第二十二天

1、 JDK5 新特性:

1）自动拆装箱
2）泛型
3）可变参数
4）静态导入
5）增强 for 循环 for(:){}
6）锁，java.util.concurrent.locks.ReentrantLock
7）枚举
8）Annotation（注解/标注），如@Override
9）格式输出，如 printf("price:%.2f",price)
...

2、 JDK7 新特性

1）二进制字面量，如 0b0001
2）数字字面量可以出现下划线，如 int a = 100_000;
3）switch 表达式可以用字符串类型
4）泛型实例化类型自动推断，如 List<Integer> list = new ArrayList<>();
5）异常的多个 catch 合并,每个异常用或|，如 catch (IOException | SQLException e) {...}
6）try-with-resources 语句，try(父接口是 AutoCloseable){}catch(){}
...

3、 JDK8 新特性

1）接口新增：默认方法与静态方法
2）Lambda 表达式，为函数式编程提供基础
4）方法引用
3）Stream 函数式操作流元素集合
…

JDK8 新特性

1、 接口

Java 8 中的接口现在支持在声明方法的同时提供实现，通过两种方式可以完成这种操作。
1）Java 8 允许在接口内声明静态方法。
2）Java 8 引入了一个新功能，叫默认方法，通过默认方法你可以指定接口方法的默认实现。实现接口的子类如果不显式地提供该方法的具体实现，就会自动继承默认的实现。
这种机制可以使你平滑地进行接口的优化和演进。
InterfaceTest.java

静态方法及接口
同时定义接口以及工具辅助类（companion class）是 Java 语言常用的一种模式，工具类定义了与接口实例协作的很多静态方法。

比如，Collections 就是处理 Collection 对象的辅助类。
由于静态方法可以存在于接口内部，你代码中的这些辅助类就没有了存在的必要，你可以把这些静态方法转移到接口内部。
为了保持后向的兼容性，这些类依然会存在于 Java 应用程序的接口之中。

默认方法
接口包含的方法签名在它的实现类中也可以不提供实现
缺失的方法实现会作为接口的一部分由实现类继承（所以命名为默认实现），而无需由实现类提供。
默认方法由 default 修饰符修饰，并像类中声明的其他方法一样包含方法体。

2、 Lambda 表达式

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，可能还有一个可以抛出的异常列表。
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。
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，还可能有可以抛出的异常列表。
传递：Lambda 表达式可以作为参数传递给方法或存储在变量中。（行为参数）
简洁：无需像匿名类那样写很多模板代码。

Lambda 的基本语法-->LambdaTest1.java
(parameters) -> expression
或（请注意语句的花括号）
(parameters) -> { statements; }

根据上述语法规则，以下哪个不是有效的 Lambda 表达式？
(1) () -> {}
(2) () -> "Raoul"
(3) () -> {return "Mario";}
(4) (Integer i) -> return "Alan" + i;
(5) (String s) -> {"IronMan";}
答案：只有 4 和 5 是无效的 Lambda。
(4) return 是一个控制流语句。要使此 Lambda 有效，需要使花括号，如下所示：
(Integer i) -> {return "Alan" + i;}。
(5)“Iron Man”是一个表达式，不是一个语句。要使此 Lambda 有效，你可以去除花括号和分号，如下所示：(String s) -> "Iron Man"。或者如果你喜欢，可以使用显式返回语句，如下所示：(String s)->{return "IronMan";}。

在哪里使用 Lambda：
你可以在函数式接口上使用 Lambda 表达式。

函数式接口：
函数式接口就是只定义一个抽象方法的接口。
哪怕有很多默认方法，只要接口只定义了一个抽象方法，它就仍然是一个函数式接口。

下面哪些接口是函数式接口？

```
public interface Adder{
    int add(int a, int b);
}
public interface SmartAdder extends Adder{
    int add(double a, double b);
}
public interface Nothing{
}
```

答案：只有 Adder 是函数式接口。
SmartAdder 不是函数式接口，因为它定义了两个叫作 add 的抽象方法（其中一个是从 Adder 那里继承来的）。
Nothing 也不是函数式接口，因为它没有声明抽象方法。

