链滴

SpringAOP 多切面

作者：zhaozhizheng

原文链接：https://ld246.com/article/1639722858589

来源网站：链滴

许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

多切面执行时，采用了责任链设计模式。

切面的配置顺序决定了切面的执行顺序，多个切面执行的过程，类似于方法调用的过程，在环绕通知proceed()执行时，去执行下一个切面或如果没有下一个切面执行目标方法，从而达成了如下的执行程：（目标方法只会执行一次）

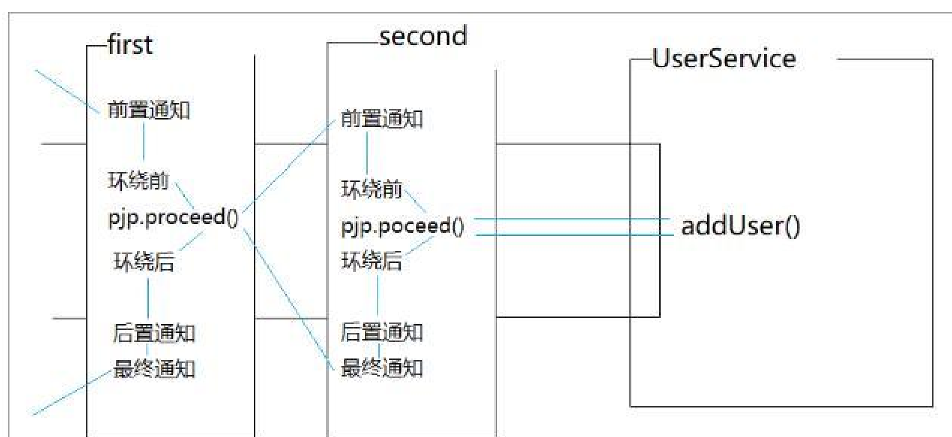
执行的顺序和配置顺序等有关

目标方法执行前： 前置，环绕前（顺序可以变）

目标方法执行后： 最终，环绕后，最终（顺序可以变）

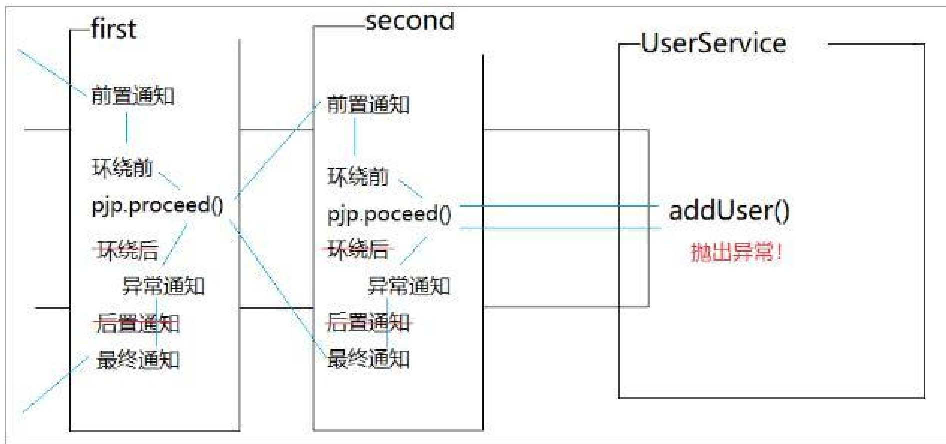
责任链设计模式：为解除请求的发送者和接收者之间的耦合，而使多个对象都有机会处理这个请求。这些对象连成一条链，并沿着这条链传递该请求，直到有对象处理它

因为责任链设计模式first切面中调用poceed方法后，继续沿着责任链去调用second切面中的环绕前----然后掉poceed方法 ----此时没有切面了---沿着责任链去调用addUser()方法---调用结束addUser()在执行环绕second环绕后。。。。。

