16. Spring原理

本节目标

1. 掌握Bean的作用域和生命周期
2. 了解SpringBoot 自动配置流程

1. Bean的作用域

1.1 概念

在Spring IoC&DI阶段, 我们学习了Spring是如何帮助我们管理对象的.

1. 通过 `@Controller` , `@Service` , `@Repository` , `@Component` , `@Configuration` , `@Bean` 来声明Bean对象.
2. 通过 `ApplicationContext` 或者 `BeanFactory` 来获取对象
3. 通过 `@Autowired` , `Setter` 方法或者构造方法等来为应用程序注入所依赖的Bean对象

我们来简单回顾一下

1. 通过 `@Bean` 声明bean , 把bean存在Spring容器中

```
public class Dog {
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

```
import org.springframework.context.annotation.Bean;
import org.springframework.stereotype.Component;

@Component
public class DogBeanConfig {
```

```
    @Bean
    public Dog dog(){
        Dog dog = new Dog();
        dog.setName("旺旺");
        return dog;
    }
}
```

2. 从Spring容器中获取Bean

```
@SpringBootApplication
public class SpringIocApplication {

    public static void main(String[] args) {
        //获取Spring上下文对象
        ApplicationContext context =
SpringApplication.run(SpringIocApplication.class, args);
        //从Spring上下文中获取对象
        Dog dog = context.getBean(Dog.class);
        System.out.println(dog);
    }
}
```

也可以通过在代码中直接注入ApplicationContext的方式来获取Spring容器

```
@SpringBootTest
class DemoApplicationTests {
   @Autowired
   private ApplicationContext applicationContext; //Spring 容器

   @Test
   void contextLoads() {
      DogBean dog1 = applicationContext.getBean(DogBean.class);
      System.out.println(dog1);
   }
}
```

修改代码, 从Spring容器中多次获取Bean

```
@SpringBootTest
class DemoApplicationTests {
```

```
    @Autowired
    private ApplicationContext applicationContext; //Spring 容器

    @Test
    void contextLoads() {
        Dog dog1 = applicationContext.getBean(Dog.class);
        System.out.println(dog1);

        Dog dog2 = applicationContext.getBean(Dog.class);
        System.out.println(dog2);
    }
}
```

观察运行结果:

```
com.example.demo.scope.Dog@3d37203b
com.example.demo.scope.Dog@3d37203b
```

发现输出的bean对象地址值是一样的, 说明每次**从Spring容器中取出来的对象都是同一个**.

> 这也是"单例模式"
>
> 单例模式: 确保一个类只有一个实例，多次创建也不会创建出多个实例

默认情况下, Spring容器中的bean都是单例的, 这种行为模式, 我们就称之为Bean的作用域.

Bean 的作用域是指 Bean 在 Spring 框架中的某种行为模式.

比如单例作用域: 表示 Bean 在整个 Spring 中只有一份, 它是全局共享的. 那么当其他人修改了这个值之后, 那么另一个人读取到的就是被修改的值.

修改上述代码, 给UserController添加属性name

修改测试代码

```
@SpringBootTest
class DemoApplicationTests {
    @Autowired
    private ApplicationContext applicationContext; //Spring 容器

    @Test
    void contextLoads() {
        Dog dog1 = applicationContext.getBean(Dog.class);
```

```
 9        dog1.setName("狗狗1");
10        System.out.println(dog1);
11        System.out.println(dog1.getName());
12
13        Dog dog2 = applicationContext.getBean(Dog.class);
14        System.out.println(dog2);
15        System.out.println(dog2.getName());
16    }
17 }
```

观察运行结果

```
com.example.demo.scope.Dog@2b10ace9
狗狗1
com.example.demo.scope.Dog@2b10ace9
狗狗1
```

`dog1` 和 `dog2` 为同一个对象，`dog2` 拿到了 `dog1` 设置的值.

那么能不能将bean对象设置为非单例的(每次获取的bean都是一个新对象呢)?

这就是Bean的不同作用域了

1.2 Bean的作用域

在Spring中支持6种作用域，后4种在Spring MVC环境才生效

1. singleton：单例作用域
2. prototype：原型作用域（多例作用域）
3. request：请求作用域
4. session：会话作用域
5. Application: 全局作用域
6. websocket：HTTP WebSocket 作用域

作用域	说明
singleton	每个Spring IoC容器内同名称的bean只有一个实例(单例)(默认)
prototype	每次使用该bean时会创建新的实例(非单例)

request	每个HTTP 请求生命周期内, 创建新的实例(web环境中, 了解)
session	每个HTTP Session生命周期内, 创建新的实例(web环境中, 了解)
application	每个ServletContext生命周期内, 创建新的实例(web环境中, 了解)
websocket	每个WebSocket生命周期内, 创建新的实例(web环境中, 了解)

> 参考文档: https://docs.spring.io/spring-framework/reference/core/beans/factory-scopes.html

我们来简单看下代码实现

定义几个不同作用域的Bean

```
import org.springframework.beans.factory.config.ConfigurableBeanFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Scope;
import org.springframework.context.annotation.ScopedProxyMode;
import org.springframework.stereotype.Component;

@Component
public class DogBeanConfig {
    @Bean
    public Dog dog(){
        Dog dog = new Dog();
        dog.setName("旺旺");
        return dog;
    }

    @Bean
    @Scope(ConfigurableBeanFactory.SCOPE_SINGLETON)
    public Dog singleDog(){
        Dog dog = new Dog();
        return dog;
    }

    @Bean
    @Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
    public Dog prototypeDog(){
        Dog dog = new Dog();
        return dog;
    }
    @Bean
    @RequestScope
    public Dog requestDog() {
        Dog dog = new Dog();
        return dog;
```

```
    }

    @Bean
    @SessionScope
    public Dog sessionDog() {
        Dog dog = new Dog();
        return dog;
    }

    @Bean
    @ApplicationScope
    public Dog applicationDog() {
        Dog dog = new Dog();
        return dog;
    }
}
```

> `@RequestScope` 等同于 `@Scope(value = WebApplicationContext.SCOPE_REQUEST, proxyMode = ScopedProxyMode.TARGET_CLASS)`
>
> `@SessionScope` 等同于 `@Scope(value =WebApplicationContext.SCOPE_SESSION, proxyMode =ScopedProxyMode.TARGET_CLASS)`
>
> `@ApplicationScope` 等同于 `@Scope(value =WebApplicationContext.SCOPE_APPLICATION, proxyMode =ScopedProxyMode.TARGET_CLASS)`
>
> proxyMode用来为spring bean设置代理. `proxyMode = ScopedProxyMode.TARGET_CLASS` 表示这个Bean基于CGLIB实现动态代理. Request, session和application作用域的Bean 需要设置 `proxyMode` .

测试不同作用域的Bean取到的对象是否一样

