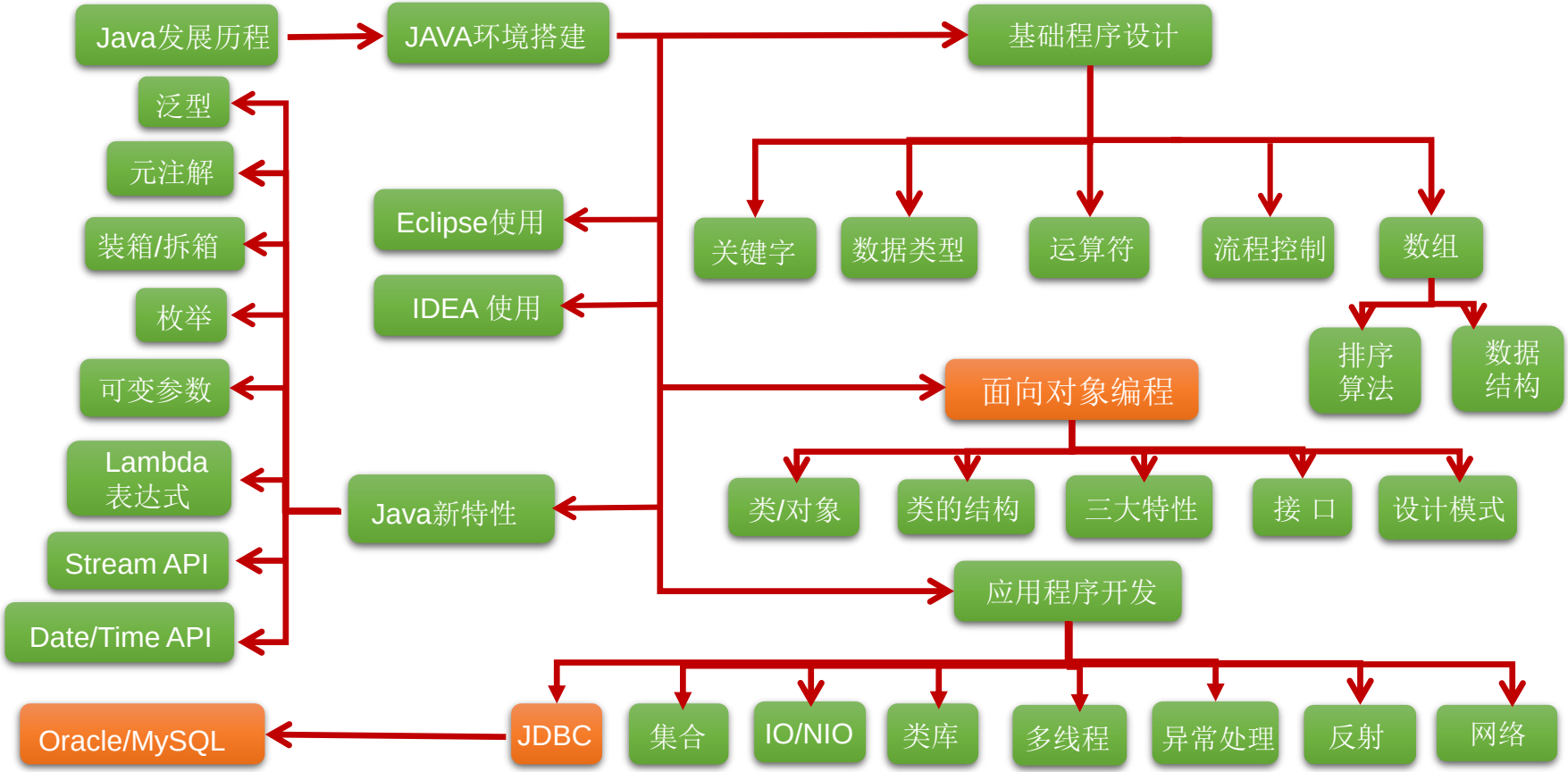
第15章
Java反射机制

讲师：宋红康
新浪微博：尚硅谷-宋红康

Java基础知识图解

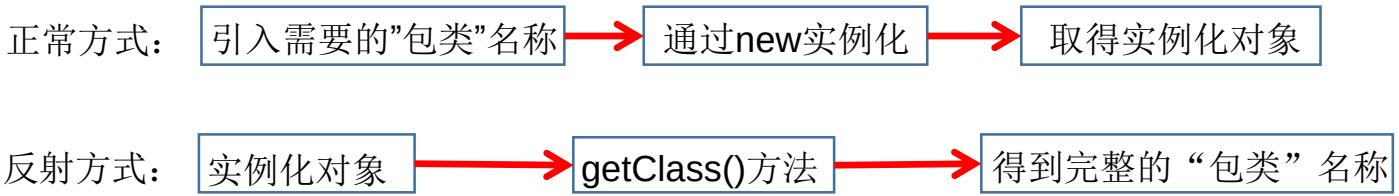


目录

15-1 Java反射机制概述

Java Reflection

- Reflection（反射）是被视为动态语言的关键，反射机制允许程序在执行期借助于Reflection API取得任何类的内部信息，并能直接操作任意对象的内部属性及方法。
- 加载完类之后，在堆内存的方法区中就产生了一个Class类型的对象（一个类只有一个Class对象），这个对象就包含了完整的类的结构信息。我们可以通过这个对象看到类的结构。这个对象就像一面镜子，透过这个镜子看到类的结构，所以，我们形象的称之为：**反射**。



