

SpringBoot原理

在前面十多天的课程当中，我们学习的都是web开发的技术使用，都是面向应用层面的，我们学会了怎么样去用。而我们今天所要学习的是web后端开发的最后一个篇章springboot原理篇，主要偏向于底层原理。

我们今天的课程安排包括这么三个部分：

1. 配置优先级：Springboot项目当中属性配置的常见方式以及配置的优先级
2. Bean的管理
3. 剖析Springboot的底层原理

1. 配置优先级

在我们前面的课程当中，我们已经讲解了SpringBoot项目当中支持的三类配置文件：

- application.properties
- application.yml
- application.yaml

在SpringBoot项目当中，我们要想配置一个属性，可以通过这三种方式当中的任意一种来配置都可以，那么如果项目中同时存在这三种配置文件，且都配置了同一个属性，如：Tomcat端口号，到底哪一份配置文件生效呢？

- application.properties

```
1  server.port=8081
```

- application.yml

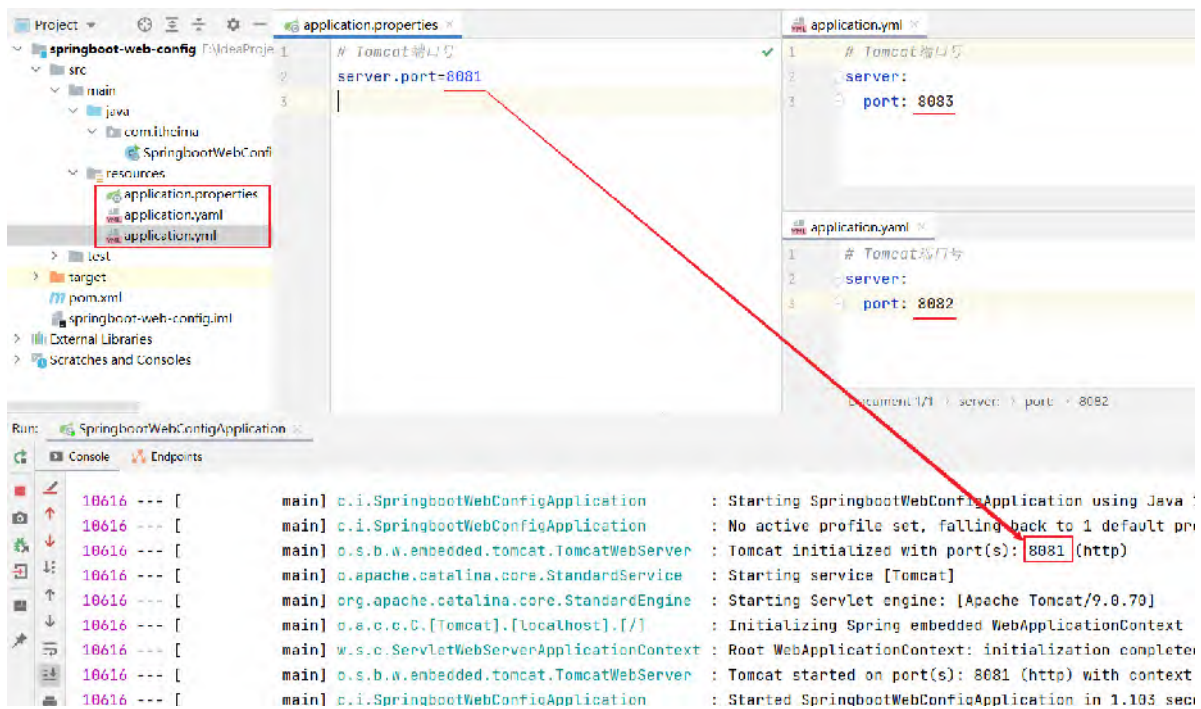
```
1  server:
2    port: 8082
```

- application.yaml

```
1  server:
2    port: 8082
```

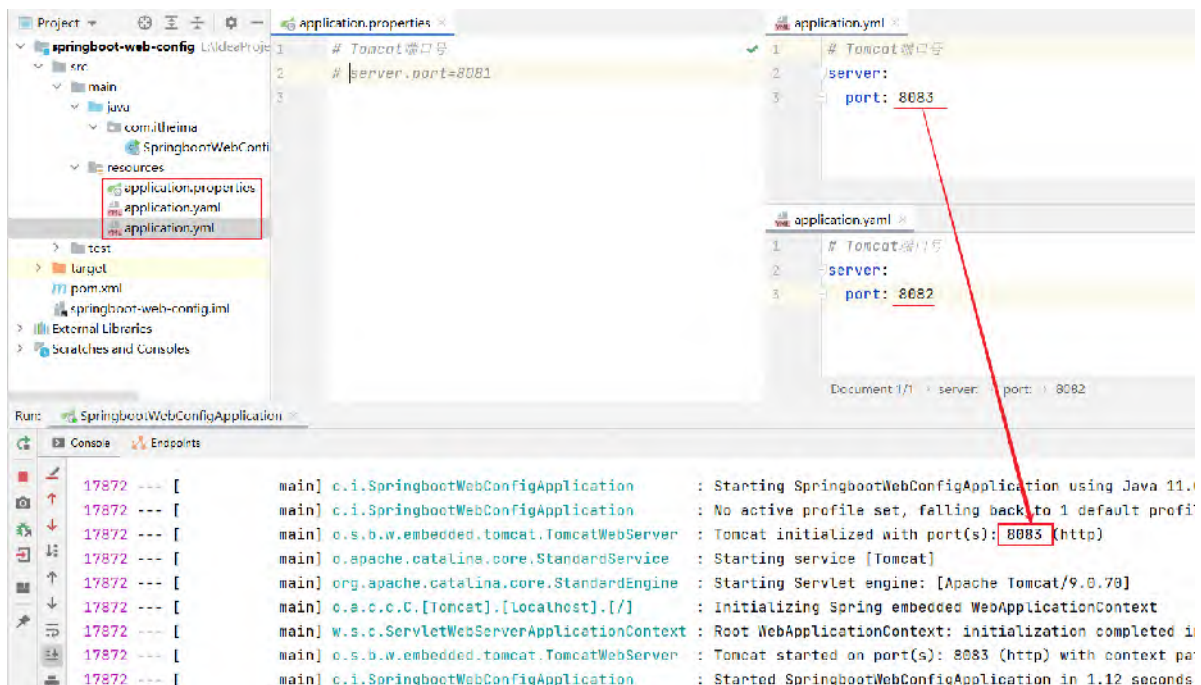
我们启动SpringBoot程序，测试下三个配置文件中哪个Tomcat端口号生效：

- properties、yaml、yml三种配置文件同时存在



properties、yaml、yml三种配置文件，优先级最高的是properties

- yaml、yml两种配置文件同时存在



配置文件优先级排名（从高到低）：

- properties配置文件
- yml配置文件
- yaml配置文件

注意事项：虽然springboot支持多种格式配置文件，但是在项目开发时，推荐统一使用一种格式的配置。（yaml是主流）

在SpringBoot项目当中除了以上3种配置文件外，SpringBoot为了增强程序的扩展性，除了支持配置文件的配置方式以外，还支持另外两种常见的配置方式：

1. Java系统属性配置 （格式： -Dkey=value）

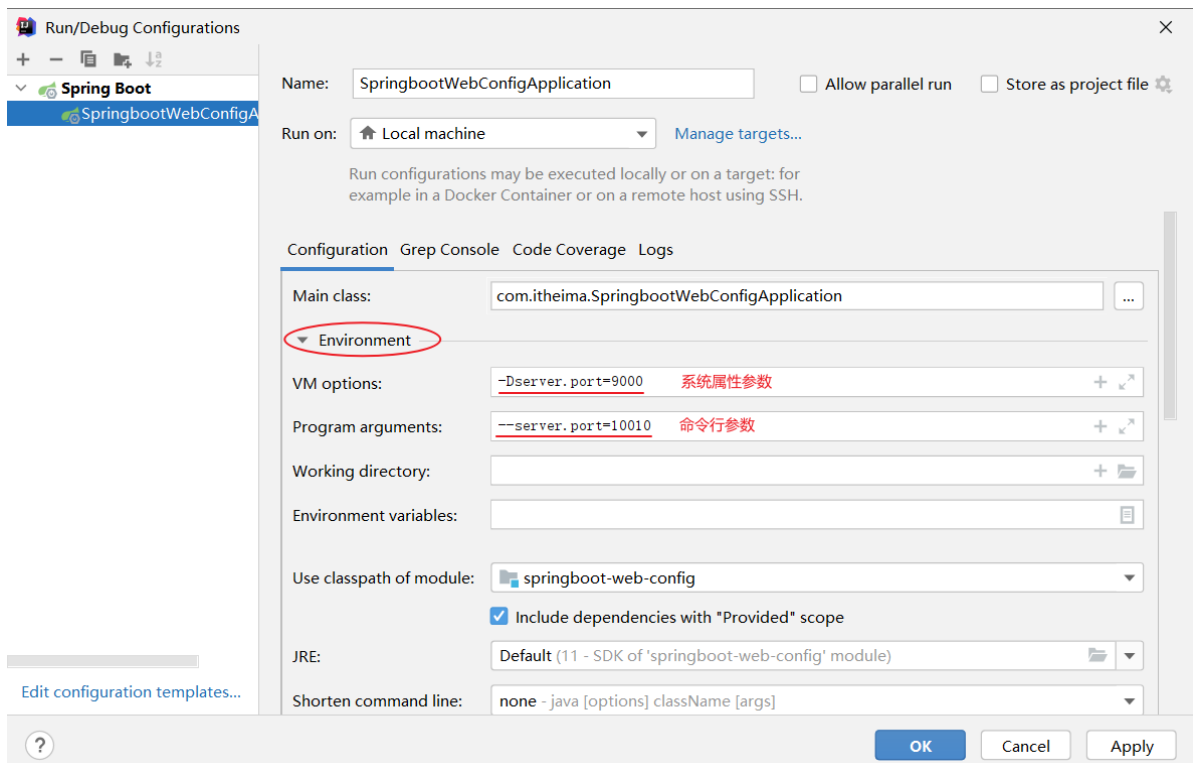
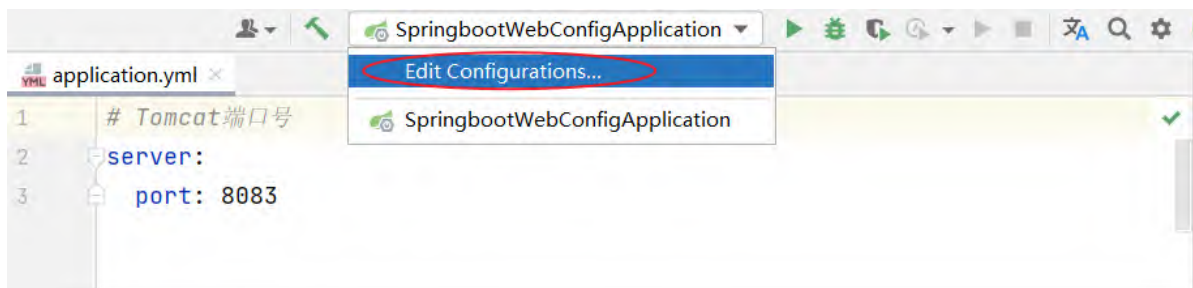
```
1 -Dserver.port=9000
```

2. 命令行参数 （格式： --key=value）

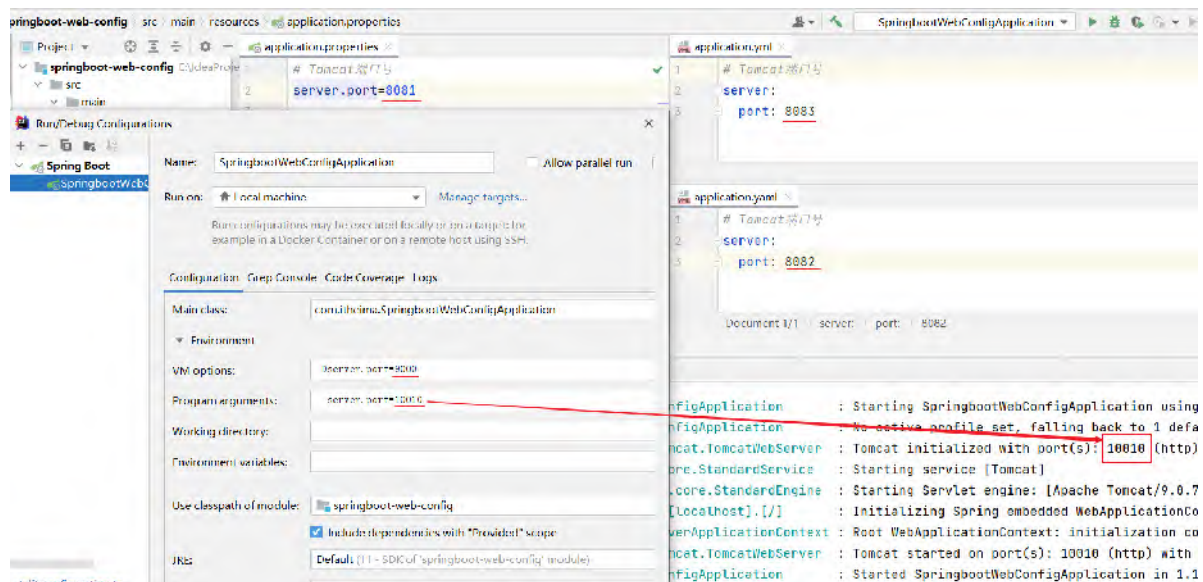
```
1 --server.port=10010
```

那在idea当中运行程序时，如何来指定Java系统属性和命令行参数呢？

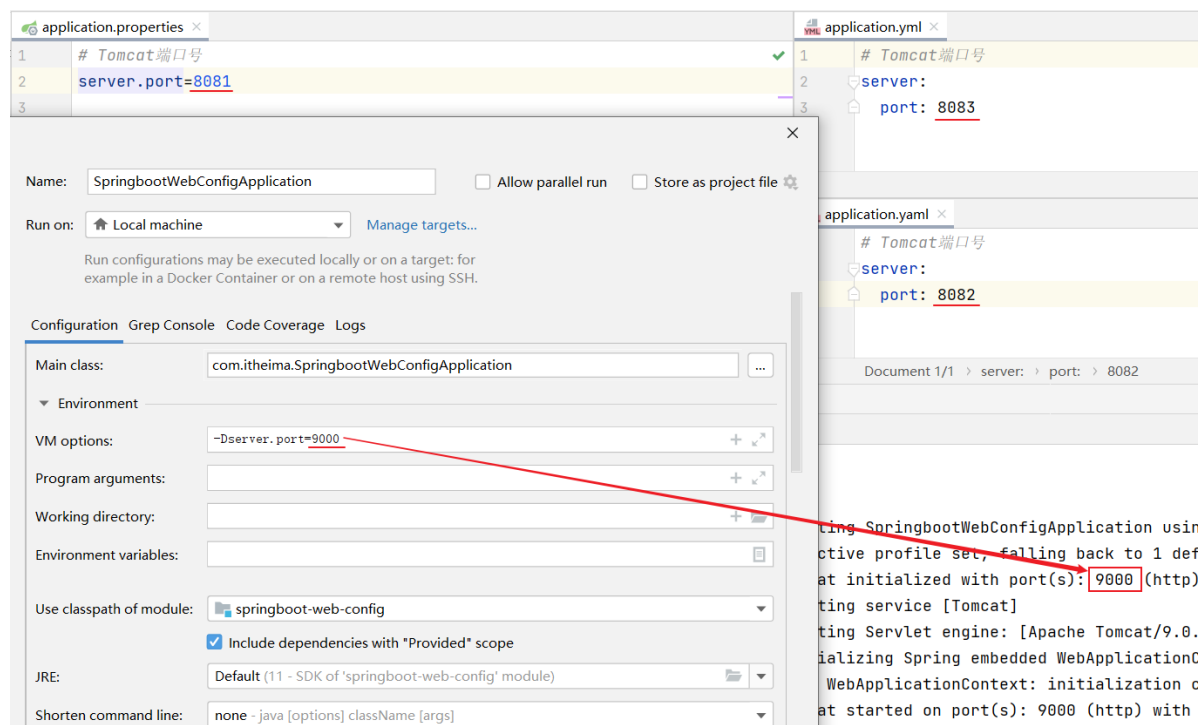
- 编辑启动程序的配置信息



重启服务，同时配置Tomcat端口（三种配置文件、系统属性、命令行参数），测试哪个Tomcat端口号生效：



删除命令行参数配置，重启SpringBoot服务：



优先级： 命令行参数 > 系统属性参数 > properties参数 > yml参数 > yaml参数

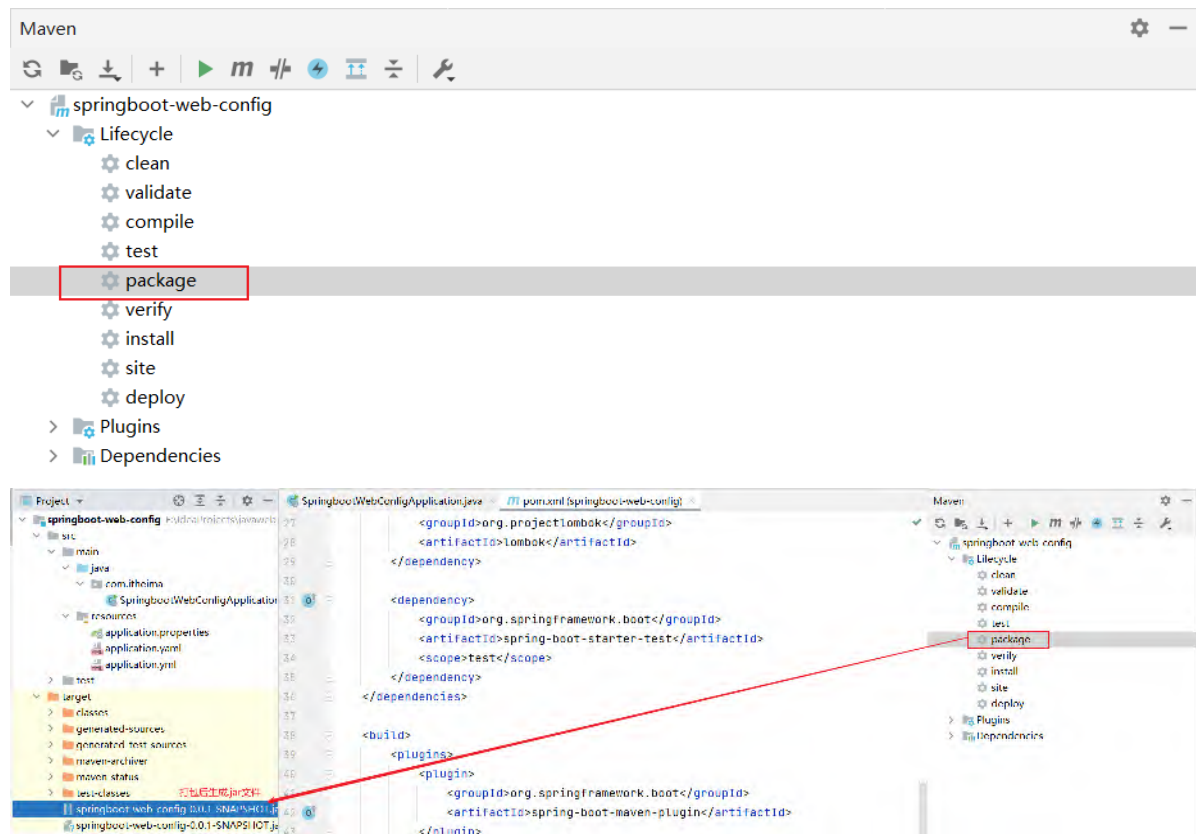
思考：如果项目已经打包上线了，这个时候我们又如何来设置Java系统属性和命令行参数呢？

```
1 java -Dserver.port=9000 -jar XXXXX.jar --server.port=10010
```

