Java

1.反射

1.1.反射的概念

反射(Reflection)是 Java 程序开发语言的特征之一，它允许运行中的 Java 程序获取自身的信息，并且可以操作类或对象的内部属性。

通过反射机制，可以在运行时访问 Java 对象的属性，方法，构造方法等。

1.2.反射的应用场景

反射的主要应用场景有：

- 开发通用框架 - 反射最重要的用途就是开发各种通用框架。很多框架（比如 Spring）都是配置化的（比如通过 XML 文件配置 JavaBean、Filter 等），为了保证框架的通用性，它们可能需要根据配置文件加载不同的对象或类，调用不同的方法，这个时候就必须用到反射——运行时动态加载需要加载的对象。
- 动态代理 - 在切面编程（AOP）中，需要拦截特定的方法，通常，会选择动态代理方式。这时，就需要反射技术来实现了。
- 注解 - 注解本身仅仅是起到标记作用，它需要利用反射机制，根据注解标记去调用注解解释器，执行行为。如果没有反射机制，注解并不比注释更有用。
- 可扩展性功能 - 应用程序可以通过使用完全限定名称创建可扩展性对象实例来使用外部的用户定义类。

1.3.反射的缺点

- 性能开销 - 由于反射涉及动态解析的类型，因此无法执行某些 Java 虚拟机优化。因此，反射操作的性能要比非反射操作的性能要差，应该在性能敏感的应用程序中频繁调用的代码段中避免。
- 破坏封装性 - 反射调用方法时可以忽略权限检查，因此可能会破坏封装性而导致安全问题。
- 内部曝光 - 由于反射允许代码执行在非反射代码中非法的操作，例如访问私有字段和方法，所以反射的使用可能会导致意想不到的副作用，这可能会导致代码功能失常并可能破坏可移植性。反射代码打破了抽象，因此可能会随着平台的升级而改变行为。

1.4.反射机制

1.4.1.类加载过程

类加载的完整过程如下：

1. 在编译时，Java 编译器编译好 .java 文件之后，在磁盘中产生 .class 文件。.class 文件是二进制文件，内容是只有 JVM 能够识别的机器码。
2. JVM 中的类加载器读取字节码文件，取出二进制数据，加载到内存中，解析.class 文件内的信息。类加载器会根据类的全限定名来获取此类的二进制字节流；然后，将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；接着，在内存中生成代表这个类的 java.lang.Class 对象。
3. 加载结束后，JVM 开始进行连接阶段（包含验证、准备、初始化）。经过这一系列操作，类的变量会被初始化。

1.4.2.Class 对象

要想使用反射，首先需要获得待操作的类所对应的 Class 对象。**Java 中，无论生成某个类的多少个对象，这些对象都会对应于同一个 Class 对象。这个 Class 对象是由 JVM 生成的，通过它能够获悉整个类的结构。** 所以，java.lang.Class 可以视为所有反射 API 的入口点。

反射的本质就是：在运行时，把 Java 类中的各种成分映射成一个个的 Java 对象。

举例来说，假如定义了以下代码：

```
User user = new User();
```

步骤说明：

1. JVM 加载方法的时候，遇到 new User()，JVM 会根据 User 的全限定名去加载 User.class 。
2. JVM 会去本地磁盘查找 User.class 文件并加载 JVM 内存中。
3. JVM 通过调用类加载器自动创建这个类对应的 Class 对象，并且存储在 JVM 的方法区。注意：一个类有且只有一个 Class 对象。

1.4.3.使用反射

java.lang.reflect 包:

Java 中的 java.lang.reflect 包提供了反射功能。java.lang.reflect 包中的类都没有 public 构造方法。

java.lang.reflect 包的核心接口和类如下：

- Member 接口 - 反映关于单个成员(字段或方法)或构造函数的标识信息。
- Field 类 - 提供一个类的域的信息以及访问类的域的接口。
- Method 类 - 提供一个类的方法的信息以及访问类的方法的接口。
- Constructor 类 - 提供一个类的构造函数的信息以及访问类的构造函数的接口。

- Array 类 - 该类提供动态地生成和访问 JAVA 数组的方法。
- Modifier 类 - 提供了 static 方法和常量，对类和成员访问修饰符进行解码。
- Proxy 类 - 提供动态地生成代理类和类实例的静态方法。

获得 Class 对象:

获得 Class 的三种方法：

（1）使用 Class 类的 forName 静态方法:

使用类的完全限定名来反射对象的类。常见的应用场景为：在 JDBC 开发中常用此方法加载数据库驱动。

```
package io.github.dunwu.javacore.reflect;

public class ReflectClassDemo01 {
    public static void main(String[] args) throws ClassNotFoundException {
        Class c1 = Class.forName("io.github.dunwu.javacore.reflect.ReflectClassDemo01");
        System.out.println(c1.getCanonicalName());
        Class c2 = Class.forName("[D");
        System.out.println(c2.getCanonicalName());
        Class c3 = Class.forName("[[Ljava.lang.String;");
        System.out.println(c3.getCanonicalName());
    }
}
//Output:
//io.github.dunwu.javacore.reflect.ReflectClassDemo01
//double[]
//java.lang.String[][]
```

（2）直接获取某一个对象的 class:

```
public class ReflectClassDemo02 {
    public static void main(String[] args) {
        Boolean b;
        // Class c = b.getClass(); // 编译错误
        Class c1 = Boolean.class;
        System.out.println(c1.getCanonicalName());
        Class c2 = java.io.PrintStream.class;
        System.out.println(c2.getCanonicalName());
        Class c3 = int[][][].class;
        System.out.println(c3.getCanonicalName());
}
}
//Output:
//boolean
//java.io.PrintStream
//int[][][]
```

(3) 调用 Object 的 getClass 方法:

Object 类中有 getClass 方法，因为所有类都继承 Object 类。从而调用 Object 类来获取

```
package io.github.dunwu.javacore.reflect;
import java.util.HashSet;
import java.util.Set;
public class ReflectClassDemo03 {
    enum E {
        A, B
    }
    public static void main(String[] args) {
        Class c = "foo".getClass();
        System.out.println(c.getCanonicalName());
        Class c2 = ReflectClassDemo03.E.A.getClass();
        System.out.println(c2.getCanonicalName());
        byte[] bytes = new byte[1024];
        Class c3 = bytes.getClass();
        System.out.println(c3.getCanonicalName());
        Set<String> set = new HashSet<>();
        Class c4 = set.getClass();
        System.out.println(c4.getCanonicalName());
    }
}
//Output:
//java.lang.String
//io.github.dunwu.javacore.reflect.ReflectClassDemo.E
//byte[]
//java.util.HashSet
```

判断是否为某个类的实例:

判断是否为某个类的实例有两种方式：

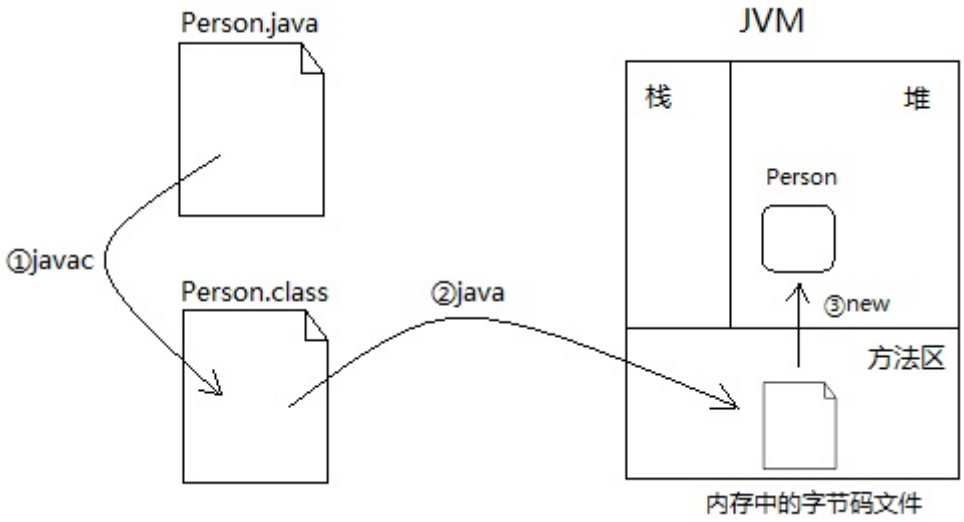
- 用 instanceof 关键字
- 用 Class 对象的 isInstance 方法（它是一个 Native 方法）

```
public class InstanceofDemo {
    public static void main(String[] args) {
        ArrayList arrayList = new ArrayList();
        if (arrayList instanceof List) {
            System.out.println("ArrayList is List");
        }
        if (List.class.isInstance(arrayList)) {
            System.out.println("ArrayList is List");
        }
    }
}
//Output:
//ArrayList is List
//ArrayList is List
```

