Spring核心源码解析

IOC容器概述

ApplicationContext接口相当于负责bean的初始化、配置和组装的IoC容器.
Spring为ApplicationContext提供了一些开箱即用的实现, 独立的应用可以使用ClassPathXmlApplicationContext或者FileSystemXmlApplicationContext，web应用在web.xml配置监听，提供xml位置和org.springframework.web.context.ContextLoaderListener即可初始化WebApplicationContextIoC容器.

配置元数据

配置元数据配置了应用中实例的实例化、配置以及组装的规则，SpringIoC容器通过此配置进行管理Bean. 配置元数据有以下几种方式:

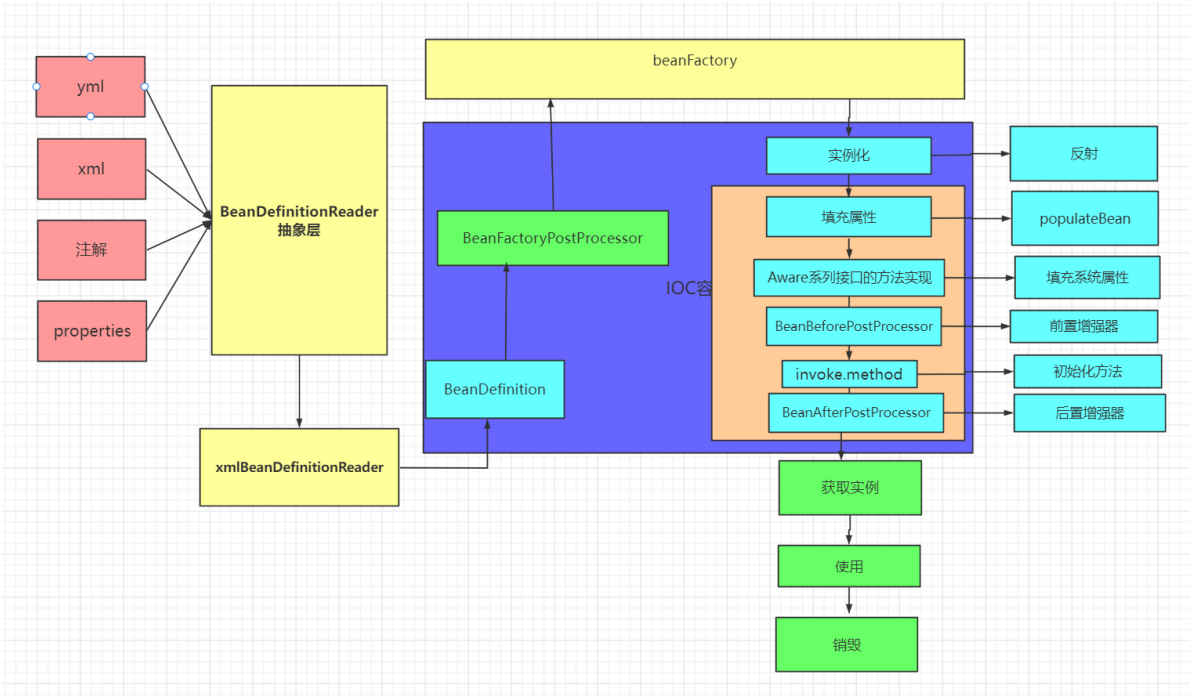
- 基于XML配置： 清晰明了，简单易用
- 基于Java代码配置：无xml,通过`@Configuration`来声明配置、对象实例化与依赖关系
- 基于Java注解配置：少量的XML(`<context:annotation-config/>`)，通过注解声明实例化类与依赖关系

后续的分析基于XML配置, 与Java代码和注解大体上的机制是一样

实例化容器

实例化容器非常简单，只需要提供本地配置路径或者根据`ApplicationContext`的构造器提供相应的资源(Spring的另一个重要抽象)即可.

```
ApplicationContext context = new
ClassPathXmlApplicationContext("application.xml");
```