下面我们来演示下打包程序运行时指定Java系统属性和命令行参数：

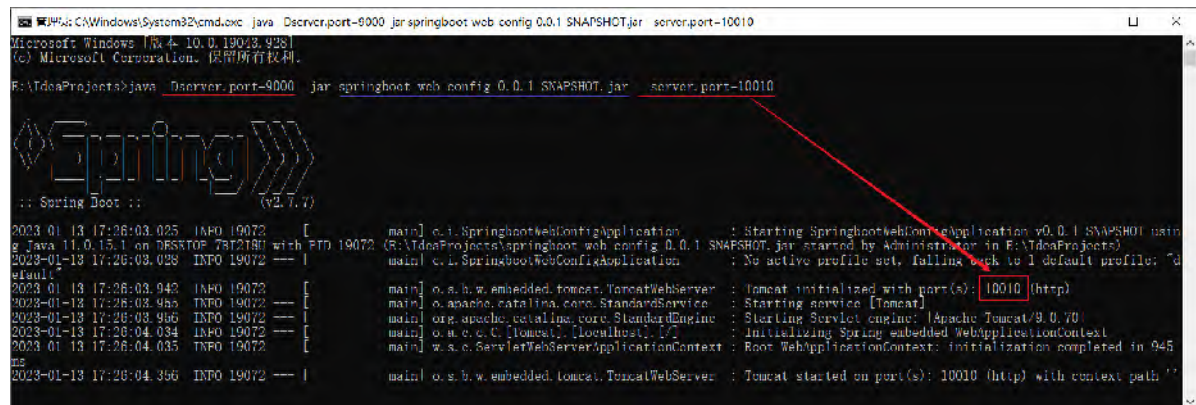
1. 执行maven打包指令package，把项目打成jar文件
2. 使用命令：java -jar 方式运行jar文件程序

项目打包:

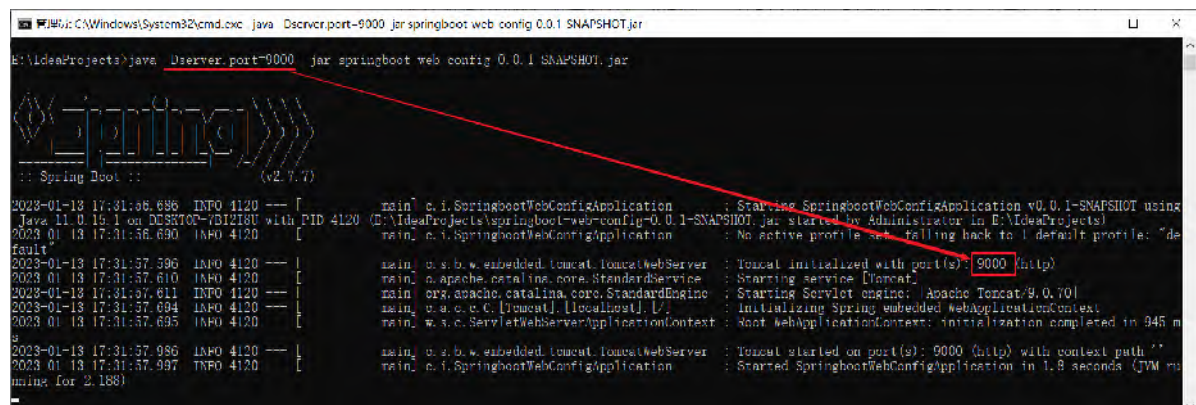


运行jar程序:

- 同时设置Java系统属性和命令行参数



- 仅设置Java系统属性



注意事项:

- Springboot项目进行打包时，需要引入插件 `spring-boot-maven-plugin`（基于官网骨架创建项目，会自动添加该插件）

在SpringBoot项目当中，常见的属性配置方式有5种，3种配置文件，加上2种外部属性的配置（Java系统属性、命令行参数）。通过以上的测试，我们也得出了优先级（从低到高）：

- `application.yaml`（忽略）
- `application.yml`
- `application.properties`
- java系统属性（`-Dxxx=xxx`）
- 命令行参数（`--xxx=xxx`）

2. Bean管理

在前面的课程当中，我们已经讲过了我们可以通过Spring当中提供的注解`@Component`以及它的三个衍生注解（`@Controller`、`@Service`、`@Repository`）来声明IOC容器中的bean对象，同时我们也学习了如何为应用程序注入运行时所需要依赖的bean对象，也就是依赖注入DI。

我们今天主要学习IOC容器中Bean的其他使用细节，主要学习以下三方面：

1. 如何从IOC容器中手动的获取到bean对象
2. bean的作用域配置
3. 管理第三方的bean对象

接下来我们先来学习第一方面，从IOC容器中获取bean对象。

2.1 获取Bean

默认情况下，SpringBoot项目在启动的时候会自动的创建IOC容器（也称为Spring容器），并且在启动的过程当中会自动的将bean对象都创建好，存放在IOC容器当中。应用程序在运行时需要依赖什么bean对象，就直接进行依赖注入就可以了。

而在Spring容器中提供了一些方法，可以主动从IOC容器中获取到bean对象，下面介绍3种常用方式：

1. 根据name获取bean

```
1 Object getBean(String name)
```

2. 根据类型获取bean

```
1 <T> T getBean(Class<T> requiredType)
```

3. 根据name获取bean (带类型转换)

```
1 <T> T getBean(String name, Class<T> requiredType)
```

思考：要从IOC容器当中来获取到bean对象，需要先拿到IOC容器对象，怎样才能拿到IOC容器呢？

- 想获取到IOC容器，直接将IOC容器对象注入进来就可以了

控制器：DeptController

```
1 @RestController
2 @RequestMapping("/depts")
3 public class DeptController {
4
5     @Autowired
6     private DeptService deptService;
7
8     public DeptController() {
9         System.out.println("DeptController constructor ....");
10    }
11
12    @GetMapping
13    public Result list() {
14        List<Dept> deptList = deptService.list();
15        return Result.success(deptList);
16    }
17
18    @DeleteMapping("/{id}")
19    public Result delete(@PathVariable Integer id) {
20        deptService.delete(id);
21        return Result.success();
22    }
23
24    @PostMapping
25    public Result save(@RequestBody Dept dept) {
26        deptService.save(dept);
27        return Result.success();
28    }
29 }
```

业务实现类：DeptServiceImpl

```

1  @Slf4j
2  @Service
3  public class DeptServiceImpl implements DeptService {
4      @Autowired
5      private DeptMapper deptMapper;
6
7      @Override
8      public List<Dept> list() {
9          List<Dept> deptList = deptMapper.list();
10         return deptList;
11     }
12
13     @Override
14     public void delete(Integer id) {
15         deptMapper.delete(id);
16     }
17
18     @Override
19     public void save(Dept dept) {
20         dept.setCreateTime(LocalDateTime.now());
21         dept.setUpdateTime(LocalDateTime.now());
22         deptMapper.save(dept);
23     }
24 }

```

Mapper接口:

```

1  @Mapper
2  public interface DeptMapper {
3      //查询全部部门数据
4      @Select("select * from dept")
5      List<Dept> list();
6
7      //删除部门
8      @Delete("delete from dept where id = #{id}")
9      void delete(Integer id);
10
11     //新增部门
12     @Insert("insert into dept(name, create_time, update_time) values
13         (#{name},#{createTime},#{updateTime})")
14     void save(Dept dept);
15 }

```


测试类：

```
1  @SpringBootTest
2  class SpringbootWebConfig2ApplicationTests {
3
4      @Autowired
5      private ApplicationContext applicationContext; //IOC容器对象
6
7      //获取bean对象
8      @Test
9      public void testGetBean() {
10         //根据bean的名称获取
11         DeptController bean1 = (DeptController)
12         applicationContext.getBean("deptController");
13         System.out.println(bean1);
14
15         //根据bean的类型获取
16         DeptController bean2 =
17         applicationContext.getBean(DeptController.class);
18         System.out.println(bean2);
19
20         //根据bean的名称 及 类型获取
21         DeptController bean3 =
22         applicationContext.getBean("deptController", DeptController.class);
23         System.out.println(bean3);
24     }
25 }
```

程序运行后控制台日志：

```
com.itheima.controller.DeptController@5ce3409b
com.itheima.controller.DeptController@5ce3409b
com.itheima.controller.DeptController@5ce3409b
```

问题：输出的bean对象地址值是一样的，说明IOC容器当中的bean对象有几个？

答案：只有一个。 （默认情况下，IOC中的bean对象是单例）

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）？

可以，在下一个知识点（bean作用域）中讲解。

注意事项：

- 上述所说的 【Spring项目启动时，会把其中的bean都创建好】还会受到作用域及延迟初始化影响，这里主要针对于默认的单例非延迟加载的bean而言。

2.2 Bean作用域

在前面我们提到的IOC容器当中，默认bean对象是单例模式（只有一个实例对象）。那么如何设置bean对象为非单例呢？需要设置bean的作用域。

在Spring中支持五种作用域，后三种在web环境才生效：

作用域	说明
singleton	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	每个请求范围内会创建新的实例（web环境中，了解）
session	每个会话范围内会创建新的实例（web环境中，了解）
application	每个应用范围内会创建新的实例（web环境中，了解）

知道了bean的5种作用域了，我们要怎么去设置一个bean的作用域呢？

- 可以借助Spring中的@Scope注解来进行配置作用域

```
@Scope("prototype")
@RestController
@RequestMapping("/depts")
public class DeptController {
}
```

1). 测试一

- 控制器

```
1 //默认bean的作用域为： singleton （单例）
2 @Lazy //延迟加载（第一次使用bean对象时，才会创建bean对象并交给ioc容器管理）
3 @RestController
4 @RequestMapping("/depts")
5 public class DeptController {
6
7     @Autowired
8     private DeptService deptService;
9 }
```

```

10     public DeptController() {
11         System.out.println("DeptController constructor ....");
12     }
13
14     //省略其他代码...
15 }

```

- 测试类

```

1  @SpringBootTest
2  class SpringbootWebConfig2ApplicationTests {
3
4      @Autowired
5      private ApplicationContext applicationContext; //IOC容器对象
6
7      //bean的作用域
8      @Test
9      public void testScope() {
10         for (int i = 0; i < 10; i++) {
11             DeptController deptController =
12                 applicationContext.getBean(DeptController.class);
13             System.out.println(deptController);
14         }
15     }
16 }

```

重启SpringBoot服务，运行测试方法，查看控制台打印的日志：

```

DeptController constructor ....
com.itheima.controller.DeptController@74b1838
com.itheima.controller.DeptController@74b1838
com.itheima.controller.DeptController@74b1838
com.itheima.controller.DeptController@74b1838
com.itheima.controller.DeptController@74b1838
com.itheima.controller.DeptController@74b1838
com.itheima.controller.DeptController@74b1838
com.itheima.controller.DeptController@74b1838
com.itheima.controller.DeptController@74b1838
com.itheima.controller.DeptController@74b1838
com.itheima.controller.DeptController@74b1838

```

注意事项：

- IOC容器中的bean默认使用的作用域：singleton（单例）
- 默认singleton的bean，在容器启动时被创建，可以使用@Lazy注解来延迟初始化（延迟到第一次使用时）

2). 测试二

修改控制器DeptController代码:

```
1  @Scope("prototype") //bean作用域为非单例
2  @Lazy //延迟加载
3  @RestController
4  @RequestMapping("/depts")
5  public class DeptController {
6
7      @Autowired
8      private DeptService deptService;
9
10     public DeptController(){
11         System.out.println("DeptController constructor ....");
12     }
13
14     //省略其他代码...
15 }
```

重启SpringBoot服务, 再次执行测试方法, 查看控制台打印的日志:

```
DeptController constructor ....
com.itheima.controller.DeptController@4d0e1a9a
DeptController constructor ....
com.itheima.controller.DeptController@25218a4d
DeptController constructor ....
com.itheima.controller.DeptController@bf2aa32
DeptController constructor ....
com.itheima.controller.DeptController@56da96b3
DeptController constructor ....
com.itheima.controller.DeptController@6b3d9c38
DeptController constructor ....
com.itheima.controller.DeptController@426710f0
DeptController constructor ....
com.itheima.controller.DeptController@5c5a91b4
DeptController constructor ....
com.itheima.controller.DeptController@5e37fb82
DeptController constructor ....
com.itheima.controller.DeptController@59ec7020
DeptController constructor ....
com.itheima.controller.DeptController@23f60b7d
```

注意事项:

- prototype的bean, 每一次使用该bean的时候都会创建一个新的实例
- 实际开发当中, 绝大部分的Bean是单例的, 也就是说绝大部分Bean不需要配置scope属性

2.3 第三方Bean

学习完bean的获取、bean的作用域之后，接下来我们再来学习第三方bean的配置。

之前我们所配置的bean，像controller、service、dao三层体系下编写的类，这些类都是我们在项目当中自己定义的类(自定义类)。当我们要声明这些bean，也非常简单，我们只需要在类上加上@Component以及它的这三个衍生注解(@Controller、@Service、@Repository)，就可以来声明这个bean对象了。

但是在我们项目开发当中，还有一种情况就是这个类它不是我们自己编写的，而是我们引入的第三方依赖当中提供的。

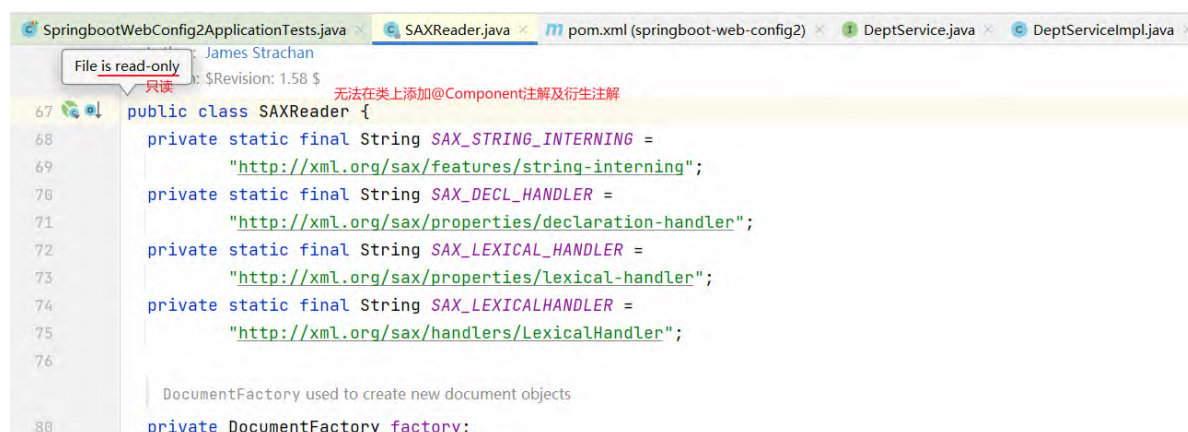
在pom.xml文件中，引入dom4j：

```
1 <!--Dom4j-->
2 <dependency>
3     <groupId>org.dom4j</groupId>
4     <artifactId>dom4j</artifactId>
5     <version>2.1.3</version>
6 </dependency>
```

dom4j就是第三方组织提供的。 dom4j中的SAXReader类就是第三方编写的。

当我们需要使用到SAXReader对象时，直接进行依赖注入是不是就可以了呢？

- 按照我们之前的做法，需要在SAXReader类上添加一个注解@Component（将当前类交给IOC容器管理）



结论：第三方提供的类是只读的。无法在第三方类上添加@Component注解或衍生注解。

那么我们应该怎样使用并定义第三方的bean呢？

- 如果要管理的bean对象来自于第三方（不是自定义的），是无法用@Component 及衍生注解声明bean的，就需要用到@Bean注解。

解决方案1：在启动类上添加@Bean标识的方法

```
1  @SpringBootApplication
2  public class SpringbootWebConfig2Application {
3
4      public static void main(String[] args) {
5          SpringApplication.run(SpringbootWebConfig2Application.class,
6              args);
7      }
8
9      //声明第三方bean
10     @Bean //将当前方法的返回值对象交给IOC容器管理，成为IOC容器bean
11     public SAXReader saxReader() {
12         return new SAXReader();
13     }
14 }
```

xml文件：

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <emp>
3      <name>Tom</name>
4      <age>18</age>
5  </emp>
6
```

测试类：

```
1  @SpringBootTest
2  class SpringbootWebConfig2ApplicationTests {
3
4      @Autowired
5      private SAXReader saxReader;
6
7      //第三方bean的管理
8      @Test
9      public void testThirdBean() throws Exception {
10         Document document =
11             saxReader.read(this.getClass().getClassLoader().getResource("1.xml"));
12     }
```

```

11         Element rootElement = document.getRootElement();
12         String name = rootElement.element("name").getText();
13         String age = rootElement.element("age").getText();
14
15         System.out.println(name + " : " + age);
16     }
17
18     //省略其他代码...
19 }
20

```

重启SpringBoot服务，执行测试方法后，控制台输出日志：

```

1    Tom : 18

```

说明：以上在启动类中声明第三方Bean的作法，不建议使用（项目中要保证启动类的纯粹性）

解决方案2：在配置类中定义@Bean标识的方法

- 如果需要定义第三方Bean时，通常会单独定义一个配置类

```

1    @Configuration //配置类 （在配置类当中对第三方bean进行集中的配置管理）
2    public class CommonConfig {
3
4        //声明第三方bean
5        @Bean //将当前方法的返回值对象交给IOC容器管理，成为IOC容器bean
6            //通过@Bean注解的name/value属性指定bean名称，如果未指定，默认
            是方法名
7        public SAXReader reader(DepartmentService deptService){
8            System.out.println(deptService);
9            return new SAXReader();
10        }
11
12    }
13

```

注释掉SpringBoot启动类中创建第三方bean对象的代码，重启服务，执行测试方法，查看控制台日志：

