上节回顾

如何clone项目并导入运行

Spring Boot

什么是 Spring Boot

它使用 “习惯优于配置”（项目中存在大量的配置，此外还内置一个习惯性的配置，让你无须手动配置）的理念让你的项目快速运行起来。
它并不是什么新的框架，而是默认配置了很多框架的使用方式，就像 Maven 整合了所有的 jar 包一样，Spring Boot 整合了所有框架

启动原理

```
// 调用静态类，参数对应的就是HelloWorldMainApplication.class以及main方法中的args
public static ConfigurableApplicationContext run(Class<?> primarySource,Strin
g... args) {
    return run(new Class<?>[] { primarySource }, args);
}
public static ConfigurableApplicationContext run(Object[] sources, String[] a
rgs) {
    return (new SpringApplication(sources)).run(args);
```

```
}
```

它实际上会构造一个SpringApplication的实例，并把我们的启动类HelloWorldMainApplication.class作为参数传进去，然后运行它的run方法

```
public SpringApplication(ResourceLoader resourceLoader, Class<?>... primarySo
urces) {
    this.resourceLoader = resourceLoader;
    Assert.notNull(primarySources, "PrimarySources must not be null");
    //把HelloWorldMainApplication.class设置为属性存储起来
    this.primarySources = new LinkedHashSet<>(Arrays.asList(primarySources));
    //设置应用类型是Standard还是Web
    this.webApplicationType = deduceWebApplicationType();
    //设置初始化器(Initializer),最后会调用这些初始化器
    setInitializers((Collection) getSpringFactoriesInstances( ApplicationCont
extInitializer.class));
    //设置监听器(Listener)
    setListeners((Collection) getSpringFactoriesInstances(ApplicationListener
.class));
    this.mainApplicationClass = deduceMainApplicationClass();
}
```

先将HelloWorldMainApplication.class存储在this.primarySources属性中

```
private WebApplicationType deduceWebApplicationType() {
    if (ClassUtils.isPresent(REACTIVE_WEB_ENVIRONMENT_CLASS, null)
            && !ClassUtils.isPresent(MVC_WEB_ENVIRONMENT_CLASS, null)) {
        return WebApplicationType.REACTIVE;
    }
    for (String className : WEB_ENVIRONMENT_CLASSES) {
        if (!ClassUtils.isPresent(className, null)) {
            return WebApplicationType.NONE;
        }
    }
    return WebApplicationType.SERVLET;
}

// 相关常量
private static final String REACTIVE_WEB_ENVIRONMENT_CLASS = "org.springframe
work."
        + "web.reactive.DispatcherHandler";
private static final String MVC_WEB_ENVIRONMENT_CLASS = "org.springframework.
```

```
"
        + "web.servlet.DispatcherServlet";
private static final String[] WEB_ENVIRONMENT_CLASSES = { "javax.servlet.Serv
let",
        "org.springframework.web.context.ConfigurableWebApplicationContext" }
;
```

这里主要是通过类加载器判断REACTIVE相关的Class是否存在，如果不存在，则web环境即为SERVLET类型。

```
private <T> Collection<T> getSpringFactoriesInstances(Class<T> type) {
    return getSpringFactoriesInstances(type, new Class<?>[] {});
}

// 这里的入参type就是ApplicationContextInitializer.class
private <T> Collection<T> getSpringFactoriesInstances(Class<T> type,
        Class<?>[] parameterTypes, Object... args) {
    ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
    // 使用Set保存names来避免重复元素
    Set<String> names = new LinkedHashSet<>(
            SpringFactoriesLoader.loadFactoryNames(type, classLoader));
    // 根据names来进行实例化
    List<T> instances = createSpringFactoriesInstances(type, parameterTypes,
            classLoader, args, names);
    // 对实例进行排序
    AnnotationAwareOrderComparator.sort(instances);
    return instances;
}
```

```
// parameterTypes: 上一步得到的names集合
private <T> List<T> createSpringFactoriesInstances(Class<T> type,
        Class<?>[] parameterTypes, ClassLoader classLoader, Object[] args,
        Set<String> names) {
    List<T> instances = new ArrayList<T>(names.size());
    for (String name : names) {
        try {
            Class<?> instanceClass = ClassUtils.forName(name, classLoader);
            //确认被加载类是ApplicationContextInitializer的子类
            Assert.isAssignable(type, instanceClass);
            Constructor<?> constructor = instanceClass.getDeclaredConstructor
(parameterTypes);
            //反射实例化对象
```

```
            T instance = (T) BeanUtils.instantiateClass(constructor, args);
            //加入List集合中
            instances.add(instance);
        }
        catch (Throwable ex) {
            throw new IllegalArgumentException(
                    "Cannot instantiate " + type + " : " + name, ex);
        }
    }
    return instances;
}
```

确认被加载的类确实是org.springframework.context.ApplicationContextInitializer的子类，然后就是得到构造器进行初始化，最后放入到实例列表中。

因此，所谓的初始化器就是org.springframework.context.ApplicationContextInitializer的实现类，这个接口是这样定义的：

```
public interface ApplicationContextInitializer<C extends ConfigurableApplicat
ionContext> {

    void initialize(C applicationContext);

}
```

在Spring上下文被刷新之前进行初始化的操作。典型地比如在Web应用中，注册Property Sources或者是激活Profiles。Property Sources比较好理解，就是配置文件。Profiles是Spring为了在不同环境下(如DEV，TEST，PRODUCTION等)，加载不同的配置项而抽象出来的一个实体。

设置监听器(Listener)
下面开始设置监听器：

```
// 这里的入参type是：org.springframework.context.ApplicationListener.class
private <T> Collection<? extends T> getSpringFactoriesInstances(Class<T> type
) {
    return getSpringFactoriesInstances(type, new Class<?>[] {});
}

private <T> Collection<? extends T> getSpringFactoriesInstances(Class<T> type
,
        Class<?>[] parameterTypes, Object... args) {
```

```
    ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
    // Use names and ensure unique to protect against duplicates
    Set<String> names = new LinkedHashSet<String>(
            SpringFactoriesLoader.loadFactoryNames(type, classLoader));
    List<T> instances = createSpringFactoriesInstances(type, parameterTypes,
            classLoader, args, names);
    AnnotationAwareOrderComparator.sort(instances);
    return instances;
}
```

可以发现，这个加载相应的类名，然后完成实例化的过程和上面在设置初始化器时如出一辙，同样，还是以spring-boot-autoconfigure这个包中的spring.factories为例，看看相应的Key-Value：

```
org.springframework.context.ApplicationListener=\
org.springframework.boot.autoconfigure.BackgroundPreinitializer

org.springframework.context.ApplicationListener=\
org.springframework.boot.ClearCachesApplicationListener,\
org.springframework.boot.builder.ParentContextCloserApplicationListener,\
org.springframework.boot.context.FileEncodingApplicationListener,\
org.springframework.boot.context.config.AnsiOutputApplicationListener,\
org.springframework.boot.context.config.ConfigFileApplicationListener,\
org.springframework.boot.context.config.DelegatingApplicationListener,\
org.springframework.boot.context.logging.ClasspathLoggingApplicationListener,
\
org.springframework.boot.context.logging.LoggingApplicationListener,\
org.springframework.boot.liquibase.LiquibaseServiceLocatorApplicationListener
```