refresh()方法的实现代码如下

```
public void refresh() throws BeansException, IllegalStateException {
    synchronized(this.startupShutdownMonitor) {
        this.prepareRefresh(); // 准备工作
        ConfigurableListableBeanFactory beanFactory =
this.obtainFreshBeanFactory(); // 获取ConfigurableListableBeanFactory最终的目的是
DefaultListableBeanFactory
        this.prepareBeanFactory(beanFactory); // 准备bean工厂

        try {
            this.postProcessBeanFactory(beanFactory); // 一个空的实现，注意这里的
spring版本号为：5.3x
            this.invokeBeanFactoryPostProcessors(beanFactory); // 注册bean的工厂
            this.registerBeanPostProcessors(beanFactory);
            this.initMessageSource(); // Spring 从所有的 @Bean 定义中抽取出来了
BeanPostProcessor，然后都注册进 beanPostProcessors，等待后面的的顺序调用 注册
BeanPostProcessor
            this.initApplicationEventMulticaster(); // 初始化事件监听多路广播器
            this.onRefresh(); // 一个空的实现
            this.registerListeners(); // 注册监听器
            this.finishBeanFactoryInitialization(beanFactory); // 到了spring加载流
程最复杂的一步，开始实例化所有的bd
            this.finishRefresh();// 刷新完成工作
        } catch (BeansException var9) {
            if (this.logger.isWarnEnabled()) {
                this.logger.warn("Exception encountered during context
initialization - cancelling refresh attempt: " + var9);
            }

            this.destroyBeans();
            this.cancelRefresh(var9);
            throw var9;
        } finally {
            this.resetCommonCaches();
        }

    }
}
```

loadBeanDefinitions:

```
ConfigurableListableBeanFactory beanFactory = this.obtainFreshBeanFactory();

this.refreshBeanFactory();

this.loadBeanDefinitions(beanFactory);

protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory)
throws BeansException, IOException {
    XmlBeanDefinitionReader beanDefinitionReader = new
XmlBeanDefinitionReader(beanFactory);
    beanDefinitionReader.setEnvironment(this.getEnvironment());
    beanDefinitionReader.setResourceLoader(this);
    beanDefinitionReader.setEntityResolver(new ResourceEntityResolver(this));
    this.initBeanDefinitionReader(beanDefinitionReader);
```

```
    this.loadBeanDefinitions(beanDefinitionReader);
}
```

SpringBoot核心源码解析

为什么会出现SpringBoot?

市场需要，首先通过对比我们来直观的感受下

1.传统的构建方式

我们沟通SpringMVC+SpringFramework来构建一个web项目，过程如下

- 创建一个maven-webapp项目
- 添加jar包依赖

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-beans</artifactId>
    <version>5.2.5.RELEASE</version>
</dependency>
<dependency>
    <groupId>commons-logging</groupId>
    <artifactId>commons-logging</artifactId>
    <version>1.2</version>
</dependency>
```

```
spring-context
spring-context-support
spring-core
spring-expression
spring-web
spring-webmvc
```

- 修改web.xml文件

```
<context-param><!--配置上下文配置路径-->
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:applicationContext.xml</param-value>
</context-param>
<!--配置监听器-->
<listener>
    <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
<listener>
    <listener-
class>org.springframework.web.util.IntrospectorCleanupListener</listener-
class>
</listener>
<!--配置Spring MVC的请求拦截-->
<servlet>
```

```
    <servlet-name>springmvc</servlet-name>
    <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:dispatcher-servlet.xml</param-value>
    </init-param>
</servlet>
<servlet-mapping>
    <servlet-name>springmvc</servlet-name>
    <url-pattern>/</url-patter>
</servlet-mapping>
```

- 在resources目录下添加dispatcher-servlet.xml文件

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd
       http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc.xsd
       http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.xsd">

<!-- 扫描 controller -->
<context:component-scan base-package="com.mashibingedu.controller" />
<!--开启注解驱动-->
<mvc:annotation-driven/>
<!-- 定义视图解析器 -->
<bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/"/>
    <property name="suffix" value=".jsp"/>
</bean>
```

- 创建一个Controller

```
@Controller
public class HelloController {

    @RequestMapping(method = RequestMethod.GET,path = "/index")
    public String index(Model model){
        model.addAttribute("key","Hello mashibing");
        return "index";
    }
}
```

- 修改默认的index.jsp，设置el表达式的解析

```
<%@ page language="java" contentType="text/html; charset=utf-8"
         pageEncoding="utf-8" isELIgnored="false" %>
${key}
```

- 运行项目

2.SpringBoot构建方式

大家能够看到这种构建方式虽然很清晰，但是花费的时间较多，配置信息也比较多。在单体架构的时候还能胜任，但是现如今今天都是分布式系统，一个系统少着十几个多着百多个子系统，如果还用这种方式来构架显然效率会很低，更快的构建项目的方式就显得很迫切了，这时候SpringBoot就应运而生了。先来看看SpringBoot构建一个web项目的效果