补充：动态语言 vs 静态语言

1、动态语言

是一类在运行时可以改变其结构的语言：例如新的函数、对象、甚至代码可以被引进，已有的函数可以被删除或是其他结构上的变化。通俗点说就是在运行时代码可以根据某些条件改变自身结构。

主要动态语言：Object-C、C#、JavaScript、PHP、Python、Erlang。

2、静态语言

与动态语言相对应的，运行时结构不可变的语言就是静态语言。如Java、C、C++。

Java不是动态语言，但Java可以称之为“准动态语言”。即Java有一定的动态性，我们可以利用反射机制、字节码操作获得类似动态语言的特性。Java的动态性让编程的时候更加灵活！

Java反射机制研究及应用

- **Java反射机制提供的功能**
  - 在运行时判断任意一个对象所属的类
  - 在运行时构造任意一个类的对象
  - 在运行时判断任意一个类所具有的成员变量和方法
  - 在运行时获取泛型信息
  - 在运行时调用任意一个对象的成员变量和方法
  - 在运行时处理注解
  - 生成动态代理

反射相关的主要API

- java.lang.Class:代表一个类
- java.lang.reflect.Method:代表类的方法
- java.lang.reflect.Field:代表类的成员变量
- java.lang.reflect.Constructor:代表类的构造器
- … …

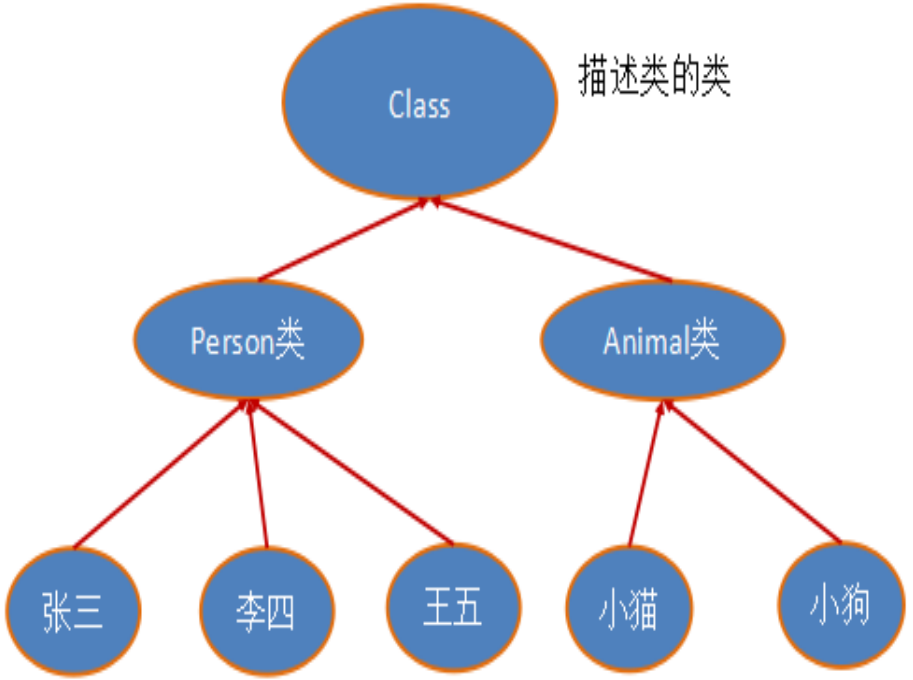
15-2 理解Class类并获取Class的实例

Class 类

在Object类中定义了以下的方法，此方法将被所有子类继承：

- public final Class getClass()

以上的方法返回值的类型是一个Class类，此类是Java反射的源头，实际上所谓反射从程序的运行结果来看也很好理解，即：可以通过对象反射求出类的名称。



Class 类

- 对象照镜子后可以得到的信息：某个类的属性、方法和构造器、某个类到底实现了哪些接口。对于每个类而言，JRE 都为其保留一个不变的 Class 类型的对象。一个 Class 对象包含了特定某个结构(class/interface/enum/annotation/primitive type/void/[])的有关信息。
  - Class本身也是一个类
  - Class 对象只能由系统建立对象
  - 一个加载的类在 JVM 中只会有一个Class实例
  - 一个Class对象对应的是一个加载到JVM中的一个.class文件
  - 每个类的实例都会记得自己是由哪个 Class 实例所生成
  - 通过Class可以完整地得到一个类中的所有被加载的结构
  - Class类是Reflection的根源，针对任何你想动态加载、运行的类，唯有先获得相应的Class对象

Class类的常用方法

方法名	功能说明
static Class forName(String name)	返回指定类名 name 的 Class 对象
Object newInstance()	调用缺省构造函数，返回该Class对象的一个实例
getName()	返回此Class对象所表示的实体（类、接口、数组类、基本类型或void）名称
Class getSuperClass()	返回当前Class对象的父类的Class对象
Class [] getInterfaces()	获取当前Class对象的接口
ClassLoader getClassLoader()	返回该类的类加载器
Class getSuperclass()	返回表示此Class所表示的实体的超类的Class
Constructor[] getConstructors()	返回一个包含某些Constructor对象的数组
Field[] getDeclaredFields()	返回Field对象的一个数组
Method getMethod(String name,Class … paramTypes)	返回一个Method对象，此对象的形参类型为paramType

反射的应用举例

- String str = "test4.Person";
- Class clazz = Class.forName(str);
- Object obj = clazz.newInstance();
- Field field = clazz.getField("name");
- field.set(obj, "Peter");
- Object name = field.get(obj);
- System.out.println(name);

注：test4.Person是test4包下的Person类

获取Class类的实例(四种方法)

1）**前提：**若已知具体的类，通过类的class属性获取，该方法最为安全可靠，程序性能最高

实例：Class clazz = String.class;