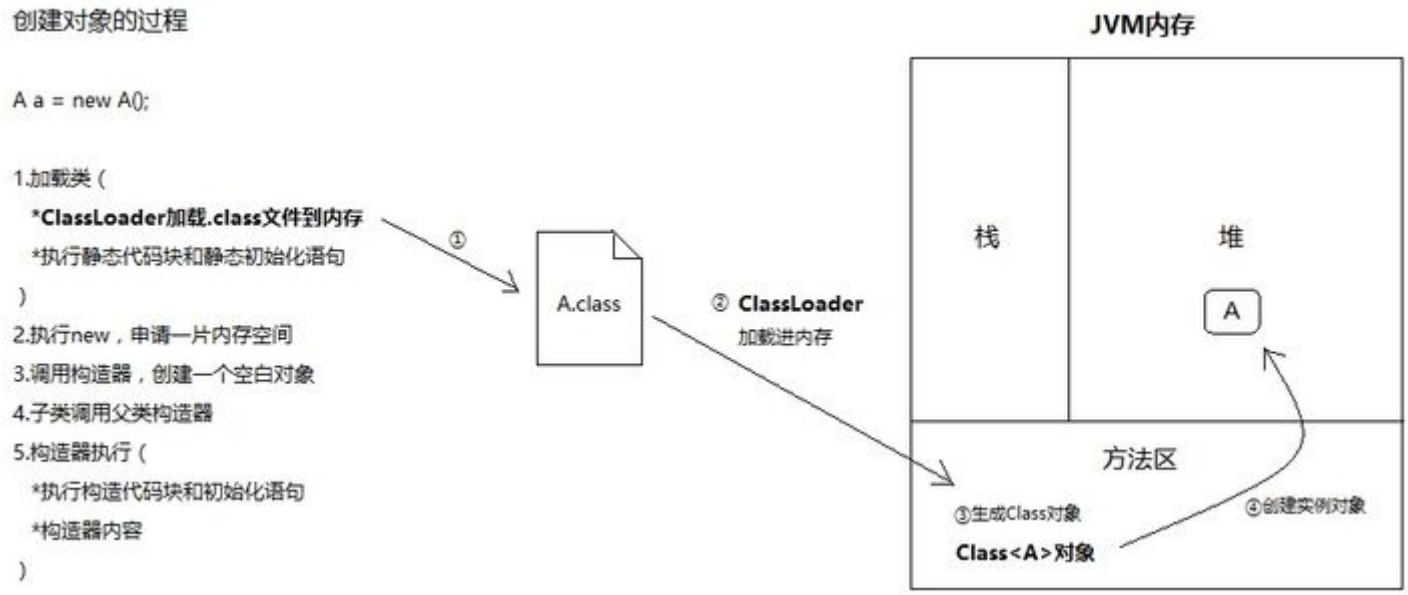
1.5.JVM是如何构建一个实例的

- 内存：即JVM内存，栈、堆、方法区啥的都是JVM内存，只是人为划分
- .class文件：就是所谓的字节码文件，这里称.class文件，直观些

```
//假设main方法中有以下代码：

Person p = new Person();

//很多初学者会以为整个创建对象的过程是下面这样的

javac Person.java
java Person
```



稍微细致一点的过程可以是下面这样的



通过new创建实例和反射创建实例，都绕不开Class对象。

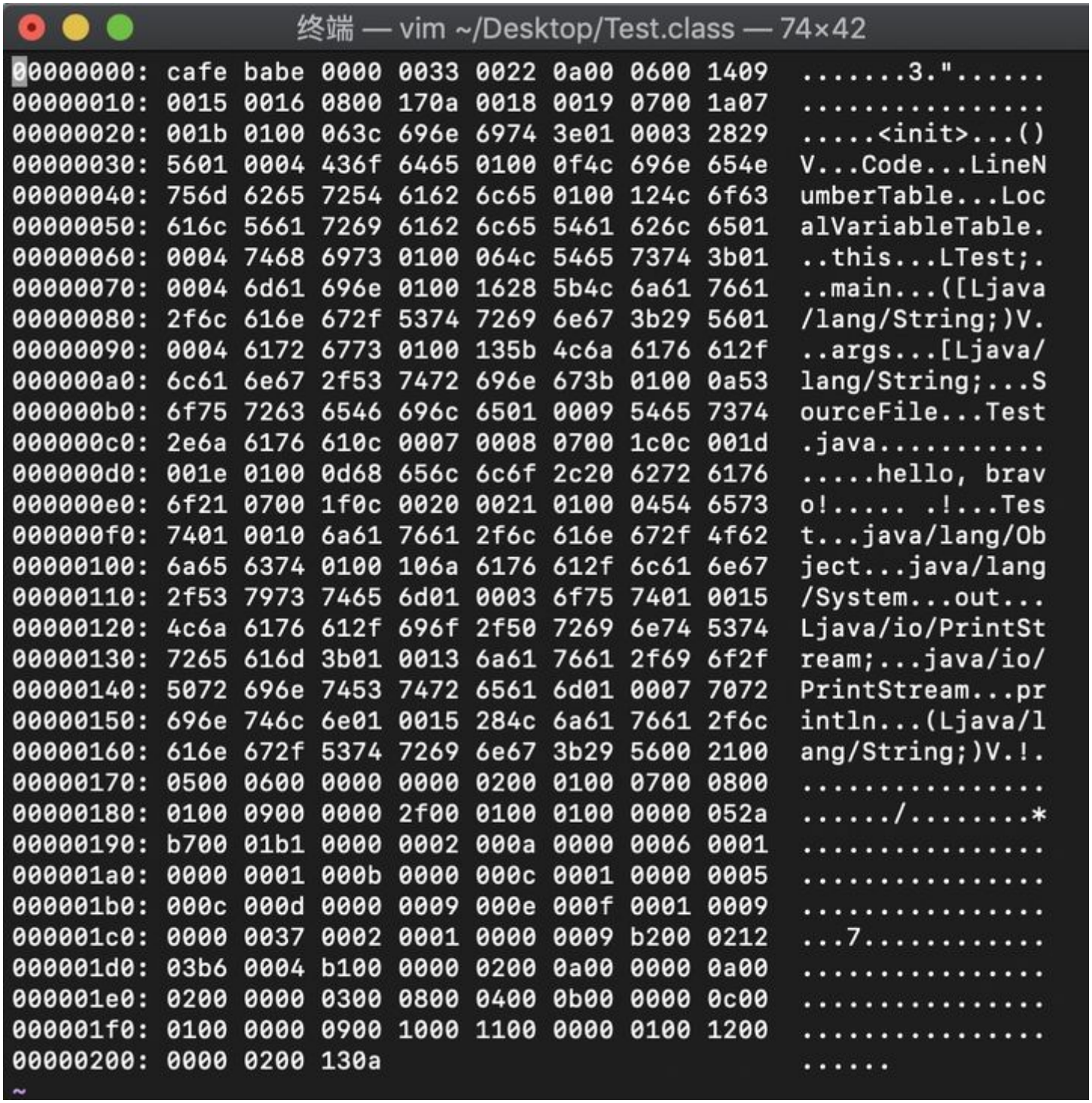
1.5.1.class文件

比如我现在写一个类

Test.java

```
1  public class Test {
2      public static void main(String[] args) {
3          System.out.println("hello, bravo!");
4      }
5  }
```

