2011

AOP 的实现机制

tengfei.fangtf

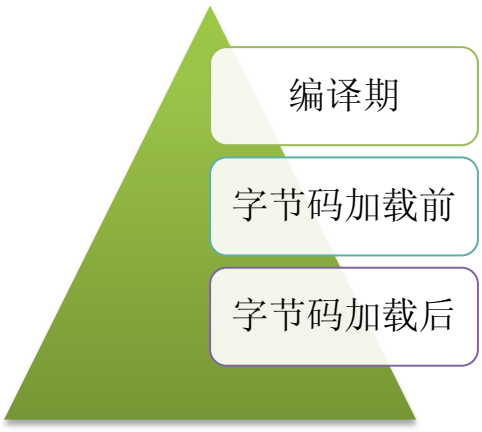
Alibaba IT

2011-10-16

目录

1 AOP 各种的实现

AOP 就是面向切面编程，我们可以从几个层面来实现 AOP。



在编译器修改源代码，在运行期字节码加载前修改字节码或字节码加载后动态创建代理类的字节码，以下是各种实现机制的比较。

类别	机制	原理	优点	缺点
静态 AOP	静态织入	在编译期，切面直接以字节码的形式编译到目标字节码文件中。	对系统无性能影响。	灵活性不够。
动态 AOP	动态代理	在运行期，目标类加载后，为接口动态生成代理类，将切面植入到代理类中。	相对于静态 AOP 更加灵活。	切入的关注点需要实现接口。对系统有一点性能影响。
	动态字节码生成	在运行期，目标类加载后，动态构建字节码文件生成目标类的子类，将切面逻辑加入到子类中。	没有接口也可以织入。	扩展类的实例方法为 final 时，则无法进行织入。
	自定义类加载器	在运行期，目标加载前，将切面逻辑加到目标字节码里。	可以对绝大部分类进行织入。	代码中如果使用了其他类加载器，则这些类将不会被织入。
	字节码转换	在运行期，所有类加载器加载字节码前，前进行拦截。	可以对所有类进行织入。	

2 AOP 里的公民

- **Joinpoint**：拦截点，如某个业务方法。
- **Pointcut**：Joinpoint 的表达式，表示拦截哪些方法。一个 Pointcut 对应多个 Joinpoint。
- **Advice**：要切入的逻辑。
  - **Before Advice** 在方法前切入。
  - **After Advice** 在方法后切入，抛出异常时也会切入。
  - **After Returning Advice** 在方法返回后切入，抛出异常则不会切入。
  - **After Throwing Advice** 在方法抛出异常时切入。
  - **Around Advice** 在方法执行前后切入，可以中断或忽略原有流程的执行。
- **公民之间的关系**

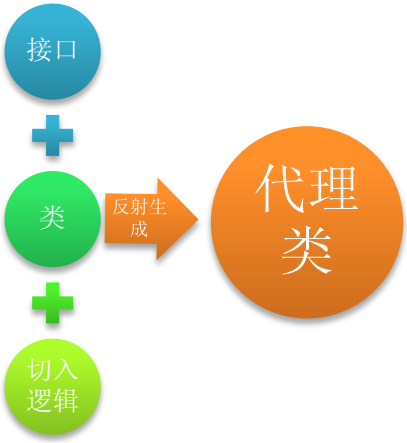
织入器通过在切面中定义 pointcut 来搜索目标（被代理类）的 JoinPoint(切入点)，然后把要切入的逻辑（Advice）织入到目标对象里，生成代理类。

3 AOP 的实现机制

本章节将详细介绍 AOP 有各种实现机制。

3.1 动态代理

Java 在 JDK1.3 后引入的动态代理机制，使我们可以在运行期动态的创建代理类。使用动态代理实现 AOP 需要有四个角色：被代理的类，被代理类的接口，织入器，和 InvocationHandler，而织入器使用接口反射机制生成一个代理类，然后在这个代理类中织入代码。被代理的类是 AOP 里所说的目标，InvocationHandler 是切面，它包含了 Advice 和 Pointcut。