2）**前提：**已知某个类的实例，调用该实例的getClass()方法获取Class对象

实例：Class clazz = “www.atguigu.com”.getClass();

3）**前提：**已知一个类的全类名，且该类在类路径下，可通过Class类的静态方法forName()获取，可能抛出ClassNotFoundException

实例：Class clazz = Class.forName(“java.lang.String”);

4）其他方式(不做要求)

```
ClassLoader cl = this.getClass().getClassLoader();
Class clazz4 = cl.loadClass(“类的全类名”);
```

哪些类型可以有Class对象？

（1）class：
外部类，成员(成员内部类，静态内部类)，局部内部类，匿名内部类
（2）interface：接口
（3）[]：数组
（4）enum：枚举
（5）annotation：注解@interface
（6）primitive type：基本数据类型
（7）void

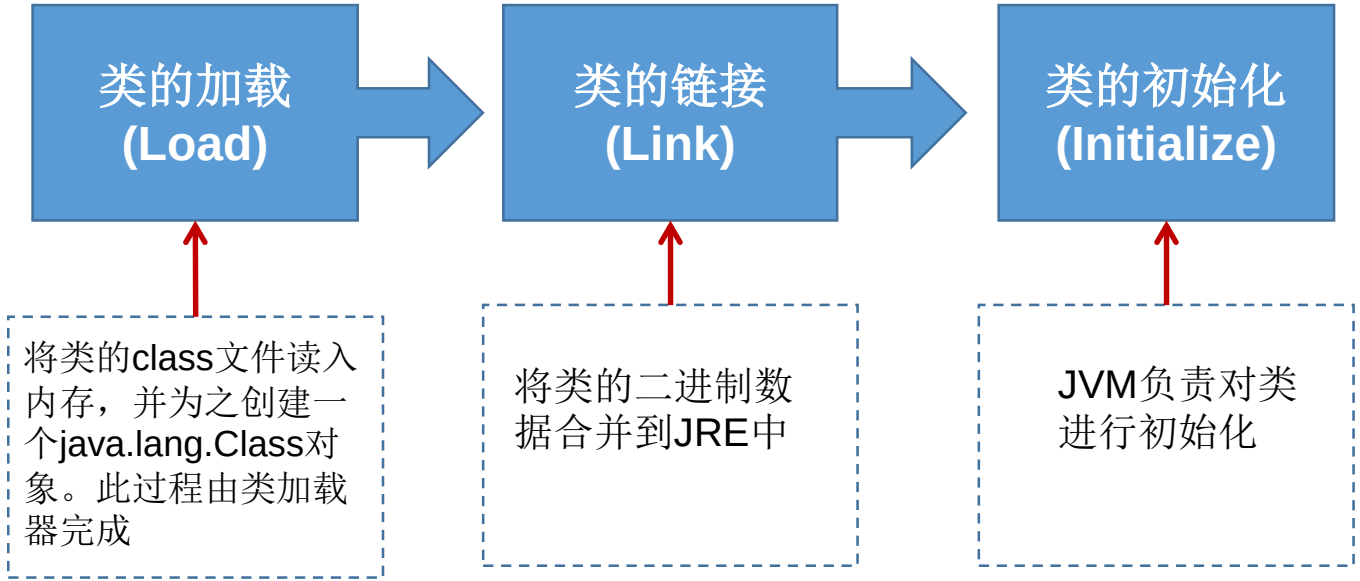
15.2 理解Class类并获取Class的实例


```
Class c1 = Object.class;
Class c2 = Comparable.class;
Class c3 = String[].class;
Class c4 = int[][].class;
Class c5 = ElementType.class;
Class c6 = Override.class;
Class c7 = int.class;
Class c8 = void.class;
Class c9 = Class.class;

int[] a = new int[10];
int[] b = new int[100];
Class c10 = a.getClass();
Class c11 = b.getClass();
// 只要元素类型与维度一样，就是同一个Class
System.out.println(c10 == c11);
```

15-3 类的加载与ClassLoader的理解

了解：类的加载过程

当程序主动使用某个类时，如果该类还未被加载到内存中，则系统会通过如下三个步骤来对该类进行初始化。



15.3 类的加载与ClassLoader的理解

- 加载：将class文件字节码内容加载到内存中，并将这些静态数据转换成方法区的运行时数据结构，然后生成一个代表这个类的java.lang.Class对象，作为方法区中类数据的访问入口（即引用地址）。所有需要访问和使用类数据只能通过这个Class对象。这个加载的过程需要类加载器参与。

- 链接：将Java类的二进制代码合并到JVM的运行状态之中的过程。
  - 验证：确保加载的类信息符合JVM规范，例如：以cafe开头，没有安全方面的问题
  - 准备：正式为类变量（static）分配内存并**设置类变量默认初始值**的阶段，这些内存都将在方法区中进行分配。
  - 解析：虚拟机常量池内的符号引用（常量名）替换为直接引用（地址）的过程。

- 初始化：
  - 执行类构造器<clinit>()方法的过程。类构造器<clinit>()方法是由编译期自动收集类中所有类变量的赋值动作和静态代码块中的语句合并产生的。（类构造器是构造类信息的，不是构造该类对象的构造器）。
  - 当初始化一个类的时候，如果发现其父类还没有进行初始化，则需要先触发其父类的初始化。
  - 虚拟机会保证一个类的<clinit>()方法在多线程环境中被正确加锁和同步。

