SpringBoot

1、SpringBoot概述

1.1、Spring是什么

Spring是Java企业版（Java Enterprise Edition，JavaEE，也称J2EE）的轻量级代替品。无需开发重量级的Enterprise JavaBean（EJB，企业Java Beans），Spring为企业级Java开发提供了一种相对简单的方法，通过依赖注入和面向切面编程，用简单的Java对象（Plain Old Java Object，POJO）实现了EJB的功能。

1.2、Spring问题分析

1.2.1、Spring优点

方便解耦，简化开发

Spring 是一个大的工厂，将所有对象的创建、依赖关系和维护都交给 Spring 管理。

方便集成各种优秀框架

Spring 不排斥各种优秀的开源框架，其内部对各种优秀框架都直接支持。

降低 Java EE API 的使用难度

Spring 封装了很多 Java EE 开发的 API都提供了封装，使 API 应用的难度大大降低。

方便程序的测试

Spring 支持 JUnit单元测试，可以通过注解方便地测试程序。

AOP 编程的支持

Spring 提供面向切面编程，可以方便地实现对程序进行权限拦截和运行监控等功能。

声明式事务的支持

只需要通过配置就可以完成对事务的管理，而无须手动编程。

1.2.2、Spring缺点

使用门槛升高，入门Spring需要较长时间；

对过时技术兼容，导致复杂度升高；

使用XML进行配置，但已不是流行配置方式；

集成第三方工具时候需要考虑兼容性；

系统启动慢，不具备热部署功能，完全依赖虚拟机或Web服务器的热部署；

1.3、SpringBoot解决Spring问题

SpringBoot对上述Spring的缺点进行的改善和优化，基于约定优于配置的思想，可以让开发人员不必在配置与逻辑业务之间进行思维的切换，全身心的投入到逻辑业务的代码编写中，从而大大提高了开发的效率，一定程度上缩短了项目周期。

1.3、SpringBoot特点

基于Spring的开发提供更快的入门体验。

开箱即用，没有代码生成，也无需XML配置。同时也可以修改默认值来满足特定的需求。

提供了一些大型项目中常见的非功能性特性，如嵌入式服务器、安全、指标，健康检测、外部配置等。

SpringBoot不是对Spring功能上的增强，而是提供了一种快速使用Spring的方式。

1.4、SpringBoot核心功能

起步依赖

起步依赖本质上是一个Maven项目对象模型（Project Object Model，POM），定义了对其他库的传递依赖，这些东西加在一起即支持某项功能。

简单的说，起步依赖就是将具备某种功能的坐标打包到一起，并提供一些默认的功能。

自动配置

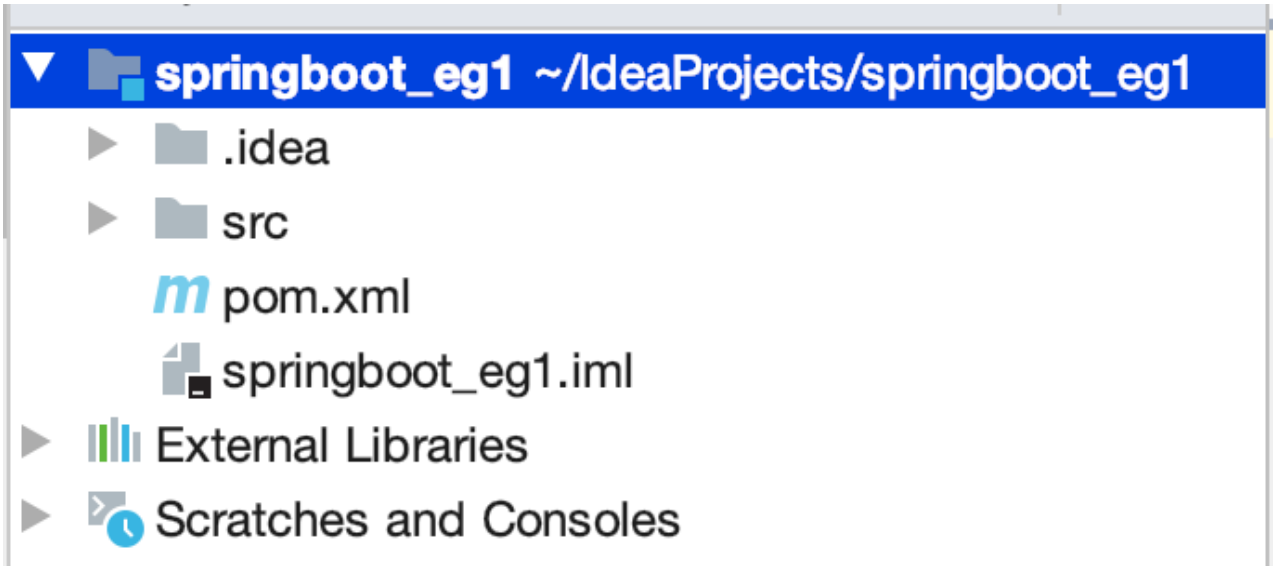
Spring Boot的自动配置是一个运行时（更准确地说，是应用程序启动时）的过程，考虑了众多因素，才决定Spring配置应该用哪个，不该用哪个。该过程是Spring自动完成的。

2、SpringBoot快速开发

2.1、案例开发

2.1.1、创建Maven工程

java工程，项目名字：springboot_eg1



2.1.2、SpringBoot基础依赖

SpringBoot要求，项目要继承SpringBoot的起步依赖spring-boot-starter-parent

```
<!--导入SpringBoot依赖-->
<parent>

<groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-
parent</artifactId>
   <version>2.3.7.RELEASE</version>
</parent>
```

SpringBoot要集成SpringMVC进行Controller的开发，所以项目要导入web的启动依赖spring-boot-starter-web

```
<!--导入Springmvc web 依赖-->
<dependency>

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

2.1.3、SpringBoot引导类

要通过SpringBoot提供的引导类起步，才能完成SpringBoot启动。

@SpringBootApplication--->org.springframework.boot.SpringApplication

SpringApplication--->org.springframework.boot.autoconfigure.SpringBootApplication

