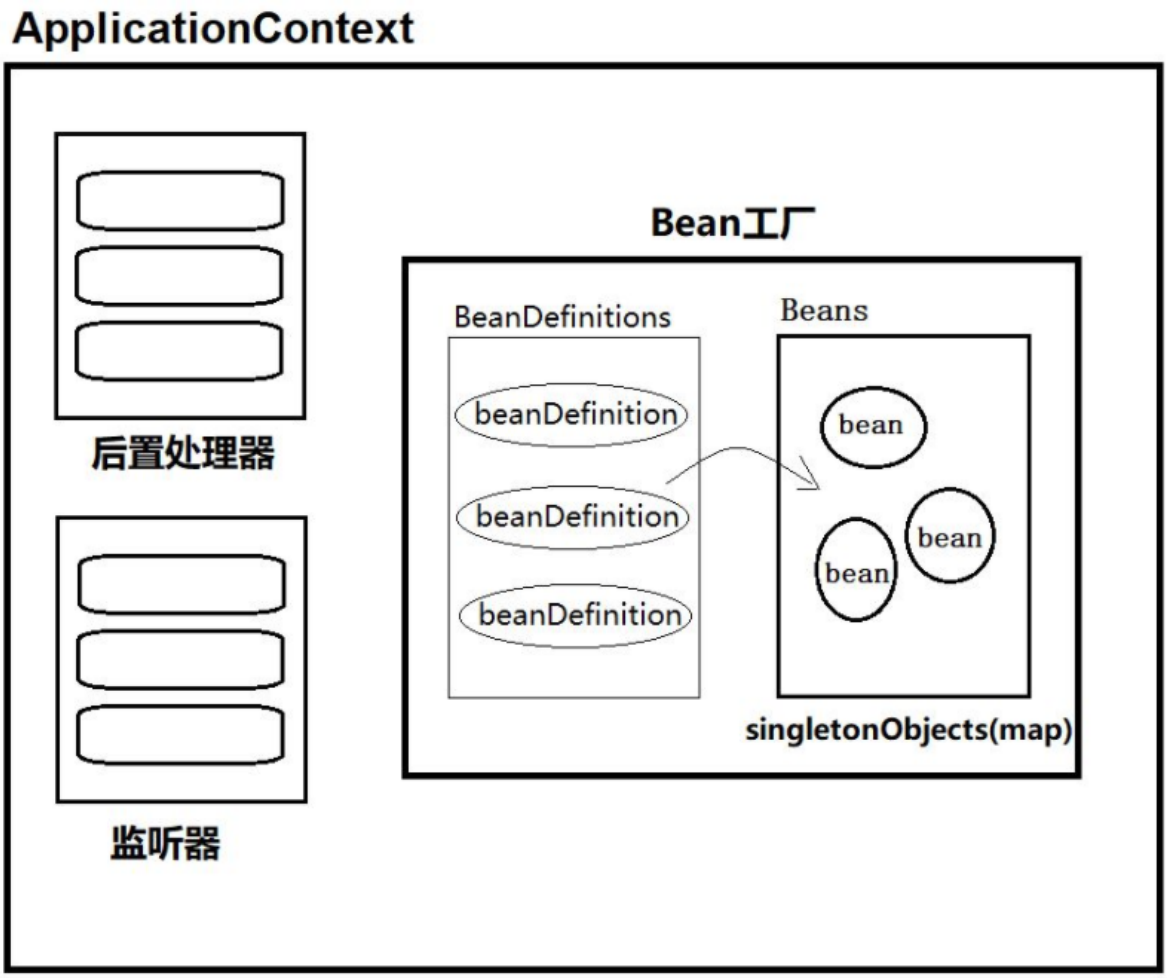
Spring Boot 启动流程



扩展接口	作用
BeanFactoryPostProcessor	处理所有bean前，对bean factory进行预处理
BeanDefinitionRegistryPostProcessor	可以添加自定义的bean
BeanPostProcessor	支持在Bean初始化前、后对bean进行处理
ApplicationContextAware	可以获得ApplicationContext及其中的bean
InitializingBean	在bean创建完成，所有属性注入完成后执行
DisposableBean	在bean销毁前执行
ApplicationListener	用来监听产生的应用事件

0、SpringApplication 构造函数

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

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

```java
public SpringApplication(Class<?>... primarySources) {
    this(null, primarySources);
}

@SuppressWarnings({"unchecked", "rawtypes"})
public SpringApplication(ResourceLoader resourceLoader, Class<?>... primarySources) {
    this.resourceLoader = resourceLoader;
    Assert.notNull(primarySources, "PrimarySources must not be null");
    this.primarySources = new LinkedHashSet<>(Arrays.asList(primarySources));
    //推断应用类型，后面会根据类型初始化对应的环境。常用的一般都是servlet环境
    this.webApplicationType = deduceWebApplicationType();//2.2.1
    //初始化classpath下 META-INF/spring.factories中已配置的ApplicationContextInitializer
    setInitializers((Collection) getSpringFactoriesInstances(ApplicationContextInitializer.class));//2.2.2
    //初始化classpath下所有已配置的 ApplicationListener
    setListeners((Collection) getSpringFactoriesInstances(ApplicationListener.class));//2.2.3
    //根据调用栈，推断出 main 方法的类名
    this.mainApplicationClass = deduceMainApplicationClass();
}
```

1）推断应用类型

根据类路径下存在的类的类型来推断是哪种应用，后面会根据类型初始化对应的环境。常用的一般都是servlet环境。

```java
this.webApplicationType = deduceWebApplicationType();
```

- NONE：不是 WEB 应用
- SERVLET：启用内嵌的 servlet web（tomcat）服务器
- REACTIVE：启用内嵌的 reactive web服务器

2）获取所有的初始化器

ApplicationContextInitializer 初始化器是一个顶级接口，可以有不同的扩展，支持自定义也有系统自带的扩展，通过类似 SPI 的机制，传入一个接口类对象，然后在指定的路径下，以这个类对象的全路径为key，取出文件中的value，即对应的扩展实现类，然后进行实例化的过程（初始化classpath下META-INF/spring.factories中已配置的ApplicationContextInitializer 初始化器）。

```java
setInitializers((Collection)
getSpringFactoriesInstances(ApplicationContextInitializer.class));
```

是在spring容器刷新之前执行的一个回调函数，回调了所有实现了该接口的初始化方法，目的是能在spring容器创建之前做一些预备工作，和定制化的需求，比如在容器刷新之前通过自定义初始化器，修改一些配置或者获取指定的bean都是可以的，或者注册属性源(property sources)或者针对上下文的环境信息environment激活相应的profile。

可以自定义初始化器，也有 SpringBoot内置的初始化器。主要作用是：

a）获取自定义扩展的实现类

就比如这个初始化器的扩展实现类，除了在 Spring指定的配置文件 spring.facyories 中定义并被SpiringFactoryLoader发现外，也可以定义在 spring 应用的配置文件 application.properties下被内置的 DelegatingApplicationContextInitializer 初始化器发现并实例。

自定义初始化器的三种方式

> **支持Order注解。并且代表着执行顺序**
>
> 都需要实现ApplicationInitializer接口

- 定义在META-INF的文件spring.factories会被SpringFactoriesLoader发现注册
- SpringApplication初始化完毕后手动添加

```
@SpringBootApplication
public class MySpringBootApplication {
    public static void main(String[] args) {
        SpringApplication application = new SpringApplication(MySpringBootApplication.class);
        application.addInitializers(new MyApplicationContextInitializer());
         application.run(args);
    }
}
```

- 定义成环境变量被DelegatingApplicationContextinitializer发现注册

b）设置 Spring 应用的上下文ID

c）对于一般配置错误在日志中作出警告

d）将内置servlet容器实际使用的监听端口写入到Environment环境属性中

SpringBoot内置的一些ApplicationContextInitializer

- **DelegatingApplicationContextInitialize**
  使用环境属性context.initializer.classes指定的初始化器(initializers)进行初始化工作，如果没有指定则什么都不做。通过它使得我们可以把自定义实现类配置在application.properties里成为了可能。
- **ContextIdApplicationContextInitializer**
  设置Spring应用上下文的ID,会参照环境属性。至于Id设置为啥值会参考环境属性：
  spring.application.name
  vcap.application.name
  spring.config.name
  spring.application.index
  vcap.application.instance_index

如果这些属性都没有，ID使用application

- **ConfigurationWarningsApplicationContextInitializer**
  对于一般配置错误在日志中作出警告
- **ServerPortInfoApplicationContextInitializer**
  将内置servlet容器实际使用的监听端口写入到Environment环境属性中。这样属性local.server.port就可以直接通过@Value注入到测试中，或者通过环境属性Environment获取。
- **SharedMetadataReaderFactoryContextInitializer**
  创建一个SpringBoot和ConfigurationClassPostProcessor共用的CachingMetadataReaderFactory对象。实现类为：ConcurrentReferenceCachingMetadataReaderFactory
- **ConditionEvaluationReportLoggingListener**
  将ConditionEvaluationReport写入日志。

3）初始化classpath下所有已配置的 ApplicationListener

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

ApplicationListener 的加载过程和上面的 ApplicationContextInitializer 类的加载过程是一样的。ApplicationListener 是spring基于观察者模式的事件监听器，通过 ApplicationEvent 类和 ApplicationListener 接口，可以让开发者实现对spring容器的生命周期的监听，当然也可以自定义监听事件。

```
/**
 * Run the Spring application, creating and refreshing a new
 * {@link ApplicationContext}.
 *
 * @param args the application arguments (usually passed from a Java main
method)
 * @return a running {@link ApplicationContext}
 *
 * 运行spring应用，并刷新一个新的 ApplicationContext（Spring的上下文）
 * ConfigurableApplicationContext 是 ApplicationContext 接口的子接口。在
ApplicationContext
 * 基础上增加了配置上下文的工具。 ConfigurableApplicationContext是容器的高级接口
 */
public ConfigurableApplicationContext run(String... args) {
    //记录程序运行时间
    StopWatch stopWatch = new StopWatch();
    stopWatch.start();
    // ConfigurableApplicationContext Spring 的上下文
    ConfigurableApplicationContext context = null;
    Collection<SpringBootExceptionReporter> exceptionReporters = new ArrayList<>
();
    configureHeadlessProperty();
    //从META-INF/spring.factories中获取监听器
    //1、获取并启动监听器
    SpringApplicationRunListeners listeners = getRunListeners(args);
    listeners.starting();
    try {
        ApplicationArguments applicationArguments = new
DefaultApplicationArguments(
                args);
        //2、构造应用上下文环境
        ConfigurableEnvironment environment = prepareEnvironment(listeners,
applicationArguments);
        //处理需要忽略的Bean
        configureIgnoreBeanInfo(environment);
        //打印banner
        Banner printedBanner = printBanner(environment);
        ///3、初始化应用上下文
        context = createApplicationContext();
        //实例化SpringBootExceptionReporter.class，用来支持报告关于启动的错误
        exceptionReporters = getSpringFactoriesInstances(
                SpringBootExceptionReporter.class,
                new Class[]{ConfigurableApplicationContext.class}, context);
        //4、刷新应用上下文前的准备阶段
        prepareContext(context, environment, listeners, applicationArguments,
printedBanner);
        //5、刷新应用上下文
        refreshContext(context);
```

```
        //刷新应用上下文后的扩展接口
        afterRefresh(context, applicationArguments);
        //时间记录停止
        stopWatch.stop();
        if (this.logStartupInfo) {
            new StartupInfoLogger(this.mainApplicationClass)
                    .logStarted(getApplicationLog(), stopWatch);
        }
        //发布容器启动完成事件
        listeners.started(context);
        callRunners(context, applicationArguments);
    } catch (Throwable ex) {
        handleRunFailure(context, ex, exceptionReporters, listeners);
        throw new IllegalStateException(ex);
    }

    try {
        listeners.running(context);
    } catch (Throwable ex) {
        handleRunFailure(context, ex, exceptionReporters, null);
        throw new IllegalStateException(ex);
    }
    return context;
}
```

总结为

1）获取并启动监听器

2）构造应用上下文环境

3）初始化应用上下文

4）上下文的准备工作

5）刷新应用上下文

6）刷新应用上下文后的扩展接口

1、获取并启动监听器

启动监听器是属于 Spring 事件机制的内容，通过事件机制我们可以监听Spring容器中正在发生的一些事件，同样也可以自定义监听事件。

Spring的事件为Bean和Bean之间的消息传递提供支持。当一个对象处理完某种任务后，通知另外的对象进行某些处理，常用的场景有进行某些操作后发送通知，消息、邮件等情况。

```
private SpringApplicationRunListeners getRunListeners(String[] args) {
    Class<?>[] types = new Class<?>[]{SpringApplication.class, String[].class};
    return new SpringApplicationRunListeners(logger, getSpringFactoriesInstances(
            SpringApplicationRunListener.class, types, this, args));
}
```

获取到 EventPublishingRunListener监听器，是Spring容器的启动监听器。

SpringApplicationRunListeners listeners = getRunListeners(args);

开启了监听事件

listeners.starting();

