SpringBoot启动原理

SpringBoot启动原理
- Spring Boot、Spring MVC 和 Spring 有什么区别？
- springboot启动原理及相关流程概览
- springboot的启动类入口
- SpringBootApplication接口用到的注解
  - @Configuration注解
  - @ComponentScan注解
  - **@EnableAutoConfiguration**
    - AutoConfigurationPackage注解：
    - Import(AutoConfigurationImportSelector.class)注解
- 自动配置幕后英雄：SpringFactoriesLoader详解
- 深入探索SpringApplication执行流程
- Bean的生命周期
- BeanFactory 和ApplicationContext的区别

Spring Boot、Spring MVC 和 Spring 有什么区别?

分别描述各自的特征：

Spring 框架就像一个家族，有众多衍生产品例如 boot、security、jpa等等；但他们的基础都是Spring 的ioc和 aop，ioc 提供了依赖注入的容器， aop解决了面向切面编程，然后在此两者的基础上实现了其他延伸产品的高级功能。

Spring MVC提供了一种轻度耦合的方式来开发web应用；它是Spring的一个模块，是一个web框架；通过DispatcherServlet, ModelAndView 和 View Resolver，开发web应用变得很容易；解决的问题领域是网站应用程序或者服务开发——URL路由、Session、模板引擎、静态Web资源等等。

Spring Boot实现了auto-configuration **自动配置**（另外三大神器actuator监控，cli命令行接口，starter依赖），降低了项目搭建的复杂度。它主要是为了解决使用Spring框架需要进行大量的配置太麻烦的问题，所以它并不是用来替代Spring的解决方案，而是和Spring框架紧密结合用于提升Spring开发者体验的工具；同时它集成了大量常用的第三方库配置(例如Jackson, JDBC, Mongo, Redis, Mail等等)，Spring Boot应用中这些第三方库几乎可以零配置的开箱即用(out-of-the-box)。

所以，用最简练的语言概括就是:

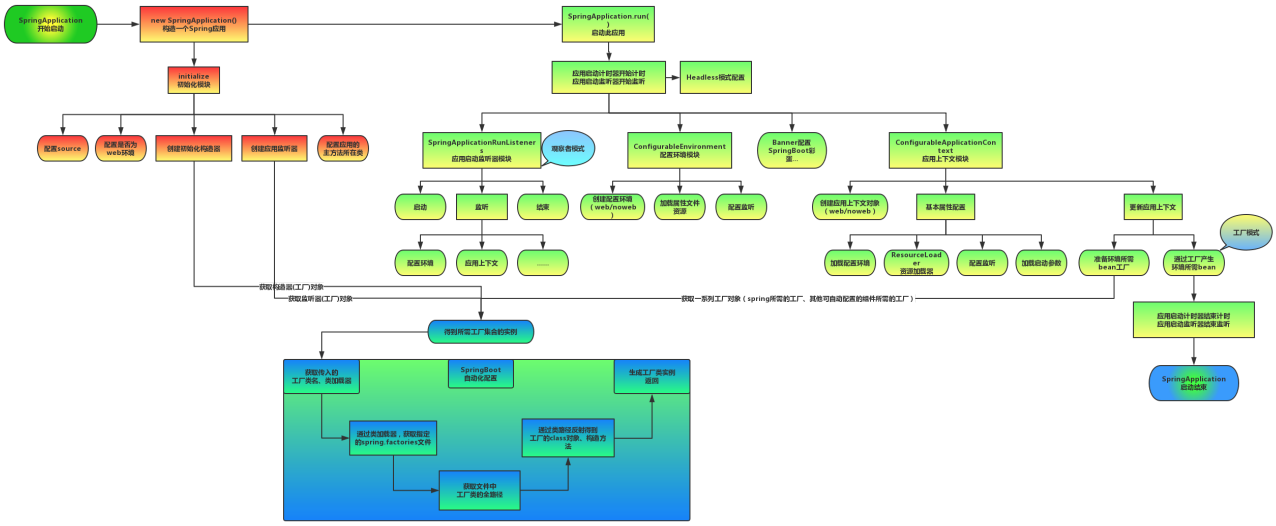
Spring 是一个“引擎”；

Spring MVC 是基于Spring的一个 MVC 框架;

