注解

简介

1. 不进入程序编译，仅可以对程序作出解释（这方面和注释 comment 很像）
2. 可以被其他程序，例如编译器读取

格式

- 注释是以@注释名在代码中存在的，还可以添加一些参数值。

```
@SuppressWarnings(value = "unchecked")
```

使用方式

- 可以被附加在**package 包**，**class 类**,**method 方法**和**field 字段**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素的访问。

内置注解

1. @Override

- 定义在**java.lang.Override**中，此注释只适用于修辞方法，表示一个方法声明打算重写超类中的另一个方法声明。

2. @Deprecated

- 定义在**java.lang.Deprecated**中，此注释可以用于修饰方法，类，属性；但是这个注解表示不鼓励程序员使用，通常因为危险问题或存在更好的选择。如果你想要使用，也是可以的。

3. @SupressWarnings

- 需要添加参数才能使用，这些参数都是已经定义好的，选择性的使用就行了。

```
import java.util.ArrayList;

```

```java
public class Annotation extends Object {
    // 重写了父类已经写好的 toString() 方法
    @Override
    public String toString(){
        return super.toString();
    }

    // 已被废弃，但是仍可以使用；有危险或有更好的替代方案
    @Deprecated
    public static void test(){
        System.out.println("Deprecated");
    }

    // 若不写，会有⚠️；若写“all”，镇压所有警告
    @SuppressWarnings("all")
    public static void test01(){
        ArrayList array = new ArrayList();
    }

    public static void main(String[] args) {
        test();
    }

}

```

元注解

- 元注解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，被用来提供对其他 annotation 类型作说明。

1. @Target

- 用于描述注解的使用范围（被描述的注解可以用在什么地方）

2. @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（runtime > class > source)

3. @Document

- 说明该注解将被包含在 javadoc 中

4. @Inherited

- 说明子类可以继承父类中的该注解

```java
import java.lang.annotation.*;

/**

```

```
 * @author cheng
 * @date 2020-12-18
 */
@myAnnotation
public class meta_annotation {
    @myAnnotation
    public void test(){
        System.out.println("...");
    }
}

/**
 * @Target 中 value 后接一个数组
 * ElementType.XXX 需要在哪边增加注解这里一定要对应上，否则会报错
 *
 * @Retention 中 value = RetentionPolicy.XXX 需要标注这个注解是存在于哪一种运行的生命周期
 * 1. CLASS 类中
 * 2. SOURCE 源代码中
 * 3. RUNTIME 运行时
 * RUNTIME > CLASS > SOURCE
 *
 * @Documented 表示是否将以下内容生成在 javadoc 文档中
 *
 * @Retention 表示子类可以继承父类的注解
 *
 * @Interface 表示自定义一个新注解
 */

@Target(value = {ElementType.METHOD,ElementType.TYPE})
@Retention(value = RetentionPolicy.RUNTIME)
@Documented
@Inherited
@interface myAnnotation{

}

```

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。

1. @interface 用来生命一个注解，格式 public @interface 注解名{code}
2. 其中的每一个方法实际上是声明了一个配置参数
3. 方法的名称就是参数的名称
4. 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）
5. 可以通过 default 来声明参数的默认值
6. 如果只有一个参数成员，一般参数名为 value
7. 注解元素必须要有值，我们定义注解元素时，经常使用空字符串，0 作为默认值。

```java
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * @author cheng
 * @date 2020-12-18
 * @description 如何自定义一个新注解
 */
public class Annotation01 {
    // 若下面 String name() 不加 default "",上面必须要加 school = "NEU"
    @myAnnotation01(schools = "NEU")
    public void test(){

    }
}

// TYPE 类 METHOD 方法上均可以加载注解
// RUNTIME 在运行时有效
@Target(value = {ElementType.TYPE, ElementType.METHOD})
@Retention(value = RetentionPolicy.RUNTIME)
@interface myAnnotation01{
    // 注解的参数: 参数类型 + 参数名
    // 如果只有一个参数，建议使用 value 命名；在上面可以省略 value = XXX；若是其他的参数必须加上 参数名 =
    String name() default "";
    int age() default 0;
    int id() default  -1;

    String[] schools();

}
```

反射

静态 vs 动态语言

- 运行时可以改变其结构的语言：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。
- Object-C,C#,JS,PHP，Python 等
- 运行结构不可变的就是静态语言，例如 Java，C，CPP
- Java 不是动态语言，但是 Java 可以被称之为准动态语言。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 的动态性让编程更加灵活。

