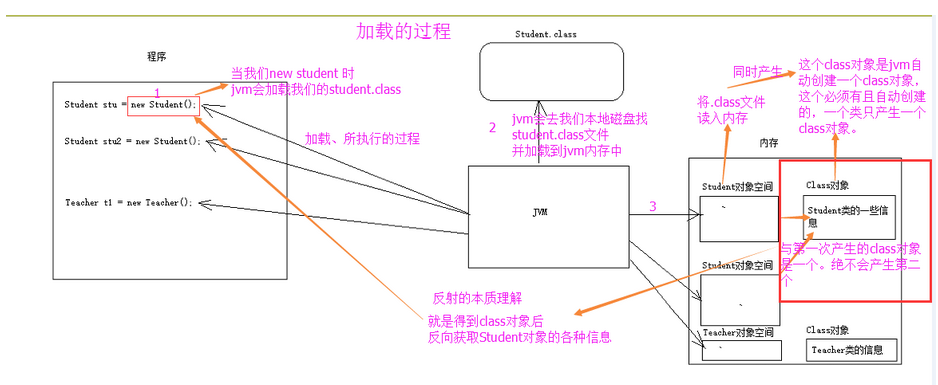
**反射机制**

## 一、基础概述

**1 Java反射机制**

在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；**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。**



**2 反射机制用途**

反射被广泛地用于那些需要在运行时检测或修改程序行为的程序中。这是一个相对高级的特性，只有那些语言基础非常扎实的开发者才应该使用它。如果能把这句警示时刻放在心里，那么反射机制就会成为一项强大的技术，可以让应用程序做一些几乎不可能做到的事情。

**3 反射机制缺点**

**（1）性能第一 Performance Overhead**

反射包括了一些动态类型，所以 JVM 无法对这些代码进行优化。因此，反射操作的效率要比那些非反射操作低得多。我们应该避免在经常被 执行的代码或对性能要求很高的程序中使用反射。

**（2）安全限制 Security Restrictions**

使用反射技术要求程序必须在一个没有安全限制的环境中运行。如果一个程序必须在有安全限制的环境中运行，如 Applet，那么这就是个问题了。

**（3）内部暴露 Exposure of Internals**

由于反射允许代码执行一些在正常情况下不被允许的操作（比如访问私有的属性和方法），所以使用反射可能会导致意料之外的副作用－－代码有功能上的错误，降低可移植性。反射代码破坏了抽象性，因此当平台发生改变的时候，代码的行为就有可能也随着变化。

## 二、技术详解

**1 关于Class**

@ Class是一个类，一个描述类的类（也就是描述类本身），封装了描述方法的Method，描述字段的Filed，描述构造器的Constructor等属性

@ 对象照镜子后（反射）可以得到的信息：某个类的数据成员名、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类的有关信息。

@ Class 对象只能由系统建立对象

@ 一个类在 JVM 中只会有一个Class实例

package com.java.reflection;

public class Person {

String name;

private int age;

public Person() {

System.out.println("无参构造器");

}

public Person(String name, int age) {

System.out.println("有参构造器");

this.name = name;

this.age = age;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

@Override

public String toString() {

return "Person{" +

"name='" + name + '\'' +

", age=" + age +

'}';

}

}

**2、反射机制获取类有三种方法**

@Test

public void testGetClass() throws ClassNotFoundException {

Class clazz = null;

//1 直接通过类名.Class的方式得到

clazz = Person.class;

System.out.println("通过类名: " + clazz);

//2 通过对象的getClass()方法获取,这个使用的少（一般是传的是Object，不知道是什么类型的时候才用）

Object obj = new Person();

clazz = obj.getClass();

System.out.println("通过getClass(): " + clazz);

//3 通过全类名获取，用的比较多，但可能抛出ClassNotFoundException异常

clazz = Class.forName("com.java.reflection.Person");

System.out.println("通过全类名获取: " + clazz);

}

通过类名: class com.java.reflection.Person

无参构造器

通过getClass(): class com.java.reflection.Person

通过全类名获取: class com.java.reflection.Person

**3、利用newInstance创建对象：调用的类必须有无参的构造器**

@Test

public void testNewInstance()

throws ClassNotFoundException, IllegalAccessException, InstantiationException {

Class clazz = Class.forName("com.java.reflection.Person");

//使用Class类的newInstance()方法创建类的一个对象

//实际调用的类的那个 无参数的构造器（这就是为什么写的类的时候，要写一个无参数的构造器，就是给反射用的）

//一般的，一个类若声明了带参数的构造器，也要声明一个无参数的构造器

Object obj = clazz.newInstance();

System.out.println(obj);

}

无参构造器

Person{name='null', age=0}

**4、ClassLoader类加载器**

类加载器详解：http://blog.csdn.net/ochangwen/article/details/51473120

/\*\*

\* ClassLoader类装载器

\*/

@Test

public void testClassLoader1() throws ClassNotFoundException, IOException {

//1、获取一个系统的类加载器

ClassLoader classLoader = ClassLoader.getSystemClassLoader();

System.out.println("系统的类加载器-->" + classLoader);