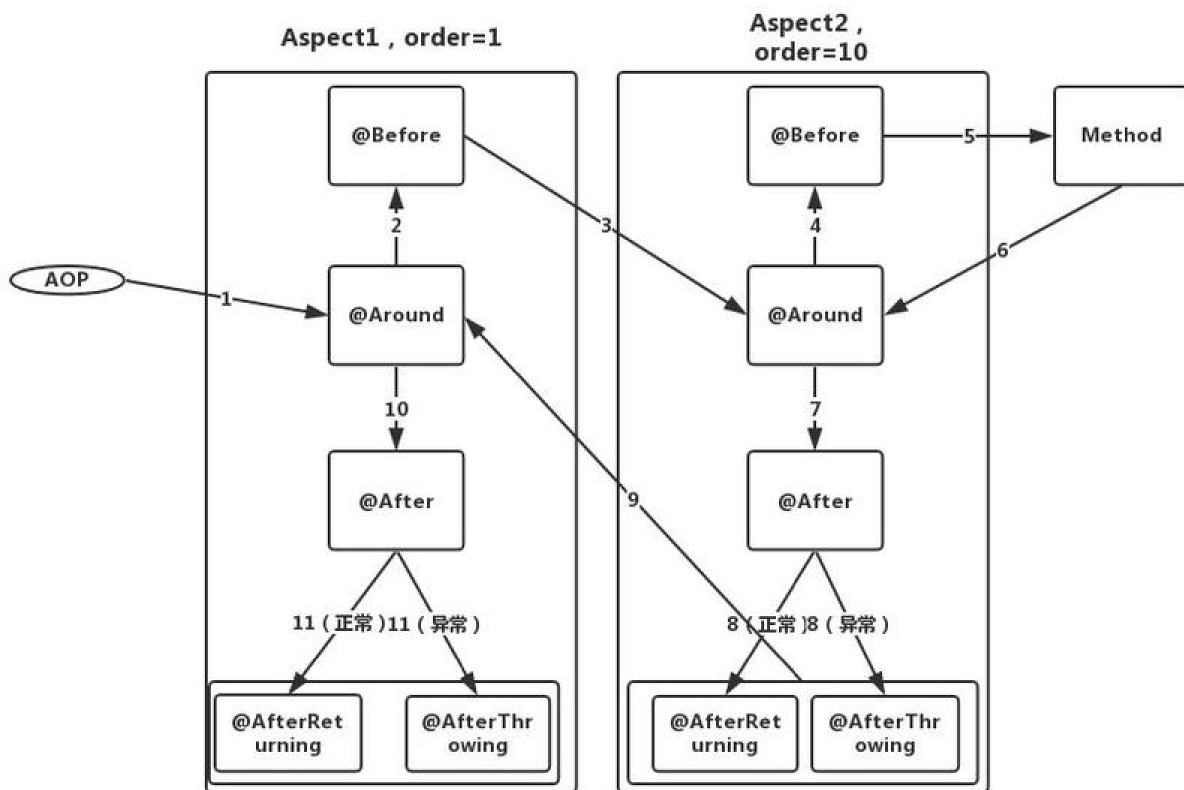


如果目标方法抛出异常

原文链接：SpringAOP 多切面



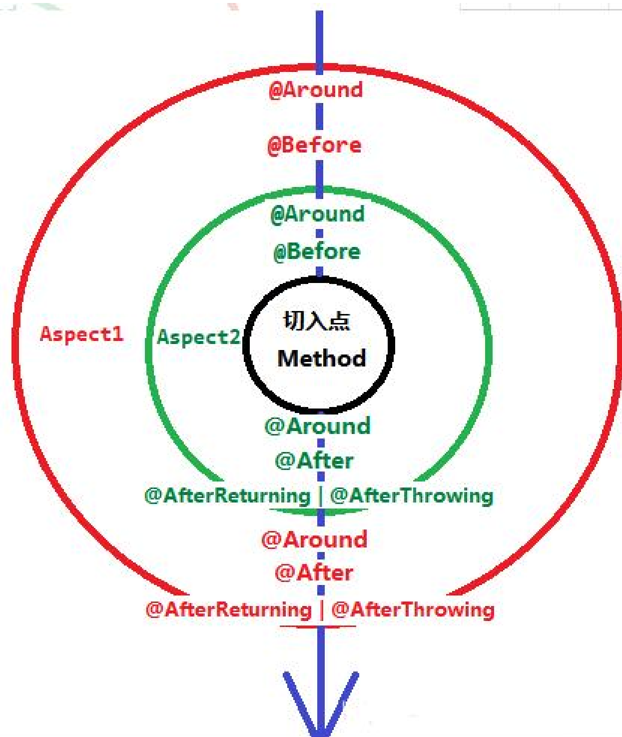
通知间的跳转顺序



多个切入点责任链模式如下图

Aspect1 和Aspect2两个切面类中所有通知类型的执行顺序，Method是具体的切入点，order代表优级，它根据一个int值来判断优先级的高低，数字越小，优先级越高！所以，不同的切面，实际上是环于切入点的同心圆