.....此处直接演示

通过构建过程大家应该能够非常直观的感受到SpringBoot带给我们的效果。

3.理解SpringBoot

Spring Boot被官方定位为“BUILD ANYTHING”，Spring Boot官方的概述是这么描述Spring Boot的。

```
Spring Boot makes it easy to create stand-alone, production-grade Spring based
Applications that you can "just run".
// 通过Spring Boot可以轻松的创建独立的、生产级别的基于Spring 生态下的应用，你只需要运行即可。
We take an opinionated view of the Spring platform and third-party libraries so
you can get started with minimum fuss. Most Spring Boot applications need
minimal Spring configuration.
//对于Spring平台和第三方库，我们提供了一个固化的视图，这个视图可以让我们在构建应用时减少很多麻
烦。大部分spring boot应用只需要最小的Spring 配置即可。
```

如果大家不习惯看英文文档，可能理解起来比较复杂，翻译成人话就是：Spring Boot能够帮助使用Spring Framework【IOC，AOP】生态的开发者快速高效的构建一个基于Spring以及spring 生态体系的应用。

了解了概念性的内容后我们再回到SpringBoot项目上来。

4.理解约定优于配置

我们知道，Spring Boot是约定由于配置理念下的产物，那么什么是约定由于配置呢?

约定优于配置是一种软件设计的范式，主要是为了减少软件开发人员需做决定的数量，获得简单的好处，而又不失灵活性。

简单来说，就是你所使用的工具默认会提供一种约定，如果这个约定和你的期待相符合，就可以省略那些基础的配置，否则，你就需要通过相关配置来达到你所期待的方式。

约定优于配置有很多地方体现，举个例子，比如交通信号灯，红灯停、绿灯行，这个是一个交通规范。你可以在红灯的时候不停，因为此时没有一个障碍物阻碍你。但是如果大家都按照这个约定来执行，那么不管是交通的顺畅度还是安全性都比较好。

而相对于技术层面来说，约定有很多地方体现，比如一个公司，会有专门的文档格式、代码提交规范、接口命名规范、数据库规范等等。这些规定的意义都是让整个项目的可读性和可维护性更强。

5.Spring Boot Web应用中约定优于配置的体现

那么在前面的案例中，我们可以思考一下，Spring Boot为什么能够把原本繁琐又麻烦的工作省略掉呢？实际上这些工作并不是真正意义上省略了，只是Spring Boot帮我们默认实现了。

而这个时候我们反过来思考一下，Spring Boot Web应用中，相对Spring MVC框架的构建而言，它的约定优于配置体现在哪些方面呢？

- Spring Boot的项目结构约定，Spring Boot默认采用Maven的目录结构，其中

  src.main.java 存放源代码文件

  src.main.resource 存放资源文件

  src.test.java 测试代码

  src.test.resource 测试资源文件

  target 编译后的class文件和jar文件

- 内置了嵌入式的Web容器，在Spring 2.2.6版本的官方文档中3.9章节中，有说明Spring Boot支持四种嵌入式的Web容器

  Tomcat

  Jetty

  Undertow

  Reactor

- Spring Boot默认提供了两种配置文件，一种是application.properties、另一种是application.yml。Spring Boot默认会从该配置文件中去解析配置进行加载。
- Spring Boot通过starter依赖，来减少第三方jar的依赖。

这些就是Spring Boot能够方便快捷的构建一个Web应用的秘密。当然Spring Boot的约定优于配置还不仅体现在这些地方，在后续的分析中还会看到Spring Boot中约定优于配置的体现。

6.Import注解

import注解是什么意思呢？ 联想到xml形式下有一个`<import resource/>`形式的注解，就明白它的作用了。import就是把多个分来的容器配置合并在一个配置中。在JavaConfig中所表达的意义是一样的。

- 创建一个包，并在里面添加一个单独的configuration

```
public class DefaultBean {
}
@Configuration
public class SpringConfig {

    @Bean
    public DefaultBean defaultBean(){
        return new DefaultBean();
    }
}
```

- 此时运行测试方法，

```
public class MainDemo {

    public static void main(String[] args) {
        ApplicationContext ac=new
AnnotationConfigApplicationContext(SpringConfig.class);
        String[] defNames=ac.getBeanDefinitionNames();
        for(String name:defNames){
            System.out.println(name);
        }
    }
}
```

