型的属性值
else if (value instanceof ManagedSet) {
    return resolveManagedSet(argName, (Set<?>) value);
}
//解析 map 类型的属性值
else if (value instanceof ManagedMap) {
    return resolveManagedMap(argName, (Map<?, ?>) value);
}
//解析 props 类型的属性值，props 其实就是 key 和 value 均为字符串的 map
else if (value instanceof ManagedProperties) {
    Properties original = (Properties) value;
    //创建一个拷贝，用于作为解析后的返回值
    Properties copy = new Properties();
    original.forEach((propKey, propValue) -> {
        if (propKey instanceof TypedStringValue) {
            propKey = evaluate((TypedStringValue) propKey);
        }
        if (propValue instanceof TypedStringValue) {
            propValue = evaluate((TypedStringValue) propValue);
        }
        if (propKey == null || propValue == null) {
            throw new BeanCreationException(
                  this.beanDefinition.getResourceDescription(), this.beanName,
                  "Error converting Properties key/value pair for " + argName + ": resolved to null");
        }
        copy.put(propKey, propValue);
    });
    return copy;
}
//解析字符串类型的属性值
else if (value instanceof TypedStringValue) {
    TypedStringValue typedStringValue = (TypedStringValue) value;
    Object valueObject = evaluate(typedStringValue);
    try {
        //获取属性的目标类型
```

```
        Class<?> resolvedTargetType = resolveTargetType(typedStringValue);
        if (resolvedTargetType != null) {
           //对目标类型的属性进行解析，递归调用
           return this.typeConverter.convertIfNecessary(valueObject, resolvedTargetType);
        }
        //没有获取到属性的目标对象，则按 Object 类型返回
        else {
           return valueObject;
        }
     }
     catch (Throwable ex) {
        throw new BeanCreationException(
              this.beanDefinition.getResourceDescription(), this.beanName,
              "Error converting typed String value for " + argName, ex);
     }
  }
  else if (value instanceof NullBean) {
     return null;
  }
  else {
     return evaluate(value);
  }
}

//解析引用类型的属性值
@Nullable
private Object resolveReference(Object argName, RuntimeBeanReference ref) {
  try {
     Object bean;
     //获取引用的 Bean 名称
     String refName = ref.getBeanName();
     refName = String.valueOf(doEvaluate(refName));
     //如果引用的对象在父类容器中，则从父类容器中获取指定的引用对象
     if (ref.isToParent()) {
        if (this.beanFactory.getParentBeanFactory() == null) {
           throw new BeanCreationException(
                 this.beanDefinition.getResourceDescription(), this.beanName,
                 "Can't resolve reference to bean '" + refName +
                 "' in parent factory: no parent factory available");
        }
        bean = this.beanFactory.getParentBeanFactory().getBean(refName);
     }
     //从当前的容器中获取指定的引用 Bean 对象，如果指定的 Bean 没有被实例化
     //则会递归触发引用 Bean 的初始化和依赖注入
```

```
      else {
         bean = this.beanFactory.getBean(refName);
         //将当前实例化对象的依赖引用对象
         this.beanFactory.registerDependentBean(refName, this.beanName);
      }
      if (bean instanceof NullBean) {
         bean = null;
      }
      return bean;
   }
   catch (BeansException ex) {
      throw new BeanCreationException(
            this.beanDefinition.getResourceDescription(), this.beanName,
            "Cannot resolve reference to bean '" + ref.getBeanName() + "' while setting " + argName, ex);
   }
}

//解析 array 类型的属性
private Object resolveManagedArray(Object argName, List<?> ml, Class<?> elementType) {
   //创建一个指定类型的数组，用于存放和返回解析后的数组
   Object resolved = Array.newInstance(elementType, ml.size());
   for (int i = 0; i < ml.size(); i++) {
      //递归解析 array 的每一个元素，并将解析后的值设置到 resolved 数组中，索引为 i
      Array.set(resolved, i,
            resolveValueIfNecessary(new KeyedArgName(argName, i), ml.get(i)));
   }
   return resolved;
}
```

通过上面的代码分析，我们明白了 Spring 是如何将引用类型，内部类以及集合类型等属性进行解析的，属性值解析完成后就可以进行依赖注入了，依赖注入的过程就是 Bean 对象实例设置到它所依赖的 Bean 对象属性上去。而真正的依赖注入是通过 bw.setPropertyValues()方法实现的，该方法也使用了委托模式，在 BeanWrapper 接口中至少定义了方法声明，依赖注入的具体实现交由其实现类 BeanWrapperImpl 来完成，下面我们就分析依 BeanWrapperImpl 中赖注入相关的源码。

注入赋值

BeanWrapperImpl 类主要是对容器中完成初始化的 Bean 实例对象进行属性的依赖注入，即把 Bean 对象设置到它所依赖的另一个 Bean 的属性中去。然而，BeanWrapperImpl 中的注入方法实际上由 AbstractNestablePropertyAccessor 来实现的，其相关源码如下：

```
//实现属性依赖注入功能
protected void setPropertyValue(PropertyTokenHolder tokens, PropertyValue pv) throws BeansException {
   if (tokens.keys != null) {
      processKeyedProperty(tokens, pv);
   }
   else {
      processLocalProperty(tokens, pv);
   }
}

//实现属性依赖注入功能
@SuppressWarnings("unchecked")
private void processKeyedProperty(PropertyTokenHolder tokens, PropertyValue pv) {
   //调用属性的 getter(readerMethod)方法，获取属性的值
   Object propValue = getPropertyHoldingValue(tokens);
   PropertyHandler ph = getLocalPropertyHandler(tokens.actualName);
   if (ph == null) {
      throw new InvalidPropertyException(
            getRootClass(), this.nestedPath + tokens.actualName, "No property handler found");
   }
   Assert.state(tokens.keys != null, "No token keys");
   String lastKey = tokens.keys[tokens.keys.length - 1];

   //注入 array 类型的属性值
   if (propValue.getClass().isArray()) {
      Class<?> requiredType = propValue.getClass().getComponentType();
      int arrayIndex = Integer.parseInt(lastKey);
      Object oldValue = null;
      try {
         if (isExtractOldValueForEditor() && arrayIndex < Array.getLength(propValue)) {
            oldValue = Array.get(propValue, arrayIndex);
         }
         Object convertedValue = convertIfNecessary(tokens.canonicalName, oldValue, pv.getValue(),
               requiredType, ph.nested(tokens.keys.length));
         //获取集合类型属性的长度
         int length = Array.getLength(propValue);
         if (arrayIndex >= length && arrayIndex < this.autoGrowCollectionLimit) {
            Class<?> componentType = propValue.getClass().getComponentType();
```

```
            Object newArray = Array.newInstance(componentType, arrayIndex + 1);
            System.arraycopy(propValue, 0, newArray, 0, length);
            setPropertyValue(tokens.actualName, newArray);
            //调用属性的 getter(readerMethod)方法，获取属性的值
            propValue = getPropertyValue(tokens.actualName);
         }
         //将属性的值赋值给数组中的元素
         Array.set(propValue, arrayIndex, convertedValue);
      }
      catch (IndexOutOfBoundsException ex) {
         throw new InvalidPropertyException(getRootClass(), this.nestedPath + tokens.canonicalName,
               "Invalid array index in property path '" + tokens.canonicalName + "'", ex);
      }
   }