Spring Boot 是基于Spring4的条件注册的一套快速开发整合包。

springboot启动原理及相关流程概览

springboot是基于spring的新型的轻量级框架，最厉害的地方当属 ***自动配置。*** 那我们就可以根据启动流程和相关原理来看看，如何实现传奇的自动配置。



springboot的启动类入口

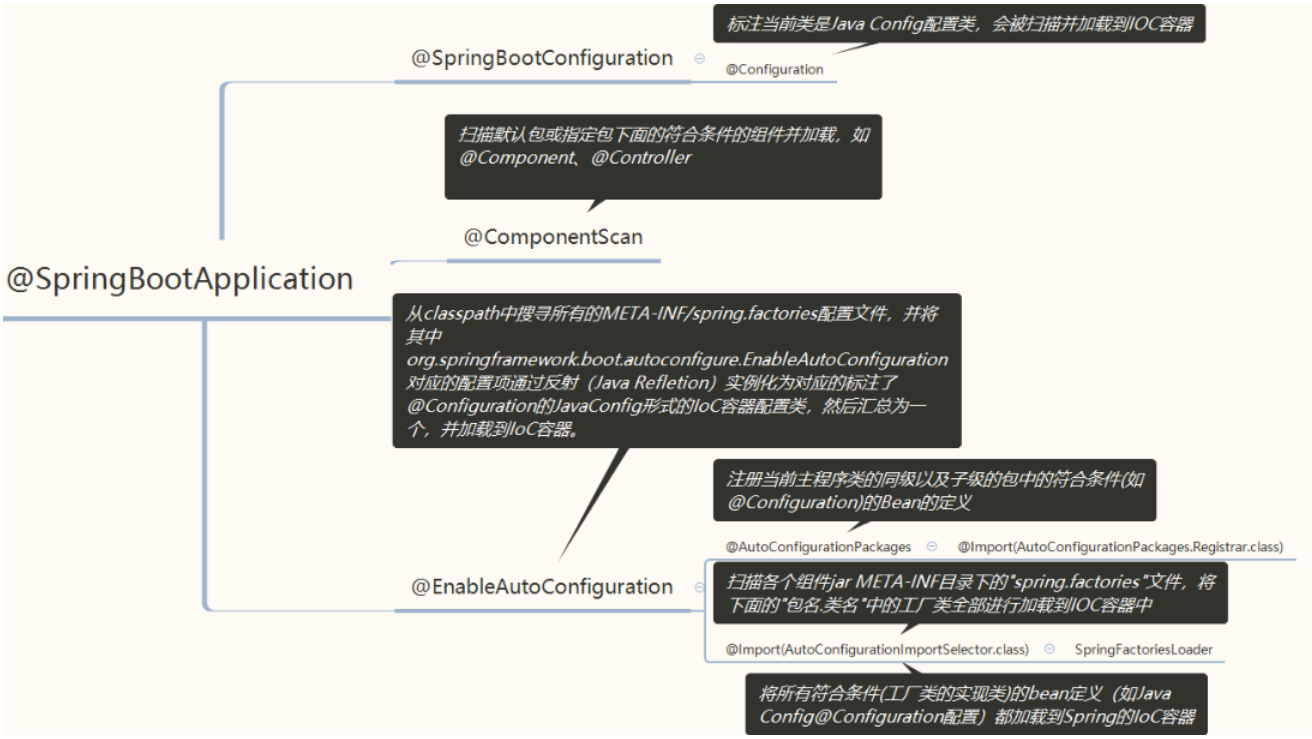
用过springboot的技术人员很显而易见的两者之间的差别就是视觉上很直观的：springboot有自己独立的启动类（独立程序）

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

从上面代码可以看出，Annotation定义（@SpringBootApplication）和类定义（SpringApplication.run）最为耀眼，所以要揭开SpringBoot的神秘面纱，我们要从这两位开始就可以了。

SpringBootApplication接口用到的注解

```
@Target(ElementType.TYPE) // 注解的适用范围，其中TYPE用于描述类、接口（包括包注解类型）或enum声明
@Retention(RetentionPolicy.RUNTIME) // 注解的生命周期，保留到class文件中（三个生命周期）
@Documented // 表明这个注解应该被javadoc记录
@Inherited // 子类可以继承该注解
@SpringBootConfiguration // 继承了Configuration，表示当前是注解类
@EnableAutoConfiguration // 开启springboot的注解功能，springboot的四大神器之一，其借助@import的帮助
@ComponentScan(excludeFilters = { // 扫描路径设置（具体使用待确认）
@Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
@Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {

    ...

}
```



在其中比较重要的有三个注解，分别是：

- @SpringBootConfiguration // 继承了Configuration，表示当前是注解类
- @EnableAutoConfiguration // 开启springboot的注解功能，springboot的四大神器之一，其借助@import的帮助
- @ComponentScan(excludeFilters = { // 扫描路径设置（具体使用待确认）

接下来对三个注解一一详解，增加对springbootApplication的理解：

@Configuration注解

按照原来xml配置文件的形式，在springboot中我们大多用配置类来解决配置问题

配置bean方式的不同：

- xml配置文件的形式配置bean

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"
default-lazy-init="true">
<!--bean定义-->
</beans>
```

- java configuration的配置形式配置bean