- 在另外一个包路径下在创建一个配置类。此时再次运行前面的测试方法，打印OtherBean实例时，这个时候会报错，提示没有该实例

```
public class OtherBean {
}
@Configuration
public class OtherConfig {

    @Bean
    public OtherBean otherBean(){
        return new OtherBean();
    }
}
```

- 修改springConfig，把另外一个配置导入过来

```
@Import(OtherConfig.class)
@Configuration
public class SpringConfig {

    @Bean
    public DefaultBean defaultBean(){
        return new DefaultBean();
    }
}
```

- 再次运行测试方法，即可看到对象实例的输出。

至此，我们已经了解了Spring Framework在注解驱动时代，完全替代XML的解决方案。至此，Spring团队就此止步了吗？你们太单纯了。虽然无配置化能够减少配置的维护带来的困扰，但是，还是会存在很对第三方组建的基础配置声明。同样很繁琐，所以Spring 推出了@Enable模块驱动。这个特性的作用是把相同职责的功能组件以模块化的方式来装配，更进一步简化了Spring Bean的配置。

深入分析EnableAutoConfiguration

EnableAutoConfiguration的主要作用其实就是帮助springboot应用把所有符合条件的@Configuration配置都加载到当前SpringBoot创建并使用的IoC容器中。

再回到EnableAutoConfiguration这个注解中，我们发现它的import是这样

```
@Import(AutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {
```

但是从EnableAutoCOnfiguration上面的import注解来看，这里面并不是引入另外一个Configuration。而是一个ImportSelector。这个是什么东西呢?

AutoConfigurationImportSelector是什么?

Enable注解不仅仅可以像前面演示的案例一样很简单的实现多个Configuration的整合，还可以实现一些复杂的场景，比如可以根据上下文来激活不同类型的bean，@Import注解可以配置三种不同的class

1. 第一种就是前面演示过的，基于普通bean或者带有@Configuration的bean进行诸如
2. 实现ImportSelector接口进行动态注入

实现ImportBeanDefinitionRegistrar接口进行动态注入

CacheService

```
public class CacheService {
}
```

LoggerService

```
public class LoggerService {
}
```

EnableDefineService

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited  --允许被继承
@Import({MyDefineImportSelector.class})
public @interface EnableDefineService {

    String[] packages() default "";
}
```

MyDefineImportSelector

```
public class MyDefineImportSelector implements ImportSelector {
    @Override
    public String[] selectImports(AnnotationMetadata annotationMetadata) {
        //获得指定注解的详细信息。我们可以根据注解中配置的属性来返回不同的class，
        //从而可以达到动态开启不同功能的目的

annotationMetadata.getAllAnnotationAttributes(EnableDefineService.class.getName(
),true)
            .forEach((k,v) -> {
                log.info(annotationMetadata.getClassName());
                log.info("k:{},v:{}",k,String.valueOf(v));
            });
        return new String[]{CacheService.class.getName()};
    }
}
```

EnableDemoTest

```
@SpringBootApplication
@EnableDefineService(name = "mashibing",value = "mashibing")
public class EnableDemoTest {
    public static void main(String[] args) {
        ConfigurableApplicationContext
ca=SpringApplication.run(EnableDemoTest.class,args);
        System.out.println(ca.getBean(CacheService.class));
        System.out.println(ca.getBean(LoggerService.class));
    }
}
```

了解了selector的基本原理之后，后续再去分析AutoConfigurationImportSelector的原理就很简单了，它本质上也是对于bean的动态加载。

@EnableAutoConfiguration注解的实现原理

了解了ImportSelector和ImportBeanDefinitionRegistrar后，对于EnableAutoConfiguration的理解就容易一些了

它会通过import导入第三方提供的bean的配置类：AutoConfigurationImportSelector

```
@Import(AutoConfigurationImportSelector.class)
```

从名字来看，可以猜到它是基于ImportSelector来实现基于动态bean的加载功能。之前我们讲过Springboot @Enable*注解的工作原理ImportSelector接口selectImports返回的数组（类的全类名）都会被纳入到spring容器中。

那么可以猜想到这里的实现原理也一定是一样的，定位到AutoConfigurationImportSelector这个类中的selectImports方法

selectImports