```
@RestController
public class DogController {
    @Autowired
    private Dog singleDog;
    @Autowired
    private Dog prototypeDog;
    @Autowired
    private Dog requestDog;
    @Autowired
    private Dog sessionDog;
    @Autowired
```

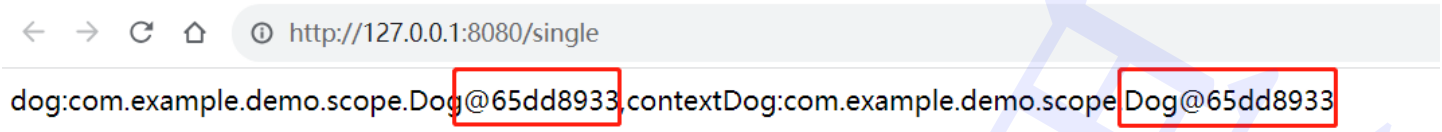
```
12      private ApplicationContext applicationContext;
13
14      @RequestMapping("/single")
15      public String single(){
16          Dog contextDog = (Dog)applicationContext.getBean("singleDog");
17          return "dog:"+singleDog.toString()+",contextDog:"+contextDog;
18      }
19
20      @RequestMapping("/prototype")
21      public String prototype(){
22          Dog contextDog = (Dog)applicationContext.getBean("prototypeDog");
23          return "dog:"+prototypeDog.toString()+",contextDog:"+contextDog;
24      }
25
26      @RequestMapping("/request")
27      public String request(){
28          Dog contextDog = (Dog)applicationContext.getBean("requestDog");
29          return
    "dog:"+requestDog.toString()+",contextDog:"+contextDog.toString();
30      }
31      @RequestMapping("/session")
32      public String session(){
33          Dog contextDog = (Dog)applicationContext.getBean("sessionDog");
34          return
    "dog:"+sessionDog.toString()+",contextDog:"+contextDog.toString();
35      }
36      @RequestMapping("/application")
37      public String application(){
38          Dog contextDog = (Dog)applicationContext.getBean("applicationDog");
39          return
    "dog:"+applicationDog.toString()+",contextDog:"+contextDog.toString();
40      }
41  }
```

> 每个请求都获取两次Bean
>
> `@Autowired` 和 `applicationContext.getBean("singleDog")` 都是从Spring 容器中获取对象.