//2、获取系统类加载器的父类加载器(扩展类加载器（extensions classLoader）)

classLoader = classLoader.getParent();

System.out.println("扩展类加载器-->" + classLoader);

//3、获取扩展类加载器的父类加载器

//输出为Null,无法被Java程序直接引用

classLoader = classLoader.getParent();

System.out.println("启动类加载器-->" + classLoader);

//

//4、测试当前类由哪个类加载器进行加载 ,结果就是系统的类加载器

classLoader = Class.forName("com.java.reflection.Person").getClassLoader();

System.out.println("当前类由哪个类加载器进行加载-->"+classLoader);

//5、测试JDK提供的Object类由哪个类加载器负责加载的

classLoader = Class.forName("java.lang.Object").getClassLoader();

System.out.println("JDK提供的Object类由哪个类加载器加载-->" + classLoader);

}

系统的类加载器-->sun.misc.Launcher$AppClassLoader@43be2d65

扩展类加载器-->sun.misc.Launcher$ExtClassLoader@7a9664a1

启动类加载器-->null

当前类由哪个类加载器进行加载-->sun.misc.Launcher$AppClassLoader@43be2d65

JDK提供的Object类由哪个类加载器加载-->null

**5 getResourceAsStream方法**

@Test

public void testGetResourceAsStream() throws ClassNotFoundException, IOException {

// 这么写的话，文件需要放到src目录下

// InputStream in = new FileInputStream("test.properties");

//5、关于类加载器的一个主要方法

//调用getResourceAsStream 获取类路径下的文件对应的输入流

InputStream in = this.getClass().getClassLoader()

.getResourceAsStream("com/java/reflection/test.properties");

System.out.println("in: " +in);

Properties properties = new Properties();

properties.load(in);

String driverClass = properties.getProperty("dirver");

String jdbcUrl = properties.getProperty("jdbcUrl");

//中文可能会出现乱码，需要转换一下

String user = new String(properties.getProperty("user").getBytes("ISO-8859-1"), "UTF-8");

String password = properties.getProperty("password");

System.out.println("diverClass: "+driverClass);

System.out.println("user: " + user);

}

test.properties内容如下：

dirver=com.mysql.jdbc.Driver;

jdbcUrl=jdbc:mysql://192.168.42.108:3306/test

user=测试

password=993803

结果：

in: java.io.BufferedInputStream@2aca0115

diverClass: com.mysql.jdbc.Driver;

user: 测试

**6 Method: 对应类中的方法**

public class Person {

private String name;

private int age;

//新增一个私有方法

private void privateMthod(){

}

public Person() {

System.out.println("无参构造器");

}

public Person(String name, int age) {

System.out.println("有参构造器");

this.name = name;

this.age = age;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

/\*\*

\*

\* @param age 类型用Integer，不用int

\*/

public void setName(String name , int age){

System.out.println("name: " + name);

System.out.println("age:"+ age);

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

@Override

public String toString() {

return "Person{" +

"name='" + name + '\'' +

", age=" + age +

'}';

}

}

@Test

public void testMethod() throws ClassNotFoundException, NoSuchMethodException,

IllegalAccessException, InstantiationException, InvocationTargetException {

Class clazz = Class.forName("com.java.reflection.Person");

//1、得到clazz 对应的类中有哪些方法,不能获取private方法

Method[] methods =clazz.getMethods();

System.out.print(" getMethods: ");

for (Method method : methods){

System.out.print(method.getName() + ", ");

}

//2、获取所有的方法(且只获取当着类声明的方法，包括private方法）

Method[] methods2 = clazz.getDeclaredMethods();

System.out.print("\ngetDeclaredMethods: ");

for (Method method : methods2){

System.out.print(method.getName() + ", ");

}

//3、获取指定的方法

Method method = clazz.getDeclaredMethod("setName",String.class);//第一个参数是方法名，后面的是方法里的参数

System.out.println("\nmethod : " + method);

Method method2 = clazz.getDeclaredMethod("setName",String.class ,int.class);//第一个参数是方法名，后面的是方法里的参数

System.out.println("method2: " + method2);

//4、执行方法！

Object obj = clazz.newInstance();

method2.invoke(obj, "changwen", 22);

}

getMethods: toString, getName, setName, setName, setAge, getAge, wait, wait, wait, equals, hashCode, getClass, notify, notifyAll,

getDeclaredMethods: toString, getName, setName, setName, setAge, getAge, privateMthod,

method : public void com.java.reflection.Person.setName(java.lang.String)

method2: public void com.java.reflection.Person.setName(java.lang.String,int)

无参构造器

name: changwen

age:22

**7 invoke方法**

public class PersonInvoke {

public PersonInvoke() {

}

private String method2() {

return "Person private String method2";

}

}

public class StudentInvoke extends PersonInvoke{

private void method1(Integer age) {

System.out.println("Student private void method1, age=:" +age);

}

}

（1）获取当前类的父类定义的私有方法

/\*\*

\* 获取当前类的父类中定义的私有方法

\* 直接调用getSuperclass()

\*/

@Test

public void testGetSuperClass() throws Exception {

String className = "com.java.reflection.StudentInvoke";

