生命力顽强的JAVA

1995年Sun公司推出的Java语言，从第一个版本诞生到现在已经有二十多年的了。时间若白驹过隙，转瞬即逝。二十多年来IT技术更新换代，编程语言层出不穷。就像自然界遵循优胜劣汰的法则，编程语言也是一样，很多老牌的编程语言被新兴的编程语言替代，逐渐没落甚至退出历史舞台，然而Java作为一门有着二十多年历史的编程语言却越发显得生机勃勃！宝刀未老！究其原因，其中很重要的一点就是Java语言不断进行版本迭代推出一系列符合技术发展趋势的新特性！

“从Java的演变路径来看，它一直致力于让并发编程更容易、出错更少。Java 1.0里有线程和锁，甚至有一个内存模型——这是当时的最佳做法。Java 5更是添加了工业级的构建模块，如线程池和并发集合。Java 7添加了分支/合并（fork/join）框架，使得并行变得更实用。而Java 8对并行又有了一个更简单的新思路！”

--摘自《Java 8 In Action》

Java8新特性简介

Java 8可谓是自Java 5以来最具革命性的版本了，她在语言、编译器、类库、开发工具以及Java虚拟机等方面都带来了不少新特性:

（其中最为核心的为Lambda 表达式与Stream API）

★ Lambda表达式

Lambda表达式可以说是Java 8最大的卖点，她将函数式编程引入了Java。Lambda允许把函数作为一个方法的参数，或者把代码看成数据。

★ Stream API

Stream API是把真正的函数式编程风格引入到Java中。其实简单来说可以把Stream理解为MapReduce。从语法上看，也很像linux的管道、或者链式编程，代码写起来简洁明了，非常酷帅！

• 接口的默认方法与静态方法

我们可以在接口中定义默认方法，使用default关键字，并提供默认的实现。所有实现这个接口的类都会接受默认方法的实现，除非子类提供的自己的实现。

• 方法引用

通常与Lambda表达式联合使用，可以直接引用已有Java类或对象的方法。

• 重复注解

Java 8引入重复注解，相同的注解在同一地方也可以声明多次。

- **扩展注解的支持**

Java 8扩展了注解的上下文，几乎可以为任何东西添加注解，包括局部变量、泛型类、父类与接口的实现，连方法的异常也能添加注解。

- **Optional**

Java 8引入Optional类来防止空指针异常，使用Optional类我们就不用显式进行空指针检查了。

- **Date/Time API (JSR 310)**

Java 8新的Date-Time API (JSR 310)受Joda-Time的影响，提供了新的java.time包，可以用来替代 java.util.Date和java.util.Calendar。

- **JavaScript引擎Nashorn**

Nashorn允许在JVM上开发运行JavaScript应用，允许Java与JavaScript相互调用。

- **Base64**

在Java 8中，Base64编码成为了Java类库的标准。Base64类同时还提供了对URL、MIME友好的编码器与解码器。

- **更好的类型推测机制**

Java 8在类型推测方面有了很大的提高，这就使代码更整洁，不需要太多的强制类型转换了。

- **编译器优化**

Java 8将方法的参数名加入了字节码中，这样在运行时通过反射就能获取到参数名，只需要在编译时使用-parameters参数。

- **并行（parallel）数组**

支持对数组进行并行处理，主要是parallelSort()方法，它可以在多核机器上极大提高数组排序的速度。

- **并发（Concurrency）**

在新增Stream机制与Lambda的基础之上，加入了一些新方法来支持聚集操作。

为什么要学习Java8新特性

通过文档我们可以看到Java8添加了很多API用于对函数式编程的支持，可以看出Java8对函数式编程的重视程度。
Java8之所以费这么大功夫引入函数式编程，原因有二：
1.代码简洁，函数式编程写出的代码简洁且意图明确，比如使用stream接口让你告别for循环。
2.多核友好，Java函数式编程使得编写并行程序如此简单，只需要调用一下parallel()方法即可。

“对于习惯了面向对象编程的开发者来说，抽象的概念并不陌生。面向对象编程是对数据进行抽象，而函数式编程是对行为进行抽象。现实世界中，数据和行为并存，程序也是如此，因此这两种编程方式我们都得学。
函数式编程这种新的抽象方式还有其他好处，例如：
不是所有人都在编写性能优先的代码，对于这些人来说，函数式编程带来的好处尤为明显。程序员能编写出更容易

