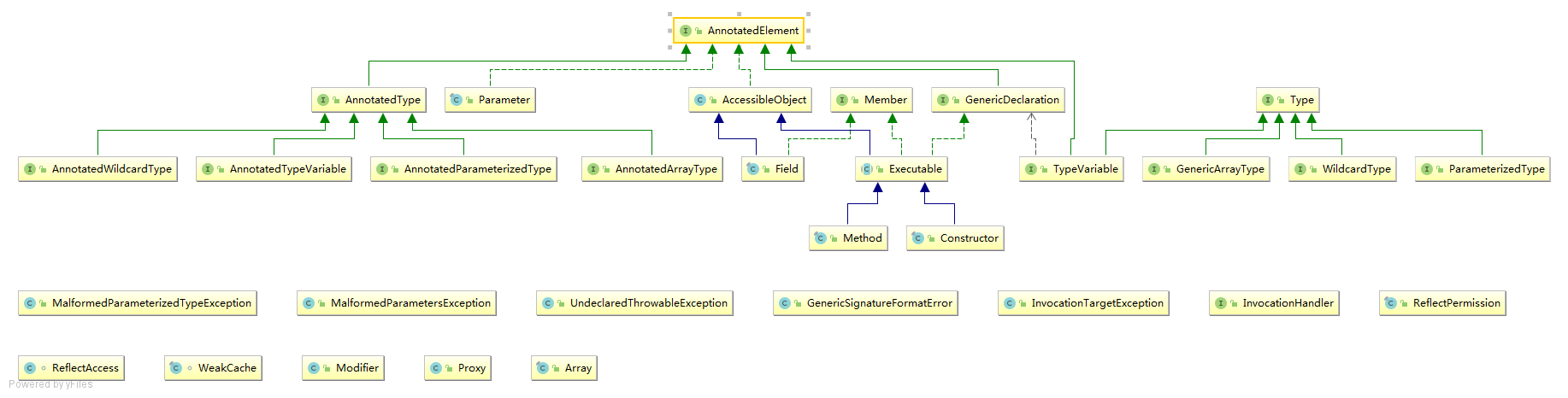
反射

## 反射机制

### 概念

根据类名通过Reflection API（java.lang.reflect）在运行时动态获取类的內部信息并操作。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.lang.reflect包



### 优点

反射提高了程序的灵活性和扩展性，降低耦合性，提高自适应能力。它允许程序创和控制任何类的对象，无需提前硬编码目标类，

### 缺点

性能问题，使用反射基本上是一种解释操作，用于字段和方法接入时要远慢于直接代码。因此反射机制主要应用在对灵活性和扩展性要求很高的系统框架上如mybtis和spring等，普通程序不建议使用

## Class理解

### 概念

java.lang.Class普通类，Class的实例是正在运行的java类或接口。枚举、注释、数组、基本类型、关键字void等都是类或接口，如Class a=int.class.Class类的构造方法是private修饰的，所以开发人员不能实例化，其是被jvm自动构造，每一个被实例化的java类，jvm中都有且仅一个java类的class。

### 获取方式

1. Class.forName(类)：装入类,并做类的静态初始化，返回Class的对象
2. 对象.getClass()：对类进行静态初始化、非静态初始化；返回引用运行时真正所指的对象所属的类的Class的对象，子类和父类equal是不等的。
3. 类.class：JVM将使用类装载器, 将类装入内存(前提是:类还没有装入内存),不做类的初始化工作.返回Class的对象

### 作用

每个类的运行时的类型信息就是用Class对象表示的，所以是反射的核心，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，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)

## 部分类分析

### Field

提供有关类或接口的单个字段的信息和动态访问。反射字段可以是类字段或实例字段。

Class blogClass = Blog.class;  
Field field = blogClass.getField("name");

### Constructor

提供有关类的单个构造函数的信息和访问权限

Class<Blog> blogClass = Blog.class;  
Constructor<Blog> constructor = blogClass.getConstructor();  
Blog blog = constructor.newInstance();

### Method

提供有关类或接口上的单个方法的信息和访问权限。反射的方法可以是类方法或实例方法（包括抽象方法）。

Class<Blog> blogClass = Blog.class;  
Blog blog= blogClass.newInstance();  
Method method = blogClass.getMethod("setNowDate", Date.class);  
method.invoke(blog,new Date());  
System.out.println(blog.getNowDate());

### Modifier

该类是静态类，其中的方法也是静态方法。getModifiers()函数返回一个用于描述类，构造器，方法和域的修饰符的整形数值。调用Modifier.toString()方法将整型数值转变成字符串，也是就我们熟悉的public，private，static，final等修饰符。

Class<Blog> blogClass = Blog.class;  
int lei= blogClass.getModifiers();  
int met=blogClass.getDeclaredMethod("say").getModifiers();  
String leiToStr=Modifier.toString(lei);  
String metToStr=Modifier.toString(met);  
System.out.println(leiToStr+"--"+metToStr);

### Array

提供了动态创建数组，以及访问数组的元素的静态方法

int[] e = (int[]) Array.newInstance(int.class,2);  
Array.set(e,0,1);  
Array.set(e,1,2);

### Proxy

提供了用于创建动态代理\*类和实例的静态方法，它也是由这些方法创建的所有\*动态代理类的超类

### InvocationHandler

代理实例的调用处理程序实现的接口，在执行实际业务前后的操作都在这里添加。

## Jdk动态代理

### 创建示例

|  |
| --- |
| //从源码可知，可以将字节码生成class文件保存在本地  System.getProperties().put("sun.misc.ProxyGenerator.saveGeneratedFiles", "true"); InvocationHandler handler = new MyInvocationHander<Person>(new PersonImp()); Person personPro = (Person) Proxy.newProxyInstance(Person.class.getClassLoader(),  new Class<?>[]{Person.class,Person2.class},handler); personPro.say(); |

### 分析

Proxy.newProxyInstance方法中关键源码,获取代理类class,Class<?> cl = getProxyClass0(loader, intfs),根据反射获取代理类实例cons.newInstance(new Object[]{h}),而getProxyClass0方法中调用ProxyClassFactory.apply产生代理类，底层实现原理大概为根据类名、方法、修饰符等参数在jvm中拼接成类的字节码，在根据字节码动态的生成代理类。

### 代理类

|  |
| --- |
| public final class $Proxy0 extends Proxy implements Person{  public $Proxy0(InvocationHandler var1) throws {  super(var1);  }  public final void say() throws {  try {  super.h.invoke(this, m3, (Object[])null);  } catch (RuntimeException | Error var2) {  throw var2;  } catch (Throwable var3) {  throw new UndeclaredThrowableException(var3);  }  ...... } |

### 总结

* 由生成的类 $Proxy0可知继承了Proxy，所以被代理的一定要是接口在ProxyClassFactory.apply中也有验证条件，这也是和cglib代理的本质区别。
* 从方法say中可知，最终走的是MyInvocationHander中的invoke方法，所以用户的逻辑都可在invoke中实现，最后调用Mehtod.invoke说明jdk动态代理基于反射。