

泛型

泛型可解决但我们需要定义这个变量, 这个变量却有很多种类型的时候, 比如学生的成绩, 有些课程的成绩可能是 `int` 直接是数字, 有些课程可能是 `String` 类型, 可能是优秀、良好、中等、等等, 这样就可能需要一个泛型了

我首先应该想到是直接用 `Object` 如果使用 `Object` 类型的话, `Object` 是所有类型的父类, 可以存放 `Integer` 和 `String`, `int` 类型可以通过自动装箱变成 `Integer`, 但是如果直接使用 `Object` 程序在获取值的时候会不知道你使用的是 `String` 还是 `Integer`

代码示例

```
1 //定义了一个成绩类
2 public class Score {
3     String course_name;
4     int course_id;
5     Object value; //这里定义的是Object类
6
7     public Score(String course_name, int course_id, Object value) {
8         this.course_name = course_name;
9         this.course_id = course_id;
10        this.value = value;
11    }
12 }
```

调用

发现这样在需要获取这个值的时候需要进过强制类型转换, 这样就无法在编译期间确定类型是否安全, 如果项目复杂, 代码量过大还会增加不必要的开销;

```
1 public class Main {
2     public static void main(String[] args) {
3         Score score = new Score("数学",1,"100");
4         int num = (int) score.value; //转换
5     }
6 }
```

下面有更好的解决方案

一、泛型类

泛型类就是一个待定的类型, 可以在你不确定它是什么类似的时候就需要使用到, 它在定义时并不会明确它是什么类型, 而是要等到使用的时候才会确定对应的类型, 可以随便使用一个特殊的名字来表示泛型,

代码示例:

★ 使用 <T> 定义一个泛型类，T 表示类型名字，它是可以随意修改的，而且此时 T 可以表示任何类型，在定义的时候也可以将他定义为任何类型，它可以定义多个

```
1 public class Score<T> {
2     String course_name;
3     int course_id;
4     T value; //泛型会更具使用的类型自动变成对应的类型
5
6     public Score(String course_name, int course_id, T value) {
7         this.course_name = course_name;
8         this.course_id = course_id;
9         this.value = value;
10    }
11 }
12
```

调用

```
1     public static void main(String[] args) {
2         Score<String> score = new Score("数学",1,"100");//在类中定义了泛型变量，在定义
           中可以使用到<>来明确需要使用的类型，这里设置为String
3         String value = score.value; //上面定义的是String那么定义的就String
4         System.out.println(value);
5     }
6
7     //输出：
8     100
```

★ 泛型将数据类型的确定控制在了编译阶段，在编写代码的时候就可以明确类型，如果类型不一致就不会通过编译，而且由于是具体对象才会明确类型，所以说静态方法中是不可以用泛型的

1 类型定义

```
1     public static void main(String[] args) {
2         Score<String> score = new Score("数学",1,100); //此时我这里是int类型的100
3         String value = score.value;
4         System.out.println(value);
5     }
```

```
Exception in thread "main" java.lang.ClassCastException Create breakpoint : java.lang.Integer cannot be cast to java.lang.String
at com.company.zy01.Main.main(Main.java:12)
```

2 静态方法中不能使用

```
public class Score<T> {
    0 个用法
    public static void test(){
        T t;
    }
    1 个用法
    String
```

无法从 static 上下文引用 'com.company.zy01.Score.this'

将 'test()' 设为非static Alt+Shift+Enter 更多操作... Alt+Enter

★ 如果未指定变量的形式，那么默认这个变量会是一个 object 类型，因为无论类型具体是什么，都是继承于它，所以说在泛型类中，如果未指定(后面会说)，使用泛型创建的变量会继承 object 的方法

```
public class Score<T> {
    0 个用法
    public void test(T t){
        t.|
    }
    1 个用法
    toString()
```

var T name = expr String

但是可以使用强制类型转换，但是没必要

```
1 public void test(T t){
2     String str = (String) t;    //都明确要用String了，那这里定义泛型不是多此一举吗
3 }
```