   //注入 list 类型的属性值
   else if (propValue instanceof List) {
      //获取 list 集合的类型
      Class<?> requiredType = ph.getCollectionType(tokens.keys.length);
      List<Object> list = (List<Object>) propValue;
      //获取 list 集合的 size
      int index = Integer.parseInt(lastKey);
      Object oldValue = null;
      if (isExtractOldValueForEditor() && index < list.size()) {
         oldValue = list.get(index);
      }
      //获取 list 解析后的属性值
      Object convertedValue = convertIfNecessary(tokens.canonicalName, oldValue, pv.getValue(),
            requiredType, ph.nested(tokens.keys.length));
      int size = list.size();
      //如果 list 的长度大于属性值的长度，则多余的元素赋值为 null
      if (index >= size && index < this.autoGrowCollectionLimit) {
         for (int i = size; i < index; i++) {
            try {
               list.add(null);
            }
            catch (NullPointerException ex) {
               throw new InvalidPropertyException(getRootClass(), this.nestedPath + tokens.canonicalName,
                     "Cannot set element with index " + index + " in List of size " +
                     size + ", accessed using property path '" + tokens.canonicalName +
                     "': List does not support filling up gaps with null elements");
            }
         }
         list.add(convertedValue);
```

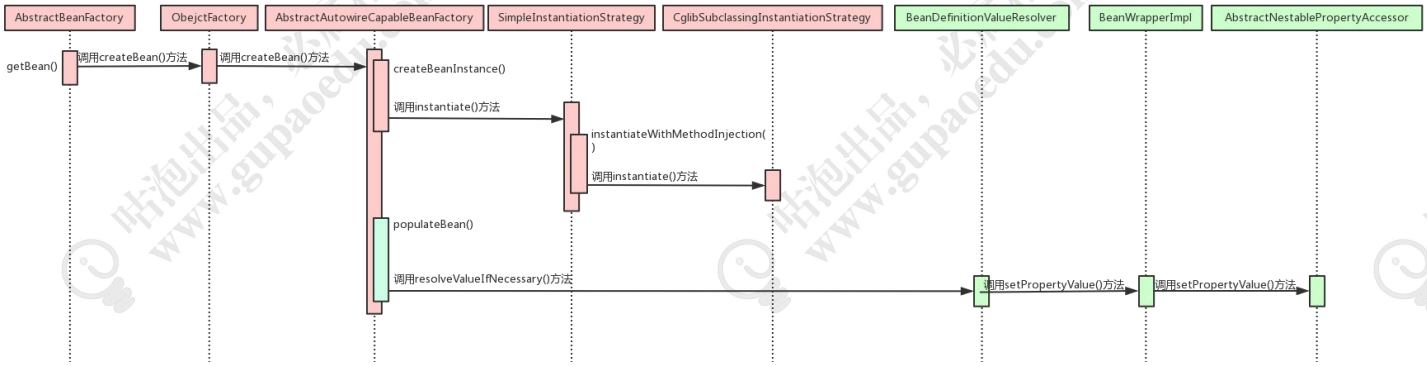
```
      }
      else {
         try {
            //将值添加到 list 中
            list.set(index, convertedValue);
         }
         catch (IndexOutOfBoundsException ex) {
            throw new InvalidPropertyException(getRootClass(), this.nestedPath + tokens.canonicalName,
                  "Invalid list index in property path '" + tokens.canonicalName + "'", ex);
         }
      }
   }

   //注入 map 类型的属性值
   else if (propValue instanceof Map) {
      //获取 map 集合 key 的类型
      Class<?> mapKeyType = ph.getMapKeyType(tokens.keys.length);
      //获取 map 集合 value 的类型
      Class<?> mapValueType = ph.getMapValueType(tokens.keys.length);
      Map<Object, Object> map = (Map<Object, Object>) propValue;
      TypeDescriptor typeDescriptor = TypeDescriptor.valueOf(mapKeyType);
      //解析 map 类型属性 key 值
      Object convertedMapKey = convertIfNecessary(null, null, lastKey, mapKeyType, typeDescriptor);
      Object oldValue = null;
      if (isExtractOldValueForEditor()) {
         oldValue = map.get(convertedMapKey);
      }
      //解析 map 类型属性 value 值
      Object convertedMapValue = convertIfNecessary(tokens.canonicalName, oldValue, pv.getValue(),
            mapValueType, ph.nested(tokens.keys.length));
      //将解析后的 key 和 value 值赋值给 map 集合属性
      map.put(convertedMapKey, convertedMapValue);
   }

   else {
      throw new InvalidPropertyException(getRootClass(), this.nestedPath + tokens.canonicalName,
            "Property referenced in indexed property path '" + tokens.canonicalName +
            "' is neither an array nor a List nor a Map; returned value was [" + propValue + "]");
   }
}

private Object getPropertyHoldingValue(PropertyTokenHolder tokens) {
   Assert.state(tokens.keys != null, "No token keys");
   PropertyTokenHolder getterTokens = new PropertyTokenHolder(tokens.actualName);
```

```
    getterTokens.canonicalName = tokens.canonicalName;
    getterTokens.keys = new String[tokens.keys.length - 1];
    System.arraycopy(tokens.keys, 0, getterTokens.keys, 0, tokens.keys.length - 1);

    Object propValue;
    try {
        //获取属性值
        propValue = getPropertyValue(getterTokens);
    }
    catch (NotReadablePropertyException ex) {
        throw new NotWritablePropertyException(getRootClass(), this.nestedPath + tokens.canonicalName,
                "Cannot access indexed value in property referenced " +
                "in indexed property path '" + tokens.canonicalName + "'", ex);
    }

