讲一下JDK1.8的新特性

题目难度：★★

知识点标签：讲一下JDK1.8的新特性

学习时长：20分钟

题目描述

讲一下JDK1.8的新特性

解题思路

> 面试官问题可以从几个方面来回答：JDK1.8新特性

JDK1.8的新特性

1.default关键字

在java里面，我们通常都是认为接口里面是只能有抽象方法，不能有任何方法的实现的，那么在jdk1.8里面打破了这个规定，引入了新的关键字default，通过使用default修饰方法，可以让我们在接口里面定义具体的方法实现，如下。

```
public interface NewCharacter {

    public void test1();

    public default void test2(){
        System.out.println("我是新特性1");
    }

}
```

2.Lambda表达式

Lambda表达式是jdk1.8里面的一个重要的更新，这意味着java也开始承认了函数式编程，并且尝试引入其中。

首先，什么是函数式编程，引用廖雪峰先生的教程里面的解释就是说：函数式编程就是一种抽象程度很高的编程范式，纯粹的函数式编程语言编写的函数没有变量，因此，任意一个函数，只要输入是确定的，输出就是确定的，这种纯函数我们称之为没有副作用。而允许使用变量的程序设计语言，由于函数内部的变量状态不确定，同样的输入，可能得到不同的输出，因此，这种函数是有副作用的。函数式编程的一个特点就是，允许把函数本身作为参数传入另一个函数，还允许返回一个函数！

简单的来说就是，函数也是一等公民了，在java里面一等公民有变量，对象，那么函数式编程语言里面函数也可以跟变量，对象一样使用了，也就是说函数既可以作为参数，也可以作为返回值了，看一下下面这个例子。

```
        // 这是常规的Collections的排序的写法，需要对接口方法重写
        public void test1(){
          List<String> list =Arrays.asList("aaa","fsa","ser","eere");
          Collections.sort(list, new Comparator<String>() {
            @Override
            public int compare(String o1, String o2) {
              return o2.compareTo(o1);
            }
          });
          for (String string : list) {
            System.out.println(string);
          }
        }
        // 这是带参数类型的Lambda的写法
        public void testLamda1(){
          List<String> list =Arrays.asList("aaa","fsa","ser","eere");
          Collections.sort(list, (Comparator<? super String>) (String a,String
b)->{
            return b.compareTo(a);
          }
                          );
          for (String string : list) {
            System.out.println(string);
          }
        }
        // 这是不带参数的lambda的写法
        public void testLamda2(){
          List<String> list =Arrays.asList("aaa","fsa","ser","eere");
          Collections.sort(list, (a,b)->b.compareTo(a));
          for (String string : list) {
            System.out.println(string);
          }
        }
```

可以看到不带参数的写法一句话就搞定了排序的问题，所以引入lambda表达式的一个最直观的作用就是大大的简化了代码的开发，像其他一些编程语言Scala，Python等都是支持函数式的写法的。当然，不是所有的接口都可以通过这种方法来调用，只有函数式接口才行，jdk1.8里面定义了好多个函数式接口，我们也可以自己定义一个来调用，下面说一下什么是函数式接口。

3.函数式接口

定义：“函数式接口”是指仅仅只包含一个抽象方法的接口，每一个该类型的lambda表达式都会被匹配到这个抽象方法。jdk1.8提供了一个@FunctionalInterface注解来定义函数式接口，如果我们定义的接口不符合函数式的规范便会报错。

```
@FunctionalInterface
public interface MyLamda {

    public void test1(String y);

    // 这里如果继续加一个抽象方法便会报错
    //    public void test1();

    // default方法可以任意定义
    default String test2(){
        return "123";
    }

    default String test3(){
        return "123";
    }

    // static方法也可以定义
    static void test4(){
        System.out.println("234");
    }

}
// 接口的调用
MyLamda m = y -> System.out.println("ss"+y);
```

4.方法与构造函数引用

jdk1.8提供了另外一种调用方式::，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，即`ClassName :: methodName`。例如，`Apple::getWeight`就是引用了Apple类中定义的方法getWeight。请记住，不需要括号，因为你没有实际调用这个方法。方法引用就是Lambda表达式`(Apple a) -> a.getWeight()`的快捷写法，如下示例。

```
// 先定义一个函数式接口
@FunctionalInterface
public interface TestConverT<T, F> {
    F convert(T t);
}
// 调用方式
public void test(){
    TestConverT<String, Integer> t = Integer::valueOf;
    Integer i = t.convert("111");
    System.out.println(i);
}
```

此外，对于构造方法也可以这么调用。

```
//实体类User和它的构造方法
```

```
public class User {
    private String name;

    private String sex;

    public User(String name, String sex) {
        super();
        this.name = name;
        this.sex = sex;
    }
}
//User工厂
public interface UserFactory {
    User get(String name, String sex);
}
//测试类
    UserFactory uf = User::new;
    User u = uf.get("ww", "man");
```

5.局部变量限制

Lambda表达式也允许使用自由变量（不是参数，而是在外层作用域中定义的变量），就像匿名类一样。 它们被称作捕获Lambda。 Lambda可以没有限制地捕获（也就是在其主体中引用）实例变量和静态变量。但局部变量必须显式声明为final，或事实上是final。

为什么局部变量有这些限制?

