众所周知，java有三大特性: 封装继承和多态，封装是为了细化权限，继承是为了多态，多态是为了灵活复用。我们又知道，继承破坏了封装，但是，它破坏的不彻底，有没有破坏的更彻底的呢？有，反射! 反射效率很低，因为它工作在运行时，为什么工作在运行时呢？我们先来看个例子:

public class User {

// 创建一个私有字段name

private String name;

public User(String name) {

this.name = name;

}

}

public void test(){

// 创建一个User对象，name为:java

User user = new User("java");

// 反射的工作目标是.class对象

Class aclass = User.class;

// 获取声明的字段:name

Field nameField = aclass.getDeclaredField("name");

// 允许访问非public的字段

nameField.setAccessible(true);

// 获取此字段在user对象上的值

Object nameValue = nameField.get(user);

// 打印

System.out.println(nameValue);

}

打印结果:

java

Process finished with exit code 0

**反射的工作目标是.class对象**，也就是说，如果把反射比作一个函数，那么它的入参是.class对象，那么这个.class对象从哪里来呢？答曰: 类加载！

**类加载机制**

JVM类加载流程:

* 1 通过全限定类名来获取定义此类的二进制字节流。
* 2 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。
* 3 在内存中生成一个代表这个类的 java.lang.Class 对象，作为方法区这个类的各种数据的访问入口。

简而言之，就是: JVM类加载机制 接收的是: .class文件的二进制字节流，产出的是: java.lang.Class对象。

所以可以说: 类加载机制产出了反射可以进行工作的目标，也就是Class对象，经过了类加载机制，反射才有了入参(Class对象)，才可以工作，而**类加载工作在运行时，反射工作在类加载之后，所以: 反射工作在运行时!**

A a = new B();

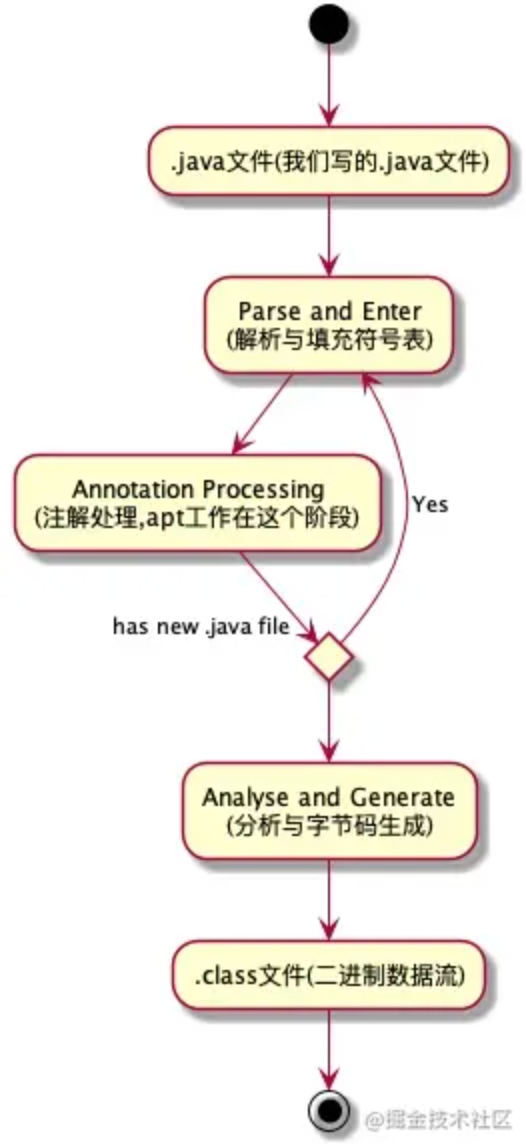
其中B是A的子类，这就是多态，只有在运行时，才知道它是什么类型，这就是java的动态特性，所以JVM的类加载是在运行时，这虽然一定程度上降低了效率，但是带来的作用是巨大的，这也是反射为什么工作在运行时，因为它需要的Class对象要在运行时才有。

那么既然JVM类加载机制 接收.class文件，产出java.lang.Class对象，那么它接收的.class文件从哪来呢？有人已经想到了，就是javac！也就是编译！

**编译过程**

我们知道，写一个Hello.java文件，然后运行指令:javac Hello.java，就会生成一个Hello.class文件，这个文件是个二进制流，正是JVM类加载需要的东西！ javac代表了编译过程，运行了这个指令，就会找到你设置的环境变量里面的相关程序，去进行编译，大家也可以直接在:com.sun.tools.javac.main.JavaCompiler找相关代码，看一下这个.class文件是怎么生成的，大概三个步骤:

* 1 解析与填充符号表，此阶段执行词法分析，语法分析，生成抽象语法树等。
* 2 注解处理，此阶段着重处理注解，也是apt的工作阶段，比如arouter。
* 3 分析与字节码生成，此阶段执行常量折叠，解语法糖等操作。



可以看到，如果在注解处理过程中，有新文件生成，则会跳转到第一步继续"解析与填充符号表"，直到不再生成java文件为止。这也是arouter的工作原理: 通过注解处理器和代码模板，在编译时动态生成文件，我们使用arouter时候，发现的一些ARouter$$Group$$app、ARouter$$Providers$$app等文件就是这么来的。

在编译阶段之后，我们就得到了.class文件，也就是jvm加载需要的东西！

我们来小结一下:

* 1 我们编写.java文件，输入相关代码
* 2 经过javac编译，通过.java文件生成了.class文件
* 3 经过JVM加载，通过.class文件生成了java.lang.Class对象
* 4 我们使用这些Class对象执行各种操作，比如反射

