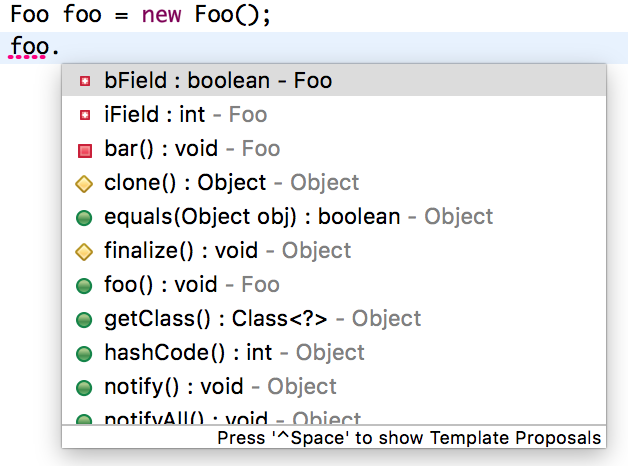
JVM如何实现反射

反射在java中的应用非常广泛。开发人员日常用到的Java集成环境（IDE）就使用了这个功能：每当敲入句点的时候，IDE便会根据点号前的内容动态展示可以访问的字段或者方法。



Spring框架的依赖反转（Ioc）就利用了反射的机制。但是反射的性能开销很大。

## 反射调用的实现

理解Method.invoke实现

public final class Method extends Executable {

...

public Object invoke(Object obj, Object... args) throws ... {

... // 权限检查

MethodAccessor ma = methodAccessor;

if (ma == null) {

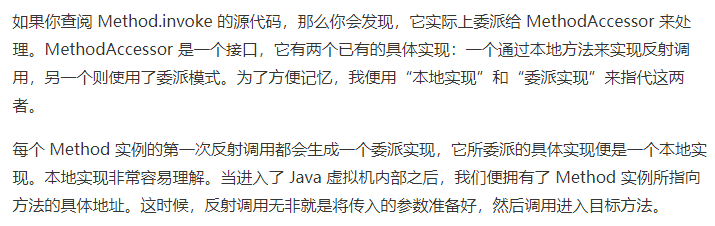
ma = acquireMethodAccessor();

}

return ma.invoke(obj, args);

}

}



// v0 版本

import java.lang.reflect.Method;

public class Test {

public static void target(int i) {

new Exception("#" + i).printStackTrace();

}

public static void main(String[] args) throws Exception {

Class<?> klass = Class.forName("Test");

Method method = klass.getMethod("target", int.class);

method.invoke(null, 0);

}

}

# 不同版本的输出略有不同，这里我使用了 Java 10。

$ java Test

java.lang.Exception: #0

at Test.target(Test.java:5)

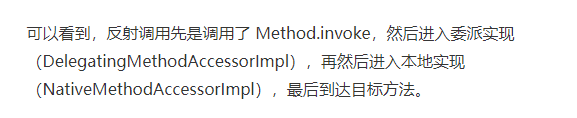
at java.base/jdk.internal.reflect.NativeMethodAccessorImpl .invoke0(Native Method)

a t java.base/jdk.internal.reflect.NativeMethodAccessorImpl. .invoke(NativeMethodAccessorImpl.java:62)

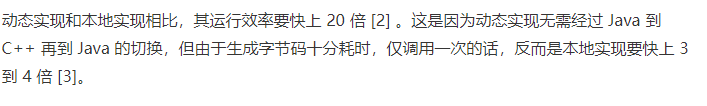
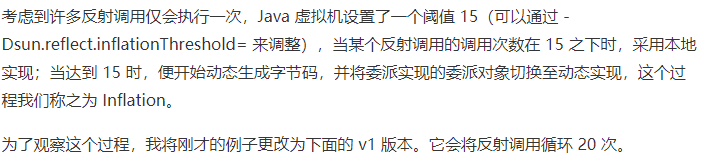
t java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.i .invoke(DelegatingMethodAccessorImpl.java:43)

java.base/java.lang.reflect.Method.invoke(Method.java:564)

t Test.main(Test.java:131



之所以使用委派实现而不直接使用本地实现，是为了能够在本地实现以及动态实现中切换。

**public class Test** {  
 **public static void** target(**int** i){  
 **new** Exception("#" + i).printStackTrace();  
 }  
  
 **public static void** main(**String**[] args) **throws Exception** {  
 **Class**<?> klass = **Class**.forName("cn.lxj.jvm40courses.course8.Test");  
 **Method** method = klass.getMethod("target", **int**.**class**);  
 **for** (**int** i =0; i < 20; i ++) {  
 method.invoke(**null**, i);  
 }  
 }  
}

# 使用 -verbose:class 打印加载的类

java.lang.Exception: #14

at cn.lxj.jvm40courses.course8.Test.target(Test.java:12)

at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)

at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)

at java.lang.reflect.Method.invoke(Method.java:498)

at cn.lxj.jvm40courses.course8.Test.main(Test.java:19)

…………..

java.lang.Exception: #15

at cn.lxj.jvm40courses.course8.Test.target(Test.java:12)

at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)

at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)

at java.lang.reflect.Method.invoke(Method.java:498)

at cn.lxj.jvm40courses.course8.Test.main(Test.java:19)

java.lang.Exception: #16

at cn.lxj.jvm40courses.course8.Test.target(Test.java:12)

at sun.reflect.GeneratedMethodAccessor1.invoke(Unknown Source)

at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)

at java.lang.reflect.Method.invoke(Method.java:498)

at cn.lxj.jvm40courses.course8.Test.main(Test.java:19)

反射调用Inflation机制可以通过参数（-Dsun.reflect.noInflation = true）来关闭。这样的话，在反射调用一开始便会直接生成动态参数，而不会使用委派实现或者本地实现。

## 反射调用的开销

Class.forname会调用本地方法，Class.getMethod会遍历该类的公有方法。如果没有匹配到，它还将遍历父类的公有方法。可想，这两个方法非常耗时。

注意：以getMethod为代表的查询方法操作，会返回查找得到结果的一份拷贝。因此，应当避免在热点代码中使用返回Method数组的getMethods或者getDeclaredMethods方法，以减少不必要的堆空间开下。

1. 变长方法导致的Object数组
2. 基本数据类型的自动拆装箱
3. 方法内联

### -verbose:class 加载的类

打印加载的类

### -Dsun.reflect.inflationThreshold=值

这个参数来调整反射调用的次数默认在15次之下的时候，采用本地实现C++。多余等于的时候会将委派实例的委派对象切换为动态实现。（动态生成字节码会十分耗时）

动态实现和本地实现相比，速度要快上20倍。这是因为动态无需经过java到c++再到java的切换，但是由于生成字节码又十分耗时。仅一次调用的话，反而是本地实现要快上三到四倍。因而Java虚机设置了一个阈值15。

### -Dsun.reflect.noInflation = true

用来关闭反射的inflation机制，这样一来，在反射开始的时候就直接生成动态实现，而不会用委派实现或者本地实现。

### -Djava.lang.Integer.IntegerCashe.high = 值

将java的自动拆装箱的缓存对象大小设置为指定的值。Int默认是127

### -XX:TypeProfileWidth=值

默认值为2，该参数为java虚机关于每个调用能够记录的类型数据