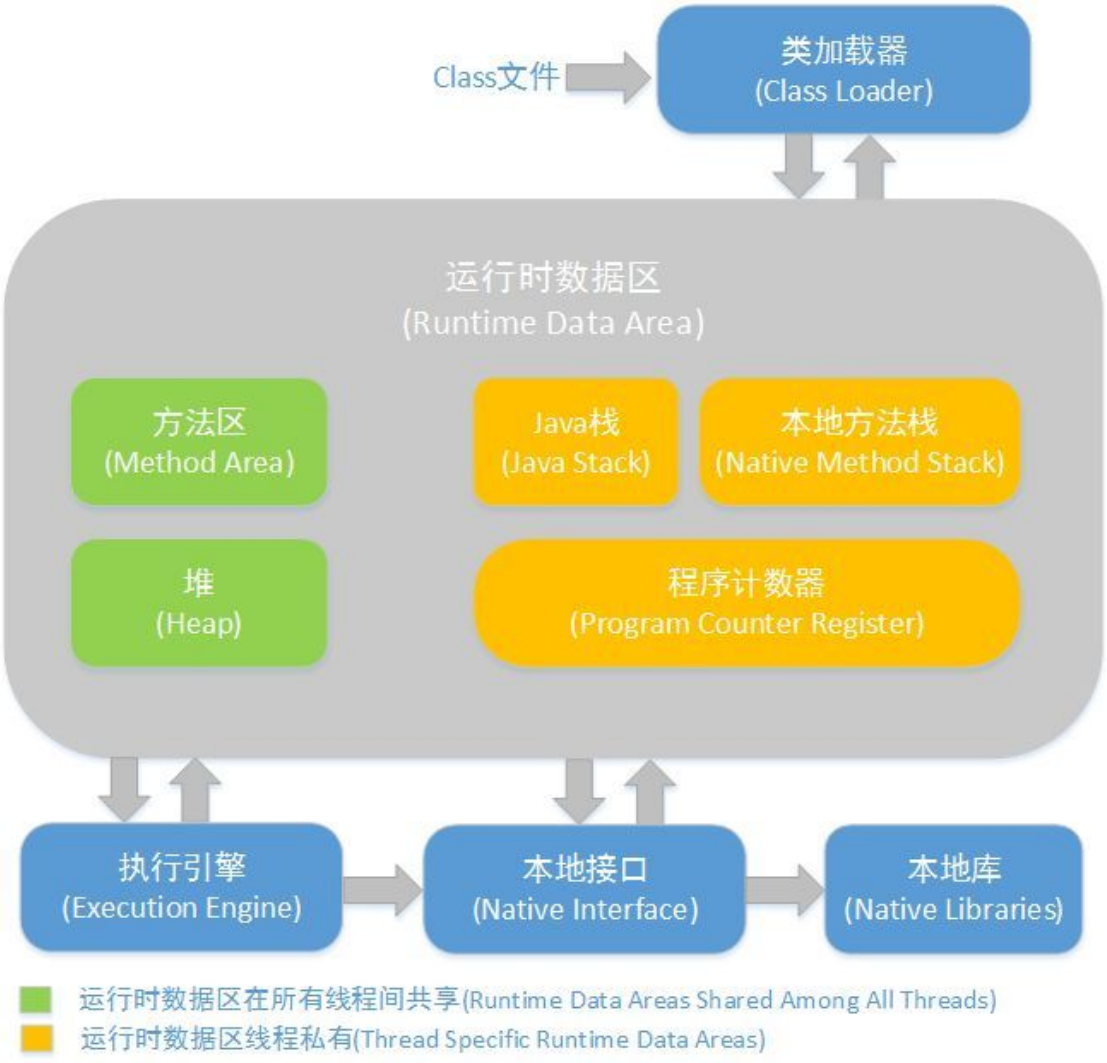
反射机制

笔记本:	java基础		
创建时间:	2019/5/4 10:40	**更新时间:**	2019/5/4 15:24
作者:	ANNER		
URL:	https://www.zhihu.com/question/24304289		

反射机制



Object o=new Object();

首先JVM会启动，代码会编译成一个.class文件，然后被类加载器加载进jvm的内存中，**类Object加载到方法区中**，创建了Object类的class对象到堆中，注意这个不是new出来的对象，而是类的类型对象，**每个类只有一个class对象，作为方法区类的数据结构的接口。**

jvm创建对象前，会先检查类是否加载，寻找类对应的class对象，若加载好，则为你的对象分配内存，初始化也就是代码:new Object()。

上面的程序对象是自己new的，程序相当于写死了给jvm去跑。假如一个服务器上突然遇到某个请求哦要用到某个类，哎呀但没加载进jvm，是不是要停下来自己写段代码，new一下，哦启动一下服务器，（脑残）！

反射是什么呢？当我们的程序在运行时，需要动态的加载一些类这些类可能之前用不到所以不用加载到jvm，而是在运行时根据需要才加载，这样的好处对于服务器来说不言而喻，举个例子我们的项目底层有时是用mysql，有时用oracle，需要动态地根据实际情况加载驱动类，这个时候反射就有用了，假设 com.java.dbtest.myqlConnection，com.java.dbtest.oracleConnection这两个类我们要用，这时候我们的程序就写得比较动态化，通过Class tc = Class.forName("com.java.dbtest.TestConnection");通过类的全类名让jvm在服务器中找到并加载这个类，而如果是oracle则传入的参数就变成另一个了。这时候就可以看到反射的好处了，这个动态性就体现出java的特性了！举多个例子，大家如果接触过spring，会发现当你配置各种各样的bean时，是以配置文件的形式配置的，你需要用到哪些bean就配哪些，spring容器就会根据你的需求去动态加载，你的程序就能健壮地运行

