6、Spring源码

笔记本： spring
创建时间： 2022/4/4 17:23
作者： 雷丰阳

1、Spring-IOC-AOP（动态代理）；多层代理

```
LogAspectpRroxy{
    try{
        @Before
        method.invoke()//pjp.procced(args){
            BAspectProxy{
                @Before
                method.invoke()//---目标方法
                 @AfterReturning
                 //xxxxxxxx
                    //修改了返回值
            }
        }
        @AfterReturning
    }catch(e){
         @AfterThrowing
    }finally{
         @After
    }
}
```

IOC：
1、IOC是一个容器
2、容器启动的时候创建所有单实例对象
3、我们可以直接从容器中获取到这个对象
SpringIOC：
1）、ioc容器的启动过程？启动期间都做了什么（什么时候创建所有单实例bean）
2）、ioc是如何创建这些单实例bean，并如何管理的；到底保存在了那里？

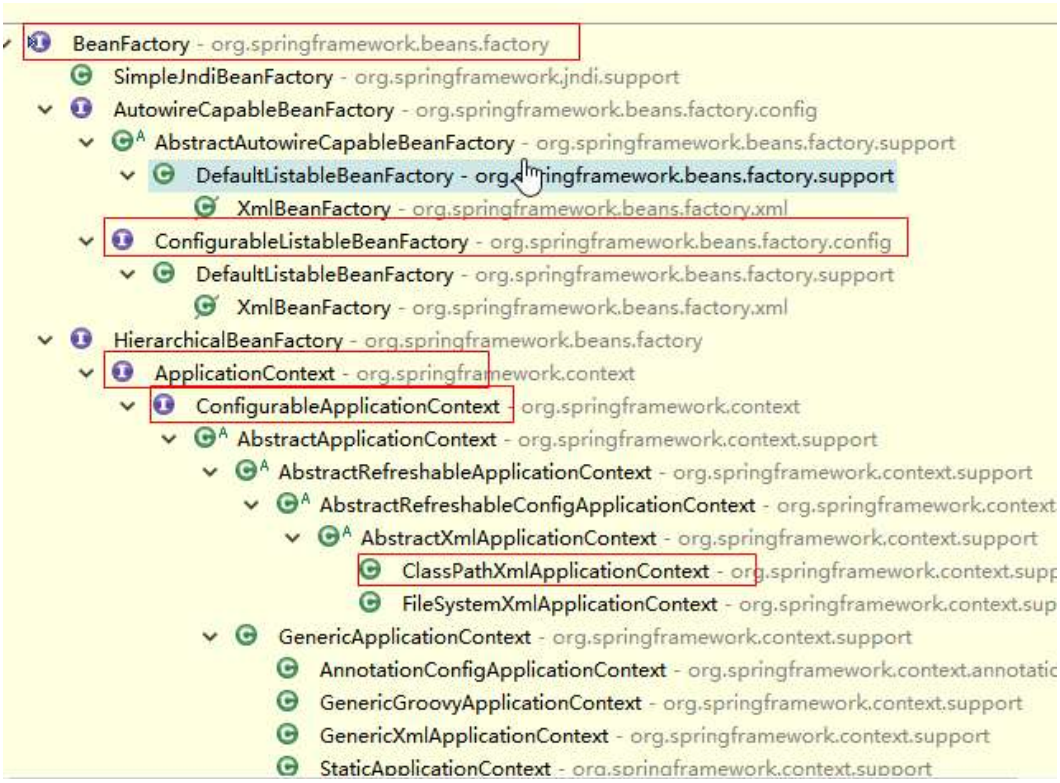
思路：
从HelloWorld开始，调试每个方法的作用；
1、ClassPathXMLApplicationContext构造器

ApplicationContext ioc =new ClassPathXmlApplicationContext("ioc.xml")

```
this(new String[] {configLocation}, true, null);
```

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

        super(parent);
        setConfigLocations(configLocations);
        if (refresh) {
               //所有单实例bean创建完成
            refresh();
        }
    }
```



BeanFactory：Bean工厂；

refresh();实现；Spring；BeanFactory的流程；也是ioc容器的创建流程