```
@Configuration
public class MockConfiguration{
   //bean定义
}
```

注入bean方式的不同：

- xml配置文件的形式注入bean

```
<bean id="mockService" class="..MockServiceImpl">
    ...
</bean>
```

- java configuration的配置形式注入bean

```
@Configuration
public class MockConfiguration{
  @Bean
  public MockService mockService(){
    return new MockServiceImpl();
  }
}
```

任何一个标注了@Bean的方法，其返回值将作为一个bean定义注册到Spring的IoC容器，方法名将默认成该bean定义的id。

表达**bean之间依赖关系**的不同：表达**bean之间依赖关系**的不同：

- xml配置文件的形式表达依赖关系

```
<bean id="mockService" class="..MockServiceImpl">
    <propery name ="dependencyService" ref="dependencyService" />
</bean>
<bean id="dependencyService" class="DependencyServiceImpl"></bean>
```

- java configuration配置的形式表达**依赖关系（重点）**

如果一个bean A的定义依赖其他bean B,则直接调用对应的JavaConfig类中依赖bean B的创建方法就可以了。

```
@Configuration
public class MockConfiguration{
    @Bean
    public MockService mockService(){
      return new MockServiceImpl(dependencyService());
    }
    @Bean
    public DependencyService dependencyService(){
      return new DependencyServiceImpl();
    }
}
```

@ComponentScan注解

作用：

a）对应xml配置中的元素；

b）**（重点）**ComponentScan的功能其实就是自动扫描并加载符合条件的组件（比如@Component和@Repository等）或者bean定义;

c）将这些bean定义加载到**IoC**容器中.

我们可以通过basePackages等属性来**细粒度**的定制@ComponentScan自动扫描的范围，如果不指定，则**默认**Spring框架实现会从声明@ComponentScan所在类的package进行扫描。

`注：所以SpringBoot的启动类最好是放在root package 下，因为默认不指定basePackages`

@EnableAutoConfiguration

此注解顾名思义是可以自动配置，所以应该是springboot中最为重要的注解。

在spring框架中就提供了各种以@Enable开头的注解，例如： @EnableScheduling、@EnableCaching、@EnableMBeanExport等； @EnableAutoConfiguration的理念和做事方式其实一脉相承简单概括一下就是，借助@Import的支持，收集和注册特定场景相关的bean定义。

- @EnableScheduling是通过@Import将Spring调度框架相关的bean定义都加载到IoC容器【定时任务、时间调度任务】
- @EnableMBeanExport是通过@Import将JMX相关的bean定义加载到IoC容器【监控JVM运行时状态】

@EnableAutoConfiguration也是借助@Import的帮助，将所有符合自动配置条件的bean定义加载到IoC容器。

@EnableAutoConfiguration作为一个复合Annotation,其自身定义关键信息如下：

```
@SuppressWarnings("deprecation")
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage【重点注解】
@Import(AutoConfigurationImportSelector.class)【重点注解】
public @interface EnableAutoConfiguration {
...
}
```

其中最重要的两个注解已经标注：

1、`@AutoConfigurationPackage【重点注解】`

2、`@Import(AutoConfigurationImportSelector.class)【重点注解】`

当然还有其中比较重要的一个类就是：AutoConfigurationImportSelector.class

AutoConfigurationPackage注解：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Import(AutoConfigurationPackages.Registrar.class)
public @interface AutoConfigurationPackage {

}
```