3.1.1 使用动态代理

那如何使用动态代理来实现 AOP。下面的例子演示在方法执行前织入一段记录日志的代码，其中 Business 是代理类，LogInvocationHandler 是记录日志的切面，IBusiness, IBusiness2 是代理类的接口，Proxy.newProxyInstance 是织入器。

清单一：动态代理的演示

```
public static void main(String[] args) {
    //需要代理的接口，被代理类实现的多个接口都必须在这里定义
    Class[] proxyInterface = new Class[] { IBusiness.class,
IBusiness2.class };
    //构建AOP的Advice，这里需要传入业务类的实例
    LogInvocationHandler handler = new LogInvocationHandler(new
Business());
    //生成代理类的字节码加载器
    ClassLoader classLoader = DynamicProxyDemo.class.getClassLoader();
    //织入器，织入代码并生成代理类
    IBusiness2 proxyBusiness = (IBusiness2)
Proxy.newProxyInstance(classLoader, proxyInterface, handler);
    //使用代理类的实例来调用方法。
```

```
    proxyBusiness.doSomeThing2();
    ((IBusiness) proxyBusiness).doSomeThing();
}

/**
 * 打印日志的切面
 */
public static class LogInvocationHandler implements InvocationHandler {

    private Object target; //目标对象

    LogInvocationHandler(Object target) {
        this.target = target;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
        //执行原有逻辑
        Object rev = method.invoke(target, args);
        //执行织入的日志，你可以控制哪些方法执行切入逻辑
        if (method.getName().equals("doSomeThing2")) {
            System.out.println("记录日志");
        }
        return rev;
    }
}
```

接口IBusiness和IBusiness2定义省略。

业务类，需要代理的类。

```
public class Business implements IBusiness, IBusiness2 {

    @Override
    public boolean doSomeThing() {
        System.out.println("执行业务逻辑");
        return true;
    }

    @Override
    public void doSomeThing2() {
        System.out.println("执行业务逻辑2");
    }
```

```
}
```

输出

```
执行业务逻辑2
记录日志
执行业务逻辑
```

可以看到“记录日志”的逻辑切入到 Business 类的 doSomeThing 方法前了。

3.1.2 动态代理原理

本节将结合动态代理的源代码讲解其实现原理。动态代理的核心其实就是代理对象的生成，即 Proxy.*newProxyInstance*(classLoader, proxyInterface, handler)。让我们进入 newProxyInstance 方法观摩下，核心代码其实就三行。

清单二：生成代理类

```
//获取代理类
Class cl = getProxyClass(loader, interfaces);
//获取带有InvocationHandler参数的构造方法
Constructor cons = cl.getConstructor(constructorParams);
//把handler传入构造方法生成实例
return (Object) cons.newInstance(new Object[] { h });
```

其中 getProxyClass(loader, interfaces)方法用于获取代理类，它主要做了三件事情：在当前类加载器的缓存里搜索是否有代理类，没有则生成代理类并缓存在本地 JVM 里。

清单三：查找代理类

```
// 缓存的key使用接口名称生成的List
Object key = Arrays.asList(interfaceNames);
synchronized (cache) {
        do {
        Object value = cache.get(key);
         // 缓存里保存了代理类的引用
        if (value instanceof Reference) {
            proxyClass = (Class) ((Reference) value).get();
        }
        if (proxyClass != null) {
            // 代理类已经存在则返回
            return proxyClass;
        } else if (value == pendingGenerationMarker) {
            // 如果代理类正在产生，则等待
            try {
            cache.wait();
            } catch (InterruptedException e) {
            }
            continue;
        } else {
            //没有代理类，则标记代理准备生成
```

```
            cache.put(key, pendingGenerationMarker);
            break;
        }
        } while (true);
    }
```

代理类的生成主要是以下这两行代码。

清单四：生成并加载代理类

```
//生成代理类的字节码文件并保存到硬盘中
proxyClassFile =  ProxyGenerator.generateProxyClass(proxyName,
interfaces);
//使用类加载器将字节码加载到内存中
proxyClass = defineClass0(loader, proxyName,proxyClassFile, 0,
proxyClassFile.length);
```

ProxyGenerator.generateProxyClass()方法属于 sun.misc 包下，Oracle 并没有提供源代码，但是我们可以使用 JD-GUI 这样的反编译软件打开 jre\lib\rt.jar 来一探究竟，以下是其核心代码的分析。

