给你一份超详细Spring Boot知识清单

2018-10-09 18:05

“

在过去两三年的 Spring 生态圈，最让人兴奋的莫过于 Spring Boot 框架。或许从命名上就能看出这个框架的设计初衷：快速的启动 Spring 应用。

因而 Spring Boot 应用本质上就是一个基于 Spring 框架的应用，它是 Spring 对“约定优先于配置”理念的最佳实践产物，它能够帮助开发者更快速高基于 Spring 生态圈的应用。

那 Spring Boot 有何魔法？自动配置、起步依赖、Actuator、命令行界面(CLI) 是 Spring Boot 最重要的 4 大核心特性。

其中 CLI 是 Spring Boot 的可选特性，虽然它功能强大，但也引入了一套不太常规的开发模型，因而这个系列的文章仅关注其他 3 种特性。

本文将为你打开 Spring Boot 的大门，重点为你剖析其启动流程以及自动配置实现原理。要掌握这部分核心内容，理解一些 Spring 框架的基础知识，事半功倍。

抛砖引玉：探索 Spring IOC 容器

如果有看过 SpringApplication.run() 方法的源码，Spring Boot 冗长无比的启动流程一定会让你抓狂。

透过现象看本质，SpringApplication 只是将一个典型的Spring应用的启动流程进行了扩展，因此，透彻理解 Spring 容器是打开 Spring Boot 大门匙。

Spring IOC 容器

可以把 Spring IOC 容器比作一间餐馆，当你来到餐馆，通常会直接招呼服务员：点菜！至于菜的原料是什么？如何用原料把菜做出来？可能你根本就

IOC 容器也是一样，你只需要告诉它需要某个 bean，它就把对应的实例（instance）扔给你，至于这个 bean 是否依赖其他组件，怎样完成它的初始就不需要你关心。

作为餐馆，想要做出菜肴，得知道菜的原料和菜谱，同样地，IOC 容器想要管理各个业务对象以及它们之间的依赖关系，需要通过某种途径来记录和信息。

BeanDefinition 对象就承担了这个责任：容器中的每一个 bean 都会有一个对应的 BeanDefinition 实例。

该实例负责保存 bean 对象的所有必要信息，包括 bean 对象的 class 类型、是否是抽象类、构造方法和参数、其他属性等等。

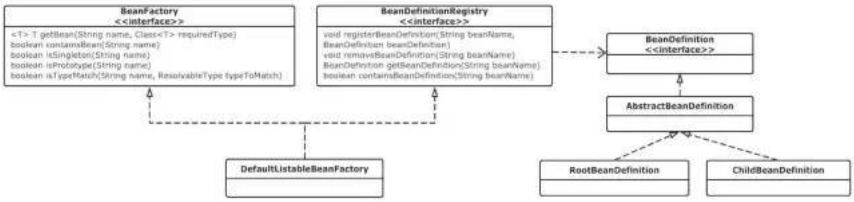
当客户端向容器请求相应对象时，容器就会通过这些信息为客户端返回一个完整可用的 bean 实例。

原材料已经准备好（把 BeanDefinition 看做原料），开始做菜吧，等等，你还需要一份菜谱。

BeanDefinitionRegistry 和 BeanFactory 就是这份菜谱，BeanDefinitionRegistry 抽象出 bean 的注册逻辑。

而 BeanFactory 则抽象出了 bean 的管理逻辑，而各个 BeanFactory 的实现类就具体承担了 bean 的注册以及管理工作。

它们之间的关系就如下图：



BeanFactory、BeanDefinitionRegistry 关系图（来自：Spring 揭秘）

DefaultListableBeanFactory 作为一个比较通用的 BeanFactory 实现，它同时也实现了 BeanDefinitionRegistry 接口，因此它就承担了 bean 的注作。

从图中也可以看出，BeanFactory 接口中主要包含 getBean、containBean、getType、getAliases 等管理 bean 的方法。

而 BeanDefinitionRegistry 接口则包含 registerBeanDefinition、removeBeanDefinition、getBeanDefinition 等注册管理 BeanDefinition 的方法。

下面通过一段简单的代码来模拟 BeanFactory 底层是如何工作的：

```
// 默认容器实现
DefaultListableBeanFactory beanRegistry = new DefaultListableBeanFactory();
// 根据业务对象构造相应的BeanDefinition
AbstractBeanDefinitio definition =ne  RootBeanDefinitio(Busines .clas ,tru )
n                                 w   n                 s       s     e   ;
// 将bean定义注册到容器中
beanRegistry.registerBeanDefinition("beanName",definition);
// 如果有多个bean，还可以指定各个bean之间的依赖关系
// ........
// 然后可以从容器中获取这个bean的实例
// 注意：这里的beanRegistry其实实现了BeanFactory接口，所以可以强转，
// 单纯的BeanDefinitionRegistry是无法强制转换到BeanFactory类型的
BeanFactory container = (BeanFactory)beanRegistry;
Business business = (Business)container.getBean("beanName");
```

这段代码仅为了说明 BeanFactory 底层的大致工作流程，实际情况会更加复杂。

比如 bean 之间的依赖关系可能定义在外部配置文件(XML/Properties)中、也可能是注解方式。

Spring IoC 容器的整个工作流程大致可以分为两个阶段：

①容器启动阶段

容器启动时，会通过某种途径加载 ConfigurationMetaData。除了代码方式比较直接外，在大部分情况下，容器需要依赖某些工具类。

比如：BeanDefinitionReader，它会对加载的 ConfigurationMetaData 进行解析和分析，并将分析后的信息组装为相应的 BeanDefinition。

最后把这些保存了 bean 定义的 BeanDefinition，注册到相应的 BeanDefinitionRegistry，这样容器的启动工作就完成了。

这个阶段主要完成一些准备性工作，更侧重于 bean 对象管理信息的收集，当然一些验证性或者辅助性的工作也在这一阶段完成。

来看一个简单的例子吧，过往，所有的 bean 都定义在 XML 配置文件中，下面的代码将模拟 BeanFactory 如何从配置文件中加载 bean 的定义以

```
// 通常为BeanDefinitionRegistry的实现类，这里以DeFaultListabeBeanFactory为例
BeanDefinitionRegistry beanRegistry = new DefaultListableBeanFactory();
// XmlBeanDefinitionReader实现了BeanDefinitionReader接口，用于解析XML文件
XmlBeanDefinitionR  beanDefinitionRead ne  XmlBeanDefinitionReade(beanRegistr
eader               er =               w   rImpl                  y);
// 加载配置文件
beanDefinitionReader.loadBeanDefinitions("classpath:spring-bean.xml");
// 从容器中获取bean实例
BeanFactory container = (BeanFactory)beanRegistry;
Business business = (Business)container.getBean("beanName");
```

②Bean 的实例化阶段

经过第一阶段，所有 bean 定义都通过 BeanDefinition 的方式注册到 BeanDefinitionRegistry 中。

当某个请求通过容器的 getBean 方法请求某个对象，或者因为依赖关系容器需要隐式的调用 getBean 时，就会触发第二阶段的活动：容器会首先检
的对象之前是否已经实例化完成。