15.3 类的加载与ClassLoader的理解

```
public class ClassLoadingTest {
    public static void main(String[] args) {
        System.out.println(A.m);
    }
}

class A {
    static {
        m = 300;
    }
    static int m = 100;
}
//第二步：链接结束后m=0
//第三步：初始化后，m的值由<clinit>()方法执行决定
//        这个A的类构造器<clinit>()方法由类变量的赋值和静态代码块中的语句按照顺序合并产生，类似于
//        <clinit>(){
//            m = 300;
//            m = 100;
//        }
```

了解：什么时候会发生类初始化？

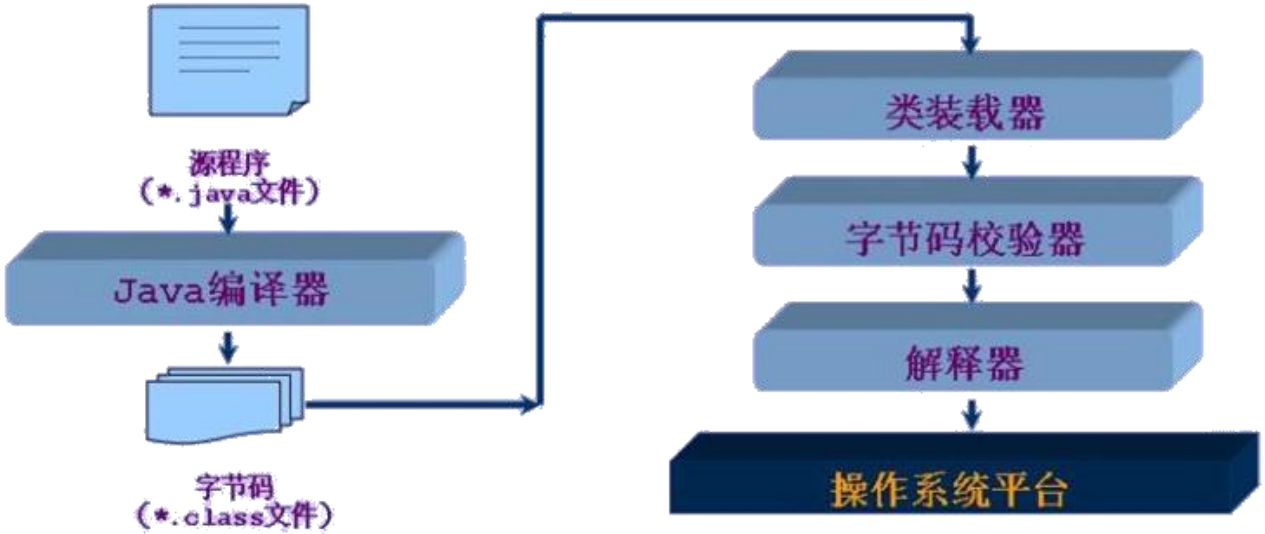
- 类的主动引用（一定会发生类的初始化）
  - 当虚拟机启动，先初始化main方法所在的类
  - new一个类的对象
  - 调用类的静态成员（除了final常量）和静态方法
  - 使用java.lang.reflect包的方法对类进行反射调用
  - 当初始化一个类，如果其父类没有被初始化，则先会初始化它的父类
- 类的被动引用（不会发生类的初始化）
  - 当访问一个静态域时，只有真正声明这个域的类才会被初始化
    - ✓ 当通过子类引用父类的静态变量，不会导致子类初始化
  - 通过数组定义类引用，不会触发此类的初始化
  - 引用常量不会触发此类的初始化（常量在链接阶段就存入调用类的常量池中了）

15.3 类的加载与ClassLoader的理解

```
public class ClassLoadingTest {
    public static void main(String[] args) {
    // 主动引用：一定会导致A和Father的初始化
    // A a = new A();
    // System.out.println(A.m);
    // Class.forName("com.atguigu.java2.A");

    // 被动引用
     A[] array = new A[5];//不会导致A和Father的初始化
    // System.out.println(A.b);//只会初始化Father
    // System.out.println(A.M);//不会导致A和Father的初始化
}

    static {
        System.out.println("main所在的类");
    }

}
```

```
class Father {
    static int b = 2;
    static {
        System.out.println("父类被加载");
    }
}

class A extends Father {
    static {
        System.out.println("子类被加载");
        m = 300;
    }
    static int m = 100;
    static final int M = 1;
}
```

15.3 类的加载与ClassLoader的理解

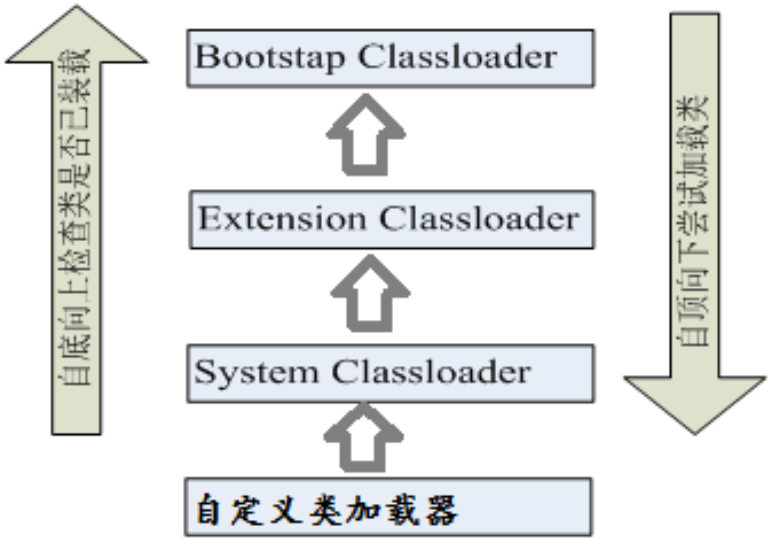


类加载器的作用：

- **类加载的作用：**将class文件字节码内容加载到内存中，并将这些静态数据转换成方法区的运行时数据结构，然后在堆中生成一个代表这个类的java.lang.Class对象，作为方法区中类数据的访问入口。
- **类缓存：**标准的JavaSE类加载器可以按要求查找类，但一旦某个类被加载到类加载器中，它将维持加载（缓存）一段时间。不过JVM垃圾回收机制可以回收这些Class对象。