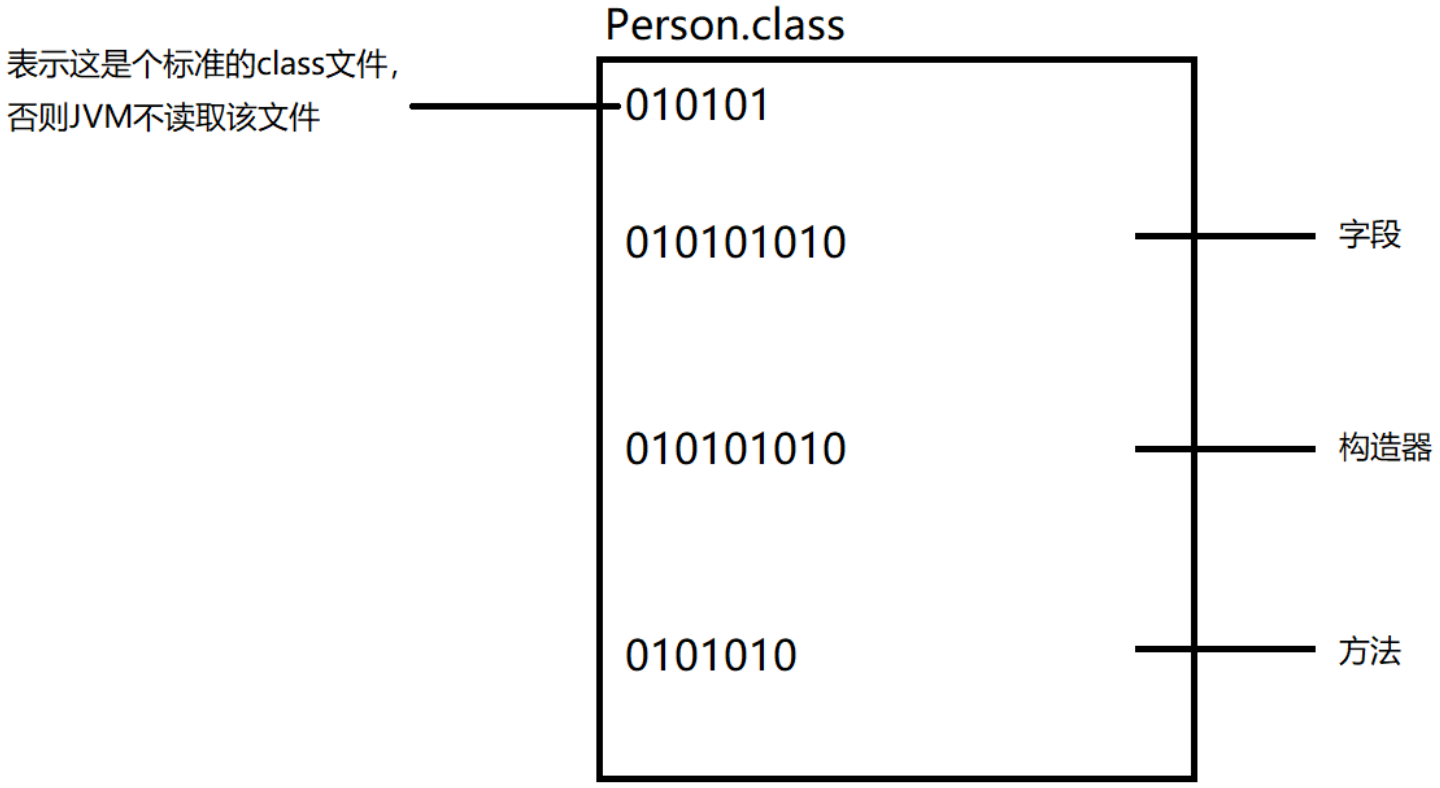
用vim命令打开.class文件，以16进制显示就是下面这副鬼样子：



在计算机中，任何东西底层保存的形式都是0101代码。

.java源码是给人类读的，而.class字节码是给计算机读的。根据不同的解读规则，可以产生不同的意思。就好比“这周日你有空吗”，合适的断句很重要。

同样的，JVM对.class文件也有一套自己的读取规则，不需要我们操心。总之，0101代码在它眼里的样子，和我们眼中的英文源码是一样的。



1.5.2类加载器

在最开始复习对象创建过程时，我们了解到.class文件是由类加载器加载的。但是核心方法只有loadClass()，告诉它需要加载的类名，它会帮你加载：

```
protected Class<?> loadClass(String name, boolean resolve)
        throws ClassNotFoundException
    {
        synchronized (getClassLoadingLock(name)) {
            // 首先，检查是否已经加载该类
            Class<?> c = findLoadedClass(name);
            if (c == null) {
                long t0 = System.nanoTime();
                try {
                    // 如果尚未加载，则遵循父优先的等级加载机制（所谓双亲委派机制）
                    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
                }

                if (c == null) {
                    // 模板方法模式：如果还是没有加载成功，调用findClass()
                    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;
        }
    }

    // 子类应该重写该方法
    protected Class<?> findClass(String name) throws ClassNotFoundException {
        throw new ClassNotFoundException(name);
    }
```

加载.class文件大致可以分为3个步骤：

1. 检查是否已经加载，有就直接返回，避免重复加载
2. 当前缓存中确实没有该类，那么遵循父优先加载机制，加载.class文件
3. 上面两步都失败了，调用findClass()方法加载

需要注意的是，ClassLoader类本身是抽象类，而抽象类是无法通过new创建对象的。所以它的findClass()方法写的很随意，直接抛了异常，反正你无法通过ClassLoader对象调用。也就是说，父类ClassLoader中的findClass()方法根本不会去加载.class文件。

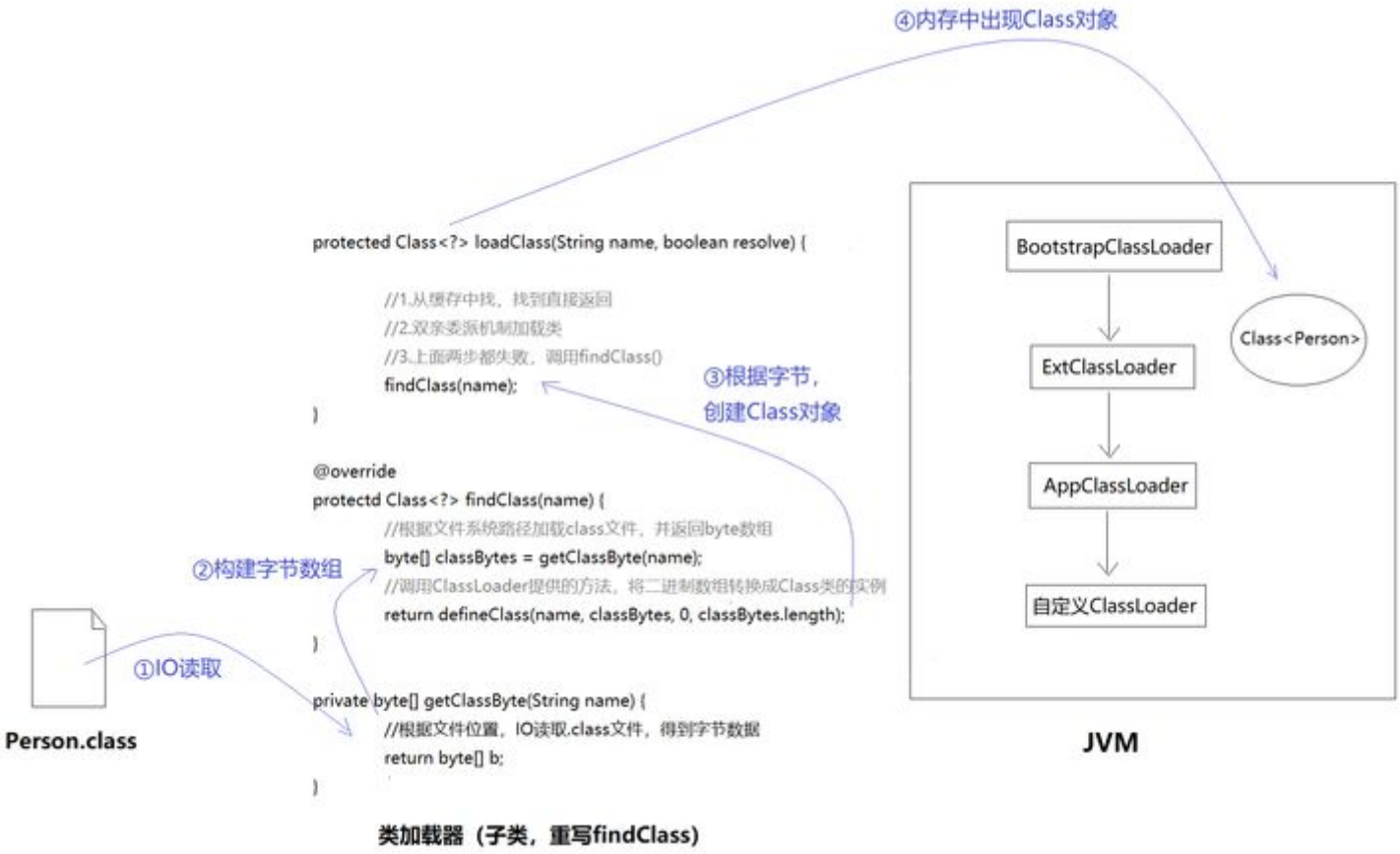
正确的做法是，子类重写覆盖findClass()，在里面写自定义的加载逻辑。比如：

```
@Override
public Class<?> findClass(String name) throws ClassNotFoundException {
    try {
        /*自己另外写一个getClassData()
                通过IO流从指定位置读取xxx.class文件得到字节数组*/
        byte[] datas = getClassData(name);
        if(datas == null) {
            throw new ClassNotFoundException("类没有找到：" + name);
        }
        //调用类加载器本身的defineClass()方法，由字节码得到Class对象
        return defineClass(name, datas, 0, datas.length);
    } catch (IOException e) {
        e.printStackTrace();
        throw new ClassNotFoundException("类找不到：" + name);
    }
}
```