★ 不能通过这个不确定的类型变量就直接去创建对象和对应的数组(后面的类型擦除会说到)

```
public class Main {
    public static void main(String[] args) {
        Score<Integer>[] score = new Score<String>[10];
    }
}
```

创建泛型数组

java.lang

public final class String

implements java.io.Serializable, Comparable<String>, CharSequence

★ 具体不同类型的泛型变量，不能使用不同的变量进行接收

```
public class Main {
    public static void main(String[] args) {
        Score<Integer> score = new Score<Integer>("数学", 1, 100);
        Score<String> score1 = score;
    }
}
```

需要的类型: Score <String>

提供的类型: Score <Integer>

将变量 'score1' 的类型更改为 'Score<Integer>' Alt+Shift+Enter 更多操作... Alt+Enter

★ 如果想让某个变量支持引用确定了任何类型的泛型，就可以使用 ? 通配符

```
1 public static void main(String[] args) {
2     Score<?> score = new Score<Integer>("数学", 1, 100); //这里是Integer类型
```

```

3      System.out.println(score.value instanceof Integer);
4      score = new Score<String>("数学",1,"100"); //这里是String类型
5      System.out.println(score.value instanceof String);
6      Object object = score.value; //可以直接赋值给Object类型，由于使用的通配符，具体
    类的类型还是不会确定所以同样会变成Object
7  }
8
9  //输出：
10
11 true
12 true
13 true
14 true

```

★ 泛型也可以定义多个

1 使用,隔开就好了

```

1 public class Test <A,B,C>{
2     A a;
3     B b;
4     C c;
5 }

```

2 如果定义了多个在明确时就需要都进行明确

```

1     public static void main(String[] args) {
2         Test<String,Integer,Character> test = new Test();
3     }

```

★ 在类中都可以使用类型变量，也就也可以创建这个泛型的方法，也可以作为传参传入

```

1 public class Test<T>{
2
3     private T value;
4
5     public void setValue(T value) {
6         this.value = value;
7     }
8
9     public T getValue() {
10        return value;
11    }
12 }

```

★ 泛型只能为一个引用类型，不能为基本类型

```

    public static void main(String[] args) {
        Test<int> test = new Test();
    }

```

类型实参不能为基元类型

1 如果要存放基本类型只可以通过包装类

```
1 public static void main(String[] args) {
2     Test<Integer> test = new Test<>();
3 }
```

2 但是基本类型的数组就可以，因为数组本身就是引用类型

```
1     public static void main(String[] args) {
2         Test<int[]> test = new Test<>();
3     }
```

二、泛型与多态

★ 仅仅是类，泛型还支持多态，也就是支持接口，抽象类都可以

```
1 public interface Study<T> {
2     T study();
3 }
```

★ 当子类实现泛型接口时，可以选择在实现类明确泛型的类型，或者进行使用泛型，让子类也称为泛型类

```
1     public static void main(String[] args) {
2
3     }
4     public class Learn implements Study<String>{ //明确了String类型
5
6         @Override
7         public String study() {
8             return null;
9         }
10    }
```

1 如果不指定默认就是 object

```
1     public static void main(String[] args) {
2
3     }
4     public class Learn implements Study{
5         @Override
6         public Object study() {
7             return null;
8         }
9     }
```

2 还可以继续使用泛型，这样就是这个类也就被定义为泛型类了

```

1      public static void main(String[] args) {
2
3      }
4      public class Learn<T> implements Study<T>{
5          @Override
6          public T study() {
7              return null;
8          }
9      }