观察Bean的作用域

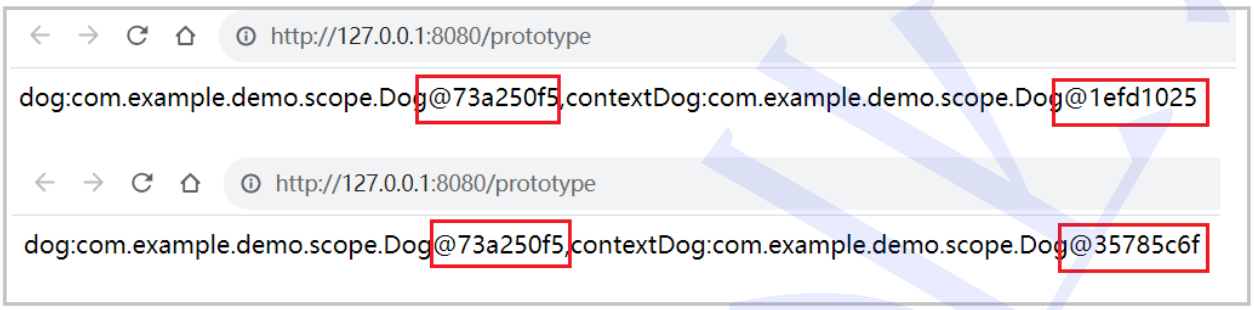
单例作用域: http://127.0.0.1:8080/single

多次访问, 得到的都是同一个对象, 并且 `@Autowired` 和 `applicationContext.getBean()` 也是同一个对象.



多例作用域: http://127.0.0.1:8080/prototype

观察ContextDog, 每次获取的对象都不一样(注入的对象在Spring容器启动时, 就已经注入了, 所以多次请求也不会发生变化)

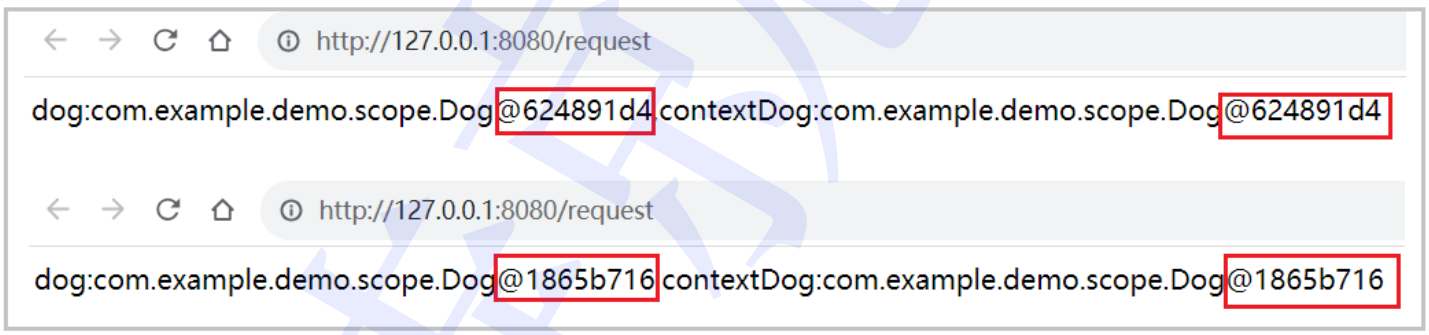


多次访问

请求作用域: http://127.0.0.1:8080/request

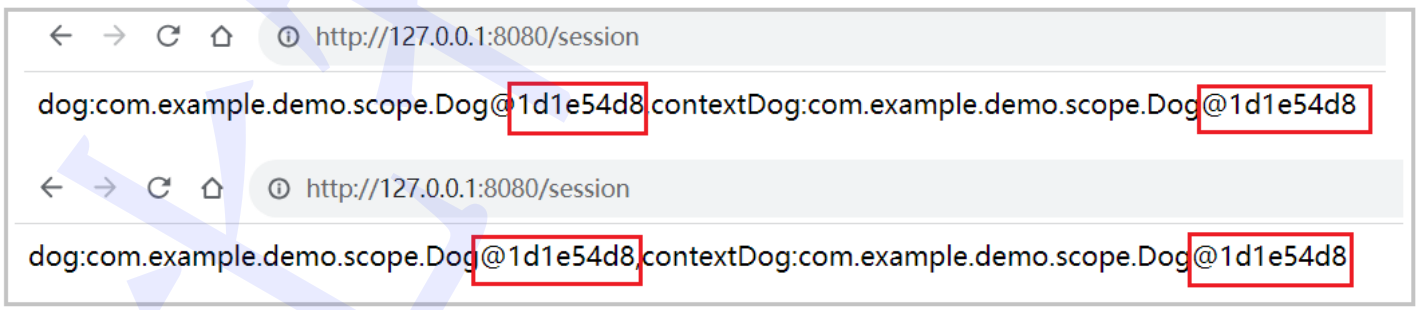
在一次请求中, `@Autowired` 和 `applicationContext.getBean()` 也是同一个对象.

但是每次请求, 都会重新创建对象



会话作用域: http://127.0.0.1:8080/session

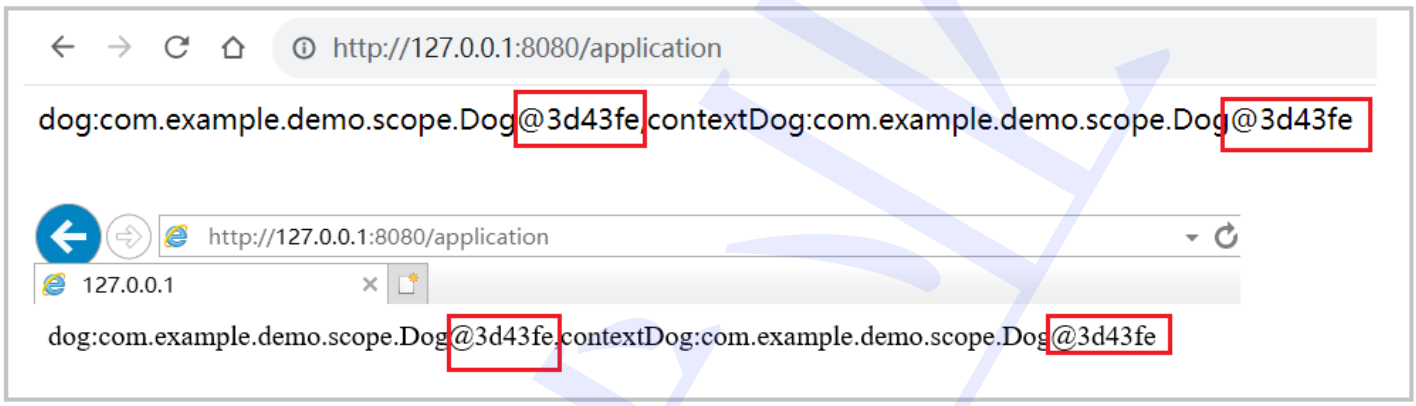
在一个session中, 多次请求, 获取到的对象都是同一个.



换一个浏览器访问, 发现会重新创建对象.(另一个Session)

Application作用域: http://127.0.0.1:8080/application

在一个应用中, 多次访问都是同一个对象



Application scope就是对于整个web容器来说, bean的作用域是ServletContext级别的. 这个和singleton有点类似，区别在于: Application scope是ServletContext的单例, singleton是一个ApplicationContext的单例. 在一个web容器中ApplicationContext可以有多个. (了解即可)

2. Bean的生命周期

生命周期指的是一个对象从诞生到销毁的整个生命过程, 我们把这个过程就叫做一个对象的生命周期.

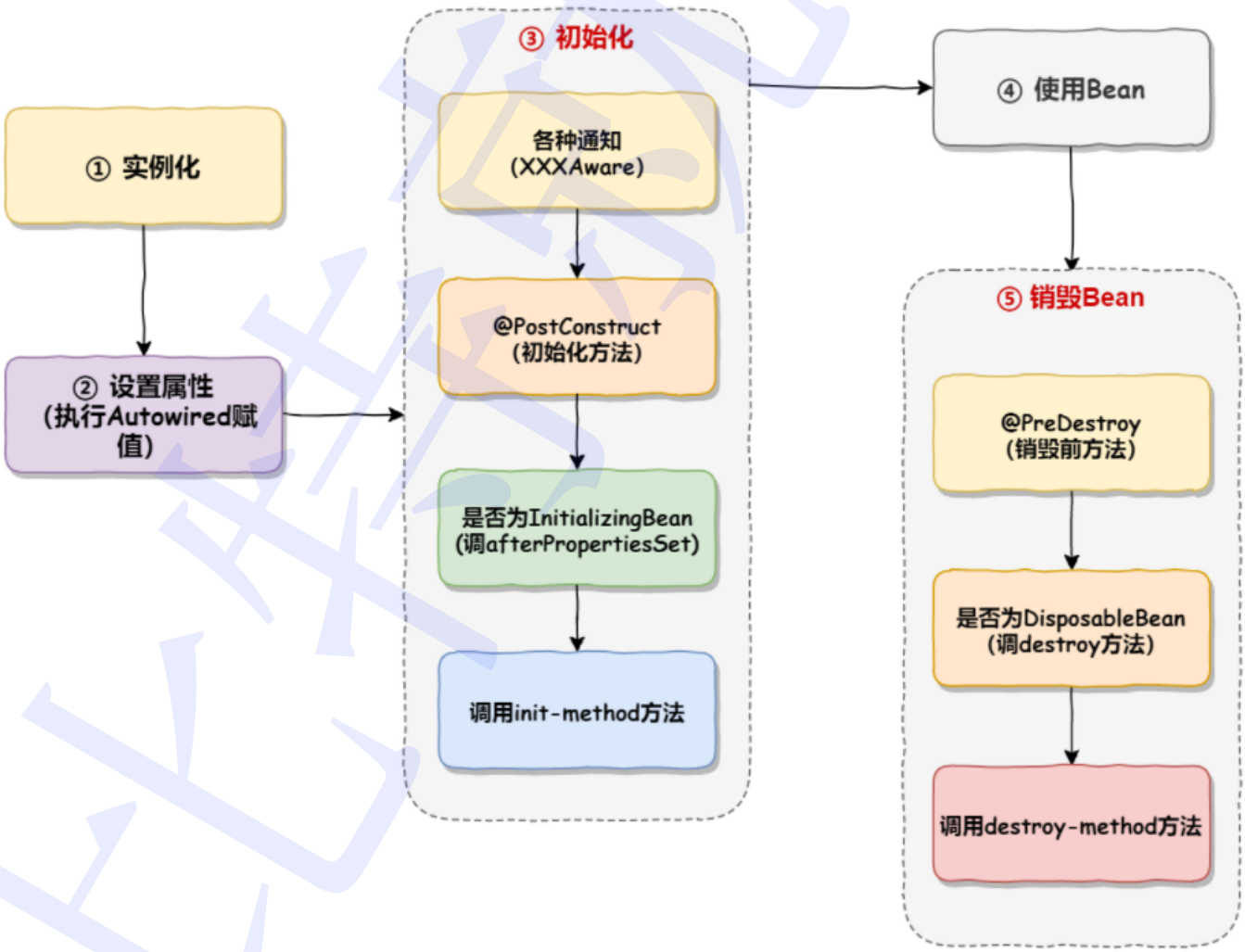
Bean 的生命周期分为以下5个部分:

1. 实例化(为Bean分配内存空间)
2. 属性赋值(Bean注入和装配, 比如 `@AutoWired`)
3. 初始化
   a. 执行各种通知, 如 `BeanNameAware` , `BeanFactoryAware` , `ApplicationContextAware` 的接口方法.
   b. 执行初始化方法
      - xml定义 `init-method`
      - 使用注解的方式 `@PostConstruct`
      - 执行初始化后置方法(`BeanPostProcessor`)
4. 使用Bean

5. 销毁Bean
    a. 销毁容器的各种方法, 如 `@PreDestroy`, `DisposableBean` 接口方法, `destroy-method.`

> 实例化和属性赋值对应构造方法和setter方法的注入. 初始化和销毁是用户能自定义扩展的两个阶段, 可以在实例化之后, 类加载完成之前进行自定义"事件"处理.
>
> 比如我们现在需要买一栋房子, 那么我们的流程是这样的:
>
> 1. 先买房(实例化, 从无到有)
> 2. 装修(设置属性)
> 3. 买家电, 如洗衣机, 冰箱, 电视, 空调等([各种]初始化,可以入住);
> 4. 入住(使用 Bean)
> 5. 卖房(Bean 销毁)

执行流程如下图所示:



2.1 代码演示

