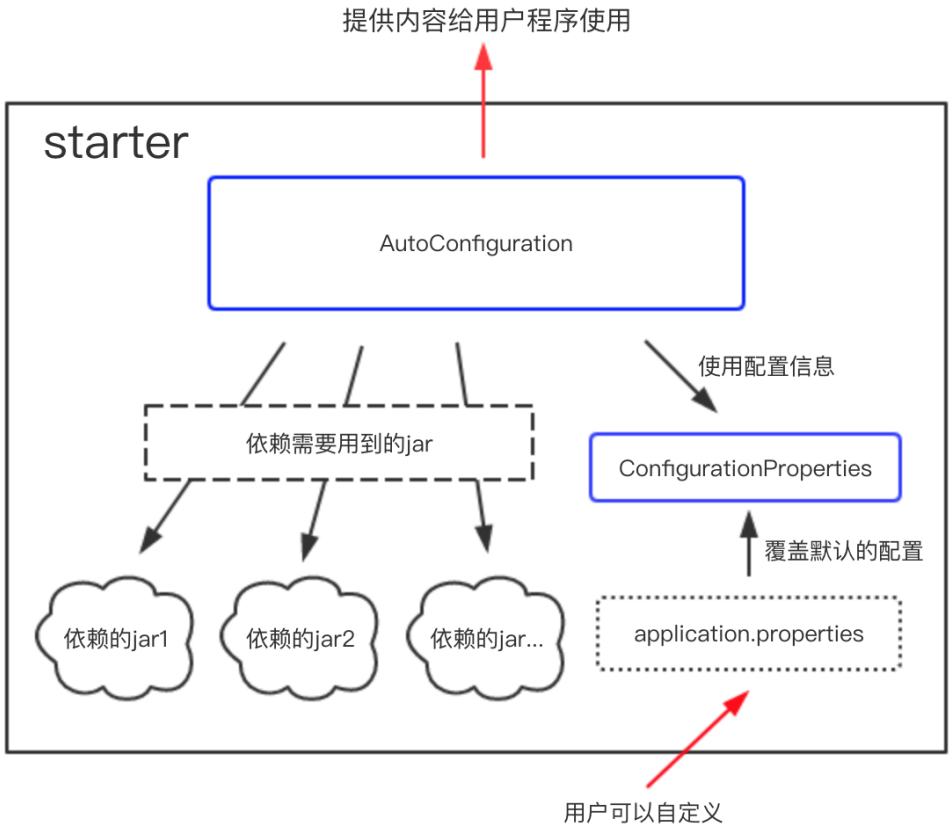
Spring Boot 核心组件之Starter

一、Starter介绍

什么是Starter

starter可以理解成pom配置了一堆jar组合的空maven项目，用来简化maven依赖配置，starter可以继承也可以依赖于别的starter。 例如spring-boot-starter-web包含以下依赖：
org.springframework.boot:spring-boot-starter org.springframework.boot:spring-boot-starter-tomcat org.springframework.boot:spring-boot-starter-validation com.fasterxml.jackson.core:jackson-databind org.springframework:spring-web org.springframework:spring-webmvc spring-boot-starter-test包含以下依赖： junit:junit org.mockito:mockito-core org.hamcrest:hamcrest-core org.hamcrest:hamcrest-library org.springframework:spring-core(excludes commons-logging:commons-logging) org.springframework:spring-test

starter负责配置好与spring整合相关的配置和相关依赖（jar和jar版本），使用者无需关心框架整合带来的问题。



两种Starter

> **如果是springboot官方的starter,命令格式为spring-boot-starter-xxxxx**

例如：

```
<dependency>
```

```
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
            <optional>true</optional>
</dependency>
<dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
</dependency>
```

备注：无需加版本号，与spring-boot 主pom中声明版本一致

> **如果是我们开发starter，命令格式为xxxxx-spring-boot-starter**

例如：

```
<dependency>
  <groupId>org.mybatis.spring.boot</groupId>
  <artifactId>mybatis-spring-boot-starter</artifactId>
  <version>1.3.4</version>
</dependency>
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-starter</artifactId>
    <version>1.1.14</version>
</dependency>
```