```
package com.ww;

import
org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.Spri
ngBootApplication;
//工程声明
@SpringBootApplication
public class MySpringBootApplication {
    public static void main(String[] args)
{
        //工程启动

 SpringApplication.run(MySpringBootApplicat
ion.class);
    }
}
```

2.1.4、Controller编写

控制器编写，完成URL地址访问

```
package com.ww.controller;
```

```
import
org.springframework.stereotype.Controller;
import
org.springframework.web.bind.annotation.Req
uestMapping;
import
org.springframework.web.bind.annotation.Res
ponseBody;
//声明控制器
@Controller
public class HelloController {
    //设置URL地址
    @RequestMapping("/hello")
    //设置return回来的值类型
    @ResponseBody
    public String hello(){
        return "Hello World!";
    }
}
```

2.1.5、启动服务运行结果

执行MySpringBootApplication.java启动类

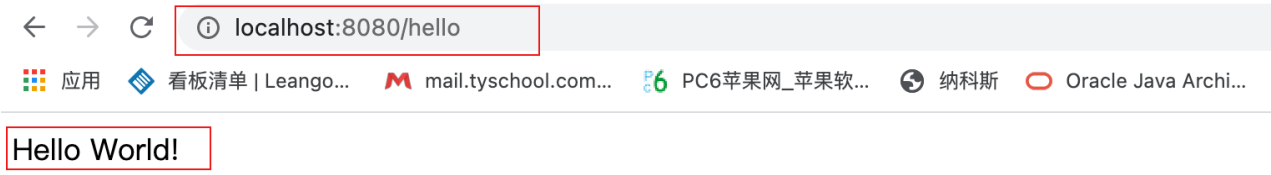
```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
```

```
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.3.7.RELEASE)

2020-12-20 19:43:49.310  INFO 22526 --- [
        main]
com.ww.MySpringBootApplication           :
Starting MySpringBootApplication on
192.168.0.100 with PID 22526
(/Users/apple/IdeaProjects/springboot_eg1/t
arget/classes started by apple in
/Users/apple/IdeaProjects/springboot_eg1)
2020-12-20 19:43:49.312  INFO 22526 --- [
        main]
com.ww.MySpringBootApplication           :
No active profile set, falling back to
default profiles: default
2020-12-20 19:43:49.908  INFO 22526 --- [
        main]
o.s.b.w.embedded.tomcat.TomcatWebServer  :
Tomcat initialized with port(s): 8080
(http)
```

```
2020-12-20 19:43:49.915  INFO 22526 --- [
        main]
o.apache.catalina.core.StandardService   :
Starting service [Tomcat]
2020-12-20 19:43:49.915  INFO 22526 --- [
        main]
org.apache.catalina.core.StandardEngine  :
Starting Servlet engine: [Apache
Tomcat/9.0.41]
2020-12-20 19:43:49.964  INFO 22526 --- [
        main] o.a.c.c.C.[Tomcat].
[localhost].[/]       : Initializing Spring
embedded WebApplicationContext
2020-12-20 19:43:49.964  INFO 22526 --- [
        main]
w.s.c.ServletWebServerApplicationContext :
Root WebApplicationContext: initialization
completed in 612 ms
2020-12-20 19:43:50.077  INFO 22526 --- [
        main]
o.s.s.concurrent.ThreadPoolTaskExecutor  :
Initializing ExecutorService
'applicationTaskExecutor'
```

```
2020-12-20 19:43:50.192  INFO 22526 --- [
        main]
o.s.b.w.embedded.tomcat.TomcatWebServer  :
Tomcat started on port(s): 8080 (http) with
context path ''
2020-12-20 19:43:50.202  INFO 22526 --- [
        main]
com.ww.MySpringBootApplication           :
Started MySpringBootApplication in 1.13
seconds (JVM running for 1.671)
```

打开浏览器访问url地址为：http://localhost:8080/hello



2.2、案例解析

2.2.1、MySpringBootApplication.java解析

@SpringBootApplication：标注SpringBoot的启动类，该注解具备多种功能（后面详细剖析）

SpringApplication.run(MySpringBootApplication.class) 代表运行SpringBoot的启动类，参数为SpringBoot启动类的字节码对象

2.2.2、工程热部署

在开发中我们会反复修改类、页面等资源内容，每次修改后都是需要重新启动服务才能够使修改生效，这样每次启动都很麻烦，浪费了大量的时间?

怎样才可以在修改代码后不重启就能生效?

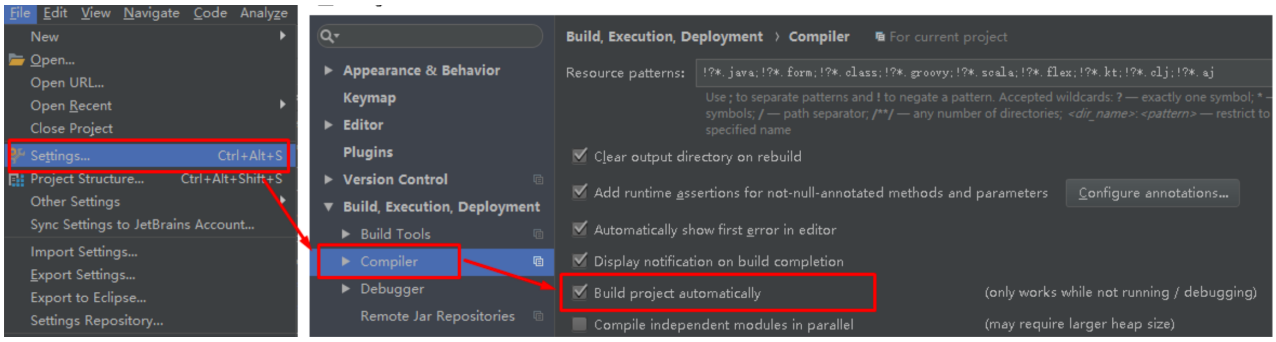
在 pom.xml 中添加如下配置