用函数式接口可以干什么呢？-->LambdaTest2.java
Lambda 表达式允许你直接以内联的形式为函数式接口的抽象方法提供实现，并把整个表达式作为函数式接口的实例（具体说来，是函数式接口一个具体实现的实例）。
你用匿名内部类也可以完成同样的事情，只不过比较笨拙。
Lambda表达式的签名要和函数式接口中抽象方法的签名(声明)保持一致

3、 函数描述符

函数式接口中的抽象方法的签名基本上就是 Lambda 表达式的签名，我们将这种抽象方法叫作函数描述符。
例如，Runnable 接口可以看作一个什么也不接受什么也不返回（void）的函数的签名() -> void 代表了参数列表为空，且返回 void 的函数。
Lambda 表达式可以被赋给一个变量，或传递给一个接受函数式接口作为参数的方法，当然这个 Lambda 表达式的签名要和函数式接口的抽象方法一样。

@FunctionalInterface-->LambdaTest3.java
如果你去看看新的 Java API，会发现函数式接口带有 @FunctionalInterface 的标注。这个标注用于表示该接口会设计成一个函数式接口。如果你用 @FunctionalInterface 定义了一个接口，而它却不是函数式接口的话，编译器将返回一个提示原因的错误。例如，错误消息可能是“Multiple non-overriding abstract methods found in interface Foo ”，表明存在多个抽象方法。@FunctionalInterface 不是必需的，但对于为此设计的接口而言，使用它是比较好的做法。它就像是 @Override 标注表示方法被重写了。

使用函数式接口
函数式接口很有用，因为抽象方法的签名可以描述 Lambda 表达式的签名。
函数式接口的抽象方法的签名称为函数描述符。
所以为了应用不同的 Lambda 表达式，你需要一套能够描述常见函数描述符的函数式接口。
Java API 中已经有了几个函数式接口，比如 Comparable、Runnable 和 Callable。
Java 8 的库设计师帮你在 java.util.function 包中引入了几个新的函数式接口，例如：Predicate、Consumer、Function 等。

Predicate：-->LambdaTest4.java

java.util.function.Predicate<T>接口定义了一个名叫 test 的抽象方法，它接受泛型 T 对象，并返回一个 boolean。

```
public interface Predicate<T>{
    boolean test(T t);
}
```

Predicate 接口是用来支持 java 函数式编程新增的一个接口,使用这个接口和 lambda 表达式就可以以更少的代码为 API 方法添加更多的动态行为。

Consumer：-->LambdaTest5.java

java.util.function.Consumer<T>定义了一个名叫 accept 的抽象方法，它接受泛型 T 的对象，没有返回（void）。

```
public interface Consumer<T>{
    void accept(T t);
}
```

Function：-->LambdaTest6.java

java.util.function.Function<T, R>接口定义了一个叫作 apply 的方法，它接受一个泛型 T 的对象，并返回一个泛型 R 的对象。

```
public interface Function<T, R>{
    R apply(T t);
}
```

Supplier：-->LambdaTest7.java

Supplier 接口返回一个任意范型的值，和 Function 接口不同的是该接口没有任何参数

```
public interface Supplier<T> {
    T get();
}
```

Java 8中的常用函数式接口

函数式接口	函数描述符	原始类型特化
Predicate<T>	T->boolean	IntPredicate,LongPredicate, DoublePredicate
Consumer<T>	T->void	IntConsumer,LongConsumer, DoubleConsumer
Function<T,R>	T->R	IntFunction<R>, IntToDoubleFunction, IntToLongFunction, LongFunction<R>, LongToDoubleFunction, LongToIntFunction, DoubleFunction<R>, ToIntFunction<T>, ToDoubleFunction<T>,
Supplier<T>	()->T	BooleanSupplier,IntSupplier, LongSupplier, DoubleSupplier
UnaryOperator<T>	T->T	IntUnaryOperator, LongUnaryOperator, DoubleUnaryOperator
BinaryOperator<T>	(T,T)->T	IntBinaryOperator, LongBinaryOperator, DoubleBinaryOperator
BiPredicate<L,R>	(L,R)->boolean	
BiConsumer<T,U>	(T,U)->void	ObjIntConsumer<T>, ObjLongConsumer<T>, ObjDoubleConsumer<T>
BiFunction<T,U,R>	(T,U)->R	ToIntBiFunction<T,U>, ToLongBiFunction<T,U>, ToDoubleBiFunction<T,U>