```

1    Tom : 18

```

在方法上加上一个@Bean注解，Spring 容器在启动的时候，它会自动的调用这个方法，并将方法的返回值声明为Spring容器当中的Bean对象。

注意事项：

- 通过@Bean注解的name或value属性可以声明bean的名称，如果不指定，默认bean的名称就是方法名。
- 如果第三方bean需要依赖其它bean对象，直接在bean定义方法中设置形参即可，容器会根据类型自动装配。

关于Bean大家只需要保持一个原则：

- 如果是在项目当中我们自己定义的类，想将这些类交给IOC容器管理，我们直接使用@Component以及它的衍生注解来声明就可以。
- 如果这个类它不是我们自己定义的，而是引入的第三方依赖当中提供的类，而且我们还想将这个类交给IOC容器管理。此时我们就需要在配置类中定义一个方法，在方法上加上一个@Bean注解，通过这种方式来声明第三方的bean对象。

3. SpringBoot原理

经过前面10多天课程的学习，大家也会发现基于SpringBoot进行web程序的开发是非常简单、非常高效的。

SpringBoot使我们能够集中精力地去关注业务功能的开发，而不用过多地关注框架本身的配置使用。而我们前面所讲解的都是面向应用层面的技术，接下来我们开始学习SpringBoot的原理，这部分内容偏向于底层的原理分析。

在剖析SpringBoot的原理之前，我们先来快速回顾一下我们前面所讲解的Spring家族的框架。



Spring Framework

Provides core support for dependency injection, transaction management, web apps, data access, messaging, and more.

Spring是目前世界上最流行的Java框架，它可以帮助我们更加快速、更加容易的来构建Java项目。而在Spring家族当中提供了很多优秀的框架，而所有的框架都是基于一个基础框架的SpringFramework(也就是Spring框架)。而前面我们也提到，如果我们直接基于Spring框架进行项目的开发，会比较繁琐。

这个繁琐主要体现在两个地方：





1. 在pom.xml中依赖配置比较繁琐，在项目开发时，需要自己去找到对应的依赖，还需要找到依赖它所配套的依赖以及对应版本，否则就会出现版本冲突问题。
2. 在使用Spring框架进行项目开发时，需要在Spring的配置文件中做大量的配置，这就造成Spring框架入门难度较大，学习成本较高。

 <h3>Spring Framework</h3> <p>Provides core support for dependency injection, transaction management, web apps, data access, messaging, and more.</p> <p>繁琐(依赖、配置)</p>	 <h3>Spring Boot</h3> <p>Takes an opinionated view of building Spring applications and gets you up and running as quickly as possible.</p> <p>简单、快捷</p>
--	---

基于Spring存在的问题，官方在Spring框架4.0版本之后，又推出了一个全新的框架：SpringBoot。

通过 SpringBoot来简化Spring框架的开发(是简化不是替代)。我们直接基于SpringBoot来构建Java项目，会让我们的项目开发更加简单，更加快捷。

SpringBoot框架之所以使用起来更简单更快捷，是因为SpringBoot框架底层提供了两个非常重要的功能：一个是起步依赖，一个是自动配置。

 <h3>Spring Boot</h3> <p>Takes an opinionated view of building Spring applications and gets you up and running as quickly as possible.</p>	 <h3>Spring Boot</h3> <p>Takes an opinionated view of building Spring applications and gets you up and running as quickly as possible.</p>
 起步依赖	 自动配置

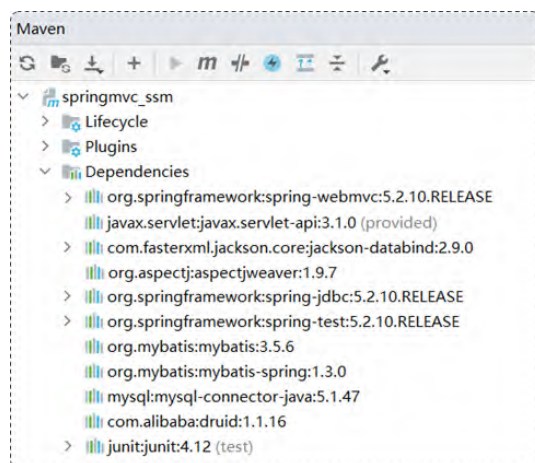
通过SpringBoot所提供的起步依赖，就可以大大的简化pom文件当中依赖的配置，从而解决了Spring框架当中依赖配置繁琐的问题。

通过自动配置的功能就可以大大的简化框架在使用时bean的声明以及bean的配置。我们只需要引入程序开发时所需要的起步依赖，项目开发时所用到的常见的配置都已经有了，我们直接使用就可以了。

简单回顾之后，接下来我们来学习下SpringBoot的原理。其实学习SpringBoot的原理就是来解析SpringBoot当中的起步依赖与自动配置的原理。我们首先来学习SpringBoot当中起步依赖的原理。

3.1 起步依赖

假如我们没有使用SpringBoot，用的是Spring框架进行web程序的开发，此时我们就需要引入web程序开发所需要的一些依赖。



spring-webmvc依赖：这是Spring框架进行web程序开发所需要的依赖

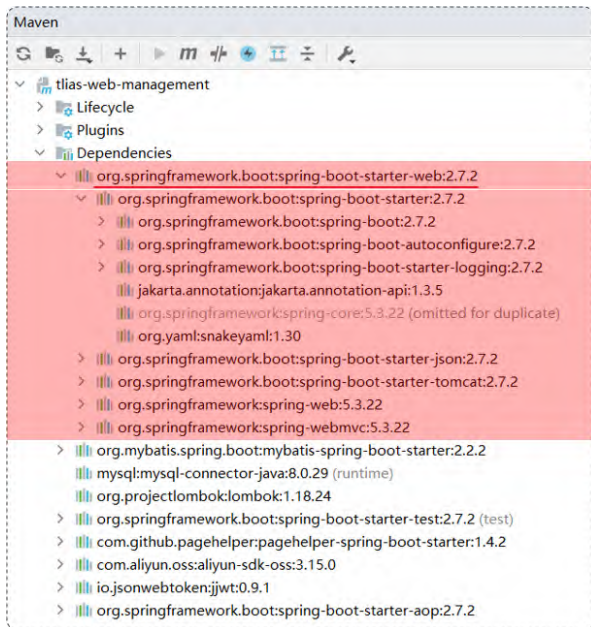
servlet-api依赖：Servlet基础依赖

jackson-databind依赖：JSON处理工具包

如果要使用AOP，还需要引入aop依赖、aspect依赖

项目中所引入的这些依赖，还需要保证版本匹配，否则就可能会出现版本冲突问题。

如果我们使用了SpringBoot，就不需要像上面这么繁琐的引入依赖了。我们只需要引入一个依赖就可以了，那就是web开发的起步依赖：springboot-starter-web。



为什么我们只需要引入一个web开发的起步依赖，web开发所需要的所有的依赖都有了昵？

- 因为Maven的依赖传递。

- 在SpringBoot给我们提供的这些起步依赖当中，已提供了当前程序开发所需要的所有的常见依赖（官网地址：<https://docs.spring.io/spring-boot/docs/2.7.7/reference/htmlsingle/#using.build-systems.starters>）。
- 比如：springboot-starter-web，这是web开发的起步依赖，在web开发的起步依赖当中，就集成了web开发中常见的依赖：json、web、webmvc、tomcat等。我们只需要引入这一个起步依赖，其他的依赖都会自动的通过Maven的依赖传递进来。

结论：起步依赖的原理就是Maven的依赖传递。

3.2 自动配置

我们讲解了SpringBoot当中起步依赖的原理，就是Maven的依赖传递。接下来我们解析下自动配置的原理，我们要分析自动配置的原理，首先要知道什么是自动配置。

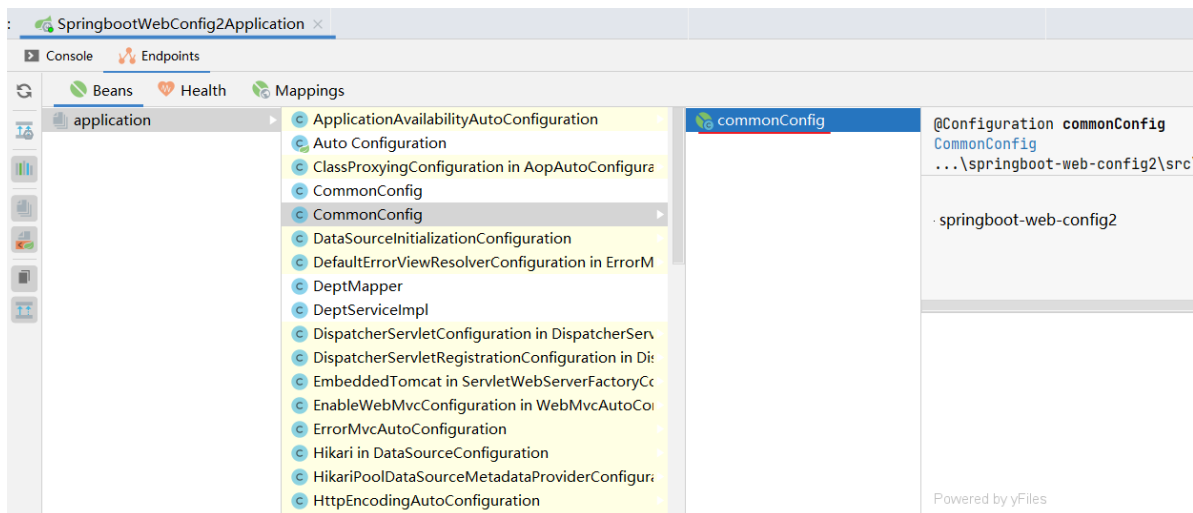
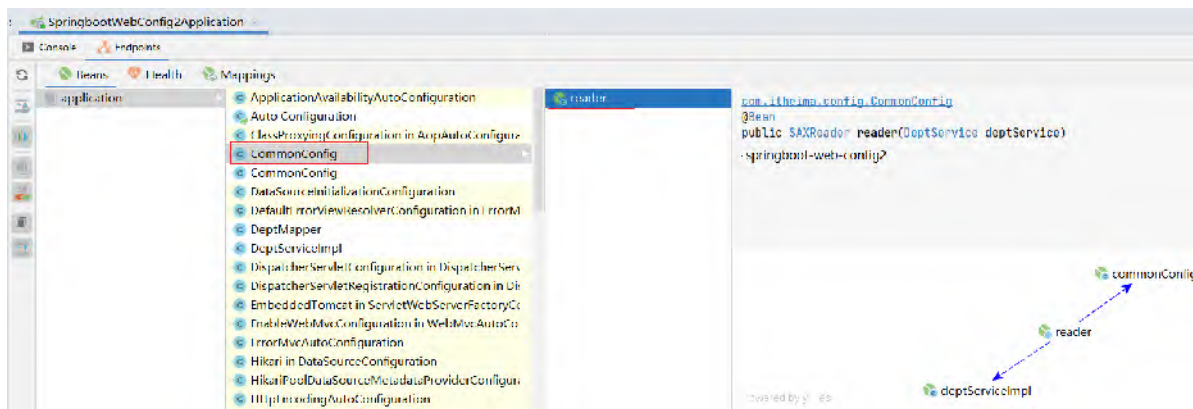
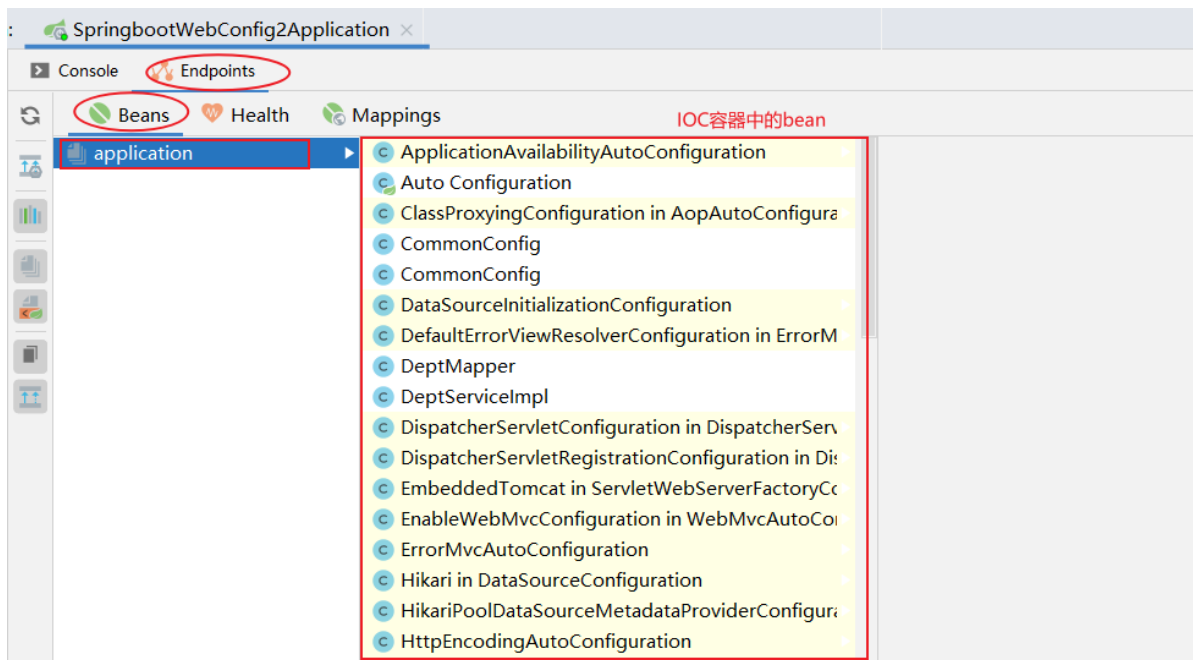
3.2.1 概述

SpringBoot的自动配置就是当Spring容器启动后，一些配置类、bean对象就自动存入到了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。

比如：我们要进行事务管理、要进行AOP程序的开发，此时就不需要我们去手动声明这些bean对象了，我们直接使用就可以从而大大的简化程序的开发，省去了繁琐的配置操作。

下面我们打开idea，一起来看下自动配置的效果：

- 运行SpringBoot启动类



大家会看到有两个CommonConfig，在第一个CommonConfig类中定义了一个bean对象，bean对象的名字叫reader。

在第二个CommonConfig中它的bean名字叫commonConfig，为什么还会有这样一个bean对象呢？原因是在CommonConfig配置类上添加了一个注解@Configuration，而@Configuration底层就是@Component

```

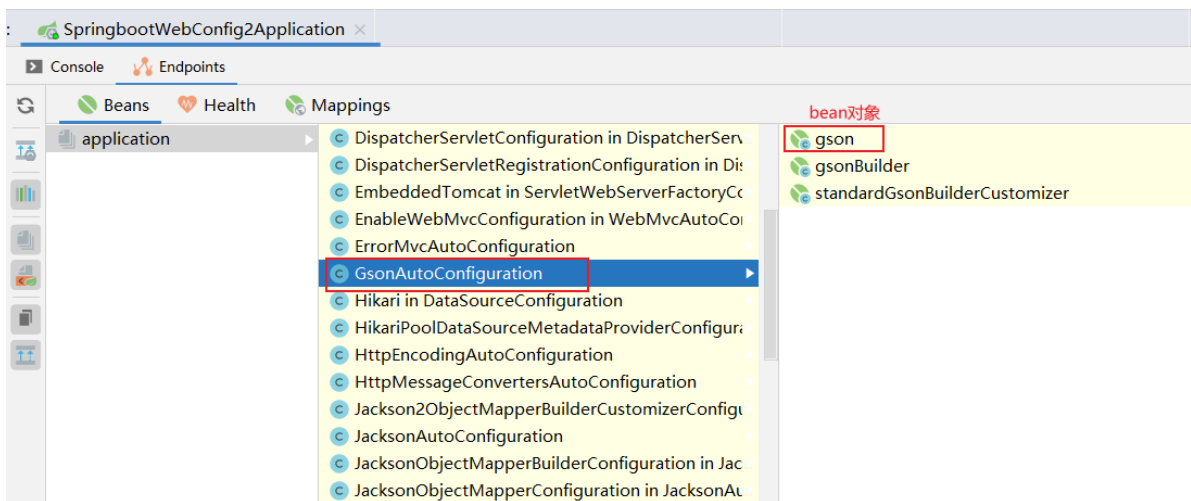
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Configuration {
    @AliasFor(
        annotation = Component.class
    )
    String value() default "";

    boolean proxyBeanMethods() default true;
}

```

所以配置类最终也是SpringIOC容器其中的一个bean对象

在IOC容器中除了我们自己定义的bean以外，还有很多配置类，这些配置类都是SpringBoot在启动的时候加载进来的配置类。这些配置类加载进来之后，它也会生成很多的bean对象。



比如：配置类GsonAutoConfiguration里面有一个bean，bean的名字叫gson，它的类型是Gson。

com.google.gson.Gson是谷歌包中提供的用来处理JSON格式数据的。

当我们想要使用这些配置类中生成的bean对象时，可以使用@Autowired就自动注入了：

```

1  import com.google.gson.Gson;
2  import com.itheima.pojo.Result;
3  import org.junit.jupiter.api.Test;
4  import org.springframework.beans.factory.annotation.Autowired;
5  import org.springframework.boot.test.context.SpringBootTest;
6
7  @SpringBootTest
8  public class AutoConfigurationTests {
9

```

```

10     @Autowired
11     private Gson gson;
12
13
14     @Test
15     public void testJson() {
16         String json = gson.toJson(Result.success());
17         System.out.println(json);
18     }
19 }

```

添加断点，使用debug模式运行测试类程序：



问题：在当前项目中我们并没有声明谷歌提供的Gson这么一个bean对象，然后我们却可以通过 @Autowired从Spring容器中注入bean对象，那么这个bean对象怎么来的？

答案：SpringBoot项目在启动时通过自动配置完成了bean对象的创建。

体验了SpringBoot的自动配置了，下面我们就来分析自动配置的原理。其实分析自动配置原理就是来解析在SpringBoot项目中，在引入依赖之后是如何将依赖jar包当中所定义的配置类以及bean加载到SpringIOC容器中的。

3.2.2 常见方案

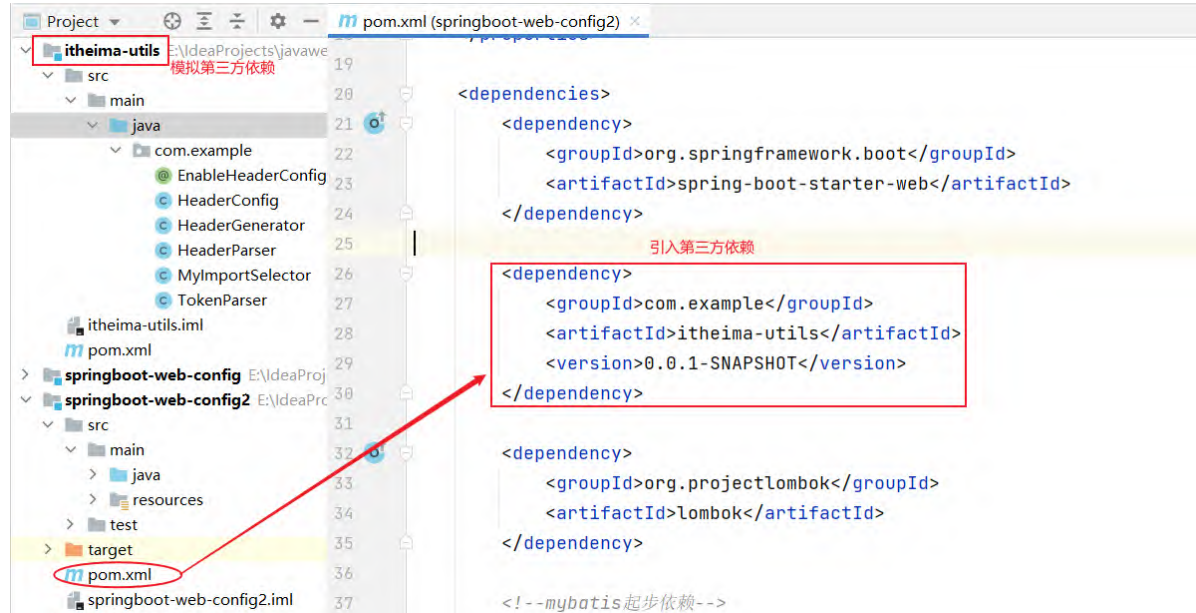
3.2.2.1 概述

我们知道了什么是自动配置之后，接下来我们就要来剖析自动配置的原理。解析自动配置的原理就是分析在 SpringBoot项目当中，我们引入对应的依赖之后，是如何将依赖jar包当中所提供的bean以及配置类直接加载到当前项目的SpringIOC容器当中的。

接下来，我们就直接通过代码来分析自动配置原理。

准备工作：在Idea中导入"资料\03. 自动配置原理"下的itheima-utils工程

1、在SpringBoot项目 spring-boot-web-config2 工程中，通过坐标引入itheima-utils依赖