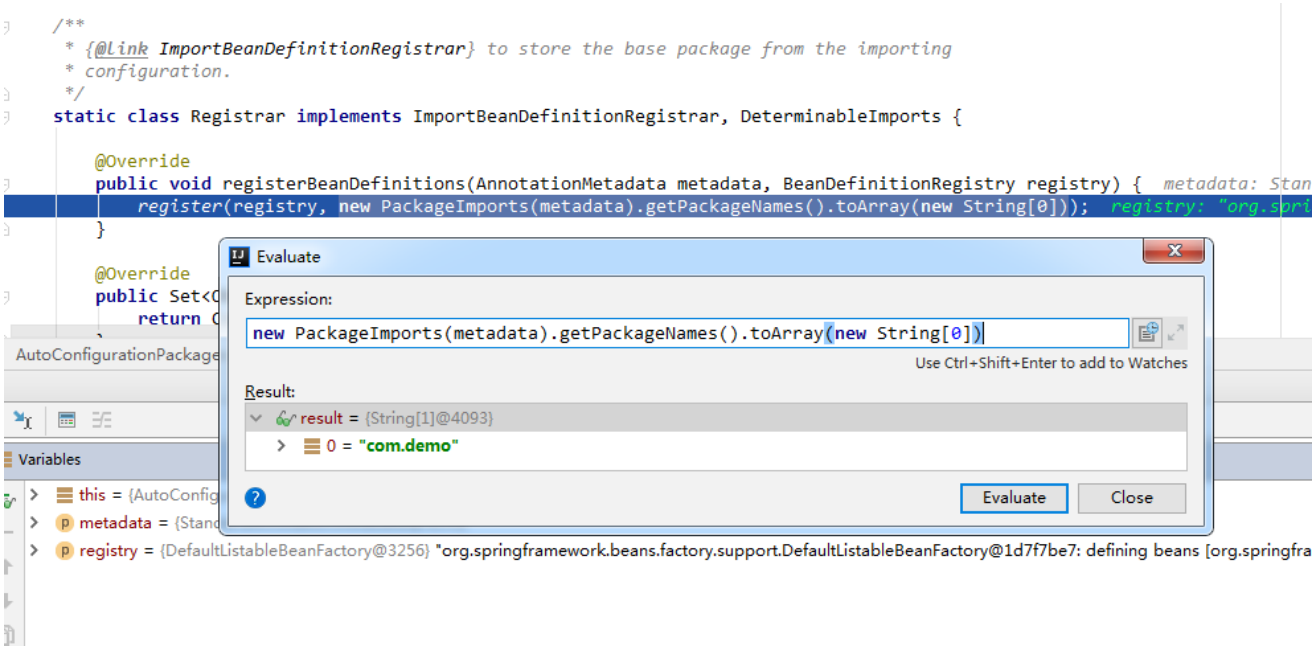
通过`@Import(AutoConfigurationPackages.Registrar.class)`

```
static class Registrar implements ImportBeanDefinitionRegistrar, DeterminableImports {

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

        ……

    }
```

