一、 JDK8 新特性

1. 基本函式数接口

函数式接口(Functional Interface)就是一个有且仅有一个抽象方法，但是可以有多个非抽象方法的接口。 函数式接口可以被隐式转换为 lambda 表达式。

基本的函数式接口主要有四个：

Supplier<T> 生产者：无输入，生产一个 T 类型的值；接口仅包含一个无参的方法: T get() 。用来获取一个泛型参数指定类型的对象数据

Consumer<T> 消费者：输入一个 T 类型的值，无输出；对给定参数执行消费操作

Function<T,R> 函数：输入一个 T 类型的值，返回一个 R 类型的值；

Predicate<T> 断言：输入一个 T 类型的值，返回 true/false

2. Lambda 表达式

Lambda 它其实是匿名函数，通过约定好怎么传入参数，怎么返回参数，由编译器负责参数类型的猜测并执行结果。

Lambda 表达式的基本语法

```
@Test
public void example1(){
    //语法格式
    //(parameters) -> expression
    //(parameters) -> { statements; }
    //即"->"操作符将 Lambda 表达式分为两个部分：左侧为参数列表，右侧为 Lambda 体。
    //每一个 Lambda 表达的返回值都是一个函数式编程的接口
}
```

com.open.newfeature.jdk8/lambdaTest.java

```
package com.open.newfeature.jdk8;

import org.junit.Test;

import java.util.function.*;

public class lambdaTest {

    /**
     * 无参数，无返回
     */
    @Test
    public void example1() {
        Runnable hello_lambda1 = () -> System.out.println("Hello Lambda");
        hello_lambda1.run();
        //或
        Runnable hello_lambda2 = () -> {
            System.out.println("Hello Lambda");
        };
        hello_lambda2.run();
    }
```

```java
/**
 * 无参数，有返回
 */
@Test
public void example2(){
    Supplier<Integer> supplier1 = () -> 10;
    System.out.println(supplier1.get());
    //或
    Supplier supplier2 = () -> {
        return 10;
    };
    System.out.println(supplier2.get());
}

/**
 * 有一个参数，无返回
 */
@Test
public void example3(){
    Consumer<String> consumer1 = x -> System.out.println(x);
    consumer1.accept("Hello Lambda1");
    //或
    Consumer<String> consumer2 = (x) -> System.out.println(x);
    consumer2.accept("Hello Lambda2");
    //或
    Consumer<String> consumer3 = x -> {
        System.out.println(x);
    };
    consumer3.accept("Hello Lambda3");
}

/**
 * 有一个参数，有返回值
 */
@Test
public void example4(){
    IntFunction<Integer> intFunction1 = x -> x + 10;
    System.out.println(intFunction1.apply(10));
    //或
    IntFunction<Integer> intFunction2 = (x) -> {
        return x + 10;
    };
    System.out.println(intFunction2.apply(10));
```

```
    }

    /**
     * 有多个参数，无返回值
     */
    @Test
    public void example5(){
        ObjIntConsumer<Integer> objIntConsumer = (x, y) -> {
            System.out.println(x + y);
        };
        objIntConsumer.accept(10,20);
        //或
        BiConsumer<Integer,Integer> biConsumer = (Integer x, Integer y)
-> {
            System.out.println(x + y);
        };
        biConsumer.accept(10,20);
    }

    /**
     * 有多个参数，有返回值
     */
    @Test
    public void example6(){
        IntBinaryOperator intBinaryOperator = (x,y) -> {
            return x > y ? x : y;
        };
        System.out.println(intBinaryOperator.applyAsInt(10,20));
        //或
        BinaryOperator<Integer> binaryOperator = (Integer x, Integer y)
-> {
            return x > y ? x : y;
        };
        System.out.println(binaryOperator.apply(10,20));
    }


}
```