```
1  @Component
2  public class TokenParser {
3      public void parse() {
4          System.out.println("TokenParser ... parse ...");
5      }
6  }
```

2、在测试类中，添加测试方法

```
1  @SpringBootTest
2  public class AutoConfigurationTests {
3
4      @Autowired
5      private ApplicationContext applicationContext;
6
7
8      @Test
9      public void testTokenParse() {
10
11          System.out.println(applicationContext.getBean(TokenParser.class));
12      }
13  }
```



```
13 //省略其他代码...
14 }
```

3、执行测试方法



异常信息描述： 没有com.example.TokenParse类型的bean

说明：在Spring容器中没有找到com.example.TokenParse类型的bean对象

思考：引入进来的第三方依赖当中的bean以及配置类为什么没有生效？

- 原因在我们之前讲解IOC的时候有提到过，在类上添加@Component注解来声明bean对象时，还需要保证@Component注解能被Spring的组件扫描到。
- SpringBoot项目中的@SpringBootApplication注解，具有包扫描的作用，但是它只会扫描启动类所在的当前包以及子包。
- 当前包：com.itheima， 第三方依赖中提供的包：com.example（扫描不到）

那么如何解决以上问题的呢？

- 方案1：@ComponentScan 组件扫描
- 方案2：@Import 导入（使用@Import导入的类会被Spring加载到IOC容器中）

3.2.2.2 方案一

@ComponentScan组件扫描

```
1  @SpringBootApplication
2  @ComponentScan({"com.itheima","com.example"}) //指定要扫描的包
3  public class SpringbootWebConfig2Application {
4      public static void main(String[] args) {
5          SpringApplication.run(SpringbootWebConfig2Application.class,
6              args);
7      }
8  }
```

重新执行测试方法，控制台日志输出：

✓ Tests passed: 1 of 1 test – 759 ms

```
Logging initialized using 'class org.apache.ibatis.logging.stdout.StdOutImpl' adapter.  
Property 'mapperLocations' was not specified.  
com.itheima.service.impl.DeptServiceImpl@28369db0  
com.example.TokenParser@98637a2
```

大家可以想象一下，如果采用以上这种方式来完成自动配置，那我们进行项目开发时，当需要引入大量的第三方的依赖，就需要在启动类上配置N多要扫描的包，这种方式会很繁琐。而且这种大面积的扫描性能也比较低。

缺点：

1. 使用繁琐
2. 性能低

结论：SpringBoot中并没有采用以上这种方案。

3.2.2.3 方案二

@Import导入

• 导入形式主要有以下几种：

1. 导入普通类
2. 导入配置类
3. 导入ImportSelector接口实现类

1) . 使用@Import导入普通类：

```
1  @Import(TokenParser.class) //导入的类会被Spring加载到IOC容器中  
2  @SpringBootApplication  
3  public class SpringbootWebConfig2Application {  
4      public static void main(String[] args) {  
5          SpringApplication.run(SpringbootWebConfig2Application.class,  
6              args);  
7      }  
8  }
```

重新执行测试方法，控制台日志输出：

✓ Tests passed: 1 of 1 test – 759 ms

```
Logging initialized using 'class org.apache.ibatis.logging.stdout.StdOutImpl' adapter.  
Property 'mapperLocations' was not specified.  
com.itheima.service.impl.DeptServiceImpl@28369db0  
com.example.TokenParser@21c75084
```

2). 使用@Import导入配置类:

- 配置类

```
1  @Configuration
2  public class HeaderConfig {
3      @Bean
4      public HeaderParser headerParser() {
5          return new HeaderParser();
6      }
7
8      @Bean
9      public HeaderGenerator headerGenerator() {
10         return new HeaderGenerator();
11     }
12 }
```

- 启动类

```
1  @Import(HeaderConfig.class) //导入配置类
2  @SpringBootApplication
3  public class SpringbootWebConfig2Application {
4      public static void main(String[] args) {
5          SpringApplication.run(SpringbootWebConfig2Application.class,
6              args);
7      }
```

- 测试类

```
1  @SpringBootTest
2  public class AutoConfigurationTests {
3      @Autowired
4      private ApplicationContext applicationContext;
5
6      @Test
7      public void testHeaderParser() {
8
9          System.out.println(applicationContext.getBean(HeaderParser.class));
10     }
11
12     @Test
13     public void testHeaderGenerator() {
```

```

13         System.out.println(applicationContext.getBean(HeaderGenerator.class
14     ));
15     }
16     //省略其他代码...
17 }

```

执行测试方法：

✓ Tests passed: 1 of 1 test – 347 ms

Logging initialized using 'class org.apache.ibatis.logging.stdout.StdOutImpl' adapter.
Property 'mapperLocations' was not specified.
com.itheima.service.impl.DeptServiceImpl@5befbac1
com.example.HeaderGenerator@40016ce1

3). 使用@Import导入ImportSelector接口实现类：

- ImportSelector接口实现类

```

1 public class MyImportSelector implements ImportSelector {
2     public String[] selectImports(AnnotationMetadata
3         importingClassMetadata) {
4         //返回值字符串数组（数组中封装了全限定名称的类）
5         return new String[]{"com.example.HeaderConfig"};
6     }
7 }

```

- 启动类

```

1 @Import(MyImportSelector.class) //导入ImportSelector接口实现类
2 @SpringBootApplication
3 public class SpringbootWebConfig2Application {
4
5     public static void main(String[] args) {
6         SpringApplication.run(SpringbootWebConfig2Application.class,
7             args);
8     }
9 }

```

执行测试方法：

✓ Tests passed: 1 of 1 test – 606 ms

```
Logging initialized using 'class org.apache.ibatis.logging.stdout.StdOutImpl' adapter.  
Property 'mapperLocations' was not specified.  
com.itheima.service.impl.DeptServiceImpl@5befbac1  
com.example.HeaderParser@40016ce1
```

我们使用`@Import`注解通过这三种方式都可以导入第三方依赖中所提供的bean或者是配置类。

思考：如果基于以上方式完成自动配置，当要引入一个第三方依赖时，是不是还要知道第三方依赖中有哪些配置类和哪些Bean对象？

- 答案：是的。（对程序员来讲，很不友好，而且比较繁琐）

思考：当我们要使用第三方依赖，依赖中到底有哪些bean和配置类，谁最清楚？

- 答案：第三方依赖自身最清楚。

结论：我们不用自己指定要导入哪些bean对象和配置类了，让第三方依赖它自己来指定。

怎么让第三方依赖自己指定bean对象和配置类？

- 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

4). 使用第三方依赖提供的 `@EnableXxxxx`注解

- 第三方依赖中提供的注解

```
1  @Retention(RetentionPolicy.RUNTIME)  
2  @Target(ElementType.TYPE)  
3  @Import(MyImportSelector.class) //指定要导入哪些bean对象或配置类  
4  public @interface EnableHeaderConfig {  
5  }
```

- 在使用时只需在启动类上加上`@EnableXxxxx`注解即可


```

1  @EnableHeaderConfig //使用第三方依赖提供的Enable开头的注解
2  @SpringBootApplication
3  public class SpringbootWebConfig2Application {
4      public static void main(String[] args) {
5          SpringApplication.run(SpringbootWebConfig2Application.class,
6              args);
7      }
8  }

```

执行测试方法：

✓ Tests passed: 1 of 1 test – 347 ms

```

Logging initialized using 'class org.apache.ibatis.logging.stdout.StdOutImpl' adapter.
Property 'mapperLocations' was not specified.
com.itheima.service.impl.DeptServiceImpl@5befbac1
com.example.HeaderGenerator@40016ce1

```

以上四种方式都可以完成导入操作，但是第4种方式会更方便更优雅，而这种方式也是SpringBoot当中所采用的方式。

3.2.3 原理分析

3.2.3.1 源码跟踪

前面我们讲解了在项目当中引入第三方依赖之后，如何加载第三方依赖中定义好的bean对象以及配置类，从而完成自动配置操作。那下面我们通过源码跟踪的形式来剖析下SpringBoot底层到底是如何完成自动配置的。

源码跟踪技巧：

在跟踪框架源码的时候，一定要抓住关键点，找到核心流程。一定不要从头到尾一行代码去看，一个方法的去研究，一定要找到关键流程，抓住关键点，先在宏观上对整个流程或者整个原理有一个认识，有精力再去研究其中的细节。

要搞清楚SpringBoot的自动配置原理，要从SpringBoot启动类上使用的核心注解

@SpringBootApplication开始分析：

```

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME) 元注解
@Documented
@Inherited
@SpringBootConfiguration 表示是配置类
@EnableAutoConfiguration Enable开头的注解
@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
    组件扫描 @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {

```

在@SpringBootApplication注解中包含了：

- 元注解（不再解释）
- @SpringBootConfiguration
- @EnableAutoConfiguration
- @ComponentScan

我们先来看第一个注解：@SpringBootConfiguration

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

@SpringBootConfiguration注解上使用了@Configuration，表明SpringBoot启动类就是一个配置类。

@Indexed注解，是用来加速应用启动的（不用关心）。

接下来再看@ComponentScan注解：

```
@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
    进行包扫描 @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {
```

@ComponentScan注解是用来进行组件扫描的，扫描启动类所在的包及其子包下所有被@Component及其衍生注解声明的类。

SpringBoot启动类，之所以具备扫描包功能，就是因为包含了@ComponentScan注解。

最后我们来看看@EnableAutoConfiguration注解（自动配置核心注解）：

```

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(AutoConfigurationImportSelector.class) 导入ImportSelector接口实现类
public @interface EnableAutoConfiguration {

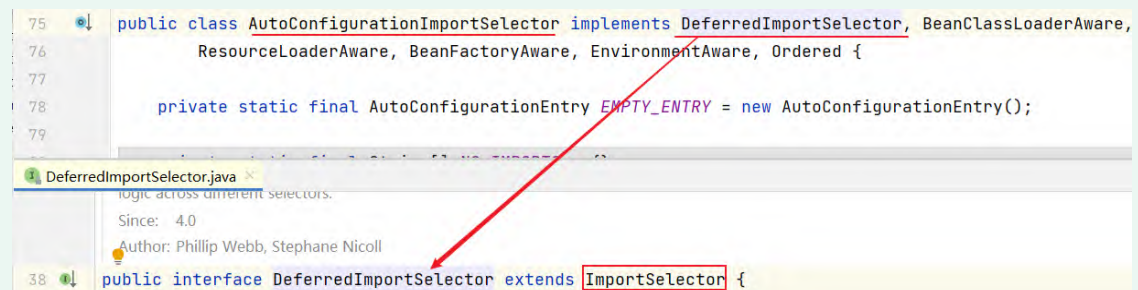
    Environment property that can be used to override when auto-configuration is enabled.

    String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautoconfiguration";
}

```

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

AutoConfigurationImportSelector类是ImportSelector接口的实现类。

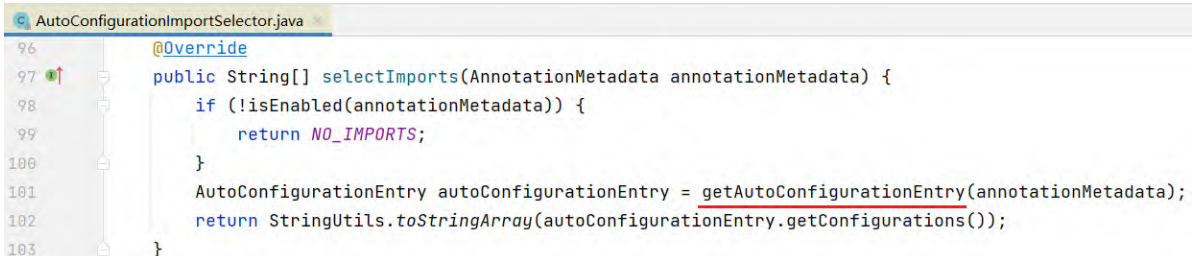


```

75 public class AutoConfigurationImportSelector implements DeferredImportSelector, BeanClassLoaderAware,
76 ResourceLoaderAware, BeanFactoryAware, EnvironmentAware, Ordered {
77
78     private static final AutoConfigurationEntry EMPTY_ENTRY = new AutoConfigurationEntry();
79
80     // ...
81
82     // ...
83
84     // ...
85
86     // ...
87
88     // ...
89
90     // ...
91
92     // ...
93
94     // ...
95
96     // ...
97
98     // ...
99
100    // ...
101
102    // ...
103
104    // ...
105
106    // ...
107
108    // ...
109
110    // ...
111
112    // ...
113
114    // ...
115
116    // ...
117
118    // ...
119
120    // ...
121
122    // ...
123
124    // ...
125
126    // ...
127
128    // ...
129
130    // ...
131
132    // ...
133
134    // ...
135
136    // ...
137
138    // ...
139
140    // ...
141
142    // ...
143
144    // ...
145
146    // ...
147
148    // ...
149
150    // ...
151
152    // ...
153
154    // ...
155
156    // ...
157
158    // ...
159
160    // ...
161
162    // ...
163
164    // ...
165
166    // ...
167
168    // ...
169
170    // ...
171
172    // ...
173
174    // ...
175
176    // ...
177
178    // ...
179
180    // ...
181
182    // ...
183
184    // ...
185
186    // ...
187
188    // ...
189
190    // ...
191
192    // ...
193
194    // ...
195
196    // ...
197
198    // ...
199
200    // ...
201
202    // ...
203
204    // ...
205
206    // ...
207
208    // ...
209
210    // ...
211
212    // ...
213
214    // ...
215
216    // ...
217
218    // ...
219
220    // ...
221
222    // ...
223
224    // ...
225
226    // ...
227
228    // ...
229
230    // ...
231
232    // ...
233
234    // ...
235
236    // ...
237
238    // ...
239
240    // ...
241
242    // ...
243
244    // ...
245
246    // ...
247
248    // ...
249
250    // ...
251
252    // ...
253
254    // ...
255
256    // ...
257
258    // ...
259
260    // ...
261
262    // ...
263
264    // ...
265
266    // ...
267
268    // ...
269
270    // ...
271
272    // ...
273
274    // ...
275
276    // ...
277
278    // ...
279
280    // ...
281
282    // ...
283
284    // ...
285
286    // ...
287
288    // ...
289
290    // ...
291
292    // ...
293
294    // ...
295
296    // ...
297
298    // ...
299
300    // ...
301
302    // ...
303
304    // ...
305
306    // ...
307
308    // ...
309
310    // ...
311
312    // ...
313
314    // ...
315
316    // ...
317
318    // ...
319
320    // ...
321
322    // ...
323
324    // ...
325
326    // ...
327
328    // ...
329
330    // ...
331
332    // ...
333
334    // ...
335
336    // ...
337
338    // ...
339
340    // ...
341
342    // ...
343
344    // ...
345
346    // ...
347
348    // ...
349
350    // ...
351
352    // ...
353
354    // ...
355
356    // ...
357
358    // ...
359
360    // ...
361
362    // ...
363
364    // ...
365
366    // ...
367
368    // ...
369
370    // ...
371
372    // ...
373
374    // ...
375
376    // ...
377
378    // ...
379
380    // ...
381
382    // ...
383
384    // ...
385
386    // ...
387
388    // ...
389
390    // ...
391
392    // ...
393
394    // ...
395
396    // ...
397
398    // ...
399
400    // ...
401
402    // ...
403
404    // ...
405
406    // ...
407
408    // ...
409
410    // ...
411
412    // ...
413
414    // ...
415
416    // ...
417
418    // ...
419
420    // ...
421
422    // ...
423
424    // ...
425
426    // ...
427
428    // ...
429
430    // ...
431
432    // ...
433
434    // ...
435
436    // ...
437
438    // ...
439
440    // ...
441
442    // ...
443
444    // ...
445
446    // ...
447
448    // ...
449
450    // ...
451
452    // ...
453
454    // ...
455
456    // ...
457
458    // ...
459
460    // ...
461
462    // ...
463
464    // ...
465
466    // ...
467
468    // ...
469
470    // ...
471
472    // ...
473
474    // ...
475
476    // ...
477
478    // ...
479
480    // ...
481
482    // ...
483
484    // ...
485
486    // ...
487
488    // ...
489
490    // ...
491
492    // ...
493
494    // ...
495
496    // ...
497
498    // ...
499
500    // ...
501
502    // ...
503
504    // ...
505
506    // ...
507
508    // ...
509
510    // ...
511
512    // ...
513
514    // ...
515
516    // ...
517
518    // ...
519
520    // ...
521
522    // ...
523
524    // ...
525
526    // ...
527
528    // ...
529
530    // ...
531
532    // ...
533
534    // ...
535
536    // ...
537
538    // ...
539
540    // ...
541
542    // ...
543
544    // ...
545
546    // ...
547
548    // ...
549
550    // ...
551
552    // ...
553
554    // ...
555
556    // ...
557
558    // ...
559
560    // ...
561
562    // ...
563
564    // ...
565
566    // ...
567
568    // ...
569
570    // ...
571
572    // ...
573
574    // ...
575
576    // ...
577
578    // ...
579
580    // ...
581
582    // ...
583
584    // ...
585
586    // ...
587
588    // ...
589
590    // ...
591
592    // ...
593
594    // ...
595
596    // ...
597
598    // ...
599
600    // ...
601
602    // ...
603
604    // ...
605
606    // ...
607
608    // ...
609
610    // ...
611
612    // ...
613
614    // ...
615
616    // ...
617
618    // ...
619
620    // ...
621
622    // ...
623
624    // ...
625
626    // ...
627
628    // ...
629
630    // ...
631
632    // ...
633
634    // ...
635
636    // ...
637
638    // ...
639
640    // ...
641
642    // ...
643
644    // ...
645
646    // ...
647
648    // ...
649
650    // ...
651
652    // ...
653
654    // ...
655
656    // ...
657
658    // ...
659
660    // ...
661
662    // ...
663
664    // ...
665
666    // ...
667
668    // ...
669
670    // ...
671
672    // ...
673
674    // ...
675
676    // ...
677
678    // ...
679
680    // ...
681
682    // ...
683
684    // ...
685
686    // ...
687
688    // ...
689
690    // ...
691
692    // ...
693
694    // ...
695
696    // ...
697
698    // ...
699
700    // ...
701
702    // ...
703
704    // ...
705
706    // ...
707
708    // ...
709
710    // ...
711
712    // ...
713
714    // ...
715
716    // ...
717
718    // ...
719
720    // ...
721
722    // ...
723
724    // ...
725
726    // ...
727
728    // ...
729
730    // ...
731
732    // ...
733
734    // ...
735
736    // ...
737
738    // ...
739
740    // ...
741
742    // ...
743
744    // ...
745
746    // ...
747
748    // ...
749
750    // ...
751
752    // ...
753
754    // ...
755
756    // ...
757
758    // ...
759
760    // ...
761
762    // ...
763
764    // ...
765
766    // ...
767
768    // ...
769
770    // ...
771
772    // ...
773
774    // ...
775
776    // ...
777
778    // ...
779
780    // ...
781
782    // ...
783
784    // ...
785
786    // ...
787
788    // ...
789
790    // ...
791
792    // ...
793
794    // ...
795
796    // ...
797
798    // ...
799
800    // ...
801
802    // ...
803
804    // ...
805
806    // ...
807
808    // ...
809
810    // ...
811
812    // ...
813
814    // ...
815
816    // ...
817
818    // ...
819
820    // ...
821
822    // ...
823
824    // ...
825
826    // ...
827
828    // ...
829
830    // ...
831
832    // ...
833
834    // ...
835
836    // ...
837
838    // ...
839
840    // ...
841
842    // ...
843
844    // ...
845
846    // ...
847
848    // ...
849
850    // ...
851
852    // ...
853
854    // ...
855
856    // ...
857
858    // ...
859
860    // ...
861
862    // ...
863
864    // ...
865
866    // ...
867
868    // ...
869
870    // ...
871
872    // ...
873
874    // ...
875
876    // ...
877
878    // ...
879
880    // ...
881
882    // ...
883
884    // ...
885
886    // ...
887
888    // ...
889
890    // ...
891
892    // ...
893
894    // ...
895
896    // ...
897
898    // ...
899
900    // ...
901
902    // ...
903
904    // ...
905
906    // ...
907
908    // ...
909
910    // ...
911
912    // ...
913
914    // ...
915
916    // ...
917
918    // ...
919
920    // ...
921
922    // ...
923
924    // ...
925
926    // ...
927
928    // ...
929
930    // ...
931
932    // ...
933
934    // ...
935
936    // ...
937
938    // ...
939
940    // ...
941
942    // ...
943
944    // ...
945
946    // ...
947
948    // ...
949
950    // ...
951
952    // ...
953
954    // ...
955
956    // ...
957
958    // ...
959
960    // ...
961
962    // ...
963
964    // ...
965
966    // ...
967
968    // ...
969
970    // ...
971
972    // ...
973
974    // ...
975
976    // ...
977
978    // ...
979
980    // ...
981
982    // ...
983
984    // ...
985
986    // ...
987
988    // ...
989
990    // ...
991
992    // ...
993
994    // ...
995
996    // ...
997
998    // ...
999
1000   // ...

```

AutoConfigurationImportSelector类中重写了ImportSelector接口的selectImports()方法：