defineClass()是ClassLoader定义的方法，目的是根据.class文件的字节数组byte[] b造出一个对应的Class对象。我们无法得知具体是如何实现的，因为最终它会调用一个native方法：

```
private native Class<?> defineClass0(String name, byte[] b, int off, int len,
                                     ProtectionDomain pd);

private native Class<?> defineClass1(String name, byte[] b, int off, int len,
                                     ProtectionDomain pd, String source);

private native Class<?> defineClass2(String name, java.nio.ByteBuffer b,
                                     int off, int len, ProtectionDomain pd,
                                     String source);
```

反正，目前我们关于类加载只需知道以下信息：



1.5.3.Class类

现在，.class文件被类加载器加载到内存中，并且JVM根据其字节数组创建了对应的Class对象。所以，我们来研究一下Class对象。

Class对象是Class类的实例，我们将在这一小节一步步分析Class类的结构。

但是，在看源码之前，我想问问聪明的各位，如果你是JDK源码设计者，你会如何设计Class类？

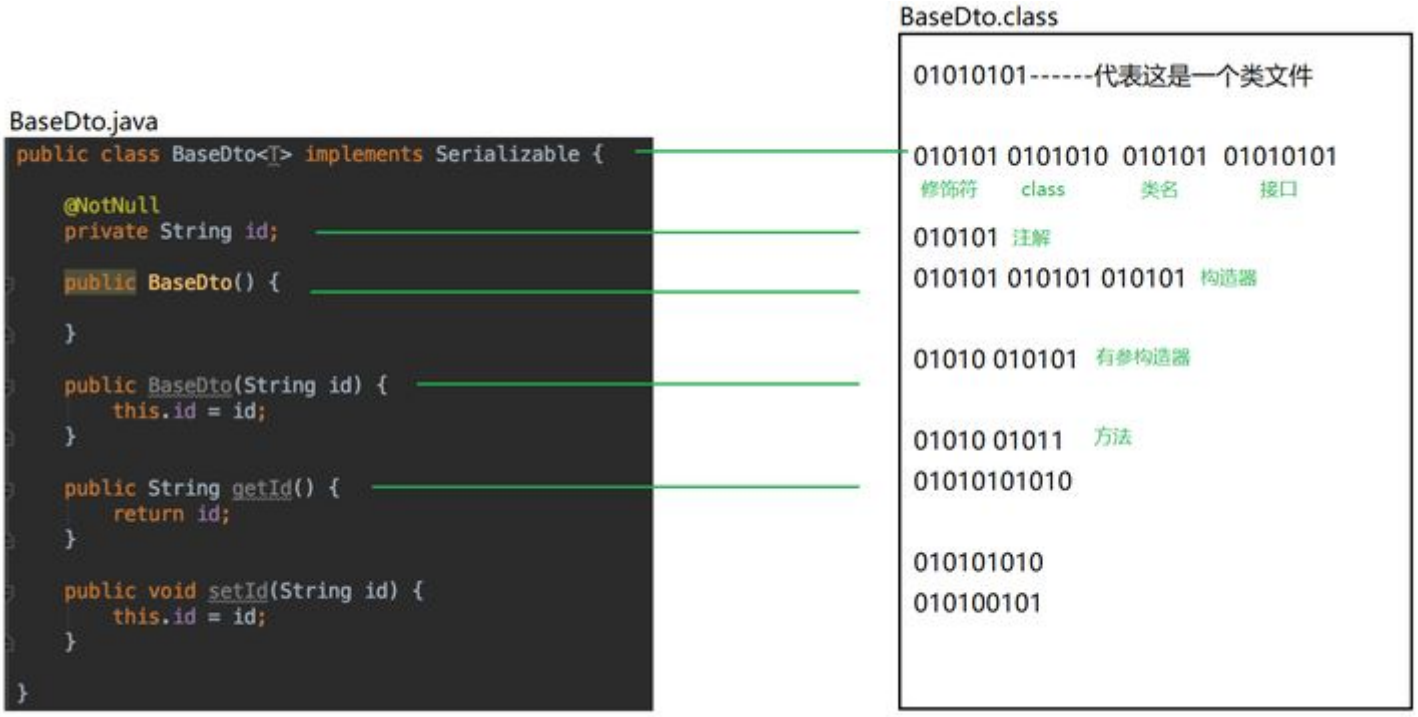
假设现在有个BaseDto类

```java
public class BaseDto<T> implements Serializable {

    @NotNull
    private String id;

    public BaseDto() {

    }

    public BaseDto(String id) {
        this.id = id;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

}
```

上面类至少包括以下信息（按顺序）：

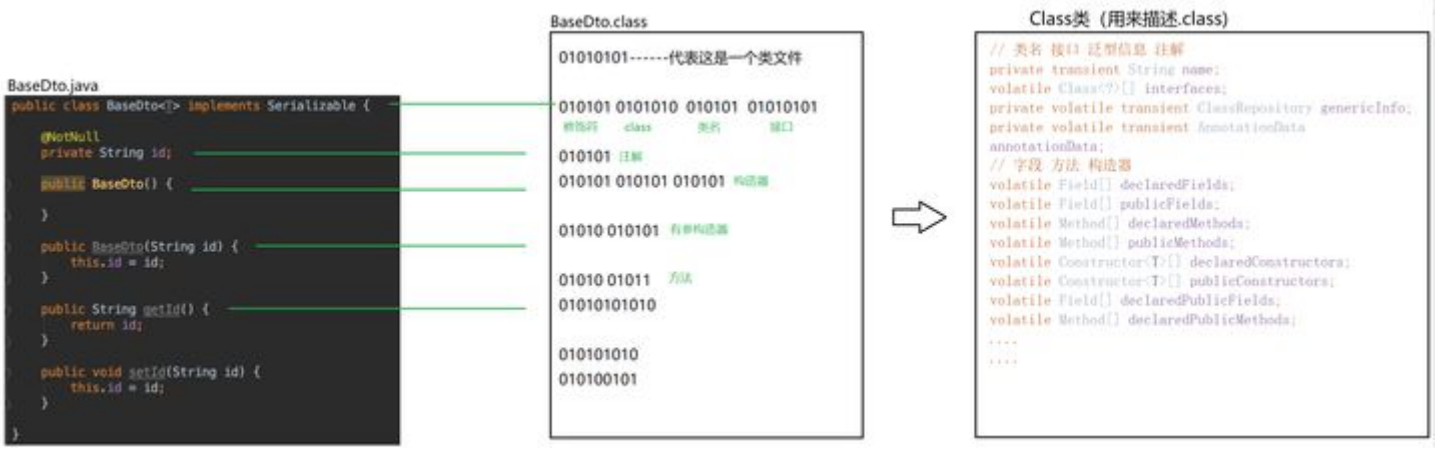
- 权限修饰符
- 类名
- 参数化类型（泛型信息）
- 接口
- 注解
- 字段（重点）
- 构造器（重点）
- 方法（重点）