这10个监听器会贯穿springBoot整个生命周期。至此，对于SpringApplication实例的初始化过程就结束了。
完成了SpringApplication实例化，下面开始调用run方法：

```
public ConfigurableApplicationContext run(String... args) {
    // 计时工具
    StopWatch stopWatch = new StopWatch();
    stopWatch.start();

    ConfigurableApplicationContext context = null;
    Collection<SpringBootExceptionReporter> exceptionReporters = new ArrayLis
t<>();
```

```
    configureHeadlessProperty();

    // 第一步：获取并启动监听器
    SpringApplicationRunListeners listeners = getRunListeners(args);
    listeners.starting();

    try {
        ApplicationArguments applicationArguments = new DefaultApplicationArg
uments(args);

        // 第二步：根据SpringApplicationRunListeners以及参数来准备环境
        ConfigurableEnvironment environment = prepareEnvironment(listeners,ap
plicationArguments);
        configureIgnoreBeanInfo(environment);

        // 准备Banner打印器 - 就是启动Spring Boot的时候打印在console上的ASCII艺术字
体
        Banner printedBanner = printBanner(environment);

        // 第三步：创建Spring容器
        context = createApplicationContext();

        exceptionReporters = getSpringFactoriesInstances(
                SpringBootExceptionReporter.class,
                new Class[] { ConfigurableApplicationContext.class }, context
);

        // 第四步：Spring容器前置处理
        prepareContext(context, environment, listeners, applicationArguments,
printedBanner);

        // 第五步：刷新容器
        refreshContext(context);
        // 第六步：Spring容器后置处理
        afterRefresh(context, applicationArguments);

        // 第七步：发出结束执行的事件
        listeners.started(context);
        // 第八步：执行Runners
        this.callRunners(context, applicationArguments);
        stopWatch.stop();
        // 返回容器
        return context;
```

```
    }
    catch (Throwable ex) {
        handleRunFailure(context, listeners, exceptionReporters, ex);
        throw new IllegalStateException(ex);
    }
}
```

跟进getRunListeners方法：

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

这里仍然利用了getSpringFactoriesInstances方法来获取实例，大家可以看看前面的这个方法分析，从META-INF/spring.factories中读取Key为org.springframework.boot.SpringApplicationRunListener的Values：

```
org.springframework.boot.SpringApplicationRunListener=\
org.springframework.boot.context.event.EventPublishingRunListener
```

getSpringFactoriesInstances中反射获取实例时会触发EventPublishingRunListener的构造函数，我们来看看EventPublishingRunListener的构造函数：

```
public class EventPublishingRunListener implements SpringApplicationRunListen
er, Ordered {
    private final SpringApplication application;
    private final String[] args;
    //广播器
    private final SimpleApplicationEventMulticaster initialMulticaster;

    public EventPublishingRunListener(SpringApplication application, String[]
 args) {
        this.application = application;
        this.args = args;
        this.initialMulticaster = new SimpleApplicationEventMulticaster();
        Iterator var3 = application.getListeners().iterator();

        while(var3.hasNext()) {
```

```
            ApplicationListener<?> listener = (ApplicationListener)var3.next(
);
            //将上面设置到SpringApplication的十一个监听器全部添加到SimpleApplicatio
nEventMulticaster这个广播器中
            this.initialMulticaster.addApplicationListener(listener);
        }

    }
    //略...
}
```

我们看到EventPublishingRunListener里面有一个广播器，EventPublishingRunListener 的构造方法将SpringApplication的十一个监听器全部添加到SimpleApplicationEventMulticaster这个广播器中，我们来看看是如何添加到广播器：

```
public abstract class AbstractApplicationEventMulticaster implements Applicat
ionEventMulticaster, BeanClassLoaderAware, BeanFactoryAware {
    //广播器的父类中存放保存监听器的内部内
    private final AbstractApplicationEventMulticaster.ListenerRetriever defau
ltRetriever = new AbstractApplicationEventMulticaster.ListenerRetriever(false
);

    @Override
    public void addApplicationListener(ApplicationListener<?> listener) {
        synchronized (this.retrievalMutex) {
            Object singletonTarget = AopProxyUtils.getSingletonTarget(listene
r);
            if (singletonTarget instanceof ApplicationListener) {
                this.defaultRetriever.applicationListeners.remove(singletonTa
rget);
            }
            //内部类对象
            this.defaultRetriever.applicationListeners.add(listener);
            this.retrieverCache.clear();
        }
    }

    private class ListenerRetriever {
        //保存所有的监听器
        public final Set<ApplicationListener<?>> applicationListeners = new L
inkedHashSet();
        public final Set<String> applicationListenerBeans = new LinkedHashSet
```

```
();
        private final boolean preFiltered;

        public ListenerRetriever(boolean preFiltered) {
            this.preFiltered = preFiltered;
        }

        public Collection<ApplicationListener<?>> getApplicationListeners() {
            LinkedList<ApplicationListener<?>> allListeners = new LinkedList(
);
            Iterator var2 = this.applicationListeners.iterator();

            while(var2.hasNext()) {
                ApplicationListener<?> listener = (ApplicationListener)var2.n
ext();
                allListeners.add(listener);
            }

            if (!this.applicationListenerBeans.isEmpty()) {
                BeanFactory beanFactory = AbstractApplicationEventMulticaster
.this.getBeanFactory();
                Iterator var8 = this.applicationListenerBeans.iterator();

                while(var8.hasNext()) {
                    String listenerBeanName = (String)var8.next();

                    try {
                        ApplicationListener<?> listenerx = (ApplicationListen
er)beanFactory.getBean(listenerBeanName, ApplicationListener.class);
                        if (this.preFiltered || !allListeners.contains(listen
erx)) {
                            allListeners.add(listenerx);
                        }
                    } catch (NoSuchBeanDefinitionException var6) {
                        ;
                    }
                }
            }

            AnnotationAwareOrderComparator.sort(allListeners);
            return allListeners;
        }
    }
```

```
    //略...
}
```

上述方法定义在SimpleApplicationEventMulticaster父类AbstractApplicationEventMulticaster中。关键代码为this.defaultRetriever.applicationListeners.add(listener);，这是一个内部类，用来保存所有的监听器。也就是在这一步，将spring.factories中的监听器传递到SimpleApplicationEventMulticaster中。我们现在知道EventPublishingRunListener中有一个广播器SimpleApplicationEventMulticaster，SimpleApplicationEventMulticaster广播器中又存放所有的监听器。

启动监听器

我们上面一步通过getRunListeners方法获取的监听器为EventPublishingRunListener，从名字可以看出是启动事件发布监听器，主要用来发布启动事件。

```
public class EventPublishingRunListener implements SpringApplicationRunListen
er, Ordered {
    private final SpringApplication application;
    private final String[] args;
    private final SimpleApplicationEventMulticaster initialMulticaster;
```

我们先来看看SpringApplicationRunListener这个接口

```
package org.springframework.boot;
public interface SpringApplicationRunListener {

    // 在run()方法开始执行时，该方法就立即被调用，可用于在初始化最早期时做一些工作
    void starting();
    // 当environment构建完成，ApplicationContext创建之前，该方法被调用
    void environmentPrepared(ConfigurableEnvironment environment);
    // 当ApplicationContext构建完成时，该方法被调用
    void contextPrepared(ConfigurableApplicationContext context);
    // 在ApplicationContext完成加载，但没有被刷新前，该方法被调用
    void contextLoaded(ConfigurableApplicationContext context);
    // 在ApplicationContext刷新并启动后，CommandLineRunners和ApplicationRunner未
被调用前，该方法被调用
    void started(ConfigurableApplicationContext context);
    // 在run()方法执行完成前该方法被调用
    void running(ConfigurableApplicationContext context);
    // 当应用运行出错时该方法被调用
    void failed(ConfigurableApplicationContext context, Throwable exception);
```

```
}
```