```

96 @Override
97 public String[] selectImports(AnnotationMetadata annotationMetadata) {
98     if (!isEnabled(annotationMetadata)) {
99         return NO_IMPORTS;
100     }
101     AutoConfigurationEntry autoConfigurationEntry = getAutoConfigurationEntry(annotationMetadata);
102     return StringUtils.toStringArray(autoConfigurationEntry.getConfigurations());
103 }

```

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

```

protected AutoConfigurationEntry getAutoConfigurationEntry(AnnotationMetadata annotationMetadata) {
    if (!isEnabled(annotationMetadata)) {
        return EMPTY_ENTRY;
    }
    AnnotationAttributes attributes = getAttributes(annotationMetadata);
    List<String> configurations = getCandidateConfigurations(annotationMetadata, attributes);
    configurations = removeDuplicates(configurations);
    Set<String> exclusions = getExclusions(annotationMetadata, attributes);
    checkExcludedClasses(configurations, exclusions);
    configurations.removeAll(exclusions);
    configurations = getConfigurationClassFilter().filter(configurations);
    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

```
protected List<String> getCandidateConfigurations(AnnotationMetadata metadata, AnnotationAttributes attributes) {  
    List<String> configurations = new ArrayList<>(  
        SpringFactoriesLoader.loadFactoryNames(getSpringFactoriesLoaderFactoryClass(), getBeanClassLoader());  
        ImportCandidates.load(AutoConfiguration.class, getBeanClassLoader()).forEach(configurations::add);  
        Assert.notEmpty(configurations,  
            message: "No auto configuration classes found in META-INF/spring.factories nor in META-INF/spring/org.springframework.  
                + "are using a custom packaging, make sure that file is correct.");  
    return configurations;  
}
```

getCandidateConfigurations 方法的功能:

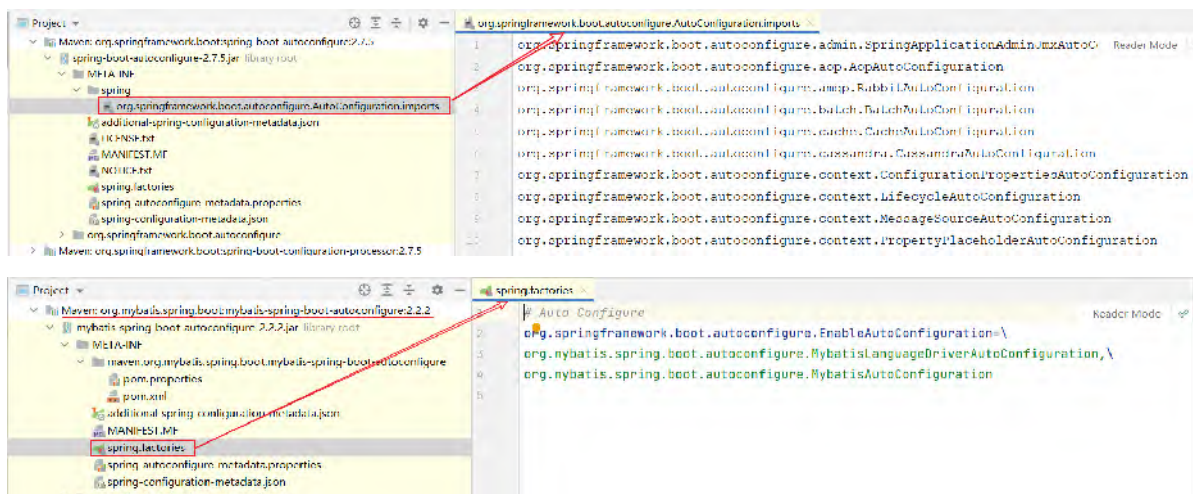
获取所有基于META-

INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports
文件、META-INF/spring.factories 文件中配置类的集合

META-

INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports
文件和META-INF/spring.factories 文件这两个文件在哪里呢?

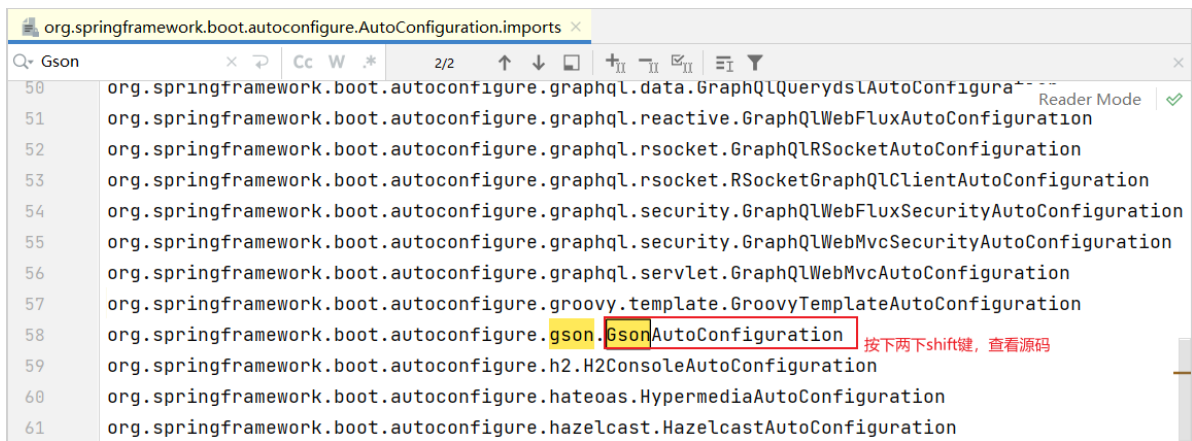
- 通常在引入的起步依赖中, 都有包含以上两个文件



在前面在给大家演示自动配置的时候, 我们直接在测试类当中注入了一个叫gson的bean对象, 进行JSON格式转换。虽然我们并没有配置bean对象, 但是我们是可以直接注入使用的。原因就是因为在自动配置类当中做了自动配置。到底是在哪个自动配置类当中做的自动配置呢? 我们通过搜索来查询一下。

在META-

INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports
配置文件中指定了第三方依赖Gson的配置类: GsonAutoConfiguration



```
org.springframework.boot.autoconfigure.AutoConfiguration.imports
50 org.springframework.boot.autoconfigure.graphql.data.GraphQLQuerydslAutoConfiguration
51 org.springframework.boot.autoconfigure.graphql.reactive.GraphQLWebFluxAutoConfiguration
52 org.springframework.boot.autoconfigure.graphql.rsocket.GraphQLRSocketAutoConfiguration
53 org.springframework.boot.autoconfigure.graphql.rsocket.RSocketGraphQLClientAutoConfiguration
54 org.springframework.boot.autoconfigure.graphql.security.GraphQLWebFluxSecurityAutoConfiguration
55 org.springframework.boot.autoconfigure.graphql.security.GraphQLWebMvcSecurityAutoConfiguration
56 org.springframework.boot.autoconfigure.graphql.servlet.GraphQLWebMvcAutoConfiguration
57 org.springframework.boot.autoconfigure.groovy.template.GroovyTemplateAutoConfiguration
58 org.springframework.boot.autoconfigure.gson.GsonAutoConfiguration
59 org.springframework.boot.autoconfigure.h2.H2ConsoleAutoConfiguration
60 org.springframework.boot.autoconfigure.hateoas.HypermediaAutoConfiguration
61 org.springframework.boot.autoconfigure.hazelcast.HazelcastAutoConfiguration
```

第三方依赖中提供的GsonAutoConfiguration类：

```
@AutoConfiguration
@ConditionalOnClass({Gson.class})
@EnableConfigurationProperties({GsonProperties.class})
public class GsonAutoConfiguration {
    public GsonAutoConfiguration() {}

    @Bean
    @ConditionalOnMissingBean
    public GsonBuilder gsonBuilder(List<GsonBuilderCustomizer> customizers) {...}

    @Bean
    @ConditionalOnMissingBean
    public Gson gson(GsonBuilder gsonBuilder) {
        return gsonBuilder.create();
    }
}
```

在GsonAutoConfiguration类上，添加了注解@AutoConfiguration，通过查看源码，可以明确：GsonAutoConfiguration类是一个配置。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Configuration(proxyBeanMethods = false)
@AutoConfigureBefore
@AutoConfigureAfter
public @interface AutoConfiguration {
```

看到这里，大家就应该明白为什么可以完成自动配置了，原理就是在配置类中定义一个@Bean标识的方法，而Spring会自动调用配置类中使用@Bean标识的方法，并把方法的返回值注册到IOC容器中。

自动配置源码小结

自动配置原理源码入口就是@SpringBootApplication注解，在这个注解中封装了3个注解，分别是：

- `@SpringBootConfiguration`
 - 声明当前类是一个配置类
- `@ComponentScan`
 - 进行组件扫描（SpringBoot中默认扫描的是启动类所在的当前包及其子包）
- `@EnableAutoConfiguration`
 - 封装了`@Import`注解（`Import`注解中指定了一个`ImportSelector`接口的实现类）
 - 在实现类重写的`selectImports()`方法，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）。

当SpringBoot程序启动时，就会加载配置文件当中所定义的配置类，并将这些配置类信息(类的全限定名)封装到String类型的数组中，最终通过`@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。

最后呢给大家抛出一个问题：在`META-`

`INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports`文件中定义的配置类非常多，而且每个配置类中又可以定义很多的bean，那这些bean都会注册到Spring的IOC容器中吗？

答案：并不是。 在声明bean对象时，上面有加一个以`@Conditional`开头的注解，这种注解的作用就是按照条件进行装配，只有满足条件之后，才会将bean注册到Spring的IOC容器中（下面会详细来讲解）

3.2.3.2 @Conditional

我们在跟踪SpringBoot自动配置的源码的时候，在自动配置类声明bean的时候，除了方法上加了一个`@Bean`注解以外，还会经常用到一个注解，就是以`Conditional`开头的这一类的注解。以`Conditional`开头的这些注解都是条件装配的注解。下面我们就来介绍下条件装配注解。

`@Conditional`注解：

- 作用：按照一定的条件进行判断，在满足给定条件后才会注册对应的bean对象到Spring的IOC容器中。
- 位置：方法、类

- @Conditional本身是一个父注解，派生出大量的子注解：
 - @ConditionalOnClass: 判断环境中存在对应字节码文件，才注册bean到IOC容器。
 - @ConditionalOnMissingBean: 判断环境中没有对应的bean (类型或名称)，才注册bean到IOC容器。
 - @ConditionalOnProperty: 判断配置文件中存在对应属性和值，才注册bean到IOC容器。

下面我们通过代码来演示下Conditional注解的使用：

- @ConditionalOnClass注解

```
1  @Configuration
2  public class HeaderConfig {
3
4      @Bean
5      @ConditionalOnClass(name="io.jsonwebtoken.Jwts") //环境中存在指定的
        这个类，才会将该bean加入IOC容器
6      public HeaderParser headerParser() {
7          return new HeaderParser();
8      }
9
10     //省略其他代码...
11 }
```

- pom.xml

```
1  <!--JWT令牌-->
2  <dependency>
3      <groupId>io.jsonwebtoken</groupId>
4      <artifactId>jjwt</artifactId>
5      <version>0.9.1</version>
6  </dependency>
```

- 测试类

```

1  @SpringBootTest
2  public class AutoConfigurationTests {
3      @Autowired
4      private ApplicationContext applicationContext;
5
6      @Test
7      public void testHeaderParser() {
8
9          System.out.println(applicationContext.getBean(HeaderParser.class));
10
11         //省略其他代码...
12     }

```

执行testHeaderParser()测试方法：

✓ Tests passed: 1 of 1 test – 693 ms

Logging initialized using 'class org.apache.ibatis.logging.stdout.StdOutImpl' adapter.
Property 'mapperLocations' was not specified.
com.itheima.service.impl.DeptServiceImpl@2e86807a
com.example.HeaderParser@6f867b0c

因为io.jsonwebtoken.Jwts字节码文件在启动SpringBoot程序时已存在，所以创建HeaderParser对象并注册到IOC容器中。

- @ConditionalOnMissingBean注解

```

1  @Configuration
2  public class HeaderConfig {
3
4      @Bean
5      @ConditionalOnMissingBean //不存在该类型的bean，才会将该bean加入IOC
6      容器
7      public HeaderParser headerParser() {
8          return new HeaderParser();
9      }
10
11     //省略其他代码...
12 }

```

执行testHeaderParser()测试方法：

✓ Tests passed: 1 of 1 test – 723 ms

```
Logging initialized using 'class org.apache.ibatis.logging.stdout.StdOutImpl' adapter.  
Property 'mapperLocations' was not specified.  
com.itheima.service.impl.DeptServiceImpl@61514735  
com.example.HeaderParser@4afd65fd
```

SpringBoot在调用@Bean标识的headerParser()前，IOC容器中是没有HeaderParser类型的bean，所以HeaderParser对象正常创建，并注册到IOC容器中。

再次修改@ConditionalOnMissingBean注解：

```
1  @Configuration  
2  public class HeaderConfig {  
3  
4      @Bean  
5      @ConditionalOnMissingBean(name="deptController2") //不存在指定名称  
        的bean，才会将该bean加入IOC容器  
6      public HeaderParser headerParser() {  
7          return new HeaderParser();  
8      }  
9  
10     //省略其他代码...  
11 }
```

执行testHeaderParser()测试方法：

✓ Tests passed: 1 of 1 test – 539 ms

```
Logging initialized using 'class org.apache.ibatis.logging.stdout.StdOutImpl' adapter.  
Property 'mapperLocations' was not specified.  
com.itheima.service.impl.DeptServiceImpl@208f0007  
com.example.HeaderParser@48a46b0f
```

因为在SpringBoot环境中不存在名字叫deptController2的bean对象，所以创建HeaderParser对象并注册到IOC容器中。

再次修改@ConditionalOnMissingBean注解：

```

1  @Configuration
2  public class HeaderConfig {
3
4      @Bean
5      @ConditionalOnMissingBean(HeaderConfig.class) //不存在指定类型的
        bean, 才会将bean加入IOC容器
6      public HeaderParser headerParser() {
7          return new HeaderParser();
8      }
9
10     //省略其他代码...
11 }

```

```

1  @SpringBootTest
2  public class AutoConfigurationTests {
3
4      @Autowired
5      private ApplicationContext applicationContext;
6
7      @Test
8      public void testHeaderParser() {
9
10         System.out.println(applicationContext.getBean(HeaderParser.class));
11     }
12
13     //省略其他代码...
14 }

```

执行testHeaderParser() 测试方法：

```

org.springframework.beans.factory.NoSuchBeanDefinitionException: No qualifying bean of type 'com.example.HeaderParser' available
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBean(DefaultListableBeanFactory.java:351)
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBean(DefaultListableBeanFactory.java:342)
    at org.springframework.context.support.AbstractApplicationContext.getBean(AbstractApplicationContext.java:1172)
    at com.itheima.AutoConfigurationTests.testHeaderParser(AutoConfigurationTests.java:37) <31 internal lines>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1541) <9 internal lines>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1541) <29 internal lines>

```

因为HeaderConfig类中添加@Configuration注解，而@Configuration注解中包含了@Component，所以SpringBoot启动时会创建HeaderConfig类对象，并注册到IOC容器中。

当IOC容器中有HeaderConfig类型的bean存在时，不会把创建HeaderParser对象注册到IOC容器中。而IOC容器中没有HeaderParser类型的对象时，通过getBean(HeaderParser.class)方法获取bean对象时，引发异常：

NoSuchBeanDefinitionException

- @ConditionalOnProperty注解（这个注解和配置文件当中配置的属性有关系）

先在application.yml配置文件中添加如下的键值对：

```
1  name: itheima
```

在声明bean的时候就可以指定一个条件@ConditionalOnProperty

```
1  @Configuration
2  public class HeaderConfig {
3
4      @Bean
5      @ConditionalOnProperty(name = "name", havingValue = "itheima") //配置文件中存在指定属性名与值，才会将bean加入IOC容器
6      public HeaderParser headerParser() {
7          return new HeaderParser();
8      }
9
10     @Bean
11     public HeaderGenerator headerGenerator() {
12         return new HeaderGenerator();
13     }
14 }
```

执行testHeaderParser() 测试方法：

✓ Tests passed: 1 of 1 test – 560 ms

Logging initialized using 'class org.apache.ibatis.logging.stdout.StdOutImpl' adapter.
Property 'mapperLocations' was not specified.
com.itheima.service.impl.DeptServiceImpl@2b7e8044
com.example.HeaderParser@76596288

修改@ConditionalOnProperty注解： havingValue的值修改为"itheima2"