```
<!--热部署配置-->
<dependency>

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

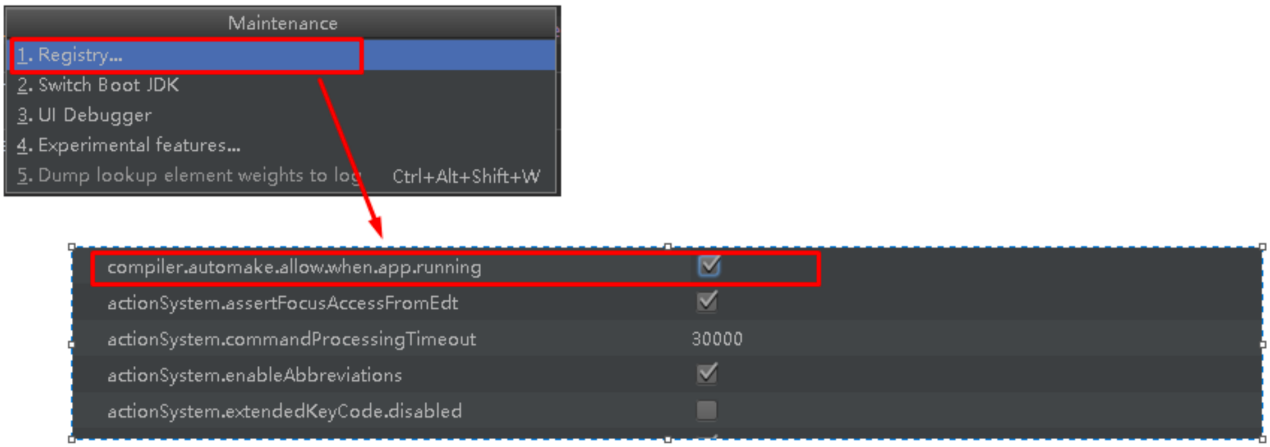
我们称之为热部署。

注意：IDEA进行SpringBoot热部署失败原因

出现这种情况，并不是热部署配置问题，其根本原因是因为Intellij IEDA默认情况下不会自动编译，需要对IDEA进行自动编译的设置，如下：



然后 Shift+Ctrl+Alt+/，选择Registry



3、SpringBoot工作原理

3.1、起步依赖原理

在SpringBoot中，我们引用的依赖变少了，配置文件也不见了，但项目却可以正常运行?

3.1.1、配置依赖

父工程：spring-boot-starter-parent

```
<!--导入SpringBoot依赖-->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.7.RELEASE</version>  Ctrl+单击
</parent>
```

父工程：spring-boot-dependencies

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-dependencies</artifactId>
    <version>2.0.7.RELEASE</version>
    <relativePath>../../spring-boot-dependencies</relativePath>
</parent>
<artifactId>spring-boot-starter-parent</artifactId>
<packaging>pom</packaging>      pom统一控制版本
<name>Spring Boot Starter Parent</name>
```

jar包版本：

```
<properties>
    <activemq.version>5.15.8</activemq.version>
    <antlr2.version>2.7.7</antlr2.version>
    <appengine-sdk.version>1.9.68</appengine-sdk.version>
    <artemis.version>2.4.0</artemis.version>
    <aspectj.version>1.8.13</aspectj.version>
    <assertj.version>3.9.1</assertj.version>
    <atomikos.version>4.0.6</atomikos.version>
    <bitronix.version>2.1.4</bitronix.version>
    <build-helper-maven-plugin.version>3.0.0</build-helper-maven-plugi
    <byte-buddy.version>1.7.11</byte-buddy.version>
    <caffeine.version>2.6.2</caffeine.version>
    <cassandra-driver.version>3.4.0</cassandra-driver.version>
    <classmate.version>1.3.4</classmate.version>
    <commons-codec.version>1.11</commons-codec.version>
    <commons-dbcp2.version>2.2.0</commons-dbcp2.version>
    <commons-lang3.version>3.7</commons-lang3.version>
    <commons-pool.version>1.6</commons-pool.version>
    <commons-pool2.version>2.5.0</commons-pool2.version>
```

我们使用了SpringBoot之后，由于父工程有对版本的统一控制，所以大部分第三方包，我们无需关注版本，个别没有纳入SpringBoot管理的，才需要设置版本号。

3.1.2、场景依赖

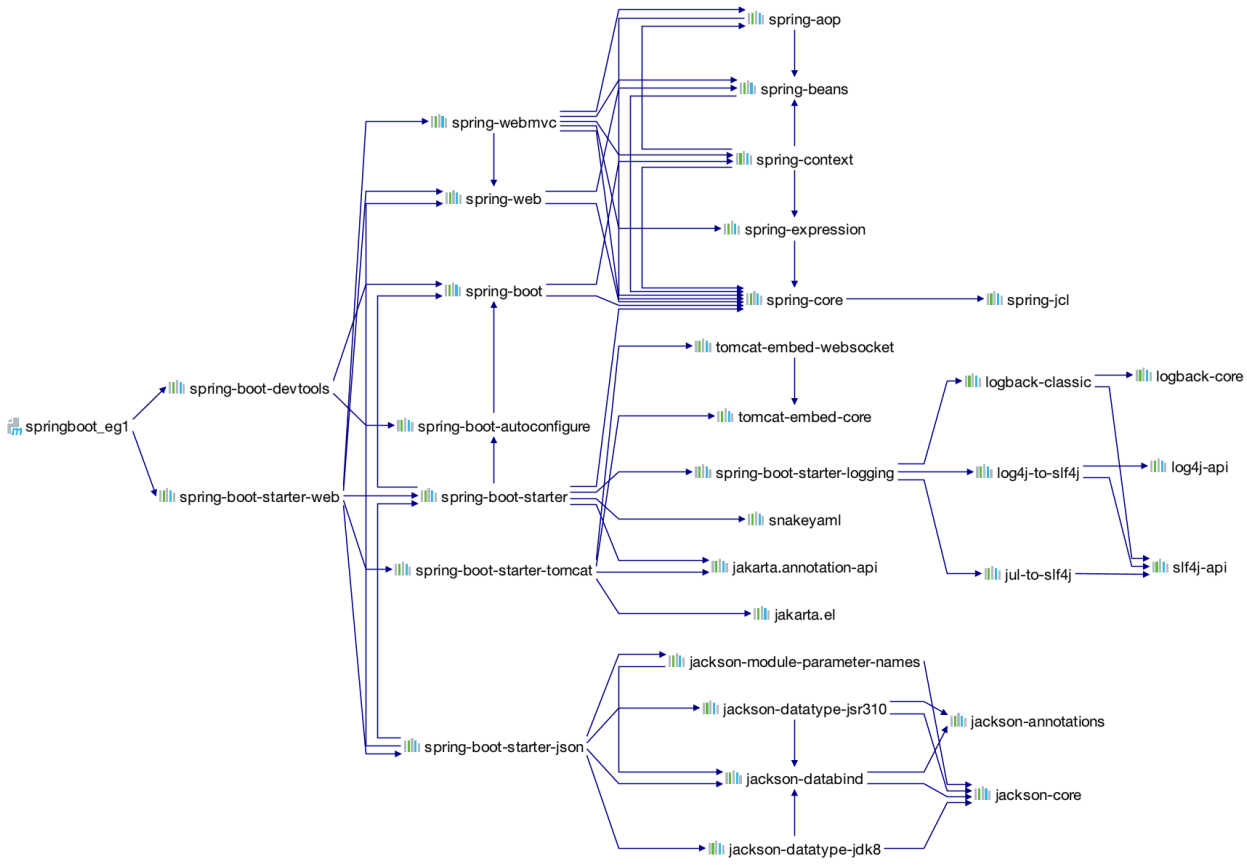
SpringBoot将所有的常见开发功能，分成了一个个场景启动器（starter），这样我们需要开发什么功能，就导入什么场景启动器依赖即可。

需求

我们要完成一个web项目开发

加入场景