SpringApplicationRunListener接口在Spring Boot 启动初始化的过程中各种状态时执行，我们也可以添加自己的监听器，在SpringBoot初始化时监听事件执行自定义逻辑，我们先来看看SpringBoot启动时第一个启动事件listeners.starting()：

```
@Override
public void starting() {
    //关键代码，先创建application启动事件`ApplicationStartingEvent`
    this.initialMulticaster.multicastEvent(new ApplicationStartingEvent(this.
application, this.args));
}
```

这里先创建了一个启动事件ApplicationStartingEvent，我们继续跟进SimpleApplicationEventMulticaster，有个核心方法：

```
@Override
public void multicastEvent(final ApplicationEvent event, @Nullable Resolvable
Type eventType) {
    ResolvableType type = (eventType != null ? eventType : resolveDefaultEven
tType(event));
    //通过事件类型ApplicationStartingEvent获取对应的监听器
    for (final ApplicationListener<?> listener : getApplicationListeners(even
t, type)) {
        //获取线程池，如果为空则同步处理。这里线程池为空，还未没初始化。
        Executor executor = getTaskExecutor();
        if (executor != null) {
            //异步发送事件
            executor.execute(() -> invokeListener(listener, event));
        }
        else {
            //同步发送事件
            invokeListener(listener, event);
        }
    }
}
```

我们选择springBoot 的日志监听器来进行讲解，核心代码如下：

```
@Override
public void onApplicationEvent(ApplicationEvent event) {
```

```
    //在springboot启动的时候
    if (event instanceof ApplicationStartedEvent) {
        onApplicationStartedEvent((ApplicationStartedEvent) event);
    }
    //springboot的Environment环境准备完成的时候
    else if (event instanceof ApplicationEnvironmentPreparedEvent) {
        onApplicationEnvironmentPreparedEvent(
                (ApplicationEnvironmentPreparedEvent) event);
    }
    //在springboot容器的环境设置完成以后
    else if (event instanceof ApplicationPreparedEvent) {
        onApplicationPreparedEvent((ApplicationPreparedEvent) event);
    }
    //容器关闭的时候
    else if (event instanceof ContextClosedEvent && ((ContextClosedEvent) eve
nt)
            .getApplicationContext().getParent() == null) {
        onContextClosedEvent();
    }
    //容器启动失败的时候
    else if (event instanceof ApplicationFailedEvent) {
        onApplicationFailedEvent();
    }
}
```

因为我们的事件类型为ApplicationEvent，所以会执行onApplicationStartedEvent((ApplicationStartedEvent) event);。springBoot会在运行过程中的不同阶段，发送各种事件，来执行对应监听器的对应方法。

第二步：环境构建

ConfigurableEnvironment environment = prepareEnvironment(listeners,applicationArguments);

```
private ConfigurableEnvironment prepareEnvironment(
        SpringApplicationRunListeners listeners,
        ApplicationArguments applicationArguments) {
    //获取对应的ConfigurableEnvironment
    ConfigurableEnvironment environment = getOrCreateEnvironment();
    //配置
    configureEnvironment(environment, applicationArguments.getSourceArgs());
    //发布环境已准备事件，这是第二次发布事件
    listeners.environmentPrepared(environment);
    bindToSpringApplication(environment);
```

```
    ConfigurationPropertySources.attach(environment);
    return environment;
}
```

来看一下getOrCreateEnvironment()方法，前面已经提到，environment已经被设置了servlet类型，所以这里创建的是环境对象是StandardServletEnvironment。

```
private ConfigurableEnvironment getOrCreateEnvironment() {
    if (this.environment != null) {
        return this.environment;
    }
    if (this.webApplicationType == WebApplicationType.SERVLET) {
        return new StandardServletEnvironment();
    }
    return new StandardEnvironment();
}
```

第三步：创建容器

第四步：Spring容器前置处理

这一步主要是在容器刷新之前的准备动作。包含一个非常关键的操作：将启动类注入容器，为后续开启自动化配置奠定基础。

```
private void prepareContext(ConfigurableApplicationContext context,
        ConfigurableEnvironment environment, SpringApplicationRunListeners li
steners,
        ApplicationArguments applicationArguments, Banner printedBanner) {
    //设置容器环境，包括各种变量
    context.setEnvironment(environment);
    //执行容器后置处理
    postProcessApplicationContext(context);
    //执行容器中的ApplicationContextInitializer（包括 spring.factories和自定义的实
例）
    applyInitializers(context);
    //发送容器已经准备好的事件，通知各监听器
    listeners.contextPrepared(context);

    //注册启动参数bean，这里将容器指定的参数封装成bean，注入容器
    context.getBeanFactory().registerSingleton("springApplicationArguments",
            applicationArguments);
    //设置banner
    if (printedBanner != null) {
```

```
        context.getBeanFactory().registerSingleton("springBootBanner", printe
dBanner);
    }
    //获取我们的启动类指定的参数，可以是多个
    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);
}
```

```
protected void applyInitializers(ConfigurableApplicationContext context) {
    // 1. 从SpringApplication类中的initializers集合获取所有的ApplicationContextIn
itializer
    for (ApplicationContextInitializer initializer : getInitializers()) {
        // 2. 循环调用ApplicationContextInitializer中的initialize方法
        Class<?> requiredType = GenericTypeResolver.resolveTypeArgument(
                initializer.getClass(), ApplicationContextInitializer.class);
        Assert.isInstanceOf(requiredType, context, "Unable to call initialize
r.");
        initializer.initialize(context);
    }
}
```

这里终于用到了在创建SpringApplication实例时设置的初始化器了，依次对它们进行遍历，并调用initialize方法。我们也可以自定义初始化器，并实现initialize方法，然后放入META-INF/spring.factories配置文件中Key为：org.springframework.context.ApplicationContextInitializer的value中，这里我们自定义的初始化器就会被调用，是我们项目初始化的一种方式

加载启动指定类（重点）

大家先回到文章最开始看看，在创建SpringApplication实例时，先将HelloWorldMainApplication.class存储在this.primarySources属性中，现在就是用到这个属性的时候了，我们来看看getAllSources（）

```
public Set<Object> getAllSources() {
    Set<Object> allSources = new LinkedHashSet();
    if (!CollectionUtils.isEmpty(this.primarySources)) {
        //获取primarySources属性，也就是之前存储的HelloWorldMainApplication.class
```

```
        allSources.addAll(this.primarySources);
    }

    if (!CollectionUtils.isEmpty(this.sources)) {
        allSources.addAll(this.sources);
    }

    return Collections.unmodifiableSet(allSources);
}
```

很明显，获取了this.primarySources属性，也就是我们的启动类HelloWorldMainApplication.class，我们接着看load(context, sources.toArray(new Object[0]));

```
protected void load(ApplicationContext context, Object[] sources) {
    BeanDefinitionLoader loader = createBeanDefinitionLoader(getBeanDefinitio
nRegistry(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();
}

private int load(Class<?> source) {
    if (isGroovyPresent()
            && GroovyBeanDefinitionSource.class.isAssignableFrom(source)) {
        // Any GroovyLoaders added in beans{} DSL can contribute beans here
        GroovyBeanDefinitionSource loader = BeanUtils.instantiateClass(source
,
                GroovyBeanDefinitionSource.class);
        load(loader);
    }
    if (isComponent(source)) {
        //以注解的方式，将启动类bean信息存入beanDefinitionMap，也就是将HelloWorldMai
nApplication.class存入了beanDefinitionMap
        this.annotatedReader.register(source);
        return 1;
```

```
    }
    return 0;
}
```