Class clazz = Class.forName(className);

Class superClazz = clazz.getSuperclass();

System.out.println(superClazz);

//输出结果：class com.java.reflection.PersonInvoke

}

（2）另一种写法

/\*\*

\* @param className 某个类的全类名

\* @param methodName 类的一个方法的方法名，该方法也可能是私有方法

\* @param args 调用该方法需要传入的参数 ...可变参数的意思

\* @return 调用方法后的返回值

\*/

public Object invoke(String className, String methodName, Object ... args) {

Object obj = null;

try {

obj = Class.forName(className).newInstance();

return invoke(obj, methodName, args);

} catch (InstantiationException e) {

e.printStackTrace();

} catch (IllegalAccessException e) {

e.printStackTrace();

} catch (ClassNotFoundException e) {

e.printStackTrace();

}

return invoke(null, methodName, args);

}

/\*\*

\* @param obj 方法执行的那个对象

\* @param methodName 类的一个方法的方法名，该方法也可能是私有方法,还可能是该方法在父类中定义的私有方法

\* @param args 调用该方法需要传入的参数 ...可变参数的意思

\* @return 调用方法后的返回值

\*/

public Object invoke(Object obj, String methodName, Object ... args) {

//1、获取Method对象

Class [] parameterTypes = new Class[args.length];

for (int i=0 ; i<args.length; i++){

parameterTypes[i] = args[i].getClass();

}

try {

//2、执行Method方法

Method method = getMethod(obj.getClass(), methodName,parameterTypes);

//通过反射执行private方法

method.setAccessible(true);

//3、返回方法的返回值

return method.invoke(obj,args);

} catch (Exception e) {

}

return null;

}

/\*\*

\* 获取clazz 的methodName 方法， 该方法可能是私有方法，还可能是父类中的私有方法

\*/

public Method getMethod(Class clazz, String methodName, Class ... parameterTypes) {

//注意这个循环里的内容！！！

for (; clazz != Object.class; clazz = clazz.getSuperclass()){

try {

return clazz.getDeclaredMethod(methodName, parameterTypes);

} catch (Exception e) { //这里要写Exception，不然会出错，应该是有部分异常没有捕获

}

}

return null;

}

（3）测试

@Test

public void testInvoke2() {

Object obj = new StudentInvoke();

invoke(obj, "method1", 10);

Object result = invoke(obj, "method2");

System.out.println(result);

}

private void method1,age:10

Person private String method2

**8 Field字段**

public class Person {

public String name;

private Integer age;

public Person() {

}

public Person(String name, Integer age) {

this.name = name;

this.age = age;

}

}

/\*\*

\* Field: 封装了字段的信息

\*/

@Test

public void testField() throws

ClassNotFoundException, NoSuchFieldException, IllegalAccessException {

Class clazz = Class.forName("com.java.reflection.Person");

//1、获取字段

//1.1 获取Field的数组,私有字段也能获取

Field[] fields = clazz.getDeclaredFields();

for (Field field: fields) {

System.out.print(field.getName() + ", ");

}

//1.2 获取指定名字的Field（如果是私有的，见下面的4)

Field field = clazz.getDeclaredField("name");

System.out.println("\n获取指定Field名=: " + field.getName());

Person person = new Person("ABC", 12);

//2、获取指定对象的Field的值

Object val = field.get(person);

System.out.println("获取指定对象字段'name'的Field的值=： " + val);

//3、设置指定对象的Field的值

field.set(person, "changwen2");

System.out.println("设置指定对象字段'name'的Field的值=: " + person.name);

//4、若该字段是私有的，需要调用setAccessible(true)方法

Field field2 = clazz.getDeclaredField("age");

field2.setAccessible(true);

System.out.println("获取指定私有字段名=: " + field2.getName());

}

name, age,

获取指定Field名=: name

获取指定对象字段'name'的Field的值=： ABC

设置指定对象字段'name'的Field的值=: changwen2

获取指定私有字段名=: age

/\*\*

\* 一个实例：

\* 反射获取一个继承Person2的Student类

\* 设置字段"age"=20(该字段可能为私有，可能在其父类中）

\*/

@Test

public void testClassField() throws ClassNotFoundException, IllegalAccessException, InstantiationException {

String className = "com.java.reflection.Student";

String fieldName = "age"; //可能为私有，可能在其父类中

Object val = 20;

//创建className 对应类的对象，并为其fieldName赋值为val

Class clazz = Class.forName(className);

Field field = null;

for (Class clazz2 = clazz; clazz2 != Object.class; clazz2 = clazz2.getSuperclass()){

try {

field = clazz2.getDeclaredField(fieldName);

} catch (Exception e) {

}

}

Object obj = clazz.newInstance();

assert field != null;

field.setAccessible(true);

field.set(obj, val);

Student stu = (Student) obj;

System.out.println("age = " + stu.getAge());

}

**9 构造器(Constructor)**

/\*\*

\* 构造器：开发用的比较少

\*/

@Test

public void testConstructor() throws ClassNotFoundException, NoSuchMethodException,

IllegalAccessException, InvocationTargetException, InstantiationException {

String className = "com.java.reflection.Person";

