

IoC

ApplicationContext与BeanFactory

BeanFactory是最基础的IOC容器，而ApplicationContext是在其基础上添加了一些新功能。
新特性

1. 支持不同的信息源。其拓展了MessageSource接口，从而支持了国际化的实现。
2. 访问资源。其体现在ResourceLoader和Resource的支持上，从而可以在不同地方获取Bean定义资源。
3. 支持应用实践。继承了接口ApplicationEventPublisher，从而引入了事件机制。
4. 其他的附加功能。

通过使用IoC容器，对象依赖关系的管理被反转了，转到IoC容器中来，对象之间的相互依赖关系由IoC容器进行管理，并由IoC容器（即BeanFactory）完成对象的注入。

DefaultListableBeanFactory作为一个默认的功能完整的IoC容器来使用。

注入方式：

1. 接口注入
2. setter注入
3. 构造器注入

BeanDefinition用于管理基于Spring应用中的各种对象以及他们之间的相互依赖关系。抽象了我们对Bean的定义，是让容器起作用的主要数据类型。（存在形式就是XML配置）

XmlBeanFactory

```
public class XmlBeanFactory extends DefaultListableBeanFactory {
    private final XmlBeanDefinitionReader reader = new
    XmlBeanDefinitionReader(this);
    public XmlBeanFactory(Resource resource) throws BeansException {
        //IoC容器初始化的时候，要对BeanDefinition进行指定，xml文件要被Resource对象进行包裹
        this(resource, null);
    }
    public XmlBeanFactory(Resource resource, BeanFactory parentBeanFactory)
    throws BeansException {
        super(parentBeanFactory);
        //使用reader来载入BeanDefinition的过程
        this.reader.loadBeanDefinitions(resource);
    }
}
```

FileSystemXmlApplicationContext

```
public FileSystemXmlApplicationContext(String[] configLocations, boolean
refresh, ApplicationContext parent)
    throws BeansException {

    super(parent);
    //将xml文件进行路径处理
    setConfigLocations(configLocations);
    if (refresh) {
        //进行IoC容器的初始化
        refresh();
    }
}
```

IoC容器的初始化时由前面介绍的refresh方法来启动的，这个方法标志着IoC容器的正式启动。包括BeanDefinition的Resource定位、载入和注册。

IoC容器中BeanDefinition是被注入到HashMap中去的，IoC容器是通过这个HashMap来持有这些BeanDefinition数据的。

需要注意的是Bean的载入和依赖注入是两独立的过程。IoC容器的初始化只是Bean载入的过程，而依赖注入是在应用第一次通过getBean想容器索取Bean的时候。但也有例外，当Bean定义时使用了lazyinit属性，则该Bean在IoC初始化的时候不会依赖注入了。

AbstractApplicationContext

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

        // Tell the subclass to refresh the internal bean factory.用于
        BeanDefinition的载入
        ConfigurableListableBeanFactory beanFactory =
        obtainFreshBeanFactory();

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

```

try {
    // 设置BeanFactory的后置处理
    postProcessBeanFactory(beanFactory);

    // 调用BeanFactory的后处理器，这些后处理器是在Bean定义中向容器注册的
    invokeBeanFactoryPostProcessors(beanFactory);

    // 注册Bean的后处理器，在Bean创建过程中调用
    registerBeanPostProcessors(beanFactory);

    // 对上下文中的消息源初始化
    initMessageSource();

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

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

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

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

    // Last step: publish corresponding event.
    finishRefresh();
}

catch (BeansException ex) {
    logger.warn("Exception encountered during context initialization
- cancelling refresh attempt", ex);

    // Destroy already created singletons to avoid dangling
resources.
    destroyBeans();

    // Reset 'active' flag.

```

```

        cancelRefresh(ex);

        // Propagate exception to caller.
        throw ex;
    }
}

```

AbstractRefreshableApplicationContext

```

@Override
protected final void refreshBeanFactory() throws BeansException {
    if (hasBeanFactory()) {
        destroyBeans();
        closeBeanFactory();
    }
    try {
        //在上下文中创建DefaultListableBeanFactory的地方
        DefaultListableBeanFactory beanFactory = createBeanFactory();
        beanFactory.setSerializationId(getId());
        customizeBeanFactory(beanFactory);
        //载入
        loadBeanDefinitions(beanFactory);
        synchronized (this.beanFactoryMonitor) {
            this.beanFactory = beanFactory;
        }
    }
    catch (IOException ex) {
        throw new ApplicationContextException("I/O error parsing bean
definition source for " + getDisplayName(), ex);
    }
}

```

Bean的生命周期

1. Bean实例的创建
2. 为bean实例设置属性
3. 调用bean的初始化方法 (initializeBean)
4. 应用可以通过IoC容器使用bean

5. 当容器关闭时，调用bean的销毁方法

Bean的销毁

1. 清除单例缓存
2. 关闭BeanFactory
3. 关闭其他的

prepareBeanFactory

完成BeanFactory的准备工作。为容器配置ClassLoader、PropertyEditor和BeanPostProcessor等。

getObjectForBeanInstance

返回FactoryBean的产物Bean

populateBean

这里完成了依赖注入的处理