```
@Value("${tom.description}")
private String description;
@Test
public void placeholderTest() {
  System.out.println(description);
}
```

上述代码中，通过@Value("${tom.description}")注解将配置文件中的tom.description属性值注入到了对应的description属性中，在测试方法placeholderTest()中对该属性值进行了输出打印。

执行测试方法placeholderTest()，查看控制台输出效果

tom的年龄可能是12

可以看出，测试方法placeholderTest()运行成功，并打印出了属性description的注入内容，该内容与配置文件中配置的属性值保持一致。接着，重复执行测试方法placeholderTest()，查看控制台输出语句中显示的年龄就会在[10,20]之间随机显示

2. SpringBoot原理深入及源码剖析

传统的Spring框架实现一个Web服务，需要导入各种依赖JAR包，然后编写对应的XML配置文件等，相较而言，Spring Boot显得更加方便、快捷和高效。那么，Spring Boot究竟如何做到这些的呢?

接下来分别针对Spring Boot框架的依赖管理、自动配置和执行流程进行深入分析

2.1 依赖管理

问题：（1）为什么导入dependency时不需要指定版本?

在Spring Boot入门程序中，项目pom.xml文件有两个核心依赖，分别是spring-boot-starter-parent和spring-boot-starter-web，关于这两个依赖的相关介绍具体如下：

1. spring-boot-starter-parent依赖

在chapter01项目中的pom.xml文件中找到spring-boot-starter-parent依赖，示例代码如下:

```
<!-- Spring Boot父项目依赖管理 -->
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent<11./artifactId>
    <version>2.2.2.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
```

上述代码中，将spring-boot-starter-parent依赖作为Spring Boot项目的统一父项目依赖管理，并将项目版本号统一为2.2.2.RELEASE，该版本号根据实际开发需求是可以修改的

使用“Ctrl+鼠标左键”进入并查看spring-boot-starter-parent底层源文件，发现spring-boot-starter-parent的底层有一个父依赖spring-boot-dependencies，核心代码具体如下

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-dependencies</artifactId>
  <version>2.2.2.RELEASE</version>
  <relativePath>../../spring-boot-dependencies</relativePath>
</parent>
```

继续查看spring-boot-dependencies底层源文件，核心代码具体如下：

```
<properties>
  <activemq.version>5.15.11</activemq.version>
  ...
    <solr.version>8.2.0</solr.version>
    <mysql.version>8.0.18</mysql.version>
    <kafka.version>2.3.1</kafka.version>
    <spring-amqp.version>2.2.2.RELEASE</spring-amqp.version>
    <spring-restdocs.version>2.0.4.RELEASE</spring-restdocs.version>
    <spring-retry.version>1.2.4.RELEASE</spring-retry.version>
    <spring-security.version>5.2.1.RELEASE</spring-security.version>
    <spring-session-bom.version>Corn-RELEASE</spring-session-bom.version>
    <spring-ws.version>3.0.8.RELEASE</spring-ws.version>
    <sqlite-jdbc.version>3.28.0</sqlite-jdbc.version>
    <sun-mail.version>${jakarta-mail.version}</sun-mail.version>
    <tomcat.version>9.0.29</tomcat.version>
    <thymeleaf.version>3.0.11.RELEASE</thymeleaf.version>
    <thymeleaf-extras-data-attribute.version>2.0.1</thymeleaf-extras-data-
attribute.version>
    ...
  </properties>
```

从spring-boot-dependencies底层源文件可以看出，该文件通过标签对一些常用技术框架的依赖文件进行了统一版本号管理，例如activemq、spring、tomcat等，都有与Spring Boot 2.2.2版本相匹配的版本，这也是pom.xml引入依赖文件不需要标注依赖文件版本号的原因。

需要说明的是，如果pom.xml引入的依赖文件不是 spring-boot-starter-parent管理的，那么在pom.xml引入依赖文件时，需要使用标签指定依赖文件的版本号。

（2）问题2： spring-boot-starter-parent父依赖启动器的主要作用是进行版本统一管理，那么项目运行依赖的JAR包是从何而来的?

2. spring-boot-starter-web依赖

查看spring-boot-starter-web依赖文件源码，核心代码具体如下

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

