Java 基础 - 反射机制详解

> JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。Java反射机制在框架设计中极为广泛，需要深入理解。

反射基础

RRIT（Run-Time Type Identification）运行时类型识别。在《Thinking in Java》一书第十四章中有提到，其作用是在运行时识别一个对象的类型和类的信息。主要有两种方式：一种是“传统的”RTTI，它假定我们在编译时已经知道了所有的类型；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。

反射就是把java类中的各种成分映射成一个个的Java对象

例如：一个类有：成员变量、方法、构造方法、包等等信息，利用反射技术可以对一个类进行解剖，把个个组成部分映射成一个个对象。

> 这里我们首先需要理解 Class类，以及类的加载机制； 然后基于此我们如何通过反射获取Class类以及类中的成员变量、方法、构造方法等。

*Class*类

Class类，Class类也是一个实实在在的类，存在于JDK的java.lang包中。Class类的实例表示java应用运行时的类(class ans enum)或接口(interface and annotation)（每个java类运行时都在JVM里表现为一个class对象，可通过类名.class、类型.getClass()、Class.forName("类名")等方法获取class对象）。数组同样也被映射为为class 对象的一个类，所有具有相同元素类型和维数的数组都共享该 Class 对象。基本类型boolean，byte，char，short，int，long，float，double和关键字void同样表现为 class 对象。

```
public final class Class<T> implements java.io.Serializable,
                              GenericDeclaration,
                              Type,
                              AnnotatedElement {
    private static final int ANNOTATION= 0x00002000;
    private static final int ENUM      = 0x00004000;
    private static final int SYNTHETIC = 0x00001000;

    private static native void registerNatives();
    static {
        registerNatives();
    }

    /*
     * Private constructor. Only the Java Virtual Machine creates Class objects.   //私有构造器，只有JVM才能调用创建Class对象
     * This constructor is not used and prevents the default constructor being
```

```
     * generated.
     */
    private Class(ClassLoader loader) {
        // Initialize final field for classLoader.  The initialization value of non-null
        // prevents future JIT optimizations from assuming this final field is null.
        classLoader = loader;
    }
```

到这我们也就可以得出以下几点信息：

- Class类也是类的一种，与class关键字是不一样的。
- 手动编写的类被编译后会产生一个Class对象，其表示的是创建的类的类型信息，而且这个Class对象保存在同名.class的文件中(字节码文件)
- 每个通过关键字class标识的类，在内存中有且只有一个与之对应的Class对象来描述其类型信息，无论创建多少个实例对象，其依据的都是用一个Class对象。
- Class类只存私有构造函数，因此对应Class对象只能有JVM创建和加载
- Class类的对象作用是运行时提供或获得某个对象的类型信息，这点对于反射技术很重要(关于反射稍后分析)。

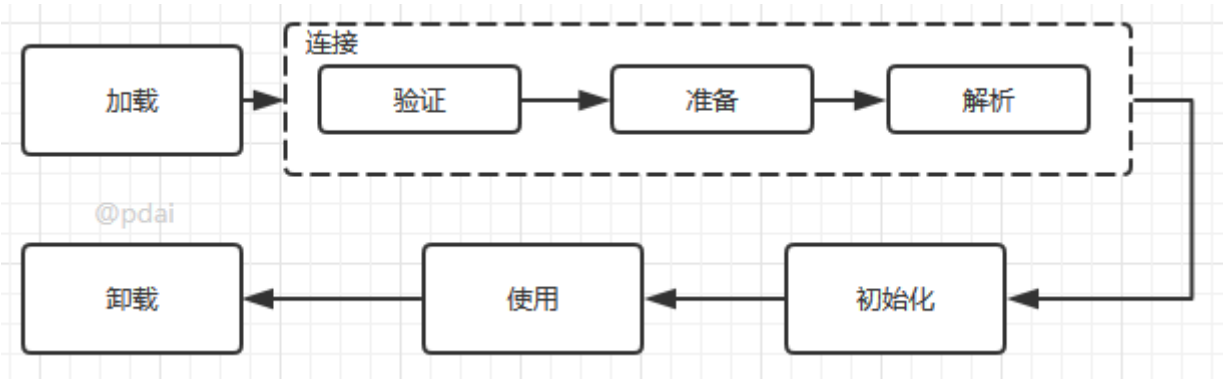
类加载

类加载机制和类字节码技术可以参考如下两篇文章：

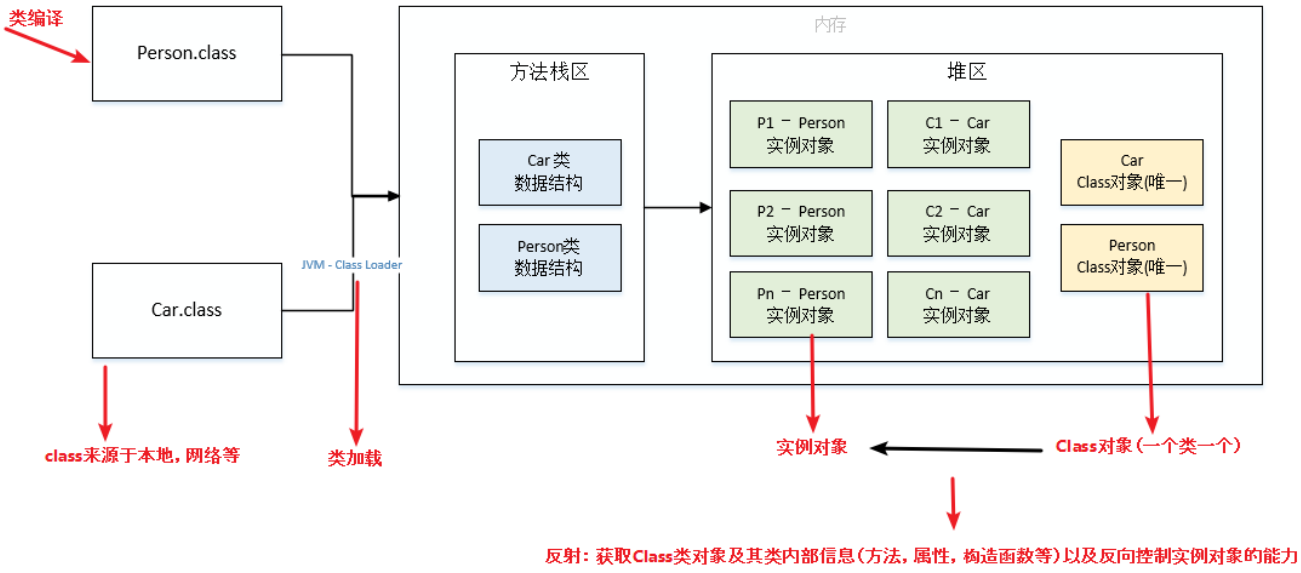
- JVM基础 - 类字节码详解
  - 源代码通过编译器编译为字节码，再通过类加载子系统进行加载到JVM中运行
- JVM基础 - Java 类加载机制
  - 这篇文章将带你深入理解Java 类加载机制

其中，这里我们需要回顾的是：

1. 类加载机制流程



2. 类的加载



反射的使用

TIP

基于此我们如何通过反射获取Class类对象以及类中的成员变量、方法、构造方法等

在Java中，Class类与java.lang.reflect类库一起对反射技术进行了全力的支持。在反射包中，我们常用的类主要有Constructor类表示的是Class 对象所表示的类的构造方法，利用它可以在运行时动态创建对象、Field表示Class对象所表示的类的成员变量，通过它可以在运行时动态修改成员变量的属性值(包含private)、Method表示Class对象所表示的类的成员方法，通过它可以动态调用对象的方法(包含private)，下面将对这几个重要类进行分别说明。

*Class*类对象的获取

在类加载的时候，jvm会创建一个class对象

class对象是可以说是反射中最常用的，获取class对象的方式的主要有三种

- 根据类名：类名.class
- 根据对象：对象.getClass()
- 根据全限定类名：Class.forName(全限定类名)

```
@Test
public void classTest() throws Exception {
    // 获取Class对象的三种方式
    logger.info("根据类名:  \t" + User.class);
    logger.info("根据对象:  \t" + new User().getClass());
    logger.info("根据全限定类名:\t" + Class.forName("com.test.User"));
    // 常用的方法
    logger.info("获取全限定类名:\t" + userClass.getName());
    logger.info("获取类名:\t" + userClass.getSimpleName());
    logger.info("实例化:\t" + userClass.newInstance());
}
```

```
// ...
package com.test;

public class User {
    private String name = "init";
    private int age;
    public User() {}
    public User(String name, int age) {
        super();
        this.name = name;
        this.age = age;
    }
    private String getName() {
        return name;
    }
    private void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    @Override
    public String toString() {
        return "User [name=" + name + ", age=" + age + "]";
    }
}
```

输出结果：

```
根据类名:   class com.test.User
根据对象:   class com.test.User
根据全限定类名:    class com.test.User
获取全限定类名:    com.test.User
获取类名:   User
实例化:    User [name=init, age=0]
```

