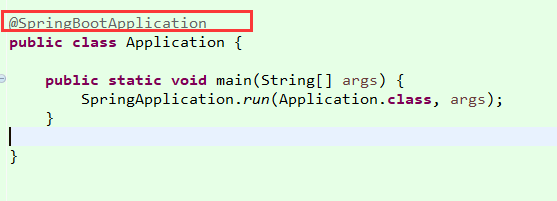
# 启动入口类剖析（spring boot 2.1.4）

启动springboot的项目，主要使用SpringApplication这个类。

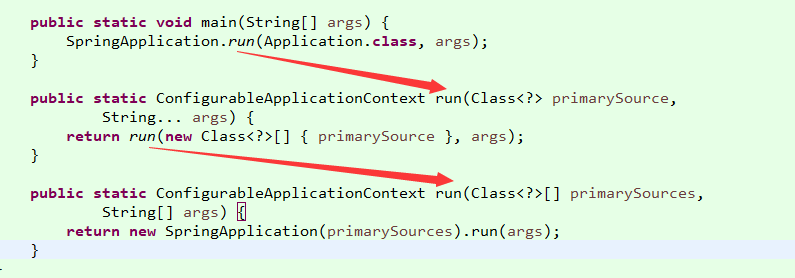


我们看下，run方法里面是实例化SpringApplication类。

第一个参数 primarySource：加载的主要资源类

第二个参数 args：传递给应用的应用参数。

看到这里，我看已经知道，分析启动，需要分为两步，实例化对象，以及调用这个对象的run方法。



## 实例化对象

public SpringApplication(ResourceLoader resourceLoader, Class<?>... primarySources) {

// 1、资源初始化资源加载器为 null

this.resourceLoader = resourceLoader;

// 2、断言主要加载资源类不能为 null，否则报错

Assert.notNull(primarySources, "PrimarySources must not be null");

// 3、初始化主要加载资源类集合并去重

this.primarySources = new LinkedHashSet<>(Arrays.asList(primarySources));

// 4、推断当前 WEB 应用类型

this.webApplicationType = WebApplicationType.deduceFromClasspath();

// 5、设置应用上线文初始化器

setInitializers((Collection) getSpringFactoriesInstances(

ApplicationContextInitializer.class));

// 6、设置监听器

setListeners((Collection) getSpringFactoriesInstances(ApplicationListener.class));

// 7、推断主入口应用类

this.mainApplicationClass = deduceMainApplicationClass();

}

### 推断当前 WEB 应用类型

根据类路径下是否有对应项目类型的类推断出不同的应用类型

public enum WebApplicationType {

/\*\* 非 WEB 项目 \*/

NONE,

/\*\* servlet WEB 项目 \*/

SERVLET,

/\*\* 响应式 WEB 项目 \*/

REACTIVE;

/\*\* 响应式 WEB 项目 启动类 \*/

private static final String WEBFLUX\_INDICATOR\_CLASS = "org."

+ "springframework.web.reactive.DispatcherHandler";

/\*\* servlet WEB 项目 启动类 \*/

private static final String[] SERVLET\_INDICATOR\_CLASSES = { "javax.servlet.Servlet",

"org.springframework.web.context.ConfigurableWebApplicationContext" };

/\*\* servlet mvc 项目 启动类\*/

private static final String WEBMVC\_INDICATOR\_CLASS = "org.springframework."

+ "web.servlet.DispatcherServlet";

/\*\* WEB 项目启动的容器 \*/

private static final String JERSEY\_INDICATOR\_CLASS = "org.glassfish.jersey.servlet.ServletContainer";

private static final String SERVLET\_APPLICATION\_CONTEXT\_CLASS = "org.springframework.web.context.WebApplicationContext";

private static final String REACTIVE\_APPLICATION\_CONTEXT\_CLASS = "org.springframework.boot.web.reactive.context.ReactiveWebApplicationContext";

/\*\* 判断web项目的类型 \*/

static WebApplicationType deduceFromClasspath() {

if (ClassUtils.isPresent(WEBFLUX\_INDICATOR\_CLASS, null)

&& !ClassUtils.isPresent(WEBMVC\_INDICATOR\_CLASS, null)