2、构造应用上下文环境

主要是应用类型、系统环境（包括java运行环境）、项目环境（项目配置文件application.properties/yml）

- **根据相应的应用类型创建并配置相应的环境**

  其中应用类型已经在SpringContext的构造器中推断出来了

- **根据用户的配置，配置系统环境**

  比如可以通过active属性指定是prod生产环境还是dev开发环境还是test测试环境，或者配置JDK版本来配置java运行环境等

- **启动监听器，解析并加载系统配置文件**

  这里主要就是项目的 properties 配置文件中定义的Key和Vlue了，比如可以是应用的名称 application.name 或者是 服务的端口号 server.port等

prepareEnvironment()

```
private ConfigurableEnvironment prepareEnvironment(
        SpringApplicationRunListeners listeners,
        ApplicationArguments applicationArguments) {
    // Create and configure the environment
    //创建并配置相应的环境
    ConfigurableEnvironment environment = getOrCreateEnvironment();
    //根据用户配置，配置 environment系统环境
    configureEnvironment(environment, applicationArguments.getSourceArgs());
    // 启动相应的监听器，其中一个重要的监听器 ConfigFileApplicationListener 就是加载项目配置文件的监听器。
    listeners.environmentPrepared(environment);
    bindToSpringApplication(environment);
    if (this.webApplicationType == WebApplicationType.NONE) {
        environment = new EnvironmentConverter(getClassLoader())
                .convertToStandardEnvironmentIfNecessary(environment);
    }
    ConfigurationPropertySources.attach(environment);
    return environment;
}
```

1）ConfigurableEnvironment environment = getOrCreateEnvironment();

看看getOrCreateEnvironment()干了些什么。

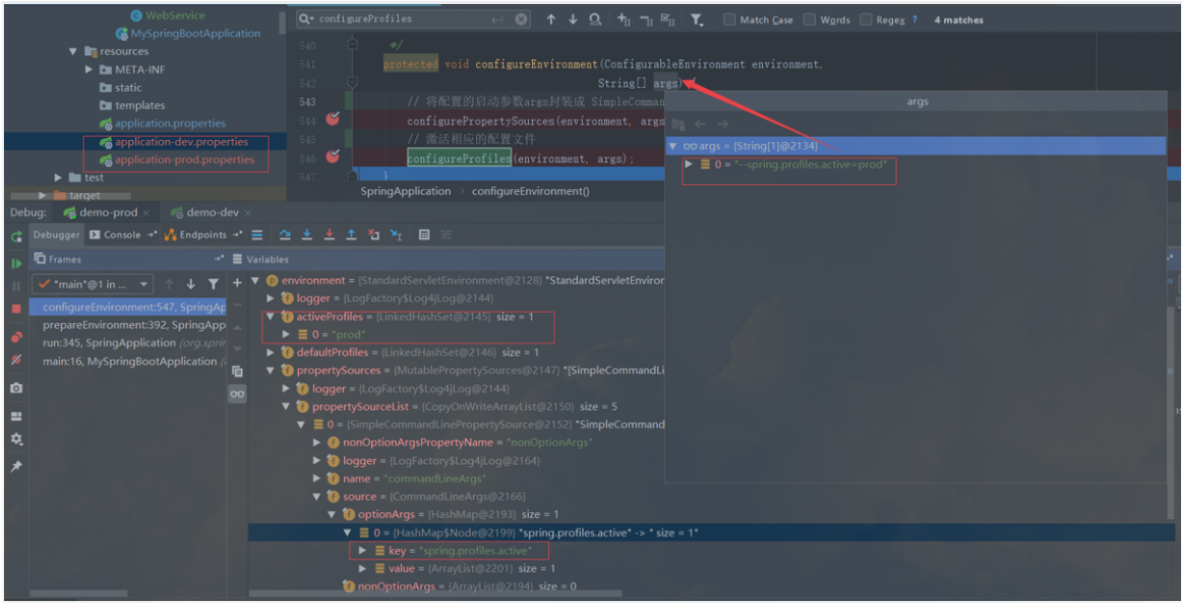
```
private ConfigurableEnvironment getOrCreateEnvironment() {
    if (this.environment != null) {
        return this.environment;
    }
    //如果应用类型是 SERVLET 则实例化 StandardServletEnvironment
    if (this.webApplicationType == WebApplicationType.SERVLET) {
        return new StandardServletEnvironment();
    }
    return new StandardEnvironment();
}
```

通过代码可以看到根据不同的应用类型初始化不同的系统环境实例。

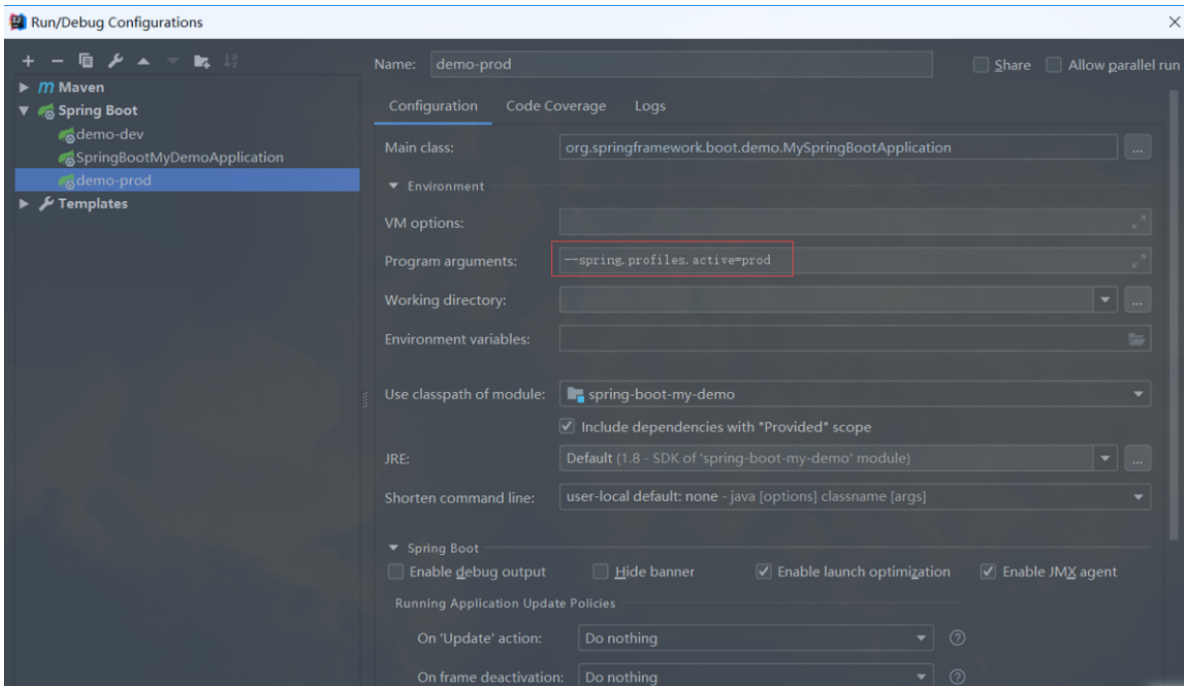
2）configureEnvironment(environment, applicationArguments.getSourceArgs());

```
protected void configureEnvironment(ConfigurableEnvironment environment,
                                    String[] args) {
    // 将main 函数的args封装成 SimpleCommandLinePropertySource 加入环境中。
    configurePropertySources(environment, args);
    // 激活相应的配置文件
    configureProfiles(environment, args);
}
```

在执行完方法中的两行代码后，debug的截图如下



如下图所示，我在spring的启动参数中指定了参数：--spring.profiles.active=prod（关于这个参数的用法，点我，其实就是启动多个实例用的）



3）listeners.environmentPrepared(environment);

进入到方法一路跟下去就到了SimpleApplicationEventMulticaster类的multicastEvent()方法。

```
ationRunListeners.java × | SimpleApplicationEventMulticaster.java × | SpringApplicationRunListener.java × | StandardServletEnvironm
128        }
129
130        @Override
131        public void multicastEvent(final ApplicationEvent event, @Nullable ResolvableType eventType) {  event:
132            ResolvableType type = (eventType != null ? eventType : resolveDefaultEventType(event));  type: "or
133            for (final ApplicationListener<?> listener : getApplicationListeners(event, type)) {  event: "org.
134                Executor executor = getTaskExecutor();
135                if (executor != null) {
136                    executor.execute(() -> invokeListener(listener, event));
137                }
138                else {
139                    invokeListener(listener, event);
140                }
141            }
142        }
```

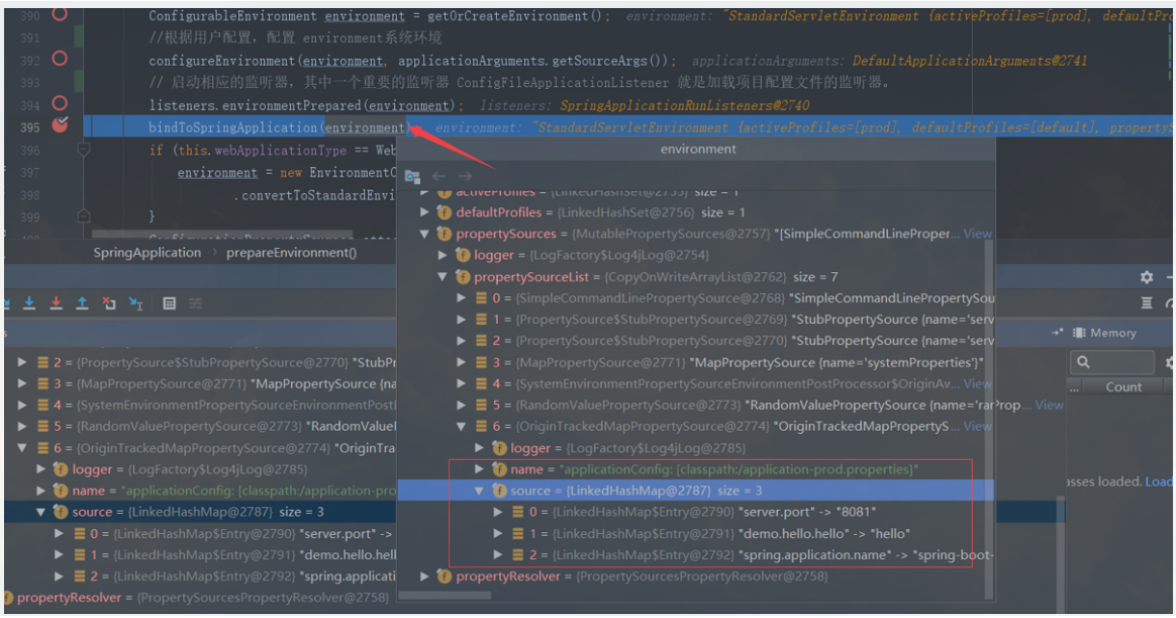
```
130        @Override
131        public void multicastEvent(final ApplicationEvent event, @Nullable ResolvableType eventType) {  event: "org.springframework.boot.
132            ResolvableType type = (eventType != null ? eventType : resolveDefaultEventType(event));  type: "org.springframework.boot.cont
133            for (final ApplicationListener<?> listener : getApplicationListeners(event, type)) {  event: "org.springframework.boot.contex
134                Executor executor = getTaskExecutor();
135
136  Evaluate
137  Expression:                                                                                                  Java
138  getApplicationListeners(event, type)
139                                                                                  Use Ctrl+Shift+Enter to add to Watches
140  Result:
141  ▼ result = {ArrayList@2319} size = 7
142      ▶ 0 = {ConfigFileApplicationListener@2321}
143      ▶ 1 = {AnsiOutputApplicationListener@2322}
144      ▶ 2 = {LoggingApplicationListener@2323}
         ▶ 3 = {ClasspathLoggingApplicationListener@2324}
         ▶ 4 = {BackgroundPreinitializer@2325}
         ▶ 5 = {DelegatingApplicationListener@2326}
         ▶ 6 = {FileEncodingApplicationListener@2327}
```

查看getApplicationListeners(event, type)执行结果，发现一个重要的监听器ConfigFileApplicationListener。

```
<li>classpath:</li>
<li>file:./</li>
<li>classpath:config/</li>
<li>file:./config/:</li>
```

这个监听器默认的从上图所示的几个位置加载配置文件，并将其加入 上下文的 environment变量中。

debug跳过 listeners.environmentPrepared(environment); 这一行，查看environment属性，果真如上面所说的，配置文件的配置信息已经添加上来了。



3、初始化应用上下文

应用上下文可以理解成IoC容器的高级表现形式，应用上下文确实是在IoC容器的基础上丰富了一些高级功能。

主要的工作：

- **根据不同的应用类型，获取不同的应用上下文实例**

  比如 web 应用对应的就是 AnnotationConfigServletWebServerApplicationContext 的上下文

- **创建 IOC 容器 BeanFactory**

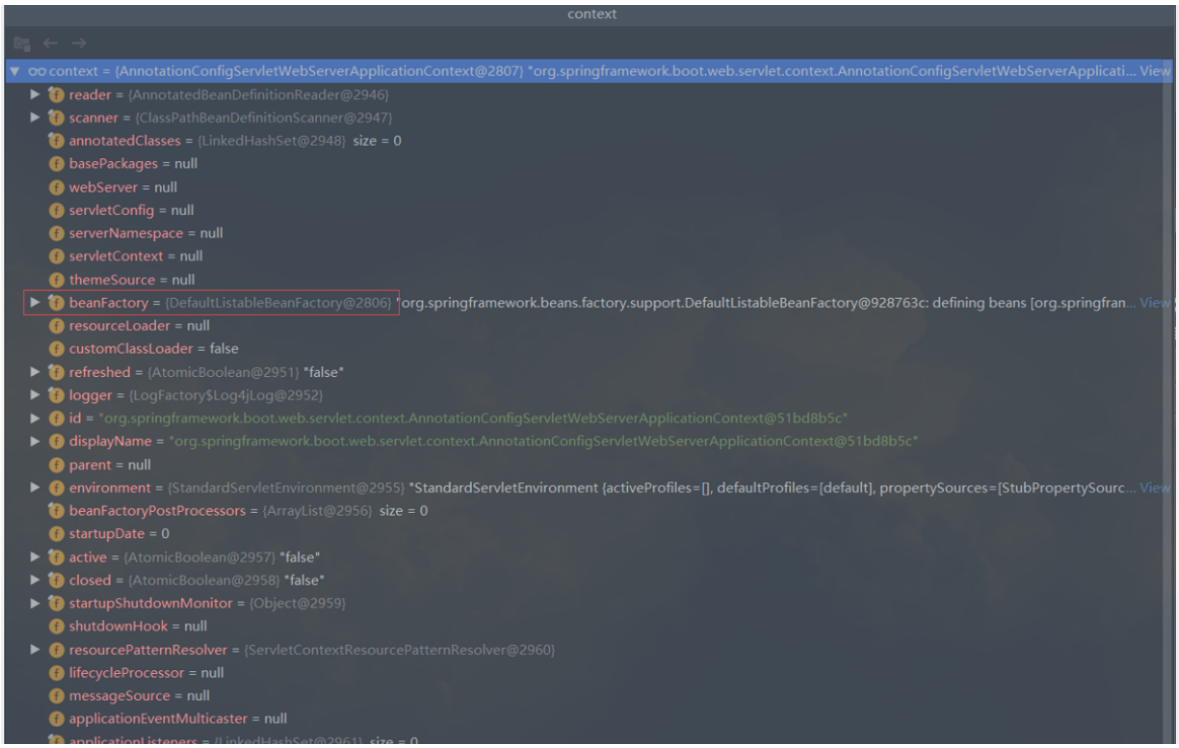
  创建 IOC 容器，然后将容器赋值给 上下文 Context 的 beanFactory 属性，也就是说让上下文持有容器