了解：ClassLoader

类加载器作用是用来把类(class)装载进内存的。JVM 规范定义了如下类型的类的加载器。



引导类加载器：用C++编写的，是JVM自带的类加载器，负责Java平台核心库，用来装载核心类库。该加载器无法直接获取

扩展类加载器：负责jre/lib/ext目录下的jar包或 –D java.ext.dirs 指定目录下的jar包装入工作库

系统类加载器：负责java –classpath 或 –D java.class.path所指的目录下的类与jar包装入工作，是最常用的加载器

15.3 类的加载与ClassLoader的理解


```
//1.获取一个系统类加载器
ClassLoader classloader = ClassLoader.getSystemClassLoader();
System.out.println(classloader);
//2.获取系统类加载器的父类加载器，即扩展类加载器
classloader = classloader.getParent();
System.out.println(classloader);
//3.获取扩展类加载器的父类加载器，即引导类加载器
classloader = classloader.getParent();
System.out.println(classloader);
//4.测试当前类由哪个类加载器进行加载
classloader = Class.forName("exer2.ClassloaderDemo").getClassLoader();
System.out.println(classloader);
```

- //5.测试JDK提供的Object类由哪个类加载器加载
- **classloader =**
- **Class.forName("java.lang.Object").getClassLoader();**
- **System.out.println(classloader);**
- //*6.关于类加载器的一个主要方法：getResourceAsStream(String str):获取类路径下的指定文件的输入流
- **InputStream in = null;**
- **in = this.getClass().getClassLoader().getResourceAsStream("exer2\\test.properties");**
- **System.out.println(in);**

15-4 创建运行时类的对象

有了Class对象，能做什么？

创建类的对象：调用Class对象的newInstance()方法

要 求：　1）类必须有一个无参数的构造器。

　　2）类的构造器的访问权限需要足够。

难道没有无参的构造器就不能创建对象了吗？

不是！只要在操作的时候明确的调用类中的构造器，并将参数传递进去之后，才可以实例化操作。

步骤如下：

1）通过Class类的**getDeclaredConstructor(Class … parameterTypes)**取得本类的指定形参类型的构造器

2）向构造器的形参中传递一个对象数组进去，里面包含了构造器中所需的各个参数。

3）通过Constructor实例化对象。

在 Constructor 类中存在一个方法：

```
public T newInstance(Object... initargs)
```

以上是反射机制应用最多的地方。

```
//1.根据全类名获取对应的Class对象
String name = “atguigu.java.Person";
Class clazz = null;
clazz = Class.forName(name);
//2.调用指定参数结构的构造器，生成Constructor的实例
Constructor con = clazz.getConstructor(String.class,Integer.class);
//3.通过Constructor的实例创建对应类的对象，并初始化类属性
Person p2 = (Person) con.newInstance("Peter",20);
System.out.println(p2);
```

15-5 获取运行时类的完整结构

通过反射获取运行时类的完整结构

Field、Method、Constructor、Superclass、Interface、Annotation

- 实现的全部接口
- 所继承的父类
- 全部的构造器
- 全部的方法
- 全部的Field

使用反射可以取得：

1.实现的全部接口

- public Class<?>[] getInterfaces()

确定此对象所表示的类或接口实现的接口。

2.所继承的父类

- public Class<? Super T> getSuperclass()

返回表示此 Class 所表示的实体（类、接口、基本类型）的父类的 Class。

3.全部的构造器

- public Constructor<T>[] getConstructors()

返回此 Class 对象所表示的类的所有public构造方法。

- public Constructor<T>[] getDeclaredConstructors()

返回此 Class 对象表示的类声明的所有构造方法。

- Constructor类中：
  - 取得修饰符: public int getModifiers();
  - 取得方法名称: public String getName();
  - 取得参数的类型：public Class<?>[] getParameterTypes();

4.全部的方法

- public Method[] getDeclaredMethods()

返回此Class对象所表示的类或接口的全部方法

- public Method[] getMethods()

返回此Class对象所表示的类或接口的public的方法

- Method类中：
  - public Class<?> getReturnType()取得全部的返回值
  - public Class<?>[] getParameterTypes()取得全部的参数
  - public int getModifiers()取得修饰符
  - public Class<?>[] getExceptionTypes()取得异常信息

5.全部的Field

- public Field[] getFields()

返回此Class对象所表示的类或接口的public的Field。

- public Field[] getDeclaredFields()

返回此Class对象所表示的类或接口的全部Field。

- Field方法中：
- public int getModifiers() 以整数形式返回此Field的修饰符
- public Class<?> getType() 得到Field的属性类型
- public String getName() 返回Field的名称。

6. Annotation相关

- get Annotation(Class<T> annotationClass)
- getDeclaredAnnotations()

7.泛型相关

获取父类泛型类型：**Type getGenericSuperclass()**
泛型类型：**ParameterizedType**
获取实际的泛型类型参数数组：**getActualTypeArguments()**

