走向自动装配

1. 手动装配 Spring Framework
2. 手动装配 - 模式注解装配
   1. 注解的 “派生性”

被@ Component直接注解的类

|  |
| --- |
| @Retention(RetentionPolicy.RUNTIME)  @Target(ElementType.TYPE)  @Documented  @Component  public @interface StudyOneAnnotation {  String value() default "";  } |

* 1. 注解的 “层次性”

间接被@ Component注解的类

|  |
| --- |
| @Retention(RetentionPolicy.RUNTIME)  @Target(ElementType.TYPE)  @Documented  @StudyOneAnnotation  public @interface StudyTwoAnnotation {  String value() default "";  } |

测试：

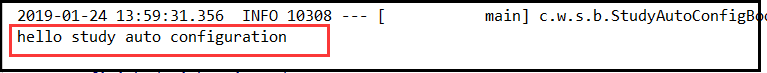
和使用@Component一样，注解到Class上面。然后进行被扫描就ok。

@StudyTwoAnnotation也可以换成@StudyOneAnnotation一样的效果

|  |
| --- |
| @StudyTwoAnnotation  public class StudyAutoConfig {  public String study(){  return "hello study auto configuration";  }  } |

结果：

|  |
| --- |
| @ComponentScan("com.wangkun.springBootStudy.configration")  public class StudyAutoConfigBootstrap {  public static void main(String[] args) {  ConfigurableApplicationContext context =  new SpringApplicationBuilder(StudyAutoConfigBootstrap.class)  .web(WebApplicationType.NONE)  .run(args);  StudyAutoConfig study = context.getBean("studyAutoConfig", StudyAutoConfig.class);  System.out.println(study.study());  //关闭  context.close();  }  } |



* 1. 其实我们写的 ”派生性” 在我们Spring 的@Configuration中得到充分的体现

@configuration源码

|  |
| --- |
| @Target(ElementType.TYPE)  @Retention(RetentionPolicy.RUNTIME)  @Documented  @Component  public @interface Configuration {    @AliasFor(annotation = Component.class)  String value() default "";  } |

给自己留个任务，为什么我们写的@Configuration中的@Bean根本不用我们扫描，怎么改变以前的配置的？

1. 手动装配 - @Enable模块装配（将功能组合到一个注解上进行独立单元）
   1. 注解方式（在我们的驱动Class上面加上@EnableXxx注解则可以导入想要的Bean）

EnableXXX 注解

|  |
| --- |
| @Retention(RetentionPolicy.RUNTIME)  @Target(ElementType.TYPE)  @Documented  @Import(StudyAutoConfig.class)  public @interface EnableStudyAutoConfig {  } |

EnableXXXConfiguration 类

|  |
| --- |
| @Configuration  public class StudyAutoConfig {  @Bean  public String study(){  return "hello study auto configuration";  }  } |

驱动类（@EnableStudyAutoConfig注解到Bootstrap上）

|  |
| --- |
| @EnableStudyAutoConfig  public class StudyAutoConfigBootstrap {  public static void main(String[] args) {  ConfigurableApplicationContext context =  new SpringApplicationBuilder(StudyAutoConfigBootstrap.class)  .web(WebApplicationType.NONE)  .run(args);  String study = context.getBean("study", String.class);  System.out.println(study);  //关闭  context.close();  }  } |

* 1. 编程方式（富有弹性的导入，按照选择导入进想要的配置）

EnableXXX

|  |
| --- |
| @Retention(RetentionPolicy.RUNTIME)  @Target(ElementType.TYPE)  @Documented  @Import(StudyAutoImportSelector.class)  public @interface EnableSelectorStudyAutoAnnotation {  } |

EnableImportSelector

实现ImportSelector接口完成方法的重写，返回类型是String[]，也就是你的配置类的返回集合。这里写出了，两个配置类。并且该方法可以根据自己的条件判断什么条件下返回什么配置类。

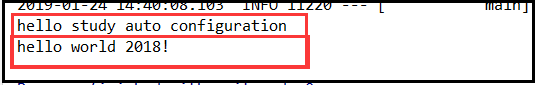
|  |
| --- |
| public class StudyAutoImportSelector implements ImportSelector {  @Override  public String[] selectImports(AnnotationMetadata importingClassMetadata) {  return new String[]{StudyAutoConfig.class.getName() ,  HelloWorldConfiguration.class.getName()};  }  } |

EnableConfiguration

|  |
| --- |
| @Configuration  public class StudyAutoConfig {  @Bean  public String study(){  return "hello study auto configuration";  }  }  @Configuration  public class HelloWorldConfiguration {  @Bean  public String hello(){  return "hello world 2018!";  }  } |

驱动类

|  |
| --- |
| @EnableSelectorStudyAutoAnnotation  public class StudyAutoConfigBootstrap {  public static void main(String[] args) {  ConfigurableApplicationContext context =  new SpringApplicationBuilder(StudyAutoConfigBootstrap.class)  .web(WebApplicationType.NONE)  .run(args);  String study = context.getBean("study", String.class);  System.out.println(study);  String hello = context.getBean("hello",String.class);  System.out.println(hello);  //关闭  context.close();  }  } |



1. 手动装配 – Spring条件装配
   1. Profile

Java7JiSuan.class 针对Java8采用的计算总和的服务

|  |
| --- |
| @Profile("Java7")  @Service("jiSuan")  public class Java7JiSuan implements JiSuan{  @Override  public int add(Integer... figures) {  System.out.println("使用for循环进行数字的累加!");  int sum = 0;  for(Integer figure : figures){  sum += figure;  }  return sum;  }  } |