1）根据应用类型获取对应的上下文实例

```
public enum WebApplicationType {
    /**
     * 应用程序不是web应用，也不应该用web服务器去启动
     */
    NONE,
    /**
     * 应用程序应作为基于servlet的web应用程序运行，并应启动嵌入式servlet web（tomcat）服务器。
     */
    SERVLET,
    /**
     * 应用程序应作为 reactive web应用程序运行，并应启动嵌入式 reactive web服务器。
     */
    REACTIVE
}
```

```
protected ConfigurableApplicationContext createApplicationContext() {
    Class<?> contextClass = this.applicationContextClass;
    if (contextClass == null) {
        try {
            switch (this.webApplicationType) {
                case SERVLET:
                    contextClass = Class.forName(DEFAULT_WEB_CONTEXT_CLASS);
                    break;
                case REACTIVE:
                    contextClass = Class.forName(DEFAULT_REACTIVE_WEB_CONTEXT_CLASS);
                    break;
                default:
                    contextClass = Class.forName(DEFAULT_CONTEXT_CLASS);
            }
        } catch (ClassNotFoundException ex) {
            throw new IllegalStateException(
                    "Unable create a default ApplicationContext, "
                            + "please specify an ApplicationContextClass",
                    ex);
        }
    }
    return (ConfigurableApplicationContext) BeanUtils.instantiateClass(contextClass);
}
```

2）debug跳过createApplicationContext()方法



4、应用上下文的准备工作

prepareContext()

```
private void prepareContext(ConfigurableApplicationContext context,
                            ConfigurableEnvironment environment,
SpringApplicationRunListeners listeners,
                            ApplicationArguments applicationArguments, Banner
printedBanner) {
    //设置容器环境
    context.setEnvironment(environment);
```

```
    //执行容器后置处理
    postProcessApplicationContext(context);
    //执行容器中的 ApplicationContextInitializer 包括spring.factories和通过三种方式自定义的
    applyInitializers(context);
    //向各个监听器发送容器已经准备好的事件
    listeners.contextPrepared(context);
    if (this.logStartupInfo) {
        logStartupInfo(context.getParent() == null);
        logStartupProfileInfo(context);
    }

    // Add boot specific singleton beans
    //将main函数中的args参数封装成单例Bean，注册进容器
    context.getBeanFactory().registerSingleton("springApplicationArguments",
            applicationArguments);
    //将 printedBanner 也封装成单例，注册进容器
    if (printedBanner != null) {
        context.getBeanFactory().registerSingleton("springBootBanner", printedBanner);
    }

    // Load the sources
    Set<Object> sources = getAllSources();
    Assert.notEmpty(sources, "Sources must not be empty");
    //加载我们的启动类，将启动类注入容器
    load(context, sources.toArray(new Object[0]));
    //发布容器已加载事件
    listeners.contextLoaded(context);
}
```

load(ApplicationContext context, Object[] sources)

```
protected void load(ApplicationContext context, Object[] sources) {
    if (logger.isDebugEnabled()) {
        logger.debug(
                "Loading source " + StringUtils.arrayToCommaDelimitedString(sources));
    }
    //创建 BeanDefinitionLoader
    BeanDefinitionLoader loader = createBeanDefinitionLoader(
            getBeanDefinitionRegistry(context), sources);
    if (this.beanNameGenerator != null) {
        loader.setBeanNameGenerator(this.beanNameGenerator);
    }
    if (this.resourceLoader != null) {
        loader.setResourceLoader(this.resourceLoader);
    }
    if (this.environment != null) {
        loader.setEnvironment(this.environment);
    }
    loader.load();
}
```

1）getBeanDefinitionRegistry()

```
private BeanDefinitionRegistry getBeanDefinitionRegistry(ApplicationContext context) {
    if (context instanceof BeanDefinitionRegistry) {
        return (BeanDefinitionRegistry) context;
    }
    ...
}
```

将我们前文创建的上下文强转为BeanDefinitionRegistry。

2）createBeanDefinitionLoader()

```
BeanDefinitionLoader(BeanDefinitionRegistry registry, Object... sources) {
    Assert.notNull(registry, "Registry must not be null");
    Assert.notEmpty(sources, "Sources must not be empty");
    this.sources = sources;
    //注解形式的Bean定义读取器 比如：@Configuration @Bean @Component @Controller @Service等等
    this.annotatedReader = new AnnotatedBeanDefinitionReader(registry);
    //XML形式的Bean定义读取器
    this.xmlReader = new XmlBeanDefinitionReader(registry);
    if (isGroovyPresent()) {
        this.groovyReader = new GroovyBeanDefinitionReader(registry);
    }
    //类路径扫描器
    this.scanner = new ClassPathBeanDefinitionScanner(registry);
    //扫描器添加排除过滤器
    this.scanner.addExcludeFilter(new ClassExcludeFilter(sources));
}
```

创建bean的加载器，为 BeanDefinitionLoader 配置这三个属性，用来将解析 bean 的定义并注册到容器的注册表中。**主要工作为：**

- **配置一个注解形式的bean的读取器**
- **配置一个xml形式的bean的读取器**
- **配置类路径扫描器**

3）loader.load() - doRegisterBean()

将主类封装成AnnotatedGenericBeanDefinition，注册进beanDefinitionMap

```
<T> void doRegisterBean(Class<T> annotatedClass, @Nullable Supplier<T> instanceSupplier, @Nullable String name,
        @Nullable Class<? extends Annotation>[] qualifiers, BeanDefinitionCustomizer... definitionCustomizers) {

    //将指定的类 封装为AnnotatedGenericBeanDefinition
    AnnotatedGenericBeanDefinition abd = new AnnotatedGenericBeanDefinition(annotatedClass);
    if (this.conditionEvaluator.shouldSkip(abd.getMetadata())) {
        return;
    }

    abd.setInstanceSupplier(instanceSupplier);
    // 获取该类的 scope 属性
    ScopeMetadata scopeMetadata = this.scopeMetadataResolver.resolveScopeMetadata(abd);
    abd.setScope(scopeMetadata.getScopeName());
    String beanName = (name != null ? name : this.beanNameGenerator.generateBeanName(abd, this.registry));

    AnnotationConfigUtils.processCommonDefinitionAnnotations(abd);
    if (qualifiers != null) {
        for (Class<? extends Annotation> qualifier : qualifiers) {
            if (Primary.class == qualifier) {
                abd.setPrimary(true);
            }
            else if (Lazy.class == qualifier) {
                abd.setLazyInit(true);
            }
            else {
                abd.addQualifier(new AutowireCandidateQualifier(qualifier));
            }
        }
    }
    for (BeanDefinitionCustomizer customizer : definitionCustomizers) {
        customizer.customize(abd);
    }

    BeanDefinitionHolder definitionHolder = new BeanDefinitionHolder(abd, beanName);
    definitionHolder = AnnotationConfigUtils.applyScopedProxyMode(scopeMetadata, definitionHolder, this.registry);
    // 将该BeanDefinition注册到IoC容器的beanDefinitionMap中
    BeanDefinitionReaderUtils.registerBeanDefinition(definitionHolder, this.registry);
}
```

5、刷新应用上下文

refresh()

```java
public void refresh() throws BeansException, IllegalStateException {
    synchronized (this.startupShutdownMonitor) {
        // Prepare this context for refreshing.
        //刷新上下文环境
         prepareRefresh();
        // Tell the subclass to refresh the internal bean factory.
        //这里是在子类中启动 refreshBeanFactory() 的地方
        ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();
        // Prepare the bean factory for use in this context.
        //准备bean工厂，以便在此上下文中使用
         prepareBeanFactory(beanFactory);
        try {
            // Allows post-processing of the bean factory in context subclasses.
            //设置 beanFactory 的后置处理
             postProcessBeanFactory(beanFactory);
            // Invoke factory processors registered as beans in the context.
            //调用 BeanFactory 的后处理器，这些处理器是在Bean 定义中向容器注册的
             invokeBeanFactoryPostProcessors(beanFactory);
            // Register bean processors that intercept bean creation.
            //注册Bean的后处理器，在Bean创建过程中调用
             registerBeanPostProcessors(beanFactory);
            // Initialize message source for this context.
            //对上下文中的消息源进行初始化
             initMessageSource();
            // Initialize event multicaster for this context.
            //初始化上下文中的事件机制
             initApplicationEventMulticaster();
            // Initialize other special beans in specific context subclasses.
            //初始化其他特殊的Bean
             onRefresh();
            // Check for listener beans and register them.
            //检查监听Bean并且将这些监听Bean向容器注册
             registerListeners();
            // Instantiate all remaining (non-lazy-init) singletons.
            //实例化所有的（non-lazy-init）单件
             finishBeanFactoryInitialization(beanFactory);
            // Last step: publish corresponding event.
            //发布容器事件，结束Refresh过程
             finishRefresh();
```

1）obtainFreshBeanFactory()

关于obtainFreshBeanFactory()方法，其实就是拿到我们之前创建的beanFactory。

2）prepareBeanFactory(beanFactory)

容器的准备工作，主要工作包括：

- **配置类加载器**

  用来加载来到 JVM 中

- **配置属性编辑器**

  用来更改 bean 的属性

- **添加 bean 的后置处理器 beanPostProcessor**

- **将内部 bean 注册为单例**

  比如环境变量（environment）、系统配置（systemProperties）、系统环境（systemEnvironment）

3）postProcessBeanFactory(beanFactory)

postProcessBeanFactory()方法向上下文中添加了一系列的Bean的后置处理器。

后置处理器工作的时机是在所有的beanDenifition加载完成之后，bean实例化之前执行。

简单来说Bean的后置处理器可以修改BeanDefinition的属性信息。

4）invokeBeanFactoryPostProcessors(beanFactory)

完成了IoC容器初始化过程的三个步骤，主要工作为：

- **BeanDefinition 的 Resource 定位**
- **BeanDefinition的载入**
- **将 BeanDefinition 注册到 IOC 容器的注册表中**

第一步：Resource定位

定位主要三种路径：

- **主类包路径**
- **SPI扩展机制定义的路径**
- **@Import注解的路径**

在SpringBoot中，包的扫描是从主类所在的包开始扫描的。在 prepareContext() 上下文准备的过程中，首先会先将主类解析成BeanDefinition，然后在 refresh() 方法也就是容器刷新时，调用 invokeBeanFactoryPostProcessors() 方法中解析主类的BeanDefinition 获取主类所在的包的路径，这样就完成了主类包路径定位的过程。

其次SpringBoot的各种starter是通过SPI扩展机制实现的自动装配，基于SPI机制从指定路径下定位到实现类的路径。

@EnableXXX注解，很大一部分都是对@Import的二次封装，对该注解的解析，底层其实就是对@Import注解指定的配置类的定位。

第二步：BeanDefinition的载入

对资源类的路径定位后，SpiringBoot会有一个资源解析器，把对应路径下的所有 .class 文件都加载进来，然后遍历判断是不是有@Component注解，如果有，就是我们要加载的 BeanDefinition。

第三步：注册BeanDefinition

得到 BeanDefinition 后，就将这些 Definition 对象注册到容器中的注册表中 ，IoC容器通过一个 HashMap 来持有这些BeanDefinition。

6、自动装配

SpringBootApplication

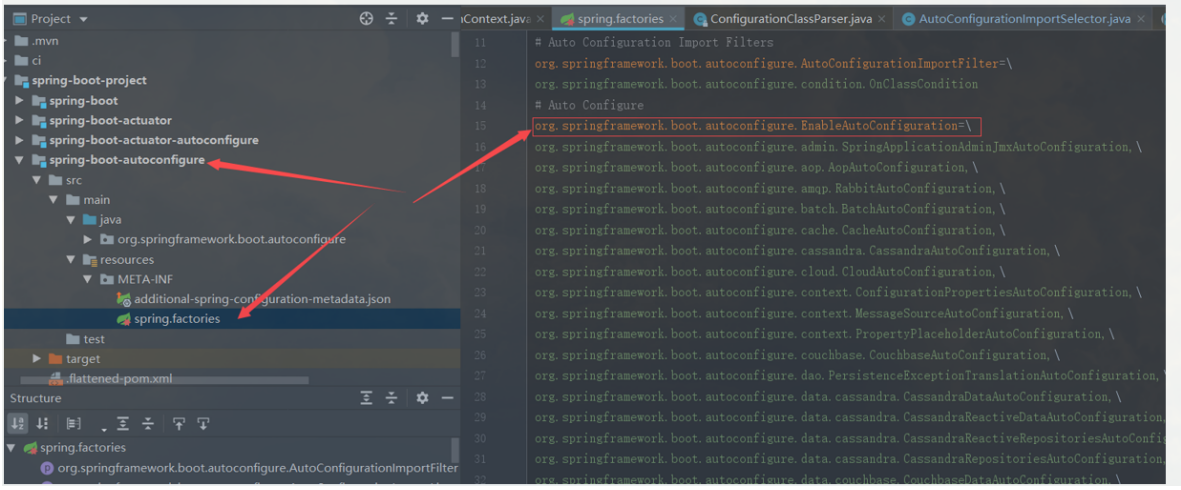
```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
        @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
        @Filter(type = FilterType.CUSTOM, classes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

EnableAutoConfiguration

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

自动装配选择器 AutoConfigurationImportSelector