&& !ClassUtils.isPresent(JERSEY\_INDICATOR\_CLASS, null)) {

return WebApplicationType.REACTIVE;

}

for (String className : SERVLET\_INDICATOR\_CLASSES) {

if (!ClassUtils.isPresent(className, null)) {

return WebApplicationType.NONE;

}

}

// 默认是servlet项目

return WebApplicationType.SERVLET;

}

}

### 初始化应用上下文

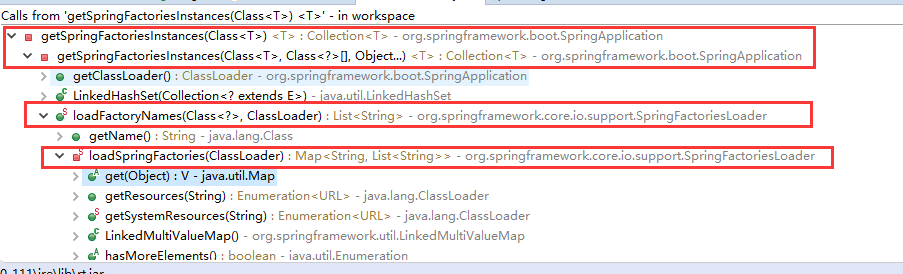
setInitializers(

(Collection) getSpringFactoriesInstances(

ApplicationContextInitializer.class)

);

调用关系如下



ApplicationContextInitializer 的作用是什么？源码如下

用来初始化指定的 Spring 应用上下文，如注册属性资源、激活 Profiles 等

public interface ApplicationContextInitializer

<C extends ConfigurableApplicationContext> {

void initialize(C applicationContext);

}

默认spring boot项目初始化加载时候的ApplicationContextInitializer 如下：

org.springframework.boot.context.config.DelegatingApplicationContextInitializer

org.springframework.boot.autoconfigure.SharedMetadataReaderFactoryContextInitializer

org.springframework.boot.context.ContextIdApplicationContextInitializer

org.springframework.boot.context.ConfigurationWarningsApplicationContextInitializer

org.springframework.boot.rsocket.context.RSocketPortInfoApplicationContextInitializer

org.springframework.boot.web.context.ServerPortInfoApplicationContextInitializer

org.springframework.boot.autoconfigure.logging.ConditionEvaluationReportLoggingListener

来看下 setInitializers 方法源码，其实就是初始化一个 ApplicationContextInitializer 应用上下文初始化器实例的集合。这样其实，重点还是在getSpringFactoriesInstances方法里面

public void setInitializers(

Collection<? extends ApplicationContextInitializer<?>> initializers) {

this.initializers = new ArrayList<>();

this.initializers.addAll(initializers);

}

再来看下这个初始化 getSpringFactoriesInstances 方法和相关的源码。

根据类路径下的 META-INF/spring.factories 文件解析并获取配置中的 ApplicationContextInitializer接口的实现类

private <T> Collection<T> getSpringFactoriesInstances(Class<T> type,

Class<?>[] parameterTypes, Object... args) {

//1. 获取当前线程上下文类加载器,类似这样的方式 Thread.currentThread().getContextClassLoader()

ClassLoader classLoader = getClassLoader();

//2. 获取 ApplicationContextInitializer 的实例名称集合并去重（是类的名字）

Set<String> names = new LinkedHashSet<>(SpringFactoriesLoader.loadFactoryNames(type, classLoader));

//3.根据以上类路径创建初始化器实例列表

List<T> instances = createSpringFactoriesInstances(type, parameterTypes, classLoader, args, names);

//4.初始化器实例列表排序

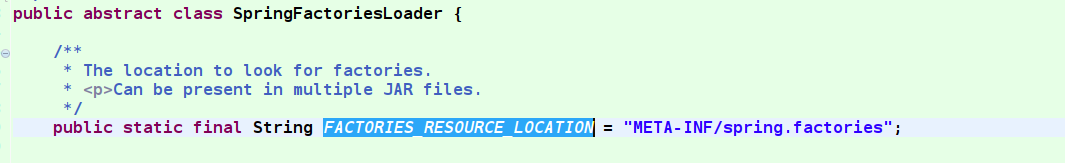
AnnotationAwareOrderComparator.sort(instances);

return instances;