3. **Optional 类的使用**

Optional 类的作用主要是为了解决空指针问题，通过对结果的包装，并使用方法来代替 if 判断，为流式编程打下了良好的基础。

Optional 类初始化

com.open.newfeature.jdk8/optionalTest.java

```
package com.open.newfeature.jdk8;
```

```java
import org.junit.Test;

import java.util.Optional;

public class optionalTest {
    /**
     * Optional 的初始化
     * Optional 的构造方法是私有的，所有只能通过静态方法去初始化
     */
    @Test
    public void example1(){
        //of 为指定的类型值创建一个 Optional，如果给定的值为 null 则会报空指针异常
        Optional<String> optional = Optional.of("Hello Optional");
        System.out.println(optional.get()); //打印：Hello Optional

        //ofNullable 为指定的类型值创建一个 Optional，它与 of 的区别在于可以传 null，当值为 null 时则创建一个空的 Optional
        Optional<Object> o = Optional.ofNullable(null);
        System.out.println(o instanceof Optional); //打印：true

        //empty 获取 Optional 的静态初始化空对象，单例实现，全局唯一
        Optional<Object> empty = Optional.empty();
        System.out.println(o == empty); //打印：true
    }

    /**
     * Optional 的 isPresent 方法：如果值存在返回 true，否则返回 false
     */
    @Test
    public void example2(){
        Optional<Object> o = Optional.ofNullable(null);
        System.out.println(o.isPresent()); //打印：false

        Optional<String> optional = Optional.ofNullable("Hello Optional");
        System.out.println(optional.isPresent()); //打印：true
    }

    /**
     * Optional 的 get 方法：如果值存在则返回，否则抛出 NoSuchElementException 异常
     */
    @Test
```

```
    public void example3(){
        Optional<String> optional = Optional.ofNullable("Hello
Optional");
        System.out.println(optional.get()); //打印：Hello Optional

        Optional<Object> o = Optional.ofNullable(null);
        System.out.println(o.get()); //抛出：NoSuchElementException 异常
    }
}
```

4. **Stream 流式编程**

Stream API 借助 Lambda 表达式，提供串行和并行两种模式进行汇聚操作，并行模式能够充分利用多核处理器的优势，使用 fork/join 来拆分任务和加速处理过程。

com.open.newfeature.jdk8/streamTest.java

```
package com.open.newfeature.jdk8;
import org.junit.Test;
import java.util.Arrays;
import java.util.List;

public class streamTest {
    @Test
    public void example1(){
        final List<String> stringList =
Arrays.asList("Tom","Jack","Alice","Lina","Pony");

        //串行流
        long count1 = stringList.stream()
                .filter(s -> {
                    System.out.println(Thread.currentThread().getId() +
"==" + Thread.currentThread().getName());
                    return s.length() > 3;
                })
                .count();
        System.out.println(count1);

        //并行流
        long count2 = stringList.parallelStream()
                .filter(s -> {
                    System.out.println(Thread.currentThread().getId() +
"==" + Thread.currentThread().getName());
                    return s.length() > 3;
                })
                .count();
        System.out.println(count2);
    }
```

```
}
```

5. **方法引用(::双冒号操作符)**

简单来说就是一个 Lambda 表达式，方法引用提供了一种引用而不执行方法的方式，运行时方法引用会创建一个函数式接口的实例。

com.open.newfeature.jdk8/funRefTest.java

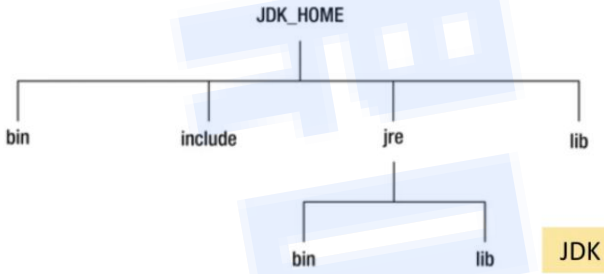
```java
package com.open.newfeature.jdk8;

import org.junit.Test;

import java.util.function.Consumer;

public class funRefTest {

    @Test
    public void example1(){
        //使用 Lambda 表达式
        Consumer<String> consumer1 = x -> System.out.println(x);
        consumer1.accept("Lambda 表达式");

        //使用方法引用
        Consumer<String> consumer2 = System.out::println;
        consumer2.accept("方法引用::");
    }

}
```

