接口

由于《Java编程思想》第四版在2007年完成，基于的是当时最新的Java，而工业界流行的Java8在14年才出来，并且Java8对于接口更新了很多特性，所以《Java编程思想》中的部分内容就是过时的了。我们后面更多得会提到的是《On Java 8》这本书，以及使用Java8标准（不知道考试的时候会不会因为这个被扣分。。。）

接口的定义和设计背景

我觉得接口是Java区分于C++的一个非常重要的特征，接口让Java中程序的耦合性不再那么的紧密，让Java更容易完成各种“设计模式”（至少我是这么认为的）。我们用一个例子来说明，顺便介绍接口的声明方法之类的基础的问题：（这里的代码来自《On Java 8》）

“完全解耦”

假设我们有这么一个需求，或者说主代码

```
public class Applicator {
    public static void apply(Processor p, Object s) { // 顺便，这里也体现了Java让所有类都是Object的子类的好处
        System.out.println("Using Processor " + p.name());
        System.out.println(p.process(s));
    }

    public static void main(String[] args) {
        String s = "We are such stuff as dreams are made on";
        apply(new Upcase(), s);
        apply(new Downcase(), s);
        apply(new Splitter(), s);
    }
}
```

`Applicator`的`apply`方法接受一个`Processor`、可以看作一个车间，和一个`Object`、可以看作需要被处理的东西

我们很容易给出一些实现，比如讲`s`的字符全部转化成大写、小写，如下：

```
class Processor {
    public String name() {
        return getClass().getSimpleName(); // 这里会返回RTT
    }

    public Object process(Object input) {
        return input; // 什么都不处理
    }
}
class Upcase extends Processor {
    // 返回协变类型
```

```
    @Override
    public String process(Object input) {
        return ((String) input).toUpperCase();
    }
}
class Downcase extends Processor {
    @Override
    public String process(Object input) {
        return ((String) input).toLowerCase();
    }
}
```

但是如果我们现在有一个`Filter`类，其也有一个方法`process`，我们很想让其也可以用在`apply`中，因为它也有方法`process`，但是很明显由于`apply`里面接受的是`Processor`，我们并不能让两个不在一条继承链上的类互相转化，所以即使`Filter`类也有`process`方法、即使我们把`apply`的第一个参数改成`Filter`就可以跑起来，但是实际上还是做不到的。我们称这种情况为`Applicator`的`apply`方法和`Processor`过于耦合，这阻止了`Applicator`的`apply()`方法被复用

我们的需求就是如果两个类都有`process`方法，能不能让他们都可以作为某个参数输入到另一个方法中呢？答案是：使用接口

我们设计一个接口叫`Processable`（大多数接口都使用形容词来命名），它应该让所有拥有这个接口的类都拥有`process`方法。具体如下：

```
package TestInterfaceProcess;

// interfaces/Applicator.java
import java.util.*;

interface Processable {
    default String name() { // Java8 default
        return getClass().getSimpleName(); // 这里会返回RTT的名字
    }

    default Object process(Object input) {
        return input; // 什么都不处理
    }

    String s="This final and static"; // right, can create static instance
    public final int cnt=0; // 默认是final的，所以这里的cnt是没有办法改变值的
    int num=0;
    static void getcnt(){ // right, can create public final static method
//        ++num; // wrong, num is final
    }

    public static void main(String[] args) {
```

```
//        new Processable(); // wrong, can't new a interface unless use lambda or
nested class
    }
}
class Upcase implements Processable {
    @Override
    public String process(Object input) { // must be 'public'
        return ((String) input).toUpperCase();
    }
}
class Downcase implements Processable {
    @Override
    public String process(Object input) {
        return ((String) input).toLowerCase();
    }
}

public class Applicator {
    public static void apply(Processable p, Object s) {
        System.out.println("Using Processor " + p.name());
        System.out.println(p.process(s));
    }

    public static void main(String[] args) {
        String s = "We are such stuff as dreams are made on";
        apply(new Upcase(), s);
        apply(new Downcase(), s);
    }
}
```