```

3 继承也是同样的

三、泛型方法

★ 泛型还可以在方法中使用，前面提到过，当某个方法(无论是静态方法还是成员方法)需要接受的参数类型并不确定也可以使用泛型

```

1      public static void main(String[] args) {
2          String str = test("你好"); //这里自动明确为字符串
3          System.out.println(str);
4      }
5      public static <T> T test(T t){ //这里返回值为泛型，而且传参也是泛型变量表示这就是一个
泛型方法
6          return t;
7      }

```

★ 数组也是可以

```

1      public static void main(String[] args) {
2          String[] strings = new String[1];
3          Main mian = new Main(); //由于不是静态方法所以需要创建一个对象在调用
4          mian.test(strings,"你好");
5          System.out.println(Arrays.toString(strings));
6      }
7      public <T> void test(T[] arr,T t){ //这里返回值为泛型，而且传参也是泛型变量表示这就是
一个泛型方法
8          arr[0] = t;
9      }

```

★ 泛型也在很多工具类也有，比如 `Arrays` 中的排序方法，正常情况下的 `sort` 只能重小到排序，如果我想从大到小排列，其实 `sort` 除了接受基本数据类型，还可以接受一个 `Comparator` 接口，它是一个比较器

```

Arrays.s;
Arra m sort(double[] a) void
m sort(Object[] a) void
m sort(T[] a, Comparator<? super T> c) void

```

```

1 Integer[] arr = {1, 4, 5, 2, 6, 3, 0, 7, 9, 8};
2 Arrays.sort(arr, new Comparator<Integer>() {
3     //通过创建泛型接口的匿名内部类，来自定义排序规则，因为匿名内部类就是接口的实现类，所以说这里
    就明确了类型
4     @Override
5     public int compare(Integer o1, Integer o2) {    //这个方法会在执行排序时被调用（别人
    来调用我们的实现）
6         return 0;
7     }
8 });

```

1 点进去看看可 `Comparator` 接口，会发现它本来就是一个泛型接口，它是一个函数式接口，后面会讲到

```

1 public interface Comparator<T> {
2 }

```

2 `compare()` 是这接口的方法，`o1` 和 `o2` 分别为接收的前一个数和后一个数，需要重写这个方法，重写这个方法就可以实现你自己的想法

```

1 int compare(T o1, T o2);

```

3 它会比较两个数的大小由前面的数减去后面的数，如果返回的值是小于0那么就是前面的数比后面的数小，大于0的数就会true，小于0的数据就会为false，如果大于0就表示前面的数比后面的数要大，通过他的 `return` 来返回，也可以自行修改一下 `return` 将默认的0换成-1试一下

这里我修改 `return` 的值，使用 `o1-o2`，也就是前面的数减去后面的数，这样就是正序排列

```

1     public static void main(String[] args) {
2         Integer[] arr = {1,3,4,5,7,8,6};
3         Arrays.sort(arr, new Comparator<Integer>() {
4             @Override
5             public int compare(Integer o1, Integer o2) {
6                 return o1-o2; //
7             }
8         });
9         System.out.println(Arrays.toString(arr));
10    }
11    //输出: [1, 3, 4, 5, 6, 7, 8]

```

4 如果是倒叙排列的话，这样的话是倒过来

```

1      public static void main(String[] args) {
2          Integer[] arr = {1,3,4,5,7,8,6};
3          Arrays.sort(arr, new Comparator<Integer>() {
4              @Override
5                  public int compare(Integer o1, Integer o2) {
6                      return o2-o1;
7                  }
8          });
9          System.out.println(Arrays.toString(arr));
10     }