介绍

> 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间借助于 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以及方法。

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

- 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，所以我们形象地称之为：反射。
  - 正常方法：引入需要的“包类”名词->通过 new 实例化->取得实例
  - 反射方式：实例化对象->getclass()方法->得到完整的“包类”名称

反射机制提供的功能

1. 在运行时判断任意一个对象所属的类
2. 在运行时构造任意一个类的对象
3. 在运行时判断任意一个类所具有的成员变量和方法
4. 在运行时获取泛型信息
5. 在运行时调用任意一个对象的成员变量和方法
6. 在运行时处理注解
7. 生成动态代理
8. ...

优缺点

优点

- 可以实现动态创建对象和编译，体现出非常强的灵活性

缺点

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

反射相关的主要 API

1. java.lang.Class: 代表一个类
2. java.lang.reflect.Method: 代表类的方法
3. java.lang.reflect.Field: 代表类的构造器
4. ...

```java
/**
 * @author cheng
 * @date 2020-12-18
 */
public class Reflection {
    public static void main(String[] args) throws ClassNotFoundException {
        @SuppressWarnings("all")
        // 这里是因为没有包，所以直接写方法名即可；若有包先写包 com.example.demo.User
        Class c =  Class.forName("User");
        System.out.println(c);

        Class c1 =  Class.forName("User");
        Class c2 =  Class.forName("User");
        // 一个类被加载后，类的整个结构都会被封装在 Class 对象中。
        if(c1.hashCode() == c2.hashCode()){
            System.out.println("一个类在内存中只有一个 Class 对象");
        }
    }
}

// 实体类：POJO 或者叫 entity
class User{
    private String name;
    private int age;
    private int id;

    public User(){}

    public User(String name,int id, int age){
        this.name = name;
        this.id = id;
        this.age = age;
    }
    public String getName(){return name;}
    public void setName(String name){this.name = name;}

    public int getId(){return id;}
    public void setId(int id){this.id = id;}

    public int getAge(){return age;}
    public void setAge(int age){this.age = age;}
}

```

Class 类

- 在 Object 类中定义了很多方法，这些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名词。
- 对象反射后可以得到的信息：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 都为其保留了一个不变的 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

1. Class 本身也是一个类
2. Class 对象只能由系统建立对象
3. 一个加载的类在 JVM 中只会有一个 Class 实例
4. 一个 Class 对象对应的是一个加载到 JVM 中的一个 .class 文件
5. 每个类的实例都会记得自己是由哪个 Class 实例所生成
6. 通过Class 可以完整地得到一个类中的所有被加载的结构。
7. Class 类是 Reflection 的根源，针对任何我们想动态加载，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 ClassforName(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()	的返回一个包含某些 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

1. 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 clazz = Person.class;
```

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

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

3. 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，可能会抛出 ClassNotFoundException 异常

```
// 包.文件.类
 Class clazz = Class.forName("com.example.demo.Student");
```

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

```
/**
 * @author cheng
 * @date 2020-12-18
 * @description 创建 Class 类的方式有哪些?
 */
public class CreateClass {
    public static void main(String[] args) throws ClassNotFoundException {
        Person p1 = new Student();
        Person p2 = new Teacher();

        System.out.println("p1: " + p1.name);
        System.out.println("p2: " + p2.name);

        // 方式 1：通过对象来获得 Class
        Class c1 = p1.getClass();
        System.out.println(c1.hashCode());
        // 方式 2：通过.forName()来获得 Class
        Class c2 = Class.forName("Student");
        System.out.println(c2.hashCode());
        // 方式 3：通过类名.class 获得 Class
        Class c3 = Student.class;
        System.out.println(c3.hashCode());
        // 方式 4：基本内置类型的包装类都有一个 Type 属性
        Class c4 = Integer.TYPE;
        System.out.println(c4);

        // 获得父类类型
        Class c5 = c1.getSuperclass();
        System.out.println(c5);

```

```
    }
}

class Person{
    public String name;

    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' +
                '}';
    }
}

class Student extends Person{
    public Student(){
        this.name = "张三";
    }
}

class Teacher extends Person{
    public Teacher(){
        this.name = "李四";
    }
}

```

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。(Class 本身也是一个类)
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

