

SpringBoot (<http://39.106.3.150/tags/#springboot>)

Spring Boot源码框架解析

Posted by Palmer on 06-01, 2019

Spring Boot是由Pivotal团队提供的全新框架，其设计目的是用来简化Spring应用的初始搭建以及开发过程。该框架使用了特定的方式来进行配置，从而使开发人员不再需要定义样板化的配置。随着基于Docker容器企业微服务架构的概念火热，在Spring3.x之前的大量的xml配置越来越让人诟病，在企业需要快速开发时，往往先面临着大量的xml文件的配置，项目团队面临巨大的技术挑战，大量繁琐的文件配置工作，复杂的环境部署工作。Spring Boot 的出现，即是来解决这些问题。Spring Boot最大的特点就是当下所倡导的一种理念“习惯优于配置”Spring Boot它其实并没有用到特别的技术，而是在项目中预先进行了许多习惯性的配置。无代码生成并且项目可以没有一个xml配置，按需自动装配，全由注解来完成这一切。其本身就提供准企业级开发的配置提过基于http ssh telnet对项目运行时的监控。内嵌Servlet容器，开发者开发完成将项目打包后，可以直接java -jar xx.jar运行程序，极大的方便了项目的部署提供许多容器接口支持。





1、 Auto-Configuration

@ComponentScan: 指定扫描的目录，实例化@Component、@Controller、@Repository、@Service ,相当于Spring XML配置文件中的: context:component-scan (context:component-scan)可以使用basePackages属性指定要扫描的包，以及扫描的条件。如果不设置的话默认扫描@ComponentScan注解所在类的同级类和同级目录下的所有类，所以对于一个Spring Boot项目，一般会把入口类放在顶层目录中，这样就能够保证源码目录下的所有类都能够被扫描到。

@Configuration: 标注此注解的类，就相当于替换xml文件的配置

@EnableAutoConfiguration: 作用是扫描META-INF/spring.factories文件中定义的类，使上@Configuration注解起作用

@SpringBootApplication: 集合了以上3个注解功能

```
@Retention(RetentionPolicy.RUNTIME) // 注解的生命周期，保留到class文件中（三个生命周期）
@Documented // 表明这个注解应该被javadoc记录
@Inherited // 子类可以继承该注解
@SpringBootConfiguration // 继承了Configuration，表示当前是注解类
@EnableAutoConfiguration // 开启springboot的注解功能，springboot的四大神器之一
@ComponentScan(excludeFilters = { // 扫描路径设置（具体使用待确认）
    @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
    @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class)
})
public @interface SpringBootApplication {
    ...
}
```

条件判断注解：

@ConditionalOnBean：仅仅在当前上下文中存在某个对象时，才会实例化一个Bean

@ConditionalOnClass：某个class位于类路径上，才会实例化一个Bean

@ConditionalOnExpression：当表达式为true的时候，才会实例化一个Bean

@ConditionalOnMissingBean：仅仅在当前上下文中不存在某个对象时，才会实例化一个Bean

@ConditionalOnMissingClass：某个class类路径上不存在的时候，才会实例化一个Bean

@ConditionalOnNotWebApplication：不是web应用 @ConditionalOnClass：该注解的参数对应的类必须存在，否则不解析该注解修饰的配置类 @ConditionalOnMissingBean：如果存在它修饰的类的bean，则不需要再创建这个bean

全由注解来完成装配

```
@EnableConfigurationProperties({SchedulerProperties.class})
// 该配置项不为空, 当前类 的内部 自动装载 才会生效
@ConditionalOnProperty(name = "framework.scheduler.cron-trigger.job-detail.target-obj")
// 在BatchAutoConfiguration类之后执行 自动装配
@AutoConfigureAfter({BatchAutoConfiguration.class})
public class SchedulerAutoConfiguration {
    .....// 类的实现代码
}
```

//xml配置方式

```
<bean id="myBatisDao"
class="com.onlyou.framework.mybatis.dao.impl.MyBatisDaoExtImpl">
<property name="sqlSessionTemplate" ref="sqlSessionTemplate"/>
</bean>
```

//对应的 自动装载 代码

```
@Bean
// 条件判断 @ConditionalOnMissingBean
// 只有在Spring上下文中, 没有MyBatisDao类型的实例, 才进行以下实例化
@ConditionalOnMissingBean
public MyBatisDao myBatisDao(SqlSessionTemplate sqlSessionTemplate) {
    MyBatisDaoImpl myBatisDao = new MyBatisDaoImpl();
    myBatisDao.setSqlSessionTemplate(sqlSessionTemplate);
    return myBatisDao;
}
```

