Java基础学习总结

函数式编程

> 函数式编程（Functional Programming）

- OO（object oriented,面向对象）是抽象数据
- FP（Functional programming，函数式编程）是抽象行为

约束

函数式编程是安全的，它增加了一些约束，即所有的数据（变量、对象）必须是不可变的，设置一次，永不改变。不修改自身外部的任何东西（函数范围之外的元素）

优点

不可变对象范式解决了**并发编程**中棘手问题（当程序的某些部分在多处理器同时运行时），意味着不同的处理器可以尝试同时修改同一块内存。

如果函数永远不会修改现有值而是产生新值，则不会对内存产生争用，这是函数式语言天生的优点。

函数式语言作为并行编程的其中之一的解决方案。

Lambda 表达式

Lambda 表达式使用最小可能的语法编写的函数定义

- Lambda 表达式产生函数，而不是类。
- Lambda 语法尽可能少，这使 Lambda 易于编写和使用

```
interface Description {
  String brief();
}

interface Body {
  String detailed(String head);
}

interface Multi {
  String twoArg(String head, Double d);
}

public class LambdaExpressions {

  static Body bod = h -> h + " No Parens!";      // [1]

  static Body bod2 = (h) -> h + " More details"; // [2]

  static Description desc = () -> "Short info";  // [3]

  static Multi mult = (h, n) -> h + n;           // [4]

  static Description moreLines = () -> {         // [5]
    System.out.println("moreLines()");
```

```
    return "from moreLines()";
  };

  public static void main(String[] args) {
    System.out.println(bod.detailed("Oh!"));
    System.out.println(bod2.detailed("Hi!"));
    System.out.println(desc.brief());
    System.out.println(mult.twoArg("Pi! ", 3.14159));
    System.out.println(moreLines.brief());
  }
}
```

> output:
>
> Oh! No Parens!
> Hi! More details
> Short info
> Pi! 3.14159
> moreLines()
> from moreLines()

语法

> a -> c

a： 参数

-> : 表示为产出

c : 方法体

上例中展示了 Lambda 的几种使用方法

[1] 当使用一个参数时，可以不需要括号()

[2] 使用括号 () 包裹参数

[3] 如果没有参数则必须使用括号 () 表示空参数列表

[4] 当使用多个参数，将参数列表包裹在括号 () 中

[5] 如果方法体一行写不完，需要使用花括号 {} 将方法体包裹其中，这种情况下需要使用return 来返回

方法引用

> 方法引用组成：类名或对象名 :: 方法名称

```
interface Callable {                        // [1]
    void call(String s);
}

class Describe {
    // show() 的签名（参数和返回类型）符合 Callable 的 call()签名
    void show(String msg) {                    // [2]
        System.out.println(msg);
    }
}

public class MethodReferences {
    // hello() 符合call() 签名
```

```
    static void hello(String name) {          // [3]
        System.out.println("Hello, " + name);
    }

    static class Description {
        String about;

        Description(String desc) {
            about = desc;
        }
        // help() 符合call() 签名，它是静态内部类中的非静态方法
        void help(String msg) {                // [4]
            System.out.println(about + " " + msg);
        }
    }

    static class Helper {
        // assist() 是静态内部类中的静态方法
        static void assist(String msg) {       // [5]
            System.out.println(msg);
        }
    }

    public static void main(String[] args) {
        Describe d = new Describe();
        // 简单理解 映射，将describe.show() 映射给 Callable.call() 方法上
        Callable c = d::show;                  // [6]
        // 通过调用 call() 来调用 show()，java 将 call() 映射到show()
        c.call("call()");                      // [7]
        // 静态方法引用
        c = MethodReferences::hello;           // [8]
        c.call("Bob");
        // 映射：将Description.help()方法映射到 Callable.call() 上
        c = new Description("valuable")::help; // [9]
        c.call("information");

        c = Helper::assist;                    // [10]
        c.call("Help!");
    }
}
```

> output:
>
> call()
> Hello, Bob
> valuable information
> Help!

未绑定的方法引用

> 未绑定的方法引用是指没有关联对象的普通（非静态）方法。
>
> 使用未绑定的引用时，我们必须先提供对象

```
class X {
    String f() {
        return "X::f()";
```

```
    }
}

interface MakeString {
    String make();
}

interface TransformX {
    String transform(X x);
}

public class UnboundMethodReference {

    public static void main(String[] args) {
        // MakeString ms = X::f;  不可以使用，因为需要绑定方法引用        // [1]
        MakeString makeString = new MakeString() {
            @Override
            public String make() {
                return "hah";
            }
        };
        System.out.println(makeString.make());

        //  TransformX sp1 = X::f; 和下面表达的意思是一样的
        TransformX sp = new TransformX() {
            @Override
            public String transform(X x) {
                return x.f();
            }
        };

        X x = new X();
        System.out.println(sp.transform(x));    // [2]
        System.out.println(x.f()); // Same effect
    }
}
/* Output:
X::f()
X::f()
*/
```

- [1] 我们尝试把 X 的 f() 方法引用赋值给 MakeString。
  - 虽然 make() 和 f() 具有相同的签名，但是编译不通过，会报无效引用（invalid method reference）错误。
  - 这是因为实际上还有一个隐藏的参数：`this` 。你不能在没有 X 对象的前提下调用 f() 。因此 X::f 表示未绑定的方法引用，因为它尚未绑定到对象。
  - 要解决这个问题，我们需要一个 X 对象，索引我们接口中需要一个额外的参数，如上例的 TransformX
  - 如果将 X : : f 赋值给 TransformX，在 Java 中是允许的。
- 使用**未绑定的引用时**，函数式方法的签名（接口中的单个方法）不再与方法引用的签名完全匹配：原因是：需要一个对象来调用方法

未绑定方法与多参数的结合使用

```
class This {
    void two(int i, double d) {
```

```
    }

    void three(int i, double d, String s) {
    }

    void four(int i, double d, String s, char c) {
    }
}

interface TwoArgs {
    void call2(This athis, int i, double d);
}

interface ThreeArgs {
    void call3(This athis, int i, double d, String s);
}

interface FourArgs {
    void call4(This athis, int i, double d, String s, char c);
}

public class MultiUnbound {

    public static void main(String[] args) {
        TwoArgs twoargs = This::two;
        ThreeArgs threeargs = This::three;
        FourArgs fourargs = This::four;

        This athis = new This();
        twoargs.call2(athis, 11, 3.14);
        threeargs.call3(athis, 11, 3.14, "Three");
        fourargs.call4(athis, 11, 3.14, "Four", 'Z');
    }
}
```

构造函数引用

```
class Dog {
    String name;
    int age = -1; // For "unknown"

    Dog() {
        name = "stray";
    }

    Dog(String nm) {
        name = nm;
    }

    Dog(String nm, int yrs) {
        name = nm;
        age = yrs;
    }
}

interface MakeNoArgs {
```

```
    Dog make();
}

interface Make1Arg {
    Dog make(String nm);
}

interface Make2Args {
    Dog make(String nm, int age);
}

public class CtorReference {

    public static void main(String[] args) {
       MakeNoArgs makeNoArgs = Dog::new;
       Make1Arg make1Arg = Dog::new;
       Make2Args make2Args = Dog::new;

       Dog noArgs = makeNoArgs.make();
       Dog args1 = make1Arg.make("ha");
       Dog args2= make2Args.make("11",2);

    }
}
```

函数式接口

- @FunctionalInterface 函数式接口：限制接口中只能存在一个抽象方法
- java 8黑魔法：**自动适配函数式接口**（适配你的值到目标接口），编译器会在后台把方法引用或lambda表达式包装进实现目标接口的类中

```
@FunctionalInterface
interface Functional {
    String goodbye(String arg);
}

interface FunctionalNoAnn {
    String goodbye(String arg);
}

// @FunctionalInterface 作用：接口中如果有多个抽象方法则会产生编译期错误
/*
@FunctionalInterface
interface NotFunctional {
  String goodbye(String arg);
  String hello(String arg);
}
Produces error message:
NotFunctional is not a functional interface
multiple non-overriding abstract methods
found in interface NotFunctional
*/

public class FunctionalAnnotation {
```

```
    public String goodbye(String arg) {
        return "Goodbye, " + arg;
    }

    public static void main(String[] args) {

        FunctionalAnnotation fa = new FunctionalAnnotation();
// TODO: 2021/8/30 java 8黑魔法：自动适配函数式接口（适配你的值到目标接口），编译器会在后台把方法引用或lambda表达式包装进实现目标接口的类中
        Functional f = fa::goodbye;
        FunctionalNoAnn fna = fa::goodbye;

        // Functional fac = fa; // Incompatible
        Functional fl = a -> "Goodbye, " + a;
        FunctionalNoAnn fnal = a -> "Goodbye, " + a;
    }
}
```

