一． Bean 的创建

1.

```
ApplicationContext beans = new ClassPathXmlApplicationContext(
        "demo/bean/create/bean.xml");
beans.getBean("singleA");
```

使用 AbstractApplicationContext

```
        public Object getBean(String name) throws
BeansException {
        assertBeanFactoryActive();
        return getBeanFactory().getBean(name);

 }
```

AbstractRefreshableApplicationContext

```
    public final ConfigurableListableBeanFactory getBeanFactory()
{
        synchronized (this.beanFactoryMonitor) {
            if (this.beanFactory == null) {
                throw new IllegalStateException("BeanFactory not
initialized or already closed - " +
                        "call 'refresh' before accessing beans via
the ApplicationContext");
            }
            return this.beanFactory;
        }
 }
```

AbstractXmlApplicationContext

```
    protected void
loadBeanDefinitions(DefaultListableBeanFactory beanFactory)
throws BeansException, IOException {
        // Create a new XmlBeanDefinitionReader for the given
BeanFactory.
        XmlBeanDefinitionReader beanDefinitionReader = new
XmlBeanDefinitionReader(beanFactory);

        // Configure the bean definition reader with this context's
        // resource loading environment.

 beanDefinitionReader.setEnvironment(this.getEnvironment());
        beanDefinitionReader.setResourceLoader(this);
        beanDefinitionReader.setEntityResolver(new
ResourceEntityResolver(this));
```

```
        // Allow a subclass to provide custom initialization of the
    reader,
        // then proceed with actually loading the bean definitions.
        initBeanDefinitionReader(beanDefinitionReader);
        loadBeanDefinitions(beanDefinitionReader);
    }
```

```
    TxNamespaceHandler
    registerBeanDefinitionParser("annotation-driven", new
    AnnotationDrivenBeanDefinitionParser());
    AnnotationDrivenBeanDefinitionParser
```

```
    AbstractBeanFactory
        public <T> T getBean(String name, @Nullable Class<T>
    requiredType,
                @Nullable Object... args) throws BeansException {

        return doGetBean(name, requiredType, args, false);
    }
```

Spring aop

```
BeanDefinitionParserDelegate
public BeanDefinition parseCustomElement(Element ele, @Nullable
BeanDefinition containingBd) {
    String namespaceUri = getNamespaceURI(ele);
    if (namespaceUri == null) {
        return null;
    }//NamespaceHandlerResolver
    NamespaceHandler handler =
this.readerContext.getNamespaceHandlerResolver().resolve(names
paceUri);
    if (handler == null) {
        error("Unable to locate Spring NamespaceHandler for XML
schema namespace [" + namespaceUri + "]", ele);
        return null;
    }//AopNamespaceHandler继承NamespaceHandlerSupport
    return handler.parse(ele, new
```

```
ParserContext(this.readerContext, this, containingBd));

    }
```

执行this.readerContext.getNamespaceHandlerResolver()获得

```
public class DefaultNamespaceHandlerResolver implements
NamespaceHandlerResolver {
    //根据namespaceUri获取AopNamespaceHandler
    public NamespaceHandler resolve(String namespaceUri) {
        Map<String, Object> handlerMappings =
    getHandlerMappings();
        Object handlerOrClassName =
handlerMappings.get(namespaceUri);
        namespaceHandler.init();//注册解析器
        //缓存handler
        handlerMappings.put(namespaceUri, namespaceHandler);
        return namespaceHandler;
    }
}
```

执行AopNamespaceHandler的init方法注册AspectJAutoProxyBeanDefinitionParser

```
AopNamespaceHandler extends NamespaceHandlerSupport

public void init() {
        registerBeanDefinitionParser("config", new
ConfigBeanDefinitionParser());
        registerBeanDefinitionParser("aspectj-autoproxy", new
AspectJAutoProxyBeanDefinitionParser());
        registerBeanDefinitionDecorator("scoped-proxy", new
ScopedProxyBeanDefinitionDecorator());
        registerBeanDefinitionParser("spring-configured", new
SpringConfiguredBeanDefinitionParser());
    }
```