备注：必须加版本号，否则maven会报错

Starter应用场景

> **场景一、简化多服务公用框架集成**

众所周知，springboot或者其他第三方所提供的starter，都是做框架集成，通过简化配置，提高开发效率，所以我们自定义starter的第一个应用场景也是基于这个思路。那我们日常开发工作中，有哪些框架是多个服务共用的，并且springboot或者其他第三方暂未提供，或者嫌弃第三方写的太烂，想自己重新实现的，都可以通过编写自定义starter来简化工作。我们公司采用微服务架构，每个服务都会使用swagger来生成在线接口文档。未封装swagger-starter之前，那么在每个服务里边，都需要增加swagger的配置类。而封装swagger-starter之后，可以省去这一步的操作，还可以通过增加自定义配置来实现一些自定义的功能。比如我们公司安全部门要求生产环境不能对外开放swagger接口文档地址，那么我们就可以添加一个enabled的参数来代表swagger是否启用，默认启用，在生产环境的配置中将enabled设为false即可达到这个目的。类似的额外功能还有很多，比如增加请求头等等 上面提到的是业务无关性的starter应用场景，那么我们抛出一个问题，是否有业务相关且多个业务场景或者多个服务会使用的应用场景？根据这个问题的描述，我们至少可以列出以下几个业务相关场景。

> **场景二、服务间调用的鉴权**

我们公司服务之间互相调用需要进行鉴权（还是安全部门的要求），由于服务间是通过feign来实现相互调用，所以无法通过网关来进行统一鉴权。实现方案是通过新增feign拦截器，在源头服务发起调用之前增加鉴权参数，请求到达目标服务后通过鉴权参数进行鉴权。这两步操作很明显是每个服务都需要的，那么这种情况下，我们就可以把这两步操作封装成starter，达到简化开发的目的。同时，我们还可以通过增加配置，实现更细粒度的调用权限控制，比如订单服务只能调用库存服务的查询商品库存接口，而无法调用更新商品库存的接口。

> **场景三：邮件，短信，验证码功能**

这些功能，在某些公司可能会放在common包里，但是这样其实会导致common包的臃肿，因为并不是所有服务都会使用到。有些公司（还是我们公司）可能对邮件服务器的访问有严格权限控制的，而且权限开通流程比较繁复的，那么会考虑做成服务，部署在已经具有访问权限的主机上，减去重复申请权限工作。如果除去这些限制，那么将这些功能封装成starter还是挺不错的，可以避免common包的臃肿。

二、手写一个Http Client Starter

1. 创建一个starter项目

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.demo</groupId>
    <artifactId>myhttp-spring-boot-autoconfigure</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.3.4.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.16.10</version>
        </dependency>

        <!--<dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-configuration-processor</artifactId>
        </dependency>-->

    </dependencies>
```

```
</project>
```

2. 创建一个ConfigurationProperties

用于保存你的配置信息（如果你的项目不使用配置信息则可以跳过这一步，不过这种情况非常少见）

```
import lombok.Data;
import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties(prefix = "http") // 自动获取配置文件中前缀为http的属性，把值传入对象参数
@Data
public class HttpProperties {

    // 如果配置文件中配置了http.url属性，则该默认属性会被覆盖
    private String url;

}
```

上面这个类就是定义了一个属性，其默认值是 `http://www.baidu.com/`，我们可以通过在 `application.properties` 中添加配置 `http.url=https://www.sohu.com` 来覆盖参数的值。

创建业务类：

```
import lombok.Getter;
import lombok.Setter;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.URL;
import java.net.URLConnection;


@Setter
@Getter
public class HttpClient {

    private String url;

    public String getHtml() {
        try {
            URL url = new URL(this.url);
            URLConnection urlConnection = url.openConnection();
                                    // 打开连接
            BufferedReader br = new BufferedReader(new
InputStreamReader(urlConnection.getInputStream(), "utf-8")); // 获取输入流
            String line = null;
            StringBuilder sb = new StringBuilder();
            while ((line = br.readLine()) != null) {
                sb.append(line).append("\n");
            }
            return sb.toString();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return "error";
```