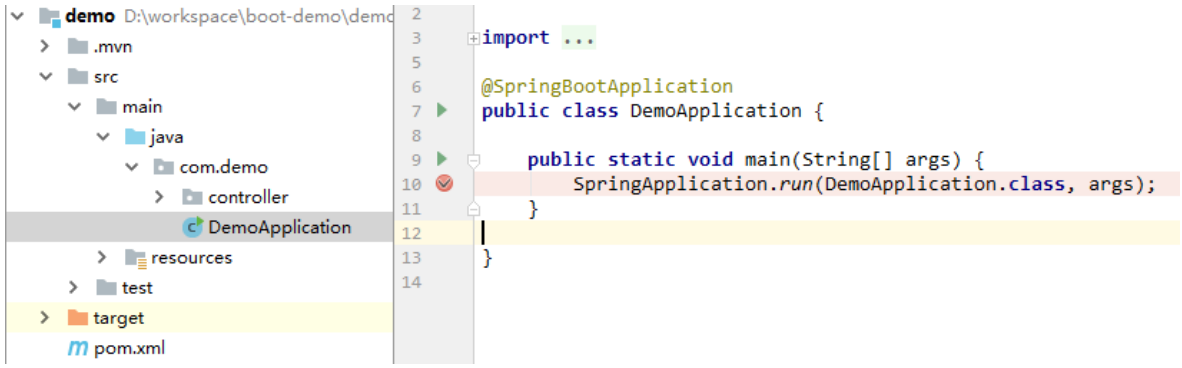
它其实是注册了一个Bean的定义；

`new PackageImport(metadata).getPackageName()`，它其实返回了**当前主程序类的**同级以及子级的包组件（重点）；

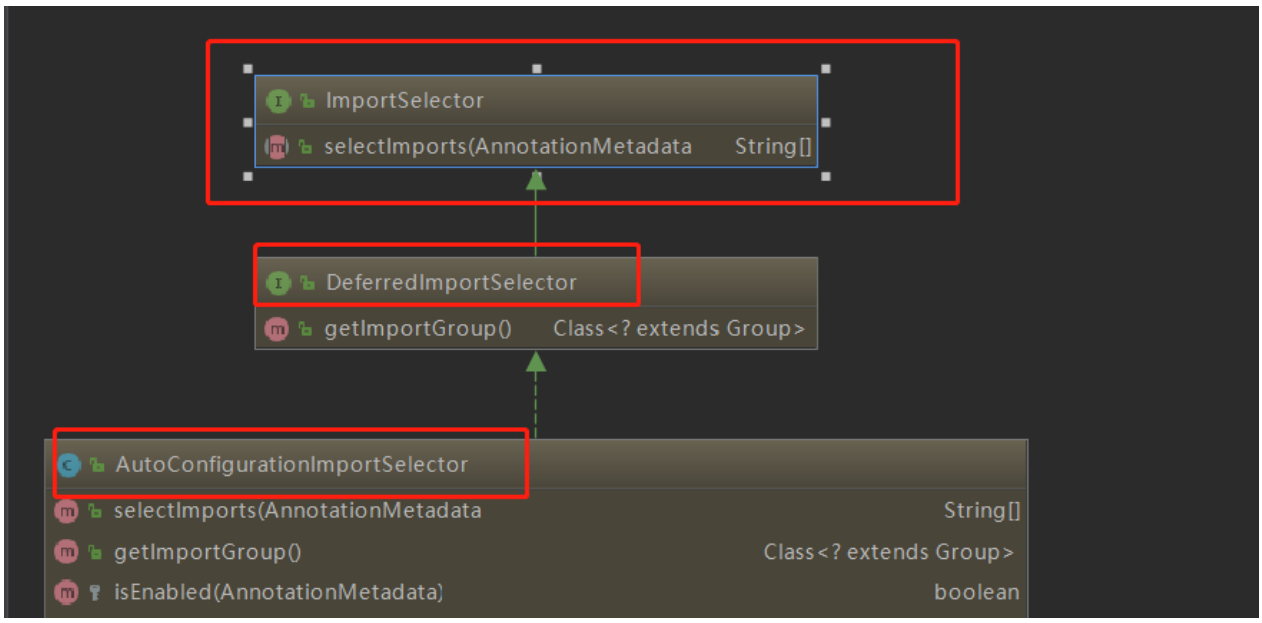
（重点）那这总体就是注册当前主程序类的同级以及子级的包中的符合条件的**Bean的定义**？



项目目录结构：



Import(AutoConfigurationImportSelector.class)注解



（重点）可以从图中看出 `AutoConfigurationImportSelector` 实现了 `DeferredImportSelector` 从 ImportSelector继承的方法： `selectImports` 。

```
@Override
public String[] selectImports(AnnotationMetadata annotationMetadata) {
    if (!isEnabled(annotationMetadata)) {
        return NO_IMPORTS;
    }
```

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

    /**
     * Return the {@link AutoConfigurationEntry} based on the {@link AnnotationMetadata}
     * of the importing {@link Configuration @Configuration} class.
     * @param annotationMetadata the annotation metadata of the configuration class
     * @return the auto-configurations that should be imported
     */
    protected AutoConfigurationEntry getAutoConfigurationEntry(AnnotationMetadata annotationMetadata) {
        if (!isEnabled(annotationMetadata)) {
            return EMPTY_ENTRY;
        }
        AnnotationAttributes attributes = getAttributes(annotationMetadata);
        // 加载各个组件jar下的   public static final String FACTORIES_RESOURCE_LOCATION = "META-
INF/spring.factories"
        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);
    }
```

该方法在springboot启动流程——`bean实例化前`被执行，返回要实例化的类信息列表；

如果获取到类信息，spring可以通过类加载器将类加载到jvm中，现在我们已经通过spring-boot的starter依赖方式依赖了我们需要的组件，那么这些组件的类信息在select方法中就可以被获取到。

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

    public static List<String> loadFactoryNames(Class<?> factoryType, @Nullable ClassLoader classLoader) {
        ClassLoader classLoaderToUse = classLoader;
        if (classLoaderToUse == null) {
            classLoaderToUse = SpringFactoriesLoader.class.getClassLoader();
        }
        String factoryTypeName = factoryType.getName();
        return loadSpringFactories(classLoaderToUse).getOrDefault(factoryTypeName, Collections.emptyList());
    }

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

        result = new HashMap<>();
        try {
            // FACTORIES_RESOURCE_LOCATION = "META-INF/spring.factories"
```

```
            // 加载配置文件
            Enumeration<URL> urls = classLoader.getResources(FACTORIES_RESOURCE_LOCATION);
            while (urls.hasMoreElements()) {
                URL url = urls.nextElement();
                UrlResource resource = new UrlResource(url);
                Properties properties = PropertiesLoaderUtils.loadProperties(resource);
                for (Map.Entry<?, ?> entry : properties.entrySet()) {
                    // spring.factories文件中找到相应的key
                    String factoryTypeName = ((String) entry.getKey()).trim();
                    String[] factoryImplementationNames =
                            StringUtils.commaDelimitedListToStringArray((String) entry.getValue());
                    for (String factoryImplementationName : factoryImplementationNames) {
                        result.computeIfAbsent(factoryTypeName, key -> new ArrayList<>())
                                .add(factoryImplementationName.trim());
                    }
                }
            }

            // Replace all lists with unmodifiable lists containing unique elements
            result.replaceAll((factoryType, implementations) -> implementations.stream().distinct()
                    .collect(Collectors.collectingAndThen(Collectors.toList(), Collections::unmodifiableList)));
            cache.put(classLoader, result);
        }
        catch (IOException ex) {
            throw new IllegalArgumentException("Unable to load factories from location [" +
                    FACTORIES_RESOURCE_LOCATION + "]", ex);
        }
        return result;
    }
```

