【济南中心】JAVA 编程阶梯：基础篇之第二十七章

反射-类的加载概述和加载时机

* A:类的加载概述

    * 当程序要使用某个类时，如果该类还未被加载到内存中，则系统会通过加载，连接，初始化三步来实现对这个类进行初始化。

    * 加载

        * 就是指将 class 文件读入内存，并为之创建一个 Class 对象。任何类被使用时系统都会建立一个 Class 对象。

    * 连接

        * 验证 是否有正确的内部结构，并和其他类协调一致

        * 准备 负责为类的静态成员分配内存，并设置默认初始化值

        * 解析 将类的二进制数据中的符号引用替换为直接引用

    * 初始化 就是我们以前讲过的初始化步骤

* B:加载时机

    * 创建类的实例

    * 访问类的静态变量，或者为静态变量赋值

    * 调用类的静态方法

    * 使用反射方式来强制创建某个类或接口对应的 java.lang.Class 对象

    * 初始化某个类的子类

    * 直接使用 java.exe 命令来运行某个主类

反射-类加载器的概述和分类

* A:类加载器的概述

    * 负责将.class 文件加载到内存中，并为之生成对应的 Class 对象。虽然我们不需要关心类加载机制，但是了解这个机制我们就能更好的理解程序的运行。

* B:类加载器的分类

    * Bootstrap ClassLoader 根类加载器

    * Extension ClassLoader 扩展类加载器

    * Sysetm ClassLoader 系统类加载器

* C:类加载器的作用

    * Bootstrap ClassLoader 根类加载器

        * 也被称为引导类加载器，负责 Java 核心类的加载

        * 比如 System,String 等。在 JDK 中 JRE 的 lib 目录下 rt.jar 文件中

    * Extension ClassLoader 扩展类加载器

        * 负责 JRE 的扩展目录中 jar 包的加载。

        * 在 JDK 中 JRE 的 lib 目录下 ext 目录

    * Sysetm ClassLoader 系统类加载器

        * 负责在 JVM 启动时加载来自 java 命令的 class 文件，以及 classpath 环境变量所指定的 jar 包和类路径

反射-反射概述

* A:反射概述

    * JAVA 反射机制是在运行状态中，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。

* 要想解剖一个类,必须先要获取到该类的字节码文件对象。

* 而解剖使用的就是 Class 类中的方法，所以先要获取到每一个字节码文件对应的 Class 类型的对象。

* B:三种方式

* a:Object 类的 getClass()方法,判断两个对象是否是同一个字节码文件

* b:静态属性 class,锁对象

* c:Class 类中静态方法 forName(),读取配置文件

* C:案例演示

* 获取 class 文件对象的三种方式

```
public
classPerson {
private String
name;
int
age;
public String
address;

public Person() {
}

private
Person(String name){
this.name = name;
}
```

```
Person(Stringname, intage) {
this.name = name;
this.age = age;
}

public Person(String name,
int age, String address) {
this.name = name;
this.age = age;
this.address = address;
}

public
void show() {
System.out.println("show");
}

public
void method(String s) {
System.out.println("method "+ s);
}

public String getString(Strings,
inti) {
return s +
"---" + i;
}

private
void function() {
System.out.println("function");
}

@Override
public String toString() {
return
"Person [name="+ name+
", age="+ age+
", address=" + address
+"]";
}

}
```

反射-Class.forName()读取配置文件举例

* 榨汁机(Juicer)榨汁的案例

* 分别有水果(Fruit)苹果(Apple)香蕉(Banana)桔子(Orange)榨汁(squeeze)

```
public class Demo2_Reflect {

	/**
	 * 榨汁机(Juicer)榨汁的案例
	 * 分别有水果(Fruit)苹果(Apple)香蕉(Banana)桔子(Orange)榨汁(squeeze)
	 * @throws Exception
	 */
	public static void main(String[] args) throws Exception {
		/*Juicer j = new Juicer();
		//j.run(new Apple());
		j.run(new Orange());*/
		BufferedReader br = new BufferedReader(new FileReader("config.properties"));	//创建输入流对象,关联配置文件
		Class<?> clazz = Class.forName(br.readLine());	//读取配置文件一行内容,获取该类的字节码对象
		Fruit f = (Fruit) clazz.newInstance();	//通过字节码对象创建实例对象
		Juicer j = new Juicer();
		j.run(f);
	}

}
interface Fruit {
	public void squeeze();
}

class Apple implements Fruit {
	public void squeeze() {
		System.out.println("榨出一杯苹果汁儿");
	}
}

class Orange implements Fruit {
	public void squeeze() {
```

```
                System.out.println("榨出一杯桔子汁儿");
        }
}

class Juicer {
        public void run(Fruit f) {
                f.squeeze();
        }

}
```

反射-通过反射获取带参构造方法并使用

* Constructor

    * Class 类的 newInstance()方法是使用该类无参的构造函数创建对象，如果一个类没有无参的构造函数，就不能这样创建了,可以调用 Class 类的 getConstructor(String.class,int.class)方法获取一个指定的构造函数然后再调用 Constructor 类的 newInstance("张三",20)方法创建对象

反射-通过反射获取成员变量并使用

* Field

    * Class.getField(String)方法可以获取类中的指定字段(可见的)，如果是私有的可以用 getDeclaedField("name")方法获取,通过 set(obj, "李四")方法可以设置指定对象上该字段的值，如果是私有的需要先调用 setAccessible(true)设置访问权限,用获取的指定的字段调用 get(obj)可以获取指定对象中该字段的值

反射-通过反射获取方法并使用

* Method

    * Class.getMethod(String, Class...) 和

Class.getDeclaredMethod(String, Class...)方法可以获取类中的指定方法,调用 invoke(Object, Object...)可以调用该方法,Class.getMethod("eat") invoke(obj) Class.getMethod("eat",int.class) invoke(obj,10)

