Lecture13-2 反射与泛型

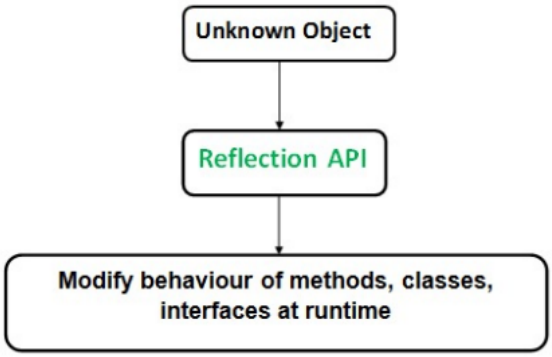
1. 反射

介绍

https://www.baeldung.com/java-reflection

反射是一种API，用于在运行时检查或修改方法、类和接口的行为

- 反射所需的类在 `java.lang.reflect` 包中提供
- 反射给我们提供了关于对象所属的类的信息，以及可以使用该对象执行的类的方法
- 通过反射，我们可以在运行时调用方法，而不用考虑与它们一起使用的访问说明符



- `Class` 对象会知道它的
  - 父类类型
  - 接口类型
  - 所在的包位置
  - 字段的名字和类型
  - 方法的名称，参数数量和类型，返回值
  - 构造器所需的参数数量和类型

```
Class getSuperclass ()
Class[] getInterfaces ()
Package getPackage ()
Field[] getDeclaredFields ()
Constructor[] getDeclaredConstructors ()
Method[] getDeclaredMethods ()
```

使用

枚举字段

- 枚举 `Math` 所有的静态字段

```
Field[] fields = Math.class.getDeclaredFields();
for (Field f : fields)
   if (Modifier.isStatic( f.getModifiers() ))
      System.out.println( f.getName() );
```

枚举构造器

- 枚举 `Rectangle` 构造器的名字和参数类型

```
for (Constructor c : cons) {
   Class[] params = cc.getParameterTypes();
   System.out.print( "Rectangle(" );
   boolean first = true;
   for (Class p : params) {
      if (first) first = false;
      else System.out.print( ", " );
      System.out.print( p.getName() );
   }
   System.out.println( ")" );
}
```

输出如下

```
Rectangle()
Rectangle( java.awt.Rectangle )
Rectangle( int, int, int, int )
Rectangle( int, int )
Rectangle( java.awt.Point, java.awt.Dimension )
Rectangle( java.awt.Point )
Rectangle( java.awt.Dimension )
```

获取一个简单的方法描述器

- 提供方法的名字
- 提供方法所需的参数类型

例如：获取 `Rectangle.contains( int, int )`

```
Method m = Rectangle.class.getDeclaredMethod(
   "contains", int.class, int.class
);
```

例如：获取 `Rectangle` 的默认构造器

```
Constructor c = Rectangle.class.getDeclaredConstructor();
```

- `getDeclaredMethod()`，`getDeclaredConstructor()` 是可变参数方法

调用一个方法

调用 `System.out.println( "Hello World ")`

```
Method m = PrintStream.class.getDeclaredMethod(
   "println", String.class
);
m.invoke( System.out, "Hello, World!" );
```

- `invoke()` 是可变参数方法

获得一个对象

- 可以在运行的时候获得一个对象，对通用调试工具有用
- 需要获得访问私有字段的权限

```
Class c = obj.getClass();
Field f = c.getDeclaredField( name );
f.setAccessible( true );
```

- 如果安全管理器禁止访问，则抛出异常

设置字段的值

```
Object value = f.get( obj );
f.set( obj, value );
```

- 对于基本数据类型，使用包装类

观察数组元素

使用 `Array` 类的静态方法

```
Object value = Array.get( a, i );
Array.set( a, i, value );
int n = Array.getLength( a );
```

构造一个新的数组

```
Object a = Array.newInstance( type, length );
```

优缺点

优点

- 可扩展性：应用程序可以通过使用扩展对象的完全限定名称来创建它们的实例，从而使用外部的用户定义类
- Debug 和测试工具：调试器使用反射属性检查类上的私有成员

