

Java 8 - 函数编程(lambda表达式)

如何写出好代码，而不是符合函数编程风格的代码。

简介

在Java世界里面，面向对象还是主流思想，对于习惯了面向对象编程的开发者来说，抽象的概念并不陌生。面向对象编程是对数据进行抽象，而函数式编程是对行为进行抽象。现实世界中，数据和行为并存，程序也是如此，因此这两种编程方式我们都得学。

这种新的抽象方式还有其他好处。很多人不总是在编写性能优先的代码，对于这些人来说，函数式编程带来的好处尤为明显。程序员能编写出更容易阅读的代码——这种代码更多地表达了业务逻辑，而不是从机制上如何实现。易读的代码也易于维护、更可靠、更不容易出错。

在写回调函数和事件处理器时，程序员不必再纠缠于匿名内部类的冗繁和可读性，函数式编程让事件处理系统变得更加简单。能将函数方便地传递也让编写惰性代码变得容易，只有在真正需要的时候，才初始化变量的值。

面向对象编程是对数据进行抽象；函数式编程是对行为进行抽象。

核心思想: 使用不可变值和函数，函数对一个值进行处理，映射成另一个值。

对核心类库的改进主要包括集合类的API和新引入的流Stream。流使程序员可以站在更高的抽象层次上对集合进行操作。

lambda表达式

- lambda表达式仅能放入如下代码: 预定义使用了 `@Functional` 注释的函数式接口，自带一个抽象函数的方法，或者SAM(Single Abstract Method 单个抽象方法)类型。这些称为lambda表达式的目标类型，可以用作返回类型，或lambda目标代码的参数。例如，若一个方法接收Runnable、Comparable或者Callable接口，都有单个抽象方法，可以传入lambda表达式。类似的，如果一个方法接受声明于 `java.util.function` 包内的接口，例如 Predicate、Function、Consumer 或 Supplier，那么可以向其传lambda表达式。
- lambda表达式内可以使用方法引用，仅当该方法不修改lambda表达式提供的参数。本例中的lambda表达式可以换为方法引用，因为这仅是一个参数相同的简单方法调用。

```
list.forEach(n -> System.out.println(n));  
list.forEach(System.out::println); // 使用方法引用
```

然而，若对参数有任何修改，则不能使用方法引用，而需键入完整地lambda表达式，如下所示：

```
list.forEach((String s) -> System.out.println("'" + s + "'"));
```

事实上，可以省略这里的lambda参数的类型声明，编译器可以从列表的类属性推测出来。

- lambda内部可以使用静态、非静态和局部变量，这称为lambda内的变量捕获。
- Lambda表达式在Java中又称为闭包或匿名函数，所以如果有同事把它叫闭包的时候，不用惊讶。
- Lambda方法在编译器内部被翻译成私有方法，并派发 invokedynamic 字节码指令来进行调用。可以使用JDK中的 javap 工具来反编译class文件。使用 javap -p 或 javap -c -v 命令来看一看lambda表达式生成的字节码。大致应该长这样:

```
private static java.lang.Object lambda$0(java.lang.String);
```

- lambda表达式有个限制，那就是只能引用 final 或 final 局部变量，这就是说不能在lambda内部修改定义在域外的变量。

```
List<Integer> primes = Arrays.asList(new Integer[]{2, 3, 5, 7});
int factor = 2;
primes.forEach(element -> { factor++; });
```

Compile time error : "local variables referenced from a lambda expression must be final or effectively final" 另外，只是访问它而不作修改是可以的，如下所示:

```
List<Integer> primes = Arrays.asList(new Integer[]{2, 3, 5, 7});
int factor = 2;
primes.forEach(element -> { System.out.println(factor*element); });
```

分类

惰性求值方法

```
lists.stream().filter(f -> f.getName().equals("p1"))
```

如上示例，这行代码并未做什么实际性的工作，filter只是描述了Stream，没有产生新的集合。

如果是多个条件组合，可以通过代码块 {}

及早求值方法

```
List<Person> list2 = lists.stream().filter(f ->
f.getName().equals("p1")).collect(Collectors.toList());
```