}

第二步中的SpringFactoriesLoader类的loadSpringFactories方法，加载FACTORIES\_RESOURCE\_LOCATION



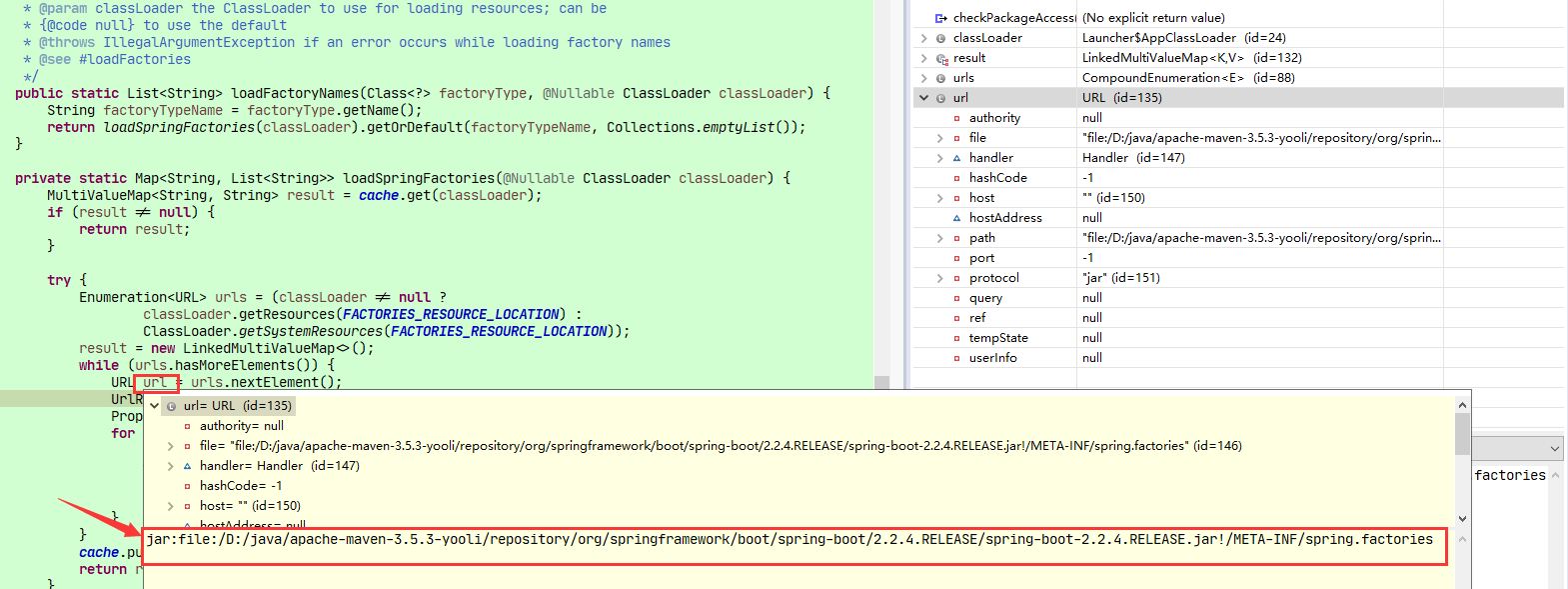
FACTORIES\_RESOURCE\_LOCATION指的是classpath下的所有META-INF/spring.factories文件。

这个地方使用的是AppClassLoader，因为classLoader已经将项目加载到JVM中了，所有通过getResources方法能获得jar和classes下的知道路径下的文件，并获得这些文件的真实地址。