```
// AutoConfigurationImportSelector类
//自动装配
@Override
public String[] selectImports(AnnotationMetadata annotationMetadata) {
    if (!isEnabled(annotationMetadata)) {
        return NO_IMPORTS;
    }
    AutoConfigurationMetadata autoConfigurationMetadata = AutoConfigurationMetadataLoader
            .loadMetadata(this.beanClassLoader);
    AnnotationAttributes attributes = getAttributes(annotationMetadata);
    //获取所有的自动配置类（META-INF/spring.factories中配置的key为org.springframework.boot.autoconfigure.EnableAutoConfigurati
    List<String> configurations = getCandidateConfigurations(annotationMetadata,
            attributes);
    configurations = removeDuplicates(configurations);
    //需要排除的自动装配类（springboot的主类上 @SpringBootApplication(exclude = {com.demo.starter.config.DemoConfig.class})指定
    Set<String> exclusions = getExclusions(annotationMetadata, attributes);
    checkExcludedClasses(configurations, exclusions);
    //将需要排除的类从 configurations remove掉
    configurations.removeAll(exclusions);
    configurations = filter(configurations, autoConfigurationMetadata);
    fireAutoConfigurationImportEvents(configurations, exclusions);
    return StringUtils.toStringArray(configurations);
}
```



其实还是通过 SPI 机制，从配置文件中，取出所有的自动装配的扩展实例。具体就是扫描项目的依赖的指定文件 spring.factories，通过传入的接口类全路径名为 key ，取出所有的扩展实现的类路径，并实例化。

同时还可以通过 @Conditional 注解排除掉一些不希望自动装配的扩展。

这里得到的扩展实例，其实相当于我们用 @Configuration 注解定义的配置类，也相当于我们在 xml 文件中定义的配置类一样，只不过 spring boot 遵从 “约定大于配置” 的规约，自动帮我们生成了这样的默认的配置文件，使得我们不用再手动配置。

7、实例化 Bean 与依赖注入

支持按名或按类型来获取 bean

最终都进入到了doGetBean()方法

1）Object sharedInstance = getSingleton(beanName);

首先会去找在容器中是不是已经存在该单例。在DefaultSingletonBeanRegistry类中的singletonObjects属性就是存singleton bean的地方。

如果getSingleton()为 null则会在当前的BeanFactory中获取BeanDefinition，首先判断是不是singleton Bean，如果是的话，开始执行创建createBean(beanName, mbd, args)，如果是原型(Prototype)Bean，每次执行getBean()都会创建一个实例。

2）getBean()

根据Bean的名字或类型取得BeanDefinition，获取当前Bean的所有依赖Bean，这里会触发getBean的递归调用。知道取到一个没有任何依赖的Bean为止。

3）createBean

通过createBean方法创建 bean，这里有单例（singleton）和原型（prototype）两种 bean 类型的实例。

主要的工作：

- Bean实例的创建
- 为Bean实例设置属性(属性注入，其实就是依赖注入真正发生的地方)
- 调用Bean的初始化方法

3.1）Bean 实例的创建

Bean对象会根据BeanDefinition定义的要求生成。实例化 bean 之前可以在 bean 的一个后处理器的前置扩展点（beforeInstance）之中对 bean 的定义 BeanDefinition中的属性做些前置处理，比如修改属性什么的。

简单地说，就是给子类一个修改BeanDefinition的机会，也就是说当程序经过这个方法后，bean可能已经不是我们认为的bean了，而是或许成为了一个经过处理的代理bean，可能是通过cglib生成的，也可能是通过其它技术生成的。

然后进行 bean 的初始化。

实例前初始化后之后在后置扩展点（afterinstance）之中应用子类的后置扩展点中的方法，因为普通的单例 bean 只会创建一次，所有如果后置扩展点有自定义的逻辑的话，就需要在这里，也就是 bean 实例化之后来触发。

doCreateBean()

创建实例

instanceWrapper = createBeanInstance(beanName, mbd, args);

把ObjectFactory放进singletonFactories中，这里在其他bean在创建的时候会先去singletonFactories中查找有没有beanName到ObjectFactory的映射，如果有ObjectFactory就调用它的getObject方法获取实例。

创建实例有两种方式，一种是如果 beanFactory 修改了 bean 的定义，就需要用 CGLIB 动态代理来生成实例，否则就用 JVM 反射生成。

3.2）Bean的属性设置（依赖注入）

populateBean(beanName, mbd, instanceWrapper);

Bean依赖注入发生的地方，对bean进行属性填充，如果存在依赖于其他的bean的属性，则会递归的调用初始化依赖的bean。

Bean的后置处理器，在Bean注入属性前改变BeanDefinition的信息

依赖注入：

AutowiredAnnotationBeanPostProcessor类实现了postProcessPropertyValues()方法。findAutowiringMetadata(beanName, bean.getClass(), pvs);方法会寻找在当前类中的被@Autowire，@Resource，@Value，@Inject等具备注入功能的注解的属性。

解析注解所定义的类。

注入的时候通过BeanFactory 解决依赖关系，比如在webController中注入了webService，这个会去BeanFactory中去获取webService，也就是getBean()的逻辑。如果存在直接返回，不存在再执行createBean()逻辑。如果在webService中依然依赖，依然会去递归。

反射注入。

3.3）Bean的初始化

exposedObject = initializeBean(beanName, exposedObject, mbd);

调用初始化方法，比如init-method方法指定的方法

initializeBean()：调用Bean的初始化方法

设置Bean的初始化方法有两种方法，一种是在xml或者@Bean指定init-method方法。另一种是让bean实现InitializingBean接口重写afterPropertiesSet()方法。

init-method

在调用Bean的初始化方法之前，调用一系列的aware接口实现，把相关的BeanName，BeanClassLoader，以及BeanFactory注入到Bean中去。接着会执行invokeInitMethods()方法。

invokeAwareMethods(beanName, bean);

初始化之前调用BeanPostProcessor。

wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);

调用指定的init-method方法

invokeInitMethods(beanName, wrappedBean, mbd);

初始化之后调用BeanPostProcessor

wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);

实现InitializingBean接口重写afterPropertiesSet()方法

((InitializingBean) bean).afterPropertiesSet();

执行afterPropertiesSet()方法进行初始化

8、总结

1、首先是 SpringApplication 构造函数，主要做了三件事：

1）根据已存在的类来推断应用类型。主要分为 WEB 应用和 非 WEB 应用

2）获得所有的初始化器。基于SPI机制，根据传入的接口类，去指定路径下找出所有的扩展实现。目的是在容器刷新之前，可以做一些扩展工作，比如修改bean的定义或者根据系统环境参数激活指定的 profile 环境。

3）加载所有的事件监听器。同样基于SPI机制找到所有的基于观察者模式的监听器，这可以让开发者实现对 Spring 容器生命周期事件的监听。

2、然后进入 SpringApplication run 方法

1）获取事件监听器并启动。就是在构造函数中创建的监听器，然后启动。

2）构造应用上下文环境

目的是让容器持有系统和项目的配置信息。

上下文环境，就是许多的配置信息，主要分为系统环境和项目环境。系统环境就是我们用户在 main 函数入口的一些配置，比如可以通过 IDEA 工具可视化地配置 active 属性是开发环境还是生产环境等，也可以配置项目使用的JDK版本比如1.8。

项目环境就是我们在项目的 resources 目录下的 properties 配置文件等。然后解析出上述两种环境的 key-value，通过事件监听机制发布环境已准备好的事件，然后有一个内置的系统监听器 ConfigFileApplicationListener
，就会专门地将这些已解析好的环境变量 key-value 加载到容器的 Environment 属性中，也就是我们平常根据参数key获取对应value时用到的一个对象。

3）初始化应用上下文

主要做两件事情，一个是根据不同的应用类型，构建相应的上下文类型。一个是创建 IOC 容器 BeanFactory。
这个应用类型在上下文的构造函数中就已经通过已存在的类来进行推断了。

如果是 WEB 应用，对应的上下文类型就是 AnnotationConfigServletWebServerApplicationContext。应用的上下文，其实是在 IOC 容器上再丰富了一些高级功能，可以把上下文看作是 IOC 容器的高级封装，它持有 IOC 容器，IOC容器专注于 Bean 的管理，比如注册 bean 的定义，提供 bean 的存储和获取功能等，

而上下文则在此基础中，通过提供多种丰富的后置处理器来作为程序应用的扩展点，可以更加丰富原有的bean
的功能，比如容器的后置处理器，针对的是容器，从容器中获取 bean 的注册表，然后修改注册表的信息来修改 bean 的定义来动态生成一个新的 bean，又比如可以在 bean 的后置处理器中，对已经实例化出来的bean进行一些扩展的操作等。比如实现一些 AOP bean ，代理 bean 等。

上下文还可以管理 bean 的生命周期，在 bean 的不同周期阶段发布不同的事件给感兴趣的监听者。但当然添加各种后置处理器和事件监听器的具体步骤，是在容器刷新时发生的。

4）准备应用上下文

就是在上下文刷新前为上下文环境再做一些预处理的准备工作，上下文刷新主要是依靠 IOC 容器进行 bean 的实例化和依赖注入。

而上下文的准备就是为容器刷新做准备。主要做了三件事情：

一个是调用初始化器的初始化方法，刚才说到在上下文的构造方法中配置了初始化器，作用就是在容器刷新
前做一些扩展的预处理，所以具体的预处理逻辑就在这里触发。

第二个是创建 bean 的加载器，在实例化 bean 之前，需要得到 bean 的定义。spring 是通过扫描包的形式
或者解析 xml 的方式来得到 bean 的定义，然后把 bean 的定义注册到容器的注册表中。所以这里需要配置一个 bean 的加载器，加载器还需要配置三个工具，就是解析注解的注解解析器，解析xml 配置的 xml 解析器，然后再配置一个扫描器。

第三个是把主类解析成 bean 定义 beanDefiniton 然后注册到注册表。目的是我们知道 spring boot 包扫描
是从主类所在包开始往下逐层扫描的，所以这里把主类定义注册起来，目的就是后面进行扫描时，可以通过
主类获取到它所在的包的全路径。

5）刷新应用上下文，核心是 bean 的实例化和依赖注入。

a. 容器的准备

刚才说的是上下文的准备工作，现在是容器的准备工作，区别就是上下文持有容器，针对的是容器的粒度，主要就是让容器拥有扫描并注册所有 bean 的能力，而容器是针对 bean 的粒度，所以容器的准备工作，也是为了支持对 bean 的操作而进行的准备工作。**主要也是几件事情。**

配置类加载器以支持将类加载到VM、配置属性编辑器以支持在容器后置处理器中扩展性地修改 bean 的定义、
添加 bean 的后置处理器、将内部 bean 注册为单例，比如环境变量（environment）、系统配置（systemProperties）、系统环境 （systemEnvironment）等。

b. 执行容器后置处理器的处理逻辑

设置bean的后置处理器，在实例化之后执行处理逻辑

c. 完成IoC容器初始化

主要为BeanDefinition 的 Resource 定位、BeanDefinition的载入、将 BeanDefinition 注册到 IOC 容器的注册表中。这个阶段也完成了 Spring Boot 的自动装配，通过@SpringBootApplication，包含了@EnableAutoConfiguration，包含了@Import，这个import注解声明了一个自动装配器接口，当扫描到import注解时会根据注解的值，基于SPI机制得到所有自动装配的扩展实例，这些实例相当于我们用 @Configuration 注解定义的配置类，也相当于我们在 xml 文件中定义的配置类一样，只不过 spring boot 遵从 “约定大于配置” 的规约，自动帮我们生成了这样的默认的配置文件，使得我们不用再手动配置。

d. 初始化信息源以支持国际化的功能

e. 注册bean生命周期的监听器，这些监听器可以看作是内部bean，也需要注册到容器的注册表中

f. 进行bean的实例化
首先会去找在容器中是不是已经存在该单例，如果没有则获取 bean 的定义然后进行实例化

- **创建实例**
  创建实例前后都有扩展点，这里先执行前置扩展点的逻辑，比如可以在这里改变bean的定义生成新的bean，或者一个代理bean。然后开始创建，创建实例有两种方式，一种是如果 beanFactory 修改了 bean 的定义，就需要用 CGLIB 动态代理来生成实例，否则就用 JVM 反射生成。
- **属性依赖注入**
  然后进行依赖注入，对bean进行属性填充，如果存在依赖于其他的bean的属性，则会递归的调用初始化依赖的bean。主要是解析@Autowire，@Resource，@Value，@Inject等注解的类，基于反射注入。
- **初始化**
  一种是在xml或者@Bean指定init-method方法。另一种是让bean实现InitializingBean接口重写afterPropertiesSet()方法。

主要流程

SpringApplication 的构造函数
1）根据已存在的类来推断应用的类型
2）获得所有的初始化器
3）加载所有的事件监听器

SpringApplication run 方法
1）构造应用上下文，让容器持有系统和项目的配置
2）初始化应用上下文，根据不同类型构建不同上下文，同时创建IOC容器
3）准备应用上下文，执行前置初始化方法，配置 bean 加载器，让容器拥有扫描并注册所有 bean 的能力
4）刷新应用上下文，准备容器配置类加载器和属性编辑器，执行容器后置处理方法，完成IOC容器的初始化，初始化信息源和注册监听器
5）进行 bean 的实例化，创建实例，进行属性依赖注入，实例化后进行初始化

小结

Spring Boot 的启动流程，首先是 Spring 应用上下文的构造函数，然后是执行上下文的 run 函数。

【构造函数】

首先是 Spring 应用上下文的构造函数，主要是配置一些全局的组件，具体通过三件事情来完成组件的配置。