内嵌Servlet容器

```
//Spring Boot官方提供（排除不使用）
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-tomcat</artifactId>
</dependency>
//framework-boot-starter-web 使用 undertow 容器
//重写并扩展对tld,jsp的支持
//web工程直接依赖
<dependency>
<groupId>com.test</groupId>
<artifactId>framework-boot-starter-web</artifactId>
</dependency>
```

2、Starter

定义spring.factories，用于指定Auto-Configuration类

定义spring-configuration-metadata.json，支持application.properties或application.yml开发配置
应用启动时Spring Boot框架加载当前应用依赖所有jar文件内部的/META-INF/spring.factories，此文件指定了当前jar带有@Configuration的类，按照类的内部编码进行自动配置。（Auto-Configuration 只是对此标记@Configuration类的别称，Starter也是。并不存在Auto-Configuration 抽象类、接口类，也不强制 *-starter.jar 这样的命名）

```
java -jar -Dspring.profiles.active=test -XX:MaxPermSize=256m -Xmx512m service.jar
#动态更换参数
java -jar -Dspring.profiles.active=test -Dframework.dubbo.protocol.port=28888 -XX...
java -jar -Dspring.profiles.active=test -Dframework.email.host=192.168.1.1 -XX...
#Web应用
java -jar -Dspring.profiles.active=test -XX:MaxPermSize=256m -Xmx512m web.jar
#动态更换参数
java -jar -Dspring.profiles.active=test -Dserver.port=8081 -XX...
java -jar -Dspring.profiles.active=test -Dframework.cache.address1=192.168.1.1:11211.
```

3、Cli

命令行工具，可用于快速搭建基于Spring的原型/支持运行Groovy脚本

4、Actuator

提供对应用的自省和监控

Maven依赖

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

主要暴露的功能

HTTP方法	路径	描述	鉴权
GET	/autoconfig	查看自动配置的使用情况	true
GET	/configprops	查看配置属性，包括默认配置	true
GET	/beans	查看bean及其关系列表	true
GET	/dump	打印线程栈	true
GET	/env	查看所有环境变量	true
GET	/env/{name}	查看具体变量值	true
GET	/health	查看应用健康指标	false
GET	/info	查看应用信息	false
GET	/mappings	查看所有url映射	true
GET	/metrics	查看应用基本指标	true
GET	/metrics/{name}	查看具体指标	true
POST	/shutdown	关闭应用	true
GET	/trace	查看基本追踪信息	true

源码解析入口类

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication {

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

以上的代码就是通过Spring Initializr配置生成的一个最简单的Web项目(只引入了Web功能)的入口方法。

一、SpringApplication的实例化

介绍完了入口类，下面开始分析关键方法：

SpringApplication.run(SpringbootApplication.class, args);
相应实现:

```
// 参数对应的就是DemoApplication.class以及main方法中的args
public static ConfigurableApplicationContext run(Class<?> primarySource,
        String... args) {
    return run(new Class<?>[] { primarySource }, args);
}
// 最终运行的这个重载方法
public static ConfigurableApplicationContext run(Class<?>[] primarySources,
        String[] args) {
    return new SpringApplication(primarySources).run(args);
}
// 实际会构造一个SpringApplication的实例, 然后运行它的run方法
public SpringApplication(ResourceLoader resourceLoader, Class<?>... primarySources) {
    this.resourceLoader = resourceLoader;
    Assert.notNull(primarySources, "PrimarySources must not be null");
    this.primarySources = new LinkedHashSet<>(Arrays.asList(primarySources));
    this.webApplicationType = WebApplicationType.deduceFromClasspath();
    setInitializers((Collection) getSpringFactoriesInstances(
        ApplicationContextInitializer.class));
    setListeners((Collection) getSpringFactoriesInstances(ApplicationListener.class));
    this.mainApplicationClass = deduceMainApplicationClass();
}
```

在构造函数中, 主要做了4件事情:

1、推断应用类型是standard还是web

```
static WebApplicationType deduceFromClasspath() {
    if (ClassUtils.isPresent(WEBFLUX_INDICATOR_CLASS, null)
        && !ClassUtils.isPresent(WEBMVC_INDICATOR_CLASS, null)
        && !ClassUtils.isPresent(JERSEY_INDICATOR_CLASS, null)) {
        return WebApplicationType.REACTIVE;
    }
    for (String className : SERVLET_INDICATOR_CLASSES) {
        if (!ClassUtils.isPresent(className, null)) {
            return WebApplicationType.NONE;
        }
    }
    return WebApplicationType.SERVLET;
}
```

// 相关常量

```
private static final String WEBMVC_INDICATOR_CLASS = "org.springframework." + "web.ser
private static final String WEBFLUX_INDICATOR_CLASS = "org." + "springframework.web.re
private static final String JERSEY_INDICATOR_CLASS = "org.glassfish.jersey.servlet.Se
```

可能会出现三种结果： `WebApplicationType.REACTIVE` - 当类路径中存在 `REACTIVE_WEB_ENVIRONMENT_CLASS`并且不存在`MVC_WEB_ENVIRONMENT_CLASS`时 `WebApplicationType.NONE` - 也就是非Web型应用(Standard型)， 此时类路径中不包含 `WEB_ENVIRONMENT_CLASSES`中定义的任何一个类时 `WebApplicationType.SERVLET` - 类路径中包含了`WEB_ENVIRONMENT_CLASSES`中定义的所有类型时

2、设置初始化器

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

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 = getClassLoader();
    // Use names and ensure unique to protect against duplicates
    // 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;
}
```

这里面首先会根据入参type读取所有的names(是一个String集合)，然后根据这个集合来完成对应的实例化操作：

```

public static List<String> loadFactoryNames(Class<?> factoryClass, @Nullable ClassLoader
    String factoryClassName = factoryClass.getName();
    return loadSpringFactories(classLoader).getOrDefault(factoryClassName, Collections
}

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

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

    try {
        Enumeration<URL> urls = (classLoader != null ?
            classLoader.getResources(FACTORIES_RESOURCE_LOCATION) :
            ClassLoader.getSystemResources(FACTORIES_RESOURCE_LOCATION));
        result = new LinkedMultiValueMap<>();
        while (urls.hasMoreElements()) {
            URL url = urls.nextElement();
            UrlResource resource = new UrlResource(url);
            Properties properties = PropertiesLoaderUtils.loadProperties(resource);
            for (Map.Entry<?, ?> entry : properties.entrySet()) {
                String factoryClassName = ((String) entry.getKey()).trim();
                for (String factoryName : StringUtils.commaDelimitedListToStringArray((String)
                    result.add(factoryClassName, factoryName.trim());
            }
        }
        cache.put(classLoader, result);
        return result;
    }
    catch (IOException ex) {
        throw new IllegalArgumentException("Unable to load factories from location [" +
            FACTORIES_RESOURCE_LOCATION + "]", ex);
    }
}

```

这个方法会尝试从类路径的META-INF/spring.factories处读取相应配置文件，然后进行遍历，读取配置文件中Key为：org.springframework.context.ApplicationContextInitializer的value。以spring-boot-autoconfigure这个包为例，它的META-INF/spring.factories部分定义如下所示：

```

#Initializers
org.springframework.context.ApplicationContextInitializer=\
org.springframework.boot.autoconfigure.SharedMetadataReaderFactoryContextInitializer,
org.springframework.boot.autoconfigure.logging.ConditionEvaluationReportLoggingListener

```

因此这两个类名会被读取出来，然后放入到集合中，准备开始下面的实例化操作：


```
// 关键参数:
// type: org.springframework.context.ApplicationContextInitializer.class
// names: 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<>(names.size());
    for (String name : names) {
        try {
            Class<?> instanceClass = ClassUtils.forName(name, classLoader);
            Assert.isAssignable(type, instanceClass);
            Constructor<?> constructor = instanceClass
                .getDeclaredConstructor(parameterTypes);
            T instance = (T) BeanUtils.instantiateClass(constructor, args);
            instances.add(instance);
        }
        catch (Throwable ex) {
            throw new IllegalArgumentException(
                "Cannot instantiate " + type + " : " + name, ex);
        }
    }
    return instances;
}
```

初始化步骤很直观，没什么好说的，类加载，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 ConfigurableApplicationConte

/**
 * Initialize the given application context.
 * @param applicationContext the application to configure
 */
void initialize(C applicationContext);

}
```

根据类文档，这个接口的主要功能是：

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。

3、设置监听器

设置完了初始化器，下面开始设置监听器：

```

public SpringApplication(ResourceLoader resourceLoader, Class<?>... primarySources) {
    this.resourceLoader = resourceLoader;
    Assert.notNull(primarySources, "PrimarySources must not be null");
    this.primarySources = new LinkedHashSet<>(Arrays.asList(primarySources));
    this.webApplicationType = WebApplicationType.deduceFromClasspath();
    setInitializers((Collection) getSpringFactoriesInstances(
        ApplicationContextInitializer.class));
    setListeners((Collection) getSpringFactoriesInstances(ApplicationListener.class));
    this.mainApplicationClass = deduceMainApplicationClass();
}

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

private <T> Collection<T> getSpringFactoriesInstances(Class<T> type,
    Class<?>[] parameterTypes, Object... args) {
    ClassLoader classLoader = getClassLoader();
    // Use names and ensure unique to protect against duplicates
    Set<String> names = new LinkedHashSet<>(
        SpringFactoriesLoader.loadFactoryNames(type, classLoader));
    List<T> instances = createSpringFactoriesInstances(type, parameterTypes,
        classLoader, args, names);
    AnnotationAwareOrderComparator.sort(instances);
    return instances;
}

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

public static List<String> loadFactoryNames(Class<?> factoryClass, @Nullable ClassLoader classLoader) {
    String factoryClassName = factoryClass.getName();
    return loadSpringFactories(classLoader).getOrDefault(factoryClassName, Collections.emptyList());
}

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

    try {
        Enumeration<URL> urls = (classLoader != null ?
            classLoader.getResources(FACTORIES_RESOURCE_LOCATION) :
            ClassLoader.getSystemResources(FACTORIES_RESOURCE_LOCATION));
        result = new LinkedMultiValueMap<>();
        while (urls.hasMoreElements()) {
            URL url = urls.nextElement();
            UrlResource resource = new UrlResource(url);
            Properties properties = PropertiesLoaderUtils.loadProperties(resource);
            for (Map.Entry<?, ?> entry : properties.entrySet()) {
                String factoryClassName = ((String) entry.getKey()).trim();
                for (String factoryName : StringUtils.commaDelimitedListToStringArray((String) entry.getValue().trim())) {
                    result.add(factoryClassName, factoryName.trim());
                }
            }
        }
    } catch (IOException ex) {
        // ignore
    }
}

```

```

        cache.put(classLoader, result);
        return result;
    }
    catch (IOException ex) {
        throw new IllegalArgumentException("Unable to load factories from location [" +
            FACTORIES_RESOURCE_LOCATION + "]", ex);
    }
}

```

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

```

这个接口基于JDK中的EventListener接口，实现了观察者模式。对于Spring框架的观察者模式实现，它限定感兴趣的事件类型需要是ApplicationEvent类型的子类，而这个类同样是继承自JDK中的EventObject类。

```

@FunctionalInterface
public interface ApplicationListener<E extends ApplicationEvent> extends EventListener {

    /**
     * Handle an application event.
     * @param event the event to respond to
     */
    void onApplicationEvent(E event);

}

```

4、推断应用入口类

```

public SpringApplication(ResourceLoader resourceLoader, Class<?>... primarySources) {
    this.resourceLoader = resourceLoader;
    Assert.notNull(primarySources, "PrimarySources must not be null");
    this.primarySources = new LinkedHashSet<>(Arrays.asList(primarySources));
    this.webApplicationType = WebApplicationType.deduceFromClasspath();
    setInitializers((Collection) getSpringFactoriesInstances(
        ApplicationContextInitializer.class));
    setListeners((Collection) getSpringFactoriesInstances(ApplicationListener.class));
    this.mainApplicationClass = deduceMainApplicationClass();
}

```

它通过构造一个运行时异常，通过异常栈中方法名为main的栈帧来得到入口类的名字。

```
private Class<?> deduceMainApplicationClass() {  
    try {  
        StackTraceElement[] stackTrace = new RuntimeException().getStackTrace();  
        for (StackTraceElement stackTraceElement : stackTrace) {  
            if ("main".equals(stackTraceElement.getMethodName())) {  
                return Class.forName(stackTraceElement.getClassName());  
            }  
        }  
    }  
    catch (ClassNotFoundException ex) {  
        // Swallow and continue  
    }  
    return null;  
}
```

至此，对于SpringApplication实例的初始化过程就结束了。

二、SpringApplication的Run方法

完成了实例化，下面开始调用run方法：

```

// 运行run方法
public ConfigurableApplicationContext run(String... args) {
    // 计时工具
    Stopwatch stopWatch = new Stopwatch();
    stopWatch.start();
    ConfigurableApplicationContext context = null;
    Collection<SpringBootExceptionHandler> exceptionReporters = new ArrayList<>();
    // 设置java.awt.headless系统属性为true - 没有图形化界面
    configureHeadlessProperty();

    // KEY 1 - 获取SpringApplicationRunListeners
    SpringApplicationRunListeners listeners = getRunListeners(args);

    // 发出开始执行的事件
    listeners.starting();
    try {
        ApplicationArguments applicationArguments = new DefaultApplicationArguments(
            args);

        // KEY 2 - 根据SpringApplicationRunListeners以及参数来准备环境
        ConfigurableEnvironment environment = prepareEnvironment(listeners,
            applicationArguments);
        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);
        // KEY 3 - 创建Spring上下文
        context = createApplicationContext();
        // 准备异常报告器
        exceptionReporters = getSpringFactoriesInstances(
            SpringBootExceptionHandler.class,
            new Class[] { ConfigurableApplicationContext.class }, context);
        // KEY 4 - Spring上下文前置处理
        prepareContext(context, environment, listeners, applicationArguments,
            printedBanner);
        // KEY 5 - Spring上下文刷新
        refreshContext(context);
        // KEY 6 - Spring上下文后置处理
        afterRefresh(context, applicationArguments);
        // 发出结束执行的事件
        stopWatch.stop();
        // 停止计时器
        if (this.logStartupInfo) {
            new StartupInfoLogger(this.mainApplicationClass)
                .logStarted(getApplicationLog(), stopWatch);
        }
        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) {

```

```

catch (Throwable ex) {
    handleRunFailure(context, ex, exceptionReporters, null);
    throw new IllegalStateException(ex);
}
return context;
}

```

这个run方法包含的内容也是有点多的，根据上面列举出的关键步骤逐个进行分析：

1、KEY 1 - 获取SpringApplicationRunListeners

```

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

// 这里的入参:
// type: SpringApplicationRunListener.class
// parameterTypes: new Class<?>[] { SpringApplication.class, String[].class };
// args: SpringApplication实例本身 + main方法传入的args
private <T> Collection<T> getSpringFactoriesInstances(Class<T> type,
    Class<?>[] parameterTypes, Object... args) {
    ClassLoader classLoader = getClassLoader();
    // Use names and ensure unique to protect against duplicates
    Set<String> names = new LinkedHashSet<>()
        SpringApplicationFactoriesLoader.loadFactoryNames(type, classLoader));
    List<T> instances = createSpringFactoriesInstances(type, parameterTypes,
        classLoader, args, names);
    AnnotationAwareOrderComparator.sort(instances);
    return instances;
}

```

所以这里还是故技重施，从META-INF/spring.factories中读取Key为org.springframework.boot.SpringApplicationRunListener的Values：比如在spring-boot包中定义的spring.factories：

```
# Run Listeners
org.springframework.boot.SpringApplicationRunListener=\
org.springframework.boot.context.event.EventPublishingRunListener

/**
 * {@link SpringApplicationRunListener} to publish {@link SpringApplicationEvent}s.
 * <p>
 * Uses an internal {@link ApplicationEventMulticaster} for the events that are fired
 * before the context is actually refreshed.
 *
 * @author Phillip Webb
 * @author Stephane Nicoll
 */
public class EventPublishingRunListener implements SpringApplicationRunListener, Order
    // ...
}
```

从类文档可以看出，它主要是负责发布SpringApplicationEvent事件的，它会利用一个内部的ApplicationEventMulticaster在上下文实际被刷新之前对事件进行处理。

2、KEY 2 - 根据SpringApplicationRunListeners以及参数来准备环境

```
private ConfigurableEnvironment prepareEnvironment(
    SpringApplicationRunListeners listeners,
    ApplicationArguments applicationArguments) {
    // Create and configure the environment
    ConfigurableEnvironment environment = getOrCreateEnvironment();
    configureEnvironment(environment, applicationArguments.getSourceArgs());
    listeners.environmentPrepared(environment);
    bindToSpringApplication(environment);
    if (!this.isCustomEnvironment) {
        environment = new EnvironmentConverter(getClassLoader())
            .convertEnvironmentIfNecessary(environment, deduceEnvironmentClass());
    }
    ConfigurationPropertySources.attach(environment);
    return environment;
}
```

配置环境的方法：


```

protected void configureEnvironment(ConfigurableEnvironment environment,
    String[] args) {
    if (this.addConversionService) {
        ConversionService conversionService = ApplicationConversionService
            .getSharedInstance();
        environment.setConversionService(
            (ConfigurableConversionService) conversionService);
    }
    configurePropertySources(environment, args);
    configureProfiles(environment, args);
}

```

所以这里实际上也包含了两个步骤：配置Property Sources 配置Profiles 对于Web应用而言，得到的environment变量是一个StandardServletEnvironment的实例。得到实例后，会调用前面RunListeners中的environmentPrepared方法：

```

private ConfigurableEnvironment prepareEnvironment(
    SpringApplicationRunListeners listeners,
    ApplicationArguments applicationArguments) {
    // Create and configure the environment
    ConfigurableEnvironment environment = getOrCreateEnvironment();
    configureEnvironment(environment, applicationArguments.getSourceArgs());
    listeners.environmentPrepared(environment);
    bindToSpringApplication(environment);
    if (!this.isCustomEnvironment) {
        environment = new EnvironmentConverter(getClassLoader())
            .convertEnvironmentIfNecessary(environment, deduceEnvironmentClass());
    }
    ConfigurationPropertySources.attach(environment);
    return environment;
}

@Override
public void environmentPrepared(ConfigurableEnvironment environment) {
    this.initialMulticaster.multicastEvent(new ApplicationEnvironmentPreparedEvent(
        this.application, this.args, environment));
}

```

在这里，定义的广播器就派上用场了，它会发布一个ApplicationEnvironmentPreparedEvent事件。

那么有发布就有监听，在构建SpringApplication实例的时候不是初始化过一些ApplicationListeners，其中的Listener就可能会监听ApplicationEnvironmentPreparedEvent事件，然后进行相应处理。

所以这里SpringApplicationRunListeners的用途和目的也比较明显了，它实际上是一个事件中转器，它能够感知到Spring Boot启动过程中产生的事件，然后有选择性的将事件进行中转。为何是有选择性的，看看它的实现就知道了：

```
@Override
public void contextPrepared(ConfigurableApplicationContext context) {
    this.initialMulticaster.multicastEvent(new ApplicationContextInitializedEvent(
        this.application, this.args, context));
}
```

它的contextPrepared方法实现为空，没有利用内部的initialMulticaster进行事件的派发。因此即便是外部有ApplicationListener对这个事件有兴趣，也是没有办法监听到的。

那么既然有事件的转发，是谁在监听这些事件呢，在这个类的构造器中交待了：

```
public EventPublishingRunListener(SpringApplication application, String[] args) {
    this.application = application;
    this.args = args;
    this.initialMulticaster = new SimpleApplicationEventMulticaster();
    for (ApplicationListener<?> listener : application.getListeners()) {
        this.initialMulticaster.addApplicationListener(listener);
    }
}
```

前面在构建SpringApplication实例过程中设置的监听器在这里被逐个添加到了initialMulticaster对应的ApplicationListener列表中。所以当initialMulticaster调用multicastEvent方法时，这些Listeners中定义的相应方法就会被触发了。

3、KEY 3 - 创建Spring上下文

```
//默认情况下将用于Web的应用程序上下文的类名称
public static final String DEFAULT_SERVLET_WEB_CONTEXT_CLASS = "org.springframework.k
//默认情况下将用于响应式Web的应用程序上下文的类名称
public static final String DEFAULT_REACTIVE_WEB_CONTEXT_CLASS = "org.springframework.
//默认情况下将用于非Web的应用程序上下文的类名称
public static final String DEFAULT_CONTEXT_CLASS = "org.springframework.context."+ "

protected ConfigurableApplicationContext createApplicationContext() {
    Class<?> contextClass = this.applicationContextClass;
    if (contextClass == null) {
        try {
            switch (this.webApplicationType) {
                case SERVLET:
                    contextClass = Class.forName(DEFAULT_SERVLET_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);
}
```

4、KEY 4 - Spring上下文前置处理

```

private void prepareContext(ConfigurableApplicationContext context,
    ConfigurableEnvironment environment, SpringApplicationRunListeners listeners,
    ApplicationArguments applicationArguments, Banner printedBanner) {
    // 将环境和上下文关联起来
    context.setEnvironment(environment);
    // 为上下文配置Bean生成器以及资源加载器(如果它们非空)
    postProcessApplicationContext(context);
    // 调用初始化器
    applyInitializers(context);
    // 触发Spring Boot启动过程的contextPrepared事件
    listeners.contextPrepared(context);
    if (this.logStartupInfo) {
        logStartupInfo(context.getParent() == null);
        logStartupProfileInfo(context);
    }
    // Add boot specific singleton beans
    ConfigurableListableBeanFactory beanFactory = context.getBeanFactory();
    // 添加两个Spring Boot中的特殊单例Beans - springApplicationArguments以及springBootBanner
    beanFactory.registerSingleton("springApplicationArguments", applicationArguments);
    if (printedBanner != null) {
        beanFactory.registerSingleton("springBootBanner", printedBanner);
    }
    if (beanFactory instanceof DefaultListableBeanFactory) {
        ((DefaultListableBeanFactory) beanFactory)
            .setAllowBeanDefinitionOverriding(this.allowBeanDefinitionOverriding);
    }
    // Load the sources
    // 加载sources - 对于DemoApplication而言, 这里的sources集合只包含了它一个class对象
    Set<Object> sources = getAllSources();
    Assert.notEmpty(sources, "Sources must not be empty");
    // 加载动作 - 构造BeanDefinitionLoader并完成Bean定义的加载
    load(context, sources.toArray(new Object[0]));
    // 触发Spring Boot启动过程的contextLoaded事件
    listeners.contextLoaded(context);
}

```

关键步骤：配置Bean生成器以及资源加载器(如果它们非空):

```
protected void postProcessApplicationContext(ConfigurableApplicationContext context)
{
    if (this.beanNameGenerator != null) {
        context.getBeanFactory().registerSingleton(
            AnnotationConfigUtils.CONFIGURATION_BEAN_NAME_GENERATOR,
            this.beanNameGenerator);
    }
    if (this.resourceLoader != null) {
        if (context instanceof GenericApplicationContext) {
            ((GenericApplicationContext) context)
                .setResourceLoader(this.resourceLoader);
        }
        if (context instanceof DefaultResourceLoader) {
            ((DefaultResourceLoader) context)
                .setClassLoader(this.resourceLoader.getClassLoader());
        }
    }
    if (this.addConversionService) {
        context.getBeanFactory().setConversionService(
            ApplicationConversionService.getSharedInstance());
    }
}
```

调用初始化器

```
protected void applyInitializers(ConfigurableApplicationContext context) {
    for (ApplicationContextInitializer initializer : getInitializers()) {
        Class<?> requiredType = GenericTypeResolver.resolveTypeArgument(
            initializer.getClass(), ApplicationContextInitializer.class);
        Assert.isInstanceOf(requiredType, context, "Unable to call initializer.");
        initializer.initialize(context);
    }
}
```

这里最终用到了在创建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。

5、Spring上下文刷新

```

private void refreshContext(ConfigurableApplicationContext context) {
    // 由于这里需要调用父类一系列的refresh操作，涉及到了很多核心操作，因此耗时会比较长，本文不做具体
    refresh(context);
    // 注册一个关闭容器时的钩子函数
    if (this.registerShutdownHook) {
        try {
            context.registerShutdownHook();
        }
        catch (AccessControlException ex) {
            // Not allowed in some environments.
        }
    }
}
// 调用父类的refresh方法完成容器刷新的基础操作
protected void refresh(ApplicationContext applicationContext) {
    Assert.isInstanceOf(AbstractApplicationContext.class, applicationContext);
    ((AbstractApplicationContext) applicationContext).refresh();
}
注册关闭容器时的钩子函数的默认实现是在AbstractApplicationContext类中：

public void registerShutdownHook() {
    if (this.shutdownHook == null) {
        // No shutdown hook registered yet.
        this.shutdownHook = new Thread() {
            @Override
            public void run() {
                synchronized (startupShutdownMonitor) {
                    doClose();
                }
            }
        };
        Runtime.getRuntime().addShutdownHook(this.shutdownHook);
    }
}

```

如果没有提供自定义的shutdownHook，那么会生成一个默认的，并添加到Runtime中。默认行为就是调用它的doClose方法，完成一些容器销毁时的清理工作。

6、Spring上下文后置处理

```

/**
 * Called after the context has been refreshed.
 * @param context the application context
 * @param args the application arguments
 */
protected void afterRefresh(ConfigurableApplicationContext context,
    ApplicationArguments args) {
    //spring boot2版本去掉了 callRunners(context, args); 调用
}

private void callRunners(ApplicationContext context, ApplicationArguments args) {
    List<Object> runners = new ArrayList<>();
    runners.addAll(context.getBeansOfType(ApplicationRunner.class).values());
    runners.addAll(context.getBeansOfType(CommandLineRunner.class).values());
    AnnotationAwareOrderComparator.sort(runners);
    for (Object runner : new LinkedHashSet<>(runners)) {
        if (runner instanceof ApplicationRunner) {
            callRunner((ApplicationRunner) runner, args);
        }
        if (runner instanceof CommandLineRunner) {
            callRunner((CommandLineRunner) runner, args);
        }
    }
}

private void callRunner(ApplicationRunner runner, ApplicationArguments args) {
    try {
        (runner).run(args);
    }
    catch (Exception ex) {
        throw new IllegalStateException("Failed to execute ApplicationRunner", ex);
    }
}

private void callRunner(CommandLineRunner runner, ApplicationArguments args) {
    try {
        (runner).run(args.getSourceArgs());
    }
    catch (Exception ex) {
        throw new IllegalStateException("Failed to execute CommandLineRunner", ex);
    }
}

```

所谓的前置操作，就是在容器完成刷新后，依次调用注册的Runners。Runners可以是两个接口的实现类：`org.springframework.boot.ApplicationRunner`
`org.springframework.boot.CommandLineRunner` 这两个接口有什么区别呢：除了接口中的run方法接受的参数类型是不一样的以外。一个是封装好的ApplicationArguments类型，另一个是直接的String不定长数组类型。因此根据需要进行选择相应的接口实现即可。