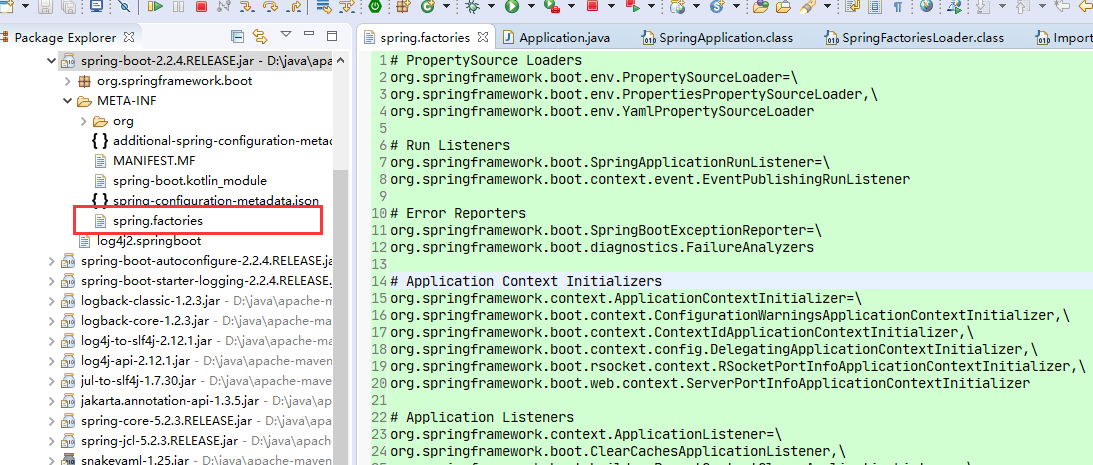
一般有2个：

jar:file:/D:/java/apache-maven-3.5.3-yooli/repository/org/springframework/boot/spring-boot-autoconfigure/2.2.4.RELEASE/spring-boot-autoconfigure-2.2.4.RELEASE.jar!/META-INF/spring.factories

jar:file:/D:/java/apache-maven-3.5.3-yooli/repository/org/springframework/boot/spring-boot-autoconfigure/2.2.4.RELEASE/spring-boot-starter-2.2.4.RELEASE.jar!/META-INF/spring.factories



我们看下这个文件是什么。



需要加载的大部分是类，少部分是依赖。Spring 启动时使用的classLoader会将其中的类进行加载

### 初始化监听器

看源码，这个接口继承了 JDK 的 java.util.EventListener 接口，实现了观察者模式，它一般用来定义感兴趣的事件类型，事件类型限定于 ApplicationEvent 的子类，这同样继承了 JDK 的 java.util.EventObject 接口。

初始化监听器和初始化上下文的方法是一样的，只是传入的类型不一样，设置监听器的接口类型为： getSpringFactoriesInstances： 实例化的监听器如下

org.springframework.boot.cloud.CloudFoundryVcapEnvironmentPostProcessor

org.springframework.boot.context.config.ConfigFileApplicationListener

org.springframework.boot.context.config.AnsiOutputApplicationListener

org.springframework.boot.context.logging.LoggingApplicationListener

org.springframework.boot.context.logging.ClasspathLoggingApplicationListener

org.springframework.boot.autoconfigure.BackgroundPreinitializer

org.springframework.boot.context.config.DelegatingApplicationListener

org.springframework.boot.builder.ParentContextCloserApplicationListener

org.springframework.boot.ClearCachesApplicationListener

org.springframework.boot.context.FileEncodingApplicationListener

org.springframework.boot.liquibase.LiquibaseServiceLocatorApplicationListener

## Run方法

public ConfigurableApplicationContext run(String... args) {

// 1、创建并启动计时监控类

StopWatch stopWatch = new StopWatch();

stopWatch.start();

// 2、初始化应用上下文和异常报告集合

ConfigurableApplicationContext context = null;

Collection<SpringBootExceptionReporter> exceptionReporters = new ArrayList<>();

// 3、设置系统属性 `java.awt.headless` 的值，默认值为：true

configureHeadlessProperty();

// 4、创建所有 Spring 运行监听器并发布应用启动事件

SpringApplicationRunListeners listeners = getRunListeners(args);

listeners.starting();

try {

// 5、初始化默认应用参数类

ApplicationArguments applicationArguments = new DefaultApplicationArguments(args);

// 6、根据运行监听器和应用参数来准备 Spring 环境

ConfigurableEnvironment environment = prepareEnvironment(listeners, applicationArguments);

configureIgnoreBeanInfo(environment);

// 7、创建 Banner 打印类

Banner printedBanner = printBanner(environment);