如果没有，则会根据注册的 BeanDefinition 所提供的信息实例化被请求对象，并为其注入依赖。当该对象装配完毕后，容器会立即将其返回给请求方

BeanFactory 只是 Spring IoC 容器的一种实现，如果没有特殊指定，它采用延迟初始化策略：只有当访问容器中的某个对象时，才对该对象进行初始
注入操作。

而在实际场景下，我们更多的使用另外一种类型的容器： ApplicationContext，它构建在 BeanFactory 之上，属于更高级的容器。

除了具有 BeanFactory 的所有能力之外，还提供对事件监听机制以及国际化的支持等。它管理的 bean，在容器启动时全部完成初始化和依赖注入操作

Spring 容器扩展机制

IoC 容器负责管理容器中所有 bean 的生命周期，而在 bean 生命周期的不同阶段，Spring 提供了不同的扩展点来改变 bean 的命运。

在容器的启动阶段， BeanFactoryPostProcessor 允许我们在容器实例化相应对象之前，对注册到容器的 BeanDefinition 所保存的信息做一些额外的
如修改bean定义的某些属性或者增加其他信息等。

如果要自定义扩展类，通常需要实现 org.springframework.beans.factory.config.BeanFactoryPostProcessor 接口。

与此同时，因为容器中可能有多个 BeanFactoryPostProcessor，可能还需要实现 org.springframework.core.Ordered 接口，
BeanFactoryPostProcessor 按照顺序执行。

Spring 提供了为数不多的 BeanFactoryPostProcessor 实现，我们以 PropertyPlaceholderConfigurer 来说明其大致的工作流程。

在 Spring 项目的 XML 配置文件中，经常可以看到许多配置项的值使用占位符，而将占位符所代表的值单独配置到独立的 properties 文件。

这样可以将散落在不同 XML 文件中的配置集中管理，而且也方便运维根据不同的环境进行配置不同的值。这个非常实用的功
PropertyPlaceholderConfigurer 负责实现的。

根据前文，当 BeanFactory 在第一阶段加载完所有配置信息时，BeanFactory 中保存的对象的属性还是以占位符方式存在的，比如 ${jdbc.mysql.url}

当 PropertyPlaceholderConfigurer 作为 BeanFactoryPostProcessor 被应用时，它会使用 properties 配置文件中的值来替换相应的 BeanDefiniti
符所表示的属性值。

当需要实例化 bean 时，bean 定义中的属性值就已经被替换成我们配置的值。

当然其实现比上面描述的要复杂一些，这里仅说明其大致工作原理，更详细的实现可以参考其源码。

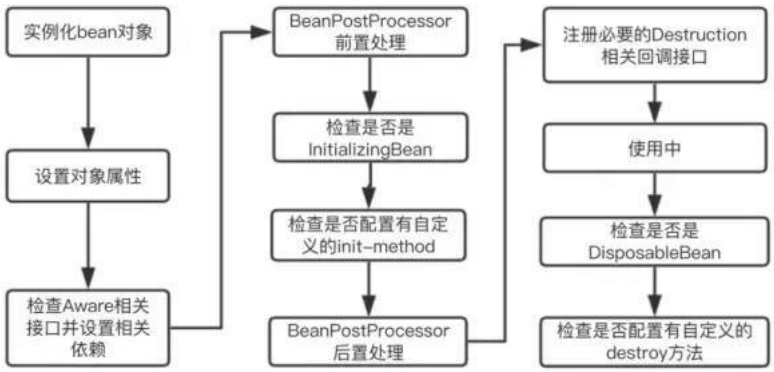
与之相似的，还有 BeanPostProcessor，其存在于对象实例化阶段。跟 BeanFactoryPostProcessor 类似，它会处理容器内所有符合条件并且已经实对象。

简单的对比，BeanFactoryPostProcessor 处理 bean 的定义，而 BeanPostProcessor 则处理 bean 完成实例化后的对象。

BeanPostProcessor 定义了两个接口：

```
public interface BeanPostProcessor {
    // 前置处理
   Obje  postProcessBeforeInitiali Obje  bean Stri  beanNam thro  BeansExcept ;
   ct   zation(                    ct   ,     ng   e)       ws    ion
    // 后置处理
   Obje  postProcessAfterInitializ Obje  bean Stri  beanNam thro  BeansExcept ;
   ct   ation(                     ct   ,     ng   e)       ws    ion
}
```

为了理解这两个方法执行的时机，简单的了解下 bean 的整个生命周期：



Bean 的实例化过程（来自：Spring 揭秘）

postProcessBeforeInitialization()方法与 postProcessAfterInitialization() 分别对应图中前置处理和后置处理两个步骤将执行的方法。

这两个方法中都传入了 bean 对象实例的引用，为扩展容器的对象实例化过程提供了很大便利，在这儿几乎可以对传入的实例执行任何操作。

注解、AOP 等功能的实现均大量使用了 BeanPostProcessor，比如有一个自定义注解，你完全可以实现 BeanPostProcessor 的接口，在其中判断 b的脑袋上是否有该注解。

如果有，你可以对这个 bean 实例执行任何操作，想想是不是非常的简单?

再来看一个更常见的例子，在 Spring 中经常能够看到各种各样的 Aware 接口，其作用就是在对象实例化完成以后将 Aware 接口定义中规定的依赖注实例中。

比如最常见的 ApplicationContextAware 接口，实现了这个接口的类都可以获取到一个 ApplicationContext 对象。

当容器中每个对象的实例化过程走到 BeanPostProcessor 前置处理这一步时，容器会检测到之前注册到容器的 ApplicationContextAwareProcessor，

然后就会调用其 postProcessBeforeInitialization() 方法，检查并设置 Aware 相关依赖。

看看代码吧，是不是很简单：

```
// 代码来自：org.springframework.context.support.ApplicationContextAwareProcessor
// 其postProcessBeforeInitialization方法调用了invokeAwareInterfaces方法
private void invokeAwareInterfaces(Object bean) {
    if (bean instanceof EnvironmentAware) {
        ((EnvironmentAware) bean).setEnvironment(this.applicationContext.getEnvironment());
    }
    if (bean instanceof ApplicationContextAware) {
        ((ApplicationContextAware) bean).setApplicationContext(this.applicationContext);
    }
    // ......
}
```

最后总结一下，本小节内容和你一起回顾了 Spring 容器的部分核心内容，限于篇幅不能写更多，但理解这部分内容，足以让您轻松理解 Spring Boot 理。

如果在后续的学习过程中遇到一些晦涩难懂的知识，再回过头来看看 Spring 的核心知识，也许有意想不到的效果。

也许 Spring Boot 的中文资料很少，但 Spring 的中文资料和书籍有太多太多，总有东西能给你启发。

夯实基础：JavaConfig 与常见 Annotation

JavaConfig

我们知道 bean 是 Spring IOC 中非常核心的概念，Spring 容器负责 bean 的生命周期的管理。

在最初，Spring 使用 XML 配置文件的方式来描述 bean 的定义以及相互间的依赖关系，但随着 Spring 的发展，越来越多的人对这种方式表示不满。

因为 Spring 项目的所有业务类均以 bean 的形式配置在 XML 文件中，造成了大量的 XML 文件，使项目变得复杂且难以管理。

后来，基于纯 Java Annotation 依赖注入框架 Guice 出世，其性能明显优于采用 XML 方式的 Spring。

甚至有部分人认为， Guice 可以完全取代 Spring（ Guice 仅是一个轻量级 IOC 框架，取代 Spring 还差的挺远）。

正是这样的危机感，促使 Spring 及社区推出并持续完善了 JavaConfig 子项目，它基于 Java 代码和 Annotation 注解来描述 bean 之间的依赖绑定关