原文链接：SpringAOP 多切面



jdk代理的源码为(这里采用默认的jdk代理模式获取代理，cglib原理类似)：

```
/*
 * Copyright 2002-2018 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.springframework.aop.framework;

import java.io.Serializable;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.util.List;

import org.aopalliance.intercept.MethodInvocation;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import org.springframework.aop.AopInvocationException;
import org.springframework.aop.RawTargetAccess;
```

```
import org.springframework.aop.TargetSource;
import org.springframework.aop.support.AopUtils;
import org.springframework.core.DecoratingProxy;
import org.springframework.lang.Nullable;
import org.springframework.util.Assert;
import org.springframework.util.ClassUtils;

/**
 * JDK-based {@link AopProxy} implementation for the Spring AOP framework,
 * based on JDK {@link java.lang.reflect.Proxy dynamic proxies}.
 *
 * <p>Creates a dynamic proxy, implementing the interfaces exposed by
 * the AopProxy. Dynamic proxies <i>cannot</i> be used to proxy methods
 * defined in classes, rather than interfaces.
 *
 * <p>Objects of this type should be obtained through proxy factories,
 * configured by an {@link AdvisedSupport} class. This class is internal
 * to Spring's AOP framework and need not be used directly by client code.
 *
 * <p>Proxies created using this class will be thread-safe if the
 * underlying (target) class is thread-safe.
 *
 * <p>Proxies are serializable so long as all Advisors (including Advices
 * and Pointcuts) and the TargetSource are serializable.
 *
 * @author Rod Johnson
 * @author Juergen Hoeller
 * @author Rob Harrop
 * @author Dave Syer
 * @see java.lang.reflect.Proxy
 * @see AdvisedSupport
 * @see ProxyFactory
 */
final class JdkDynamicAopProxy implements AopProxy, InvocationHandler, Serializable {

    /** use serialVersionUID from Spring 1.2 for interoperability. */
    private static final long serialVersionUID = 5531744639992436476L;


    /*
     * NOTE: We could avoid the code duplication between this class and the CGLIB
     * proxies by refactoring "invoke" into a template method. However, this approach
     * adds at least 10% performance overhead versus a copy-paste solution, so we sacrifice
     * elegance for performance. (We have a good test suite to ensure that the different
     * proxies behave the same :-)
     * This way, we can also more easily take advantage of minor optimizations in each class.
     */

    /** We use a static Log to avoid serialization issues. */
    private static final Log logger = LogFactory.getLog(JdkDynamicAopProxy.class);

    /** Config used to configure this proxy. */
    private final AdvisedSupport advised;
```

```
    /**
     * Is the {@link #equals} method defined on the proxied interfaces?
     */
    private boolean equalsDefined;

    /**
     * Is the {@link #hashCode} method defined on the proxied interfaces?
     */
    private boolean hashCodeDefined;


    /**
     * Construct a new JdkDynamicAopProxy for the given AOP configuration.
     * @param config the AOP configuration as AdvisedSupport object
     * @throws AopConfigException if the config is invalid. We try to throw an informative
     * exception in this case, rather than let a mysterious failure happen later.
     */
    public JdkDynamicAopProxy(AdvisedSupport config) throws AopConfigException {
        Assert.notNull(config, "AdvisedSupport must not be null");
        if (config.getAdvisors().length == 0 && config.getTargetSource() == AdvisedSupport.E
PTY_TARGET_SOURCE) {
            throw new AopConfigException("No advisors and no TargetSource specified");
        }
        this.advised = config;
    }


    @Override
    public Object getProxy() {
        return getProxy(ClassUtils.getDefaultClassLoader());
    }

    @Override
    public Object getProxy(@Nullable ClassLoader classLoader) {
        if (logger.isTraceEnabled()) {
            logger.trace("Creating JDK dynamic proxy: " + this.advised.getTargetSource());
        }
        Class<?>[] proxiedInterfaces = AopProxyUtils.completeProxiedInterfaces(this.advised, tr
e);
        findDefinedEqualsAndHashCodeMethods(proxiedInterfaces);
        return Proxy.newProxyInstance(classLoader, proxiedInterfaces, this);
    }

    /**
     * Finds any {@link #equals} or {@link #hashCode} method that may be defined
     * on the supplied set of interfaces.
     * @param proxiedInterfaces the interfaces to introspect
     */
    private void findDefinedEqualsAndHashCodeMethods(Class<?>[] proxiedInterfaces) {
        for (Class<?> proxiedInterface : proxiedInterfaces) {
            Method[] methods = proxiedInterface.getDeclaredMethods();
            for (Method method : methods) {
                if (AopUtils.isEqualsMethod(method)) {
                    this.equalsDefined = true;
```

```
            }
            if (AopUtils.isHashCodeMethod(method)) {
                this.hashCodeDefined = true;
            }