// 8、创建应用上下文

context = createApplicationContext();

// 9、准备异常报告器

exceptionReporters = getSpringFactoriesInstances(

SpringBootExceptionReporter.class,

new Class[] { ConfigurableApplicationContext.class }, context);

// 10、准备应用上下文

prepareContext(context, environment, listeners, applicationArguments,

printedBanner);

// 11、刷新应用上下文

refreshContext(context);

// 12、应用上下文刷新后置处理

afterRefresh(context, applicationArguments);

// 13、停止计时监控类

stopWatch.stop();

// 14、输出日志记录执行主类名、时间信息

if (this.logStartupInfo) {

new StartupInfoLogger(this.mainApplicationClass)

.logStarted(getApplicationLog(), stopWatch);

}

// 15、发布应用上下文启动完成事件

listeners.started(context);

// 16、执行所有 Runner 运行器

callRunners(context, applicationArguments);

}

catch (Throwable ex) {

handleRunFailure(context, ex, exceptionReporters, listeners);

throw new IllegalStateException(ex);

}

try {

// 17、发布应用上下文就绪事件

listeners.running(context);

} catch (Throwable ex) {

handleRunFailure(context, ex, exceptionReporters, null);

throw new IllegalStateException(ex);

}

// 18、返回应用上下文

return context;

}

# 自动启动原理

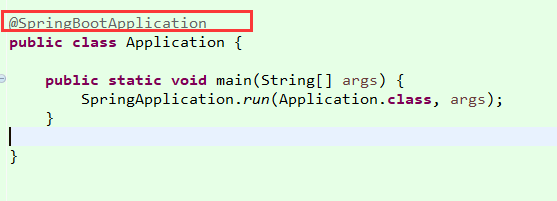
面试简述启动过程： <https://mp.weixin.qq.com/s/R4xyvmxsmxL44EnSJw6veA>

概述： <https://www.cnblogs.com/canmeng-cn/p/8627660.html>

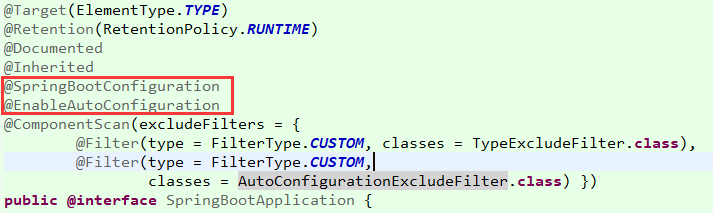
代码分析：<https://blog.csdn.net/u010811939/article/details/80592461>