```
import java.lang.annotation.ElementType;

/**
 * @author cheng
 * @date 2020-12-19
 * @description 哪些类型可以有 Class 对象?
 */
public class OOPClass {
    public static void main(String[] args) {
        // 1. 类的 Class
        Class c1 = Object.class;
```

```
        // 2. 接口的 Class
        Class c2 = Comparable.class;
        // 3. 一维数组的 Class
        Class c3 = String[].class;
        // 3.0 二维数组的 Class
        Class c3_0 = int[][].class;
        // 4. 注解类型的 Class
        Class c4 = Override.class;
        // 5. 枚举类型的 Class
        Class c5 = ElementType.class;
        // 6. 基本数据类型的 Class
        Class c6 = Integer.class;
        // 7. void 的 Class
        Class c7 = void.class;
        // 8.  Class 本身也是一个类，Class 的 class
        Class c8 = Class.class;

        System.out.println(c1);
        System.out.println(c2);
        System.out.println(c3);
        System.out.println(c3_0);
        System.out.println(c4);
        System.out.println(c5);
        System.out.println(c6);
        System.out.println(c7);
        System.out.println(c8);

        // 判断不同大小的同类型数组是否拥有同一个 Class
        int[] a = new int[10];
        int[] b = new int[100];
        if(a.getClass().hashCode() == b.getClass().hashCode()){
            System.out.println("只要元素类型与维度一样，就是一个 Class");
        }else{
            System.out.println("这句话不可能出现！");
        }

    }
}
```

Java 内存分析

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。

1. 类的加载（Load）
   - 将类的 class 文件读入内存，并为之创建一个 java.lang.Class 对象。
   - 此过程由类加载器完成。
2. 类的链接（Link）
   - 将类的二进制数据合并到 JRE 中
3. 类的初始化（Initialize）
   - JVM 负责对类进行初始化

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，并将这些景泰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

1. 验证
   - 确保加载的类信息符合 JVM 规范，没有安全方面的问题。
2. 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中进行分配。
3. 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杯正确地加锁和同步。

类是如何被加载的?

```
/**
 * @author cheng
 * @date 2020-12-19
 * @description 类是如何被加载的?
 *
 */
public class LoadClass {
    public static void main(String[] args) {

        A a = new A();
        System.out.println(A.m);
    }
}

class A{
    static{
        System.out.println("A 类静态代码初始化");
        m = 300;
    }
    static int m = 100;
    public A(){
        System.out.println("A 类的无参构造初始化");
    }
}
```

什么时候会发生类初始化?

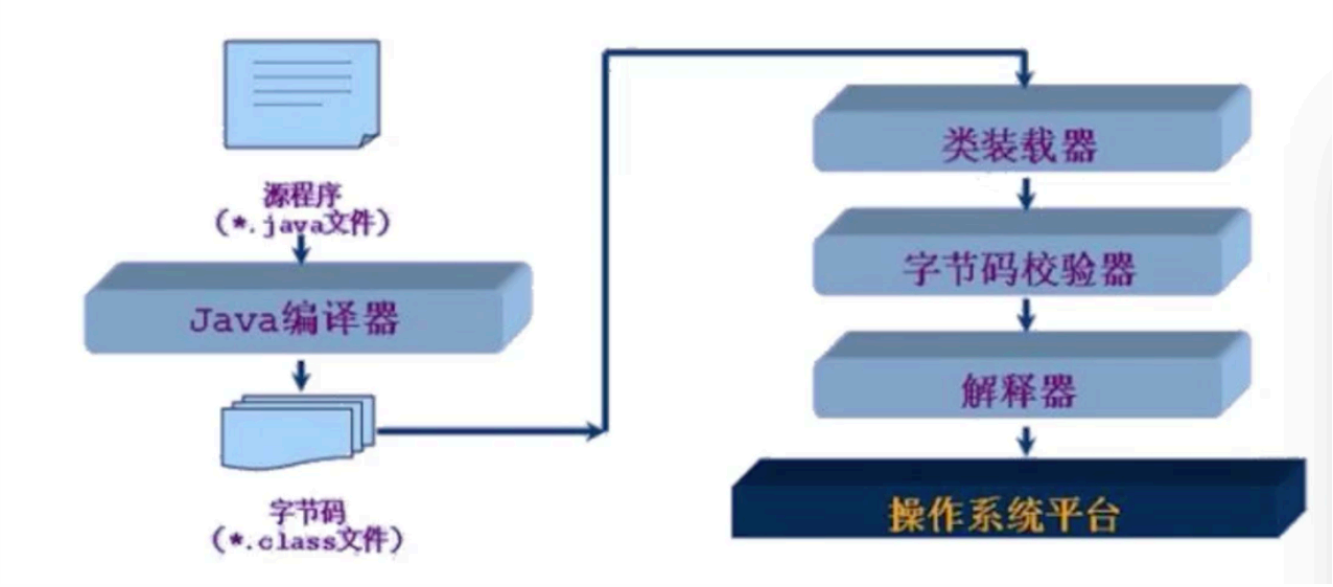
1. 类的主动引用（一定会发生类的初始化）
   - 当虚拟机启动，先初始化 main 方法所在的类
   - new 一个类的对象
   - 调用类的静态成员（除了 final）常量和静态方法
   - 使用 java.lang.reflect 包的方法对类进行反射调用
   - 当初始化一个类，如果其父类没有被初始化则先会初始化其父类
2. 类的被动引用（不会发生类的初始化）
   - 当访问一个静态域时，只有真正声明这个域的类才会被初始化。如：当通过子类引用父类的静态变量，不会导致子类初始化
   - 通过数组定义类引用，不会触发此类的初始化
   - 引用常量不会触发此类的初始化（常量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

