Spring注解驱动开发第35讲——声明式事务原理的源码分析

文章目录

写在前面

上一讲，我向大家简单介绍了一下注解版的声明式事务 ，相信大家已经会初步使用了。这一讲，咱们就从源码的角度分析一下声明式事务的原理，让大家对声明式事务的原理有一个更加深入的认识。

声明式事务的原理

其实，**要想知道声明式事务的原理，只需要搞清楚@EnableTransactionManagement注解给容器中注册了什么组件，以及这些组件工作时候的功能是什么就行了，一旦把这个研究透了，那么声明式事务的原理我们就清楚了。**

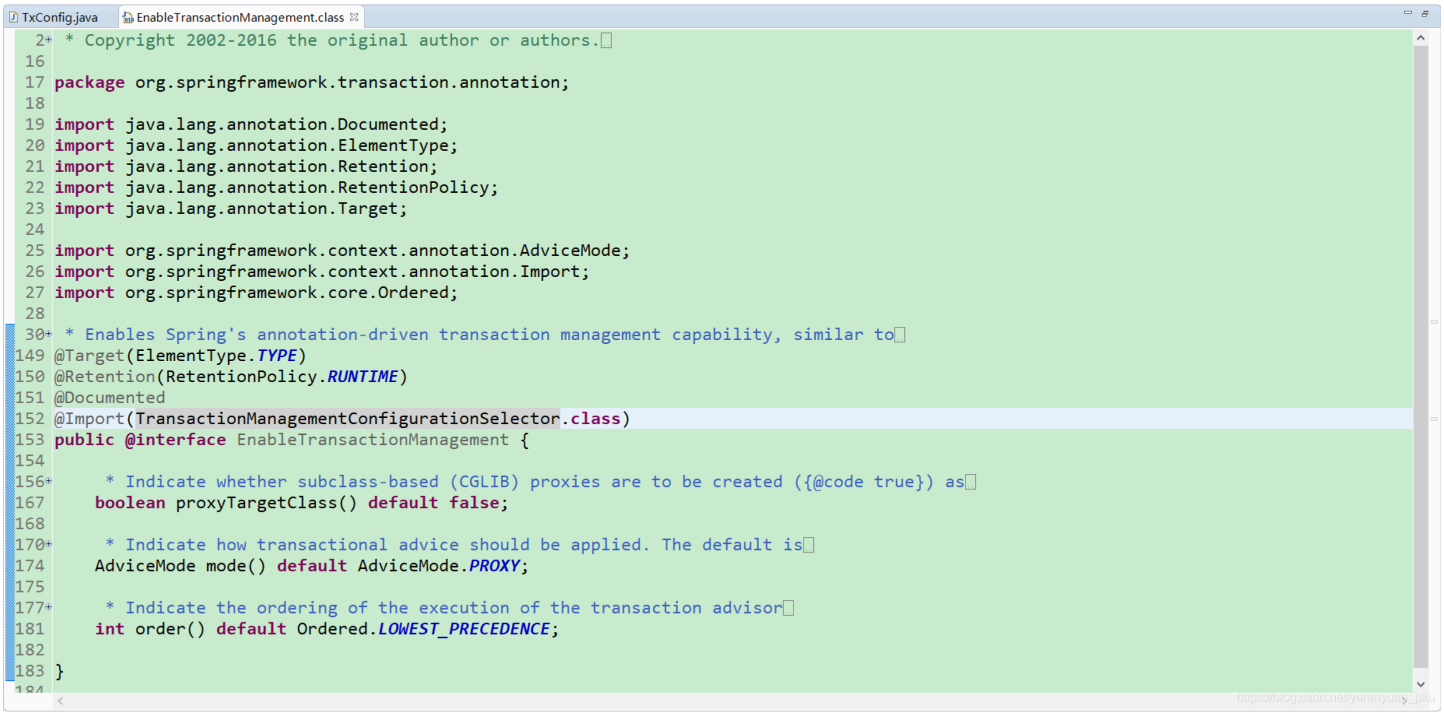
你有没有发现，咱们之前研究AOP的原理时，是从@EnableAspectJAutoProxy注解开始入手研究的，这时你就应该能想到，研究声明式事务的原理，也是应该从@EnableTransactionManagement注解开始入手研究。

其实，当你研究完声明式事务的原理，你就会发现这一过程与研究AOP原理 的过程是非常相似的，也可以说这俩原理几乎是一模一样的。

接下来，我们就要从@EnableTransactionManagement注解开始一步一步分析声明式事务的原理了。只不过这次不会像之前分析AOP原理一样，以debug的模式来分析，而是直接点进源码里面看源码来分析。

@EnableTransactionManagement注解利用TransactionManagementConfigurationSelector给容器中导入组件

在配置类上添加@EnableTransactionManagement注解，便能够开启基于注解的事务管理功能。那下面我们就来看一看它的源码，如下图所示。

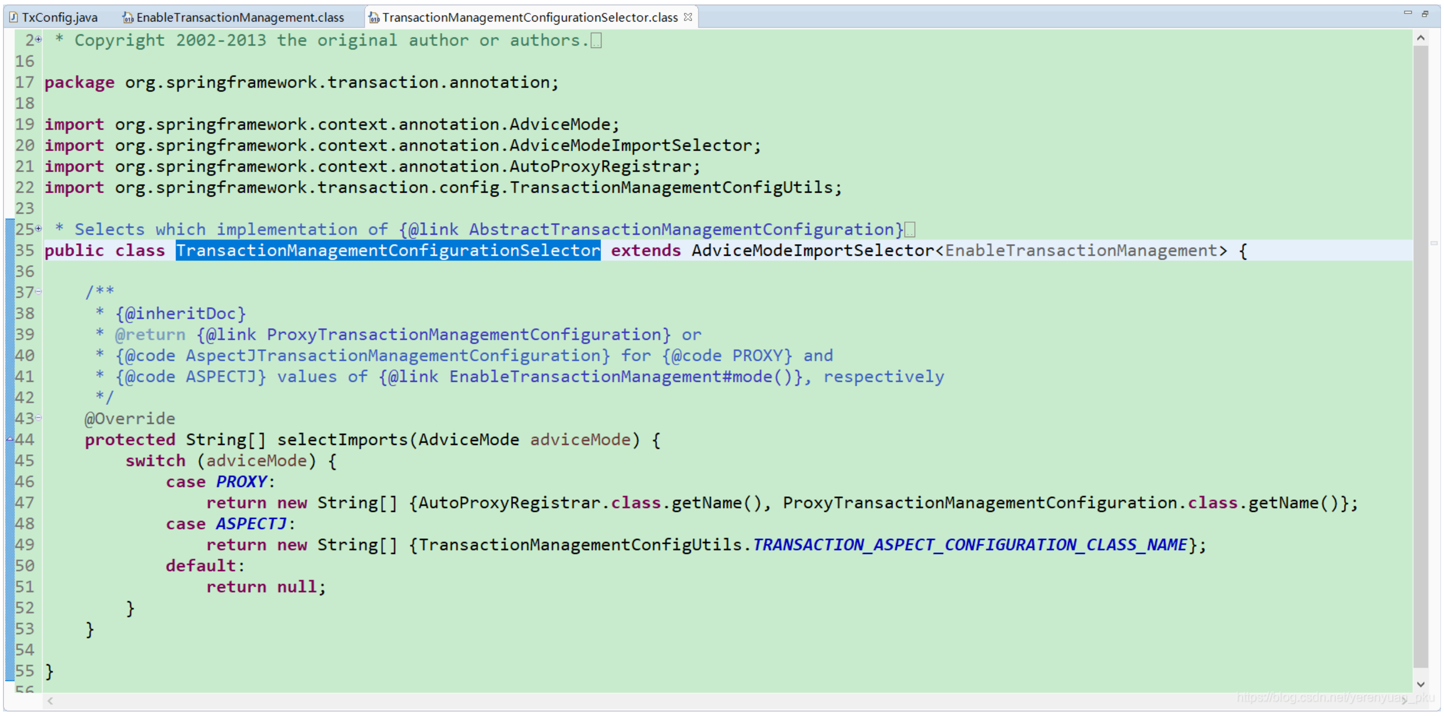


```java
 * Copyright 2002-2016 the original author or authors.

package org.springframework.transaction.annotation;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import org.springframework.context.annotation.AdviceMode;
import org.springframework.context.annotation.Import;
import org.springframework.core.Ordered;

 * Enables Spring's annotation-driven transaction management capability, similar to
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import(TransactionManagementConfigurationSelector.class)
public @interface EnableTransactionManagement {

     * Indicate whether subclass-based (CGLIB) proxies are to be created ({@code true}) as
    boolean proxyTargetClass() default false;

     * Indicate how transactional advice should be applied. The default is
    AdviceMode mode() default AdviceMode.PROXY;

     * Indicate the ordering of the execution of the transaction advisor
    int order() default Ordered.LOWEST_PRECEDENCE;

}
```

https://blog.csdn.net/yerenyuan_pku

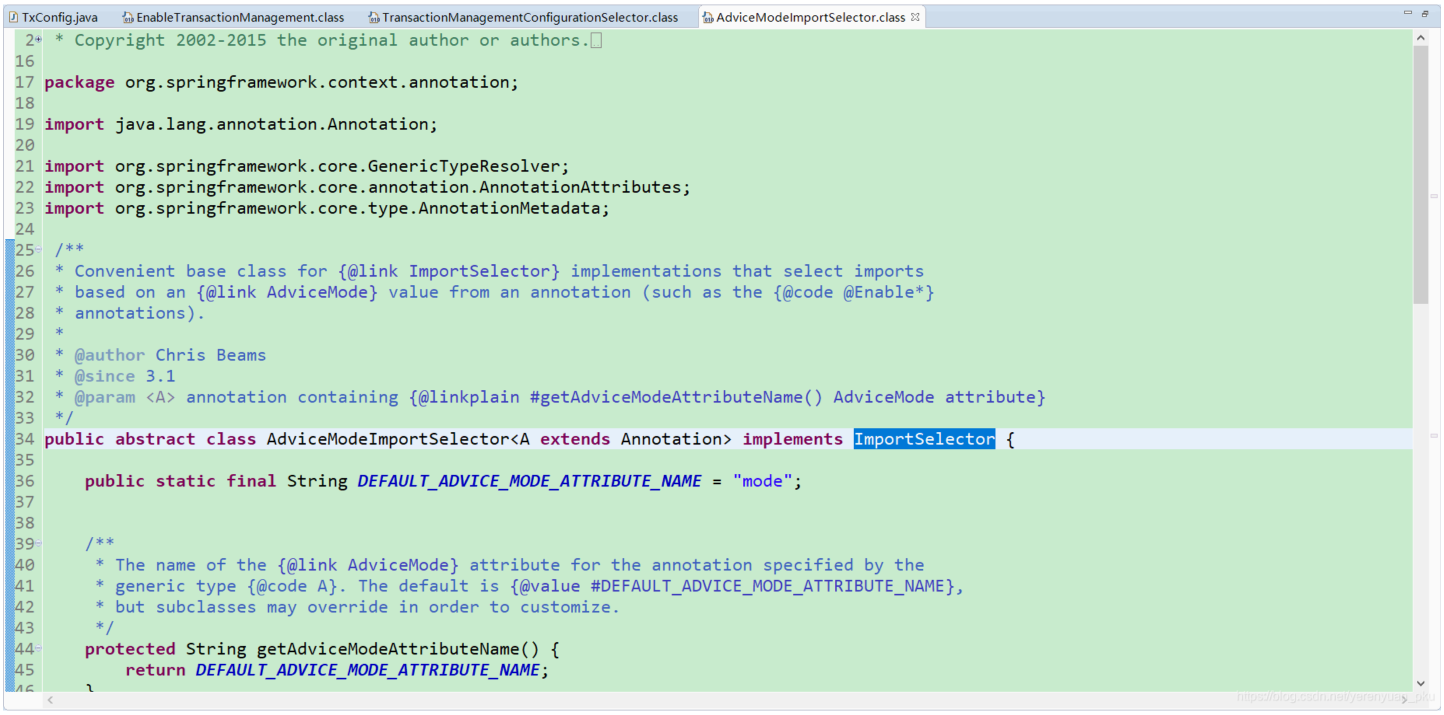
从源码中可以看出，@EnableTransactionManagement注解使用@Import注解给容器中引入了TransactionManagementConfigurationSelector组件。那这个TransactionManagementConfigurationSelector又是啥呢？它其实是一个ImportSelector。

这是怎么得出来的呢？我们可以点到TransactionManagementConfigurationSelector类中一看究竟，如下图所示，发现它继承了一个类，叫AdviceModeImportSelector。


TxConfig.java
EnableTransactionManagement.class
TransactionManagementConfigurationSelector.class

```
 * Copyright 2002-2013 the original author or authors.□

package org.springframework.transaction.annotation;

import org.springframework.context.annotation.AdviceMode;
import org.springframework.context.annotation.AdviceModeImportSelector;
import org.springframework.context.annotation.AutoProxyRegistrar;
import org.springframework.transaction.config.TransactionManagementConfigUtils;

 * Selects which implementation of {@link AbstractTransactionManagementConfiguration}□
public class TransactionManagementConfigurationSelector extends AdviceModeImportSelector<EnableTransactionManagement> {

    /**
     * {@inheritDoc}
     * @return {@link ProxyTransactionManagementConfiguration} or
     * {@code AspectJTransactionManagementConfiguration} for {@code PROXY} and
     * {@code ASPECTJ} values of {@link EnableTransactionManagement#mode()}, respectively
     */
    @Override
    protected String[] selectImports(AdviceMode adviceMode) {
        switch (adviceMode) {
            case PROXY:
                return new String[] {AutoProxyRegistrar.class.getName(), ProxyTransactionManagementConfiguration.class.getName()};
            case ASPECTJ:
                return new String[] {TransactionManagementConfigUtils.TRANSACTION_ASPECT_CONFIGURATION_CLASS_NAME};
            default:
                return null;
        }
    }

}
```

然后再次点到AdviceModeImportSelector类中，如下图所示，发现它实现了一个接口，叫ImportSelector。


TxConfig.java
EnableTransactionManagement.class
TransactionManagementConfigurationSelector.class
AdviceModeImportSelector.class

```
 * Copyright 2002-2015 the original author or authors.□

package org.springframework.context.annotation;

import java.lang.annotation.Annotation;

import org.springframework.core.GenericTypeResolver;
import org.springframework.core.annotation.AnnotationAttributes;
import org.springframework.core.type.AnnotationMetadata;

/**
 * Convenient base class for {@link ImportSelector} implementations that select imports
 * based on an {@link AdviceMode} value from an annotation (such as the {@code @Enable*}
 * annotations).
 *
 * @author Chris Beams
 * @since 3.1
 * @param <A> annotation containing {@linkplain #getAdviceModeAttributeName() AdviceMode attribute}
 */
public abstract class AdviceModeImportSelector<A extends Annotation> implements ImportSelector {

    public static final String DEFAULT_ADVICE_MODE_ATTRIBUTE_NAME = "mode";


    /**
     * The name of the {@link AdviceMode} attribute for the annotation specified by the
     * generic type {@code A}. The default is {@value #DEFAULT_ADVICE_MODE_ATTRIBUTE_NAME},
     * but subclasses may override in order to customize.
     */
    protected String getAdviceModeAttributeName() {
        return DEFAULT_ADVICE_MODE_ATTRIBUTE_NAME;
```

这时，你总该相信TransactionManagementConfigurationSelector是一个ImportSelector了吧！其实，我们已经对ImportSelector接口有了一定程度的认识了，如果你还不知道它，那么不妨看看我写的《Spring注解驱动开发第9讲——在@Import注解中使用ImportSelector接口导入bean》这篇博客。

说到底，其实它是用于给容器中快速导入一些组件的，到底要导入哪些组件，就看它会返回哪些要导入到容器中的组件的全类名。

我们可以看一下TransactionManagementConfigurationSelector类的源码，看看它里面到底是怎么写的。其实在上面我们就看清楚该类的源码了，在它里面会做一个switch判断，如果adviceMode是PROXY，那么就会返回一个 `String[]` ，该String数组如下所示：

```
new String[] {AutoProxyRegistrar.class.getName(), ProxyTransactionManagementConfiguration.class.getName()};
```

AI写代码java运行

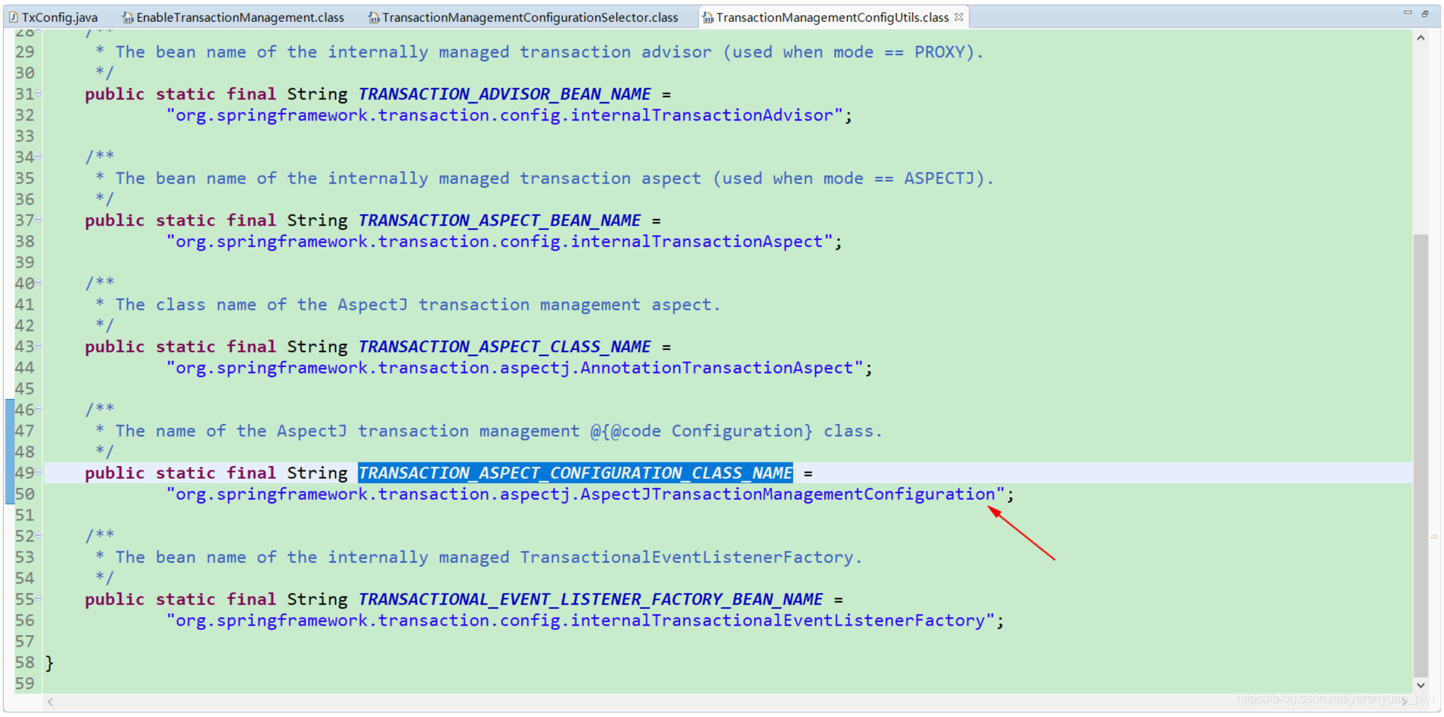
这说明会向容器中导入AutoProxyRegistrar和ProxyTransactionManagementConfiguration这两个组件。

如果adviceMode是ASPECTJ，那么便会返回如下这样一个 `String[]` 。

```
new String[] {TransactionManagementConfigUtils.TRANSACTION_ASPECT_CONFIGURATION_CLASS_NAME};
```