接着执行NamespaceHandlerSupport的parse方法

```
handler.parse(ele, new ParserContext(this.readerContext, this,
containingBd));
// element=[aop:aspectj-autoproxy: null]
public BeanDefinition parse(Element element, ParserContext
parserContext) {
//获取AspectJAutoProxyBeanDefinitionParser
        BeanDefinitionParserparser=findParserForElement(element,
parserContext);//寻找解析器
        return (parser != null ? parser.parse(element,
```

```
parserContext) : null);
    }

class AspectJAutoProxyBeanDefinitionParser implements
BeanDefinitionParser {
    @Nullable//注册AnnotationAwareAspectJAutoProxyCreator
    public BeanDefinition parse(Element element, ParserContext
parserContext) {
    AopNamespaceUtils.registerAspectJAnnotationAutoProxyCreator
    IfNecessary(parserContext, element);
        extendBeanDefinition(element, parserContext);
        return null;
    }

}
```

开始创建代理

AbstractAutowireCapableBeanFactory

```
protected Object doCreateBean(final String beanName, final
RootBeanDefinition mbd, final @Nullable Object[] args)
          throws BeanCreationException {
    BeanWrapper  instanceWrapper = createBeanInstance(beanName,
mbd, args);
        populateBean(beanName, mbd, instanceWrapper);
        exposedObject = initializeBean(beanName, exposedObject,
mbd);
}
protected Object initializeBean(final String beanName, final
Object bean, @Nullable RootBeanDefinition mbd) {
    applyBeanPostProcessorsAfterInitialization(wrappedBean,
beanName);
}
```

