[**大白话说Java反射：入门、使用、原理**](https://www.cnblogs.com/chanshuyi/p/head_first_of_reflection.html)

https://www.cnblogs.com/chanshuyi/p/head\_first\_of\_reflection.html

[**目录**](https://www.cnblogs.com/chanshuyi/p/head_first_of_reflection.html#cb_post_title_url)

* [一个简单的例子](https://www.cnblogs.com/chanshuyi/p/head_first_of_reflection.html#%E4%B8%80%E4%B8%AA%E7%AE%80%E5%8D%95%E7%9A%84%E4%BE%8B%E5%AD%90)
* [反射常用API](https://www.cnblogs.com/chanshuyi/p/head_first_of_reflection.html#%E5%8F%8D%E5%B0%84%E5%B8%B8%E7%94%A8api)
  + [获取反射中的Class对象](https://www.cnblogs.com/chanshuyi/p/head_first_of_reflection.html#%E8%8E%B7%E5%8F%96%E5%8F%8D%E5%B0%84%E4%B8%AD%E7%9A%84class%E5%AF%B9%E8%B1%A1)
  + [通过反射创建类对象](https://www.cnblogs.com/chanshuyi/p/head_first_of_reflection.html#%E9%80%9A%E8%BF%87%E5%8F%8D%E5%B0%84%E5%88%9B%E5%BB%BA%E7%B1%BB%E5%AF%B9%E8%B1%A1)
  + [通过反射获取类属性、方法、构造器](https://www.cnblogs.com/chanshuyi/p/head_first_of_reflection.html#%E9%80%9A%E8%BF%87%E5%8F%8D%E5%B0%84%E8%8E%B7%E5%8F%96%E7%B1%BB%E5%B1%9E%E6%80%A7%E6%96%B9%E6%B3%95%E6%9E%84%E9%80%A0%E5%99%A8)
* [反射源码解析](https://www.cnblogs.com/chanshuyi/p/head_first_of_reflection.html#%E5%8F%8D%E5%B0%84%E6%BA%90%E7%A0%81%E8%A7%A3%E6%9E%90)

**文章首发于**[**【博客园-陈树义】**](http://www.cnblogs.com/chanshuyi)**，点击跳转到原文**[**《大白话说Java反射：入门、进阶、原理》**](http://www.cnblogs.com/chanshuyi/p/head_first_of_reflection.html)

反射之中包含了一个「反」字，所以想要解释反射就必须先从「正」开始解释。

一般情况下，我们使用某个类时必定知道它是什么类，是用来做什么的。于是我们直接对这个类进行实例化，之后使用这个类对象进行操作。

Apple apple = new Apple(); //直接初始化，「正射」

apple.setPrice(4);

上面这样子进行类对象的初始化，我们可以理解为「正」。

而反射则是一开始并不知道我要初始化的类对象是什么，自然也无法使用 new 关键字来创建对象了。

这时候，我们使用 JDK 提供的反射 API 进行反射调用：

Class clz = Class.forName("com.chenshuyi.reflect.Apple");

Method method = clz.getMethod("setPrice", int.class);

Constructor constructor = clz.getConstructor();

Object object = constructor.newInstance();

method.invoke(object, 4);

上面两段代码的执行结果，其实是完全一样的。但是其思路完全不一样，第一段代码在未运行时就已经确定了要运行的类（Apple），而第二段代码则是在运行时通过字符串值才得知要运行的类（com.chenshuyi.reflect.Apple）。

所以说什么是反射？

**反射就是在运行时才知道要操作的类是什么，并且可以在运行时获取类的完整构造，并调用对应的方法。**

**一个简单的例子**

上面提到的示例程序，其完整的程序代码如下：

public class Apple {

private int price;

public int getPrice() {

return price;

}

public void setPrice(int price) {

this.price = price;

}

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

//正常的调用

Apple apple = new Apple();

apple.setPrice(5);