```java
import com.example.demo.component.UserComponent;
import org.springframework.beans.factory.BeanNameAware;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import jakarta.annotation.PostConstruct;
import jakarta.annotation.PreDestroy;

@Component
public class BeanLifeComponent implements BeanNameAware {

    private UserComponent userComponent;

    public BeanLifeComponent() {
        System.out.println("执行构造函数");
    }

    @Autowired
    public void setUserComponent(UserComponent userComponent) {
        System.out.println("设置属性userComponent");
        this.userComponent = userComponent;
    }

    @Override
    public void setBeanName(String s) {
        System.out.println("执行了 setBeanName 方法: " + s);
    }

    /**
     * 初始化
     */
    @PostConstruct
    public void postConstruct() {
        System.out.println("执行 PostConstruct()");
    }

    public void use() {
        System.out.println("执行了use方法");
    }

    /**
     * 销毁前执行方法
     */
    @PreDestroy
    public void preDestroy() {
        System.out.println("执行: preDestroy()");
```

```
    }
}
```

执行结果

```
执行构造函数
设置属性
执行了 setBeanName 方法：beanLifeComponent
执行 PostConstruct()
2023-10-15 17:28:46.828  INFO 16028 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2023-10-15 17:28:46.834  INFO 16028 --- [           main] com.example.demo.SpringIocApplication    : Started SpringIocApplication in 1.6 seconds (JVM running for 1.913)
执行了use方法
执行：preDestroy()
```

通过运行结果观察

1. 先执行构造函数
2. 设置属性
3. Bean初始化
4. 使用Bean
5. 销毁Bean

2.2 源码阅读

以上步骤在源码中皆有体现.

创建Bean的代码入口在 `AbstractAutowireCapableBeanFactory#createBean`

```
public abstract class AbstractAutowireCapableBeanFactory extends
AbstractBeanFactory
      implements AutowireCapableBeanFactory {

      protected Object createBean(String beanName, RootBeanDefinition mbd,
@Nullable Object[] args)
      throws BeanCreationException {

      //...代码省略

       try {
          // Give BeanPostProcessors a chance to return a proxy instead of the
target bean instance.
          //在实例化之前，是否有快捷创建的Bean，也就是通过
PostProcessorsBeforeInstantiation返回的Bean
          //如果存在，则会替代原来正常通过target bean生成的bean的流程
          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
            //方法中包含了实例化、属性赋值、初始化过程
            Object beanInstance = doCreateBean(beanName, mbdToUse, args);
            if (logger.isTraceEnabled()) {
                logger.trace("Finished creating instance of bean '" + beanName + "'");
            }
            return beanInstance;
        }
        //...代码省略
     }
}
```

点进去继续看源码: `AbstractAutowireCapableBeanFactory#doCreateBean`

```
protected Object doCreateBean(String beanName, RootBeanDefinition mbd, @Nullable Object[] args)
      throws BeanCreationException {

   //...代码省略
   // Instantiate the bean.
   if (instanceWrapper == null) {
      //实例化Bean
      instanceWrapper = createBeanInstance(beanName, mbd, args);
   }
   //...代码省略
   //初始化bean
   // Initialize the bean instance.
   Object exposedObject = bean;
   try {
      //依据bean definition 完成bean属性赋值
      populateBean(beanName, mbd, instanceWrapper);
      //执行bean初始化
      exposedObject = initializeBean(beanName, exposedObject, mbd);
   }
   catch (Throwable ex) {
```

```
21        if (ex instanceof BeanCreationException &&
beanName.equals(((BeanCreationException) ex).getBeanName())) {
22           throw (BeanCreationException) ex;
23        }
24        else {
25           throw new BeanCreationException(
26                 mbd.getResourceDescription(), beanName, "Initialization of bean
failed", ex);
27        }
28     }
29     //...代码省略
30     return exposedObject;
31  }
```

这三个方法与三个生命周期阶段一一对应

1. createBeanInstance() -> 实例化
2. populateBean() -> 属性赋值
3. initializeBean() -> 初始化

继续点进去 initializeBean

```
1  //初始化Bean
2  protected Object initializeBean(String beanName, Object bean, @Nullable
RootBeanDefinition mbd) {
3     if (System.getSecurityManager() != null) {
4        AccessController.doPrivileged((PrivilegedAction<Object>) () -> {
5           invokeAwareMethods(beanName, bean);
6           return null;
7        }, getAccessControlContext());
8     }
9     else {
10       //调用的三个Bean开头的Aware方法
11
12       invokeAwareMethods(beanName, bean);
13    }
14
15    Object wrappedBean = bean;
16    if (mbd == null || !mbd.isSynthetic()) {
17       wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean,
beanName);
18    }
19
20    try {
21      //
```