AI写代码java运行

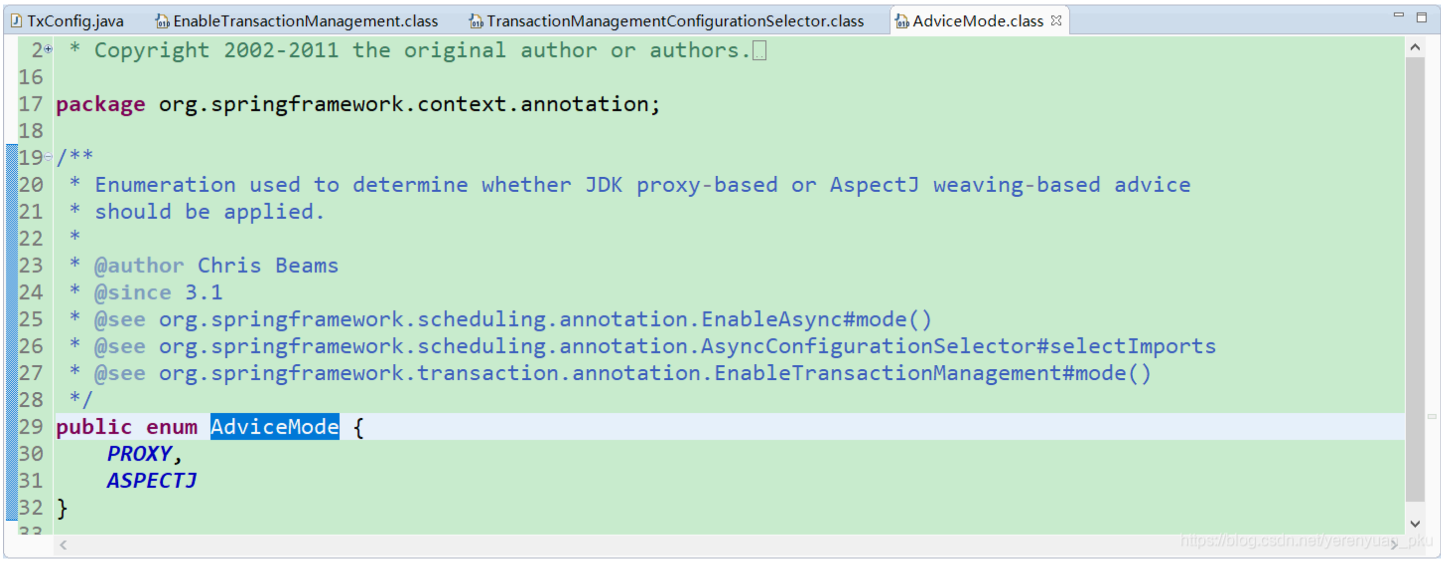
点 `TRANSACTION_ASPECT_CONFIGURATION_CLASS_NAME` 一下，可以看到，它其实就是AspectJTransactionManagementConfiguration类的全类名，如下图所示。


TxConfig.java
EnableTransactionManagement.class
TransactionManagementConfigurationSelector.class
TransactionManagementConfigUtils.class
https://blog.csdn.net/yerenyuan_pku

```
    /**
     * The bean name of the internally managed transaction advisor (used when mode == PROXY).
     */
    public static final String TRANSACTION_ADVISOR_BEAN_NAME =
            "org.springframework.transaction.config.internalTransactionAdvisor";

    /**
     * The bean name of the internally managed transaction aspect (used when mode == ASPECTJ).
     */
    public static final String TRANSACTION_ASPECT_BEAN_NAME =
            "org.springframework.transaction.config.internalTransactionAspect";

    /**
     * The class name of the AspectJ transaction management aspect.
     */
    public static final String TRANSACTION_ASPECT_CLASS_NAME =
            "org.springframework.transaction.aspectj.AnnotationTransactionAspect";

    /**
     * The name of the AspectJ transaction management @{@code Configuration} class.
     */
    public static final String TRANSACTION_ASPECT_CONFIGURATION_CLASS_NAME =
            "org.springframework.transaction.aspectj.AspectJTransactionManagementConfiguration";

    /**
     * The bean name of the internally managed TransactionalEventListenerFactory.
     */
    public static final String TRANSACTIONAL_EVENT_LISTENER_FACTORY_BEAN_NAME =
            "org.springframework.transaction.config.internalTransactionalEventListenerFactory";

}
```

也就是说，如果adviceMode是ASPECTJ，那么就会向容器中导入一个AspectJTransactionManagementConfiguration组件。只可惜，它和我们研究声明式事务的原理没有半毛钱的关系。

那么问题来了，AdviceMode又是个啥呢？点它，发现它是一个枚举，如下图所示。


TxConfig.java
EnableTransactionManagement.class
TransactionManagementConfigurationSelector.class
AdviceMode.class
https://blog.csdn.net/yerenyuan_pku

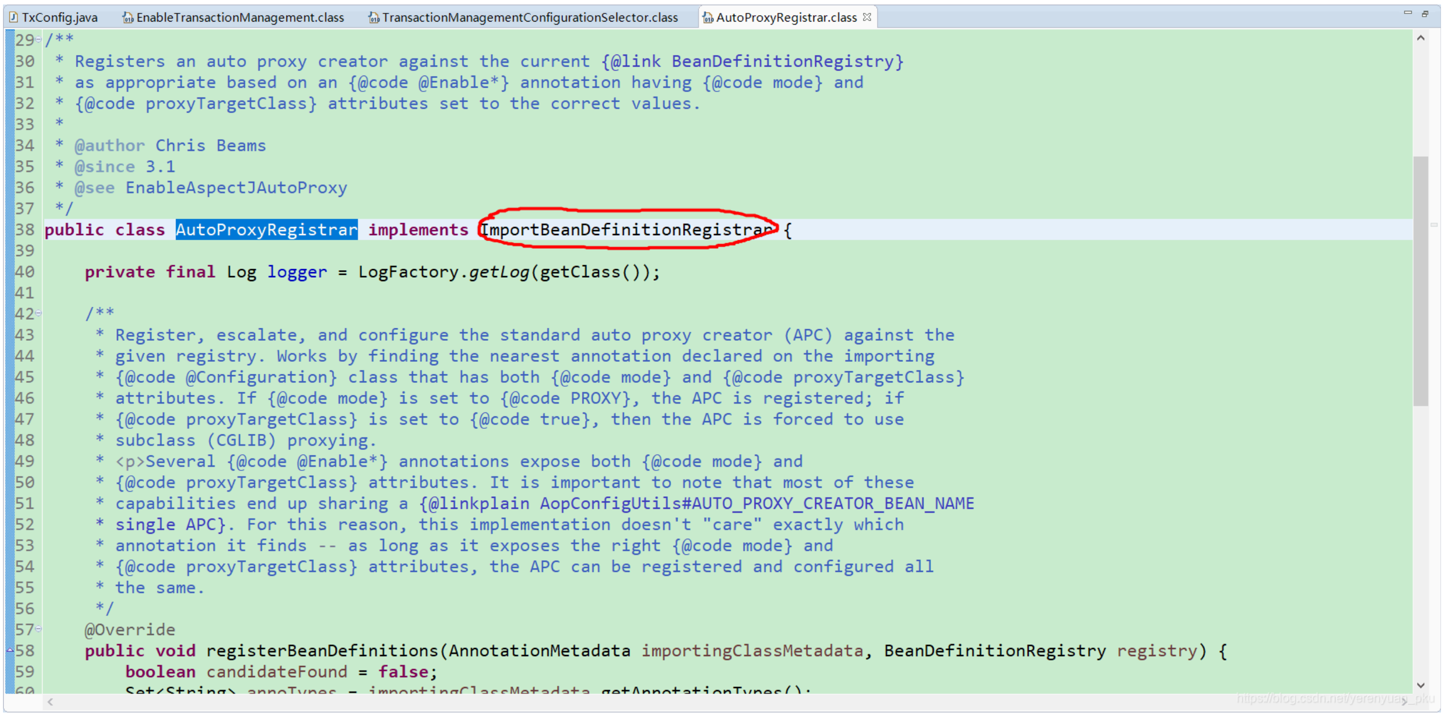
```
 * Copyright 2002-2011 the original author or authors.

package org.springframework.context.annotation;

/**
 * Enumeration used to determine whether JDK proxy-based or AspectJ weaving-based advice
 * should be applied.
 *
 * @author Chris Beams
 * @since 3.1
 * @see org.springframework.scheduling.annotation.EnableAsync#mode()
 * @see org.springframework.scheduling.annotation.AsyncConfigurationSelector#selectImports
 * @see org.springframework.transaction.annotation.EnableTransactionManagement#mode()
 */
public enum AdviceMode {
    PROXY,
    ASPECTJ
}
```

这个枚举有啥子用呢？我们可以再来看一下@EnableTransactionManagement注解的源码，发现它里面会定义一个mode属性，且其默认值就是 `AdviceMode.PROXY` 。既然如此，那么便会进入到TransactionManagementConfigurationSelector类的switch语句的 `case PROXY` 选项中，这时，就会向容器中快速导入两个组件，一个叫AutoProxyRegistrar，一个叫ProxyTransactionManagementConfiguration。

接下来，我们便要来分析这两个组件的功能了，只要分析清楚了，声明式事务的原理就呼之欲出了。

导入的第一个组件（即AutoProxyRegistrar），它到底做了些啥呢？

我们来看导入的第一个组件，即AutoProxyRegistrar，它都做了些啥？我们点进去该类里面看一看，发现它实现了一个接口，叫ImportBeanDefinitionRegistrar。

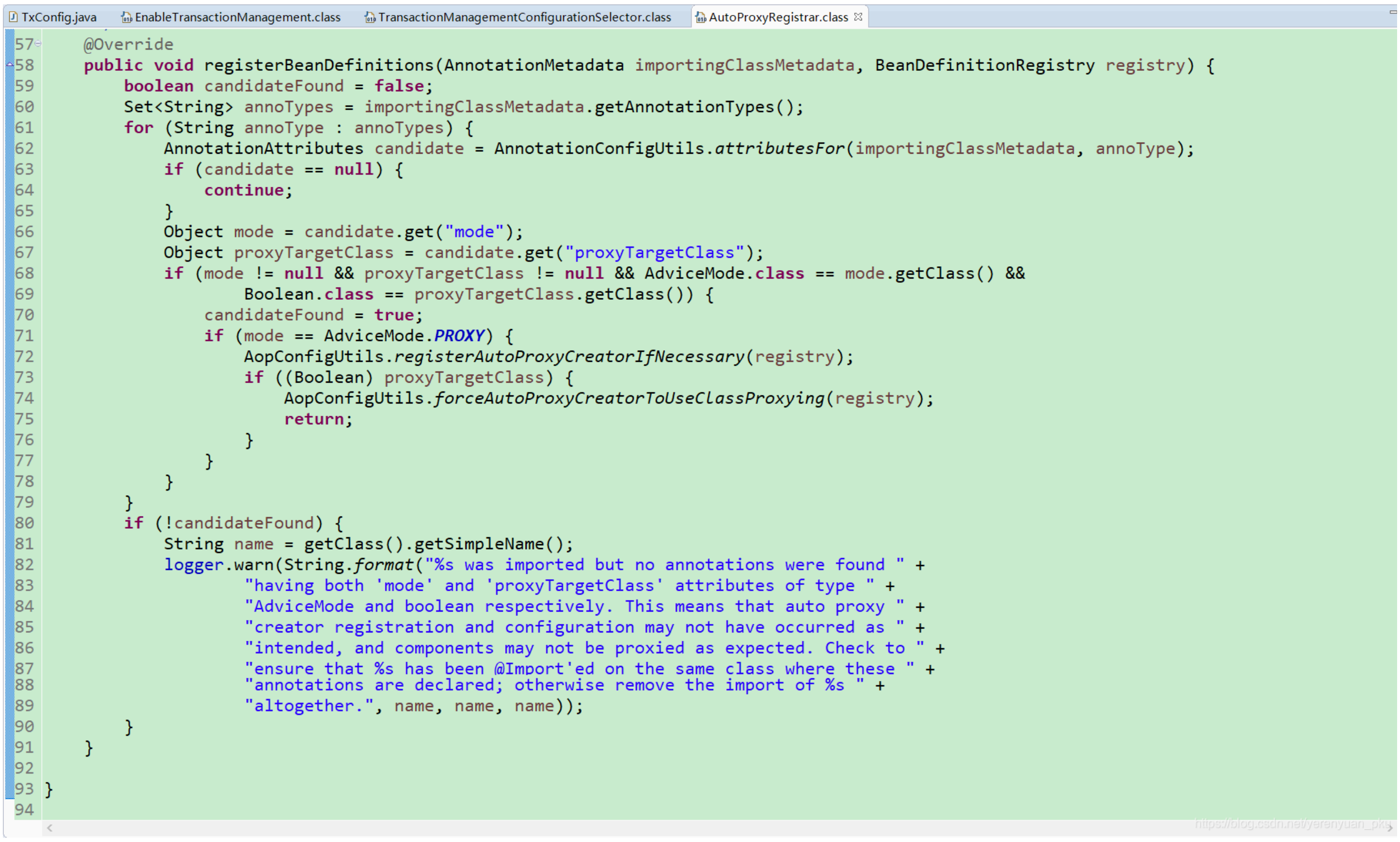

TxConfig.java
EnableTransactionManagement.class
TransactionManagementConfigurationSelector.class
AutoProxyRegistrar.class

```java
/**
 * Registers an auto proxy creator against the current {@link BeanDefinitionRegistry}
 * as appropriate based on an {@code @Enable*} annotation having {@code mode} and
 * {@code proxyTargetClass} attributes set to the correct values.
 *
 * @author Chris Beams
 * @since 3.1
 * @see EnableAspectJAutoProxy
 */
public class AutoProxyRegistrar implements ImportBeanDefinitionRegistrar {

    private final Log logger = LogFactory.getLog(getClass());

    /**
     * Register, escalate, and configure the standard auto proxy creator (APC) against the
     * given registry. Works by finding the nearest annotation declared on the importing
     * {@code @Configuration} class that has both {@code mode} and {@code proxyTargetClass}
     * attributes. If {@code mode} is set to {@code PROXY}, the APC is registered; if
     * {@code proxyTargetClass} is set to {@code true}, then the APC is forced to use
     * subclass (CGLIB) proxying.
     * <p>Several {@code @Enable*} annotations expose both {@code mode} and
     * {@code proxyTargetClass} attributes. It is important to note that most of these
     * capabilities end up sharing a {@linkplain AopConfigUtils#AUTO_PROXY_CREATOR_BEAN_NAME
     * single APC}. For this reason, this implementation doesn't "care" exactly which
     * annotation it finds -- as long as it exposes the right {@code mode} and
     * {@code proxyTargetClass} attributes, the APC can be registered and configured all
     * the same.
     */
    @Override
    public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {
        boolean candidateFound = false;
        Set<String> annoTypes = importingClassMetadata.getAnnotationTypes();
```

关于该接口的详细介绍，你可以参考我写的《Spring注解驱动开发第10讲——在@Import注解中使用ImportBeanDefinitionRegistrar向容器中注册bean》这篇博客。说到底，这个AutoProxyRegistrar组件其实就是用来向容器中注册bean的，那你就应该清楚，最终会调用该组件的registerBeanDefinitions()方法来向容器中注册bean。

AutoProxyRegistrar向容器中注入了一个自动代理创建器，即InfrastructureAdvisorAutoProxyCreator

那么会向容器中注册什么bean呢？我们仔细地看一下AutoProxyRegistrar类中的registerBeanDefinitions()方法，如下图所示。


TxConfig.java
EnableTransactionManagement.class
TransactionManagementConfigurationSelector.class
AutoProxyRegistrar.class

```java
    @Override
    public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {
        boolean candidateFound = false;
        Set<String> annoTypes = importingClassMetadata.getAnnotationTypes();
        for (String annoType : annoTypes) {
            AnnotationAttributes candidate = AnnotationConfigUtils.attributesFor(importingClassMetadata, annoType);
            if (candidate == null) {
                continue;
            }
            Object mode = candidate.get("mode");
            Object proxyTargetClass = candidate.get("proxyTargetClass");
            if (mode != null && proxyTargetClass != null && AdviceMode.class == mode.getClass() &&
                    Boolean.class == proxyTargetClass.getClass()) {
                candidateFound = true;
                if (mode == AdviceMode.PROXY) {
                    AopConfigUtils.registerAutoProxyCreatorIfNecessary(registry);
                    if ((Boolean) proxyTargetClass) {
                        AopConfigUtils.forceAutoProxyCreatorToUseClassProxying(registry);
                        return;
                    }
                }
            }
        }
        if (!candidateFound) {
            String name = getClass().getSimpleName();
            logger.warn(String.format("%s was imported but no annotations were found " +
                    "having both 'mode' and 'proxyTargetClass' attributes of type " +
                    "AdviceMode and boolean respectively. This means that auto proxy " +
                    "creator registration and configuration may not have occurred as " +
                    "intended, and components may not be proxied as expected. Check to " +
                    "ensure that %s has been @Import'ed on the same class where these " +
                    "annotations are declared; otherwise remove the import of %s " +
                    "altogether.", name, name, name));
        }
    }

}
```

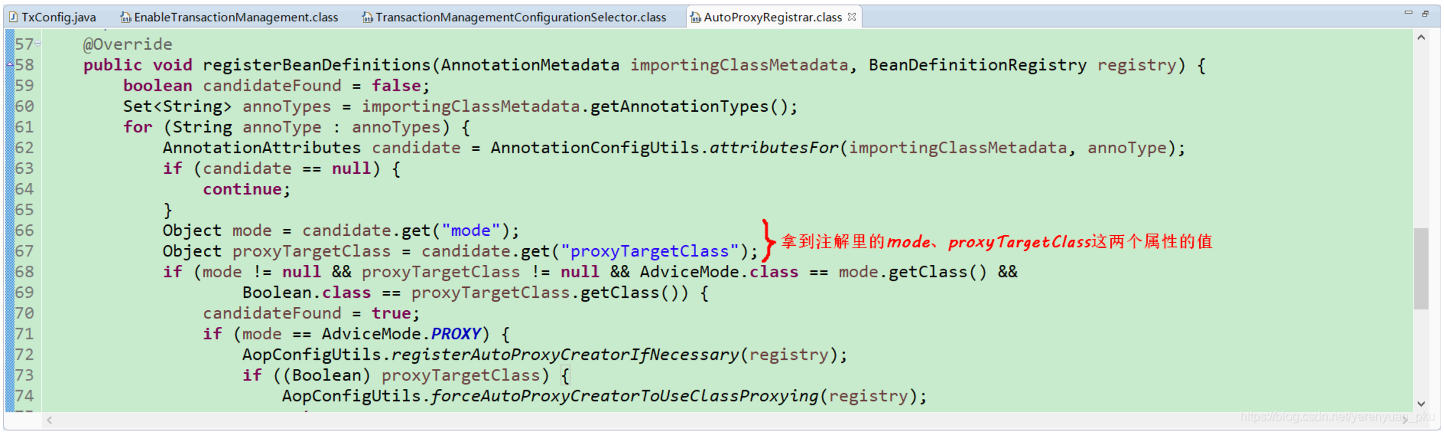
这里，我就粗略地分析一下该方法，真的就只是粗略地分析一下，如有错误，可以告知笔者改正。

在该方法中先是通过如下一行代码来获取各种注解类型，这儿需要特别注意的是，这里是拿到所有的注解类型，而不是只拿@EnableAspectJAutoProxy这个类型的。因为mode、proxyTargetClass等属性会直接影响到代理的方式，而拥有这些属性的注解至少有@EnableTransactionManagement、@EnableAsync以及@EnableCaching等等，甚至还有启用AOP的注解，即@EnableAspectJAutoProxy，它也能设置proxyTargetClass这个属性的值，因此也会产生关联影响。

```java
Set<String> annoTypes = importingClassMetadata.getAnnotationTypes();
```

AI写代码java运行

然后是拿到注解里的mode、proxyTargetClass这两个属性的值，如下图所示。


TxConfig.java
EnableTransactionManagement.class
TransactionManagementConfigurationSelector.class
AutoProxyRegistrar.class

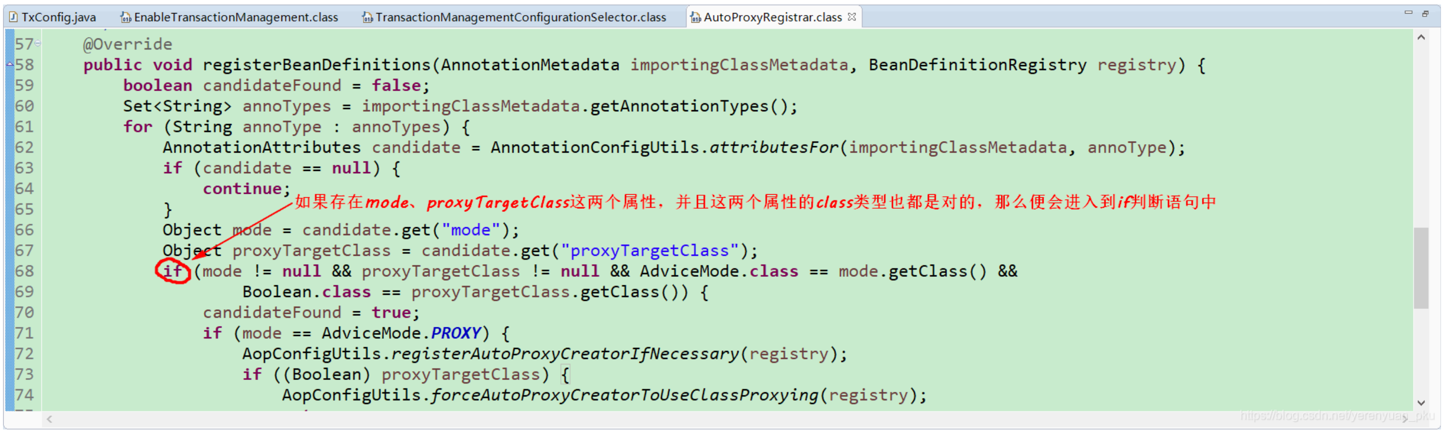
```java
    @Override
    public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {
        boolean candidateFound = false;
        Set<String> annoTypes = importingClassMetadata.getAnnotationTypes();
        for (String annoType : annoTypes) {
            AnnotationAttributes candidate = AnnotationConfigUtils.attributesFor(importingClassMetadata, annoType);
            if (candidate == null) {
                continue;
            }
            Object mode = candidate.get("mode");
            Object proxyTargetClass = candidate.get("proxyTargetClass");
            if (mode != null && proxyTargetClass != null && AdviceMode.class == mode.getClass() &&
                    Boolean.class == proxyTargetClass.getClass()) {
                candidateFound = true;
                if (mode == AdviceMode.PROXY) {
                    AopConfigUtils.registerAutoProxyCreatorIfNecessary(registry);
                    if ((Boolean) proxyTargetClass) {
                        AopConfigUtils.forceAutoProxyCreatorToUseClassProxying(registry);
```