函数式接口命名准则

1. 如果只处理对象而非基本类型，名称则为 Function，Consumer，Predicate 等。参数类型通过泛型添加
2. 如果接收的参数是基本类型，则由名称的第一部分表示，如 LongConsumer，DoubleFunction，IntPredicate 等，但返回基本类型的 Supplier 接口例外。
3. 如果返回值为基本类型，则用 To 表示，如 ToLongFunction 和 IntToLongFunction。
4. 如果返回值类型与参数类型一致，则是一个运算符：单个参数使用 UnaryOperator，两个参数使用 BinaryOperator。
5. 如果接收两个参数且返回值为布尔值，则是一个谓词（Predicate）。
6. 如果接收的两个参数类型不同，则名称中有一个 Bi。

特征	函数式方法名	示例
无参数；无返回值	Runnable (java.lang) run()	Runnable
无参数；返回类型任意	Supplier get() getAs 类型 ()	Supplier BooleanSupplier IntSupplier LongSupplier DoubleSupplier
无参数；返回类型任意	Callable(java.util.concurrent)call()	Callable
1 参数；无返回值	Consumer accept()	Consumer IntConsumer LongConsumer DoubleConsumer
2 参数 Consumer	BiConsumer accept()	BiConsumer<T,U>
2 参数 Consumer；1 引用；1 基本类型	Obj 类型 Consumer accept()	ObjIntConsumer ObjLongConsumer ObjDoubleConsumer
1 参数；返回类型不同	Function apply() To 类型和 类型 To 类型<applyAs 类型 ()	Function<T,R> IntFunction LongFunction DoubleFunction ToIntFunction ToLongFunction ToDoubleFunction IntToLongFunction IntToDoubleFunction LongToIntFunction LongToDoubleFunction DoubleToIntFunction DoubleToLongFunction
1 参数；返回类型相同	UnaryOperator apply()	UnaryOperator IntUnaryOperator LongUnaryOperator DoubleUnaryOperator
2 参数类型相同；返回类型相同	BinaryOperator apply()	BinaryOperator IntBinaryOperator LongBinaryOperator DoubleBinaryOperator
2 参数类型相同; 返回整型	Comparator (java.util)compare()	Comparator
2 参数；返回布尔型	Predicate test()	Predicate BiPredicate<T,U> IntPredicate LongPredicate DoublePredicate

特征	函数式方法名	示例
参数基本类型；返回基本类型	类型 To 类型 Function applyAs 类型 ()	IntToLongFunction IntToDoubleFunction LongToIntFunction LongToDoubleFunction DoubleToIntFunction DoubleToLongFunction
2 参数类型不同	Bi 操作 (不同方法名)	BiFunction<T,U,R> BiConsumer<T,U> BiPredicate<T,U> ToIntBiFunction<T,U> ToLongBiFunction<T,U> ToDoubleBiFunction

使用函数式接口时，名称无关紧要，只要参数类型和返回类型相同即可，Java会将你的方法映射到接口方法。

等同 final 效果

```
// 如果局部变量的初始值永远不会改变，那么它实际上就是 final 的。
class Closure2 {
    // TODO: 2021/8/30  x等同 final 效果
    IntSupplier makeFun(int x) {
        int i = 0;
        return () -> x + i;
    }
}

class Closure3 {
    IntSupplier makeFun(int x) {
        int i = 0;
        // TODO: 2021/8/30 被 Lambda 表达式引用的局部变量必须是 final 或者是等同 final
效果的
        //return () -> x++ + i++; error
        return () -> x;
    }
}

class Closure4 {
    IntSupplier makeFun(final int x) {
        final int i = 0;
        return () -> x + i;
    }
}
```

函数组合

组合方法	支持接口
andThen(argument) 根据参数执行原始操作	Function BiFunction Consumer BiConsumer IntConsumer LongConsumer DoubleConsumer UnaryOperator IntUnaryOperator LongUnaryOperator DoubleUnaryOperator BinaryOperator
compose(argument) 根据参数执行原始操作	Function UnaryOperator IntUnaryOperator LongUnaryOperator DoubleUnaryOperator
and(argument) 短路逻辑与原始谓词和参数谓词	Predicate BiPredicate IntPredicate LongPredicate DoublePredicate
or(argument) 短路逻辑或原始谓词和参数谓词	Predicate BiPredicate IntPredicate LongPredicate DoublePredicate
negate() 该谓词的逻辑否谓词	Predicate BiPredicate IntPredicate LongPredicate DoublePredicate

```
public class FunctionComposition {

    // TODO: 2021/8/30 函数组合
    static Function<String, String> f1 = s -> {
        System.out.println(s);
        return s.replace('A', '_'); },
            f2 = s -> s.substring(3),
            f3 = s -> s.toLowerCase(),
            f4 = f1.compose(f2).andThen(f3);

    public static void main(String[] args) {
        System.out.println(
                f4.apply("GO AFTER ALL AMBULANCES"));
    }
}
/* Output:
AFTER ALL AMBULANCES
_fter _ll _mbul_nces
*/
```

```
public class PredicateComposition {
    // TODO: 2021/8/30 短路 与 或 非
    static Predicate<String>
            p1 = s -> s.contains("bar"),
            p2 = s -> s.length() < 5,
            p3 = s -> s.contains("foo"),
            p4 = p1.negate().and(p2).or(p3);

    public static void main(String[] args) {
        Stream.of("bar", "foobar", "foobaz", "fongopuckey")
```

```
                .filter(p4)
                .forEach(System.out::println);
    }
}
```

流式编程

开胃菜

```
// TODO: 2021/8/31 流式编程与普通编程对比
class ImperativeRandoms {
    // TODO: 2021/8/31 外部迭代
    public static void main(String[] args) {
        Random rand = new Random(47);
        SortedSet<Integer> rints = new TreeSet<>();
        while (rints.size() < 7) {
            int r = rand.nextInt(20);
            if (r < 5) {
                continue;
            }
            rints.add(r);
        }
        System.out.println(rints);
    }
}

// TODO: 2021/8/31 使用流的好处：内部迭代产生的代码可读性更强，而且能更简单的的使用多核处理器，
//  通过放弃对迭代的控制，可以把控制权交给并行化机制。
//  流是懒加载的，
public class Randoms {
    // TODO: 2021/8/31 内部迭代
    public static void main(String[] args) {
        // seed（种子）作用，再次运行产生相同的输出
        new Random(47)
                .ints(5, 20) // 产生一个整数流在两个区间内
                .distinct()  // 去重
                .limit(7)  // 保留个数
                .sorted()  // 升序（默认）逆序Comparator.reverseOrder()
                .forEach(System.out::println); // 方法引用


    }
}
/* Output:
6
10
13
16
17
18
19
*/
```