```
<!--导入Springmvc web 依赖-->
<dependency>

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



SpringBoot是通过定义各种各样的starter来管理这些依赖的

比如：我们需要开发web的功能，那么引入spring-boot-starter-web

比如：我们需要开发模板页的功能，那么引入spring-boot-starter-thymeleaf

比如：我们需要整合redis，那么引入spring-boot-starter-data-redis

比如：我们需要整合amqp，实现异步消息通信机制，那么引入spring-boot-starter-amqp

等等，就是这么方便

3.2、自动配置原理

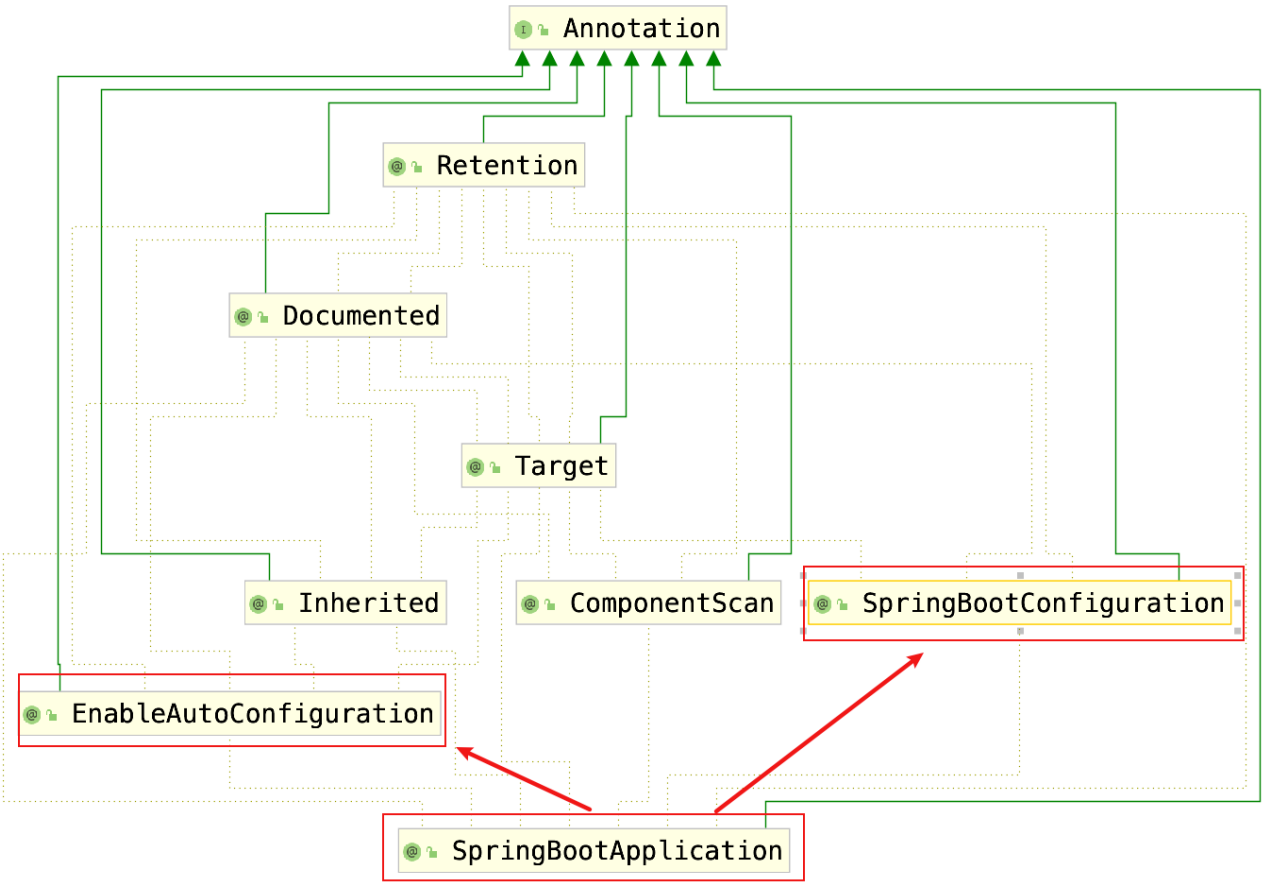
3.2.1、@SpringBootApplication

看看SpringBoot的启动类代码，除了一个关键的注解，其他都是普通的类和main方法定义

```
@SpringBootApplication
public class MySpringBootApplication {
    public static void main(String[] args)
{

 SpringApplication.run(MySpringBootApplicat
ion.class);
    }
}
```

那么，我们来观察下这个注解背后的东西，发现，这个注解是一个复合注解，包含了很多的信息

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(
    excludeFilters = {@Filter(
    type = FilterType.CUSTOM,
    classes = {TypeExcludeFilter.class}
), @Filter(
    type = FilterType.CUSTOM,
    classes =
{AutoConfigurationExcludeFilter.class}
)}
)
public @interface SpringBootApplication {
    @AliasFor(
        annotation =
EnableAutoConfiguration.class
    )
...........
```



@SpringBootConfiguration：等同与@Configuration，既标注该类是Spring的一个配置类

@EnableAutoConfiguration：SpringBoot自动配置功能开启

3.2.2、@SpringBootConfiguration

```
package org.springframework.boot;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
```

```
import java.lang.annotation.Retention;
import
java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import
org.springframework.context.annotation.Conf
iguration;
import
org.springframework.core.annotation.AliasFo
r;

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