拿到注解里的mode、proxyTargetClass这两个属性的值

https://blog.csdn.net/yerenyuan_pku

注意，如果这儿的注解是@Configuration或者别的其他注解的话，那么获取到的这俩属性的值就是null了。

接着做一个判断，如果存在mode、proxyTargetClass这两个属性，并且这两个属性的class类型也都是对的，那么便会进入到if判断语句中，这样，其余注解就相当于都被挡在外面了。


TxConfig.java
EnableTransactionManagement.class
TransactionManagementConfigurationSelector.class
AutoProxyRegistrar.class

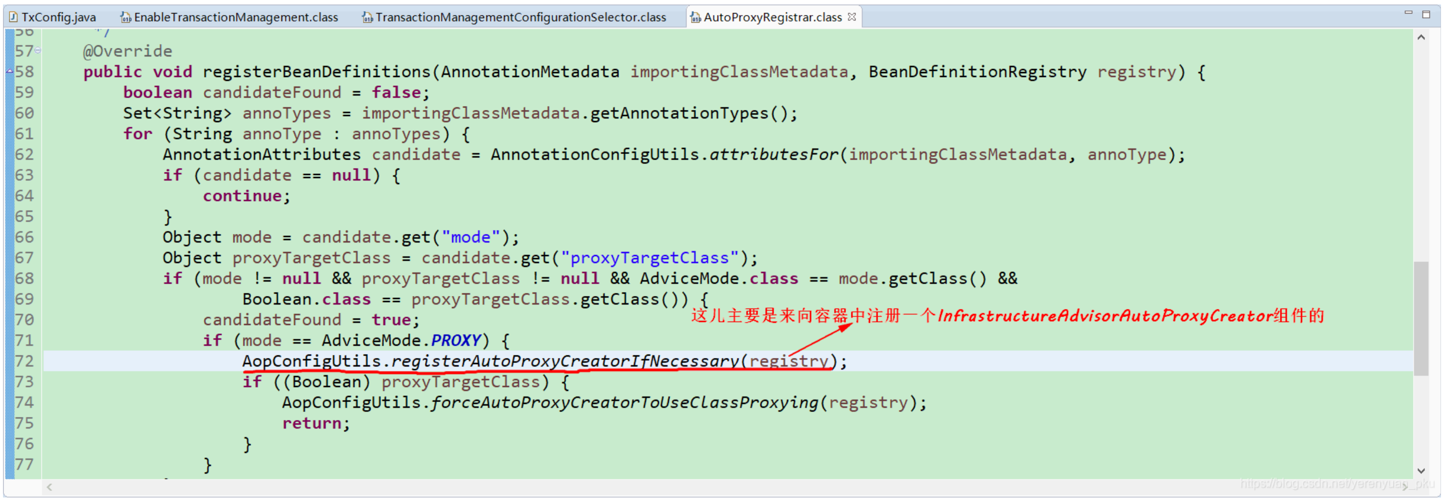
```java
    @Override
    public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {
        boolean candidateFound = false;
        Set<String> annoTypes = importingClassMetadata.getAnnotationTypes();
        for (String annoType : annoTypes) {
            AnnotationAttributes candidate = AnnotationConfigUtils.attributesFor(importingClassMetadata, annoType);
            if (candidate == null) {
                continue;
            }
            Object mode = candidate.get("mode");
            Object proxyTargetClass = candidate.get("proxyTargetClass");
            if (mode != null && proxyTargetClass != null && AdviceMode.class == mode.getClass() &&
                    Boolean.class == proxyTargetClass.getClass()) {
                candidateFound = true;
                if (mode == AdviceMode.PROXY) {
                    AopConfigUtils.registerAutoProxyCreatorIfNecessary(registry);
                    if ((Boolean) proxyTargetClass) {
                        AopConfigUtils.forceAutoProxyCreatorToUseClassProxying(registry);
```

如果存在mode、proxyTargetClass这两个属性，并且这两个属性的class类型也都是对的，那么便会进入到if判断语句中

https://blog.csdn.net/yerenyuan_pku

要是真进入到了if判断语句中，是不是意味着找到了候选的注解（例如@EnableTransactionManagement）呢？你仔细想一下，是不是这回事。找到了候选的注解之后，就将candidateFound标识置为true。

紧接着会再做一个判断，即判断找到的候选注解中的mode属性的值是否为 `AdviceMode.PROXY`，若是则会调用我们熟悉的AopConfigUtils工具类的registerAutoProxyCreatorIfNecessary方法。相信大家也很熟悉这个方法了，它主要是来向容器中注册一个InfrastructureAdvisorAutoProxyCreator组件的。


TxConfig.java
EnableTransactionManagement.class
TransactionManagementConfigurationSelector.class
AutoProxyRegistrar.class

```java
    @Override
    public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {
        boolean candidateFound = false;
        Set<String> annoTypes = importingClassMetadata.getAnnotationTypes();
        for (String annoType : annoTypes) {
            AnnotationAttributes candidate = AnnotationConfigUtils.attributesFor(importingClassMetadata, annoType);
            if (candidate == null) {
                continue;
            }
            Object mode = candidate.get("mode");
            Object proxyTargetClass = candidate.get("proxyTargetClass");
            if (mode != null && proxyTargetClass != null && AdviceMode.class == mode.getClass() &&
                    Boolean.class == proxyTargetClass.getClass()) {
                candidateFound = true;
                if (mode == AdviceMode.PROXY) {
                    AopConfigUtils.registerAutoProxyCreatorIfNecessary(registry);
                    if ((Boolean) proxyTargetClass) {
                        AopConfigUtils.forceAutoProxyCreatorToUseClassProxying(registry);
                        return;
                    }
                }
```

这儿主要是来向容器中注册一个InfrastructureAdvisorAutoProxyCreator组件的

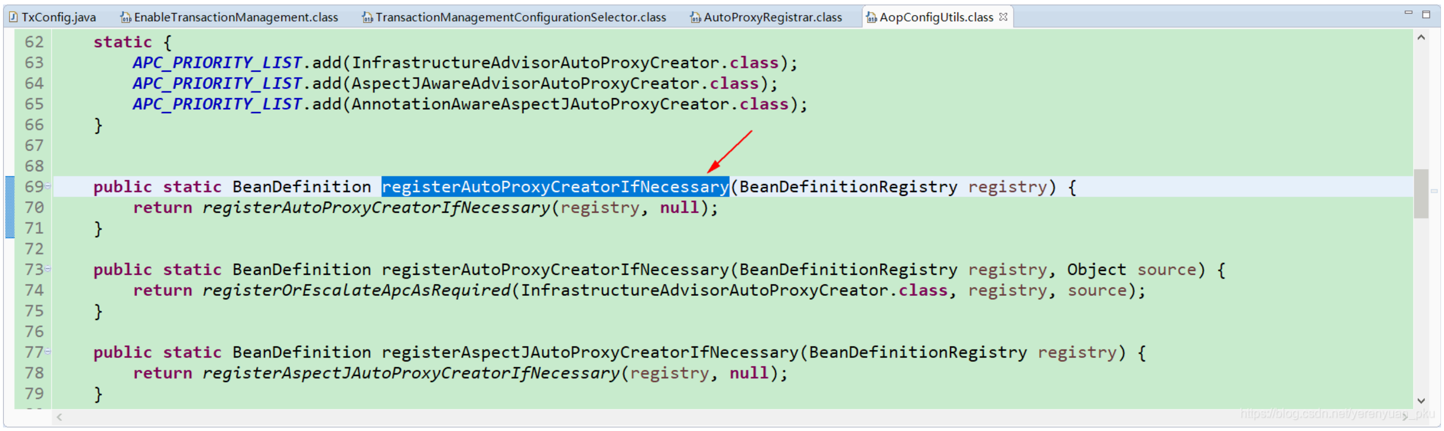
https://blog.csdn.net/yerenyuan_pku

我是为啥知道的这么清楚的呢？待会再来告诉你，哈哈哈😊

我们继续往下看AutoProxyRegistrar类的registerBeanDefinitions()方法。这时，又会做一个判断，要是找到的候选注解设置了proxyTargetClass这个属性的值，并且值为true，那么便会进入到下面的if判断语句中，**看要不要强制使用CGLIB的方式**。

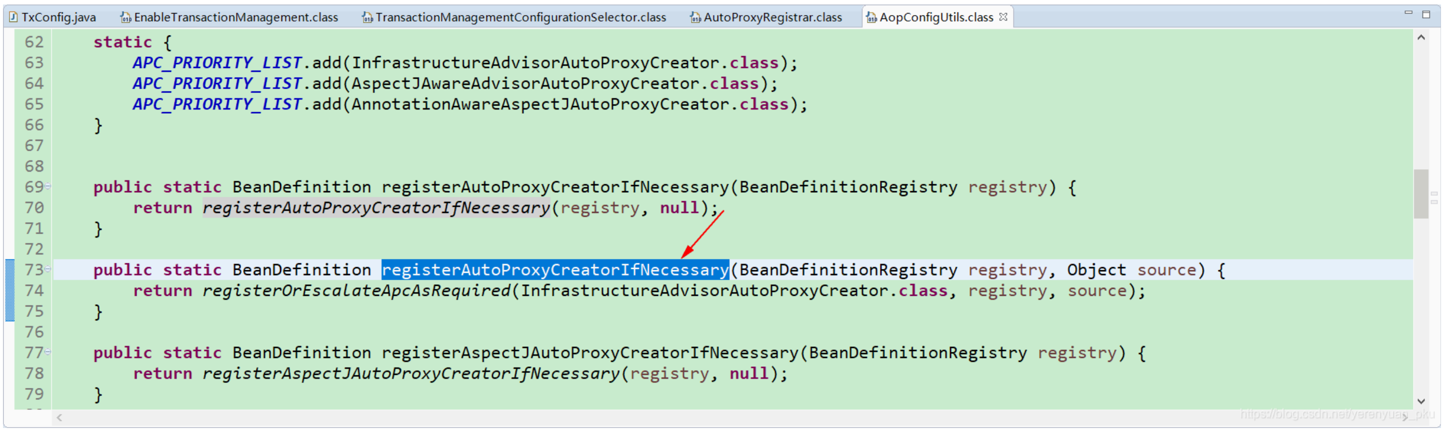
如果此时找到的候选注解是@EnableTransactionManagement，想一想会发生什么事情？查看该注解的源码，你会发现它里面就拥有一个proxyTargetClass属性，并且其默认值是false。所以此时压根就不会进入到if判断语句中，而只会调用我们熟悉的AopConfigUtils工具类的registerAutoProxyCreatorIfNecessary方法。

这个咱们再熟悉不过的registerAutoProxyCreatorIfNecessary方法会向容器中注册什么呢？上面我也说到了，它会向容器中注册一个InfrastructureAdvisorAutoProxyCreator组件，即自动代理创建器。我是咋知道的呢？只能是看源码呗，还能是什么。点进去registerAutoProxyCreatorIfNecessary方法中，如下图所示，可以看到这个方法又调用了一个同名的重载方法。


TxConfig.java
EnableTransactionManagement.class
TransactionManagementConfigurationSelector.class
AutoProxyRegistrar.class
AopConfigUtils.class

```java
62      static {
63          APC_PRIORITY_LIST.add(InfrastructureAdvisorAutoProxyCreator.class);
64          APC_PRIORITY_LIST.add(AspectJAwareAdvisorAutoProxyCreator.class);
65          APC_PRIORITY_LIST.add(AnnotationAwareAspectJAutoProxyCreator.class);
66      }
67
68
69      public static BeanDefinition registerAutoProxyCreatorIfNecessary(BeanDefinitionRegistry registry) {
70          return registerAutoProxyCreatorIfNecessary(registry, null);
71      }
72
73      public static BeanDefinition registerAutoProxyCreatorIfNecessary(BeanDefinitionRegistry registry, Object source) {
74          return registerOrEscalateApcAsRequired(InfrastructureAdvisorAutoProxyCreator.class, registry, source);
75      }
76
77      public static BeanDefinition registerAspectJAutoProxyCreatorIfNecessary(BeanDefinitionRegistry registry) {
78          return registerAspectJAutoProxyCreatorIfNecessary(registry, null);
79      }
```

然后点进去同名的重载方法中，如下图所示，可以看到这个方法又调用了一个registerOrEscalateApcAsRequired方法，而且还传入了一个参数，即 `InfrastructureAdvisorAutoProxyCreator.class` 。

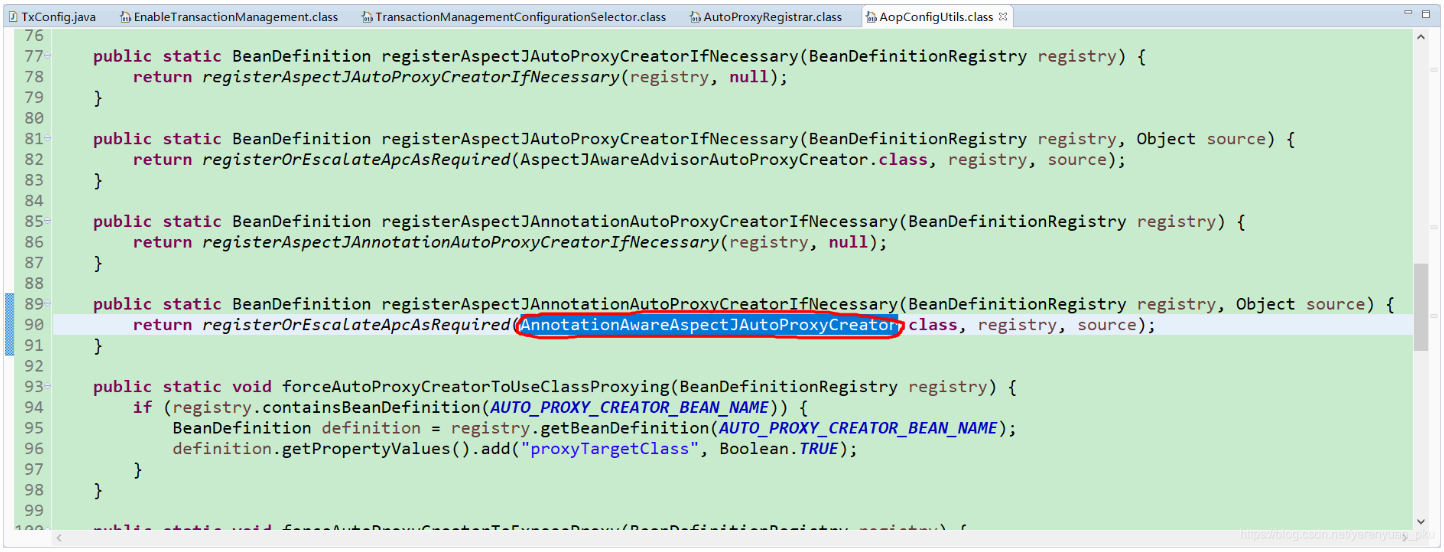

TxConfig.java
EnableTransactionManagement.class
TransactionManagementConfigurationSelector.class
AutoProxyRegistrar.class
AopConfigUtils.class

```java
62      static {
63          APC_PRIORITY_LIST.add(InfrastructureAdvisorAutoProxyCreator.class);
64          APC_PRIORITY_LIST.add(AspectJAwareAdvisorAutoProxyCreator.class);
65          APC_PRIORITY_LIST.add(AnnotationAwareAspectJAutoProxyCreator.class);
66      }
67
68
69      public static BeanDefinition registerAutoProxyCreatorIfNecessary(BeanDefinitionRegistry registry) {
70          return registerAutoProxyCreatorIfNecessary(registry, null);
71      }
72
73      public static BeanDefinition registerAutoProxyCreatorIfNecessary(BeanDefinitionRegistry registry, Object source) {
74          return registerOrEscalateApcAsRequired(InfrastructureAdvisorAutoProxyCreator.class, registry, source);
75      }
76
77      public static BeanDefinition registerAspectJAutoProxyCreatorIfNecessary(BeanDefinitionRegistry registry) {
78          return registerAspectJAutoProxyCreatorIfNecessary(registry, null);
79      }
```

你现在该知道调用AopConfigUtils工具类的registerAutoProxyCreatorIfNecessary方法会向容器中注册什么组件了吧！

现在我们可以得出这样一个结论：**导入的第一个组件（即AutoProxyRegistrar）向容器中注入了一个自动代理创建器，即InfrastructureAdvisorAutoProxyCreator**。

其实，大家可以好好看一下AopConfigUtils工具类的源码，因为它里面还有一个我们非常熟悉的东东，这个东东是什么呢？我就不卖关子了，直接查看AopConfigUtils工具类第90行源码，你就能看到异常熟悉的东东了，它就是AnnotationAwareAspectJAutoProxyCreator，如下图所示。

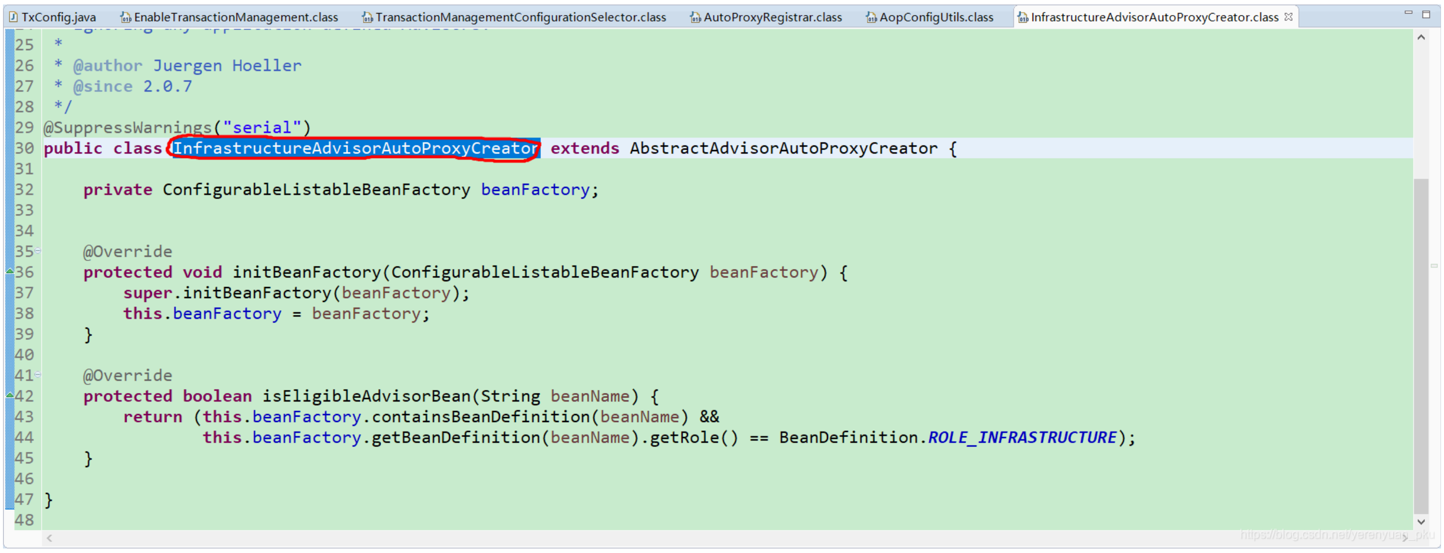

TxConfig.java
EnableTransactionManagement.class
TransactionManagementConfigurationSelector.class
AutoProxyRegistrar.class
AopConfigUtils.class

```java
76
77      public static BeanDefinition registerAspectJAutoProxyCreatorIfNecessary(BeanDefinitionRegistry registry) {
78          return registerAspectJAutoProxyCreatorIfNecessary(registry, null);
79      }
80
81      public static BeanDefinition registerAspectJAutoProxyCreatorIfNecessary(BeanDefinitionRegistry registry, Object source) {
82          return registerOrEscalateApcAsRequired(AspectJAwareAdvisorAutoProxyCreator.class, registry, source);
83      }
84
85      public static BeanDefinition registerAspectJAnnotationAutoProxyCreatorIfNecessary(BeanDefinitionRegistry registry) {
86          return registerAspectJAnnotationAutoProxyCreatorIfNecessary(registry, null);
87      }
88
89      public static BeanDefinition registerAspectJAnnotationAutoProxyCreatorIfNecessary(BeanDefinitionRegistry registry, Object source) {
90          return registerOrEscalateApcAsRequired(AnnotationAwareAspectJAutoProxyCreator.class, registry, source);
91      }
92
93      public static void forceAutoProxyCreatorToUseClassProxying(BeanDefinitionRegistry registry) {
94          if (registry.containsBeanDefinition(AUTO_PROXY_CREATOR_BEAN_NAME)) {
95              BeanDefinition definition = registry.getBeanDefinition(AUTO_PROXY_CREATOR_BEAN_NAME);
96              definition.getPropertyValues().add("proxyTargetClass", Boolean.TRUE);
97          }
98      }
99
100     public static void forceAutoProxyCreatorToExposeProxy(BeanDefinitionRegistry registry) {
```