这里逐一介绍下：

接口的声明和内容

一个接口表示：所有实现了该接口的类看起来都像上面的样子

声明`interface`只需要把声明类的时候的`class`改成`interface`就行，但是要注意`interface`一定是`abstract`的，即使没有写明，所以我们并不能直接`new`一个接口类。另外接口的访问控制和类的是一样的就不介绍了

接口中的所有方法都会被默认是`public`的，而且一个类实现接口的方法的时候一定要把覆写部分的访问控制设定成`public`，这是Java规定的（相对的，在继承中一个继承类的方法和基类的被覆写的方法的访问控制权限完全可以不一样）

接口中可以有方法的实现（Java8特性），但是需要增加域作用符`default`修饰

接口中可以有静态方法（Java8特性），这样可以方便存放一些应该属于接口的方法

以上两个Java8特性考试的时候当不存在（MDZZ）

接口中也可以有变量，但是需要注意变量默认是`public static final`的（这是接口和抽象类的最大不同），即使你不这么声明（声明成别的样的会被报错。。。）。另外和类的`static final`量一样，Java会尽量优化这个过程，但是`static final`的量一旦被赋值就不能再改变（或指向别处）

实现接口的方法

如果你的类实现了某个接口，需要在类名和继承内容后面加上`implements [interface,...]`。值得注意的是，一个类可以实现多个接口，我们稍后会介绍解决命名冲突等实现多个接口的时候需要注意的问题

在类内应该覆写所有没有`default`的方法，如果`default`在此类里的实现应该不一样，也需要覆写。覆写最好加上关键字`@Override`

覆写的方法访问权限必须是`public`

一次实现多个接口

Java支持一个类实现多个接口——注意这并不会导致数据的重复问题，因为接口里并没有非`static`的属性（或者说变量）

想要实现多个接口，只需要在`implements`后面多写接口就行

下面是一个例子，这里`TesInterface`同时有两个功能：实现比较接口实现Java自带排序的比较功能；实现`Readable`接口使之可以构造`Scanner`

```
package TestInterfaceProcess;

import org.jetbrains.annotations.NotNull;

import java.io.IOException;
import java.nio.CharBuffer;
import java.util.*;

public class TesInterface implements Comparator<Integer> ,Readable {
    @Override
    public int compare(Integer o1, Integer o2) {
        System.err.println("o1: " + o1 + " ");
        System.err.println("o2: " + o2 + " ");
        return o1.compareTo(o2);
    }

    private static final char [] CHS= "0123456789".toCharArray();
    private int count;
    private final Random r=new Random();
    public TesInterface(int count){
//        System.err.println("count: " + count + " ");
        this.count=count;
    }
```

```
    @Override
    public int read(@NotNull CharBuffer cb) throws IOException {
        if (count == 0) {
            return -1; // indicates end of input
        }
        --count;
        int len=r.nextInt(Integer.MAX_VALUE)%8+1; // r.nextInt()貌似会出现负数。。。
        System.err.println("len: " + len + " ");
        for(int i=0;i<len;++i){
            cb.append(CHS[r.nextInt(CHS.length)]);
//            System.out.println(cb);
        }
        cb.append(' ');
        return len+1;
    }

    public static void main(String[] args) {
        int len=10;
        Scanner cin=new Scanner(new TesInterface(len));
        Integer [] nums=new Integer[len];
        int tot=0;
        while(cin.hasNext()){
            nums[tot++]= cin.nextInt(); // 没有任何问题。。。包装类会在后面介绍
            System.err.println("tot: " + tot + " ");
        }
        Arrays.sort(nums,0,tot,new TesInterface(0));
        for(int i=0;i<tot;++i){
            System.out.println(nums[i]);
        }
    }
}
```

当然这里的例子很生硬，但是确实反映了这种特性的特征和写法

接口的继承

接口是可以继承自接口的，产生的新接口会拥有基接口的所有方法和自己的所有方法。在单继承的时候，一个基接口的方法无论有没有`default`关键字，都可以在继承接口里使用`default`来覆写。当然你也可以选择不覆写，这样实现继承接口的类的时候，就需要覆写所有基接口没有实现并且在继承接口也没有实现的、和继承接口新增的没有实现的方法

