朱晔和你聊 Spring 系列 S1E2：SpringBoot 并不神秘

文本我们会一步一步做一个例子来看看 SpringBoot 的自动配置是如何实现的，然后来看一些 SpringBoot 留给我们的扩展点。

自己制作一个 SpringBoot Starter

我们知道 SpringBoot 提供了非常多的启动器，引入了启动器依赖即可直接享受到自动依赖配置和自动属性配置：

spring-boot-starter-jetty	Support Tomcat 9 and Undertow 2	3 months ago
spring-boot-starter-jooq	Remove redundant spring-boot-starter dependencies from starters	2 months ago
spring-boot-starter-json	Remove spring.provides	6 months ago
spring-boot-starter-jta-atomikos	Remove spring.provides	6 months ago
spring-boot-starter-jta-bitronix	Remove spring.provides	6 months ago
spring-boot-starter-log4j2	Merge branch '2.0.x'	4 months ago
spring-boot-starter-logging	Remove spring.provides	6 months ago
spring-boot-starter-mail	Remove spring.provides	6 months ago
spring-boot-starter-mustache	Remove spring.provides	6 months ago
spring-boot-starter-oauth2-oidc-...	Revert "Polish dependency management for OIDC starter"	a month ago
spring-boot-starter-parent	Add execution id to `repackage` goal	a month ago
spring-boot-starter-quartz	Remove spring.provides	6 months ago
spring-boot-starter-reactor-netty	Upgrade to Spring Framework 5.1	4 months ago
spring-boot-starter-security	Remove spring.provides	6 months ago
spring-boot-starter-test	Remove spring.provides	6 months ago
spring-boot-starter-thymeleaf	Remove spring.provides	6 months ago
spring-boot-starter-tomcat	Remove spring.provides	6 months ago
spring-boot-starter-undertow	Support Tomcat 9 and Undertow 2	3 months ago
spring-boot-starter-validation	Remove spring.provides	6 months ago
spring-boot-starter-web-services	Rework "Remove redundant spring-boot-starter dependency"	a month ago

https://github.com/spring-projects/spring-boot/tree/master/spring-boot-project/spring-boot-starters

在第一篇文章中我提到，在 SpringBoot 出现之前，我们需要使用 SpringMVC、Spring Data、Spring Core 都需要对 Spring 内部的各种组件进行 Bean 以及 Bean 依赖的配置，在90%的时候我们用的是默认的配置，不会自定义任何扩展类，这个时候也需要由使用者来手动配置显然不合理，有了 SpringBoot，我们只需引入启动器依赖，然后启动器就可以自己做为自己的一些内部组件做自动配置，大大方便了使用者。启动器的实现非常简单，我们来看下实现过程。

首先创建一个 Maven 空项目，引入 SpringBoot：

```xml
<?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>me.josephzhu</groupId>
    <artifactId>spring101</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>spring101</name>
    <description></description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.5.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>

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

```xml
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>


</project>
```

然后我们创建一个 Starter 模块隶属于父项目：

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://maven.apache.org/POM/4.0.0"
         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>me.josephzhu</groupId>
    <artifactId>spring101-customstarter</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>spring101-customstarter</name>
    <description></description>

    <parent>
        <groupId>me.josephzhu</groupId>
        <artifactId>spring101</artifactId>
        <version>0.0.1-SNAPSHOT</version>
    </parent>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-autoconfigure</artifactId>
        </dependency>
    </dependencies>

</project>
```

接下去我们创建一个服务抽象基类和实现，这个服务非常简单，会依赖到一些属性，然后会有不同的实现（无参构造函数设置了打招呼用语的默认值为 Hello）：

```
package me.josephzhu.spring101customstarter;

import org.springframework.beans.factory.annotation.Autowired;

public abstract class AbstractMyService {

    protected String word;
    public AbstractMyService(String word) {
        this.word = word;
    }

    public AbstractMyService() {
        this ("Hello");
    }

    @Autowired
    protected MyServiceProperties properties;

    public abstract String hello();
}
```

这里注入了自定义属性类：

```
package me.josephzhu.spring101customstarter;

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

@ConfigurationProperties(prefix = "spring101")
@Data
public class MyServiceProperties {
    /**
     * user name
     */
    private String name;
    /**
     * user age *Should between 1 and 120
     */
    private Integer age;
    /**
     * determine the service version you want use
     */
```

```
    private String version;
}
```
这里看到了如果我们需要定义一个自定义类来关联配置源（比如 application.properties 文件配置）是多么简单，使用@ConfigurationProperties 注解标注我们的 POJO 告知注解我们配置的前缀即可。额外提一句，如果希望我们的 IDE 可以针对自定义配置有提示的话（自动完成，而且带上注解中的提示语），可以引入如下的依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>
```
这样编译后就会在 META-INF 下面生成一个叫做 spring-configuration-metadata.json 的文件：