启动类HelloWorldMainApplication.class被加载到 beanDefinitionMap中，后续该启动类将作为开启自动化配置的入口，后面一篇文章我会详细的分析，启动类是如何加载，以及自动化配置开启的详细流程。

通知监听器，容器已准备就绪

listeners.contextLoaded(context);
主还是针对一些日志等监听器的响应处理。
第五步：刷新容器
执行到这里，springBoot相关的处理工作已经结束，接下的工作就交给了spring。我们来看看refreshContext(context);

```
protected void refresh(ApplicationContext applicationContext) {
    Assert.isInstanceOf(AbstractApplicationContext.class, applicationContext)
;
    //调用创建的容器applicationContext中的refresh()方法
    ((AbstractApplicationContext)applicationContext).refresh();
}
public void refresh() throws BeansException, IllegalStateException {
    synchronized (this.startupShutdownMonitor) {
        /**
         * 刷新上下文环境
         */
        prepareRefresh();

        /**
         * 初始化BeanFactory，解析XML，相当于之前的XmlBeanFactory的操作，
         */
        ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory(
);

        /**
         * 为上下文准备BeanFactory，即对BeanFactory的各种功能进行填充，如常用的注解@Au
towired @Qualifier等
         * 添加ApplicationContextAwareProcessor处理器
         * 在依赖注入忽略实现*Aware的接口，如EnvironmentAware、ApplicationEventPub
lisherAware等
         * 注册依赖，如一个bean的属性中含有ApplicationEventPublisher(beanFactory)
```

```
，则会将beanFactory的实例注入进去
         */
        prepareBeanFactory(beanFactory);

        try {
            /**
             * 提供子类覆盖的额外处理，即子类处理自定义的BeanFactoryPostProcess
             */
            postProcessBeanFactory(beanFactory);

            /**
             * 激活各种BeanFactory处理器,包括BeanDefinitionRegistryBeanFactoryPo
stProcessor和普通的BeanFactoryPostProcessor
             * 执行对应的postProcessBeanDefinitionRegistry方法 和  postProcessBe
anFactory方法
             */
            invokeBeanFactoryPostProcessors(beanFactory);

            /**
             * 注册拦截Bean创建的Bean处理器，即注册BeanPostProcessor，不是BeanFacto
ryPostProcessor，注意两者的区别
             * 注意，这里仅仅是注册，并不会执行对应的方法，将在bean的实例化时执行对应的方
法
             */
            registerBeanPostProcessors(beanFactory);

            /**
             * 初始化上下文中的资源文件，如国际化文件的处理等
             */
            initMessageSource();

            /**
             * 初始化上下文事件广播器，并放入applicatioEventMulticaster,如Applicati
onEventPublisher
             */
            initApplicationEventMulticaster();

            /**
             * 给子类扩展初始化其他Bean
             */
            onRefresh();

            /**
```

```
                * 在所有bean中查找listener bean，然后注册到广播器中
                */
               registerListeners();

               /**
                * 设置转换器
                * 注册一个默认的属性值解析器
                * 冻结所有的bean定义，说明注册的bean定义将不能被修改或进一步的处理
                * 初始化剩余的非惰性的bean，即初始化非延迟加载的bean
                */
               finishBeanFactoryInitialization(beanFactory);

               /**
                * 通过spring的事件发布机制发布ContextRefreshedEvent事件，以保证对应的监听器做进一步的处理
                * 即对那种在spring启动后需要处理的一些类，这些类实现了ApplicationListener<ContextRefreshedEvent>，
                * 这里就是要触发这些类的执行(执行onApplicationEvent方法)
                * 另外，spring的内置Event有ContextClosedEvent、ContextRefreshedEvent、ContextStartedEvent、ContextStoppedEvent、RequestHandleEvent
                * 完成初始化，通知生命周期处理器lifeCycleProcessor刷新过程，同时发出ContextRefreshEvent通知其他人
                */
               finishRefresh();
           }

           finally {

               resetCommonCaches();
           }
       }
   }
```

第六步：Spring容器后置处理

扩展接口，设计模式中的模板方法，默认为空实现。如果有自定义需求，可以重写该方法。比如打印一些启动结束log，或者一些其它后置处理。

第七步：发出结束执行的事件

```
public void started(ConfigurableApplicationContext context) {
    //这里就是获取的EventPublishingRunListener
    Iterator var2 = this.listeners.iterator();
```

```
    while(var2.hasNext()) {
        SpringApplicationRunListener listener = (SpringApplicationRunListener
)var2.next();
        //执行EventPublishingRunListener的started方法
        listener.started(context);
    }
}

public void started(ConfigurableApplicationContext context) {
    //创建ApplicationStartedEvent事件，并且发布事件
    //我们看到是执行的ConfigurableApplicationContext这个容器的publishEvent方法，和
前面的starting是不同的
    context.publishEvent(new ApplicationStartedEvent(this.application, this.a
rgs, context));
}
```

获取EventPublishingRunListener监听器，并执行其started方法，并且将创建的Spring容器传进去了，创建一个ApplicationStartedEvent事件，并执行ConfigurableApplicationContext 的publishEvent方法，也就是说这里是在Spring容器中发布事件，并不是在SpringApplication中发布事件，和前面的starting是不同的，前面的starting是直接向SpringApplication中的11个监听器发布启动事件。

第八步：执行Runners
我们再来看看最后一步callRunners(context, applicationArguments);

```
private void callRunners(ApplicationContext context, ApplicationArguments arg
s) {
    List<Object> runners = new ArrayList<Object>();
    //获取容器中所有的ApplicationRunner的Bean实例
    runners.addAll(context.getBeansOfType(ApplicationRunner.class).values());
    //获取容器中所有的CommandLineRunner的Bean实例
    runners.addAll(context.getBeansOfType(CommandLineRunner.class).values());
    AnnotationAwareOrderComparator.sort(runners);
    for (Object runner : new LinkedHashSet<Object>(runners)) {
        if (runner instanceof ApplicationRunner) {
            //执行ApplicationRunner的run方法
            callRunner((ApplicationRunner) runner, args);
        }
        if (runner instanceof CommandLineRunner) {
            //执行CommandLineRunner的run方法
            callRunner((CommandLineRunner) runner, args);
```

```
        }
    }
}
```

核心注解@SpringBootApplication

Spring Boot应用标注在某个类上，说明这个类是SpringBoot的主配置类，SpringBoot就应该运行这个类的main方法来启动SpringBoot应用

```
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
        @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
        @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeF
ilter.class) })
public @interface SpringBootApplication {}
```

@SpringBootConfiguration

Spring Boot的配置类
标注在某个类上，表示这是一个Spring Boot的配置类
注解定义如下：

```
@Configuration
public @interface SpringBootConfiguration {}
```

其实就是一个Configuration配置类，意思是HelloWorldMainApplication最终会被注册到Spring容器中

@EnableAutoConfiguration
开启自动配置功能
以前使用Spring需要配置的信息，Spring Boot帮助自动配置；
@EnableAutoConfiguration通知SpringBoot开启自动配置功能，这样自动配置才能生效。
注解定义如下：

```
@AutoConfigurationPackage
@Import(EnableAutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {}
@AutoConfigurationPackage
```