AnnotationAwareAspectJAutoProxyCreator

```
public class AnnotationAwareAspectJAutoProxyCreator extends
AspectJAwareAdvisorAutoProxyCreator

public abstract class AbstractAdvisorAutoProxyCreator extends
AbstractAutoProxyCreator

public abstract class AbstractAutoProxyCreator extends
ProxyProcessorSupport
```

```
      implements SmartInstantiationAwareBeanPostProcessor,
BeanFactoryAware

AbstractAutoProxyCreator中的
@Override
   public Object postProcessAfterInitialization(@Nullable Object
bean, String beanName) {
      if (bean != null) {
         Object cacheKey = getCacheKey(bean.getClass(),
beanName);
         if (!this.earlyProxyReferences.contains(cacheKey)) {
            return wrapIfNecessary(bean, beanName, cacheKey);
         }
      }
      return bean;
   }

protected Object wrapIfNecessary(Object bean, String beanName,
Object cacheKey) {
      Object[] specificInterceptors =
getAdvicesAndAdvisorsForBean(bean.getClass(), beanName, null);
      if (specificInterceptors != DO_NOT_PROXY) {
         this.advisedBeans.put(cacheKey, Boolean.TRUE);
         Object proxy = createProxy(
               bean.getClass(), beanName, specificInterceptors,
new SingletonTargetSource(bean));
         this.proxyTypes.put(cacheKey, proxy.getClass());
         return proxy;
      }
      this.advisedBeans.put(cacheKey, Boolean.FALSE);
      return bean;
   }


protected Object createProxy(Class<?> beanClass, @Nullable String
beanName,
         @Nullable Object[] specificInterceptors, TargetSource
targetSource) {

      ProxyFactory proxyFactory = new ProxyFactory();
      proxyFactory.copyFrom(this);// this代表
AnnotationAwareAspectJAutoProxyCreator

      if (!proxyFactory.isProxyTargetClass()) {
```

```
            if (shouldProxyTargetClass(beanClass, beanName)) {
                proxyFactory.setProxyTargetClass(true);
            }
            else {
                evaluateProxyInterfaces(beanClass, proxyFactory);
            }
        }

        Advisor[] advisors = buildAdvisors(beanName,
specificInterceptors);
        proxyFactory.addAdvisors(advisors);
        proxyFactory.setTargetSource(targetSource);
        customizeProxyFactory(proxyFactory);

        proxyFactory.setFrozen(this.freezeProxy);
        if (advisorsPreFiltered()) {
            proxyFactory.setPreFiltered(true);
        }
        return proxyFactory.getProxy(getProxyClassLoader());
    }

public Object getProxy(@Nullable ClassLoader classLoader) {
        return createAopProxy().getProxy(classLoader);
    }

protected final synchronized AopProxy createAopProxy() {
        if (!this.active) {
            activate();
        }//this =ProxyFactory;  ProxyFactory extends
ProxyCreatorSupport, ProxyCreatorSupport构造器里创建
DefaultAopProxyFactory,故DefaultAopProxyFactory
=getAopProxyFactory()
        return getAopProxyFactory().createAopProxy(this);
    }

public AopProxy createAopProxy(AdvisedSupport config) throws
AopConfigException {
        if (config.isOptimize() || config.isProxyTargetClass() ||
hasNoUserSuppliedProxyInterfaces(config)) {
            Class<?> targetClass = config.getTargetClass();
            if (targetClass.isInterface() ||
Proxy.isProxyClass(targetClass)) {
                return new JdkDynamicAopProxy(config);
            }
```

```
            return new ObjenesisCglibAopProxy(config);
        }
        else {
            return new JdkDynamicAopProxy(config);
        }
    }
CglibAopProxy implements AopProxy

public Object getProxy(@Nullable ClassLoader classLoader) {
        try {
            // advised =ProxyFactory
            Class<?> rootClass = this.advised.getTargetClass();
            Class<?> proxySuperClass = rootClass;
            if (ClassUtils.isCglibProxyClass(rootClass)) {