（重点）其中，最关键的要属`@Import(AutoConfigurationImportSelector.class)`，借助`AutoConfigurationImportSelector`，`@EnableAutoConfiguration`可以帮助SpringBoot应用将`所有符合条件(spring.factories)的bean定义`（如Java Config`@Configuration配置`）都加载到当前SpringBoot创建并使用的`IoC容器`。就像一只“八爪鱼”一样。

自动配置幕后英雄：SpringFactoriesLoader详解

借助于Spring框架`原有`的一个工具类：`SpringFactoriesLoader`的支持，@EnableAutoConfiguration可以智能的自动配置功效才得以大功告成！

SpringFactoriesLoader属于Spring框架私有的一种`扩展`方案，其主要功能就是从`指定的配置文件META-INF/spring.factories`加载配置，`加载工厂类`。

SpringFactoriesLoader为`Spring工厂加载器`，该对象提供了`loadFactoryNames`方法，入参为factoryClass和classLoader即需要传入`工厂类`名称和对应的类加载器，方法会根据指定的classLoader，加载该类加器搜索路径下的指定文件，即`spring.factories`文件；

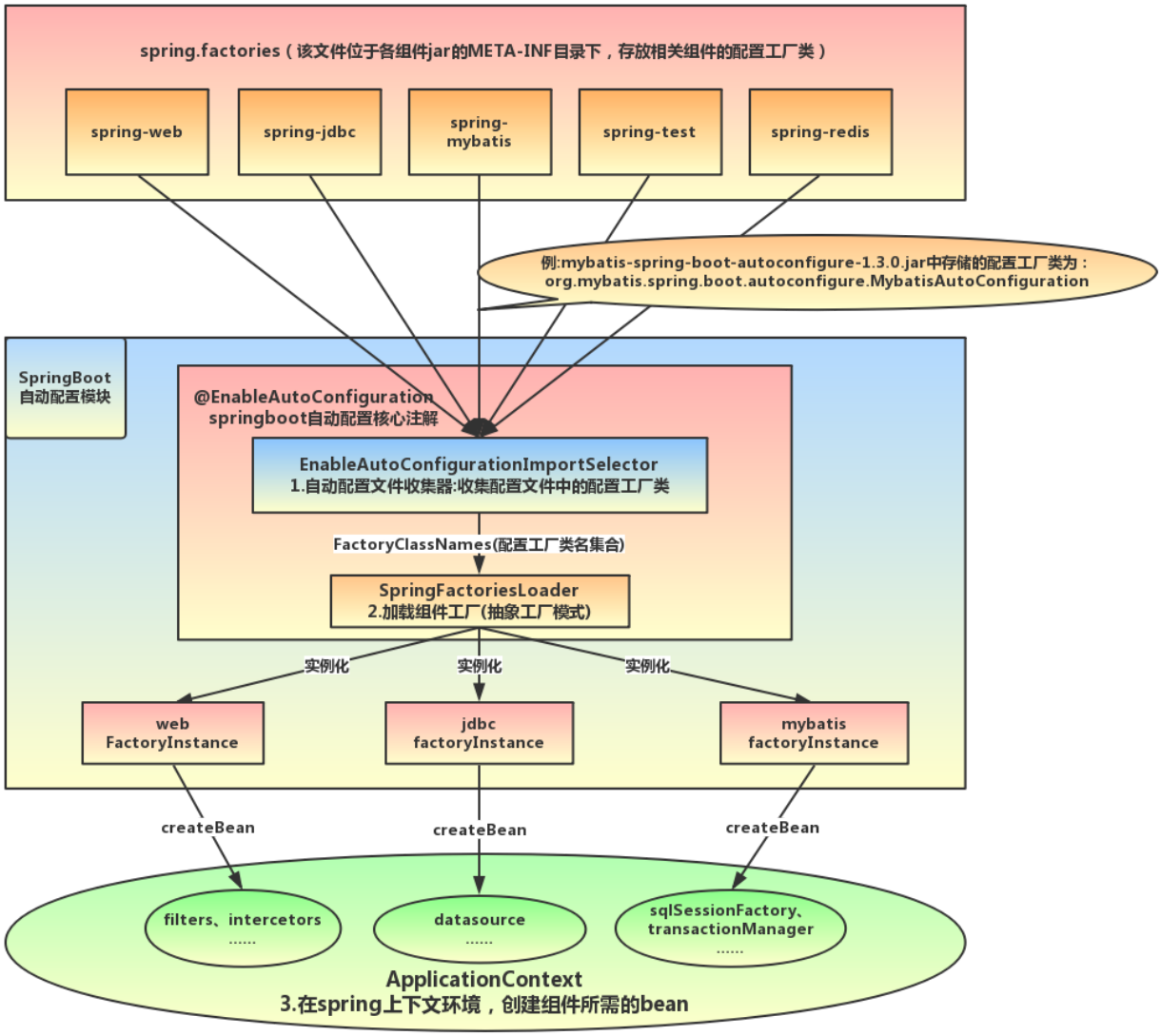
传入的工厂类为接口，而文件中对应的类则是接口的实现类，或最终作为实现类。

```
public abstract class SpringFactoriesLoader {
//...
    public static <T> List<T> loadFactories(Class<T> factoryClass, ClassLoader classLoader) {