```
    }

}
```

这个业务类的操作非常简单，只包含了一个 `url` 属性和一个 `getHtml` 方法，用于获取一个网页的HTML数据

3. 创建一个AutoConfiguration

```
@Configuration
@EnableConfigurationProperties(HttpProperties.class)
public class HttpAutoConfiguration {

    @Autowired
    private HttpProperties httpProperties;

    // 在Spring上下文中创建一个对象
    @Bean("httpClient")
    @Profile("default")
    @ConditionalOnProperty(value = "http.enable", havingValue = "true",
matchIfMissing = false)
    public HttpClient httpClientDefault() {
        HttpClient httpClient = new HttpClient();
        String url = httpProperties.getUrl();
        httpClient.setUrl(url);
        System.out.println("动态读取地址");
        return httpClient;
    }

    @Bean("httpClient")
    @Profile("fixed")
    @ConditionalOnProperty(value = "http.enable", havingValue = "true",
matchIfMissing = false)
    public HttpClient httpClientFixed() {
        HttpClient httpClient = new HttpClient();
        httpClient.setUrl("https://sohu.com");
        System.out.println("使用固定地址");
        return httpClient;
    }

}
```

注意：`@Configuration` 是 Spring 3.0 引入的

在上面的AutoConfiguration中我们实现了自己要求：在Spring的上下文中创建了一个HttpClient类的bean，并且我们把properties中的一个参数赋给了该bean。 关于`@ConditionalOnProperty` 这个注解，它的意思是在该bean要在符合属性配置的情况下此方法才会执行，这个相当于开关的角色，更多关于开关系列的注解可以参考这里。

Spring 条件装配

从 Spring Framework 3.1 开始，允许在 Bean 装配时增加前置条件判断

条件装配举例

Spring 注解	场景说明	起始版本
@Profile	配置化条件装配	3.1
@Conditional	编程条件装配	4.0

4. 装配HttpAutoConfiguration

1. 使用@ComponentScan装配

Spring 模式注解装配

模式注解 Stereotype Annotations

A ***stereotype annotation*** is an annotation that is used to declare the role that a component plays within the application. For example, the `@Repository` annotation in the Spring Framework is a marker for any class that fulfills the role or *stereotype* of a repository (also known as Data Access Object or DAO).

`@Component` is a generic stereotype for any Spring-managed component. Any component annotated with `@Component` is a candidate for component scanning. Similarly, any component annotated with an annotation that is itself meta-annotated with `@Component` is also a candidate for component scanning. For example, `@Service` is meta-annotated with `@Component`.

Core Spring provides several stereotype annotations out of the box, including but not limited to: `@Component`, `@Service`, `@Repository`, `@Controller`, `@RestController`, and `@Configuration`. `@Repository`, `@Service`, etc. are specializations of `@Component`.

一种用于声明在应用中扮演“组件”角色的注解，如 Spring Framework 中的 `@Repository` 标注在任何类上，用于扮演仓储角色（DAO）的模式注解。

`@Component` 作为一种由 Spring 容器托管的通用模式组件，任何被 `@Component` 标注的组件均为组件扫描的候选对象。类似的，凡是被 `@Component` 元标注（meta-annotated）的注解，如`@Service`，`@Repository`，`@Controller`，`@RestController`，and `@Configuration`。当任何组件标注它时，也被视作组件扫描的候选对象。

模式注解举例

Spring Framework 注解	场景说明	起始版本
`@Repository`	数据仓库模式注解	2.0
`@Component`	通用组件模式注解	2.5
`@Service`	服务模式注解	2.5
`@Controller`	Web 控制器模式注解	2.5
`@Configuration`	配置类模式注解	3.0

代码实现