```
{
  "hints": [],
  "groups": [
    {
      "sourceType": "me.josephzhu.spring101customstarter.MyServiceProperties",
      "name": "spring101",
      "type": "me.josephzhu.spring101customstarter.MyServiceProperties"
    }
  ],
  "properties": [
    {
      "sourceType": "me.josephzhu.spring101customstarter.MyServiceProperties",
      "name": "spring101.age",
      "description": "user age *Should between 1 and 120",
      "type": "java.lang.Integer"
    },
    {
      "sourceType": "me.josephzhu.spring101customstarter.MyServiceProperties",
      "name": "spring101.name",
      "description": "user name",
      "type": "java.lang.String"
    },
    {
      "sourceType": "me.josephzhu.spring101customstarter.MyServiceProperties",
      "name": "spring101.version",
      "description": "determine the service version you want use",
      "type": "java.lang.String"
    }
```

```
  ]
}
```

之后在使用配置的时候就可以有提示:

```
1  spring101.age=35
2  spring101.name=zhuye
3  spring101.version=v3
4    p spring101.version (determine the service version …   String
     ^↓ and ^↑ will move caret down and up in the editor >>
```

我们先来写第一个服务实现，如下，只是输出一下使用到的一些自定义属性:

```java
package me.josephzhu.spring101customstarter;

import org.springframework.stereotype.Service;

@Service
public class MyService extends AbstractMyService {

    public MyService(String word) {
        super(word);
    }

    public MyService(){}

    @Override
    public String hello() {
        return String.format("V1 %s >> %s:%s !!", word, properties.getName(), properties.getAge());
    }

}
```

关键的一步来了，接下去我们需要定义自动配置类:

```java
package me.josephzhu.spring101customstarter;

import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
@EnableConfigurationProperties(MyServiceProperties.class)
public class MyAutoConfiguration {
    @Bean
    MyService getMyService(){
```

```java
            return new MyService();
        }
}
```

通过 EnableConfigurationProperties 来告诉 Spring 我们需要关联一个配置文件配置类（配置类斗不需要设置@Component），通过@Configuration 告诉 Spring 这是一个 Bean 的配置类，下面我们定义了我们 Service 的实现。

最后，我们需要告诉 SpringBoot 如何来找到我们的自动配置类，在合适的时候自动配置。我们需要在项目资源目录建一个 META-INF 文件夹，然后创建一个 spring.factories 文件，写入下面的内容：

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=me.josephzhu.spring101customstarter.MyAutoConfiguration
```

好了，就是这么简单，接下去我们创建一个项目来使用我们的自定义启动器来试试：

```xml
<?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>me.josephzhu</groupId>
    <artifactId>spring101-main</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>spring101-main</name>
    <description></description>

    <parent>
        <groupId>me.josephzhu</groupId>
        <artifactId>spring101</artifactId>
        <version>0.0.1-SNAPSHOT</version>
    </parent>

    <dependencies>
        <dependency>
            <groupId>me.josephzhu</groupId>
            <artifactId>spring101-customstarter</artifactId>
            <version>0.0.1-SNAPSHOT</version>
        </dependency>

    </dependencies>
```

```
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>


</project>
```

创建一个 Runner 来调用服务:

```
package me.josephzhu.spring101main;

import lombok.extern.slf4j.Slf4j;
import me.josephzhu.spring101customstarter.AbstractMyService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

@Component
@Slf4j
public class Runner1 implements CommandLineRunner {

    @Autowired
    private AbstractMyService service;

    @Override
    public void run(String... args) {
        log.info(service.hello());
    }
}
```

创建主程序:

```
package me.josephzhu.spring101main;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Spring101MainApplication {

    public static void main(String[] args) {
        SpringApplication.run(Spring101MainApplication.class, args);
```

```
    }

}
```

然后在 main 模块的资源目录下创建 application.properties 文件，写入两个配置：

```
spring101.age=35
spring101.name=zhuye
```

运行后可以看到输出：

```
2018-09-30 14:55:00.848  INFO 12704 --- [           main]
me.josephzhu.spring101main.Runner1       : V1 Hello >> zhuye:35 !!
```