缺点

- 性能：反射操作比非反射操作的性能要慢，应该避免在性能敏感的应用程序中频繁调用的代码部分中使用反射操作
- 内部暴露：反射代码破坏了抽象，因此可能会随着平台的升级而改变行为

2. 泛型

介绍

- 泛型类型具有一个或多个类型变量
- 类型变量是用类或接口类型实例化的
- 不能使用**基本类型**，例如不存在 `ArrayList<int>`
- 定义泛型类时，在定义中使用类型变量

```
public class ArrayList<E> {
   public E get (int i) { . . . }
   public E set (int i, E newValue) { . . . }
   . . .
   private E[] elementData;
}
```

- 注意：**如果 S 是 T 的子类型，** `ArrayList<S>` **不是** `ArrayList<T>` **的子类型**
- 泛型通常使用 T、K、V、E 表示

泛型类 / 泛型接口

```
public class Order<T>{
    T orderT;
    public Order (T orderT){
        this.orderT = orderT;
    }

    public T getOrderT(){
        return orderT;
    }
}
```

继承

如果有子类继承的话

```
public class SubOrder1 extends Order<Integer>{
    //...
}
```

- 如果继承的时候指明了父类的类型，那么子类就不是泛型类了
- 实例化子类对象时，不再需要指明泛型

```
public class SubOrder2<T> extends Order<T>{
    // ...
}
```

- 如果继承的时候仍然以泛型符号继承，则基类仍然是泛型类
- 实例化子类对象时，需要指明泛型

注意

- 泛型类可能有多个参数，多个参数写在同一个尖括号内，比如 `<E1,E2,E3>`
- 泛型类的构造器如下

  ```
  public GenericClass(){}
  public GenericClass<E>(){} // 这种是错误的
  ```

- 泛型如果不指定，将被**擦除**，泛型对应的类型均按照 `Object` 处理，但不等价于 `Object`
- **静态方法中不能使用类的泛型**
- **异常类不可能是泛型的**
- 注意泛型的数组声明

```
T[] arr = new T[10]; // 编译不通过
T[] arr = (T[]) new Object[10]; // 可以这样
```

泛型方法

在方法中，出现了泛型的结构，泛型参数与类的泛型参数没有直接关系

泛型方法所属的类或者接口是不是泛型的无所谓，泛型方法自己定义了泛型

```
public class Order<T>{
    T orderT;
    public Order (T orderT){
        this.orderT = orderT;
    }

    // 这个不是泛型方法
    public T getOrderT(){
        return orderT;
    }
}
```

- 泛型方法 = 带有类型参数的方法

```
public static <E> void fill ( ArrayList<E> a, E value, int count) {
       for (int i = 0; i < count; i++)
          a.add( value );
}
```

- 泛型方法在调用时，指明泛型的类型
- 泛型方法可以是 `static` 的，泛型参数是在调用方法的时候确定的，而不是在实例化的时候确定的

类型边界

可以用类型界限约束类型变量，约束可以使方法更有用

```
public static <E> void append( ArrayList<E> a, ArrayList<E> b, int count) {
   for (int i = 0; i < count && i < b.size(); i++)
      a.add( b.get( i ) );
}
```

- 上面的方法中，即使 `Rectangle` 继承了 `Shape`，我们也不能将 `ArrayList<Rectangle>` 的内容 append 到 `ArrayList<Shape>` 中

但是，如果有了类型边界

```
public static <E, F extends E> void append( ArrayList<E> a, ArrayList<F> b, int count) {
   for (int i = 0; i < count && i < b.size(); i++)
      a.add( b.get( i ) );
}
```

- `extends` 意味着是**子类型**（例如继承 / 实现）

可以指定多个边界

```
E extends Cloneable & Serializable
```

通配符 Wildcards

上面的描述中，我们从来没有使用过类型 F，可以使用通配符替代

```
<?>
```

- `?`：通配符，表示匹配**任何**类

```
public static <E> void append( ArrayList<E> a, ArrayList<? extends E> b, int count) {
   for (int i = 0; i < count && i < b.size(); i++)
      a.add( b.get( i ) );
}
```