            if (this.equalsDefined && this.hashCodeDefined) {
                return;
            }
        }
    }
}


/**
 * Implementation of {@code InvocationHandler.invoke}.
 * <p>Callers will see exactly the exception thrown by the target,
 * unless a hook method throws an exception.
 */
@Override
@Nullable
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    MethodInvocation invocation;
    Object oldProxy = null;
    boolean setProxyContext = false;

    TargetSource targetSource = this.advised.targetSource;
    Object target = null;

    try {
        if (!this.equalsDefined && AopUtils.isEqualsMethod(method)) {
            // The target does not implement the equals(Object) method itself.
            return equals(args[0]);
        }
        else if (!this.hashCodeDefined && AopUtils.isHashCodeMethod(method)) {
            // The target does not implement the hashCode() method itself.
            return hashCode();
        }
        else if (method.getDeclaringClass() == DecoratingProxy.class) {
            // There is only getDecoratedClass() declared -> dispatch to proxy config.
            return AopProxyUtils.ultimateTargetClass(this.advised);
        }
        else if (!this.advised.opaque && method.getDeclaringClass().isInterface() &&
                method.getDeclaringClass().isAssignableFrom(Advised.class)) {
            // Service invocations on ProxyConfig with the proxy config...
            return AopUtils.invokeJoinpointUsingReflection(this.advised, method, args);
        }

        Object retVal;

        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();
      Class<?> targetClass = (target != null ? target.getClass() : null);

      // Get the interception chain for this method.
      List<Object> chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice(met
od, targetClass);

      // Check whether we have any advice. If we don't, we can fallback on direct
      // reflective invocation of the target, and avoid creating a MethodInvocation.
      if (chain.isEmpty()) {
        // 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 p
oxying.
        Object[] argsToUse = AopProxyUtils.adaptArgumentsIfNecessary(method, args);
        retVal = AopUtils.invokeJoinpointUsingReflection(target, method, argsToUse);
      }
      else {
        // We need to create a method invocation...
        invocation = new ReflectiveMethodInvocation(proxy, target, method, args, targetCl
ss, chain);
        // Proceed to the joinpoint through the interceptor chain.
        retVal = invocation.proceed();
      }

      // Massage return value if necessary.
      Class<?> returnType = method.getReturnType();
      if (retVal != null && retVal == target &&
          returnType != Object.class && returnType.isInstance(proxy) &&
          !RawTargetAccess.class.isAssignableFrom(method.getDeclaringClass())) {
        // Special case: it returned "this" and the return type of the method
        // is type-compatible. Note that we can't help if the target sets
        // a reference to itself in another returned object.
        retVal = proxy;
      }
      else if (retVal == null && returnType != Void.TYPE && returnType.isPrimitive()) {
        throw new AopInvocationException(
            "Null return value from advice does not match primitive return type for: " + me
hod);
      }
      return retVal;
    }
    finally {
      if (target != null && !targetSource.isStatic()) {
        // Must have come from TargetSource.
        targetSource.releaseTarget(target);
      }
      if (setProxyContext) {
        // Restore old proxy.
        AopContext.setCurrentProxy(oldProxy);
      }
    }
```

```
    }


    /**
     * Equality means interfaces, advisors and TargetSource are equal.
     * <p>The compared object may be a JdkDynamicAopProxy instance itself
     * or a dynamic proxy wrapping a JdkDynamicAopProxy instance.
     */
    @Override
    public boolean equals(@Nullable Object other) {
        if (other == this) {
            return true;
        }
        if (other == null) {
            return false;
        }

        JdkDynamicAopProxy otherProxy;
        if (other instanceof JdkDynamicAopProxy) {
            otherProxy = (JdkDynamicAopProxy) other;
        }
        else if (Proxy.isProxyClass(other.getClass())) {
            InvocationHandler ih = Proxy.getInvocationHandler(other);
            if (!(ih instanceof JdkDynamicAopProxy)) {
                return false;
            }
            otherProxy = (JdkDynamicAopProxy) ih;
        }
        else {
            // Not a valid comparison...
            return false;
        }

        // If we get here, otherProxy is the other AopProxy.
        return AopProxyUtils.equalsInProxy(this.advised, otherProxy.advised);
    }

    /**
     * Proxy uses the hash code of the TargetSource.
     */
    @Override
    public int hashCode() {
        return JdkDynamicAopProxy.class.hashCode() * 13 + this.advised.getTargetSource().hash
ode();
    }

}
```

