2、具体内容

从JDK1.8开始为了简化使用者进行代码的开发，专门提供有Lambda表达式的支持，利用此操作形式可以实现函数式的编程，对于函数式编程比较著名的语言：Haskell、Scala，利用函数式的编程可以避免面向对象编程之中的一些繁琐的处理问题。

面向对象在其长期发展的过程之中一直有一部分的反对者，这些反对者认为面向对象的设计过于复杂了，并且过于繁琐了。以一个最简单的程序为例。

范例：观察传统开发中的问题

```
interface IMessage {
	public void send(String str) ;
}
public class JavaDemo {
	public static void main(String args[]) {
		IMessage msg = new IMessage() {
			public void send(String str) {
				System.out.println("消息发送：" + str) ;
			}
		} ;
		msg.send("www.mldn.cn") ;
	}
}
```

在这样的一个程序里面，实际上核心的功能只有一行语句“System.out.println(”消息发送：“ + str) ;”，但是为了这一行的核心语句依然需要按照完整的面向对象给出的设计结构及进行开发。于是这些问题随着技术的不断发展也是越来越突出了。

范例：使用lambda表达式实现与之前完全一样的功能

```
interface IMessage {
	public void send(String str) ;
}
public class JavaDemo {
	public static void main(String args[]) {
		IMessage msg = (str)->{
```

```
            System.out.println("发送消息：" + str) ;
        } ;
        msg.send("www.mldn.cn") ;
    }
}
```

现在整个程序代码里面会发现真的只是编写了一行语句，于是利用这种形式就避免了复杂的面向对象结构化的要求。

lambda表达式如果要想使用，那么必须有一个重要的实现要求：SAM(Single Abstract Method)，只有一个抽象方法，以之前的IMessage接口为例，在这个接口里面发现只是提供有一个send()方法，除此之外没有如何其它方法定义，所以这样的接口就被称为函数式接口，而只有函数式接口才可以被Lambda表达式所使用。

范例：使用函数式接口注解

```
@FunctionalInterface    // 函数式接口
interface IMessage {
    public void send(String str) ;
}
public class JavaDemo {
    public static void main(String args[]) {
        IMessage msg = (str)->{
            System.out.println("发送消息：" + str) ;
        } ;
        msg.send("www.mldn.cn") ;
    }
}
```

对于Lambda表达式而言，提供有如下几种格式：

- 方法没有参数：()->{};
- 方法有参数：(参数,参数)->{};
- 如果现在只有一行语句返回：(参数,参数)->语句;

范例：定义没有参数的方法

```
@FunctionalInterface    // 函数式接口
interface IMessage {
    public void send() ;
}
public class JavaDemo {
    public static void main(String args[]) {
        IMessage msg = ()->{
            System.out.println("发送消息：www.mldn.cn") ;
        } ;
        msg.send() ;
    }
}
```

范例：定义有参数的处理形式

```
@FunctionalInterface    // 函数式接口
interface IMath {
    public int add(int x,int y) ;
```

```
}
public class JavaDemo {
    public static void main(String args[]) {
        IMath math = (t1,t2)->{
            return t1 + t2 ;
        } ;
        System.out.println(math.add(10,20)) ;
    }
}
```

以上的表达式之中你会发现只有一行语句，这个时候也可以进一步简化。

范例：简化Lambda操作

```
@FunctionalInterface    // 函数式接口
interface IMath {
    public int add(int x,int y) ;
}
public class JavaDemo {
    public static void main(String args[]) {
        IMath math = (t1,t2)-> t1 + t2 ;
        System.out.println(math.add(10,20)) ;
    }
}
```

利用lambda表达式的确可以摆脱传统面向对象之中关于结构的限制，使得代码更加的简便。

2、具体内容

引用数据类型最大的特点是可以进行内存的指向处理，但是在传统的开发之中一直所使用的只是对象引用操作，而从JDK1.8之后也提供有方法的引用，即：不同的方法名称可以描述同一个方法。如果要进行方法的引用在Java里面提供有如下的四种形式：

- 引用静态方法：类名称 :: static 方法名称；
- 引用某个实例对象的方法：实例化对象 :: 普通方法；
- 引用特定类型的方法：特定类 :: 普通方法；

- 引用构造方法：类名称 :: new；

范例：引用静态方法

- 在String类里面提供有String.valueOf()方法，这个方法就属于静态方法。
  |- 方法定义：public static String valueOf(int i)；该方法有参数，并且有返回值；

```
@FunctionalInterface    // 函数式接口
// P描述的是参数、R描述的是返回值
interface IFunction<P,R> {
    public R change(P p) ;
}
public class JavaDemo {
    public static void main(String args[]) {
        IFunction<Integer,String> fun = String :: valueOf ;
        String str = fun.change(100) ;
        System.out.println(str.length()) ;
    }
}
```