大家还记得它是什么吗？这个时候你就需要回顾一下以前学习的内容了。当初咱们在研究AOP的原理时，不是得出了这样一个结论吗？**即@EnableAspectJAutoProxy注解会利用AspectJAutoProxyRegistrar向容器中注入一个AnnotationAwareAspectJAutoProxyCreator组件**。现在，你总算该记起来了吧😊

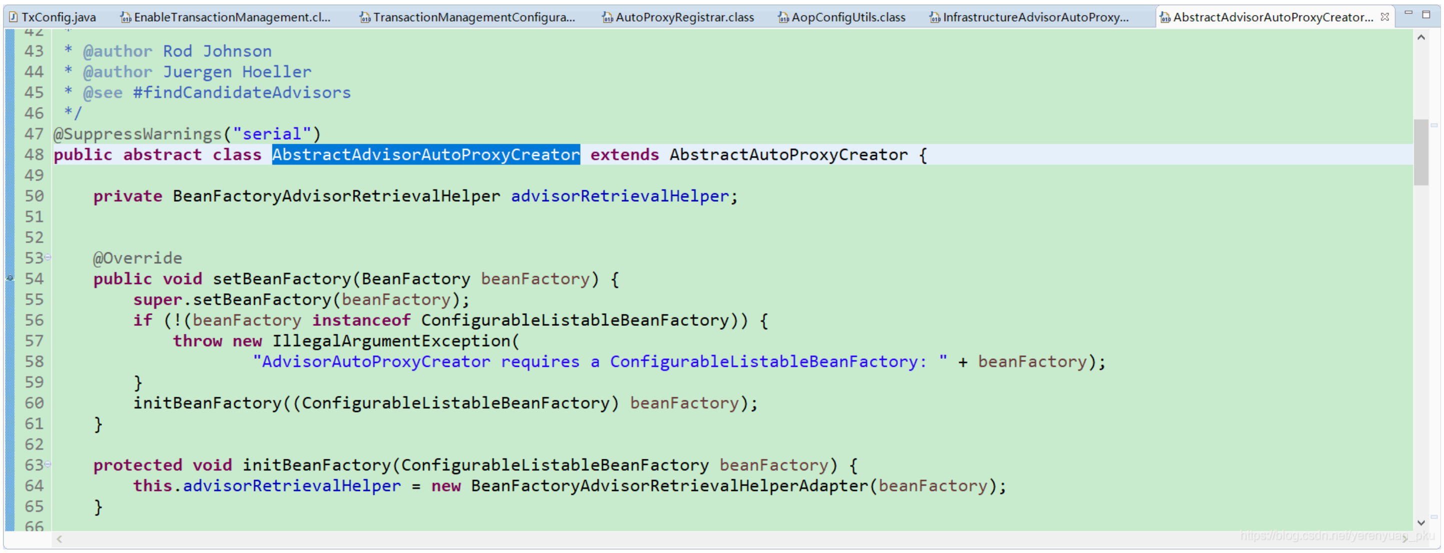
声明式事务的原理跟AOP的原理很相似，只不过对于声明式事务原理而言，它注入的是InfrastructureAdvisorAutoProxyCreator组件而已。我们都知道，在研究AOP原理时，AnnotationAwareAspectJAutoProxyCreator实质上是一个后置处理器，那么InfrastructureAdvisorAutoProxyCreator实质上又是一个什么呢？也会是一个后置处理器吗？

点进去InfrastructureAdvisorAutoProxyCreator类里面去看一看，如下图所示，发现它继承了一个AbstractAdvisorAutoProxyCreator类。


TxConfig.java
EnableTransactionManagement.class
TransactionManagementConfigurationSelector.class
AutoProxyRegistrar.class
AopConfigUtils.class
InfrastructureAdvisorAutoProxyCreator.class

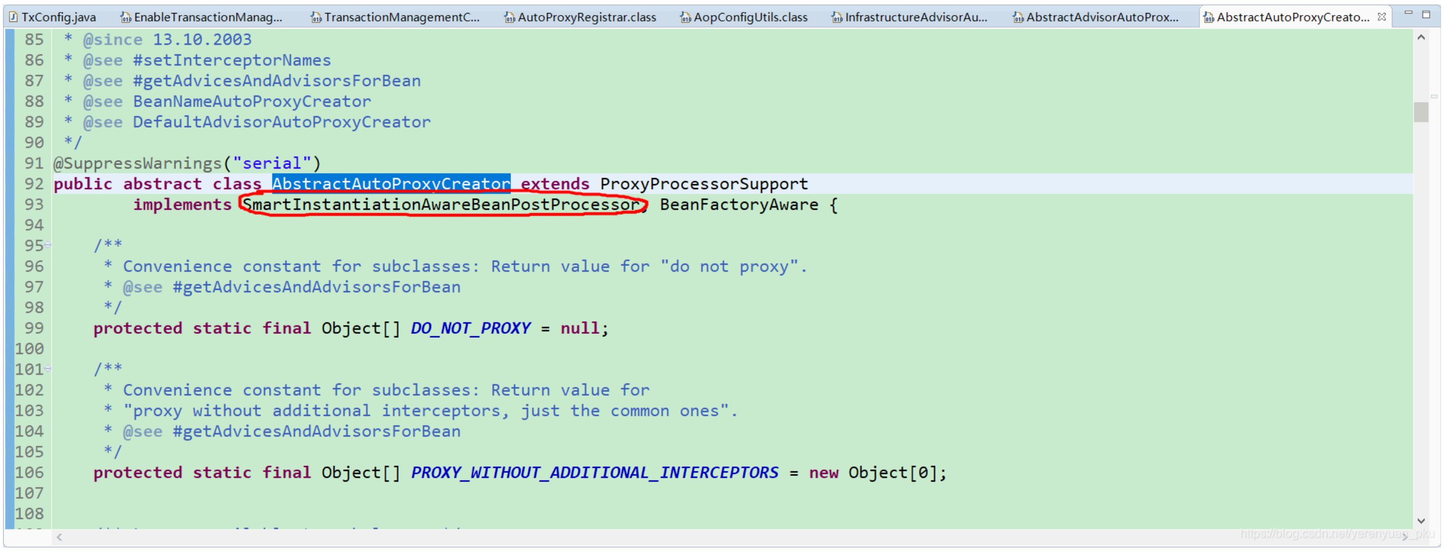
```
 *
 * @author Juergen Hoeller
 * @since 2.0.7
 */
@SuppressWarnings("serial")
public class InfrastructureAdvisorAutoProxyCreator extends AbstractAdvisorAutoProxyCreator {

    private ConfigurableListableBeanFactory beanFactory;


    @Override
    protected void initBeanFactory(ConfigurableListableBeanFactory beanFactory) {
        super.initBeanFactory(beanFactory);
        this.beanFactory = beanFactory;
    }

    @Override
    protected boolean isEligibleAdvisorBean(String beanName) {
        return (this.beanFactory.containsBeanDefinition(beanName) &&
                this.beanFactory.getBeanDefinition(beanName).getRole() == BeanDefinition.ROLE_INFRASTRUCTURE);
    }

}
```

然后再点进去AbstractAdvisorAutoProxyCreator类里面去看一看，如下图所示，发现它继承了一个AbstractAutoProxyCreator类。


TxConfig.java
EnableTransactionManagement.cl...
TransactionManagementConfigura...
AutoProxyRegistrar.class
AopConfigUtils.class
InfrastructureAdvisorAutoProxy...
AbstractAdvisorAutoProxyCreator...

```
 * @author Rod Johnson
 * @author Juergen Hoeller
 * @see #findCandidateAdvisors
 */
@SuppressWarnings("serial")
public abstract class AbstractAdvisorAutoProxyCreator extends AbstractAutoProxyCreator {

    private BeanFactoryAdvisorRetrievalHelper advisorRetrievalHelper;


    @Override
    public void setBeanFactory(BeanFactory beanFactory) {
        super.setBeanFactory(beanFactory);
        if (!(beanFactory instanceof ConfigurableListableBeanFactory)) {
            throw new IllegalArgumentException(
                    "AdvisorAutoProxyCreator requires a ConfigurableListableBeanFactory: " + beanFactory);
        }
        initBeanFactory((ConfigurableListableBeanFactory) beanFactory);
    }

    protected void initBeanFactory(ConfigurableListableBeanFactory beanFactory) {
        this.advisorRetrievalHelper = new BeanFactoryAdvisorRetrievalHelperAdapter(beanFactory);
    }
```

接着再点进去AbstractAutoProxyCreator类里面去看一看，如下图所示，发现它实现了一个SmartInstantiationAwareBeanPostProcessor接口。


TxConfig.java
EnableTransactionManag...
TransactionManagementC...
AutoProxyRegistrar.class
AopConfigUtils.class
InfrastructureAdvisorAu...
AbstractAdvisorAutoProx...
AbstractAutoProxyCreato...

```
 * @since 13.10.2003
 * @see #setInterceptorNames
 * @see #getAdvicesAndAdvisorsForBean
 * @see BeanNameAutoProxyCreator
 * @see DefaultAdvisorAutoProxyCreator
 */
@SuppressWarnings("serial")
public abstract class AbstractAutoProxyCreator extends ProxyProcessorSupport
        implements SmartInstantiationAwareBeanPostProcessor, BeanFactoryAware {

    /**
     * Convenience constant for subclasses: Return value for "do not proxy".
     * @see #getAdvicesAndAdvisorsForBean
     */
    protected static final Object[] DO_NOT_PROXY = null;

    /**
     * Convenience constant for subclasses: Return value for
     * "proxy without additional interceptors, just the common ones".
     * @see #getAdvicesAndAdvisorsForBean
     */
    protected static final Object[] PROXY_WITHOUT_ADDITIONAL_INTERCEPTORS = new Object[0];

```

这说明注入的InfrastructureAdvisorAutoProxyCreator组件同样也是一个后置处理器。接下来我们就来分析一下该组件的功能。

InfrastructureAdvisorAutoProxyCreator组件的功能

在这一小节中，我们来粗略地分析一下注入的InfrastructureAdvisorAutoProxyCreator组件到底都做了些什么。

其实，它做的事情也很简单，和之前研究AOP原理时向容器中注入的AnnotationAwareAspectJAutoProxyCreator组件所做的事情基本上没差别，只是利用后置处理器机制在对象创建以后进行包装，然后返回一个代理对象，并且该代理对象里面会存有所有的增强器。最后，代理对象执行目标方法，在此过程中会利用拦截器的链式机制，依次进入每一个拦截器中进行执行。

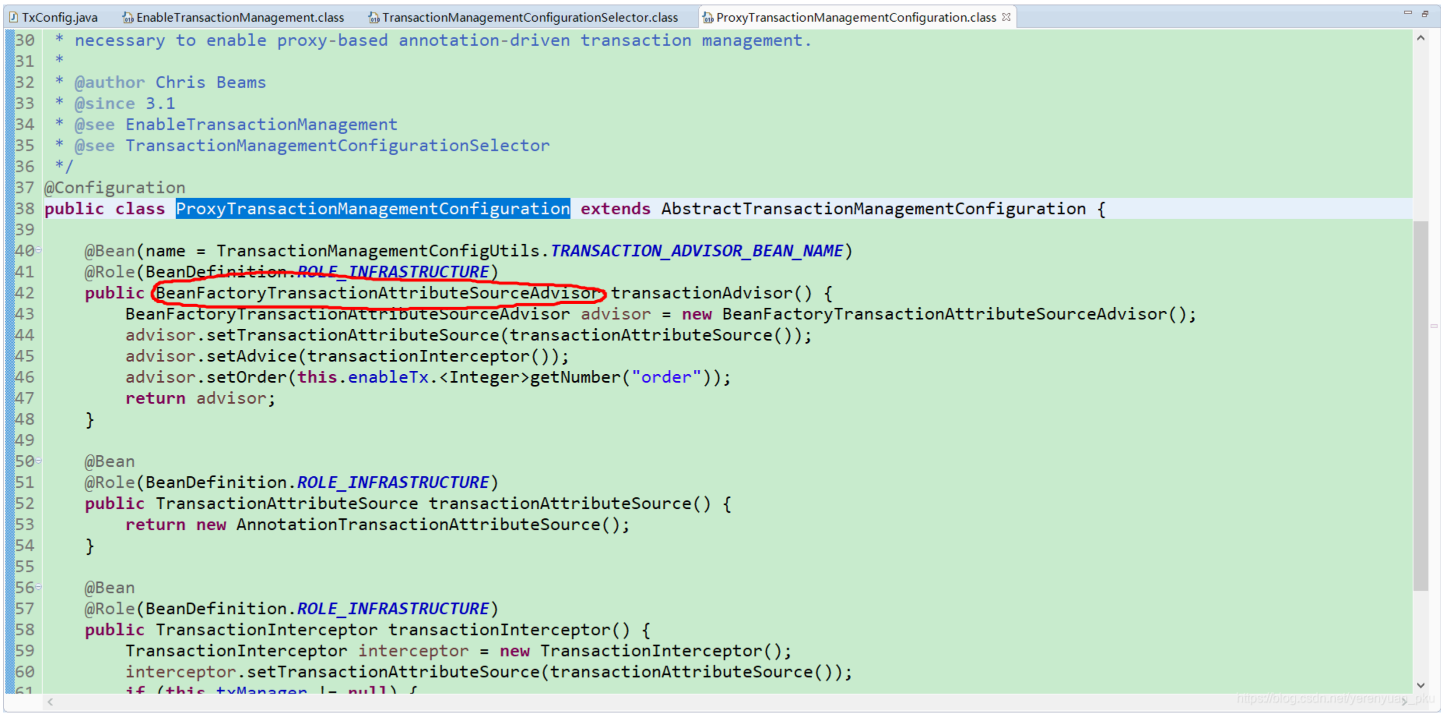
这儿，我也只是寥寥几笔概括了一下，并没有仔细地分析，主要是之前我在研究AOP原理的时候，详细分析过了，而且是一步一步地认真分析，这消耗了我大量的精力与时间，令我倍感疲惫。

导入的第二个组件（即ProxyTransactionManagementConfiguration），它又到底做了些啥呢？

接下来，我们再来看导入的第二个组件，即ProxyTransactionManagementConfiguration，它又做了些啥？

向容器中注册事务增强器

点进去ProxyTransactionManagementConfiguration类里面去看一看，很快你就会发现它是一个配置类，它会利用@Bean注解向容器中注册各种组件，而且注册的第一个组件就是BeanFactoryTransactionAttributeSourceAdvisor，这个Advisor可是事务的核心内容，可以暂时称之为事务增强器。


TxConfig.java
EnableTransactionManagement.class
TransactionManagementConfigurationSelector.class
ProxyTransactionManagementConfiguration.class

```java
 * necessary to enable proxy-based annotation-driven transaction management.
 *
 * @author Chris Beams
 * @since 3.1
 * @see EnableTransactionManagement
 * @see TransactionManagementConfigurationSelector
 */
@Configuration
public class ProxyTransactionManagementConfiguration extends AbstractTransactionManagementConfiguration {

    @Bean(name = TransactionManagementConfigUtils.TRANSACTION_ADVISOR_BEAN_NAME)
    @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
    public BeanFactoryTransactionAttributeSourceAdvisor transactionAdvisor() {
        BeanFactoryTransactionAttributeSourceAdvisor advisor = new BeanFactoryTransactionAttributeSourceAdvisor();
        advisor.setTransactionAttributeSource(transactionAttributeSource());
        advisor.setAdvice(transactionInterceptor());
        advisor.setOrder(this.enableTx.<Integer>getNumber("order"));
        return advisor;
    }

    @Bean
    @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
    public TransactionAttributeSource transactionAttributeSource() {
        return new AnnotationTransactionAttributeSource();
    }

    @Bean
    @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
    public TransactionInterceptor transactionInterceptor() {
        TransactionInterceptor interceptor = new TransactionInterceptor();
        interceptor.setTransactionAttributeSource(transactionAttributeSource());
        if (this.txManager != null) {
```

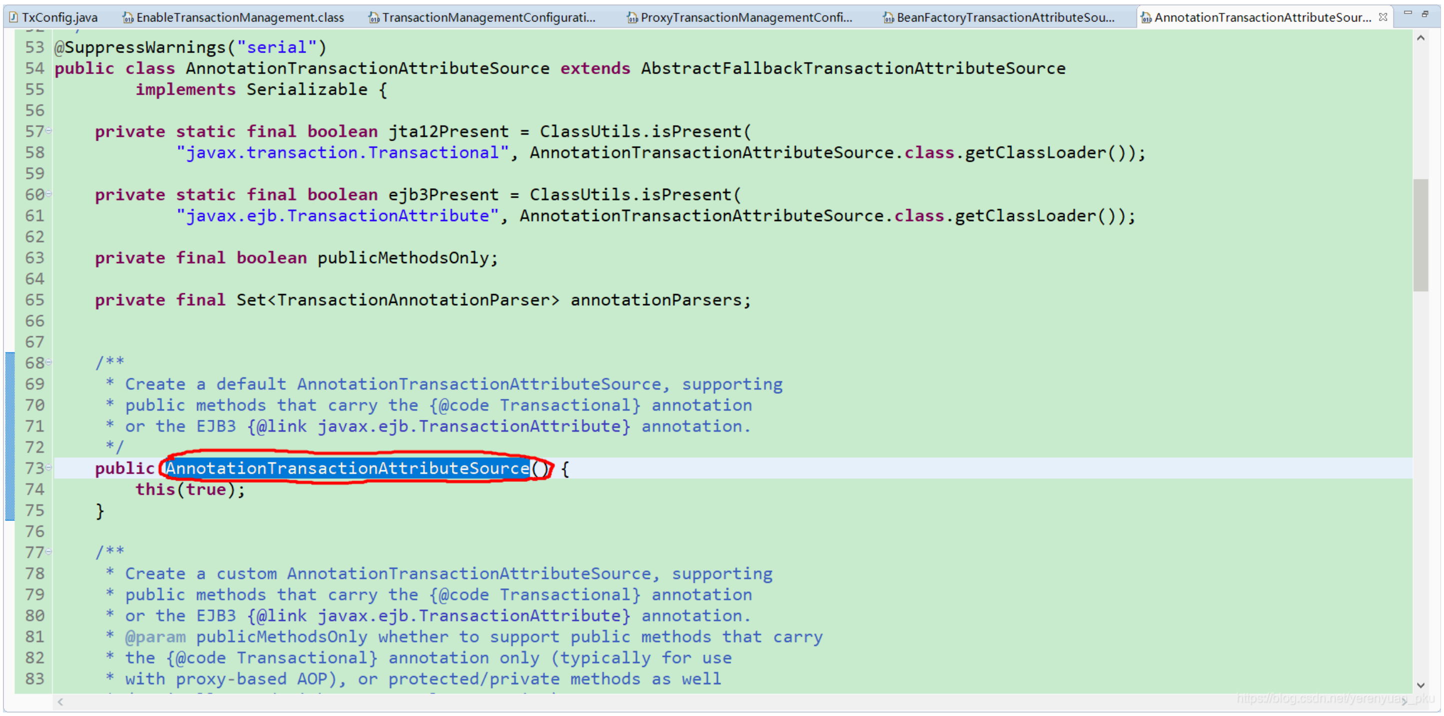
总之一句话，以上配置类会利用@Bean注解向容器中注册一个事务增强器。

在向容器中注册事务增强器时，需要用到事务属性源

那么这个所谓的事务增强器又是什么呢？从上面的配置类中可以看出，在向容器中注册事务增强器时，它会需要一个TransactionAttributeSource，翻译过来应该是事务属性源。