内部迭代产生的代码可读性更强，而且能更简单的使用多核处理器。通过放弃对迭代流程的控制，可以把控制权交给并行化机制。

流是懒加载的

创建流

```
public class StreamOf {
    // TODO: 2021/8/31 创建流、遍历流
    public static void main(String[] args) {
        // createStream1();
        collectionToStream();
    }

    private static void collectionToStream() {
        List<Bubble> bubbles = Arrays.asList(
                new Bubble(1), new Bubble(2), new Bubble(3));
        // 所有集合都有的stream () , map()获取流中所有的元素，并进行深克隆
        System.out.println(
                // TODO: 2021/8/31 stream()
                bubbles.stream()
                        // 将一个对象流转换成为包含整形数字的IntStream
                        .mapToInt(b -> b.i)
                        .sum());

        Set<String> w = new HashSet<>(Arrays.asList(
                "It's a wonderful day for pie!".split(" ")));
        // strean() 产生一个流，map()获取流中所有的元素，并且对流中元素应用操作从而产生新的元素，并将其传递到后续的流中。
        w.stream()
                .map(x -> x + ",")
                .forEach(System.out::print);
        System.out.println();

        Map<String, Double> m = new HashMap<>();
        m.put("pi", 3.14159);
        m.put("e", 2.718);
        m.put("phi", 1.618);
        // 首先调用 entrySet()产生一个对象流，通过 map()获取所有的对象
        m.entrySet().stream()
                .map(e -> e.getKey() + ": " + e.getValue())
                .forEach(System.out::println);
    }

    private static void createStream1() {
        // TODO: 2021/8/31 Stream.of()
        Stream.of(
                new Bubble(1), new Bubble(2), new Bubble(3))
                .forEach(System.out::println);

        Stream.of("It's ", "a ", "wonderful ",
                "day ", "for ", "pie!")
                .forEach(System.out::print);

        System.out.println();

        Stream.of(3.14159, 2.718, 1.618)
```

```
                .forEach(System.out::println);
    }


}
/* Output:
Bubble(1)
Bubble(2)
Bubble(3)
It's a wonderful day for pie!
3.14159
2.718
1.618
*/
```

随机数

```
public class RandInts {

    // TODO: 2021/8/31 toArray() 将流转换成适当类型的数组
    private static int[] rints = new Random(47)
            .ints(0, 1000)
            .limit(100)
            .toArray();

    /**
     * 将 100 个数值范围在 0 到 1000 之间的随机数流转换成为数组并将其存储在
     * rints 中。这样一来，每次调用 rands() 的时候可以重复获取相同的整数流。
     */
    public static IntStream rands() {
        return Arrays.stream(rints);
    }
}
```

常用方法

stream()

- 将集合转换成一个流

ints(5, 20)

- 产生一个整数流在两个区间内

distinct()

- 去重

limit(7)

- 保留7个元素

sorted()

- 升序（默认）
- 逆序 Comparator.reverseOrder()

boxed()

- 将基本类型装箱　　.

Stream.generate()

- 把任意Supplier 用于生成T类型的流

collect()

- 根据参数来结合所有的流元素
- 对参数进行分割
    - collect(Collectors.joining(", ")) 使用逗号分割

sum()

- 求和

```
// TODO: 2021/8/31 使用流读取文件
public class RandomWords implements Supplier<String> {

    List<String> words = new ArrayList<>();
    Random rand = new Random(47);

    RandomWords(String fname) throws IOException {
        List<String> lines =
                Files.readAllLines(Paths.get(fname));
        // Skip the first line:
        for (String line : lines.subList(1, lines.size())) {
            // 使用空格 点 逗号 问号 分隔 +表示出现一次或者多次
            for (String word : line.split("[ .?,]+")) {
                words.add(word.toLowerCase());
            }
        }
    }

    @Override
    public String get() {
        return words.get(rand.nextInt(words.size()));
    }

    @Override
    public String toString() {

        return words.stream()
                .collect(Collectors.joining(" "));
    }

    public static void main(String[] args) throws Exception {
        System.out.println(
                // TODO: 2021/8/31 Stream.generate(), 把任意Supplier<T> 用于生成 T
类型的流
```

```
                Stream.generate(new
RandomWords("src/main/resources/Cheese.dat"))
                        .limit(10)
                        // TODO: 2021/8/31 collect() 根据参数来结合所有的流元素，使用
逗号分割
                        .collect(Collectors.joining(", ")));
    }
}
/* Output:
it shop sir the much cheese by conclusion district is
*/
```

IntStream.range()

- 左闭右开区间
- 生成区间内的整型序列流
- 可以替换简单的for循环
    - ```
      import static java.util.stream.IntStream.*;
      range(0, 5).forEach(System.out::println);
      ```

```
public class Ranges {
    // TODO: 2021/8/31 range()流操作大用处，转换成数组，计算区间总和
    public static void main(String[] args) {

        // The traditional way:
        int result = 0;
        for (int i = 10; i < 20; i++) {
            result += i;
        }
        System.out.println(result);
        // range(a,b) 左闭右开区间[a,b)
        int[] ints = range(10, 20).toArray();
        System.out.println(Arrays.toString(ints));
        // for-in with a range:
        result = 0;
        for (int i : range(10, 20).toArray()) {
            result += i;
        }
        System.out.println(result);

        // Use streams:
        System.out.println(range(10, 20).sum());
    }
}
/* Output:
145
145
145
*/
```
```

Stream.iterate()

- 产生的流的第一个元素是种子，然后将种子传递给方法（iterate 方法的第二个参数）
- 方法运行的结果被添加到流，作为流的第二个元素，并存储起来作为下次调用 iterate()时的第一个参数，以此类推

skip()

- 跳过的数据个数

```
public class Fibonacci {
    int x = 1;

    // TODO: 2021/8/31 Stream.iterate()
    Stream<Integer> numbers() {
        return Stream.iterate(0, i -> {
            int result = x + i;
            x = i;
            return result;
        });
    }

    public static void main(String[] args) {
        new Fibonacci().numbers()
                .skip(20) // Don't use the first 20
                .limit(10) // Then take 10 of them
                .forEach(System.out::println);
    }
}
/* Output:
6765
10946
17711
28657
46368
75025
121393
196418
317811
514229
*/
```

Arrays.stream()

- 把数组转换成为流

```
public class ArrayStreams {
    // TODO: 2021/8/31 Arrays.stream()将数组转换成流
    public static void main(String[] args) {
        Arrays.stream(
                new double[]{3.14159, 2.718, 1.618})
                .forEach(n -> System.out.format("%f ", n));
        System.out.println();

        Arrays.stream(new int[]{1, 3, 5})
                .forEach(n -> System.out.format("%d ", n));
```

```
        System.out.println();

        Arrays.stream(new long[]{11, 22, 44, 66})
                .forEach(n -> System.out.format("%d ", n));
        System.out.println();

        // 第二个参数是从数组哪个位置开始选择元素，第三个参数是在哪里停止，左闭右开区间
        Arrays.stream(
                new int[]{1, 3, 5, 7, 15, 28, 37}, 3, 6)
                .forEach(n -> System.out.format("%d ", n));
    }
}
/* Output:
3.141590 2.718000 1.618000
1 3 5
11 22 44 66
7 15 28
*/
```

正则表达式

splitAsStream()

- Pattern.compile("[.,?]+").splitAsStream(all);

```
// TODO: 2021/8/31 使用流将文件转换为一个字符串，接着使用正则表达式将字符串转化为单词流
public class FileToWordsRegexp {
    private String all;

    public FileToWordsRegexp(String filePath) throws Exception {
        all = Files.lines(Paths.get(filePath))
                .skip(1) // First (comment) line
                .collect(Collectors.joining(" "));
    }

    public Stream<String> stream() {
        return Pattern.compile("[ .,?]+").splitAsStream(all);
    }

    public static void main(String[] args) throws Exception {

        FileToWordsRegexp fw = new
FileToWordsRegexp("src/main/resources/Cheese.dat");
        fw.stream()
                .limit(7)
                .map(w -> w + " ")
                .forEach(System.out::print);
      System.out.println();
        // output Not much of a cheese shop really
        fw.stream()
                .skip(7)
                .limit(2)
                .map(w -> w + " ")
                .forEach(System.out::print);
        // output Not much of a cheese shop really is it
    }
```

```
}
/* Output:
Not much of a cheese shop really is it
*/
```

peek()

- 无修改的查看流中元素

```
class Peeking {
    // TODO: 2021/8/31 peek() 瞟一眼 中间操作
    public static void main(String[] args) throws Exception {
        new FileToWordsRegexp("src/main/resources/Cheese.dat").stream()
                .skip(21)
                .limit(4)
                .map(w -> w + " ")
                .peek(System.out::print)
                .map(String::toUpperCase)
                .peek(System.out::print)
                .map(String::toLowerCase)
                .forEach(System.out::print);
    }
}
/* Output:
Well WELL well it IT it s S s so SO so
*/
```

filter()

- 若元素传递给过滤函数产生的结果为true，则过滤操作保留这些元素

rangeClosed()

- rangeClosed() 包含了上限值 左闭右闭区间

```
import java.util.stream.*;
import static java.util.stream.LongStream.*;
public class Prime {