```
@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.
               //Spring解析xml配置文件将要创建的所有bean的配置信息保存起来，观看
Spring对xml的解析
            ConfigurableListableBeanFactory beanFactory =
obtainFreshBeanFactory();

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

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

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

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

                // Initialize message source for this context.
                //支持国际化功能的;
                initMessageSource();

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

                // Initialize other special beans in specific context
subclasses.
               //留给子类的方法
                onRefresh();

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

```
                // Instantiate all remaining (non-lazy-init) singletons.
                //初始化所有单实例bean的地方
                finishBeanFactoryInitialization(beanFactory);

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

            catch (BeansException ex) {
                // Destroy already created singletons to avoid dangling
resources.
                destroyBeans();

                // Reset 'active' flag.
                cancelRefresh(ex);

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

finishBeanFactoryInitialization(beanFactory);实现AbstractApplicationContext：

```
protected void
finishBeanFactoryInitialization(ConfigurableListableBeanFactory beanFactory)
{
        // Initialize conversion service for this context.
        if (beanFactory.containsBean(CONVERSION_SERVICE_BEAN_NAME) &&
                beanFactory.isTypeMatch(CONVERSION_SERVICE_BEAN_NAME,
ConversionService.class)) {
            beanFactory.setConversionService(
                    beanFactory.getBean(CONVERSION_SERVICE_BEAN_NAME,
ConversionService.class));
        }

        // Initialize LoadTimeWeaverAware beans early to allow for
registering their transformers early.
        String[] weaverAwareNames =
beanFactory.getBeanNamesForType(LoadTimeWeaverAware.class, false, false);
        for (String weaverAwareName : weaverAwareNames) {
            getBean(weaverAwareName);
        }

        // Stop using the temporary ClassLoader for type matching.
        beanFactory.setTempClassLoader(null);

        // Allow for caching all bean definition metadata, not expecting
further changes.
        beanFactory.freezeConfiguration();

        // Instantiate all remaining (non-lazy-init) singletons.初始化所有单实
例
        beanFactory.preInstantiateSingletons();
    }
```

DefaultListableBeanFactory：bean工厂；创建bean

```
@Override
    public void preInstantiateSingletons() throws BeansException {
        if (this.logger.isDebugEnabled()) {
            this.logger.debug("Pre-instantiating singletons in " + this);
        }

          //拿到所有要创建的bean的名字
        List<String> beanNames;
        synchronized (this.beanDefinitionMap) {
            // Iterate over a copy to allow for init methods which in turn
register new bean definitions.
```

```
            // While this may not be part of the regular factory bootstrap,
it does otherwise work fine.
            beanNames = new ArrayList<String>(this.beanDefinitionNames);
        }


          //按顺序创建bean
        for (String beanName : beanNames) {
            //根据beandid获取到bean的定义信息;
            RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);

               //判断bean是单实例的，并且不是抽象的，并且不是懒加载的
            if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {
                   //是否是一个实现了FactoryBean接口的bean
                if (isFactoryBean(beanName)) {
                    final FactoryBean<?> factory = (FactoryBean<?>)
getBean(FACTORY_BEAN_PREFIX + beanName);
                    boolean isEagerInit;
                    if (System.getSecurityManager() != null && factory
instanceof SmartFactoryBean) {
                        isEagerInit = AccessController.doPrivileged(new
PrivilegedAction<Boolean>() {
                            @Override
                            public Boolean run() {
                                return ((SmartFactoryBean<?>)
factory).isEagerInit();
                            }
                        }, getAccessControlContext());
                    }
                    else {
                        isEagerInit = (factory instanceof SmartFactoryBean
&&
                                ((SmartFactoryBean<?>)
factory).isEagerInit());
                    }
                    if (isEagerInit) {
                        getBean(beanName);
                    }
                }
                else {
                    getBean(beanName);
                }
            }
        }
    }
```

getBean(beanName);创建bean的细节

```
getBean(){
           //所有的getBean掉的是它
          doGetBean(name, null, null, false);
}
```

AbstractBeanFactory：doGetBean(name, null, null, false);

```
@SuppressWarnings("unchecked")
    protected <T> T doGetBean(
            final String name, final Class<T> requiredType, final Object[]
args, boolean typeCheckOnly)
            throws BeansException {

        final String beanName = transformedBeanName(name);
        Object bean;