如上示例，collect最终会从Stream产生新值，拥有终止操作。

理想方式是形成一个惰性求值的链，最后用一个及早求值的操作返回想要的结果。与建造者模式相似，建造者模式先是使用一系列操作设置属性和配置，最后调用build方法，创建对象。

stream & parallelStream

stream & parallelStream

每个Stream都有两种模式: 顺序执行和并行执行。

顺序流:

```
List <Person> people = list.getStream.collect(Collectors.toList());
```

并行流:

```
List <Person> people = list.getStream.parallel().collect(Collectors.toList());
```

顾名思义, 当使用顺序方式去遍历时, 每个item读完后读下一个item。而使用并行去遍历时, 数组会被分成多个段, 其中每一个都在不同的线程中处理, 然后将结果一起输出。

*parallelStream*原理:

```
List originalList = someData;  
split1 = originalList(0, mid); //将数据分小部分  
split2 = originalList(mid, end);  
new Runnable(split1.process()); //小部分执行操作  
new Runnable(split2.process());  
List revisedList = split1 + split2; //将结果合并
```

大家对hadoop有稍微了解就知道, 里面的 MapReduce 本身就是用于并行处理大数据集的软件框架, 其处理大数据的核心思想就是大而化小, 分配到不同机器去运行map, 最终通过reduce将所有机器的结果结合起来得到一个最终结果, 与MapReduce不同, Stream则是利用多核技术可将大数据通过多核并行处理, 而MapReduce则可以分布式的。

stream与parallelStream性能测试对比

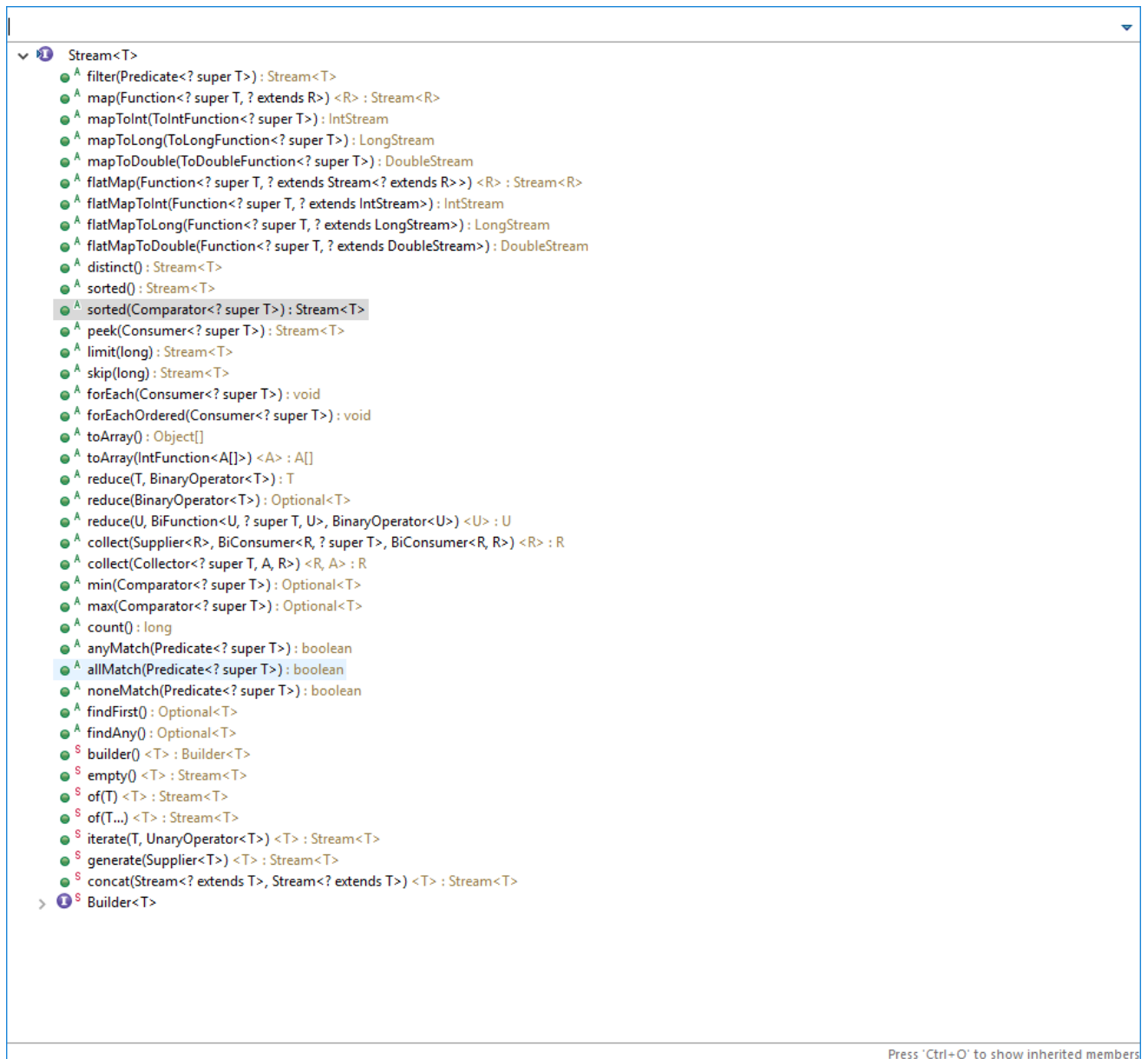
如果是多核机器, 理论上并行流则会比顺序流快上一倍, 下面是测试代码

```
long t0 = System.nanoTime();  
  
//初始化一个范围100万整数流, 求能被2整除的数字, toArray()是终点方法  
  
int a[] = IntStream.range(0, 1_000_000).filter(p -> p % 2 == 0).toArray();  
  
long t1 = System.nanoTime();  
  
//和上面功能一样, 这里是用并行流来计算  
  
int b[] = IntStream.range(0, 1_000_000).parallel().filter(p -> p % 2 == 0).toArray();  
  
long t2 = System.nanoTime();  
  
//我本机的结果是serial: 0.06s, parallel 0.02s, 证明并行流确实比顺序流快  
  
System.out.printf("serial: %.2fs, parallel %.2fs\n", (t1 - t0) * 1e-9, (t2 - t1) * 1e-9);
```