4、 方法引用

方法引用让你可以重复使用现有的方法定义，并像 Lambda 一样传递它们。
方法引用可以被看作仅仅调用特定方法的 Lambda 的一种快捷写法。
当你需要使用方法引用时，目标引用放在分隔符::前，方法的名称放在后面。

方法引用主要有三类。
(1) 指向静态方法的方法引用（例如 Integer 的 parseInt 方法，写作 Integer::parseInt）。
(2) 指向任意类型实例方法的方法引用（ 例如 String 的 length 方法，写作 String::length）。
(3) 指向现有对象的实例方法的方法引用（假设你有一个局部变量 expensiveTransaction 用于存放 Transaction 类型的对象，它支持实例方法 getValue，那么你就可以写 expensive-Transaction::getValue）。

```
class Transaction{
    public int getValue(){…}
}
Transaction expensiveTransaction
```

第二种和第三种方法引用可能乍看起来有点儿晕。类似于 String::length 的第二种方法引用的思想就是你在引用一个对象的方法，而这个对象本身是 Lambda 的一个参数。例如，Lambda 表达式(String s) -> s.toUppeCase()可以写作 String::toUpperCase。但第三种方法引用指的是，你在 Lambda 中调用一个已经存在的外部对象中的方法。例如，Lambda 表达式()->expensiveTransaction.getValue()可以写作 expensiveTransaction::getValue。

对于一个现有构造函数，你可以利用它的名称和关键字 new 来创建它的一个引用：ClassName::new。它的功能与指向静态方法的引用类似。

5、 stream

简短的定义就是“从支持数据处理操作的源生成的元素序列”

元素序列：就像集合一样，流也提供了一个接口，可以访问特定元素类型的一组有序值。因为集合是数据结构，所以它的主要目的是以特定的时间/空间复杂度存储和访问元素（如 ArrayList 与 LinkedList）。但流的目的在于表达计算，比如你前面见到的 filter、sorted 和 map。集合讲的是数据，流讲的是计算。

源：流会使用一个提供数据的源，如集合、数组或输入/输出资源。 请注意，从有序集合生成流时会保留原有的顺序。由列表生成的流，其元素顺序与列表一致。

数据处理操作：流的数据处理功能支持类似于数据库的操作，以及函数式编程语言中的常用操作，如 filter、map、reduce、find、match、sort 等。流操作可以顺序执行，也可并行执行。

流操作有两个重要的特点。

流水线：很多流操作本身会返回一个流，这样多个操作就可以链接起来，形成一个大的流水线。

内部迭代：与使用迭代器显式迭代的集合不同，流的迭代操作是在背后进行的。

filter 接受：Lambda，从流中排除某些元素。在本例中，通过传递 lambda d -> d.getCalories() > 500，选择出热量超过 500 卡路里的菜肴。
map：接受一个 Lambda，将元素转换成其他形式或提取信息。在本例中，通过传递方法引用 Dish::getName，相当于 Lambda d -> d.getName()，提取了每道菜的菜名。
limit 截断流，使其元素不超过给定数量。
collect：将流转换为其他形式。在本例中，流被转换为一个列表。

流与集合
集合讲的是数据，流讲的是计算。
集合是一个内存中的数据结构，它包含数据结构中目前所有的值。
流则是在概念上固定的数据结构（你不能添加或删除元素），其元素则是按需计算的。
流就像是一个延迟创建的集合：只有在消费者要求的时候才会计算值。
和迭代器类似，流只能遍历一次。
使用 Collection 接口需要用户去做迭代（比如用 for-each），这称为外部迭代。 相反，Streams 库使用内部迭代——它帮你把迭代做了，还把得到的流值存在了某个地方，你只要给出一个函数说要干什么就可以了。

流操作
java.util.stream.Stream 中的 Stream 接口定义了许多操作。它们可以分为两大类：
可以连接起来的流操作称为中间操作，关闭流的操作称为终端操作。
除非流水线上触发一个终端操作，否则中间操作不会执行任何处理。
终端操作会从流的流水线生成结果，其结果是任何不是流的值，比如 List、Integer，甚至 void。