```
    <version>2.2.2.RELEASE</version>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-json</artifactId>
    <version>2.2.2.RELEASE</version>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-tomcat</artifactId>
    <version>2.2.2.RELEASE</version>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
    <version>2.2.2.RELEASE</version>
    <scope>compile</scope>
    <exclusions>
      <exclusion>
        <artifactId>tomcat-embed-el</artifactId>
        <groupId>org.apache.tomcat.embed</groupId>
      </exclusion>
    </exclusions>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>5.2.2.RELEASE</version>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.2.2.RELEASE</version>
    <scope>compile</scope>
  </dependency>
</dependencies>
```

从上述代码可以发现，spring-boot-starter-web依赖启动器的主要作用是提供Web开发场景所需的底层所有依赖

正是如此，在pom.xml中引入spring-boot-starter-web依赖启动器时，就可以实现Web场景开发，而不需要额外导入Tomcat服务器以及其他Web依赖文件等。当然，这些引入的依赖文件的版本号还是由spring-boot-starter-parent父依赖进行的统一管理。

Spring Boot除了提供有上述介绍的Web依赖启动器外，还提供了其他许多开发场景的相关依赖，我们可以打开Spring Boot官方文档，搜索“Starters”关键字查询场景依赖启动器

Table 13.1. Spring Boot application starters

Name	Description	Pom
spring-boot-starter	Core starter, including auto-configuration support, logging and YAML	Pom
spring-boot-starter-activemq	Starter for JMS messaging using Apache ActiveMQ	Pom
spring-boot-starter-amqp	Starter for using Spring AMQP and Rabbit MQ	Pom
spring-boot-starter-aop	Starter for aspect-oriented programming with Spring AOP and AspectJ	Pom
spring-boot-starter-artemis	Starter for JMS messaging using Apache Artemis	Pom
spring-boot-starter-batch	Starter for using Spring Batch	Pom
spring-boot-starter-cache	Starter for using Spring Framework's caching support	Pom
spring-boot-starter-cloud-connectors	Starter for using Spring Cloud Connectors which simplifies connecting to services in cloud platforms like Cloud Foundry and Heroku	Pom
spring-boot-starter-data-cassandra	Starter for using Cassandra distributed database and Spring Data Cassandra	Pom
spring-boot-starter-data-cassandra-reactive	Starter for using Cassandra distributed database and Spring Data Cassandra Reactive	Pom
spring-boot-starter-data-couchbase	Starter for using Couchbase document-oriented database and Spring Data Couchbase	Pom

列出了Spring Boot官方提供的部分场景依赖启动器，这些依赖启动器适用于不同的场景开发，使用时只需要在pox.xml文件中导入对应的依赖启动器即可。

需要说明的是，Spring Boot官方并不是针对所有场景开发的技术框架都提供了场景启动器，例如数据库操作框架MyBatis、阿里巴巴的Druid数据源等，Spring Boot官方就没有提供对应的依赖启动器。为了充分利用Spring Boot框架的优势，在Spring Boot官方没有整合这些技术框架的情况下，MyBatis、Druid等技术框架所在的开发团队主动与Spring Boot框架进行了整合，实现了各自的依赖启动器，例如mybatis-spring-boot-starter、druid-spring-boot-starter等。我们在pom.xml文件中引入这些第三方的依赖启动器时，切记要配置对应的版本号

2.2 自动配置（启动流程）

概念：能够在我们添加jar包依赖的时候，自动为我们配置一些组件的相关配置，我们无需配置或者只需要少量配置就能运行编写的项目

问题：Spring Boot到底是如何进行自动配置的，都把哪些组件进行了自动配置?

Spring Boot应用的启动入口是@SpringBootApplication注解标注类中的main()方法，@SpringBootApplication能够扫描Spring组件并自动配置Spring Boot

下面，查看@SpringBootApplication内部源码进行分析 ，核心代码具体如下

```
@SpringBootApplication
public class SpringbootDemoApplication {

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

```
@Target({ElementType.TYPE}) //注解的适用范围,Type表示注解可以描述在类、接口、注解或枚举中
@Retention(RetentionPolicy.RUNTIME) //表示注解的生命周期，Runtime运行时
@Documented //表示注解可以记录在javadoc中
@Inherited  //表示可以被子类继承该注解
@SpringBootConfiguration      // 标明该类为配置类
@EnableAutoConfiguration      // 启动自动配置功能
@ComponentScan(                 // 包扫描器
    excludeFilters = {@Filter(
    type = FilterType.CUSTOM,
    classes = {TypeExcludeFilter.class}
), @Filter(
    type = FilterType.CUSTOM,
    classes = {AutoConfigurationExcludeFilter.class}
)}
)
public @interface SpringBootApplication {
...
}
```

