Spring系列-1 启动流程

原创 Ewen Seong 已于 2024-06-02 14:44:39 修改 阅读量4.3k 收藏 13 点赞数 2
分类专栏： Spring系列 文章标签： spring java spring boot

背景

从本文开始，开启一个新的专题Spring系列，用于收集Spring框架相关的文章；通过使用方式、案例演示、源码分析等方式对Spring进行介绍。
该系列将包括以下文章：

1.Spring系列—启动流程
2.Spring系列—Bean的生命周期
3.Spring系列—Bean实例化与依赖注入
4.Spring系列—循环依赖与三级缓存
5.Spring系列—事件机制
6.Spring系列—占位符使用和原理
7.Spring系列—国际化
8.Spring系列—AOP原理
9.Spring系列—Async注解使用与原理
10.Spring系列—事务机制

预计每周末更新一篇，预计持续时间3个月左右。

本文介绍 Spring启动流程 ，重点在于容器的刷新过程。

1.Spring启动流程

BeanFactory提供了IOC相关的能力，称为IOC容器；SpringApplication作为BeanFactory的子类，在其基础上提供了事件机制、国际化、资源处理等功
Spring上下文或者Spring容器。
SpringApplication的核心实现在AbstractSpringApplication类中，Spring启动流程也是在该类的refresh()方法中完成。AbstractSpringApplication类在内
个BeanFactory对象(默认为DefaultListableBeanFactory类型)；容器相关的所有功能由该BeanFactory对象提供，而AbstractSpringApplication在此基础
一层代理封装。
相对于BeanFactory的懒加载机制，Spring容器在启动过程中会将所有的非lazy类型的Bean对象加载到IOC容器中。
Spring启动流程可以看成Spring容器组件初始化、向IOC容器注册Bean对象以及对需要AOP的对象完成代理的过程；其中，组件初始化包括IOC容器的
事件组件、国际化组件等。

2.使用方式

> 如果准备开始阅读源码，第一件事应该写一个demo

2.1 xml 配置文件方式

spring-context.xml 配置文件如下：

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
         http://www.springframework.org/schema/context
         https://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="com.seong.xml.componentscan"/>

    <bean id="componentA" class="com.seong.xml.component.ComponentA"/>
</beans>
```

如上所示：可以通过bean标签进行Bean对象的声明；也可以通过component-scan进行扫描，要求被扫描的对象为Component(至少被@Component注
ComponentA和ComponentB为普通POJO(这里

测试用例如下：

```
@Test
public void testXml(){
    AbstractApplicationContext context = new ClassPathXmlApplicationContext("spring-context.xml");
    final String[] beanDefinitionNames = context.getBeanDefinitionNames();
    Arrays.stream(beanDefinitionNames).forEach(beanDefinitionName -> log.info("name contains {}.", beanDefinitionName));
}
```

2.2 配置类方式

```
@Configuration
@Import(ComponentA.class)
@ComponentScan(basePackages = "com.seong.annotation.componentscan")
public class ConfigurationA {
    @Bean
    public ComponentB beanMethodE() {
        return new ComponentB();
    }
}
```

如上所示，在使用@Configuration注解配置类，可以在配置类中可以通过@Import导入Bean定义，也可以通过@ComponentScan注解进行类路径的扫描。

测试用例如下：

