#### 反射机制定义

在运行状态中，对于任意的一个类，都能够知道这个类的所有属性和方法，对任意一个对象都能够通过反射机制调用一个类的任意方法，这种动态获取信息及动态调用类对象方法的功能称为Java的反射机制。

#### 反射的作用

1. 动态地创建类的实例，将类绑定到现有的对象中，或从现在的对象中获取类型；
2. 应用程序需要在运行时从某个特定的程序集中载入一个特定的类。

#### Java反射机制原理

[java反射机制原理](https://blog.csdn.net/p785810989/article/details/6495956)

反射一般使用Class.forName()方法；

动态代理就是实现InvocationHandler接口；

要想理解反射的原理，首先要了解什么是类型信息。Java让我们在运行时识别对象和类的信息，主要有2种方式：一种是传统的RTTI（运行时类型识别），它假定我们在编译时已经知道了所有的类型信息；另一种是反射机制，它允许我们在运行时发现和使用类的信息。

#### 1、Class对象

理解RTTI（运行时类型识别）在Java中的工作原理，首先需要知道类型信息在运行时是如何表示的，这是由Class对象来完成的，它包含了与类有关的信息。Class对象就是用来创建所有“常规”对象的，Java使用Class对象来执行RTTI，即使你正在执行的是类似类型转换这样的操作。

每个类都会产生一个对应的Class对象，也就是保存在.class文件。所有类都是在对其第一次使用时，动态加载到JVM的，当程序创建一个对类的静态成员的引用时，就会加载这个类。Class对象仅在需要的时候才会加载，static初始化是在类加载时进行的。

public class Test {

public static void main(String[] args) throws ClassNotFoundException {  
 System.*out*.println(XYZ.*name*);  
 }  
}  
  
class XYZ {  
 public static String *name* = "wcy";  
  
 static {  
 System.*out*.println("xyz静态块");  
 }  
  
 public XYZ() {  
 System.*out*.println("xyz构造了");  
 }  
}

类加载器首先会检查这个类的Class对象是否已被加载过，如果尚未加载，默认的类加载器就会根据类名查找对应的.class文件。想在运行时使用类型信息，必须获取对象（比如类Base对象）的Class对象的引用，使用Class.forName(“Base”)，或者使用Base.class。注意，使用“.class”来创建Class对象的引用时，不会自动初始化该Class对象，使用forName()会自动初始化该Class对象。为了使用类而做的准备工作一般有以下3个步骤：

* 加载：由类加载器完成，找到对应的字节码，创建一个Class对象；
* 链接：验证类中的字节码，为静态域分配空间；
* 初始化：如果该类有超类，则对其初始化，执行静态初始化器和静态初始化块。

public class Base {

static int *num* = 1;  
  
 static {  
 System.*out*.println("Base " + *num*);  
 }  
}  
  
class Base2 {  
 public static void main(String[] args) throws ClassNotFoundException {  
 //不会初始化静态块  
 Class clazz1 = Base.class;  
 System.*out*.println("-----");  
 //会初始化静态块  
 Class clazz2 = Class.*forName*("com.chance.basis.reflection.Base");  
 }  
}

#### 2、类型转换前先做检查

编译器将检查类型向下转型是否合法，如果不合法将抛出异常。向下转型前，可以使用instanceof判断。

instanceof应用场合：当你拿到一个对象的引用时（例如参数），你可能需要判断这个引用真正指向的类。所以你需要从该类继承树的最底层开始，使用instanceof操作符判断，第一个结果为true的类即为引用真正指向的类。

#### 3、运行时类信息

如果不知道某个对象的确切类型，RTTI（Run-Time Type Identification，运行时类型识别）可以告诉你，但是有一个前提：这个类型在编译时必须已知，这样才能使用RTTI来识别它。Class类与java.lang.reflect类库一起对反射进行了支持，该类库包含Field、Method和Constructor类，这些类的对象由JVM在启动时创建，用以表示未知类里对应的成员。这样的话就可以使用Constructor创建新的对象，用get()和get()方法获取和修改类中与Field对象关联的字段，用invoke()方法调用与Method对象关联的方法。另外，还可以调用getFields()、getMethods()和getConstructors()等许多便利的方法，以返回表示字段、方法、以及构造器对象的数组，这样，对象信息可以在运行时被完全确定下来，而在编译时不需要知道关于类的任何事情。

反射机制并没有什么神奇之处，当通过反射与一个未知类型的对象打交道时，JVM只是简单地检查这个对象，看它属于哪个特定的类。因此，那个类的.class对于JVM来说必须是可获取的，要么在本地机器上，要么从网络获取。要么在本地机器上，要么从网络获取。所以对于RTTI和反射之间的真正区别只在于：

* RTTI，编译器在编译时打开和检查.class文件。
* 反射，运行时打开和检查.class文件。

Person person = new Person("luoxn28", 23);

Class clazz = person.getClass();  
  
Field[] fields = clazz.getDeclaredFields();  
for (Field field : fields) {  
 String key = field.getName();  
 PropertyDescriptor descriptor = new PropertyDescriptor(key, clazz);  
 Method method = descriptor.getReadMethod();  
 Object value = method.invoke(person);  
  
 System.*out*.println(key + ":" + value);  
}

以上通过getReadMethod()方法调用类的get函数，可以通过getWriteMethod()方法来调用类的set方法。通常来说，我们不需要使用反射工具，但是它们在创建动态代码会更有用，反射在Java中用来支持其他特性的，例如对象的序列化JavaBean等。

#### 4、动态代理

代理模式是为了提供额外或不同的操作，而插入的用来替代“实际”对象的对象，这些操作涉及到与“实际”对象的通信，因此代理通常充当中间人角色。Java的动态代理比代理的思想更前进了一步，它可以动态地创建并代理并动态地处理对所代理方法的调用。在动态代理上所做的所有调用都会被重定向到单一的调用处理器上，它的工作是揭示调用的类型并确定相应的策略。

Spring主要有两大思想：IoC和AOP，对于IoC，它利用的是反射机制，依赖注入就不用多说了，而对于Spring的核心AOP来说，使用了动态代理，其实底层也是反射。我们不但要知道怎么通过AOP来满足我们的功能，更需要学习的是其底层是怎么样的一个原理，而AOP的原理就是Java的动态代理机制。

在Java的动态代理机制中，有两个重要的类或接口，一个是InvocationHandler(接口)、另一个则是Proxy(类)，这一个类和接口是实现我们动态代理所必需用到的。首先我们先来看看Java的API帮助文档对这两个类进行描述的：

InvocationHandler接口:

每一个动态代理类都必须要实现InvocationHandler这个接口，并且每个代理类的实例都关联到了一个handler，当我们通过代理对象调用一个方法的时候，这个方法的调用就会被转发为由InvocationHandler这个接口的invoke方法来进行调用。该接口的唯一方法invoke：

public Object invoke(Object proxy, Method method, Object[] args)

throws Throwable;

proxy：指代我们所代理的那个真实对象；

method：指代的是我们所要调用真实对象的某个方法的Method对象；

args：指代的是调用真实对象某个方法时接收的参数。

Proxy类：

用来动态创建一个代理对象的类，它提供了许多的方法，但是我们用的最多的就是newProxyInstance方法：

public static Object newProxyInstance(ClassLoader loader,

Class<?>[] interfaces,

InvocationHandler h)

throws IllegalArgumentException

{

Objects.requireNonNull(h);

final Class<?>[] intfs = interfaces.clone();

final SecurityManager sm = System.getSecurityManager();

if (sm != null) {

checkProxyAccess(Reflection.getCallerClass(), loader, intfs);

}

loader：一个ClassLoader对象，定义了由哪个ClassLoader对象来对生成的代理对象进行加载；

interfaces：一个Interface对象的数组，表示的是我将要给我需要代理的对象提供一组什么借口，如果我提供了一组借口给它，那么这个代理对象就宣城实现了该接口（多态），这样我就能调用这组接口中的方法了；

h：一个InvocationHandler对象，表示的是当我这个动态代理对象在调用方法的时候，会关联到哪一个InvocationHandler对象上。

首先定义了一个Subject类型的接口，为其声明了两个方法：

public interface Subject {

void rent();  
void hello(String str);  
}

接着，定义一个类来实现这个接口，这个类就是我们的真实对象，RealSubject类：

public class DynamicProxy implements InvocationHandler {

*/\*\*  
 \* 这个就是我们要代理的真实对象  
 \*/* private Object subject;  
  
 */\*\*  
 \* 构造方法，给我们要代理的真实对象赋初值  
 \*/* public DynamicProxy(Object subject) {  
 this.subject = subject;  
 }  
  
  
 @Override  
 public Object invoke(Object proxy, Method method, Object[] args)

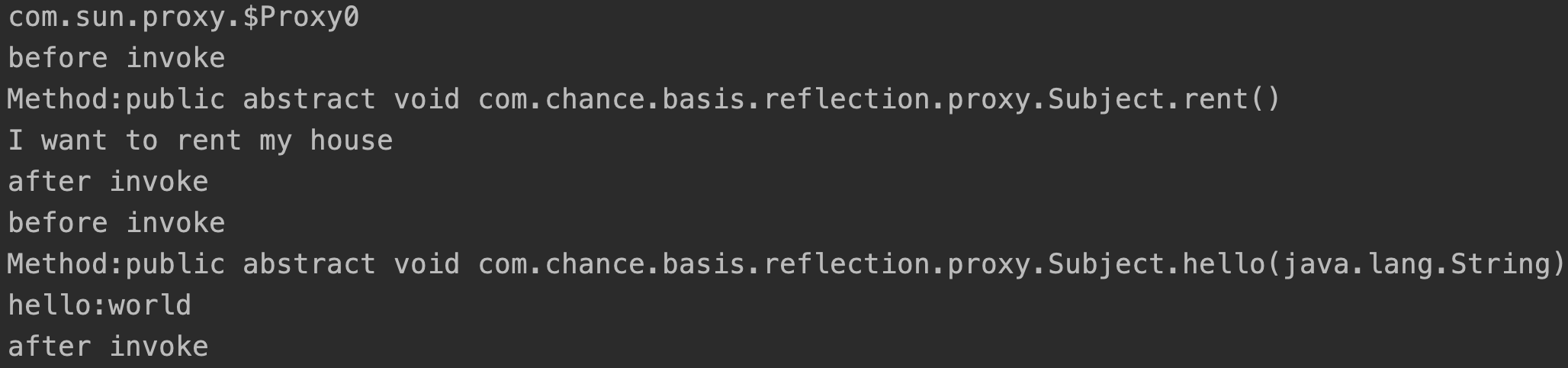
throws Throwable {  
 //1、在代理真实对象前我们可以添加一些自己的操作  
 System.*out*.println("before invoke");  
 System.*out*.println("Method:" + method);  
 //2、当代理对象调用真实对象的方法时，其会自动的跳转到代理对象关联的handler对象的invoke方法来进行调用  
 method.invoke(subject, args);  
 //3、在代理真实对象后我们也可以添加一些自己的操作  
 System.*out*.println("after invoke");  
 return null;  
 }  
}

Client类：

public class Client {

public static void main(String[] args) {  
 //要代理的真实对象  
 RealSubject realSubject = new RealSubject();  
 //要代理的哪个真实对象，就将该对象传进去，最后是通过该真实对象来调用其方法的  
 DynamicProxy handler = new DynamicProxy(realSubject);  
  
 //通过Proxy的newProxyInstance方法来创建我们的代理对象  
 Subject subject = (Subject) Proxy.*newProxyInstance*(handler.getClass().getClassLoader(), realSubject.getClass().getInterfaces(), handler);  
 System.*out*.println(subject.getClass().getName());  
 subject.rent();  
 subject.hello("world");  
 }  
}

运行结果：



首先看$Proxy0，这个东西是由System.out.println(subject.getClass().getName());这条语句打印出来的，那么为什么我们返回的这个代理对象的类名是这样的呢？

Subject subject = (Subject)

Proxy.newProxyInstance(handler.getClass().getClassLoader(),   
 realSubject.getClass().getInterfaces(), handler);

解释下为什么我们这里可以将其转化为Subject类型的对象？原因就在newProxyInstance这个方法的第二个参数上，我们给这个代理对象提供了一组什么接口，那么这个代理对象就会实现了这组接口，这个时候我们当然可以将这个代理对象强制类型转化为这组接口中的任意一个，因为这里的接口是Subject类型，所以就可以将其转化为Subject类型了。

同时一定要记住，通过Proxy.newProxyInstance创建的代理对象是在JVM运行时动态生成的一个对象，它并不是我们的InvocationHandler类型，也不是我们定义的那组接口的类型，而是在运行时动态生成的一个对象，命名方式（以$开头，proxy为中，最后一个数字表示对象的标号）。

接下来看看这两句：

subject.rent();

subject.hello("world");

这里是通过代理对象来调用实现那种接口中的方法，这个时候程序就会跳转到由这个代理对象关联到的handler中的invoke方法去执行，而我们的这个handler对象又接受了一个RealSubject类型的参数，表示我要代理的就是这个真是对象，所以此时就会调用handler中的invoke方法去执行：

@Override

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

System.out.println("before invoke");

System.out.println("Method:" + method);

//当代理对象调用真实对象的方法时，其会自动的跳转到代理对象关联的handler对象的invoke方法来进行调用

method.invoke(subject, args);

System.out.println("after invoke");

return null;

}

我们看到，在真正通过代理对象来调用真实对象的方法的时候，我们可以在该方法前后添加自己的一些操作，同时我们看到我们的这个method对象是这样的：

public abstract void com.xiaoluo.dynamicproxy.Subject.rent()

public abstract void com.xiaoluo.dynamicproxy.Subject.hello(java.lang.String)

正好就是我们的Subject接口中的两个方法，这也就证明了当我通过代理对象来调用方法的时候，其实际就是委托由其关联到的handler对象的invoke方法中来调用，并不是自己来真实调用，而是通过代理的方式来调用的。这就是我们的Java动态代理机制。