从上述源码可以看出，@SpringBootApplication注解是一个组合注解，前面 4 个是注解的元数据信息， 我们主要看后面 3 个注解：@SpringBootConfiguration、@EnableAutoConfiguration、@ComponentScan三个核心注解，关于这三个核心注解的相关说明具体如下：

1. @SpringBootConfiguration注解

@SpringBootConfiguration注解表示Spring Boot配置类。查看@SpringBootConfiguration注解源码，核心代码具体如下。

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Configuration //配置IOC容器
public @interface SpringBootConfiguration {
}
```

从上述源码可以看出，@SpringBootConfiguration注解内部有一个核心注解@Configuration，该注解是Spring框架提供的，表示当前类为一个配置类（XML配置文件的注解表现形式），并可以被组件扫描器扫描。由此可见，@SpringBootConfiguration注解的作用与@Configuration注解相同，都是标识一个可以被组件扫描器扫描的配置类，只不过@SpringBootConfiguration是被Spring Boot进行了重新封装命名而已

2. @EnableAutoConfiguration注解

@EnableAutoConfiguration注解表示开启自动配置功能，该注解是Spring Boot框架最重要的注解，也是实现自动化配置的注解。同样，查看该注解内部查看源码信息，核心代码具体如下

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage                                    // 自动配置包
@Import({AutoConfigurationImportSelector.class}) // 自动配置类扫描导入
public @interface EnableAutoConfiguration {
    String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautoconfiguration";
    Class<?>[] exclude() default {};
    String[] excludeName() default {};
```

可以发现它是一个组合注解，Spring 中有很多以Enable开头的注解，其作用就是借助@Import来收集并注册特定场景相关的bean，并加载到IoC容器。@EnableAutoConfiguration就是借助@Import来收集所有符合自动配置条件的bean定义，并加载到IoC容器。

下面，对这两个核心注解分别讲解：

（1）@AutoConfigurationPackage注解

查看@AutoConfigurationPackage注解内部源码信息，核心代码具体如下：

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Import({Registrar.class})       // 导入Registrar中注册的组件
public @interface AutoConfigurationPackage {
}
```

从上述源码可以看出，@AutoConfigurationPackage注解的功能是由@Import注解实现的，它是spring框架的底层注解，它的作用就是给容器中导入某个组件类，例如@Import(AutoConfigurationPackages.Registrar.class)，它就是将Registrar这个组件类导入到容器中，可查看Registrar类中registerBeanDefinitions方法，这个方法就是导入组件类的具体实现：

```
static class Registrar implements ImportBeanDefinitionRegistrar, DeterminableImports {
    Registrar() {
    }

    public void registerBeanDefinitions(AnnotationMetadata metadata, BeanDefinitionRegistry
registry) {
        AutoConfigurationPackages.register(registry, (
          new AutoConfigurationPackages.PackageImport(metadata)).getPackageName());
    }
```

从上述源码可以看出，在Registrar类中有一个registerBeanDefinitions()方法，使用Debug模式启动项目， 可以看到选中的部分就是com.lagou。也就是说，@AutoConfigurationPackage注解的主要作用就是将主程序类所在包及所有子包下的组件到扫描到spring容器中。

因此 在定义项目包结构时，要求定义的包结构非常规范，项目主程序启动类要定义在最外层的根目录位置，然后在根目录位置内部建立子包和类进行业务开发，这样才能够保证定义的类能够被组件扫描器扫描

（2）**@Import({AutoConfigurationImportSelector.class})：**
将**AutoConfigurationImportSelector**这个类导入到spring容器中，**AutoConfigurationImportSelector**可以帮助springboot应用将所有符合条件的@Configuration配置都加载到当前SpringBoot创建并使用的IoC容器(ApplicationContext)中

继续研究**AutoConfigurationImportSelector**这个类，通过源码分析这个类中是通过selectImports这个方法告诉springboot都需要导入那些组件：

```
@Override
public String[] selectImports(AnnotationMetadata annotationMetadata) {
        if (!isEnabled(annotationMetadata)) {
                return NO_IMPORTS;
        }
//获得自动配置元信息，需要传入beanClassLoader这个类加载器
        AutoConfigurationMetadata autoConfigurationMetadata = AutoConfigurationMetadataLoader
                        .loadMetadata(this.beanClassLoader);

        AutoConfigurationEntry autoConfigurationEntry = getAutoConfigurationEntry(
                        autoConfigurationMetadata, annotationMetadata);
        return StringUtils.toStringArray(autoConfigurationEntry.getConfigurations());
}
```

深入研究loadMetadata方法