```
1  @Bean
2  @ConditionalOnProperty(name = "name", havingValue = "itheima2") //配置文件中存在指定属性名与值，才会将bean加入IOC容器
3  public HeaderParser headerParser() {
4      return new HeaderParser();
5  }
```

再次执行testHeaderParser() 测试方法：

```

org.springframework.beans.factory.NoSuchBeanDefinitionException: No qualifying bean of type 'com.example.HeaderParser' available
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBean(DefaultListableBeanFactory.java:351)
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBean(DefaultListableBeanFactory.java:342)
    at org.springframework.context.support.AbstractApplicationContext.getBean(AbstractApplicationContext.java:1172)
    at com.itheima.AutoConfigurationTests.testHeaderParser(AutoConfigurationTests.java:37) <31 internal lines>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1541) <9 internal lines>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1541) <25 internal lines>

```

因为application.yml配置文件中, 不存在: name: itheima2, 所以HeaderParser对象在IOC容器中不存在

我们再回头看看之前讲解SpringBoot源码时提到的一个配置类: GsonAutoConfiguration

```

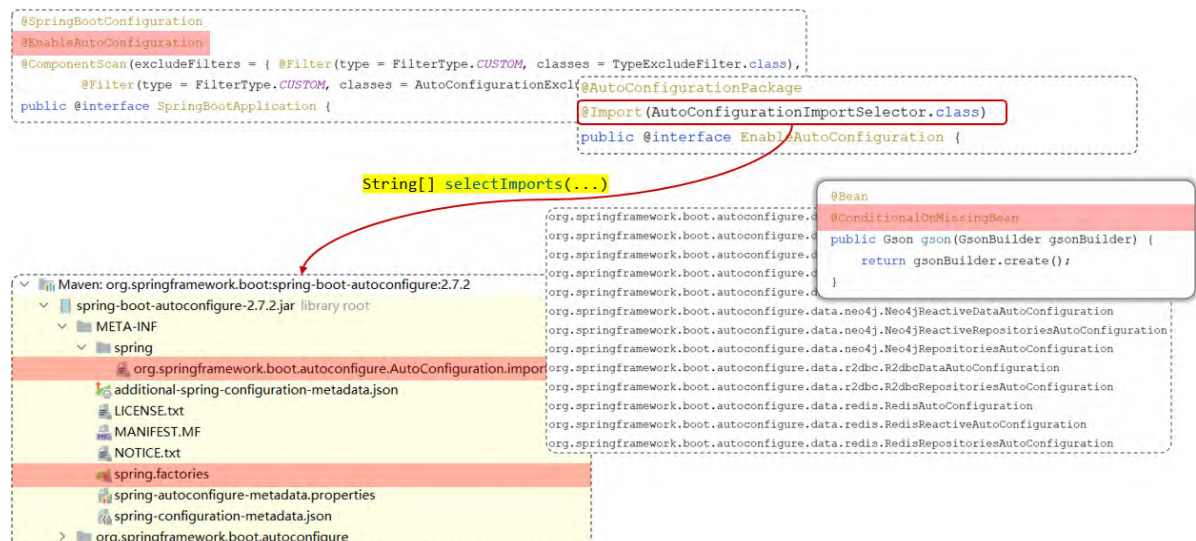
@AutoConfiguration
@ConditionalOnClass({Gson.class}) 当前SpringBoot环境中存在Gson.class文件时, 才会创建当前配置类对象
@EnableConfigurationProperties({GsonProperties.class})
public class GsonAutoConfiguration {
    public GsonAutoConfiguration() {
    }

    @Bean
    @ConditionalOnMissingBean 当前GsonBuilder对象不存在时, 创建该对象并注册到IOC容器中
    public GsonBuilder gsonBuilder(List<GsonBuilderCustomizer> customizers) {
        GsonBuilder builder = new GsonBuilder();
        customizers.forEach((c) -> {
            c.customize(builder);
        });
        return builder;
    }

    @Bean
    @ConditionalOnMissingBean 当前Gson对象不存在时, 创建该对象并注册到IOC容器中
    public Gson gson(GsonBuilder gsonBuilder) { return gsonBuilder.create(); }
}

```

最后再给大家梳理一下自动配置原理:



自动配置的核心就在@SpringBootApplication注解上，SpringBootApplication这个注解底层包含了3个注解，分别是：

- @SpringBootConfiguration
- @ComponentScan
- @EnableAutoConfiguration

@EnableAutoConfiguration这个注解才是自动配置的核心。

- 它封装了一个@Import注解，Import注解里面指定了一个ImportSelector接口的实现类。
- 在这个实现类中，重写了ImportSelector接口中的selectImports()方法。
- 而selectImports()方法中会去读取两份配置文件，并将配置文件中定义的配置类做为selectImports()方法的返回值返回，返回值代表的就是需要将哪些类交给Spring的IOC容器进行管理。
- 那么所有自动配置类中声明的bean都会加载到Spring的IOC容器中吗？其实并不会，因为这些配置类中在声明bean时，通常都会添加@Conditional开头的注解，这个注解就是进行条件装配。而Spring会根据Conditional注解有选择性的进行bean的创建。
- @Enable 开头的注解底层，它就封装了一个注解 import 注解，它里面指定了一个类，是 ImportSelector 接口的实现类。在实现类当中，我们需要去实现 ImportSelector 接口当中的一个方法 selectImports 这个方法。这个方法的返回值代表的就是我需要将哪些类交给 spring 的 IOC容器进行管理。
- 此时它会去读取两份配置文件，一份儿是 spring.factories，另外一份儿是 autoConfiguration.imports。而在 autoConfiguration.imports 这份儿文件当中，它就会去配置大量的自动配置的类。
- 而前面我们也提到过这些所有的自动配置类当中，所有的 bean都会加载到 spring 的 IOC 容器当中吗？其实并不会，因为这些配置类当中，在声明 bean 的时候，通常会加上这么一类@Conditional 开头的注解。这个注解就是进行条件装配。所以SpringBoot非常的智能，它会根据 @Conditional 注解来进行条件装配。只有条件成立，它才会声明这个bean，才会将这个 bean 交给 IOC 容器管理。

3.2.4 案例

3.2.4.1 自定义starter分析

前面我们解析了SpringBoot中自动配置的原理，下面我们就通过一个自定义starter案例来加深大家对于自动配置原理的理解。首先介绍一下自定义starter的业务场景，再分析一下具体的操作步骤。

所谓starter指的就是SpringBoot当中的起步依赖。在SpringBoot当中已经给我们提供了很多的起步依赖了，我们为什么还需要自定义 starter 起步依赖？这是因为在实际的项目开发当中，我们可能会用到很多第三方的技术，并不是所有的第三方的技术官方都给我们提供了与SpringBoot整合的starter起步依赖，但是这些技术又非常的通用，在很多项目组当中都在使用。

业务场景：

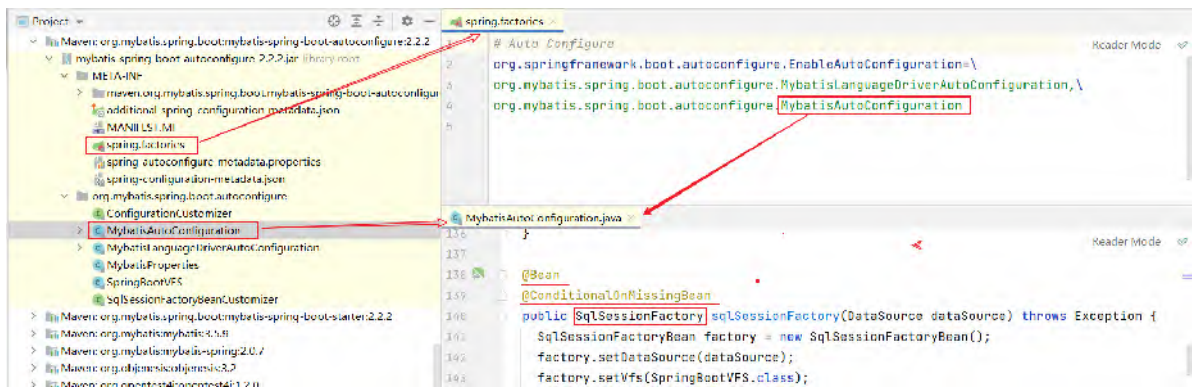
- 我们前面案例当中所使用的阿里云OSS对象存储服务，现在阿里云的官方是没有给我们提供对应的起步依赖的，这个时候使用起来就会比较繁琐，我们需要引入对应的依赖。我们还需要在配置文件当中进行配置，还需要基于官方SDK示例来改造对应的工具类，我们在项目当中才可以进行使用。
- 大家想在我们当前项目当中使用了阿里云OSS，我们需要进行这么多步的操作。在别的项目组当中要想使用阿里云OSS，是不是也需要进行这么多步的操作，所以这个时候我们就可以自定义一些公共组件，在这些公共组件当中，我就可以提前把需要配置的bean都提前配置好。将来在项目当中，我要想使用这个技术，我直接将组件对应的坐标直接引入进来，就已经自动配置好了，就可以直接使用了。我们也可以把公共组件提供给别的项目组进行使用，这样就可以大大的简化我们的开发。

在SpringBoot项目中，一般都会将这些公共组件封装为SpringBoot当中的starter，也就是我们所说的起步依赖。

<div>> Maven: org.springframework.boot:spring-boot-autoconfigure:2.7.5</div> <div>> Maven: org.springframework.boot:spring-boot-starter-web:2.7.5</div>	<div>自动配置功能</div> <div>依赖管理功能</div>	SpringBoot官方
<div>> Maven: org.mybatis.spring.boot:mybatis-spring-boot-autoconfigure:2.2.2</div> <div>> Maven: org.mybatis.spring.boot:mybatis-spring-boot-starter:2.2.2</div>	<div>自动配置功能</div> <div>依赖管理功能</div>	
<div>> Maven: com.github.pagehelper:pagehelper-spring-boot-autoconfigure:1.4.2</div> <div>> Maven: com.github.pagehelper:pagehelper-spring-boot-starter:1.4.2</div>	<div>自动配置功能</div> <div>依赖管理功能</div>	其它技术提供

SpringBoot官方starter命名： spring-boot-starter-xxxx

第三组织提供的starter命名： xxxx-spring-boot-starter



Mybatis提供了配置类，并且也提供了springboot会自动读取的配置文件。当SpringBoot项目启动时，会读取到spring.factories配置文件中的配置类并加载配置类，生成相关bean对象注册到IOC容器中。

结果：我们可以直接在SpringBoot程序中使用Mybatis自动配置的bean对象。

在自定义一个起步依赖starter的时候，按照规范需要定义两个模块：

1. starter模块（进行依赖管理[把程序开发所需要的依赖都定义在starter起步依赖中]）
2. autoconfigure模块（自动配置）

将来在项目当中进行相关功能开发时，只需要引入一个起步依赖就可以了，因为它会将autoconfigure自动配置的依赖给传递下来。

上面我们简单介绍了自定义starter的场景，以及自定义starter时涉及到的模块之后，接下来我们就来完成一个自定义starter的案例。

需求：自定义aliyun-oss-spring-boot-starter，完成阿里云OSS操作工具类AliyunOSSUtils的自动配置。

目标：引入起步依赖引入之后，要想使用阿里云OSS，注入AliyunOSSUtils直接使用即可。

之前阿里云OSS的使用：

- 配置文件

```

1  #配置阿里云OSS参数
2  aliyun:
3      oss:
4          endpoint: https://oss-cn-shanghai.aliyuncs.com
5          accessKeyId: LTAI5t9MZK8iq5T2Av5GLDxX
6          accessKeySecret: C0IrHzKZGKqU8S7YQcevcotD3Zd5Tc
7          bucketName: web-framework01

```

- AliOSSProperties类

```

1  @Data
2  @Component
3  @ConfigurationProperties(prefix = "aliyun.oss")
4  public class AliOSSProperties {
5      //区域
6      private String endpoint;
7      //身份ID
8      private String accessKeyId ;
9      //身份密钥
10     private String accessKeySecret ;
11     //存储空间
12     private String bucketName;
13 }
14

```

- AliOSSUtils工具类

```

1  @Component //当前类对象由Spring创建和管理
2  public class AliOSSUtils {
3      @Autowired
4      private AliOSSProperties aliOSSProperties;
5
6      /**
7       * 实现上传图片到oss
8       */
9      public String upload(MultipartFile multipartFile) throws
IOException {
10         // 获取上传的文件的输入流
11         InputStream inputStream = multipartFile.getInputStream();
12
13         // 避免文件覆盖
14         String originalFilename =
multipartFile.getOriginalFilename();
15         String fileName = UUID.randomUUID().toString() +
originalFilename.substring(originalFilename.lastIndexOf("."));

```

```

16
17         //上传文件到 oss
18         OSS ossClient = new
        OSSClientBuilder().build(alioSSProperties.getEndpoint(),
19                                 alioSSProperties.getAccessKeyId(),
        alioSSProperties.getAccessKeySecret());
20         ossClient.putObject(alioSSProperties.getBucketName(),
        fileName, inputStream);
21
22         //文件访问路径
23         String url =alioSSProperties.getEndpoint().split("//")[0] +
        "//" + alioSSProperties.getBucketName() + "." +
        alioSSProperties.getEndpoint().split("//")[1] + "/" + fileName;
24
25         // 关闭ossClient
26         ossClient.shutdown();
27         return url;// 把上传到oss的路径返回
28     }
29 }

```

当我们在项目当中要使用阿里云OSS，就可以注入AliOSSUtils工具类来进行文件上传。但这种方式其实是比较繁琐的。

大家再思考，现在我们使用阿里云OSS，需要做这么几步，将来大家在开发其他的项目的时候，你使用阿里云OSS，这几步你要不要做？当团队中其他小伙伴也在使用阿里云OSS的时候，步骤 不也是一样的。

所以这个时候我们就可以制作一个公共组件(自定义starter)。starter定义好之后，将来要使用阿里云OSS进行文件上传，只需要将起步依赖引入进来之后，就可以直接注入AliOSSUtils使用了。

需求明确了，接下来我们再来分析一下具体的实现步骤：

- 第1步：创建自定义starter模块（进行依赖管理）
 - 把阿里云OSS所有的依赖统一管理起来
- 第2步：创建autoconfigure模块
 - 在starter中引入autoconfigure（我们使用时只需要引入starter起步依赖即可）
- 第3步：在autoconfigure中完成自动配置
 1. 定义一个自动配置类，在自动配置类中将所要配置的bean都提前配置好
 2. 定义配置文件，把自动配置类的全类名定义在配置文件中

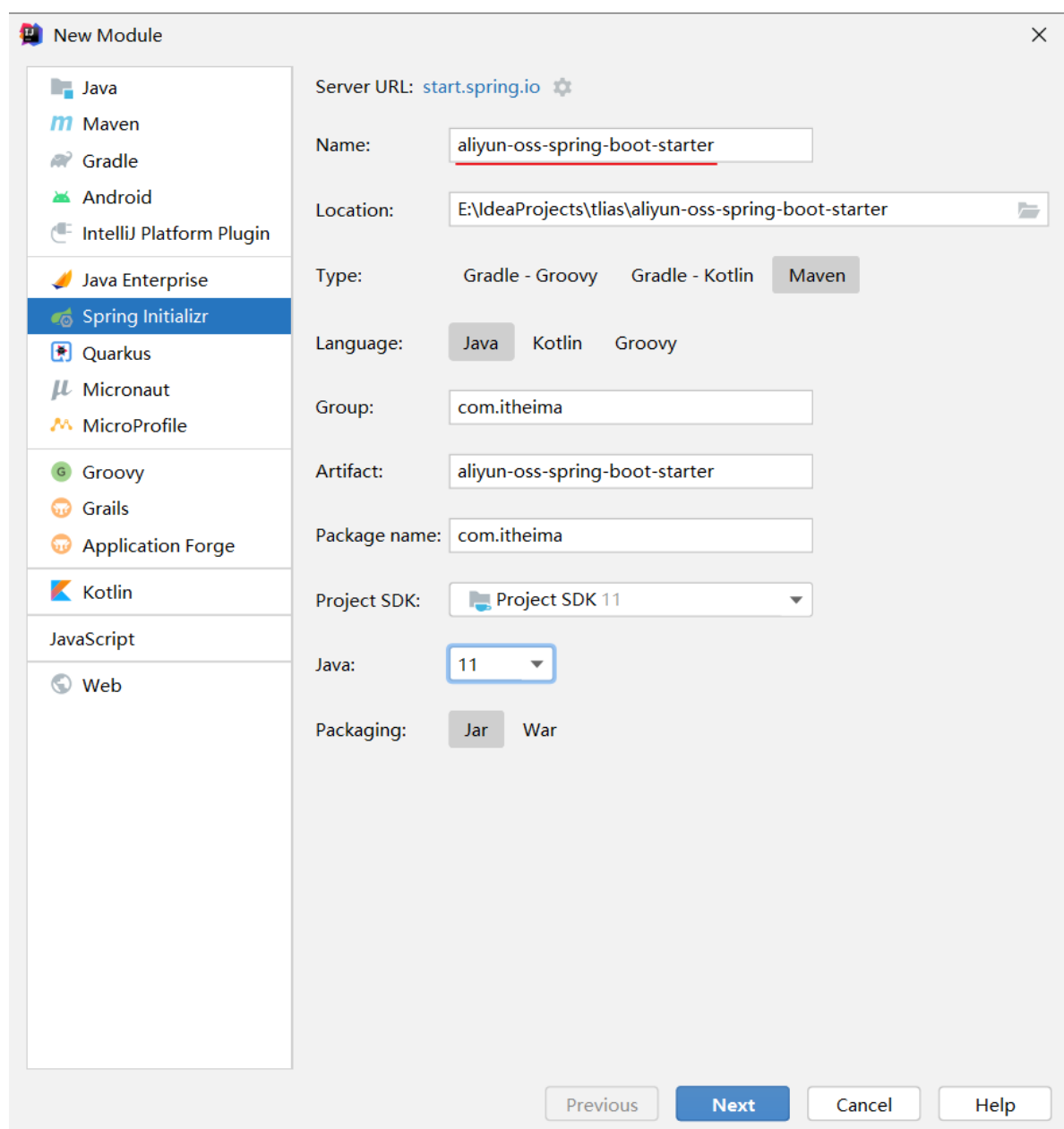
我们分析完自定义阿里云OSS自动配置的操作步骤了，下面我们就按照分析的步骤来实现自定义starter。

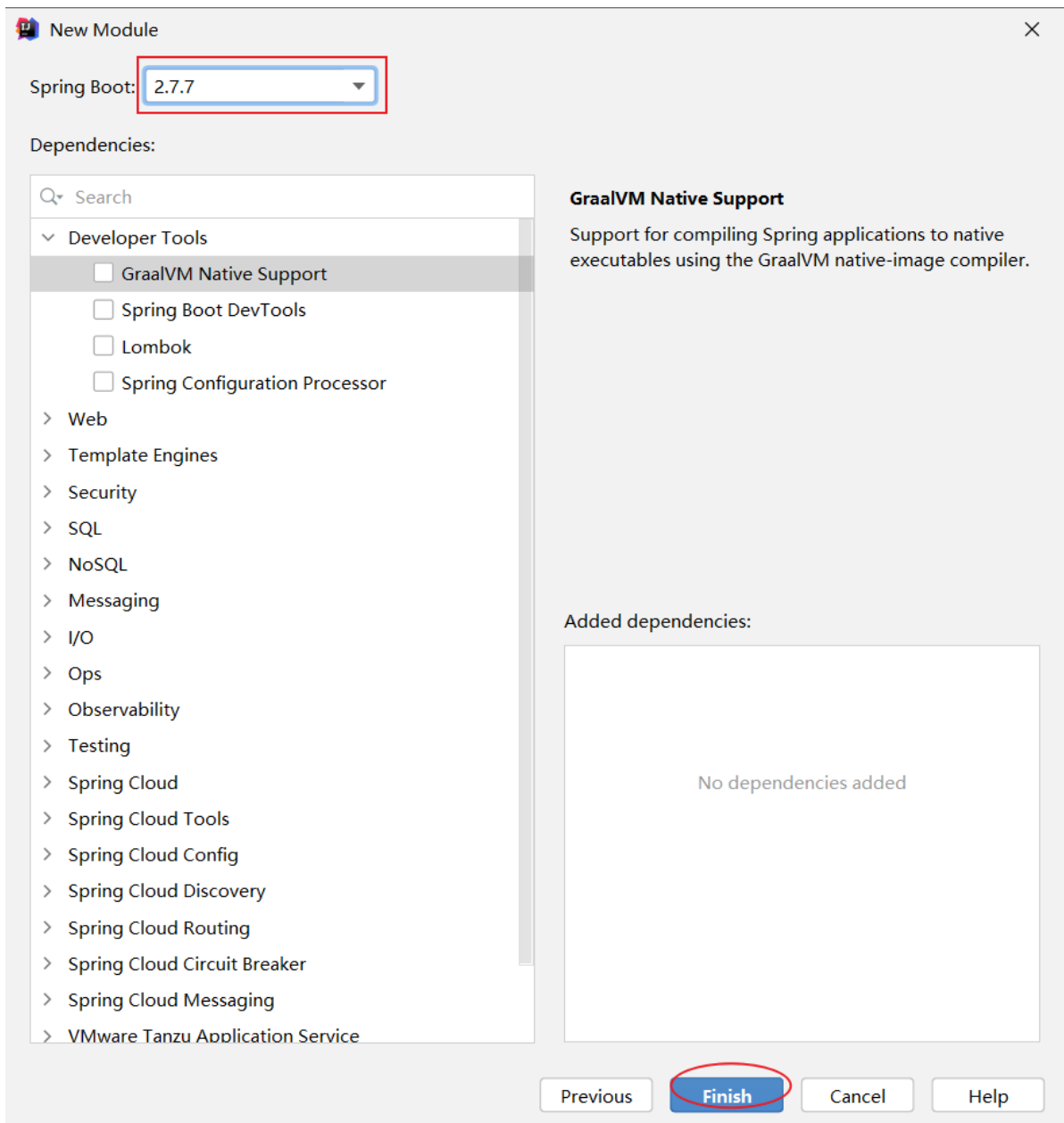
3.2.4.2 自定义starter实现

自定义starter的步骤我们刚才已经分析了，接下来我们就按照分析的步骤来完成自定义starter的开发。

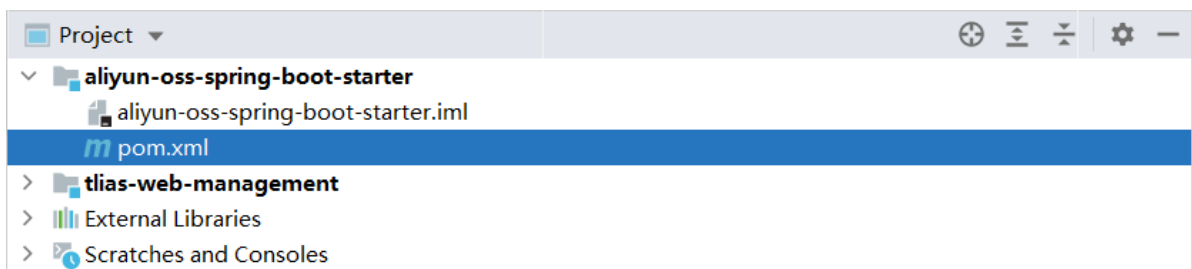
首先我们先来创建两个Maven模块：

1). aliyun-oss-spring-boot-starter模块





创建完starter模块后，删除多余的文件，最终保留内容如下：



删除pom.xml文件中多余的内容后：

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
   https://maven.apache.org/xsd/maven-4.0.0.xsd">
4     <modelVersion>4.0.0</modelVersion>
5     <parent>
6         <groupId>org.springframework.boot</groupId>
```

```
7         <artifactId>spring-boot-starter-parent</artifactId>
8         <version>2.7.5</version>
9         <relativePath/> <!-- lookup parent from repository -->
10    </parent>
11
12    <groupId>com.aliyun.oss</groupId>
13    <artifactId>aliyun-oss-spring-boot-starter</artifactId>
14    <version>0.0.1-SNAPSHOT</version>
15
16    <properties>
17        <java.version>11</java.version>
18    </properties>
19
20    <dependencies>
21        <dependency>
22            <groupId>org.springframework.boot</groupId>
23            <artifactId>spring-boot-starter</artifactId>
24        </dependency>
25    </dependencies>
26
27 </project>
```