        // Eagerly check singleton cache for manually registered singletons.
          //先从已经注册的所有单实例bean中看有没有个bean。第一次创建bean是没有的;
        Object sharedInstance = getSingleton(beanName);
        if (sharedInstance != null && args == null) {
            if (logger.isDebugEnabled()) {
                if (isSingletonCurrentlyInCreation(beanName)) {
                    logger.debug("Returning eagerly cached instance of
singleton bean '" + beanName +
```

```
                            "' that is not fully initialized yet - a
consequence of a circular reference");
                }
                else {
                    logger.debug("Returning cached instance of singleton
bean '" + beanName + "'");
                }
            }
            bean = getObjectForBeanInstance(sharedInstance, name, beanName,
null);
        }

        else {
            // Fail if we're already creating this bean instance:
            // We're assumably within a circular reference.
            if (isPrototypeCurrentlyInCreation(beanName)) {
                throw new BeanCurrentlyInCreationException(beanName);
            }

            // Check if bean definition exists in this factory.
            BeanFactory parentBeanFactory = getParentBeanFactory();
            if (parentBeanFactory != null &&
!containsBeanDefinition(beanName)) {
                // Not found -> check parent.
                String nameToLookup = originalBeanName(name);
                if (args != null) {
                    // Delegation to parent with explicit args.
                    return (T) parentBeanFactory.getBean(nameToLookup,
args);
                }
                else {
                    // No args -> delegate to standard getBean method.
                    return parentBeanFactory.getBean(nameToLookup,
requiredType);
                }
            }

            if (!typeCheckOnly) {
                markBeanAsCreated(beanName);
            }

            try {
                final RootBeanDefinition mbd =
getMergedLocalBeanDefinition(beanName);
                checkMergedBeanDefinition(mbd, beanName, args);

                // Guarantee initialization of beans that the current bean
depends on.
               //拿到创建当前bean之前需要提前创建的bean。depends-on属性；如果有就
循环创建
                String[] dependsOn = mbd.getDependsOn();

                if (dependsOn != null) {
                    for (String dependsOnBean : dependsOn) {
                        if (isDependent(beanName, dependsOnBean)) {
                            throw new BeanCreationException("Circular
depends-on relationship between '" +
                                    beanName + "' and '" + dependsOnBean +
"'");
                        }
                        registerDependentBean(dependsOnBean, beanName);
                        getBean(dependsOnBean);
                    }
                }


                // Create bean instance.创建bean实例
                if (mbd.isSingleton()) {
                    sharedInstance = getSingleton(beanName, new
ObjectFactory<Object>() {
                        @Override
                        public Object getObject() throws BeansException {
                            try {
                                return createBean(beanName, mbd, args);
```

```
                            }
                            catch (BeansException ex) {
                                // Explicitly remove instance from singleton
cache: It might have been put there
                                // eagerly by the creation process, to allow
for circular reference resolution.
                                // Also remove any beans that received a
temporary reference to the bean.
                                destroySingleton(beanName);
                                throw ex;
                            }
                        }
                    });
                    bean = getObjectForBeanInstance(sharedInstance, name,
beanName, mbd);
                }

                else if (mbd.isPrototype()) {
                    // It's a prototype -> create a new instance.
                    Object prototypeInstance = null;
                    try {
                        beforePrototypeCreation(beanName);
                        prototypeInstance = createBean(beanName, mbd, args);
                    }
                    finally {
                        afterPrototypeCreation(beanName);
                    }
                    bean = getObjectForBeanInstance(prototypeInstance, name,
beanName, mbd);
                }

                else {
                    String scopeName = mbd.getScope();
                    final Scope scope = this.scopes.get(scopeName);
                    if (scope == null) {
                        throw new IllegalStateException("No Scope registered
for scope '" + scopeName + "'");
                    }
                    try {
                        Object scopedInstance = scope.get(beanName, new
ObjectFactory<Object>() {
                            @Override
                            public Object getObject() throws BeansException
{
                                beforePrototypeCreation(beanName);
                                try {
                                    return createBean(beanName, mbd, args);
                                }
                                finally {
                                    afterPrototypeCreation(beanName);
                                }
                            }
                        });
                        bean = getObjectForBeanInstance(scopedInstance,
name, beanName, mbd);
                    }
                    catch (IllegalStateException ex) {
                        throw new BeanCreationException(beanName,
                                "Scope '" + scopeName + "' is not active for
the current thread; " +
                                "consider defining a scoped proxy for this
bean if you intend to refer to it from a singleton",
                                ex);
                    }
                }
            }
            catch (BeansException ex) {
                cleanupAfterBeanCreationFailure(beanName);
                throw ex;
            }
        }