```
protected static final String PATH = "META-INF/"
                + "spring-autoconfigure-metadata.properties";  //文件中为需要加载的配置类的类路径

   public static AutoConfigurationMetadata loadMetadata(ClassLoader classLoader) {
           return loadMetadata(classLoader, PATH);
   }

   static AutoConfigurationMetadata loadMetadata(ClassLoader classLoader, String path) {
           try {

      //读取spring-boot-autoconfigure-2.1.5.RELEASE.jar包中spring-autoconfigure-metadata.properties的信息生成url
                Enumeration<URL> urls = (classLoader != null) ? classLoader.getResources(path)
                                : ClassLoader.getSystemResources(path);
                Properties properties = new Properties();

      //解析urls枚举对象中的信息封装成properties对象并加载
                while (urls.hasMoreElements()) {
                        properties.putAll(PropertiesLoaderUtils
                                        .loadProperties(new UrlResource(urls.nextElement())));
                }

      //根据封装好的properties对象生成AutoConfigurationMetadata对象返回
                return loadMetadata(properties);
           }
           catch (IOException ex) {
                throw new IllegalArgumentException(
                                "Unable to load @ConditionalOnClass location [" + path + "]", ex);
           }
   }
```

深入getCandidateConfigurations方法

个方法中有一个重要方法loadFactoryNames，这个方法是让SpringFactoryLoader去加载一些组件的名字。

```
    protected List<String> getCandidateConfigurations(AnnotationMetadata metadata,
                    AnnotationAttributes attributes) {

     /**
     * 这个方法需要传入两个参数getSpringFactoriesLoaderFactoryClass()和getBeanClassLoader()
     * getSpringFactoriesLoaderFactoryClass()这个方法返回的是EnableAutoConfiguration.class
     * getBeanClassLoader()这个方法返回的是beanClassLoader（类加载器）
     */
            List<String> configurations = SpringFactoriesLoader.loadFactoryNames(
                            getSpringFactoriesLoaderFactoryClass(), getBeanClassLoader());
            Assert.notEmpty(configurations,
                            "No auto configuration classes found in META-INF/spring.factories. If you "
                                            + "are using a custom packaging, make sure that file is correct.");
            return configurations;
    }

/**
     * Return the class used by {@link SpringFactoriesLoader} to load configuration
     * candidates.
     * @return the factory class
     */
    protected Class<?> getSpringFactoriesLoaderFactoryClass() {
            return EnableAutoConfiguration.class;
    }


protected ClassLoader getBeanClassLoader() {
            return this.beanClassLoader;
    }
```

继续点开loadFactory方法

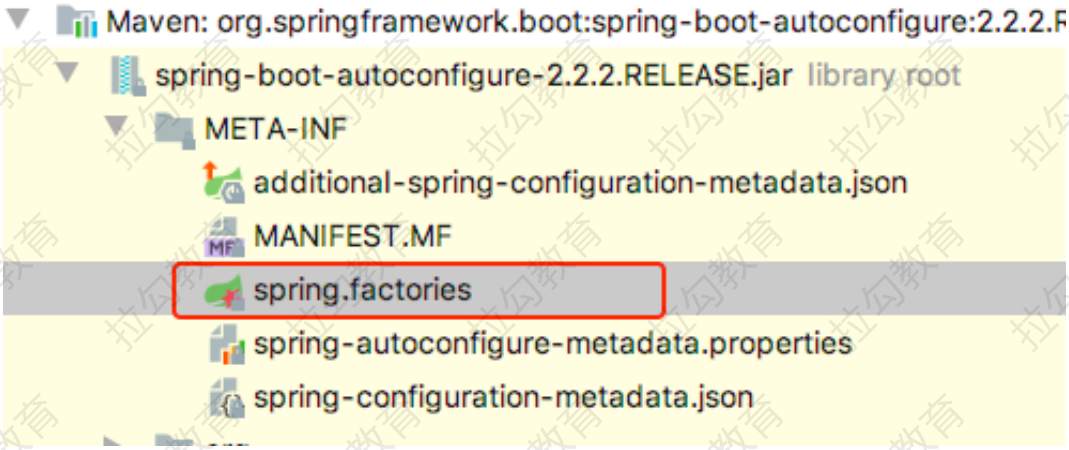
```
 public static List<String> loadFactoryNames(Class<?> factoryClass, @Nullable
ClassLoader classLoader) {