先看获取代理的方法

```
@Override
    public Object getProxy() {
        return getProxy(ClassUtils.getDefaultClassLoader());
    }
```

```
    @Override
    public Object getProxy(@Nullable ClassLoader classLoader) {
        if (logger.isTraceEnabled()) {
            logger.trace("Creating JDK dynamic proxy: " + this.advised.getTargetSource());
        }
        Class<?>[] proxiedInterfaces = AopProxyUtils.completeProxiedInterfaces(this.advised, tr
e);
        findDefinedEqualsAndHashCodeMethods(proxiedInterfaces);
        return Proxy.newProxyInstance(classLoader, proxiedInterfaces, this);
    }
```

最终调用了jdk生成代理的方法，并且该类实现类InvocationHandler接口，那么继续看invoke方法

```
/**
     * Implementation of {@code InvocationHandler.invoke}.
     * <p>Callers will see exactly the exception thrown by the target,
     * unless a hook method throws an exception.
     */
    @Override
    @Nullable
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        MethodInvocation invocation;
        Object oldProxy = null;
        boolean setProxyContext = false;

        TargetSource targetSource = this.advised.targetSource;
        Object target = null;

        try {
            if (!this.equalsDefined && AopUtils.isEqualsMethod(method)) {
                // The target does not implement the equals(Object) method itself.
                return equals(args[0]);
            }
            else if (!this.hashCodeDefined && AopUtils.isHashCodeMethod(method)) {
                // The target does not implement the hashCode() method itself.
                return hashCode();
            }
            else if (method.getDeclaringClass() == DecoratingProxy.class) {
                // There is only getDecoratedClass() declared -> dispatch to proxy config.
                return AopProxyUtils.ultimateTargetClass(this.advised);
            }
            else if (!this.advised.opaque && method.getDeclaringClass().isInterface() &&
                    method.getDeclaringClass().isAssignableFrom(Advised.class)) {
                // Service invocations on ProxyConfig with the proxy config...
                return AopUtils.invokeJoinpointUsingReflection(this.advised, method, args);
            }

            Object retVal;

            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();
            Class<?> targetClass = (target != null ? target.getClass() : null);

            // Get the interception chain for this method.
            List<Object> chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice(met
od, targetClass);

            // Check whether we have any advice. If we don't, we can fallback on direct
            // reflective invocation of the target, and avoid creating a MethodInvocation.
            if (chain.isEmpty()) {
                // 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 p
oxying.
                Object[] argsToUse = AopProxyUtils.adaptArgumentsIfNecessary(method, args);
                retVal = AopUtils.invokeJoinpointUsingReflection(target, method, argsToUse);
            }
            else {
                // We need to create a method invocation...
                invocation = new ReflectiveMethodInvocation(proxy, target, method, args, targetCl
ss, chain);
                // Proceed to the joinpoint through the interceptor chain.
                retVal = invocation.proceed();
            }

            // Massage return value if necessary.