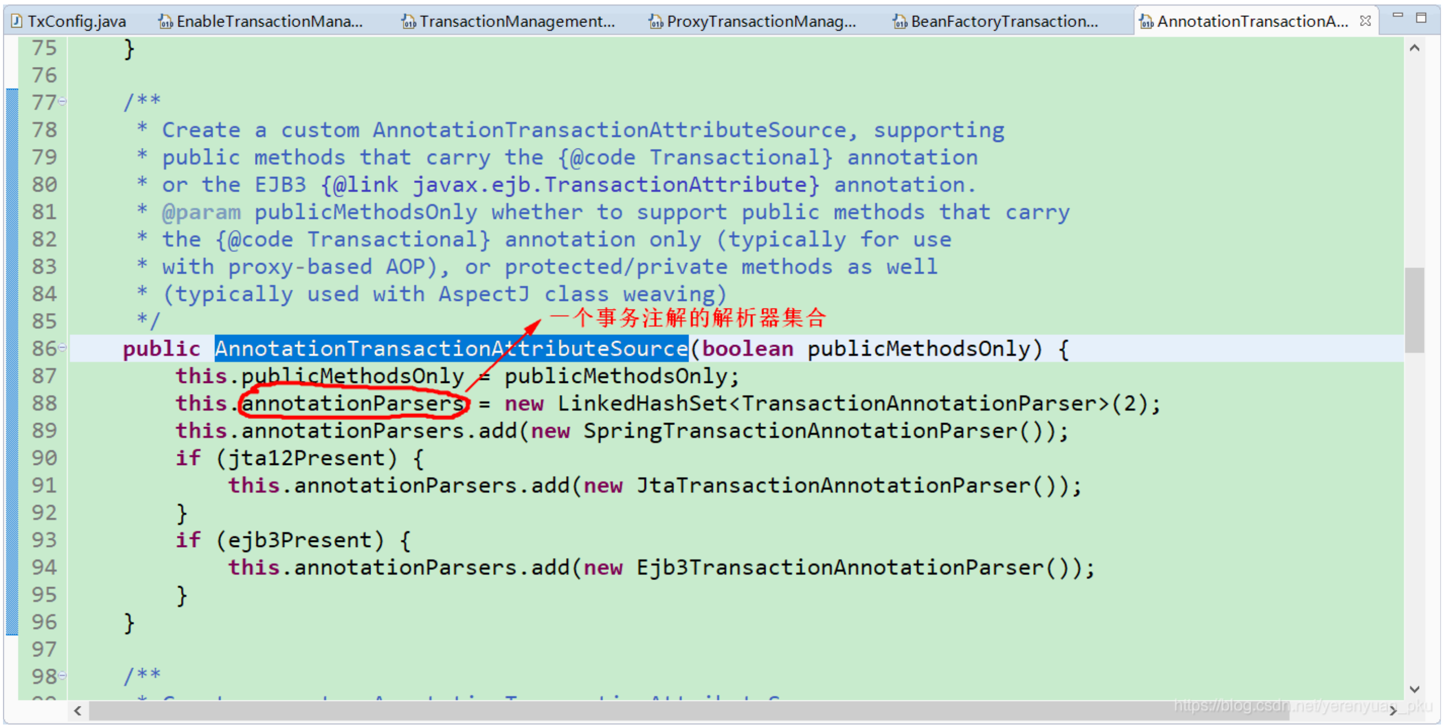
很快，你就会发现所需的TransactionAttributeSource又是容器中的一个bean，而且从transactionAttributeSource方法中可以看出，它是new出来了一个AnnotationTransactionAttributeSource对象。这个是重点，它是基于注解驱动的事务管理的事务属性源，和@Transactional注解相关，也是现在使用得最多的方式，其基本作用是遇上比如@Transactional注解标注的方法时，此类会分析此事务注解。

然后，点进AnnotationTransactionAttributeSource类的无参构造方法中去看一看，发现该方法又调用了如下一个this(true)方法，即本类的另一个重载的有参构造方法。


TxConfig.java
EnableTransactionManagement.class
TransactionManagementConfigurati...
ProxyTransactionManagementConfi...
BeanFactoryTransactionAttributeSou...
AnnotationTransactionAttributeSour...

```java
@SuppressWarnings("serial")
public class AnnotationTransactionAttributeSource extends AbstractFallbackTransactionAttributeSource
        implements Serializable {

    private static final boolean jta12Present = ClassUtils.isPresent(
            "javax.transaction.Transactional", AnnotationTransactionAttributeSource.class.getClassLoader());

    private static final boolean ejb3Present = ClassUtils.isPresent(
            "javax.ejb.TransactionAttribute", AnnotationTransactionAttributeSource.class.getClassLoader());

    private final boolean publicMethodsOnly;

    private final Set<TransactionAnnotationParser> annotationParsers;


    /**
     * Create a default AnnotationTransactionAttributeSource, supporting
     * public methods that carry the {@code Transactional} annotation
     * or the EJB3 {@link javax.ejb.TransactionAttribute} annotation.
     */
    public AnnotationTransactionAttributeSource() {
        this(true);
    }

    /**
     * Create a custom AnnotationTransactionAttributeSource, supporting
     * public methods that carry the {@code Transactional} annotation
     * or the EJB3 {@link javax.ejb.TransactionAttribute} annotation.
     * @param publicMethodsOnly whether to support public methods that carry
     * the {@code Transactional} annotation only (typically for use
     * with proxy-based AOP), or protected/private methods as well
```

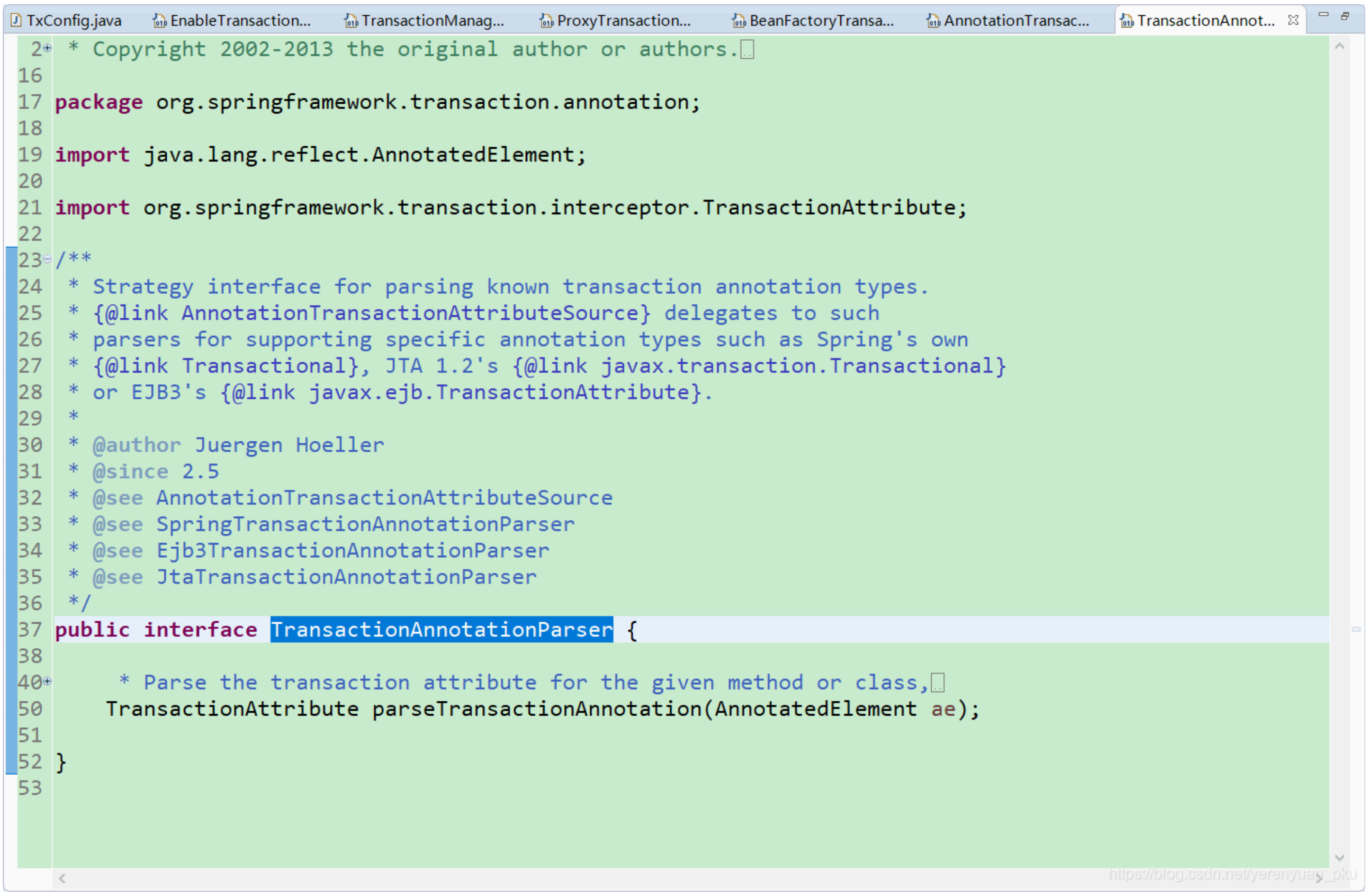
接着，点击一下this(true)方法，这时会跳到如下的一个有参构造方法处。


TxConfig.java
EnableTransactionMana...
TransactionManagement...
ProxyTransactionManag...
BeanFactoryTransaction...
AnnotationTransactionA...
https://blog.csdn.net/yerenyuan_pku

一个事务注解的解析器集合

```java
	}

	/**
	 * Create a custom AnnotationTransactionAttributeSource, supporting
	 * public methods that carry the {@code Transactional} annotation
	 * or the EJB3 {@link javax.ejb.TransactionAttribute} annotation.
	 * @param publicMethodsOnly whether to support public methods that carry
	 * the {@code Transactional} annotation only (typically for use
	 * with proxy-based AOP), or protected/private methods as well
	 * (typically used with AspectJ class weaving)
	 */
	public AnnotationTransactionAttributeSource(boolean publicMethodsOnly) {
		this.publicMethodsOnly = publicMethodsOnly;
		this.annotationParsers = new LinkedHashSet<TransactionAnnotationParser>(2);
		this.annotationParsers.add(new SpringTransactionAnnotationParser());
		if (jta12Present) {
			this.annotationParsers.add(new JtaTransactionAnnotationParser());
		}
		if (ejb3Present) {
			this.annotationParsers.add(new Ejb3TransactionAnnotationParser());
		}
	}

	/**
```

在该方法中，你会看到一个TransactionAnnotationParser接口，源码如下图所示。



```java
 * Copyright 2002-2013 the original author or authors.□

package org.springframework.transaction.annotation;

import java.lang.reflect.AnnotatedElement;

import org.springframework.transaction.interceptor.TransactionAttribute;

/**
 * Strategy interface for parsing known transaction annotation types.
 * {@link AnnotationTransactionAttributeSource} delegates to such
 * parsers for supporting specific annotation types such as Spring's own
 * {@link Transactional}, JTA 1.2's {@link javax.transaction.Transactional}
 * or EJB3's {@link javax.ejb.TransactionAttribute}.
 *
 * @author Juergen Hoeller
 * @since 2.5
 * @see AnnotationTransactionAttributeSource
 * @see SpringTransactionAnnotationParser
 * @see Ejb3TransactionAnnotationParser
 * @see JtaTransactionAnnotationParser
 */
public interface TransactionAnnotationParser {

	 * Parse the transaction attribute for the given method or class,□
	TransactionAttribute parseTransactionAnnotation(AnnotatedElement ae);

}
```

顾名思义，它是解析方法/类上事务注解的，当然了，你也可以称它为事务注解的解析器。

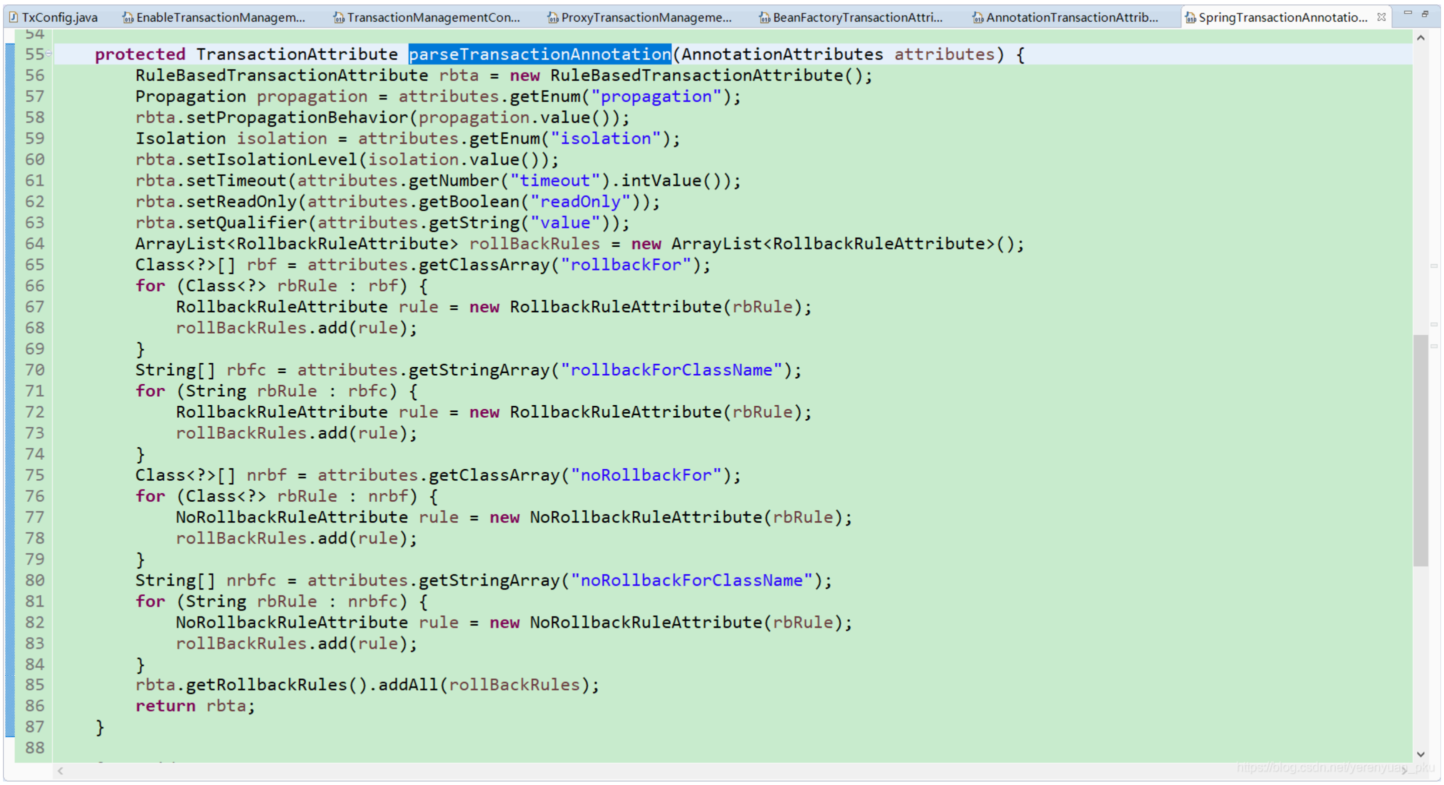
这里我要说明的一点是，Spring支持三个不同的事务注解，它们分别是：

1. Spring事务注解，即 `org.springframework.transaction.annotation.Transactional`（纯正血统，官方推荐）
2. JTA事务注解，即 `javax.transaction.Transactional`
3. EJB 3事务注解，即 `javax.ejb.TransactionAttribute`

因为现在基本上都是Spring的天下了，所以我们一般都会使用Spring事务注解。另外，上面三个注解虽然语义上一样，但是使用方式上不完全一样，若真要使用其它的则请注意各自的使用方式。

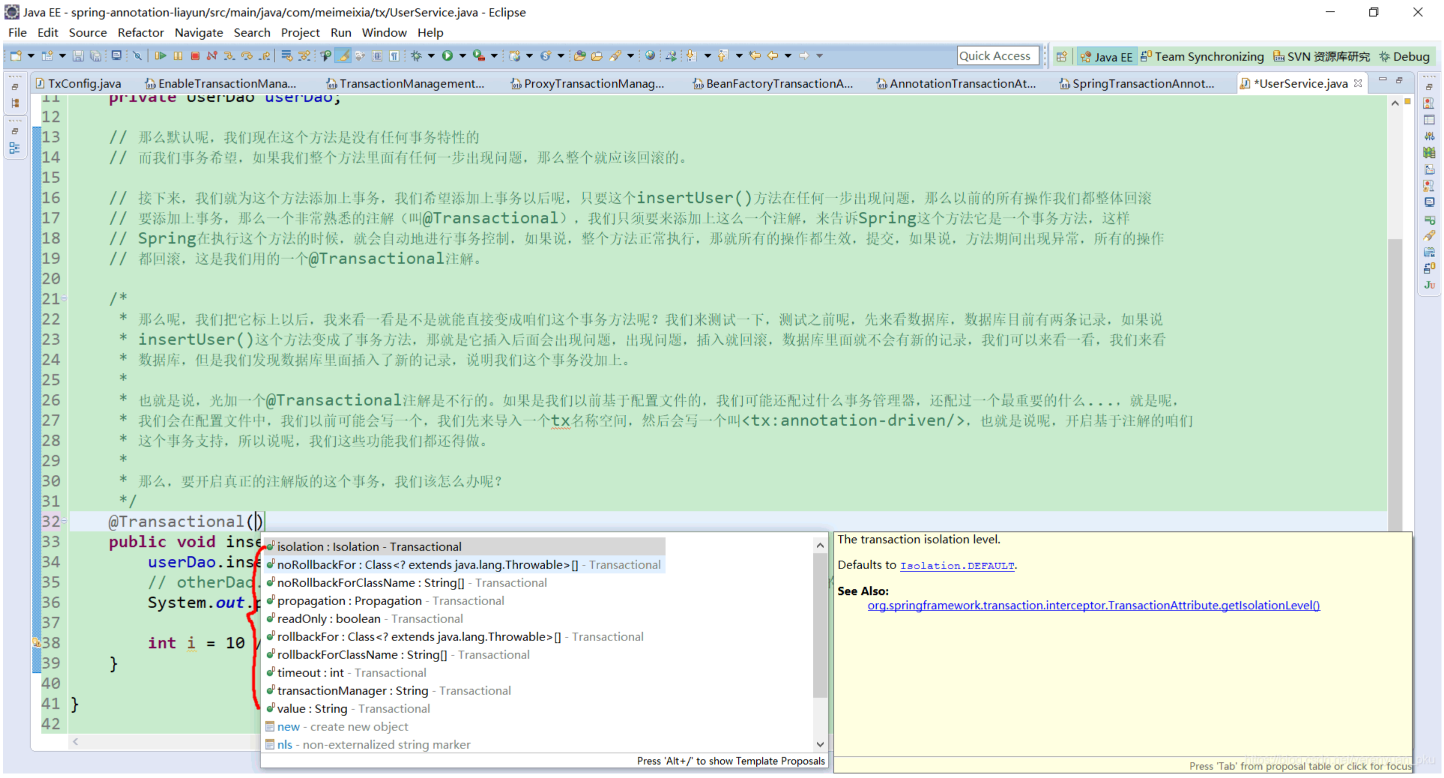
上面说到了Spring支持三个不同的事务注解，这里很显然，它们都对应了三个不同的注解解析器，即SpringTransactionAnnotationParser、JtaTransactionAnnotationParser以及Ejb3TransactionAnnotationParser。

也是因为现在基本上都是Spring的天下了，所以本文只会讲述SpringTransactionAnnotationParser，其它的雷同。我们可以点进去该类里面看一看，尤其要注意翻阅parseTransactionAnnotation方法，你会发现它就是来解析@Transactional注解里面的每一个信息的，包括它里面的每一个属性，例如rollbackFor、noRollbackFor、…