清单五：代理类的生成过程

```
//添加接口中定义的方法，此时方法体为空
for (int i = 0; i < this.interfaces.length; i++) {
   localObject1 = this.interfaces[i].getMethods();
   for (int k = 0; k < localObject1.length; k++) {
       addProxyMethod(localObject1[k], this.interfaces[i]);
   }
}

//添加一个带有 InvocationHandler 的构造方法
MethodInfo localMethodInfo = new MethodInfo("<init>",
"(Ljava/lang/reflect/InvocationHandler;)V", 1);

//循环生成方法体代码（省略）
//方法体里生成调用 InvocationHandler 的 invoke 方法代码。（此处有所省略）
this.cp.getInterfaceMethodRef("InvocationHandler", "invoke", "Object; Method; Object;")

//将生成的字节码，写入硬盘
localFileOutputStream = new FileOutputStream(ProxyGenerator.access$000(this.val$name) +
".class");
localFileOutputStream.write(this.val$classFile);
```

那么通过以上分析，我们可以推出动态代理为我们生成了一个这样的代理类。把方法 doSomeThing 的方法体修改为调用 LogInvocationHandler 的 invoke 方法。

清单六：生成的代理类源码

```
public class ProxyBusiness implements IBusiness, IBusiness2 {

private LogInvocationHandler h;
```

```
@Override
public void doSomeThing2() {
    try {
        Method m = (h.target).getClass().getMethod("doSomeThing", null);
        h.invoke(this, m, null);
    } catch (Throwable e) {
        // 异常处理（略）
    }
}

@Override
public boolean doSomeThing() {
    try {
       Method m = (h.target).getClass().getMethod("doSomeThing2", null);
       return (Boolean) h.invoke(this, m, null);
    } catch (Throwable e) {
        // 异常处理（略）
    }
    return false;
}

public ProxyBusiness(LogInvocationHandler h) {
    this.h = h;
}

//测试用
public static void main(String[] args) {
    //构建AOP的Advice
    LogInvocationHandler handler = new LogInvocationHandler(new
Business());
    new ProxyBusiness(handler).doSomeThing();
    new ProxyBusiness(handler).doSomeThing2();
}
}
```

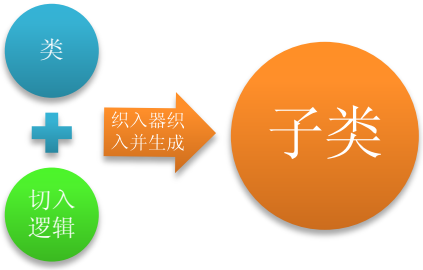
3.1.3 小结

从前两节的分析我们可以看出，动态代理在运行期通过接口动态生成代理类，这为其带来了一定的灵活性，但这个灵活性却带来了两个问题，第一代理类必须实现一个接口，如果没实现接口会抛出一个异常。第二性能影响，因为动态代理使用反射的机制实现的，首先反射肯定比直接调用要慢，经过测试大概每个代理类比静态代理多出 10 几毫秒的消耗。其次使用反射大量生成类文件可能引起 Full GC 造成性能影响，因为字节码文件加载后会存放在 JVM 运行时区的方法区（或者叫持久代）中，当方法区满的时候，会引起 Full GC，所以当

你大量使用动态代理时，可以将持久代设置大一些，减少 Full GC 次数。

3.2 动态字节码生成

使用动态字节码生成技术实现 AOP 原理是在运行期间目标字节码加载后，生成目标类的子类，将切面逻辑加入到子类中，所以使用 Cglib 实现 AOP 不需要基于接口。



本节介绍如何使用 Cglib 来实现动态字节码技术。Cglib 是一个强大的,高性能的 Code 生成类库，它可以在运行期间扩展 Java 类和实现 Java 接口，它封装了 Asm，所以使用 Cglib 前需要引入 Asm 的 jar。