- 再来看看 **Class类的方法**

方法名	说明
forName()	(1)获取Class对象的一个引用，但引用的类还没有加载(该类的第一个对象没有生成)就加载了这个类。
(2)为了产生Class引用，forName()立即就进行了初始化。	
Object-getClass()	获取Class对象的一个引用，返回表示该对象的实际类型的Class引用。
getName()	取全限定的类名(包括包名)，即类的完整名字。
getSimpleName()	获取类名(不包括包名)
getCanonicalName()	获取全限定的类名(包括包名)

方法名	说明
isInterface()	判断Class对象是否是表示一个接口
getInterfaces()	返回Class对象数组，表示Class对象所引用的类所实现的所有接口。
getSupercalss()	返回Class对象，表示Class对象所引用的类所继承的直接基类。应用该方法可在运行时发现一个对象完整的继承结构。
newInstance()	返回一个Oject对象，是实现“虚拟构造器”的一种途径。使用该方法创建的类，必须带有无参的构造器。
getFields()	获得某个类的所有的公共（public）的字段，包括继承自父类的所有公共字段。类似的还有getMethods和getConstructors。
getDeclaredFields	获得某个类的自己声明的字段，即包括public、private和proteced，默认但是不包括父类声明的任何字段。似的还有getDeclaredMethods和getDeclaredConstructors。

简单测试（这里例子源于https://blog.csdn.net/mcryeasy/article/details/52344729）

```
package com.cry;
import java.lang.reflect.Field;
interface I1 {
}
interface I2 {
}
class Cell{
    public int mCellPublic;
}
class Animal extends  Cell{
    private int mAnimalPrivate;
    protected int mAnimalProtected;
    int mAnimalDefault;
    public int mAnimalPublic;
    private static int sAnimalPrivate;
    protected static int sAnimalProtected;
    static int sAnimalDefault;
    public static int sAnimalPublic;
}
class Dog extends Animal implements I1, I2 {
    private int mDogPrivate;
    public int mDogPublic;
    protected int mDogProtected;
    private int mDogDefault;
    private static int sDogPrivate;
    protected static int sDogProtected;
    static int sDogDefault;
    public static int sDogPublic;
}
public class Test {
    public static void main(String[] args) throws IllegalAccessException, InstantiationException
{
        Class<Dog> dog = Dog.class;
        //类名打印
        System.out.println(dog.getName()); //com.cry.Dog
        System.out.println(dog.getSimpleName()); //Dog
        System.out.println(dog.getCanonicalName());//com.cry.Dog
        //接口
```

```
        System.out.println(dog.isInterface()); //false
        for (Class iI : dog.getInterfaces()) {
            System.out.println(iI);
        }
         /*
          interface com.cry.I1
          interface com.cry.I2
         */

        //父类
        System.out.println(dog.getSuperclass());//class com.cry.Animal
        //创建对象
        Dog d = dog.newInstance();
        //字段
        for (Field f : dog.getFields()) {
            System.out.println(f.getName());
        }
        /*
            mDogPublic
            sDogPublic
            mAnimalPublic
            sAnimalPublic
            mCellPublic  //父类的父类的公共字段也打印出来了
         */
        System.out.println("---------");
        for (Field f : dog.getDeclaredFields()) {
            System.out.println(f.getName());
        }
        /** 只有自己类声明的字段
         mDogPrivate
         mDogPublic
         mDogProtected
         mDogDefault
         sDogPrivate
         sDogProtected
         sDogDefault
         sDogPublic
         */
    }
}
```

getName、getCanonicalName与getSimpleName的区别：

- getSimpleName：只获取类名
- getName：类的全限定名，jvm中Class的表示，可以用于动态加载Class对象，例如Class.forName。
- getCanonicalName：返回更容易理解的表示，主要用于输出（toString）或log打印，大多数情况下和getName一样，但是在内部类、数组等类型的表示形式就不同了。

```
package com.cry;
public class Test {
    private  class inner{
    }
    public static void main(String[] args) throws ClassNotFoundException {
        //普通类
        System.out.println(Test.class.getSimpleName()); //Test
        System.out.println(Test.class.getName()); //com.cry.Test
        System.out.println(Test.class.getCanonicalName()); //com.cry.Test
        //内部类
        System.out.println(inner.class.getSimpleName()); //inner
        System.out.println(inner.class.getName()); //com.cry.Test$inner
```

```
        System.out.println(inner.class.getCanonicalName()); //com.cry.Test.inner
        //数组
        System.out.println(args.getClass().getSimpleName()); //String[]
        System.out.println(args.getClass().getName()); //[Ljava.lang.String;
        System.out.println(args.getClass().getCanonicalName()); //java.lang.String[]
        //我们不能用getCanonicalName去加载类对象，必须用getName
        //Class.forName(inner.class.getCanonicalName()); 报错
        Class.forName(inner.class.getName());
    }
}
```

*Constructor*类及其用法

> Constructor类存在于反射包(java.lang.reflect)中，反映的是Class 对象所表示的类的构造方法。

获取Constructor对象是通过Class类中的方法获取的，Class类与Constructor相关的主要方法如下：

方法返回值	方法名称	方法说明
static Class<?>	forName(String className)	返回与带有给定字符串名的类或接口相关联的 Class 对象。
Constructor	getConstructor(Class<?>... parameterTypes)	返回指定参数类型、具有public访问权限的构造函数对象
Constructor<?>[]	getConstructors()	返回所有具有public访问权限的构造函数的Constructor对象数组
Constructor	getDeclaredConstructor(Class<?>... parameterTypes)	返回指定参数类型、所有声明的（包括private）构造函数对象
Constructor<?>[]	getDeclaredConstructor()	返回所有声明的（包括private）构造函数对象
T	newInstance()	调用无参构造器创建此 Class 对象所表示的类的一个新实例。

下面看一个简单例子来了解Constructor对象的使用：

```
public class ConstructionTest implements Serializable {
    public static void main(String[] args) throws Exception {

        Class<?> clazz = null;

        //获取Class对象的引用
        clazz = Class.forName("com.example.javabase.User");

        //第一种方法，实例化默认构造方法，User必须无参构造函数,否则将抛异常
        User user = (User) clazz.newInstance();
        user.setAge(20);
        user.setName("Jack");
        System.out.println(user);

        System.out.println("----------------------------------------");

        //获取带String参数的public构造函数
```

```
        Constructor cs1 =clazz.getConstructor(String.class);
        //创建User
        User user1= (User) cs1.newInstance("hiway");
        user1.setAge(22);
        System.out.println("user1:"+user1.toString());

        System.out.println("----------------------------------------------");

        //取得指定带int和String参数构造函数,该方法是私有构造private
        Constructor cs2=clazz.getDeclaredConstructor(int.class,String.class);
        //由于是private必须设置可访问
        cs2.setAccessible(true);
        //创建user对象
        User user2= (User) cs2.newInstance(25,"hiway2");
        System.out.println("user2:"+user2.toString());

        System.out.println("----------------------------------------------");

        //获取所有构造包含private
        Constructor<?> cons[] = clazz.getDeclaredConstructors();
        // 查看每个构造方法需要的参数
        for (int i = 0; i < cons.length; i++) {
            //获取构造函数参数类型
            Class<?> clazzs[] = cons[i].getParameterTypes();
            System.out.println("构造函数["+i+"]:"+cons[i].toString() );
            System.out.print("参数类型["+i+"]:(");
            for (int j = 0; j < clazzs.length; j++) {
                if (j == clazzs.length - 1)
                    System.out.print(clazzs[j].getName());
                else
                    System.out.print(clazzs[j].getName() + ",");
            }
            System.out.println(")");
        }
    }
}


class User {
    private int age;
    private String name;
    public User() {
        super();
    }
    public User(String name) {
        super();
        this.name = name;
    }

    /**
     * 私有构造
     * @param age
     * @param name
     */
    private User(int age, String name) {
        super();
        this.age = age;
        this.name = name;
    }

    public int getAge() {
```

```
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "User{" +
                "age=" + age +
                ", name='" + name + '\'' +
                '}';
    }
}
```

输出结果

```
/* output
User{age=20, name='Jack'}
-----------------------------------------
user1:User{age=22, name='hiway'}
-----------------------------------------
user2:User{age=25, name='hiway2'}
-----------------------------------------
构造函数[0]:private com.example.javabase.User(int,java.lang.String)
参数类型[0]:(int,java.lang.String)
构造函数[1]:public com.example.javabase.User(java.lang.String)
参数类型[1]:(java.lang.String)
构造函数[2]:public com.example.javabase.User()
参数类型[2]:()
```

关于**Constructor类本身一些常用方法**如下(仅部分，其他可查API)