```
@ComponentScan(basePackages = {"com.yuandengta.boot",
"com.yuandengta.http.autoconfigure"})
```

2. 使用@Enable 模块装配

Spring @Enable 模块装配

Spring Framework 3.1 开始支持 "@Enable 模块驱动"。所谓"模块"是指具备相同领域的功能组件集合， 组合所形成一个独立的单元。比如 Web MVC 模块、AspectJ 代理模块、Caching（缓存）模块、JMX（Java 管 理扩展）模块、Async（异步处理）模块等。

@Enable注解模块举例

框架实现	@Enable 注解模块	激活模块
Spring Framework	`@EnableWebMvc`	Web MVC 模块
	`@EnableTransactionManagement`	事务管理模块
	`@EnableCaching`	Caching 模块
	`@EnableMBeanExport`	JMX 模块
	`@EnableAsync`	异步处理模块
	`@EnableWebFlux`	Web Flux 模块
	`@EnableAspectJAutoProxy`	AspectJ 代理模块
Spring Boot	@EnableAutoConfiguration	自动装配模块
	@EnableManagementContext	Actuator管理模块
	@EnableConfigurationProperties	配置属性绑定模块
	@EnableOAuth2Sso	OAuth2 单点登录模块
Spring Cloud	@EnableEurekaServer	Eureka服务器模块
	@EnableConfigServer	配置服务器模块
	@EnableFeignClients	Feign客户端模块
	@EnableZuulProxy	服务网关 Zuul 模块
	@EnableCircuitBreaker	服务熔断模块

实现方式

注解驱动方式

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
@Documented
@Import({DelegatingWebMvcConfiguration.class})
public @interface EnableWebMvc {
}
```

```
@Configuration(
    proxyBeanMethods = false
)
public class DelegatingWebMvcConfiguration extends WebMvcConfigurationSupport {
    ...
}
```

接口编程方式

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import({CachingConfigurationSelector.class})
public @interface EnableCaching {
    ...
}
```

```
/**
 * Selects which implementation of {@link AbstractCachingConfiguration} should
 * be used based on the value of {@link EnableCaching#mode} on the importing
 * {@code @Configuration} class.
 *
 * <p>Detects the presence of JSR-107 and enables JCache support accordingly.
 *
 * @author Chris Beams
 * @author Stephane Nicoll
 * @since 3.1
 * @see EnableCaching
 * @see ProxyCachingConfiguration
 */
public class CachingConfigurationSelector extends
AdviceModeImportSelector<EnableCaching> {
    /**
     * Returns {@link ProxyCachingConfiguration} or {@code
AspectJCachingConfiguration}
     * for {@code PROXY} and {@code ASPECTJ} values of {@link
EnableCaching#mode()},
     * respectively. Potentially includes corresponding JCache configuration as
well.
     */
    public String[] selectImports(AdviceMode adviceMode) {
        switch(adviceMode) {
        case PROXY:
            return this.getProxyImports(); // 使用代理方式
```

```
            case ASPECTJ:
                return this.getAspectJImports(); // 使用 AspectJ 方式
            default:
                return null;
            }
        }
    }
```

接口编程方式（ImportSelector）来实现更加灵活，底层可以有多种实现。

自定义 @EnableHttpClient 模块

基于注解驱动实现 - @EnableHttpClientByAnnotation

定义 `@EnableHttpClientByAnnotation`

EnableHttpClientByAnnotation > HttpAutoConfiguration-> HttpClient

```
/**
 * 激活Http模块
 */
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import(HttpAutoConfiguration.class)
public @interface EnableHttpClientByAnnotation {
}
```

定义主启动类 EnableHttpAnnotationBootstrap

```
@SpringBootApplication(exclude = DataSourceAutoConfiguration.class)
@EnableHttpClientByAnnotation
public class EnableHttpAnnotationBootstrap implements CommandLineRunner {

    @Autowired
    private HttpClient httpClient;

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

    public void run(String... strings) throws Exception {
        //执行请求
        String result = httpClient.getHtml();
        System.out.println(result);
    }
}
```

基于接口驱动实现 - @EnableHttpClientByInterface

EnableHttpClientByInterface -> HttpSelector --> HttpAutoConfiguration --> HttpClient

定义 ImportSelector 实现类