但是Java中接口是支持多继承的：C++中类默认支持多继承，但是这样会导致非常多的问题，比如如果`A`是`B`和`C`的基类，而`D`继承自`B`和`C`，那`D`的实例中应该有几个`A`的数据呢？这种简单的多继承C++还能通过虚继承控制，但是对于更复杂的继承情况，多继承就显得很笨重了。在Java中类只有单继承，但是接口可以多继承，因为接口没有非`static`量，不存在数据重复的问题

接口的多继承实际上也没什么特别的。。。因为接口不存在成员变量，所以少了很多类里继承的麻烦事；同时接口的方法在多继承的过程中的表现实际上和单继承差别不大，值得注意的只有：如果有接口`A`和`B`，其有相同形式的方法`fun`，`A`实现了这个方法而`B`没有，接口`C`继承自`A`和`B`，那么由于`fun`在`A`和`B`中一个有一个没有默认实现，所以需要专门覆写；还是`A`、`B`、`C`，如果`A`和`B`都没有实现`fun`，那么`C`就不会有问题；如果`A`和`B`都实现了`fun`，`C`需要专门覆写`fun`

另外在一个继承类同时实现接口并继承的时候，如果基类中和接口中有一个相同形式的方法的话，也需要专门去实现

和类的表现一样，接口中的静态不变内容是不会被覆写的，也不能说被隐藏了。。。因为即使是在继承接口的类中也只能使用`Base.num`的方式访问/调用静态内容

lambda表达式

Java的lambda表达式和接口有不可分割的联系，所以放在一起讲

声明方式

在C++里面，lambda表达式是非常灵活多样的，但是在Java中，为了和接口适应，lambda表达式在一定程度上被阉割了

不多说，我们看看怎么声明lambda表达式：

引入

在比较数组的时候，Java会让你传入一个`Comparator<? super T> c`参数，它会被用来进行比较，所以一种写法是这样的：

```
String[] strs={"123","23","243","244"};
Arrays.sort(strs, new Comparator<String>() {
    @Override
    public int compare(String o1, String o2) {
        return o1.length()-o2.length();
    }
});
```

这是匿名类的写法，但是这看起来就比较麻烦，我们能不能像C++一样传一个函数进去呢?

行，但不完全行

行是因为，lambda就是被用来做这事的；不行是因为，lambda的本质还是一个类的对象，不过写起来像函数。我们先关注形式上java怎么传递函数到方法里的（因为我们知道这个函数本质是类，所以我说的是“形式上”）

声明

```
Arrays.sort(strs, (String s1, String s2) -> {
    return s1.length() - s2.length();
});
```

这里的`(String s1, String s2) -> {return s1.length() - s2.length();}`就是lambda表达式

这么看起来，Java的lambda表达式就是新建一个变量传给需要接收接口的地方？差不多接近真相了，不过我们看下下面的代码：

```
interface BiFun<T> {
    void fun();

    boolean cmp(T t1, T t2);
}

public class Main {
    static <T> void sort(T[] ts, BiFun<? super T> bf) {
        // do things
    }
    public static void main(String[] args) throws IOException {
        String[] strs = {"123", "23", "243", "244"};
        sort(strs, (String s1, String s2) ->{return s1.length() - s2.length();}); // #
WRONG!
        sort(strs, new BiFun<String>() { // RIGHT!
            @Override
            public void fun() {
                System.out.println("haha");
            }
            @Override
            public boolean cmp(String t1, String t2) {
                return t1.length()-t2.length()>0;
            }
        });
    }
}
```

这里的`#`行就不能通过编译，为什么呢？观察发现，`BiFun`有两个没有实现的方法，这就是原因！lambda表达式只能用在 被传的接口类型里面只有一个未实现的方法 内！这就是形式上使用lambda表达式的方法

理解

一种不对的理解

下面的语句是跑得通的：

```
String []strs={"iplee","Iplee","ipLee","HYY","hyy"}
Comparator<String>cmp=(String o1,String o2)->{
    return o1.length()-o2.length();
};
Arrays.sort(strs,cmp); // cmp是一个实现了Comparator<String>接口的类的实例
```