阅读的代码——这种代码更多地表达了业务逻辑的意图，而不是它的实现机制。易读的代码也易于维护、更可靠、更不容易出错。比如，在写回调函数和事件处理程序时，程序员不必再纠缠于匿名内部类的冗繁和可读性，函数式编程让事件处理系统变得更加简单。能将函数方便地传递也让编写惰性代码变得容易，惰性代码在真正需要时才初始化变量的值。
总而言之，Java 已经不是祖辈们当年使用的Java 了，嗯， 这不是件坏事。”

--摘自《Java 8 Lambdas Functional Programming For The Masses》

面向对象的束缚

在正式讲Lambda表达式和函数式编程之前我们先看一下我们之前分析一下面向对象编程的不足！

需求:启动一个线程并输出一句话

本着“万物皆对象”的思想，创建一个Runnable接口的匿名内部类对象(线程对象)来指定任务内容，再启动该线程。
这种做法是Ok的,但是.....
我们真的想创建一个匿名内部类对象吗?
不。我们只是为了做这件事情而不得不创建一个对象。
我们真正希望做的事情是：将 run方法方法体内的代码传递给Thread类。
传递并执行这一段代码，才是我们真正的目的。
而创建对象只是受限于面向对象语法而不得不采取的一种手段方式。
那，有没有更加简单的办法?
如果我们将关注点从“怎么做”回归到“做什么”的本质上，就会发现只要能够更好地达到目的，过程与形式其实并不重要。

```
new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("开一个子线程处理一个耗时操作");
            }
  }).start();
```

函数式编程的解放

函数式编程思想让我们更加关注我们的目标也就是:“做什么”

```
    @Test
    public void testLambda() throws Exception {
       new Thread(()->System.out.println("开一个子线程处理一个耗时操作")).start();;
    }
```

这段代码和刚才的执行效果是完全一样的。
从代码的语义中可以看出：我们启动了一个线程，而线程任务的内容以一种更加简洁的形式被指定。
瞬间感觉得到了解放！不再有“不得不创建 Runnable接口对象”的束缚，不再有“覆盖重写抽象方法”的负担，就是这么简单！

总结：

Java语言一大特点与优势就是面向对象，但是随着技术的发展，我们会发现面向对象思想在某些场景下并非是最优的思想。能够简单快速满足需求、达到目标效果的思想有时候可能更加适合。这种思想就是函数式编程思想！
Java8中的Lambda表达式就可以让我们把这种思想发挥的淋漓尽致！

通过案例体验Java8新特性

根据用户提出的不同条件筛选出满足相应条件的商品

比如：
// 筛选出所有名称包含手机的商品
// 筛选出所有价格大于1000的商品

准备数据

```
private static List<Product> products = new ArrayList<>();

static {
    products.add(new Product(1L,"苹果手机",8888.88));
    products.add(new Product(2L,"华为手机",6666.66));
    products.add(new Product(3L,"联想笔记本",7777.77));
    products.add(new Product(4L,"机械键盘",999.99));
    products.add(new Product(5L,"雷蛇鼠标",222.22));
}
```

传统方式：

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

// 筛选出所有名称包含手机的商品
List<Product> list = findPhoneByName(products);
for (Product product : list) {
     System.out.println("product = " + product);
}
// 筛选出所有价格大于1000的商品
List<Product> productList=  findPhoneByPrice(products);
for (Product product : productList) {
  System.out.println("product = " + product);
}

}
private static List<Product> findPhoneByPrice(List<Product> products) {
```

```
 List<Product> list = new ArrayList<>();
      for (Product product : products) {
          if (product.getPrice()>1000) {
                list.add(product);
            }
        }
        return list;
    }
 private static List<Product> findPhoneByName(List<Product> products) {
       List<Product> list = new ArrayList<>();
        for (Product product : products) {
            if (product.getName().contains("手机")) {
                list.add(product);
            }
        }
        return list;
    }
```

匿名内部类改造：

```
public static void main(String[] args) {
     // 筛选出所有名称包含手机的商品

    List<Product> productList = findPhoneByCondition(products, new MyPredicate() {
        @Override
        public boolean test(Product product) {
            return product.getName().contains("手机");
        }
    });
    for (Product product : productList) {
        System.out.println(product);
    }