```

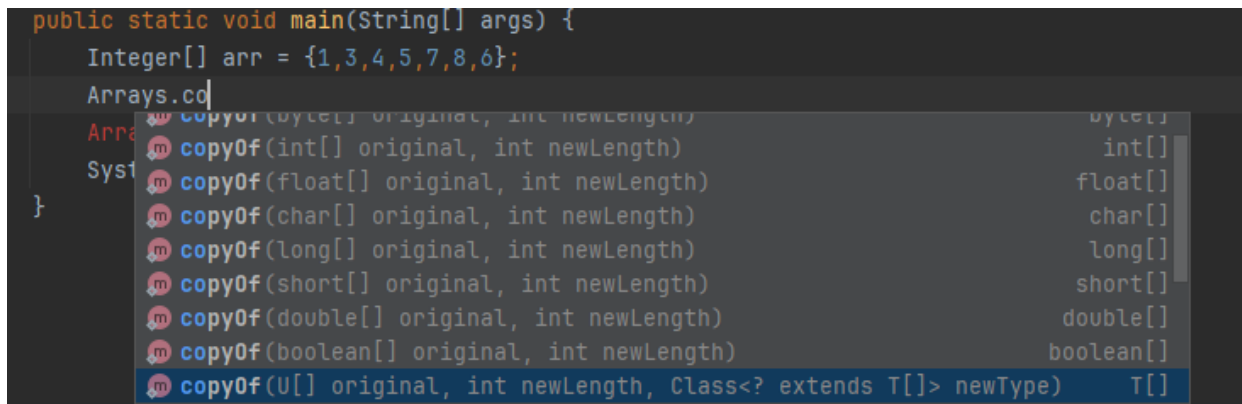
5 这个匿名表达式可以通过 `lambda` 表达式来简化

```

1      public static void main(String[] args) {
2          Integer[] arr = {1,3,4,5,7,8,6};
3          Arrays.sort(arr, (o1, o2) -> o2-o1);
4          System.out.println(Arrays.toString(arr));
5      }

```

★ 数组的复制方法 `copyOf` 也是一样可



```

public static void main(String[] args) {
    Integer[] arr = {1,3,4,5,7,8,6};
    Arrays.co

```

The screenshot shows a code completion menu for the `Arrays.copyOf` method. The menu lists several overloads for different array types: `byte[]`, `int[]`, `float[]`, `char[]`, `long[]`, `short[]`, `double[]`, `boolean[]`, and a generic `copyOf(U[] original, int newLength, Class<? extends T[]> newType) T[]`. The generic version is currently selected.

四、泛型的界限

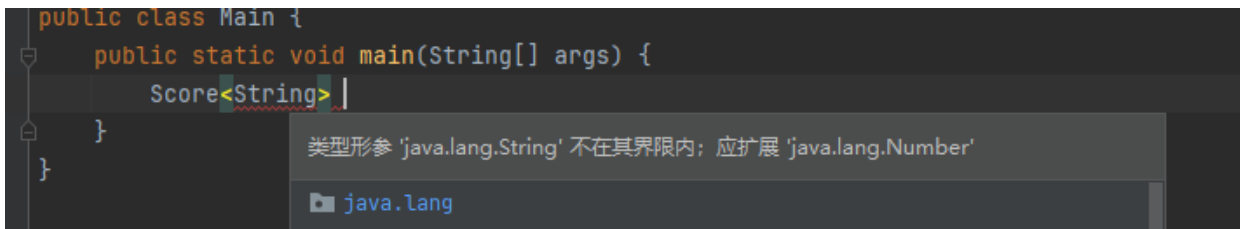
★ 前面提到了泛型默认编译会切换成 `object`，可以是整数也可以是字符串，如果我想要定义这个泛型只能是数字类型不能是字符串类型就可以使用到泛型的上界，上界就是限定泛型的范围，这里我定义了 `Number` 也就是可以使用 `Number` 类和继承它的子类


```

1 public class Score<T extends Number> { //和继承一样的关键字表示上界为Number类型
2     String course_name;
3     int course_id;
4     T value;
5
6     public Score(String course_name, int course_id, T value) {
7         this.course_name = course_name;
8         this.course_id = course_id;
9         this.value = value;
10    }
11 }

```

❶ 如果我定义了一个 String 类型，会直接报错



```

1     public static void main(String[] args) {
2         Score<Integer> score = new Score("数学",1,100); //由于Integer继承与Number
        类，所以说可以定义
3     }

