注解和反射

注解

注解入门

什么是注解

- Annotation 是从JDK5.0开始引入的新技术
- Annotation 的作用：
  - 不是程序本身，可以对程序作出解释（这一点和注释（comment）没什么区别）
  - 可以被其他程序（比如编译器等）读取
- Annotation 的格式：
  - 注解是以 `@注解名` 在代码中存在的，还可以添加一些参数值，例如：`@SuppressWarnings(value = "unchecked")`
- Annotation 在那里使用：
  - 可以附加在 package、class、method、field 等上面，相当于给他们添加了额外的辅助信息，我们可以通过反射机制编程实现对这些元数据的访问

内置注解

- `@Override` ：定义在 java.lang.Override 中，此注解只适用于修饰方法，表示一个方法声明打算重写超类中的另一个方法声明
- `@Deprecated` ：定义在 java.lang.Deprecated 中，此注解可以用于修饰方法、属性、类，表示不鼓励程序员使用这样的元素，通常是因为它很危险或者存在更好的选择
- `@SuppressWarnings` ：定义在 java.lang.SuppressWarnings 中，用来抑制编译时的警告信息
  - 与前两个注解不同，此注解需要添加一个参数才能正确使用，这些参数都是已经定义好了的，直接使用就好了
    - `@SuppressWarnings("all")`
    - `@SuppressWarnings("unchecked")`
    - `@SuppressWarnings(value = {"unchecked", "deprecation"})`

- ……

自定义注解、元注解

元注解

- 元注解的作用就是负责注解其他注解，Java 定义了4个标准的 meta-annotation 类型，它们被用来提供对其他 annotation 类型做说明
- 这些类型和它们所支持的类在 java.lang.annotation 包中可以找到。（`@Tatget`、`@Retention`、`@Documented`、`@Inherited`）
  - `@Target`：用于描述注解的适用范围（即被描述的注解可以用在什么地方）
  - `@Retention`：表示需要在什么级别保存该注释信息，用于描述注解的生命周期（**SOURCE < CLASS < RUNTIME**）
  - `@Documented`：说明该注解将被包含在 javadoc 中
  - `@Inherited`：说明子类可以**继承**父类中的该注解

自定义注解

- 使用 `@interface` 自定义注解时，自动继承了 java.lang.annotation.Annotation 接口
- 分析：
  - `@interface` 用来声明一个注解，格式：`public @interface 注解名 { 定义内容 }`
  - 其中每一个方法实际上就是声明了一个配置参数
  - 方法的名称就是参数的名称
  - 返回值类型就是参数的类型（返回值只能是基本类型，Class、String、enum）
  - 可以通过 `default` 来声明参数的默认值
  - 如果只有一个参数成员，一般参数名为 value
  - 注解元素必须要有值，定义注解元素时，经常使用空字符串、0作为默认值

反射机制

Java反射机制概述

静态 VS 动态语言

- 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等。
- 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 Reflection

- Reflection（反射）是Java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，并能直接操作任意类的内部属性及方法。

```
1   Class c = Class.forName("java.lang.String");
```

- 加载完类之后，在对内存的方法区就产生了一个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，透过这个镜子可以看到类的结构，所以，我们形象地称之为：反射。
  - 正常方式：引入需要的“包类”名称 ---> 通过new实例化 ---> 取得实例化对象
  - 反射方式：实例化对象 ---> getClass()方法 ---> 得到完整的“包类”名称

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
- ……

Java反射优点和缺点

优点：

- 可以实现动态创建对象和编译，体现出很大的灵活性

缺点：

- 对性能有影响。使用反射基本上是一种解释操作，我们可以 JVM，我们希望做什么并且它会满足我i们的要求。这类操作总是慢于直接执行相同的操作。

Class 类

Class 类概述

在 Object 类中定义了以下方法，此方法将被所有子类继承：

```
1   public final Class getClass()
```

- 以上的方法返回值的类型就是一个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。
  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
- 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 可以完整的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，唯有获得相应的 Class 对象