    // 筛选出所有价格大于1000的商品

    List<Product> productList1 = findPhoneByCondition(products, new MyPredicate() {
        @Override
        public boolean test(Product product) {
            return product.getPrice()>1000;
        }
    });
    for (Product product : productList1) {
        System.out.println("product = " + product);
    }
}
private static List<Product> findPhoneByCondition(List<Product> products, MyPredicate
predicate){
```

```
    List<Product> productList = new ArrayList<>();

    for (Product product : products) {
        if (predicate.test(product)) {
            productList.add(product);
        }
    }
    return productList;
}


public interface MyPredicate {

      boolean test(Product product);
}
```

策略设计模式：

```
  public static void main(String[] args) {
         // 筛选出所有名称包含手机的商品

        List<Product> productList = findPhoneByCondition(products, new
FilterProductByNamePredicate());
        for (Product product : productList) {
            System.out.println(product);
        }

        // 筛选出所有价格大于1000的商品

        List<Product> productList1 = findPhoneByCondition(products,new
FilterProductByPricePredicate());
        for (Product product : productList1) {
            System.out.println("product = " + product);
        }
  }

 private static List<Product> findPhoneByCondition(List<Product> products, MyPredicate
predicate){

        List<Product> productList = new ArrayList<>();

        for (Product product : products) {
            if (predicate.test(product)) {
                productList.add(product);
            }
        }
        return productList;
```

```
    }
```

```
public class FilterProductByNamePredicate implements MyPredicate {

    @Override
    public boolean test(Product product) {
        return product.getName().contains("手机");
    }
}

public class FilterProductByPricePredicate implements MyPredicate {

    @Override
    public boolean test(Product product) {
        return product.getPrice()>1000;
    }
}
```

```
public interface MyPredicate {

      boolean test(Product product);
}
```

Lambda表达式改造：

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

        // 筛选出所有名称包含手机的商品

        List<Product> productList = findPhoneByCondition(products, (p)-
>p.getName().contains("手机"));
        for (Product product : productList) {
            System.out.println(product);
        }

        // 筛选出所有价格大于1000的商品

        List<Product> productList1 = findPhoneByCondition(products,(p)->p.getPrice()>1000);
        for (Product product : productList1) {
            System.out.println("product = " + product);
        }
```

```
    }

    private static List<Product> findPhoneByCondition(List<Product> products, MyPredicate
predicate){

        List<Product> productList = new ArrayList<>();

        for (Product product : products) {
            if (predicate.test(product)) {
                productList.add(product);
            }
        }
        return productList;
    }
```

```
public interface MyPredicate {

    boolean test(Product product);

}
```

Stream改造：

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

        // 筛选出所有名称包含手机的商品

        products.stream().filter(p->p.getName().contains("手机")).forEach(System.out::println);

        System.out.println("----------------------");

        // 筛选出所有价格大于1000的商品

        products.stream().filter(p->p.getPrice()>1000).forEach(System.out::println);

    }
```

Lambda语法

lambda语法图解

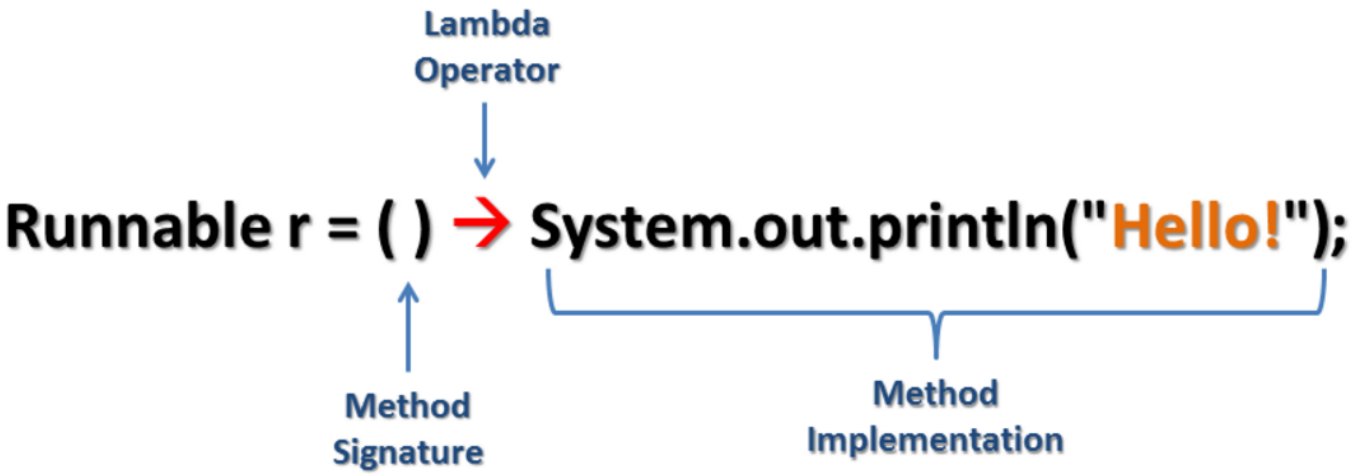
```java
@Test
public void testOOP() throws Exception {
    new Thread(
            new Runnable() {
                @Override
                public void run() {
                    System.out.println("OOP");
                }
            }
    ).start();
}