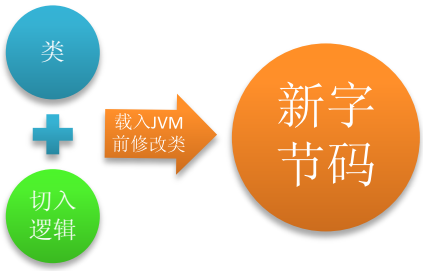
清单七：使用CGLib实现AOP

```
public static void main(String[] args) {
        byteCodeGe();
    }

    public static void byteCodeGe() {
        //创建一个织入器
        Enhancer enhancer = new Enhancer();
        //设置父类
        enhancer.setSuperclass(Business.class);
        //设置需要织入的逻辑
        enhancer.setCallback(new LogIntercept());
        //使用织入器创建子类
        IBusiness2 newBusiness = (IBusiness2) enhancer.create();
        newBusiness.doSomeThing2();
    }

    /**
     * 记录日志
     */
    public static class LogIntercept implements MethodInterceptor {

        @Override
        public Object intercept(Object target, Method method, Object[]
args, MethodProxy proxy) throws Throwable {
            //执行原有逻辑，注意这里是invokeSuper
```

```
            Object rev = proxy.invokeSuper(target, args);
            //执行织入的日志
            if (method.getName().equals("doSomeThing2")) {
                System.out.println("记录日志");
            }
            return rev;
        }
    }
```

3.3 自定义类加载器

如果我们实现了一个自定义类加载器，在类加载到 JVM 之前直接修改某些类的方法，并将切入逻辑织入到这个方法里，然后将修改后的字节码文件交给虚拟机运行，那岂不是更直接。



Javassist 是一个编辑字节码的框架，可以让你很简单地操作字节码。它可以在运行期定义或修改 Class。使用 Javassist 实现 AOP 的原理是在字节码加载前直接修改需要切入的方法。这比使用 Cglib 实现 AOP 更加高效，并且没太多限制，实现原理如下图：

我们使用系统类加载器启动我们自定义的类加载器，在这个类加载器里加一个类加载监听器，监听器发现目标类被加载时就织入切入逻辑，咱们再看看使用 Javassist 实现 AOP 的代码：

清单八：启动自定义的类加载器

```
//获取存放CtClass的容器ClassPool
ClassPool cp = ClassPool.getDefault();
//创建一个类加载器
Loader cl = new Loader();
//增加一个转换器
cl.addTranslator(cp, new MyTranslator());
```

```
//启动MyTranslator的main函数
cl.run("jsvassist.JavassistAopDemo$MyTranslator", args);
```

清单九：类加载监听器

```
public static class MyTranslator implements Translator {

	public void start(ClassPool pool) throws NotFoundException,
CannotCompileException {
	}

	/* *
	 * 类装载到JVM前进行代码织入
	 */
	public void onLoad(ClassPool pool, String classname) {
	    if (!"model$Business".equals(classname)) {
	        return;
	    }
	    //通过获取类文件
	    try {
	        CtClass  cc = pool.get(classname);
	        //获得指定方法名的方法
	        CtMethod m = cc.getDeclaredMethod("doSomeThing");
	        //在方法执行前插入代码
	        m.insertBefore("{ System.out.println(\"记录日志\"); }");
	    } catch (NotFoundException e) {
	    } catch (CannotCompileException e) {
	    }
	}

	public static void main(String[] args) {
	    Business b = new Business();
	    b.doSomeThing2();
	    b.doSomeThing();
	}
}
```

输出：

```
执行业务逻辑2
记录日志
执行业务逻辑
```

其中 Bussiness 类在本文的清单一中定义。看起来是不是特别简单，CtClass 是一个 class 文件的抽象描述。咱们也可以使用 insertAfter()在方法的末尾插入代码，使用 insertAt()在指定行插入代码。

3.3.1 小结

从本节中可知，使用自定义的类加载器实现 AOP 在性能上要优于动态代理和 Cglib，因为它不会产生新类，但是它仍然存在一个问题，就是如果其他的类加载器来加载类的话，这些类将不会被拦截。

3.4 字节码转换

自定义的类加载器实现 AOP 只能拦截自己加载的字节码，那么有没有一种方式能够监控所有类加载器加载字节码呢？有，使用 Instrumentation，它是 Java 5 提供的新特性，使用 Instrumentation，开发者可以构建一个字节码转换器，在字节码加载前进行转换。本节使用 Instrumentation 和 javassist 来实现 AOP。