最终这些信息在.class文件中都会以0101表示：



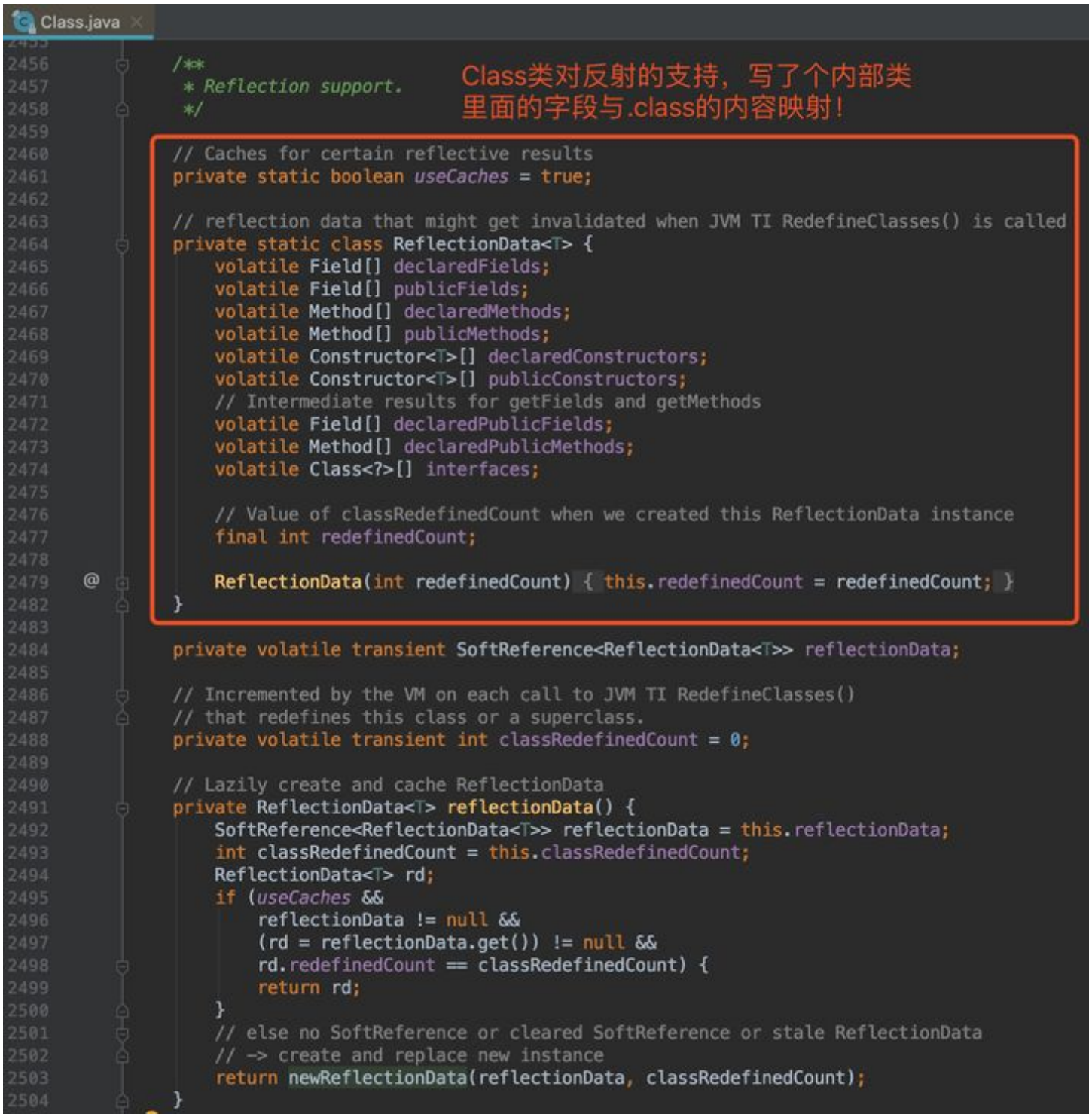
整个.class文件最终都成为字节数组byte[] b，里面的构造器、方法等各个“组件”，其实也是字节。

所以，我猜Class类的字段至少是这样的：



好了，看一下源码是不是如我所料：

字段、方法、构造器对象：



注解数据:

```
// Annotations cache
/UnusedDeclaration/
private volatile transient AnnotationData annotationData;
```

泛型信息:

```
// Generic info repository; lazily initialized
private volatile transient ClassRepository genericInfo;
```

而且，针对字段、方法、构造器，因为信息量太大了，JDK还单独写了三个类，比如Method类：

```
public final class Method extends Executable {
    private Class<?>            clazz;
    private int                 slot;
    // This is guaranteed to be interned by the VM in the 1.4
    // reflection implementation
    private String              name;
    private Class<?>            returnType;
    private Class<?>[]          parameterTypes;
    private Class<?>[]          exceptionTypes;
    private int                 modifiers;
    // Generics and annotations support
    private transient String              signature;
    // generic info repository; lazily initialized
    private transient MethodRepository genericInfo;
    private byte[]              annotations;
    private byte[]              parameterAnnotations;
    private byte[]              annotationDefault;
    private volatile MethodAccessor methodAccessor;
```

方法名、返回值类型

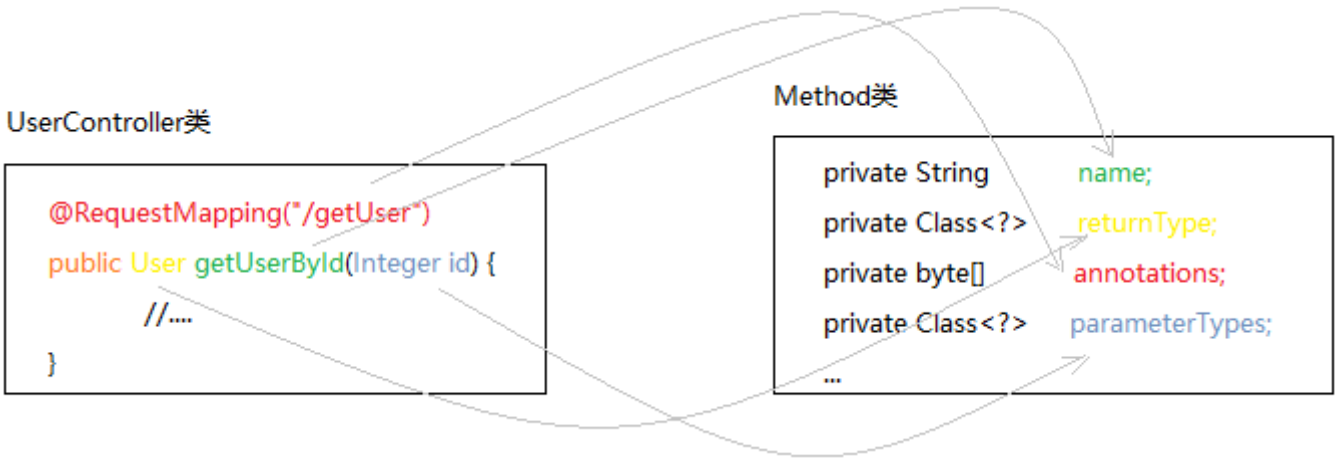
参数类型、异常类型

修饰符

方法泛型

方法上的注解

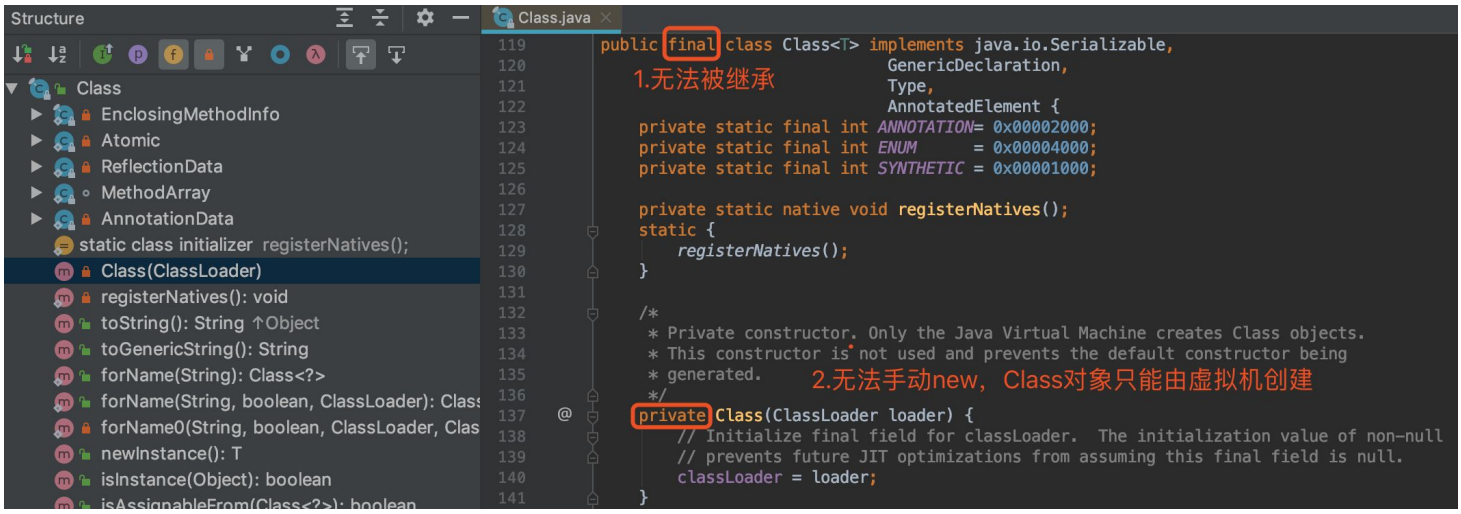
也就是说，Class类准备了很多字段用来表示一个.class文件的信息，对于字段、方法、构造器等，为了更详细地描述这些重要信息，还写了三个类，每个类里面都有很详细的对应。