```
22          invokeInitMethods(beanName, wrappedBean, mbd);
23       }
24       catch (Throwable ex) {
25          throw new BeanCreationException(
26                (mbd != null ? mbd.getResourceDescription() : null),
27                beanName, "Invocation of init method failed", ex);
28       }
29       if (mbd == null || !mbd.isSynthetic()) {
30          wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean,
    beanName);
31       }
32
33       return wrappedBean;
34    }
35
36    //调用的三个Bean开头的Aware方法
37    private void invokeAwareMethods(String beanName, Object bean) {
38       if (bean instanceof Aware) {
39          if (bean instanceof BeanNameAware) {
40             ((BeanNameAware) bean).setBeanName(beanName);
41          }
42          if (bean instanceof BeanClassLoaderAware) {
43             ClassLoader bcl = getBeanClassLoader();
44             if (bcl != null) {
45                ((BeanClassLoaderAware) bean).setBeanClassLoader(bcl);
46             }
47          }
48          if (bean instanceof BeanFactoryAware) {
49             ((BeanFactoryAware)
    bean).setBeanFactory(AbstractAutowireCapableBeanFactory.this);
50          }
51       }
52    }
```

3. Spring Boot自动配置

SpringBoot的自动配置就是当Spring容器启动后, 一些配置类, bean对象等就自动存入到了IoC容器中, 不需要我们手动去声明, 从而简化了开发, 省去了繁琐的配置操作.

SpringBoot自动配置, 就是指SpringBoot是如何将依赖jar包中的配置类以及Bean加载到Spring IoC容器中的.

我们学习主要分以下两个方面:

1. Spring 是如何把对象加载到SpringIoC容器中的
2. SpringBoot 是如何实现的

3.1 Spring 加载Bean

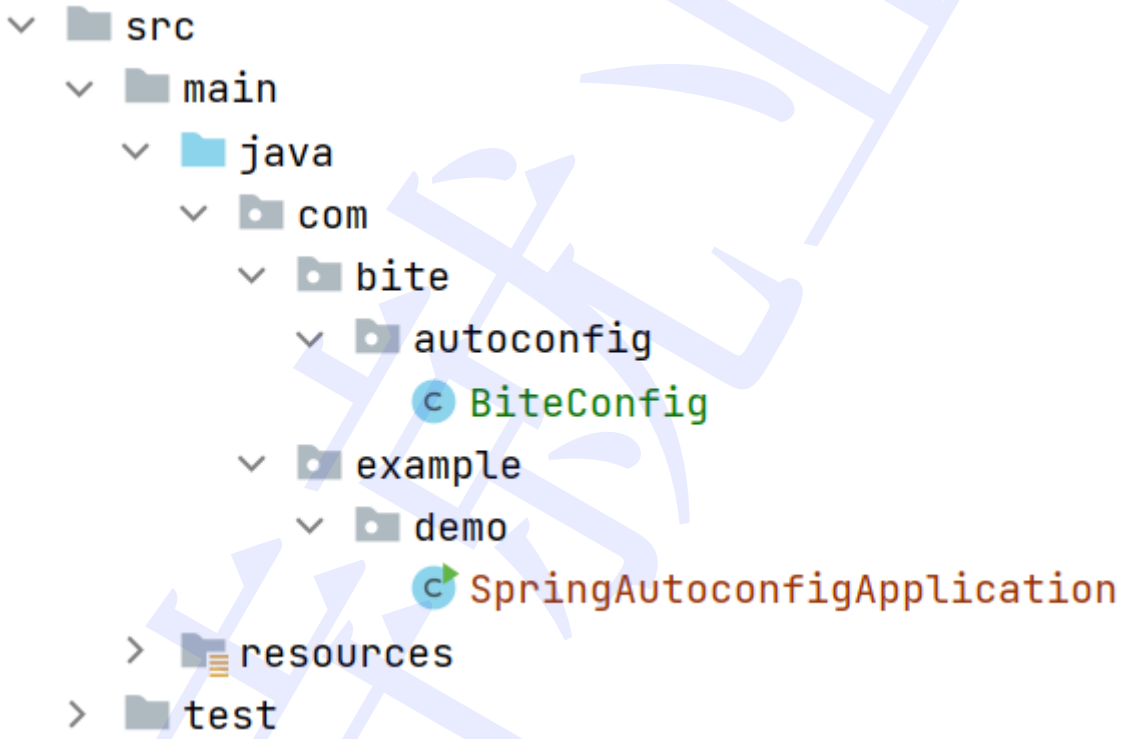
3.1.1 问题描述

需求: 使用Spring管理第三方的jar包的配置

> 引入第三方的包, 其实就是在该项目下,引入第三方的代码, 我们采用在该项目下创建不同的目录来模拟第三方的代码引入

数据准备:

1. 创建项目spring-autoconfig, 当前项目目录为 `com.example.demo`
2. 模拟第三方代码文件在 `com.bite.autoconfig` 目录下



第三方文件代码:

```
@Component
public class BiteConfig {
    public void study(){
        System.out.println("start study...");
    }
}
```

3. 获取 `BiteConfig` 这个Bean

写测试代码

```java
@SpringBootTest
class SpringAutoconfigApplicationTests {
    @Autowired
    private ApplicationContext applicationContext;
    @Test
    void contextLoads() {
        BiteConfig biteConfig = applicationContext.getBean(BiteConfig.class,
"biteConfig");
        System.out.println(biteConfig);
    }

}
```

4. 运行程序:

```
org.springframework.beans.factory.NoSuchBeanDefinitionException: No qualifying bean of type 'com.bite.autoconfig.BiteConfig'
 available

    at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBean(DefaultListableBeanFactory.java:351)
    at org.springframework.context.support.AbstractApplicationContext.getBean(AbstractApplicationContext.java:1184)
    at com.example.demo.SpringAutoconfigApplicationTests.contextLoads(SpringAutoconfigApplicationTests.java:15) <31 internal lines>
    at java.util.ArrayList.forEach(ArrayList.java:1259) <9 internal lines>
    at java.util.ArrayList.forEach(ArrayList.java:1259) <27 internal lines>
```

观察日志: No qualifying bean of type 'com.bite.autoconfig.BiteConfig' available

没有 `com.bite.autoconfig.BiteConfig` 这个类型的Bean

3.1.2 原因分析

Spring通过五大注解和 `@Bean` 注解可以帮助我们把Bean加载到SpringIoC容器中, 以上有个前提就是这些**注解类需要和SpringBoot启动类在同一个目录下** (`@SpringBootApplication` 标注的类 就是SpringBoot项目的启动类).

> 启动类所在目录为: `com.example.demo` , 而 `BiteConfig` 这个类在 `com.bite.autoconfig` 下, 所以SpringBoot并没有扫描到.

当我们引入第三方的Jar包时, 第三方的Jar代码目录肯定不在启动类的目录下, 如何告诉Spring帮我们管理这些Bean呢?

3.1.3 解决方案

我们需要指定路径或者引入的文件, 告诉Spring, 让Spring进行扫描到.

常见的解决方案有两种:

1. `@ComponentScan` 组件扫描

2. `@Import` 导入(使用@Import导入的类会被Spring加载到IoC容器中)

我们通过代码来看如何解决

3.1.3.1 `@ComponentScan`

通过 `@ComponentScan` 注解, 指定Spring扫描路径