TxConfig.java
EnableTransactionManagem...
TransactionManagementCon...
ProxyTransactionManageme...
BeanFactoryTransactionAttri...
AnnotationTransactionAttrib...
SpringTransactionAnnotatio...

```java
    protected TransactionAttribute parseTransactionAnnotation(AnnotationAttributes attributes) {
        RuleBasedTransactionAttribute rbta = new RuleBasedTransactionAttribute();
        Propagation propagation = attributes.getEnum("propagation");
        rbta.setPropagationBehavior(propagation.value());
        Isolation isolation = attributes.getEnum("isolation");
        rbta.setIsolationLevel(isolation.value());
        rbta.setTimeout(attributes.getNumber("timeout").intValue());
        rbta.setReadOnly(attributes.getBoolean("readOnly"));
        rbta.setQualifier(attributes.getString("value"));
        ArrayList<RollbackRuleAttribute> rollBackRules = new ArrayList<RollbackRuleAttribute>();
        Class<?>[] rbf = attributes.getClassArray("rollbackFor");
        for (Class<?> rbRule : rbf) {
            RollbackRuleAttribute rule = new RollbackRuleAttribute(rbRule);
            rollBackRules.add(rule);
        }
        String[] rbfc = attributes.getStringArray("rollbackForClassName");
        for (String rbRule : rbfc) {
            RollbackRuleAttribute rule = new RollbackRuleAttribute(rbRule);
            rollBackRules.add(rule);
        }
        Class<?>[] nrbf = attributes.getClassArray("noRollbackFor");
        for (Class<?> rbRule : nrbf) {
            NoRollbackRuleAttribute rule = new NoRollbackRuleAttribute(rbRule);
            rollBackRules.add(rule);
        }
        String[] nrbfc = attributes.getStringArray("noRollbackForClassName");
        for (String rbRule : nrbfc) {
            NoRollbackRuleAttribute rule = new NoRollbackRuleAttribute(rbRule);
            rollBackRules.add(rule);
        }
        rbta.getRollbackRules().addAll(rollBackRules);
        return rbta;
    }
```

rollbackFor、noRollbackFor等等这些属性就是我们可以在@Transactional注解里面能写的。



```java
    private UserDao userDao;

    // 那么默认呢，我们现在这个方法是没有任何事务特性的
    // 而我们事务希望，如果我们整个方法里面有任何一步出现问题，那么整个就应该回滚的。

    // 接下来，我们就为这个方法添加上事务，我们希望添加上事务以后呢，只要这个insertUser()方法在任何一步出现问题，那么以前的所有操作我们都整体回滚
    // 要添加上事务，那么一个非常熟悉的注解（叫@Transactional），我们只须要来添加上这么一个注解，来告诉Spring这个方法它是一个事务方法，这样
    // Spring在执行这个方法的时候，就会自动地进行事务控制，如果说，整个方法正常执行，那就所有的操作都生效，提交，如果说，方法期间出现异常，所有的操作
    // 都回滚，这是我们用的一个@Transactional注解。

    /*
     * 那么呢，我们把它标上以后，我来看一看是不是就能直接变成咱们这个事务方法呢？我们来测试一下，测试之前呢，先来看数据库，数据库目前有两条记录，如果说
     * insertUser()这个方法变成了事务方法，那就是它插入后面会出现问题，出现问题，插入就回滚，数据库里面就不会有新的记录，我们可以来看一看，我们来看
     * 数据库，但是我们发现数据库里面插入了新的记录，说明我们这个事务没加上。
     *
     * 也就是说，光加一个@Transactional注解是不行的。如果是我们以前基于配置文件的，我们可能还配过什么事务管理器，还配过一个最重要的什么....，就是呢，
     * 我们会在配置文件中，我们以前可能会写一个，我们先来导入一个tx名称空间，然后会写一个叫<tx:annotation-driven/>，也就是说呢，开启基于注解的咱们
     * 这个事务支持，所以说呢，我们这些功能我们都还得做。
     *
     * 那么，要开启真正的注解版的这个事务，我们该怎么办呢？
     */
    @Transactional()
    public void inse
        userDao.ins
        // otherDao
        System.out.

        int i = 10
    }

}
```

```
isolation : Isolation - Transactional
noRollbackFor : Class<? extends java.lang.Throwable>[] - Transactional
noRollbackForClassName : String[] - Transactional
propagation : Propagation - Transactional
readOnly : boolean - Transactional
rollbackFor : Class<? extends java.lang.Throwable>[] - Transactional
rollbackForClassName : String[] - Transactional
timeout : int - Transactional
transactionManager : String - Transactional
value : String - Transactional
new - create new object
nls - non-externalized string marker
Press 'Alt+/' to show Template Proposals
```

The transaction isolation level.

Defaults to Isolation.DEFAULT.

See Also:
org.springframework.transaction.interceptor.TransactionAttribute.getIsolationLevel()

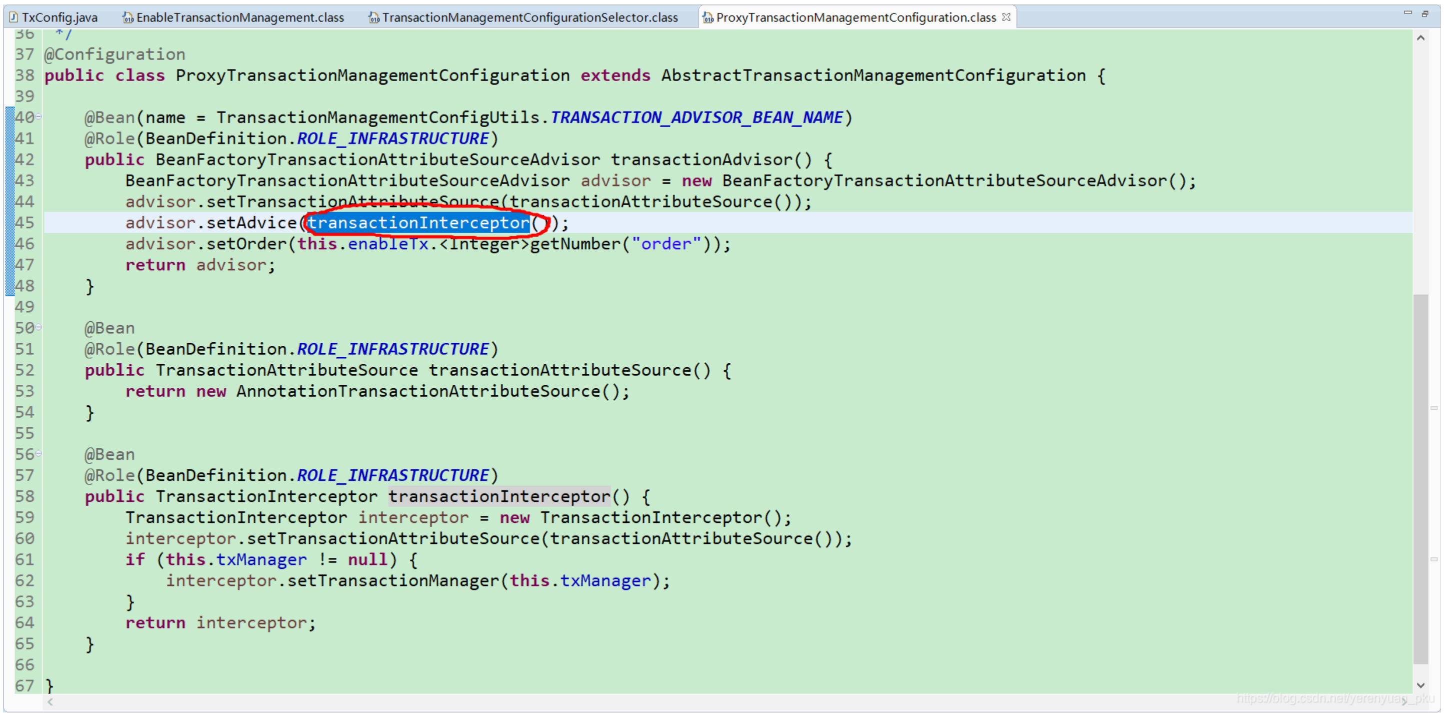
Press 'Tab' from proposal table or click for focus

小结

事务增强器要用到事务注解的信息，它总该得知道哪个方法是事务吧😊，所以这儿会使用到一个叫AnnotationTransactionAttributeSource的类，用它来解析事务注解。

在向容器中注册事务增强器时，还需要用到事务的拦截器

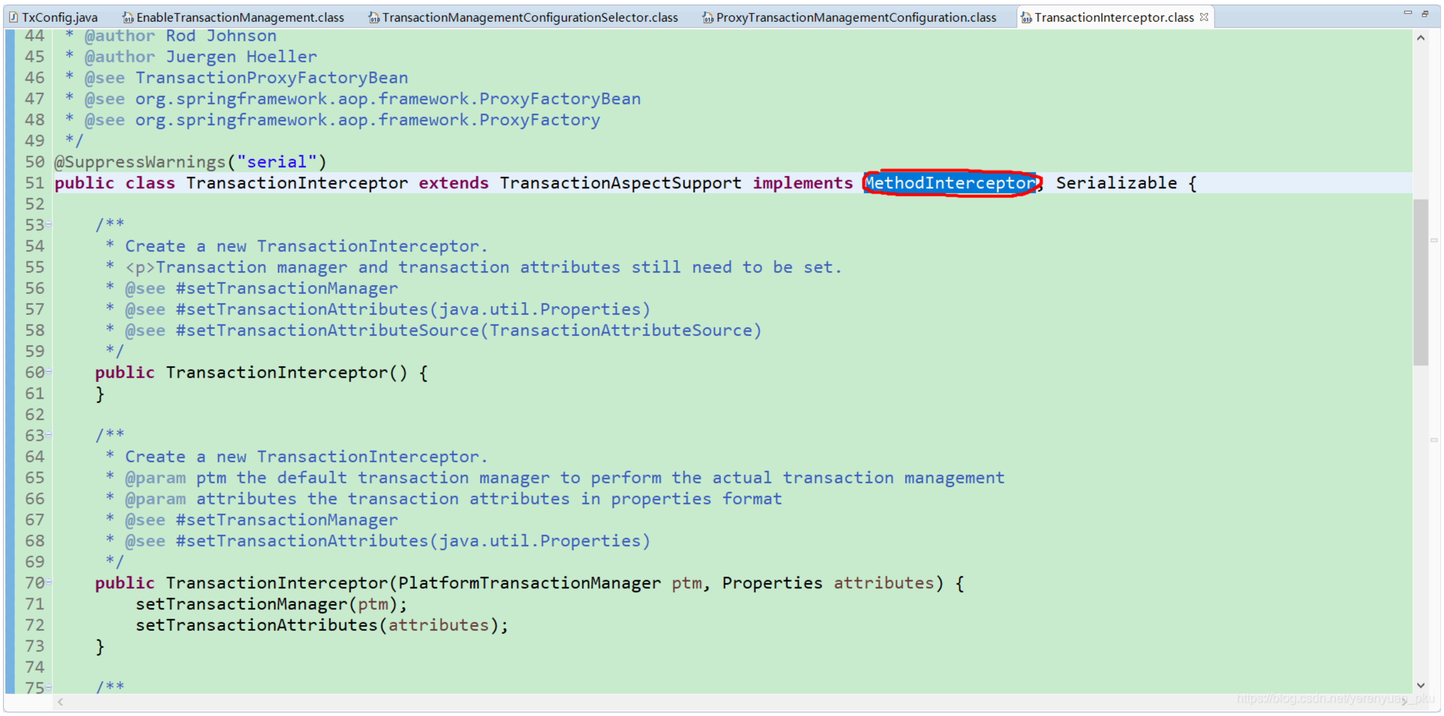
接下来，我们再来看看向容器中注册事务增强器时，还得做些什么。回到ProxyTransactionManagementConfiguration类中，发现在向容器中注册事务增强器时，除了需要事务注解信息，还需要一个事务的拦截器，看到那个transactionInterceptor方法没，它就是表示事务增强器还要用到一个事务的拦截器。


TxConfig.java
EnableTransactionManagement.class
TransactionManagementConfigurationSelector.class
ProxyTransactionManagementConfiguration.class
https://blog.csdn.net/yerenyuan_pku

```java
 */
@Configuration
public class ProxyTransactionManagementConfiguration extends AbstractTransactionManagementConfiguration {

    @Bean(name = TransactionManagementConfigUtils.TRANSACTION_ADVISOR_BEAN_NAME)
    @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
    public BeanFactoryTransactionAttributeSourceAdvisor transactionAdvisor() {
        BeanFactoryTransactionAttributeSourceAdvisor advisor = new BeanFactoryTransactionAttributeSourceAdvisor();
        advisor.setTransactionAttributeSource(transactionAttributeSource());
        advisor.setAdvice(transactionInterceptor());
        advisor.setOrder(this.enableTx.<Integer>getNumber("order"));
        return advisor;
    }

    @Bean
    @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
    public TransactionAttributeSource transactionAttributeSource() {
        return new AnnotationTransactionAttributeSource();
    }

    @Bean
    @Role(BeanDefinition.ROLE_INFRASTRUCTURE)
    public TransactionInterceptor transactionInterceptor() {
        TransactionInterceptor interceptor = new TransactionInterceptor();
        interceptor.setTransactionAttributeSource(transactionAttributeSource());
        if (this.txManager != null) {
            interceptor.setTransactionManager(this.txManager);
        }
        return interceptor;
    }

}
```

仔细查看上面的transactionInterceptor方法，你会看到在里面创建了一个TransactionInterceptor对象，创建完毕之后，不但会将事务属性源设置进去，而且还会将事务管理器（txManager）设置进去。也就是说，事务拦截器里面不仅保存了事务属性信息，还保存了事务管理器。

我们点进去TransactionInterceptor类里面去看一下，发现该类实现了一个MethodInterceptor接口，如下图所示。


TxConfig.java
EnableTransactionManagement.class
TransactionManagementConfigurationSelector.class
ProxyTransactionManagementConfiguration.class
TransactionInterceptor.class
https://blog.csdn.net/yerenyuan_pku

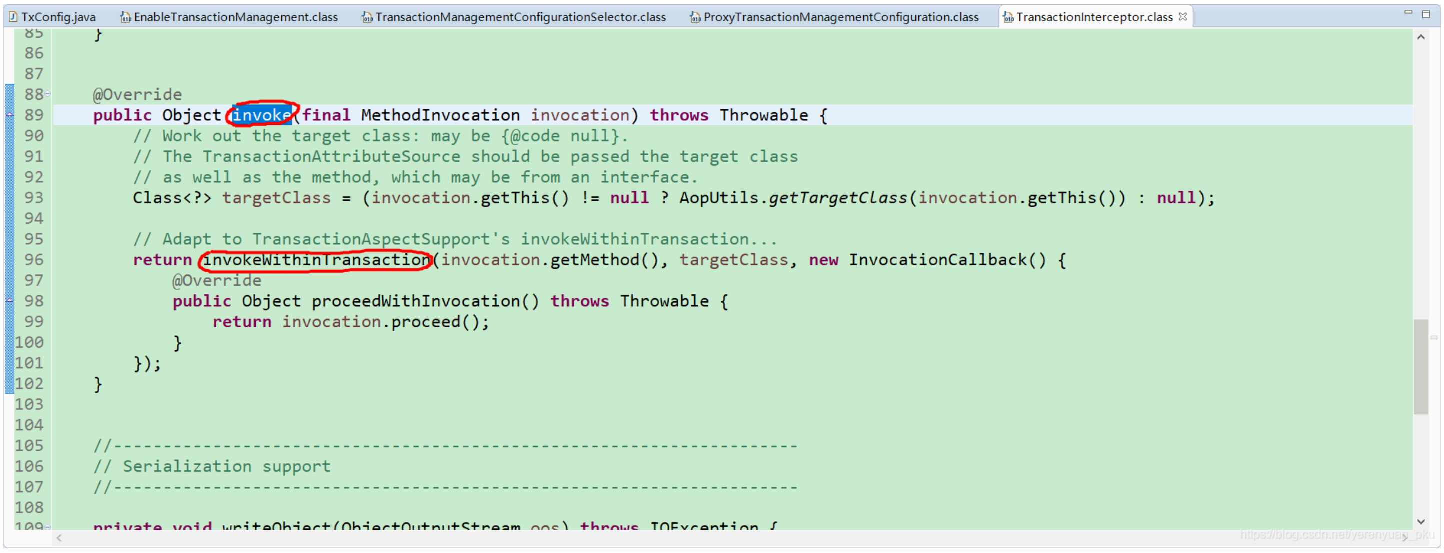
```java
 * @author Rod Johnson
 * @author Juergen Hoeller
 * @see TransactionProxyFactoryBean
 * @see org.springframework.aop.framework.ProxyFactoryBean
 * @see org.springframework.aop.framework.ProxyFactory
 */
@SuppressWarnings("serial")
public class TransactionInterceptor extends TransactionAspectSupport implements MethodInterceptor, Serializable {

    /**
     * Create a new TransactionInterceptor.
     * <p>Transaction manager and transaction attributes still need to be set.
     * @see #setTransactionManager
     * @see #setTransactionAttributes(java.util.Properties)
     * @see #setTransactionAttributeSource(TransactionAttributeSource)
     */
    public TransactionInterceptor() {
    }

    /**
     * Create a new TransactionInterceptor.
     * @param ptm the default transaction manager to perform the actual transaction management
     * @param attributes the transaction attributes in properties format
     * @see #setTransactionManager
     * @see #setTransactionAttributes(java.util.Properties)
     */
    public TransactionInterceptor(PlatformTransactionManager ptm, Properties attributes) {
        setTransactionManager(ptm);
        setTransactionAttributes(attributes);
    }

    /**
```

看到它，你是不是倍感亲切，因为咱们在研究AOP的原理时，就已经认识它了。相信你应该还记得这样一个知识点，**切面类里面的通知方法最终都会被整成增强器，而增强器又会被转换成MethodInterceptor**。所以，这样看来，这个事务拦截器实质上还是一个MethodInterceptor（方法拦截器）。

啥叫方法拦截器呢？简单来说就是，现在会向容器中放一个代理对象，代理对象要执行目标方法，那么方法拦截器就会进行工作。

其实，跟我们以前研究AOP的原理一模一样，在代理对象执行目标方法的时候，它便会来执行拦截器链，而现在这个拦截器链，只有一个TransactionInterceptor，它正是这个事务拦截器。接下来，我们就来看看这个事务拦截器是怎样工作的，即它的作用是什么。

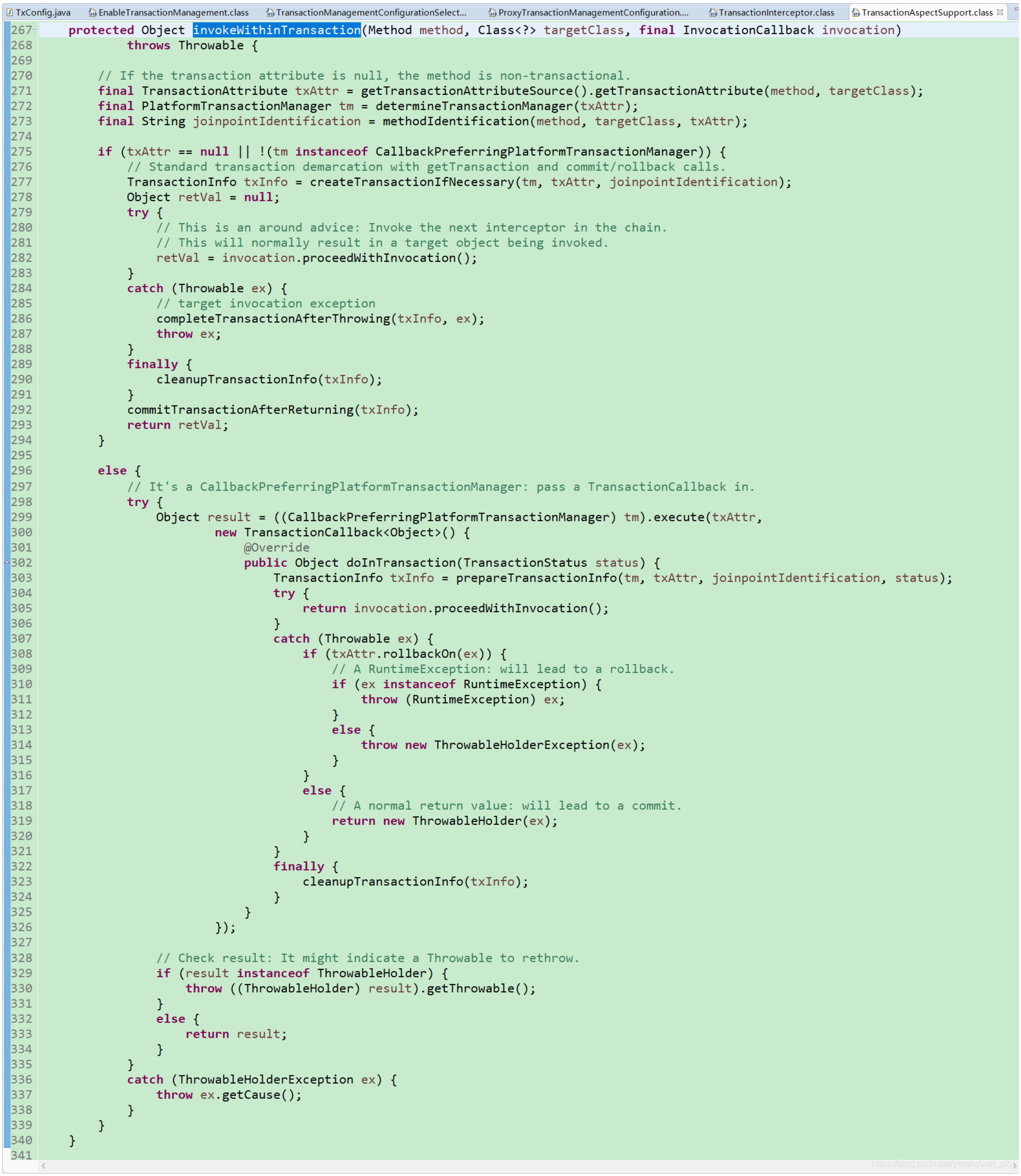
仔细翻阅TransactionInterceptor类的源码，你会发现它里面有一个invoke方法，而且还会看到在该方法里面又调用了一个invokeWithinTransaction方法，如下图所示。