比如，下面是使用 XML 配置方式来描述 bean 的定义：

而基于 JavaConfig 的配置形式是这样的：

```
@Configuration
public class MoonBookConfiguration {

    // 任何标志了@Bean的方法，其返回值将作为一个bean注册到Spring的IOC容器中
    // 方法名默认成为该bean定义的id
    @Bean
    public BookService bookService() {
        return new BookServiceImpl();
    }
}
```

如果两个 bean 之间有依赖关系的话，在 XML 配置中应该是这样：

```
<bean id="bookService" class="cn.moondev.service.BookServiceImpl">
    <property name="dependencyService" ref="dependencyService"/>
</bean>

<bean id="otherService" class="cn.moondev.service.OtherServiceImpl">
    <property name="dependencyService" ref="dependencyService"/>
</bean>

<bean id="dependencyService" class="DependencyServiceImpl"/>
```

而在 JavaConfig 中则是这样：

```
@Configuration
public class MoonBookConfiguration {

    // 如果一个bean依赖另一个bean，则直接调用对应JavaConfig类中依赖bean的创建方法即可
    // 这里直接调用dependencyService()
    @Bean
    public BookService bookService() {
        return new BookServiceImpl(dependencyService());
    }

    @Bean
    public OtherService otherService() {
        return new OtherServiceImpl(dependencyService());
    }

    @Bean
    public DependencyService dependencyService() {
        return new DependencyServiceImpl();
    }
}
```

你可能注意到这个示例中，有两个 bean 都依赖于 dependencyService，也就是说当初始化 bookService 时会调用 dependencyService()，otherService 时也会调用 dependencyService()，那么问题来了？

这时候 IOC 容器中是有一个 dependencyService 实例还是两个？这个问题留着大家思考吧，这里不再赘述。

@ComponentScan

@ComponentScan 注解对应XML配置形式中的 <context:component-scan> 元素，表示启用组件扫描，Spring 会自动扫描所有通过注解配置的 be
将其注册到 IOC 容器中。

我们可以通过 basePackages 等属性来指定 @ComponentScan 自动扫描的范围，如果不指定，默认从声明 @ComponentScan 所在类的 packag
描。正因为如此，SpringBoot 的启动类都默认在 src/main/java 下。

@Import

@Import 注解用于导入配置类，举个简单的例子：

```
@Configuration
public class MoonBookConfiguration {
    @Bean
    public BookService bookService() {
        return new BookServiceImpl();
    }
}
```

现在有另外一个配置类，比如：MoonUserConfiguration，这个配置类中有一个 bean 依赖于 MoonBookConfiguration 中的 bookService，如何
bean 组合在一起？

借助 @Import 即可：

```
@Configuration
// 可以同时导入多个配置类，比如：@Import({A.class,B.class})
@Import(MoonBookConfiguration.class)
public class MoonUserConfiguration {
    @Bean
    public UserService userService(BookService bookService) {
        return new BookServiceImpl(bookService);
    }
}
```

需要注意的是，在 4.2 之前， @Import 注解只支持导入配置类，但是在 4.2 之后，它支持导入普通类，并将这个类作为一个 bean 的定义注册到中。

@Conditional

@Conditional 注解表示在满足某种条件后才初始化一个 bean 或者启用某些配置。

它一般用在由 @Component、@Service、@Configuration 等注解标识的类上面，或者由 @Bean 标记的方法上。

如果一个 @Configuration 类标记了 @Conditional，则该类中所有标识了 @Bean 的方法和 @Import 注解导入的相关类将遵从这些条件。

在 Spring 里可以很方便的编写你自己的条件类，所要做的就是实现 Condition 接口，并覆盖它的 matches()方法。

举个例子，下面的简单条件类表示只有在 Classpath 里存在 JdbcTemplate 类时才生效：

```
public class JdbcTemplateCondition implements Condition {
    @Override
   publ bool  match ConditionCo  conditionCon AnnotatedTypeM  annotatedTypeMet
   ic   ean  es(   ntext      text,        etadata       adata) {
        try {
        conditionContext.getClassLoader( "org.springframework.jdbc.core.Jdb )
).loadClass(                            cTemplate"                          ;
            return true;
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        return false;
    }
}
```

当你用 Java 来声明 bean 的时候，可以使用这个自定义条件类：

```
@Conditional(JdbcTemplateCondition.class)
@Service
public MyService service() {
    ......
}
```

这个例子中只有当 JdbcTemplateCondition 类的条件成立时才会创建 MyService 这个 bean。

也就是说 MyService 这 bean 的创建条件是 classpath 里面包含 JdbcTemplate，否则这个 bean 的声明就会被忽略掉。

Spring Boot 定义了很多有趣的条件，并把他们运用到了配置类上，这些配置类构成了 Spring Boot 的自动配置的基础。

Spring Boot 运用条件化配置的方法是：定义多个特殊的条件化注解，并将它们用到配置类上。

下面列出了 Spring Boot 提供的部分条件化注解：

条件化注解	配置生效条件
@ConditionalOnBean	配置了某个特定bean
@ConditionalOnMissingBean	没有配置特定的bean
@ConditionalOnClass	Classpath里有指定的类
@ConditionalOnMissingClass	Classpath里没有指定的类
@ConditionalOnExpression	给定的Spring Expression Language表达式计算结果为true
@ConditionalOnJava	Java的版本匹配特定指或者一个范围值
@ConditionalOnProperty	指定的配置属性要有一个明确的值
@ConditionalOnResource	Classpath里有指定的资源
@ConditionalOnWebApplication	这是一个Web应用程序
@ConditionalOnNotWebApplication	这不是一个Web应用程序

@ConfigurationProperties与@EnableConfigurationProperties

当某些属性的值需要配置的时候，我们一般会在 application.properties 文件中新建配置项，然后在 bean 中使用 @Value 注解来获取配置的值。

比如下面配置数据源的代码：

```
// jdbc config
jdbc.mysql.url=jdbc:mysql://localhost:3306/sampledb
jdbc.mysql.username=root
jdbc.mysql.password=123456
......
// 配置数据源
@Configuration
public class HikariDataSourceConfiguration {

    @Value("jdbc.mysql.url")
    public String url;
    @Value("jdbc.mysql.username")
    public String user;
    @Value("jdbc.mysql.password")
    public String password;

    @Bean
    public HikariDataSource dataSource() {
        HikariConfig hikariConfig = new HikariConfig();
        hikariConfig.setJdbcUrl(url);
        hikariConfig.setUsername(user);
        hikariConfig.setPassword(password);
        // 省略部分代码
        return new HikariDataSource(hikariConfig);
    }
}
```