我们可以看到，内部是包含了@Configuration，这是Spring定义配置类的注解，而@Configuration实际上就是一个@Component，表示一个受Spring管理的组件。

@SpringBootConfiguration这个注解只是更好区分这是SpringBoot的配置注解，本质还是用了Spring提供的@Configuration注解。

3.2.3、@EnableAutoConfiguration

注解的作用是告诉SpringBoot开启自动配置功能。

```
package
org.springframework.boot.autoconfigure;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import
java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import
org.springframework.context.annotation.Impo
rt;

@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
```

```
@Import({AutoConfigurationImportSelector.class})
public @interface EnableAutoConfiguration {
    String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautoconfiguration";

    Class<?>[] exclude() default {};

    String[] excludeName() default {};
}
```

先来分析@AutoConfigurationPackage

观察其内部实现，内部是采用了@Import，来给容器导入一个Registrar组件

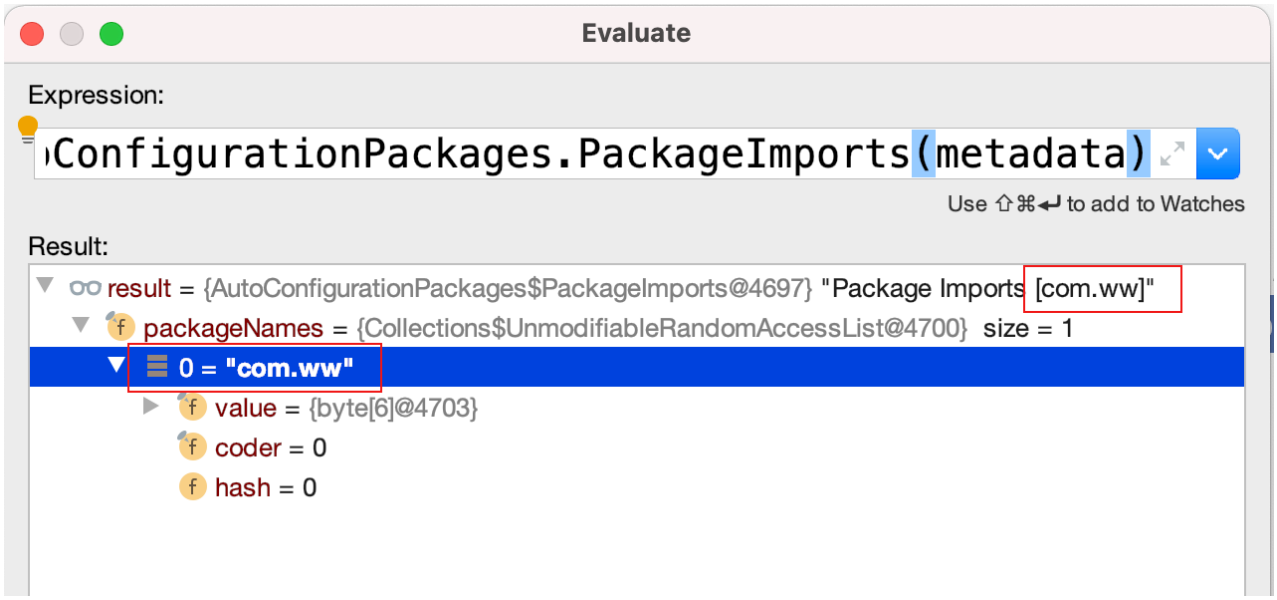
```
import ...

@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Import({Registrar.class})
public @interface AutoConfigurationPackage {
    String[] basePackages() default {};

    Class<?>[] basePackageClasses() default {};
}
```

Registrar.class里的内容

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

        public void
registerBeanDefinitions(AnnotationMetadata
metadata, BeanDefinitionRegistry registry)
{

 AutoConfigurationPackages.register(registr
y, (String[])(new
AutoConfigurationPackages.PackageImports(me
tadata)).getPackageNames().toArray(new
String[0]));
        }

        public Set<Object>
determineImports(AnnotationMetadata
metadata) {
            return
Collections.singleton(new
AutoConfigurationPackages.PackageImports(me
tadata));
        }
```