```
/**
```

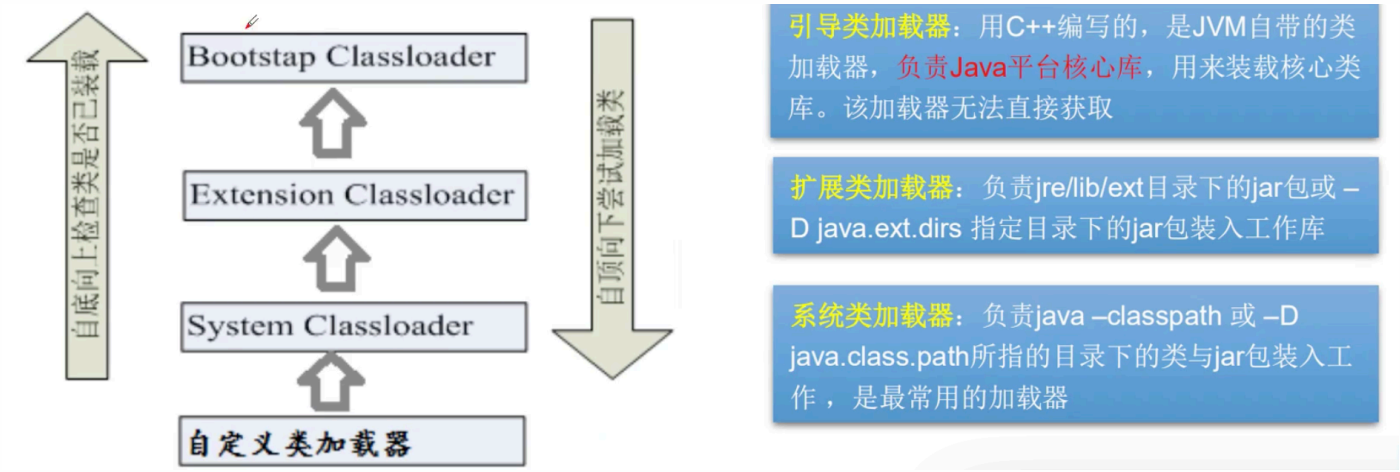
```java
 * @author cheng
 * @date 2020-12-21
 * @description 类什么时候会被初始化?
 */
public class ClassInit {
    static {
        System.out.println("main 类静态代码块被加载");
    }

    public static void main(String[] args) throws ClassNotFoundException {
        // 1. 类的主动引用
        //Son s = new Son();
        // 2. Class.forName() 反射;抛出找不到 Class 的异常
        //Class.forName("Son");