也就是说，原本UserController类中所有信息，都被“解构”后保存在Class类、Method类等的字段中。

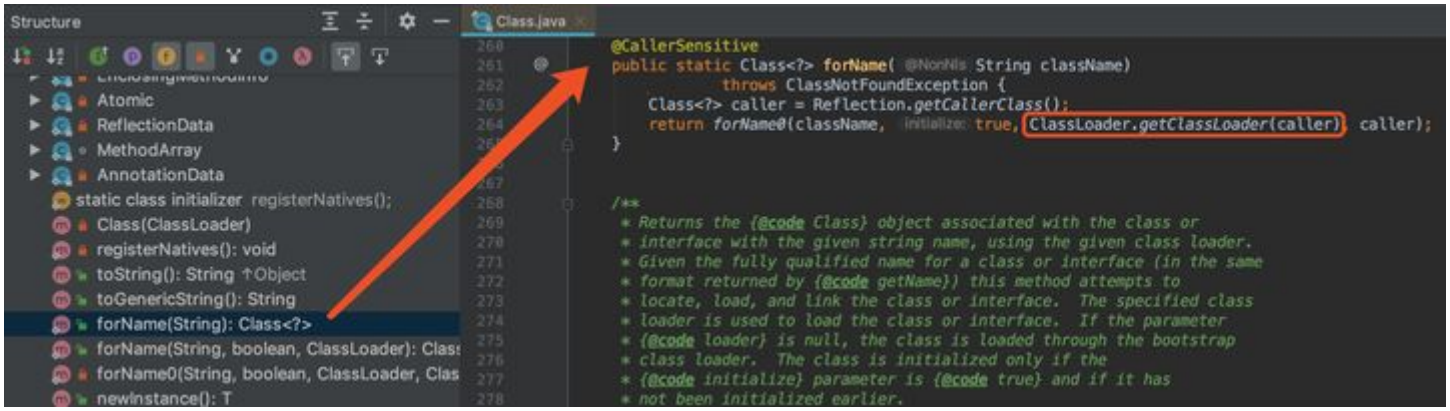
大概了解完Class类的字段后，我们看看Class类的方法。

构造器



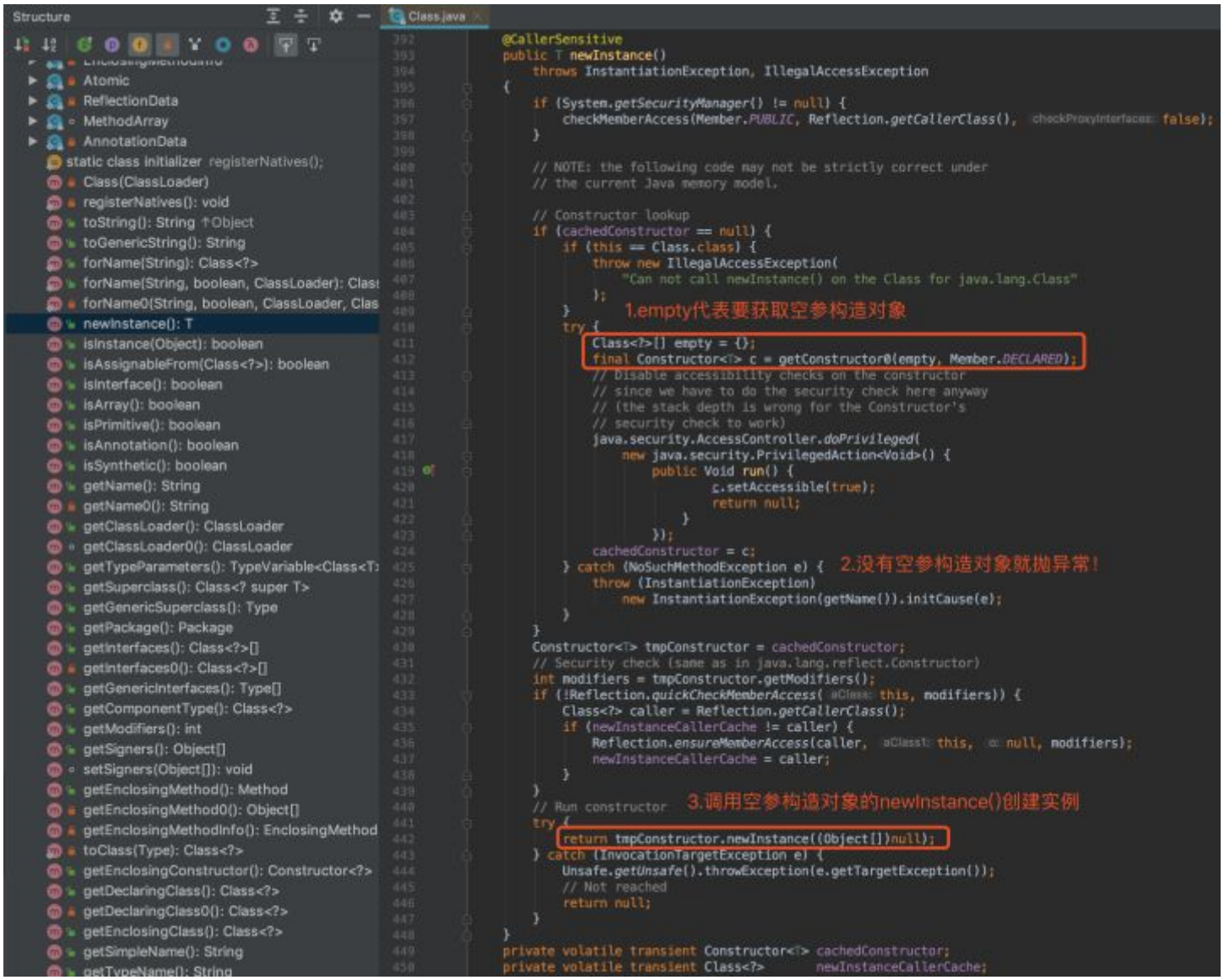
可以发现，Class类的构造器是私有的，我们无法手动new一个Class对象，只能由JVM创建。JVM在构造Class对象时，需要传入一个类加载器，然后才有我们上面分析的一连串加载、创建过程。

Class.forName()方法



反正还是类加载器去搞呗。

newInstance()



也就是说，newInstance()底层就是调用无参构造对象的newInstance()。

所以，本质上Class对象要想创建实例，其实都是通过构造器对象。如果没有空参构造对象，就无法使用clazz.newInstance()，必须要获取其他有参的构造对象然后调用构造对象的newInstance()。