Stream中常用方法如下:

- stream(), parallelStream()
- filter()
- findAny() findFirst()
- sort
- forEach void
- map(), reduce()
- flatMap() - 将多个Stream连接成一个Stream
- collect(Collectors.toList())
- distinct, limit
- count
- min, max, summaryStatistics

看下所有API:



The screenshot shows the Java Stream API documentation for `Stream<T>` in an IDE. The list of methods includes:

- `filter(Predicate<? super T>) : Stream<T>`
- `map(Function<? super T, ? extends R>) <R> : Stream<R>`
- `mapToInt(ToIntFunction<? super T>) : IntStream`
- `mapToLong(ToLongFunction<? super T>) : LongStream`
- `mapToDouble(ToDoubleFunction<? super T>) : DoubleStream`
- `flatMap(Function<? super T, ? extends Stream<? extends R>>) <R> : Stream<R>`
- `flatMapToInt(Function<? super T, ? extends IntStream>) : IntStream`
- `flatMapToLong(Function<? super T, ? extends LongStream>) : LongStream`
- `flatMapToDouble(Function<? super T, ? extends DoubleStream>) : DoubleStream`
- `distinct() : Stream<T>`
- `sorted() : Stream<T>`
- `sorted(Comparator<? super T>) : Stream<T>`
- `peek(Consumer<? super T>) : Stream<T>`
- `limit(long) : Stream<T>`
- `skip(long) : Stream<T>`
- `forEach(Consumer<? super T>) : void`
- `forEachOrdered(Consumer<? super T>) : void`
- `toArray() : Object[]`
- `toArray(IntFunction<A[]>) <A> : A[]`
- `reduce(T, BinaryOperator<T>) : T`
- `reduce(BinaryOperator<T>) : Optional<T>`
- `reduce(U, BiFunction<U, ? super T, U>, BinaryOperator<U>) <U> : U`
- `collect(Supplier<R>, BiConsumer<R, ? super T>, BiConsumer<R, R>) <R> : R`
- `collect(Collector<? super T, A, R>) <R, A> : R`
- `min(Comparator<? super T>) : Optional<T>`
- `max(Comparator<? super T>) : Optional<T>`
- `count() : long`
- `anyMatch(Predicate<? super T>) : boolean`
- `allMatch(Predicate<? super T>) : boolean`
- `noneMatch(Predicate<? super T>) : boolean`
- `findFirst() : Optional<T>`
- `findAny() : Optional<T>`
- `builder() <T> : Builder<T>`
- `empty() <T> : Stream<T>`
- `of(T) <T> : Stream<T>`
- `of(T...) <T> : Stream<T>`
- `iterate(T, UnaryOperator<T>) <T> : Stream<T>`
- `generate(Supplier<T>) <T> : Stream<T>`
- `concat(Stream<? extends T>, Stream<? extends T>) <T> : Stream<T>`

At the bottom, there is a note: "Press 'Ctrl+O' to show inherited members".