2). aliyun-oss-spring-boot-autoconfigure模块

Java

Maven

Gradle

Android

IntelliJ Platform Plugin

Java Enterprise

Spring Initializr

Quarkus

Micronaut

MicroProfile

Groovy

Grails

Application Forge

Kotlin

JavaScript

Web

Server URL: start.spring.io

Name:

Location:

Type:

Gradle - Groovy

Gradle - Kotlin

Maven

Language:

Java

Kotlin

Groovy

Group:

Artifact:

Package name:

Project SDK:

Project SDK 11

Java:

11

Packaging:

Jar

War

Previous

Next

Cancel

Help

Spring Boot:

Dependencies:

Search

Developer Tools

☐ GraalVM Native Support

☐ Spring Boot DevTools

☐ Lombok

☐ Spring Configuration Processor

> Web

> Template Engines

> Security

> SQL

> NoSQL

> Messaging

> I/O

> Ops

> Observability

> Testing

> Spring Cloud

> Spring Cloud Tools

GraalVM Native Support

Support for compiling Spring applications to native executables using the GraalVM native-image compiler.

Added dependencies:

No dependencies added

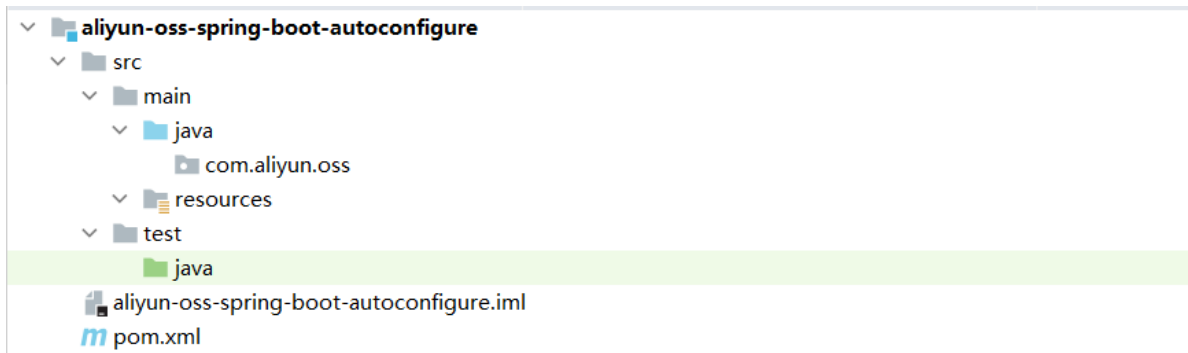
Previous

Finish

Cancel

Help

创建完starter模块后，删除多余的文件，最终保留内容如下：



删除pom.xml文件中多余的内容后：

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
   https://maven.apache.org/xsd/maven-4.0.0.xsd">
4     <modelVersion>4.0.0</modelVersion>
5     <parent>
6         <groupId>org.springframework.boot</groupId>
7         <artifactId>spring-boot-starter-parent</artifactId>
8         <version>2.7.5</version>
9         <relativePath/> <!-- lookup parent from repository -->
10    </parent>
11
12    <groupId>com.aliyun.oss</groupId>
13    <artifactId>aliyun-oss-spring-boot-autoconfigure</artifactId>
14    <version>0.0.1-SNAPSHOT</version>
15
16    <properties>
17        <java.version>11</java.version>
18    </properties>
19
20    <dependencies>
21        <dependency>
22            <groupId>org.springframework.boot</groupId>
23            <artifactId>spring-boot-starter</artifactId>
24        </dependency>
25    </dependencies>
26
27 </project>
```

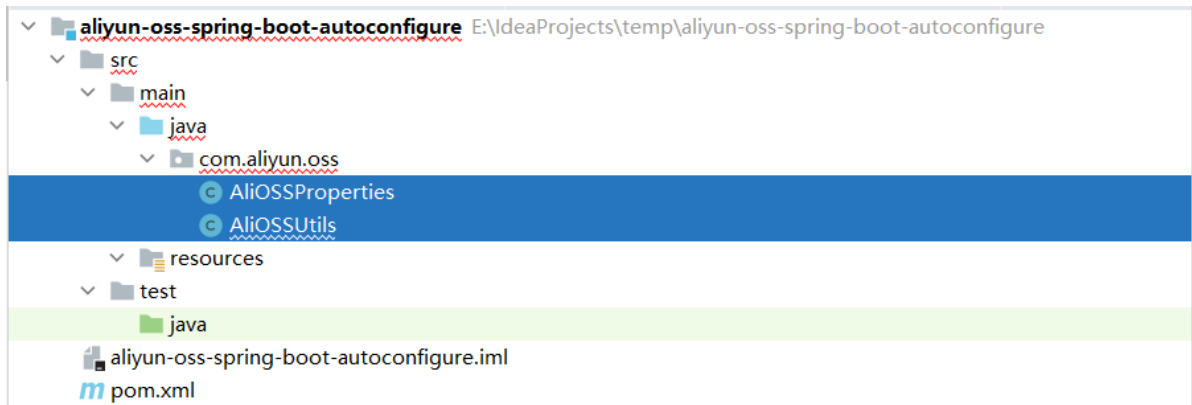
按照我们之前的分析，是需要在starter模块中来引入autoconfigure这个模块的。打开starter模块中的pom文件：

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
   https://maven.apache.org/xsd/maven-4.0.0.xsd">
4     <modelVersion>4.0.0</modelVersion>
5     <parent>
6         <groupId>org.springframework.boot</groupId>
7         <artifactId>spring-boot-starter-parent</artifactId>
8         <version>2.7.5</version>
9         <relativePath/> <!-- lookup parent from repository -->
10    </parent>
11
12    <groupId>com.aliyun.oss</groupId>
13    <artifactId>aliyun-oss-spring-boot-starter</artifactId>
14    <version>0.0.1-SNAPSHOT</version>
15
16    <properties>
17        <java.version>11</java.version>
18    </properties>
19
20    <dependencies>
21        <!--引入autoconfigure模块-->
22        <dependency>
23            <groupId>com.aliyun.oss</groupId>
24            <artifactId>aliyun-oss-spring-boot-
autoconfigure</artifactId>
25            <version>0.0.1-SNAPSHOT</version>
26        </dependency>
27
28        <dependency>
29            <groupId>org.springframework.boot</groupId>
30            <artifactId>spring-boot-starter</artifactId>
31        </dependency>
32    </dependencies>
33
34 </project>
```

前两步已经完成了，接下来是最关键的就是第三步：

在autoconfigure模块当中来完成自动配置操作。

我们将之前案例中所使用的阿里云OSS部分的代码直接拷贝到autoconfigure模块下，然后进行改造就行了。



拷贝过来后，还缺失一些相关的依赖，需要把相关依赖也拷贝过来：

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5      https://maven.apache.org/xsd/maven-4.0.0.xsd">
6      <modelVersion>4.0.0</modelVersion>
7      <parent>
8          <groupId>org.springframework.boot</groupId>
9          <artifactId>spring-boot-starter-parent</artifactId>
10         <version>2.7.5</version>
11         <relativePath/> <!-- lookup parent from repository -->
12     </parent>
13     <groupId>com.aliyun.oss</groupId>
14     <artifactId>aliyun-oss-spring-boot-autoconfigure</artifactId>
15     <version>0.0.1-SNAPSHOT</version>
16     <properties>
17         <java.version>11</java.version>
18     </properties>
19     <dependencies>
20         <dependency>
21             <groupId>org.springframework.boot</groupId>
```

```
23         <artifactId>spring-boot-starter</artifactId>
24     </dependency>
25
26     <!--引入web起步依赖-->
27     <dependency>
28         <groupId>org.springframework.boot</groupId>
29         <artifactId>spring-boot-starter-web</artifactId>
30     </dependency>
31
32     <!--Lombok-->
33     <dependency>
34         <groupId>org.projectlombok</groupId>
35         <artifactId>lombok</artifactId>
36     </dependency>
37
38     <!--阿里云OSS-->
39     <dependency>
40         <groupId>com.aliyun.oss</groupId>
41         <artifactId>aliyun-sdk-oss</artifactId>
42         <version>3.15.1</version>
43     </dependency>
44
45     <dependency>
46         <groupId>javax.xml.bind</groupId>
47         <artifactId>jaxb-api</artifactId>
48         <version>2.3.1</version>
49     </dependency>
50     <dependency>
51         <groupId>javax.activation</groupId>
52         <artifactId>activation</artifactId>
53         <version>1.1.1</version>
54     </dependency>
55     <!-- no more than 2.3.3-->
56     <dependency>
57         <groupId>org.glassfish.jaxb</groupId>
58         <artifactId>jaxb-runtime</artifactId>
59         <version>2.3.3</version>
60     </dependency>
61 </dependencies>
62 </project>
```

现在大家思考下，在类上添加的@Component注解还有用吗？


```

@Data
@Component
@ConfigurationProperties(prefix = "aliyun.oss")
public class AliOSSProperties {

@Component
public class AliOSSUtils {

    @Autowired
    private AliOSSProperties aliOSSProperties;

```

答案：没用了。 在SpringBoot项目中，并不会去扫描com.aliyun.oss这个包，不扫描这个包那类上的注解也就失去了作用。

@Component注解不需要使用了，可以从类上删除了。

删除后报红色错误，暂时不理睬，后面再来处理。

```

@Data
@ConfigurationProperties(prefix = "aliyun.oss")
public class AliOSSProperties {

```

删除AliOSSUtils类中的@Component注解、@Autowired注解

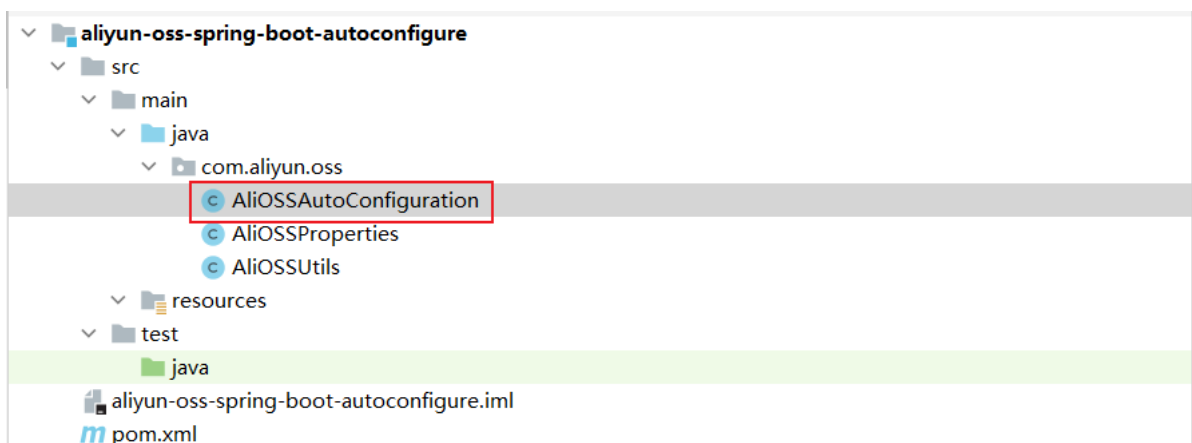
```

public class AliOSSUtils {

    private AliOSSProperties aliOSSProperties;

```

下面我们就要定义一个自动配置类了，在自动配置类当中来声明AliOSSUtils的bean对象。



AliOSSAutoConfiguration类：

```

1  @Configuration//当前类为Spring配置类
2  @EnableConfigurationProperties (AliOSSProperties.class)//导入
    AliOSSProperties类，并交给SpringIOC管理
3  public class AliOSSAutoConfiguration {
4
5
6      //创建AliOSSUtils对象，并交给SpringIOC容器
7      @Bean
8      public AliOSSUtils alioSSUtils (AliOSSProperties
    AliOSSProperties){
9          AliOSSUtils alioSSUtils = new AliOSSUtils();
10         alioSSUtils.setAliOSSProperties (alioSSProperties);
11         return alioSSUtils;
12     }
13 }

```

AliOSSProperties类:

```

1  /*阿里云OSS相关配置*/
2  @Data
3  @ConfigurationProperties (prefix = "aliyun.oss")
4  public class AliOSSProperties {
5      //区域
6      private String endpoint;
7      //身份ID
8      private String accessKeyId ;
9      //身份密钥
10     private String accessKeySecret ;
11     //存储空间
12     private String bucketName;
13 }

```

AliOSSUtils类:

```

1  @Data
2  public class AliOSSUtils {
3      private AliOSSProperties alioSSProperties;
4
5      /**
6       * 实现上传图片到oss
7       */

```

```

8      public String upload(MultipartFile multipartFile) throws
IOException {
9          // 获取上传的文件的输入流
10         InputStream inputStream = multipartFile.getInputStream();
11
12         // 避免文件覆盖
13         String originalFilename =
multipartFile.getOriginalFilename();
14         String fileName = UUID.randomUUID().toString() +
originalFilename.substring(originalFilename.lastIndexOf("."));
15
16         //上传文件到 oss
17         OSS ossClient = new
OSSClientBuilder().build(alioSSProperties.getEndpoint(),
18             alioSSProperties.getAccessKeyId(),
alioSSProperties.getAccessKeySecret());
19         ossClient.putObject(alioSSProperties.getBucketName(),
fileName, inputStream);
20
21         //文件访问路径
22         String url =alioSSProperties.getEndpoint().split("//")[0] +
"//" + alioSSProperties.getBucketName() + "." +
alioSSProperties.getEndpoint().split("//")[1] + "/" + fileName;
23
24         // 关闭ossClient
25         ossClient.shutdown();
26         return url;// 把上传到oss的路径返回
27     }
28 }

```

在aliyun-oss-spring-boot-autoconfigure模块中的resources下，新建自动配置文件：

- META-

