反射是 Java 语言中一个相当重要的特性，它允许正在运行的 Java 程序观测，甚至是修改程序的动态行为。

举例来说，我们可以通过 Class 对象枚举该类中的所有方法，我们还可以通过 Method.setAccessible（位于 java.lang.reflect 包，该方法继承自 AccessibleObject）绕过 Java 语言的访问权限，在私有方法所在类之外的地方调用该方法。

反射在 Java 中的应用十分广泛。开发人员日常接触到的 Java 集成开发环境（IDE）便运用了这一功能：每当我们敲入点号时，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。

在 Web 开发中，我们经常能够接触到各种可配置的通用框架。为了保证框架的可扩展性，它们往往借助 Java 的反射机制，根据配置文件来加载不同的类。举例来说，Spring 框架的依赖反转（IoC），便是依赖于反射机制。

然而，我相信不少开发人员都嫌弃反射机制比较慢。甚至是甲骨文关于反射的教学网页 [1]，也强调了反射性能开销大的缺点。

今天我们便来了解一下反射的实现机制，以及它性能糟糕的原因。

1. 反射调用的实现

首先，我们来看看方法的反射调用，也就是 Method.invoke，是怎么实现的。

```
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);
  }
}
```

如果你查阅 Method.invoke 的源代码，那么你会发现，它实际上委派给 MethodAccessor 来处理。MethodAccessor 是一个接口，它有两个已有的具体实现：一个通过本地方法来实现反射调用，另一个则使用了委派模式。为了方便记忆，我便用“本地实现”和“委派实现”来指代这两者。

每个 Method 实例的第一次反射调用都会生成一个委派实现，它所委派的具体实现便是一个本地实现。本地实现非常容易理解。当进入了 Java 虚拟机内部之后，我们便拥有了 Method 实例所指向方法的具体地址。这时候，反射调用无非就是将传入的参数准备好，然后调用进入目标方法。

```
// 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)
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.
.invoke(NativeMethodAccessorImpl.java:62)
        at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.i
.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.base/java.lang.reflect.Method.invoke(Method.java:564)
        at Test.main(Test.java:131)
```

为了方便理解，我们可以打印一下反射调用到目标方法时的栈轨迹。在上面的 v0 版本代码中，我们获取了一个指向 Test.target 方法的 Method 对象，并且用它来进行反射调用。在 Test.target 中，我会打印出栈轨迹。

可以看到，反射调用先是调用了 Method.invoke，然后进入委派实现（DelegatingMethodAccessorImpl），再然后进入本地实现（NativeMethodAccessorImpl），最后到达目标方法。

这里你可能会疑问，为什么反射调用还要采取委派实现作为中间层？直接交给本地实现不可以么？

其实，Java 的反射调用机制还设立了另一种动态生成字节码的实现（下称动态实现），直接使用 invoke 指令来调用目标方法。之所以采用委派实现，便是为了能够在本地实现以及动态实现中切换。

动态实现和本地实现相比，其运行效率要快上 20 倍 [2] 。这是因为动态实现无需经过 Java 到 C++ 再到 Java 的切换，但由于生成字节码十分耗时，仅调用一次的话，反而是本地实现要快上 3 到 4 倍 [3]。

考虑到许多反射调用仅会执行一次，Java 虚拟机设置了一个阈值 15（可以通过 -Dsun.reflect.inflationThreshold= 来调整），当某个反射调用的调用次数在 15 之下时，采用本地实现；当达到 15 时，便开始动态生成字节码，并将委派实现的委派对象切换至动态实现，这个过程我们称之为 Inflation。

```
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);
    for (int i = 0; i < 20; i++) {
      method.invoke(null, i);
    }
  }
}

# 使用 -verbose:class 打印加载的类
$ java -verbose:class Test
...
java.lang.Exception: #14
        at Test.target(Test.java:5)
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl .invoke0(Native
Method)
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl
.invoke(NativeMethodAccessorImpl.java:62)
        at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl
.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.base/java.lang.reflect.Method.invoke(Method.java:564)
```

```
        at Test.main(Test.java:12)
[0.158s][info][class,load] ...
...
[0.160s][info][class,load] jdk.internal.reflect.GeneratedMethodAccessor1 source:
__JVM_DefineClass__
java.lang.Exception: #15
        at Test.target(Test.java:5)
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl .invoke0(Native
Method)
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl
.invoke(NativeMethodAccessorImpl.java:62)
        at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl
.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.base/java.lang.reflect.Method.invoke(Method.java:564)
        at Test.main(Test.java:12)
java.lang.Exception: #16
        at Test.target(Test.java:5)
        at jdk.internal.reflect.GeneratedMethodAccessor1 .invoke(Unknown Source)
        at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl
.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.base/java.lang.reflect.Method.invoke(Method.java:564)
        at Test.main(Test.java:12)
...
```