Class<Person> clazz = (Class<Person>) Class.forName(className);

//1.获取Constructor对象

Constructor<Person>[] constructors =

(Constructor<Person>[]) Class.forName(className).getConstructors();

for (Constructor<Person> constructor: constructors) {

System.out.println(constructor);

}

Constructor<Person> constructor = clazz.getConstructor(String.class, Integer.class);

System.out.println("指定的-->" + constructor);

//2.调用构造器的newInstance()方法创建对象

Object obj= constructor.newInstance("changwen", 11);

}

public com.java.reflection.Person()

public com.java.reflection.Person(java.lang.String,java.lang.Integer)

指定的-->public com.java.reflection.Person(java.lang.String,java.lang.Integer)

**10 注解(Annotation)**

•从 JDK5.0 开始,Java 增加了对元数据(MetaData)的支持,也就是Annotation(注释)

•Annotation其实就是代码里的特殊标记,这些标记可以在编译,类加载, 运行时被读取,并执行相应的处理.通过使用Annotation,程序员可以在不改变原有逻辑的情况下,在源文件中嵌入一些补充信息.

•Annotation 可以像修饰符一样被使用,可用于修饰包,类,构造器, 方法,成员变量, 参数,局部变量的声明,这些信息被保存在Annotation的 “name=value”对中.

•Annotation能被用来为程序元素(类,方法,成员变量等)设置元数据

**（1）基本的 Annotation**

•使用 Annotation时要在其前面增加@符号,并把该Annotation 当成一个修饰符使用.用于修饰它支持的程序元素

•三个基本的Annotation:

–@Override:限定重写父类方法,该注释只能用于方法

–@Deprecated:用于表示某个程序元素(类,方法等)已过时

–@SuppressWarnings:抑制编译器警告.

**（2）自定义 Annotation**

•定义新的 Annotation类型使用@interface关键字

•Annotation 的成员变量在Annotation 定义中以无参数方法的形式来声明.其方法名和返回值定义了该成员的名字和类型.

•可以在定义Annotation的成员变量时为其指定初始值,指定成员变量的初始值可使用default关键字

•没有成员定义的Annotation称为标记;包含成员变量的Annotation称为元数据Annotation

@Retention(RetentionPolicy.RUNTIME) //运行时检验

@Target(value = {ElementType.METHOD}) //作用在方法上

public @interface AgeValidator {

int min();

int max();

}

@Test

public void testAnnotation() throws Exception {

//这样的方式不能使用注解

Person3 person3 = new Person3();

person3.setAge(10);\*/

String className = "com.java.reflection.Person3";

Class clazz = Class.forName(className);

Object obj = clazz.newInstance();

Method method = clazz.getDeclaredMethod("setAge",Integer.class);

int val =40;

//获取注解

Annotation annotation = method.getAnnotation(AgeValidator.class);

if (annotation != null){

if (annotation instanceof AgeValidator){

AgeValidator ageValidator = (AgeValidator) annotation;

if (val< ageValidator.min() || val>ageValidator.max()){

throw new RuntimeException("数值超出范围");

}

}

}

method.invoke(obj, val);

System.out.println(obj);

}

**（3）提取 Annotation信息**

•JDK5.0 在 java.lang.reflect包下新增了 AnnotatedElement接口,该接口代表程序中可以接受注释的程序元素

•当一个 Annotation类型被定义为运行时Annotation后,该注释才是运行时可见,当 class文件被载入时保存在 class文件中的 Annotation才会被虚拟机读取

•程序可以调用AnnotationElement对象的如下方法来访问 Annotation信息

–获取 Annotation实例：

•getAnnotation(Class<T> annotationClass)

•getDeclaredAnnotations()

•getParameterAnnotations()

**（4）JDK 的元Annotation**

•JDK 的元Annotation 用于修饰其他Annotation 定义

•@Retention:只能用于修饰一个 Annotation定义,用于指定该 Annotation可以保留多长时间,@Rentention包含一个RetentionPolicy类型的成员变量,使用 @Rentention时必须为该 value成员变量指定值:

–RetentionPolicy.CLASS:编译器将把注释记录在 class文件中.当运行 Java程序时,JVM 不会保留注释.这是默认值

–RetentionPolicy.RUNTIME:编译器将把注释记录在class文件中. 当运行 Java 程序时, JVM 会保留注释. 程序可以通过反射获取该注释

–RetentionPolicy.SOURCE:编译器直接丢弃这种策略的注释

•@Target: 用于修饰Annotation 定义,用于指定被修饰的 Annotation能用于修饰哪些程序元素.@Target 也包含一个名为 value的成员变量.

•@Documented:用于指定被该元 Annotation修饰的 Annotation类将被 javadoc工具提取成文档.