1.5.4.反射API

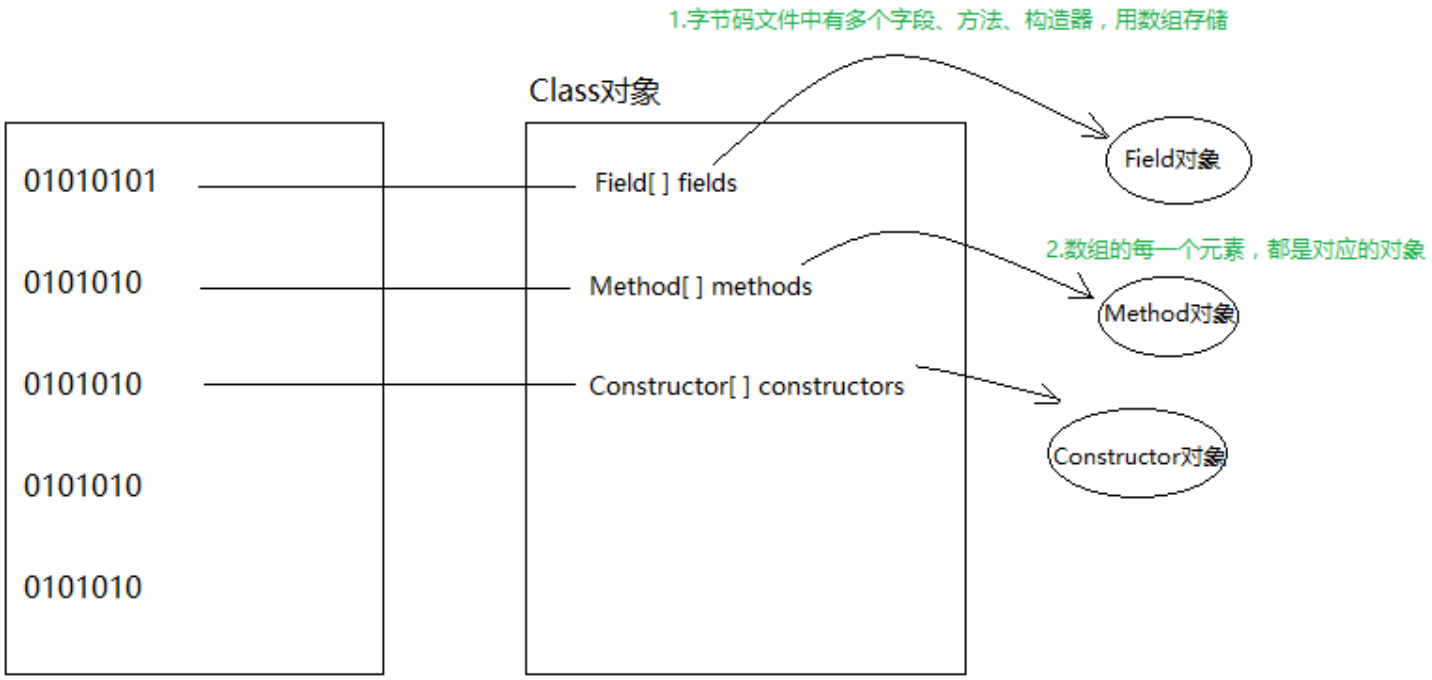
没啥好说的，在日常开发中反射最终目的主要两个：

- 创建实例
- 反射调用方法

创建实例的难点在于，很多人不知道clazz.newInstance()底层还是调用Contructor对象的newInstance()。所以，要想调用clazz.newInstance()，必须保证编写类的时候有个无参构造。

反射调用方法的难点，有两个，初学者可能会不理解。

再此之前，先来理清楚Class、Field、Method、Constructor四个对象的关系：



Field、Method、Constructor对象内部有对字段、方法、构造器更详细的描述：

```
public final class Method extends Executable {
    private Class<?>            clazz;
    private int                 slot;
    // This is guaranteed to be interned by the VM in the 1.4
    // reflection implementation
    private String              name;
    private Class<?>            returnType;
    private Class<?>[]          parameterTypes;
    private Class<?>[]          exceptionTypes;
    private int                 modifiers;
    // Generics and annotations support
    private transient String              signature;
    // generic info repository; lazily initialized
    private transient MethodRepository genericInfo;
    private byte[]              annotations;
    private byte[]              parameterAnnotations;
    private byte[]              annotationDefault;
    private volatile MethodAccessor methodAccessor;
```

方法名、返回值类型

参数类型、异常类型

修饰符

方法泛型

方法上的注解

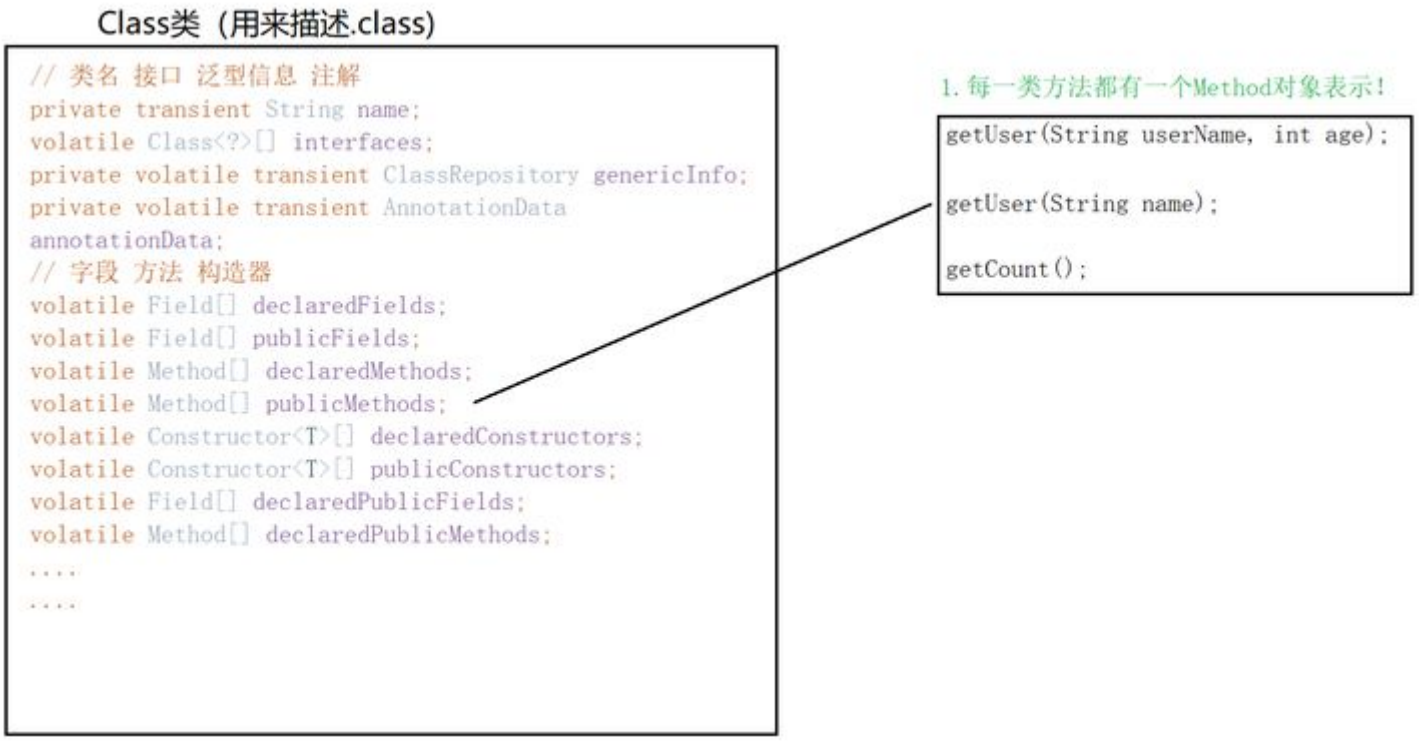
- 难点一：为什么根据Class对象获取Method时，需要传入方法名+参数的Class类型

```
public class Test {
    public static void main(String[] args) {
        System.out.println("hello, bravo!");

        Class<BaseDto> baseDtoClass = BaseDto.class;
        baseDtoClass.getm
    }
}
```

```
getMethod(String name, Class<?>... parameterTypes)          Method
getMethods()                                                Method[]
getModifiers()                                              int
getDeclaredMethod(String name, Class<?>... parameterTyp…    Method
getDeclaredMethods()                                        Method[]
getEnclosingMethod()                                        Method
Press ^. to choose the selected (or first) suggestion and insert a dot afterwards >>   π
```

为什么要传name和ParameterType?