有限制通配符

```
<? extends ClassA>  // ClassA 以及它的子类
<? super ClassA>  // ClassA 以及它的父类
```

```
List<? extends Integer> list; // 任何是 Integer以及它的子类的 list
```

通配符使用示例

以这个实例开始

```java
public static <E extends Comparable<E>> E getMax(ArrayList<E> a) {
   E max = a.get( 0 );
   for (int i = 1; i < a.size(); i++)
      if (a.get(i).compareTo( max ) > 0)
         max = a.get(i);
   return max;
}
```

- 因为 `E` 是 `Comparable<E>` 的子类型，所以我们可以调用 `compareTo()` 方法
- 但是我们不能传入一个 `ArrayList<GregorianCalendar>`
  - 因为它实现的不是 `Comparable<GregorianCalendar>` 而是 `Comparable<Calendar>`
- 如何使用通配符取挽救

```java
public static <E extends Comparable<? super E>> E getMax(ArrayList<E> a) {
   E max = a.get( 0 );
   for (int i = 1; i < a.size(); i++)
      if (a.get(i).compareTo( max ) > 0)
         max = a.get(i);
   return max;
}
```

类型擦除

- 虚拟机不知道泛型类型，泛型信息只存在于编译阶段，在进入 JVM 之前，与泛型相关的信息会被擦除掉，替换为**类型绑定**或 `Object` 类（如果没有绑定）

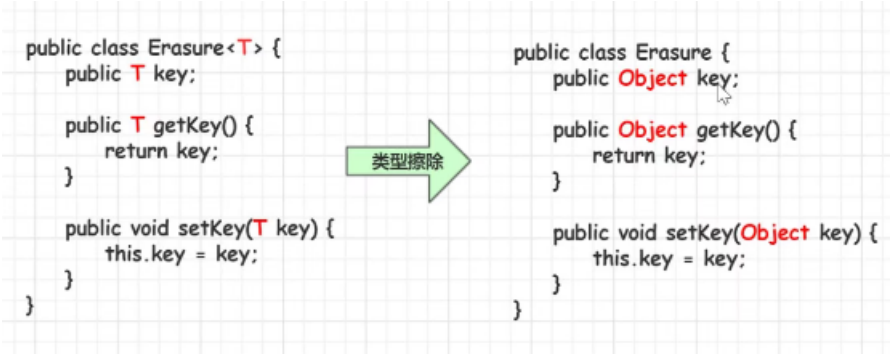
例如 `ArrayList<E>` 会变成

```java
public class ArrayList {
   public Object get (int i) { . . . }
   public Object set (int i, Object newValue) { . . . }
   . . .
   private Object[] elementData;
}
```

例如 `getMax()` 会变成

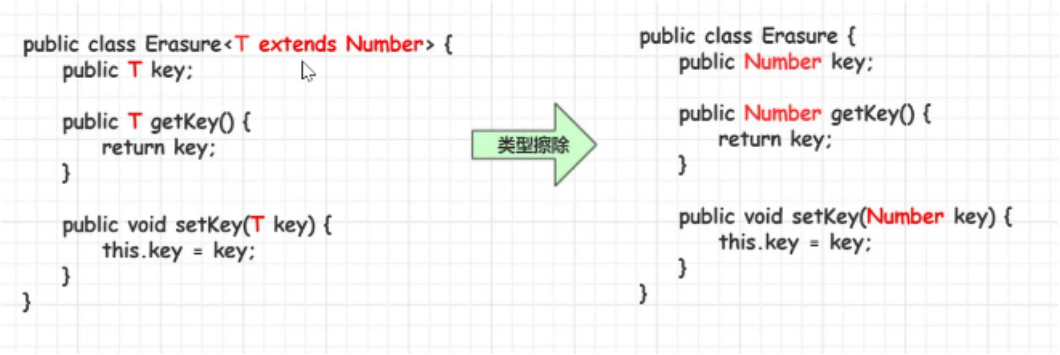
```java
public static Comparable getMax(ArrayList a)
// E extends Comparable<? super E> 被替换成 Comparable
```