```
    }
```

new AutoConfigurationPackages.PackageImports(metadata)测试导入包是?



通过源码跟踪，我们知道，程序运行到这里，会去加载启动类所在包下面的所有类

这就是为什么，默认情况下，我们要求定义的类，比如controller，service必须在启动类的同级目录或子级目录的原因

再来分析@Import({AutoConfigurationImportSelector.class})

```
public class
AutoConfigurationImportSelector implements
DeferredImportSelector,
BeanClassLoaderAware, ResourceLoaderAware,
BeanFactoryAware, EnvironmentAware, Ordered
{
.............
    public String[]
selectImports(AnnotationMetadata
annotationMetadata) {
        if
(!this.isEnabled(annotationMetadata)) {
            return NO_IMPORTS;
        } else {

 AutoConfigurationImportSelector.AutoConfig
urationEntry autoConfigurationEntry =
this.getAutoConfigurationEntry(annotationMe
tadata);
            return
StringUtils.toStringArray(autoConfiguration
Entry.getConfigurations());
        }
    }
```

```
  protected
AutoConfigurationImportSelector.AutoConfigu
rationEntry
getAutoConfigurationEntry(AnnotationMetadat
a annotationMetadata) {
        if
(!this.isEnabled(annotationMetadata)) {
            return EMPTY_ENTRY;
        } else {
            AnnotationAttributes attributes
= this.getAttributes(annotationMetadata);
            List<String> configurations =
this.getCandidateConfigurations(annotationM
etadata, attributes);
            configurations =
this.removeDuplicates(configurations);
            Set<String> exclusions =
this.getExclusions(annotationMetadata,
attributes);

 this.checkExcludedClasses(configurations,
exclusions);

 configurations.removeAll(exclusions);
```

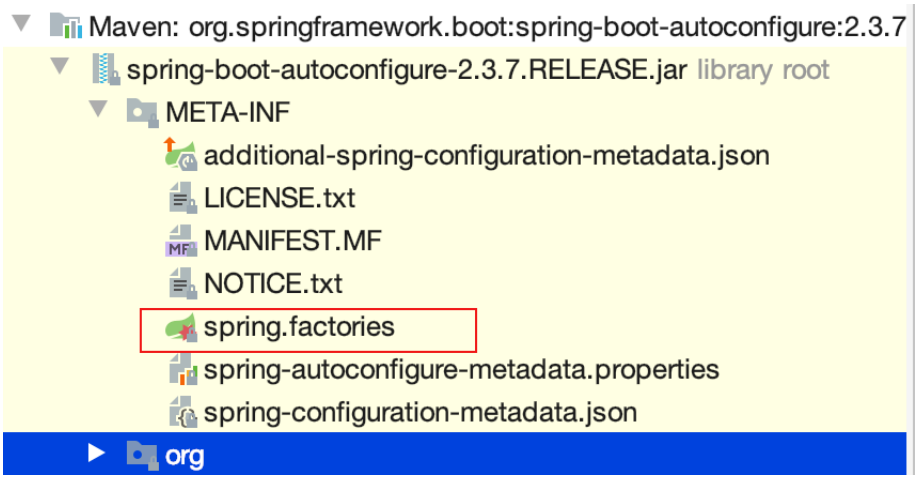
```
            configurations =
this.getConfigurationClassFilter().filter(c
onfigurations);

 this.fireAutoConfigurationImportEvents(con
figurations, exclusions);
            return new
AutoConfigurationImportSelector.AutoConfigu
rationEntry(configurations, exclusions);
        }
    }

.............
```

该selectImports调用了loadMetadata静态方法。该静态方法会扫描所有具有META-INF/spring.factories的jar包。

AutoConfigurationMetadataLoader

```
final class AutoConfigurationMetadataLoader
{
.............
    static AutoConfigurationMetadata
loadMetadata(ClassLoader classLoader) {
        return loadMetadata(classLoader,
"META-INF/spring-autoconfigure-
metadata.properties");
    }
.............
```

spring-boot-autoconfigure-x.x.x.x.jar里就有一个这样的spring.factories文件。



spring.factories

spring.factories文件也是一组一组的key=value的形式，其中一个key是EnableAutoConfiguration类的全类名，而它的value是一个xxxxAutoConfiguration的类名的列表，这些类名以逗号分隔

```
.............
# Auto Configuration Import Listeners
org.springframework.boot.autoconfigure.Auto
ConfigurationImportListener=\
org.springframework.boot.autoconfigure.cond
ition.ConditionEvaluationReportAutoConfigur
ationImportListener