8.类所在的包　**Package getPackage()**

小 结：

1.在实际的操作中，取得类的信息的操作代码，并不会经常开发。

2.一定要熟悉java.lang.reflect包的作用，反射机制。

3.如何取得属性、方法、构造器的名称，修饰符等。

15-6 调用运行时类的指定结构

1.调用指定方法

通过反射，调用类中的方法，通过Method类完成。步骤：

1.通过Class类的**getMethod(String name,Class…parameterTypes)**方法取得一个Method对象，并设置此方法操作时所需要的参数类型。

2.之后使用**Object invoke(Object obj, Object[] args)**进行调用，并向方法中传递要设置的obj对象的参数信息。

Object invoke(Object obj, Object … args)

说明：

1.Object 对应原方法的返回值，若原方法无返回值，此时返回null

2.若原方法若为静态方法，此时形参Object obj可为null

3.若原方法形参列表为空，则Object[] args为null

4.若原方法声明为private,则需要在调用此invoke()方法前，显式调用方法对象的setAccessible(true)方法，将可访问private的方法。

2.调用指定属性

在反射机制中，可以直接通过Field类操作类中的属性，通过Field类提供的set()和get()方法就可以完成设置和取得属性内容的操作。

- public Field getField(String name) 返回此Class对象表示的类或接口的指定的public的Field。
- public Field getDeclaredField(String name)返回此Class对象表示的类或接口的指定的Field。

- **在Field中：**
- public Object get(Object obj) 取得指定对象obj上此Field的属性内容
- public void set(Object obj,Object value) 设置指定对象obj上此Field的属性内容

关于setAccessible方法的使用

- Method和Field、Constructor对象都有setAccessible()方法。
- setAccessible启动和禁用访问安全检查的开关。
- 参数值为true则指示反射的对象在使用时应该取消Java语言访问检查。
  - 提高反射的效率。如果代码中必须用反射，而该句代码需要频繁的被调用，那么请设置为true。
  - 使得原本无法访问的私有成员也可以访问
- 参数值为false则指示反射的对象应该实施Java语言访问检查。

15-7 反射的应用：动态代理

●代理设计模式的原理:

使用一个代理将对象包装起来, 然后用该代理对象取代原始对象。任何对原始对象的调用都要通过代理。代理对象决定是否以及何时将方法调用转到原始对象上。

- 之前为大家讲解过代理机制的操作，属于静态代理，特征是代理类和目标对象的类都是在编译期间确定下来，不利于程序的扩展。同时，每一个代理类只能为一个接口服务，这样一来程序开发中必然产生过多的代理。**最好可以通过一个代理类完成全部的代理功能。**

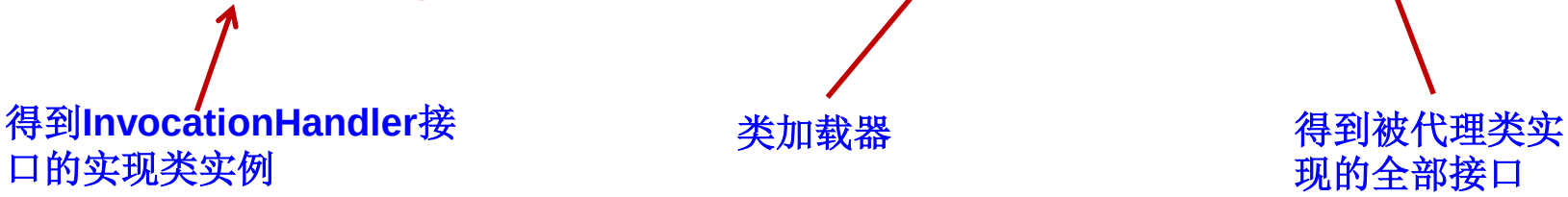
15.7 反射的应用：动态代理

- 动态代理是指客户通过代理类来调用其它对象的方法，并且是在程序运行时根据需要动态创建目标类的代理对象。
- 动态代理使用场合:
  - 调试
  - 远程方法调用
- **动态代理相比于静态代理的优点：**

抽象角色中（接口）声明的所有方法都被转移到调用处理器一个集中的方法中处理，这样，我们可以更加灵活和统一的处理众多的方法。

Java动态代理相关API

- **Proxy**：专门完成代理的操作类，是所有动态代理类的父类。通过此类为一个或多个接口动态地生成实现类。
- 提供用于创建动态代理类和动态代理对象的静态方法
  - **static Class<?> getProxyClass(ClassLoader loader, Class<?>... interfaces)** 创建一个动态代理类所对应的Class对象
  - **static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)** 直接创建一个动态代理对象



动态代理步骤

1.创建一个实现接口**InvocationHandler**的类，它必须实现invoke方法，以完成代理的具体操作。

```
public Object invoke(Object theProxy, Method method, Object[] params)
  throws Throwable{
      try{
          Object retval = method.invoke(targetObj, params);
          // Print out the result
          System.out.println(retval);
          return retval;
      }catch (Exception exc){}
}
```