常用例子

匿名类简写

```
new Thread( () -> System.out.println("In Java8, Lambda expression rocks !!") ).start();

// 用法
(params) -> expression
(params) -> statement
(params) -> { statements }
```

forEach

```
// forEach
List features = Arrays.asList("Lambdas", "Default Method", "Stream API", "Date and Time API");
features.forEach(n -> System.out.println(n));

// 使用Java 8的方法引用更方便，方法引用由::双冒号操作符标示，
features.forEach(System.out::println);
```

方法引用

构造引用

```
// Supplier<Student> s = () -> new Student();
Supplier<Student> s = Student::new;
```

对象::实例方法 Lambda表达式的(形参列表)与实例方法的(实参列表)类型，个数是对应

```
// set.forEach(t -> System.out.println(t));
set.forEach(System.out::println);
```

类名::静态方法

```
// Stream<Double> stream = Stream.generate(() -> Math.random());
Stream<Double> stream = Stream.generate(Math::random);
```

类名::实例方法

```
// TreeSet<String> set = new TreeSet<>((s1,s2) -> s1.compareTo(s2));
/* 这里如果使用第一句话，编译器会有提示：Can be replaced with Comparator.naturalOrder，这句话告诉我们
String已经重写了compareTo()方法，在这里写是多此一举，这里为什么这么写，是因为为了体现下面
这句编译器的提示：Lambda can be replaced with method reference。好了，下面的这句就是改写成方法引用
之后：
*/
TreeSet<String> set = new TreeSet<>(String::compareTo);
```

Filter & Predicate

常规用法

```
public static void main(args[]){
    List languages = Arrays.asList("Java", "Scala", "C++", "Haskell", "Lisp");

    System.out.println("Languages which starts with J :");
    filter(languages, (str)->str.startsWith("J"));

    System.out.println("Languages which ends with a ");
    filter(languages, (str)->str.endsWith("a"));

    System.out.println("Print all languages :");
    filter(languages, (str)->true);

    System.out.println("Print no language : ");
    filter(languages, (str)->false);

    System.out.println("Print language whose length greater than 4:");
    filter(languages, (str)->str.length() > 4);
}

public static void filter(List names, Predicate condition) {
    names.stream().filter((name) -> (condition.test(name))).forEach((name) -> {
        System.out.println(name + " ");
    });
}
```

多个Predicate组合filter

```
// 可以用and()、or()和xor()逻辑函数来合并Predicate,
// 例如要找到所有以J开始, 长度为四个字母的名字, 你可以合并两个Predicate并传入
Predicate<String> startsWithJ = (n) -> n.startsWith("J");
Predicate<String> fourLetterLong = (n) -> n.length() == 4;
names.stream()
    .filter(startsWithJ.and(fourLetterLong))
    .forEach((n) -> System.out.print("nName, which starts with 'J' and four letter long is : " +
n));
```

Map&Reduce

map将集合类(例如列表)元素进行转换的。还有一个 reduce() 函数可以将所有值合并成一个

```
List costBeforeTax = Arrays.asList(100, 200, 300, 400, 500);
double bill = costBeforeTax.stream().map((cost) -> cost + .12*cost).reduce((sum, cost) -> sum +
cost).get();
System.out.println("Total : " + bill);
```

Collectors

// 将字符串换成大写并用逗号链接起来

```
List<String> G7 = Arrays.asList("USA", "Japan", "France", "Germany", "Italy", "U.K.", "Canada");
String G7Countries = G7.stream().map(x -> x.toUpperCase()).collect(Collectors.joining(", "));
System.out.println(G7Countries);
```