## 项目启动入口run()（spring boot 2.1.4）

我们都知道springboot的项目启动类需要注解@SpringBootApplication修饰。

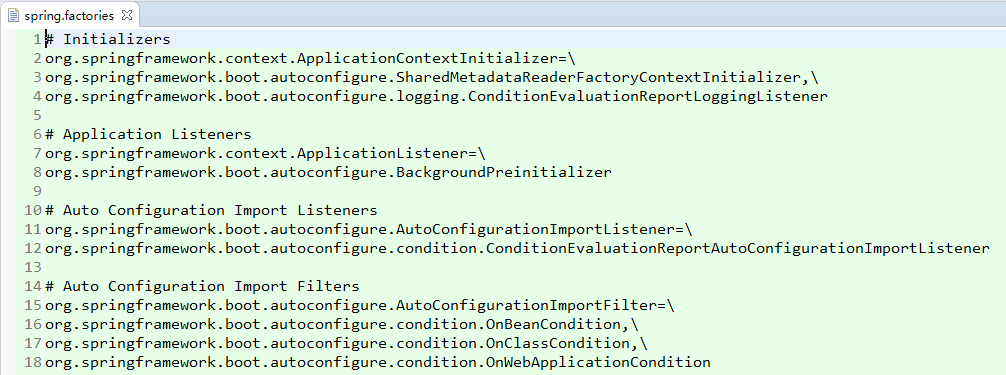


而SpringBootApplication注解，也同时使用了@EnableAutoConfiguration，用来自动启动

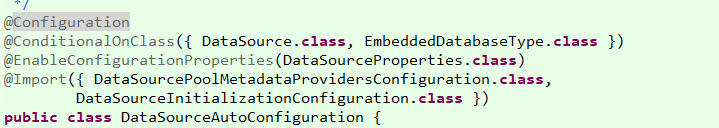


## Spring boot的案例

下面是spring-boot-autoconfigure-2.1.4.RELEASE.jar中的部分配置



数据源自动配置的实现注解



@Configuration,@ConditionalOnClass就是自动配置的核心，首先它得是一个配置文件，其次根据类路径下是否有这个类去自动配置。

## 常用注解

@ConditionalOnBean:当容器中有指定的Bean的条件下

@ConditionalOnClass：当类路径下有指定的类的条件下

@ConditionalOnExpression:基于SpEL表达式作为判断条件

@ConditionalOnJava:基于JVM版本作为判断条件

@ConditionalOnJndi:在JNDI存在的条件下查找指定的位置

@ConditionalOnMissingBean:当容器中没有指定Bean的情况下

@ConditionalOnMissingClass:当类路径下没有指定的类的条件下

@ConditionalOnNotWebApplication:当前项目不是Web项目的条件下

@ConditionalOnProperty:指定的属性是否有指定的值

@ConditionalOnResource:类路径下是否有指定的资源

@ConditionalOnWebApplication:当前项目是Web项目的条件下

@ConditionalOnSingleCandidate: 当指定的Bean在容器中只有一个，或者在有多个Bean的情下，用来指定首选的Bean

## 【美团】spring boot 怎么加载start的 spring.factories 文件的

Spring boot 获得spring.factories场景比如：初始化spring应用上下文、spring监听器、加载spring 自动配置等等。

要获得spring.factories的话，因为项目启动的时候，已经把classpath通过APPClassLoader加载到jvm中了，所有我们想要获得项目中的文件，比如jar和classes。这里只读取AppClassLoader和ExtClassLoader两个classloader下管理的路径去找。

<https://www.jianshu.com/p/b8e331840961>

## springboot中读取配置文件的几种方式和加载优选顺序?

<https://blog.csdn.net/it_faquir/article/details/80869578>

# 自定义Spring Boot Starter

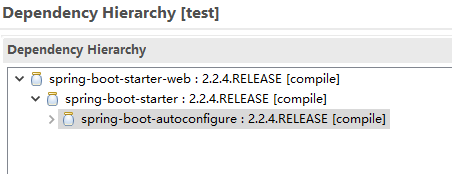
<https://blog.csdn.net/zhongzunfa/article/details/80365132>

## 【美团】spring boot 什么是约定大于配置？

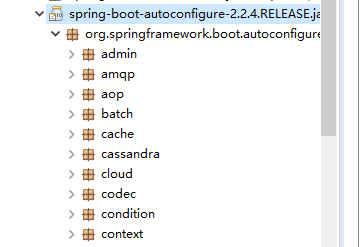
约定大于配置，我们也可以理解为约定优化配置，或者习惯优于配置，甚至再简单理解的话就是大家约定俗成的事情就不需要配置了

### 案例1：框架的简答案例：Redis

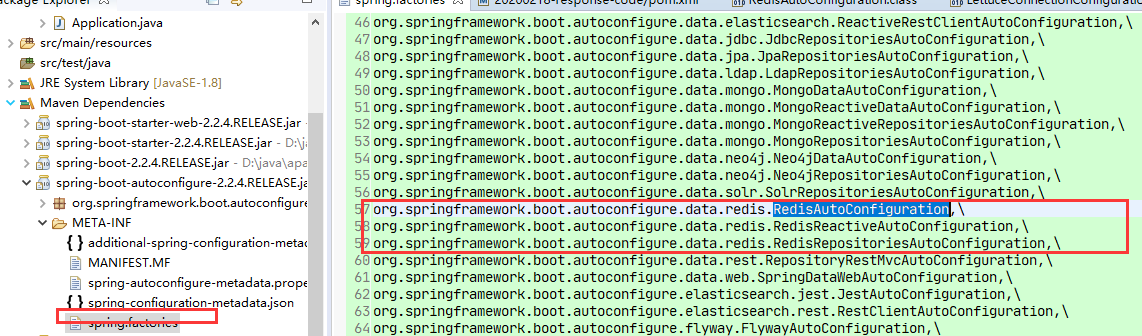
spring boot 对很多框架中间件等提供了自动配置的类，位置如下：其实就是我们自动引用了这个jar，