        ...
    }


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

配合`@EnableAutoConfiguration`使用的话，它更多是提供一种配置查找的功能支持，即根据`@EnableAutoConfiguration`的完整类名org.springframework.boot.autoconfigure.EnableAutoConfiguration作为`查找的Key`,获取对应的一组`@Configuration`类

`（重点）`所以，`@EnableAutoConfiguration`自动配置的魔法其实就变成了：

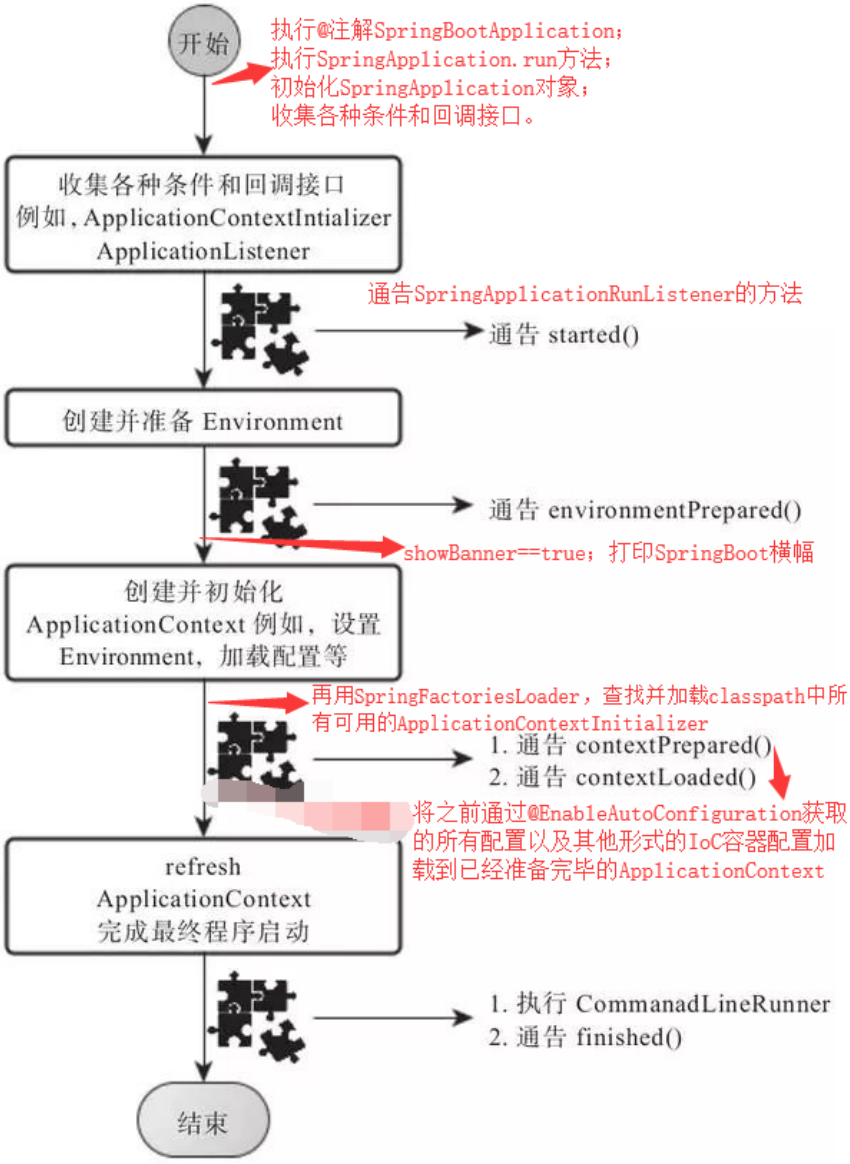
从`classpath`中搜寻所有的`META-INF/spring.factories`配置文件，并将其中org.springframework.boot.autoconfigure.EnableAutoConfiguration对应的`配置项`通过`反射（Java Refletion）`实例化为对应的标注了`@Configuration`的JavaConfig形式的IoC容器配置类，然后汇总为一个并加载到IoC容器。



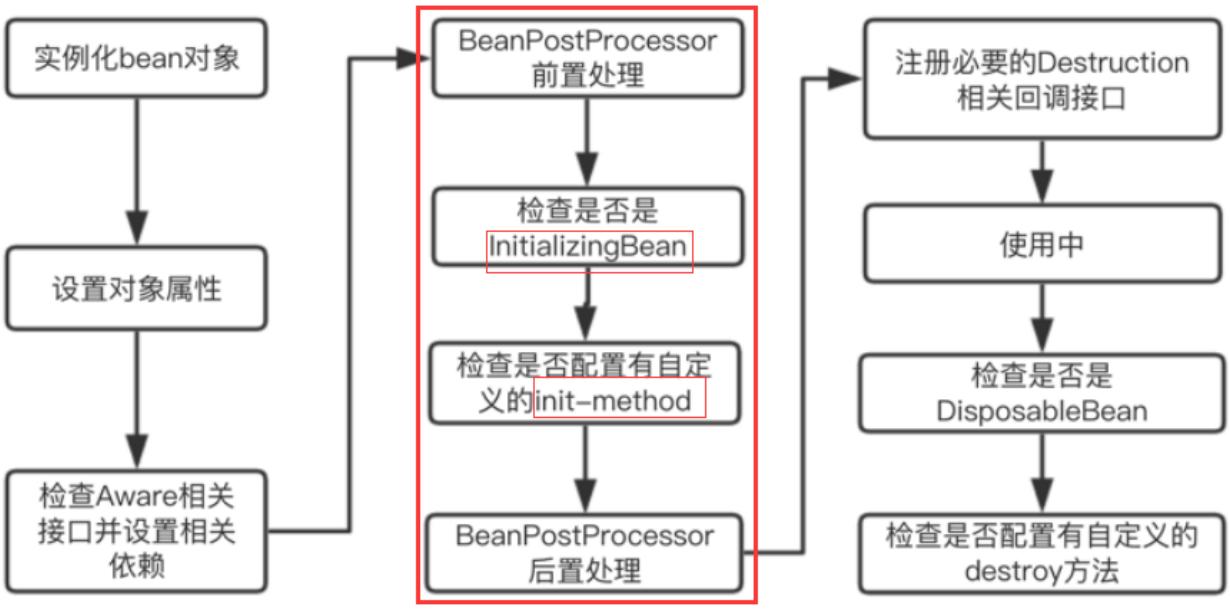
SpringBoot自动化配置关键组件关系图

mybatis-spring-boot-starter、spring-boot-starter-web等组件的META-INF文件下均含有spring.factories文件，自动配置模块中，SpringFactoriesLoader收集到文件中的类全名并返回一个类全名的数组，返回的类全名通过反射被实例化，就形成了具体的工厂实例，工厂实例来生成组件具体需要的bean。

深入探索SpringApplication执行流程