```
/**
 * ImportSelector接口
 */
public class HttpSelector implements ImportSelector {
    @Override
    public String[] selectImports(AnnotationMetadata annotationMetadata) {
        return new String[]
{"com.yuandengta.autoconfigure.HttpAutoConfiguration"};
    }
}
```

注意：ImportSelector 是 Spring 3.1 开始引入的

修改 `@EnableHttpClientByInterface`

```
/**
 * Enable注解
 */
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import(HttpSelector.class)
public @interface EnableHttpClientByInterface {
}
```

启动 主启动类 EnableHttpBootstrap进行测试，结果一致

```
@SpringBootApplication(exclude = DataSourceAutoConfiguration.class)
@EnableHttpClientByInterface
public class EnableHttpInterfaceBootstrap implements CommandLineRunner {

    @Autowired
    private HttpClient httpClient;

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

    public void run(String... strings) throws Exception {
        //执行请求
        String result = httpClient.getHtml();
        System.out.println(result);
    }
}
```

3. 使用spring.factories自动装配

Spring Boot 自动装配

上面我们将了三种Spring 的手动装配实现方式，那么 Spring Boot 自动装配和手动装配有什么关系？我们看一个比较熟悉的 `WebMvcAutoConfiguration` ，看到了我们熟悉的身影 `@Configuration` ，第二个 `@ConditionalOnWebApplication` ，它的本质是 Spring 4 中 Condition 来实现的，因此我们发现这个里面既有模式注解，又有 Condition 条件。

在 Spring Boot 场景下，基于约定大于配置的原则，实现 Spring 组件自动装配的目的。其中使用了

底层装配技术

- Spring 模式注解装配
- Spring `@Enable` 模块装配
- Spring 条件装配装配
- Spring 工厂加载机制
    - 实现类： `SpringFactoriesLoader`
    - 配置资源： `META-INF/spring.factories`

工厂加载机制其实也是 Spring 里面的一个机制，它的实现类是 `SpringFactoriesLoader`，我们可以打开这个类查看源码，这个类我们看到是 Spring 3.2 提供的，这个工具是干嘛用的呢，其实非常简单，就是用于装载配置的方法，从字面意思也可以看出，我们称之为 Spring 工厂，它有一个方法 `loadFactories`，这个方法里面首先有一个工厂类，还有一个加载器，它的实现方式其实很简单，就是通过一个 for 循环进行迭代装配。那这个是用在什么地方呢？我们可以搜索 `myhttp-spring-boot-starter` 下面的 `spring.factories`，这个文件也是在我们规约的位置 `META-INF/` 下面， 那什么是工厂类呢，也就是第一个参数， 这个第一个参数其实是另外一个东西，就是激活自动装配 `EnableAutoConfiguration`，当这个类被装配在我们引导环时，那么下面的一些具体的自动化配置类会被相应的组装。具体的实现方式是放在了 Starter 模块里面进行组装的。我们看一下 Maven 的依赖，Starter 是模块化的，Starter 其实就是一个场景启动器，启动的时候就会装配一些相应的配置来完成自动化配置的实现，这就是 Spring Boot 自动化配置的本质。

自动装配举例

参考 `myhttp-spring-boot-autoconfigure` 模块下的 `META-INF/spring.factories`

实现方法

1. 激活自动装配 - `@EnableAutoConfiguration`
2. 实现自动装配 - `XxxAutoConfiguration`
3. 配置自动装配实现 - `META-INF/spring.factories`

自定义自动装配

编写自动配置类

`HttpAutoConfiguration`

```
@Configuration
@EnableConfigurationProperties(HttpProperties.class)
public class HttpAutoConfiguration {

    @Autowired
    private HttpProperties httpProperties;

    // 在Spring上下文中创建一个对象
```

```
    @Bean
    @ConditionalOnMissingBean
    public HttpClient httpClient() {
        HttpClient httpClient = new HttpClient();
        String url = httpProperties.getUrl();
        httpClient.setUrl(url);
        return httpClient;
    }
}
```

配置自动装配实现

在当前工程 resources 目录下新建 META-INF 目录，并在 META-INF 目录下创建 spring.factories 文件