使用 @Value 注解注入的属性通常都比较简单，如果同一个配置在多个地方使用，也存在不方便维护的问题（考虑下，如果有几十个地方在使用某个现在你想改下名字，你该怎么做？）。

对于更为复杂的配置，Spring Boot 提供了更优雅的实现方式，那就是 @ConfigurationProperties 注解。

我们可以通过下面的方式来改写上面的代码：

```
@Component
//  还可以通过@PropertySource("classpath:jdbc.properties")来指定配置文件
@ConfigurationProperties("jdbc.mysql")
// 前缀=jdbc.mysql，会在配置文件中寻找jdbc.mysql.*的配置项
pulic class JdbcConfig {
    public String url;
    public String username;
    public String password;
}

@Configuration
public class HikariDataSourceConfiguration {

    @AutoWired
    public JdbcConfig config;

    @Bean
    public HikariDataSource dataSource() {
        HikariConfig hikariConfig = new HikariConfig();
        hikariConfig.setJdbcUrl(config.url);
        hikariConfig.setUsername(config.username);
        hikariConfig.setPassword(config.password);
        // 省略部分代码
        return new HikariDataSource(hikariConfig);
    }
}
```

@ConfigurationProperties 对于更为复杂的配置，处理起来也是得心应手，比如有如下配置文件：

```
#App
app.menus[0].title=Home
app.menus[0].name=Home
app.menus[0].path=/
app.menus[1].title=Login
app.menus[1].name=Login
app.menus[1].path=/login

app.compiler.timeout=5
app.compiler.output-folder=/temp/

app.error=/error/
```

可以定义如下配置类来接收这些属性：

```
@Component
@ConfigurationProperties("app")
public class AppProperties {

    public String error;
    public List<Menu> menus = new ArrayList<>();
    public Compiler compiler = new Compiler();

    public static class Menu {
        public String name;
        public String path;
        public String title;
    }

    public static class Compiler {
        public String timeout;
        public String outputFolder;
    }
}
```

@EnableConfigurationProperties 注解表示对 @ConfigurationProperties 的内嵌支持，默认会将对应 Properties Class 作为 bean 注入的 IOC 容器
相应的 Properties 类上不用加 @Component 注解。

削铁如泥：SpringFactoriesLoader 详解

JVM 提供了 3 种类加载器： BootstrapClassLoader、 ExtClassLoader、 AppClassLoader 分别加载 Java 核心类库、扩展类库以及应用的
CLASSPATH)下的类库。

JVM 通过双亲委派模型进行类的加载，我们也可以通过继承 java.lang.classloader 实现自己的类加载器。

何为双亲委派模型？当一个类加载器收到类加载任务时，会先交给自己的父加载器去完成。

因此最终加载任务都会传递到最顶层的 BootstrapClassLoader，只有当父加载器无法完成加载任务时，才会尝试自己来加载。

采用双亲委派模型的一个好处是保证使用不同类加载器最终得到的都是同一个对象，这样就可以保证 Java 核心库的类型安全。

比如，加载位于 rt.jar 包中的 java.lang.Object 类，不管是哪个加载器加载这个类，最终都是委托给顶层的 BootstrapClassLoader 来加载的，这样就任何的类加载器最终得到的都是同样一个 Object 对象。

查看 ClassLoader 的源码，对双亲委派模型会有更直观的认识：

```
protected Class<?> loadClass(String name, boolean resolve) {
    synchronized (getClassLoadingLock(name)) {
    // 首先，检查该类是否已经被加载，如果从JVM缓存中找到该类，则直接返回
    Class<?> c = findLoadedClass(name);
    if (c == null) {
        try {
            // 遵循双亲委派的模型，首先会通过递归从父加载器开始找，
            // 直到父类加载器是BootstrapClassLoader为止
            if (parent != null) {
                c = parent.loadClass(name, false);
            } else {
                c = findBootstrapClassOrNull(name);
            }
        } catch (ClassNotFoundException e) {}
        if (c == null) {
            // 如果还找不到，尝试通过findClass方法去寻找
            // findClass是留给开发者自己实现的，也就是说
            // 自定义类加载器时，重写此方法即可
           c = findClass(name);
        }
    }
    if (resolve) {
        resolveClass(c);
    }
    return c;
    }
}
```

但双亲委派模型并不能解决所有的类加载器问题，比如，Java 提供了很多服务提供者接口(ServiceProviderInterface，SPI)，允许第三方为这些接现。

常见的 SPI 有 JDBC、JNDI、JAXP 等，这些 SPI 的接口由核心类库提供，却由第三方实现。

这样就存在一个问题：SPI 的接口是 Java 核心库的一部分，是由 BootstrapClassLoader 加载的；SPI 实现的 Java 类一般是由 AppClassLoader 来

BootstrapClassLoader 是无法找到 SPI 的实现类的，因为它只加载 Java 的核心库。

它也不能代理给 AppClassLoader，因为它是最顶层的类加载器。也就是说，双亲委派模型并不能解决这个问题。

线程上下文类加载器(ContextClassLoader)正好解决了这个问题。从名称上看，可能会误解为它是一种新的类加载器，实际上，它仅仅是 Thread 类量而已。

可以通过 setContextClassLoader(ClassLoadercl)和 getContextClassLoader()来设置和获取该对象。

如果不做任何的设置，Java 应用的线程的上下文类加载器默认就是 AppClassLoader。

在核心类库使用 SPI 接口时，传递的类加载器使用线程上下文类加载器，就可以成功的加载到 SPI 实现的类。

线程上下文类加载器在很多 SPI 的实现中都会用到。但在 JDBC 中，你可能会看到一种更直接的实现方式。

比如，JDBC 驱动管理 java.sql.Driver 中的 loadInitialDrivers() 方法中，你可以直接看到 JDK 是如何加载驱动的：

```
for (String aDriver : driversList) {
    try {
        // 直接使用AppClassLoader
        Class.forName(aDriver, true, ClassLoader.getSystemClassLoader());
    } catch (Exception ex) {
        println("DriverManager.Initialize: load failed: " + ex);
    }
}
```

其实讲解线程上下文类加载器，最主要是让大家在看到 Thread.currentThread().getClassLoader()和Thread.currentThread().getContextClassLoade
一脸懵逼。

这两者除了在许多底层框架中取得的 ClassLoader 可能会有所不同外，其他大多数业务场景下都是一样的，大家只要知道它是为了解决什么问题而
可。

类加载器除了加载 class 外，还有一个非常重要功能，就是加载资源，它可以从 jar 包中读取任何资源文件。

比如， ClassLoader.getResources(Stringname) 方法就是用于读取 jar 包中的资源文件，其代码如下：

```
public Enumeration<URL> getResources(String name) throws IOException {
    Enumeration<URL>[] tmp = (Enumeration<URL>[]) new Enumeration<?>[2];
    if (parent != null) {
        tmp[0] = parent.getResources(name);
    } else {
        tmp[0] = getBootstrapResources(name);
    }
    tmp[1] = findResources(name);
    return new CompoundEnumeration<>(tmp);
}
```