- Collectors.joining(", ")
- Collectors.toList()
- Collectors.toSet(), 生成set集合
- Collectors.toMap(MemberModel::getUid, Function.identity())
- Collectors.toMap(ImageModel::getAid, o -> IMAGE_ADDRESS_PREFIX + o.getUrl())

flatMap

将多个Stream连接成一个Stream

```
List<Integer> result= Stream.of(Arrays.asList(1,3),Arrays.asList(5,6)).flatMap(a-
->a.stream()).collect(Collectors.toList());
```

结果: [1, 3, 5, 6]

distinct

去重

```
List<LikeDO> likeDOs=new ArrayList<LikeDO>();
List<Long> likeTidList = likeDOs.stream().map(LikeDO::getTid)
    .distinct().collect(Collectors.toList());
```

count

计总数

```
int countOfAdult=persons.stream()
    .filter(p -> p.getAge() > 18)
    .map(person -> new Adult(person))
    .count();
```

Match

```
boolean anyStartsWithA =
    stringCollection
        .stream()
        .anyMatch((s) -> s.startsWith("a"));

System.out.println(anyStartsWithA);    // true

boolean allStartsWithA =
    stringCollection
        .stream()
        .allMatch((s) -> s.startsWith("a"));

System.out.println(allStartsWithA);    // false

boolean noneStartsWithZ =
    stringCollection
        .stream()
        .noneMatch((s) -> s.startsWith("z"));

System.out.println(noneStartsWithZ);    // true
```

min,max,summaryStatistics

最小值，最大值

```
List<Person> lists = new ArrayList<Person>();
lists.add(new Person(1L, "p1"));
lists.add(new Person(2L, "p2"));
lists.add(new Person(3L, "p3"));
lists.add(new Person(4L, "p4"));
Person a = lists.stream().max(Comparator.comparing(t -> t.getId())).get();
System.out.println(a.getId());
```

如果比较器涉及多个条件，比较复杂，可以定制

```
Person a = lists.stream().min(new Comparator<Person>() {

    @Override
    public int compare(Person o1, Person o2) {
        if (o1.getId() > o2.getId()) return -1;
        if (o1.getId() < o2.getId()) return 1;
        return 0;
    }
}).get();
```

summaryStatistics


```
//获取数字的个数、最小值、最大值、总和以及平均值
List<Integer> primes = Arrays.asList(2, 3, 5, 7, 11, 13, 17, 19, 23, 29);
IntSummaryStatistics stats = primes.stream().mapToInt((x) -> x).summaryStatistics();
System.out.println("Highest prime number in List : " + stats.getMax());
System.out.println("Lowest prime number in List : " + stats.getMin());
System.out.println("Sum of all prime numbers : " + stats.getSum());
System.out.println("Average of all prime numbers : " + stats.getAverage());
```

peek

可以使用peek方法，peek方法可只包含一个空的方法体，只要能设置断点即可，但有些IDE不允许空，可以如下文示例，简单写一个打印逻辑。

注意，调试完后要删掉。

```
List<Person> lists = new ArrayList<Person>();
lists.add(new Person(1L, "p1"));
lists.add(new Person(2L, "p2"));
lists.add(new Person(3L, "p3"));
lists.add(new Person(4L, "p4"));
System.out.println(lists);

List<Person> list2 = lists.stream()
    .filter(f -> f.getName().startsWith("p"))
    .peek(t -> {
        System.out.println(t.getName());
    })
    .collect(Collectors.toList());
System.out.println(list2);
```

FunctionalInterface

理解注解 *@FunctionInterface*

```
/**
 * An informative annotation type used to indicate that an interface
 * type declaration is intended to be a <i>functional interface</i> as
 * defined by the Java Language Specification.
 *
 * Conceptually, a functional interface has exactly one abstract
 * method. Since {@linkplain java.lang.reflect.Method#isDefault()}
 * default methods} have an implementation, they are not abstract. If
 * an interface declares an abstract method overriding one of the
 * public methods of {@code java.lang.Object}, that also does
 * <em>not</em> count toward the interface's abstract method count
 * since any implementation of the interface will have an
 * implementation from {@code java.lang.Object} or elsewhere.
 *
 * <p>Note that instances of functional interfaces can be created with
 * lambda expressions, method references, or constructor references.
 *
 * <p>If a type is annotated with this annotation type, compilers are
```

```

* required to generate an error message unless:
*
* <ul>
* <li> The type is an interface type and not an annotation type, enum, or class.
* <li> The annotated type satisfies the requirements of a functional interface.
* </ul>
*
* <p>However, the compiler will treat any interface meeting the
* definition of a functional interface as a functional interface
* regardless of whether or not a {@code FunctionalInterface}
* annotation is present on the interface declaration.
*
* @jls 4.3.2. The Class Object
* @jls 9.8 Functional Interfaces
* @jls 9.4.3 Interface Method Body
* @since 1.8
*/
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface FunctionalInterface{}