•@Inherited:被它修饰的 Annotation将具有继承性.如果某个类使用了被@Inherited 修饰的Annotation, 则其子类将自动具有该注释

## 三 、java的核心机制

1 java有两种核心机制:**java虚拟机(JavaVirtual Machine)与垃圾收集机制(Garbage collection)：**

2 Java虚拟机：是运行所有Java程序的抽象计算机，是Java语言的运行环境，在其上面运行Java代码编译后的字节码程序，java虚拟机实现了平台无关性。

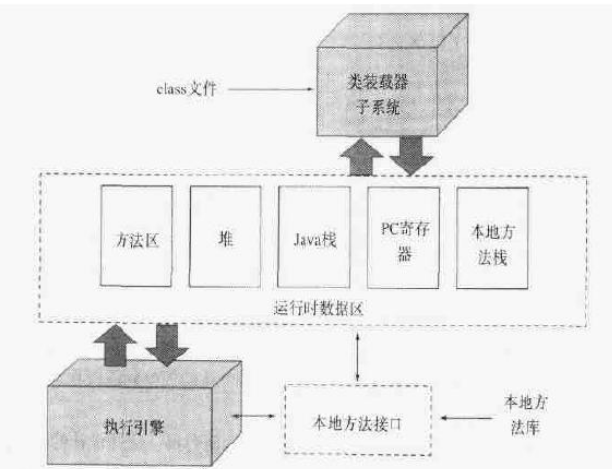
3 Java垃圾回收（Garbage Collection）：自动释放不用对象内存空间，在java程序运行过程中自动进行，垃圾收集机制可大大缩短编程时间，保护程序的完整性，是Java语言安全性策略的一个重要部份。

## 四、java虚拟机及其结构

1 java垃圾回收不需要程序员手动操作，我们经常需要关注的是java虚拟机，java虚拟机承载着程序从源码到运行的全部工作。

2 Java虚拟机是可运行Java代码的假想计算机，有自己想象中的硬件，如处理器、堆栈、寄存器等，还具有相应的指令系统，可以执行 Java 的字节码程序。Java语言的一个非常重要的特点就是与平台的无关性。而使用Java虚拟机是实现这一特点的关键。Java语言使用模式Java虚拟机屏蔽了与具体平台相关的信息，使得Java语言编译程序只需生成在Java虚拟机上运行的目标代码（字节码），就可以在多种平台上不加修改地运行。Java虚拟机在执行字节码时，把字节码解释成具体平台上的机器指令执行。

3 对于 JVM 的基本结构，我们可以从下图可以大致了解：



## 五、程序的运行过程

1 从源文件创建到程序运行，Java程序要经过两大步骤：编译，运行；（1）源文件由编译器编译成字节码（ByteCode） （2）字节码由java虚拟机解释运行。

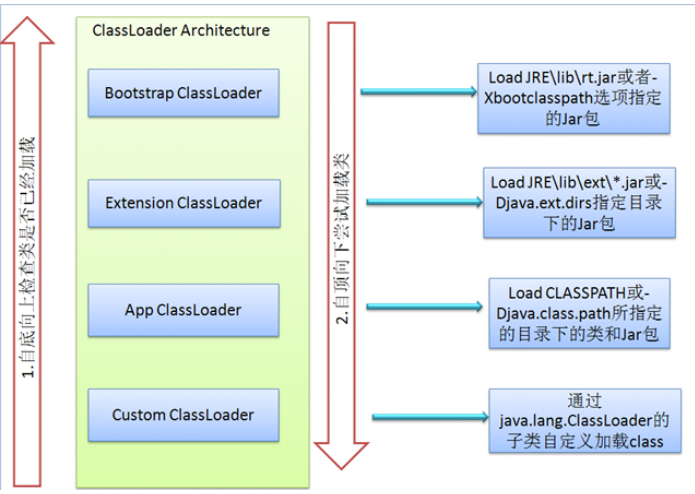
（1）第一步(编译): 创建完源文件之后，程序会被编译器编译为.class文件。Java编译一个类时，如果这个类所依赖的类还没有被编译，编译器就会先编译这个被依赖的类，然后引用，否则直接引用。。编译后的字节码文件格式主要分为两部分：常量池和方法字节码。

（2）第二步（运行）：java类运行的过程大概可分为两个过程：1、类的加载 2、执行。

## 六、类的加载

**1 类加载过程**

java程序经过编译后形成\*.class文件。通过类加载器将字节码(\*.class)加载入JVM的内存中。JVM将类加载过程分成加载，连接，初始化三个阶段，其中连接阶段又可分为验证，准备，解析三个阶段。JVM 的类加载是通过 ClassLoader 及其子类来完成的，类的层次关系和加载顺序可以由下图来描述：



1）Bootstrap ClassLoader启动类加载器

负责加载$JAVA\_HOME中jre/lib/里所有的 class(JDK 代表 JDK 的安装目录，下同)，或被-Xbootclasspath参数指定的路径中的，并且能被虚拟机识别的类库（如 rt.jar，所有的java.\*开头的类均被 Bootstrap ClassLoader 加载）。启动类加载器由 C++ 实现，不是 ClassLoader 子类。无法被 Java 程序直接引用的。

2）Extension ClassLoader扩展类加载器

该加载器由sun.misc.LauncherExtClassLoader实现，负责加载Java平台中扩展功能的一些jar包，包括JAVA\_HOME中jre/lib/.jar或-Djava.ext.dirs指定目录下的 jar 包。即JDK\jre\lib\ext目录中，或者由 java.ext.dirs 系统变量指定的路径中的所有类库（如javax.开头的类），开发者可以直接使用扩展类加载器