一是根据已存在的类的类型来推断应用类型，二是获取所有初始化器，三是获取所有的监听器。

第一步**推断应用类型**，可以分为 WEB 应用和非 WEB 应用，区分不同的应用类型，是为了后面可以生成对应的应用上下文，应用上下文可以看作是对 IOC 容器的封装，是对 IOC 容器的丰富和扩展，比如可以设置很多的后置处理器来对 bean 进行扩展，可以在扩展中修改 bean 的定义，或者通过动态代理的方式来增加 bean 的功能等。

第二件事情是**获取初始化器**，获取的过程是基于 SPI 机制，通过传入一个顶级的初始化器的接口为 SPI 的 key，
然后在指定的路径下 spring.factories 得到对应的value，得到的 value 就是不同的初始化器的扩展实例。初始化器的功能主要是在后面在容器的刷新之前，做一些定制化的操作，比如可以通过容器的 environment 属性即系统配置的active属性来激活相应的 profile 环境，比如生产环境就是prod开发环境是dev这些。

第三件事情就是**获取监听器**，也是基于 SPI 机制获取各种监听器的扩展，不同的扩展对应处理不同的组件的生命周期的事件。一方面，无论是 spring 应用上下文还是 IOC 容器的创建和初始化过程，都伴随着许多组件的创建，而一些组件的创建时机需要依赖于其他组件的完成后，才可以进行创建，这时就需要有一套事件发布机制，所以 Spring 的监听注册，这是基于观察者模式构建的事件发布机制，以此来协调上下文和 IOC 容器创建的各个阶段的不同组件的创建和依赖等。另一方面，spring 支持在不同的阶段提供一些扩展点来给开发者丰富 bean 的功能，所以基于事件发布机制和监听就能够实现开发者监听自己感兴趣的事件，才能在合适的时机进行相应的扩展。

刚才解释了 spring 上下文的构造，主要是配置了几个全局的组件，为后面 IOC 容器的创建和初始化、以及容器功能的增强和扩展准备了必要的工具。

> **全局组件：推断应用类型（丰富扩展、后置扩展）、获取初始化器（SPI、容器刷新）、监听器（组件依赖、监听扩展）**

【run 函数】

然后进入 Spring 上下文的 run 方法，围绕着应用的上下文，主要可以划分为三个阶段。分别是应用上下文环境的创建、应用上下文的初始化（包含准备）、应用上下文的刷新这三步。

【第一阶段】

首先是第一阶段，创建应用上下文环境，主要分为创建系统环境和项目环境，目的是为了让上下文拥有系统和项目的配置信息。系统环境就是我们用户在 main 函数入口的一些配置，比如可以通过 IDEA 工具可视化地配置 active 属性是开发环境还是生产环境等，也可以配置项目使用的JDK版本比如1.8。项目环境就是我们在项目的 resources 目录下的 properties 配置文件等，将这些配置文件的 key-value 键值对解析并加载到上下文的 Environment 属性中，也就是我们平常根据参数key获取对应value时用到的一个对象。所以这一步主要是配置信息的解析和加载。

> **系统环境（active属性、jdk版本），项目环境（resources.properties、Environment属性 ）**

【第二阶段】

然后是第二阶段，应用上下文的初始化，初始化应用上下文，主要目的是使得 IOC 容器拥有扫描并注册所有 bean 的能力。围绕这个目标的达成，系统构建了核心组件和主要组件。核心组件指的是具体的应用上下文以及IOC 容器，主要组件指的是 bean 加载器。在上下文构造函数中已经推断出上下文的类型，这里只要根据类型创建对应的上下文即可，比如 WEB 应用对应一种上下文。（刚才提到，上下文的主要作用就是**丰富 bean 的功能**，它作为 IOC 容器的封装，通过提供不同的后置处理器作为扩展，可以让开发者通过扩展来加工 bean，比如可以加工成代理 bean，或者 AOP bean等。）

然后是**创建一个 IOC 容器**，这个容器负责后面 bean 的实例化以及依赖注入。接着是**创建一个 bean 的加载器**，因为我们要实例化 bean，首先需要知道 bean 在哪里，然后才能在具体的位置得到 bean 的相关定义，然后再把 bean 的定义扫描注册到 IOC 容器的注册表中。这就是 bean 加载器的由来，所以它包含**三个组件**，一个是扫描器，它负责扫描相关的 bean 的注解的位置，比如@Controller注解、@Service注解、@Compont注解等，扫描到这些注解所在的类的路径后，还需要有解析器，一个是注解形式的解析器，一个是xml形式的解析器，分别对应我们定义bean的两种方式，用解析器将 bean 的定义解析成beanDefinition 对象后，注册到 IOC 容器的注册表中。

这就是核心组件和主要组件的构造，主要目的是使得 IOC 容器拥有扫描并注册所有 bean 的能力。除此以外，这个阶段还做了两件事情，一件是**执行初始化器的扩展实例的初始化逻辑**，这些初始化扩展就是在一开始 spring 应用上下文构造器中得到的，这里就是为了在后面 bean 实例化之前再做一些预处理的扩展逻辑。第二件事是**把主类也注册到 IOC 容器**中。因为我们知道 spring boot 的组件扫描，是从主类所在的包下开始的，所以这里把主类注册到 IOC 容器中，是为了后面开始扫描各种组件时，可以通过 IOC 容器获取到主类所在的包路径，然后从该路径开始描。

所以到这里第二阶段，也就是上下文的初始化工作就完成了，**前面两个阶段，主要完成了三件事情，一个是加载到了系统和项目的配置信息，第二个是使得 IOC 容器拥有扫描并注册所有 bean 的能力。第三个是在 IOC 容器创建的前后，通过后置处理器完成了一些预处理和扩展逻辑等。**

> **IOC 拥有容器扫描并注册 bean 的能力：**
>
> **核心组件（应用上下文、IOC 容器）、主要组件 bean 加载器（在哪、定义、注册，扫描器、解析器）**
>
> **初始化器方法、主类加载**

【第三阶段】

最后到了第三阶段，就是上下文的刷新。这里说上下文的刷新，其实是比较抽象的，具体来讲，其实到这一步，这个上下文已经通过上面应用的推断得到了一个具体的上下文对象了，而这个上下文对象主要持有了一个 IOC 容器，以及支持基于 IOC 容器的一些丰富的扩展功能。

所以对于上下文的刷新，主要是对于 IOC 容器的刷新，而 IOC 容器此时只是创建出来，还没有进行初始化，所以对于上下文的刷新，主要就是对于 IOC 容器的初始化，而 IOC 容器的初始化，主要就是 bean 的实例化、bean 的属性依赖注入，以及 bean 的初始化等。**所以上下文的刷新主要内容就是 bean 的实例化、依赖注入、初始化，以及围绕 bean 做一些功能性的扩展和增强。**

具体来说，容器要能对 bean 实例化，首先需要为容器配置**类加载器**以支持加载到 JVM 中，并且需要配置**属性编辑器**以支持对属性定义的修改。然后需要找到各类 bean 所在的位置，就需要进行一个资源的定位，通过sanner扫描器扫描主类所在包下的注解得到 bean 所在的位置，这个过程也包含了Spring Boot **自动装配**各类依赖的过程，通过 SPI 机制，基于“约定大于配置” 的规约，Spring 自动帮我们生成了各种默认的配置文件 bean 并注册到 IOC 容器中的注册表中，为后面实例化到做准备，使得我们不用再手动配置。

然后**开始实例化，通过CGLIB 动态代理或者JVM反射**得到 bean 的实例，再判断bean的属性是否有依赖于其他的 bean，有则会**递归的从容器中获取依赖的 bean** ，如果没有则进行创建，这里就是属性的依赖注入。最后是 **bean 的初始化**，有两种方法，一种是我们通过xml或注解的形式指定了初始化的 init-method 方法。另一种是让bean实现InitializingBean接口重写afterPropertiesSet()方法。在实例化bean的前后，还可以在系统提供的 beanPostProcessor 扩展点中进行功能的增强，比如包装一个代理bean，或者注入应用的上下文等。

到这就是 IOC 容器的初始化，除此以外还进行信息源的初始化以支持国际化的功能，以及注册监听器等。

也可以分为 bean 实例化之前和之后。之前就是应用上下文的构建和初始化，使得上下文加载到了系统和项目的配置信息，同时使得 IOC 容器拥有扫描并注册所有 bean 的能力。之后就是开始进入 bean 的实例化、属性依赖注入、初始化的过程。在这一整个阶段，还伴随着各类后置处理器的加入以及组件生命周期事件的发布和监听等，作用都是为了更加丰富和完善 bean 的功能以及协调 bean 的构建过程。

> **Context - > IOC -> Init(实例化、依赖注入、初始化)**
>
> **实例化：类加载器、属性编辑器、自动装配（约定大于规约、自动生成并注册）**
>
> **依赖注入：实例化（动态代理、反射），依赖注入、初始化（xml-init、afterPropertiesSet）**
>
> **其他扩展：初始化信息源、注册监听器**

Q&A

Q1、自动装配

Starter是一组非常方便的依赖集合，应用不需要再单独依赖，只需要依赖Starter，就可以**一站式**解决所需的依赖。自动配置需要依赖大量 **spring java configuration**，这些配置往往都放在spring-boot-autoconfigure模块中，这些配置通过类似Java **SPI**机制将其载入。在类路径上有个spring.factories文件，其中存需要自动配置的Java Configure类，然后通过Spring Factory Loader将其载入，作为spring的configuration class，再**解析注册**相应的Bean，这样就完成了自动配置，从而避免应用开发者手动配置的不便。

Spring boot 的自动配置正是借助了 SPI 机制，是通过一个**自动配置选择器**来实现 SPI 机制的。而这个自动配置选择器就在 Spring boot 应用启动时就会被初始化的。具体的实现就是在应用的主类上有**@Spring BootApplication注解，该注解包含了@EnableAutoConfiguration注解，这个注解又包含了@import注解**，而 import 注解声明了自动配置选择器这个类，当应用启动时，扫描到这个 import 注解时就会生成自动装配选择器，然后传入EnableAutoConfiguration接口类，以该类为Key，去指定路径下找出所有的value，就是自动配置的具体实现。

spring boot中预内置了大量配置类，其实就是将平时传统的spring应用的XML配置或者Java Config配置预先编排好，形成配置类放在spring-boot-autoconfigure项目中，当spring boot启动时就会载入这些预先编排的配置类。

> **SpringBootApplication 包含 EnableAutoConfiguration，包含 import，声明一个自动配置选择器。**

> **应用启动，扫描并生成自动配置选择器，传入 EnableAutoConfiguration 接口类作为 SPI KEY，去指定路径 spring.factories 找到各种配置的实例，这些实例被定义在 stater 的 spring-boot-autoconfigure 模块中， 然后将这些自动配置 bean 定义实例化到 IOC 容器中，后面再通过这些配置 bean 完成了各种项目依赖的导入。**

Q2、核心接口类

1）ApplicationContext 保存了 IOC 容器的应用上下文，是 IOC 容器的一个封装。通过上下文的初始化过程，可以实现对 IOC 容器进行一个丰富和扩展的功能。比如可以通过一些后置扩展器来修改 bean 的定义（属性、是否单例等），或者通过动态代理来增强 bean 的功能等，可以通过其中的 beanfactory 获取到任意到bean。以及其他的丰富功能，比如国际化功能，上下文和 IOC 容器声明周期的事件的发布机制等。

2）BeanFactory 是接口，可以有具体的 IOC 容器的实现，可以抽象认为是 IOC 容器。 IOC 容器包括一张 beanDefinition 的注册信息表，以及一个 bean 的单例对象池等。主要负责 bean 的资源定位、实例化、属性编辑、依赖注入和初始化等。发现扫描并将 bean 的定义注册到注册表中，然后再负责将注册表中的 bean 进行初始化，放到单例对象池中。主要实现了对 bean 的管理。

3）BeanWrapper 是对 bean 的封装，一般是在 IOC 内部使用，供了访问 bean 属性的编辑、访问和相关类型转换等功能，方便ioc容器用统一的方式来访问bean的属性。BeanFactory 通过 BeanWrapper 操作 bean。

4）FactoryBean 是一个接口，是在 IOC 容器的基础上加上了简单工厂模式和装饰模式的一个工厂，主要通过 getObject 方法来生产实现了该接口的 bean。主要是为了降低实例化 bean 的复杂性，因为通过动态代理和反射的方式有时不太灵活比较复杂，用户可以自定义实现 FactoryBean 接口的 bean，再通过 getObject 方法即可获得一个简单配置的 bean 对象，而无需配置很多的复杂内容。

Q3、常用注解

通用注解

@Component注解在类上，表明该类是个组件 bean。

@Bean注解使用在方法上，告诉Spring这个方法将会返回一个Bean对象， 需要把返回的对象注册到Spring的应用上下文中。

设置类注解

@Autowire和、Qualifie（区分不同的实现）

web类注解

@RequestMapping、@GetMapping、@PostMapping等路径匹配注解，以及@PathVariable、@RequestParam 等参数获取注解。

功能类注解

@ComponentScan注解自动扫描、@Transactional事务注解

Constructor(构造方法) -> @Autowired(依赖注入) -> @PostConstruct(注释的方法)

Q4、Sping Context 初始化流程

1）初始化上下文环境

2）创建 IOC 容器即 bean factory， 并进行准备工作，比如设置类加载器和后置处理器，主要有两种

3）执行 IOC 容器的后置处理逻辑

4）实例化 bean，进行属性依赖注入

5）bean 的后置处理器前置方法

6）bean 的初始化

7）bean 的后置处理器后置方法

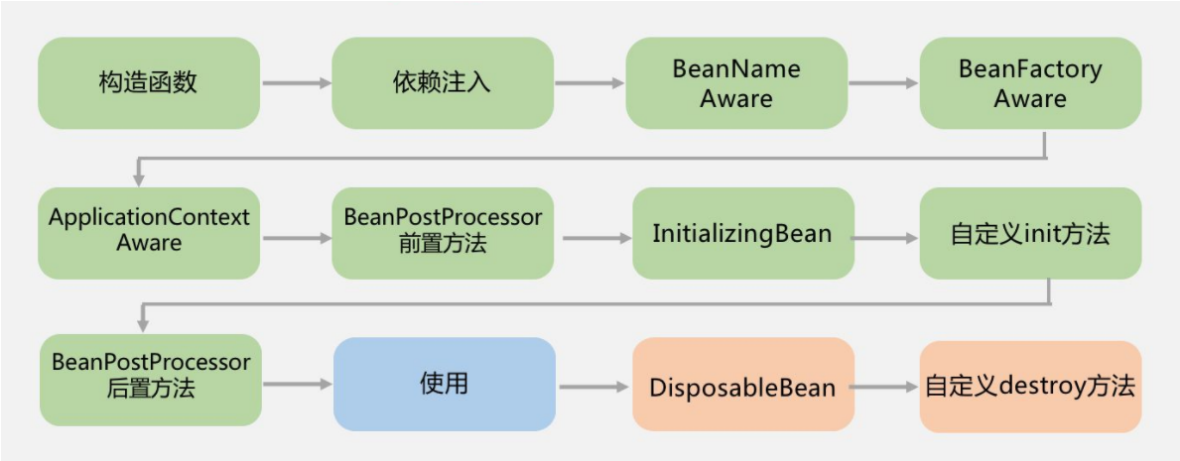
8）初始化信息源，支持消息国际化功能

9）初始化应用事件广播器， 把实现了ApplicationListener的监听器 bean 注册到事件广播器，并对广播器中的早期未广播事件进行通知

10）冻结 bean 定义的修改，完成 IOC 容器的初始化，并发布相关事件

11）重置公共缓存等

Q5、生命周期



第1步：调用bean的构造方法创建bean；

第2步：通过反射调用setter方法进行属性的依赖注入；

--->aware

第3步：如果实现BeanNameAware接口的话，会设置bean的name；

第4步：如果实现了BeanFactoryAware，会把bean factory设置给bean；

第5步：如果实现了ApplicationContextAware，会把上下文设置给bean；

--->初始化方法

第6步：如果实现了BeanPostProcessor接口，则执行前置处理方法；

第7步：实现了InitializingBean接口的话，执行afterPropertiesSet方法；

第8步：执行自定义的init方法；

第9步：执行BeanPostProcessor接口的后置处理方法。

> **构造函数、依赖注入**
>
> **aware（name、beanFactory、applicationContex）**

> **初始化（后置处理器的前置后置方法、自定义方法）**

Q6、Spring 和 Spring Boot 的区别

Spring是一个**低侵入式**的**应用开发框架**，致力于为开发Java应用程序提**供了全面的基础架构支持。**包括统一的 **bean 管理**功能和**依赖自动注入**功能，支持**AOP**切面编程，还有很多**开箱即用**的模块， 比如spring **mvc**，**JDBC**等。

Spring Boot是Spring框架的**扩展**，它的目标是**简化Spring应用的创建、开发与部署。**把许多常用依赖封装到一个starter依赖包中，基于类 SPI 和注解扫描机制实现了配置的**自动装配**，简化了配置文件，同时内嵌入了web服务器如tomcat。

简单来说，spring boot 是 spring 的扩展，主要功能是简化配置，不用写很多配置文件和依赖，让开发人员可以更加关注业务逻辑的开发。

> **低侵入式、应用开发框架、开箱即用（mvc、jdbc）**
>
> **扩展、简化 spring 应用的配置和部署，使用基于 SPI 机制和注解扫描的方式实现了项目依赖的自动装配**
>
> **内嵌 web 服务器**
>
> **相当于是 Context 是 IOC 的封装和扩展**