自动配置包注解

```
@Import(AutoConfigurationPackages.Registrar.class)
public @interface AutoConfigurationPackage {}
```

@Import(AutoConfigurationPackages.Registrar.class)：默认将主配置类(@SpringBootApplication)所在的包及其子包里面的所有组件扫描到Spring容器中。如下

```
@Order(Ordered.HIGHEST_PRECEDENCE)
static class Registrar implements ImportBeanDefinitionRegistrar, Determinable
Imports {

    @Override
    public void registerBeanDefinitions(AnnotationMetadata metadata,
            BeanDefinitionRegistry registry) {
          //默认将会扫描@SpringBootApplication标注的主配置类所在的包及其子包下所有组件
        register(registry, new PackageImport(metadata).getPackageName());
    }

    @Override
    public Set<Object> determineImports(AnnotationMetadata metadata) {
        return Collections.<Object>singleton(new PackageImport(metadata));
    }
}
```

@Import(EnableAutoConfigurationImportSelector.class)

EnableAutoConfigurationImportSelector： 导入哪些组件的选择器，将所有需要导入的组件以全类名的方式返回，这些组件就会被添加到容器中。

```
//EnableAutoConfigurationImportSelector的父类: AutoConfigurationImportSelector
@Override
public String[] selectImports(AnnotationMetadata annotationMetadata) {
    if (!isEnabled(annotationMetadata)) {
        return NO_IMPORTS;
    }
    try {
        AutoConfigurationMetadata autoConfigurationMetadata = AutoConfigurati
onMetadataLoader
            .loadMetadata(this.beanClassLoader);
```

```
        AnnotationAttributes attributes = getAttributes(annotationMetadata);
        List<String> configurations = getCandidateConfigurations(annotationMe
tadata, attributes);
        configurations = removeDuplicates(configurations);
        configurations = sort(configurations, autoConfigurationMetadata);
        Set<String> exclusions = getExclusions(annotationMetadata, attributes
);
        checkExcludedClasses(configurations, exclusions);
        configurations.removeAll(exclusions);
        configurations = filter(configurations, autoConfigurationMetadata);
        fireAutoConfigurationImportEvents(configurations, exclusions);
        return configurations.toArray(new String[configurations.size()]);
    }
    catch (IOException ex) {
        throw new IllegalStateException(ex);
    }
}
```

List configurations = getCandidateConfigurations(annotationMetadata, attributes);会给容器中注入众多的自动配置类（xxxAutoConfiguration），就是给容器中导入这个场景需要的所有组件，并配置好这些组件。我们跟进去看看

```
protected List<String> getCandidateConfigurations(AnnotationMetadata metadata
,
            AnnotationAttributes attributes) {
    List<String> configurations = SpringFactoriesLoader.loadFactoryNames(
        getSpringFactoriesLoaderFactoryClass(), getBeanClassLoader());
    //...
    return configurations;
}

protected Class<?> getSpringFactoriesLoaderFactoryClass() {
    return EnableAutoConfiguration.class;
}

public static final String FACTORIES_RESOURCE_LOCATION = "META-INF/spring.fac
tories";

public static List<String> loadFactoryNames(Class<?> factoryClass, ClassLoade
r classLoader) {
    String factoryClassName = factoryClass.getName();
    try {
```

```
        //从类路径的META-INF/spring.factories中加载所有默认的自动配置类
        Enumeration<URL> urls = (classLoader != null ? classLoader.getResources(FACTORIES_RESOURCE_LOCATION) :
                                 ClassLoader.getSystemResources(FACTORIES_RESOURCE_LOCATION));
        List<String> result = new ArrayList<String>();
        while (urls.hasMoreElements()) {
            URL url = urls.nextElement();
            Properties properties = PropertiesLoaderUtils.loadProperties(new UrlResource(url));
            //获取EnableAutoConfiguration指定的所有值,也就是EnableAutoConfiguration.class的值
            String factoryClassNames = properties.getProperty(factoryClassName);
            result.addAll(Arrays.asList(StringUtils.commaDelimitedListToStringArray(factoryClassNames)));
        }
        return result;
    }
    catch (IOException ex) {
        throw new IllegalArgumentException("Unable to load [" + factoryClass.getName() + "] factories from location [" + FACTORIES_RESOURCE_LOCATION + "]", ex);
    }
}
```

SpringBoot启动的时候从类路径下的 META-INF/spring.factories中获取EnableAutoConfiguration指定的值，并将这些值作为自动配置类导入到容器中，自动配置类就会生效，最后完成自动配置工作。

自定义starter

```
@Configuration
@ConditionalOnProperty(name = "enabled.autoConfituration", matchIfMissing = true)
public class MyAutoConfiguration {

    static {
        System.out.println("myAutoConfiguration init...");
    }
```

```
    @Bean
    public SimpleBean simpleBean(){
        return new SimpleBean();
    }

}
```

然后在在META-INF下面建一个 spring.factories文件，添加如下配置：

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
spring.study.startup.bean.MyAutoConfiguration
```

最后只需在启动类项目的pom中引入我们的 starter 模块即可

AOP原理

spring boot 中自动化配置是读取/META-INF/spring.factories 中读取org.springframework.boot.autoconfigure.EnableAutoConfiguration配置的值,其中有关aop的只有一个,即org.springframework.boot.autoconfigure.aop.AopAutoConfiguration.该类就是打开宝箱的钥匙

1. AopAutoConfiguration 声明了如下注解:

   ```
   @Configuration
   @ConditionalOnClass({ EnableAspectJAutoProxy.class, Aspect.class, Advice.
   class })
   @ConditionalOnProperty(prefix = "spring.aop", name = "auto", havingValue
   = "true", matchIfMissing = true)
   ```

   @Configuration –> 配置类
   @ConditionalOnClass({ EnableAspectJAutoProxy.class, Aspect.class, Advice.class }) –> 在当前的类路径下存在EnableAspectJAutoProxy.class, Aspect.class, Advice.class时该配置才被解析
   @ConditionalOnProperty(prefix = “spring.aop”, name = “auto”, havingValue = “true”, matchIfMissing = true) –> 当配置有spring.aop.auto= true时生效.如果没有配置,则默认生效

2. AopAutoConfiguration中只有2个内部类:

   i. JdkDynamicAutoProxyConfiguration,代码如下:

```
@Configuration
@EnableAspectJAutoProxy(proxyTargetClass = false)
@ConditionalOnProperty(prefix = "spring.aop", name = "proxy-target-class"
, havingValue = "false", matchIfMissing = true)
public static class JdkDynamicAutoProxyConfiguration {
}
```

@Configuration –> 配置类
@EnableAspectJAutoProxy(proxyTargetClass = false) –> 开启aop注解,关于该注解,后面有解析.
@ConditionalOnProperty(prefix = “spring.aop”, name = “proxy-target-class”, havingValue = “false”, matchIfMissing = true)–> 当配置有spring.aop.proxy-target-class= false时生效,如果没有配置,默认生效.

i. CglibAutoProxyConfiguration,代码如下:

```
@Configuration
@EnableAspectJAutoProxy(proxyTargetClass = true)
@ConditionalOnProperty(prefix = "spring.aop", name = "proxy-target-class"
, havingValue = "true", matchIfMissing = false)
public static class CglibAutoProxyConfiguration {
}
```

@Configuration –> 配置类
@EnableAspectJAutoProxy(proxyTargetClass = false) –> 开启aop注解
@ConditionalOnProperty(prefix = “spring.aop”, name = “proxy-target-class”, havingValue = “true”, matchIfMissing = false)–> 当配置有spring.aop.proxy-target-class= true时生效,如果没有配置,默认不生效.
因此,我们可以知道,aop 默认生效的JdkDynamicAutoProxyConfiguration.这也符合我们关于spring aop的认知,默认是使用jdk,如果使用面向类的代理,则需要配置spring.aop.proxy-target-class=true,来使用cglib进行代理

3. @EnableAspectJAutoProxy 代码如下:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import(AspectJAutoProxyRegistrar.class)
public @interface EnableAspectJAutoProxy {
```

```
    // 表明是否使用基于类的代理,false--> 使用面向接口的代理(java 动态代理).true--> 使用cglib
    boolean proxyTargetClass() default false;
    // 表明是否暴露代理(通过ThreadLocal的方式),该属性设置为true可解决目标对象内部的自我调用将无法实施切面中的增强的问题.
    boolean exposeProxy() default false;
}
```

这里通过@Import(AspectJAutoProxyRegistrar.class),导入了AspectJAutoProxyRegistrar的配置.

还是由之前的文章可知,此时会调用ConfigurationClassParser#processImports.由于AspectJAutoProxyRegistrar实现了ImportBeanDefinitionRegistrar接口,因此会加入到JdkDynamicAutoProxyConfiguration所对应的ConfigurationClass 的importBeanDefinitionRegistrars中.

最后,在ConfigurationClassBeanDefinitionReader#loadBeanDefinitionsFromRegistrars会对importBeanDefinitionRegistrars依次调用其registerBeanDefinitions 方法.

AspectJAutoProxyRegistrar#registerBeanDefinitions 代码如下:

```
public void registerBeanDefinitions(
        AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {

    AopConfigUtils.registerAspectJAnnotationAutoProxyCreatorIfNecessary(registry);

    AnnotationAttributes enableAspectJAutoProxy =
            AnnotationConfigUtils.attributesFor(importingClassMetadata, EnableAspectJAutoProxy.class);
    if (enableAspectJAutoProxy.getBoolean("proxyTargetClass")) {
        AopConfigUtils.forceAutoProxyCreatorToUseClassProxying(registry);
    }
    if (enableAspectJAutoProxy.getBoolean("exposeProxy")) {
        AopConfigUtils.forceAutoProxyCreatorToExposeProxy(registry);
    }
}
```

注册id为org.springframework.aop.config.internalAutoProxyCreator,class 为AnnotationAwareAspectJAutoProxyCreator的bean.这里经过层层调用,最终调用

AopConfigUtils#registerOrEscalateApcAsRequired方法,代码如下:

```
private static BeanDefinition registerOrEscalateApcAsRequired(Class<?> cls, B
eanDefinitionRegistry registry, Object source) {
Assert.notNull(registry, "BeanDefinitionRegistry must not be null");
// 如果已经存在了自动代理创建器且存在的自动代理创建器与现状的不一致那么需要根据优先级来判断
到底需要使用哪一个
if (registry.containsBeanDefinition(AUTO_PROXY_CREATOR_BEAN_NAME)) {
    BeanDefinition apcDefinition = registry.getBeanDefinition(AUTO_PROXY_CREA
TOR_BEAN_NAME);
    if (!cls.getName().equals(apcDefinition.getBeanClassName())) {
        int currentPriority = findPriorityForClass(apcDefinition.getBeanClass
Name());
        int requiredPriority = findPriorityForClass(cls);
        if (currentPriority < requiredPriority) {
            // 改变bean最重要的就是改变bean所对应的ClassName
            apcDefinition.setBeanClassName(cls.getName());
        }
    }
    // 如果已经存在自动代理创建器并且与将要创建的一致,那么无需再次创建
    return null;
}
RootBeanDefinition beanDefinition = new RootBeanDefinition(cls);
beanDefinition.setSource(source);
beanDefinition.getPropertyValues().add("order", Ordered.HIGHEST_PRECEDENCE);
beanDefinition.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
registry.registerBeanDefinition(AUTO_PROXY_CREATOR_BEAN_NAME, beanDefinition)
;
return beanDefinition;
}
```

1. 这里传入的cls 为AnnotationAwareAspectJAutoProxyCreator.class
2. 如果已经存在了自动代理创建器且存在的自动代理创建器与现状的不一致那么需要根据优先级来判断到底需要使用哪一个–> 如果此时传入的AnnotationAwareAspectJAutoProxyCreator的优先级比已注册的优先级高,则替换为AnnotationAwareAspectJAutoProxyCreator.然后直接return
3. 如果已经存在自动代理创建器并且与将要创建(AnnotationAwareAspectJAutoProxyCreator)的一致,那么无需再次创建
4. 进行注册,id为org.springframework.aop.config.internalAutoProxyCreator,class 为AnnotationAwareAspectJAutoProxyCreator,优先级为Integer.MIN_VALUE.角色为内部使用

如果配置了proxyTargetClass等于true,则对id为org.springframework.aop.config.internalAutoProxyCreator的BeanDefinition 添加proxyTargetClass 的属性,值为true.代码如下:

```
public static void forceAutoProxyCreatorToUseClassProxying(BeanDefinitionRegi
stry registry) {
    if (registry.containsBeanDefinition(AUTO_PROXY_CREATOR_BEAN_NAME)) {
        BeanDefinition definition = registry.getBeanDefinition(AUTO_PROXY_CRE
ATOR_BEAN_NAME);
        definition.getPropertyValues().add("proxyTargetClass", Boolean.TRUE);
    }
}
```

如果配置了exposeProxy等于true,则对id为org.springframework.aop.config.internalAutoProxyCreator的BeanDefinition 添加exposeProxy的属性,值为true. 代码如下:

```
public static void forceAutoProxyCreatorToExposeProxy(BeanDefinitionRegistry
registry) {
if (registry.containsBeanDefinition(AUTO_PROXY_CREATOR_BEAN_NAME)) {
    BeanDefinition definition = registry.getBeanDefinition(AUTO_PROXY_CREATOR
_BEAN_NAME);
    definition.getPropertyValues().add("exposeProxy", Boolean.TRUE);
}
}
```