```

★ 通配符也可以实现泛型的界限

```

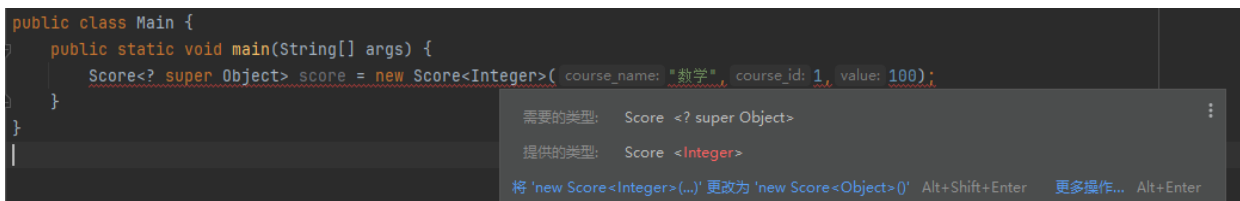
1     public static void main(String[] args) {
2         Score<? extends Integer> score = new Score("数学",1,100);
3     }

```

★ 有上界那么肯定就有下界，下界就是定义不能是定义的这个类型的子类，但是可以是父类

❶ 下界需要通过 super，而且下界只支持通配符？，对于类型变量来说是不支持的

这里我定义的下界为 Object，但是我 new 的对象是 Integer 类型，就会提示类型不匹配



类不能定义下界

```

public class Score<T super Object> {
    1 个用法
    String course_name;
    1 个用法
    int course_id;
    1 个用法
    T value;

    0 个用法
    public Score(String course_name, int course_id, T value) {
        this.course_name = course_name;
        this.course_id = course_id;
        this.value = value;
    }
}

```

★ 限定了上界之后，这个泛型定义的泛型成员，就不会在是 `Object`，除非定义的就是 `Object`，那么它会变成泛型上界定义的那个类型

```

1 public static void main(String[] args) {
2     Score<? extends Number> score = new Score<Integer>("数学",1,100);
3     Number value = score.value; //变成Number类型了
4 }

```

★ 但是如果是限定下界，那么就还有可能是 `Object`

```

1 public static void main(String[] args) {
2     Score<? super Number> score = new Score<>("数据结构与算法基础", "EP074512", 10);
3     Object o = score.getValue();
4 }

```

设置泛型的界限可以更加灵活的控制泛型的具体类型范围

五、类型擦除

★ 实际上在Java中并不是真的有泛型类型（为了兼容之前的Java版本）因为所有的对象都是属于一个普通的类型，一个泛型类型编译之后，实际上会直接使用默认的类型

```

1 //这里是反编译的代码
2 public abstract class A {
3     abstract Object test(Object t); //默认就是Object
4 }

```

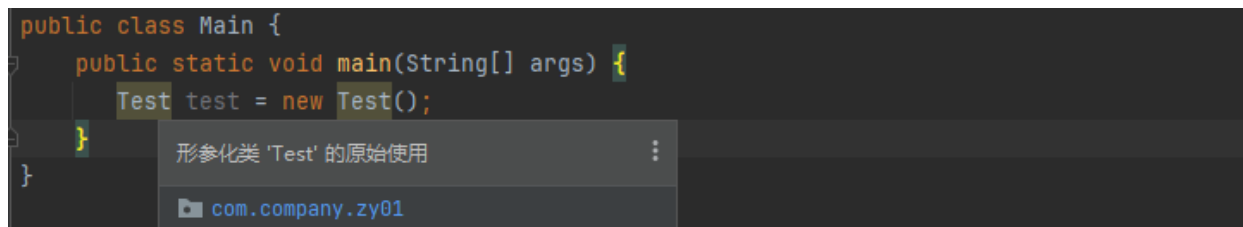
★ 如果给定了上界的话，那么默认类型就会变成上界定义的类型

```

1 public abstract class A <T extends Number>{    //设定上界为Number
2     abstract T test(T t);
3 }
4
5
6 ////这里是反编译的代码
7 public abstract class A {
8     abstract Number test(Number t);    //上界Number，因为现在只可能出现Number的子类
9 }
10