    /**
     * 过滤函数，用于检测质数
     */
    public static boolean isPrime(long n) {
        // TODO: 2021/8/31 rangeClosed() 包含了上限值，
        return rangeClosed(2, (long) Math.sqrt(n))
                .noneMatch(i -> n % i == 0);
        // 如果不能整除，即余数不等于 0，则 noneMatch()
        //操作返回 true，如果出现任何等于 0 的结果则返回 false。noneMatch() 操作一旦有
        //失败就会退出。
    }

    public LongStream numbers() {
        // seed 初始元素，每次加1操作
        return iterate(2, i -> i + 1)
                .filter(Prime::isPrime);
    }
```

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

        new Prime().numbers()
                .limit(10)
                .forEach(n -> System.out.format("%d ", n));
        System.out.println();

        new Prime().numbers()
                .skip(90)
                .limit(10)
                .forEach(n -> System.out.format("%d ", n));
    }
}
/* Output:
2 3 5 7 11 13 17 19 23 29
467 479 487 491 499 503 509 521 523 541
*/
```

map(Function)

- 将函数操作应用在输入流的元素中，并将返回值传递到输出流中
- 获取对象并产生新的对象

mapToInt(ToIntFunction)

- 将一个对象流（object stream）转换成为包含整形数字的 IntStream

mapToLong(ToLongFunction):

- 操作同上，但结果是 LongStream。

mapToDouble(ToDoubleFunction):

- 操作同上，但结果是 DoubleStream。

```
class FunctionMap {
    static String[] elements = {"12", "", "23", "45"};

    /**
     * @return 将数组转换成流返回
     */
    static Stream<String> testStream() {
        return Arrays.stream(elements);
    }

    // TODO: 2021/8/31 map() 的使用方法
    static void test(String descr, Function<String, String> func) {
        System.out.println(" ---( " + descr + " )---");
        testStream()
                .map(func)
                .forEach(System.out::println);
    }

    public static void main(String[] args) {
        // s 表示的是每个集合中的元素
        test("add brackets", s -> "[" + s + "]");
```

```
        test("Increment", s -> {
            try {
                return Integer.parseInt(s) + 1 + "";
            } catch (NumberFormatException e) {
                return s;
            }
        });

        test("Replace", s -> s.replace("2", "9"));

        test("Take last digit", s -> s.length() > 0 ?
                s.charAt(s.length() - 1) + "" : s);
    }
}
/* Output:
 ---( add brackets )---
[12]
[]
[23]
[45]
 ---( Increment )---
13

24
46
 ---( Replace )---
19

93
45
 ---( Take last digit )---
2

3
5
*/
```

```
class FunctionMap3 {
    // TODO: 2021/8/31 使用 mapToInt 将流进行转化
    public static void main(String[] args) {
        Stream.of("5", "7", "9")
                .mapToInt(Integer::parseInt)
                .forEach(n -> System.out.format("%d ", n));
        System.out.println();

        Stream.of("17", "19", "23")
                .mapToLong(Long::parseLong)
                .forEach(n -> System.out.format("%d ", n));
        System.out.println();

        Stream.of("17", "1.9", ".23")
                .mapToDouble(Double::parseDouble)
                .forEach(n -> System.out.format("%f ", n));
    }
}
```

```
/* Output:
5 7 9
17 19 23
17.000000 1.900000 0.230000
*/
```

flatMap(Function)

- 将产生流的函数应用在每个元素上（与 map()，所做的相同），
- 然后将每个流都扁平化为元素，因而最终产生的仅仅是元素

flatMapToInt(Function):

- 当 Function 产生 IntStream 时使用。

flatMapToLong(Function):

- 当 Function 产生 LongStream 时使用。

flatMapToDouble(Function):

- 当 Function 产生 DoubleStream 时使用。

```
public class FlatMap {
    public static void main(String[] args) {
        Stream.of(1, 2, 3)
        .flatMap(i -> Stream.of("Gonzo", "Fozzie", "Beaker"))
        .forEach(System.out::println);
    }
}
output:
Gonzo
Fozzie
Beaker
Gonzo
Fozzie
Beaker
Gonzo
Fozzie
Beaker
```

Stream.of(1，2，3)

- 每次执行几次任务
- 第一次执行1次，第二次执行2次，第三次执行3次

concat()

- 以参数顺序组合两个流

使用整数流创建随机数

```
public class StreamOfRandoms {
    static Random rand = new Random(47);

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

```
        // Stream.of()每次执行几次任务
        Stream.of(1, 2, 3, 4, 5)
                .flatMapToInt(i -> IntStream.concat(
                        rand.ints(0, 100).limit(i), IntStream.of(-1)))
                .forEach(n -> System.out.format("%d ", n));
    }
}
/* Output:
58 -1 55 93 -1 61 61 29 -1 68 0 22 7 -1 88 28 51 89 9
-1
*/
```

Optional

findFirst()

- 返回一个包含第一个元素的 Optional 对象，如果流为空则返回Optional.empty

findAny()

- 返回包含任意元素的 Optional 对象，如果流为空则返回 Op-tional.empty

max() 和 min()

- 返回一个包含最大值或者最小值的 Optional 对象，如果流为空则返回 Optional.empty

> 当流为空的时候你会获得一个 Optional.empty 对象，而不是抛出异常

便利函数

ifPresent(Consumer)

- 当值存在时调用 Consumer，否则什么也不做。

orElse(otherObject)

- 如果值存在则直接返回，否则生成 otherObject。

orElseGet(Supplier)

- 如果值存在则直接返回，否则使用 Supplier 函数生成一个可替代对象。

orElseThrow(Supplier)

- 如果值存在直接返回，否则使用 Supplier 函数生成一个异常。

```
public class Optionals {

    static void basics(Optional<String> optString) {
        if (optString.isPresent()) {
            System.out.println(optString.get());
        } else {
            System.out.println("Nothing inside!");
        }
    }

    // TODO: 2021/8/31 ifPresent 当值存在时调用 consumer，否则什么也不做
    static void ifPresent(Optional<String> optString) {
```

```
        optString.ifPresent(System.out::println);
    }

    // TODO: 2021/8/31 orElse 如果值存在直接返回，否则生成otherObject
    static void orElse(Optional<String> optString) {
        System.out.println(optString.orElse("Nada"));
    }

    // TODO: 2021/8/31  orElseGet 如果值存在直接返回，否则使用 Supplier生成一个可替代对
象
    static void orElseGet(Optional<String> optString) {
        System.out.println(
                optString.orElseGet(() -> "Generated"));
    }

    // TODO: 2021/8/31  orElseThrow 如果值存在直接返回，否则使用 Supplier生成一个异常
    static void orElseThrow(Optional<String> optString) {
        try {
            System.out.println(optString.orElseThrow(
                    () -> new Exception("Supplied")));
        } catch (Exception e) {
            System.out.println("Caught " + e);
        }
    }

    static void test(String testName, Consumer<Optional<String>> cos) {
        System.out.println(" === " + testName + " === ");
        cos.accept(Stream.of("Epithets").findFirst());
        cos.accept(Stream.<String>empty().findFirst());
    }

    public static void main(String[] args) {
        test("basics", Optionals::basics);
        test("ifPresent", Optionals::ifPresent);
        test("orElse", Optionals::orElse);
        test("orElseGet", Optionals::orElseGet);
        test("orElseThrow", Optionals::orElseThrow);
    }
}
/* Output:
 === basics ===
Epithets
Nothing inside!
 === ifPresent ===
Epithets
 === orElse ===
Epithets
Nada
 === orElseGet ===
Epithets
Generated
 === orElseThrow ===
Epithets
Caught java.lang.Exception: Supplied
*/
```

创建Optional

empty()

- 生成一个空 Optional。

of(value)

- 将一个非空值包装到 Optional 里。

ofNullable(value)

- 针对一个可能为空的值，为空时自动生成 Optional.empty，否则将值包装在 Optional 中。

```
class CreatingOptionals {

    static void test(String testName, Optional<String> opt) {
        System.out.println(" === " + testName + " === ");
        System.out.println(opt.orElse("Null"));
    }

    public static void main(String[] args) {
        // TODO: 2021/8/31 empty() 生成一个空 Optional
        test("empty", Optional.empty());