```

- interface做注解的注解类型，被定义成java语言规范
- 一个被它注解的接口只能有一个抽象方法，有两种例外
- 第一是接口允许有实现的方法，这种实现的方法是用 default 关键字来标记的 (java 反射中 java.lang.reflect.Method#isDefault()方法用来判断是否是default方法)
- 第二如果声明的方法和java.lang.Object中的某个方法一样，它可以不当做未实现的方法，不违背这个原则: 一个被它注解的接口只能有一个抽象方法, 比如: java public interface Comparator<T> { int compare(T o1, T o2); boolean equals(Object obj); }
- 如果一个类型被这个注解修饰，那么编译器会要求这个类型必须满足如下条件:
 - 这个类型必须是一个interface，而不是其他的注解类型、枚举enum或者类class
 - 这个类型必须满足function interface的所有要求，如你个包含两个抽象方法的接口增加这个注解，会有编译错误。
- 编译器会自动把满足function interface要求的接口自动识别为function interface，所以你不需要对上面示例中的 ITest接口增加@FunctionalInterface注解。

自定义函数接口

```

@FunctionalInterface
public interface IMyInterface {
    void study();
}

package com.isea.java;
public class TestIMyInterface {
    public static void main(String[] args) {
        IMyInterface iMyInterface = () -> System.out.println("I like study");
        iMyInterface.study();
    }
}

```

内置四大函数接口

- 消费型接口: `Consumer< T> void accept(T t)`有参数, 无返回值的抽象方法;

比如: `map.forEach(BiConsumer<A, T>)`

```
Consumer<Person> greeter = (p) -> System.out.println("Hello, " + p.firstName);
greeter.accept(new Person("Luke", "Skywalker"));
```

- 供给型接口: `Supplier< T> T get()` 无参有返回值的抽象方法;

以`stream().collect(Collector<? super T, A, R> collector)`为例:

比如:

```
Supplier<Person> personSupplier = Person::new;
personSupplier.get();    // new Person
```

再如:

```
// 调用方法
<R, A> R collect(Collector<? super T, A, R> collector)

// Collectors.toSet
public static <T>
    Collector<T, ?, Set<T>> toSet() {
    return new CollectorImpl<>((Supplier<Set<T>>) HashSet::new, Set::add,
        (left, right) -> { left.addAll(right); return left; },
        CH_UNORDERED_ID);
}

// CollectorImpl
private final Supplier<A> supplier;
private final BiConsumer<A, T> accumulator;
private final BinaryOperator<A> combiner;
private final Function<A, R> finisher;
private final Set<Characteristics> characteristics;

CollectorImpl(Supplier<A> supplier,
    BiConsumer<A, T> accumulator,
    BinaryOperator<A> combiner,
    Function<A, R> finisher,
    Set<Characteristics> characteristics) {
    this.supplier = supplier;
    this.accumulator = accumulator;
    this.combiner = combiner;
    this.finisher = finisher;
    this.characteristics = characteristics;
}

CollectorImpl(Supplier<A> supplier,
    BiConsumer<A, T> accumulator,
    BinaryOperator<A> combiner,
    Set<Characteristics> characteristics) {
    this(supplier, accumulator, combiner, castingIdentity(), characteristics);
}

// collect()方法实现
public final <R, A> R collect(Collector<? super P_OUT, A, R> collector) {
```

```

A container;
if (isParallel()
    && (collector.characteristics().contains(Collector.Characteristics.CONCURRENT))
    && (!isOrdered() ||
collector.characteristics().contains(Collector.Characteristics.UNORDERED))) {
    container = collector.supplier().get();
    BiConsumer<A, ? super P_OUT> accumulator = collector.accumulator();
    forEach(u -> accumulator.accept(container, u));
}
else {
    container = evaluate(ReduceOps.makeRef(collector));
}
return collector.characteristics().contains(Collector.Characteristics.IDENTITY_FINISH)
    ? (R) container
    : collector.finisher().apply(container);
}