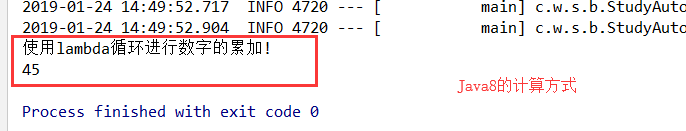
Java8JiSuan.class 针对Java8采用的计算总和的服务

|  |
| --- |
| @Profile("Java8")  @Service("jiSuan")  public class Java8JiSuan implements JiSuan{  @Override  public int add(Integer... figures) {  System.out.println("使用lambda循环进行数字的累加!");  int sum = Stream.of(figures).reduce(0,Integer::sum);  return sum;  }  } |

驱动类

启动程序的时候选择Profile的测试版本

|  |
| --- |
| @ComponentScan("com.wangkun.springBootStudy.service")  public class StudyAutoConfigBootstrap {  public static void main(String[] args) {  ConfigurableApplicationContext context =  new SpringApplicationBuilder(StudyAutoConfigBootstrap.class)  .web(WebApplicationType.NONE)  .profiles("Java8")  .run(args);  JiSuan jiSuan = context.getBean("jiSuan", JiSuan.class);  int add = jiSuan.add(1, 2, 3, 4, 5, 6, 7, 8, 9);  System.out.println(add);  //关闭  context.close();  }  } |



* 1. Conditional

XXXConditionAnnotation

我们这里想演示在方法上面创建一个Bean，所以使用的是方法级别的Target

|  |
| --- |
| @Retention(RetentionPolicy.RUNTIME)  @Target(ElementType.METHOD)  @Documented  @Conditional(StudyAutoConfigConditional.class)  public @interface StudyAutoConfigConditionAnnotation {  String name() default "";  String value() default "";  } |

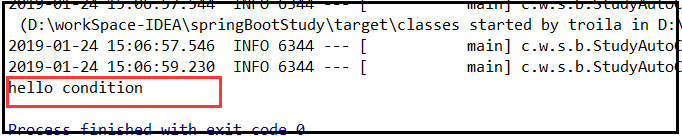
XXXConditional

实现Condition中的matches方法，可以根据自己的判断条件进行变更。我们这里使用的是取出注解上面的参数和我们系统中的数据进行比对。True则创建Bean，否则不创建Bean。

|  |
| --- |
| public class StudyAutoConfigConditional implements Condition {  @Override  public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {  Map<String, Object> attrs = metadata.getAnnotationAttributes(StudyAutoConfigConditionAnnotation.class.getName());  String name = String.valueOf(attrs.get("name"));  String value = String.valueOf(attrs.get("value"));  String property = System.getProperty(name);  return value.equals(property);  }  } |

驱动类

|  |
| --- |
| public class StudyAutoConfigBootstrap {  @Bean  @StudyAutoConfigConditionAnnotation(name = "user.name",value = "troila")  private String hello(){  return "hello condition";  }  public static void main(String[] args) {  ConfigurableApplicationContext context =  new SpringApplicationBuilder(StudyAutoConfigBootstrap.class)  .web(WebApplicationType.NONE)  .profiles("Java8")  .run(args);  String hello = context.getBean("hello", String.class);  System.out.println(hello);  //关闭  context.close();  }  } |



###### 自动装配

* 1. 必要条件
     + 1. EnableAutoXXX类
       2. Spring.factories
       3. 驱动类
  2. 实现

EnableAutoXXX注解

自动配置将我们上面学到的手动配置全部使用到了。

|  |
| --- |
| @Configuration //模式注解驱动  @EnableStudyAutoConfig //模块注解驱动  @StudyAutoConfigConditionAnnotation(name = "user.name",value = "troila") //条件驱动  public class EnableAutoConfigStudyAutoConfigConditional {  } |

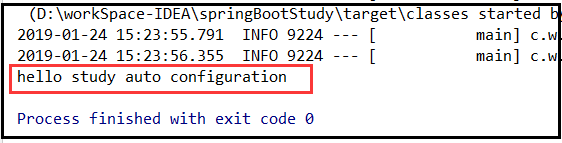
Spring.factories (放到我们resources -> META-INF 下)

对应我们的自动配置的Class，是我们自动配置的关键

|  |
| --- |
| org.springframework.boot.autoconfigure.EnableAutoConfiguration=\  com.wangkun.springBootStudy.configration.HelloWorldEnableAutoConfiguration,\  com.wangkun.springBootStudy.conditional.EnableAutoConfigStudyAutoConfigConditional |

驱动类:

|  |
| --- |
| @EnableAutoConfiguration  public class StudyAutoConfigBootstrap {  public static void main(String[] args) {  ConfigurableApplicationContext context =  new SpringApplicationBuilder(StudyAutoConfigBootstrap.class)  .web(WebApplicationType.NONE)  .run(args);  String hello = context.getBean("study", String.class);  System.out.println(hello);  //关闭  context.close();  }  } |



* 1. 启动流程

@EnableAutoConfiguration –> spring.factories –> conditional –> importselector –> configuration –> hello的Bean

驱动类被@EnableAutoConfiguration注解，spring去spring.factories中寻找自动配置的类，找到之后查看我们是否符合conditional的条件，如果符合spring去选择符合条件的configuration类，根据我们的需要生成Bean。

总结：

通过了解spring boot的自动化配置，让我对springboot又多了一点的了解，不再迷茫为什么驱动类上要有自动配置注解，不再迷茫为什么@EnableXXX注解配置之后，我们有了那么多的功能。