可以证明，第一我们的 main 模块引入的 starter 正确被 SpringBoot 识别加载，第二 starter 中的 Configuration 正确执行不但加载了配置类，而且也正确注入了 Service 的一个实现。

如何实现条件配置？

作为组件的开发者，我们有的时候希望针对环境、配置、类的加载情况等等进行各种更智能的自动配置，这个时候就需要使用 Spring 的 Conditional 特性。我们来看一个例子，如果我们的 Service 随着发展演化出了 v2 版本，我们希望用户在默认的时候使用 v1，如果需要的话可以进行 version 属性配置允许用户切换到 v2 版本。实现起来非常简单，首先定义另一个 v2 版本的服务：

```java
package me.josephzhu.spring101customstarter;

import org.springframework.stereotype.Service;

@Service
public class MyServiceV2 extends AbstractMyService {

    public MyServiceV2(String word) {
        super(word);
    }

    public MyServiceV2(){}

    @Override
    public String hello() {
        return String.format("V2 %s >> %s:%s !!", word, properties.getName(), properties.getAge());
    }

}
```

和版本 v1 没有任何区别，只是标记了一下 v2 关键字。

然后我们改造一下我们的自动配置类：

```
package me.josephzhu.spring101customstarter;

import org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
@EnableConfigurationProperties(MyServiceProperties.class)
public class MyAutoConfiguration {
    @Bean
    @ConditionalOnProperty(prefix = "spring101", name = "version", havingValue = "v1", matchIfMissing = true)
    MyService getMyService(){
        return new MyService();
    }

    @Bean
    @ConditionalOnProperty(prefix = "spring101", name = "version", havingValue = "v2")
    MyServiceV2 getMyServiceV2(){
        return new MyServiceV2();
    }
}
```

这里主要是为两个 Bean 分别添加了@ConditionalOnProperty 注解，注解是自解释的。这里说了如果 version 的值是 v1 或没有定义 version 的话匹配到默认的 v1 版本的服务，如果配置设置为 v2 的话匹配到 v2 版本的服务，就这么简单。

再来看一个例子，如果我们的使用者希望自己定义服务的实现，这个时候我们需要覆盖自动配置为我们自动装配的 v1 和 v2，可以使用另一个注解@ConditionalOnMissingBean 来告知 SpringBoot，如果找不到 Bean 的话再来自动配置：

```
package me.josephzhu.spring101customstarter;

import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;
import org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
@EnableConfigurationProperties(MyServiceProperties.class)
public class MyAutoConfiguration {
```

```java
    @Bean
    @ConditionalOnMissingBean(MyService.class)
    @ConditionalOnProperty(prefix = "spring101", name = "version", havingValue = "v1", matchIfMissing = true)
    MyService getMyService(){
        return new MyService();
    }

    @Bean
    @ConditionalOnMissingBean(MyServiceV2.class)
    @ConditionalOnProperty(prefix = "spring101", name = "version", havingValue = "v2")
    MyServiceV2 getMyServiceV2(){
        return new MyServiceV2();
    }
}
```

这样的话，如果客户端自己定义了 Service 的实现的话，就可以让自动配置放弃自动配置使用客户端自己定义的 Bean。还有 N 多的 Conditional 注解可以使用，甚至可以自定义条件，具体可以查看官方文档。

进行一下测试

接下去，我们来写一下单元测试来验证一下我们之前的代码，使用 ApplicationContextRunner 可以方便得设置带入的各种外部配置项以及自定义配置类：

在这里我们写了三个测试用例：

- 在提供了合适的属性配置后，可以看到服务的输出正确获取到了属性。
- 使用配置项 version 来切换服务的实现，在省略 version，设置 version 为 1，设置 version 为 2 的情况下得到正确的输出，分别是 v1、v1 和 v2。
- 在客户端自定义实现（MyServiceConfig）后可以看到并没有加载使用自动配置里定义的服务实现，最后输出了打招呼用语 Hi 而不是 Hello。

```java
package me.josephzhu.spring101main;

import me.josephzhu.spring101customstarter.AbstractMyService;
import me.josephzhu.spring101customstarter.MyAutoConfiguration;
import me.josephzhu.spring101customstarter.MyService;
import org.junit.Test;
import org.springframework.boot.autoconfigure.AutoConfigurations;
import org.springframework.boot.test.context.runner.ApplicationContextRunner;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

```java
import static org.assertj.core.api.Assertions.assertThat;


public class Spring101MainApplicationTests {

    private ApplicationContextRunner applicationContextRunner = new ApplicationContextRunner()
            .withConfiguration(AutoConfigurations.of(MyAutoConfiguration.class));