（1）实例变量和局部变量背后的实现有一个关键不同。实例变量都存储在堆中，而局部变量则保存在栈上。如果Lambda可以直接访问局部变量，而且Lambda是在一个线程中使用的，则使用Lambda的线程，可能会在分配该变量的线程将这个变量收回之后，去访问该变量。因此， Java在访问自由局部变量时，实际上是在访问它的副本，而不是访问原始变量。如果局部变量仅仅赋值一次那就没有什么区别了——因此就有了这个限制。

（2）这一限制不鼓励你使用改变外部变量的典型命令式编程模式。

```
final int num = 1;
Converter<Integer, String> stringConverter =
        (from) -> String.valueOf(from + num);
stringConverter.convert(2);
```

6.Date Api更新

1.8之前JDK自带的日期处理类非常不方便，我们处理的时候经常是使用的第三方工具包，比如commons-lang包等。不过1.8出现之后这个改观了很多，比如日期时间的创建、比较、调整、格式化、时间间隔等。这些类都在java.time包下。比原来实用了很多。

6.1 LocalDate/LocalTime/LocalDateTime

LocalDate为日期处理类、LocalTime为时间处理类、LocalDateTime为日期时间处理类，方法都类似，具体可以看API文档或源码，选取几个代表性的方法做下介绍。

now相关的方法可以获取当前日期或时间，of方法可以创建对应的日期或时间，parse方法可以解析日期或时间，get方法可以获取日期或时间信息，with方法可以设置日期或时间信息，plus或minus方法可以增减日期或时间信息；

*6.2*TemporalAdjusters

这个类在日期调整时非常有用，比如得到当月的第一天、最后一天，当年的第一天、最后一天，下一周或前一周的某天等。

6.3DateTimeFormatter

以前日期格式化一般用SimpleDateFormat类，但是不怎么好用，现在1.8引入了DateTimeFormatter类，默认定义了很多常量格式（ISO打头的），在使用的时候一般配合LocalDate/LocalTime/LocalDateTime使用，比如想把当前日期格式化成yyyy-MM-dd hh:mm:ss的形式:

```
LocalDateTime dt = LocalDateTime.now();
DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy-MM-dd hh:mm:ss");

System.out.println(dtf.format(dt));
```

7.流

定义：流是Java API的新成员，它允许我们以声明性方式处理数据集合（通过查询语句来表达，而不是临时编写一个实现）。就现在来说，我们可以把它们看成遍历数据集的高级迭代器。此外，流还可以透明地并行处理，也就是说我们不用写多线程代码了。

Stream 不是集合元素，它不是数据结构并不保存数据，它是有关算法和计算的，它更像一个高级版本的 Iterator。原始版本的 Iterator，用户只能显式地一个一个遍历元素并对其执行某些操作；高级版本的 Stream，用户只要给出需要对其包含的元素执行什么操作，比如 “过滤掉长度大于 10 的字符串”、“获取每个字符串的首字母”等，Stream 会隐式地在内部进行遍历，做出相应的数据转换。

Stream 就如同一个迭代器（Iterator），单向，不可往复，数据只能遍历一次，遍历过一次后即用尽了，就好比流水从面前流过，一去不复返。而和迭代器又不同的是，Stream 可以并行化操作，迭代器只能命令式地、串行化操作。顾名思义，当使用串行方式去遍历时，每个 item 读完后再读下一个 item。而使用并行去遍历时，数据会被分成多个段，其中每一个都在不同的线程中处理，然后将结果一起输出。Stream 的并行操作依赖于 Java7 中引入的 Fork/Join 框架（JSR166y）来拆分任务和加速处理过程。

流的操作类型分为两种：

- Intermediate：一个流可以后面跟随零个或多个 intermediate 操作。其目的主要是打开流，做出某种程度的数据映射/过滤，然后返回一个新的流，交给下一个操作使用。这类操作都是惰性化的（lazy），就是说，仅仅调用到这类方法，并没有真正开始流的遍历。
- Terminal：一个流只能有一个 terminal 操作，当这个操作执行后，流就被使用“光”了，无法再被操作。所以这必定是流的最后一个操作。Terminal 操作的执行，才会真正开始流的遍历，并且会生成一个结果，或者一个 side effect。

在对于一个 Stream 进行多次转换操作 (Intermediate 操作)，每次都对 Stream 的每个元素进行转换，而且是执行多次，这样时间复杂度就是 N（转换次数）个 for 循环里把所有操作都做掉的总和吗？其实不是这样的，转换操作都是 lazy 的，多个转换操作只会在 Terminal 操作的时候融合起来，一次循环完成。我们可以这样简单的理解，Stream 里有个操作函数的集合，每次转换操作就是把转换函数放入这个集合中，在 Terminal 操作的时候循环 Stream 对应的集合，然后对每个元素执行所有的函数。

```
// 构造流的几种方式
// 1. Individual values
Stream stream = Stream.of("a", "b", "c");
// 2. Arrays
String [] strArray = new String[] {"a", "b", "c"};
stream = Stream.of(strArray);
stream = Arrays.stream(strArray);
// 3. Collections
List<String> list = Arrays.asList(strArray);
stream = list.stream();
```