    if (propValue == null) {
        if (isAutoGrowNestedPaths()) {
            int lastKeyIndex = tokens.canonicalName.lastIndexOf('[');
            getterTokens.canonicalName = tokens.canonicalName.substring(0, lastKeyIndex);
            propValue = setDefaultValue(getterTokens);
        }
        else {
            throw new NullValueInNestedPathException(getRootClass(), this.nestedPath + tokens.canonicalName,
                    "Cannot access indexed value in property referenced " +
                    "in indexed property path '" + tokens.canonicalName + "': returned null");
        }
    }
    return propValue;
}
```

通过对上面注入依赖代码的分析，我们已经明白了 Spring IOC 容器是如何将属性的值注入到 Bean 实例对象中去的：

1)、对于集合类型的属性，将其属性值解析为目标类型的集合后直接赋值给属性。

2)、对于非集合类型的属性，大量使用了 JDK 的反射机制，通过属性的 getter()方法获取指定属性注入以前的值，同时调用属性的 setter()方法为属性设置注入后的值。看到这里相信很多人都明白了 Spring 的 setter()注入原理。



至此 Spring IOC 容器对 Bean 定义资源文件的定位，载入、解析和依赖注入已经全部分析完毕，现在 Spring IOC 容器中管理了一系列靠依赖关系联系起来的 Bean，程序不需要应用自己手动创建所需的对象，Spring IOC 容器会在我们使用的时候自动为我们创建，并且为我们注入好相关的依赖，这就是 Spring 核心功能的控制反转和依赖注入的相关功能。

IOC 容器中那些鲜为人知的事

通过前面章节中对 Spring IOC 容器的源码分析，我们已经基本上了解了 Spring IOC 容器对 Bean 定义资源的定位、载入和注册过程，同时也清楚了当用户通过 getBean()方法向 IOC 容器获取被管理的 Bean 时，IOC 容器对 Bean 进行的初始化和依赖注入过程，这些是 Spring IOC 容器的基本功能特性。Spring IOC 容器还有一些高级特性，如使用 lazy-init 属性对 Bean 预初始化、FactoryBean 产生或者修饰 Bean 对象的生成、IOC 容器初始化 Bean 过程中使用 BeanPostProcessor 后置处理器对 Bean 声明周期事件管理等。

关于延时加载

通过前面我们对 IOC 容器的实现和工作原理分析，我们已经知道 IOC 容器的初始化过程就是对 Bean 定义资源的定位、载入和注册，此时容器对 Bean 的依赖注入并没有发生，依赖注入主要是在应用程序第一次向容器索取 Bean 时，通过 getBean()方法的调用完成。

当 Bean 定义资源的<Bean>元素中配置了 lazy-init=false 属性时，容器将会在初始化的时候对所配置的 Bean 进行预实例化，Bean 的依赖注入在容器初始化的时候就已经完成。这样，当应用程序第一次向容器索取被管理的 Bean 时，就不用再初始化和对 Bean 进行依赖注入了，直接从容器中获取已经完成依赖注入的现成 Bean，可以提高应用第一次向容器获取 Bean 的性能。

1、refresh()方法

先从 IOC 容器的初始化过程开始，我们知道 IOC 容器读入已经定位的 Bean 定义资源是从 refresh()方法开始的，我们首先从 AbstractApplicationContext 类的 refresh()方法入手分析，源码如下：

```
@Override
public void refresh() throws BeansException, IllegalStateException {
   synchronized (this.startupShutdownMonitor) {
      //调用容器准备刷新的方法，获取容器的当时时间，同时给容器设置同步标识
      prepareRefresh();

      //告诉子类启动 refreshBeanFactory()方法，Bean 定义资源文件的载入从
      //子类的 refreshBeanFactory()方法启动
      ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

      //为 BeanFactory 配置容器特性，例如类加载器、事件处理器等
      prepareBeanFactory(beanFactory);

      try {
         //为容器的某些子类指定特殊的 BeanPost 事件处理器
         postProcessBeanFactory(beanFactory);

         //调用所有注册的 BeanFactoryPostProcessor 的 Bean
         invokeBeanFactoryPostProcessors(beanFactory);

         //为 BeanFactory 注册 BeanPost 事件处理器.
         //BeanPostProcessor 是 Bean 后置处理器，用于监听容器触发的事件
         registerBeanPostProcessors(beanFactory);

         //初始化信息源，和国际化相关.
         initMessageSource();

         //初始化容器事件传播器.
         initApplicationEventMulticaster();
```

```
            //调用子类的某些特殊 Bean 初始化方法
            onRefresh();

            //为事件传播器注册事件监听器.
            registerListeners();

            //初始化所有剩余的单例 Bean
            finishBeanFactoryInitialization(beanFactory);

            //初始化容器的生命周期事件处理器，并发布容器的生命周期事件
            finishRefresh();
        }

        catch (BeansException ex) {
            if (logger.isWarnEnabled()) {
                logger.warn("Exception encountered during context initialization - " +
                        "cancelling refresh attempt: " + ex);
            }

            //销毁已创建的 Bean
            destroyBeans();

            //取消 refresh 操作，重置容器的同步标识.
            cancelRefresh(ex);

            throw ex;
        }

        finally {
            resetCommonCaches();
        }
    }
}
```

在 refresh()方法中 ConfigurableListableBeanFactorybeanFactory = obtainFreshBeanFactory();启动了 Bean 定义资源的载入、注册过程，而 finishBeanFactoryInitialization 方法是对注册后的 Bean 定义中的预实例化(lazy-init=false,Spring 默认就是预实例化,即为 true)的 Bean 进行处理的地方。

2、finishBeanFactoryInitialization 处理预实例化 Bean

当 Bean 定义资源被载入 IOC 容器之后，容器将 Bean 定义资源解析为容器内部的数据结构 BeanDefinition 注册到容器中，AbstractApplicationContext 类中的 finishBeanFactoryInitialization() 方法对配置了预实例化属性的 Bean 进行预初始化过程，源码如下：

```
//对配置了 lazy-init 属性的 Bean 进行预实例化处理
protected void finishBeanFactoryInitialization(ConfigurableListableBeanFactory beanFactory) {
   //这是 Spring3 以后新加的代码，为容器指定一个转换服务(ConversionService)
   //在对某些 Bean 属性进行转换时使用
   if (beanFactory.containsBean(CONVERSION_SERVICE_BEAN_NAME) &&
         beanFactory.isTypeMatch(CONVERSION_SERVICE_BEAN_NAME, ConversionService.class)) {
      beanFactory.setConversionService(
            beanFactory.getBean(CONVERSION_SERVICE_BEAN_NAME, ConversionService.class));
   }

   if (!beanFactory.hasEmbeddedValueResolver()) {
      beanFactory.addEmbeddedValueResolver(strVal -> getEnvironment().resolvePlaceholders(strVal));
   }

   String[] weaverAwareNames = beanFactory.getBeanNamesForType(LoadTimeWeaverAware.class, false, false);
   for (String weaverAwareName : weaverAwareNames) {
      getBean(weaverAwareName);
   }

   //为了类型匹配，停止使用临时的类加载器
   beanFactory.setTempClassLoader(null);

   //缓存容器中所有注册的 BeanDefinition 元数据，以防被修改
   beanFactory.freezeConfiguration();

   //对配置了 lazy-init 属性的单态模式 Bean 进行预实例化处理
   beanFactory.preInstantiateSingletons();
}
```

ConfigurableListableBeanFactory 是一个接口，其 preInstantiateSingletons() 方法由其子类 DefaultListableBeanFactory 提供。

3、DefaultListableBeanFactory 对配置 lazy-init 属性单态 Bean 的预实例化

```
public void preInstantiateSingletons() throws BeansException {
   if (this.logger.isDebugEnabled()) {
      this.logger.debug("Pre-instantiating singletons in " + this);
   }
```