        // TODO: 2021/8/31 Optional.of 将一个非空值包装到 Optional里
        test("of", Optional.of("Howdy"));
        System.out.println();
        try {
            test("of", Optional.of(null));
        } catch (Exception e) {
            System.out.println(e);
        }

        // TODO: 2021/8/31 对于一个可能为空的值，为空时自动生成Optional.empty()，否则将
值包装在Optional中
        test("ofNullable", Optional.ofNullable("Hi"));
        test("ofNullable", Optional.ofNullable(null));
    }
}
/* Output:
 === empty ===
Null
 === of ===
Howdy
java.lang.NullPointerException
 === ofNullable ===
Hi
 === ofNullable ===
Null
*/
```

filter(Predicate):

- 对 Optional 中的内容应用 Predicate 并将结果返回。如果Optional 不满足 Predicate ，将 Optional 转化为空 Optional 。如果 Optional已经为空，则直接返回空 Optional 。

```
// TODO: 2021/8/31 使用流 进行过滤数组，不满足条件的为空
class OptionalFilter {

    static String[] elements = {
            "Foo", "", "Bar", "Baz", "Bingo"
    };

    static Stream<String> testStream() {
        return Arrays.stream(elements);
    }

    static void test(String descr, Predicate<String> pred) {
        System.out.println(" ---( " + descr + " )---");
        for (int i = 0; i <= elements.length; i++) {
            System.out.println(
                    testStream()
                            //执行完就跳过这个循环，下次不执行
                            .skip(i)
                            .findFirst()
                            .filter(pred));
        }
    }

    public static void main(String[] args) {
        test("true", str -> true);
        test("false", str -> false);
        test("str != \"\"", str -> str != "");
        test("str.length() == 3", str -> str.length() == 3);
        test("startsWith(\"B\")",
                str -> str.startsWith("B"));
    }
}
/* Output:
 ---( true )---
Optional[Foo]
Optional[]
Optional[Bar]
Optional[Baz]
Optional[Bingo]
Optional.empty
 ---( false )---
Optional.empty
Optional.empty
Optional.empty
Optional.empty
Optional.empty
Optional.empty
 ---( str != "" )---
Optional[Foo]
Optional.empty
Optional[Bar]
Optional[Baz]
```

```
Optional[Bingo]
Optional.empty
 ---( str.length() == 3 )---
Optional[Foo]
Optional.empty
Optional[Bar]
Optional[Baz]
Optional.empty
Optional.empty
 ---( startsWith("B") )---
Optional.empty
Optional.empty
Optional[Bar]
Optional[Baz]
Optional[Bingo]
Optional.empty
*/
```

map(Function):

- 如果 Optional 不为空，应用 Function 于 Optional 中的内容，并返回结果。否则直接返回 Optional.empty。

> 同 map() 一样，Optional.map() 执行一个函数。它仅在 Optional 不为空时才执行这个映射函数。并将 Optional 的内容提取出来，传递给映射函数。

```
class OptionalMap {
    static String[] elements = {"12", "", "23", "45"};

    static Stream<String> testStream() {
        return Arrays.stream(elements);
    }

    // TODO: 2021/8/31 Optional.map() 执行一个函数
    static void test(String descr, Function<String, String> func) {
        System.out.println(" ---( " + descr + " )---");
        for (int i = 0; i <= elements.length; i++) {
            System.out.println(
                    testStream()
                            .skip(i)
                            .findFirst() // Produces an Optional
                            .map(func));
        }
    }

    public static void main(String[] args) {

        // If Optional is not empty, map() first extracts
        // the contents which it then passes
        // to the function:

        test("Add brackets", s -> "[" + s + "]");

        test("Increment", s -> {
```

```
            try {
                return Integer.parseInt(s) + 1 + "";
            } catch (NumberFormatException e) {
                return s;
            }
        });

        test("Replace", s -> s.replace("2", "9"));

        test("Take last digit", s -> s.length() > 0 ?
                s.charAt(s.length() - 1) + "" : s);
    }
    // After the function is finished, map() wraps the
    // result in an Optional before returning it:
}
/* Output:
 ---( Add brackets )---
Optional[[12]]
Optional[[]]
Optional[[23]]
Optional[[45]]
Optional.empty
 ---( Increment )---
Optional[13]
Optional[]
Optional[24]
Optional[46]
Optional.empty
 ---( Replace )---
Optional[19]
Optional[]
Optional[93]
Optional[45]
Optional.empty
 ---( Take last digit )---
Optional[2]
Optional[]
Optional[3]
Optional[5]
Optional.empty
*/
```

flatMap(Function):

- 同 map()，但是提供的映射函数将结果包装在 Optional 对象中，因此 flatMap() 不会在最后进行任何包装。

```
// TODO: 2021/8/31 flatmap() 将提取非空 Optional的内容并将其应用在映射函数,
//  flatMap()不会把结果包装在 Optional中，因为映射函数已经被包装过
//  Optional.flatMap() 是为那些自己已经生成Optiional 的函数设计的
class OptionalFlatMap {
    static String[] elements = {"12", "", "23", "45"};

    static Stream<String> testStream() {
        return Arrays.stream(elements);
    }
```

```
    static void test(String descr, Function<String, Optional<String>> func) {
        System.out.println(" ---( " + descr + " )---");
        for (int i = 0; i <= elements.length; i++) {
            System.out.println(
                    testStream()
                            .skip(i)
                            .findFirst()
                            .flatMap(func));
        }
    }

    public static void main(String[] args) {

        test("Add brackets", s -> Optional.of("[" + s + "]"));

        test("Increment", s -> {
            try {
                return Optional.of(
                        Integer.parseInt(s) + 1 + "");
            } catch (NumberFormatException e) {
                return Optional.of(s);
            }
        });

        test("Replace", s -> Optional.of(s.replace("2", "9")));

        test("Take last digit", s -> Optional.of(s.length() > 0 ?
                        s.charAt(s.length() - 1) + "" : s));
    }
}
/* Output:
 ---( Add brackets )---
Optional[[12]]
Optional[[]]
Optional[[23]]
Optional[[45]]
Optional.empty
 ---( Increment )---
Optional[13]
Optional[]
Optional[24]
Optional[46]
Optional.empty
 ---( Replace )---
Optional[19]
Optional[]
Optional[93]
Optional[45]
Optional.empty
 ---( Take last digit )---
Optional[2]
Optional[]
Optional[3]
Optional[5]
Optional.empty
*/
```

Optional 流

```
public class StreamOfOptionals {
    public static void main(String[] args) {
        Signal.stream()
                .limit(10)
                .forEach(System.out::println);

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

        Signal.stream()
                .limit(10)
                // 过滤掉 Optional.empty()
                .filter(Optional::isPresent)
                // 使用get 获取元素
                .map(Optional::get)
                .forEach(System.out::println);
    }
}
/* Output:
Optional[Signal(dash)]
Optional[Signal(dot)]
Optional[Signal(dash)]
Optional.empty
Optional.empty
Optional[Signal(dash)]
Optional.empty
Optional[Signal(dot)]
Optional[Signal(dash)]
Optional[Signal(dash)]
 ---
Signal(dot)
Signal(dot)
Signal(dash)
Signal(dash)
*/
```

终端操作

数组

toArray():

- 将流转换成适当类型的数组。

toArray(generator):

- 在特殊情况下，生成自定义类型的数组。

```
public class RandInts {

    // TODO: 2021/8/31 toArray() 将流转换成适当类型的数组
    private static int[] rints = new Random(47)
            .ints(0, 1000)
            .limit(100)
            .toArray();
```

```
    /**
     * 将 100 个数值范围在 0 到 1000 之间的随机数流转换成为数组并将其存储在
     * rints 中。这样一来，每次调用 rands() 的时候可以重复获取相同的整数流。
     */
    public static IntStream rands() {
        return Arrays.stream(rints);
    }
}
```

循环

forEach(Consumer)

- 常见如 System.out::println 作为 Consumer 函数。
- 无序操作

forEachOrdered(Consumer)

- 保证 forEach 按照原始流顺序操作。

parallel()

- 实现多处理器并行操作
- 实现原理：将流分割为多个（通常为CPU核心数）并在不同处理器上分别执行操作

```
public class ForEach {
    static final int SZ = 14;

    public static void main(String[] args) {
        RandInts.rands().limit(SZ)
                .forEach(n -> System.out.format("%d ", n));
        System.out.println();