使用流
流的使用一般包括三件事：
一个数据源（如集合）来执行一个查询；
一个中间操作链，形成一条流的流水线；
一个终端操作，执行流水线，并能生成结果。

Stream 对象的构建

```
// 1.使用值构建
Stream<String> stream = Stream.of("a", "b", "c");
// 2. 使用数组构建
String[] strArray = new String[] {"a", "b", "c"};
Stream<String> stream = Stream.of(strArray);
Stream<String> stream = Arrays.stream(strArray);
// 3. 利用集合构建(不支持 Map 集合)
List<String> list = Arrays.asList(strArray);
stream = list.stream();
```

对于基本数值型，目前有三种对应的包装类型 Stream：IntStream、LongStream、DoubleStream。当然我们也可以用 Stream<Integer>、Stream<Long> 、Stream<Double>，但是自动拆箱装箱会很耗时，所以特别为这三种基本数值型提供了对应的 Stream。
Java 8 中还没有提供其它基本类型数值的 Stream

数值 Stream 的构建:

```
IntStream stream1 = IntStream.of(new int[]{1, 2, 3});
//[1,3)
IntStream stream2 = IntStream.range(1, 3);
//[1,3]
IntStream stream3 = IntStream.rangeClosed(1, 3);
```

Stream 转换为其它类型:

```
Stream<String> stream = Stream.of("hello","world","tom");
// 1. 转换为 Array
String[] strArray    = stream.toArray(String[]::new);
// 2. 转换为 Collection
List<String> list1 = stream.collect(Collectors.toList());
List<String> list2 = stream.collect(Collectors.toCollection(ArrayList::new));
Set<String> set3 = stream.collect(Collectors.toSet());
Set<String> set4 = stream.collect(Collectors.toCollection(HashSet::new));
// 3. 转换为 String
String str = stream.collect(Collectors.joining()).toString();
```

特别注意 ：一个 Stream 只可以使用一次，上面的代码为了简洁而重复使用了多次。
这个代码直接运行会抛出异常的:

```
java.lang.IllegalStateException: stream has already been operated upon or closed
```

流操作:

中间操作和终端操作

操　作	类　型	返回类型	使用的类型/函数式接口	函数描述符
filter	中间	Stream<T>	Predicate<T>	T -> boolean
distinct	中间 (有状态-无界)	Stream<T>		
skip	中间 (有状态-有界)	Stream<T>	long	
limit	中间 (有状态-有界)	Stream<T>	long	
map	中间	Stream<R>	Function<T, R>	T -> R
flatMap	中间	Stream<R>	Function<T, Stream<R>>	T -> Stream<R>
sorted	中间 (有状态-无界)	Stream<T>	Comparator<T>	(T, T) -> int
anyMatch	终端	boolean	Predicate<T>	T -> boolean
noneMatch	终端	boolean	Predicate<T>	T -> boolean
allMatch	终端	boolean	Predicate<T>	T -> boolean
findAny	终端	Optional<T>		
findFirst	终端	Optional<T>		
forEach	终端	void	Consumer<T>	T -> void
collect	终端	R	Collector<T, A, R>	
reduce	终端 (有状态-有界)	Optional<T>	BinaryOperator<T>	(T, T) -> T
count	终端	long		

并行 Streams

Stream 有串行和并行两种，串行 Stream 上的操作是在一个线程中依次完成，而并行 Stream 则是在多个线程上同时执行。

可以通过对收集源调用 parallelStream 方法来把集合转换为并行流。

并行流就是一个把内容分成多个数据块，并用不同的线程分别处理每个数据块的流。

对顺序流调用 parallel 方法把流转换成并行流。

对顺序流调用 parallel 方法并不意味着流本身有任何实际的变化。它在内部实际上就是设了一个 boolean 标志，表示你想让调用 parallel 之后进行的所有操作都并行执行。

只需要对并行流调用 sequential 方法就可以把它变成顺序流。

```
stream.parallel()
      .filter(...)
      .sequential()
      .map(...)
      .parallel()
      .reduce();
```