反射-通过反射越过泛型检查

* A:案例演示

    * ArrayList<Integer>的一个对象，在这个集合中添加一个字符串数据，如何实现呢？

反射-通过反射写一个通用的设置某个对象的某个属性为指定的值

* A:案例演示

    * public void setProperty(Object obj, String propertyName, Object value){} ,此方法可将 obj 对象中名为 propertyName 的属性的值设置为 value。

反射-动态代理的概述和实现

* A:动态代理概述

    * 代理：本来应该自己做的事情，请了别人来做，被请的人就是代理对象。

    * 举例：春节回家买票让人代买

    * 动态代理：在程序运行过程中产生的这个对象,而程序运行过程中产生对象其实就是我们刚才反射讲解的内容，所以，动态代理其实就是通过反射来生成一个代理

* 在 Java 中 java.lang.reflect 包下提供了一个 Proxy 类和一个 InvocationHandler 接口，通过使用这个类和接口就可以生成动态代理对象。JDK 提供的代理只能针对接口做代理。我们有更强大的代理 cglib，Proxy 类中的方法创建动态代理类对象

* public static Object newProxyInstance(ClassLoader loader,Class<?>[] interfaces,InvocationHandler h)

* 最终会调用 InvocationHandler 的方法

* InvocationHandler Object invoke(Object proxy,Method method,Object[] args)

设计模式-模版(Template)设计模式概述和使用

* A:模版设计模式概述

* 模版方法模式就是定义一个算法的骨架，而将具体的算法延迟到子类中来实现

* B:优点和缺点

* a:优点

* 使用模版方法模式，在定义算法骨架的同时，可以很灵活的实现具体的算法，满足用户灵活多变的需求

* b:缺点

* 如果算法骨架有修改的话，则需要修改抽象类

常用的设计模式：

1,装饰

2,单例

3,简单工厂

4,工厂方法

5,适配器

6,模版

JDK5 新特性

* A:枚举概述

    * 是指将变量的值一一列出来,变量的值只限于列举出来的值的范围内。举例：一周只有 7 天，一年只有 12 个月等。

* B:回想单例设计模式：单例类是一个类只有一个实例

    * 那么多例类就是一个类有多个实例，但不是无限个数的实例，而是有限个数的实例。这才能是枚举类。

* C:案例演示

    * 自己实现枚举类

1,自动拆装箱

2,泛型

3,可变参数

4,静态导入

5,增强 for 循环

6,互斥锁

7,枚举

JDK5 新特性(通过 enum 实现枚举类)

* A:案例演示

    * 通过 enum 实现枚举类

```
public enum Planet {
MERCURY(3.302e+23, 2.439e6),
VENUS (4.869e+24, 6.052e6),
EARTH (5.975e+24, 6.378e6),
MARS (6.419e+23, 3.393e6),
JUPITER(1.899e+27, 7.149e7),
SATURN (5.685e+26, 6.027e7),
URANUS (8.683e+25, 2.556e7),
NEPTUNE(1.024e+26, 2.477e7);
private final double mass; // In kilograms
private final double radius; // In meters
private final double surfaceGravity; // In m / s^2

// Universal gravitational constant in m^3 / kg s^2
private static final double G = 6.67300E-11;

// Constructor
Planet(double mass, double radius) {
this.mass = mass;
this.radius = radius;
surfaceGravity = G * mass / (radius * radius);
}

public double mass() { return mass; }
public double radius() { return radius; }
public double surfaceGravity() { return surfaceGravity; }

public double surfaceWeight(double mass) {
return mass * surfaceGravity; // F = ma
}
}
```

JDK5 新特性-枚举的注意事项

* A:案例演示

* 定义枚举类要用关键字 enum

* 所有枚举类都是 Enum 的子类

* 枚举类的第一行上必须是枚举项，最后一个枚举项后的分号是可以省略的，但是如果枚举类有其他的东西，这个分号就不能省略。建议不要省略

* 枚举类可以有构造器 ,但必须是 private 的 ,它默认的也是 private 的。

* 枚举类也可以有抽象方法，但是枚举项必须重写该方法

* 枚举在 switch 语句中的使用

JDK5 新特性-枚举类的常见方法

* A:枚举类的常见方法

* int ordinal()

* int compareTo(E o)

* String name()

* String toString()

* <T> T valueOf(Class<T> type,String name)

* values()

* 此方法虽然在 JDK 文档中查找不到 ,但每个枚举类都具有该方法，它遍历枚举类的所有枚举值非常方便

* B:案例演示

* 枚举类的常见方法

JDK7 新特性-JDK7 的六个新特性回顾和讲解

* A:二进制字面量

* B:数字字面量可以出现下划线

* C:switch 语句可以用字符串

* D:泛型简化,菱形泛型

* E:异常的多个 catch 合并,每个异常用或|

* F:try-with-resources 语句

JDK8 新特性-JDK8 的新特性

* 接口中可以定义有方法体的方法,如果是非静态,必须用 default 修饰

* 如果是静态的就不用了

```
class Test {
                public void run() {
                        final int x = 10;
                        class Inner {
                                public void method() {
                                        System.out.println(x);
                                }
                        }

                        Inner i = new Inner();
                        i.method();
                }

        }
```

局部内部类在访问他所在方法中的局部变量必须用 final 修饰,为什么?

因为当调用这个方法时,局部变量如果没有用 final 修饰,他的生命

周期和方法的生命周期是一样的,当方法弹栈,这个局部变量也会消失,那么如果局部内部类对象还没有马上消失想用这个局部变量,就没有了,如果用 final 修饰会在类加载的时候进入常量池,即使方法弹栈,常量池的常量还在,也可以继续使用