        //获取出入的键
        String factoryClassName = factoryClass.getName();
        return
(List)loadSpringFactories(classLoader).getOrDefault(factoryClassName,
Collections.emptyList());
    }

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

                //如果类加载器不为null，则加载类路径下spring.factories文件，将其中设置
的配置类的全路径信息封装 为Enumeration类对象
                Enumeration<URL> urls = classLoader != null ?
classLoader.getResources("META-INF/spring.factories") :
ClassLoader.getSystemResources("META-INF/spring.factories");
                LinkedMultiValueMap result = new LinkedMultiValueMap();
```

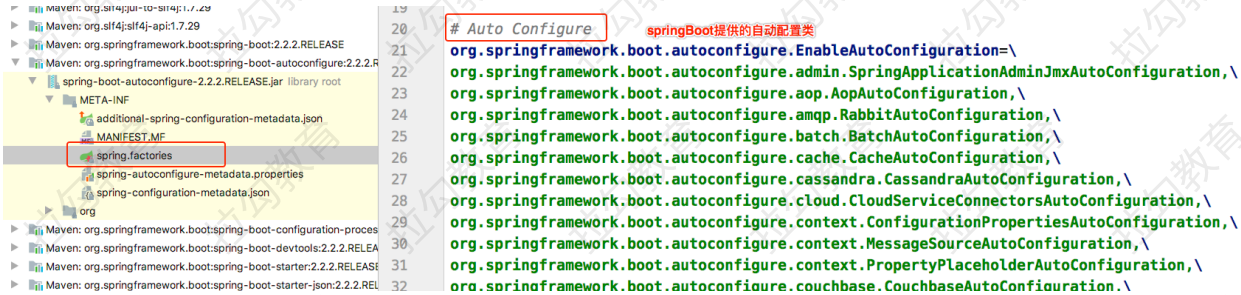
```
                //循环Enumeration类对象，根据相应的节点信息生成Properties对象，通过传
入的键获取值，在将值切割为一个个小的字符串转化为Array，方法result集合中
                while(urls.hasMoreElements()) {
                    URL url = (URL)urls.nextElement();
                    UrlResource resource = new UrlResource(url);
                    Properties properties =
PropertiesLoaderUtils.loadProperties(resource);
                    Iterator var6 = properties.entrySet().iterator();

                    while(var6.hasNext()) {
                        Entry<?, ?> entry = (Entry)var6.next();
                        String factoryClassName =
((String)entry.getKey()).trim();
                        String[] var9 =
StringUtils.commaDelimitedListToStringArray((String)entry.getValue());
                        int var10 = var9.length;

                        for(int var11 = 0; var11 < var10; ++var11) {
                            String factoryName = var9[var11];
                            result.add(factoryClassName, factoryName.trim());
                        }
                    }
                }

                cache.put(classLoader, result);
                return result;
```

会去读取一个 sprin g.factories 的文件，读取不到会表这个错误，我们继续根据会看到，最终路径的长这样，而这个是spring提供的一个工具类

```
public final class SpringFactoriesLoader {
    public static final String FACTORIES_RESOURCE_LOCATION = "META-
INF/spring.factories";
}
```

它其实是去加载一个外部的文件，而这文件是在





@EnableAutoConfiguration就是从classpath中搜寻META-INF/spring.factories配置文件，并将其中org.springframework.boot.autoconfigure.EnableutoConfiguration对应的配置项通过反射（Java Refletion）实例化为对应的标注了@Configuration的JavaConfig形式的配置类，并加载到IOC容器中

以刚刚的项目为例，在项目中加入了Web环境依赖启动器，对应的WebMvcAutoConfiguration自动配置类就会生效，打开该自动配置类会发现，在该配置类中通过全注解配置类的方式对Spring MVC运行所需环境进行了默认配置，包括默认前缀、默认后缀、视图解析器、MVC校验器等。而这些自动配置类的本质是传统Spring MVC框架中对应的XML配置文件，只不过在Spring Boot中以自动配置类的形式进行了预先配置。因此，在Spring Boot项目中加入相关依赖启动器后，基本上不需要任何配置就可以运行程序，当然，我们也可以对这些自动配置类中默认的配置进行更改

总结

因此springboot底层实现自动配置的步骤是：

1. springboot应用启动；
2. @SpringBootApplication起作用；
3. @EnableAutoConfiguration；
4. @AutoConfigurationPackage：这个组合注解主要是@Import(AutoConfigurationPackages.Registrar.class)，它通过将Registrar类导入到容器中，而Registrar类作用是扫描主配置类同级目录以及子包，并将相应的组件导入到springboot创建管理的容器中；
5. @Import(AutoConfigurationImportSelector.class)：它通过将AutoConfigurationImportSelector类导入到容器中，AutoConfigurationImportSelector类作用是通过selectImports方法执行的过程中，会使用内部工具类SpringFactoriesLoader，查找classpath上所有jar包中的META-INF/spring.factories进行加载，实现将配置类信息交给SpringFactory加载器进行一系列的容器创建过程

3. @ComponentScan注解

@ComponentScan注解具体扫描的包的根路径由Spring Boot项目主程序启动类所在包位置决定，在扫描过程中由前面介绍的@AutoConfigurationPackage注解进行解析，从而得到Spring Boot项目主程序启动类所在包的具体位置

总结：

@SpringBootApplication 的注解的功能就分析差不多了， 简单来说就是 3 个注解的组合注解：