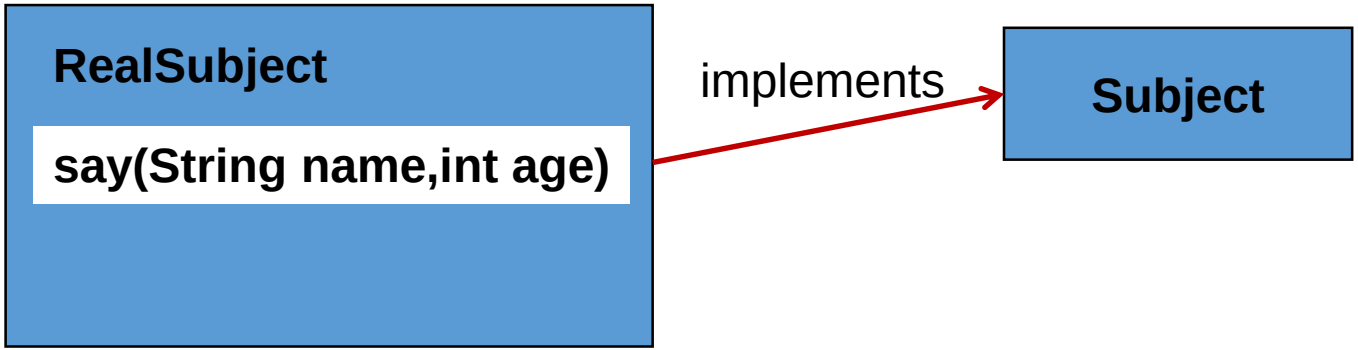
代理类的对象

要调用的方法

方法调用时所需要的参数

动态代理步骤

2.创建被代理的类以及接口



动态代理步骤

3.通过Proxy的静态方法

newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler h) 创建一个**Subject**接口代理

```
RealSubject target = new RealSubject();
// Create a proxy to wrap the original implementation
DebugProxy proxy = new DebugProxy(target);
 // Get a reference to the proxy through the Subject interface
Subject sub = (Subject) Proxy.newProxyInstance(
Subject.class.getClassLoader(),new Class[] { Subject.class }, proxy);
```

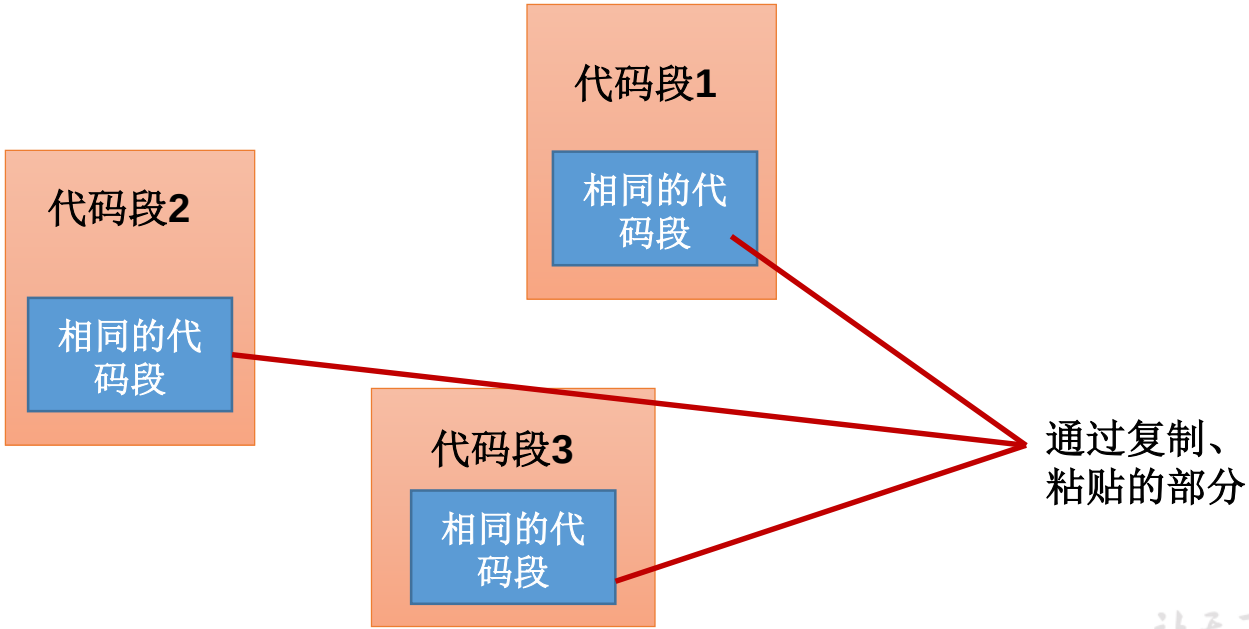
动态代理步骤

4.通过 Subject代理调用RealSubject实现类的方法

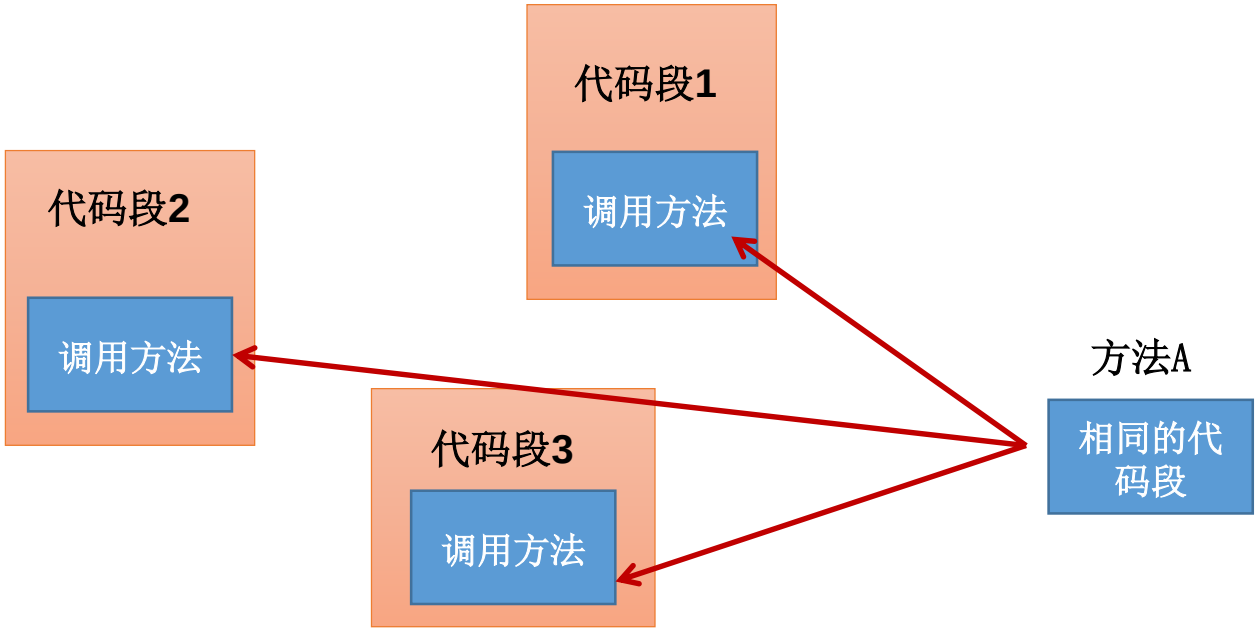
```
String info = sub.say(“Peter", 24);
System.out.println(info);
```