```
public String[] selectImports(AnnotationMetadata annotationMetadata) {
   if (!isEnabled(annotationMetadata)) {
      return NO_IMPORTS;
   }
// 从配置文件（spring-autoconfigure-metadata.properties）中加载
AutoConfigurationMetadata
   AutoConfigurationMetadata autoConfigurationMetadata =
AutoConfigurationMetadataLoader
         .loadMetadata(this.beanClassLoader);
// 获取所有候选配置类EnableAutoConfiguration
   AutoConfigurationEntry autoConfigurationEntry = getAutoConfigurationEntry(
         autoConfigurationMetadata, annotationMetadata);
   return StringUtils.toStringArray(autoConfigurationEntry.getConfigurations());
}
```

getAutoConfigurationEntry

```
protected AutoConfigurationEntry getAutoConfigurationEntry(
      AutoConfigurationMetadata autoConfigurationMetadata,
      AnnotationMetadata annotationMetadata) {
   if (!isEnabled(annotationMetadata)) {
      return EMPTY_ENTRY;
   }
//获取元注解中的属性
   AnnotationAttributes attributes = getAttributes(annotationMetadata);
//使用SpringFactoriesLoader 加载classpath路径下META-INF\spring.factories中，
//key= org.springframework.boot.autoconfigure.EnableAutoConfiguration对应的value
   List<String> configurations = getCandidateConfigurations(annotationMetadata,
         attributes);
//去重
   configurations = removeDuplicates(configurations);
//应用exclusion属性
   Set<String> exclusions = getExclusions(annotationMetadata, attributes);
   checkExcludedClasses(configurations, exclusions);
   configurations.removeAll(exclusions);
//过滤，检查候选配置类上的注解@ConditionalOnClass，如果要求的类不存在，则这个候选类会被过滤不
被加载
   configurations = filter(configurations, autoConfigurationMetadata);
   //广播事件
fireAutoConfigurationImportEvents(configurations, exclusions);
   return new AutoConfigurationEntry(configurations, exclusions);
}
```

本质上来说，其实EnableAutoConfiguration会帮助springboot应用把所有符合@Configuration配置都加载到当前SpringBoot创建的IoC容器，而这里面借助了Spring框架提供的一个工具类SpringFactoriesLoader的支持。以及用到了Spring提供的条件注解@Conditional，选择性的针对需要加载的bean进行条件过滤

SpringFactoriesLoader

为了给大家补一下基础，我在这里简单分析一下SpringFactoriesLoader这个工具类的使用。它其实和java中的SPI机制的原理是一样的，不过它比SPI更好的点在于不会一次性加载所有的类，而是根据key进行加载。

首先，SpringFactoriesLoader的作用是从classpath/META-INF/spring.factories文件中，根据key来加载对应的类到spring IoC容器中。接下来带大家实践一下

创建外部项目jar

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>4.3.13.RELEASE</version>
</dependency>
```

创建bean以及config

```
public class mashibingCore {
    public String study(){
        System.out.println("good good study, day day up");
        return "mashibingEdu.com";
    }
}
@Configuration
public class mashibingConfig {
    @Bean
    public mashibingCore mashibingCore(){
        return new mashibingCore();
    }
}
```

创建另外一个工程（spring-boot）

把前面的工程打包成jar，当前项目依赖该jar包

```
<dependency>
    <groupId>com.mashibingedu.practice</groupId>
    <artifactId>mashibing-Core</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
```

通过下面代码获取依赖包中的属性

运行结果会报错，原因是mashibingCore并没有被Spring的IoC容器所加载，也就是没有被EnableAutoConfiguration导入

```
@SpringBootApplication
public class SpringBootStudyApplication {
    public static void main(String[] args) throws IOException {
        ConfigurableApplicationContext
ac=SpringApplication.run(SpringBootStudyApplication.class, args);
        mashibingCore Myc=ac.getBean(mashibingCore.class);
        System.out.println(Myc.study());
    }
}
```

解决方案

在mashibing-Core项目resources下新建文件夹META-INF，在文件夹下面新建spring.factories文件，文件中配置，key为自定配置类EnableAutoConfiguration的全路径，value是配置类的全路径