```
1  import org.springframework.boot.SpringApplication;
2  import org.springframework.boot.autoconfigure.SpringBootApplication;
3  import org.springframework.context.annotation.ComponentScan;
4
5  @ComponentScan("com.bite.autoconfig")
6  @SpringBootApplication
7  public class SpringAutoconfigApplication {
8
9      public static void main(String[] args) {
10         SpringApplication.run(SpringAutoconfigApplication.class, args);
11     }
12
13 }
```

> 可以指定扫描多个包
>
> @ComponentScan({"com.bite.autoconfig","com.example.demo"})

运行程序:

可以看到, 这次 `biteConfig` Bean获取到了

> Spring 是否使用了这种方式呢?
>
> 非常明显, 没有.(因为我们引入第三方框架时, 没有加扫描路径. 比如mybatis)
>
> 如果Spring Boot采用这种方式, 当我们引入大量的第三方依赖, 比如Mybatis, jackson等时, 就需要在启动类上配置不同
>
> 依赖需要扫描的包, 这种方式会非常繁琐.

3.1.3.2 `@Import`

`@Import` 导入主要有以下几种形式:

1. 导入类
2. 导入 `ImportSelector` 接口实现类

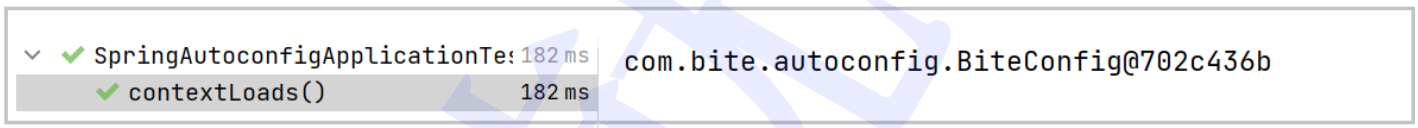
1. 导入类

```
@Import(BiteConfig.class)
@SpringBootApplication
public class SpringAutoconfigApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringAutoconfigApplication.class, args);
    }

}
```

运行程序:



可以看到, 这种方式也可以告诉Spring加载 `biteConfig`

问题: 如果又多了一些配置项呢?

```
@Component
public class BiteConfig2 {
    public void study2(){
        System.out.println("start study2...");
    }
}
```

我们可以采用导入多个类

```
@Import({BiteConfig.class, BiteConfig2.class})
@SpringBootApplication
public class SpringAutoconfigApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringAutoconfigApplication.class, args);
```

```
    }

}
```

很明显, 这种方式也很繁琐.

所以, SpringBoot依然没有采用.

2. 导入ImportSelector接口实现类

`ImportSelector` 接口实现类

```
public class MyImportSelector implements ImportSelector {
    @Override
    public String[] selectImports(AnnotationMetadata importingClassMetadata) {
        //需要导入的全限定类名
        return new String[]
{"com.bite.autoconfig.BiteConfig","com.bite.autoconfig.BiteConfig2"};
    }
}
```

启动类:

```
@Import(MyImportSelector.class)
@SpringBootApplication
public class SpringAutoconfigApplication {

   public static void main(String[] args) {
      SpringApplication.run(SpringAutoconfigApplication.class, args);
   }

}
```

运行程序:

```
SpringAutoconfigApplicationTe: 179 ms
    contextLoads()              179 ms
com.bite.autoconfig.BiteConfig@23f3dbf0
com.bite.autoconfig.BiteConfig2@31d6f3fe
```

可以看到, 我们采用这种方式也可以导入第三方依赖提供的Bean.

问题:

但是他们都有一个明显的问题, 就是使用者需要知道第三方依赖中有哪些Bean对象或配置类. 如果漏掉其中一些Bean, 很可能导致我们的项目出现大的事故.

这对程序员来说非常不友好.

依赖中有哪些Bean, 使用时需要配置哪些bean, 第三方依赖最清楚, 那能否由第三方依赖来做这件事呢?

- 比较常见的方案就是第三方依赖给我们提供一个注解, 这个注解一般都以@EnableXxxx开头的注解, 注解中封装的就是 `@Import` 注解

1. 第三方依赖提供注解

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

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Import(MyImportSelector.class)//指定要导入哪些类
public @interface EnableBiteConfig {
}
```

注解中封装 `@Import` 注解, 导入 `MyImportSelector.class`

2. 在启动类上使用第三方提供的注解

```
@EnableBiteConfig
@SpringBootApplication
public class SpringAutoconfigApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringAutoconfigApplication.class, args);
    }

}
```

3. 运行程序

可以看到, 这种方式也可以导入第三方依赖提供的Bean.

并且这种方式更优雅一点. SpringBoot采用的也是这种方式

3.2 SpringBoot原理分析

3.2.1 源码阅读

SpringBoot 是如何帮助我们做的呢? 一切的来自起源SpringBoot的启动类开始

> `@SpringBootApplication` 标注的类 就是SpringBoot项目的启动类

```
1  @SpringBootApplication
2  public class SpringIocApplication {
3
4      public static void main(String[] args) {
5          //获取Spring上下文对象
6          ApplicationContext context =
   SpringApplication.run(SpringIocApplication.class, args);
7          //从Spring上下文中获取对象
8          BeanLifeComponent beanLifeComponent =
   context.getBean(BeanLifeComponent.class);
9          beanLifeComponent.use();
10     }
11 }
```

这个类和普通类唯一的区别就是 `@SpringBootApplication` 注解, 这个注解也是SpringBoot实现自动配置的核心.

```
1  @Target({ElementType.TYPE})
2  @Retention(RetentionPolicy.RUNTIME)
3  @Documented
4  @Inherited
5  @SpringBootConfiguration
6  @EnableAutoConfiguration
7  @ComponentScan(
8      excludeFilters = {@Filter(
9      type = FilterType.CUSTOM,
10     classes = {TypeExcludeFilter.class}
11 ), @Filter(
```

```
    type = FilterType.CUSTOM,
    classes = {AutoConfigurationExcludeFilter.class}
)}
)
public @interface SpringBootApplication {
    //...代码省略
}
```

`@SpringBootApplication` 是一个组合注解, 注解中包含了:

1. 元注解

   > JDK中提供了4个标准的用来对注解类型进行注解的注解类, 我们称之为 meta-annotation(元注解), 他们分别是:
   >
   > - @Target 描述注解的使用范围(即被修饰的注解可以用在什么地方)
   > - @Retention 描述注解保留的时间范围
   > - @Documented 描述在使用 javadoc 工具为类生成帮助文档时是否要保留其注解信息
   > - @Inherited 使被它修饰的注解具有继承性(如果某个类使用了被@Inherited修饰的注解，则其子类将自动具有该注解)

2. `@SpringBootConfiguration`

   里面就是@Configuration, 标注当前类为配置类, 其实只是做了一层封装改了个名字而已.
   (@Indexed注解，是用来加速应用启动的, 不用关心)

   ```
   @Target({ElementType.TYPE})
   @Retention(RetentionPolicy.RUNTIME)
   @Documented
   @Configuration
   @Indexed
   public @interface SpringBootConfiguration {
       @AliasFor(
           annotation = Configuration.class
       )
       boolean proxyBeanMethods() default true;
   }
   ```

3. `@EnableAutoConfiguration` (开启自动配置)

   Spring自动配置的核心注解, 下面详细讲解

4. `@ComponentScan` (包扫描)

可以通过 `basePackageClasses` 或 `basePackages` 来定义要扫描的特定包，如果没有定义特定的包，将从声明该注解的类的包开始扫描，这也是为什么SpringBoot项目声明的注解类必须要在启动类的目录下.

> excludeFilters 自定义过滤器，通常用于排除一些类，注解等.

3.2.2 `@EnableAutoConfiguration` 详解

看下 `@EnableAutoConfiguration` 注解的实现:

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import({AutoConfigurationImportSelector.class})
public @interface EnableAutoConfiguration {
    String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautoconfiguration";

    Class<?>[] exclude() default {};

    String[] excludeName() default {};
}
```

这个注解包含两部分:

1. **@Import({AutoConfigurationImportSelector.class})**

使用@Import注解，导入了实现ImportSelector接口的实现类.

```
public class AutoConfigurationImportSelector implements
DeferredImportSelector, BeanClassLoaderAware, ResourceLoaderAware,
BeanFactoryAware, EnvironmentAware, Ordered {
    public String[] selectImports(AnnotationMetadata annotationMetadata) {
    if (!this.isEnabled(annotationMetadata)) {
        return NO_IMPORTS;
    } else {
        //获取自动配置的配置类信息
        AutoConfigurationEntry autoConfigurationEntry =
this.getAutoConfigurationEntry(annotationMetadata);
        return
StringUtils.toStringArray(autoConfigurationEntry.getConfigurations());
    }
}

```

```
}

```

`selectImports()` 方法底层调用 `getAutoConfigurationEntry()` 方法, 获取可自动配置的配置类信息集合.

点进去:

```
protected AutoConfigurationEntry getAutoConfigurationEntry(AnnotationMetadata annotationMetadata) {
    if (!this.isEnabled(annotationMetadata)) {
        return EMPTY_ENTRY;
    } else {
        AnnotationAttributes attributes = this.getAttributes(annotationMetadata);
        //获取在配置文件中配置的所有自动配置类的集合
        List<String> configurations = this.getCandidateConfigurations(annotationMetadata, attributes);
        configurations = this.removeDuplicates(configurations);
        Set<String> exclusions = this.getExclusions(annotationMetadata, attributes);
        this.checkExcludedClasses(configurations, exclusions);
        configurations.removeAll(exclusions);
        configurations = this.getConfigurationClassFilter().filter(configurations);
        this.fireAutoConfigurationImportEvents(configurations, exclusions);
        return new AutoConfigurationEntry(configurations, exclusions);
    }
}
```

`getAutoConfigurationEntry()` 方法通过调用 `getCandidateConfigurations(annotationMetadata, attributes)` 方法获取在配置文件中配置的所有自动配置类的集合

点进去:

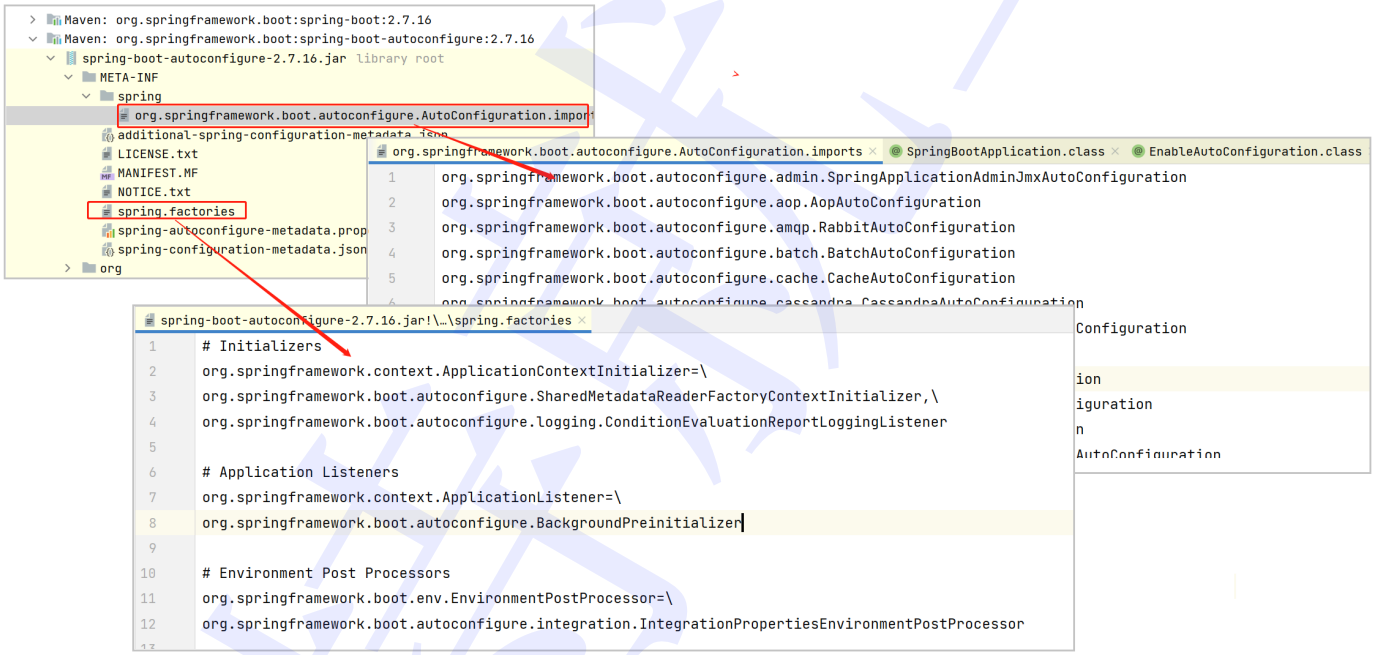
```
//获取所有基于
//META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports文件
//META-INF/spring.factories文件中配置类的集合
protected List<String> getCandidateConfigurations(AnnotationMetadata metadata, AnnotationAttributes attributes) {
    List<String> configurations = new ArrayList(SpringFactoriesLoader.loadFactoryNames(this.getSpringFactoriesLoaderF
```