二、 JDK9 新特性

1. 目录变化

9、10、11、12 面向开发者的新特性其实并不是很多，大部分都是一些优化、收集器加强以及增加了一些新功能等。

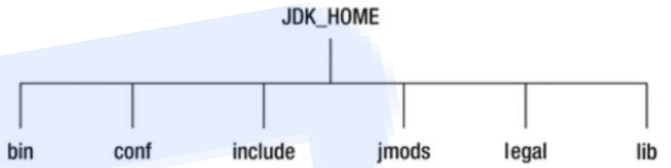
JDK1.8 及以前的目录



JDK 8 的目录结构

bin 目录	包含命令行开发和调试工具，如javac，jar和javadoc。
include目录	包含在编译本地代码时使用的C/C++头文件
lib 目录	包含JDK工具的几个JAR和其他类型的文件。 它有一个tools.jar文件，其中包含javac编译器的Java类
jre/bin 目录	包含基本命令，如java命令。 在Windows平台上，它包含系统的运行时动态链接库（DLL）。
jre/lib 目录	包含用户可编辑的配置文件，如.properties和.policy文件。包含几个JAR。rt.jar文件包含运行时的Java类和资源文件。

JDK1.9 目录



JDK 9 的目录结构

没有名为jre的子目录	
bin 目录	包含所有命令。 在Windows平台上，它继续包含系统的运行时动态链接库。
conf 目录	包含用户可编辑的配置文件，例如以前位于jre\lib目录中的.properties和.policy文件
include 目录	包含要在以前编译本地代码时使用的C/C++头文件。 它只存在于JDK中
jmods 目录	包含JMOD格式的平台模块。 创建自定义运行时映像时需要它。 它只存在于JDK中
legal 目录	包含法律声明
lib 目录	包含非Windows平台上的动态链接本地库。 其子目录和文件不应由开发人员直接编辑或使用

模块化系统

Java 已经 发展超过 20 年(95 年最初发布)，Java 和相关生态在不断丰富的同时也越来越暴露出一些问题:

Java 运行环境的膨胀和臃肿，每次 JVM 启动的时候，至少会有 30~60MB 的内存加载，主要原因是 JVM 需要加载 rt.jar，不管其中的类是否被 classloader 加载，第一步整个 jar 都会被 JVM 加载到内存当中去。

而模块化就是解决这问题，按需加载。

当代码库越来越大，创建复杂，不同版本的类库交叉依赖导致让人头疼的问题，这些都阻碍了 Java 开发和 运行效率的提升很难真正地对代码进行封装，系统并没有对不同 JAR 文件之间的依赖关系有个明确的概念。每一个公共类都可以被类路径之下任何其它的公共类所访问到，这样就会导致无意中使用了并不想被公开访问的 API。

实现目标

模块化的主要目的在于减少内存的开销

只须必要模块，而非全部 jdk 模块

改进 Java SE 平台，使其可以适应不同大小的计算设备

改进其安全性，可维护性，提高性能

2. 接口私有方法

JDK8 新增了静态方法和默认方法，但是不支持私有方法。

JDK9 中新增了私有方法

jdk9_12/OrderPay.java

```
package com.open.newfeature.jdk9_12;

public interface OrderPay {
    void pay();

    default void defaultPay() {
```

```
        privateMethod();
    }

    //接口的私有方法可以在 JDK9 中使用
    private void privateMethod() {
        System.out.println("调用接口的私有方法");
    }
}
```

jdk9_12/OrderPayImpl.java

```
package com.open.newfeature.jdk9_12;

public class OrderPayImpl implements OrderPay {
    @Override
    public void pay() {
        System.out.println("OrderPayImpl 实现了 pay()方法");
    }

    public static void main(String[] args) {
        OrderPayImpl orderPay = new OrderPayImpl();
        orderPay.defaultPay();
        orderPay.pay();
    }
}
```

3. **集合加强**

jdk9 为所有集合（List/Set/Map）都增加了 of 和 copyOf 方法，用来创建不可变集合，即一旦创建就无法再执行添加、删除、替换、排序等操作，否则将报 java.lang.UnsupportedOperationException 异常。

jdk9_12/newFeature.java

```
@Test
public void example1() {
    List<String> list = new ArrayList<>();
    list.add("a");
    list.add("b");
    list.add("c");
    //设置为只读 list
    list = Collections.unmodifiableList(list);
    System.out.println(list);
    //执行 add 操作时会抛出 java.lang.UnsupportedOperationException 的异常
    list.add("d");

}
```

```java
@Test
public void example2() {
    List<String> list = List.of("a", "b", "c");
    System.out.println(list);
    Set<String> set = Set.of("a", "b", "c");
    System.out.println(set);
    Map<String, String> map = Map.of("k1", "a", "k2", "b");
    System.out.println(map);
}
```