        // 即使流很小，输出的结果顺序也和前面的不一样，这是因为处理器并行处理的原因
        RandInts.rands().limit(SZ)
                .parallel()
                .forEach(n -> System.out.format("%d ", n));
        System.out.println();
        // TODO: 2021/8/31 在使用 parallel()并行计算后，对于并行流必须使用
forEachOrdered()保证流的顺序性
        RandInts.rands().limit(SZ)
                .parallel()
                .forEachOrdered(n -> System.out.format("%d ", n));
    }
}
/* Output:
258 555 693 861 961 429 868 200 522 207 288 128 551 589
551 589 861 555 288 128 429 207 693 200 258 522 868 961
258 555 693 861 961 429 868 200 522 207 288 128 551 589
*/
```

集合

collect(Collector)

- 使用 Collector 收集流元素到结果集合中。

collect(Supplier, BiConsumer, BiConsumer):

- 同上，第一个参数 Supplier 创建了一个新的结果集合，第二个参数 BiConsumer 将下一个元素收集到结果集合中，第三个参数 BiConsumer 用于将两个结果集合合并起来。

```
// TODO: 2021/8/31 将文件中的单词收集到 TreeSet 中，使用Collections.toCollection()构建任何类型的集合
public class TreeSetOfWords {

    public static void main(String[] args) throws Exception {
        Set<String> words2 =

 Files.lines(Paths.get("src/main/java/stream/readfilesforwords/TreeSetOfWords.java"))
                        .flatMap(s -> Arrays.stream(s.split("\\W+")))
                        .filter(s -> !s.matches("\\d+")) // No numbers 移除数字
                        .map(String::trim)
                        // 过滤单词数小于3的people
                        .filter(s -> s.length() > 2)
                        .limit(100)
                        .collect(Collectors.toCollection(TreeSet::new));
        System.out.println(words2);
    }
}
/* Output:
[Arrays, Collectors, Exception, Files, Output, Paths,
Set, String, System, TreeSet, TreeSetOfWords, args,
class, collect, file, filter, flatMap, get, import,
java, length, limit, lines, main, map, matches, new,
nio, numbers, out, println, public, split, static,
stream, streams, throws, toCollection, trim, util,
void, words2]
*/
```

在流中生成map

```
class Pair {
    public final Character c;
    public final Integer i;

    Pair(Character c, Integer i) {
        this.c = c;
        this.i = i;
    }

    public Character getC() {
        return c;
    }

    public Integer getI() {
```

```
        return i;
    }

    @Override
    public String toString() {
        return "Pair(" + c + ", " + i + ")";
    }
}

class RandomPair {
    Random rand = new Random(47);
    // TODO: 2021/8/31  随机大写字母的无限迭代器:
    Iterator<Character> capChars = rand.ints(65, 91)
            .mapToObj(i -> (char) i)
            .iterator();

    public Stream<Pair> stream() {
        return rand.ints(100, 1000).distinct()
                .mapToObj(i -> new Pair(capChars.next(), i));
    }
}

public class MapCollector {
    public static void main(String[] args) {
        // TODO: 2021/8/31 生成随机map 使用Collections.toMap，从两个流中获取键和值的函
数
        Map<Integer, Character> map = new RandomPair()
                .stream()
                .limit(8)
                .collect(Collectors.toMap(Pair::getI, Pair::getC));
        System.out.println(map);
    }
}
/* Output:
{688=W, 309=C, 293=B, 761=N, 858=N, 668=G, 622=F,
751=N}
*/
```

组合

reduce(BinaryOperator):

- 使用 BinaryOperator 来组合所有流中的元素。因为流可能为空，其返回值为 Optional。

reduce(identity, BinaryOperator):

- 功能同上，但是使用 identity 作为其组合的初始值。因此如果流为空，identity 就是结果。

reduce(identity, BiFunction, BinaryOperator):

- 更复杂的使用形式（暂不介绍），这里把它包含在内，因为它可以提高效率。通常，我们可以显式地组合 map()和 reduce() 来更简单的表达它

```
class Frobnitz {
    int size;

    Frobnitz(int sz) {
```

```
        size = sz;
    }

    @Override
    public String toString() {
        return "Frobnitz(" + size + ")";
    }

    // Generator:
    static Random rand = new Random(47);
    static final int BOUND = 100;

    static Frobnitz supply() {
        return new Frobnitz(rand.nextInt(BOUND));
    }
}

public class Reduce {
    // TODO: 2021/9/1 reduce
    public static void main(String[] args) {
        // 创建流 create stream
        Stream.generate(Frobnitz::supply)
                .limit(100)
                .peek(System.out::println)
                // fro 是 reduce 中上一次调用的结果，fr1 是从流传递过来的值
                .reduce((fr0, fr1) -> fr0.size < 50 ? fr0 : fr1)
                .ifPresent(System.out::println);
    }
}
/* Output:
Frobnitz(58)
Frobnitz(55)
Frobnitz(93)
Frobnitz(61)
Frobnitz(61)
Frobnitz(29)
Frobnitz(68)
Frobnitz(0)
Frobnitz(22)
Frobnitz(7)
Frobnitz(29)
*/
```

匹配

allMatch(Predicate)

- 如果流的每个元素提供给 Predicate 都返回 true，结果返回为 true。在第一个 false 时，则停止执行计算。

anyMatch(Predicate)

- 如果流的任意一个元素提供给 Predicate 返回 true，结果返回为 true。在第一个 true 是停止执行计算。

noneMatch(Predicate)

- 如果流的每个元素提供给 Predicate 都返回 false 时，结果返回为 true。在第一个 true 时停止执行计算

```
// TODO: 2021/9/1 BiPredicate 是一个二元谓词，
//  第一个参数是我们要测试的流，第二个参数是谓词Predicate
//  简单的说第一个参数是模式匹配，第二个参数是模式匹配的值
interface Matcher extends BiPredicate<Stream<Integer>, Predicate<Integer>> {
}

// TODO: 2021/9/1 匹配
public class Matching {

    static void show(Matcher match, int val) {
        System.out.println(match.test(
                IntStream.rangeClosed(1, 9)
                        .boxed()
                        .peek(n -> System.out.format("%d ", n)),
                n -> n < val));
    }

    public static void main(String[] args) {
        // TODO: 2021/9/1 allMatch 如果流的每个元素提供给Predicate 都返回true，结果返回为true，在第一个false时，则停止执行计算
        show(Stream::allMatch, 10);

        // TODO: 2021/9/1 anyMatch 如果流的任意一个元素提供给 Predicate 返回true，结果返回为true，在第一个false时，则停止执行计算
        show(Stream::allMatch, 4);
        show(Stream::anyMatch, 2);
        show(Stream::anyMatch, 0);

        // TODO: 2021/9/1 noneMatch 如果流的任意一个元素提供给 Predicate 返回false，结果返回为true，在第一个true时，则停止执行计算
        show(Stream::noneMatch, 5);
        show(Stream::noneMatch, 0);
    }
}
/* Output:
1 2 3 4 5 6 7 8 9 true
1 2 3 4 false
1 true
1 2 3 4 5 6 7 8 9 false
1 false
1 2 3 4 5 6 7 8 9 true
*/
```

查找

findFirst()

- 返回第一个流元素的 Optional，如果流为空返回 Optional.empty。

findAny()

- 返回含有任意流元素的 Optional，如果流为空返回 Optional.empty

```
public class SelectElement {

    public static void main(String[] args) {

        System.out.println(RandInts.rands().findFirst().getAsInt());

        System.out.println(RandInts.rands().parallel().findFirst().getAsInt());

        // TODO: 2021/9/1 对于非并行流，findAny()会选择流中的第一个元素
        System.out.println(RandInts.rands().findAny().getAsInt());

        // TODO: 2021/9/1 对于并行流，findAny()不会选择第一个元素
        System.out.println(RandInts.rands().parallel().findAny().getAsInt());
        System.out.println("=====");
        RandInts.rands().limit(4).forEach(System.out::println);
        System.out.println("=====");