System.out.println("Apple Price:" + apple.getPrice());

//使用反射调用

Class clz = Class.forName("com.chenshuyi.api.Apple");

Method setPriceMethod = clz.getMethod("setPrice", int.class);

Constructor appleConstructor = clz.getConstructor();

Object appleObj = appleConstructor.newInstance();

setPriceMethod.invoke(appleObj, 14);

Method getPriceMethod = clz.getMethod("getPrice");

System.out.println("Apple Price:" + getPriceMethod.invoke(appleObj));

}

}

从代码中可以看到我们使用反射调用了 setPrice 方法，并传递了 14 的值。之后使用反射调用了 getPrice 方法，输出其价格。上面的代码整个的输出结果是：

Apple Price:5

Apple Price:14

从这个简单的例子可以看出，一般情况下我们使用反射获取一个对象的步骤：

* 获取类的 Class 对象实例

Class clz = Class.forName("com.zhenai.api.Apple");

* 根据 Class 对象实例获取 Constructor 对象

Constructor appleConstructor = clz.getConstructor();

* 使用 Constructor 对象的 newInstance 方法获取反射类对象

Object appleObj = appleConstructor.newInstance();

而如果要调用某一个方法，则需要经过下面的步骤：

* 获取方法的 Method 对象

Method setPriceMethod = clz.getMethod("setPrice", int.class);

* 利用 invoke 方法调用方法

setPriceMethod.invoke(appleObj, 14);

到这里，我们已经能够掌握反射的基本使用。但如果要进一步掌握反射，还需要对反射的常用 API 有更深入的理解。

在 JDK 中，反射相关的 API 可以分为下面几个方面：获取反射的 Class 对象、通过反射创建类对象、通过反射获取类属性方法及构造器。

**反射常用API**

**获取反射中的Class对象**

在反射中，要获取一个类或调用一个类的方法，我们首先需要获取到该类的 Class 对象。

在 Java API 中，获取 Class 类对象有三种方法：

**第一种，使用 Class.forName 静态方法。**当你知道该类的全路径名时，你可以使用该方法获取 Class 类对象。

Class clz = Class.forName("java.lang.String");

**第二种，使用 .class 方法。**

这种方法只适合在编译前就知道操作的 Class。

Class clz = String.class;

**第三种，使用类对象的 getClass() 方法。**

String str = new String("Hello");

Class clz = str.getClass();

**通过反射创建类对象**

通过反射创建类对象主要有两种方式：通过 Class 对象的 newInstance() 方法、通过 Constructor 对象的 newInstance() 方法。

第一种：通过 Class 对象的 newInstance() 方法。

Class clz = Apple.class;

Apple apple = (Apple)clz.newInstance();

第二种：通过 Constructor 对象的 newInstance() 方法

Class clz = Apple.class;

Constructor constructor = clz.getConstructor();

Apple apple = (Apple)constructor.newInstance();

通过 Constructor 对象创建类对象可以选择特定构造方法，而通过 Class 对象则只能使用默认的无参数构造方法。下面的代码就调用了一个有参数的构造方法进行了类对象的初始化。

Class clz = Apple.class;

Constructor constructor = clz.getConstructor(String.class, int.class);

Apple apple = (Apple)constructor.newInstance("红富士", 15);

**通过反射获取类属性、方法、构造器**

我们通过 Class 对象的 getFields() 方法可以获取 Class 类的属性，但无法获取私有属性。

Class clz = Apple.class;

Field[] fields = clz.getFields();

for (Field field : fields) {

System.out.println(field.getName());

}

输出结果是：

price

而如果使用 Class 对象的 getDeclaredFields() 方法则可以获取包括私有属性在内的所有属性：

Class clz = Apple.class;

Field[] fields = clz.getDeclaredFields();

for (Field field : fields) {

System.out.println(field.getName());

}

输出结果是：

name

price

与获取类属性一样，当我们去获取类方法、类构造器时，如果要获取私有方法或私有构造器，则必须使用有 declared 关键字的方法。

**反射源码解析**

当我们懂得了如何使用反射后，今天我们就来看看 JDK 源码中是如何实现反射的。或许大家平时没有使用过反射，但是在开发 Web 项目的时候会遇到过下面的异常：

java.lang.NullPointerException

...