3.4.1 构建字节码转换器

首先需要创建字节码转换器，该转换器负责拦截 Business 类，并在 Business 类的 doSomeThing 方法前使用 javassist 加入记录日志的代码。

```
public class MyClassFileTransformer implements ClassFileTransformer {

    /**
     * 字节码加载到虚拟机前会进入这个方法
     */
    @Override
    public byte[] transform(ClassLoader loader, String className,
Class<?> classBeingRedefined,
                            ProtectionDomain protectionDomain, byte[]
classfileBuffer)
            throws IllegalClassFormatException {
        System.out.println(className);
        //如果加载Business类才拦截
        if (!"model/Business".equals(className)) {
            return null;
        }

        //javassist的包名是用点分割的，需要转换下
        if (className.indexOf("/") != -1) {
            className = className.replaceAll("/", ".");
        }
        try {
            //通过包名获取类文件
```

```
            CtClass cc = ClassPool.getDefault().get(className);
            //获得指定方法名的方法
            CtMethod m = cc.getDeclaredMethod("doSomeThing");
            //在方法执行前插入代码
            m.insertBefore("{ System.out.println(\"记录日志\"); }");
            return cc.toBytecode();
        } catch (NotFoundException e) {
        } catch (CannotCompileException e) {
        } catch (IOException e) {
            //忽略异常处理
        }
        return null;
}
```

3.4.2 注册转换器

使用 premain 函数注册字节码转换器，该方法在 main 函数之前执行。

```
public class MyClassFileTransformer implements ClassFileTransformer {
    public static void premain(String options, Instrumentation ins) {
        //注册我自己的字节码转换器
        ins.addTransformer(new MyClassFileTransformer());
    }
}
```