```

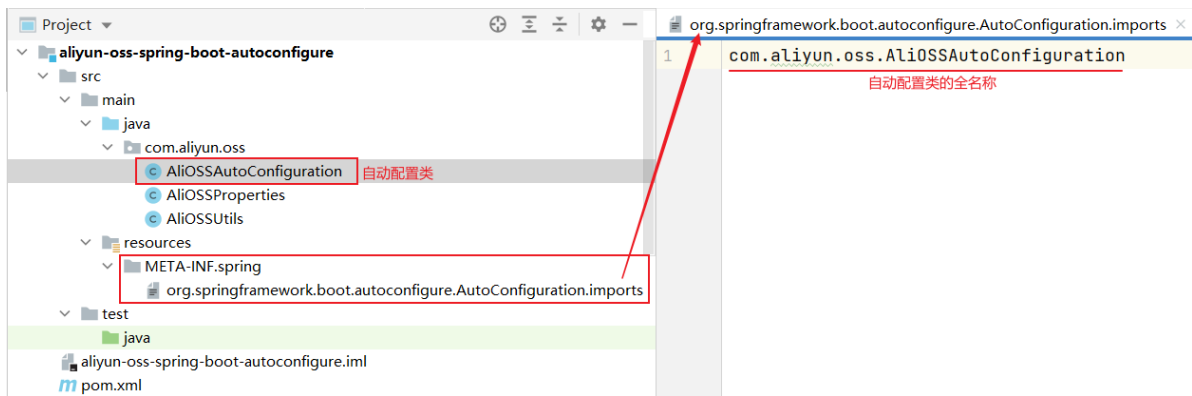
INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.impo
rts

```

```

1    com.aliyun.oss.AliOSSAutoConfiguration

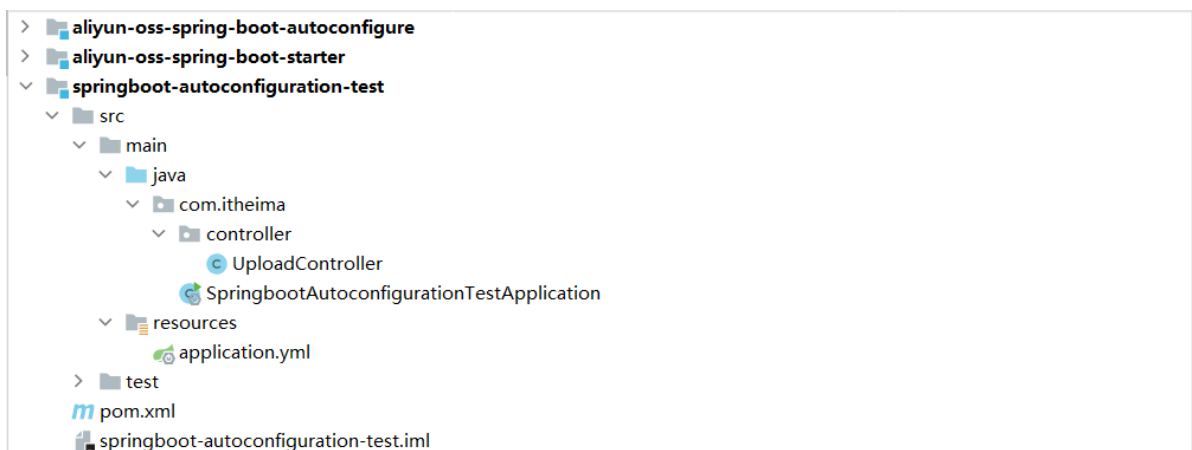
```



3.2.4.3 自定义starter测试

阿里云OSS的starter我们刚才已经定义好了，接下来我们就来做一个测试。

今天的课程资料当中，提供了一个自定义starter的测试工程。我们直接打开文件夹，里面有一个测试工程。测试工程就是springboot-autoconfiguration-test，我们只需要将测试工程直接导入到Idea当中即可。



测试前准备：

1. 在test工程中引入阿里云starter依赖
 - 通过依赖传递，会把autoconfigure依赖也引入了

```
1 <!--引入阿里云OSS起步依赖-->
2 <dependency>
3     <groupId>com.aliyun.oss</groupId>
4     <artifactId>aliyun-oss-spring-boot-starter</artifactId>
5     <version>0.0.1-SNAPSHOT</version>
6 </dependency>
```

2. 在test工程中的application.yml文件中，配置阿里云OSS配置参数信息（从以前的工程中拷贝即可）

```

1  #配置阿里云OSS参数
2  aliyun:
3    oss:
4      endpoint: https://oss-cn-shanghai.aliyuncs.com
5      accessKeyId: LTAI5t9MZK8iq5T2Av5GLDxX
6      accessKeySecret: C0IrHzKZGKqU8S7YQcevcotD3Zd5Tc
7      bucketName: web-framework01

```

3. 在test工程中的UploadController类编写代码

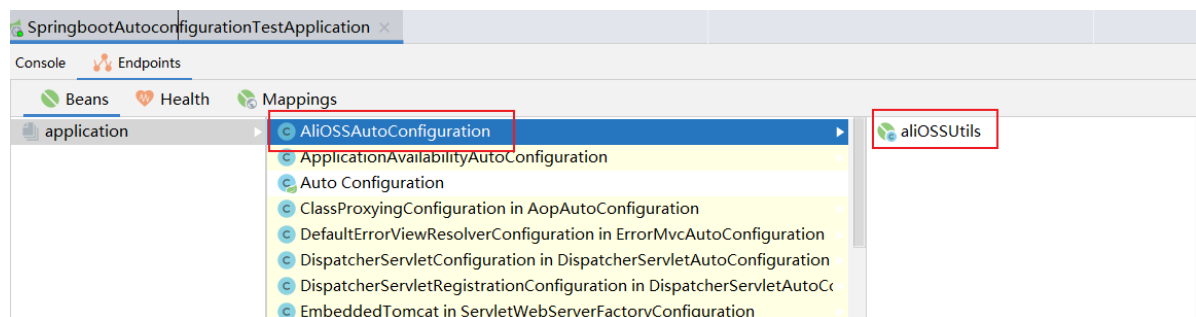
```

1  @RestController
2  public class UploadController {
3
4      @Autowired
5      private AliOSSUtils aliOSSUtils;
6
7      @PostMapping("/upload")
8      public String upload(MultipartFile image) throws Exception {
9          //上传文件到阿里云 oss
10         String url = aliOSSUtils.upload(image);
11         return url;
12     }
13
14 }

```

编写完代码后，我们启动当前的SpringBoot测试工程：

- 随着SpringBoot项目启动，自动配置会把AliOSSUtils的bean对象装配到IOC容器中



用postman工具进行文件上传：



通过断点可以看到自动注入AliOSSUtils的bean对象：

```
@RestController
public class UploadController {

    @Autowired
    private AliOSSUtils aliOSSUtils;    aliOSSUtils: "AliOSSUtils.aliOSSProperties=AliOSSProperties(
                                     自动注入

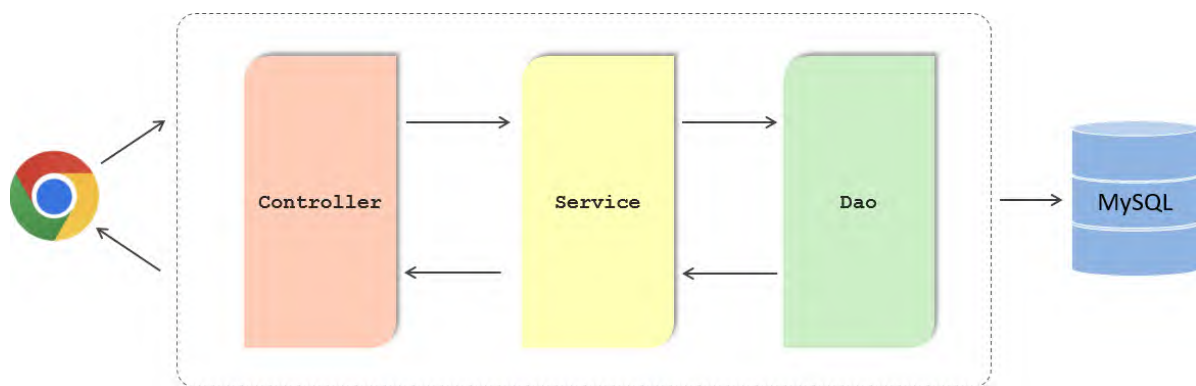
    @PostMapping("/upload")
    public String upload(MultipartFile image) throws Exception {    image: StandardMultipartHttpServ
        // 上传文件到阿里云 OSS
        String url = aliOSSUtils.upload(image);    image: StandardMultipartHttpServletRequest$Stand
        return url;
    }
}
```

4. Web后端开发总结

到此基于SpringBoot进行web后端开发的相关知识我们已经学习完毕了。下面我们一起针对这段web课程做一个总结。

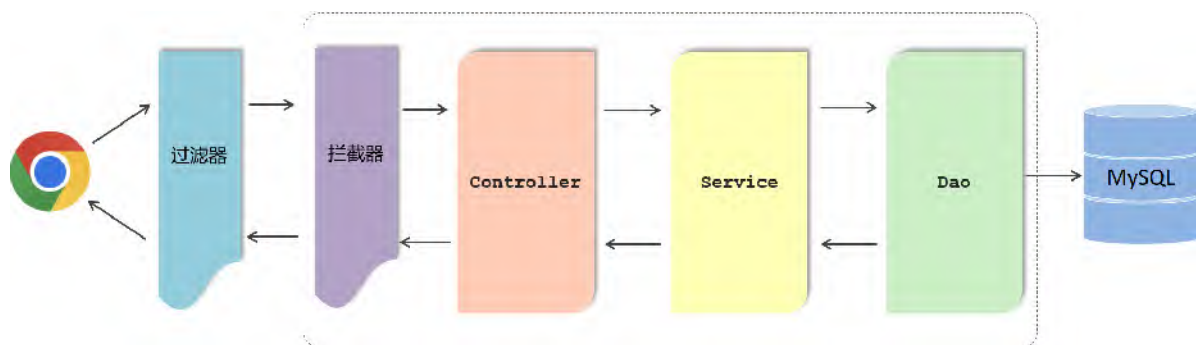
我们来回顾一下关于web后端开发，我们都学习了哪些内容，以及每一块知识，具体是属于哪个框架的。

web后端开发现在基本上都是基于标准的三层架构进行开发的，在三层架构当中，Controller控制器层负责接收请求响应数据，Service业务层负责具体的业务逻辑处理，而Dao数据访问层也叫持久层，就是用来处理数据访问操作的，来完成数据库当中数据的增删改查操作。



在三层架构当中，前端发起请求首先会到达Controller(不进行逻辑处理)，然后Controller会直接调用Service 进行逻辑处理， Service再调用Dao完成数据访问操作。

如果我们在执行具体的业务处理之前，需要去做一些通用的业务处理，比如：我们要进行统一的登录校验，我们要进行统一的字符编码等这些操作时，我们就可以借助于Javaweb当中三大组件之一的过滤器Filter或者是Spring当中提供的拦截器Interceptor来实现。

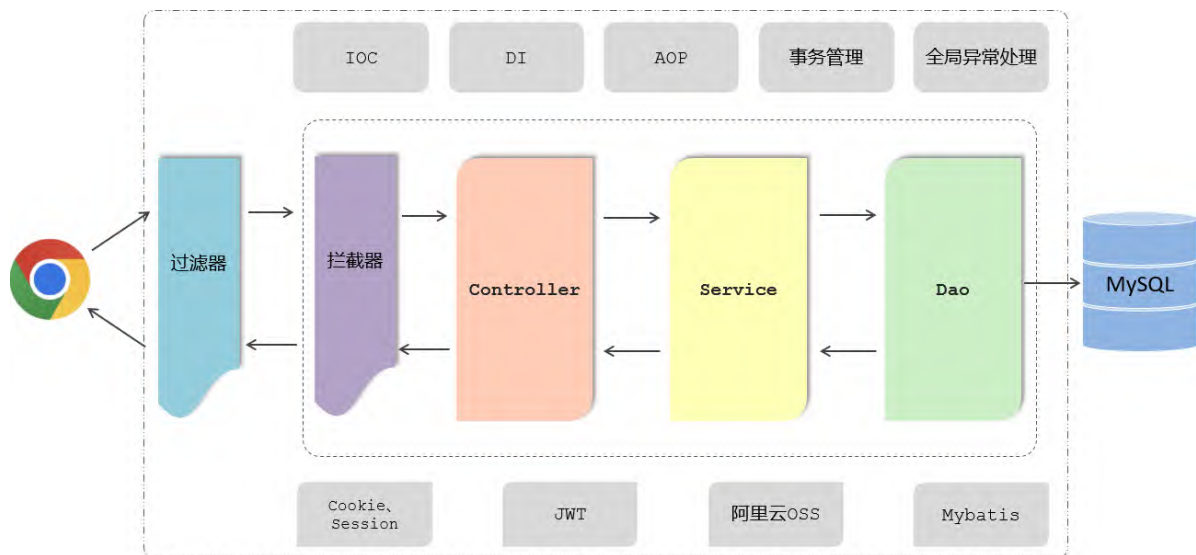


而为了实现三层架构层与层之间的解耦，我们学习了Spring框架当中的第一大核心：IOC控制反转与DI依赖注入。

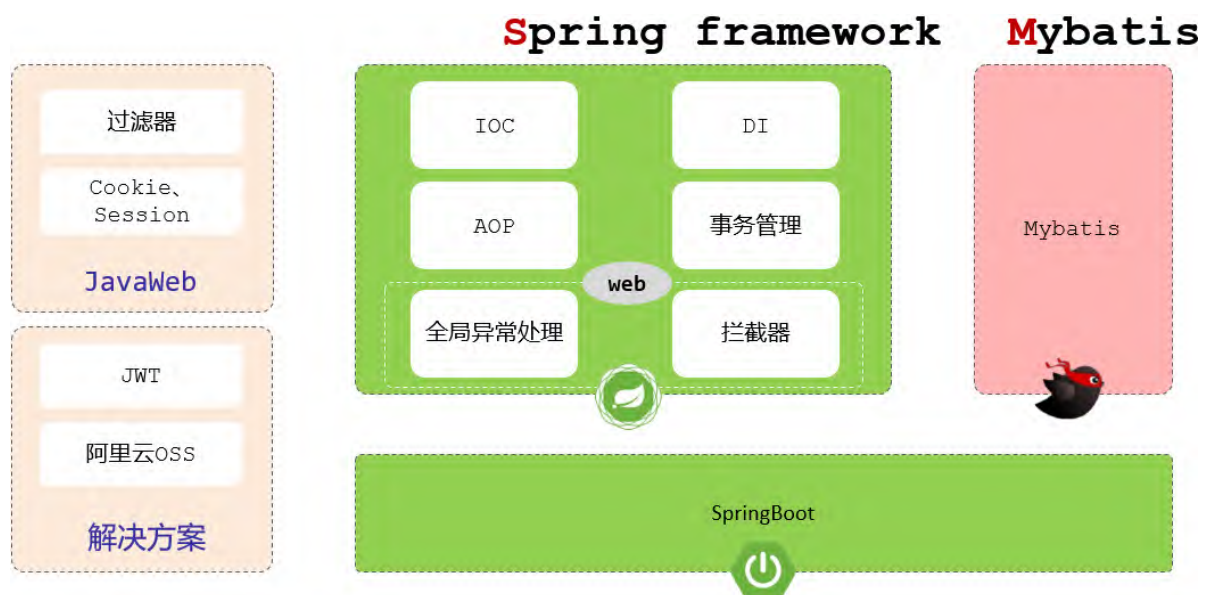
所谓控制反转，指的是将对象创建的控制权由应用程序自身交给外部容器，这个容器就是我们常说的IOC容器或Spring容器。

而DI依赖注入指的是容器为程序提供运行时所需要的资源。

除了IOC与DI我们还讲到了AOP面向切面编程，还有Spring中的事务管理、全局异常处理器，以及传递会话技术Cookie、Session以及新的会话跟踪解决方案JWT令牌，阿里云OSS对象存储服务，以及通过Mybatis持久层架构操作数据库等技术。



我们在学习这些web后端开发技术的时候，我们都是基于主流的SpringBoot进行整合使用的。而SpringBoot又是用来简化开发，提高开发效率的。像过滤器、拦截器、IOC、DI、AOP、事务管理等这些技术到底是哪个框架提供的核心功能？



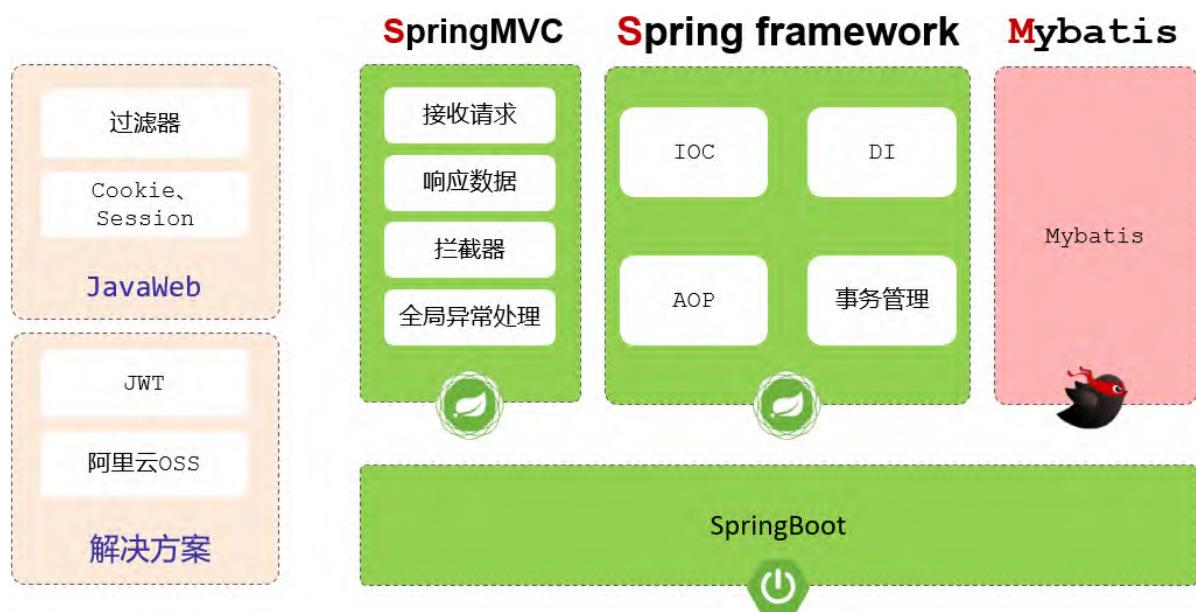
Filter过滤器、Cookie、Session这些都是传统的JavaWeb提供的技术。

JWT令牌、阿里云OSS对象存储服务，是现在企业项目中常见的一些解决方案。

IOC控制反转、DI依赖注入、AOP面向切面编程、事务管理、全局异常处理、拦截器等，这些技术都是 Spring Framework框架当中提供的核心功能。

Mybatis就是一个持久层的框架，是用来操作数据库的。

在Spring框架的生态中，对web程序开发提供了很好的支持，如：全局异常处理器、拦截器这些都是Spring框架中web开发模块所提供的功能，而Spring框架的web开发模块，我们也称为：SpringMVC



SpringMVC不是一个单独的框架，它是Spring框架的一部分，是Spring框架中的web开发模块，是用来简化原始的Servlet程序开发的。

外界俗称的SSM，就是由：SpringMVC、Spring Framework、Mybatis三块组成。

基于传统的SSM框架进行整合开发项目会比较繁琐，而且效率也比较低，所以在现在的企业项目开发当中，基本上都是直接基于SpringBoot整合SSM进行项目开发的。

到此我们web后端开发的内容就已经全部讲解结束了。