```

★ 泛型只是在编译阶段的时候进行类型检测，在程序运行的时候并不会去检查对应类型，你不指定类型都可以，只不过编译器就会警告



★ 由于类型擦除机制的存在，实际上我们在使用的時候，编译后的代码是进行了强制转换的

```

1 //这次编写的代码
2 public static void main(String[] args) {
3     A<String> a = new B();
4     String i = a.test("10");    //因为类型A只有返回值为原始类型Object的方法
5 }
6
7
8 //这是编译后的代码
9 public static void main(String[] args) {
10     A a = new B();
11     String i = (String) a.test("10");    //依靠强制类型转换完成的
12 }

```

❓ 不过，我们思考一个问题，既然继承泛型类之后可以明确具体类型，那么为什么 `@Override` 不会出现错误呢？我们前面说了，重写的条件是需要和父类的返回值类型和形参一致，而泛型默认的原始类型是 `Object` 类型，子类明确后变为其他类型，这显然不满足重写的条件，但是为什么依然能编译通过呢？

1 这里定义了一个类继承自 `A` 类，`A` 类是一个泛型类

```

1 public class B extends A<String>{
2     @Override
3     String test(String s) {
4         return null;
5     }
6 }

```

2 编译后

```

1 // Compiled from "B.java"
2 public class com.test.entity.B extends com.test.entity.A<java.lang.String> {
3     public com.test.entity.B();
4     java.lang.String test(java.lang.String);
5     java.lang.Object test(java.lang.Object);    //桥接方法，这才是真正重写的方法，但是使用
        时会调用上面的方法
6 }

```

3 使用反编译观察

```

1 public class B extends A {
2
3     public Object test(Object obj) {    //这才是重写的桥接方法
4         return this.test((Integer) obj);    //桥接方法调用我们自己写的方法
5     }
6
7     public String test(String str) {    //我们自己写的方法
8         return null;
9     }
10 }

```

类型擦除机制其实就是为了方便使用后面集合类（不然每次都要强制类型转换）同时为了向下兼容采取的方案。因此，泛型的使用会有一些限制

★ 泛型的限制

1 在类型判断时不能使用泛型类，只能使用原始类

```

public class Main {
    public static void main(String[] args) {
        Test test = new Test();
        System.out.println(test instanceof Test<String>);
    }
}

```

instanceof 的泛型类型非法

java.lang

2 泛型类不能创建参数化类型数组,但是可以使用原始类

```

public class Main {
    public static void main(String[] args) {
        Score<Integer>[] score = new Score<String>[10];
    }
}

```

创建泛型数组

java.lang

public final class String
implements java.io.Serializable, Comparable<String>, CharSequence

```

1 public static void main(String[] args) {
2     Test[] test = new Test[10];
3 }

```

3 可以创建泛型类型数组

```

1 public static void main(String[] args) {
2     Test<String>[] test = new Test[10];
3     test[1] = new Test<String>();
4 }

```

六、函数式接口

函数式接口就是 JDK1.8 专门为我们提供好的用于 Lambda 表达式的接口，这些接口都可以直接使用 Lambda 表达式，非常方便，这里我们主要介绍一下四个主要的函数式接口，

★ 函数式接口，它是一个接口，只能有一个未实现的方法，当然也可以有其它已经实现好的方法，一个匿名内部类要使用 Lambda 表达式也是需要有这个条件

★ 首先查看 Supplier 供给函数式接口，它只有一个 get 方法用于获取到需要的对象，可以理解为一个加工工厂，每调用一次就创建或者输出一次；

```

1 @FunctionalInterface //表示这是一个函数式接口
2 public interface Supplier<T> { //它还是一个泛型接口
3
4     /**
5      * Gets a result.
6      *
7      * @return a result
8      */
9     T get(); //只有一个为要实现的方法
10 }