            Class<?> returnType = method.getReturnType();
            if (retVal != null && retVal == target &&
                    returnType != Object.class && returnType.isInstance(proxy) &&
                    !RawTargetAccess.class.isAssignableFrom(method.getDeclaringClass())) {
                // Special case: it returned "this" and the return type of the method
                // is type-compatible. Note that we can't help if the target sets
                // a reference to itself in another returned object.
                retVal = proxy;
            }
            else if (retVal == null && returnType != Void.TYPE && returnType.isPrimitive()) {
                throw new AopInvocationException(
                        "Null return value from advice does not match primitive return type for: " + me
hod);
            }
            return retVal;
        }
        finally {
            if (target != null && !targetSource.isStatic()) {
                // Must have come from TargetSource.
                targetSource.releaseTarget(target);
            }
            if (setProxyContext) {
                // Restore old proxy.
                AopContext.setCurrentProxy(oldProxy);
```

原文链接：SpringAOP 多切面

```
            }
        }
    }
```

忽略了几种特殊情况，然后判断是否报漏代理，即业务代码能不能通过

AopContext.currentProxy()

方法获得，之后真正的进行多个切面的执行

1.先获取拦截器链 2.遍历该拦截器链 3.通过ReflectiveMethodInvocation方法的proceed()方法执行

获取拦截器链，应该是加载的时候每个方法对应的一个缓存

```
/**
     * Determine a list of {@link org.aopalliance.intercept.MethodInterceptor} objects
     * for the given method, based on this configuration.
     * @param method the proxied method
     * @param targetClass the target class
     * @return a List of MethodInterceptors (may also include InterceptorAndDynamicMethod
atchers)
     */
    public List<Object> getInterceptorsAndDynamicInterceptionAdvice(Method method, @Nul
able Class<?> targetClass) {
        MethodCacheKey cacheKey = new MethodCacheKey(method);
        List<Object> cached = this.methodCache.get(cacheKey);
        if (cached == null) {
            cached = this.advisorChainFactory.getInterceptorsAndDynamicInterceptionAdvice(
                    this, method, targetClass);
            this.methodCache.put(cacheKey, cached);
        }
        return cached;
    }
```

ReflectiveMethodInvocation的procceed()执行逻辑

```
@Override
    @Nullable
    public Object proceed() throws Throwable {
        //    We start with an index of -1 and increment early.
        if (this.currentInterceptorIndex == this.interceptorsAndDynamicMethodMatchers.size() -
1) {
            return invokeJoinpoint();
        }

        Object interceptorOrInterceptionAdvice =
                this.interceptorsAndDynamicMethodMatchers.get(++this.currentInterceptorIndex);
        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.getDecl
ringClass());
            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 {
            // It's an interceptor, so we just invoke it: The pointcut will have
            // been evaluated statically before this object was constructed.
            return ((MethodInterceptor) interceptorOrInterceptionAdvice).invoke(this);
        }
    }
```

可以看出该方法是一个递归，默认currentInterceptorIndex为0,如果不是链最后一个则执行，之后通递归实现链中下一个拦截器执行，如果是最后一个则执行实际被代理的方法，这样递归实现，能有效解决环绕通知中前后的执行顺序的正确性

版权声明：本文为博主原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接和本声明。

本文链接：https://blog.csdn.net/qq_32224047/article/details/107146328