```
List<String> beanNames = new ArrayList<>(this.beanDefinitionNames);

for (String beanName : beanNames) {
   //获取指定名称的 Bean 定义
   RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);
   //Bean 不是抽象的，是单态模式的，且 lazy-init 属性配置为 false
   if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {
      //如果指定名称的 bean 是创建容器的 Bean
      if (isFactoryBean(beanName)) {
         //FACTORY_BEAN_PREFIX="&"，当 Bean 名称前面加"&"符号
         //时，获取的是产生容器对象本身，而不是容器产生的 Bean.
         //调用 getBean 方法，触发容器对 Bean 实例化和依赖注入过程
         final FactoryBean<?> factory = (FactoryBean<?>) getBean(FACTORY_BEAN_PREFIX + beanName);
         //标识是否需要预实例化
         boolean isEagerInit;
         if (System.getSecurityManager() != null && factory instanceof SmartFactoryBean) {
            //一个匿名内部类
            isEagerInit = AccessController.doPrivileged((PrivilegedAction<Boolean>) () ->
                  ((SmartFactoryBean<?>) factory).isEagerInit(),
                  getAccessControlContext());
         }
         else {
            isEagerInit = (factory instanceof SmartFactoryBean &&
                  ((SmartFactoryBean<?>) factory).isEagerInit());
         }
         if (isEagerInit) {
            //调用 getBean 方法，触发容器对 Bean 实例化和依赖注入过程
            getBean(beanName);
         }
      }
      else {
         getBean(beanName);
      }
   }
}
```

通过对 lazy-init 处理源码的分析，我们可以看出，如果设置了 lazy-init 属性，则容器在完成 Bean 定义的注册之后，会通过 getBean 方法，触发对指定 Bean 的初始化和依赖注入过程，这样当应用第一次向容器索取所需的 Bean 时，容器不再需要对 Bean 进行初始化和依赖注入，直接从已经完成实例化和依赖注入的 Bean 中取一个现成的 Bean，这样就提高了第一次获取 Bean 的性能。

关于 FactoryBean

在 Spring 中，有两个很容易混淆的类：BeanFactory 和 FactoryBean。

BeanFactory：Bean 工厂，是一个工厂(Factory)，我们 Spring IOC 容器的最顶层接口就是这个 BeanFactory，它的作用是管理 Bean，即实例化、定位、配置应用程序中的对象及建立这些对象间的依赖。

FactoryBean：工厂 Bean，是一个 Bean，作用是产生其他 bean 实例。通常情况下，这种 Bean 没有什么特别的要求，仅需要提供一个工厂方法，该方法用来返回其他 Bean 实例。通常情况下，Bean 无须自己实现工厂模式，Spring 容器担任工厂角色；但少数情况下，容器中的 Bean 本身就是工厂，其作用是产生其它 Bean 实例。

当用户使用容器本身时，可以使用转义字符”&”来得到 FactoryBean 本身，以区别通过 FactoryBean 产生的实例对象和 FactoryBean 对象本身。在 BeanFactory 中通过如下代码定义了该转义字符：

String FACTORY_BEAN_PREFIX = "&";

如果 myJndiObject 是一个 FactoryBean，则使用&myJndiObject 得到的是 myJndiObject 对象，而不是 myJndiObject 产生出来的对象。

1、FactoryBean 源码：

```
//工厂 Bean，用于产生其他对象
public interface FactoryBean<T> {

  //获取容器管理的对象实例
  @Nullable
  T getObject() throws Exception;

  //获取 Bean 工厂创建的对象的类型
  @Nullable
  Class<?> getObjectType();

  //Bean 工厂创建的对象是否是单态模式，如果是单态模式，则整个容器中只有一个实例
  //对象，每次请求都返回同一个实例对象
  default boolean isSingleton() {
```

```
        return true;
    }

}
```

2、AbstractBeanFactory 的 getBean()方法调用 FactoryBean：

在前面我们分析 Spring IOC 容器实例化 Bean 并进行依赖注入过程的源码时，提到在 getBean()方法触发容器实例化 Bean 的时候会调用 AbstractBeanFactory 的 doGetBean()方法来进行实例化的过程，源码如下：

```
//真正实现向 IOC 容器获取 Bean 的功能，也是触发依赖注入功能的地方
protected <T> T doGetBean(final String name, @Nullable final Class<T> requiredType,
      @Nullable final Object[] args, boolean typeCheckOnly) throws BeansException {

   //根据指定的名称获取被管理 Bean 的名称，剥离指定名称中对容器的相关依赖
   //如果指定的是别名，将别名转换为规范的 Bean 名称
   final String beanName = transformedBeanName(name);
   Object bean;

   // Eagerly check singleton cache for manually registered singletons.
   //先从缓存中取是否已经有被创建过的单态类型的 Bean
   //对于单例模式的 Bean 整个 IOC 容器中只创建一次，不需要重复创建
   Object sharedInstance = getSingleton(beanName);
   //IOC 容器创建单例模式 Bean 实例对象
   if (sharedInstance != null && args == null) {
      if (logger.isDebugEnabled()) {
         //如果指定名称的 Bean 在容器中已有单例模式的 Bean 被创建
         //直接返回已经创建的 Bean
         if (isSingletonCurrentlyInCreation(beanName)) {
            logger.debug("Returning eagerly cached instance of singleton bean '" + beanName +
                  "' that is not fully initialized yet - a consequence of a circular reference");
         }
         else {
            logger.debug("Returning cached instance of singleton bean '" + beanName + "'");
         }
      }
      //获取给定 Bean 的实例对象，主要是完成 FactoryBean 的相关处理
      //注意：BeanFactory 是管理容器中 Bean 的工厂，而 FactoryBean 是
      //创建创建对象的工厂 Bean，两者之间有区别
      bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);
   }
```

```
else {
   // Fail if we're already creating this bean instance:
   // We're assumably within a circular reference.
   //缓存没有正在创建的单例模式 Bean
   //缓存中已经有已经创建的原型模式 Bean
   //但是由于循环引用的问题导致实例化对象失败
   if (isPrototypeCurrentlyInCreation(beanName)) {
      throw new BeanCurrentlyInCreationException(beanName);
   }

   // Check if bean definition exists in this factory.
   //对 IOC 容器中是否存在指定名称的 BeanDefinition 进行检查，首先检查是否
   //能在当前的 BeanFactory 中获取的所需要的 Bean，如果不能则委托当前容器
   //的父级容器去查找，如果还是找不到则沿着容器的继承体系向父级容器查找
   BeanFactory parentBeanFactory = getParentBeanFactory();
   //当前容器的父级容器存在，且当前容器中不存在指定名称的 Bean
   if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {
      // Not found -> check parent.
      //解析指定 Bean 名称的原始名称
      String nameToLookup = originalBeanName(name);
      if (parentBeanFactory instanceof AbstractBeanFactory) {
         return ((AbstractBeanFactory) parentBeanFactory).doGetBean(
               nameToLookup, requiredType, args, typeCheckOnly);
      }
      else if (args != null) {
         // Delegation to parent with explicit args.
         //委派父级容器根据指定名称和显式的参数查找
         return (T) parentBeanFactory.getBean(nameToLookup, args);
      }
      else {
         // No args -> delegate to standard getBean Method.
         //委派父级容器根据指定名称和类型查找
         return parentBeanFactory.getBean(nameToLookup, requiredType);
      }
   }

   //创建的 Bean 是否需要进行类型验证，一般不需要
   if (!typeCheckOnly) {
      //向容器标记指定的 Bean 已经被创建
      markBeanAsCreated(beanName);
   }

   try {
      //根据指定 Bean 名称获取其父级的 Bean 定义
```

```
        //主要解决 Bean 继承时子类合并父类公共属性问题
        final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);
        checkMergedBeanDefinition(mbd, beanName, args);