```

1 我们在使用它的时候就需要使用到匿名内部类来重写 get() 方法

首先需要创建一个类

```

1 public class Student { //名为Student
2     public void test(){
3         System.out.println("你是猪");
4     }
5 }
6

```

在 Main 类中创建 Supplier，在重写 get 方法的时候这里由于接受的类型需要和泛型类定义的一样，所以就是 Student，这里每调用一个 get 就会创建一个 Student

```

1      public static final Supplier<Student> SUPPLIER_STUDENT = new
Supplier<Student>() {
2          @Override
3          public Student get() {
4              return new Student(); //new一个
5          }
6      };
7      public static void main(String[] args) {
8          Student stu1 = new Student();
9          Student student = SUPPLIER_STUDENT.get();
10         student.test();
11     }

```

可以通过方法引用来简化

```

1      public static final Supplier<Student> SUPPLIER_STUDENT = Student::new; //方法
应用
2      public static void main(String[] args) {
3          Student stu1 = new Student();
4          Student student = SUPPLIER_STUDENT.get();
5          student.test();
6      }

```

★ **Consumer** 消费型函数接口，用于消费对象，上面是供给，消费这里就是需要你提供一个对象，然后你可以对这个对象进行操作，这就叫消费

1 首先先看一下这个函数式接口内部，它由一个未实现的方法，还有一个实现的方法组成

```

1  @FunctionalInterface
2  public interface Consumer<T> {
3
4      void accept(T t); //未实现的方法
5
6      default Consumer<T> andThen(Consumer<? super T> after) { //实现的方法
7          Objects.requireNonNull(after);
8          return (T t) -> { accept(t); after.accept(t); };
9      }
10 }
11

```

2 首先需要先创建 **Consumer** 接口对象，这里我重写了 **accept** 方法，使用者需要传入一个 **Student** 类型变量,就会打印一段话

```

1      public static final Consumer<Student> STUDENT_CONSUMER = student ->
      System.out.println("我是学生我叫: "+student.name); //定义静态方法需要传入一个学生类
2      public static void main(String[] args) {
3          Student stu1 = new Student();
4          stu1.setName("小猪");
5          STUDENT_CONSUMER.accept(stu1); //消费学生
6      }
7
8      //输出: 我是学生我叫: 小猪

```

3 `andThen` 方法表示为消费之后的操作，解读一下它的代码

`andThen` 会先接收一个新创建的 `Consumer` 对象，新的对象也是需要重写 `accept` 方法的，然后先判断新对象是不是为 `null`，然后 `return` 部分会先执行当前实例的 `accept`，然后在执行新创建的对对象的 `accept` (我是这么理解的)

```

1      default Consumer<T> andThen(Consumer<? super T> after) { //它需要传入一个after的
      泛型对象
2          Objects.requireNonNull(after); //这里判断这个after是不是为空，after表示前一步操
      作;
3          return (T t) -> { accept(t); after.accept(t); }; //返回先执行accept，然后将
      执行后的对象传递给泛型对象
4      }

```

4 `andThen`

```

1      public static final Consumer<Student> STUDENT_CONSUMER = student ->
      System.out.println("我是学生我叫: "+student.name); //定义静态方法需要传入一个学生类
2      public static void main(String[] args) {
3          Student stu1 = new Student();
4          stu1.setName("小猪");
5          STUDENT_CONSUMER
6              .andThen(student -> System.out.println("我不是猪")) //这里的
      student表示创建的接受变量
7              .accept(stu1);
8      }
9
10     //输出:
11     我是学生我叫: 小猪
12     我不是猪