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=com.mashibingedu.
practice.mashibingConfig
```

重新打包，重新运行SpringBootStudyApplication这个类。

可以发现，我们编写的那个类，就被加载进来了。

Spring Boot中的条件过滤

在分析AutoConfigurationImportSelector的源码时，会先扫描spring-autoconfiguration-metadata.properties文件，最后在扫描spring.factories对应的类时，会结合前面的元数据进行过滤，为什么要过滤呢？ 原因是很多的@Configuration其实是依托于其他的框架来加载的，如果当前的classpath环境下没有相关联的依赖，则意味着这些类没必要进行加载，所以，通过这种条件过滤可以有效的减少@configuration类的数量从而降低SpringBoot的启动时间。

修改mashibing-Core

在META-INF/增加配置文件，spring-autoconfigure-metadata.properties。

```
com.mashibingedu.practice.mashibingConfig.ConditionalOnClass=com.mashibingedu.TestClass
```

格式：自动配置的类全名.条件=值

上面这段代码的意思就是，如果当前的classpath下存在TestClass，则会对mashibingConfig这个Configuration进行加载

演示过程(spring-boot)

1. 沿用前面spring-boot工程的测试案例，直接运行main方法，发现原本能够被加载的mashibingCore，发现在ioc容器中找不到了。

   ```
   public static void main(String[] args) throws IOException {
       ConfigurableApplicationContext
   ac=SpringApplication.run(SpringBootStudyApplication.class, args);
       mashibingCore Myc=ac.getBean(mashibingCore.class);
       System.out.println(Myc.study());
   }
   ```

2. 在当前工程中指定的包com.mashibingedu下创建一个TestClass以后，再运行上面这段代码，程序能够正常执行

手写Starter

我们通过手写Starter来加深对于自动装配的理解

1.创建一个Maven项目，quick-starter

定义相关的依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
    <version>2.1.6.RELEASE</version>
</dependency>
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>fastjson</artifactId>
    <version>1.2.56</version>
    <!-- 可选 -->
    <optional>true</optional>
</dependency>
```

2.定义Formate接口

定义的格式转换的接口，并且定义两个实现类

```
public interface FormatProcessor {
    /**
     * 定义一个格式化的方法
     * @param obj
     * @param <T>
     * @return
     */
    <T> String formate(T obj);
}
```

```
public class JsonFormatProcessor implements FormatProcessor {
    @Override
    public <T> String formate(T obj) {
        return "JsonFormatProcessor:" + JSON.toJSONString(obj);
    }
}
```

```
public class StringFormatProcessor implements FormatProcessor {
    @Override
    public <T> String formate(T obj) {
        return "StringFormatProcessor:" + obj.toString();
    }
}
```

3.定义相关的配置类

首先定义格式化加载的Java配置类

```
@Configuration
public class FormatAutoConfiguration {
```

```
    @ConditionalOnMissingClass("com.alibaba.fastjson.JSON")
    @Bean
    @Primary // 优先加载
    public FormatProcessor stringFormatProcessor(){
        return new StringFormatProcessor();
    }

    @ConditionalOnClass(name="com.alibaba.fastjson.JSON")
    @Bean
    public FormatProcessor jsonFormatProcessor(){
        return new JsonFormatProcessor();
    }
}
```

定义一个模板工具类

```
public class HelloFormatTemplate {

    private FormatProcessor formatProcessor;


    public HelloFormatTemplate(FormatProcessor processor){
        this.formatProcessor = processor;
    }

    public <T> String doFormat(T obj){
        StringBuilder builder = new StringBuilder();
        builder.append("Execute format : ").append("<br>");
        builder.append("Object format result:"
).append(formatProcessor.formate(obj));
        return builder.toString();
    }
}
```

再就是整合到SpringBoot中去的Java配置类

```
@Configuration
@Import(FormatAutoConfiguration.class)
public class HelloAutoConfiguration {

    @Bean
    public HelloFormatTemplate helloFormatTemplate(FormatProcessor
formatProcessor){
        return new HelloFormatTemplate(formatProcessor);
    }
}
```

4.创建spring.factories文件

在resources下创建META-INF目录，再在其下创建spring.factories文件

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
  org.mashibingedu.autoconfiguration.HelloAutoConfiguration
```

install 打包，然后就可以在SpringBoot项目中依赖改项目来操作了。

5.测试

在SpringBoot中引入依赖

```
<dependency>
    <groupId>org.example</groupId>
    <artifactId>format-spring-boot-starter</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
```

在controller中使用

```
@RestController
public class UserController {

    @Autowired
    private HelloFormatTemplate helloFormatTemplate;

    @GetMapping("/format")
    public String format(){
        User user = new User();
        user.setName("BoBo");
        user.setAge(18);
        return helloFormatTemplate.doFormat(user);
    }
}
```