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

at java.lang.reflect.Method.invoke(Method.java:497)

可以看到异常堆栈指出了异常在 Method 的第 497 的 invoke 方法中，其实这里指的 invoke 方法就是我们反射调用方法中的 invoke。

Method method = clz.getMethod("setPrice", int.class);

method.invoke(object, 4); //就是这里的invoke方法

例如我们经常使用的 Spring 配置中，经常会有相关 Bean 的配置：

<bean class="com.chenshuyi.Apple">

</bean>

当我们在 XML 文件中配置了上面这段配置之后，Spring 便会在启动的时候利用反射去加载对应的 Apple 类。而当 Apple 类不存在或发生启发异常时，异常堆栈便会将异常指向调用的 invoke 方法。

从这里可以看出，我们平常很多框架都使用了反射，而反射中最最终的就是 Method 类的 invoke 方法了。

下面我们来看看 JDK 的 invoke 方法到底做了些什么。

进入 Method 的 invoke 方法我们可以看到，一开始是进行了一些权限的检查，最后是调用了 MethodAccessor 类的 invoke 方法进行进一步处理，如下图红色方框所示。

那么 MethodAccessor 又是什么呢？

其实 MethodAccessor 是一个接口，定义了方法调用的具体操作，而它有三个具体的实现类：

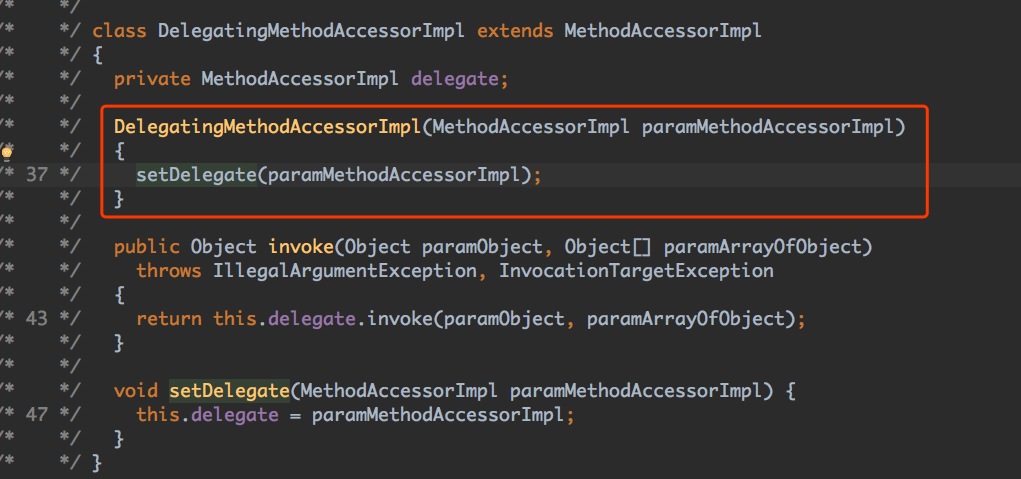
* sun.reflect.DelegatingMethodAccessorImpl
* sun.reflect.MethodAccessorImpl
* sun.reflect.NativeMethodAccessorImpl

而要看 ma.invoke() 到底调用的是哪个类的 invoke 方法，则需要看看 MethodAccessor 对象返回的到底是哪个类对象，所以我们需要进入 acquireMethodAccessor() 方法中看看。