```
@Test
public void annotationConfig(){
    AbstractApplicationContext context = new AnnotationConfigApplicationContext(ConfigurationA.class);
    final String[] beanDefinitionNames = context.getBeanDefinitionNames();
    Arrays.stream(beanDefinitionNames).forEach(beanDefinitionName -> log.info("name contains {}.", beanDefinitionName));
}
```

3.实现原理

章节2 中的AnnotationConfigApplicationContext或者ClassPathXmlApplicationContext都是AbstractApplicationContext的子类，而 Spring框架启动 时辑在于AbstractApplicationContext中的refresh()方法；AnnotationConfigApplicationContext、ClassPathXmlApplicationContext以refresh()方法为基础板方法进行类一些扩展。

3.1 属性介绍

在进入refresh()方法前，对涉及的重要属性进行简要说明：
在AbstractApplicationContext对象中：

> (1) private ConfigurableEnvironment environment
> 存储上下文关联的环境变量

> (2) private final List<BeanFactoryPostProcessor> beanFactoryPostProcessors
> 存储全局的beanFactoryPostProcessor对象，在容器启动之初便利调用其postProcessBeanFactory方法

> (3) private ResourcePatternResolver resourcePatternResolver
> 资源解析器，从类路径下读取资源文件进入内存

> (4) private MessageSource messageSource
> 国际化组件，国际化功能依赖于该对象实现

> (5) private ApplicationEventMulticaster applicationEventMulticaster
> 事件广播器，事件能力依赖该对象进行

> (6) Set<ApplicationListener<?>> applicationL
> 监听器收集对象，用于收集容器启动过程中的

> (7) Set earlyApplicationEvents;
> 收集事件对象，用于收集容器启动过程中触发的事件，当事件广播器初始化后立刻触发

在AbstractBeanFactory对象中：

> (8) private final List beanPostProcessors = new BeanPostProcessorCacheAwareList();
> 存储全局的beanPostProcessor对象，在Bean对象初始化阶段依次调用其postProcessBeforeInitialization和postProcessAfterInitialization方法.

3.2 AbstractApplicationContext

refresh()方法的主线逻辑如下所示可被分为12个步骤，AbstractApplicationContext进行了相当程度的实现，子类也可基于此进行扩展：

```
public void refresh() throws BeansException, IllegalStateException {
    synchronized (this.startupShutdownMonitor) {
        // ⚠1.Prepare this context for refreshing.
        prepareRefresh();

        //  ⚠2.Tell the subclass to refresh the internal bean factory.
        ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

        // 3.Prepare the bean factory for use in this context.
        prepareBeanFactory(beanFactory);

        try {
            //  ⚠4.Allows post-processing of the bean factory in context subclasses.
            postProcessBeanFactory(beanFactory);

            //  ⚠5.Invoke factory processors registered as beans in the context.
            invokeBeanFactoryPostProcessors(beanFactory);

            //  ⚠6.Register bean processors that intercept bean creation.
            registerBeanPostProcessors(beanFactory);

            //  ⚠7.Initialize message source for this context.
            initMessageSource();

            //  ⚠8.Initialize event multicaster for this context.
            initApplicationEventMulticaster();

            //  ⚠9.Initialize other special beans in specific context subclasses.
            onRefresh();

            //  ⚠10.Check for listener beans and register them.
            registerListeners();

            //  ⚠11.Instantiate all remaining (non-lazy-init) singletons.
            finishBeanFactoryInitialization(beanFactory);

            //  ⚠12.Last step: publish corresponding event.
            finishRefresh();
        } catch (BeansException ex) {
            //...
        } finally {
            //...
        }
    }
}
```

3.3 refresh()方法介绍

3.3.1 容器刷新前的准备

```
protected void prepareRefresh() {
    this.startupDate = System.current
```

```
    this.closed.set(false);
    this.active.set(true);

    initPropertySources();

    getEnvironment().validateRequiredProperties();

    if (this.earlyApplicationListeners == null) {
        this.earlyApplicationListeners = new LinkedHashSet<>(this.applicationListeners);
    } else {
        this.applicationListeners.clear();
        this.applicationListeners.addAll(this.earlyApplicationListeners);
    }
    this.earlyApplicationEvents = new LinkedHashSet<>();
}
```

进行了容器刷新前的准备工作，如记录开始时间(用于计算启动时长)、容器启动状态的设置、属性的初始化操作。
其中，子类可扩展 `initPropertySources()` 方法，Spring web框架对该方法进行了扩展，实现从环境变量中获取属性值填充占位符。
`getEnvironment().validateRequiredProperties()` 可用于进行容器启动前的环境变量校验，要求指定的变量必须被赋值。
`this.earlyApplicationEvents` 属性用于收集事件广播器被初始化前的事件，在广播器创建后再触发这些事件，因此需要提前被初始化；当容器启动完
性需要被再次设置为null。

3.3.2 获取beanFactory

```
protected ConfigurableListableBeanFactory obtainFreshBeanFactory() {
    refreshBeanFactory();
    return getBeanFactory();
}
```

`refreshBeanFactory()` 在子类中有不同的实现，而 `getBeanFactory()` 返回的都是new出来的DefaultListableBeanFactory类型的对象。
对于AbstractRefreshableApplicationContext类型的Spring容器，refreshBeanFactory()进行了以下扩展：

```
protected final void refreshBeanFactory() throws BeansException {
    DefaultListableBeanFactory beanFactory = createBeanFactory();
    beanFactory.setSerializationId(getId());
    customizeBeanFactory(beanFactory);
    loadBeanDefinitions(beanFactory);
    this.beanFactory = beanFactory;
}
```

`createBeanFactory()` 方法通过直接new方式创建DefaultListableBeanFactory类型的IOC容器；通过调用容器的 `setSerializationId` 方法设置serializa
性。
重点在于 `customizeBeanFactory(beanFactory);` 和 `loadBeanDefinitions(beanFactory);` 方法；
customizeBeanFactory方法允许对容器进行一些设置，如同名Bean是否覆盖问题、是否支持循环依赖等，如下所示：

```
protected void customizeBeanFactory(DefaultListableBeanFactory beanFactory) {
    if (this.allowBeanDefinitionOverriding != null) {
        beanFactory.setAllowBeanDefinitionOverriding(this.allowBeanDefinitionOverriding);
    }
    if (this.allowCircularReferences != null) {
        beanFactory.setAllowCircularReferences(this.allowCircularReferences);
    }
}
```

`loadBeanDefinitions(beanFactory)` 方法的功能是向IOC容器中注册BeanDefinition信息，这些BeanDefinition信息可以来自于XML配置文件、属性文件
配置文件等。

3.3.3 对beanFactory准备工作

`prepareBeanFactory(beanFactory)` 方法为beanFactory进行容器刷新前的准备工作，可以分为如下几类：
（1）初始化Spring组件
包括类加载器BeanClassLoader、Aware处理器ApplicationContextAwareProcessor、属性编辑器PropertyEditorRegistrar、bean表达式解析器
BeanExpressionResolver、监听器监测器ApplicationListenerDetector；注意：ApplicationListenerDetector在前后两次被加入到容器的beanPostProc
中。

（2）beanFactory其他属性初始化

对框架引入的Aware接口，如EnvironmentAware、ApplicationEventPublisherAware、MessageSourceAware、ApplicationContextAware等，需要添加
ignoreDependencyInterfaces属性中标记不需要进行依赖检查和自动注入；因为ApplicationContextAwareProcessor组件对于实现Aware接口的类在回
置了属性信息。

（3）LTW配置

AOP切面的织入方式有三种：编译阶段，通过特殊的编译器实现，如AspectJ；类加载阶段，通过LTW实现；运行时，通过JDK或者CGLIB动态代理实
中未见过LTW的实际使用场景，不是本文关注的对象。

（4）注入环境信息相关的Bean对象

包括环境对象Bean(environment)，系统属性Bean(systemProperties)，系统环境变量Bean(systemEnvironment)，这些Bean对象的直接数据来源为 `System.getProperties()` 、 `System.getenv()` ，即将机器的环境变量信息使用Bean的方式进行了包装。

3.3.4 postProcessBeanFactory

预留给子类容器扩展，在容器刷新前进行的定制化操作。

3.3.4 invokeBeanFactoryPostProcessors(beanFactory)

Spring容器按照 PriorityOrder接口 > Ordered接口 > non的顺序依次调用BeanFactoryPostProcessor对象的postProcessBeanFactory方法。该方法为容
即容器启动过程中postProcessBeanFactory之后调用一次。

3.3.4 registerBeanPostProcessors(beanFactory)

Spring容器按照 PriorityOrder接口 > Ordered接口 > non的顺序依次将BeanPostProcessor加入到IOC容器的beanPostProcessors属性中。在Bean对象
阶段会调用BeanPostProcessor的勾子方法，即每个Bean在创建过程中都需要经历BeanPostProcessor的装饰和处理。
另外，在该方法的最后，Spring再次将ApplicationListenerDetector加入到IOC中，读者可以在Spring系列-5 事件机制文章中找到答案。

3.3.4 initMessageSource();

初始化国际化资源，请参考：Spring系列-7 国际化

3.3.4 initApplicationEventMulticaster();

初始化Spring容器的事件广播器，请参考：Spring系列-5 事件机制

3.3.4 onRefresh();

预留给子类容器扩展，扩展向IOC容器注册单例Bean前的定制行为。SpringBoot对此方法进行了扩展，后续在介绍SpringBoot启动流程时进行详细介

3.3.4 registerListeners();

```
protected void registerListeners() {
    for (ApplicationListener<?> listener : getApplicationListeners()) {
        getApplicationEventMulticaster().addApplicationListener(listener);
    }
    String[] listenerBeanNames = getBeanNamesForType(ApplicationListener.class, true, false);
    for (String listenerBeanName : listenerBeanNames) {
        getApplicationEventMulticaster().addApplicationListenerBean(listenerBeanName);
    }

    Set<ApplicationEvent> earlyEventsToProcess = this.earlyApplicationEvents;
    this.earlyApplicationEvents = null;
    if (!CollectionUtils.isEmpty(earlyEventsToProcess)) {
        for (ApplicationEvent earlyEvent : earlyEventsToProcess) {
            getApplicationEventMulticaster().multicastEvent(earlyEvent);
        }
    }
}
```

`registerListeners()` 方法做了两件事件：

（1）向事件广播器注册监听器

在Spring容器的事件广播器被初始化前，向Spring容器注册的监听器都会保存在 `this.applicationListeners` 属性上：

```
public void addApplicationListener(ApplicationListener<?> listener) {
    Assert.notNull(listener, "Applica
    if (this.applicationEventMulticas
```

```
        this.applicationEventMulticaster.addApplicationListener(listener);
    }
    this.applicationListeners.add(listener);
}
```

因此，需要在事件广播器被初始化后，将监听器注册到广播器上：

```
for (ApplicationListener<?> listener : getApplicationListeners()) {
    getApplicationEventMulticaster().addApplicationListener(listener);
}
```

同时，从IOC中取出所有ApplicationListener类型的Bean对象，即用户自定义的监听器对象，将其注册到Spring事件广播器上：

```
String[] listenerBeanNames = getBeanNamesForType(ApplicationListener.class, true, false);
for (String listenerBeanName : listenerBeanNames) {
    getApplicationEventMulticaster().addApplicationListenerBean(listenerBeanName);
}
```

（2）触发earlyEvent

在容器的准备阶段，Spring对 `this.earlyApplicationEvents` 属性进行了初始化，即不会为空；当向Spring容器发生事件时，被记录在 `this.earlyApplicationEvents` 属性中：

```
protected void publishEvent(Object event, @Nullable ResolvableType eventType) {
    //...
    if (this.earlyApplicationEvents != null) {
        this.earlyApplicationEvents.add(applicationEvent);
    } else {
        getApplicationEventMulticaster().multicastEvent(applicationEvent, eventType);
    }
    //...
}
```

在事件广播器被初始化后，需要立刻触发寄存在 `this.earlyApplicationEvents` 属性中的事件，并将 `this.earlyApplicationEvents` 属性设置为空，以
事件触发可以经过广播器，不再寄存于 `this.earlyApplicationEvents` 属性：

```
Set<ApplicationEvent> earlyEventsToProcess = this.earlyApplicationEvents;
this.earlyApplicationEvents = null;
if (!CollectionUtils.isEmpty(earlyEventsToProcess)) {
    for (ApplicationEvent earlyEvent : earlyEventsToProcess) {
        getApplicationEventMulticaster().multicastEvent(earlyEvent);
    }
}
```

3.3.4 finishBeanFactoryInitialization(beanFactory);

完成向容器中注册所有单例非lazy的bean对象的操作，请参考：Spring系列-2 Bean的生命周期、Spring系列-3 Bean实例化与依赖注入、Spring系列-
与三级缓存。

3.3.4 finishRefresh()

完成容器刷新后的清理工作。

Spring 启动流程_spring启动流程
创建Spring容器 ApplicationContext applicationContext = new AnnotationConfigApplicationContext(SpringTest.class); 1 进入构造方法 public AnnotationConfigApplication(

15、Spring~容器启动过程 **最新发布** sproutBoy的
15、Spring~容器启动过程容器启动过程AnnotationConfigApplicationContext类的四个构造器：启动过程详解无参构造方法refresh()方法prepareRefresh()方法prepareBean

spring启动流程源码 qq_38923630的
spring容器启动流程： BeanDefinitionReader 配置文件的读取： （xml、yaml、json、properties） public void refresh() throws BeansException, IllegalStateException { sy

Spring启动流程解析_spring启动过程详解
一、启动Spring通常只需要一到两行代码如: 1、ConfigurableApplicationContext context =newClassPathXmlApplicationContext("abc.xml"); 或SpringBoot常用类似如下方

SpringBoot系列之【启动流程详解】_springboot启动流程
这个【SpringBoot启动流程】就是当初没回答好的部分,这次来补上,机会虽然错过,但知识绝不能错过,以下整理相关知识点。 一、SpringBoot特点 这边先简单扼要叙述一下

【Spring容器启动流程是怎样的】 YYT9527的
Spring容器的启动流程可以归纳为三个步骤

spring 启动过程 weixin_30755393
首先，对于一个web应用，其部署在web容器中，web容器提供其一个全局的上下文环境，这个上下文就是ServletContext，其为后面的spring IoC容器提供宿主环境； 其

spring(15) SpringBoot启动过程
spring-boot:2.2.x spring-framework:5.2.x 话不多说,下面就让我们开始了解 SpringBoot 的启动过程吧。 一、过程简介 首先,SpringBoot 启动的时候,会构造一个SpringAppli

Spring容器启动流程_spring容器启动流程是怎样的
Spring容器启动流程包含主要有两个过程:容器初始化、容器刷新 我们常用的容器有如下2种 基于xml配置Bean(ClassPathXmlApplicationContext) 基于注解配置Bean(Anno

Spring启动流程 m0_56737477的
主要介绍了Spring启动的流程

spring的启动过程 暖心~的
spring启动的过程其实就是Ioc容器的启动过程，对于web程序来说，Ioc容器启动即是建立上下文的过程。spring的启动过程： 1、首先，对于一个web应用，其部署在web

分析SpringBoot启动配置原理_spring启动加载顺序及原理
(十)SpringApplication启动异常处理 四、SpringBoot自动配置分析 (一)自动装配原理分析 (二)条件化自动装配 (三)自动配置原理举例:HttpEncodingAutoConfiguration(HTTP

spring启动流程 ice-wee的
未验证，仅供参考： 链接：http://blog.csdn.net/chjttony/article/details/6358814 --《Spring技术内幕》学习笔记10——Web环境中Spring的启动过程 链接：http://www.cnb