所以，仅通过上面的代码lambda表达式可以看作：一种构造新变量的形式，这种形式只关注被构造变量需要实现的接口的唯一的方法

那为什么不对呢？我们看下面的语句：

```
Comparator<String> lambdacmp = (String o1, String o2) -> {
    return o1.length() - o2.length();
};
Comparator<String>classcmp=new Comparator<String>() {
    @Override
    public int compare(String o1, String o2) {
        return o1.length()-o2.length();
    }
};
System.out.println(lambdacmp);
System.out.println(classcmp);
```

这里输出的分别是：

```
TestLambda.Main$$Lambda$14/0x0000000800066840@1cd072a9
TestLambda.Main$1@7c75222b
```

也就是，lambda表达式的实例真的会被当作“Lambda”，和接口匿名类实例不同

正确的理解

我也不知道。。。百度的资料也都没啥意义

方法引用

在形式上说，Java其实是可以传一个方法的，如下：

```
public class Main {
    static int cmp(String o1,String o2){
        return o1.length()-o2.length();
    }
    public static void main(String[] args) throws IOException {
        String[] strs = {"123", "23", "243", "244"};
        Arrays.sort(strs,Main::cmp);
        Arrays.sort(strs,String::compareToIgnoreCase);
    }
}
```

这个和lambda的本质其实是一样的，形式上看是传了一个函数（实际传的是一个新的类）

再比如，利用`removeIf`语句删除`List`里面的`NULL`内容可以使用下面的语句

```
ls.removeIf(ele->{return ele==null;}); // lambda
ls.removeIf(Objects::isNull); // 方法引用
```

也可以传构造器的引用，不过这里就不介绍了

闭包

引入

看下面的语句：

```
public static void main(String[] args) {
    String textFinal1 = "Comparing!";
    String textFinal2;
    textFinal2 = "Comparing!";
    String textNotFinal = "Comparing!";
    textNotFinal = "Comparing!";
    String[] strs = {"iplee", "Iplee", "ipLee", "HYY", "hyy"};
//        Arrays.sort(strs,(String o1,String o2)->{int first=o1.length();return first-
o2.length();});
    Arrays.sort(strs, (String s1, String s2) -> {
        System.out.println(textFinal1); // *
        return s1.length() - s2.length();
    });
    Arrays.sort(strs, (String s1, String s2) -> {
        System.out.println(textFinal2); // *
        return s1.length() - s2.length();
    });
```

```
    Arrays.sort(strs, (String s1, String s2) -> {
        System.out.println(textNotFinal); // #
        return s1.length() - s2.length();
    });
}
```

这里，#那一行会报错，*那两行则不会。这是为什么呢?

作用域

在之前，我们先明确lambda表达式的作用域

```
String first="First";
String[] strs = {"iplee", "Iplee", "ipLee", "HYY", "hyy"};
Arrays.sort(strs,(String first,String second)->{return first.length()-
second.length();});
```

这个代码过不了编译，因为lambda的参数`first`和外部的`first`冲突了

下面这个也同样过不了编译

```
int first=0;
String[] strs = {"iplee", "Iplee", "ipLee", "HYY", "hyy"};
Arrays.sort(strs,(String o1,String o2)->{int first=o1.length();return first-
o2.length();});
```

换句话说，lambda表达式的体与嵌套块有相同的作用域。为什么要这么限制呢?

闭包

传入限制

有时候我们希望将外部的变量传入lambda表达式，比如引入部分写的输出正在比较的语句，但是我们发现几乎是一样的语句，为什么有的可以被引用，有的不可以呢?