AOP实战

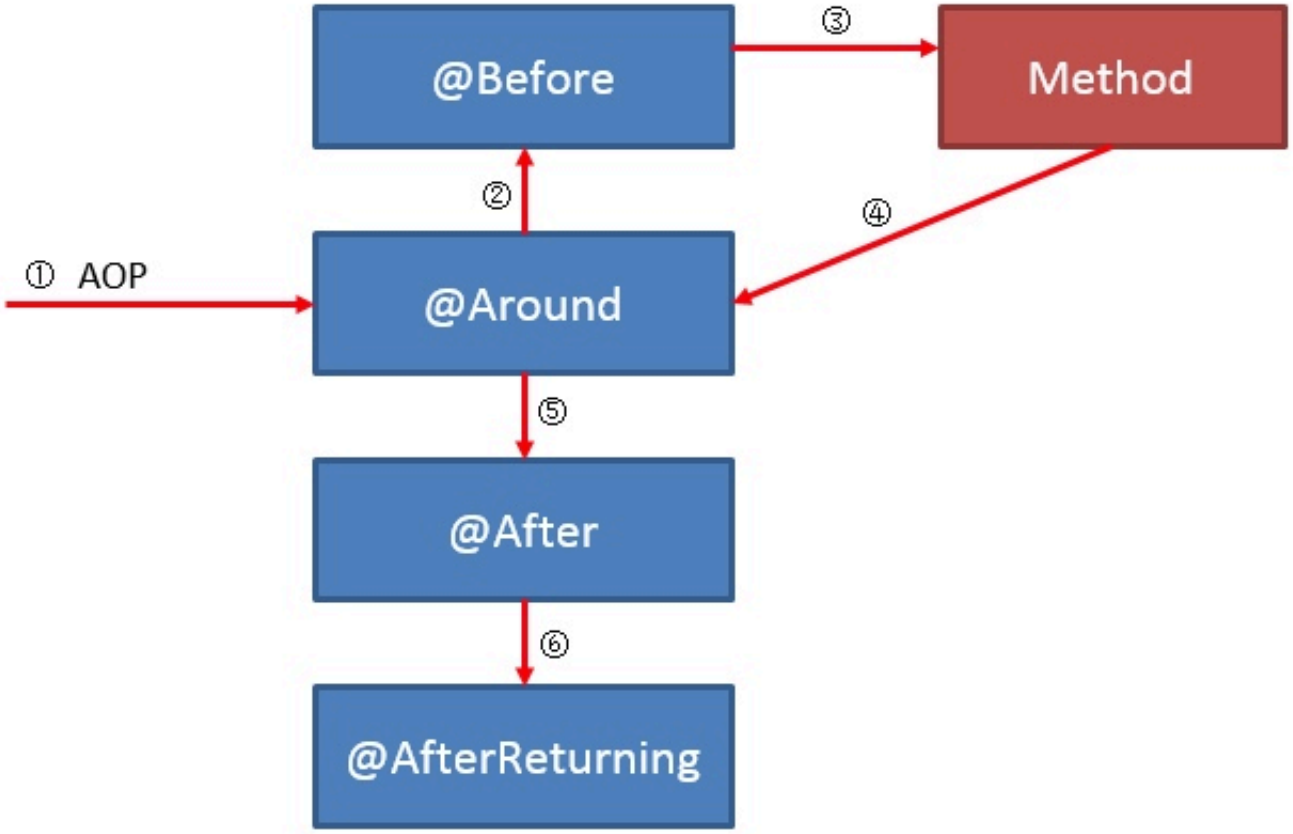
AOP基础知识

Spring 中声明 AspectJ 切面, 只需要在 IOC 容器中将切面声明为 Bean 实例. 当在 Spring IOC 容器中初始化 AspectJ 切面之后, Spring IOC 容器就会为那些与 AspectJ 切面相匹配的 Bean 创建代理。
在切面类中需要定义切面方法用于响应响应的目标方法，切面方法即为通知方法，通知方法需要用注解标识，AspectJ 支持 5 种类型的通知注解:

@Before: 前置通知, 在方法执行之前执行
@After: 后置通知, 在方法执行之后执行 。

@AfterRunning: 返回通知, 在方法返回结果之后执行
@AfterThrowing: 异常通知, 在方法抛出异常之后
@Around: 环绕通知, 围绕着方法执行

单切面




```
public interface Person {
    String sayHello(String name);

    void eat(String food);
}
```

```
@Component
public class Chinese implements Person {
    @Timer
    @Override
    public String sayHello(String name) {
        System.out.println("-- sayHello() --");
        return name + " hello, AOP";
    }
```

```
    @Override
    public void eat(String food) {
        System.out.println("我正在吃: " + food);
    // sayHello("hello");
    }
}
```java
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Timer {
}
```

```
@SpringBootApplication
@RestController
public class SpringBootDemoApplication {

    //这里必须使用Person接口做注入
    @Autowired
    private Person chinese;

    @RequestMapping("/test")
    public void test() {
        chinese.sayHello("listen");
        System.out.println(chinese.getClass());
    }

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

```
@Aspect
@Component
public class AdviceTest {
    @Pointcut("@annotation(com.haocdp.coding.spring.Timer)")
    public void pointcut() {
    }

    @Before("pointcut()")
    public void before() {
        System.out.println("before");
    }
```
```

```
    @After("pointcut()")
    public void after() {
        System.out.println("after");
    }

    @AfterReturning(pointcut = "pointcut()")
    public void afterReturning() {
        System.out.println("after return");
    }

    @AfterThrowing(pointcut = "pointcut()")
    public void afterThrowing() {
        System.out.println("after throwing");
    }

    @Around("pointcut()")
    public void around(ProceedingJoinPoint pjp) throws Throwable {
        System.out.println("before around");
        pjp.proceed();
        System.out.println("after around");
    }
}
```

执行结果

```
before around
before
-- sayHello() --
after around
after
after return
```

多切面

spring aop就是一个同心圆，要执行的方法为圆心，最外层的order最小。从最外层按照AOP1、AOP2的顺序依次执行doAround方法，doBefore方法。然后执行method方法，最后按照AOP2、AOP1的顺序依次执行doAfter、doAfterReturn方法。也就是说对多个AOP来说，先before的，一定后after。

如果我们要在同一个方法事务提交后执行自己的AOP，那么把事务的AOP order设置为2，自己的AOP order设置为1，然后在doAfterReturn里边处理自己的业务逻辑。

```
@Order(1)
@Aspect
@Component
public class AdviceTest2 {
    @Pointcut("@annotation(com.haocdp.coding.spring.Timer)")
    public void pointcut() {
    }

    @Before("pointcut()")
    public void before() {
        System.out.println("before-2");
    }

    @After("pointcut()")
    public void after() {
        System.out.println("after-2");
    }

    @AfterReturning(pointcut = "pointcut()")
    public void afterReturning() {
        System.out.println("after return -2");
    }

    @AfterThrowing(pointcut = "pointcut()")
    public void afterThrowing() {
        System.out.println("after throwing -2");
    }

    @Around("pointcut()")
    public void around(ProceedingJoinPoint pjp) throws Throwable {
        System.out.println("before around -2");
        pjp.proceed();
        System.out.println("after around -2");
    }
}
```

```
@Order(2)
@Aspect
@Component
public class AdviceTest {
    @Pointcut("@annotation(com.haocdp.coding.spring.Timer)")
    public void pointcut() {
```

```
    }

    @Before("pointcut()")
    public void before() {
        System.out.println("before");
    }

    @After("pointcut()")
    public void after() {
        System.out.println("after");
    }

    @AfterReturning(pointcut = "pointcut()")
    public void afterReturning() {
        System.out.println("after return");
    }

    @AfterThrowing(pointcut = "pointcut()")
    public void afterThrowing() {
        System.out.println("after throwing");
    }

    @Around("pointcut()")
    public void around(ProceedingJoinPoint pjp) throws Throwable {
        System.out.println("before around");
        pjp.proceed();
        System.out.println("after around");
    }
}
```

执行结果就一直会是：

```
before around -2
before-2
before around
before
-- sayHello() --
after around
after
after return
after around -2
after-2
after return -2
```

AOP 实现 Web 日志处理

在项目 pom.xml 文件中添加依赖：

```
<!-- aop 依赖 -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>

<!-- 用于日志切面中，以 json 格式打印出入参 -->
<dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.8.5</version>
</dependency>
```

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
@Documented
public @interface WebLog {
    /**
     * 日志描述信息
     *
     * @return
     */
    String description() default "";

}
```