                proxySuperClass = rootClass.getSuperclass();
                Class<?>[] additionalInterfaces =
rootClass.getInterfaces();
                for (Class<?> additionalInterface :
additionalInterfaces) {
                    this.advised.addInterface(additionalInterface);
                }
            }

            // Validate the class, writing log messages as necessary.
            validateClassIfNecessary(proxySuperClass,
classLoader);

            // Configure CGLIB Enhancer...
            Enhancer enhancer = createEnhancer();
            if (classLoader != null) {
                enhancer.setClassLoader(classLoader);
                if (classLoader instanceof SmartClassLoader &&
                        ((SmartClassLoader)
classLoader).isClassReloadable(proxySuperClass)) {
                    enhancer.setUseCache(false);
                }
            }
            enhancer.setSuperclass(proxySuperClass);

    enhancer.setInterfaces(AopProxyUtils.completeProxiedInterfa
ces(this.advised));
            enhancer.setNamingPolicy(SpringNamingPolicy.INSTANCE);
            enhancer.setStrategy(new
ClassLoaderAwareUndeclaredThrowableStrategy(classLoader));
```

```
            Callback[] callbacks = getCallbacks(rootClass);
            Class<?>[] types = new Class<?>[callbacks.length];
            for (int x = 0; x < types.length; x++) {
               types[x] = callbacks[x].getClass();
            }
            // fixedInterceptorMap only populated at this point,
after getCallbacks call above
            enhancer.setCallbackFilter(new ProxyCallbackFilter(
                  this.advised.getConfigurationOnlyCopy(),
this.fixedInterceptorMap, this.fixedInterceptorOffset));
            enhancer.setCallbackTypes(types);

            // Generate the proxy class and create a proxy instance.
            return createProxyClassAndInstance(enhancer,
callbacks);
      }
      catch (CodeGenerationException | IllegalArgumentException
ex) {
   }

private Callback[] getCallbacks(Class<?> rootClass) throws
Exception {
      Callback aopInterceptor = new
DynamicAdvisedInterceptor(this.advised);
      Callback[] mainCallbacks = new Callback[] {
            aopInterceptor,  // for normal advice
            targetInterceptor,  // invoke target without
considering advice, if optimized
            new SerializableNoOp(),  // no override for methods
mapped to this
            targetDispatcher, this.advisedDispatcher,
            new EqualsInterceptor(this.advised),
            new HashCodeInterceptor(this.advised)
      };

      Callback[] callbacks;
 calls
      if (isStatic && isFrozen) {
         Method[] methods = rootClass.getMethods();
         Callback[] fixedCallbacks = new
Callback[methods.length];
         this.fixedInterceptorMap = new
HashMap<>(methods.length);
```

```
            for (int x = 0; x < methods.length; x++) {
                List<Object> chain =
this.advised.getInterceptorsAndDynamicInterceptionAdvice(metho
ds[x], rootClass);
                fixedCallbacks[x] = new
FixedChainStaticTargetInterceptor(
                        chain,
this.advised.getTargetSource().getTarget(),
this.advised.getTargetClass());
                this.fixedInterceptorMap.put(methods[x].toString(),
x);
            }
            callbacks = new Callback[mainCallbacks.length +
fixedCallbacks.length];
            System.arraycopy(mainCallbacks, 0, callbacks, 0,
mainCallbacks.length);
            System.arraycopy(fixedCallbacks, 0, callbacks,
mainCallbacks.length, fixedCallbacks.length);
            this.fixedInterceptorOffset = mainCallbacks.length;
        }
        else {
            callbacks = mainCallbacks;
        }
        return callbacks;
    }
```