自动配置的框架有很多，但是不是我们启动的时候就全都进行自动配置



jar里面的META-INF/spring.factories，定义很多AutoConfiguration类。Spring boot启动的时候先加载这些配置。



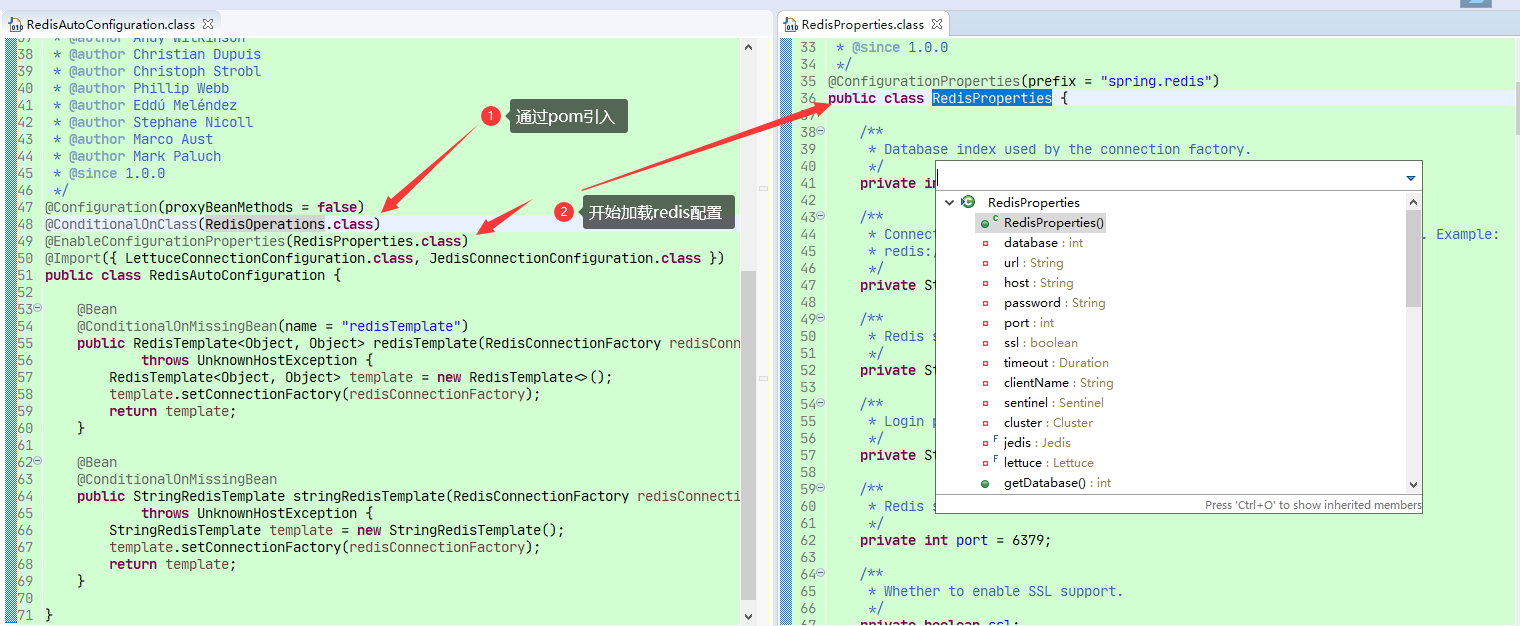
以redis为例子：我们使用redis的时候，先要引入依赖，RedisAutoConfiguration自动配置的时候，依赖于RedisOperations类的存在，所有我们不引入pom依赖的时候是不会自动配置redis的时候，而引入后，才开始自动配置，并且RedisProperties类中，给了大量的默认值，甚至不需要任何配置我们就能连接到默认的Redis本地单机配置了。甚至提供的线程池的配置。保证基础的开箱即用