①：什么时候使用该注解，我们定义为运行时；
②：注解用于什么地方，我们定义为作用于方法上；
③：注解是否将包含在 JavaDoc 中；
④：注解名为 WebLog;
⑤：定义一个属性，默认为空字符串；

配置 AOP 切面

```
@Aspect
```

```java
@Component
@Profile({"dev", "test"})
public class WebLogAspect {

    private final static Logger logger         = LoggerFactory.getLogger(WebL
ogAspect.class);
    /** 换行符 */
    private static final String LINE_SEPARATOR = System.lineSeparator();

    /** 以自定义 @WebLog 注解为切点 */
    @Pointcut("@annotation(site.exception.springbootaopwebrequest.aspect.WebL
og)")
    public void webLog() {}

    /**
     * 在切点之前织入
     * @param joinPoint
     * @throws Throwable
     */
    @Before("webLog()")
    public void doBefore(JoinPoint joinPoint) throws Throwable {
        // 开始打印请求日志
        ServletRequestAttributes attributes = (ServletRequestAttributes) Requ
estContextHolder.getRequestAttributes();
        HttpServletRequest request = attributes.getRequest();

        // 获取 @WebLog 注解的描述信息
        String methodDescription = getAspectLogDescription(joinPoint);

        // 打印请求相关参数
        logger.info("========================================== Start =======
==================================");
        // 打印请求 url
        logger.info("URL            : {}", request.getRequestURL().toString()
);
        // 打印描述信息
        logger.info("Description    : {}", methodDescription);
        // 打印 Http method
        logger.info("HTTP Method    : {}", request.getMethod());
        // 打印调用 controller 的全路径以及执行方法
        logger.info("Class Method   : {}.{}", joinPoint.getSignature().getDec
laringTypeName(), joinPoint.getSignature().getName());
        // 打印请求的 IP
```

```
        logger.info("IP             : {}", request.getRemoteAddr());
        // 打印请求入参
        logger.info("Request Args   : {}", new Gson().toJson(joinPoint.getArg
s()));
    }

    /**
     * 在切点之后织入
     * @throws Throwable
     */
    @After("webLog()")
    public void doAfter() throws Throwable {
        // 接口结束后换行，方便分割查看
        logger.info("========================================== End ========
=================================" + LINE_SEPARATOR);
    }

    /**
     * 环绕
     * @param proceedingJoinPoint
     * @return
     * @throws Throwable
     */
    @Around("webLog()")
    public Object doAround(ProceedingJoinPoint proceedingJoinPoint) throws Th
rowable {
        long startTime = System.currentTimeMillis();
        Object result = proceedingJoinPoint.proceed();
        // 打印出参
        logger.info("Response Args  : {}", new Gson().toJson(result));
        // 执行耗时
        logger.info("Time-Consuming : {} ms", System.currentTimeMillis() - st
artTime);
        return result;
    }


    /**
     * 获取切面注解的描述
     *
     * @param joinPoint 切点
     * @return 描述信息
     * @throws Exception
```

```
     */
    public String getAspectLogDescription(JoinPoint joinPoint)
            throws Exception {
        String targetName = joinPoint.getTarget().getClass().getName();
        String methodName = joinPoint.getSignature().getName();
        Object[] arguments = joinPoint.getArgs();
        Class targetClass = Class.forName(targetName);
        Method[] methods = targetClass.getMethods();
        StringBuilder description = new StringBuilder("");
        for (Method method : methods) {
            if (method.getName().equals(methodName)) {
                Class[] clazzs = method.getParameterTypes();
                if (clazzs.length == arguments.length) {
                    description.append(method.getAnnotation(WebLog.class).des
cription());
                    break;
                }
            }
        }
        return description.toString();
    }

}
```

Spring Boot 常用注解

1. @RequestMapping 注解提供路由信息。它告诉Spring任何来自"/"路径的HTTP请求都应该被映射到 home 方法。 @RestController 注解告诉Spring以字符串的形式渲染结果，并直接返回给调用者。
2. @Profiles
   Spring Profiles提供了一种隔离应用程序配置的方式，并让这些配置只能在特定的环境下生效。任何@Component或@Configuration都能被@Profile标记，从而限制加载它的时机。
3. @ResponseBody
   表示该方法的返回结果直接写入HTTP response body中
   一般在异步获取数据时使用，在使用@RequestMapping后，返回值通常解析为跳转路径，加上
   @responsebody后返回结果不会被解析为跳转路径，而是直接写入HTTP response body中。比如
   异步获取json数据，加上@responsebody后，会直接返回json数据。

4. @Component：
   泛指组件，当组件不好归类的时候，我们可以使用这个注解进行标注。一般公共的方法我会用上这个注解
5. @AutoWired
   byType方式。把配置好的Bean拿来用，完成属性、方法的组装，它可以对类成员变量、方法及构
   造函数进行标注，完成自动装配的工作。
   当加上（required=false）时，就算找不到bean也不报错。
6. @RequestParam：用在方法的参数前面。
   @RequestParam String a =request.getParameter("a")。
7. @PathVariable:路径变量。
   RequestMapping("user/get/mac/{macAddress}")
   public String getByMacAddress(@PathVariable String macAddress){
   //do something;
   }

8. 全局处理异常的：
   @ControllerAdvice：
   包含@Component。可以被扫描到。
   统一处理异常。
   @ExceptionHandler（Exception.class）：
   用在方法上面表示遇到这个异常就执行以下方法。

```
/**
* 全局异常处理
*/
@ControllerAdvice
class GlobalDefaultExceptionHandler {
public static final String DEFAULT_ERROR_VIEW = "error";
@ExceptionHandler({TypeMismatchException.class,NumberFormatException.clas
s})
public ModelAndView formatErrorHandler(HttpServletRequest req, Exception
e) throws Exception {
    ModelAndView mav = new ModelAndView();
    mav.addObject("error","参数类型错误");
    mav.addObject("exception", e);
    mav.addObject("url", RequestUtils.getCompleteRequestUrl(req));
    mav.addObject("timestamp", new Date());
    mav.setViewName(DEFAULT_ERROR_VIEW);
```

```
    return mav;
}}
```

9. 通过@value注解来读取application.properties里面的配置

10. @Configuration标注在类上，相当于把该类作为spring的xml配置文件中的，作用为：配置spring容器(应用上下文)

11. @Bean标注在方法上(返回某个实例的方法)，等价于spring的xml配置文件中的，作用为：注册bean对象
    注：
    (1)、@Bean注解在返回实例的方法上，如果未通过@Bean指定bean的名称，则默认与标注的方法名相同；
    (2)、@Bean注解默认作用域为单例singleton作用域，可通过@Scope(“prototype”)设置为原型作用域；

```
@Configuration
public class TestConfiguration {
    public TestConfiguration(){
        System.out.println("spring容器启动初始化。。。");
    }
//@Bean注解注册bean,同时可以指定初始化和销毁方法
//@Bean(name="testNean",initMethod="start",destroyMethod="cleanUp")
@Bean
@Scope("prototype")
public TestBean testBean() {
    return new TestBean();
}
}
```

```
public class TestMain {
public static void main(String[] args) {
    ApplicationContext context = new AnnotationConfigApplicationContext(T
estConfiguration.class);
    //获取bean
    TestBean tb = context.getBean("testBean");
    tb.sayHello();
}
}
```