

在 Java 中反射随处可见，它底层的原也比较有意思，这篇文章来详细介绍反射背后的原理。

先来看下面这个例子：

```
public class ReflectionTest {  
  
    private static int count = 0;  
    public static void foo() {  
        new Exception("test#" +  
(count++)).printStackTrace();  
    }  
  
    public static void main(String[] args) throws  
Exception {  
        Class<?> clz =  
Class.forName("ReflectionTest");  
        Method method = clz.getMethod("foo");  
        for (int i = 0; i < 20; i++) {  
            method.invoke(null);  
        }  
    }  
}
```

运行结果如下

可以看到同一段代码，运行的堆栈结果与执行次数有关系，在 0 ~ 15 次调用方式

为sun.reflect.NativeMethodAccessorImpl.invoke0，从第 16 次开始调用方式变为了

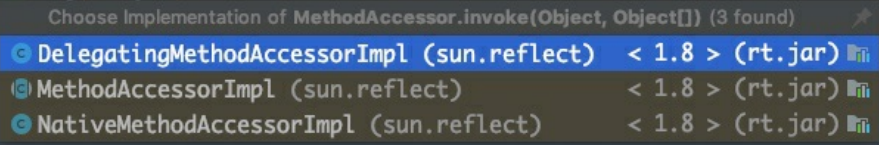
sun.reflect.GeneratedMethodAccessor1.invoke。原因是什么呢？继续往下看。

# 反射方法源码分析

Method.invoke 源码如下:

```
@CallerSensitive
public Object invoke(Object obj, Object... args)
    throws IllegalAccessException, IllegalArgumentException,
        InvocationTargetException
{
    if (!override) {
        if (!Reflection.quickCheckMemberAccess(clazz, modifiers)) {
            Class<?> caller = Reflection.getCallerClass();
            checkAccess(caller, clazz, obj, modifiers);
        }
    }
    MethodAccessor ma = methodAccessor; // read volatile
    if (ma == null) {
        ma = acquireMethodAccessor();
    }
    return ma.invoke(obj, args);
}

/**
 * Returns {@code}
 */
```



可以最终调用了MethodAccessor.invoke方法,  
MethodAccessor 是一个接口

```
public interface MethodAccessor {
    public Object invoke(Object obj, Object[]
args)
        throws IllegalArgumentException,
        InvocationTargetException;
}
```

从输出的堆栈可以看到 MethodAccessor 的实现类是委托类  
DelegatingMethodAccessorImpl, 它的 invoke 函数非常简单,  
就是把调用委托给了真正的实现类。

```
class DelegatingMethodAccessorImpl extends
MethodAccessorImpl {
    private MethodAccessorImpl delegate;
    public Object invoke(Object obj, Object[]
args)
        throws IllegalArgumentException,
InvocationTargetException
    {
        return delegate.invoke(obj, args);
    }
}
```

通过堆栈可以看到在第 0 ~ 15 次调用中，实现类是 NativeMethodAccessorImpl，从第 16 次调用开始实现类是 GeneratedMethodAccessor1，为什么是这样呢？玄机就在 NativeMethodAccessorImpl 的 invoke 方法中

```
public Object invoke(Object obj, Object[] args)
    throws IllegalArgumentException, InvocationTargetException
{
    // We can't inflate methods belonging to vm-anonymous classes because
    // that kind of class can't be referred to by name, hence can't be
    // found from the generated bytecode
    if (++numInvocations > ReflectionFactory.inflationThreshold() 默认值为 15
        && !ReflectUtil.isVMAnonymousClass(method.getDeclaringClass())) {
        MethodAccessorImpl acc = (MethodAccessorImpl)
            new MethodAccessorGenerator().
                generateMethod(method.getDeclaringClass(),
                    method.getName(),
                    method.getParameterTypes(),
                    method.getReturnType(),
                    method.getExceptionTypes(),
                    method.getModifiers());
        parent.setDelegate(acc); 15 次以后使用新的实现类
    }
    return invoke0(method, obj, args);
}
```

前 0 ~ 15 次都会调用到 invoke0，这是一个 native 的函数。

```
private static native Object invoke0(Method m,
Object obj, Object[] args);
```