是不是觉得有点眼熟，没错，它的逻辑其实跟类加载的逻辑是一样的，首先判断父类加载器是否为空，不为空则委托父类加载器执行资源查找任
BootstrapClassLoader，最后才轮到自己查找。

而不同的类加载器负责扫描不同路径下的 jar 包，就如同加载 class 一样，最后会扫描所有的 jar 包，找到符合条件的资源文件。

类加载器的 findResources(name) 方法会遍历其负责加载的所有 jar 包，找到 jar 包中名称为 name 的资源文件，这里的资源可以是任何文件，甚至
文件。

比如下面的示例，用于查找 Array.class 文件：

```
// 寻找Array.class文件
public static void main(String[] args) throws Exception{
    // Array.class的完整路径
    String name = "java/sql/Array.class";
   Enumerat <URL> urls Thre .currentThread().getContextClassLoader().getReso
   ion       =         ad   urces(name);
    while (urls.hasMoreElements()) {
        URL url = urls.nextElement();
        System.out.println(url.toString());
    }
}
```

运行后可以得到如下结果：

根据资源文件的 URL，可以构造相应的文件来读取资源内容。

看到这里，你可能会感到挺奇怪的，你不是要详解 SpringFactoriesLoader 吗？上来讲了一堆 ClassLoader 是几个意思？

看下它的源码你就知道了：

```
publi  stati  fina  Strin  FACTORIES_RESOURCE_LOCATIO "META-INF/spring.factori ;
c      c      l     g      N =                         es"
// spring.factories文件的格式为: key=value1,value2,value3
// 从所有的jar包中找到META-INF/spring.factories文件
// 然后从文件中解析出key=factoryClass类名称的所有value值
publ   stat   Lis<Stri > loadFactoryNa Clas<?> factoryCla ClassLoad classLoader
ic     ic     t    ng    mes(           s    ss,           er       ) {
    String factoryClassName = factoryClass.getName();
    // 取得资源文件的URL
  Enume<URL> urls = n  ? classLoader.getResource Class.getSystemResources(FA
  ratio(classLoader  u s(FACTORIES_RESOURCE_LOCAT LoadeCTORIES_RESOURCE_LOCAT
  n     !=           l ION) :                    r    ION));
                     l
    List<String> result = new ArrayList<String>();
    // 遍历所有的URL
    while (urls.hasMoreElements()) {
        URL url = urls.nextElement();
        // 根据资源文件URL解析properties文件
       Properti  properties PropertiesLoaderU .loadProperti ne  UrlResour (url)
       es        =          tils              es(           w   ce        );
        String factoryClassNames = properties.getProperty(factoryClassName);
        // 组装数据，并返回
        result.a Arra .asLisStringUt .commaDelimitedListToStringArray(factory
ddAll(           ys   t(    ils      ClassNames)));
    }
    return result;
}
```

有了前面关于 ClassLoader 的知识，再来理解这段代码，是不是感觉豁然开朗：从 CLASSPATH 下的每个 jar 包中搜寻所有 META-INF/spring.fac
置文件，然后将解析 properties 文件，找到指定名称的配置后返回。

需要注意的是，其实这里不仅仅是会去 ClassPath 路径下查找，会扫描所有路径下的 jar 包，只不过这个文件只会在 ClassPath 下的 jar 包中。

来简单看下 spring.factories 文件的内容吧：

```
// 来自 org.springframework.boot.autoconfigure下的META-INF/spring.factories
// EnableAutoConfiguration后文会讲到，它用于开启Spring Boot自动配置功能
org.springframework.boot.autoconfigure.EnableAutoConfiguration=
org.springframework.boot.autoconfigure.SpringApplicationAdminJmxAutoConfigu ,
admin.                                 ration
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration
```

执行 loadFactoryNames(EnableAutoConfiguration.class,classLoader)后，得到对应的一组 @Configuration 类。

我们就可以通过反射实例化这些类然后注入到 IOC 容器中，最后容器里就有了一系列标注了 @Configuration 的 JavaConfig 形式的配置类。

这就是 SpringFactoriesLoader，它本质上属于 Spring 框架私有的一种扩展方案，类似于 SPI，Spring Boot 在 Spring 基础上的很多核心功能都是基
望大家可以理解。

另一件武器：Spring 容器的事件监听机制

过去，事件监听机制多用于图形界面编程，比如：点击按钮、在文本框输入内容等操作被称为事件。

而当事件触发时，应用程序作出一定的响应则表示应用监听了这个事件，而在服务器端，事件的监听机制更多的用于异步通知以及监控和异常处理。

Java 提供了实现事件监听机制的两个基础类：自定义事件类型扩展自 java.util.EventObject、事件的监听器扩展自 java.util.EventListener。

来看一个简单的实例：简单的监控一个方法的耗时。

首先定义事件类型，通常的做法是扩展 EventObject，随着事件的发生，相应的状态通常都封装在此类中：

```
public class MethodMonitorEvent extends EventObject {
    // 时间戳，用于记录方法开始执行的时间
    public long timestamp;

    public MethodMonitorEvent(Object source) {
        super(source);
    }
}
```

事件发布之后，相应的监听器即可对该类型的事件进行处理，我们可以在方法开始执行之前发布一个 begin 事件。

在方法执行结束之后发布一个 end 事件，相应地，事件监听器需要提供方法对这两种情况下接收到的事件进行处理：

```
// 1、定义事件监听接口
public interface MethodMonitorEventListener extends EventListener {
    // 处理方法执行之前发布的事件
    public void onMethodBegin(MethodMonitorEvent event);
    // 处理方法结束时发布的事件
    public void onMethodEnd(MethodMonitorEvent event);
}
// 2、事件监听接口的实现：如何处理
publi clas AbstractMethodMonitorEventLis implemen MethodMonitorEventList
c     s    tener                        ts       ener                    {

    @Override
    public void onMethodBegin(MethodMonitorEvent event) {
        // 记录方法开始执行时的时间
        event.timestamp = System.currentTimeMillis();
    }

    @Override
    public void onMethodEnd(MethodMonitorEvent event) {
        // 计算方法耗时
        long duration = System.currentTimeMillis() - event.timestamp;
        System.out.println("耗时：" + duration);
    }
}
```

事件监听器接口针对不同的事件发布实际提供相应的处理方法定义，最重要的是，其方法只接收 MethodMonitorEvent 参数，说明这个监听器类只负
对应的事件并进行处理。