@Test
public void testLambda() throws Exception {
    new Thread(
            () -> System.out.println("Lambda")
    ).start();
}
```

在上图中，Thread类的构造器中需要传入一个Runnable接口的实现类,而Runnable接口中只有一个run方法(实现类中需要实现该方法并将任务代码写该方法中)

那既然我们知道Thread需要一个Runnable,并且只需要重写一个run方法,并且run方法中就是我们要写的任务代码,那我们为何不直接把我们要完成的任务代码传递给Thread呢?

所以我们就可以规定一种语法,将我们要完成的任务直接传递给Thread:
()->System.out.println("使用Lambda方式开启线程")



这种新的语法就是:“->”

```
“->”被称为Lambda 操作符或箭头操作符。----固定语法
它将Lambda 分为两个部分：
左侧：指定了Lambda 表达式需要的所有参数 ---参数
run方法不需要参数,所以可以直接写()
右侧：指定了Lambda 体，即Lambda 表达式要执行的任务、功能或者说是行为。---方法体
我们要完成的功能就是在run方法中输出一句话,所以直接写
System.out.println("使用Lambda方式开启线程")
```

可传递的匿名函数

可以把Lambda表达式理解为简洁地表示可传递的匿名函数的一种方式：它没有名称，但它有参数列表、函数主体、返回类型。这个定义还是太模糊的，让我们从以下几个关键词慢慢道来。
匿名——我们说匿名，是因为它不像普通的方法那样有一个明确的名称：写得少而做得多！
函数——我们说它是函数，是因为Lambda函数不像方法那样属于某个特定的类。但和方法一样，Lambda有参数列表、函数主体、返回类型。
传递——Lambda表达式可以作为参数传递给方法或存储在变量中。
简洁——无需像匿名类那样写很多模板代码。

也可以简单理解为:Lambda表达式是匿名内部类对象的方法实现

那到这里我们已经清楚了Lambda表达式的基本语法格式,那再继续思考一个问题:是不是所有的接口类型作为参数传递(传统写法就是匿名内部类)都可以使用Lambda表达式呢?

答案是不可以！因为只有接口中只有一个抽象方法的时候,Lambda表达式才可以正确的“猜测”到你传递的任务实际上是在实现接口中的唯一的那一个抽象方法！
那么我们把这种只有一个抽象方法的接口称作为“函数式接口”，只有函数式接口才可以使用Lambda表达式进行函数式编程！

什么是函数式接口?

定义

只要确保接口中有且仅有一个抽象方法即可

格式

```
修饰符 interface 接口名称 {
    [public abstract] 返回值类型 方法名称(可选参数信息);
    // 其他
}
```

为了便于区分并便于编译器进行语法校验,JDK8中还引入了一个新的注解:

```
@FunctionalInterface
```

该注解可用于一个接口的定义上,一旦使用该注解来定义接口，编译器将会强制检查该接口是否确实有且仅有一个抽象方法，否则将会报错。
需要注意的是，即使不使用该注解，只要满足函数式接口的定义，该接口仍然是一个函数式接口，使用起来都一样。(可以类比下@Override 注解的作用)
我们可以去看下Runnable接口的源码,发现它就是一个函数式接口

那么我们也可以来定义一个简单的函数式接口:

```
@FunctionalInterface
public interface MyFunctionalInterface {
    void myMethod();
}
```

特征

1.可选类型声明：不需要声明参数类型，编译器可以统一识别参数值。
2.可选的参数圆括号()：一个参数无需定义圆括号，但多个参数需要定义圆括号()。
3.可选的大括号{}：如果主体包含了一个语句，就不需要使用{}。
4.可选的返回关键字return：如果主体只有一个表达式返回值则可以省略return和{}

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

        List<String> list = Arrays.asList("a","c","b");
        Map<String,String> map = new HashMap<>(){
            {
                this.put("手机","小米");
                this.put("笔记本","苹果笔记本");
                this.put("鼠标","罗技鼠标");
            }
        };

        //1.可选类型声明：不需要声明参数类型，编译器可以统一识别参数值

        list.forEach((String x)->System.out.println(x));
        list.forEach((x)->System.out.println(x));
        System.out.println("------------------------");

        //2.可选的参数圆括号()：一个参数无需定义圆括号，但多个参数需要定义圆括号()
        list.forEach((x)->System.out.println(x));
        list.forEach(x->System.out.println(x));
        map.forEach((x,y)->System.out.println(x+":" + y));
        System.out.println("------------------------");

        //3.可选的大括号{}：如果主体包含了一个语句，就不需要使用{}
        list.forEach(x->System.out.println(x));
```

```
        list.forEach(x->{
            System.out.println(x);
            System.out.println(x);

        });
        System.out.println("-------------------------");

        //4.可选的返回关键字return：如果主体只有一个表达式返回值则可以省略return和{}

        list.stream().sorted((s1,s2)->s2.compareTo(s1)).forEach(x->System.out.println(x));

        System.out.println("-------------------------");

        list.stream().sorted(Comparator.reverseOrder()).forEach(x->System.out.println(x));

        list.stream().sorted(Comparator.comparing(String::hashCode).reversed())
                .forEach(x->System.out.println(x));
    }
```

总结
Lambda表达式极大的丰富了Java语言的表现力,简化了Java代码的编写,但是语法细节我们或许很难记住,那么只需要记住一条:可推断则可省略！能省则省

注意
在某些情况下，类型信息可以帮助阅读者理解代码，这时需要手动声明类型，让代码便于阅读。有时省略类型信息更加简洁明了，还可以减少干扰，让阅读者关注业务逻辑，这时就可以省略；所以在开发中建议大家根据自己或项目组的习惯结合实际情况，进行类型声明或省略。

常见的函数式接口

JDK提供了大量常用的函数式接口以丰富Lambda的典型使用场景，它们主要在java.util.function包中被提供。
上面用到过的Consumer接口就是其中之一，接下来给大家介绍下其他的常用函数式接口

函数式接口	参数类型	返回类型	说明
Consumer<T> 消费型接口	T	void	对类型为T的对象操作， 方法：void accept(T t)
Supplier<T> 供给型接口	无	T	返回类型为T的对象， 方法：T get();可用作工厂
Function<T, R> 函数型接口	T	R	对类型为T的对象操作，并返回结果是R类型的对象。 方法：R apply(T t);
Predicate<T> 断言型接口	T	boolean	判断类型为T的对象是否满足条件，并返回boolean 值。 方法boolean test(T t);

说明

//函数式接口类型　　参数类型　　返回类型　　说明

Consumer消费型接口　　T　　void　　对类型为T的对象操作，方法：void accept(T t)

Supplier供给型接口　　无　　T　　返回类型为T的对象，方法：T get();可用作工厂

Function<T, R>函数型接口　　T　　R　　对类型为T的对象操作，并返回结果是R类型的对象。方法：R apply(T t);

Predicate断言型接口　　T　　boolean　　判断类型为T的对象是否满足条件，并返回boolean 值。方法boolean test(T t);

练习

```
// 需求1:编写shop方法输出消费多少元
// 需求2:编写getCode方法返回指定位数的随机验证码字符串
// 需求3:编写getStringRealLength方法返回字符串真实长度
// 需求4:编写getString方法返回长度大于5的字符串的集合
```

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

        //调用shop 方法
        // 普通方式:
        obj.shop(1000, new Consumer<Integer>() {
            @Override
            public void accept(Integer money) {
                System.out.println("这次大保健总共消费了" + money + "元");
```