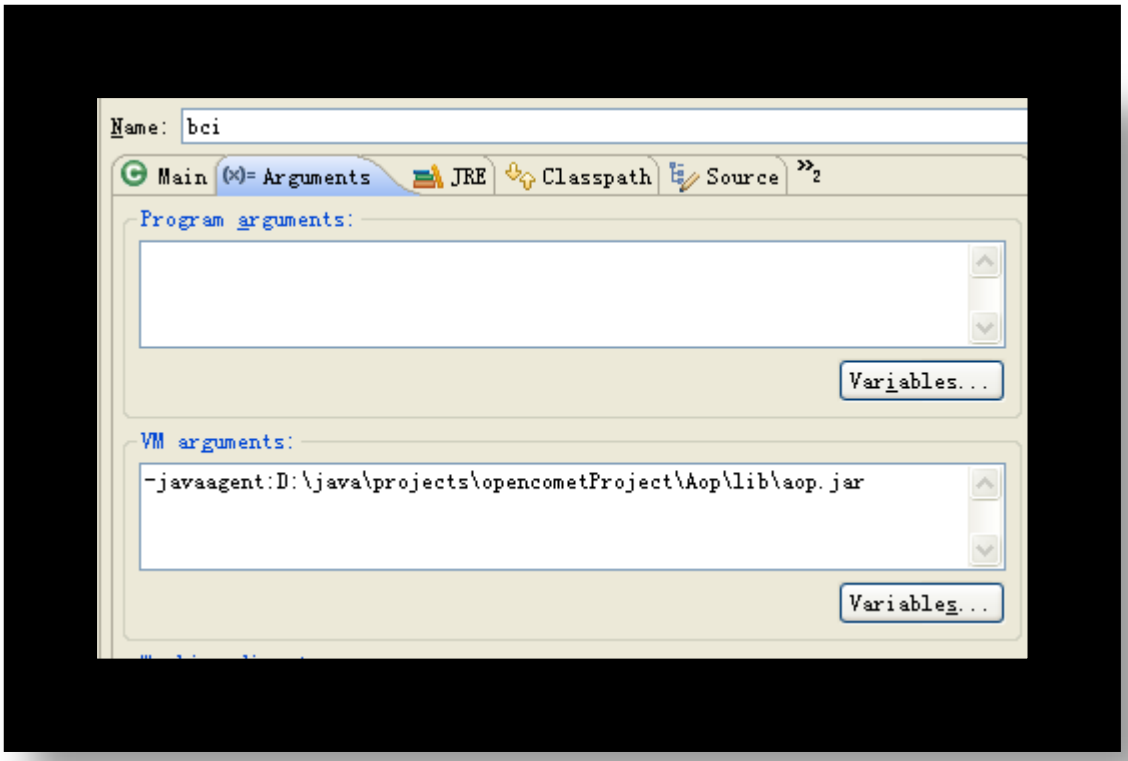
3.4.3 配置和执行

需要告诉 JVM 在启动 main 函数之前，需要先执行 premain 函数。首先需要将 premain 函数所在的类打成 jar 包。并修改该 jar 包里的 META-INF\MANIFEST.MF 文件。

```
Manifest-Version: 1.0
Premain-Class: bci. MyClassFileTransformer
```

然后在 JVM 的启动参数里加上。

```
-javaagent:D:\java\projects\opencometProject\Aop\lib\aop.jar
```



3.4.4 输出

执行 main 函数，你会发现切入的代码无侵入性的织入进去了。

```
public static void main(String[] args) {
   new Business().doSomeThing();
   new Business().doSomeThing2();
}
```

输出

```
model/Business
sun/misc/Cleaner
java/lang/Enum
model/IBusiness
model/IBusiness2
记录日志
执行业务逻辑
执行业务逻辑2
java/lang/Shutdown
java/lang/Shutdown$Lock
```

从输出中可以看到系统类加载器加载的类也经过了这里。

4 AOP 实战

说了这么多理论，那 AOP 到底能做什么呢？ AOP 能做的事情非常多。

- 性能监控，在方法调用前后记录调用时间，方法执行太长或超时报警。
- 缓存代理，缓存某方法的返回值，下次执行该方法时，直接从缓存里获取。
- 软件破解，使用 AOP 修改软件的验证类的判断逻辑。
- 记录日志，在方法执行前后记录系统日志。
- 工作流系统，工作流系统需要将业务代码和流程引擎代码混合在一起执行，那么我们可以使用 AOP 将其分离，并动态挂接业务。
- 权限验证，方法执行前验证是否有权限执行当前方法，没有则抛出没有权限执行异常，又业务代码捕捉。

以下实战是我在询盘管理的天使瀑布项目中使用 AOP 实现的一个简单的方法监控。代码不是很复杂，关键是将监控代码和业务代码的分离和复用。

4.1 方法监控

我使用 Spring AOP 监控询盘生成方法的调用次数，以便于观察整个询盘生成的过程。设计思路如下：

每个方法调用成功后，统计调用次数并存入缓存服务器，每天晚上 11 点 50 分从缓存服务器中获取数据并存入数据库。因为每天的方法调用次数近百万，为了降低数据库压力不能实时入库。

4.1.1 如何使用

只要配置了注解的方法将会被统计调用次数，有的方法需要方法调用成功后才记录，而下面这个方法要求返回值为 false 才记录:

```
@MethodInvokeTimesMonitor(value = "KEY_FILTER_NUM", returnValue = false)
public boolean evaluateMsg(String message) {}
```

4.1.2 如何配置

我使用的是 Spring2.5.5 和 AspectJ 的方式来配置 AOP，首先需要启用对 AspectJ 的支持。

```
启用AOP
xsi:schemaLocation="
```

```
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd"

<!--启用对aspectJ的支持-->
<aop:aspectj-autoproxy proxy-target-class="true"/>
```

proxy-target-class 设置为 true 表示让 Spring 使用 CGlib 来实现 AOP，配置为 false 表示使用动态代理实现 AOP，默认使用动态代理。其次定义`@MethodInvokeTimesMonitor`注解。

定义注解

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface MethodInvokeTimesMonitor {

    /**
     * 监控名称，和数据库存储字段名称保持一致
     */
    String value();

    /**
     * 要求返回值为空或等returnValue才记录
     */
    boolean returnValue() default true;
}
```

最后定义一个切面，在切面中定义拦截的方法和在方法返回后记录调用次数的 Advice，我们在这里定义了拦截所有配置了注解的方法。

切面