    }
}
output:
258
258
258
242
=====
258
555
693
861
=====

Process finished with exit code 0
```

信息

count():

- 流中的元素个数。

max(Comparator)

- 根据所传入的 Comparator 所决定的 “最大” 元素。

min(Comparator)

- 根据所传入的 Comparator 所决定的 “最小” 元素。

orElse(A)

- 如果存在即返回，否则返回A

```
public class Informational {

    public static void main(String[] args) throws Exception {


 System.out.println(FileToWords.stream("src/main/resources/Cheese.dat").count())
;

        System.out.println(FileToWords.stream("src/main/resources/Cheese.dat")
                        .min(String.CASE_INSENSITIVE_ORDER)
                        .orElse("NONE"));

        System.out.println(FileToWords.stream("src/main/resources/Cheese.dat")
                        .max(String.CASE_INSENSITIVE_ORDER)
                        .orElse("NONE"));

    }
}
/* Output:
32
a
you
*/
```

代码校验

相关注解

@BeforeAll 在任何其他测试操作之前运行一次的方法

@AfterAll 是所有其他操作之后只运行一次的方法

@BeforeEach 通常用于创建和初始化公共对象的方法，并在每次测试前运行

@AfterEach 通常执行清理操作：如果修改了需要恢复的静态文件，打开文件需要关闭，打开数据库或者网络连接

```
public class CountedListTest {

    private CountedList list;

    // TODO: 2021/9/1 @BeforeAll 在任何其他测试操作之前运行一次的方法
    @BeforeAll
    static void beforeAllMsg() {
        System.out.println(">>> Starting CountedListTest");
    }

    // TODO: 2021/9/1 @AfterAll 是所有其他操作之后只运行一次的方法
    @AfterAll
    static void afterAllMsg() {
        System.out.println(">>> Finished CountedListTest");
    }
```

```java
    // TODO: 2021/9/1 @BeforeEach 通常用于创建和初始化公共对象的方法，并在每次测试前运行
    @BeforeEach
    public void initialize() {
        list = new CountedList();
        System.out.println("Set up for " + list.getId());
        for (int i = 0; i < 3; i++)
            list.add(Integer.toString(i));
    }

    @AfterEach
    public void cleanup() {
        System.out.println("Cleaning up " + list.getId());
    }

    @Test
    public void insert() {
        System.out.println("Running testInsert()");
        assertEquals(list.size(), 3);
        list.add(1, "Insert");
        assertEquals(list.size(), 4);
        assertEquals(list.get(1), "Insert");
    }

    @Test
    public void replace() {
        System.out.println("Running testReplace()");
        assertEquals(list.size(), 3);
        list.set(1, "Replace");
        assertEquals(list.size(), 3);
        assertEquals(list.get(1), "Replace");
    }

    // A helper method to simplify the code. As
    // long as it's not annotated with @Test, it will
    // not be automatically executed by JUnit.
    private void compare(List<String> lst, String[] strs) {
        assertArrayEquals(lst.toArray(new String[0]), strs);
    }

    @Test
    public void order() {
        System.out.println("Running testOrder()");
        compare(list, new String[]{"0", "1", "2"});
    }

    @Test
    public void remove() {
        System.out.println("Running testRemove()");
        assertEquals(list.size(), 3);
        list.remove(1);
        assertEquals(list.size(), 2);
        compare(list, new String[]{"0", "2"});
    }

    @Test
    public void addAll() {
        System.out.println("Running testAddAll()");
        list.addAll(Arrays.asList(new String[]{
```

```
                "An", "African", "Swallow"}));
        assertEquals(list.size(), 6);
        compare(list, new String[]{"0", "1", "2",
                "An", "African", "Swallow"});
    }
}
/* Output:
>>> Starting CountedListTest
CountedList #0
Set up for 0
Running testRemove()
Cleaning up 0
CountedList #1
Set up for 1
Running testReplace()
Cleaning up 1
CountedList #2
Set up for 2
Running testAddAll()
Cleaning up 2
CountedList #3
Set up for 3
Running testInsert()
Cleaning up 3
CountedList #4
Set up for 4
Running testOrder()
Cleaning up 4
>>> Finished CountedListTest
*/
```

Guava pom

```
<!-- https://mvnrepository.com/artifact/com.google.guava/guava -->
        <dependency>
            <groupId>com.google.guava</groupId>
            <artifactId>guava</artifactId>
            <version>21.0</version>
        </dependency>
```

Guava 断言

```
import com.google.common.base.*;
import static com.google.common.base.Verify.*;
public class GuavaAssertions {
    // TODO: 2021/9/1 guava包中 verify和verfyNotNull 的使用
    public static void main(String[] args) {

        verify(2 + 2 == 4);
        try {
            verify(1 + 2 == 4);
        } catch (VerifyException e) {
            e.printStackTrace();
```

```
        }
        System.out.println("=====");
        try {
            verify(1 + 2 == 4, "Bad math");
        } catch (VerifyException e) {
            System.out.println(e.getMessage());
        }
        System.out.println("=====");
        try {
            verify(1 + 2 == 4, "Bad math: %s", "not 4");
        } catch (VerifyException e) {
            System.out.println(e.getMessage());
        }
        System.out.println("=====");
        String s = "";
        s = verifyNotNull(s);
        s = null;
        try {
            verifyNotNull(s);
        } catch (VerifyException e) {
            System.out.println(e.getMessage());
        }
        System.out.println("=====");
        try {
            verifyNotNull(
                    s, "Shouldn't be null: %s", "arg s");
        } catch (VerifyException e) {
            System.out.println(e.getMessage());
        }
    }
}
/* Output:
com.google.common.base.VerifyException
    at com.google.common.base.Verify.verify(Verify.java:99)
    at com.validatingcode.GuavaAssertions.main(GuavaAssertions.java:17)
=====
Bad math
=====
Bad math: not 4
=====
expected a non-null reference
=====
Shouldn't be null: arg s
*/
```

Guava 前置条件

checkArgument()接收布尔表达式来对参数进行更具体的测试， 失败抛出IllegalArgumentException

checkState（expression）获取表达式的状态

checkElementIndex(a,b) 确保 a 是b 有效元素索引，大小由b 指定

checkPositionindex(index,b) 确保 index 在 0到 b.length 范围内

checkPositionIndexes(start,end,a) 检查 strat 到 end 是 a 的有效子列表

```java
package com.validatingcode;// validating/GuavaPreconditions.java
// (c)2021 MindView LLC: see Copyright.txt
// We make no guarantees that this code is fit for any purpose.
// Visit http://OnJava8.com for more book information.
// Demonstrating Guava Preconditions

import java.util.function.Consumer;

import static com.google.common.base.Preconditions.*;

public class GuavaPreconditions {
    static int i = 1;
    static int i2 = 1;
    static String flag2 = "error_";
    static String flag1 = "success_";

    static void test(Consumer<String> c, String s) {
        try {
            System.out.println(s);
            c.accept(s);
            System.out.println(flag1 + i++);
        } catch (Exception e) {
            String type = e.getClass().getSimpleName();
            String msg = e.getMessage();
            System.out.println(flag2 + i2++ + "_: " + type +
                    (msg == null ? "" : ": " + msg));
        }
    }

    // TODO: 2021/9/1 Goava 前置条件
    public static void main(String[] args) {
        test(s -> s = checkNotNull(s), "X");
        test(s -> s = checkNotNull(s), null);
        test(s -> s = checkNotNull(s, "s was null"), null);
        test(s -> s = checkNotNull(
                s, "s was null, %s %s", "arg2", "arg3"), null);
        System.out.println("=======");
        System.out.println("=======");
        System.out.println("=======");
        // TODO: 2021/9/1 checkArgument()接收布尔表达式来对参数进行更具体的测试 ， 失败抛
出IllegalArgumentException
        test(s -> checkArgument(s == "Fozzie"), "Fozzie");
        test(s -> checkArgument(s == "Fozzie"), "X");
        test(s -> checkArgument(s == "Fozzie"), null);
        test(s -> checkArgument(
                s == "Fozzie", "Bear Left!"), null);
        test(s -> checkArgument(
                s == "Fozzie", "Bear Left! %s Right!", "Frog"),
                null);
        System.out.println("=======");
        System.out.println("=======");
        System.out.println("=======");
        // TODO: 2021/9/1 checkState (expression) 获取表达式的状态
        test(s -> checkState(s.length() > 6), "Mortimer");
        test(s -> checkState(s.length() > 6), "Mort");
        test(s -> checkState(s.length() > 6), null);
        System.out.println("=======");
        System.out.println("=======");
```