```
actoryClass(), this.getBeanClassLoader()));
    ImportCandidates.load(AutoConfiguration.class,
this.getBeanClassLoader()).forEach(configurations::add);
    Assert.notEmpty(configurations, "No auto configuration classes found in
META-INF/spring.factories nor in META-
INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports.
If you are using a custom packaging, make sure that file is correct.");
    return configurations;
}
```

`getCandidateConfigurations` 方法的功能:

获取所有基于 `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports` 文件, `META-INF/spring.factories` 文件中配置类的集合.

在引入的起步依赖中, 通常都有包含以上两个文件



这里面包含了很多第三方依赖的配置文件(连续按两下shift可以查看对应的源码)

1. 在加载自动配置类的时候, 并不是将所有的配置全部加载进来, 而是通过@Conditional等注解的判断进行动态加载.

@Conditional是spring底层注解, 意思就是根据不同的条件, 来进行自己不同的条件判断，如果满足指定的条件，那么配置类里边的配置才会生效.

2. META-INF/spring.factories文件是Spring内部提供的一个约定俗成的加载方式, 只需要在模块的META-INF/spring.factories文件中配置即可，Spring就会把相应的实现类注入到Spring容器中.

   注: 会加载所有jar包下的classpath路径下的META-INF/spring.factories文件, 这样文件不止一个.

比如 Redis的配置: `RedisAutoConfiguration`

可以看到, 配置文件中使用 `@Bean` 声明了一些对象, spring就会自动调用配置类中使用 `@Bean` 标识的方法, 并把对象注册到Spring IoC容器中.

> 在加载自动配置类的时候, 并不是将所有的配置全部加载进来, 而是通过@Conditional等注解的判断进行动态加载
>
> @Conditional是spring底层注解, 意思就是根据不同的条件, 来进行自己不同的条件判断，如果满足指定的条件，那么配置类里边的配置才会生效。

2. @AutoConfigurationPackage

源码如下:

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Import({AutoConfigurationPackages.Registrar.class})
public @interface AutoConfigurationPackage {
    String[] basePackages() default {};

    Class<?>[] basePackageClasses() default {};
}
```

这个注解主要是导入一个配置文件 `AutoConfigurationPackages.Registrar.class`

```
static class Registrar implements ImportBeanDefinitionRegistrar,
```

```
    DeterminableImports {
2       Registrar() {
3       }
4
5       public void registerBeanDefinitions(AnnotationMetadata metadata,
    BeanDefinitionRegistry registry) {
6           AutoConfigurationPackages.register(registry, (String[])(new
    PackageImports(metadata)).getPackageNames().toArray(new String[0]));
7       }
8
9       public Set<Object> determineImports(AnnotationMetadata metadata) {
10          return Collections.singleton(new PackageImports(metadata));
11      }
12  }
```

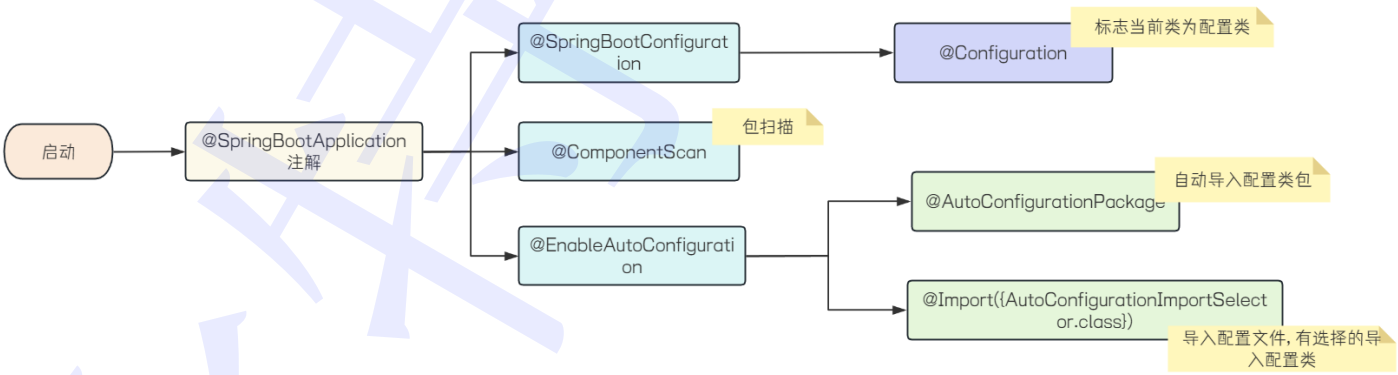
> Registrar实现了 `ImportBeanDefinitionRegistrar` 类, 就可以被注解@Import导入到spring容器里.

`(String[])(new PackageImports(metadata)).getPackageNames().toArray(new String[0])` : 当前启动类所在的包名.

所以: `@AutoConfigurationPackage` 就是将启动类所在的包下面所有的组件都扫描注册到spring 容器中.

3.2.3 总结

SpringBoot 自动配置原理的大概流程如下:



当SpringBoot程序启动时, 会加载配置文件当中所定义的配置类, 通过 `@Import` 注解将这些配置类全部加载到Spring的IOC容器中, 交给IOC容器管理.

4. 总结

1. Bean的作用域共分为6种: singleton, prototype, request, session, application和websocket.
2. Bean的生命周期共分为5大部分: 实例化, 属性复制, 初始化, 使用和销毁

3. SpringBoot的自动配置原理源码口是 `@SpringBootApplication` 注解, 这个注解封装了3个注解
    - **`@SpringBootConfiguration`** 标志当前类为配置类
    - **`@ComponentScan`** 进行包扫描(默认扫描的是启动类所在的当前包及其子包)
    - **`@EnableAutoConfiguration`**
        - **`@Import` 注解**: 读取当前项目下所有依赖jar包中 `META-INF/spring.factories` , `META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports` 两个文件里面定义的配置类(配置类中定义了 `@Bean` 注解标识的方法)
        - **`@AutoConfigurationPackage`** : 把启动类所在的包下面所有的组件都注入到 Spring 容器中