因为.class文件中有多个方法，比如



所以必须传入name，以方法名区分哪个方法，得到对应的Method。

那参数parameterTypes为什么要用Class类型，我想和调用方法时一样直接传变量名不行吗，比如userName， age。

答案是：我们无法根据变量名区分方法

```
User getUser(String userName, int age);
User getUser(String mingzi, int nianling);
```

这不叫重载，这就是同一个方法。只能根据参数类型。

我知道，你还会问：变量名不行，那我能不能传String, int。

不好意思，这些都是基本类型和引用类型，类型不能用来传递。我们能传递的要么值，要么对象（引用）。而String.class, int.class是对象，且是Class对象。

实际上，调用Class对象的getMethod()方法时，内部会循环遍历所有Method，然后根据方法名和参数类型匹配唯一的Method返回。

循环遍历所有Method，根据name和parameterType匹配

```
private static Method searchMethods(Method[] methods,
                                    String name,
                                    Class<?>[] parameterTypes)
{
    Method res = null;
    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));
}
```

- 难点二：调用method.invoke(obj, args);时为什么要传入一个目标对象？

上面分析过，.class文件通过IO被加载到内存后，JDK创造了至少四个对象：Class、Field、Method、Constructor，这些对象其实都是0101010的抽象表示。

以Method对象为例，它到底是什么，怎么来的？我们上面已经分析过，Method对象有好多字段，比如name（方法名），returnType（返回值类型）等。也就是说我们在.java文件中写的方法，被“解构”以后存入了Method对象中。所以对象本身是一个方法的映射，一个方法对应一个Method对象。

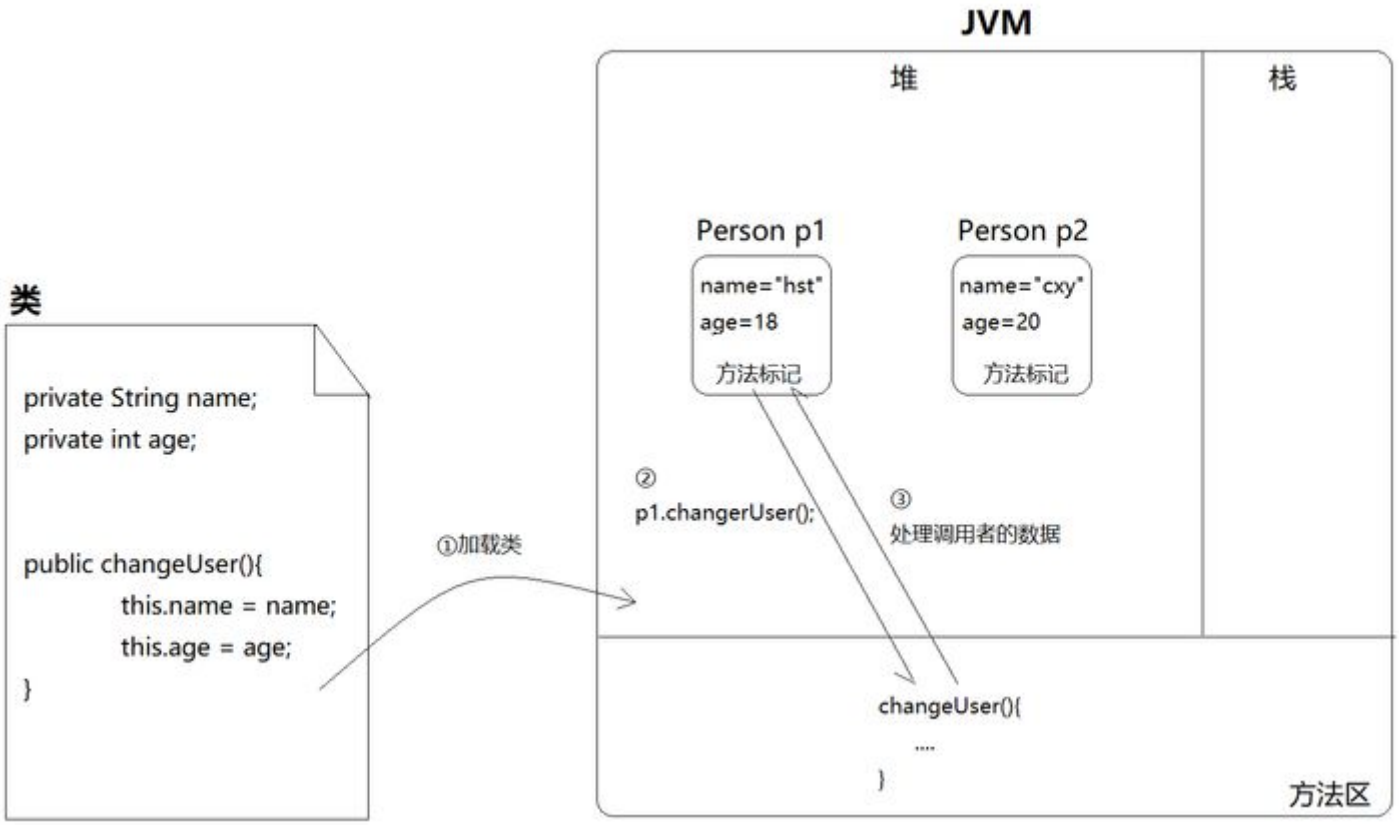
对象的本质就是用来存储数据的。而方法作为一种行为描述，是所有对象共有的，不属于某个对象独有。比如现有两个Person实例

```
Person p1 = new Person();
Person p2 = new Person();
```

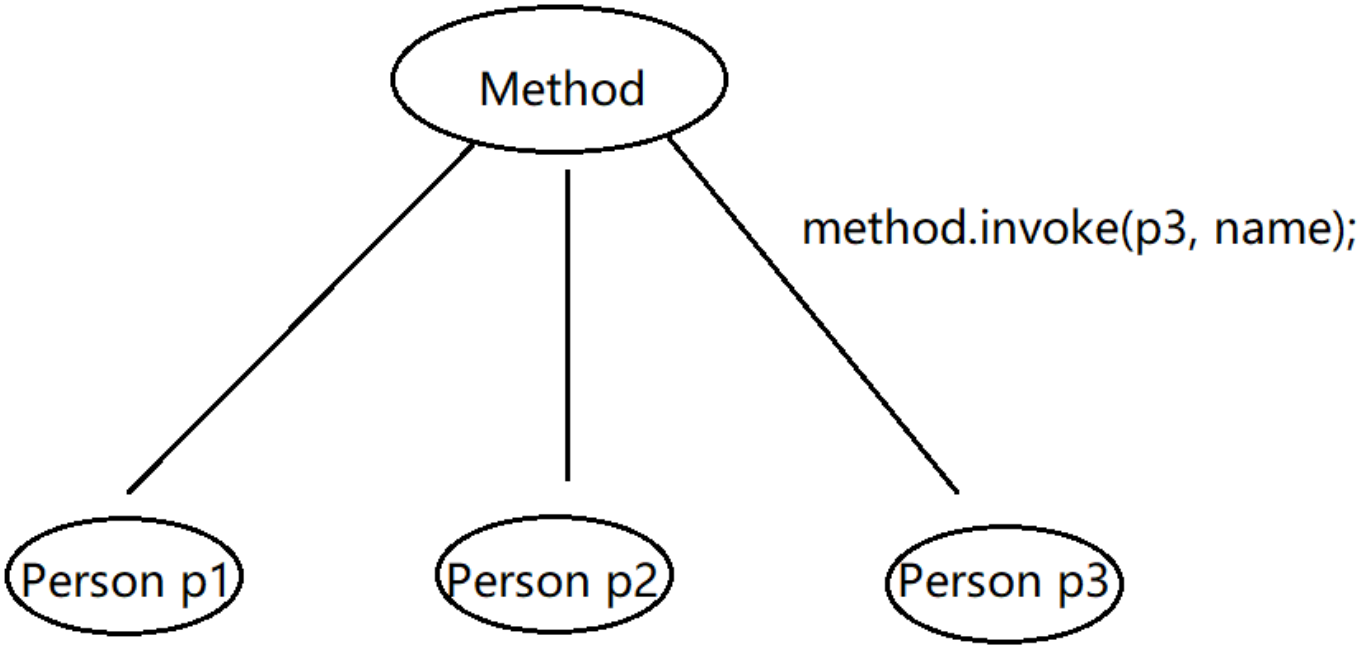
对象 p1保存了"hst"和18，p2保存了"cxy"和20。但是不管是p1还是p2，都会有changeUser()，但是每个对象里面写一份太浪费。既然是共性行为，可以抽取出来，放在方法区共用。

但这又产生了一个棘手的问题，方法是共用的，JVM如何保证p1调用changeUser()时，changeUser()不会跑去把p2的数据改掉呢?

所以JVM设置了一种隐性机制，每次对象调用方法时，都会隐性传递当前调用该方法的对象参数，方法可以根据这个对象参数知道当前调用本方法的是哪个对象!



同样的，在反射调用方法时，本质还是希望方法处理数据，所以必须告诉它执行哪个对象的数据。



所以，把Method理解为方法执行指令吧，它更像是一个方法执行器，必须告诉它要执行的对象（数据）。

当然，如果是invoke一个静态方法，不需要传入具体的对象。因为静态方法并不能处理对象中保存的数据。