```
        System.out.println("=======");
        // TODO: 2021/9/1 checkElementIndex(a,b) 确保 a 是b 有效元素索引，大小由b 指定
        test(s ->
                checkElementIndex(6, s.length()), "Robert");
        test(s ->
                checkElementIndex(5, s.length()), "Robert");
        test(s ->
                checkElementIndex(6, s.length()), "Bob");
        test(s ->
                checkElementIndex(6, s.length()), null);
        System.out.println("=======");
        System.out.println("=======");
        System.out.println("=======");
        // TODO: 2021/9/1 checkPositionindex(index,b) 确保 index 在 0到 b.length
范围内
        test(s ->
                checkPositionIndex(3, s.length()), "Robert");
        test(s ->
                checkPositionIndex(6, s.length()), "Robert");
        test(s ->
                checkPositionIndex(6, s.length()), "Bob");
        test(s ->
                checkPositionIndex(6, s.length()), null);
        System.out.println("=======");
        System.out.println("=======");
        System.out.println("=======");
        // TODO: 2021/9/1 checkPositionIndexes(start,end,a) 检查 strat 到 end 是 a
的有效子列表
        test(s -> checkPositionIndexes(
                0, 6, s.length()), "Hieronymus");
        test(s -> checkPositionIndexes(
                0, 10, s.length()), "Hieronymus");
        test(s -> checkPositionIndexes(
                0, 11, s.length()), "Hieronymus");
        test(s -> checkPositionIndexes(
                -1, 6, s.length()), "Hieronymus");
        test(s -> checkPositionIndexes(
                7, 6, s.length()), "Hieronymus");
        test(s -> checkPositionIndexes(
                0, 6, s.length()), null);
    }
}
/* Output:
X
success_1
null
error_1_: NullPointerException
null
error_2_: NullPointerException: s was null
null
error_3_: NullPointerException: s was null, arg2 arg3
=======
=======
=======
Fozzie
success_2
X
error_4_: IllegalArgumentException
```

```
null
error_5_: IllegalArgumentException
null
error_6_: IllegalArgumentException: Bear Left!
null
error_7_: IllegalArgumentException: Bear Left! Frog Right!
=======
=======
=======
Mortimer
success_3
Mort
error_8_: IllegalStateException
null
error_9_: NullPointerException
=======
=======
=======
Robert
error_10_: IndexOutOfBoundsException: index (6) must be less than size (6)
Robert
success_4
Bob
error_11_: IndexOutOfBoundsException: index (6) must be less than size (3)
null
error_12_: NullPointerException
=======
=======
=======
Robert
success_5
Robert
success_6
Bob
error_13_: IndexOutOfBoundsException: index (6) must not be greater than size
(3)
null
error_14_: NullPointerException
=======
=======
=======
Hieronymus
success_7
Hieronymus
success_8
Hieronymus
error_15_: IndexOutOfBoundsException: end index (11) must not be greater than
size (10)
Hieronymus
error_16_: IndexOutOfBoundsException: start index (-1) must not be negative
Hieronymus
error_17_: IndexOutOfBoundsException: end index (6) must not be less than start
index (7)
null
error_18_: NullPointerException

Process finished with exit code 0
```

```
*/
```

基准测试

> 我们应该忘掉微小的效率提升，说的就是这些 97% 的时间做的事：过早的优化是万恶之源。

> 基准测试意味着对代码或算法片段进行计时看哪个跑得更快

微基准测试

1. 创建一个 Timer 对象，执行一些操作然后调用 Timer 的 duration() 方法产生以毫秒为单位的运行时间。
2. 向静态的 duration() 方法中传入 Runnable。任何符合 Runnable 接口的类都有一个函数式方法 run()，该方法没有入参，且没有返回。

```
import static java.util.concurrent.TimeUnit.*;

public class Timer {
    private long start = System.nanoTime();

    public long duration() {
        return NANOSECONDS.toMillis(
                System.nanoTime() - start);
    }

    public static long duration(Runnable test) {
        Timer timer = new Timer();
        test.run();
        return timer.duration();
    }
}
```

测试demo

```
public class BadMicroBenchmark {
    static final int SIZE = 250_000_000;

    public static void main(String[] args) {
        try { // For machines with insufficient memory
            long[] la = new long[SIZE];
            System.out.println("setAll: " +
                    Timer.duration(() ->
                            Arrays.setAll(la, n -> n)));

            System.out.println("parallelSetAll: " +
                    Timer.duration(() ->
                            Arrays.parallelSetAll(la, n -> n)));
        } catch (OutOfMemoryError e) {
            System.out.println("Insufficient memory");
            System.exit(0);
        }
    }
}
/* Output:
```

```
setAll: 153
parallelSetAll: 160
*/
```

测试结果表明：并行的性能没有非并行的好。因为如果你的操作以爱于同一个资源，那么并行版本的运行速度会骤降，因为不同的进程会竞争相同的那个资源

SplittableRandom

- 是为并行算法设计的，它当然看起来比普通的 Random 在parallelSetAll() 中运行得更快

```
import java.util.Arrays;
import java.util.Random;
import java.util.SplittableRandom;

// TODO: 2021/9/1 微基准测试 SplittableRandom 加快random 的速度
public class BadMicroBenchmark2 {
    //减小 SIZE 以使其运行速度更快:
    static final int SIZE = 5_000_000;

    public static void main(String[] args) {
        long[] la = new long[SIZE];
        Random r = new Random();
        System.out.println("parallelSetAll: " +
                Timer.duration(() ->
                        Arrays.parallelSetAll(la, n -> r.nextLong())));

        System.out.println("setAll: " +
                Timer.duration(() ->
                        Arrays.setAll(la, n -> r.nextLong())));

        SplittableRandom sr = new SplittableRandom();
        System.out.println("parallelSetAll: " +
                Timer.duration(() ->
                        Arrays.parallelSetAll(la, n -> sr.nextLong())));

        System.out.println("setAll: " +
                Timer.duration(() ->
                        Arrays.setAll(la, n -> sr.nextLong())));

    }
}
/* Output:
parallelSetAll: 1008
setAll: 294
parallelSetAll: 78
setAll: 88
*/
```

JMH基准测试

环境搭建

```
 mvn archetype:generate -DinteractiveMode=false -
DarchetypeGroupId="org.openjdk.jmh" -DarchetypeArtifactId=jmh-java-benchmark-
archetype -DarchetypeVersion="1.21" -DgroupId=com.jenkov -DartifactId=first-
benchmark -Dversion="1.0"
```

执行

```
mvn clean package
java -jar target/benchmarks.jar
```

优化

jdk下面的VisualVM

启动命令： cmd --> jvisualvm

优化准则

- 避免为了性能牺牲代码的可读性。
- 不要独立地看待性能。衡量与带来的收益相比所需投入的工作量。
- 程序的大小很重要。性能优化通常只对运行了长时间的大型项目有价值。性能通常不是小项目的关注点。
- 运行起来程序比一心钻研它的性能具有更高的优先级。一旦你已经有了可工作的程序，如有必要的话，你可以使用剖析器提高它的效率。只有当性能是关键因素时，才需要在设计/开发阶段考虑性能。
- 不要猜测瓶颈发生在哪。运行剖析器，让剖析器告诉你。
- 无论何时有可能的话，显式地设置实例为 null 表明你不再用它。这对垃圾收集器来说是个有用的暗示。
- static final 修饰的变量会被 JVM 优化从而提高程序的运行速度。因而程序中的常量应该声明 static final。

bug检测

在 IDE 装一个 SpotBugs 插件，他会检测代码会出现的相关问题，并告诉你如何解决它

泛型

枚举

RTTI

> RTTI（RunTime Type Information，运行时类型信息）能够在程序运行时发现和使用类型信息

Class对象

类是程序的一部分，每个类都有一个Class 对象。每当我们编写并编译一个新类时，就会产生一个Class 对象（保存在一个同名的 .class 文件下）。为了生成这个类的对象，JVM先会调用类加载器子系统把这个类加载到内存中。

所有的类都是第一次使用时动态加载到JVM中，当程序创建第一个对类的静态成员引用时，就会加载这个类