Q7、Spring 管理事务的方式有几种?

1. 编程式事务，在代码中硬编码。(不推荐使用)
2. 声明式事务，在配置文件中配置（推荐使用）

声明式事务又分为两种：

1. 基于XML的声明式事务
2. 基于注解的声明式事务

声明式事务管理建立在AOP之上的。其本质是通过AOP功能，对方法前后（环绕）进行拦截，将事务处理的功能编织到拦截的方法中，也就是在目标方法开始之前加入一个事务，在执行完目标方法之后根据执行情况提交或者回滚事务。

声明式事务最大的优点就是不需要在业务逻辑代码中掺杂事务管理的代码，只需在配置文件中做相关的事务规则声明或通过@Transactional注解的方式，便可以将事务规则应用到业务逻辑中。

> **编程式事务、声明式事务（XML、注解）**
>
> **声明式事务基于AOP，拦截方法、织入事务逻辑**
>
> **方法开始前加入一个事务，结束后提交或者回滚事务**

Q8、Spring 事务中的隔离级别有哪几种?

TransactionDefinition.ISOLATION_DEFAULT: 使用后端数据库默认的隔离级别，Mysql 默认采用的REPEATABLE_READ隔离级别 Oracle 默认采用的 READ_COMMITTED隔离级别.

TransactionDefinition.ISOLATION_READ_UNCOMMITTED: 最低，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读

TransactionDefinition.ISOLATION_READ_COMMITTED: 允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生

TransactionDefinition.ISOLATION_REPEATABLE_READ: 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，**可以阻止脏读和不可重复读，但幻读仍有可能发生。**

TransactionDefinition.ISOLATION_SERIALIZABLE: 最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，**该级别可以防止脏读、不可重复读以及幻读。**但是这将严重影响程序的**性能。**通常情况下也不会用到该级别。

> **读未提交、读已提交、可重复读、串行化**

Q9、Spring 事务中哪几种事务传播行为?

支持当前事务的情况：

- **TransactionDefinition.PROPAGATION_REQUIRED：** 如果当前存在事务，则加入该事务；如果当前没有事务，则**创建一个新的事务。**（默认）
- **TransactionDefinition.PROPAGATION_SUPPORTS：** 如果当前存在事务，则加入该事务；如果当前没有事务，则**以非事务的方式继续运行。**
- **TransactionDefinition.PROPAGATION_MANDATORY：** 如果当前存在事务，则加入该事务；如果当前没有事务，则**抛出异常。**（mandatory：强制性）

不支持当前事务的情况：

- **TransactionDefinition.PROPAGATION_REQUIRES_NEW： 创建一个新的事务**，如果当前存在事务，则把当前事务**挂起。**
- **TransactionDefinition.PROPAGATION_NOT_SUPPORTED： 以非事务方式运行**，如果当前存在事务，则把当前事务**挂起。**
- **TransactionDefinition.PROPAGATION_NEVER： 以非事务方式运行**，如果当前存在事务，则抛出异常。
- **TransactionDefinition.PROPAGATION_ NESTED**：如果没有，就新建一个事务；如果有，就在当前事务中嵌套其他事务。

> **7种，支持与不支持当前事务**
>
> **如果当前存在事务，则加入该事务（创建、非事务方式运行、抛异常）**
>
> **创建、非事务方式运行、异常（如果存在当前事务）**

Q10、IOC

IoC（Inverse of Control:控制反转）是一种**设计思想**，就是**将原本在程序中手动创建对象的控制权，交由Spring框架来管理。 IoC 容器是 Spring 用来实现 IoC 的载体， 主要包括一个 bean 定义的注册表，以及一个 bean 的单例缓存池，相当于一个Map（key，value）。**

将对象之间的相互**依赖关系**交给 IOC 容器来管理，并由 IOC 容器完成对象的注入。这样可以很大程度上简化应用的开发，解耦模块之间的复杂关系。

IOC 容器初始化过程

> **设计思想，控制权交给Spring框架管理。**

> **IOC容器是Spring用来实现IOC的载体，包含注册表和单例池**
>
> **简化开发，解耦模块之间的关系**

Q11、AOP

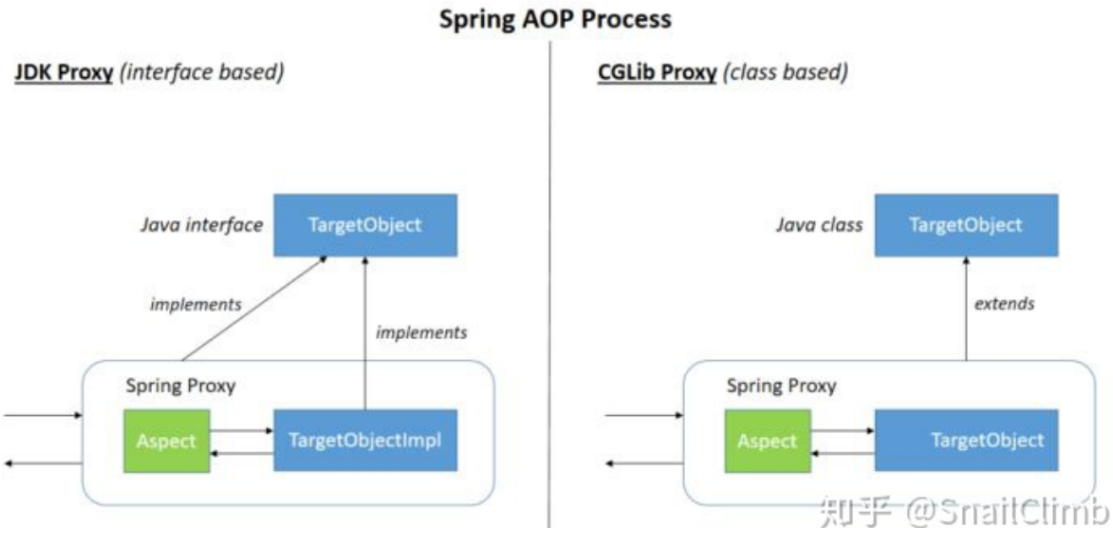
AOP(Aspect-Oriented Programming:面向切面编程)，就是能够将那些与业务无关，**却为业务模块所共同调用的逻辑（例如事务处理、日志管理、权限控制等）封装起来**，便于**减少系统的重复代码**，**降低模块间的耦合度**，并**有利于未来的可拓展性和可维护性。**

AOP实现的关键在于代理模式，AOP代理主要分为**动态代理和静态代理。**动态代理则以 **Spring AOP** 为代表。，静态代理的代表为**AspectJ。**

（1）AspectJ是静态代理的增强，所谓静态代理，就是AOP框架会在**编译阶段生成AOP代理类**，因此也称为编译期增强，他会**在编译阶段将切面逻辑的字节码，然后织入到原有的目标Java字节码中**，运行的时候就是增强之后的AOP对象。

（2）Spring AOP使用的动态代理，所谓的动态代理就是说AOP框架**不会去修改字节码**，而是**每次运行时在内存中临时为方法生成一个代理对象，对象包含了目标对象的全部方法**，并且在特定的切点做了增强处理，最终执行了增强的逻辑与原有方法。

Spring AOP就是基于动态代理的，如果要代理的对象，实现了某个**接口**，那么Spring AOP会使用 **JDK Proxy**，去创建代理对象， 通过 **InvocationHandler** 的**invoke**()方法**反射来调用目标类**中的方法，将**业务逻辑和切面逻辑**交织在一起，而对于没有实现接口的对象，就无法使用 JDK Proxy 去进行代理了，这时候Spring AOP会使用**Cglib** ，这时候Spring AOP会使用 **Cglib** 生成一个被**代理对象的子类来作为代理**，**父类代表业务逻辑，在子类实现了代理逻辑**，这样来将切面逻辑进行织入，如下图所示：



> **业务模块共同调用的逻辑（日志、权限控制），减少重复代码、降低模块耦合度、可扩性**
>
> **静态代理（AspectJ、编译期增强、将切面逻辑的字节码织入原有的目标字节码中）**
>
> **动态代理（Spring AOP、内存生成代理对象，封装了原有对象以及其所有方法，切点增强）**
>
> **实现了接口，JDK 代理，InvocationHandler的invock**
>
> **未实现接口，CGLIB代理，被代理对象的子类执行切面逻辑**

Q12、AOP 几个名词

切面：抽取出来的公共逻辑

连接点：可以织入切面逻辑的点，因为 Spring Aop 只支持到在方法级别的增强，所以这里的连接点指的就是业务逻辑中的每个方法

切点：就是最终确定的连接点，最终织入切面逻辑的点

通知：就是织入切面的时机点，有前置后置通知，环绕通知，意思是在方法执行的前后或者前后都有触发的织入逻辑，还有返回通知，就是方法执行完进行织入，还有异常通知，出了异常后开始织入

> **切面、连接点、切点、通知**

Q13、Spring 中的 bean 的作用域有哪些?

- **singleton** : 默认单例模式
- **prototype** : 原型模式，每次请求会创建一个新的 bean 实例。
- **request** : 每一次HTTP请求都会产生一个新的bean，仅在当次 request 中有效
- **session** : 每一次HTTP请求都会产生一个新的 bean，仅在当次 session 中有效
- **global-session**： 全局session作用域

> **单例、原型、request、session、全局session**

Q14、Spring框架中的单例Beans是线程安全的吗

Spring框架并没有对单例bean进行多线程的处理，比如添加任务 JVM 锁或者其他同步机制等。关于单例bean的线程安全和并发问题需要开发者自行去搞定。如果业务逻辑中的 bean 存在状态的话，就需要自己保证线程安全，最简单的方式就是把 bean 的作用域从单例变成原型。每次请求每个线程创建一个 bean 的实例。

还可以采用 ThreadLocal 进行处理，解决线程安全问题。ThreadLocal会为每一个线程提供一个独立的变量副本，从而隔离了多个线程对数据的访问冲突。在编写多线程代码时，可以把不安全的变量封装进 ThreadLocal。

ThreadLocal和线程同步机制都是为了解决多线程中相同变量的访问冲突问题。同步机制采用了“时间换空间”的方式，仅提供一份变量，不同的线程在访问前需要获取锁，没获得锁的线程则需要排队。而 ThreadLocal采用了“空间换时间”的方式。

> **默认是单例 bean，没有线程安全处理，需要自己处理。**
>
> **设置成原型 bean，或者用 ThreadLocal 提供线程内部的一个变量副本**

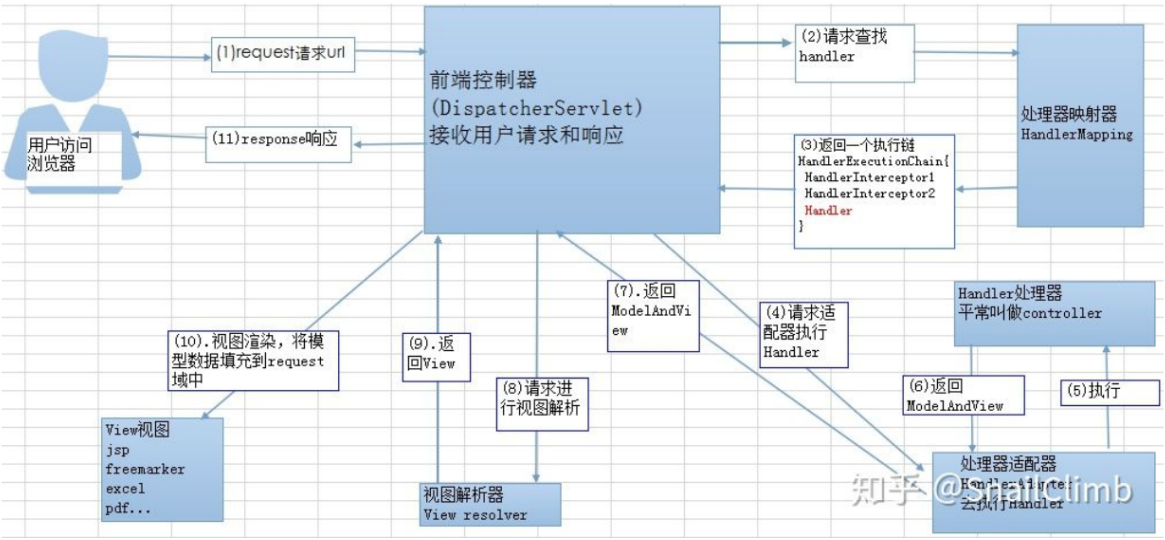
Q15、Spring MVC 的控制器是不是单例模式

默认情况下是单例模式，在多线程进行访问的时候,有线程安全问题。但是不建议使用同步，因为会影响性能。

解决方案是在控制器里面不能写成员变量，通过函数来传递变量，或者使用 ThreadLocal 的方式让每个线程都持有一个独立的变量副本。

> **不写成员变量，通过函数来传递，在栈中的对象不能共享没有线程安全问题**

Q16、SpringMVC 工作原理



主要包括了前端控制器、请求映射器、请求处理器、视图解析器这几个部分。

前端控制器接收到请求后，分发给一个请求映射器，请求映射器根据预设的规则寻找请求的处理器。比如我们在 Controller 的接口上用注解定义了相关的路径、参数和请求方式等，就是规定了一系列规则，请求映射器就会将请求分发给合适的处理器来处理。

处理器处理完毕会返回 ModelAndView 模型数据给前端控制器，前端控制器再转发给视图解析器，解析出一个逻辑视图，就是一个字符串名称代表的视图返回给前端控制器，前端控制器再根据这个逻辑视图名到资源文件下找到真实的视图文件，将模型数据填充到视图文件中进行渲染，最后再返回给用户。

> **前端控制器 -> 请求映射器（按规则查找） -> 请求处理器（填充ModelAndView） -> 视图解析（解析出逻辑视图名）-> 渲染数据到页面**

Q17、Spring 框架中用到了哪些设计模式?

- **工厂模式**：spring中的BeanFactory就是简单工厂模式的体现，根据传入一个唯一的标识 beanid 来获得bean对象
- **单例模式**：bean 默认单例模式
- **代理模式**：使用 JDK 或者 CGLIB 动态代理实现 AOP 功能
- **观察者模式**：Spring 应用上下文以及 IOC 容器 以及 bean 等组件的生命周期的事件发布和监听机制
- **适配器模式**：SpringMVC 中前端控制器接收到请求后分发给一个请求映射器，请求映射器根据预设的规则寻找请求的处理器的过程

Q18、列举一些 Spring 的重要模块

Spring Core： 基础核心模块，提供 IOC 以及依赖注入功能

Spring AOP ： 提供了面向切面编程功能

Spring JDBC : 提供 Java 数据库连接API

Spring JMS ： 提供 Java 消息服务API

Spring Test : 提供对单元测试的支持

> **core、aop、jdbc、jms、test**

Q19、Spring IOC 注入方式

构造器注入、setter方法注入、根据注解注入。

Q20、哪种注入的方式比较好

构造器方式：会固定地依赖于注入的顺序，而且无法解决 bean 之间的循环依赖的问题，而且构造器注入的方式不能使用 CGLIB 动态代理，只能使用基于接口的代理或者虚拟的无参构造函数，而且如果构造器的参数过多，那么这个构造函数责任更大，不利于维护，而且代码很累赘不雅观。

setter方式：只有当对象需要注入的时候才会为我们注入，而不是在类初始化的时候就都注入，缺点是不能把对象定义为 final。

成员变量方式：可读性高，方便维护。但是容易引起空指针问题，也不能把对象定义为 final。

1.强制性的依赖性或者当目标不可变时，使用构造函数注入（**应该说尽量都使用构造器来注入**）

2.可选或多变的依赖使用setter注入（**建议可以使用构造器结合setter的方式来注入**）

3.在大多数的情况下避免field域注入（**感觉大多数同学可能会有异议，毕竟这个方式写起来非常简便，但是它的弊端确实远大于这些优点**）

4.Spring 4.3+ 的同学可以试一试构造器的隐式注入，采用此方式注入后，使得我们的代码更优雅，更独立，减少了对Spring的依赖性。

> **构造器注入（依赖于固定的注入顺序、且不能解决循环依赖的问题、而且不能用CGLIB的动态代理、参数过多也不好维护）**
>
> **setter注入（只有在需要的时候才注入，不用在构造的时候就注入，缺点是不能把对象定义为final）**
>
> **@Autowire成员变量注入（可读性高、方便维护、但也不能定义为final）**
>
> **三种方式具体情况具体分析**

Q21、哪种自动装配的方式比较好

通常情况下，推荐使用通过注解和扫描器的方式来进行组件的自动装配，这样的好处是可读性高，便捷灵活，而且模块之间的耦合性更低，我们可以更加专注于业务代码而不是写配置文件。

当第三方类库无法通过注解的方式进行自动扫描和装配，这时需要用 JavaConfig 的形式来定义一个bean进行装配，同时相比于 XML 方式而言，这种显示的方式更加安全，因为 XML 的方式会存在当我们用 bean 标签定义了一个 bean，并且用 class 属性指定了一个类的路径后，后面当我们重构的时候不小心改了类名，但是这里的配置文件没有改回来的情况，就容易出错。

> **推荐用注解+扫描器，可读性高，耦合性低**
>
> **当第三方类库不能注解时，用 JavaBean 定义，比 XML 安全**

Q22、自动装配的局限性

不能自动装配简单的属性，如基本数据类型，String。

不能自动装配第三方类库，需要用 Java 代码手动编写配置类的方式来实现。

Q23、循环依赖

循环依赖是指n个类中循环套用，比如a类依赖了b类，b类依赖了c类，c类又依赖了a类。在 Spring 中，大概有三种情况会出现。

1）构造器的循环依赖

比如在上述三个类的构造函数中进行了循环依赖，那么当 C 类的构造函数依赖了 A 类时，就会报循环依赖的异常。因为 Spring 在实例化 bean 的过程中，如果当前 bean 是正在被创建的，那么这个 bean 会被放在一个当前创建的 bean 池中，同时会有一个标识符，来标识这个类正在被创建，而且这个标识符也会被缓存在池子中。一旦在 bean 的构建过程中发现自己正在被创建，那么就会抛出循环依赖的异常，这种情况由于 Spring IOC 容器的初始化机制，无法解决。

2）setter 方法注入单例 bean

这种情况 Spring 可以解决循环依赖问题，因为 Spring 创建 bean 的过程，是先实例化 bean ，然后把 bean 缓存到一个单例 bean 的缓存池里，然后根据需要取出再注入。所以 setter 的方式，由于三个类已经创建完毕，spring 就会把三个类的单例 bean 缓存起来，然后当执行到 setter 方法时，就会去单例池中取出再注入，就可以解决循环依赖的问题。

3）setter 方法注入原型 bean

因为 spring 只会缓存单例的 bean ，并不会缓存原型 bean ，原型 bean 是指在每次请求的时候才去创建一个实例 bean，所以执行当 setter 方法时，由于还没有创建，所以这时也会报错。

> **构造器循环依赖（如果依赖的对象正在被创建，那么就会抛出异常）**
>
> **setter方法注入单例依赖（由于依赖的对象已经创建好并放在 IOC 容器的单例对象池汇中，所以可以解决）**
>
> **setter方法注入原型依赖（因为原型 bean 不会被缓存到单例缓冲池里，所以找不到实例对象，抛出异常）**

Q24、内部 bean 是什么

只有将 bean 用作另一个 bean 的属性时，才能将 bean 声明为内部 bean。

Q25、Bean 的实例化方式有几种

【xml 方式】

1）构造器方式

比如在 xml 配置文件中用标签来声明无参和有参的构造函数