        // 不会产生类的使用方法
        // 1. 子类引用父类的静态变量，不会导致子类初始化
        //System.out.println(Son.f);
        // 2.通过数组定义类引用，不会触发此类的初始化
        //Son[] array = new Son[10];
        // 3.引用常量不会触发此类的初始化
        System.out.println(Son.F);
    }
}

class Father{

    static int f = 4;
    static {
        System.out.println("父类静态代码块被加载");
    }
}

class Son extends Father{
    static {
        System.out.println("子类静态代码块被加载");
        s = 3;
    }
    static int s = 1;
    static final int F = 2;
}

```

类加载器的作用

- 类的加载作用
  - 将 class 文件字节码内容加载到内存中，并将这些景泰数据转换成方法区的运行时数据结构，然后再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得访问入口。
- 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，他将维持加载（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。




```java
/**
 * @author cheng
 * @date 2020-12-21
 * @description 获得系统类加载器
 */
public class GetSysClassLoader {
    public static void main(String[] args) throws ClassNotFoundException {
        // 获得系统类加载器
        ClassLoader sysCL = ClassLoader.getSystemClassLoader();
        System.out.println("获取系统类加载器：" + sysCL);

        // 获得系统类加载器的父类加载器 --> 扩展类加载器
```

```
        System.out.println(sysCL.getParent());

        // 获得西戎类加载器的父类加载器的加载器 --> 根加载器 (C/C++写的)
        // 根加载器是核心类库加载器，无法直接获取
        System.out.println(sysCL.getParent().getParent());

        // 测试当前类是哪一个加载器加载的(注意 main 要抛出 ClassNotFoundException 异常)
        ClassLoader current_Class_loader =
Class.forName("GetSysClassLoader").getClassLoader();
        System.out.println(current_Class_loader);
        // 测试 JDK 内置类是由什么加载的
        ClassLoader JDK_Class_loader =
Class.forName("java.lang.Object").getClassLoader();
        System.out.println(JDK_Class_loader);

        // 如何获得系统类加载器的路径
        System.out.println(System.getProperty("java.class.path"));

        /*
            /Library/Java/JavaVirtualMachines/jdk1.8.0_171.jdk/Contents/

 Home/jre/lib/charsets.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_171.jdk/Cont
ents
            ...还有就不复制了
         */
    }
}
```

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
- 注解
- ...

```
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;

/**
 * @author cheng
 * @date 2020-12-21
 * @description 获取类的信息结构
```

```
 */
public class GetClassStructure {
    public static void main(String[] args) throws ClassNotFoundException, NoSuchFieldException, NoSuchMethodException {
        Class c1 = Class.forName("User");
        // 下面两句话作用等同于上面一句话
        User user = new User();
        Class c2 = user.getClass();
        System.out.println(c1.getName());// 因为缺少包，这里只有 User 的名字，若有包，例:com.example.demo.User
        System.out.println(c1.getSimpleName());
        System.out.println("-----------------------");
        System.out.println(c2.getName());// 因为缺少包，这里只有 User 的名字，若有包，例:com.example.demo.User
        System.out.println(c2.getSimpleName());

        // 获得类的属性
        //System.out.println(c1.getFields());

        Field[] fields = c1.getFields();// 只能得到 public 属性，private 无法得到
//        for(Field field:fields){
//            System.out.println(field);
//        }

        fields = c1.getDeclaredFields();// 得到全部属性
        for(Field field:fields){
            System.out.println(field);
        }

        // 获得指定属性的值
        //Field name = c1.getField("name");// 会报错找不到，因为只能找公有属性
        Field name = c1.getDeclaredField("name");
        System.out.println("=======================\n" + name);

        // 获得类方法
        System.out.println("=======================");
        // 获得本类及其父类的 public 方法
        Method[] methods = c1.getMethods();
        for(Method method:methods){

            System.out.println("得到public getMethods：" + method);
        }
        // 获得本类的全部方法
        methods = c1.getDeclaredMethods();
        for(Method method:methods){
            System.out.println("得到全部 getDeclaredMethods：" + method);
        }

        // 获得指定方法
        // 重载
        Method getName = c1.getMethod("getName",null);
        System.out.println(getName);
```