动态代理与AOP（Aspect Orient Programming)

前面介绍的Proxy和InvocationHandler，很难看出这种动态代理的优势，下面介绍一种更实用的动态代理机制



动态代理与AOP（Aspect Orient Programming)



改进后的说明：代码段1、代码段2、代码段3和深色代码段分离开了，但代码段1、2、3又和一个特定的方法A耦合了！最理想的效果是：代码块1、2、3既可以执行方法A，又无须在程序中以硬编码的方式直接调用深色代码的方法

动态代理与AOP（Aspect Orient Programming)

```
public interface Dog{
    void info();
    void run();
}
```

```
public class HuntingDog implements Dog{
        public void info(){
                System.out.println("我是一只猎狗");
        }
        public void run(){
                System.out.println("我奔跑迅速");
        }
}
```

动态代理与AOP（Aspect Orient Programming)

```
public class DogUtil{
        public void method1(){
                System.out.println("=====模拟通用方法一=====");
        }
        public void method2(){
                System.out.println("=====模拟通用方法二=====");
        }
}
```

动态代理与AOP（Aspect Orient Programming)

```
public class MyInvocationHandler implements InvocationHandler{
        // 需要被代理的对象
        private Object target;
        public void setTarget(Object target){
                this.target = target;}
// 执行动态代理对象的所有方法时，都会被替换成执行如下的invoke方法
        public Object invoke(Object proxy, Method method, Object[] args)
                throws Exception{
                DogUtil du = new DogUtil();
                // 执行DogUtil对象中的method1。
                du.method1();
                // 以target作为主调来执行method方法
                Object result = method.invoke(target , args);
                // 执行DogUtil对象中的method2。
                du.method2();
                return result;}}
```

动态代理与AOP（Aspect Orient Programming)

```
public class MyProxyFactory{
        // 为指定target生成动态代理对象
        public static Object getProxy(Object target)
                throws Exception{
                // 创建一个MyInvokationHandler对象
                MyInvokationHandler handler =
                new MyInvokationHandler();
                // 为MyInvokationHandler设置target对象
                handler.setTarget(target);
                // 创建、并返回一个动态代理对象
                return
Proxy.newProxyInstance(target.getClass().getClassLoader()
                        , target.getClass().getInterfaces() , handler);
        }
}
```

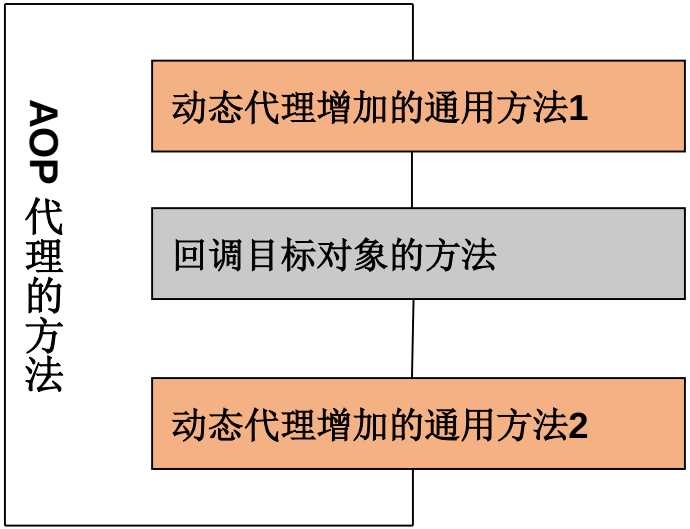
动态代理与AOP（Aspect Orient Programming)

```
public class Test{
        public static void main(String[] args)
                throws Exception{
                // 创建一个原始的HuntingDog对象，作为target
                Dog target = new HuntingDog();
                // 以指定的target来创建动态代理
                Dog dog = (Dog)MyProxyFactory.getProxy(target);
                dog.info();
                dog.run();
        }
}
```

动态代理与AOP（Aspect Orient Programming)

- 使用Proxy生成一个动态代理时，往往并不会凭空产生一个动态代理，这样没有太大的意义。通常都是为指定的目标对象生成动态代理
- 这种动态代理在AOP中被称为AOP代理，AOP代理可代替目标对象，AOP代理包含了目标对象的全部方法。但AOP代理中的方法与目标对象的方法存在差异：**AOP代理里的方法可以在执行目标方法之前、之后插入一些通用处理**

动态代理与AOP (Aspect Orient Programming)




尚硅谷
让天下没有难学的的技术