```

★ `Function` 函数式接口，这个接口表示消费一个对象，然后又会对外供给一个对象

1 本体代码

它也是一个泛型类，并且还需要明确两个泛型类型，一个是接受参数，一个是返回的参数

```

1
2      @FunctionalInterface
3      public interface Function<T, R> {

```

```

4
5
6     R apply(T t);
7
8
9     default <V> Function<V, R> compose(Function<? super V, ? extends T> before)
10    {
11        Objects.requireNonNull(before);
12        return (V v) -> apply(before.apply(v));
13    }
14
15    default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {
16        Objects.requireNonNull(after);
17        return (T t) -> after.apply(apply(t));
18    }
19
20
21    static <T> Function<T, T> identity() {
22        return t -> t;
23    }
24 }
25

```

2 `apply` 方法是需要实现的，它就是供给一个对象

这里创建了一个 `Function` 对象，明确了两个泛型类，`Student` 为传入，`String` 为输出，使用匿名内部类重写 `apply` 方法，使用 `student` 的 `toString` 方法

```

1     public static final Function<Student,String> STUDENT_FUNCTION = new
Function<Student, String>() {
2         @Override
3         public String apply(Student student) {
4             return student.toString();
5         }
6     };
7     public static void main(String[] args) {
8         Student stu1 = new Student();
9         stu1.setName("小猪");
10        System.out.println(STUDENT_FUNCTION.apply(stu1));
11    }

```

简化


```

1      public static final Function<Student,String> STUDENT_FUNCTION =
Object::toString;
2      public static void main(String[] args) {
3          Student stu1 = new Student();
4          stu1.setName("小猪");
5          System.out.println(STUDENT_FUNCTION.apply(stu1)); //输出toString
6      }
7
8      //输出:
9      Student{name='小猪'} //需要在类中定义toString方法

```

2 `compose` , 将之前函数式的结果作为当前函数式的实参:

```

1      default <V> Function<V, R> compose(Function<? super V, ? extends T> before) {
2          Objects.requireNonNull(before);
3          return (V v) -> apply(before.apply(v));
4      }

```

★ Predicate断言型函数式接口: 接受一个参数, 然后进自定义判断并返回一个布尔类型的结果

```

1      @FunctionalInterface
2      public interface Predicate<T> {
3          boolean test(T t);    //这个方法就是我们要实现的
4
5          default Predicate<T> and(Predicate<? super T> other) {
6              Objects.requireNonNull(other);
7              return (t) -> test(t) && other.test(t);
8          }
9
10         default Predicate<T> negate() {
11             return (t) -> !test(t);
12         }
13
14         default Predicate<T> or(Predicate<? super T> other) {
15             Objects.requireNonNull(other);
16             return (t) -> test(t) || other.test(t);
17         }
18
19         static <T> Predicate<T> isEqual(Object targetRef) {
20             return (null == targetRef)
21                 ? Objects::isNull
22                 : object -> targetRef.equals(object);
23         }
24     }

```

1 代码实现

```

1 public class Student {
2     public int score;
3 }

```

```

1 private static final Predicate<Student> STUDENT_PREDICATE = student ->
  student.score >= 60;
2 public static void main(String[] args) {
3     Student student = new Student();
4     student.score = 80;
5     if(STUDENT_PREDICATE.test(student)) { //test方法的返回值是一个boolean结果
6         System.out.println("及格了，真不错，今晚奖励自己一次");
7     } else {
8         System.out.println("不是，Java都考不及格？隔壁初中生都在打ACM了");
9     }
10 }

```

五、判断空包装类

使用 `Optional` 可以用来判断空包装类，这个类可以有效的处理空指针的问题

```

1 private static void test(String str){
2     Optional
3         .ofNullable(str) //将传入的对象包装进Optional中
4         .ifPresent(s -> System.out.println("字符串长度为: "+s.length()));
5         //如果不为空，则执行这里的Consumer实现
6 }

```