3）App ClassLoader应用程序类加载器

。该类加载器由 sun.misc.Launcher$AppClassLoader 来实现，负责记载 classpath 中指定的 jar 包及目录中 class，开发者可以直接使用该类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。启动类加载器：它使用 C++ 实现（这里仅限于 Hotspot，也就是 JDK1.5 之后默认的虚拟机，有很多其他的虚拟机是用 Java 语言实现的），是虚拟机自身的一部分。 所有其他的类加载器：这些类加载器都由 Java 语言实现，独立于虚拟机之外，并且全部继承自抽象类 java.lang.ClassLoader，这些类加载器需要由启动类加载器加载到内存中之后才能去加载其他的类。 应用程序都是由这三种类加载器互相配合进行加载的，我们还可以加入自定义的类加载器。

**2 加载**

（1）加载时类加载过程的第一个阶段，在加载阶段，虚拟机需要完成以下三件事情：通过一个类的全限定名来获取其定义的二进制字节流。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。在 Java 堆中生成一个代表这个类的 java.lang.Class 对象，作为对方法区中这些数据的访问入口。

（2）注意，这里第 1 条中的二进制字节流并不只是单纯地从 Class 文件中获取，比如它还可以从 Jar 包中获取、从网络中获取（最典型的应用便是 Applet）、由其他文件生成（JSP 应用）等。相对于类加载的其他阶段而言，加载阶段（准确地说，是加载阶段获取类的二进制字节流的动作）是可控性最强的阶段，因为开发人员既可以使用系统提供的类加载器来完成加载，也可以自定义自己的类加载器来完成加载。JVM主要在程序第一次主动使用类的时候，才会去加载该类。也就是说，JVM并不是在一开始就把一个程序就所有的类都加载到内存中，而是到用的时候才把它加载进来，而且只加载一次。

（3）加载过程中会先检查类是否被已加载，检查顺序是自底向上，从 Custom ClassLoader 到 BootStrap ClassLoader 逐层检查，只要某个 Classloader 已加载就视为已加载此类，保证此类只所有 ClassLoade r加载一次。而加载的顺序是自顶向下，也就是由上层来逐层尝试加载此类。

这几种类加载器的层次关系如下图所示：



这种层次关系称为类加载器的双亲委派模型。双亲委派模型的工作流程是：

（4）如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把请求委托给父加载器去完成，依次向上，因此，所有的类加载请求最终都应该被传递到顶层的启动类加载器中，只有当父加载器在它的搜索范围中没有找到所需的类时，即无法完成该加载，子加载器才会尝试自己去加载该类。

**3 验证**

验证的目的是为了确保 Class 文件中的字节流包含的信息符合当前虚拟机的要求，而且不会危害虚拟机自身的安全。不同的虚拟机对类验证的实现可能会有所不同，但大致都会完成以下四个阶段的验证：文件格式的验证、元数据的验证、字节码验证和符号引用验证。

**4 准备**

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些内存都将在方法区中分配。对于该阶段有以下几点需要注意：这时候进行内存分配的仅包括类变量（static），而不包括实例变量，实例变量会在对象实例化时随着对象一块分配在 Java 堆中。这里所设置的初始值通常情况下是数据类型默认的零值（如 0、0L、null、false 等），而不是被在 Java 代码中被显式地赋予的值。

**5 解析**

解析阶段是虚拟机将常量池中的符号引用转化为直接引用的过程。

解析动作主要针对类或接口、字段、类方法、接口方法四类符号引用进行，分别对应于常量池中的 CONSTANT\_Class\_info、CONSTANT\_Fieldref\_info、CONSTANT\_Methodref\_info、CONSTANT\_InterfaceMethodref\_info 四种常量类型。

**（1）类或接口的解析**：判断所要转化成的直接引用是对数组类型，还是普通的对象类型的引用，从而进行不同的解析。

**（2）字段解析：**对字段进行解析时，会先在本类中查找是否包含有简单名称和字段描述符都与目标相匹配的字段，如果有，则查找结束；如果没有，则会按照继承关系从上往下递归搜索该类所实现的各个接口和它们的父接口，还没有，则按照继承关系从上往下递归搜索其父类，直至查找结束

**6 初始化**

类初始化是类加载过程的最后一个阶段，到初始化阶段，才真正开始执行类中的 Java 程序代码。虚拟机规范严格规定了有且只有四种情况必须立即对类进行初始化：

（1）遇到 new、getstatic、putstatic、invokestatic 这四条字节码指令时，如果类还没有进行过初始化，则需要先触发其初始化。生成这四条指令最常见的 Java 代码场景是：使用 new 关键字实例化对象时、读取或设置一个类的静态字段（static）时（被 static 修饰又被 final 修饰的，已在编译期把结果放入常量池的静态字段除外）、以及调用一个类的静态方法时。使用 Java.lang.refect 包的方法对类进行反射调用时，如果类还没有进行过初始化，则需要先触发其初始化。