# Auto Configuration Import Filters
org.springframework.boot.autoconfigure.Auto
ConfigurationImportFilter=\
org.springframework.boot.autoconfigure.cond
ition.OnBeanCondition,\
org.springframework.boot.autoconfigure.cond
ition.OnClassCondition,\
org.springframework.boot.autoconfigure.cond
ition.OnWebApplicationCondition
.............
```

这个@EnableAutoConfiguration注解通过@SpringBootApplication被间接的标记在了Spring Boot的启动类上。在SpringApplication.run(...)的内部就会执行selectImports()方法，找到所有JavaConfig自动配置类的全限定名对应的class，然后将所有自动配置类加载到Spring容器中。

4、SpringBoot的配置文件

4.1、配置文件分类

SpringBoot是基于约定的，所以很多配置都有默认值，但如果想使用自己的配置替换默认配置的话，就可以使用application.properties或者application.yml（application.yaml）进行配置。

SpringBoot默认会从Resources目录下加载application.properties或application.yml（application.yaml）文件。

其中，application.properties文件是键值对类型的文件，之前一直在使用，所以此处不在对properties文件的格式进行阐述。除了properties文件外，SpringBoot还可以使用yml文件进行配置，下面对yml文件进行讲解。

4.2、yml配置文件

4.2.1、yml概述

YML文件格式是YAML (YAML Aint Markup Language)编写的文件格式，YAML是一种直观的能够被电脑识别的的数据数据序列化格式，并且容易被人类阅读，容易和脚本语言交互的，可以被支持YAML库的不同的编程语言程序导入，比如： C/C++, Ruby, Python, Java, Perl, C#, PHP等。YML文件是以数据为核心的，比传统的xml方式更加简洁。

YML文件的扩展名可以使用.yml或者.yaml。

yml设计目的：

容易阅读

可用于不同程序间的数据交换

适合描述程序所使用的数据结构，特别是脚本语言

丰富的表达能力与可扩展性

易于使用

yml基本语法：

大小写敏感

使用缩进表示层级关系

缩进时不允许使用Tab键，只允许使用空格。

缩进的空格数目不重要，只要相同层级的元素左侧对齐即可

yml基础规范：

文档使用 Unicode 编码作为字符标准编码，例如 UTF-8

使用“#”来表示注释内容

使用空格作为嵌套缩进工具。通常建议使用两个空格缩进，不建议使用 tab （甚至不支持）

4.2.2、基础数据

语法：

key: value

案例：

```
name: zhangsan
```

注意：

value前面一定要有一个空格

4.2.3、对象/Map数据

语法：

key:

key1: value1

key2: value2

或者：

key: {key1: value1,key2: value2}

案例：

```yaml
person:
  name: zhangsan
  age: 19
  score: 90
```

或

```yaml
person: {name: zhangsan, age: 19, score: 90
}
```

注意：

key1前面的空格个数不限定，在yml语法中，相同缩进代表同一个级别

4.2.4、数组/List/Set数据

语法：

key:

- value1

- value2

或者：

key: [value1,value2]

案例：

```yaml
city:
  - chongqing
  - beijing
  - hebei
  - tianjin
```

或

```yaml
city: [chongqing,beijing,hebei,tianjin]
```

集合中的元素形式（对象）

```yaml
persons:
  - name: zhangsan
    age: 18
    score: 90
  - name: lisi
    age: 19
    score: 60
  - name: wangwu
    age: 20
    score: 68
```

或

```yaml
persons: [{name: zhangsan,age: 18,score: 90},{name: lisi,age: 19,score: 60},{name: wangwu,age: 20,score: 68}]
```

注意：

value1与之间的 - 之间存在一个空格

4.3、配置信息查询

4.3.1、配置查询位置

SpringBoot的配置文件，主要的目的就是对配置信息进行修改的，但在配置时的key从哪里去查询

呢?

https://docs.spring.io/spring-boot/docs/2.0.1.RELEASE/reference/htmlsingle/#common-application-properties

```
# ===================================================================
# COMMON SPRING BOOT PROPERTIES
#
# This sample file is provided as a guideline. Do NOT copy it in its
# entirety to your own application.                  ^^^
# ===================================================================


# ----------------------------------------
# CORE PROPERTIES
# ----------------------------------------
debug=false # Enable debug logs.
trace=false # Enable trace logs.

# LOGGING
logging.config= # Location of the logging configuration file. For instance, `classpath:logback.xml` for Logback.
logging.exception-conversion-word=%wEx # Conversion word used when logging exceptions.
logging.file= # Log file name (for instance, `myapp.log`). Names can be an exact location or relative to the current directory.
logging.file.max-history=0 # Maximum of archive log files to keep. Only supported with the default logback setup.
logging.file.max-size=10MB # Maximum log file size. Only supported with the default logback setup.
logging.level.*= # Log levels severity mapping. For instance, `logging.level.org.springframework=DEBUG`.
logging.path= # Location of the log file. For instance, `/var/log`.
logging.pattern.console= # Appender pattern for output to the console. Supported only with the default Logback setup.
logging.pattern.dateformat=yyyy-MM-dd HH:mm:ss.SSS # Appender pattern for log date format. Supported only with the default Logback s
logging.pattern.file= # Appender pattern for output to a file. Supported only with the default Logback setup.
logging.pattern.level=%5p # Appender pattern for log level. Supported only with the default Logback setup.
logging.register-shutdown-hook=false # Register a shutdown hook for the logging system when it is initialized.

# AOP
spring.aop.auto=true # Add @EnableAspectJAutoProxy.
spring.aop.proxy-target-class=true # Whether subclass-based (CGLIB) proxies are to be created (true), as opposed to standard Java in

# IDENTITY (ContextIdApplicationContextInitializer)
spring.application.name= # Application name.
```

4.3.2、常见配置

```
# LOGGING
logging.config= # Location of the logging configuration file. For instance, `classpath:logback.xml` for Logback.
logging.exception-conversion-word=%wEx # Conversion word used when logging exceptions.
```

```
logging.file= # Log file name (for
instance, `myapp.log`). Names can be an
exact location or relative to the current
directory.
logging.file.max-history=0 # Maximum of
archive log files to keep. Only supported
with the default logback setup.
logging.file.max-size=10MB # Maximum log
file size. Only supported with the default
logback setup.
logging.level.*= # Log levels severity
mapping. For instance,
`logging.level.org.springframework=DEBUG`.
logging.path= # Location of the log file.
For instance, `/var/log`.
logging.pattern.console= # Appender pattern
for output to the console. Supported only
with the default Logback setup.
logging.pattern.dateformat=yyyy-MM-dd
HH:mm:ss.SSS # Appender pattern for log
date format. Supported only with the
default Logback setup.
logging.pattern.file= # Appender pattern
for output to a file. Supported only with
the default Logback setup.
```

```
logging.pattern.level=%5p # Appender
pattern for log level. Supported only with
the default Logback setup.
logging.register-shutdown-hook=false #
Register a shutdown hook for the logging
system when it is initialized.