上面完成代理的所有准备工作

```
ApplicationContext beans = new ClassPathXmlApplicationContext(
                "demo/bean/create/bean.xml");
        Poem juggler = (Poem) beans.getBean("test");
        juggler.getName();//当执行此方法时候会触发callback,执行拦截
器链，入口为下面的intercept
```

CglibAopProxy下的

```
private static class DynamicAdvisedInterceptor implements
MethodInterceptor, Serializable {

        private final AdvisedSupport advised;

        public DynamicAdvisedInterceptor(AdvisedSupport advised) {
            this.advised = advised;
        }

        @Override
```

```
        @Nullable
        public Object intercept(Object proxy, Method method,
Object[] args, MethodProxy methodProxy) throws Throwable {
            Object oldProxy = null;
            boolean setProxyContext = false;
            Object target = null;
            TargetSource targetSource =
this.advised.getTargetSource();//SingletonTargetSource
            try {
                if (this.advised.exposeProxy) {
                    // Make invocation available if necessary.
                    oldProxy = AopContext.setCurrentProxy(proxy);
                    setProxyContext = true;
                }
                // Get as late as possible to minimize the time we "own"
the target, in case it comes from a pool...
                target =
targetSource.getTarget();//SingletonTargetSource
                Class<?> targetClass = (target != null ?
target.getClass() : null);
                List<Object> chain =
this.advised.getInterceptorsAndDynamicInterceptionAdvice(metho
d, targetClass);
                Object retVal;
                // Check whether we only have one InvokerInterceptor:
that is,
                // no real advice, but just reflective invocation of
the target.
                if (chain.isEmpty() &&
Modifier.isPublic(method.getModifiers())) {
                    // We can skip creating a MethodInvocation: just
invoke the target directly.
                    // Note that the final invoker must be an
InvokerInterceptor, so we know
                    // it does nothing but a reflective operation on
the target, and no hot
                    // swapping or fancy proxying.
                    Object[] argsToUse =
AopProxyUtils.adaptArgumentsIfNecessary(method, args);
                    retVal = methodProxy.invoke(target, argsToUse);
                }
                else {
                    // We need to create a method invocation...
                    retVal = new CglibMethodInvocation(proxy, target,
```

```
method, args, targetClass, chain, methodProxy).proceed();
            }
            retVal = processReturnType(proxy, target, method,
retVal);
            return retVal;
         }
         finally {
            if (target != null && !targetSource.isStatic()) {
               targetSource.releaseTarget(target);
            }
            if (setProxyContext) {
               // Restore old proxy.
               AopContext.setCurrentProxy(oldProxy);
            }
         }
      }

      @Override
      public boolean equals(Object other) {
         return (this == other ||
               (other instanceof DynamicAdvisedInterceptor &&

   this.advised.equals(((DynamicAdvisedInterceptor)
other).advised)));
      }

      /**
       * CGLIB uses this to drive proxy creation.
       */
      @Override
      public int hashCode() {
         return this.advised.hashCode();
      }
   }
private static class CglibMethodInvocation extends
ReflectiveMethodInvocation

执行ReflectiveMethodInvocation中的proceed(),完成所有的增强
public Object proceed() throws Throwable {
      if (this.currentInterceptorIndex ==
this.interceptorsAndDynamicMethodMatchers.size() - 1) {
         return invokeJoinpoint();
      }
      Object interceptorOrInterceptionAdvice =
```

```
   this.interceptorsAndDynamicMethodMatchers.get(++this.curren
tInterceptorIndex);
      if (interceptorOrInterceptionAdvice instanceof
InterceptorAndDynamicMethodMatcher) {
         // Evaluate dynamic method matcher here: static part will
already have
         // been evaluated and found to match.
         InterceptorAndDynamicMethodMatcher dm =
               (InterceptorAndDynamicMethodMatcher)
interceptorOrInterceptionAdvice;
         Class<?> targetClass = (this.targetClass != null ?
this.targetClass : this.method.getDeclaringClass());
         if (dm.methodMatcher.matches(this.method, targetClass,
this.arguments)) {
            return dm.interceptor.invoke(this);
         }
         else {
            // Dynamic matching failed.
            // Skip this interceptor and invoke the next in the
chain.
            return proceed();
         }
      }
      else {
         return ((MethodInterceptor)
interceptorOrInterceptionAdvice).invoke(this);
      }
   }
```

```
public class BeanDefinitionParserDelegate {
public BeanDefinition parseCustomElement(Element ele, @Nullable
BeanDefinition containingBd) {
       String namespaceUri = getNamespaceURI(ele);
       if (namespaceUri == null) {
          return null;
       }//DefaultNamespaceHandlerResolver
       NamespaceHandler handler =
this.readerContext.getNamespaceHandlerResolver().resolve(names
paceUri);
       if (handler == null) {
          error("Unable to locate Spring NamespaceHandler for XML
schema namespace [" + namespaceUri + "]", ele);
          return null;
       }//AopNamespaceHandler继承NamespaceHandlerSupport
       return handler.parse(ele, new
ParserContext(this.readerContext, this, containingBd));
    }

}

public class DefaultNamespaceHandlerResolver implements

NamespaceHandlerResolver {

public NamespaceHandler resolve(String namespaceUri) {
       Map<String, Object> handlerMappings = getHandlerMappings();
       Object handlerOrClassName =
handlerMappings.get(namespaceUri);
             NamespaceHandler namespaceHandler =
(NamespaceHandler) BeanUtils.instantiateClass(handlerClass);
             namespaceHandler.init();//注册解析器
             handlerMappings.put(namespaceUri,
namespaceHandler);
             return namespaceHandler;
          }
          catch (ClassNotFoundException ex) {
             throw new FatalBeanException("Could not find
NamespaceHandler class [" + className +
                   "] for namespace [" + namespaceUri + "]", ex);
          }
          catch (LinkageError err) {
             throw new FatalBeanException("Unresolvable class
definition for NamespaceHandler class [" +
                   className + "] for namespace [" + namespaceUri
+ "]", err);
```

```
            }
        }
    }
}
public class TxNamespaceHandler extends NamespaceHandlerSupport
{
        public void init() {
        registerBeanDefinitionParser("advice", new
TxAdviceBeanDefinitionParser());
        registerBeanDefinitionParser("annotation-driven", new
AnnotationDrivenBeanDefinitionParser());
        registerBeanDefinitionParser("jta-transaction-manager",
            new JtaTransactionManagerBeanDefinitionParser());
    }
}

handler.parse(ele, new ParserContext(this.readerContext, this,
containingBd));
```