        // Guarantee initialization of beans that the current bean depends on.
        //获取当前 Bean 所有依赖 Bean 的名称
        String[] dependsOn = mbd.getDependsOn();
        //如果当前 Bean 有依赖 Bean
        if (dependsOn != null) {
           for (String dep : dependsOn) {
              if (isDependent(beanName, dep)) {
                 throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                       "Circular depends-on relationship between '" + beanName + "' and '" + dep + "'");
              }
              //递归调用 getBean 方法，获取当前 Bean 的依赖 Bean
              registerDependentBean(dep, beanName);
              //把被依赖 Bean 注册给当前依赖的 Bean
              getBean(dep);
           }
        }

        // Create bean instance.
        //创建单例模式 Bean 的实例对象
        if (mbd.isSingleton()) {
           //这里使用了一个匿名内部类，创建 Bean 实例对象，并且注册给所依赖的对象
           sharedInstance = getSingleton(beanName, () -> {
              try {
                 //创建一个指定 Bean 实例对象，如果有父级继承，则合并子类和父类的定义
                 return createBean(beanName, mbd, args);
              }
              catch (BeansException ex) {
                 // Explicitly remove instance from singleton cache: It might have been put there
                 // eagerly by the creation process, to allow for circular reference resolution.
                 // Also remove any beans that received a temporary reference to the bean.
                 //显式地从容器单例模式 Bean 缓存中清除实例对象
                 destroySingleton(beanName);
                 throw ex;
              }
           });
           //获取给定 Bean 的实例对象
           bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);
        }

        //IOC 容器创建原型模式 Bean 实例对象
```

```
        else if (mbd.isPrototype()) {
            // It's a prototype -> create a new instance.
            //原型模式(Prototype)是每次都会创建一个新的对象
            Object prototypeInstance = null;
            try {
                //回调 beforePrototypeCreation 方法，默认的功能是注册当前创建的原型对象
                beforePrototypeCreation(beanName);
                //创建指定 Bean 对象实例
                prototypeInstance = createBean(beanName, mbd, args);
            }
            finally {
                //回调 afterPrototypeCreation 方法，默认的功能告诉 IOC 容器指定 Bean 的原型对象不再创建
                afterPrototypeCreation(beanName);
            }
            //获取给定 Bean 的实例对象
            bean = getObjectForBeanInstance(prototypeInstance, name, beanName, mbd);
        }

        //要创建的 Bean 既不是单例模式，也不是原型模式，则根据 Bean 定义资源中
        //配置的生命周期范围，选择实例化 Bean 的合适方法，这种在 Web 应用程序中
        //比较常用，如：request、session、application 等生命周期
        else {
            String scopeName = mbd.getScope();
            final Scope scope = this.scopes.get(scopeName);
            //Bean 定义资源中没有配置生命周期范围，则 Bean 定义不合法
            if (scope == null) {
                throw new IllegalStateException("No Scope registered for scope name '" + scopeName + "'");
            }
            try {
                //这里又使用了一个匿名内部类，获取一个指定生命周期范围的实例
                Object scopedInstance = scope.get(beanName, () -> {
                    beforePrototypeCreation(beanName);
                    try {
                        return createBean(beanName, mbd, args);
                    }
                    finally {
                        afterPrototypeCreation(beanName);
                    }
                });
                //获取给定 Bean 的实例对象
                bean = getObjectForBeanInstance(scopedInstance, name, beanName, mbd);
            }
            catch (IllegalStateException ex) {
                throw new BeanCreationException(beanName,
```

```
                    "Scope '" + scopeName + "' is not active for the current thread; consider " +
                    "defining a scoped proxy for this bean if you intend to refer to it from a singleton",
                    ex);
            }
         }
      }
      catch (BeansException ex) {
         cleanupAfterBeanCreationFailure(beanName);
         throw ex;
      }
   }
   ...
   return (T) bean;
}

//获取给定 Bean 的实例对象，主要是完成 FactoryBean 的相关处理
protected Object getObjectForBeanInstance(
      Object beanInstance, String name, String beanName, @Nullable RootBeanDefinition mbd) {

   //容器已经得到了 Bean 实例对象，这个实例对象可能是一个普通的 Bean，
   //也可能是一个工厂 Bean，如果是一个工厂 Bean，则使用它创建一个 Bean 实例对象，
   //如果调用本身就想获得一个容器的引用，则指定返回这个工厂 Bean 实例对象
   //如果指定的名称是容器的解引用(dereference，即是对象本身而非内存地址)，
   //且 Bean 实例也不是创建 Bean 实例对象的工厂 Bean
   if (BeanFactoryUtils.isFactoryDereference(name) && !(beanInstance instanceof FactoryBean)) {
      throw new BeanIsNotAFactoryException(transformedBeanName(name), beanInstance.getClass());
   }

   //如果 Bean 实例不是工厂 Bean，或者指定名称是容器的解引用，
   //调用者向获取对容器的引用，则直接返回当前的 Bean 实例
   if (!(beanInstance instanceof FactoryBean) || BeanFactoryUtils.isFactoryDereference(name)) {
      return beanInstance;
   }

   //处理指定名称不是容器的解引用，或者根据名称获取的 Bean 实例对象是一个工厂 Bean
   //使用工厂 Bean 创建一个 Bean 的实例对象
   Object object = null;
   if (mbd == null) {
      //从 Bean 工厂缓存中获取给定名称的 Bean 实例对象
      object = getCachedObjectForFactoryBean(beanName);
   }
   //让 Bean 工厂生产给定名称的 Bean 对象实例
   if (object == null) {
      FactoryBean<?> factory = (FactoryBean<?>) beanInstance;
```

```
        //如果从 Bean 工厂生产的 Bean 是单态模式的，则缓存
        if (mbd == null && containsBeanDefinition(beanName)) {
            //从容器中获取指定名称的 Bean 定义，如果继承基类，则合并基类相关属性
            mbd = getMergedLocalBeanDefinition(beanName);
        }
        //如果从容器得到 Bean 定义信息，并且 Bean 定义信息不是虚构的，
        //则让工厂 Bean 生产 Bean 实例对象
        boolean synthetic = (mbd != null && mbd.isSynthetic());
        //调用 FactoryBeanRegistrySupport 类的 getObjectFromFactoryBean 方法，
        //实现工厂 Bean 生产 Bean 对象实例的过程
        object = getObjectFromFactoryBean(factory, beanName, !synthetic);
    }
    return object;
}
```

在上面获取给定 Bean 的实例对象的 getObjectForBeanInstance() 方法中，会调用 FactoryBeanRegistrySupport 类的 getObjectFromFactoryBean()方法，该方法实现了 Bean 工厂生产 Bean 实例对象。

Dereference(解引用)：一个在 C/C++中应用比较多的术语，在 C++中，”*”是解引用符号，而”&”是引用符号，解引用是指变量指向的是所引用对象的本身数据，而不是引用对象的内存地址。

3、AbstractBeanFactory 生产 Bean 实例对象

AbstractBeanFactory 类中生产 Bean 实例对象的主要源码如下：

```
//Bean 工厂生产 Bean 实例对象
protected Object getObjectFromFactoryBean(FactoryBean<?> factory, String beanName, boolean shouldPostProcess)
{
    //Bean 工厂是单态模式，并且 Bean 工厂缓存中存在指定名称的 Bean 实例对象
    if (factory.isSingleton() && containsSingleton(beanName)) {
        //多线程同步，以防止数据不一致
        synchronized (getSingletonMutex()) {
            //直接从 Bean 工厂缓存中获取指定名称的 Bean 实例对象
            Object object = this.factoryBeanObjectCache.get(beanName);
            //Bean 工厂缓存中没有指定名称的实例对象，则生产该实例对象
            if (object == null) {
                //调用 Bean 工厂的 getObject 方法生产指定 Bean 的实例对象
                object = doGetObjectFromFactoryBean(factory, beanName);
                Object alreadyThere = this.factoryBeanObjectCache.get(beanName);
                if (alreadyThere != null) {
                    object = alreadyThere;
```