方法返回值	方法名称	方法说明
Class	getDeclaringClass()	返回 Class 对象，该对象表示声明由此 Constructor 对象表示的构造方法的类,其实就是返回真实类型（不包含参数）
Type[]	getGenericParameterTypes()	按照声明顺序返回一组 Type 对象，返回的就是 Constructor对象构造函数的形参类型。
String	getName()	以字符串形式返回此构造方法的名称。
Class<?>[]	getParameterTypes()	按照声明顺序返回一组 Class 对象，即返回Constructor 对象所表示构造方法的形参类型
T	newInstance(Object... initargs)	使用此 Constructor对象表示的构造函数来创建新实例

方法返回值	方法名称	方法说明
String	toGenericString()	返回描述此 Constructor 的字符串，其中包括类型参数。

代码演示如下：

```
Constructor cs3 = clazz.getDeclaredConstructor(int.class,String.class);
System.out.println("-----getDeclaringClass-----");
Class uclazz=cs3.getDeclaringClass();
//Constructor对象表示的构造方法的类
System.out.println("构造方法的类:"+uclazz.getName());

System.out.println("-----getGenericParameterTypes-----");
//对象表示此 Constructor 对象所表示的方法的形参类型
Type[] tps=cs3.getGenericParameterTypes();
for (Type tp:tps) {
    System.out.println("参数名称tp:"+tp);
}
System.out.println("-----getParameterTypes-----");
//获取构造函数参数类型
Class<?> clazzs[] = cs3.getParameterTypes();
for (Class claz:clazzs) {
    System.out.println("参数名称:"+claz.getName());
}
System.out.println("-----getName-----");
//以字符串形式返回此构造方法的名称
System.out.println("getName:"+cs3.getName());

System.out.println("-----getoGenericString-----");
//返回描述此 Constructor 的字符串，其中包括类型参数。
System.out.println("getoGenericString():"+cs3.toGenericString())
```

输出结果

```
-----getDeclaringClass-----
构造方法的类:com.example.javabase.User
-----getGenericParameterTypes-----
参数名称tp:int
参数名称tp:class java.lang.String
-----getParameterTypes-----
参数名称:int
参数名称:java.lang.String
-----getName-----
getName:com.example.javabase.User
-----getoGenericString-----
getoGenericString():private com.example.javabase.User(int,java.lang.String)
```

*Field*类及其用法

> Field 提供有关类或接口的单个字段的信息，以及对它的动态访问权限。反射的字段可能是一个类（静态）字段或实例字段。

同样的道理，我们可以通过Class类的提供的方法来获取代表字段信息的Field对象，Class类与Field对象相关方法如下：

方法返回值	方法名称	方法说明
Field	getDeclaredField(String name)	获取指定name名称的(包含private修饰的)字段，不包括继承的字段
Field[]	getDeclaredField()	获取Class对象所表示的类或接口的所有(包含private修饰的)字段,不包括继承的字段
Field	getField(String name)	获取指定name名称、具有public修饰的字段，包含继承字段
Field[]	getField()	获取修饰符为public的字段，包含继承字段

下面的代码演示了上述方法的使用过程

```
public class ReflectField {

    public static void main(String[] args) throws ClassNotFoundException, NoSuchFieldException {
        Class<?> clazz = Class.forName("reflect.Student");
        //获取指定字段名称的Field类,注意字段修饰符必须为public而且存在该字段,
        // 否则抛NoSuchFieldException
        Field field = clazz.getField("age");
        System.out.println("field:"+field);

        //获取所有修饰符为public的字段,包含父类字段,注意修饰符为public才会获取
        Field fields[] = clazz.getFields();
        for (Field f:fields) {
            System.out.println("f:"+f.getDeclaringClass());
        }

        System.out.println("================getDeclaredFields====================");
        //获取当前类所字段(包含private字段),注意不包含父类的字段
        Field fields2[] = clazz.getDeclaredFields();
        for (Field f:fields2) {
            System.out.println("f2:"+f.getDeclaringClass());
        }
        //获取指定字段名称的Field类,可以是任意修饰符的自动,注意不包含父类的字段
        Field field2 = clazz.getDeclaredField("desc");
        System.out.println("field2:"+field2);
    }
    /**
      输出结果:
     field:public int reflect.Person.age
     f:public java.lang.String reflect.Student.desc
     f:public int reflect.Person.age
     f:public java.lang.String reflect.Person.name

     ================getDeclaredFields====================
     f2:public java.lang.String reflect.Student.desc
     f2:private int reflect.Student.score
     field2:public java.lang.String reflect.Student.desc
     */
}

class Person{
    public int age;
    public String name;
    //省略set和get方法
}
```

```
class Student extends Person{
    public String desc;
    private int score;
    //省略set和get方法
}
```

上述方法需要注意的是，如果我们不期望获取其父类的字段，则需使用Class类的getDeclaredField/getDeclaredFields方法来获取字段即可，倘若需要连带获取到父类的字段，那么请使用Class类的getField/getFields，但是也只能获取到public修饰的的字段，无法获取父类的私有字段。下面将通过Field类本身的方法对指定类属性赋值，代码演示如下：

```
//获取Class对象引用
Class<?> clazz = Class.forName("reflect.Student");

Student st= (Student) clazz.newInstance();
//获取父类public字段并赋值
Field ageField = clazz.getField("age");
ageField.set(st,18);
Field nameField = clazz.getField("name");
nameField.set(st,"Lily");

//只获取当前类的字段,不获取父类的字段
Field descField = clazz.getDeclaredField("desc");
descField.set(st,"I am student");
Field scoreField = clazz.getDeclaredField("score");
//设置可访问，score是private的
scoreField.setAccessible(true);
scoreField.set(st,88);
System.out.println(st.toString());

//输出结果：Student{age=18, name='Lily ,desc='I am student', score=88}

//获取字段值
System.out.println(scoreField.get(st));
// 88
```

其中的set(Object obj, Object value)方法是Field类本身的方法，用于设置字段的值，而get(Object obj)则是获取字段的值，当然关于Field类还有其他常用的方法如下：

方法返回值	方法名称	方法说明
void	set(Object obj, Object value)	将指定对象变量上此 Field 对象表示的字段设置为指定的新值。
Object	get(Object obj)	返回指定对象上此 Field 表示的字段的值
Class<?>	getType()	返回一个 Class 对象，它标识了此Field 对象所表示字段的声明类型。
boolean	isEnumConstant()	如果此字段表示枚举类型的元素则返回 true；否则返回 false
String	toGenericString()	返回一个描述此 Field（包括其一般类型）的字符串
String	getName()	返回此 Field 对象表示的字段的名称
Class<?>	getDeclaringClass()	返回表示类或接口的 Class 对象，该类或接口声明由此 Field 对象表示的字段

方法返回值	方法名称	方法说明
void	setAccessible(boolean flag)	将此对象的 accessible 标志设置为指示的布尔值,即设置其可访问性

上述方法可能是较为常用的，事实上在设置值的方法上，Field类还提供了专门针对基本数据类型的方法，如`setInt()/getInt()`、`setBoolean()/getBoolean`、`setChar()/getChar()`等等方法，这里就不全部列出了，需要时查API文档即可。需要特别注意的是被final关键字修饰的Field字段是安全的，在运行时可以接收任何修改，但最终其实际值是不会发生改变的。

*Method*类及其用法

> Method 提供关于类或接口上单独某个方法（以及如何访问该方法）的信息，所反映的方法可能是类方法或实例方法（包括抽象方法）。

下面是Class类获取Method对象相关的方法：

方法返回值	方法名称	方法说明
Method	getDeclaredMethod(String name, Class<?>... parameterTypes)	返回一个指定参数的Method对象，该对象反映此 Class 对象所表示的类或接口的指定已声明方法。
Method[]	getDeclaredMethod()	返回 Method 对象的一个数组，这些对象反映此 Class 对象表示的类或接口声明的所有方法，包括公共、保护、默认（包）访问和私有方法，但不包括继承的方法。
Method	getMethod(String name, Class<?>... parameterTypes)	返回一个 Method 对象，它反映此 Class 对象所表示的类或接口的指定公共成员方法。
Method[]	getMethods()	返回一个包含某些 Method 对象的数组，这些对象反映此 Class 对象所表示的类或接口（包括那些由该类或接口声明的以及从超类和超接口继承的那些的类或接口）的公共 member 方法。

同样通过案例演示上述方法：

```
import java.lang.reflect.Method;

public class ReflectMethod  {


    public static void main(String[] args) throws ClassNotFoundException, NoSuchMethodException
{

        Class clazz = Class.forName("reflect.Circle");

        //根据参数获取public的Method,包含继承自父类的方法
        Method method = clazz.getMethod("draw",int.class,String.class);

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

        //获取所有public的方法:
```

```
        Method[] methods =clazz.getMethods();
        for (Method m:methods){
            System.out.println("m::"+m);
        }

        System.out.println("========================================");

        //获取当前类的方法包含private,该方法无法获取继承自父类的method
        Method method1 = clazz.getDeclaredMethod("drawCircle");
        System.out.println("method1::"+method1);
        //获取当前类的所有方法包含private,该方法无法获取继承自父类的method
        Method[] methods1=clazz.getDeclaredMethods();
        for (Method m:methods1){
            System.out.println("m1::"+m);
        }
    }
}

class Shape {
    public void draw(){
        System.out.println("draw");
    }

    public void draw(int count , String name){
        System.out.println("draw "+ name +",count="+count);
    }

}
class Circle extends Shape{

    private void drawCircle(){
        System.out.println("drawCircle");
    }
    public int getAllCount(){
        return 100;
    }
}
```