```
# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=com.yuandengta.http.autoconfigure.HttpAutoConfiguration
```

编写主启动类

```
@SpringBootApplication
public class SpringApplicationBootstrap implements CommandLineRunner {

    @Autowired
    private HttpClient httpClient;

    public static void main(String[] args) {
        ConfigurableApplicationContext context =
                new SpringApplicationBuilder(SpringApplicationBootstrap.class)
                        .web(WebApplicationType.NONE)
                        .profiles("default")
                        .run(args);
    }

    public void run(String... strings) throws Exception {
        //执行请求
        String result = httpClient.getHtml();
        System.out.println(result);
    }
}
```

5、打包项目，使用该starter

到此，我们的starter已经创建完毕了，使用Maven打包该项目。之后创建一个SpringBoot项目，在项目中添加我们之前打包的starter作为依赖，

```
<dependency>
    <groupId>com.demo</groupId>
    <artifactId>myhttp-spring-boot-starter</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
```

使用SringBoot来运行我们的starter，代码如下：

```
@SpringBootApplication
public class SpringApplicationBootstrap implements CommandLineRunner {

    @Autowired
    private HttpClient httpClient;

    public static void main(String[] args) {
        ConfigurableApplicationContext context =
                new SpringApplicationBuilder(SpringApplicationBootstrap.class)
                        .web(WebApplicationType.NONE)
                        .profiles("default")
                        .run(args);
    }

    public void run(String... strings) throws Exception {
        //执行请求
        String result = httpClient.getHtml();
        System.out.println(result);
    }
}
```

正常情况下此方法的执行会打印出url `http://www.baidu.com/` 的HTML内容，之后我们在application.properties中加入配置：

```
http.url=https://www.zhihu.com/
```

再次运行程序，此时打印的结果应该是百度首页的HTML了，证明properties中的数据确实被覆盖了。

三、分析Starter启动原理

引入starter依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

springboot-boot-starter:就是springboot的场景启动器。springboot将所有的功能场景都抽取出来，做成一个个的starter，只需要在项目中引入这些starter即可，所有相关的依赖都会导入进来，根据公司业务需求决定导入什么启动器即可。

加载@SpringBootApplication注解

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
//SpringBootApplication注解用来标注一个主程序类，说明是一个springboot应用
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

加载@EnableAutoConfiguration注解

```
/*
@ComponentScan:自动扫描并加载符合条件的组件或者bean，将这个bean定义加载到IOC容器中

@SpringBootConfiguration:标注在某个类上，表示这是一个springboot的配置类。

@EnableAutoConfiguration:开启自动配置功能,之前在使用springboot的时候，springboot可以自动帮我们完成配置功能，@EnableAutoConfiguration告诉springboot开启自动配置功能，这样自动配置才能生效
*/
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes =
TypeExcludeFilter.class),
        @Filter(type = FilterType.CUSTOM, classes =
AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {}
```

加载@Configuration注解

```
/*
可以看到SpringBootConfiguration使用了Configuration注解来标注
*/
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Configuration
public @interface SpringBootConfiguration {}
```

加载@Component注解

```
/*
可以看到Configuration也是容器中的一个组件
*/
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Configuration {}
```

加载AutoConfigurationImportSelector类

```
/*
*@AutoConfigurationPackage:自动配置包
*@Import(AutoConfigurationImportSelector.class)：导入哪些组件的选择器，它将所有需要导入的组件以全类名的方式返回，这些组件就会被添加到容器中，它会给* *容器中导入非常多的自动配置类，就是给容器中导入这个场景需要的所有组件，并配置好这些组件
*/
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(AutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {}
```

```
/*
给容器导入一个组件，导入的组件由AutoConfigurationPackages.Registrar.class将主配置类（@SpringBootApplication标注的类）的所在包及包下面所有子包里面的所有组件扫描到spring容器
*/
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Import(AutoConfigurationPackages.Registrar.class)
public @interface AutoConfigurationPackage {}
```

- 利用AutoConfigurationImportSelector给容器导入一些组件。
- 查看selectImports方法的内容，返回一个AutoConfigurationEntry