```
            }
            else {
               if (shouldPostProcess) {
                  try {
                     object = postProcessObjectFromFactoryBean(object, beanName);
                  }
                  catch (Throwable ex) {
                     throw new BeanCreationException(beanName,
                           "Post-processing of FactoryBean's singleton object failed", ex);
                  }
               }
               //将生产的实例对象添加到 Bean 工厂缓存中
               this.factoryBeanObjectCache.put(beanName, object);
            }
         }
         return object;
      }
   }
   //调用 Bean 工厂的 getObject 方法生产指定 Bean 的实例对象
   else {
      Object object = doGetObjectFromFactoryBean(factory, beanName);
      if (shouldPostProcess) {
         try {
            object = postProcessObjectFromFactoryBean(object, beanName);
         }
         catch (Throwable ex) {
            throw new BeanCreationException(beanName, "Post-processing of FactoryBean's object failed", ex);
         }
      }
      return object;
   }
}

//调用 Bean 工厂的 getObject 方法生产指定 Bean 的实例对象
private Object doGetObjectFromFactoryBean(final FactoryBean<?> factory, final String beanName)
      throws BeanCreationException {

   Object object;
   try {
      if (System.getSecurityManager() != null) {
         AccessControlContext acc = getAccessControlContext();
         try {
            //实现 PrivilegedExceptionAction 接口的匿名内置类
            //根据 JVM 检查权限，然后决定 BeanFactory 创建实例对象
```

```
            object = AccessController.doPrivileged((PrivilegedExceptionAction<Object>) () ->
                  factory.getObject(), acc);
         }
         catch (PrivilegedActionException pae) {
            throw pae.getException();
         }
      }
      else {
         //调用 BeanFactory 接口实现类的创建对象方法
         object = factory.getObject();
      }
   }
   catch (FactoryBeanNotInitializedException ex) {
      throw new BeanCurrentlyInCreationException(beanName, ex.toString());
   }
   catch (Throwable ex) {
      throw new BeanCreationException(beanName, "FactoryBean threw exception on object creation", ex);
   }

   //创建出来的实例对象为 null，或者因为单态对象正在创建而返回 null
   if (object == null) {
      if (isSingletonCurrentlyInCreation(beanName)) {
         throw new BeanCurrentlyInCreationException(
               beanName, "FactoryBean which is currently in creation returned null from getObject");
      }
      object = new NullBean();
   }
   return object;
}
```

从上面的源码分析中，我们可以看出，BeanFactory 接口调用其实现类的 getObject 方法来实现创建 Bean 实例对象的功能。

4、工厂 Bean 的实现类 getObject 方法创建 Bean 实例对象

FactoryBean 的实现类有非常多，比如：Proxy、RMI、JNDI、ServletContextFactoryBean 等等，FactoryBean 接口为 Spring 容器提供了一个很好的封装机制，具体的 getObject()有不同的实现类根据不同的实现策略来具体提供，我们分析一个最简单的 AnnotationTestFactoryBean 的实现源码：

```
public class AnnotationTestBeanFactory implements FactoryBean<FactoryCreatedAnnotationTestBean> {
   private final FactoryCreatedAnnotationTestBean instance = new FactoryCreatedAnnotationTestBean();
   public AnnotationTestBeanFactory() {
```

```
        this.instance.setName("FACTORY");
    }
    @Override
    public FactoryCreatedAnnotationTestBean getObject() throws Exception {
        return this.instance;
    }
    //AnnotationTestBeanFactory 产生 Bean 实例对象的实现
    @Override
    public Class<? extends IJmxTestBean> getObjectType() {
        return FactoryCreatedAnnotationTestBean.class;
    }
    @Override
    public boolean isSingleton() {
        return true;
    }
}
```

其他的 Proxy，RMI，JNDI 等等，都是根据相应的策略提供 getObject()的实现。这里不做一一分析，这已经不是 Spring 的核心功能，感兴趣的小伙可以再去深入研究。

关于 BeanPostProcessor 后置处理器

BeanPostProcessor 后置处理器是 Spring IOC 容器经常使用到的一个特性，这个 Bean 后置处理器是一个监听器，可以监听容器触发的 Bean 声明周期事件。后置处理器向容器注册以后，容器中管理的 Bean 就具备了接收 IOC 容器事件回调的能力。

BeanPostProcessor 的使用非常简单，只需要提供一个实现接口 BeanPostProcessor 的实现类，然后在 Bean 的配置文件中设置即可。

1、BeanPostProcessor 源码

```
public interface BeanPostProcessor {

    //为在 Bean 的初始化前提供回调入口
    @Nullable
    default Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
        return bean;
    }

    //为在 Bean 的初始化之后提供回调入口
    @Nullable
```

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

}
```

这两个回调的入口都是和容器管理的 Bean 的生命周期事件紧密相关，可以为用户提供在 Spring IOC 容器初始化 Bean 过程中自定义的处理操作。

2、AbstractAutowireCapableBeanFactory 类对容器生成的 Bean 添加后置处理器

BeanPostProcessor 后置处理器的调用发生在 Spring IOC 容器完成对 Bean 实例对象的创建和属性的依赖注入完成之后，在对 Spring 依赖注入的源码分析过程中我们知道，当应用程序第一次调用 getBean()方法(lazy-init 预实例化除外)向 Spring IOC 容器索取指定 Bean 时触发 Spring IOC 容器创建 Bean 实例对象并进行依赖注入的过程，其中真正实现创建 Bean 对象并进行依赖注入的方法是 AbstractAutowireCapableBeanFactory 类的 doCreateBean()方法，主要源码如下：

```
//真正创建 Bean 的方法
protected Object doCreateBean(final String beanName, final RootBeanDefinition mbd, final @Nullable Object[] args)
      throws BeanCreationException {

   //创建 Bean 实例对象

    ...

   Object exposedObject = bean;
   try {
      //对 Bean 属性进行依赖注入
      populateBean(beanName, mbd, instanceWrapper);
      //在对 Bean 实例对象生成和依赖注入完成以后，开始对 Bean 实例对象
      //进行初始化 ，为 Bean 实例对象应用 BeanPostProcessor 后置处理器
      exposedObject = initializeBean(beanName, exposedObject, mbd);
   }
   catch (Throwable ex) {
      if (ex instanceof BeanCreationException && beanName.equals(((BeanCreationException) ex).getBeanName())) {
         throw (BeanCreationException) ex;
      }
      else {
         throw new BeanCreationException(
               mbd.getResourceDescription(), beanName, "Initialization of bean failed", ex);
```

```
        }
    }

    ...

  //为应用返回所需要的实例对象
    return exposedObject;
}
```