    @Test
    public void testService() {
        applicationContextRunner
                .withPropertyValues("spring101.age=35")
                .withPropertyValues("spring101.name=zhuye")
                .run(context -> {
                    assertThat(context).hasSingleBean(AbstractMyService.class);
                    assertThat(context.getBean(AbstractMyService.class).hello()).containsSequence("zhuye:35");
                    System.out.println(context.getBean(MyService.class).hello());
                });
    }

    @Test
    public void testConditionalOnProperty() {
        applicationContextRunner
                .run(context -> {
                    assertThat(context).hasSingleBean(AbstractMyService.class);
                    assertThat(context.getBean(AbstractMyService.class).hello()).containsSequence("V1 Hello");
                    System.out.println(context.getBean(AbstractMyService.class).hello());
                });
        applicationContextRunner
                .withPropertyValues("spring101.version=v1")
                .run(context -> {
                    assertThat(context).hasSingleBean(AbstractMyService.class);
                    assertThat(context.getBean(AbstractMyService.class).hello()).containsSequence("V1 Hello");
                    System.out.println(context.getBean(AbstractMyService.class).hello());
                });
        applicationContextRunner
                .withPropertyValues("spring101.version=v2")
                .run(context -> {
                    assertThat(context).hasSingleBean(AbstractMyService.class);
                    assertThat(context.getBean(AbstractMyService.class).hello()).containsSequence("V2 Hello");
                    System.out.println(context.getBean(AbstractMyService.class).hello());
                });

    }
```

```java
    @Test
    public void testConditionalOnMissingBean() {
        applicationContextRunner
                .withUserConfiguration(MyServiceConfig.class)
                .run(context -> {
                    assertThat(context).hasSingleBean(MyService.class);
                    assertThat(context.getBean(MyService.class).hello()).containsSequence("V1 Hi");
                    System.out.println(context.getBean(MyService.class).hello());
                });
    }

}

@Configuration
class MyServiceConfig {
    @Bean
    MyService getService() {
        return new MyService("Hi");
    }
}
```

运行测试可以看到三个测试都可以通过，控制台也输出了 hello 方法的返回值：

实现自定义的配置数据源

接下去我们来看一下如何利用 EnvironmentPostProcessor 来实现一个自定义的配置数据源。我们在 starter 项目中新建一个类，这个类使用了一个 Yaml 配置源加载器，然后我们把加载到的自定义的 PropertySource 加入到 PropertySource 候选列表的第一个，这样就可以实现属性优先从我们定义的（classpath 下的）config.yml 来读取：

```java
package me.josephzhu.spring101customstarter;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.env.EnvironmentPostProcessor;
import org.springframework.boot.env.YamlPropertySourceLoader;
```

```java
import org.springframework.core.env.ConfigurableEnvironment;
import org.springframework.core.env.PropertySource;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

public class MyPropertySourceEnvironmentPostProcessor implements EnvironmentPostProcessor {

    private final YamlPropertySourceLoader loader = new YamlPropertySourceLoader();

    @Override
    public void postProcessEnvironment(ConfigurableEnvironment environment,
                                       SpringApplication application) {
        PropertySource<?> propertySource = loadYaml(new ClassPathResource("config.yml"));
        environment.getPropertySources().addFirst(propertySource);
    }

    private PropertySource<?> loadYaml(Resource path) {
        if (!path.exists()) {
            throw new IllegalArgumentException("Resource " + path + " does not exist");
        }
        try {
            return this.loader.load(path.getFile().getAbsolutePath(), path).get(0);
        }
        catch (Exception ex) {
            throw new IllegalStateException("Failed to load yaml configuration from " + path, ex);
        }
    }
}
```

最关键的一步就是让 SpringBoot 能加载到我们这个 PostProcessor，还是老样子，在 spring,factories 文件中加入一项配置即可：

```
org.springframework.boot.env.EnvironmentPostProcessor=me.josephzhu.spring101customstarter.MyPropertySourceEnvironmentPostProcessor
```

现在，我们可以在 starter 项目下的 resrouces 目录下创建一个 config.yml 来验证一下：