```
58          Method setName = c1.getMethod("setName",String.class);
59          System.out.println("setName = " + setName);
60
61          // 获得构造器
62          System.out.println("======================");
63          // 获得所有构造器
64          Constructor[] constructors = c1.getConstructors();
65          for(Constructor constructor:constructors){
66              System.out.println("constructor [public] = " + constructor);
67          }
68          constructors = c1.getDeclaredConstructors();
69          for(Constructor constructor:constructors){
70              System.out.println("constructor [all] = " + constructor);
71          }
72
73          // 获得指定构造器
74          Constructor declaredConstructor =
    c1.getDeclaredConstructor(String.class,int.class,int.class);
75          System.out.println("获得指定构造器 declaredConstructor = " +
    declaredConstructor);
76      }
77  }
```

小结

- 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。
- 一定要熟悉 java.lang.reflect 包的作用，反射机制。
- 知道如何获得属性，方法，构造器的名词，修饰符等。

有了 Class 对象，能做什么?

- 创建类的对象：调用 Class 对象的 newInstance()方法
  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
- 问：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。
  1. 通过 Class 类的 getDeclaredConstrutcor（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

调用指定的方法

- 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 obj 对象的参数信息。

invoke()

> 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，显示调用方法对象的 setAccessible(true)方法，将可访问 private 的方法。

setAccessible()

- Method 和 Field，Constructor对象都有 setAccessible()方法。
- setAccessible作用是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地被调用，那么需要被设置为特
  - 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。

```
1  import java.lang.reflect.Constructor;
2  import java.lang.reflect.Field;
3  import java.lang.reflect.InvocationTargetException;
4  import java.lang.reflect.Method;
5
6  /**
7   * @author cheng
8   * @date 2020-12-21
9   * @description 如何动态地创建对象?
10  */
11 public class DynamicNew {
12     public static void main(String[] args) throws ClassNotFoundException,
   IllegalAccessException, InstantiationException, NoSuchMethodException,
   InvocationTargetException, NoSuchFieldException {
13         // 获得 Class 对象
14         Class c1 = Class.forName("User");
15
16         // 构造一个对象
17         //User user = (User)c1.newInstance();// 调用无参构造器
```

```
        //System.out.println("user = " + user);

        // 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",002,23);
//        System.out.println(user2);

        // 通过反射调用方法
        User user3 = (User)c1.newInstance();
        // 通过反射获取一个方法
        Method setName = c1.getDeclaredMethod("setName", String.class);
        // invoke：激活    xxx.invoke(对象，"方法的值");
        setName.invoke(user3,"李四");
        System.out.println("user3.getName() = " + user3.getName());

        // 通过反射操作属性
        System.out.println("------------------------------");
        User user4 = (User)c1.newInstance();
        Field name = c1.getDeclaredField("name");
        // 若没设置 setAccessible()，不能直接操作私有属性。
        // 强行反射：简称强吻
        name.setAccessible(true);
        name.set(user4,"李四他哥");
        System.out.println("user4.getName() = " + user4.getName());
    }

}

```

setAccessible

- Method 和 Field，Constructor 对象都有 setAccessible() 方法。
- setAccessible 作用是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。
  - 提高反射效率。如果代码中必须使用反射，而该代码需要频繁地被调用，那么设置为 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

```
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

/**
 * @author cheng
 * @date 2020-12-21
 * @description 分析性能问题
 */
```

```java
public class AnalyzeCapability {
    // 通过普通方法创建对象
    public static void test01(){
        User user = new User();
        long start_time = System.currentTimeMillis();
        for (int i = 0; i < 100000000; i++) {
            user.getName();
        }
        long end_time = System.currentTimeMillis();
        System.out.println("普通方式执行 1亿次用时：" +(end_time - start_time)+ "ms");
    }


    // 通过反射方法创建对象
    public static void test02() throws NoSuchMethodException, InvocationTargetException, IllegalAccessException, InstantiationException {
        //User user = new User();
        Class c1 = User.class;
        User user =(User)c1.newInstance();
        Method getName = c1.getDeclaredMethod("getName",null);

        long start_time = System.currentTimeMillis();
        for (int i = 0; i < 100000000; i++) {
            getName.invoke(user,null);
        }
        long end_time = System.currentTimeMillis();
        System.out.println("反射方法创建对象执行 1亿次用时：" +(end_time - start_time)+" ms");
    }



    // 通过反射方法（关闭检测）创建对象
    public static void test03() throws InvocationTargetException, IllegalAccessException, NoSuchMethodException {
        User user = new User();
        Class c1 = user.getClass();

        Method getName = c1.getDeclaredMethod("getName",null);
        getName.setAccessible(true);
        long start_time = System.currentTimeMillis();
        for (int i = 0; i < 100000000; i++) {
            getName.invoke(user,null);
        }
        long end_time = System.currentTimeMillis();
        System.out.println("关闭检测执行 1亿次用时：" +(end_time - start_time)+ "ms");
    }


```