```
@Aspect
public class MethodAspect {

    @Resource
    private XpCacheClient eqUserFloattedCacheClient;

    /**
     * 切入点，所有配置MethodInvokeTimesMonitor注解的方法
     */

@Pointcut("@annotation(com.alibaba.myalibaba.eq.monitor.MethodInvokeTimesMonitor)")
    public void allMethodInvokeTimesMonitor() {
    }

    /**
     * 统计方法的调用次数
     *
     * @param methodInvokeTimesMonitor 注解传递的参数
```

```
     */
    @AfterReturning(value = "MethodAspect.allMethodInvokeTimesMonitor()
&& @annotation(methodInvokeTimesMonitor)", returning = "retVal")
    public void statInvokeTimes(MethodInvokeTimesMonitor
methodInvokeTimesMonitor, Object retVal) {
        String name = methodInvokeTimesMonitor.value();
        //获取方法的返回值
        boolean returnValue = methodInvokeTimesMonitor.returnValue();
        //如果返回值不为空，则判断返回值是否和要求的返回值一致，如果一致则记录调用
次数
        if (retVal != null && retVal instanceof Boolean && ((Boolean) retVal
== returnValue)) {
            statInvokeTimes(name);
        }
        //如果无返回值，则直接记录调用次数
        if (retVal == null) {
            statInvokeTimes(name);
        }
    }

    private void statInvokeTimes(String name) {
        //只缓存当天的数据
        String key = getCacheKey(name);

        //没有则为1，有则自增长1
        Integer num = eqUserFloattedCacheClient.get(key);
        if (num == null) {
            eqUserFloattedCacheClient.put(key, 1);
        } else {
            eqUserFloattedCacheClient.syncPut(key, ++num);
        }
    }

    private String getCacheKey(String name) {
        return Calendar.getInstance().get(Calendar.DAY_OF_MONTH) + "_" +
name;
    }

}
```

@Pointcut 用于定义切入点表达式，为了表达式可以复用，所以在单独的方法上配置。@AfterReturning 表示在方法执行后进行切入，里面的 MethodAspect.allMethodInvokeTimesMonitor()表示使用这个方法的切入点表达式，而@annotation(methodInvokeTimesMonitor)表示将当参数传递给 statInvokeTimes 方法，returning = "retVal"则表示将被切入方法的返回值赋值给 retVal，并传递给

statInvokeTimes 方法。

定义 MethodAspect 切面为 Spring 的 Bean，如果不配置则 AOP 不会生效。

```
<bean class="com.alibaba.myalibaba.eq.commons.monitor.MethodAspect"/>
```

4.1.3 Spring 的 AOP

Spring 默认采取的**动态代理**机制实现 AOP，当动态代理不可用时（代理类无接口）会使用 **CGlib** 机制。但 Spring 的 AOP 有一定的缺点，第一个**只能对方法进行切入**，不能对接口，字段，静态代码块进行切入（切入接口的某个方法，则该接口下所有实现类的该方法将被切入）。第二个**同类中的互相调用方法将不会使用代理类。**因为要使用代理类必须从 Spring 容器中获取 Bean。第三个**性能不是最好的**，从 3.3 章节我们得知使用自定义类加载器，性能要优于动态代理和 CGlib。

可以获取代理类

```
public IMsgFilterService getThis()
{
       return (IMsgFilterService) AopContext.currentProxy();
}

public boolean evaluateMsg () {
   // 执行此方法将不会织入切入逻辑
  return getThis().evaluateMsg(String message);
}

@MethodInvokeTimesMonitor("KEY_FILTER_NUM")
public boolean evaluateMsg(String message) {
```

不能获取代理类

```
public boolean evaluateMsg () {
   // 执行此方法将不会织入切入逻辑
  return evaluateMsg(String message);
}

@MethodInvokeTimesMonitor("KEY_FILTER_NUM")
public boolean evaluateMsg(String message) {
```

4.2 参考资料

- Java 动态代理机制分析及扩展
- CGlib 的官方网站
- ASM 官方网站
- JbossAOP
- Java5 特性 Instrumenttation 实践