```yaml
spring101:
  name: zhuye_yml
```

重新运行 main 项目可以看到如下的输出结果中包含了 yml 字样：

```
2018-09-30 15:27:05.123  INFO 12769 --- [           main] me.josephzhu.spring101main.Runner1       : V1
Hello >> zhuye_yml:35 !!
```

我们可以为项目添加一下 Actuator 模块进行进一步验证：

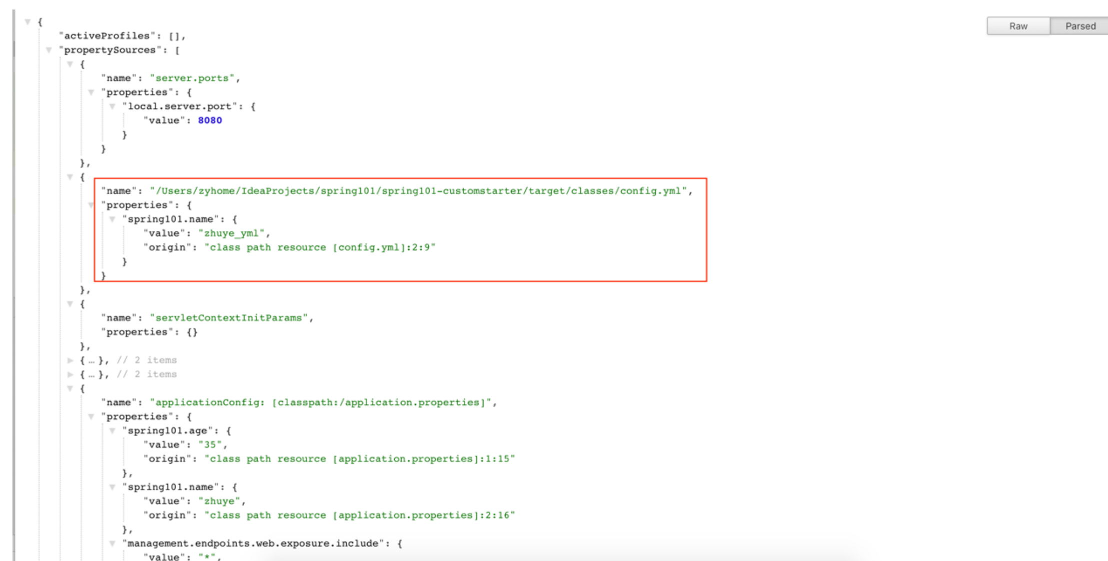
```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
```

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

在配置文件中加入设置来放开所有的端口访问：

```
management.endpoints.web.exposure.include=*
```

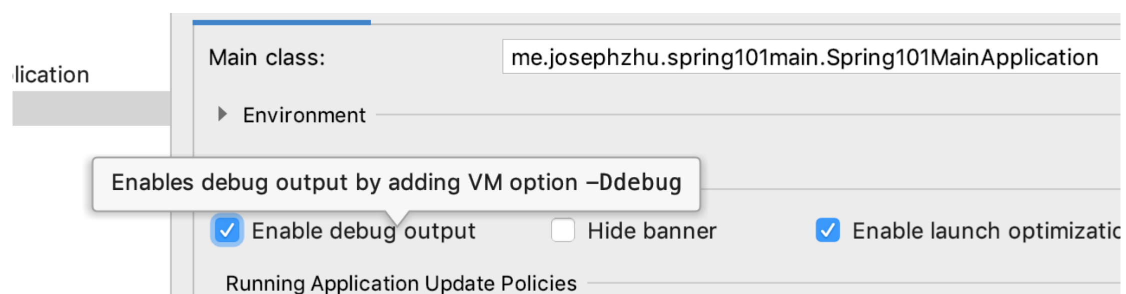
然后打开浏览器访问 http://127.0.0.1:8080/actuator/env：



可以看到，的确是添加了我们自定义的 config.yml 作为 PropertySource。

自动配置的调试问题

对于复杂的项目，如果我们发现自动配置不起作用，要搞清楚框架是如何在各种条件中做自动配置以及自动配置的匹配过程是比较麻烦的事情，这个时候我们可以打开 SpringBoot 的 Debug 来查看日志：



我们可以在日志中搜索我们关注的类型的匹配情况，是不是很直观呢：

MyAutoConfiguration#getMyService matched:

- @ConditionalOnProperty (spring101.version=v1) matched (OnPropertyCondition)

- @ConditionalOnMissingBean (types: me.josephzhu.spring101customstarter.MyService; SearchStrategy: all) did not find any beans (OnBeanCondition)

MyAutoConfiguration#getMyServiceV2:

Did not match:

- @ConditionalOnProperty (spring101.version=v2) did not find property 'version' (OnPropertyCondition)

我们可以试试在出错的时候系统给我们的提示，来把配置文件中的 version 设置为 3：