```

* Interface used to indicate that a bean should <em>run</em> when it is contained wit
* a {@link SpringApplication}. Multiple {@link ApplicationRunner} beans can be define
* within the same application context and can be ordered using the {@link Ordered}
* interface or {@link Order @Order} annotation.
*
* @author Phillip Webb
* @since 1.3.0
* @see CommandLineRunner
*/
@FunctionalInterface
public interface ApplicationRunner {

    /**
     * Callback used to run the bean.
     * @param args incoming application arguments
     * @throws Exception on error
     */
    void run(ApplicationArguments args) throws Exception;

}

/**
 * Interface used to indicate that a bean should <em>run</em> when it is contained wit
 * a {@link SpringApplication}. Multiple {@link CommandLineRunner} beans can be define
 * within the same application context and can be ordered using the {@link Ordered}
 * interface or {@link Order @Order} annotation.
 * <p>
 * If you need access to {@link ApplicationArguments} instead of the raw String array
 * consider using {@link ApplicationRunner}.
 *
 * @author Dave Syer
 * @see ApplicationRunner
 */
@FunctionalInterface
public interface CommandLineRunner {

    /**
     * Callback used to run the bean.
     * @param args incoming main method arguments
     * @throws Exception on error
     */
    void run(String... args) throws Exception;

}

```

三、总结

Spring Boot启动时的关键步骤，主要包含以下两个方面：

1、SpringApplication实例的构建过程

其中主要涉及到了初始化器(Initializer)以及监听器(Listener)这两大概念，它们都通过META-

INF/spring.factories完成定义。

2、SpringApplication实例run方法的执行过程

其中主要有一个SpringApplicationRunListeners的概念，它作为Spring Boot容器初始化时各阶段事件的中转器，将事件派发给感兴趣的Listeners(在SpringApplication实例的构建过程中得到的)。这些阶段性事件将容器的初始化过程给构造起来，提供了比较强大的可扩展性。

如果从可扩展性的角度出发，应用开发者可以在Spring Boot容器的启动阶段，扩展哪些内容呢：

- 1、初始化器(Initializer)
- 2、监听器(Listener)
- 3、容器刷新后置Runners(ApplicationRunner或者CommandLineRunner接口的实现类)
- 4、启动期间在Console打印Banner的具体实现类

因此在下一篇文章中，将介绍如何对Spring Boot容器的启动过程进行扩展，实现对该过程的定制化。

← PREVIOUS POST

([HTTP://39.106.3.150/ARCHIVES/LINUX](http://39.106.3.150/archives/linux))



...

撰写评论...



[上一页](#) [下一页](#)

FEATURED TAGS (<http://39.106.3.150/tags/>)

[java 8 \(<http://39.106.3.150/tags/#java-8>\)](http://39.106.3.150/tags/#java-8)

[java8 \(<http://39.106.3.150/tags/#java8>\)](http://39.106.3.150/tags/#java8)

[redis \(<http://39.106.3.150/tags/#redis>\)](http://39.106.3.150/tags/#redis)

[监控 \(<http://39.106.3.150/tags/#1561876610222>\)](http://39.106.3.150/tags/#1561876610222)

[全链路 \(<http://39.106.3.150/tags/#1561876610220>\)](http://39.106.3.150/tags/#1561876610220)

[容器 \(<http://39.106.3.150/tags/#1560852708518>\)](http://39.106.3.150/tags/#1560852708518)

[开源框架 \(<http://39.106.3.150/tags/#1560569459781>\)](http://39.106.3.150/tags/#1560569459781)

[Spring \(<http://39.106.3.150/tags/#spring>\)](http://39.106.3.150/tags/#spring)

[设计模式 \(<http://39.106.3.150/tags/#1559888728999>\)](http://39.106.3.150/tags/#1559888728999)

[linux \(<http://39.106.3.150/tags/#linux>\)](http://39.106.3.150/tags/#linux)

[SpringBoot \(<http://39.106.3.150/tags/#springboot>\)](http://39.106.3.150/tags/#springboot)

[大数据 \(<http://39.106.3.150/tags/#1559363598973>\)](http://39.106.3.150/tags/#1559363598973)

[区块链 \(<http://39.106.3.150/tags/#1559363594390>\)](http://39.106.3.150/tags/#1559363594390)

[Java \(<http://39.106.3.150/tags/#java>\)](http://39.106.3.150/tags/#java)

FRIENDS



(<https://github.com/PowehiEdge>)

Copyright © powehi

你的世界不止在眼前