开始
执行@注解SpringBootApplication;
执行SpringApplication.run方法;
初始化SpringApplication对象;
收集各种条件和回调接口。
收集各种条件和回调接口
例如，ApplicationContextIntializer
ApplicationListener
通告SpringApplicationRunListener的方法
通告 started()
创建并准备 Environment
通告 environmentPrepared()
showBanner==true；打印SpringBoot横幅
创建并初始化
ApplicationContext 例如，设置
Environment，加载配置等
再用SpringFactoriesLoader，查找并加载classpath中所有可用的ApplicationContextInitializer
1. 通告 contextPrepared()
2. 通告 contextLoaded()
将之前通过@EnableAutoConfiguration获取的所有配置以及其他形式的IoC容器配置加载到已经准备完毕的ApplicationContext
refresh
ApplicationContext
完成最终程序启动
1. 执行 CommanadLineRunner
2. 通告 finished()
结束

Bean的生命周期


实例化bean对象
设置对象属性
检查Aware相关接口并设置相关依赖
BeanPostProcessor
前置处理
检查是否是
InitializingBean
检查是否配置有自定义的init-method
BeanPostProcessor
后置处理
注册必要的Destruction
相关回调接口
使用中
检查是否是
DisposableBean
检查是否配置有自定义的
destroy方法

BeanFactory 和ApplicationContext的区别