有了事件和监听器，剩下的就是发布事件，然后让相应的监听器监听并处理。

通常情况，我们会有一个事件发布者，它本身作为事件源，在合适的时机，将相应的事件发布给对应的事件监听器：

```
public class MethodMonitorEventPublisher {

   priva Lis<MethodMonitorEventL > listener ne ArrayL <MethodMonitorEventL >()
   te    t   istener             s =        w  ist     istener             ;

    public void methodMonitor() {
        MethodMonitorEvent eventObject = new MethodMonitorEvent(this);
        publishEvent("begin",eventObject);
        // 模拟方法执行：休眠5秒钟
        TimeUnit.SECONDS.sleep(5);
        publishEvent("end",eventObject);

    }

    private void publishEvent(String status,MethodMonitorEvent event) {
        // 避免在事件处理期间，监听器被移除，这里为了安全做一个复制操作
      Li <MethodMonitorEven > copyListenerne ArrayL<MethodMonitorEven >(listen
      st  tListener         s = ➥         w  ist    tListener         ers);
        for (MethodMonitorEventListener listener : copyListeners) {
            if ("begin".equals(status)) {
                listener.onMethodBegin(event);
            } else {
                listener.onMethodEnd(event);
            }
        }
    }

    public static void main(String[] args) {
       MethodMonitorEventPublish  publisher =ne  MethodMonitorEventPublish ()
       er                                    w   er                        ;
        publisher.addEventListener ne  AbstractMethodMonitorEventListene ())
(                                  w   r                                  ;
        publisher.methodMonitor();
    }
    // 省略实现
    public void addEventListener(MethodMonitorEventListener listener) {}
    public void removeEventListener(MethodMonitorEventListener listener) {}
    public void removeAllListeners() {}
```

对于事件发布者（事件源）通常需要关注两点：

- 在合适的时机发布事件。此例中的 methodMonitor() 方法是事件发布的源头，其在方法执行之前和结束之后两个时间点发布 MethodMonitorEven
每个时间点发布的事件都会传给相应的监听器进行处理。 在具体实现时需要注意的是，事件发布是顺序执行，为了不影响处理性能，事件监听器的
应尽量简单。
- 事件监听器的管理。publisher 类中提供了事件监听器的注册与移除方法，这样客户端可以根据实际情况决定是否需要注册新的监听器或者移除某个
器。 如果这里没有提供 remove 方法，那么注册的监听器示例将一直被 MethodMonitorEventPublisher 引用，即使已经废弃不用了，也依然在发
听器列表中，这会导致隐性的内存泄漏。

Spring 容器内的事件监听机制

Spring 的 ApplicationContext 容器内部中的所有事件类型均继承自 org.springframework.context.AppliationEvent。

容器中的所有监听器都实现 org.springframework.context.ApplicationListener 接口，并且以 bean 的形式注册在容器中。

一旦在容器内发布 ApplicationEvent 及其子类型的事件，注册到容器的 ApplicationListener 就会对这些事件进行处理。

你应该已经猜到是怎么回事了。

ApplicationEvent 继承自 EventObject，Spring 提供了一些默认的实现，比如：ContextClosedEvent 表示容器在即将关闭时发布的事
ContextRefreshedEvent 表示容器在初始化或者刷新的时候发布的事件类型.....

容器内部使用 ApplicationListener 作为事件监听器接口定义，它继承自 EventListener。

ApplicationContext 容器在启动时，会自动识别并加载 EventListener 类型的bean，一旦容器内有事件发布，将通知这些注册到容器的 EventListene

ApplicationContext 接口继承了 ApplicationEventPublisher 接口，该接口提供了 voidpublishEvent(ApplicationEventevent) 方法定义，不
ApplicationContext 容器担当的就是事件发布者的角色。

如果有兴趣可以查看 AbstractApplicationContext.publishEvent(ApplicationEventevent) 方法的源码：ApplicationContext 将事件的发布以及监听器
作委托给 ApplicationEventMulticaster 接口的实现类。

在容器启动时，会检查容器内是否存在名为 applicationEventMulticaster 的 ApplicationEventMulticaster 对象实例。

如果有就使用其提供的实现，没有就默认初始化一个 SimpleApplicationEventMulticaster 作为实现。

最后，如果我们业务需要在容器内部发布事件，只需要为其注入 ApplicationEventPublisher 依赖即可实现 ApplicationEventPublisherAware
ApplicationContextAware 接口(Aware 接口相关内容请回顾上文)。

出神入化：揭秘自动配置原理

典型的 Spring Boot 应用的启动类一般均位于 src/main/java 根路径下，比如 MoonApplication 类：

```
@SpringBootApplication
public class MoonApplication {

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

其中 @SpringBootApplication 开启组件扫描和自动配置，而 SpringApplication.run 则负责启动引导应用程序。

@SpringBootApplication 是一个复合 Annotation，它将三个有用的注解组合在一起：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
        @Filte (type = FilterTyp .CUSTOM, classes = TypeExcludeFilte .clas )
        r              e                          r                s    ,
       @Filt (type  FilterTy.CUSTOM, classe AutoConfigurationExclude .clas) }
       er    =      pe       s =            Filter                   s    )
public @interface SpringBootApplication {
    // ......
}
```

@SpringBootConfiguration 就是 @Configuration，它是 Spring 框架的注解，标明该类是一个 JavaConfig 配置类。

而 @ComponentScan 启用组件扫描，前文已经详细讲解过，这里着重关注 @EnableAutoConfiguration。

@EnableAutoConfiguration 注解表示开启 Spring Boot 自动配置功能，Spring Boot 会根据应用的依赖、自定义的 bean、Classpath 下有没有某个
素来猜测你需要的 bean，然后注册到 IOC 容器中。

那 @EnableAutoConfiguration 是如何推算出你的需求？首先看下它的定义：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(EnableAutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {
    // ......
}
```

你的关注点应该在 @Import(EnableAutoConfigurationImportSelector.class)上了。

前文说过，@Import 注解用于导入类，并将这个类作为一个 bean 的定义注册到容器中，这里它将把 EnableAutoConfigurationImportSelector 作为

入到容器中。

而这个类会将所有符合条件的 @Configuration 配置都加载到容器中，看看它的代码：

```
public String[] selectImports(AnnotationMetadata annotationMetadata) {
    // 省略了大部分代码，保留一句核心代码
    // 注意：SpringBoot最近版本中，这句代码被封装在一个单独的方法中
    // SpringFactoriesLoader相关知识请参考前文
    Lis <Strin > factories = ne  ArrayLis <Strin > ne  LinkedHashSe <Strin >(
    t    g                   w   t         g     ( w   t            g
      SpringFactories .loadFactoryN EnableAutoConfigu .cla , thi.beanClassLoade
      Loader          ames(         ration            ss    s  r)));
}
```

这个类会扫描所有的 Jar 包，将所有符合条件的 @Configuration 配置类注入到容器中，何为符合条件，看看 META-INF/spring.factories 的文件内

```
// 来自 org.springframework.boot.autoconfigure下的META-INF/spring.factories
// 配置的key = EnableAutoConfiguration，与代码中一致
org.springframework.boot.autoconfigure.EnableAutoConfiguration=
org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration
.....
```

以 DataSourceAutoConfiguration 为例，看看 Spring Boot 是如何自动配置的：

```
@Configuration
@ConditionalOnClass({ DataSource.class, EmbeddedDatabaseType.class })
@EnableConfigurationProperties(DataSourceProperties.class)
@Impor ({ Registra .clas , DataSourcePoolMetadataProvidersConfigurati.clas  }
t         r         s      on                                          s    )
public class DataSourceAutoConfiguration {
}
```

分别说一说：

- @ConditionalOnClass({DataSource.class,EmbeddedDatabaseType.class})：当 Classpath 中存在 DataSource 或者 EmbeddedDatabaseType 用这个配置，否则这个配置将被忽略。
- @EnableConfigurationProperties(DataSourceProperties.class)：将 DataSource 的默认配置类注入到 IOC 容器中，DataSourceproperties 定义