8.Objects方法新特性

```
//比较两个对象是否相等（首先比较内存地址，然后比较a.equals(b)，只要符合其中之一返回true）
public static boolean equals(Object a, Object b);

//深度比较两个对象是否相等(首先比较内存地址,相同返回true;如果传入的是数组，则比较数组内的对应下标值是否相同)
public static boolean deepEquals(Object a, Object b);

//返回对象的hashCode，若传入的为null，返回0
public static int hashCode(Object o);

//返回传入可变参数的所有值的hashCode的总和（这里说总和有点牵强，具体参考Arrays.hashCode()方法）
public static int hash(Object... values);

//返回对象的String表示，若传入null，返回null字符串
public static String toString(Object o)

//返回对象的String表示，若传入null，返回默认值nullDefault
public static String toString(Object o, String nullDefault)

//使用指定的比较器c 比较参数a和参数b的大小（相等返回0，a大于b返回整数，a小于b返回负数）
public static <T> int compare(T a, T b, Comparator<? super T> c)

//如果传入的obj为null抛出NullPointerException,否者返回obj
public static <T> T requireNonNull(T obj)

//如果传入的obj为null抛出NullPointerException并可以指定错误信息message,否者返回obj
public static <T> T requireNonNull(T obj, String message)
```

```
----------------------------以下是jdk8新增方法--------------------------

//判断传入的obj是否为null，是返回true,否者返回false
public static boolean isNull(Object obj)

//判断传入的obj是否不为null，不为空返回true,为空返回false （和isNull()方法相反）
public static boolean nonNull(Object obj)

//如果传入的obj为null抛出NullPointerException并且使用参数messageSupplier指定错误信息，
否者返回obj
public static <T> T requireNonNull(T obj, Supplier<String> messageSupplier)
```

```
package cn.cupcat.java8;

import org.junit.Test;

import java.util.Comparator;
import java.util.Objects;

/**
 * Created by xy on 2017/12/25.
 */
public class ObjectsTest {

    /**
     *  因为Objects类比较简单，所以只用这一个测试用例进行测试
     * */
    @Test
    public void equalsTest(){
        String str1 = "hello";
        String str2 = "hello";
        //传入对象
        //Objects.equals(str1, str2) ?  true
        boolean equals = Objects.equals(str1, str2);
        System.out.println("Objects.equals(str1, str2) ?  "+ equals);



    }
    @Test
    public void deepEqualsTest(){
        String str1 = "hello";
        String str2 = "hello";
        //传入对象
        boolean deepEquals = Objects.deepEquals(str1, str2);
        //Objects.deepEquals(str1, str2) ?  true
        System.out.println("Objects.deepEquals(str1, str2) ?  "+ deepEquals);
```

```
        int[] arr1 = {1,2};
        int[] arr2 = {1,2};
        //传入数组
        deepEquals = Objects.deepEquals(arr1, arr2);
        //Objects.deepEquals(arr1, arr2) ?  true
        System.out.println("Objects.deepEquals(arr1, arr2) ?  "+ deepEquals);
    }
    @Test
    public void hashCodeTest(){
        String str1 = "hello";

        //传入对象
        int hashCode = Objects.hashCode(str1);
        //Objects.hashCode(str1) ?  99162322
        System.out.println("Objects.hashCode(str1) ?  "+ hashCode);

        //传入null
        hashCode = Objects.hashCode(null);
        //Objects.hashCode(null) ?  0
        System.out.println("Objects.hashCode(null) ?  "+ hashCode);


    }

    @Test
    public void hashTest(){

        int  a  = 100;

        //传入对象
        int hashCode = Objects.hashCode(a);
        //Objects.hashCode(str1) ?  100
        System.out.println("Objects.hashCode(str1) ?  "+ hashCode);

        //输入数组
        int[] arr = {100,100};
        hashCode = Objects.hash(arr);
        //Objects.hashCode(arr) ?  1555093793
        System.out.println("Objects.hashCode(arr) ?  "+ hashCode);
    }


    @Test
    public void compareTest(){
        int a = 10;
        int b = 11;
        int compare = Objects.compare(a, b, new Comparator<Integer>() {
            @Override
            public int compare(Integer o1, Integer o2) {
```

```
                return o1.compareTo(o2);
            }
        });
        // compare = -1
        System.out.println(" compare = "+ compare);

    }

    @Test
    public void requireNonNullTest(){
        String test = null;
        //java.lang.NullPointerException
       // String s = Objects.requireNonNull(test);

        //java.lang.NullPointerException: 这是空指针异常提示的信息
        //String s = Objects.requireNonNull(test, "这是空指针异常提示的信息");


        //java.lang.NullPointerException: 我是返回的异常信息
        String s = Objects.requireNonNull(test,()->"我是返回的异常信息");
    }
}
```

总结

说清楚JDK1.8新特性，已经做了哪些优化。