```
<bean name="people" class="com.nieshenkuan.model.People"></bean>
```

```
<bean name="people2" class="com.nieshenkuan.model.People">
    <constructor-arg index="0" value="lyy"></constructor-arg>
    <constructor-arg index="1" value="18"></constructor-arg>
    <constructor-arg index="2" ref="collection"></constructor-arg>
</bean>
```

2）setter 属性注入的方式

同样是在配置文件中通过 标签进行属性的注入来实例化

```
<bean name="people" class="com.nieshenkuan.model.People" id="people">
    <property name="name" value="nsk"></property>
    <property name="age" value="24"></property>
</bean>
```

3）静态工厂和实例工厂的方式

同样是在配置文件声明 bean， 同时需要通过factory-method 属性来声明实例化Bean的方法。

```
<!-- 1、定义实例工厂Bean -->
<bean id="beanInstanceFactory" class="com.nieshenkuan.staticFactory.HelloInstanceFactory1"></bean>
<!-- 2、使用实例工厂Bean创建Bean -->
<bean id="hello2" factory-bean="beanInstanceFactory"
    factory-method="newInstanceHello">
    <constructor-arg index="0" value="Helloworld2!"></constructor-arg>
</bean>
```

【注解的方式 @Service、@Controller、@Component】

【FactoryBean 的方式】

> XML，注解 @Service、@Controller、@Component，FactoryBean
>
> 构造器方式（、无参、有参）
>
> setter属性注入方式（）
>
> 静态和动态工厂（定义个实例工厂bean、使用实例工厂 bean 创建 bean）

Q26、如何使用 Spring MVC 完成 JSON 操作

Spring mvc 提供了一种不同**格式数据交互**的机制(JSON、XML以及其他数据格式)，会将前台传来的JSON/XML等格式信息自动转换为相应的**包装类**，或者将输出的信息转换为JSON/XML等格式的数据。

Spring mvc 主要利用**类型转换器**(messageConverters)**将前台信息转换成开发者需要的格式**，然后在相应的**Controller方法**接收参数前添加@**RequestBody**注解，进行数据转换，或在方法的**返回值类型**处添加@**ResponseBody**注解，将返回信息转换成相关格式的数据。

@RequestBody注解特点：根据请求参数的contentType决定是否将相关格式转换至包装类，如果是目标类型，就进行转换。这里转换的是JSON数据，所以要求将前端请求的**contentType**指定为"**application/json**"类型，而普通的key/value请求参数的contentType默认为"**application/x-www-form-urlen**"类型。

> **@RequestBody 与@RequestParam()可以同时使用时，原SpringMVC接收参数的机制不变，只不过RequestBody 接收的是请求体里面的数据；而RequestParam接收的是key-value里面的参数。**
>
> **如果参数时放在请求体中，传入后台的话，那么后台要用@RequestBody才能接收到；如果不是放在请求体中的话，那么后台接收前台传过来的参数时，要用@RequestParam来接收，或则形参前什么也不写也能接收。**
>
> **后端@RequestBody注解对应的类在将HTTP的输入流(含请求体)装配到目标类(即：@RequestBody后面的类)时，会根据json字符串中的key来匹配对应实体类的属性。**

Q27、在 Spring 中注入一个 null 和一个空字符串

通过配置文件的属性注入