# AOP
spring.aop.auto=true # Add
@EnableAspectJAutoProxy.
spring.aop.proxy-target-class=true #
Whether subclass-based (CGLIB) proxies are
to be created (true), as opposed to
standard Java interface-based proxies
(false).

# EMBEDDED SERVER CONFIGURATION
(ServerProperties)
server.address= # Network address to which
the server should bind.
server.port=8080 # Server HTTP port.
server.server-header= # Value to use for
the Server response header (if empty, no
header is sent).
```

```
server.use-forward-headers= # Whether X-
Forwarded-* headers should be applied to
the HttpRequest.
server.servlet.context-parameters.*= #
Servlet context init parameters.
server.servlet.context-path= # Context path
of the application.

........
```

4.3.3、properties到yaml的转换

application.properties

```
server.address= # Network address to which
the server should bind.
server.port=8080 # Server HTTP port.
server.servlet.context-path= # Context path
of the application.
```

application.yml

```
server:
  address: localhost
  port: 8080
  servlet:
    context-path: /demo
```

4.4、配置文件详解

在 Spring Boot 中有两种上下文，一种是 bootstrap配置文件，另外一种是 application配置文件。这两种配置文件的区别是：

4.4.1、加载顺序

若application.yml 和bootstrap.yml 在同一目录下：

bootstrap.yml 先加载 ， application.yml后加载

bootstrap.yml 用于应用程序上下文的引导阶段。bootstrap.yml 由父Spring ApplicationContext加载。

4.4.2、配置内容区别

bootstrap.yml 和 application.yml 都可以用来配置参数。

bootstrap.yml 用来程序引导时执行，应用于更加早期配置信息读取。可以理解成系统级别的一些参数配置，这些参数一般是不会变动的。一旦bootStrap.yml 被加载，则内容不会被覆盖。

application.yml 可以用来定义应用级别的， 应用程序特有配置信息，可以用来配置后续各个模块中需使用的公共参数等。

4.4.3、覆盖问题

启动上下文时，Spring Cloud会创建一个 Bootstrap Context，作为 Spring 应用的 Application Context 的父上下文。

初始化的时候，Bootstrap Context 负责从外部源加载配置属性并解析配置。这两个上下文共享一个从外部获取的 Environment（生态环境）。Bootstrap 属性有高优先级，默认情况下，它们不会被本地配置覆盖。

也就是说**如果加载的 application.yml 的内容标签与 bootstrap 的标签一致，application 也不会覆盖 bootstrap，而 application.yml 里面的内容可以动态替换**。

4.5、配置文件获取

4.5.1、@Value

通过@Value注解将配置文件中的值映射到一个Spring管理的Bean的字段上，所在包：org.springframework.beans.factory.annotation.Value

案例：

application.yml

```
person:
  name: zhangsan
  age: 19
  score: 90
```

注意：

配置文件application.yml在resources目录下

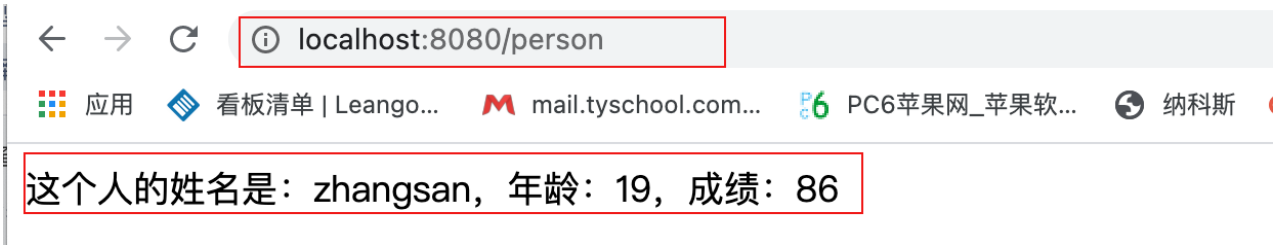
PersonController.java

```
package com.ww.controller;

import
org.springframework.beans.factory.annotatio
n.Value;
import
org.springframework.stereotype.Controller;
```

```
import
org.springframework.web.bind.annotation.Req
uestMapping;
import
org.springframework.web.bind.annotation.Res
ponseBody;

@Controller
public class PersonController {
    @Value("${person.name}")
    private String name;
    @Value("${person.age}")
    private int age;
    @Value("${person.score}")
    private int score;

    @RequestMapping("/person")
    @ResponseBody
    public String person(){
        return "这个人的姓名是: "+name+", 年
龄: "+age+", 成绩: "+score;
    }
}
```

启动服务，测试：

http://localhost:8080/person



4.5.2、@ConfigurationProperties

通过@ConfigurationProperties(prefix="配置文件中的key的前缀")可以将配置文件中的配置自动与实体进行映射，所在包：org.springframework.boot.context.properties.ConfigurationProperties

案例：

application.yml

```yaml
person:
  name: zhangsan
  age: 19
  score: 90
```

注意：

配置文件application.yml在resources目录下

Person01Controller.java

```
package com.ww.controller;
```

```
import
org.springframework.boot.context.properties
.ConfigurationProperties;
import
org.springframework.stereotype.Controller;
import
org.springframework.web.bind.annotation.Req
uestMapping;
import
org.springframework.web.bind.annotation.Res
ponseBody;

@Controller
@ConfigurationProperties(prefix = "person")
public class Person01Controller {
    private String name;
    private int age;
    private int score;

    @RequestMapping("/person01")
    @ResponseBody
    public String person01(){
        return "这个人的姓名是: "+name+", 年
龄: "+age+", 成绩: "+score;
    }
```

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

    public void setAge(int age) {
        this.age = age;
    }

    public void setScore(int score) {
        this.score = score;
    }
}
```

注意：

使用@ConfigurationProperties方式可以进行配置文件与实体字段的自动映射，但需要字段必须提供set方法才可以，而使用@Value注解修饰的字段不需要提供set方法

启动服务，测试：

http://localhost:8080/person01