```
spring101.version=v3
```

重新运行后看到如下输出：

****

APPLICATION FAILED TO START

****

Description:

Field service in me.josephzhu.spring101main.Runner1 required a bean of type 'me.josephzhu.spring101customstarter.AbstractMyService' that could not be found.

- Bean method 'getMyService' in 'MyAutoConfiguration' not loaded because @ConditionalOnProperty (spring101.version=v1) found different value in property 'version'

- Bean method 'getMyServiceV2' in 'MyAutoConfiguration' not loaded because @ConditionalOnProperty (spring101.version=v2) found different value in property 'version'

Action:

Consider revisiting the entries above or defining a bean of type 'me.josephzhu.spring101customstarter.AbstractMyService' in your configuration.

这个所谓的分析报告是比较机械性的，作为框架的开发者，我们甚至可以自定义叫做 FailureAnalyzer 的东西来做更明确的提示。实现上和之前的步骤几乎一样，首先自定义一个类：

```java
package me.josephzhu.spring101customstarter;

import org.springframework.beans.factory.NoSuchBeanDefinitionException;
import org.springframework.boot.diagnostics.AbstractFailureAnalyzer;
import org.springframework.boot.diagnostics.FailureAnalysis;

public class MyFailureAnalyzer extends AbstractFailureAnalyzer<NoSuchBeanDefinitionException> {
    @Override
    protected FailureAnalysis analyze(Throwable rootFailure, NoSuchBeanDefinitionException cause) {
        if(cause.getBeanType().equals(AbstractMyService.class))
            return new FailureAnalysis("加载 MyService 失败", "请检查配置文件中的 version 属性设置是否是 v1 或 v2", rootFailure);

        return null;
    }
}
```

这里我们根据 cause 的 Bean 类型做了简单判断，如果发生错误的是我们的 Service 类型的话，告知使用者明确的错误原因（Description）以及怎么来纠正这个错误（Action）。

然后老规矩，在 spring.factories 中进行关联：

```
org.springframework.boot.diagnostics.FailureAnalyzer=me.josephzhu.spring101customstarter.MyFailureAnalyzer
```

重新运行程序后可以看到如下的结果：

```
***************************

APPLICATION FAILED TO START

***************************

Description:

加载 MyService 失败

Action:

请检查配置文件中的 version 属性设置是否是 v1 或 v2
```

是不是直观很多呢？这里我的实现比较简单，在正式的实现中你可以根据上下文以及环境的各种情况为用户进行全面的分析，分析服务启动失败的原因。

SpringBoot 的扩展点

在之前的几个例子中，我们进行了各种扩展配置，通过 spring.factories 进行了自动配置、环境后处理器配置以及错误分析器配置：

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=me.josephzhu.spring101customstarter.MyAutoConfiguration
org.springframework.boot.env.EnvironmentPostProcessor=me.josephzhu.spring101customstarter.MyPropertySourceEnvironmentPostProcessor
org.springframework.boot.diagnostics.FailureAnalyzer=me.josephzhu.spring101customstarter.MyFailureAnalyzer
```