```
            }
        });
        //lambda方式:
        obj.shop(1000,money->System.out.println("这次大保健总共消费了" + money + "元"));

        // 调用getCode方法
        // 普通方式
        String code = obj.getCode(4, new Supplier<Integer>() {
            @Override
            public Integer get() {
                return new Random().nextInt(10);
            }
        });
        System.out.println("code = " + code);

        // lambda方式
        String code1 = obj.getCode(4,()->new Random().nextInt(10));
        System.out.println("code1 = " + code1);

        //调用 getStringRealLength  方法
        // 普通方式
        int realLength = obj.getStringRealLength("  wolfcode.cn ", new Function<String,
Integer>() {
            @Override
            public Integer apply(String s) {
                return s.trim().length();
            }
        });
        System.out.println("realLength = " + realLength);
        // lambda方式
        int realLength1 = obj.getStringRealLength("  用心做教育 ",s->s.trim().length());
        System.out.println("realLength1 = " + realLength1);

        // 调用 getString 方法
        List<String> list = Arrays.asList("abc","wolf","wolfcode","wolfcode.cn");
        // 普通方式
        obj.getString(list, new Predicate<String>() {
            @Override
            public boolean test(String s) {
                return s.length()>5;
            }
        }).forEach(x->System.out.println(x));

        // lambda方式
        obj.getString(list,s->s.length()>5).forEach(x->System.out.println(x));


    }
```

```
    // 需求1:编写shop方法输出消费多少元
    public void shop(int money , Consumer<Integer> consumer){
        consumer.accept(money);
    }
    // 需求2:编写getCode方法返回指定位数的随机验证码字符串
    public String getCode(int number, Supplier<Integer> supplier){

        StringBuffer buffer = new StringBuffer();
        for (int i = 0; i < number; i++) {
            buffer.append(supplier.get());
        }
        return buffer.toString();

    }

    // 需求3:编写getStringRealLength方法返回字符串真实长度
    public int getStringRealLength(String str, Function<String,Integer> function){

        return function.apply(str);
    }
    // 需求4:编写getString方法返回长度大于5的字符串的集合
    public List<String> getString(List<String> list, Predicate<String> predicate){
         List<String> newList = new ArrayList<>();
        for (String s : list) {
            if (predicate.test(s)) {
                newList.add(s);
            }
        }
        return newList;
    }
```

方法引用

在使用Lambda表达式的时候，我们实际上传递进去的代码就是一种解决方案：拿什么参数做什么操作。那么考虑一种情况：如果我们在Lambda中所指定的操作方案，已经有地方存在相同方案，那是否还有必要再写重复逻辑呢?

之前方法引用存在的问题

1.不够精简的Lambda

看一下我们之前写的代码

Arrays.asList("a", "b", "c").forEach((x) -> System.out.print(x));

2.问题分析

这段代码的问题在于，对字符串进行控制台打印输出的操作方案，明明已经有了现成的实现，那就是System.out对象中的println(String) 方法。
既然Lambda希望做的事情就是调用println(String) 方法，那何必自己手动调用呢?
能否省去Lambda的语法格式呢（尽管它已经相当简洁）？

3.问题解决

只要“引用”过去就好了!
请注意其中的双冒号:: 写法，这被称为“方法引用”，而双冒号是一种新的语法。
Arrays.asList("a", "b", "c").forEach(System.out::println);

4.语法分析

双冒号:: 为引用运算符，而它所在的表达式被称为方法引用。如果Lambda要表达的函数方案已经存在于某个方法的实现中，那么则可以通过双冒号来引用该方法作为Lambda的替代者。
下面两种写法的执行效果完全一样，而方法二方法引用的写法复用了已有方案，更加简洁。
方法一:Lambda表达式：s -> System.out.println(s);
拿到参数之后经Lambda之手，继而传递给System.out.println 方法去处理。

方法二:方法引用： System.out::println
直接让System.out 中的println 方法来取代Lambda。

方法引用的语法

1.对象的引用 :: 实例方法名

2.类名 :: 静态方法名

3.类名 :: 实例方法名

4.类名 :: new (构造器引用)

5.类型[] :: new (数组引用)

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

        //1.对象的引用 :: 实例方法名
        Arrays.asList("a","b","c").forEach(new Consumer<String>() {
            @Override
            public void accept(String s) {
                System.out.println(s);
            }
        });

        Arrays.asList("a","b","c").forEach(s->System.out.println(s)); // 前
        Arrays.asList("a","b","c").forEach(System.out::println); // 后
```

```
        System.out.println("----------------------");

        //2.类名 :: 静态方法名

        Supplier<Double> sup = new Supplier<Double>() {
            @Override
            public Double get() {
                return Math.random();
            }
        };



        Supplier<Double> supp = ()->Math.random();// 前

        Supplier<Double> suppp = Math::random;// 后

        System.out.println(suppp.get());
        System.out.println("----------------------");

        //3.类名 :: 实例方法名
        Function<Product,String> fun = new Function<Product, String>() {
            @Override
            public String apply(Product product) {
                return product.getName();
            }
        };

        Function<Product,String> funn = (p)->p.getName(); // 前

        Function<Product,String> funnn = Product::getName; // 后

        String productName = funnn.apply(new Product(2L,"小米手机",3200d));
        System.out.println("productName = " + productName);
        System.out.println("----------------------");

        //4.类名 :: new (构造器引用)
        Supplier<Product> snew = new Supplier<Product>() {
            @Override
            public Product get() {
                return new Product();
            }
        };

        Supplier<Product> snew1 = ()->new Product(); // 前

        Supplier<Product> snew2 = Product::new; // 后
        System.out.println(snew2.get());
        System.out.println("----------------------");
```

```
        //5.类型[] :: new (数组引用)

        Function<Integer,String[]> f2 = String[]::new;
        String[] strings = f2.apply(20);
        System.out.println(strings.length);
        System.out.println("----------------------");


    }
```