4. **Stream 方法增强**

Stream 中增加了新的方法 ofNullable、dropWhile、takeWhile 和 iterate。

takeWhile()方法：从 Stream 中获取一部分数据，返回从头开始的尽可能多的元素，直到遇到第一个 false 结果（不包含该元素），如果第一个值不满足断言条件，将返回一个空的 Stream

dropWhile()方法：与 takeWhile()相反，返回第一个 false 结果和其之后的元素，和 takeWhile()方法形成互补

jdk9_12/newFeature.java

```java
@Test
public void example3() {
    List<String> list1 = List.of("SpringBoot", "", "SpringCloud",
"Redis").stream()
    .takeWhile(obj -> !obj.isEmpty()).collect(Collectors.toList());
    System.out.println(list1);

    List<String> list2 = List.of("SpringBoot", "", "SpringCloud",
"Redis").stream()
   .dropWhile(obj -> !obj.isEmpty()).collect(Collectors.toList());
    System.out.println(list2);
 }
```

5. **jshell 命令**

jdk9 引入了 jshell 这个交互性工具，让 Java 也可以像脚本语言一样来运行，可以从控制台启动 jshell ，在 jshell 中直接输入表达式并查看其执行结果。当需要测试一个方法的运行效果，或是快速的对表达式进行求值时，jshell 都非常实用。

```
C:\jdk12\bin>jshell
|  欢迎使用 JShell -- 版本 12.0.2
|  要大致了解该版本, 请键入: /help intro

jshell> System.out.println("ok");
ok

jshell> int n=20;
n ==> 20

jshell> int m=30;
m ==> 30

jshell> int k=m+n;
k ==> 50

jshell> System.out.println(k);
50
```

三、 JDK10 新特性

1. 局部变量类型推断

JDK10 可以使用var 作为局部变量类型推断标识符

仅适用于局部变量，如增强 for 循环的索引，传统 for 循环局部变量

不能使用于方法形参、构造函数形参、方法返回类型或任何其他类型的变量声明

标识符 var 不是关键字，而是一个保留类型名称，而且不支持类或接口叫 var，也不符合命名规范

jdk9_12/newFeature.java

```
@Test
public void testVar() {
    var strVar = "SpringBoot";
    System.out.println(strVar instanceof String);//true


    //repeat(int count)方法:用于字符串循环输出
    System.out.println(strVar.repeat(2));

    var flagVar = Boolean.valueOf(true);
    System.out.println(flagVar instanceof Boolean);//true

    for (var i = 0; i < 10; ++i) {
        System.out.println(i);
    }
}
```

四、 JDK11 新特性

1. 字符串加强

jdk9_12/newFeature.java

```
@Test
public void testString() {
```

```
    " ".isBlank(); // true
    // 去除首尾空格
    " Javastack ".strip(); // "Javastack"
    // 去除尾部空格
    " Javastack ".stripTrailing();
    // 去除首部空格
    " Javastack ".stripLeading(); // "Javastack "
    // 复制字符串
    System.out.println("Java".repeat(3));// "JavaJavaJava"
    // 行数统计
    System.out.println("A\nB\nC".lines().count()); // 3
}
```

2. HttClient Api

新增 Http 客户端 HTTP，用于传输网页的协议，早在 1997 年就被采用在目前的 1.1 版本中。直到 2015 年，HTTP2 才成为标准。

HTTP/1.1 和 HTTP/2 的主要区别是如何在客户端和服务器之间构建和传输数据 。HTTP/1.1 依赖于请求/响应周期。HTTP/2 允许服务器“push”数据：它可以发送比客户端请求更多的数据。

这是 Java 9 开始引入的一个处理 HTTP 请求的 HTTP Client API，该 API 支持同步和异步，而在 Java 11 中已经为正式可用状态，你可以在 java.net 包中找到这个 API 它将替代仅适用于 blocking 模式的 HttpURLConnection，并提供对 WebSocket 和 HTTP/2 的支持。