```
|- @SpringBootConfiguration
   |- @Configuration  //通过javaConfig的方式来添加组件到IOC容器中
|- @EnableAutoConfiguration
   |- @AutoConfigurationPackage //自动配置包，与@ComponentScan扫描到的添加到IOC
   |- @Import(AutoConfigurationImportSelector.class) //到META-
INF/spring.factories中定义的bean添加到IOC容器中
|- @ComponentScan //包扫描
```

2.3 自定义Stater

SpringBoot starter机制

SpringBoot由众多Starter组成（一系列的自动化配置的starter插件），SpringBoot之所以流行，也是因为starter。

starter是SpringBoot非常重要的一部分，可以理解为一个可拔插式的插件，正是这些starter使得使用某个功能的开发者不需要关注各种依赖库的处理，不需要具体的配置信息，由Spring Boot自动通过classpath路径下的类发现需要的Bean，并织入相应的Bean。

例如，你想使用Reids插件，那么可以使用spring-boot-starter-redis；如果想使用MongoDB，可以使用spring-boot-starter-data-mongodb

为什么要自定义starter

开发过程中，经常会有一些独立于业务之外的配置模块。如果我们将这些可独立于业务代码之外的功能配置模块封装成一个个starter，复用的时候只需要将其在pom中引用依赖即可，SpringBoot为我们完成自动装配

自定义starter的命名规则

SpringBoot提供的starter以`spring-boot-starter-xxx`的方式命名的。官方建议自定义的starter使用`xxx-spring-boot-starter`命名规则。以区分SpringBoot生态提供的starter

整个过程分为两部分：

- 自定义starter
- 使用starter

首先，先完成自定义starter

（1）新建maven jar工程，工程名为zdy-spring-boot-starter，导入依赖：

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-autoconfigure</artifactId>
        <version>2.2.2.RELEASE</version>
    </dependency>
</dependencies>
```

（2）编写javaBean

```
@EnableConfigurationProperties(SimpleBean.class)
@ConfigurationProperties(prefix = "simplebean")
public class SimpleBean {

    private int id;
    private String name;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "SimpleBean{" +
                "id=" + id +
                ", name='" + name + '\'' +
                '}';
    }
}
```

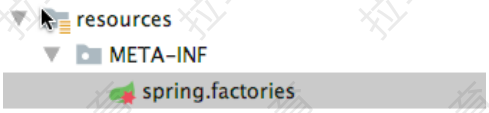
（3）编写配置类MyAutoConfiguration

```
@Configuration
```

```
@ConditionalOnClass //@ConditionalOnClass：当类路径classpath下有指定的类的情况下进行自动配置
public class MyAutoConfiguration {


    static {
        System.out.println("MyAutoConfiguration init....");
    }


    @Bean
    public SimpleBean simpleBean(){
        return new SimpleBean();
    }

}
```

（4）resources下创建/META-INF/spring.factories

注意：META-INF是自己手动创建的目录，spring.factories也是手动创建的文件,在该文件中配置自己的自动配置类




```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.lagou.config.MyAutoConfiguration
```

使用自定义starter

（1）导入自定义starter的依赖

```
<dependency>
   <groupId>com.lagou</groupId>
   <artifactId>zdy-spring-boot-starter</artifactId>
   <version>1.0-SNAPSHOT</version>

</dependency>
```

（2）在全局配置文件中配置属性值

```
simplebean.id=1
simplebean.name=自定义starter
```

（3）编写测试方法

```
//测试自定义starter
@Autowired
private SimpleBean simpleBean;

@Test
public void zdyStarterTest(){
   System.out.println(simpleBean);
}
```

2.4 执行原理

每个Spring Boot项目都有一个主程序启动类，在主程序启动类中有一个启动项目的main()方法，在该方法中通过执行SpringApplication.run()即可启动整个Spring Boot程序。

问题：那么SpringApplication.run()方法到底是如何做到启动Spring Boot项目的呢?