camunda-bpm-spring-boot-starter：Camunda BPM已启动!
4. **Java类集成**：Camunda提供了一系列的Java API，如`ProcessEngine`、`RuntimeService`、`TaskService`等，用于在代码中启动流程、处理任务等。 5. **Spring Be

spring-5.3.9-dist.zip
Spring 5.3.9 还会支持最新的Spring Boot版本，Spring Boot简化了Spring应用的初始搭建和运行过程，通过预配置和自动配置，让开发者能够快速启动项目。此外，Sprin

java spring-boot shell 启动器
1. **启动应用**：使用`java`命令运行包含Spring-Boot应用的JAR文件，通常附带一些参数，如内存分配、JVM选项等。 2. **停止应用**：通过查找与应用相关的PID（进程

spring-boot-reference2018最新版
Spring Boot提供了丰富的测试支持，包括`@SpringBootTest`用于启动整个Spring Boot应用进行集成测试，`@WebMvcTest`用于只启动Web层进行Controller测试，`@Da

springcloud-demo-master_spring-cloud_cloud_
1. **Spring Boot**: 是Spring框架的一个扩展，它通过自动配置、起步依赖和内嵌服务器等特性，极大地简化了Java应用的初始搭建以及开发过程。 2. **Spring Cloud**: 基

Spring容器的启动流程 一个喜欢诗和远方的程
（本文基于 Spring 的 5.1.6.RELEASE 版本） Spring的启动流程可以归纳为三个步骤： 1、初始化Spring容器，注册内置的BeanPostProcessor的BeanDefinition到容器中

Spring启动过程（一） Chenbug666的
Spring的启动过程，就是其IoC容器的启动过程，本质就是创建和初始化bean的工厂（BeanFactory），BeanFactory其实就是整个SpringIoc的核心，Spring 使用 Bean

flowable-spring-boot-starter-process 流程定义设计
Flowable 是一个开源的 BPM（Business Process Management）框架，它可以帮助开发者快速高效地实现业务流程管理功能。Flowable 提供了一系列的 API 和工具，可

Ewen Seong
码龄6年 暂无认证

83	3464	1万+	15万+	
原创	周排名	总排名	访问	等级
1685	2535	800	31	815
积分	粉丝	获赞	评论	收藏

私信 已关注

搜博主文章

热门文章

Spring系列-9 Async注解使用与原理 4592
Spring系列-6 占位符使用和原理 4402
Spring系列-1 启动流程 4324
SpringMVC系列-1 使用方式和启动流程 4023
事务-2 Spring与Mybatis事务实现原理 3909

分类专栏

工具类 6篇
笔记 3篇
前端 9篇
Nginx系列 12篇
三方件 7篇
SpringMVC系列 6篇

最新评论

前端系列-7 Vue3响应式数据
全栈小5: 文章写的很详细，条理清晰，很容易看进去，学到了很多知识，感谢博主分 ...

前端系列-7 Vue3响应式数据
ha_lydms: 非常不错的技术领域文章分享，解决了我在实践中的大问题！博主很有 ...

多线程系列-2 线程中断机制

Ewen Seong: 可以结合"多线程系列-1 线程的状态"理解线程中断的概念

Nginx系列-7 upstream与负载均衡

阿登_: 描述得很详细 很到位👍

Lua使用方式介绍

Ewen Seong: lua官网地址: https://www.lua.org/

最新文章

LocalDateTime的序列化和反序列化

前端系列-9 Vue3生命周期和computed和watch

Nginx系列-12 Nginx使用Lua脚本进行JWT校验

2024年 35篇	2023年 18篇
2022年 17篇	2021年 13篇

目录

背景

1.Spring启动流程

2.使用方式

2.1 xml 配置文件方式

2.2 配置类方式

3.实现原理

3.1 属性介绍

3.2 AbstractApplicationContext

3.3 refresh()方法介绍

3.3.1 容器刷新前的准备

3.3.2 获取beanFactory

3.3.3 对beanFactory准备工作

3.3.4 postProcessBeanFactory

3.3.4 invokeBeanFactorvPost…