```

- 断定型接口: Predicate boolean test(T t):有参, 但是返回值类型是固定的boolean

比如: stream().filter()中参数就是Predicate

```

Predicate<String> predicate = (s) -> s.length() > 0;

predicate.test("foo");           // true
predicate.negate().test("foo");   // false

Predicate<Boolean> nonNull = Objects::nonNull;
Predicate<Boolean> isNull = Objects::isNull;

Predicate<String> isEmpty = String::isEmpty;
Predicate<String> isEmpty = isEmpty.negate();

```

- 函数型接口: Function<T,R> R apply(T t)有参有返回值的抽象方法;

比如: stream().map() 中参数就是 Function<? super T, ? extends R>; reduce() 中参数 BinaryOperator (ps: BinaryOperator extends BiFunction<T,T,T>)

```

Function<String, Integer> toInteger = Integer::valueOf;
Function<String, String> backToString = toInteger.andThen(String::valueOf);

backToString.apply("123");       // "123"

```

一些例子

- 输出 年龄>25的女程序员中名字排名前3位的姓名

```

javaProgrammers.stream()
    .filter((p) -> (p.getAge() > 25))
    .filter((p) -> ("female".equals(p.getGender())))
    .sorted((p, p2) -> (p.getFirstName().compareTo(p2.getFirstName())))
    .limit(3)
    //.forEach(e -> e.setSalary(e.getSalary() / 100 * 5 + e.getSalary()))//涨工资
    .forEach((p) -> System.out.printf("%s %s; ", p.getFirstName(), p.getLastName()));

```

- 工资最高的 Java programmer

```

Person person = javaProgrammers
    .stream()
    .max((p, p2) -> (p.getSalary() - p2.getSalary()))
    .get()

```

- 将 Java programmers 的 first name 存放到 TreeSet

```

TreeSet<String> javaDevLastName = javaProgrammers
    .stream()
    .map(Person::getLastName)
    .collect(toCollection(TreeSet::new))

```

- 计算付给 Java programmers 的所有 money

```

int totalSalary = javaProgrammers
    .parallelStream()
    .mapToInt(p -> p.getSalary())
    .sum();

```

- Comparator多属性排序: 先按名字不分大小写排, 再按GID倒序排, 最后按年龄正序排

```

public static void main(String[] args) {
    List<Person> personList = getTestList();
    personList.sort(Comparator.comparing(Person::getName, String.CASE_INSENSITIVE_ORDER)
        .thenComparing(Person::getGid, (a, b) -> b.compareTo(a))
        .thenComparingInt(Person::getAge));
    personList.stream().forEach(System.out::println);
}

public static List<Person> getTestList() {
    return Lists.newArrayList(new Person("dai", "301", 10), new Person("dai", "303", 10),
        new Person("dai", "303", 8), new Person("dai", "303", 6), new Person("dai", "303",
11),
        new Person("dai", "302", 9), new Person("zhang", "302", 9), new Person("zhang",
"301", 9),
        new Person("Li", "301", 8));
}

// 输出结果
// Person [name=dai, gid=303, age=6]
// Person [name=dai, gid=303, age=8]
// Person [name=dai, gid=303, age=10]
// Person [name=dai, gid=303, age=11]
// Person [name=dai, gid=302, age=9]
// Person [name=dai, gid=301, age=10]
// Person [name=Li, gid=301, age=8]
// Person [name=zhang, gid=302, age=9]
// Person [name=zhang, gid=301, age=9]

```

- 处理字符串

两个新的方法可在字符串类上使用: join和chars。第一个方法使用指定的分隔符, 将任何数量的字符串连接为一个字符串。

```

String.join(":", "foobar", "foo", "bar");
// => foobar:foo:bar

```

第二个方法chars从字符串所有字符创建数据流，所以你可以在这些字符上使用流式操作。

```
"foobar:foo:bar"
  .chars()
  .distinct()
  .mapToObj(c -> String.valueOf((char)c))
  .sorted()
  .collect(Collectors.joining());
// => :abfor
```