```
<bean class="Boweifeng">

<property name="email" /><null/></property>

</bean>

<bean class="Boweifeng">

<property name="email" /><""/></property>

</bean>
```

Q28、如何注入 java 集合?

通过 property 标签和对应的集合标签比如 等以及元素标签来实现

```
<property name="addressList">
   <list>
      <value>INDIA</value>
      <value>Pakistan</value>
      <value>USA</value>
      <value>USA</value>
   </list>
</property>


<property name="addressSet">
   <set>
      <value>INDIA</value>
      <value>Pakistan</value>
      <value>USA</value>
      <value>USA</value>
   </set>
</property>
```

```
<property name="addressMap">
   <map>
      <entry key="1" value="INDIA"/>
      <entry key="2" value="Pakistan"/>
      <entry key="3" value="USA"/>
      <entry key="4" value="USA"/>
   </map>
</property>
```

> 、、

Q29、在 Spring 中如何实现时间处理?

spring 系统提供了一个格式化的日期工厂bean对象，通过注入这个工厂，然后使用其工厂方法即可。

```
<bean id="dateFormat" class="java.text.SimpleDateFormat">
        <constructor-arg value="yyyy-MM-dd" />
</bean>
<bean  id="user" class="zxy.demo.springmvc.domain.UserInfo">
        <constructor-arg value="10" />
        <constructor-arg index="1" value="名字" />
        <constructor-arg index="2">
                <bean factory-bean="dateFormat" factory-method="parse">
                        <constructor-arg value="2016-12-19" />
                </bean>
        </constructor-arg>
</bean>
```

> **dateFormat**

Q30、Spring MVC 里面拦截器是怎么写的?

有两种写法,一种是实现HandlerInterceptor接口，另外一种是继承适配器类HandlerInterceptorAdapter，接着在接口方法当中，实现处理逻辑；然后在SpringMvc的配置文件中顺序配置拦截器即可。项目用的是第二种方式。

> **实现HandlerInterceptor接口、继承HandlerInterceptorAdapter类**

Q31、Autowire 实现原理是什么?

在 bean 进行实例化时，需要进行属性的依赖注入，会有一个后置处理器

AutowiredAnnotationBeanPostProcessor类，会扫描寻找被@Autowire，@Resource，@Value，@Inject注解的类，然后在容器中查找有没有已经实例好的对象，

- 如果查询结果刚好为一个，就将该bean装配给@Autowired指定的数据
- 如果查询的结果不止一个，那么@Autowired会根据qulifie声明的名称来查找。
- 如果查询的结果为空，那么会抛出异常。解决方法时，使用required=false

> **实例化时，依赖注入，扫描注解，解析寻找**

Q32、项目怎么用到 AOP 的?

读写分离、日志记录等

1）定义一个基础数据库 baseDataSource bean，通过 druid 配置数据库

```
<bean id="baseDataSource" class="com.alibaba.druid.pool.DruidDataSource"
destroy-method="close"  abstract="true">
    <property name="driverClassName" value="${db.driverClassName}" />

    <!-- 连接池启动时创建的初始化连接数量（默认值为0） -->
    <property name="initialSize" value="${db.initialSize}" />
    <!-- 连接池中可同时连接的最大的连接数（默认值为8，调整为20，高峰单机器在20并发左右，自己根
据应用场景定） -->
    <property name="maxActive" value="${db.maxActive}" />   <!-- 最大连接数 -->
    <!-- 连接池中最小的空闲的连接数，低于这个数量会被创建新的连接（默认为0，调整为5，该参数越
接近maxIdle，性能越好，因为连接的创建和销毁，都是需要消耗资源的；但是不能太大，因为在机器很空闲
的时候，也会创建低于minidle个数的连接，类似于jvm参数中的Xmn设置） -->
    <property name="minIdle" value="${db.minIdle}" />
    <!-- 最大等待时间，当没有可用连接时，连接池等待连接释放的最大时间，超过该时间限制会抛出异
常，如果设置-1表示无限等待（默认为无限，调整为60000ms，避免因线程池不够用，而导致请求被无限制挂
起） -->
    <property name="maxWait" value="${db.maxWait}" />

    <property name="defaultAutoCommit" value="${db.defaultAutoCommit}" />
    <!-- sql心跳检查 -->
    <!-- 1. 性能问题。 -->
    <!-- 目前网站的应用大部分的瓶颈还是在I/O这一块，大部分的I/O还是在数据库的这一层面上，每一
个请求可能会调用10来次SQL查询，如果不走事务，一个请求会重复获取链接，如果每次获取链接,比如在
testOnBorrow都进行validateObject，性能开销不是很能接受，可以假定一次SQL操作消毫0.5~1ms(一
般走了网络请求基本就这数) -->
    <!-- 2 . 成本和收益 -->
    <!-- 网站异常数据库重启，网络异常断开的频率是非常低的，一般也就在数据库升级，演习维护时才会
进行，而且一般也是选在晚上，访问量相对比较低的请求，而且一般会有人员值班关注，所以异步的
validateObject是可以接受，但一个前提需要确保能保证在一个合理的时间段内，数据库能完成自动重联。
-->
    <!-- GenericObjectPool中针对pool管理，起了一个 异步Evict的TimerTask定时线程进行控制
( 可通过设置参数
      timeBetweenEvictionRunsMillis>0)，定时对线程池中的链接进行validateObject校验，
对无效的链接进行关闭后，会调用ensureMinIdle，适当建立链接保证最小的minIdle连接数 -->
    <property name="testWhileIdle" value="${db.testWhileIdle}" />
    <!-- 在进行borrowObject进行处理时，对拿到的connection进行validateObject校验 -->
    <property name="testOnBorrow" value="${db.testOnBorrow}" />
    <!-- 在进行returnObject对返回的connection进行validateObject校验，个人觉得对数据库连
接池的管理意义不大 -->
    <property name="testOnReturn" value="${db.testOnReturn}" />
    <!-- 代表检查的sql -->
    <property name="validationQuery" value="${db.validationQuery}" />
    <!-- 设置的Evict线程的时间，单位ms，大于0才会开启evict检查线程 -->
    <property name="validationQueryTimeout" value="${db.validationQueryTimeout}"
/>
    <!-- 代表在执行检查时，通过statement设置，
statement.setQueryTimeout(validationQueryTimeout) -->
    <property name="timeBetweenEvictionRunsMillis"
value="${db.timeBetweenEvictionRunsMillis}" />
    <!-- bcp默认是30分，需要开启异步线程Evict，否则不生效。原理很简单，就是通过一个异步线程，
每次检查connnection上一次使用的时间戳，看看是否已经超过这个timeout时间设置 -->
    <!-- <property name="minEvictableIdleTimeMillis ">
<value>${db.minEvictableIdleTimeMillis}</value></property> -->
    <!-- 主要是用于在出现链接紧张时候，会扫描一些链接未超过removeAbandonedTimeout时间还未
被释放，会主动的关闭该链接 -->
```

```
    <!-- <property name="removeAbandoned" ><value>${db.removeAbandoned}</value>
</property> -->
    <!-- <property name="removeAbandonedTimeout ">
<value>${db.removeAbandonedTimeout}</value></property> -->

    <!-- druid统计监控信息 -->
    <property name="filters" value="stat,slf4j,wall" />
    <!-- druid慢SQL记录 -->
    <property name="connectionProperties" value="druid.stat.slowSqlMillis=3000"
/>
    <!-- 合并多个DruidDataSource的监控数据 -->
    <!-- <property name="useGlobalDataSourceStat" value="true" /> -->
  </bean>
```

2）配置读库的 bean 和两个写库的 bean 对象，继承于 baseDataSource

```
  <bean id="writeDataSource" parent="baseDataSource">
    <property name="url" value="${db.master.url}" />
    <property name="username" value="${db.master.username}" />
    <property name="password" value="${db.master.password}" />
  </bean>

  <bean id="readDataSource1" parent="baseDataSource">
    <property name="url" value="${db.slave1.url}" />
    <property name="username" value="${db.slave1.username}" />
    <property name="password" value="${db.slave1.password}" />
  </bean>

  <bean id="readDataSource2" parent="baseDataSource">
    <property name="url" value="${db.slave2.url}" />
    <property name="username" value="${db.slave2.username}" />
    <property name="password" value="${db.slave2.password}" />
  </bean>
```

3）配置多数据源 bean，将读写库作为属性注入

```
  <bean id="dataSource"
class="com.feidee.message.support.database.datasource.readwrite.ReadWriteDynamic
DataSource">
    <property name="readDataSources">
      <list>
        <ref bean="readDataSource1" />
        <ref bean="readDataSource2" />
      </list>
    </property>
    <property name="writeDataSource" ref="writeDataSource" />
  </bean>
```

4）配置 sharding 分库分表中间件

- 分库策略不分库、分表策略自增id取模64
- 配置逻辑表名和实际表名

```
<rdb:strategy id="databaseShardingStrategy" sharding-columns="fid" algorithm-
class="com.dangdang.ddframe.rdb.sharding.api.strategy.database.NoneDatabaseShard
ingAlgorithm"/>
```

```
  <rdb:strategy id="tableShardingStrategy" sharding-columns="fid" algorithm-
expression="t_terminal_sign_${fid.longValue() % 64}"/>

  <rdb:data-source id="shardingDataSource">
    <rdb:sharding-rule data-sources="dataSource" default-data-
source="dataSource">
      <rdb:table-rules>
        <rdb:table-rule logic-table="t_terminal_sign" actual-
tables="t_terminal_sign_${0..63}" database-strategy="databaseShardingStrategy"
table-strategy="tableShardingStrategy" />
      </rdb:table-rules>
    </rdb:sharding-rule>
    <rdb:props>
      <prop key="executor.size">20</prop>
      <prop key="sql.show">false</prop>
    </rdb:props>
  </rdb:data-source>
```

5）配置数据源读写分离的切面实现逻辑

- 设置切点在各个 Dao 类中
- 通过解析 Dao 类中的方法，用正则表达式来匹配，设置不同的方法走不同的读写库
- 通过 determineReadOrWriteDB 方法，本质是通过 DataSourceType 的 setType 方法设置当前线程的数据源

```
<bean id="dateSourceAspect"
class="com.feidee.message.support.database.datasource.readwrite.ReadWriteDateSou
rceAspect">
    <property name="writeMethodNamePattern"> <!-- 正则表达式方式 -->
      <list>
        <value><![CDATA[^add.*]]></value>
        <value><![CDATA[^insert.*]]></value>
        <value><![CDATA[^update.*]]></value>
        <value><![CDATA[^save.*]]></value>
        <value><![CDATA[^delete.*]]></value>
        <value><![CDATA[^getForUpdate.*]]></value>
        <value><![CDATA[.*InWrite.*]]></value>
        <value><![CDATA[^logoff.*]]></value>
        <value><![CDATA[^uninstall.*]]></value>
      </list>
    </property>
  </bean>

  <aop:config expose-proxy="true">
    <aop:aspect ref="dateSourceAspect" order="0">
      <aop:pointcut id="dateSourcePointcut" expression="execution(*
com.feidee.message..dao..*.*(..))"/>
      <aop:around pointcut-ref="dateSourcePointcut"
method="determineReadOrWriteDB" />
    </aop:aspect>
  </aop:config>
```

6）设置 Mybatis框架的 sqlSessionFactory bean 对象

```
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="dataSource" ref="shardingDataSource" />
    <property name="configLocation" value="classpath:config/mybatis.xml" />
    <property name="typeAliasesPackage"
value="com.feidee.message.support.common.entity,com.feidee.message.user.entity"
/>  <!-- 实体类包 -->
    <property name="typeAliasesSuperType"
value="com.feidee.message.support.common.entity.BasePojo" />  <!-- 实体类父类，如果
没有，则mybatis默认为Object -->
    <property name="mapperLocations">
      <list>
        <value>classpath*:config/mappers/*.xml</value>
        <value>classpath*:config/mappers/message/*.xml</value>
      </list>
    </property>
  </bean>
```

7）配置 sqlSessionTemplate bean 对象，注入 sqlSessionFactory bean 对象

```
<bean id="sqlSessionTemplate" class="org.mybatis.spring.SqlSessionTemplate">
    <constructor-arg index="0" ref="sqlSessionFactory" />
  </bean>
```

【汇总】

1）定义一个基础数据库 baseDataSource bean，通过 druid 配置数据库

2）配置读库的 bean 和两个写库的 bean 对象，继承于 baseDataSource

3）配置多数据源 bean，将读写库作为属性注入

4）配置 sharding 分库分表中间件

- 分库策略不分库、分表策略自增id取模64
- 配置逻辑表名和实际表名

5）配置数据源读写分离的切面实现逻辑

- 设置切点在各个 Dao 类中
- 通过解析 Dao 类中的方法，用正则表达式来匹配，设置不同的方法走不同的读写库
- 通过 determineReadOrWriteDB 方法，本质是通过 DataSourceType 的 setType 方法设置当前线程的数据源

6）设置 Mybatis 框架的 sqlSessionFactory bean 对象

7）配置 sqlSessionTemplate bean 对象，注入 sqlSessionFactory bean 对象

> **配置基础数据库信息 bean 对象，通过druid 配置数据库，配置读库和写库的 bean 继承于基础数据库**
>
> **配置多数据源 bean，注入读写库**
>
> **配置 sharding 分库分表中间件，分表（id%64）不分库**
>
> **配置数据源读写分离的切面逻辑（切点、正则匹配方法名、数据库选择方法）**
>
> **配置 Mybatis 的 sqlSessionFactory ，配置 sqlSessionTemplate ，注入前者**