利用方法引用这一概念可以为一个方法定义多个名字，但是要求必须是函数式接口。

范例：引用实例化对象中的方法

- 在String类里面有一个转大写的方法：public String toUpperCase()；
  |- 这个方法是必须在有实例化对象提供的情况下才可以调用；

```
@FunctionalInterface    // 函数式接口
// P描述的是参数、R描述的是返回值
interface IFunction<R> {
    public R upper() ;
}
public class JavaDemo {
    public static void main(String args[]) {
        IFunction<String> fun = "www.mldn.cn" :: toUpperCase ;
        System.out.println(fun.upper()) ;
    }
}
```

在进行方法引用的时候也可以引用特定类中的一些操作方法，在String类里面提供有一个字符串大小关系的比较：

- 比较大小：public int compareTo(String anotherString)；

这是一个普通方法，如果要引用普通方法，则往往都需要实例化对象，但是如果说现在你不想给出实例化对象，只是想引用这个方法，则就可以使用特定类来进行引用处理。

范例：引用指定类中的方法

```
@FunctionalInterface    // 函数式接口
// P描述的是参数、R描述的是返回值
interface IFunction<P> {
    public int compare(P p1,P p2) ;
}
public class JavaDemo {
```

```
        public static void main(String args[]) {
                IFunction<String> fun = String :: compareTo ;
                System.out.println(fun.compare("A","a")) ;
        }
}
```

在方法引用里面最具有杀伤力的就是构造方法的引用。

范例：引用构造方法

```
class Person {
        private String name ;
        private int age ;
        public Person(String name,int age) {
                this.name = name ;
                this.age = age ;
        }
        public String toString() {
                return "姓名：" + this.name + "、年龄：" + this.age ;
        }
}
@FunctionalInterface     // 函数式接口
interface IFunction<R> {
        public R create(String s,int a) ;
}
public class JavaDemo {
        public static void main(String args[]) {
                IFunction<Person> fun = Person :: new ;
                System.out.println(fun.create("张三",20)) ;
        }
}
```

提供方法引用的概念更多的情况下也只是弥补了对于引用的支持功能。

2、具体内容

在JDK1.8之中提供有lambda表达式也提供有方法引用，但是你会发现现在如果有开发者自己定义函数式接口，往往都需要使用“@FunctionalInterface”注解来进行大量声明，于是很多的情况下如果为了方便则可以直接引用系统中提供的函数式接口。

在系统之中专门提供有一个java.util.function的开发包，里面可以直接使用函数式接口，在这个包下面一共有如下的几个核心接口供使用：

1、功能性函数式接口：

• 在String类中有一个方法判断是否以指定的字符串开头：public boolean startsWith(String str);

接口定义：	**接口使用：**

@FunctionalInterface
public interface
Function<T,R>{
    public R apply(T t)
}
	import java.util.function.* ;
	public class JavaDemo {
	    public static void main(String args[]) {
	        Function<String,Boolean> fun = "**Hello" ::
	startsWith ;
	        System.out.println(fun.apply("**")) ;
	    }
	}

2、消费型函数接口：只能够进行数据的处理操作，而没有任何的返回；

• 在进行系统数据输出的时候使用的是：System.out.println();

接口定义：	**接口使用：**

@FunctionalInterface
public interface
Consumer<T>{
    public void accept(T t)
}
	import java.util.function.* ;
	public class JavaDemo {
	    public static void main(String args[]) {
	        Consumer<String> con = System.out :: println ;
	        con.accept("www.mldn.cn") ;
	    }
	}

3、供给型函数式结构：

• 在String类中提供有转小写方法，这个方法没有接受参数，但是有返回值；

|- 方法：public StringtoLowerCase();

接口定义：	**接口使用：**

@FunctionalInterface
public interface
Supplier<T>{
    public T get();
}
	import java.util.function.* ;
	public class JavaDemo {
	    public static void main(String args[]) {
	        Supplier<String> sup = "www.MLDNJAVA.cn" ::
	toLowerCase ;
	        System.out.println(sup.get()) ;
	    }
	}

4、断言型函数式接口：进行判断处理

- 在String类有一个equalsToIgnoreCase()方法;

接口定义：	**接口使用：**

@FunctionalInterface
public interface
Predicate<T>{
public boolean test(T t)
}
	import java.util.function.* ;
	public class JavaDemo {
	public static void main(String args[]) {
	Predicate<String> pre = "mldn" :: equalsIgnoreCase
	;
	System.out.println(pre.test("MLDN")) ;
	}
	}

以后对于实际项目开发之中，如果JDK本身提供的函数式接口可以被我们所使用，那么就没有必要进行重新定义了。