Class 类中的常用方法

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 Class 对象的一个实例
getName()	返回此 Class 对象所表示的实体（类，接口，数组类或 void）的名称
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
Constructor[] getConstructors()	返回一个包含某些 Constructor 对象的数组
Method getMethod(String name, Class.. T)	返回一个 Method 对象，此对象的形参类型为 paramType
Field[] getDeclaredFields()	返回 Field 对象的一个数组

获取 Class 类的实例

- 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。

```
Class class = Person.class;
```

- 已知某个类的实例，调用该实例的 `getClass()` 方法获取 Class 对象。

```
Class class = person.getClass();
```

- 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。

```
Class class = Class.forName("demo01.Student");
```

- 内置基本数据类型可以直接用 类名.Type
- 还可以利用 ClassLoader

哪些类型可以有 Class 对象?

- 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
- interface：接口
- []：数组
- enum：枚举
- annotation：注解 `@interface`
- primitive type：基本数据类型
- void

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。
- 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
  - 准备：正式为类变量（static）分配内存并设置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>()` 方法是由编译器自动收集类中所有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。

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。如：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）

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。
- 类加载器作用：用来把类装载进内存的。JVM 规范定义了如下类型的类加载器：
  - 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 所指定的目录下的类与 jar 包装入工作库，是最常用的加载器。

```java
// 获取系统类的加载器
ClassLoader systemClassLoader = ClassLoader.getSystemClassLoader();

// 获取系统类加载器的父类加载器 ⟶ 扩展类加载器
ClassLoader parent = systemClassLoader.getParent();

// 获取扩展类加载器的父类加载器 ⟶ 根加载器(C/C++)
ClassLoader root = parent.getParent();
```

• 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
  - 类必须有一个无参构造器
  - 类的构造器的访问权限需要足够

> “
> ❓思考：难道没有无参的构造器就不能创建对象了吗?

只要在操作的时候明确地调用类中的构造器，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 ... paramterTypes)` 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

- 调用指定方法

通过反射，调用类中的方法，通过 Method 类完成。

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 object 对象的参数信息。
3. `Object invoke(Object obj, Object ... args)`
   - Object 对应原方法的返回值，若原方法无返回值，此时返回null
   - 若原方法为静态方法，此时形参 Object obj 可为 null
   - 若原方法形参列表为空，则 Object[] args 为 null
   - 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 方法

setAccessible

- Method 和 Field、Constructor 对象都有 setAccessible() 方法
- setAccessible 作用是启动和禁止访问安全检查的开关
- 参数值为 true 则表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得被调用，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则表示反射的对象应该实施Java语言访问检查

```
// 获得Class对象
Class c1 = Class.forName("com.test.reflection.User");

// 构造一个对象
User user = (User) c1.newInstance(); // 本质是调用了类的无参构造器

// 通过构造器创建对象
Constructor constructor = c1.getDeclaredConstructor(String.class, int.class, int.class);
User user2 = (User) constructor.newInstance("张三", 001, 18);

// 通过反射调用普通方法
User user3 = (User) c1.newInstance();

// 通过反射获取一个方法
Mrthod setName = c1.getDeclaredMethod("setName", String.class);

// invoke: 激活的意思
// (对象, “方法的值”)
setName(user3, "李四");
System.out.println(user3.getName());

// 通过反射操作属性
User user4 = (User) c1.newInstance();
Field name = c1.getDeclaredField("name");

// 不能直接操作私有属性，需要关闭程序的安全检测，属性或者方法的setAccessible(true)
name.setAccessible(true);
name.set(user4, "王五");
System.out.println(user4.getName());
```

反射操作泛型

- Java 采用泛型擦除的机制来引入泛型，Java 中的泛型仅仅是给编译器 javac 使用的，确保数据的安全性和免去强制类型转换的问题，但是，一旦编译完成，所有和泛型有关的类型全部擦除
- 为了通过反射操作这些类型，Java 新增了 ParameterizedType，GenericArrayType，TypeVariable 和 WildcardType 几种类型来代表不能被归一到 Class 类中的类型但是又和原始类型齐名的类型

- ParameterizedType：表示一种参数化类型，比如：`Collection<String>`
- GenericArrayType：表示一种元素类型是参数化类型或者类型变量的数组类型
- TypeVariable：是各种类型变量的公共父接口
- WildcardType：代表一种通配符类型表达式