```
// 提供对datasource配置信息的支持，所有的配置前缀为：spring.datasource
@ConfigurationProperties(prefix = "spring.datasource")
public class DataSourceProperties  {
    private ClassLoader classLoader;
    private Environment environment;
    private String name = "testdb";
    ......
}
```

- @Import({ Registrar.class, DataSourcePoolMetadataProvidersConfiguration.class })：导入其他额外的配置，就以 DataSourcePoolMetadataProvidersConfiguration 为例吧：

```
@Configuration
public class DataSourcePoolMetadataProvidersConfiguration {

    @Configuration
    @ConditionalOnClass(org.apache.tomcat.jdbc.pool.DataSource.class)
    static class TomcatDataSourcePoolMetadataProviderConfiguration {
        @Bean
       publi DataSourcePoolMetadataProv tomcatPoolDataSourceMetadataProvide
       c     ider                       r() {
            .....
        }
    }
  ......
}
```

DataSourcePoolMetadataProvidersConfiguration 是数据库连接池提供者的一个配置类。

即 Classpath 中存在 org.apache.tomcat.jdbc.pool.DataSource.class，则使用 tomcat-jdbc 连接池，如果 Classpath 中存在 HikariDataSource.c 用 Hikari 连接池。

这里仅描述了 DataSourceAutoConfiguration 的冰山一角，但足以说明 Spring Boot 如何利用条件化配置来实现自动配置的。

回顾一下，@EnableAutoConfiguration 中导入了 EnableAutoConfigurationImportSelector 类。

而这个类的 selectImports() 通过 SpringFactoriesLoader 得到了大量的配置类，而每一个配置类则根据条件化配置来做出决策，以实现自动配置。

整个流程很清晰，但漏了一个大问题：EnableAutoConfigurationImportSelector.selectImports() 是何时执行的?

其实这个方法会在容器启动过程中执行： AbstractApplicationContext.refresh()，更多的细节在下一小节中说明。

启动引导：Spring Boot 应用启动的秘密

SpringApplication 初始化

Spring Boot 整个启动流程分为两个步骤：初始化一个 SpringApplication 对象、执行该对象的 run 方法。

看下 SpringApplication 的初始化流程，SpringApplication 的构造方法中调用 initialize(Object[] sources) 方法，其代码如下：

```
private void initialize(Object[] sources) {
     if (sources != null && sources.length > 0) {
         this.sources.addAll(Arrays.asList(sources));
     }
     // 判断是否是Web项目
     this.webEnvironment = deduceWebEnvironment();
     setInitiali Collect ) getSpringFactoriesInsApplicationContextInit .cla ))
zers((           ion     tances(                ializer                ss  ;
     setListener Collecti ) getSpringFactoriesInstan ApplicationListe .clas ))
s((              on       ces(                      ner              s    ;
     // 找到入口类
     this.mainApplicationClass = deduceMainApplicationClass();
}
```

初始化流程中最重要的就是通过 SpringFactoriesLoader 找到 spring.factories 文件中配置的 ApplicationContextInitializer 和 ApplicationListener 的实现类名称，以便后期构造相应的实例。

ApplicationContextInitializer 的主要目的是在 ConfigurableApplicationContext 做 refresh 之前，对 ConfigurableApplicationContext 实例做进一或处理。

ConfigurableApplicationContext 继承自 ApplicationContext，其主要提供了对 ApplicationContext 进行设置的能力。

实现一个 ApplicationContextInitializer 非常简单，因为它只有一个方法，但大多数情况下我们没有必要自定义一个 ApplicationContextInitializer。

即便是 Spring Boot 框架，它默认也只是注册了两个实现，毕竟 Spring 的容器已经非常成熟和稳定，你没有必要来改变它。

而 ApplicationListener 的目的就没什么好说的了，它是 Spring 框架对 Java 事件监听机制的一种框架实现，具体内容在前文 Spring 事件监听机制详细讲解。

这里主要说说，如果你想为 Spring Boot 应用添加监听器，该如何实现?

Spring Boot 提供两种方式来添加自定义监听器:

- 通过 SpringApplication.addListeners(ApplicationListener<?>...listeners)或者 SpringApplication.setListeners(Collection<? extendsApplicationListener<?>>listeners)两个方法来添加一个或者多个自定义监听器。
- 既然 SpringApplication 的初始化流程中已经从 spring.factories 中获取到 ApplicationListener 的实现类，那么我们直接在自己的 jar 包的 META-INF/spring.factories 文件中新增配置即可:

关于 SpringApplication 的初始化，我们就说这么多。

Spring Boot 启动流程

Spring Boot 应用的整个启动流程都封装在 SpringApplication.run 方法中，其整个流程真的是太长太长了，但本质上就是在 Spring 容器启动的基础量的扩展，按照这个思路来看看源码:

```
public ConfigurableApplicationContext run(String... args) {
        StopWatch stopWatch = new StopWatch();
        stopWatch.start();
        ConfigurableApplicationContext context = null;
        FailureAnalyzers analyzers = null;
        configureHeadlessProperty();
        // ①
        SpringApplicationRunListeners listeners = getRunListeners(args);
        listeners.starting();
        try {
            // ②
          ApplicationArgum applicationArgumen ne DefaultApplicationArgu(args
          ents            ts =              w  ments                  );
         ConfigurableEnvir environment = prepareEnvironment(listeners,applic
         onment           ationArguments);
            // ③
            Banner printedBanner = printBanner(environment);
            // ④
            context = createApplicationContext();
            // ⑤
            analyzers = new FailureAnalyzers(context);
            // ⑥
            prepareContext(context, environment, listeners, applicationArgum
ents,printedBanner);
            // ⑦
            refreshContext(context);
            // ⑧
            afterRefresh(context, applicationArguments);
            // ⑨
            listeners.finished(context, null);
            stopWatch.stop();
            return context;
        }
        catch (Throwable ex) {
            handleRunFailure(context, listeners, analyzers, ex);
            throw new IllegalStateException(ex);
        }
    }
```

①通过 SpringFactoriesLoader 查找并加载所有的 SpringApplicationRunListeners。

通过调用 starting() 方法通知所有的 SpringApplicationRunListeners：应用开始启动了。

SpringApplicationRunListeners 其本质上就是一个事件发布者，它在 SpringBoot 应用启动的不同时间点发布不同应用事件类型(ApplicationEvent)。

如果有哪些事件监听者(ApplicationListener)对这些事件感兴趣，则可以接收并且处理。

还记得初始化流程中，SpringApplication 加载了一系列 ApplicationListener 吗?

这个启动流程中没有发现有发布事件的代码，其实都已经在 SpringApplicationRunListeners 这儿实现了。

简单的分析一下其实现流程，首先看下 SpringApplicationRunListener 的源码：