```
    public static void main(String[] args) throws NoSuchMethodException, IllegalAccessException, InvocationTargetException, InstantiationException {
        test01();
        test02();
        test03();
    }
}
```

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，确保数据得安全性和免去强制类型转换问题，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
  - 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

```
import java.lang.reflect.Method;
import java.lang.reflect.ParameterizedType;
import java.lang.reflect.Type;
import java.util.List;
import java.util.Map;

/**
 * @author cheng
 * @date 2020-12-21
 * @description 通过反射获取泛型
 */
public class ReflectEnum {
    // 用泛型传参
    public void test01(Map<String,User> map, List<User> list){
        System.out.println("test01");
    }

    // 返回值的泛型
    public Map<String,User> test02(){
        System.out.println("test02");
        return null;
    }

    public static void main(String[] args) throws NoSuchMethodException {
        // 通过注解获得泛型
        Method method = ReflectEnum.class.getMethod("test01", Map.class, List.class);
        Type[] getGenericParameterTypes = method.getGenericParameterTypes();
        for (Type getGenericParameterType:getGenericParameterTypes){
```

```
            if(getGenericParameterType instanceof ParameterizedType){
                Type[] actualTypeArguments = ((ParameterizedType)
getGenericParameterType).getActualTypeArguments();
                for(Type actualTypeArgument:actualTypeArguments){
                    System.out.println("actualTypeArgument = " +
actualTypeArgument);
                }
            }
            //System.out.println("getGenericParameterType = " +
getGenericParameterType);
        }
        method = ReflectEnum.class.getMethod("test02",null);
        Type returnType = method.getGenericReturnType();
        if(returnType instanceof ParameterizedType){
            Type[] actualTypeArguments = ((ParameterizedType)
returnType).getActualTypeArguments();
            for(Type actualTypeArgument:actualTypeArguments){
                System.out.println("actualTypeArgument = " + actualTypeArgument);
            }
        }
    }
}

```

获取注解信息

反射操作注解

- getAnnotations
- getAnnotation

ORM

Object relationship Mapping: 对象关系映射

```
class Student{
    int id;
    String name;
    int age;
}
```

id	name	age
001	张三	18
002	李四	21

- 类和表结构对应
- 属性和字段对应
- 对象和记录对应

==利用注解和反射完成类和表结构的映射关系==

```
import jdk.nashorn.internal.ir.annotations.Reference;

import java.lang.annotation.Annotation;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.lang.reflect.Field;

/**
 * @author cheng
 * @date 2020-12-21
 * @description 利用注解和反射完成类和表结构的映射关系(反射操作注解)
 */
public class OrmPractice {
    public static void main(String[] args) throws ClassNotFoundException,
NoSuchFieldException {
        Class c1 = Class.forName("Student1");

        // 通过反射获得注解
        Annotation[] annotations = c1.getAnnotations();
        for (Annotation annotation : annotations) {
            System.out.println("annotation = " + annotation);
        }

        // 获得注解的 value 的值
        Table table = (Table)c1.getAnnotation(Table.class);
        String value = table.value();
        System.out.println("value = " + value);

        // 获得类指定的注解
        Field name = c1.getDeclaredField("name");
        FieldAnnotation annotation = name.getAnnotation(FieldAnnotation.class);
        System.out.println("annotation.columnName() = " +
annotation.columnName());
        System.out.println("annotation.columnName() = " + annotation.type());
        System.out.println("annotation.columnName() = " + annotation.length());


    }
}

@Table("db_student")
class Student1{
    @FieldAnnotation(columnName = "db_id",type="int",length = 10)
    private int id;
```

```
    @FieldAnnotation(columnName = "db_name",type="varchar",length = 3)
    private String name;
    @FieldAnnotation(columnName = "db_age",type="int",length = 10)
    private int age;


    Student1(){}

    Student1(int id,String name,int age){
        this.id = id;
        this.name = name;
        this.age = age;
    }
    @Override
    public String toString() {
        return "Student1{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", age=" + age +
                '}';
    }

    public int getId() {
        return id;
    }

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

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}

// 类名的注解
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
/**
 * @author cheng
```

```
 */
@interface Table{
    String value();
}

// 属性的注解
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)

/**
 * @author cheng
 */
@interface FieldAnnotation{
    String columnName();
    String type();
    int length();
}
```