```
AutoConfigurationEntry autoConfigurationEntry =
getAutoConfigurationEntry(autoConfigurationMetadata,
      annotationMetadata);
------
List<String> configurations = getCandidateConfigurations(annotationMetadata,
attributes);
------
protected List<String> getCandidateConfigurations(AnnotationMetadata metadata,
AnnotationAttributes attributes) {
        List<String> configurations =
SpringFactoriesLoader.loadFactoryNames(getSpringFactoriesLoaderFactoryClass(),
                getBeanClassLoader());
        Assert.notEmpty(configurations, "No auto configuration classes found in
META-INF/spring.factories. If you "
                + "are using a custom packaging, make sure that file is
correct.");
        return configurations;
    }
```

```
/*
在AutoConfigurationImportSelector类中有如下方法，可以看到

*/
```

```
protected List<String> getCandidateConfigurations(AnnotationMetadata metadata,
AnnotationAttributes attributes) {
        List<String> configurations =
SpringFactoriesLoader.loadFactoryNames(getSpringFactoriesLoaderFactoryClass(),
                getBeanClassLoader());
        Assert.notEmpty(configurations, "No auto configuration classes found in
META-INF/spring.factories. If you "
                + "are using a custom packaging, make sure that file is
correct.");
        return configurations;
    }
/*

此时返回的就是启动自动导入配置文件的注解类
*/
protected Class<?> getSpringFactoriesLoaderFactoryClass() {
        return EnableAutoConfiguration.class;
    }

//进入SpringFactoriesLoader类中
/*
 * 看到会读取对应的配置文件，位置在META-INF/spring.factories中
 */
public final class SpringFactoriesLoader {

    /**
     * The location to look for factories.
     * <p>Can be present in multiple JAR files.
     */
public static final String FACTORIES_RESOURCE_LOCATION = "META-
INF/spring.factories";}

//进入loadFactoryNames方法中
public static List<String> loadFactoryNames(Class<?> factoryType, @Nullable
ClassLoader classLoader) {
        String factoryTypeName = factoryType.getName();
        return loadSpringFactories(classLoader).getOrDefault(factoryTypeName,
Collections.emptyList());
    }

    private static Map<String, List<String>> loadSpringFactories(@Nullable
ClassLoader classLoader) {
        MultiValueMap<String, String> result = cache.get(classLoader);
        if (result != null) {
            return result;
        }

        try {
            Enumeration<URL> urls = (classLoader != null ?
                    classLoader.getResources(FACTORIES_RESOURCE_LOCATION) :

ClassLoader.getSystemResources(FACTORIES_RESOURCE_LOCATION));
            result = new LinkedMultiValueMap<>();
            while (urls.hasMoreElements()) {
                URL url = urls.nextElement();
                UrlResource resource = new UrlResource(url);
                Properties properties =
PropertiesLoaderUtils.loadProperties(resource);
```

```
                for (Map.Entry<?, ?> entry : properties.entrySet()) {
                    String factoryTypeName = ((String) entry.getKey()).trim();
                    for (String factoryImplementationName :
StringUtils.commaDelimitedListToStringArray((String) entry.getValue())) {
                        result.add(factoryTypeName,
factoryImplementationName.trim());
                    }
                }
            }
            cache.put(classLoader, result);
            return result;
        }
        catch (IOException ex) {
            throw new IllegalArgumentException("Unable to load factories from
location [" +
                    FACTORIES_RESOURCE_LOCATION + "]", ex);
        }
    }
```

springboot在启动的时候从类路径下的META-INF/spring.factories中获取EnableAutoConfiguration指定的值，将这些值作为自动配置类导入容器，自动配置类就生效，帮我们进行自动配置的工作：spring.factories文件位于springboot-autoconfigure.jar包中。

所以真正实现是从classpath中搜寻所有的**META-INF/spring.factories**配置文件，并将其中对应org.springframework.boot.autoconfigure.包下的配置项通过反射实例化为对应标注了@Configuration的JavaConfig形式的IOC容器配置类，然后将这些都汇总称为一个实例并加载到IOC容器中。

以HttpEncodingAutoConfiguration为例，来解释自动装配原理