获取刚才注册的解析器AnnotationDrivenBeanDefinitionParser

```
public BeanDefinition parse(Element element, ParserContext
parserContext) {
        registerTransactionalEventListenerFactory(parserContext);
        String mode = element.getAttribute("mode");
        if ("aspectj".equals(mode)) {
            // mode="aspectj"
            registerTransactionAspect(element, parserContext);
            if
(ClassUtils.isPresent("javax.transaction.Transactional",
getClass().getClassLoader())) {
                registerJtaTransactionAspect(element,
parserContext);
            }
        }
        else {
            // mode="proxy"

    AopAutoProxyConfigurer.configureAutoProxyCreator(element,
parserContext);
        }
        return null;
    }
```

```
private void
registerTransactionalEventListenerFactory(ParserContext
parserContext) {
       RootBeanDefinition def = new RootBeanDefinition();

    def.setBeanClass(TransactionalEventListenerFactory.class);
       parserContext.registerBeanComponent(new
BeanComponentDefinition(def,

    TransactionManagementConfigUtils.TRANSACTIONAL_EVENT_LISTEN
ER_FACTORY_BEAN_NAME));//org.springframework.transaction.confi
g.internalTransactionalEventListenerFactory
    }



       public static void configureAutoProxyCreator(Element
element, ParserContext parserContext) {

    AopNamespaceUtils.registerAutoProxyCreatorIfNecessary(parse
rContext, element);

          String txAdvisorBeanName =
TransactionManagementConfigUtils.TRANSACTION_ADVISOR_BEAN_NAME;
          if
(!parserContext.getRegistry().containsBeanDefinition(txAdvisor
BeanName)) {
             Object eleSource =
parserContext.extractSource(element);

             // Create the TransactionAttributeSource definition.
             RootBeanDefinition sourceDef = new
RootBeanDefinition(

    "org.springframework.transaction.annotation.AnnotationTrans
actionAttributeSource");
             sourceDef.setSource(eleSource);

    sourceDef.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
             String sourceName =
parserContext.getReaderContext().registerWithGeneratedName(sou
rceDef);

             // Create the TransactionInterceptor definition.
```

```
            RootBeanDefinition interceptorDef = new
RootBeanDefinition(TransactionInterceptor.class);
            interceptorDef.setSource(eleSource);

   interceptorDef.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
            registerTransactionManager(element,
interceptorDef);

   interceptorDef.getPropertyValues().add("transactionAttribut
eSource", new RuntimeBeanReference(sourceName));
            String interceptorName =
parserContext.getReaderContext().registerWithGeneratedName(int
erceptorDef);

            // Create the TransactionAttributeSourceAdvisor
definition.
            RootBeanDefinition advisorDef = new
RootBeanDefinition(BeanFactoryTransactionAttributeSourceAdviso
r.class);
            advisorDef.setSource(eleSource);

   advisorDef.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);

   advisorDef.getPropertyValues().add("transactionAttributeSou
rce", new RuntimeBeanReference(sourceName));

   advisorDef.getPropertyValues().add("adviceBeanName",
interceptorName);
            if (element.hasAttribute("order")) {
               advisorDef.getPropertyValues().add("order",
element.getAttribute("order"));
            }

   parserContext.getRegistry().registerBeanDefinition(txAdviso
rBeanName, advisorDef);

            CompositeComponentDefinition compositeDef = new
CompositeComponentDefinition(element.getTagName(), eleSource);
            compositeDef.addNestedComponent(new
BeanComponentDefinition(sourceDef, sourceName));
            compositeDef.addNestedComponent(new
BeanComponentDefinition(interceptorDef, interceptorName));
            compositeDef.addNestedComponent(new
BeanComponentDefinition(advisorDef, txAdvisorBeanName));
```

```
                parserContext.registerComponent(compositeDef);
            }
        }

public static void registerAutoProxyCreatorIfNecessary(
            ParserContext parserContext, Element sourceElement) {
        BeanDefinition beanDefinition =
AopConfigUtils.registerAutoProxyCreatorIfNecessary(parserConte
xt.getRegistry(), parserContext.extractSource(sourceElement));
        useClassProxyingIfNecessary(parserContext.getRegistry(),
sourceElement);
        registerComponentIfNecessary(beanDefinition,
parserContext);
    }

public static BeanDefinition
registerAutoProxyCreatorIfNecessary(
            BeanDefinitionRegistry registry, @Nullable Object
source) {

        return
registerOrEscalateApcAsRequired(InfrastructureAdvisorAutoProxy
Creator.class, registry, source);
    }

Cls=org.springframework.aop.framework.autoproxy.Infrastructure
AdvisorAutoProxyCreator
Registry= DefaultListableBeanFactory
private static BeanDefinition registerOrEscalateApcAsRequired(
            Class<?> cls, BeanDefinitionRegistry registry,
@Nullable Object source) {
        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_NAM
E, beanDefinition);// internalAutoProxyCreator
        return beanDefinition;
    }
```

InfrastructureAdvisorAutoProxyCreator