从上面的代码中我们知道，为 Bean 实例对象添加 BeanPostProcessor 后置处理器的入口的是 initializeBean()方法。

3、initializeBean()方法为容器产生的 Bean 实例对象添加 BeanPostProcessor 后置处理器

同样在 AbstractAutowireCapableBeanFactory 类中，initializeBean()方法实现为容器创建的 Bean 实例对象添加 BeanPostProcessor 后置处理器，源码如下：

```
//初始容器创建的 Bean 实例对象，为其添加 BeanPostProcessor 后置处理器
protected Object initializeBean(final String beanName, final Object bean, @Nullable RootBeanDefinition mbd) {
    //JDK 的安全机制验证权限
    if (System.getSecurityManager() != null) {
        //实现 PrivilegedAction 接口的匿名内部类
        AccessController.doPrivileged((PrivilegedAction<Object>) () -> {
            invokeAwareMethods(beanName, bean);
            return null;
        }, getAccessControlContext());
    }
    else {
        //为 Bean 实例对象包装相关属性，如名称，类加载器，所属容器等信息
        invokeAwareMethods(beanName, bean);
    }

    Object wrappedBean = bean;
    //对 BeanPostProcessor 后置处理器的 postProcessBeforeInitialization
    //回调方法的调用，为 Bean 实例初始化前做一些处理
    if (mbd == null || !mbd.isSynthetic()) {
        wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);
    }

    //调用 Bean 实例对象初始化的方法，这个初始化方法是在 Spring Bean 定义配置
    //文件中通过 init-Method 属性指定的
    try {
        invokeInitMethods(beanName, wrappedBean, mbd);
```

```
    }
    catch (Throwable ex) {
        throw new BeanCreationException(
                (mbd != null ? mbd.getResourceDescription() : null),
                beanName, "Invocation of init Method failed", ex);
    }
    //对 BeanPostProcessor 后置处理器的 postProcessAfterInitialization
    //回调方法的调用，为 Bean 实例初始化之后做一些处理
    if (mbd == null || !mbd.isSynthetic()) {
        wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);
    }

    return wrappedBean;
}

@Override
//调用 BeanPostProcessor 后置处理器实例对象初始化之前的处理方法
public Object applyBeanPostProcessorsBeforeInitialization(Object existingBean, String beanName)
        throws BeansException {
    Object result = existingBean;
    //遍历容器为所创建的 Bean 添加的所有 BeanPostProcessor 后置处理器
    for (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {
        //调用 Bean 实例所有的后置处理中的初始化前处理方法，为 Bean 实例对象在
        //初始化之前做一些自定义的处理操作
        Object current = beanProcessor.postProcessBeforeInitialization(result, beanName);
        if (current == null) {
            return result;
        }
        result = current;
    }
    return result;
}

@Override
//调用 BeanPostProcessor 后置处理器实例对象初始化之后的处理方法
public Object applyBeanPostProcessorsAfterInitialization(Object existingBean, String beanName)
        throws BeansException {

    Object result = existingBean;
    //遍历容器为所创建的 Bean 添加的所有 BeanPostProcessor 后置处理器
    for (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {
        //调用 Bean 实例所有的后置处理中的初始化后处理方法，为 Bean 实例对象在
        //初始化之后做一些自定义的处理操作
        Object current = beanProcessor.postProcessAfterInitialization(result, beanName);
```

```
        if (current == null) {
            return result;
        }
        result = current;
    }
    return result;
}
```

BeanPostProcessor 是一个接口，其初始化前的操作方法和初始化后的操作方法均委托其实现子类来实现，在 Spring 中，BeanPostProcessor 的实现子类非常的多，分别完成不同的操作，如：AOP 面向切面编程的注册通知适配器、Bean 对象的数据校验、Bean 继承属性、方法的合并等等，我们以最简单的 AOP 切面织入来简单了解其主要的功能。

4、AdvisorAdapterRegistrationManager 在 Bean 对象初始化后注册通知适配器

AdvisorAdapterRegistrationManager 是 BeanPostProcessor 的一个实现类，其主要的作用为容器中管理的 Bean 注册一个面向切面编程的通知适配器，以便在 Spring 容器为所管理的 Bean 进行面向切面编程时提供方便，其源码如下：

```
//为容器中管理的 Bean 注册一个面向切面编程的通知适配器
public class AdvisorAdapterRegistrationManager implements BeanPostProcessor {

    //容器中负责管理切面通知适配器注册的对象
    private AdvisorAdapterRegistry advisorAdapterRegistry = GlobalAdvisorAdapterRegistry.getInstance();

    public void setAdvisorAdapterRegistry(AdvisorAdapterRegistry advisorAdapterRegistry) {
        //如果容器创建的 Bean 实例对象是一个切面通知适配器，则向容器的注册
        this.advisorAdapterRegistry = advisorAdapterRegistry;
    }
    //BeanPostProcessor 在 Bean 对象初始化后的操作
    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
        return bean;
    }

    //BeanPostProcessor 在 Bean 对象初始化前的操作
    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
        if (bean instanceof AdvisorAdapter){
            this.advisorAdapterRegistry.registerAdvisorAdapter((AdvisorAdapter) bean);
```

```
        }
        //没有做任何操作，直接返回容器创建的 Bean 对象
        return bean;
    }

}
```

其他的 BeanPostProcessor 接口实现类的也类似，都是对 Bean 对象使用到的一些特性进行处理，或者向 IOC 容器中注册，为创建的 Bean 实例对象做一些自定义的功能增加，这些操作是容器初始化 Bean 时自动触发的，不需要人为干预。

再述 autowiring

Spring IOC 容器提供了两种管理 Bean 依赖关系的方式：

1)、显式管理：通过 BeanDefinition 的属性值和构造方法实现 Bean 依赖关系管理。

2)、autowiring：Spring IOC 容器的依赖自动装配功能，不需要对 Bean 属性的依赖关系做显式的声明，只需要在配置好 autowiring 属性，IOC 容器会自动使用反射查找属性的类型和名称，然后基于属性的类型或者名称来自动匹配容器中管理的 Bean，从而自动地完成依赖注入。

通过对 autowiring 自动装配特性的理解，我们知道容器对 Bean 的自动装配发生在容器对 Bean 依赖注入的过程中。在前面对 Spring IOC 容器的依赖注入过程源码分析中，我们已经知道了容器对 Bean 实例对象的属性注入的处理发生在 AbstractAutoWireCapableBeanFactory 类中的 populateBean()方法中，我们通过程序流程分析 autowiring 的实现原理：