Lambda表达式的延迟执行(了解)

到这里我们已经学会了使用Lambda表达式以及它的孪生兄弟方法引用来简化我们代码的编写，但是Lambda表达式的强大可不仅仅只是简化语法哦！

1.记录日志时的性能浪费

一种典型的场景就是对参数进行有条件使用，例如对日志消息进行拼接后，在满足条件的情况下进行打印输出：

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

        String name = "张无忌";
        String action="夜店消费";
        String money = "20000元";

        // 立即拼接字符串
        logMethod("info",name + action + money);

    }

    private static void logMethod(String info, String s) {

        if ("info".equals(info)) {
            System.out.println(s);
        }
    }
```

这段代码存在问题：无论级别是否满足要求，作为log 方法的第二个参数，三个字符串一定会首先被拼接并传入方法内，然后才会进行级别判断。如果级别不符合要求，那么字符串的拼接操作就白做了，存在性能浪费。

备注：SLF4J(应用非常广泛的日志框架)在记录日志时为了解决这种性能浪费的问题，并不推荐首先进行字符串的拼接，而是将字符串的若干部分作为可变参数传入方法中，仅在日志级别满足要求的情况下才会进行字符串拼接。例如： LOGGER.debug("变量{}的取值为{}。", "os", "windows") ，其中的大括号{} 为占位符。如果满足日志级别要求，则会将“os”和“windows”两个字符串依次拼接到大括号的位置；否则不会进行字符串拼接。这也是一种可行解决方案，但Lambda可以做到更好。

2.使用Lambda拒绝性能浪费

定义一个函数式接口并对log方法进行改造
这样一来，只有当级别满足要求的时候，才会进行三个字符串的拼接；否则三个字符串将不会进行拼接。

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

        String name = "张无忌";
        String action="夜店消费";
        String money = "20000元";

        // 延迟拼写字符串
        logLambdaMethod("info",()->{
            System.out.println("come in");
            return name + action+ money;
        });
    }

    private static void logLambdaMethod(String level, Supplier<String> sup) {
        if ("info".equals(level)){
            System.out.println("come in");
            System.out.println(sup.get());
        }
    }
```

更强大的接口（了解）

在JDK 1.7及更老的版本中，接口中只能包含常量与抽象方法两种内容，不允许包含其他。但是这种情况在JDK 1.8中已经改变：接口中允许包含default方法和static方法并指定方法体的具体实现。