（2）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。

（3）当虚拟机启动时，用户需要指定一个要执行的主类，虚拟机会先执行该主类。

（4）虚拟机规定只有这四种情况才会触发类的初始化，称为对一个类进行主动引用，除此之外所有引用类的方式都不会触发其初始化，称为被动引用。本大段参考引用及图片来自深入理解 Java 虚拟机

## 七、静态加载和动态加载

1 Java初始化一个类的时候可以用new 操作符来初始化,也可通过Class.forName的方式来得到一个Class类型的实例,然后通过这个Class类型的实例的newInstance来初始化.我们把前者叫做JAVA的静态加载,把后者叫做动态加载.。

2 有时候我们说某个语言具有很强的动态性，有时候我们会区分动态和静态的不同技术与作法。我们朗朗上口动态绑定（dynamic binding）、动态链接（dynamic linking）、动态加载（dynamic loading）等。然而“动态”一词其实没有绝对而普遍适用的严格定义，有时候甚至像面向对象当初被导入编程领域一样，一人一把号，各吹各的调。

一般而言，开发者社群说到动态语言，大致认同的一个定义是：“程序运行时，允许改变程序结构或变量类型，这种语言称为动态语言”。从这个观点看，Perl，Python，Ruby是动态语言，C++，Java，C#不是动态语言。

尽管在这样的定义与分类下Java不是动态语言，它却有着一个非常突出的动态相关机制：Reflection。Java程序可以加载一个运行时才得知名称的class，获悉其完整构造（但不包括methods定义），并生成其对象实体、或对其fields设值、或唤起其methods。

（1）静态加载的时候如果在运行环境中找不到要初始化的类,抛出的是NoClassDefFoundError,它在JAVA的异常体系中是一个Error.

（2）动态态加载的时候如果在运行环境中找不到要初始化的类,抛出的是ClassNotFoundException,它在JAVA的异常体系中是一个checked异常,在写代码的时候就需要catch.

## 八、反射

**1 JAVA反射机制：**

在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；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。

**2 Java反射机制主要提供了以下功能：**

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；在运行时调用任意一个对象的方法；生成动态代理。

**3 Java有个Object 类**，是所有Java 类的继承根源，其内声明了数个应该在所有Java 类中被改写的方法：hashCode()、equals()、clone()、toString()、getClass()等。其中getClass()返回一个Class 对象。 Class 类十分特殊。它和一般类一样继承自Object，其实体用以表达Java程序运行时的classes和interfaces，也用来表达enum、array、primitive Java types（boolean, byte, char, short, int, long, float, double）以及关键词void。当一个class被加载，或当加载器（class loader）的defineClass()被JVM调用，JVM 便自动产生一个Class 对象。

**4 如果您想借由“修改Java标准库源码”来观察Class 对象的实际生成时机**（例如在Class的constructor内添加一个println()），这样是行不通的！因为Class并没有public constructor。

Class是Reflection故事起源。针对任何您想探勘的类，唯有先为它产生一个Class 对象，接下来才能经由后者唤起为数十多个的Reflection APIs。Reflection机制允许程序在正在执行的过程中，利用Reflection APIs取得任何已知名称的类的内部信息，包括：package、 type parameters、 superclass、 implemented interfaces、 inner classes、 outer classes、 fields、 constructors、 methods、 modifiers等，并可以在执行的过程中，动态生成instances、变更fields内容或唤起methods。

**5 从Class中获取信息**

Class类提供了大量的实例方法来获取该Class对象所对应的详细信息,Class类大致包含如下方法,其中每个方法都包含多个重载版本,因此我们只是做简单的介绍。

**获取内容 方法签名**

构造器 Constructor<T> getConstructor(Class<?>... parameterTypes)

包含的方法 Method getMethod(String name, Class<?>... parameterTypes)

包含的属性 Field getField(String name)

包含的Annotation <A extends Annotation> A getAnnotation(Class<A> annotationClass)

内部类 Class<?>[] getDeclaredClasses()

外部类 Class<?> getDeclaringClass()

所实现的接口 Class<?>[] getInterfaces()

修饰符 int getModifiers()

所在包 Package getPackage()

类名 String getName()

简称 String getSimpleName()

**6 一些判断类本身信息的方法**

判断内容 方法签名

注解类型? boolean isAnnotation()

使用了该Annotation修饰? boolean isAnnotationPresent(Class<? extends Annotation> annotationClass)

匿名类? boolean isAnonymousClass()

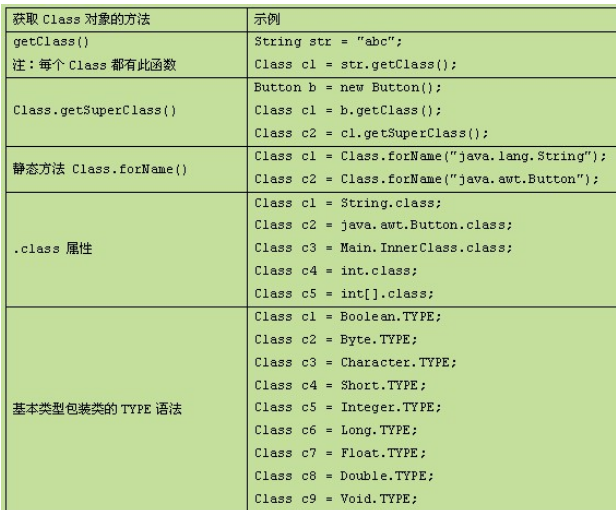
数组? boolean isArray()