1、AbstractAutoWireCapableBeanFactory 对 Bean 实例进行属性依赖注入

应用第一次通过 getBean()方法(配置了 lazy-init 预实例化属性的除外)向 IOC 容器索取 Bean 时，容器创建 Bean 实例对象，并且对 Bean 实例对象进行属性依赖注入，AbstractAutoWireCapableBeanFactory 的 populateBean()方法就是实现 Bean 属性依赖注入的功能，其主要源码如下：

```
//将 Bean 属性设置到生成的实例对象上
protected void populateBean(String beanName, RootBeanDefinition mbd, @Nullable BeanWrapper bw) {
```

```
if (bw == null) {
   if (mbd.hasPropertyValues()) {
      throw new BeanCreationException(
            mbd.getResourceDescription(), beanName, "Cannot apply property values to null instance");
   }
   else {
      // Skip property population phase for null instance.
      return;
   }
}

boolean continueWithPropertyPopulation = true;

if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {
   for (BeanPostProcessor bp : getBeanPostProcessors()) {
      if (bp instanceof InstantiationAwareBeanPostProcessor) {
         InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;
         if (!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(), beanName)) {
            continueWithPropertyPopulation = false;
            break;
         }
      }
   }
}

if (!continueWithPropertyPopulation) {
   return;
}
//获取容器在解析 Bean 定义资源时为 BeanDefiniton 中设置的属性值
PropertyValues pvs = (mbd.hasPropertyValues() ? mbd.getPropertyValues() : null);

//对依赖注入处理，首先处理 autowiring 自动装配的依赖注入
if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_NAME ||
      mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_TYPE) {
   MutablePropertyValues newPvs = new MutablePropertyValues(pvs);

   //根据 Bean 名称进行 autowiring 自动装配处理
   if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_NAME) {
      autowireByName(beanName, mbd, bw, newPvs);
   }

   //根据 Bean 类型进行 autowiring 自动装配处理
   if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_TYPE) {
      autowireByType(beanName, mbd, bw, newPvs);
```

```
        }

        pvs = newPvs;
    }

    //对非 autowiring 的属性进行依赖注入处理
    ...
}
```

2、Spring IOC 容器根据 Bean 名称或者类型进行 autowiring 自动依赖注入

```
//根据类型对属性进行自动依赖注入
protected void autowireByType(
        String beanName, AbstractBeanDefinition mbd, BeanWrapper bw, MutablePropertyValues pvs) {

    //获取用户定义的类型转换器
    TypeConverter converter = getCustomTypeConverter();
    if (converter == null) {
        converter = bw;
    }

    //存放解析的要注入的属性
    Set<String> autowiredBeanNames = new LinkedHashSet<>(4);
    //对 Bean 对象中非简单属性(不是简单继承的对象，如 8 中原始类型，字符
    //URL 等都是简单属性)进行处理
    String[] propertyNames = unsatisfiedNonSimpleProperties(mbd, bw);
    for (String propertyName : propertyNames) {
        try {
            //获取指定属性名称的属性描述器
            PropertyDescriptor pd = bw.getPropertyDescriptor(propertyName);
            // Don't try autowiring by type for type Object: never makes sense,
            // even if it technically is a unsatisfied, non-simple property.
            //不对 Object 类型的属性进行 autowiring 自动依赖注入
            if (Object.class != pd.getPropertyType()) {
                //获取属性的 setter 方法
                MethodParameter MethodParam = BeanUtils.getWriteMethodParameter(pd);
                // Do not allow eager init for type matching in case of a prioritized post-processor.
                //检查指定类型是否可以被转换为目标对象的类型
                boolean eager = !PriorityOrdered.class.isInstance(bw.getWrappedInstance());
                //创建一个要被注入的依赖描述
                DependencyDescriptor desc = new AutowireByTypeDependencyDescriptor(MethodParam, eager);
                //根据容器的 Bean 定义解析依赖关系，返回所有要被注入的 Bean 对象
                Object autowiredArgument = resolveDependency(desc, beanName, autowiredBeanNames, converter);
                if (autowiredArgument != null) {
                    //为属性赋值所引用的对象
```

```
                pvs.add(propertyName, autowiredArgument);
            }
            for (String autowiredBeanName : autowiredBeanNames) {
                //指定名称属性注册依赖 Bean 名称，进行属性依赖注入
                registerDependentBean(autowiredBeanName, beanName);
                if (logger.isDebugEnabled()) {
                    logger.debug("Autowiring by type from bean name '" + beanName + "' via property '" +
                            propertyName + "' to bean named '" + autowiredBeanName + "'");
                }
            }
            //释放已自动注入的属性
            autowiredBeanNames.clear();
        }
    }
    catch (BeansException ex) {
        throw new UnsatisfiedDependencyException(mbd.getResourceDescription(), beanName, propertyName, ex);
    }
  }
}
```

通过上面的源码分析，我们可以看出来通过属性名进行自动依赖注入的相对比通过属性类型进行自动依赖注入要稍微简单一些，但是真正实现属性注入的是 DefaultSingletonBeanRegistry 类的 registerDependentBean()方法。

3、DefaultSingletonBeanRegistry 的 registerDependentBean()方法对属性注入

```
//为指定的 Bean 注入依赖的 Bean
public void registerDependentBean(String beanName, String dependentBeanName) {
  //处理 Bean 名称，将别名转换为规范的 Bean 名称
  String canonicalName = canonicalName(beanName);
  Set<String> dependentBeans = this.dependentBeanMap.get(canonicalName);
  if (dependentBeans != null && dependentBeans.contains(dependentBeanName)) {
    return;
  }

  // No entry yet -> fully synchronized manipulation of the dependentBeans Set
  //多线程同步，保证容器内数据的一致性
  //先从容器中：bean 名称-->全部依赖 Bean 名称集合找查找给定名称 Bean 的依赖 Bean
  synchronized (this.dependentBeanMap) {
    //获取给定名称 Bean 的所有依赖 Bean 名称
    dependentBeans = this.dependentBeanMap.get(canonicalName);
    if (dependentBeans == null) {
      //为 Bean 设置依赖 Bean 信息
```

```
            dependentBeans = new LinkedHashSet<>(8);
            this.dependentBeanMap.put(canonicalName, dependentBeans);
        }
        //向容器中：bean 名称-->全部依赖 Bean 名称集合添加 Bean 的依赖信息
        //即，将 Bean 所依赖的 Bean 添加到容器的集合中
        dependentBeans.add(dependentBeanName);
    }
    //从容器中：bean 名称-->指定名称 Bean 的依赖 Bean 集合找查找给定名称 Bean 的依赖 Bean
    synchronized (this.dependenciesForBeanMap) {
        Set<String> dependenciesForBean = this.dependenciesForBeanMap.get(dependentBeanName);
        if (dependenciesForBean == null) {
            dependenciesForBean = new LinkedHashSet<>(8);
            this.dependenciesForBeanMap.put(dependentBeanName, dependenciesForBean);
        }
        //向容器中：bean 名称-->指定 Bean 的依赖 Bean 名称集合添加 Bean 的依赖信息
        //即，将 Bean 所依赖的 Bean 添加到容器的集合中
        dependenciesForBean.add(canonicalName);
    }
}
```

通过对 autowiring 的源码分析，我们可以看出，autowiring 的实现过程：

a、对 Bean 的属性代调用 getBean()方法，完成依赖 Bean 的初始化和依赖注入。

b、将依赖 Bean 的属性引用设置到被依赖的 Bean 属性上。

c、将依赖 Bean 的名称和被依赖 Bean 的名称存储在 IOC 容器的集合中。

Spring IOC 容器的 autowiring 属性自动依赖注入是一个很方便的特性，可以简化开发时的配置，但是凡是都有两面性，自动属性依赖注入也有不足，首先，Bean 的依赖关系在 配置文件中无法很清楚地看出来，对于维护造成一定困难。其次，由于自动依赖注入是 Spring 容器自动执行的，容器是不会智能判断的，如果配置不当，将会带来无法预料的后果，所以自动依赖注入特性在使用时还是综合考虑。