HttpClient.Builder：HttpClient 构建工具类

HttpRequest.Builder：HttpRequest 构建工具类

HttpRequest.BodyPublisher：将 Java 对象转换为可发送的 HTTP Request Body 字节流，如 form 表单提交

HttpResponse.BodyHandler：处理接收到的 Response Body

3. 新一代的垃圾回收器 ZGC

ZGC，这应该是 JDK11 最为瞩目的特性，没有之一。但是后面带了 Experimental，说明这还不建议用到生产环境。

ZGC 是一个并发,基于 region，压缩型的垃圾收集器,只有 root 扫描阶段会 STW(stop the world)，因此 GC 停顿时间不会随着堆的增长和存活对象的增长而变长。

优势

GC 暂停时间不会超过 10ms

既能处理几百兆的小堆，也能处理几个 T 的大堆(OMG)

和 G1 相比，应用吞吐能力不会下降超过 15%

初始只支持 64 位系统

五、 JDK12 新特性

1. Switch Expressions

switch 语句如果漏写了一个 break，那么逻辑往往就跑偏了，这种方式既繁琐，又容易出错。如果换成 switch 表达式，能够自然地保证只有单一路径会被执行。

```
switch (day) {
    case MONDAY:
    case FRIDAY:
    case SUNDAY:
        System.out.println(6);
        break;
    case TUESDAY:
        System.out.println(7);
        break;
    case THURSDAY:
    case SATURDAY:
        System.out.println(8);
        break;
    case WEDNESDAY:
        System.out.println(9);
        break;
}
```

现在可以用以下写法

```
switch (day) {
    case MONDAY, FRIDAY, SUNDAY -> System.out.println(6);
    case TUESDAY                -> System.out.println(7);
    case THURSDAY, SATURDAY     -> System.out.println(8);
    case WEDNESDAY              -> System.out.println(9);
}
```

2. **Shenandoah GC**

新增了一个名为 Shenandoah 的 GC 算法，通过与正在运行的 Java 线程同时进行 evacuation（撤离） 工作来减少 GC 暂停时间。使用 Shenandoah 的暂停时间与堆大小无关，这意味着无论堆是 200 MB 还是 200 GB，都将具有相同的暂停时间。

六、 JDK13 新特性

1. **默认生成类数据共享归档文件**

在同一个虚拟机上启动多个 JVM 时，如果每个虚拟机都单独装载自己需要的所有类，启动成本和内存占用成本很高。所以 java 团队引入了类数据共享机制（CDS），将一些核心类在每个 JVM 中共享，使启动时间减少，JVM 内存占用减少。

2. **可中断的 G1 混合 GC**

当 G1 垃圾回收器的回收超过暂停时间的目标，能中止垃圾回收过程，目的是使用户能够设置预期的 JVM 停顿时间。

G1 将 GC 回收集分为了 mandatory（紧急）和 optional（可选）两部分，若处理完 mandatory 后的时间小于设置的预期时间则会继续处理 optional。

3. **ZGC 取消提交未使用的内存**

ZGC 从 JDK11 中被引入进来，在进行 GC 的时候保证更短的停顿时间，10ms 以下，

在 JDK13 中新增了归还未提交，未使用的内存给操作系统

4. **文字块**

将文本块添加至 java 语言，多行字符串不需要转义，提高代码的可读性。

HTML 示例

使用一维字符串：

```
String html = "<html>\n" +
              "    <body>\n" +
              "        <p>Hello, world</p>\n" +
              "    </body>\n" +
              "</html>\n";
```

使用二维文本块

```
String html = """
              <html>
                  <body>
                      <p>Hello, world</p>
                  </body>
              </html>
              """;
```

SQL 示例

使用一维字符串：

```
String query = "SELECT `EMP_ID`, `LAST_NAME` FROM `EMPLOYEE_TB`\n" +
               "WHERE `CITY` = 'INDIANAPOLIS'\n" +
               "ORDER BY `EMP_ID`, `LAST_NAME`;\n";
```

使用二维文本块：

```
String query = """
               SELECT `EMP_ID`, `LAST_NAME` FROM `EMPLOYEE_TB`
               WHERE `CITY` = 'INDIANAPOLIS'
               ORDER BY `EMP_ID`, `LAST_NAME`
                """;
```