无限制类型擦除

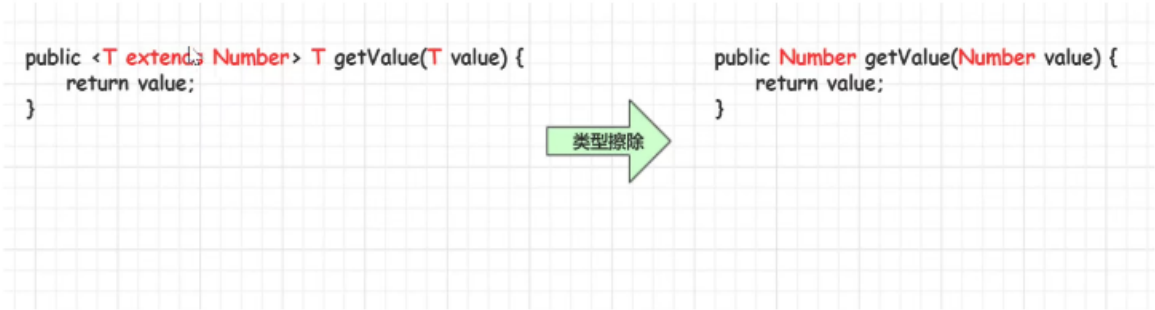


有限制类型擦除

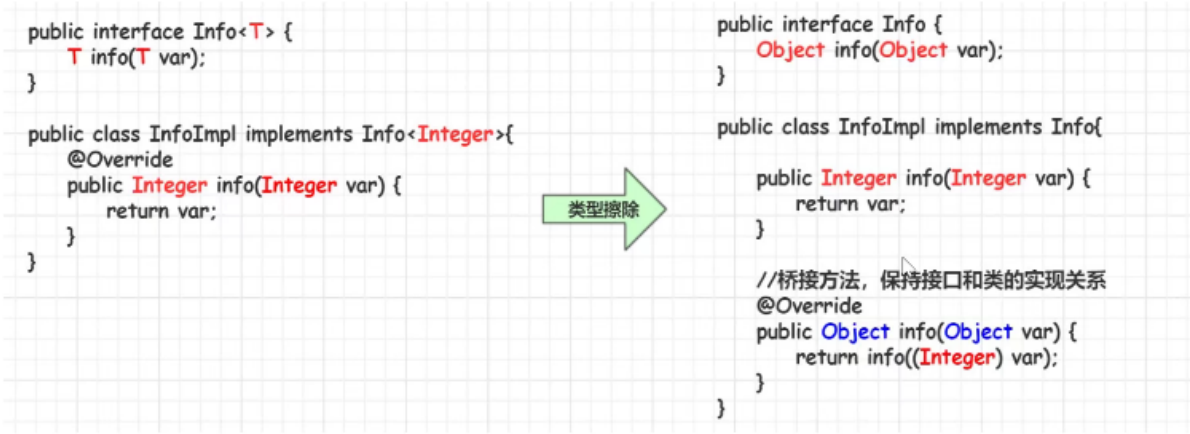
转换成**上限类型**



擦除方法中类型定义的参数



桥接方法



泛型的局限

- 不能用基本类型替换类型变量
- 无法构造泛型类型的新对象
  - `a.add(new E());` 会报错
  - 补救方法，使用反射

```
1  public static <E> void fillWithDefaults( ArrayList<E>, Class<? extends E> cl, int
   count) throws InstantiationException, IllegalAccessException {
2     for (int i = 0; i < count; i++)
3        a.add( cl.newInstance() );
4  }
```

- 调用类似于 `fillWithDefaults( a, Rectangle.class, count )`
- 不能形成参数化类型的数组
  - `Comparable<E>[]` 是不合法的
  - 补救方法：使用 `ArrayList<Comparable<E>>`
- 不能在静态上下文中引用类型参数，静态字段、方法、内部类
- 无法抛出或捕获泛型类型
- 虚拟机替换后不能有类型冲突
  - `GregorianCalendar` 不能实现 `GregorianCalendar`，因为它已经实现了 `GregorianCalendar`，而这些都会被替换成 `Comparable`