可以看到，在第 15 次（从 0 开始数）反射调用时，我们便触发了动态实现的生成。这时候，Java 虚拟机额外加载了不少类。其中，最重要的当属 GeneratedMethodAccessor1（第 30 行）。并且，从第 16 次反射调用开始，我们便切换至这个刚刚生成的动态实现（第 40 行）。

反射调用的 Inflation 机制是可以通过参数（-Dsun.reflect.noInflation=true）来关闭的。这样一来，在反射调用一开始便会直接生成动态实现，而不会使用委派实现或者本地实现。

2.反射调用的开销

在刚才的例子中，我们先后进行了 Class.forName，Class.getMethod 以及 Method.invoke 三个操作。其中，Class.forName 会调用本地方法，Class.getMethod 则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。可想而知，这两个操作都非常费时。

值得注意的是，以 getMethod 为代表的查找方法操作，会返回查找得到结果的一份拷贝。因此，我们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消耗。

在实践中，我们往往会在应用程序中缓存 Class.forName 和 Class.getMethod 的结果。因此，下面我就只关注反射调用本身的性能开销。

第一，由于 Method.invoke 是一个变长参数方法，在字节码层面它的最后一个参数会是 Object 数组（感兴趣的同学私下可以用 javap 查看）。Java 编译器会在方法调用处生成一个长度为传入参数数量的 Object 数组，并将传入参数一一存储进该数组中。

第二，由于 Object 数组不能存储基本类型，Java 编译器会对传入的基本类型参数进行自动装箱。

3.总结

在默认情况下，方法的反射调用为委派实现，委派给本地实现来进行方法调用。在调用超过 15 次之后，委派实现便会将委派对象切换至动态实现。这个动态实现的字节码是自动生成的，它将直接使用 invoke 指令来调用目标方法。

方法的反射调用会带来不少性能开销，原因主要有三个：变长参数方法导致的 Object 数组，基本类型的自动装箱、拆箱，还有最重要的方法内联。

通常来说，使用反射 API 的第一步便是获取 Class 对象。在 Java 中常见的有这么三种。

1. 使用静态方法 Class.forName 来获取。
2. 调用对象的 getClass() 方法。
3. 直接用类名 +“.class”访问。对于基本类型来说，它们的包装类型（wrapper classes）拥有一个名为“TYPE”的 final 静态字段，指向该基本类型对应的 Class 对象。

例如，Integer.TYPE 指向 int.class。对于数组类型来说，可以使用类名 +“[ ].class”来访问，如 int[ ].class。

除此之外，Class 类和 java.lang.reflect 包中还提供了许多返回 Class 对象的方法。例如，对于数组类的 Class 对象，调用 Class.getComponentType() 方法可以获得数组元素的类型。

一旦得到了 Class 对象，我们便可以正式地使用反射功能了。下面我列举了较为常用的几项。

1. 使用 newInstance() 来生成一个该类的实例。它要求该类中拥有一个无参数的构造器。
2. 使用 isInstance(Object) 来判断一个对象是否该类的实例，语法上等同于 instanceof 关键字（JIT 优化时会有差别，我会在本专栏的第二部分详细介绍）。
3. 使用 Array.newInstance(Class,int) 来构造该类型的数组。
4. 使用 getFields()/getConstructors()/getMethods() 来访问该类的成员。除了这三个之外，Class 类还提供了许多其他方法，详见 [4]。需要注意的是，方法名中带 Declared 的不会返回父类的成员，但是会返回私有成员(本类声明位private的field和method)；而不带 Declared 的则相反。