其实，SpringBoot 还有一些其它的扩展槽，如下是 SpringBoot 自带的一些配置类：

```
# Initializers
org.springframework.context.ApplicationContextInitializer=\
org.springframework.boot.autoconfigure.SharedMetadataReaderFactoryContextInitializer,\
org.springframework.boot.autoconfigure.logging.ConditionEvaluationReportLoggingListener

# Application Listeners
org.springframework.context.ApplicationListener=\
org.springframework.boot.autoconfigure.BackgroundPreinitializer

# Environment Post Processors
org.springframework.boot.env.EnvironmentPostProcessor=\
org.springframework.boot.autoconfigure.security.oauth2.client.OAuth2ClientPropertiesEnvironmentPostProcessor

# Auto Configuration Import Listeners
org.springframework.boot.autoconfigure.AutoConfigurationImportListener=\
org.springframework.boot.autoconfigure.condition.ConditionEvaluationReportAutoConfigurationImportListener

# Auto Configuration Import Filters
org.springframework.boot.autoconfigure.AutoConfigurationImportFilter=\
org.springframework.boot.autoconfigure.condition.OnBeanCondition,\
org.springframework.boot.autoconfigure.condition.OnClassCondition,\
org.springframework.boot.autoconfigure.condition.OnWebApplicationCondition

# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,\
org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration,\
org.springframework.boot.autoconfigure.cloud.CloudServiceConnectorsAutoConfiguration,\
org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration,\
org.springframework.boot.autoconfigure.context.MessageSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.context.PropertyPlaceholderAutoConfiguration,\
```

```
org.springframework.boot.autoconfigure.couchbase.CouchbaseAutoConfiguration,\
org.springframework.boot.autoconfigure.dao.PersistenceExceptionTranslationAutoConfiguration,\
org.springframework.boot.autoconfigure.data.cassandra.CassandraDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.cassandra.CassandraReactiveDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.cassandra.CassandraReactiveRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.cassandra.CassandraRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseReactiveDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseReactiveRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchAutoConfiguration,\
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.jdbc.JdbcRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.jpa.JpaRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.ldap.LdapRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.mongo.MongoDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.mongo.MongoReactiveDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.mongo.MongoReactiveRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.mongo.MongoRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.neo4j.Neo4jDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.neo4j.Neo4jRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.solr.SolrRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.redis.RedisAutoConfiguration,\
org.springframework.boot.autoconfigure.data.redis.RedisReactiveAutoConfiguration,\
org.springframework.boot.autoconfigure.data.redis.RedisRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.rest.RepositoryRestMvcAutoConfiguration,\
org.springframework.boot.autoconfigure.data.web.SpringDataWebAutoConfiguration,\
org.springframework.boot.autoconfigure.elasticsearch.jest.JestAutoConfiguration,\
org.springframework.boot.autoconfigure.elasticsearch.rest.RestClientAutoConfiguration,\
org.springframework.boot.autoconfigure.flyway.FlywayAutoConfiguration,\
org.springframework.boot.autoconfigure.freemarker.FreeMarkerAutoConfiguration,\
org.springframework.boot.autoconfigure.gson.GsonAutoConfiguration,\
org.springframework.boot.autoconfigure.h2.H2ConsoleAutoConfiguration,\
org.springframework.boot.autoconfigure.hateoas.HypermediaAutoConfiguration,\
org.springframework.boot.autoconfigure.hazelcast.HazelcastAutoConfiguration,\
org.springframework.boot.autoconfigure.hazelcast.HazelcastJpaDependencyAutoConfiguration,\
org.springframework.boot.autoconfigure.http.HttpMessageConvertersAutoConfiguration,\
org.springframework.boot.autoconfigure.http.codec.CodecsAutoConfiguration,\
org.springframework.boot.autoconfigure.influx.InfluxDbAutoConfiguration,\
org.springframework.boot.autoconfigure.info.ProjectInfoAutoConfiguration,\
org.springframework.boot.autoconfigure.integration.IntegrationAutoConfiguration,\
org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,\
```

```
org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.JndiDataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.XADataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.JmsAutoConfiguration,\
org.springframework.boot.autoconfigure.jmx.JmxAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.JndiConnectionFactoryAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.activemq.ActiveMQAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.artemis.ArtemisAutoConfiguration,\
org.springframework.boot.autoconfigure.groovy.template.GroovyTemplateAutoConfiguration,\
org.springframework.boot.autoconfigure.jersey.JerseyAutoConfiguration,\
org.springframework.boot.autoconfigure.jooq.JooqAutoConfiguration,\
org.springframework.boot.autoconfigure.jsonb.JsonbAutoConfiguration,\
org.springframework.boot.autoconfigure.kafka.KafkaAutoConfiguration,\