在作用域上，我们提到，lambda表达式的体与嵌套块有相同的作用域，但是lambda并不能使用所有的嵌套块的变量，实际上只有“事实最终变量”，可以被使用

那么，下一个问题，什么是“事实最终变量”? 根据StackOverflow的这个帖子，一个变量如果其值永远不会被改变，那么他就是事实最终变量。这里的值不改变指引用不改变，比如下面的语句，从内容的层面，`lsFinal`是被改变了的（`add`了新东西），但是Java编译器还是将它当作了事实最终变量

```
String[] strs = {"iplee", "Iplee", "ipLee", "HYY", "hyy"};
List<String> lsFinal = new ArrayList<>();
lsFinal.add("Hello");lsFinal.add("World");
Arrays.sort(strs, (String s1, String s2) -> {
    System.out.println(lsFinal); // using effectively final
    return s1.length() - s2.length();
});
```

使用限制

传入的事实最终变量，会被当作真的`final`处理，也就是不能重新赋值（本质是改变引用），比如：

```
String[] strs = {"iplee", "Iplee", "ipLee", "HYY", "hyy"};
List<String> lsFinal = new ArrayList<>();
lsFinal.add("Hello");lsFinal.add("World");
Arrays.sort(strs, (String s1, String s2) -> {
    System.out.println(lsFinal);
    lsFinal=new ArrayList<>();
    return s1.length() - s2.length();
});
```

就是不合法的，因为lambda里面改变了`lsFinal`引用指向的对象

特例

不过有一个特例，这个特例在一开始还混淆了我的认知，那就是`static`变量

非`final`的`static`变量，不仅可以被拉入lambda，还可以在lambda内被改变：

```
static int static_Cnt = -1;

public static void main(String[] args) throws IOException {
    static_Cnt = 0;
    int non_static_Cnt = 0;
    String text = "comparing";
    String[] strs = {"iplee", "Iplee", "ipLee", "HYY", "hyy"};
    Arrays.sort(strs, (String o1, String o2) -> {
        ++static_Cnt; // ALL RIGHT!
        System.out.println(text);
        return o1.length() - o2.length();
    });
    System.out.println(static_Cnt);
    System.out.println(Arrays.toString(strs));
    Arrays.sort(strs, (String o1, String o2) -> {
        ++non_static_Cnt; // WRONG!
```

```
        return o1.length() - o2.length();
    });
}
```

按照前面的定义，这里的`static_Cnt`肯定不是一个“事实最终变量”，因为在一开始被赋值为`-1`之后还被赋值成了`0`，但是它确实被传入了lambda，而且在lambda里面被改变了值

另外，`static`变量在lambda里面会有一些问题：

```
package TestJavaFX;

import java.util.LinkedList;
import java.util.Queue;

public class TestLambda {
    static int[] h = new int[10];
    public static void main(String[] args) {
        for (int i = 0; i < h.length; ++i) {
            h[i] = i;
        }
        Queue<Runnable> qr = new LinkedList<>();
        qr.add(new Runnable() {
            @Override
            public void run() {
                System.out.println(1 + " : " + h[1]);
            }
        });
        qr.add(new Runnable() {
            @Override
            public void run() {
                System.out.println(2 + " : " + h[2]);

            }
        });
        for (int i = 0; i < h.length; ++i) {
            h[i] = -i;
        }
        while(qr.size()>0){
            qr.poll().run();
        }
    }
}
/* 输出:
1 : -1
2 : -2
*/
```

由于`static`变量并没有被捕捉到闭包里，所以实际上`Runnable`里面的内容会输出实时`h[1]`的值而不是在`add`进队列时`h[1]`的值

常用函数式接口

直接截图《Java核心技术 卷I》

函数式接口	参数类型	返回类型	抽象方法名	描述	其他方法
Runnable	无	void	run	作为无参数或返回值的动作运行	
Supplier<T>	无	T	get	提供一个T类型的值	
Consumer<T>	T	void	accept	处理一个T类型的值	andThen
BiConsumer<T, U>	T, U	void	accept	处理T和U类型的值	andThen
Function<T, R>	T	R	apply	有一个T类型参数的函数	compose, andThen, identity
BiFunction<T, U, R>	T, U	R	apply	有T和U类型参数的函数	andThen
UnaryOperator<T>	T	T	apply	类型T上的一元操作符	compose, andThen, identity
BinaryOperator<T>	T, T	T	apply	类型T上的二元操作符	andThen, maxBy, minBy
Predicate<T>	T	boolean	test	布尔值函数	and, or, negate, isEqual
BiPredicate<T, U>	T, U	boolean	test	有两个参数的布尔值函数	and, or, negate

看到需要传入的接口时直接查就行

（实际上这里与设计模式也有一定关联）