下面我们查看run()方法内部的源码，核心代码具体如下：

```
@SpringBootApplication
public class SpringbootDemoApplication {

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

}
```

```
public static ConfigurableApplicationContext run(Class<?> primarySource,
String... args) {
  return run(new Class[]{primarySource}, args);
}
public static ConfigurableApplicationContext run(Class<?>[] primarySources,
String[] args) {
  return (new SpringApplication(primarySources)).run(args);
}
```

从上述源码可以看出，SpringApplication.run()方法内部执行了两个操作，分别是SpringApplication实例的初始化创建和调用run()启动项目，这两个阶段的实现具体说明如下

1. SpringApplication实例的初始化创建

查看SpringApplication实例对象初始化创建的源码信息，核心代码具体如下

```
public SpringApplication(ResourceLoader resourceLoader, Class...
primarySources) {
  this.sources = new LinkedHashSet();
  this.bannerMode = Mode.CONSOLE;
```

```
    this.logStartupInfo = true;
    this.addCommandLineProperties = true;
    this.addConversionService = true;
    this.headless = true;
    this.registerShutdownHook = true;
    this.additionalProfiles = new HashSet();
    this.isCustomEnvironment = false;
    this.resourceLoader = resourceLoader;
    Assert.notNull(primarySources, "PrimarySources must not be null");
    //把项目启动类.class设置为属性存储起来
    this.primarySources = new LinkedHashSet(Arrays.asList(primarySources));

    //判断当前webApplicationType应用的类型
    this.webApplicationType = WebApplicationType.deduceFromClasspath();

    // 设置初始化器(Initializer),最后会调用这些初始化器
    this.setInitializers(this.getSpringFactoriesInstances(
ApplicationContextInitializer.class));
    // 设置监听器(Listener)

this.setListeners(this.getSpringFactoriesInstances(ApplicationListener.class))
;
    // 用于推断并设置项目main()方法启动的主程序启动类
    this.mainApplicationClass = this.deduceMainApplicationClass();
```

从上述源码可以看出，SpringApplication的初始化过程主要包括4部分，具体说明如下。

（1）this.webApplicationType = WebApplicationType.deduceFromClasspath()

用于判断当前webApplicationType应用的类型。deduceFromClasspath()方法用于查看Classpath类路径下是否存在某个特征类，从而判断当前webApplicationType类型是SERVLET应用（Spring 5之前的传统MVC应用）还是REACTIVE应用（Spring 5开始出现的WebFlux交互式应用）

（2）this.setInitializers(this.getSpringFactoriesInstances(ApplicationContextInitializer.class))

用于SpringApplication应用的初始化器设置。在初始化器设置过程中，会使用Spring类加载器SpringFactoriesLoader从META-INF/spring.factories类路径下的META-INF下的spring.factores文件中获取所有可用的应用初始化器类ApplicationContextInitializer。

（3）this.setListeners(this.getSpringFactoriesInstances(ApplicationListener.class))

用于SpringApplication应用的监听器设置。监听器设置的过程与上一步初始化器设置的过程基本一样，也是使用SpringFactoriesLoader从META-INF/spring.factories类路径下的META-INF下的spring.factores文件中获取所有可用的监听器类ApplicationListener。

（4）this.mainApplicationClass = this.deduceMainApplicationClass()