输出结果:

```
method:public void reflect.Shape.draw(int,java.lang.String)

m::public int reflect.Circle.getAllCount()
m::public void reflect.Shape.draw()
m::public void reflect.Shape.draw(int,java.lang.String)
m::public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException
m::public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException
m::public final void java.lang.Object.wait() throws java.lang.InterruptedException
m::public boolean java.lang.Object.equals(java.lang.Object)
m::public java.lang.String java.lang.Object.toString()
m::public native int java.lang.Object.hashCode()
m::public final native java.lang.Class java.lang.Object.getClass()
m::public final native void java.lang.Object.notify()
m::public final native void java.lang.Object.notifyAll()

========================================
method1::private void reflect.Circle.drawCircle()

m1::public int reflect.Circle.getAllCount()
m1::private void reflect.Circle.drawCircle()
```

在通过getMethods方法获取Method对象时，会把父类的方法也获取到，如上的输出结果，把Object类的方法都打印出来了。而getDeclaredMethod/getDeclaredMethods方法都只能获取当前类的方法。我们在使用时根据情况选择即可。下面将演示通过Method对象调用指定类的方法：

```
Class clazz = Class.forName("reflect.Circle");
//创建对象
Circle circle = (Circle) clazz.newInstance();

//获取指定参数的方法对象Method
Method method = clazz.getMethod("draw",int.class,String.class);

//通过Method对象的invoke(Object obj,Object... args)方法调用
method.invoke(circle,15,"圈圈");

//对私有无参方法的操作
Method method1 = clazz.getDeclaredMethod("drawCircle");
//修改私有方法的访问标识
method1.setAccessible(true);
method1.invoke(circle);

//对有返回值得方法操作
Method method2 =clazz.getDeclaredMethod("getAllCount");
Integer count = (Integer) method2.invoke(circle);
System.out.println("count:"+count);
```

输出结果

```
draw 圈圈,count=15
drawCircle
count:100
```

在上述代码中调用方法，使用了Method类的invoke(Object obj,Object... args)第一个参数代表调用的对象，第二个参数传递的调用方法的参数。这样就完成了类方法的动态调用。

方法返回值	方法名称	方法说明
Object	invoke(Object obj, Object... args)	对带有指定参数的指定对象调用由此 Method 对象表示的底层方法。
Class<?>	getReturnType()	返回一个 Class 对象，该对象描述了此 Method 对象所表示的方法的正式返回类型,即方法的返回类型
Type	getGenericReturnType()	返回表示由此 Method 对象所表示方法的正式返回类型的 Type 对象，也是方法的返回类型。
Class<?>[]	getParameterTypes()	按照声明顺序返回 Class 对象的数组，这些对象描述了此 Method 对象所表示的方法的形参类型。即返回方法的参数类型组成的数组
Type[]	getGenericParameterTypes()	按照声明顺序返回 Type 对象的数组，这些对象描述了此 Method 对象所表示的方法的形参类型的，也是返回方法的参数类型
String	getName()	以 String 形式返回此 Method 对象表示的方法名称，即返回方法的名称
boolean	isVarArgs()	判断方法是否带可变参数，如果将此方法声明为带有可变数量的参数，则返回 true；否则，返回 false。
String	toGenericString()	返回描述此 Method 的字符串，包括类型参数。

getReturnType方法/getGenericReturnType方法都是获取Method对象表示的方法的返回类型，只不过前者返回的Class类型后者返回的Type(前面已分析过)，Type就是一个接口而已，在Java8中新增一个默认的方法实现，返回的就参数类型信息

```
public interface Type {
    //1.8新增
    default String getTypeName() {
        return toString();
    }
}
```

而getParameterTypes/getGenericParameterTypes也是同样的道理，都是获取Method对象所表示的方法的参数类型，其他方法与前面的Field和Constructor是类似的。

反射机制执行的流程

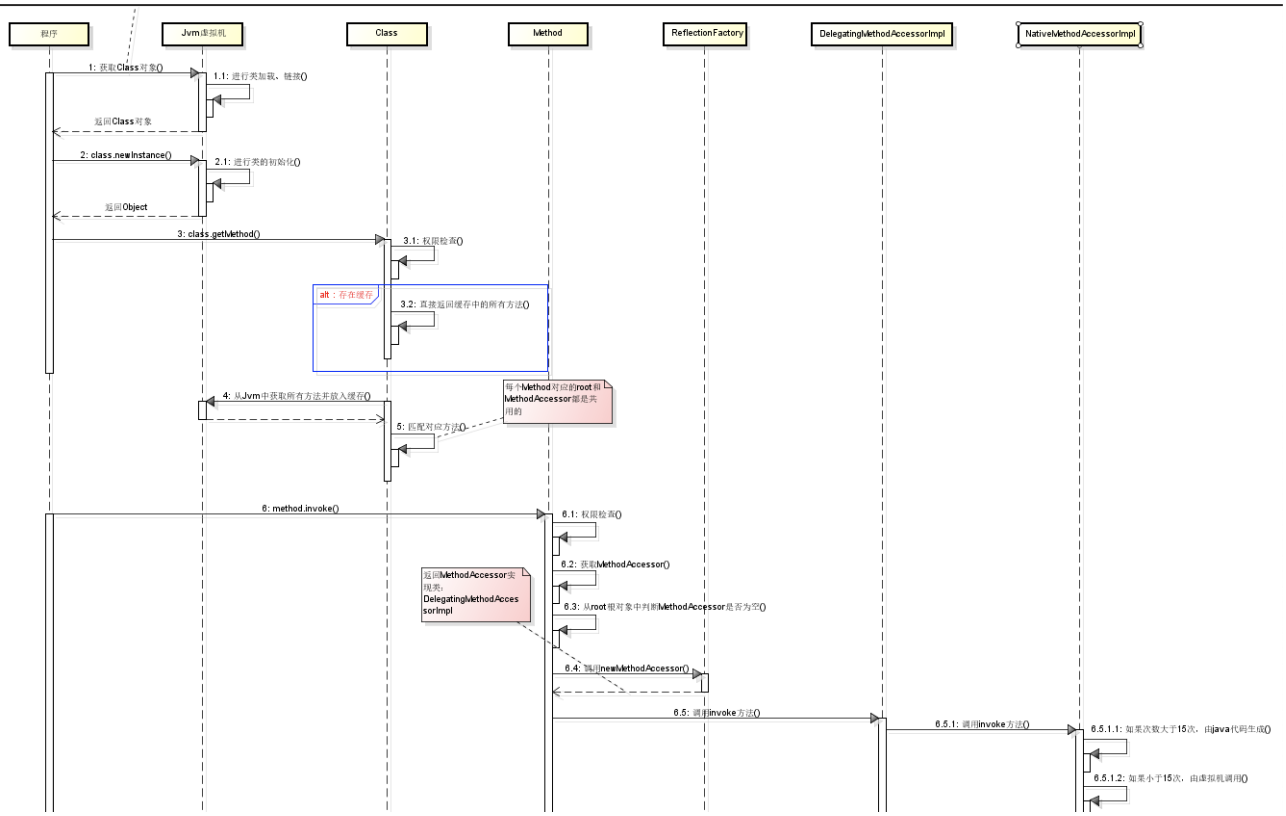
先看个例子

```
public class HelloReflect {
    public static void main(String[] args) {
        try {
            // 1. 使用外部配置的实现，进行动态加载类
            TempFunctionTest test =
(TempFunctionTest)Class.forName("com.tester.HelloReflect").newInstance();
            test.sayHello("call directly");
            // 2. 根据配置的函数名，进行方法调用（不需要通用的接口抽象）
            Object t2 = new TempFunctionTest();
            Method method = t2.getClass().getDeclaredMethod("sayHello", String.class);
            method.invoke(test, "method invoke");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (NoSuchMethodException e ) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
    }

    public void sayHello(String word) {
        System.out.println("hello," + word);
    }

}
```

来看执行流程



反射获取类实例

首先调用了 java.lang.Class 的静态方法，获取类信息。

```
    @CallerSensitive
    public static Class<?> forName(String className)
                throws ClassNotFoundException {
        // 先通过反射，获取调用进来的类信息，从而获取当前的 classLoader
        Class<?> caller = Reflection.getCallerClass();
        // 调用native方法进行获取class信息
        return forName0(className, true, ClassLoader.getClassLoader(caller), caller);
    }
```

forName()反射获取类信息，并没有将实现留给了java,而是交给了jvm去加载。

主要是先获取 ClassLoader, 然后调用 native 方法，获取信息，加载类则是回调 java.lang.ClassLoader.

最后，jvm又会回调 ClassLoader 进类加载。

```
    //
    public Class<?> loadClass(String name) throws ClassNotFoundException {
        return loadClass(name, false);
    }

        // sun.misc.Launcher
        public Class<?> loadClass(String var1, boolean var2) throws ClassNotFoundException {
            int var3 = var1.lastIndexOf(46);
            if(var3 != -1) {
                SecurityManager var4 = System.getSecurityManager();
                if(var4 != null) {
```

```
                var4.checkPackageAccess(var1.substring(0, var3));
            }
        }

        if(this.ucp.knownToNotExist(var1)) {
            Class var5 = this.findLoadedClass(var1);
            if(var5 != null) {
                if(var2) {
                    this.resolveClass(var5);
                }

                return var5;
            } else {
                throw new ClassNotFoundException(var1);
            }
        } else {
            return super.loadClass(var1, var2);
        }
    }
// java.lang.ClassLoader
protected Class<?> loadClass(String name, boolean resolve)
    throws ClassNotFoundException
{
    // 先获取锁
    synchronized (getClassLoadingLock(name)) {
        // First, check if the class has already been loaded
        // 如果已经加载了的话，就不用再加载了
        Class<?> c = findLoadedClass(name);
        if (c == null) {
            long t0 = System.nanoTime();
            try {
                // 双亲委托加载
                if (parent != null) {
                    c = parent.loadClass(name, false);
                } else {
                    c = findBootstrapClassOrNull(name);
                }
            } catch (ClassNotFoundException e) {
                // ClassNotFoundException thrown if class not found
                // from the non-null parent class loader
            }

            // 父类没有加载到时，再自己加载
            if (c == null) {
                // If still not found, then invoke findClass in order
                // to find the class.
                long t1 = System.nanoTime();
                c = findClass(name);

                // this is the defining class loader; record the stats
                sun.misc.PerfCounter.getParentDelegationTime().addTime(t1 - t0);
                sun.misc.PerfCounter.getFindClassTime().addElapsedTimeFrom(t1);
                sun.misc.PerfCounter.getFindClasses().increment();
            }
        }
        if (resolve) {
            resolveClass(c);
        }
        return c;
    }
}
```

```
protected Object getClassLoadingLock(String className) {
    Object lock = this;
    if (parallelLockMap != null) {
        // 使用 ConcurrentHashMap来保存锁
        Object newLock = new Object();
        lock = parallelLockMap.putIfAbsent(className, newLock);
        if (lock == null) {
            lock = newLock;
        }
    }
    return lock;
}

protected final Class<?> findLoadedClass(String name) {
    if (!checkName(name))
        return null;
    return findLoadedClass0(name);
}
```

下面来看一下 newInstance() 的实现方式。

```
// 首先肯定是 Class.newInstance
@CallerSensitive
public T newInstance()
    throws InstantiationException, IllegalAccessException
{
    if (System.getSecurityManager() != null) {
        checkMemberAccess(Member.PUBLIC, Reflection.getCallerClass(), false);
    }

    // NOTE: the following code may not be strictly correct under
    // the current Java memory model.

    // Constructor lookup
    // newInstance() 其实相当于调用类的无参构造函数，所以，首先要找到其无参构造器
    if (cachedConstructor == null) {
        if (this == Class.class) {
            // 不允许调用 Class 的 newInstance() 方法
            throw new IllegalAccessException(
                "Can not call newInstance() on the Class for java.lang.Class"
            );
        }
        try {
            // 获取无参构造器
            Class<?>[] empty = {};
            final Constructor<T> c = getConstructor0(empty, Member.DECLARED);
            // Disable accessibility checks on the constructor
            // since we have to do the security check here anyway
            // (the stack depth is wrong for the Constructor's
            // security check to work)
            java.security.AccessController.doPrivileged(
                new java.security.PrivilegedAction<Void>() {
                    public Void run() {
                            c.setAccessible(true);
                            return null;
                        }
                    });
            cachedConstructor = c;
        } catch (NoSuchMethodException e) {
```

```
                throw (InstantiationException)
                    new InstantiationException(getName()).initCause(e);
            }
        }
        Constructor<T> tmpConstructor = cachedConstructor;
        // Security check (same as in java.lang.reflect.Constructor)
        int modifiers = tmpConstructor.getModifiers();
        if (!Reflection.quickCheckMemberAccess(this, modifiers)) {
            Class<?> caller = Reflection.getCallerClass();
            if (newInstanceCallerCache != caller) {
                Reflection.ensureMemberAccess(caller, this, null, modifiers);
                newInstanceCallerCache = caller;
            }
        }
        // Run constructor
        try {
            // 调用无参构造器
            return tmpConstructor.newInstance((Object[])null);
        } catch (InvocationTargetException e) {
            Unsafe.getUnsafe().throwException(e.getTargetException());
            // Not reached
            return null;
        }
    }
```

newInstance() 主要做了三件事：

- 1. 权限检测，如果不通过直接抛出异常；
- 1. 查找无参构造器，并将其缓存起来；
- 1. 调用具体方法的无参构造方法，生成实例并返回；

下面是获取构造器的过程：

```
    private Constructor<T> getConstructor0(Class<?>[] parameterTypes,
                                           int which) throws NoSuchMethodException
    {
        // 获取所有构造器
        Constructor<T>[] constructors = privateGetDeclaredConstructors((which ==
Member.PUBLIC));
        for (Constructor<T> constructor : constructors) {
            if (arrayContentsEq(parameterTypes,
                                constructor.getParameterTypes())) {
                return getReflectionFactory().copyConstructor(constructor);
            }
        }
        throw new NoSuchMethodException(getName() + ".<init>" +
argumentTypesToString(parameterTypes));
    }
```

getConstructor0() 为获取匹配的构造方器；分三步：

- 1. 先获取所有的constructors, 然后通过进行参数类型比较；
- 1. 找到匹配后，通过 ReflectionFactory copy一份constructor返回；
- 1. 否则抛出 NoSuchMethodException;

```
    // 获取当前类所有的构造方法，通过jvm或者缓存
    // Returns an array of "root" constructors. These Constructor
    // objects must NOT be propagated to the outside world, but must
    // instead be copied via ReflectionFactory.copyConstructor.
```

```
    private Constructor<T>[] privateGetDeclaredConstructors(boolean publicOnly) {
        checkInitted();
        Constructor<T>[] res;
        // 调用 reflectionData(), 获取保存的信息，使用软引用保存，从而使内存不够可以回收
        ReflectionData<T> rd = reflectionData();
        if (rd != null) {
            res = publicOnly ? rd.publicConstructors : rd.declaredConstructors;
            // 存在缓存，则直接返回
            if (res != null) return res;
        }
        // No cached value available; request value from VM
        if (isInterface()) {
            @SuppressWarnings("unchecked")
            Constructor<T>[] temporaryRes = (Constructor<T>[]) new Constructor<?>[0];
            res = temporaryRes;
        } else {
            // 使用native方法从jvm获取构造器
            res = getDeclaredConstructors0(publicOnly);
        }
        if (rd != null) {
            // 最后，将从jvm中读取的内容，存入缓存
            if (publicOnly) {
                rd.publicConstructors = res;
            } else {
                rd.declaredConstructors = res;
            }
        }
        return res;
    }

    // Lazily create and cache ReflectionData
    private ReflectionData<T> reflectionData() {
        SoftReference<ReflectionData<T>> reflectionData = this.reflectionData;
        int classRedefinedCount = this.classRedefinedCount;
        ReflectionData<T> rd;
        if (useCaches &&
            reflectionData != null &&
            (rd = reflectionData.get()) != null &&
            rd.redefinedCount == classRedefinedCount) {
            return rd;
        }
        // else no SoftReference or cleared SoftReference or stale ReflectionData
        // -> create and replace new instance
        return newReflectionData(reflectionData, classRedefinedCount);
    }

    // 新创建缓存，保存反射信息
    private ReflectionData<T> newReflectionData(SoftReference<ReflectionData<T>>
oldReflectionData,
                                                int classRedefinedCount) {
        if (!useCaches) return null;

        // 使用cas保证更新的线程安全性，所以反射是保证线程安全的
        while (true) {
            ReflectionData<T> rd = new ReflectionData<>(classRedefinedCount);
            // try to CAS it...
            if (Atomic.casReflectionData(this, oldReflectionData, new SoftReference<>(rd))) {
                return rd;
            }
            // 先使用CAS更新，如果更新成功，则立即返回，否则测查当前已被其他线程更新的情况，如果和自己想
要更新的状态一致，则也算是成功了
```

```
            oldReflectionData = this.reflectionData;
            classRedefinedCount = this.classRedefinedCount;
            if (oldReflectionData != null &&
                (rd = oldReflectionData.get()) != null &&
                rd.redefinedCount == classRedefinedCount) {
                return rd;
            }
        }
    }
```

如上，privateGetDeclaredConstructors(), 获取所有的构造器主要步骤；

- 1. 先尝试从缓存中获取；
- 1. 如果缓存没有，则从jvm中重新获取，并存入缓存，缓存使用软引用进行保存，保证内存可用；

另外，使用 relactionData() 进行缓存保存；ReflectionData 的数据结构如下。

```
    // reflection data that might get invalidated when JVM TI RedefineClasses() is called
    private static class ReflectionData<T> {
        volatile Field[] declaredFields;
        volatile Field[] publicFields;
        volatile Method[] declaredMethods;
        volatile Method[] publicMethods;
        volatile Constructor<T>[] declaredConstructors;
        volatile Constructor<T>[] publicConstructors;
        // Intermediate results for getFields and getMethods
        volatile Field[] declaredPublicFields;
        volatile Method[] declaredPublicMethods;
        volatile Class<?>[] interfaces;

        // Value of classRedefinedCount when we created this ReflectionData instance
        final int redefinedCount;

        ReflectionData(int redefinedCount) {
            this.redefinedCount = redefinedCount;
        }
    }
```

其中，还有一个点，就是如何比较构造是否是要查找构造器，其实就是比较类型完成相等就完了，有一个不相等则返回false。

```
    private static boolean arrayContentsEq(Object[] a1, Object[] a2) {
        if (a1 == null) {
            return a2 == null || a2.length == 0;
        }

        if (a2 == null) {
            return a1.length == 0;
        }

        if (a1.length != a2.length) {
            return false;
        }

        for (int i = 0; i < a1.length; i++) {
            if (a1[i] != a2[i]) {
                return false;
            }
        }
```

```
        return true;
    }
    // sun.reflect.ReflectionFactory
    /** Makes a copy of the passed constructor. The returned
        constructor is a "child" of the passed one; see the comments
        in Constructor.java for details. */
    public <T> Constructor<T> copyConstructor(Constructor<T> arg) {
        return langReflectAccess().copyConstructor(arg);
    }

    // java.lang.reflect.Constructor, copy 其实就是新new一个 Constructor 出来
    Constructor<T> copy() {
        // This routine enables sharing of ConstructorAccessor objects
        // among Constructor objects which refer to the same underlying
        // method in the VM. (All of this contortion is only necessary
        // because of the "accessibility" bit in AccessibleObject,
        // which implicitly requires that new java.lang.reflect
        // objects be fabricated for each reflective call on Class
        // objects.)
        if (this.root != null)
            throw new IllegalArgumentException("Can not copy a non-root Constructor");

        Constructor<T> res = new Constructor<>(clazz,
                                               parameterTypes,
                                               exceptionTypes, modifiers, slot,
                                               signature,
                                               annotations,
                                               parameterAnnotations);
        // root 指向当前 constructor
        res.root = this;
        // Might as well eagerly propagate this if already present
        res.constructorAccessor = constructorAccessor;
        return res;
    }
```

通过上面，获取到 Constructor 了。

接下来就只需调用其相应构造器的 newInstance()，即返回实例了。

```
    // return tmpConstructor.newInstance((Object[])null);
    // java.lang.reflect.Constructor
    @CallerSensitive
    public T newInstance(Object ... initargs)
        throws InstantiationException, IllegalAccessException,
               IllegalArgumentException, InvocationTargetException
    {
        if (!override) {
            if (!Reflection.quickCheckMemberAccess(clazz, modifiers)) {
                Class<?> caller = Reflection.getCallerClass();
                checkAccess(caller, clazz, null, modifiers);
            }
        }
        if ((clazz.getModifiers() & Modifier.ENUM) != 0)
            throw new IllegalArgumentException("Cannot reflectively create enum objects");
        ConstructorAccessor ca = constructorAccessor;   // read volatile
        if (ca == null) {
            ca = acquireConstructorAccessor();
        }
        @SuppressWarnings("unchecked")
```

```
        T inst = (T) ca.newInstance(initargs);
        return inst;
    }
    // sun.reflect.DelegatingConstructorAccessorImpl
    public Object newInstance(Object[] args)
      throws InstantiationException,
             IllegalArgumentException,
             InvocationTargetException
    {
        return delegate.newInstance(args);
    }
    // sun.reflect.NativeConstructorAccessorImpl
    public Object newInstance(Object[] args)
        throws InstantiationException,
               IllegalArgumentException,
               InvocationTargetException
    {
        // We can't inflate a constructor belonging to a vm-anonymous class
        // 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()
                && !ReflectUtil.isVMAnonymousClass(c.getDeclaringClass())) {
            ConstructorAccessorImpl acc = (ConstructorAccessorImpl)
                new MethodAccessorGenerator().
                    generateConstructor(c.getDeclaringClass(),
                                        c.getParameterTypes(),
                                        c.getExceptionTypes(),
                                        c.getModifiers());
            parent.setDelegate(acc);
        }

        // 调用native方法，进行调用 constructor
        return newInstance0(c, args);
    }
```

返回构造器的实例后，可以根据外部进行进行类型转换，从而使用接口或方法进行调用实例功能了。

反射获取方法

- **第一步，先获取 Method**;

```
    // java.lang.Class
    @CallerSensitive
    public Method getDeclaredMethod(String name, Class<?>... parameterTypes)
        throws NoSuchMethodException, SecurityException {
        checkMemberAccess(Member.DECLARED, Reflection.getCallerClass(), true);
        Method method = searchMethods(privateGetDeclaredMethods(false), name, parameterTypes);
        if (method == null) {
            throw new NoSuchMethodException(getName() + "." + name +
argumentTypesToString(parameterTypes));
        }
        return method;
    }
```

忽略第一个检查权限，剩下就只有两个动作了。

- 1. 获取所有方法列表；
- 1. 根据方法名称和方法列表，选出符合要求的方法；
- 1. 如果没有找到相应方法，抛出异常，否则返回对应方法；

所以，先看一下怎样获取类声明的所有方法？

```
    // Returns an array of "root" methods. These Method objects must NOT
    // be propagated to the outside world, but must instead be copied
    // via ReflectionFactory.copyMethod.
    private Method[] privateGetDeclaredMethods(boolean publicOnly) {
        checkInitted();
        Method[] res;
        ReflectionData<T> rd = reflectionData();
        if (rd != null) {
            res = publicOnly ? rd.declaredPublicMethods : rd.declaredMethods;
            if (res != null) return res;
        }
        // No cached value available; request value from VM
        res = Reflection.filterMethods(this, getDeclaredMethods0(publicOnly));
        if (rd != null) {
            if (publicOnly) {
                rd.declaredPublicMethods = res;
            } else {
                rd.declaredMethods = res;
            }
        }
        return res;
    }
```

很相似，和获取所有构造器的方法很相似，都是先从缓存中获取方法，如果没有，则从jvm中获取。

不同的是，方法列表需要进行过滤 Reflection.filterMethods;当然后面看来，这个方法我们一般不会派上用场。

```
    // sun.misc.Reflection
    public static Method[] filterMethods(Class<?> containingClass, Method[] methods) {
        if (methodFilterMap == null) {
            // Bootstrapping
            return methods;
        }
        return (Method[])filter(methods, methodFilterMap.get(containingClass));
    }
    // 可以过滤指定的方法，一般为空，如果要指定过滤，可以调用 registerMethodsToFilter(), 或者...
    private static Member[] filter(Member[] members, String[] filteredNames) {
        if ((filteredNames == null) || (members.length == 0)) {
            return members;
        }
        int numNewMembers = 0;
        for (Member member : members) {
            boolean shouldSkip = false;
            for (String filteredName : filteredNames) {
                if (member.getName() == filteredName) {
                    shouldSkip = true;
                    break;
                }
            }
            if (!shouldSkip) {
                ++numNewMembers;
            }
        }
        Member[] newMembers =
```

```
            (Member[])Array.newInstance(members[0].getClass(), numNewMembers);
        int destIdx = 0;
        for (Member member : members) {
            boolean shouldSkip = false;
            for (String filteredName : filteredNames) {
                if (member.getName() == filteredName) {
                    shouldSkip = true;
                    break;
                }
            }
            if (!shouldSkip) {
                newMembers[destIdx++] = member;
            }
        }
        return newMembers;
    }
```

- **第二步，根据方法名和参数类型过滤指定方法返回**：

```
    private static Method searchMethods(Method[] methods,
                                        String name,
                                        Class<?>[] parameterTypes)
    {
        Method res = null;
        // 使用常量池，避免重复创建String
        String internedName = name.intern();
        for (int i = 0; i < methods.length; i++) {
            Method m = methods[i];
            if (m.getName() == internedName
                && arrayContentsEq(parameterTypes, m.getParameterTypes())
                && (res == null
                    || res.getReturnType().isAssignableFrom(m.getReturnType())))
                res = m;
        }

        return (res == null ? res : getReflectionFactory().copyMethod(res));
    }
```

大概意思看得明白，就是匹配到方法名，然后参数类型匹配，才可以。

- 但是可以看到，匹配到一个方法，并没有退出for循环，而是继续进行匹配。
- 这里是匹配最精确的子类进行返回（最优匹配）
- 最后，还是通过 ReflectionFactory, copy 方法后返回。

调用 *method.invoke()* 方法

```
    @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);
    }
```

invoke 时，是通过 MethodAccessor 进行调用的，而 MethodAccessor 是个接口，在第一次时调用 acquireMethodAccessor() 进行新创建。

```
    // probably make the implementation more scalable.
    private MethodAccessor acquireMethodAccessor() {
        // First check to see if one has been created yet, and take it
        // if so
        MethodAccessor tmp = null;
        if (root != null) tmp = root.getMethodAccessor();
        if (tmp != null) {
            // 存在缓存时，存入 methodAccessor，否则调用 ReflectionFactory 创建新的 MethodAccessor
            methodAccessor = tmp;
        } else {
            // Otherwise fabricate one and propagate it up to the root
            tmp = reflectionFactory.newMethodAccessor(this);
            setMethodAccessor(tmp);
        }

        return tmp;
    }
    // sun.reflect.ReflectionFactory
    public MethodAccessor newMethodAccessor(Method method) {
        checkInitted();

        if (noInflation && !ReflectUtil.isVMAnonymousClass(method.getDeclaringClass())) {
            return new MethodAccessorGenerator().
                generateMethod(method.getDeclaringClass(),
                               method.getName(),
                               method.getParameterTypes(),
                               method.getReturnType(),
                               method.getExceptionTypes(),
                               method.getModifiers());
        } else {
            NativeMethodAccessorImpl acc =
                new NativeMethodAccessorImpl(method);
            DelegatingMethodAccessorImpl res =
                new DelegatingMethodAccessorImpl(acc);
            acc.setParent(res);
            return res;
        }
    }
}
```

两个Accessor详情：

```
//     NativeMethodAccessorImpl / DelegatingMethodAccessorImpl
class NativeMethodAccessorImpl extends MethodAccessorImpl {
    private final Method method;
    private DelegatingMethodAccessorImpl parent;
    private int numInvocations;
```

```
    NativeMethodAccessorImpl(Method method) {
        this.method = method;
    }

    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()
                && !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);
        }

        return invoke0(method, obj, args);
    }

    void setParent(DelegatingMethodAccessorImpl parent) {
        this.parent = parent;
    }

    private static native Object invoke0(Method m, Object obj, Object[] args);
}
class DelegatingMethodAccessorImpl extends MethodAccessorImpl {
    private MethodAccessorImpl delegate;

    DelegatingMethodAccessorImpl(MethodAccessorImpl delegate) {
        setDelegate(delegate);
    }

    public Object invoke(Object obj, Object[] args)
        throws IllegalArgumentException, InvocationTargetException
    {
        return delegate.invoke(obj, args);
    }

    void setDelegate(MethodAccessorImpl delegate) {
        this.delegate = delegate;
    }
}
```

进行 ma.invoke(obj, args); 调用时，调用 DelegatingMethodAccessorImpl.invoke();

最后被委托到 NativeMethodAccessorImpl.invoke(), 即：

```
    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()
                && !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);
        }

        // invoke0 是个 native 方法，由jvm进行调用业务方法。从而完成反射调用功能。
        return invoke0(method, obj, args);
    }
```

其中， generateMethod() 是生成具体类的方法：

```
    /** This routine is not thread-safe */
    public MethodAccessor generateMethod(Class<?> declaringClass,
                                         String   name,
                                         Class<?>[] parameterTypes,
                                         Class<?>   returnType,
                                         Class<?>[] checkedExceptions,
                                         int modifiers)
    {
        return (MethodAccessor) generate(declaringClass,
                                         name,
                                         parameterTypes,
                                         returnType,
                                         checkedExceptions,
                                         modifiers,
                                         false,
                                         false,
                                         null);
    }
```