org.springframework.boot.autoconfigure.ldap.embedded.EmbeddedLdapAutoConfiguration,\
org.springframework.boot.autoconfigure.ldap.LdapAutoConfiguration,\
org.springframework.boot.autoconfigure.liquibase.LiquibaseAutoConfiguration,\
org.springframework.boot.autoconfigure.mail.MailSenderAutoConfiguration,\
org.springframework.boot.autoconfigure.mail.MailSenderValidatorAutoConfiguration,\
org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongoAutoConfiguration,\
org.springframework.boot.autoconfigure.mongo.MongoAutoConfiguration,\
org.springframework.boot.autoconfigure.mongo.MongoReactiveAutoConfiguration,\
org.springframework.boot.autoconfigure.mustache.MustacheAutoConfiguration,\
org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration,\
org.springframework.boot.autoconfigure.quartz.QuartzAutoConfiguration,\
org.springframework.boot.autoconfigure.reactor.core.ReactorCoreAutoConfiguration,\
org.springframework.boot.autoconfigure.security.servlet.SecurityAutoConfiguration,\
org.springframework.boot.autoconfigure.security.servlet.SecurityRequestMatcherProviderAutoConfiguration,\
org.springframework.boot.autoconfigure.security.servlet.UserDetailsServiceAutoConfiguration,\
org.springframework.boot.autoconfigure.security.servlet.SecurityFilterAutoConfiguration,\
org.springframework.boot.autoconfigure.security.reactive.ReactiveSecurityAutoConfiguration,\
org.springframework.boot.autoconfigure.security.reactive.ReactiveUserDetailsServiceAutoConfiguration,\
org.springframework.boot.autoconfigure.sendgrid.SendGridAutoConfiguration,\
org.springframework.boot.autoconfigure.session.SessionAutoConfiguration,\
org.springframework.boot.autoconfigure.security.oauth2.client.servlet.OAuth2ClientAutoConfiguration,\
org.springframework.boot.autoconfigure.security.oauth2.client.reactive.ReactiveOAuth2ClientAutoConfiguration,\
org.springframework.boot.autoconfigure.security.oauth2.resource.servlet.OAuth2ResourceServerAutoConfiguration,\
org.springframework.boot.autoconfigure.security.oauth2.resource.reactive.ReactiveOAuth2ResourceServerAutoConf
iguration,\
org.springframework.boot.autoconfigure.solr.SolrAutoConfiguration,\
org.springframework.boot.autoconfigure.task.TaskExecutionAutoConfiguration,\
org.springframework.boot.autoconfigure.task.TaskSchedulingAutoConfiguration,\
org.springframework.boot.autoconfigure.thymeleaf.ThymeleafAutoConfiguration,\
org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration,\
```

```
org.springframework.boot.autoconfigure.transaction.jta.JtaAutoConfiguration,\
org.springframework.boot.autoconfigure.validation.ValidationAutoConfiguration,\
org.springframework.boot.autoconfigure.web.client.RestTemplateAutoConfiguration,\
org.springframework.boot.autoconfigure.web.embedded.EmbeddedWebServerFactoryCustomizerAutoConfiguration,
\
org.springframework.boot.autoconfigure.web.reactive.HttpHandlerAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.ReactiveWebServerFactoryAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.WebFluxAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.error.ErrorWebFluxAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.function.client.ClientHttpConnectorAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.function.client.WebClientAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.DispatcherServletAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.ServletWebServerFactoryAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.error.ErrorMvcAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.HttpEncodingAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.MultipartAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.WebMvcAutoConfiguration,\
org.springframework.boot.autoconfigure.websocket.reactive.WebSocketReactiveAutoConfiguration,\
org.springframework.boot.autoconfigure.websocket.servlet.WebSocketServletAutoConfiguration,\
org.springframework.boot.autoconfigure.websocket.servlet.WebSocketMessagingAutoConfiguration,\
org.springframework.boot.autoconfigure.webservices.WebServicesAutoConfiguration,\
org.springframework.boot.autoconfigure.webservices.client.WebServiceTemplateAutoConfiguration

# Failure analyzers
org.springframework.boot.diagnostics.FailureAnalyzer=\
org.springframework.boot.autoconfigure.diagnostics.analyzer.NoSuchBeanDefinitionFailureAnalyzer,\
org.springframework.boot.autoconfigure.jdbc.DataSourceBeanCreationFailureAnalyzer,\
org.springframework.boot.autoconfigure.jdbc.HikariDriverConfigurationFailureAnalyzer,\
org.springframework.boot.autoconfigure.session.NonUniqueSessionRepositoryFailureAnalyzer

# Template availability providers
org.springframework.boot.autoconfigure.template.TemplateAvailabilityProvider=\
org.springframework.boot.autoconfigure.freemarker.FreeMarkerTemplateAvailabilityProvider,\
org.springframework.boot.autoconfigure.mustache.MustacheTemplateAvailabilityProvider,\
org.springframework.boot.autoconfigure.groovy.template.GroovyTemplateAvailabilityProvider,\
org.springframework.boot.autoconfigure.thymeleaf.ThymeleafTemplateAvailabilityProvider,\
org.springframework.boot.autoconfigure.web.servlet.JspTemplateAvailabilityProvider
```

我们可以看到一共有 8 大类的扩展槽，这些槽贯穿了整个 SpringBoot 加载的整个过程，感兴趣的读者可以逐一搜索查看 Spring 的文档和源码进行进一步的分析。

本文从如何做一个 Starter 实现自动配置开始，进一步阐述了如何实现智能的条件配置，如何，如何进行自动配置的测试，然后我们自定义了环境后处理器来加载额外的配置源（你

当然可以实现更复杂的配置源，比如从 Redis 和数据库中获取配置）以及通过开启 Actutor 来验证，定义了配置错误分析器来给用户明确的错误提示。最后，我们看了一下 spring.factories 中的内容了解了 SpringBoot 内部定义的一些扩展槽和实现。