TxConfig.java
EnableTransactionManagement.class
TransactionManagementConfigurationSelector.class
ProxyTransactionManagementConfiguration.class
TransactionInterceptor.class
https://blog.csdn.net/yerenyuan_pku

```
    }


    @Override
    public Object invoke(final MethodInvocation invocation) throws Throwable {
        // Work out the target class: may be {@code null}.
        // The TransactionAttributeSource should be passed the target class
        // as well as the method, which may be from an interface.
        Class<?> targetClass = (invocation.getThis() != null ? AopUtils.getTargetClass(invocation.getThis()) : null);

        // Adapt to TransactionAspectSupport's invokeWithinTransaction...
        return invokeWithinTransaction(invocation.getMethod(), targetClass, new InvocationCallback() {
            @Override
            public Object proceedWithInvocation() throws Throwable {
                return invocation.proceed();
            }
        });
    }


    //---------------------------------------------------------------------
    // Serialization support
    //---------------------------------------------------------------------

    private void writeObject(ObjectOutputStream oos) throws IOException {
```

点进去invokeWithinTransaction方法里面看一下，你就能知道这个事务拦截器是怎样工作的了。

哎呀🙃！你不仅感叹一声，这个方法未免也写得太长了吧！确实是太长了，不过为了大家能看得更加清楚，我还是把整个方法给截出来给大家看看。


TxConfig.java
EnableTransactionManagement.class
TransactionManagementConfigurationSelect...
ProxyTransactionManagementConfiguration....
TransactionInterceptor.class
TransactionAspectSupport.class
https://blog.csdn.net/yerenyuan_pku

```java
    protected Object invokeWithinTransaction(Method method, Class<?> targetClass, final InvocationCallback invocation)
            throws Throwable {

        // If the transaction attribute is null, the method is non-transactional.
        final TransactionAttribute txAttr = getTransactionAttributeSource().getTransactionAttribute(method, targetClass);
        final PlatformTransactionManager tm = determineTransactionManager(txAttr);
        final String joinpointIdentification = methodIdentification(method, targetClass, txAttr);

        if (txAttr == null || !(tm instanceof CallbackPreferringPlatformTransactionManager)) {
            // Standard transaction demarcation with getTransaction and commit/rollback calls.
            TransactionInfo txInfo = createTransactionIfNecessary(tm, txAttr, joinpointIdentification);
            Object retVal = null;
            try {
                // This is an around advice: Invoke the next interceptor in the chain.
                // This will normally result in a target object being invoked.
                retVal = invocation.proceedWithInvocation();
            }
            catch (Throwable ex) {
                // target invocation exception
                completeTransactionAfterThrowing(txInfo, ex);
                throw ex;
            }
            finally {
                cleanupTransactionInfo(txInfo);
            }
            commitTransactionAfterReturning(txInfo);
            return retVal;
        }

        else {
            // It's a CallbackPreferringPlatformTransactionManager: pass a TransactionCallback in.
            try {
                Object result = ((CallbackPreferringPlatformTransactionManager) tm).execute(txAttr,
                        new TransactionCallback<Object>() {
                            @Override
                            public Object doInTransaction(TransactionStatus status) {
                                TransactionInfo txInfo = prepareTransactionInfo(tm, txAttr, joinpointIdentification, status);
                                try {
                                    return invocation.proceedWithInvocation();
                                }
                                catch (Throwable ex) {
                                    if (txAttr.rollbackOn(ex)) {
                                        // A RuntimeException: will lead to a rollback.
                                        if (ex instanceof RuntimeException) {
                                            throw (RuntimeException) ex;
                                        }
                                        else {
                                            throw new ThrowableHolderException(ex);
                                        }
                                    }
                                    else {
                                        // A normal return value: will lead to a commit.
                                        return new ThrowableHolder(ex);
                                    }
                                }
                                finally {
                                    cleanupTransactionInfo(txInfo);
                                }
                            }
                        });

                // Check result: It might indicate a Throwable to rethrow.
                if (result instanceof ThrowableHolder) {
                    throw ((ThrowableHolder) result).getThrowable();
                }
                else {
                    return result;
                }
            }
            catch (ThrowableHolderException ex) {
                throw ex.getCause();
            }
        }
    }
```

看看，是不是足够长啊😱！下面我就来详细讲述一下该方法。

先来获取事务相关的一些属性信息

从invokeWithinTransaction方法的第一行代码，即：

```java
final TransactionAttribute txAttr = getTransactionAttributeSource().getTransactionAttribute(method, targetClass);
```

AI写代码java运行

我们便可以知道，这儿是来获取事务相关的一些属性信息的。

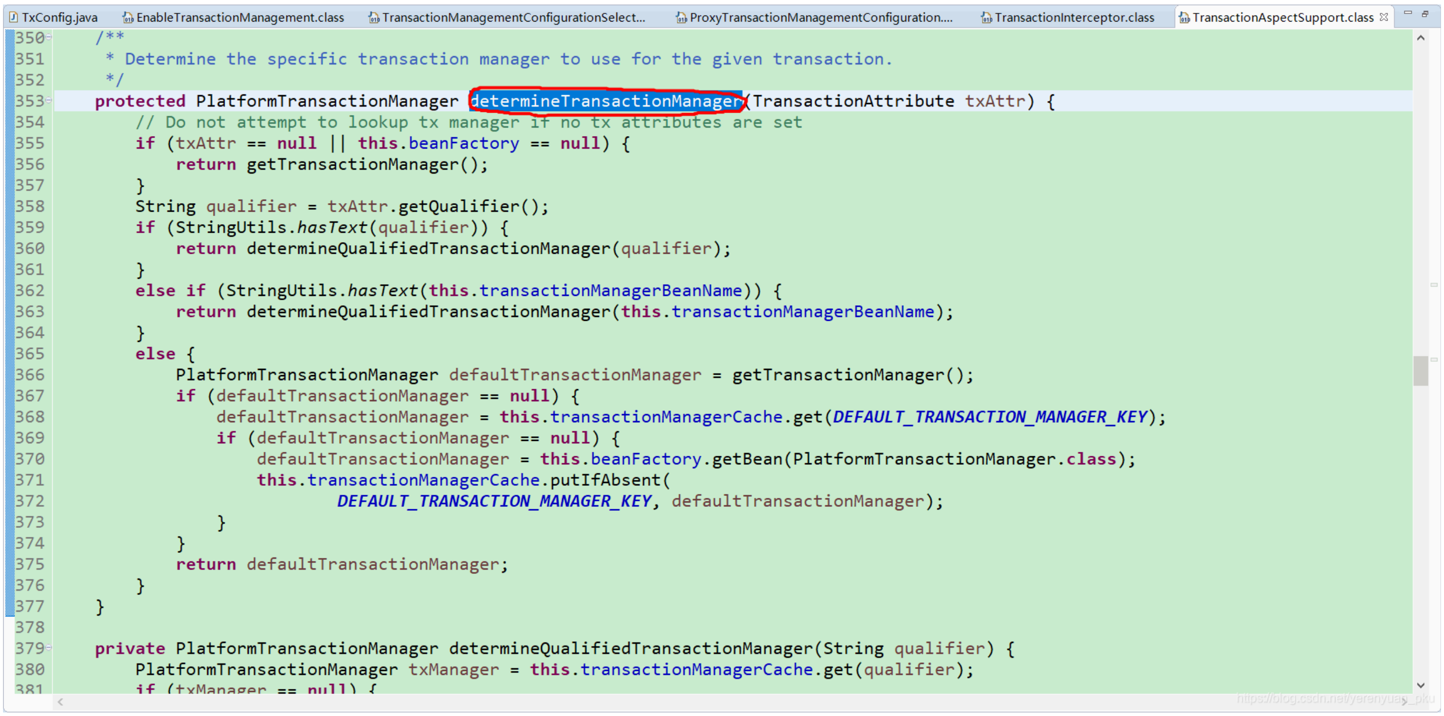
再来获取PlatformTransactionManager

接着往下看invokeWithinTransaction方法，可以看到它的第二行代码是这样写的：

```java
final PlatformTransactionManager tm = determineTransactionManager(txAttr);
```

AI写代码java运行

这就是来获取PlatformTransactionManager的，还记得我们之前就已经向容器中注册了一个吗，现在就是来获取它的。那到底又是怎么来获取的呢？我们不妨点进去determineTransactionManager方法里面去看一下。


TxConfig.java
EnableTransactionManagement.class
TransactionManagementConfigurationSelect...
ProxyTransactionManagementConfiguration....
TransactionInterceptor.class
TransactionAspectSupport.class
https://blog.csdn.net/yerenyuan_pku

```java
    /**
     * Determine the specific transaction manager to use for the given transaction.
     */
    protected PlatformTransactionManager determineTransactionManager(TransactionAttribute txAttr) {
        // Do not attempt to lookup tx manager if no tx attributes are set
        if (txAttr == null || this.beanFactory == null) {
            return getTransactionManager();
        }
        String qualifier = txAttr.getQualifier();
        if (StringUtils.hasText(qualifier)) {
            return determineQualifiedTransactionManager(qualifier);
        }
        else if (StringUtils.hasText(this.transactionManagerBeanName)) {
            return determineQualifiedTransactionManager(this.transactionManagerBeanName);
        }
        else {
            PlatformTransactionManager defaultTransactionManager = getTransactionManager();
            if (defaultTransactionManager == null) {
                defaultTransactionManager = this.transactionManagerCache.get(DEFAULT_TRANSACTION_MANAGER_KEY);
                if (defaultTransactionManager == null) {
                    defaultTransactionManager = this.beanFactory.getBean(PlatformTransactionManager.class);
                    this.transactionManagerCache.putIfAbsent(
                            DEFAULT_TRANSACTION_MANAGER_KEY, defaultTransactionManager);
                }
            }
            return defaultTransactionManager;
        }
    }

    private PlatformTransactionManager determineQualifiedTransactionManager(String qualifier) {
        PlatformTransactionManager txManager = this.transactionManagerCache.get(qualifier);
        if (txManager == null) {
```

这个方法写的还是蛮长的，不过没关系啊，下面我会为大家详细说说该方法。

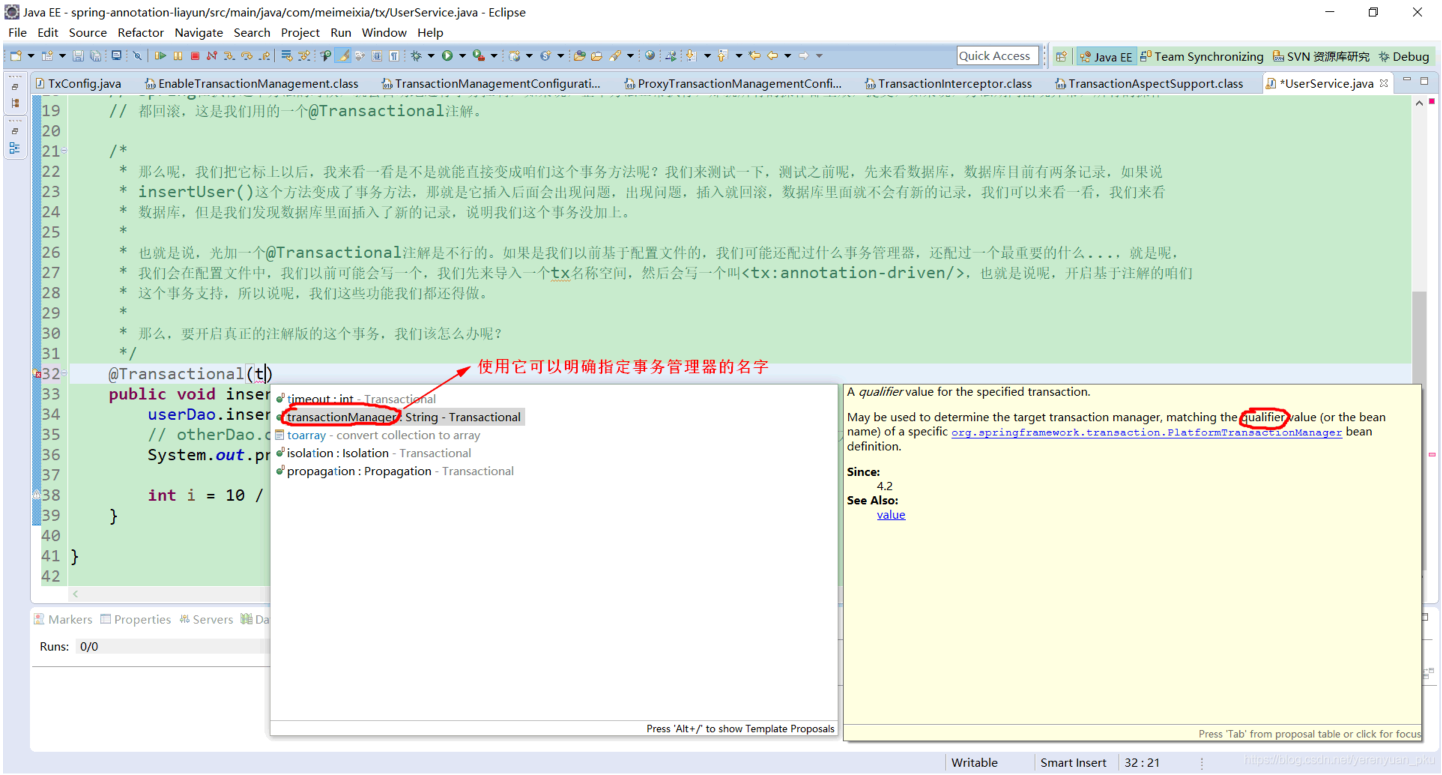
先来看看下面这几行代码，即：

```java
// ···
String qualifier = txAttr.getQualifier();
if (StringUtils.hasText(qualifier)) {
    return determineQualifiedTransactionManager(qualifier);
}
// ···
```

AI写代码java运行

这几行代码说的是啥意思呢？它是说，如果事务属性里面有Qualifier这个注解，并且这个注解还有值，那么就会直接从容器中按照这个指定的值来获取PlatformTransactionManager。

我这样一讲，相信你更加摸不着头脑了，这说的是啥啊😱！且听我娓娓道来，其实我们在为某个业务方法标注@Transactional注解的时候，是可以明确地指定事务管理器的名字的，不信你看：


Java EE - spring-annotation-liayun/src/main/java/com/meimeixia/tx/UserService.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Java EE
Team Synchronizing
SVN 资源库研究
Debug
TxConfig.java
EnableTransactionManagement.class
TransactionManagementConfigurati...
ProxyTransactionManagementConfi...
TransactionInterceptor.class
TransactionAspectSupport.class
*UserService.java
使用它可以明确指定事务管理器的名字
Markers
Properties
Servers
Da
Runs: 0/0
Writable
Smart Insert
32 : 21
https://blog.csdn.net/yerenyuan_pku

```java
    // 都回滚，这是我们用的一个@Transactional注解。

    /*
     * 那么呢，我们把它标上以后，我来看一看是不是就能直接变成咱们这个事务方法呢？我们来测试一下，测试之前呢，先来看数据库，数据库目前有两条记录，如果说
     * insertUser()这个方法变成了事务方法，那就是它插入后面会出现问题，出现问题，插入就回滚，数据库里面就不会有新的记录，我们可以来看一看，我们来看
     * 数据库，但是我们发现数据库里面插入了新的记录，说明我们这个事务没加上。
     *
     * 也就是说，光加一个@Transactional注解是不行的。如果是我们以前基于配置文件的，我们可能还配过什么事务管理器，还配过一个最重要的什么....，就是呢，
     * 我们会在配置文件中，我们以前可能会写一个，我们先来导入一个tx名称空间，然后会写一个叫<tx:annotation-driven/>，也就是说呢，开启基于注解的咱们
     * 这个事务支持，所以说呢，我们这些功能我们都还得做。
     *
     * 那么，要开启真正的注解版的这个事务，我们该怎么办呢？
     */
    @Transactional(t)
    public void inser
        userDao.inser
        // otherDao.c
        System.out.pr

        int i = 10 /
    }

}
```

timeout : int - Transactional
transactionManager : String - Transactional
toarray - convert collection to array
isolation : Isolation - Transactional
propagation : Propagation - Transactional
Press 'Alt+/' to show Template Proposals

A *qualifier* value for the specified transaction.

May be used to determine the target transaction manager, matching the qualifier value (or the bean name) of a specific `org.springframework.transaction.PlatformTransactionManager` bean definition.

Since:
4.2
See Also:
value

Press 'Tab' from proposal table or click for focus

从上图中可以看到，指定事务管理器的名字，其实就等同于Qualifier这个注解。虽说是可以明确指定事务管理器的名字，但我们一般都不这么做，即不指定。

如果真要是指定了的话，那么就应该是到这儿来判断了。

```java
else if (StringUtils.hasText(this.transactionManagerBeanName)) {
    return determineQualifiedTransactionManager(this.transactionManagerBeanName);
}
```

AI写代码java运行

上面这几行代码应该是来判断PlatformTransactionManager是否有名，若有则就应该像上面这么来获取。希望我理解的没有错😊

如果没指定的话，那么就是来获取默认的了，这时很显然会进入到最下面的else判断中。

```java
else {
    PlatformTransactionManager defaultTransactionManager = getTransactionManager();
    if (defaultTransactionManager == null) {
        defaultTransactionManager = this.transactionManagerCache.get(DEFAULT_TRANSACTION_MANAGER_KEY);
        if (defaultTransactionManager == null) {
            defaultTransactionManager = this.beanFactory.getBean(PlatformTransactionManager.class);
            this.transactionManagerCache.putIfAbsent(
                    DEFAULT_TRANSACTION_MANAGER_KEY, defaultTransactionManager);
        }
    }
    return defaultTransactionManager;
}
```

AI写代码java运行

可以看到，会先调用getTransactionManager方法，获取的是默认向容器中自动装配进去的PlatformTransactionManager。

首次获取肯定就为null，但没关系，因为最终会从容器中按照类型来获取，这可以从下面这行代码中看出来。

```java
defaultTransactionManager = this.beanFactory.getBean(PlatformTransactionManager.class);
```

AI写代码java运行

所以，我们只需要给容器中注入一个PlatformTransactionManager，正如我们前面写的这样：

```java
// 注册事务管理器在容器中
@Bean
public PlatformTransactionManager platformTransactionManager() throws Exception {
    return new DataSourceTransactionManager(dataSource());
}
```

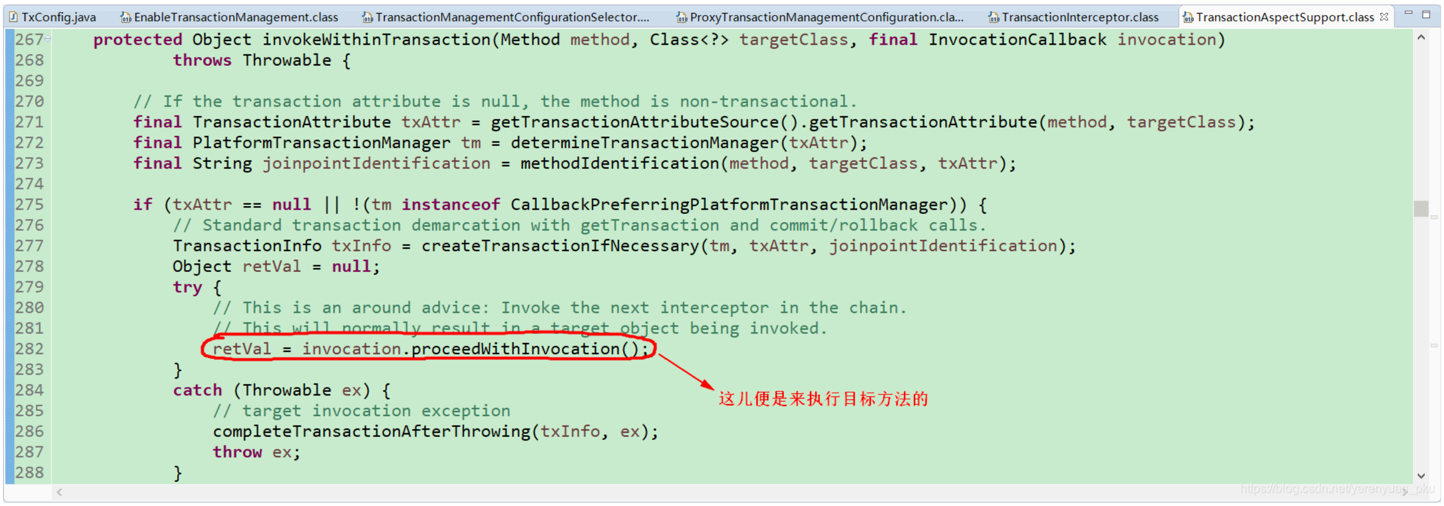
AI写代码java运行

然后就能获取到PlatformTransactionManager了。获取到了之后，当然就可以使用它了。

总结：如果事先没有添加指定任何TransactionManager，那么最终会从容器中按照类型来获取一个PlatformTransactionManager。

执行目标方法

接下来，继续往下看invokeWithinTransaction方法，来看它接下去又做了些什么。其实，很容易就能看出来，获取到事务管理器之后，然后便要来执行目标方法了，而且如果目标方法执行时一切正常，那么还能拿到一个返回值，如下图所示。