generate() 戳详情。

```
    /** This routine is not thread-safe */
    private MagicAccessorImpl generate(final Class<?> declaringClass,
                                       String name,
                                       Class<?>[] parameterTypes,
                                       Class<?>   returnType,
                                       Class<?>[] checkedExceptions,
                                       int modifiers,
                                       boolean isConstructor,
                                       boolean forSerialization,
                                       Class<?> serializationTargetClass)
    {
        ByteVector vec = ByteVectorFactory.create();
        asm = new ClassFileAssembler(vec);
        this.declaringClass = declaringClass;
        this.parameterTypes = parameterTypes;
        this.returnType = returnType;
        this.modifiers = modifiers;
        this.isConstructor = isConstructor;
        this.forSerialization = forSerialization;
```

```
        asm.emitMagicAndVersion();

        // Constant pool entries:
        // ( * = Boxing information: optional)
        // (+  = Shared entries provided by AccessorGenerator)
        // (^  = Only present if generating SerializationConstructorAccessor)
        //     [UTF-8] [This class's name]
        //     [CONSTANT_Class_info] for above
        //     [UTF-8]
"sun/reflect/{MethodAccessorImpl,ConstructorAccessorImpl,SerializationConstructorAccessorImpl}"
        //     [CONSTANT_Class_info] for above
        //     [UTF-8] [Target class's name]
        //     [CONSTANT_Class_info] for above
        // ^   [UTF-8] [Serialization: Class's name in which to invoke constructor]
        // ^   [CONSTANT_Class_info] for above
        //     [UTF-8] target method or constructor name
        //     [UTF-8] target method or constructor signature
        //     [CONSTANT_NameAndType_info] for above
        //     [CONSTANT_Methodref_info or CONSTANT_InterfaceMethodref_info] for target method
        //     [UTF-8] "invoke" or "newInstance"
        //     [UTF-8] invoke or newInstance descriptor
        //     [UTF-8] descriptor for type of non-primitive parameter 1
        //     [CONSTANT_Class_info] for type of non-primitive parameter 1
        //     ...
        //     [UTF-8] descriptor for type of non-primitive parameter n
        //     [CONSTANT_Class_info] for type of non-primitive parameter n
        // +   [UTF-8] "java/lang/Exception"
        // +   [CONSTANT_Class_info] for above
        // +   [UTF-8] "java/lang/ClassCastException"
        // +   [CONSTANT_Class_info] for above
        // +   [UTF-8] "java/lang/NullPointerException"
        // +   [CONSTANT_Class_info] for above
        // +   [UTF-8] "java/lang/IllegalArgumentException"
        // +   [CONSTANT_Class_info] for above
        // +   [UTF-8] "java/lang/InvocationTargetException"
        // +   [CONSTANT_Class_info] for above
        // +   [UTF-8] "<init>"
        // +   [UTF-8] "()V"
        // +   [CONSTANT_NameAndType_info] for above
        // +   [CONSTANT_Methodref_info] for NullPointerException's constructor
        // +   [CONSTANT_Methodref_info] for IllegalArgumentException's constructor
        // +   [UTF-8] "(Ljava/lang/String;)V"
        // +   [CONSTANT_NameAndType_info] for "<init>(Ljava/lang/String;)V"
        // +   [CONSTANT_Methodref_info] for IllegalArgumentException's constructor taking a
String
        // +   [UTF-8] "(Ljava/lang/Throwable;)V"
        // +   [CONSTANT_NameAndType_info] for "<init>(Ljava/lang/Throwable;)V"
        // +   [CONSTANT_Methodref_info] for InvocationTargetException's constructor
        // +   [CONSTANT_Methodref_info] for "super()"
        // +   [UTF-8] "java/lang/Object"
        // +   [CONSTANT_Class_info] for above
        // +   [UTF-8] "toString"
        // +   [UTF-8] "()Ljava/lang/String;"
        // +   [CONSTANT_NameAndType_info] for "toString()Ljava/lang/String;"
        // +   [CONSTANT_Methodref_info] for Object's toString method
        // +   [UTF-8] "Code"
        // +   [UTF-8] "Exceptions"
        //  *  [UTF-8] "java/lang/Boolean"
        //  *  [CONSTANT_Class_info] for above
        //  *  [UTF-8] "(Z)V"
```

```
//  *  [CONSTANT_NameAndType_info] for above
//  *  [CONSTANT_Methodref_info] for above
//  *  [UTF-8] "booleanValue"
//  *  [UTF-8] "()Z"
//  *  [CONSTANT_NameAndType_info] for above
//  *  [CONSTANT_Methodref_info] for above
//  *  [UTF-8] "java/lang/Byte"
//  *  [CONSTANT_Class_info] for above
//  *  [UTF-8] "(B)V"
//  *  [CONSTANT_NameAndType_info] for above
//  *  [CONSTANT_Methodref_info] for above
//  *  [UTF-8] "byteValue"
//  *  [UTF-8] "()B"
//  *  [CONSTANT_NameAndType_info] for above
//  *  [CONSTANT_Methodref_info] for above
//  *  [UTF-8] "java/lang/Character"
//  *  [CONSTANT_Class_info] for above
//  *  [UTF-8] "(C)V"
//  *  [CONSTANT_NameAndType_info] for above
//  *  [CONSTANT_Methodref_info] for above
//  *  [UTF-8] "charValue"
//  *  [UTF-8] "()C"
//  *  [CONSTANT_NameAndType_info] for above
//  *  [CONSTANT_Methodref_info] for above
//  *  [UTF-8] "java/lang/Double"
//  *  [CONSTANT_Class_info] for above
//  *  [UTF-8] "(D)V"
//  *  [CONSTANT_NameAndType_info] for above
//  *  [CONSTANT_Methodref_info] for above
//  *  [UTF-8] "doubleValue"
//  *  [UTF-8] "()D"
//  *  [CONSTANT_NameAndType_info] for above
//  *  [CONSTANT_Methodref_info] for above
//  *  [UTF-8] "java/lang/Float"
//  *  [CONSTANT_Class_info] for above
//  *  [UTF-8] "(F)V"
//  *  [CONSTANT_NameAndType_info] for above
//  *  [CONSTANT_Methodref_info] for above
//  *  [UTF-8] "floatValue"
//  *  [UTF-8] "()F"
//  *  [CONSTANT_NameAndType_info] for above
//  *  [CONSTANT_Methodref_info] for above
//  *  [UTF-8] "java/lang/Integer"
//  *  [CONSTANT_Class_info] for above
//  *  [UTF-8] "(I)V"
//  *  [CONSTANT_NameAndType_info] for above
//  *  [CONSTANT_Methodref_info] for above
//  *  [UTF-8] "intValue"
//  *  [UTF-8] "()I"
//  *  [CONSTANT_NameAndType_info] for above
//  *  [CONSTANT_Methodref_info] for above
//  *  [UTF-8] "java/lang/Long"
//  *  [CONSTANT_Class_info] for above
//  *  [UTF-8] "(J)V"
//  *  [CONSTANT_NameAndType_info] for above
//  *  [CONSTANT_Methodref_info] for above
//  *  [UTF-8] "longValue"
//  *  [UTF-8] "()J"
//  *  [CONSTANT_NameAndType_info] for above
//  *  [CONSTANT_Methodref_info] for above
//  *  [UTF-8] "java/lang/Short"
```

```
//  *  [CONSTANT_Class_info] for above
//  *  [UTF-8] "(S)V"
//  *  [CONSTANT_NameAndType_info] for above
//  *  [CONSTANT_Methodref_info] for above
//  *  [UTF-8] "shortValue"
//  *  [UTF-8] "()S"
//  *  [CONSTANT_NameAndType_info] for above
//  *  [CONSTANT_Methodref_info] for above

short numCPEntries = NUM_BASE_CPOOL_ENTRIES + NUM_COMMON_CPOOL_ENTRIES;
boolean usesPrimitives = usesPrimitiveTypes();
if (usesPrimitives) {
    numCPEntries += NUM_BOXING_CPOOL_ENTRIES;
}
if (forSerialization) {
    numCPEntries += NUM_SERIALIZATION_CPOOL_ENTRIES;
}

// Add in variable-length number of entries to be able to describe
// non-primitive parameter types and checked exceptions.
numCPEntries += (short) (2 * numNonPrimitiveParameterTypes());

asm.emitShort(add(numCPEntries, S1));

final String generatedName = generateName(isConstructor, forSerialization);
asm.emitConstantPoolUTF8(generatedName);
asm.emitConstantPoolClass(asm.cpi());
thisClass = asm.cpi();
if (isConstructor) {
    if (forSerialization) {
        asm.emitConstantPoolUTF8
            ("sun/reflect/SerializationConstructorAccessorImpl");
    } else {
        asm.emitConstantPoolUTF8("sun/reflect/ConstructorAccessorImpl");
    }
} else {
    asm.emitConstantPoolUTF8("sun/reflect/MethodAccessorImpl");
}
asm.emitConstantPoolClass(asm.cpi());
superClass = asm.cpi();
asm.emitConstantPoolUTF8(getClassName(declaringClass, false));
asm.emitConstantPoolClass(asm.cpi());
targetClass = asm.cpi();
short serializationTargetClassIdx = (short) 0;
if (forSerialization) {
    asm.emitConstantPoolUTF8(getClassName(serializationTargetClass, false));
    asm.emitConstantPoolClass(asm.cpi());
    serializationTargetClassIdx = asm.cpi();
}
asm.emitConstantPoolUTF8(name);
asm.emitConstantPoolUTF8(buildInternalSignature());
asm.emitConstantPoolNameAndType(sub(asm.cpi(), S1), asm.cpi());
if (isInterface()) {
    asm.emitConstantPoolInterfaceMethodref(targetClass, asm.cpi());
} else {
    if (forSerialization) {
        asm.emitConstantPoolMethodref(serializationTargetClassIdx, asm.cpi());
    } else {
        asm.emitConstantPoolMethodref(targetClass, asm.cpi());
    }
}
```

```
targetMethodRef = asm.cpi();
if (isConstructor) {
    asm.emitConstantPoolUTF8("newInstance");
} else {
    asm.emitConstantPoolUTF8("invoke");
}
invokeIdx = asm.cpi();
if (isConstructor) {
    asm.emitConstantPoolUTF8("([Ljava/lang/Object;)Ljava/lang/Object;");
} else {
    asm.emitConstantPoolUTF8
        ("(Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object;");
}
invokeDescriptorIdx = asm.cpi();

// Output class information for non-primitive parameter types
nonPrimitiveParametersBaseIdx = add(asm.cpi(), S2);
for (int i = 0; i < parameterTypes.length; i++) {
    Class<?> c = parameterTypes[i];
    if (!isPrimitive(c)) {
        asm.emitConstantPoolUTF8(getClassName(c, false));
        asm.emitConstantPoolClass(asm.cpi());
    }
}

// Entries common to FieldAccessor, MethodAccessor and ConstructorAccessor
emitCommonConstantPoolEntries();

// Boxing entries
if (usesPrimitives) {
    emitBoxingContantPoolEntries();
}

if (asm.cpi() != numCPEntries) {
    throw new InternalError("Adjust this code (cpi = " + asm.cpi() +
                            ", numCPEntries = " + numCPEntries + ")");
}

// Access flags
asm.emitShort(ACC_PUBLIC);

// This class
asm.emitShort(thisClass);

// Superclass
asm.emitShort(superClass);

// Interfaces count and interfaces
asm.emitShort(S0);

// Fields count and fields
asm.emitShort(S0);

// Methods count and methods
asm.emitShort(NUM_METHODS);

emitConstructor();
emitInvoke();

// Additional attributes (none)
asm.emitShort(S0);
```

```
        // Load class
        vec.trim();
        final byte[] bytes = vec.getData();
        // Note: the class loader is the only thing that really matters
        // here -- it's important to get the generated code into the
        // same namespace as the target class. Since the generated code
        // is privileged anyway, the protection domain probably doesn't
        // matter.
        return AccessController.doPrivileged(
            new PrivilegedAction<MagicAccessorImpl>() {
                public MagicAccessorImpl run() {
                        try {
                        return (MagicAccessorImpl)
                        ClassDefiner.defineClass
                                (generatedName,
                                 bytes,
                                 0,
                                 bytes.length,
                                 declaringClass.getClassLoader()).newInstance();
                        } catch (InstantiationException | IllegalAccessException e) {
                            throw new InternalError(e);
                        }
                    }
                });
    }
```

主要看这一句：`ClassDefiner.defineClass(xx, declaringClass.getClassLoader()).newInstance();`

在`ClassDefiner.defineClass`方法实现中，每被调用一次都会生成一个DelegatingClassLoader类加载器对象 ，这里每次都生成新的类加载器，是为了性能考虑，在某些情况下可以卸载这些生成的类，因为类的卸载是只有在类加载器可以被回收的情况下才会被回收的，如果用了原来的类加载器，那可能导致这些新创建的类一直无法被卸载。

而反射生成的类，有时候可能用了就可以卸载了，所以使用其独立的类加载器，从而使得更容易控制反射类的生命周期。

反射调用流程小结

最后，总结反射的实现原理：

1. 反射类及反射方法的获取，都是通过从列表中搜寻查找匹配的方法，所以查找性能会随类的大小方法多少而变化；
2. 每个类都会有一个与之对应的Class实例，从而每个类都可以获取method反射方法，并作用到其他实例身上；
3. 反射也是考虑了线程安全的，放心使用；
4. 反射使用软引用relectionData缓存class信息，避免每次重新从jvm获取带来的开销；
5. 反射调用多次生成新代理Accessor, 而通过字节码生存的则考虑了卸载功能，所以会使用独立的类加载器；
6. 当找到需要的方法，都会copy一份出来，而不是使用原来的实例，从而保证数据隔离；
7. 调度反射方法，最终是由jvm执行invoke0()执行；