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-redis</artifactId>

</dependency>



以上就是框架的约定大于配置的一个简答的案例

总结：就是我们一引用某个spring boot 直接支持的框架，他就会帮助我们按照默认的配置去启动项目，比如 添加 start-web 项目就是web项目， 添加start-data-redis，就启动redisTemplate等等

### 案例2：我们自己的配置

1.默认加载resources目录下的配置文件application.properties 或.yml文件

2.多环境配置，比普通的maven项目方便不少

3.项目的类型可以通过引入的包进行判断，引入starter-web，就是web项目，还有非web项目与reactive项目

## 需要启动的类

import org.springframework.context.EnvironmentAware;

import org.springframework.core.env.Environment;

public class EnvConfig implements EnvironmentAware {

private Environment env;

public String getStringValue(String key) {

return this.env.getProperty(key);

}

public Integer getIntValue(String key) {

return Integer.parseInt(this.env.getProperty(key));

}

@Override

public void setEnvironment(Environment environment) {

this.env = environment;

}

}

## 自动启动的配置

@Configuration

@ConditionalOnClass(PropertyResolver.class)

public class EnvAutoConfig {

@Bean

public EnvConfig envConfig() {

return new EnvConfig();

}

}

## 配置启动

方式1：自动启动

在resources目录创建META-INF/spring.factories文件

org.springframework.boot.autoconfigure.EnableAutoConfiguration=\

com.zx.EnvAutoConfig

方式2：手动启动

@Retention(RetentionPolicy.RUNTIME)

@Target({ElementType.TYPE})

@Documented

@Inherited

// 重点在这注解导入启动配置

@Import({EnvAutoConfig .class})

public @interface EnableEnvAutoConfig {

// 使用只需要在 java 类中使用 @EnableEnvAutoConfig 启动

}

## 查看自己的类是否被加载

第一步：设置日志级别为debug

第二步：启动项目，查看控制台

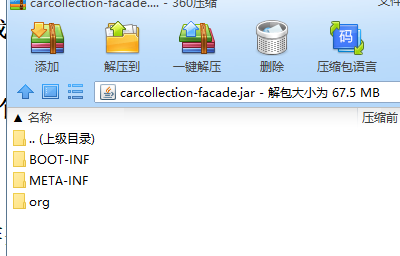


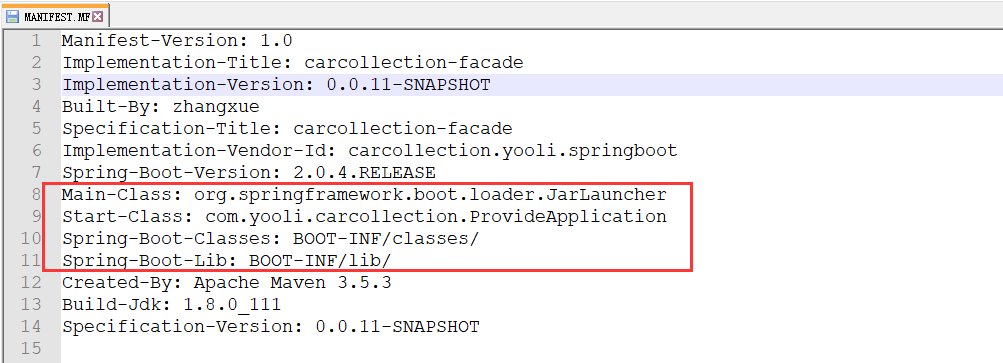
Positive matches：已经启用的自动配置

Negative matches：未启用的自动配置

# Spring boot 的jar为什么可以直接运行

<https://mp.weixin.qq.com/s/sMiOftOMXV5dooFO4R1vxg>





根据项目的类型，jar，war。启动不同的launcher类。以JarLauncher为例子，JarLauncher主要是为了  
 1.为了使用自定义的classloader。LaunchedURLClassLoader加载jar与class（主要是为了按照自己的规则加载）

2.新启动一个线程去启动我们定义的main方法

# @Import的作用

<https://blog.csdn.net/panchao888888/article/details/82882279>

# SpringBoot内置的Tomcat是如何启动的

<https://mp.weixin.qq.com/s/qeHM2A60dQ2ztyoMTiVXBw>