泛型

public static void getGeneral() {

Class aClass = ArrayList.class;

// Class的范型参数

TypeVariable[] typeParameters = aClass.getTypeParameters();

Field field = null;

// field的范型类型

Type genericType = field.getGenericType();

Method method = null;

// method的范型参数

method.getGenericParameterTypes();

// method的范型返回值

method.getGenericReturnType();

// method的范型异常

method.getExceptionTypes();

Constructor constructor = null;

// 构造器的范型参数

constructor.getGenericParameterTypes();

// 构造器的范型异常

constructor.getGenericExceptionTypes();

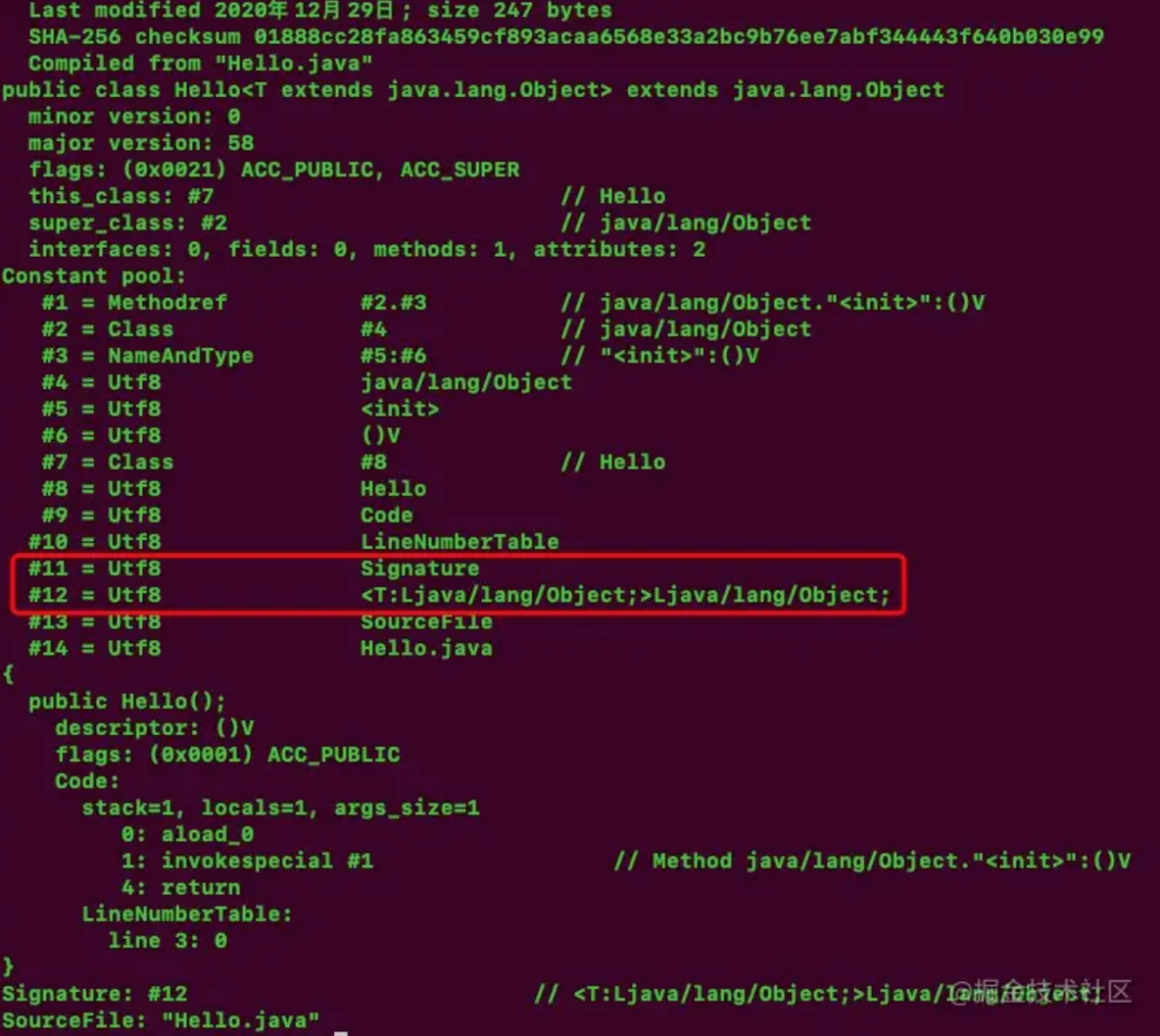
}

这里有人会问，java不是有泛型擦除吗，既然都擦除了，怎么还有泛型？没错，java的泛型在编译期就擦除了，但是擦除的同时，会在.class文件的属性表(attribute\_info)里面写入一个Signature信息，这里面存放了泛型的相关信息，我们获取的泛型信息就是从这个属性里面获取的，所以java的泛型称为"伪泛型"。我们来验证一下，随便写一个泛型类:

public class Hello<T> {

}

然后javac Hello.java得到Hello.class对象，再执行javap -verbose Hello.class查看:



我们再去掉范型信息看下:

public class Hello {

}



可以看到，里面的Signature属性没有了，也就是说，javac在编译的时候，擦除了泛型，然后把相关信息保存在常量池的Signature里面去了。

Java 反射效率低主要原因是：

1、Method#invoke 方法会对参数做封装和解封操作

2、需要检查方法可见性

3、需要校验参数

4、反射方法难以内联

5、JIT 无法优化