不仅仅是字符串，正则表达式模式串也能受益于数据流。我们可以分割任何模式串，并创建数据流来处理它们，而不是将字符串分割为单个字符的数据流，像下面这样：

```
Pattern.compile(":")
  .splitAsStream("foobar:foo:bar")
  .filter(s -> s.contains("bar"))
  .sorted()
  .collect(Collectors.joining(":"));
// => bar:foobar
```

此外，正则模式串可以转换为谓词。这些谓词可以像下面那样用于过滤字符串流：

```
Pattern pattern = Pattern.compile(".*@gmail\\.com");
Stream.of("bob@gmail.com", "alice@hotmail.com")
  .filter(pattern.asPredicate())
  .count();
// => 1
```

上面的模式串接受任何以@gmail.com结尾的字符串，并且之后用作Java8的Predicate来过滤电子邮件地址流。

■ Local Cache实现

```
public class TestLocalCache {

    private static ConcurrentHashMap<Integer, Long> cache = new ConcurrentHashMap<>();

    static long fibonacci(int i) {
        if (i == 0)
            return i;

        if (i == 1)
            return 1;

        return cache.computeIfAbsent(i, (key) -> {
            System.out.println("Slow calculation of " + key);

            return fibonacci(i - 2) + fibonacci(i - 1);
        });
    }

    public static void main(String[] args) {
        // warm up
        for (int i = 0; i < 101; i++)
            System.out.println(
                "f(" + i + ") = " + fibonacci(i));

        // read -> cal
        long current = System.currentTimeMillis();
    }
}
```

```

        System.out.println(fibonacci(100));
        System.out.println(System.currentTimeMillis()-current);
    }
}

```

- 集合--》取元素的一个属性--》去重---》组装成List--》返回

```

List<LikeDO> likeDOs=new ArrayList<LikeDO>();
List<Long> likeTidList = likeDOs.stream().map(LikeDO::getTid)
    .distinct().collect(Collectors.toList());

```

- 集合--》按表达式过滤--》遍历、每个元素处理--》放入预先定义的集合中

```

Map<String, StkProduct> newStockName2Product = Maps.newConcurrentMap();
stockProducts.stream().filter(stkProduct -> stkProduct.enabled).forEach(stkProduct -> {
    String newName = BCConvert.bj2qj(StringUtils.replace(stkProduct.name, " ", ""));
    newStockName2Product.put(newName, stkProduct);
});

```

```

Set<String> qjStockNames;
qjStockNames.stream().filter(name -> !acAutomaton.getKey2link().containsKey(name)).forEach(name
-> {
    String value = "";
    StkProduct stkProduct = stockNameQj2Product.get(name);
    if (stkProduct != null) {
        value = stkProduct.name;
    }
    acAutomaton.getKey2link().put(name, value);
});

```

- 集合--》map

```

List<ImageModel> imageModelList = null;
Map<Long, String> imagesMap = null;
imagesMap = imageModelList.stream().collect(Collectors.toMap(ImageModel::getAid, o ->
    IMAGE_ADDRESS_PREFIX + o.getUrl()));

Map<String, String> kvMap = postDetailCacheList.stream().collect(Collectors.toMap((detailCache)
->
    getBbsSimplePostKey(detailCache.getTid()), JSON::toJSONString));

Map<Long, Long> pidToTid;
List<String> pidKeyList = pidToTid.entrySet().stream().map((o) ->
    getKeyBbsReplyPid(o.getValue(), o.getKey())).collect(Collectors.toList());

```

- DO模型---》Model模型

```

List<AdDO> adDOList;
adDOList.stream().map(adDo -> convertAdModel(adDo))
    .collect(Collectors.toList());

```

- phones 是一个List, 将相同的元素分组、归类

```
List<String> phones=new ArrayList<String>();
    phones.add("a");
    phones.add("b");
    phones.add("a");
    phones.add("a");
    phones.add("c");
    phones.add("b");
    Map<String, List<String>> phoneClassify =
phones.stream().collect(Collectors.groupingBy(item -> item));
    System.out.println(phoneClassify);
```

返回结果:

```
{a=[a, a, a], b=[b, b], c=[c]}
```