从 acquireMethodAccessor() 方法我们可以看到，代码先判断是否存在对应的 MethodAccessor 对象，如果存在那么就复用之前的 MethodAccessor 对象，否则调用 ReflectionFactory 对象的 newMethodAccessor 方法生成一个 MethodAccessor 对象。

在 ReflectionFactory 类的 newMethodAccessor 方法里，我们可以看到首先是生成了一个 NativeMethodAccessorImpl 对象，再这个对象作为参数调用 DelegatingMethodAccessorImpl 类的构造方法。

这里的实现是使用了代理模式，将 NativeMethodAccessorImpl 对象交给 DelegatingMethodAccessorImpl 对象代理。我们查看 DelegatingMethodAccessorImpl 类的构造方法可以知道，其实是将 NativeMethodAccessorImpl 对象赋值给 DelegatingMethodAccessorImpl 类的 delegate 属性。



所以说ReflectionFactory 类的 newMethodAccessor 方法最终返回 DelegatingMethodAccessorImpl 类对象。所以我们在前面的 ma.invoke() 里，其将会进入 DelegatingMethodAccessorImpl 类的 invoke 方法中。

进入 DelegatingMethodAccessorImpl 类的 invoke 方法后，这里调用了 delegate 属性的 invoke 方法，它又有两个实现类，分别是：DelegatingMethodAccessorImpl 和 NativeMethodAccessorImpl。按照我们前面说到的，这里的 delegate 其实是一个 NativeMethodAccessorImpl 对象，所以这里会进入 NativeMethodAccessorImpl 的 invoke 方法。

而在 NativeMethodAccessorImpl 的 invoke 方法里，其会判断调用次数是否超过阀值（numInvocations）。如果超过该阀值，那么就会生成另一个MethodAccessor 对象，并将原来 DelegatingMethodAccessorImpl 对象中的 delegate 属性指向最新的 MethodAccessor 对象。

到这里，其实我们可以知道 MethodAccessor 对象其实就是具体去生成反射类的入口。通过查看源码上的注释，我们可以了解到 MethodAccessor 对象的一些设计信息。

"Inflation" mechanism. Loading bytecodes to implement Method.invoke() and Constructor.newInstance() currently costs 3-4x more than an invocation via native code for the first invocation (though subsequent invocations have been benchmarked to be over 20x faster).Unfortunately this cost increases startup time for certain applications that use reflection intensively (but only once per class) to bootstrap themselves.

Inflation 机制。初次加载字节码实现反射，使用 Method.invoke() 和 Constructor.newInstance() 加载花费的时间是使用原生代码加载花费时间的 3 - 4 倍。这使得那些频繁使用反射的应用需要花费更长的启动时间。

To avoid this penalty we reuse the existing JVM entry points for the first few invocations of Methods and Constructors and then switch to the bytecode-based implementations. Package-private to be accessible to NativeMethodAccessorImpl and NativeConstructorAccessorImpl.

为了避免这种痛苦的加载时间，我们在第一次加载的时候重用了 JVM 的入口，之后切换到字节码实现的实现。

就像注释里说的，实际的 MethodAccessor 实现有两个版本，一个是 Native 版本，一个是 Java 版本。

Native 版本一开始启动快，但是随着运行时间边长，速度变慢。Java 版本一开始加载慢，但是随着运行时间边长，速度变快。正是因为两种存在这些问题，所以第一次加载的时候我们会发现使用的是 NativeMethodAccessorImpl 的实现，而当反射调用次数超过 15 次之后，则使用 MethodAccessorGenerator 生成的 MethodAccessorImpl 对象去实现反射。

Method 类的 invoke 方法整个流程可以表示成如下的时序图：

讲到这里，我们了解了 Method 类的 invoke 方法的具体实现方式。知道了原来 invoke 方法内部有两种实现方式，一种是 native 原生的实现方式，一种是 Java 实现方式，这两种各有千秋。而为了最大化性能优势，JDK 源码使用了代理的设计模式去实现最大化性能。