TxConfig.java
EnableTransactionManagement.class
TransactionManagementConfigurationSelector....
ProxyTransactionManagementConfiguration.cla...
TransactionInterceptor.class
TransactionAspectSupport.class

```java
    protected Object invokeWithinTransaction(Method method, Class<?> targetClass, final InvocationCallback invocation)
            throws Throwable {

        // If the transaction attribute is null, the method is non-transactional.
        final TransactionAttribute txAttr = getTransactionAttributeSource().getTransactionAttribute(method, targetClass);
        final PlatformTransactionManager tm = determineTransactionManager(txAttr);
        final String joinpointIdentification = methodIdentification(method, targetClass, txAttr);

        if (txAttr == null || !(tm instanceof CallbackPreferringPlatformTransactionManager)) {
            // Standard transaction demarcation with getTransaction and commit/rollback calls.
            TransactionInfo txInfo = createTransactionIfNecessary(tm, txAttr, joinpointIdentification);
            Object retVal = null;
            try {
                // This is an around advice: Invoke the next interceptor in the chain.
                // This will normally result in a target object being invoked.
                retVal = invocation.proceedWithInvocation();
            }
            catch (Throwable ex) {
                // target invocation exception
                completeTransactionAfterThrowing(txInfo, ex);
                throw ex;
            }
```

这儿便是来执行目标方法的

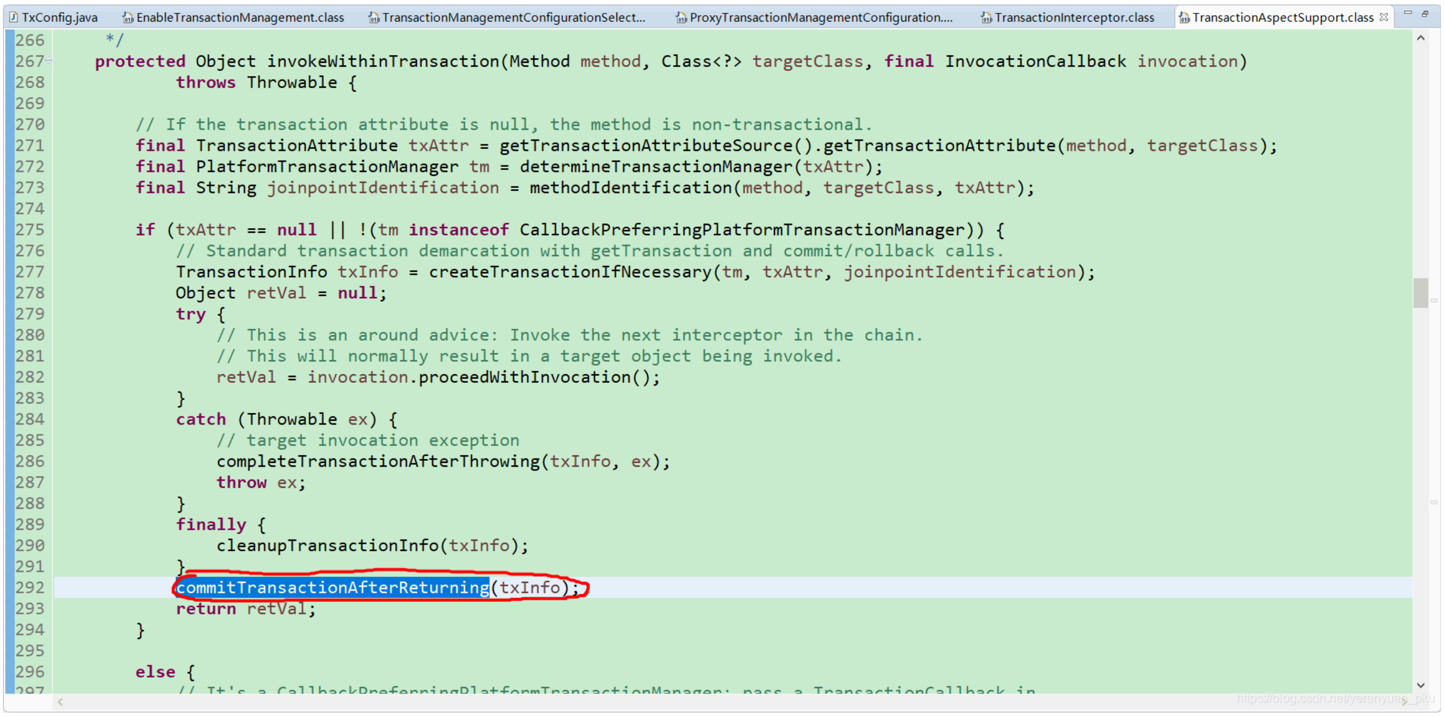
不知你有没有看到，在执行上面这句代码之前，还有这样一句代码：

```java
TransactionInfo txInfo = createTransactionIfNecessary(tm, txAttr, joinpointIdentification);
```

AI写代码java运行

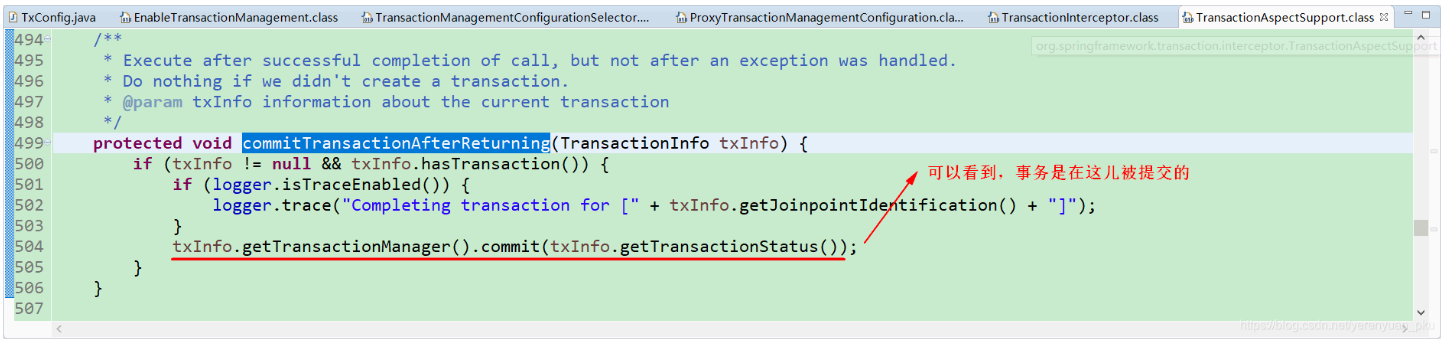
上面这个方法翻译成中文，就是如果是必须的话，那么得先创建一个Transaction。说人话，就是如果目标方法是一个事务，那么便开启事务。

如果目标方法执行时一切正常，那么接下来该怎么办呢？这时，会调用一个叫commitTransactionAfterReturning的方法，如下图所示。


TxConfig.java
EnableTransactionManagement.class
TransactionManagementConfigurationSelect...
ProxyTransactionManagementConfiguration....
TransactionInterceptor.class
TransactionAspectSupport.class

```java
     */
    protected Object invokeWithinTransaction(Method method, Class<?> targetClass, final InvocationCallback invocation)
            throws Throwable {

        // If the transaction attribute is null, the method is non-transactional.
        final TransactionAttribute txAttr = getTransactionAttributeSource().getTransactionAttribute(method, targetClass);
        final PlatformTransactionManager tm = determineTransactionManager(txAttr);
        final String joinpointIdentification = methodIdentification(method, targetClass, txAttr);

        if (txAttr == null || !(tm instanceof CallbackPreferringPlatformTransactionManager)) {
            // Standard transaction demarcation with getTransaction and commit/rollback calls.
            TransactionInfo txInfo = createTransactionIfNecessary(tm, txAttr, joinpointIdentification);
            Object retVal = null;
            try {
                // This is an around advice: Invoke the next interceptor in the chain.
                // This will normally result in a target object being invoked.
                retVal = invocation.proceedWithInvocation();
            }
            catch (Throwable ex) {
                // target invocation exception
                completeTransactionAfterThrowing(txInfo, ex);
                throw ex;
            }
            finally {
                cleanupTransactionInfo(txInfo);
            }
            commitTransactionAfterReturning(txInfo);
            return retVal;
        }

        else {
            // It's a CallbackPreferringPlatformTransactionManager: pass a TransactionCallback in
```

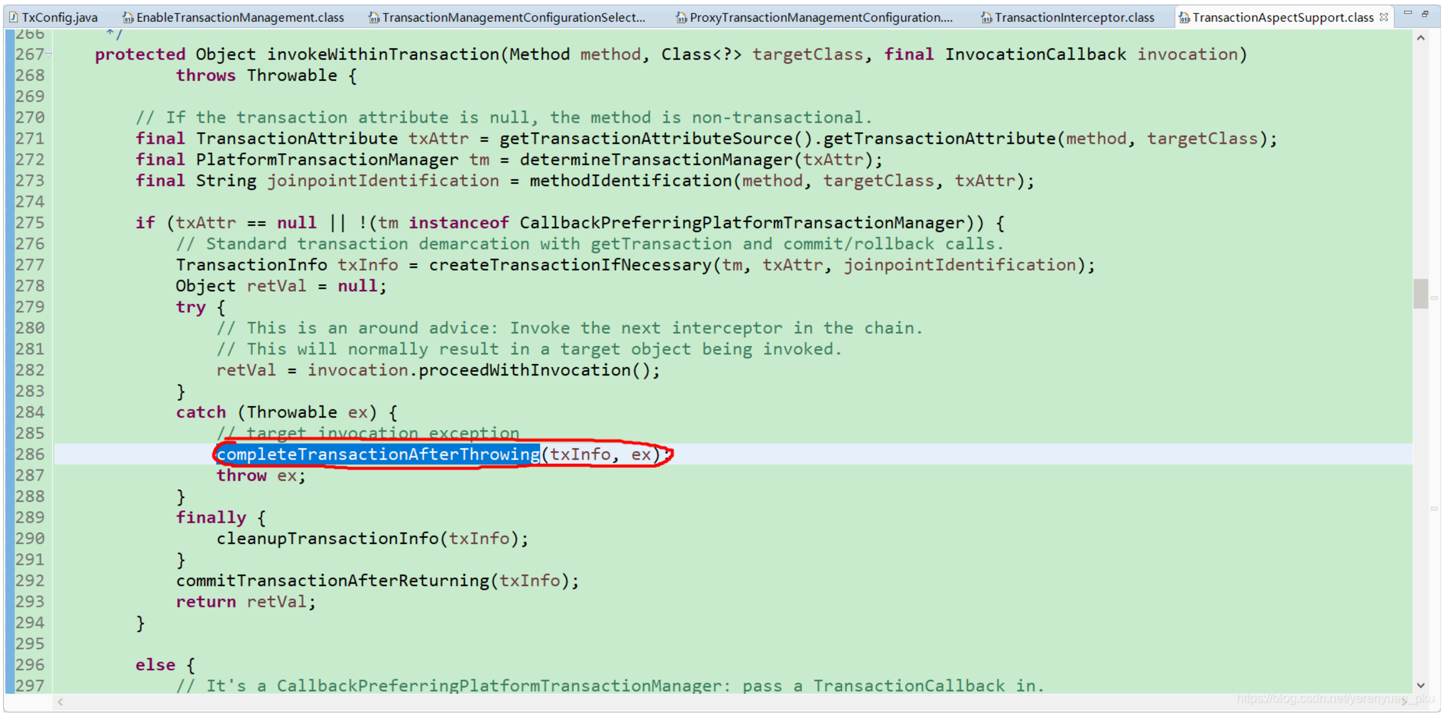
我们可以点进去commitTransactionAfterReturning方法里面去看一看，发现它是先获取到事务管理器，然后再利用事务管理器提交事务，如下图所示。


TxConfig.java
EnableTransactionManagement.class
TransactionManagementConfigurationSelector....
ProxyTransactionManagementConfiguration.cla...
TransactionInterceptor.class
TransactionAspectSupport.class
org.springframework.transaction.interceptor.TransactionAspectSupport
https://blog.csdn.net/yerenyuan_pku

```java
    /**
     * Execute after successful completion of call, but not after an exception was handled.
     * Do nothing if we didn't create a transaction.
     * @param txInfo information about the current transaction
     */
    protected void commitTransactionAfterReturning(TransactionInfo txInfo) {
        if (txInfo != null && txInfo.hasTransaction()) {
            if (logger.isTraceEnabled()) {
                logger.trace("Completing transaction for [" + txInfo.getJoinpointIdentification() + "]");
            }
            txInfo.getTransactionManager().commit(txInfo.getTransactionStatus());
        }
    }
```

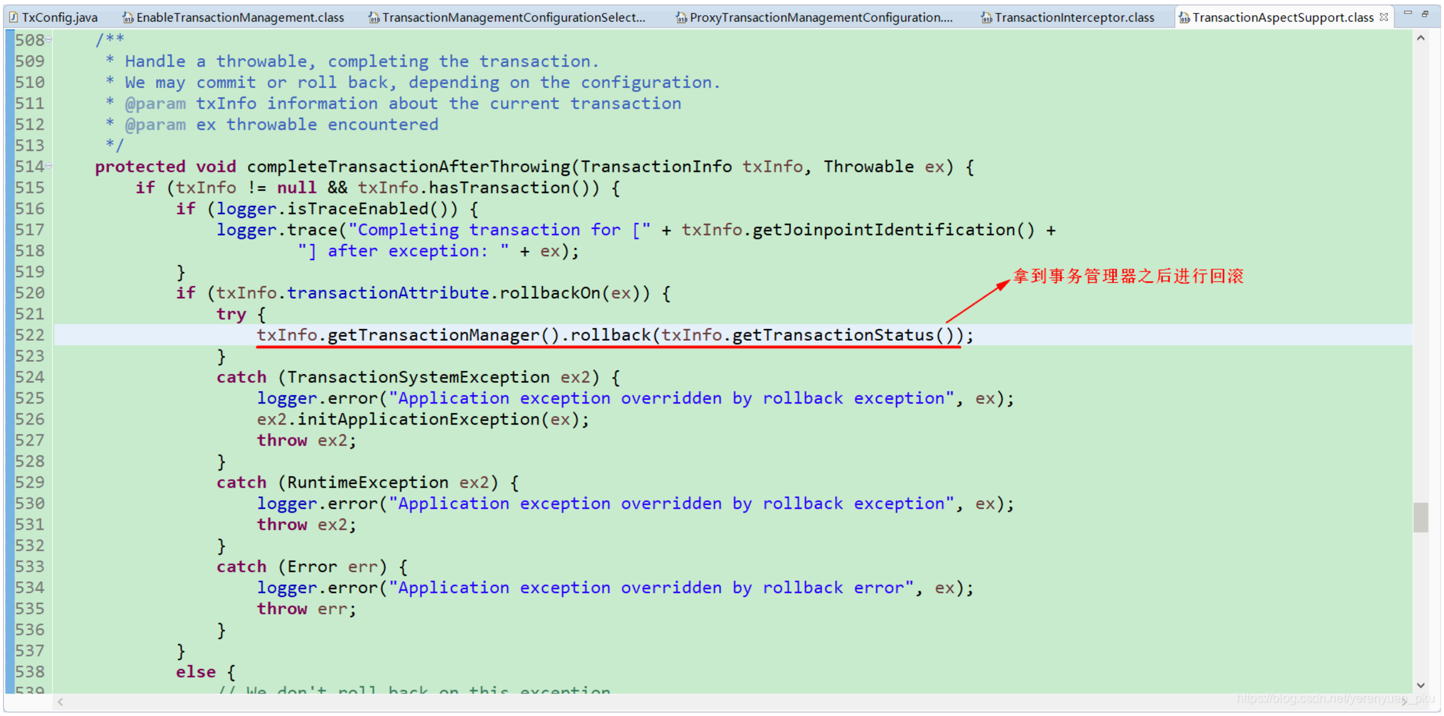
可以看到，事务是在这儿被提交的

如果执行目标方法时出现异常，那么又该怎么办呢？这时，会调用一个叫completeTransactionAfterThrowing的方法，如下图所示。


TxConfig.java
EnableTransactionManagement.class
TransactionManagementConfigurationSelect...
ProxyTransactionManagementConfiguration....
TransactionInterceptor.class
TransactionAspectSupport.class
https://blog.csdn.net/yerenyuan_pku

```java
     */
    protected Object invokeWithinTransaction(Method method, Class<?> targetClass, final InvocationCallback invocation)
            throws Throwable {

        // If the transaction attribute is null, the method is non-transactional.
        final TransactionAttribute txAttr = getTransactionAttributeSource().getTransactionAttribute(method, targetClass);
        final PlatformTransactionManager tm = determineTransactionManager(txAttr);
        final String joinpointIdentification = methodIdentification(method, targetClass, txAttr);

        if (txAttr == null || !(tm instanceof CallbackPreferringPlatformTransactionManager)) {
            // Standard transaction demarcation with getTransaction and commit/rollback calls.
            TransactionInfo txInfo = createTransactionIfNecessary(tm, txAttr, joinpointIdentification);
            Object retVal = null;
            try {
                // This is an around advice: Invoke the next interceptor in the chain.
                // This will normally result in a target object being invoked.
                retVal = invocation.proceedWithInvocation();
            }
            catch (Throwable ex) {
                // target invocation exception
                completeTransactionAfterThrowing(txInfo, ex);
                throw ex;
            }
            finally {
                cleanupTransactionInfo(txInfo);
            }
            commitTransactionAfterReturning(txInfo);
            return retVal;
        }

        else {
            // It's a CallbackPreferringPlatformTransactionManager: pass a TransactionCallback in.
```

我们可以点进去completeTransactionAfterThrowing方法里面去看一看，发现它是先获取到事务管理器，然后再利用事务管理器回滚这次操作，如下图所示。


TxConfig.java
EnableTransactionManagement.class
TransactionManagementConfigurationSelect...
ProxyTransactionManagementConfiguration....
TransactionInterceptor.class
TransactionAspectSupport.class
https://blog.csdn.net/yerenyuan_pku

```java
    /**
     * Handle a throwable, completing the transaction.
     * We may commit or roll back, depending on the configuration.
     * @param txInfo information about the current transaction
     * @param ex throwable encountered
     */
    protected void completeTransactionAfterThrowing(TransactionInfo txInfo, Throwable ex) {
        if (txInfo != null && txInfo.hasTransaction()) {
            if (logger.isTraceEnabled()) {
                logger.trace("Completing transaction for [" + txInfo.getJoinpointIdentification() +
                        "] after exception: " + ex);
            }
            if (txInfo.transactionAttribute.rollbackOn(ex)) {
                try {
                    txInfo.getTransactionManager().rollback(txInfo.getTransactionStatus());
                }
                catch (TransactionSystemException ex2) {
                    logger.error("Application exception overridden by rollback exception", ex);
                    ex2.initApplicationException(ex);
                    throw ex2;
                }
                catch (RuntimeException ex2) {
                    logger.error("Application exception overridden by rollback exception", ex);
                    throw ex2;
                }
                catch (Error err) {
                    logger.error("Application exception overridden by rollback error", ex);
                    throw err;
                }
            }
            else {
                // We don't roll back on this exception.
```

拿到事务管理器之后进行回滚

也就是说，真正的回滚与提交事务的操作都是由事务管理器来做的，而TransactionInterceptor只是用来拦截目标方法的。

以上就是我们通过简单地来分析源码，粗略地了解了一下整个事务控制的原理。

总结

最后，我来总结一下声明式事务的原理。

首先，使用AutoProxyRegistrar向Spring容器里面注册一个后置处理器，这个后置处理器会负责给我们包装代理对象。然后，使用ProxyTransactionManagementConfiguration（配置类）再向Spring容器里面注册一个事务增强器，此时，需要用到事务拦截器。最后，代理对象执行目标方法，在这一过程中，便会执行到当前Spring容器里面的拦截器链，而且每次在执行目标方法时，如果出现了异常，那么便会利用事务管理器进行回滚事务，如果执行过程中一切正常，那么则会利用事务管理器提交事务。