```
/*
表名这是一个配置类，
*/
@Configuration(proxyBeanMethods = false)
/*
启动指定类的ConfigurationProperties功能,进入HttpProperties查看，将配置文件中对应的值和
HttpProperties绑定起来，并把HttpProperties加入到ioc容器中
*/
@EnableConfigurationProperties(HttpProperties.class)
/*
spring底层@Confitional注解，根据不同的条件判断，如果满足指定的条件，整个配置类里面的配置就会
生效
此时表示判断当前应用是否是web应用，如果是，那么配置类生效
*/
@ConditionalOnWebApplication(type = ConditionalOnWebApplication.Type.SERVLET)
/*
判断当前项目由没有这个类CharacterEncodingFilter，springmvc中进行乱码解决的过滤器
*/
@ConditionalOnClass(CharacterEncodingFilter.class)
/*
判断配置文件中是否存在某个配置：spring.http.encoding.enabled
如果不存在，判断也是成立的，
即使我们配置文件中不配置spring.http.encoding.enabled=true，也是默认生效的
*/
@ConditionalOnProperty(prefix = "spring.http.encoding", value = "enabled",
matchIfMissing = true)
public class HttpEncodingAutoConfiguration {
```

```
    //和springboot的配置文件映射
    private final HttpProperties.Encoding properties;

    //只有一个有参构造器的情况下，参数的值就会从容器中拿
    public HttpEncodingAutoConfiguration(HttpProperties properties) {
        this.properties = properties.getEncoding();
    }

    //给容器中添加一个组件，这个组件的某些值需要从properties中获取
    @Bean
    @ConditionalOnMissingBean//判断容器中是否有此组件
    public CharacterEncodingFilter characterEncodingFilter() {
        CharacterEncodingFilter filter = new OrderedCharacterEncodingFilter();
        filter.setEncoding(this.properties.getCharset().name());

filter.setForceRequestEncoding(this.properties.shouldForce(Type.REQUEST));

filter.setForceResponseEncoding(this.properties.shouldForce(Type.RESPONSE));
        return filter;
    }

    @Bean
    public LocaleCharsetMappingsCustomizer localeCharsetMappingsCustomizer() {
        return new LocaleCharsetMappingsCustomizer(this.properties);
    }

    private static class LocaleCharsetMappingsCustomizer
            implements
WebServerFactoryCustomizer<ConfigurableServletWebServerFactory>, Ordered {

        private final HttpProperties.Encoding properties;

        LocaleCharsetMappingsCustomizer(HttpProperties.Encoding properties) {
            this.properties = properties;
        }

        @Override
        public void customize(ConfigurableServletWebServerFactory factory) {
            if (this.properties.getMapping() != null) {
                factory.setLocaleCharsetMappings(this.properties.getMapping());
            }
        }

        @Override
        public int getOrder() {
            return 0;
        }
    }
}
```

根据当前不同的条件判断，决定这个配置类是否生效!

总结:

1、springboot启动会加载大量的自动配置类

2、查看需要的功能有没有在springboot默认写好的自动配置类中

3、查看这个自动配置类到底配置了哪些组件

4、给容器中自动配置类添加组件的时候，会从properties类中获取属性

@Conditional：自动配置类在一定条件下才能生效

@Conditional扩展注解	**作用**
@ConditionalOnJava	系统的java版本是否符合要求
@ConditionalOnBean	容器中存在指定Bean
@ConditionalOnMissingBean	容器中不存在指定Bean
@ConditionalOnExpression	满足SpEL表达式
@ConditionalOnClass	系统中有指定的类
@ConditionalOnMissingClass	系统中没有指定的类
@ConditionalOnSingleCandidate	容器中只有一个指定的Bean，或者是首选Bean
@ConditionalOnProperty	系统中指定的属性是否有指定的值
@ConditionalOnResource	类路径下是否存在指定资源文件
@ConditionOnWebApplication	当前是web环境
@ConditionalOnNotWebApplication	当前不是web环境
@ConditionalOnJndi	JNDI存在指定项