        // Check if required type matches the type of the actual bean
instance.
```

```
        if (requiredType != null && bean != null &&
!requiredType.isAssignableFrom(bean.getClass())) {
            try {
                return getTypeConverter().convertIfNecessary(bean,
requiredType);
            }
            catch (TypeMismatchException ex) {
                if (logger.isDebugEnabled()) {
                    logger.debug("Failed to convert bean '" + name + "' to
required type [" +
                            ClassUtils.getQualifiedName(requiredType) + "]",
ex);
                }
                throw new BeanNotOfRequiredTypeException(name, requiredType,
bean.getClass());
            }
        }
        return (T) bean;
    }
```

DefaultSingletonBeanRegistry:

```
/** Cache of singleton objects: bean name --> bean instance */
private final Map<String, Object> singletonObjects = new ConcurrentHashMap<String, Object>(64);
```

getSingleton方法;

```
public Object getSingleton(String beanName, ObjectFactory<?>
singletonFactory) {
        Assert.notNull(beanName, "'beanName' must not be null");
        synchronized (this.singletonObjects) {
               //先从一个地方将这个beanget出来
            Object singletonObject = this.singletonObjects.get(beanName);
            if (singletonObject == null) {
                if (this.singletonsCurrentlyInDestruction) {
                    throw new BeanCreationNotAllowedException(beanName,
                            "Singleton bean creation not allowed while the
singletons of this factory are in destruction " +
                            "(Do not request a bean from a BeanFactory in a
destroy method implementation!)");
                }
                if (logger.isDebugEnabled()) {
                    logger.debug("Creating shared instance of singleton bean
'" + beanName + "'");
                }
                beforeSingletonCreation(beanName);
                boolean recordSuppressedExceptions =
(this.suppressedExceptions == null);
                if (recordSuppressedExceptions) {
                    this.suppressedExceptions = new LinkedHashSet<Exception>
();
                }
                try {
                    //创建bean
                    singletonObject = singletonFactory.getObject();
                }
                catch (BeanCreationException ex) {
                    if (recordSuppressedExceptions) {
                        for (Exception suppressedException :
this.suppressedExceptions) {
                            ex.addRelatedCause(suppressedException);
                        }
                    }
```

```
                    throw ex;
                }
                finally {
                    if (recordSuppressedExceptions) {
                        this.suppressedExceptions = null;
                    }
                    afterSingletonCreation(beanName);
                }
               //添加创建的bean
                addSingleton(beanName, singletonObject);
            }
            return (singletonObject != NULL_OBJECT ? singletonObject :
null);
        }
    }
```

创建好的对象最终会保存在一个map中；
ioc容器之一：保存单实例bean的地方；
ioc就是一个容器，单实例bean保存在一个map中；

```
DefaultSingletonBeanRegistry-singletonObjects
```

BeanFactory和ApplicationContext的区别；

ApplicationContext是BeanFactory的子接口；
BeanFactory：bean工厂接口；负责创建bean实例；容器里面保存的所有单例bean其实是一个map；
　　Spring最底层的接口；
ApplicationContext：是容器接口；更多的负责容器功能的实现；（可以基于beanFactory创建好的对象之上完成强大的容器）
　　容器可以从map获取这个bean，并且aop。di。在ApplicationContext接口的下的这些类里面；

BeanFactory最底层的接口，ApplicationContext留给程序员使用的ioc容器接口；
ApplicationContext是BeanFactory的子接口；
ApplicationContext

Spring里面最大的模式就是工厂模式；
　　<bean class=""></bean>
　　BeanFactory：bean工厂；工厂模式；帮用户创建bean