1.复习面向接口编程

接口是一种契约，一种规范。在多数情况下，我们可以优先使用接口来体现多态性。下面的代码很好地诠释了这一点：

首先是所有手机都拥有的打电话接口：

```
public interface Callable {
    void call();
}
```

然后是苹果的iPhone手机实现了该接口，可以打电话：
而三星的Galaxy手机也实现了该接口，也可以打电话：

有了接口，我们在打电话的时候就可以屏蔽掉不同手机之间的差别，只专注于打电话功能即可：

```
interface  Callable{
        void call();
    }

class IPhone implements Callable{

        @Override
        public void call() {

            System.out.println("使用苹果手机打电话");
        }
    }
class HuaWei implements Callable{

        @Override
        public void call() {

            System.out.println("使用华为手机打电话");
        }
    }
```

2.场景：接口升级

想像一下如果Callable接口除了普通的call方法之外，还希望引入一个autoCall（自动拨号）方法，那怎么办？如果继续添加一个抽象方法，那么已经使用Callable接口的两个实现类都必须进行代码修改以进行覆盖重写。然而问题是，接口的制定者往往在交付接口进行使用之后，无法控制其使用者再“追加协议内容”。这就破坏了接口的向后兼容性。
这时候常见的做法是定义一个新的接口来继承已有接口并添加新的内容，借助对象转型来保证向后兼容性。但是，有没有更好的做法?

3.解决：默认方法

从JDK 1.8开始，可以在接口名称不变的情况下，追加新的方法定义，而对已有的若干实现类不产生任何影响。这种新添加的方法需要使用default关键字进行修饰并指定方法体实现，被称为“默认方法”。
例如上例中的Callable接口，如果添加一个autoCall方法：

```
 interface  Callable{
        void call();
        default void autoCall(){
            System.out.println("自动拨号");
        }
    }
```

而IPhone和Galaxy两个实现类不需要任何修改，即可继续使用,这是保证接口向后兼容性的更优手段。

4.问题：接口冲突

考虑一种场景，如果有两个接口中都指定了default方法且签名一样，那么当一个实现类同时实现了两个接口时会怎么样?
例如
接口A:

接口B:

实现类（编译时报错）：
//Duplicate default methods named method with the parameters () and ()
//are inherited from the types InterfaceB and InterfaceA
public class InterfaceImpl implements InterfaceA, InterfaceB {
}

这段代码是错误的，编译错误将会发生在实现类中。

```
 interface  A{
        default void print(){
            System.out.println("A");
        }

    }
 interface  B{
        default void print(){
            System.out.println("B");
        }
    }

  class C implements A,B{

    }
```

5.解决：方法覆盖

当同时实现的多个接口中存在签名一样的若干个方法体实现（无论内容是否完全相同）时，实现类必须进行覆盖重写以解决冲突。

```
interface  A{
        default void print(){
            System.out.println("A");
        }

    }
interface  B{
```

```
        default void print(){
            System.out.println("B");
        }
    }

class C implements A,B{


        @Override
        public void print() {

           System.out.println("这个print方法必须被覆写");

        }
    }
```

总结:

为什么接口中可以有具体方法?

接口中的default方法是一个无奈之举，在Java 7及之前要想在定义好的接口中加入新的抽象方法是很困难甚至不可能的，因为所有实现了该接口的类都要重新实现。default方法就是用来解决这个尴尬问题的，直接在接口中实现新加入的方法。既然已经引入了default方法，为何不再加入static方法来避免专门的工具类呢!

6.静态方法

Collection和Collections、File和Files.....这些带有s后缀的类从名称上约定为对应的接口或类的工具类。然而这是一种妥协方案，并不是最优方案。
考虑以下几个问题：
1)对于多个对象的公共特征，如何复用？提高其抽象层次。
2)为什么优先实现接口而不是继承父类？因为Java有单继承的限制。
3)对于不依托于对象的内容，应该如何设计？使用静态。

结合这三个问题，JDK 1.8中允许在接口中定义静态方法并指定方法体实现。例如：

```
interface  Callable{
        void call();

        default void autoCall(){
            System.out.println("自动拨号");
        }

        static void videoCall(){
            System.out.println("视频通话");
        }

    }
```

这样我们就不再需要Collections和Files这样单独的工具类了。
当然，为了保证向后兼容性，它们也被保留。
并且在Java9已经在java.util.List接口中 加入了大量的static静态方法of的重载以设计出更优的API：直接返回本接口对象。
当然在Java8中很多API接口中也加入了静态方法比如:Comparator接口

https://docs.oracle.com/javase/9/docs/api/java/util/List.html