枚举? boolean isEnum()

原始类型? boolean isPrimitive()

接口? boolean isInterface()obj是否是该Class的实例 boolean isInstance(Object obj)

**7 获取class**



主有三种获得class的途径，使用时要注意区别

**a,类型.class**  如： String.class使用类名加“.class”的方式即会返回与该类对应的Class对象。这个方法可以直接获得与指定类关联的Class对象，而并不需要有该类的对象存在。

**b,Class.forName("类名");**该方法可以根据字符串参数所指定的类名获取与该类关联的Class对象。如果该类还没有被装入，该方法会将该类装入JVM。forName方法的参数是类的完 整限定名（即包含包名）。通常用于在程序运行时根据类名动态的载入该类并获得与之对应的Class对象。

**c, obj.getClass();**所有Java对象都具备这个方法，该方法用于返回调用该方法的对象的所属类关联的Class对象

**8 获取构造方法**

Class类提供了四个public方法，用于获取某个类的构造方法：

**Constructor getConstructor(Class[] params**)根据构造函数的参数，返回一个具体的具有public属性的构造函数

**Constructor getConstructors()**  返回所有具有public属性的构造函数数组

**Constructor getDeclaredConstructor(Class[] params)**  根据构造函数的参数，返回一个具体的构造函数（不分public和非public属性）

**Constructor getDeclaredConstructors()**  返回该类中所有的构造函数数组（不分public和非public属性）

\* 1 反射出无参的构造方法并得到对象

　　\* 注意:

　　\* 1 在Class.forName()中应该传入含有包名的类全名

　　\* 2 newInstance()方法的本质是调用类的无参Public构造方法

　　\*/

　　String className1="cn.testreflect.Worker";

　　Class clazz1=Class.forName(className1);

　　Object object1=clazz1.newInstance();

　　System.out.println("object1.toString()="+object1.toString());

　　/\*\*

　　\* 2 反射出带参数的构造方法并得到对象

　　\*/

　　String className2="cn.testreflect.Worker";

　　Class clazz2=Class.forName(className2);

　　Constructor constructor1=clazz2.getConstructor(int.class,String.class);

　　Object object2=constructor1.newInstance(18,"小明");

　　System.out.println("object2.toString()="+object2.toString());



**9 获取类的成员方法**

与获取构造方法的方式相同，存在四种获取成员方法的方式。

**Method getMethod(String name, Class[] params)** 根据方法名和参数，返回一个具体的具有public属性的方法

**Method[] getMethods()** 返回所有具有public属性的方法数组

**Method getDeclaredMethod(String name, Class[] params)**  根据方法名和参数，返回一个具体的方法（不分public和非public属性）

**Method[] getDeclaredMethods()** 返回该类中的所有的方法数组（不分public和非public属性）

\* 调用对象的带参数的方法

　　\*/

　　String className5="cn.testreflect.Worker";

　　Class clazz5=Class.forName(className5);

　　Method method=clazz5.getMethod("printMessage",

String.class,int.class,int.class);

　　Object object5=clazz5.newInstance();

　　method.invoke(object5, "周星星",50,9527);

　　} catch (Exception e) {

　　System.out.println(e.toString());

　　}

**10 获取类的成员变量（成员属性）**

存在四种获取成员属性的方法

**Field getField(String name)**  根据变量名，返回一个具体的具有public属性的成员变量

**Field[] getFields()**  返回具有public属性的成员变量的数组

**Field getDeclaredField(String name)** 根据变量名，返回一个成员变量（不分public和非public属性）

**Field[] getDelcaredFields()** 返回所有成员变量组成的数组（不分public和非public属性）

\* 1 获取类的私有字段

　　\* 注意:

　　\* 获取共有字段应调用clazz3.getField(name)方法

　　\*/

　　String className3="cn.testreflect.Worker";

　　Class clazz3=Class.forName(className3);

　　Field ageField1=clazz3.getDeclaredField("age");

　　System.out.println("ageField1="+ageField1);

　　/\*\*

　　\* 2 获取和更改某个对象的私有字段

　　\* 即模拟get()和set()方法

　　\*/

　　String className4="cn.testreflect.Worker";

　　Class clazz4=Class.forName(className4);

　　Field ageField2=clazz4.getDeclaredField("age");

　　Object object4=constructor1.newInstance(18,"小明");

　　//取消访问私有字段的合法性检查

　　ageField2.setAccessible(true);

　　//获取对象的私有字段

　　Object ageObject4=ageField2.get(object4);

　　System.out.println("ageObject4="+ageObject4);

　　//再更改对象的私有字段的值

　　ageField2.set(object4, 9527);

　　//重新获得

　　Object ageObject5=ageField2.get(object4);

　　System.out.println("ageObject5="+ageObject5);

1. 使用场景

**1 Spring事务注解**

**2 Spring MVC ,Struts1和Struts2**