```
public interface SpringApplicationRunListener {

    // 运行run方法时立即调用此方法，可以用户非常早期的初始化工作
    void starting();

    // Environment准备好后，并且ApplicationContext创建之前调用
    void environmentPrepared(ConfigurableEnvironment environment);

    // ApplicationContext创建好后立即调用
    void contextPrepared(ConfigurableApplicationContext context);

    // ApplicationContext加载完成，在refresh之前调用
    void contextLoaded(ConfigurableApplicationContext context);

    // 当run方法结束之前调用
    voi  finished ConfigurableApplicationContex context, Throwabl  exception)
    d   (         t                                  e        ;
}
```

SpringApplicationRunListener 只有一个实现类： EventPublishingRunListener。

①处的代码只会获取到一个 EventPublishingRunListener 的实例，我们来看看 starting() 方法的内容：

```
public void starting() {
    // 发布一个ApplicationStartedEvent
   thi.initialMulticaster.multic ne ApplicationStarte (thi.applicati thi.args)
   s  astEvent(                 w  dEvent            s  on,        s  );
}
```

顺着这个逻辑，你可以在 ② 处的 prepareEnvironment() 方法的源码中找到 listeners.environmentPrepared(environment)。

即 SpringApplicationRunListener 接口的第二个方法，那不出你所料， environmentPrepared() 又发布了另外一个 ApplicationEnvironmentPreparedEvent。接下来会发生什么，就不用我多说了吧。

②创建并配置当前应用将要使用的 Environment。

Environment 用于描述应用程序当前的运行环境，其抽象了两个方面的内容：配置文件(profile)和属性(properties)。

开发经验丰富的同学对这两个东西一定不会陌生：不同的环境(eg：生产环境、预发布环境)可以使用不同的配置文件，而属性则可以从配置文件、环境变量、命令行参数等来源获取。

因此，当 Environment 准备好后，在整个应用的任何时候，都可以从 Environment 中获取资源。

总结起来，②处的两句代码，主要完成以下几件事：

- 判断 Environment 是否存在，不存在就创建（如果是 Web 项目就创建 StandardServletEnvironment，否则创建 StandardEnvironment）。
- 配置 Environment：配置 profile 以及 properties。
- 调用 SpringApplicationRunListener 的 environmentPrepared() 方法，通知事件监听者：应用的 Environment 已经准备好。

③Spring Boot 应用在启动时会输出这样的东西：

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.5.6.RELEASE)
```

如果想把这个东西改成自己的涂鸦，你可以研究一下 Banner 的实现，这个任务就留给你们吧。

④根据是否是 Web 项目，来创建不同的 ApplicationContext 容器。

⑤创建一系列 FailureAnalyzer。

创建流程依然是通过 SpringFactoriesLoader 获取到所有实现 FailureAnalyzer 接口的 class，然后再创建对应的实例。FailureAnalyzer 用于分析故障关诊断信息。

⑥初始化 ApplicationContext。

主要完成以下工作：

- 将准备好的 Environment 设置给 ApplicationContext。
- 遍历调用所有的 ApplicationContextInitializer 的 initialize() 方法来对已经创建好的 ApplicationContext 进行进一步的处理。
- 调用 SpringApplicationRunListener 的 contextPrepared() 方法，通知所有的监听者：ApplicationContext 已经准备完毕。
- 将所有的 bean 加载到容器中。
- 调用 SpringApplicationRunListener 的 contextLoaded() 方法，通知所有的监听者：ApplicationContext 已经装载完毕。

⑦调用 ApplicationContext 的 refresh() 方法，完成 IOC 容器可用的最后一道工序。

从名字上理解为刷新容器，那何为刷新？就是插手容器的启动，联系一下第一小节的内容。那如何刷新呢？

且看下面代码：

```
// 摘自refresh()方法中一句代码
invokeBeanFactoryPostProcessors(beanFactory);
```

看看这个方法的实现：

```
protect voi  invokeBeanFactoryPostProce ConfigurableListableBeanF  beanFactory
ed       d   ssors(                     actory                    ) {
   PostProcessorRegistrat.invokeBeanFactoryPostProcessors(beanFactory, getBe
   ionDelegate           anFactoryPostProcessors());
     ......
}
```

获取到所有的 BeanFactoryPostProcessor 来对容器做一些额外的操作。BeanFactoryPostProcessor 允许我们在容器实例化相应对象之前，对注册 BeanDefinition 所保存的信息做一些额外的操作。

这里的 getBeanFactoryPostProcessors() 方法可以获取到 3 个 Processor：

```
ConfigurationWarningsApplicationContextInitializer$ConfigurationWarningsPost
Processor
SharedMetadataReaderFactoryContextInitializer$CachingMetadataReaderFactoryPo
stProcessor
ConfigFileApplicationListener$PropertySourceOrderingPostProcessor
```

不是有那么多 BeanFactoryPostProcessor 的实现类，为什么这儿只有这 3 个?

因为在初始化流程获取到的各种 ApplicationContextInitializer 和 ApplicationListener 中，只有上文 3 个做了类似于如下操作：

```
public void initialize(ConfigurableApplicationContext context) {
    context.addBeanFactoryPostPr ne ConfigurationWarningsPostPro(getChecks()
ocessor(                        w   cessor                       ));
}
```

然后你就可以进入到 PostProcessorRegistrationDelegate.invokeBeanFactoryPostProcessors() 方法了。

这个方法除了会遍历上面的 3 个 BeanFactoryPostProcessor 处理外，还会获取类型为 BeanDefinitionRegistryPostProcessor 的 org.springframework.context.annotation.internalConfigurationAnnotationProcessor，对应的 Class 为 ConfigurationClassPostProcessor。

ConfigurationClassPostProcessor 用于解析处理各种注解，包括：

- @Configuration
- @ComponentScan
- @Import
- @PropertySource
- @ImportResource
- @Bean

当处理 @import 注解的时候，就会调用<自动配置>这一小节中的 EnableAutoConfigurationImportSelector.selectImports() 来完成自动配置功能。其不再多讲，如果你有兴趣，可以查阅参考资料 6。

⑧查找当前 context 中是否注册有 CommandLineRunner 和 ApplicationRunner，如果有则遍历执行它们。

⑨执行所有 SpringApplicationRunListener 的 finished() 方法。

这就是 Spring Boot 的整个启动流程，其核心就是在 Spring 容器初始化并启动的基础上加入各种扩展点。

这些扩展点包括：ApplicationContextInitializer、ApplicationListener 以及各种 BeanFactoryPostProcessor 等等。

你对整个流程的细节不必太过关注，甚至没弄明白也没有关系，你只要理解这些扩展点是在何时如何工作的，能让它们为你所用即可。

整个启动流程确实非常复杂，可以查询参考资料中的部分章节和内容，对照着源码，多看看，我想最终你都能弄清楚的。

言而总之，Spring 才是核心，理解清楚 Spring 容器的启动流程，那 Spring Boot 启动流程就不在话下了。

作者：CHEN川

编辑：陶家龙、孙淑娟

出处：http://www.jianshu.com/p/83693d3d0a65 返回搜狐，查看更多