获取class对象

被反射机制加载的类必须有无参数构造方法,否则运行会抛出异常

```
/**
 * @author anner
 * @date 2019/5/4 11:21
 * 获取Class对象三种方法
 */
public class TestGetClass {
    public static void main(String[] args) throws Exception{
        Class c1=Class.forName("java_base.reflaction.Person");
        Class c2=new Person().getClass();
        Class c3=Person.class;
    }
}
```

获取构造方法

方法	描述
public Constructor getConstructor(Class... parameterTypes)	获得指定的构造方法，**注意只能获得 public 权限的构造方法，其他访问权限的获取不到**
public Constructor getDeclaredConstructor(Class... parameterTypes)	**获得指定的构造方法，注意可以获取到任何访问权限的构造方法。**
public Constructor[] getConstructors() throws SecurityException	获得所有 public 访问权限的构造方法
public Constructor[] getDeclaredConstructors() throws SecurityException	获得所有的构造方法，包括（public, private,protected,默认权限的）

```java
public class TestConstructor {
    public static void main(String[] args) {
        String className="java_base.reflaction.Person";
        try {
            Class c=Class.forName(className);
            //获取全部的构造函数对象
            Constructor []constructors=c.getDeclaredConstructors();
            //打印对应的信息
            for(Constructor constructor:constructors){
                System.out.print("构造函数:"+constructor.getName()+" 参数类型: ");
                Class []paramtypes=constructor.getParameterTypes();
                for(Class param:paramtypes){
                    System.out.print(param.getName()+" , ");
                }
                System.out.println();
            }
            //调用构造函数构造信息
            Constructor constructor=c.getConstructor(String.class,String.class,int.class,long.class,boolean.class,String.class);
            Person p= (Person) constructor.newInstance( ...initargs: "anner","123",20,181,true,"whut");
            System.out.println(p.toString());
        }catch (Exception e){
            e.printStackTrace();
        }
    }
}
```

获取成员方法

```
        StudentInfo studentInfo = new StudentInfo();
        Class<? extends StudentInfo> class1 = studentInfo.getClass();

        Method[] methods = class1.getDeclaredMethods();
        for (Method method : methods) {
            System.out.print("方法名 :  "+method.getName());
            Class<?>[] parameterTypes = method.getParameterTypes();
            for (Class<?> class2 : parameterTypes) {
                System.out.print(" 参数类型:  "+class2.getName());
            }
            Class<?> returnType = method.getReturnType();
            System.out.println(" 返回值类型:  "+returnType.getName());
        }

        try {
            //1、调用非静态方法
            Method method1 = class1.getDeclaredMethod("setName", String.class);
            Method method2 = class1.getDeclaredMethod("getName");
            method1.invoke(studentInfo, "李四");
            String name = (String) method2.invoke(studentInfo);
            System.out.println("调用非静态方法 getName()==name: "+name);

            //2、调用静态方法：将invoke的第一个参数设置为null
            System.out.println("调用静态方法 test(): ");
            Method method3 = class1.getDeclaredMethod("test");
            method3.invoke(null);
            Method method4 = class1.getDeclaredMethod("test", String.class);
            method4.invoke(null, "123456");

            //3、调用私有方法：方法调用之前，需要将方法setAccessible
            Method method5 = class1.getDeclaredMethod("getAge");
            method5.setAccessible(true);
            System.out.println("调用私有方法 getAge()=="+method5.invoke(studentInfo));
```

获取成员变量

```
private static void class2GetField() {
        Class<?> class1 =  StudentInfo.class;
        Field[] fields = class1.getDeclaredFields();
        for (Field field : fields) {
            System.out.print("变量名 :  "+field.getName());
            Class<?> type = field.getType();
            System.out.println(" 类型 :  "+type.getName());
        }

        try {
            //访问非私有变量
            StudentInfo studentInfo = new StudentInfo();
            Field field = class1.getDeclaredField("name");
            System.out.println(" 变量name== "+field.get(studentInfo));
            Field field2 = class1.getDeclaredField("school");
            System.out.println(" 静态变量school== "+field2.get(studentInfo));

            //访问私有变量
            Field field3 = class1.getDeclaredField("age");
            field3.setAccessible(true);
            System.out.println(" 私有变量age== "+field3.get(studentInfo));
            field3.set(studentInfo, 18);
            System.out.println(" 私有变量age== "+field3.get(studentInfo));

        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
```

获取类、方法、属性的修饰域

类Class、Method、Constructor、Field都有一个public方法int getModifiers()。该方法返回一个int类型的数，表示被修饰对象（Class、 Method、 Constructor、 Field）的修饰类型的组合值。

。

int	ABSTRACT	The `int` value representing the `abstract` modifier.
int	FINAL	The `int` value representing the `final` modifier.
int	INTERFACE	The `int` value representing the `interface` modifier.
int	NATIVE	The `int` value representing the `native` modifier.
int	PRIVATE	The `int` value representing the `private` modifier.
int	PROTECTED	The `int` value representing the `protected` modifier.
int	PUBLIC	The `int` value representing the `public` modifier.
int	STATIC	The `int` value representing the `static` modifier.
int	STRICT	The `int` value representing the `strict` modifier.
int	SYNCHRONIZED	The `int` value representing the `synchronized` modifier.
int	TRANSIENT	The `int` value representing the `transient` modifier.
int	VOLATILE	The `int` value representing the `volatile` modifier.