有兴趣的同学可以去看一下 Hotspot 的源码，依次跟踪下面的代码和函数：

```
./jdk/src/share/native/sun/reflect/NativeAccessors
.C
JNIEXPORT jobject JNICALL
Java_sun_reflect_NativeMethodAccessorImpl_invoke0
(JNIEnv *env, jclass unused, jobject m, jobject
obj, jobjectArray args)

./hotspot/src/share/vm/prims/jvm.cpp
JVM_ENTRY(jobject, JVM_InvokeMethod(JNIEnv *env,
jobject method, jobject obj, jobjectArray args0))

./hotspot/src/share/vm/runtime/reflection.cpp
oop Reflection::invoke_method(oop method_mirror,
Handle receiver, objArrayHandle args, TRAPS)
```

这里不详细展开 native 实现的细节。

15 次以后会走新的逻辑，使用 GeneratedMethodAccessor1 来调用反射的方法。MethodAccessorGenerator 的作用是通过 ASM 生成新的类 sun.reflect.GeneratedMethodAccessor1。为了查看整个类的内容，可以使用阿里的 [arthas](https://alibaba.github.io/arthas) (<https://alibaba.github.io/arthas>) 工具。修改上面的代码，在 main 函数的最后加上 System.in.read(); 让 JVM 进程不要退出。

执行 arthas 工具中的 ./as.sh，会要求输入 JVM 进程

选择在运行的 ReflectionTest 进程号 7 就进入到了 arthas 交互性界面。执行 dump sun.reflect.GeneratedMethodAccessor1 文件就保存到了本地。

来看下这个类的字节码

人肉翻译一下这个字节码，忽略掉异常处理以后的代码如下

```
public class GeneratedMethodAccessor1 extends
MethodAccessorImpl {
    @Override
    public Object invoke(Object obj, Object[]
args)
        throws IllegalArgumentException,
InvocationTargetException {
        ReflectionTest.foo();
        return null;
    }
}
```

那为什么要采用 0 ~ 15 次使用 native 方式来调用，15 次以后使用 ASM 新生成的类来处理反射的调用呢？

一切都是基于性能的考虑。JNI native 调用的方式要比动态生成类调用的方式慢 20 倍，但是又由于第一次字节码生成的过程比较慢。如果反射仅调用一次的话，采用生成字节码的方式反而比 native 调用的方式慢 3 ~ 4 倍。

## inflation 机制

因为很多情况下，反射只会调用一次，因此 JVM 想了一招，设置了 15 这个 `sun.reflect.inflationThreshold` 阈值，反射方法调用超过 15 次时（从 0 开始），采用 ASM 生成新的类，保证后面的调用比 native 要快。如果小于 15 次的情况下，还不如生成直接 native 来的简单直接，还不造成额外类的生成、校验、加载。这种方式被称为「inflation 机制」。inflation 这个单词也比较有意思，它的字面意思是「膨胀；通货膨胀」。

JVM 与 inflation 相关的属性有两个，一个是刚提到的阈值 `sun.reflect.inflationThreshold`，还有一个是是否禁用 inflation 的属性 `sun.reflect.noInflation`，默认值为 `false`。如果把这个值设置成 `true` 的话，从第 0 次开始就使用动态生成类的方式来调用反射方法了，不会使用 `native` 的方式。

增加 `noInflation` 属性重新执行上述 Java 代码

```
java -cp . -Dsun.reflect.noInflation=true  
ReflectionTest
```

输出结果为

```
java.lang.Exception: test#0  
    at  
ReflectionTest.foo(ReflectionTest.java:10)  
    at  
sun.reflect.GeneratedMethodAccessor1.invoke(Unknown  
Source)  
    at  
java.lang.reflect.Method.invoke(Method.java:497)  
    at  
ReflectionTest.main(ReflectionTest.java:18)  
java.lang.Exception: test#1  
    at  
ReflectionTest.foo(ReflectionTest.java:10)  
    at  
sun.reflect.GeneratedMethodAccessor1.invoke(Unknown  
Source)  
    at  
java.lang.reflect.Method.invoke(Method.java:497)  
    at  
ReflectionTest.main(ReflectionTest.java:18)
```

可以看到，从第 0 次开始就已经没有使用 native 方法来调用反射方法了。

## 小结

这篇文章主要介绍了 Java 方法反射调用底层的原理，主要有两种方式

- native 方法
- 动态生成类的方式

native 调用的方式比 Java 类直接调用的方式慢 20 倍，但是第一次生成动态类又比较耗时，因此 JVM 才有了一个优化策略，在某阈值之前使用 native 调用，在此阈值之后使用动态生成类的方式。这样既可以保证在反射方法少数调用的情况下，不用生成新的类，又可以保证调用次数很多的情况下使用性能更优的动态类的方式。

## 思考题

现实中大量使用反射调用的项目，inflation 机制可能造成哪些隐患呢？