用于推断并设置项目main()方法启动的主程序启动类

2. 项目的初始化启动

分析完(new SpringApplication(primarySources)).run(args)源码前一部分SpringApplication实例对象的初始化创建后，查看run(args)方法执行的项目初始化启动过程，核心代码具体如下：

```
public ConfigurableApplicationContext run(String... args) {
  StopWatch stopWatch = new StopWatch();
  stopWatch.start();
  ConfigurableApplicationContext context = null;
  Collection<SpringBootExceptionReporter> exceptionReporters = new
ArrayList();
  this.configureHeadlessProperty();
     // 第一步：获取并启动监听器
  SpringApplicationRunListeners listeners = this.getRunListeners(args);
  listeners.starting();
  Collection exceptionReporters;
  try {
    ApplicationArguments applicationArguments =
new DefaultApplicationArguments(args);
    // 第二步：根据SpringApplicationRunListeners以及参数来准备环境
    ConfigurableEnvironment environment =
this.prepareEnvironment(listeners, applicationArguments);
    this.configureIgnoreBeanInfo(environment);
     // 准备Banner打印器 - 就是启动Spring Boot的时候打印在console上的ASCII艺术字体
    Banner printedBanner = this.printBanner(environment);

    // 第三步：创建Spring容器
    context = this.createApplicationContext();
    exceptionReporters =

this.getSpringFactoriesInstances(SpringBootExceptionReporter.class,
new Class[]{ConfigurableApplicationContext.class}, new Object[]{context});

    // 第四步：Spring容器前置处理
this.prepareContext(context, environment, listeners,
applicationArguments, printedBanner);

     // 第五步：刷新容器
    this.refreshContext(context);

     // 第六步：Spring容器后置处理
    this.afterRefresh(context, applicationArguments);
    stopWatch.stop();
    if(this.logStartupInfo) {
      (new StartupInfoLogger(this.mainApplicationClass))
.logStarted(this.getApplicationLog(), stopWatch);
    }
    // 第七步：发出结束执行的事件
    listeners.started(context);

     // 返回容器
```

```
            this.callRunners(context, applicationArguments);
        } catch (Throwable var10) {
            this.handleRunFailure(context, var10, exceptionReporters, listeners);
            throw new IllegalStateException(var10);
        }
        try {
            listeners.running(context);
            return context;
        } catch (Throwable var9) {
            this.handleRunFailure(context, var9, exceptionReporters,
    (SpringApplicationRunListeners)null);
            throw new IllegalStateException(var9);
        }
    }
```

从上述源码可以看出，项目初始化启动过程大致包括以下部分：

- 第一步：获取并启动监听器

this.getRunListeners(args)和listeners.starting()方法主要用于获取SpringApplication实例初始化过程中初始化的SpringApplicationRunListener监听器并运行。

- 第二步：根据SpringApplicationRunListeners以及参数来准备环境

this.prepareEnvironment(listeners, applicationArguments)方法主要用于对项目运行环境进行预设置，同时通过this.configureIgnoreBeanInfo(environment)方法排除一些不需要的运行环境

- 第三步：创建Spring容器

根据webApplicationType进行判断， 确定容器类型，如果该类型为SERVLET类型，会通过反射装载对应的字节码，也就是AnnotationConfigServletWebServerApplicationContext，接着使用之前初始化设置的context（应用上下文环境）、environment（项目运行环境）、listeners（运行监听器）、applicationArguments（项目参数）和printedBanner（项目图标信息）进行应用上下文的组装配置，并刷新配置

- 第四步：Spring容器前置处理

这一步主要是在容器刷新之前的准备动作。设置容器环境，包括各种变量等等，其中包含一个非常关键的操作：将启动类注入容器，为后续开启自动化配置奠定基础

- 第五步：刷新容器

开启刷新spring容器，通过refresh方法对整个IOC容器的初始化(包括bean资源的定位，解析，注册等等)，同时向JVM运行时注册一个关机钩子，在JVM关机时会关闭这个上下文，除非当时它已经关闭

- 第六步：Spring容器后置处理

> 扩展接口，设计模式中的模板方法，默认为空实现。如果有自定义需求，可以重写该方法。比如打印一些启动结束log，或者一些其它后置处理。

- 第七步：发出结束执行的事件

> 获取EventPublishingRunListener监听器，并执行其started方法，并且将创建的Spring容器传进去了，创建一个ApplicationStartedEvent事件，并执行ConfigurableApplicationContext 的publishEvent方法，也就是说这里是在Spring容器中发布事件，并不是在SpringApplication中发布事件，和前面的starting是不同的，前面的starting是直接向SpringApplication中的监听器发布启动事件。

- 第八步：执行Runners

> 用于调用项目中自定义的执行器XxxRunner类，使得在项目启动完成后立即执行一些特定程序。其中，Spring Boot提供的执行器接口有ApplicationRunner 和CommandLineRunner两种，在使用时只需要自定义一个执行器类实现其中一个接口并重写对应的run()方法接口，然后Spring Boot项目启动后会立即执行这些特定程序

下面，通过一个Spring Boot执行流程图，让大家更清晰的知道Spring Boot的整体执行流程和主要启动阶段: