链滴

Java 反射技术

作者：jchain
原文链接：https://ld246.com/article/1595581784724
来源网站：链滴

为什么需要反射，反射解决了什么问题，有什么好处

反射英文为 Reflection 其实更为贴切的翻译为 内省（自省），这里可以认为 “我是谁”，表明可以道自己本身的数据，在程序中这些“本省”的数据就是 类， 属性，方法，构造器 等

这样对于静态编译型语言来说 在运行时 就会有一些动态的能力，比如 获取(设置)对象的属性和方法 创建对象等，提升了程序的灵活性

按照一般的解释：Java反射是在运行状态中，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方法和属性，是一种动态获取信息以及动态调用对象的方法功能

反射技术应用场景举例

1. 实现通用框架基础功能

用户可以基于**框架定义的描述手段**进行系统功能定义开发,框架在**运行时** 解析 获取用户 在开发阶段定的描述信息，然后根据这些描述信息 动态地 去在运行时获取所需的 对象，对象的方法/属性等信息 而执行相应的操作，完成系统的功能。这样可以大大的提升程序的开发效率。比如spring框架，框架身定义了一些Bean的描述手段(xml 或者 配置注解),用户只需要按照描述规则编写代码，那么在运行，spring就会根据你的描述信息 去动态的 创建你的bean,并且填充属性，初始化等操作，这样你就可直接拿到这个bean去使用，而不是对于系统中的每个bean，你都自己去手写每一个bean的对象创建属性填充，初始化，注入等操作，如果这样写的话就够你喝一壶的了。其实更简单一点可以理解为 这的框架 就像一个模子（模板），你按照他的规则来，它就能给你想要的东西。底层得话 大致上就是解析描述信息，然后根据反射技术从描述中动态创建对象/填充属性/初始化/注入等等

2. 实现作用于 "任意类型" 的函数（或者json序列化或者属性编辑器）

这里比方说要实现一个 打印的 函数，参数可以是任意类型(Java中Object，go中interface{}),然后需打印出 这个对象的属性以及值。

这样的话就可以 通过反射技术 动态获取传进来对象的属性和值，然后组装成想要的格式打印出来，样不管什么属性和值都能打印出来。比如用Java写一个打印方法：

```
public static void print(Object object) {
    Field[] declaredFields = object.getClass().getDeclaredFields();
    for (Field declaredField : declaredFields) {
        declaredField.setAccessible(true);
        try {
            System.out.println(declaredField.getName()+":"+declaredField.get(object));
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }
}
```

另外 golang的 `Println` 函数底层 部分 就是用了反射技术

反射有什么弊端

在执行时对于CPU和内存资源会进行占用，在数据量很大的时候，需谨慎使用反射

在封装性上有些破坏，不过这也是一种权衡，毕竟解决问题才是目的

Java反射

Java大体上分为编译期和运行期。我们编写好源代码后，会通过 javac 命令编译成 字节码，然后程序行时 会根据 字节码 装载到Java虚拟机，那么在运行期时是怎么表示 我们源代码的呢？其实也就是自的数据，在运行时 有一些内建类来表示我们的程序，这些类如下：

类名	用途
Class	类的表示，可以表示类或者接口
Field	属性(成员变成)的表示
Method	方法的表示
Constructor	构造方法的表示

获取Class类对象的方式

一般可以通过 有下面3中方式

1. Class.forName("全限定类名")

这种方式 我们会在 加载数据库驱动的会用到 比如 Class.forName("com.mysql.jdbc.Driver")

2. 类名.class

Class.class

3. 实例对象.getClass()

obj.getClass()

```
class Clazz {

}

public class ClazzTest {
    public static void main(String[] args) throws Exception {
        Class<Clazz> clazz1 = Clazz.class;
        Class<Clazz> clazz2 = (Class<Clazz>) Class.forName("com.linn.slarn.spring.bean.register.
efrect.Clazz");
        Class<? extends Clazz> clazz3 = new Clazz().getClass();
        // clazz1 == clazz2 == clazz3
        // ClassLoader一样的 所以相等
        // 参考 https://segmentfault.com/a/1190000023108785
    }
}
```

反射API

当拿到Class对象后，就可以通过语言本身提供的API 来进行我们想要的操作了

Class API

获取Class类相关方法

方法	作用
getClassLoader()	获取类加载器
getClasses()	返回包含该类中所有**公共(public)和公共(public)**接口的对象 的数组
getDeclaredClasses()	返回包含该类中所有类和口类的对象的数组
getName()	获得类的完整路径名字
getSimpleName()	获得类（简单）的名字
getPackage()	获得类的包
getSuperclass()	获得当前类继承的父类的名字
getInterfaces()	getInterfaces()
forName(String className)	根据类名返回类的象（这种在类的加载时是有初始化操作的）
newInstance()	创建内的实例

```
package com.linn.slarn.spring.bean.register.refrect;

class SuperClazz {

}

interface Interface1 {

}

interface Interface2 {

}

class Clazz extends SuperClazz implements Interface1, Interface2 {

    public class MemberClazz1 {

    }

    class MemberClazz2 {

    }

    interface MemberInterface1 {

    }

    public interface MemberInterface2 {

    }
}
```

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

        // 返回包含该类中所有 公共(public)类 和 公共接口(public) 的对象 的数组
        Class<?>[] classes = Clazz.class.getClasses();
        for (Class<?> aClass : classes) {
            System.out.println(aClass);
        }
        //interface com.linn.slarn.spring.bean.register.refrect.Clazz$MemberInterface2
        //class com.linn.slarn.spring.bean.register.refrect.Clazz$MemberClazz1

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

        // 返回包含该类中所有类和接口的数组
        Class<?>[] declaredClasses = Clazz.class.getDeclaredClasses();
        for (Class<?> aClass : declaredClasses) {
            System.out.println(aClass);
        }
        //interface com.linn.slarn.spring.bean.register.refrect.Clazz$MemberInterface2
        //interface com.linn.slarn.spring.bean.register.refrect.Clazz$MemberInterface1
        //class com.linn.slarn.spring.bean.register.refrect.Clazz$MemberClazz2
        //class com.linn.slarn.spring.bean.register.refrect.Clazz$MemberClazz1

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

        // 获取 Name（全限定的）： com.linn.slarn.spring.bean.register.refrect.Clazz
        System.out.println(Clazz.class.getName());

        // 获取简单Name： Clazz
        System.out.println(Clazz.class.getSimpleName());

        // 获取包名： package com.linn.slarn.spring.bean.register.refrect
        System.out.println(Clazz.class.getPackage());

        // 获取父类 class com.linn.slarn.spring.bean.register.refrect.SuperClazz 如果没有的就是
级父类Object
        Class<? super Clazz> superclass = Clazz.class.getSuperclass();
        System.out.println(superclass);

        // 获取 ClassLoader: sun.misc.Launcher$AppClassLoader@18b4aac2
        ClassLoader classLoader = Clazz.class.getClassLoader();
        System.out.println(classLoader);

        // 获取类的接口
        // interface com.linn.slarn.spring.bean.register.refrect.Interface1
        // interface com.linn.slarn.spring.bean.register.refrect.Interface2
        Class<?>[] interfaces = Clazz.class.getInterfaces();
        for (Class<?> anInterface : interfaces) {
            System.out.println(anInterface);
        }

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

        // 创建对象 每种方式创建的对象都是不一样的
```

```
        Clazz obj1 = Clazz.class.newInstance();
        System.out.println("T.class.newInstance() 创建的对象" + obj1); //Clazz@5cad8086

        Clazz clazz = new Clazz(); //@6e0be858
        System.out.println("new T() 创建的对象：" + clazz);
        Clazz obj2 = clazz.getClass().newInstance();
        System.out.println("new T().getClass().newInstance创建的对象" + obj2); //@6e0be858

        Clazz obj3 = (Clazz) Class.forName("com.linn.slarn.spring.bean.register.refrect.Clazz").new
nstance();
        System.out.println("Class.forName 创建的对象 obj3:" + obj3); //@610455d6

    }
}
```

获取和注解相关的方法

方法	作用
getAnnotations()	返回该类所有的**公有**注解对象
getAnnotation(Class<A> annotationClass)	获
类上的指定**公有**注解	
getDeclaredAnnotation(Class<A> annotationClass)	
回该类中与参数类型匹配的所有注解对象	
getDeclaredAnnotations()	返回该类所有的注
对象	
getAnnotationsByType(Class<A> annotationClass)	
据类型获取注解	

```
// 两个注解
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Annotation1 {
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Annotation2 {
}



@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@interface Annotation0 {

}

@Annotation0
@Annotation1
@Annotation2
class Clazz {}
```

```java
public class ClazzTest {

    public static void main(String[] args) {
        Class<Clazz> clazzClass = Clazz.class;

        // 需要注意所有的注解 比如是 @Retention(RetentionPolicy.RUNTIME) 运行时的才行

        // 获取类上的指定公有（public）注解
        Annotation1 annotationOne = clazzClass.getAnnotation(Annotation1.class);
        System.out.println(annotationOne);
        // @com.linn.slarn.spring.bean.register.refrect.annotation.Annotation1()

        System.out.println("=====");
        // 返回该类所有的公有(public)注解对象
        Annotation[] annotations = clazzClass.getAnnotations();
        for (Annotation annotation : annotations) {
            System.out.println(annotation);
        }
        //@com.linn.slarn.spring.bean.register.refrect.annotation.Annotation1()
        //@com.linn.slarn.spring.bean.register.refrect.annotation.Annotation2()

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

        // 获取类上 所有的 注解 包括不是public的
        Annotation[] declaredAnnotations = clazzClass.getDeclaredAnnotations();
        for (Annotation declaredAnnotation : declaredAnnotations) {
            System.out.println(declaredAnnotation);
        }
        //@com.linn.slarn.spring.bean.register.refrect.Annotation0()
        //@com.linn.slarn.spring.bean.register.refrect.annotation.Annotation1()
        //@com.linn.slarn.spring.bean.register.refrect.annotation.Annotation2()
    }
}
```

获取和构造器相关的方法

方法	作用
getConstructors()	获得该类的所有公有构造方法
getDeclaredConstructors()	获得该类所有构造法
getConstructor(Class...<?> parameterTypes)	得该类中与参数类型匹配的公有构造方法
getDeclaredConstructor(Class...<?> parameterTypes)	得该类中与参数类型匹配的构造方法

```java
class Clazz {
    public Clazz() {

    }
```

```
    public Clazz(String name) {

    }

    public Clazz(String name,Integer age) {

    }

    Clazz(Long id, String name,Integer age) {

    }

    Clazz(Long id, String name,Integer age,Object object) {

    }

}

public class ClazzTest {
    public static void main(String[] args) throws Exception {
        Class<Clazz> clazzClass = Clazz.class;

        // 获取所有的 public 构造器哦
        Constructor<?>[] constructors = clazzClass.getConstructors();
        for (Constructor<?> constructor : constructors) {
            System.out.println(constructor);
        }
        //public com.linn.slarn.spring.bean.register.refrect.Clazz(java.lang.String,java.lang.Integer)
        //public com.linn.slarn.spring.bean.register.refrect.Clazz(java.lang.String)
        //public com.linn.slarn.spring.bean.register.refrect.Clazz()

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

        // 获取指定 public 构造器
        Constructor<Clazz> constructor = clazzClass.getConstructor(String.class, Integer.class);
        System.out.println(constructor);
        //public com.linn.slarn.spring.bean.register.refrect.Clazz(java.lang.String,java.lang.Integer)

        System.out.println("=====");
        //获取所有的构造器
        Constructor<?>[] declaredConstructors = clazzClass.getDeclaredConstructors();
        for (Constructor<?> declaredConstructor : declaredConstructors) {
            System.out.println(declaredConstructor);
        }
        //com.linn.slarn.spring.bean.register.refrect.Clazz(java.lang.Long,java.lang.String,java.lang.
nteger,java.lang.Object)
        //com.linn.slarn.spring.bean.register.refrect.Clazz(java.lang.Long,java.lang.String,java.lang.
nteger)
        //public com.linn.slarn.spring.bean.register.refrect.Clazz(java.lang.String,java.lang.Integer)
        //public com.linn.slarn.spring.bean.register.refrect.Clazz(java.lang.String)
        //public com.linn.slarn.spring.bean.register.refrect.Clazz()

        System.out.println("=====");
        // 获取指定的构造器
```

```
        Constructor<Clazz> declaredConstructor = clazzClass.getDeclaredConstructor(Long.class String.class, Integer.class);
        System.out.println(declaredConstructor);
        //com.linn.slarn.spring.bean.register.refrect.Clazz(java.lang.Long,java.lang.String,java.lang.nteger)

    }
}
```

获取和方法相关的方法

方法	作用
getMethods()	获得该类所有**公有(public)**的法
getDeclaredMethods()	获得该类所有方法
getMethod(String name, Class...<?> parameterTypes)	据参数类型获得该类某个**公有(public)**的方法
getDeclaredMethod(String name, Class...<?> parameterTypes)	据参数类型获得该类某个方法

```
class SuperClazz {
    public void superM1() {
    }

    void superM2(Long id, String s) {
    }
}

class Clazz extends SuperClazz {

    public void m1(int a) {

    }

    public int m2(int a, int b) {
        return a + b;
    }

    void m3() {
    }

    void m4(String s) {
    }
}

public class ClazzTest {
    public static void main(String[] args) throws Exception {
        Class<Clazz> clazzClass = Clazz.class;
        // 获取所有的公共（public）方法 包括Object中的方法
        Method[] methods = clazzClass.getMethods();
        for (Method method : methods) {
```

```
    System.out.println(method);
}
//public int com.linn.slarn.spring.bean.register.refrect.Clazz.m2(int,int)
//public void com.linn.slarn.spring.bean.register.refrect.Clazz.m1(int)
//public void com.linn.slarn.spring.bean.register.refrect.SuperClazz.superM1()
// Object中方法 很多个

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

// 获取所有的方法 但是不能获取父类的方法
Method[] declaredMethod = clazzClass.getDeclaredMethods();
for (Method method : declaredMethod) {
    System.out.println(method);
}
//public void com.linn.slarn.spring.bean.register.refrect.Clazz.m1(int)
//void com.linn.slarn.spring.bean.register.refrect.Clazz.m4(java.lang.String)
//public int com.linn.slarn.spring.bean.register.refrect.Clazz.m2(int,int)
//void com.linn.slarn.spring.bean.register.refrect.Clazz.m3()

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

// 获取指定方法 (只能获取public的) ,
Method m1 = clazzClass.getMethod("m1", int.class);
System.out.println(m1);

// 报错 （本类中不是public的）
//Method m4 = clazzClass.getMethod("m4", String.class);

// 获取父类的public方法
Method superM1 = clazzClass.getMethod("superM1");
System.out.println(superM1);
//public void com.linn.slarn.spring.bean.register.refrect.SuperClazz.superM1()

// 报错
//Method superM2 = clazzClass.getMethod("superM2",Long.class,String.class);

System.out.println("======");
Method m11 = clazzClass.getDeclaredMethod("m1", int.class);
System.out.println(m11);
//public void com.linn.slarn.spring.bean.register.refrect.Clazz.m1(int)

// 获取不是public的方法
Method m4 = clazzClass.getDeclaredMethod("m4", String.class);
System.out.println(m4);
//void com.linn.slarn.spring.bean.register.refrect.Clazz.m4(java.lang.String)

// 报错 不能获取父类 public 方法
//Method superM11 = clazzClass.getDeclaredMethod("superM1");
//System.out.println(superM11);

// 获取父类 非 public 方法 报错， 不能获取父类的非public方法
//Method superM2 = clazzClass.getDeclaredMethod("superM2", Long.class, String.class);

// getMethods 获取所有 本类和父类的所有 public 方法
```

```
        // getMethod  指定 获取 本类和父类的所有 public 方法

        // getDeclaredMethods 只能获取本类所有方法，不能获取其他的 比如父类的任务方法
        // getDeclaredMethod  指定获取到本类所有方法，不能获取到父类任何方法
    }
}
```

注意：

```
getMethods：  获取所有 本类和父类的所有 public 方法
getMethod：   指定 获取 本类和父类的所有 public 方法
etDeclaredMethods：  只能获取本类所有方法，不能获取其他的 比如父类的任务方法
getDeclaredMethod：  指定获取到本类所有方法，不能获取到父类任何方法
```

获取和方法相关的方法

方法	作用
getFields()	获得该类所有**公有(public)**的属，以及父级public属性
getDeclaredFields()	获得该类所有属性，不能获父级的
getField(String name)	根据参数名 获得该类（者父类，接口）的某个**公有(public)**的属性
getDeclaredField(String name)	根据参数名获该类某个属性，只能是当前类

```
interface Interface0{
    String interfaceName = "0";
}

class SuperClazz {
    public String superName;
    Integer superAge;
}

class Clazz extends SuperClazz implements Interface0{
    public String name;
    Integer age;
}

public class ClazzTest {
    public static void main(String[] args) {
        Class<Clazz> clazzClass = Clazz.class;

        // 获取公共（public）属性,包括父类的或者接口的
        // getField 获取单个
        Field[] fields = clazzClass.getFields();
        for (Field field : fields) {
            System.out.println(field);
        }
        //public java.lang.String com.linn.slarn.spring.bean.register.refrect.Clazz.name
```

```
        //public static final java.lang.String com.linn.slarn.spring.bean.register.refrect.Interface0.i
terfaceName
        //public java.lang.String com.linn.slarn.spring.bean.register.refrect.SuperClazz.superName

        System.out.println("=====");
        // 获取所有本类属性
        // getDeclaredField 获取单个
        Field[] declaredFields = clazzClass.getDeclaredFields();
        for (Field declaredField : declaredFields) {
            System.out.println(declaredField);
        }
        //public java.lang.String com.linn.slarn.spring.bean.register.refrect.Clazz.name
        //java.lang.Integer com.linn.slarn.spring.bean.register.refrect.Clazz.age
    }
}
```

Field API

方法	作用
equals(Object obj)	属性与obj相等则返回true
get(Object obj)	获得obj中对应的属性值
set(Object obj, Object value)	设置obj中对应属值

```
class Clazz {
    public String name;
    private Integer age;
}

public class ClazzTest {

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

        Field name = Clazz.class.getField("name");

        Clazz clazz = new Clazz();
        // 给 clazz 的属性 name 设置 值
        name.set(clazz, "lijun");

        // 获取 clazz 中 name 的值
        System.out.println(name.get(clazz));

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

        Field age = Clazz.class.getDeclaredField("age");
        // 对于 private 修饰的属性 设置为 true 才能操作
        age.setAccessible(true);
        age.set(clazz, 18);
        System.out.println(age.get(clazz));
    }
}
```

也有一块和注解相关的

Method API

方法	作用
invoke(Object obj, Object... args)	传递object象及参数调用该对象对应的方法

```
class Clazz {
    public void add(int a, int b) {
        System.out.println("计算a+b的值"+(a+b));
    }
    public int add(int a, int b,int c) {
        int ret = a + b + c;
        System.out.println("计算a+b+c的值"+ret);
        return ret;
    }
}

public class ClazzTest {
    public static void main(String[] args) throws Exception {
        Method method = Clazz.class.getMethod("add",int.class,int.class);
        Object invoke1 = method.invoke(new Clazz(), 1, 1);
        System.out.println(invoke1);  // null
        Object invoke2 = method.invoke(Clazz.class.newInstance(), 1, 1);
        System.out.println(invoke2); //null

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

        Method method1 = Clazz.class.getMethod("add",int.class,int.class,int.class);
        Object invoke3 = method1.invoke(Class.forName("com.linn.slarn.spring.bean.register.ref
ect.Clazz").newInstance(), 1, 1, 2);
        System.out.println(invoke3); // 4

        // 执行invoke时 ，第一个参数是 目标对象，后面的参数都是 方法的参数，
        // 如果目标方法的返回值为 void， 则 invoke执行后 返回的Object 为 null
        // 如果目标方法的返回值不为 void， 则invoke执行后 返回的 Object 为 目标方法本省返回的值
    }
}
```

Constructor API

方法	作用
newInstance(Object... initargs)	根据传递的参创建类的对象

```
class Clazz {
    public Clazz() {}

    public Clazz(String name,Integer age) {}
```

```
    Clazz(String name,Integer age,Double d) {}
    private Clazz(String name,Integer age,Long id) {
        System.out.println("Clazz(String name,Integer age,Long id)");
    }
}
public class ClazzTest {
    public static void main(String[] args) throws Exception {
        Constructor<Clazz> declaredConstructor = Clazz.class.getDeclaredConstructor(String.cla
s, Integer.class, Long.class);
        System.out.println("=====");
        Class<?>[] parameterTypes = declaredConstructor.getParameterTypes();
        for (Class<?> parameterType : parameterTypes) {
            System.out.println(parameterType);
        }
        //class java.lang.String
        //class java.lang.Integer
        //class java.lang.Long

        // 创建对象 相应的构造方法会被调用 这里的话会在控制台中打印
        // 如果 构造方法为private 则需要设置 setAccessible(true)
        declaredConstructor.setAccessible(true);
        Clazz name = declaredConstructor.newInstance("lijun", 2, 3L);
    }
}
```