Java 8 Stream API 实用指南

本文作为 Stream API 的“使用指南”，主要侧重于“实用”，并不会关注太多的实现细节，当然，不是简单地罗列接口，而是尽可能地向读者展示 Stream API 的全貌。

开始之前

作为 Java API 的新成员，Stream API “允许以声明式的方式处理数据集合”。回顾“内容介绍”部分，我们阅读了以下的代码：

```
class Good {
  String name;                 // 商品名称
  long price;                  // 价格
  long sales;                  // 销量
  List categories;  // 类别

  // ... 省略 constructor、getter / setter

  // ... 省略 toString
}

void process(List goods) {
  //
  // 筛选 price > 500 & sales
```


Lambda 表达式 & 方法引用

```
interface PriceCalculator {
  long calculate(Good good);
};

public void process(List goods, PriceCalculator calculator) {
  // 计算商品价格
}

//
// 实现 PriceCalculator 接口的匿名类实例，作为 process 参数
//
process(goods, new PriceCalculator() {
  @Override
  public long calculate(Good good) {
    return good.getPrice();
  }
});
```

接口 PriceCalculator 只有一个方法，我们将只有一个抽象方法的接口，称为“函数式接口”，并以 @FunctionalInterface 进行标记。（注意，Java 8 允许接口提供方法实现，即“默认方法”，函数式接口必须包含且仅包含一个抽象方法，对于提供实现的默认方法，没有限制）

Lambda 表达式，其本质即为函数式接口的一个实例：


```
//
// 示例 #1: args -> { statement; }
//
process(goods, (good) -> {
  return good.getPrice();
});

//
// 示例 #2: args -> expression
```

```
process(goods, Good::getPrice);
```

对于 Lambda 表达式到方法引用的简化，我们提供以下规则：

Lambda 表达式	方法引用
`(args) -> ClassName.staticMethod(args)`	`ClassName::staticMethod`
`(arg0, ...) -> arg0.instanceMethod(...)`	`ClassName::instanceMethod`
`(args) -> expression.instanceMethod(args)`	`expression::instanceMethod`

特别的，对于构造函数的方法引用： `ClassName::new`

开始使用 Stream API

本章节将阐述 Stream 的生成、操作、数据收集，主要介绍 Stream API 的常用接口与辅助方法。为了便于我们试验示例的代码，我们先说明 `forEach(Consumer`

生成 Stream

由集合 & 数组生成 Stream

Stream 作为元素的“序列”，自然而然地，我们想到通过集合、数组生成 Stream。

Java 8 的 Collection 接口添加了 `Stream stream()` 方法，由集合生成 Stream，例如：

```
,,
```

```
    Arrays.stream(goods).forEach(System.out::println);
  }
```

特别地，当数组元素类型 `T` 是原始类型，静态方法 `stream(T[])` 将返回原始类型的 Stream。

通过集合或数组获得的 Stream，是“有限”的。

直接创建 Stream

除了由集合和数组生成 Stream，Stream API 提供了静态方法 `Stream.generate(Supplier)`、`Stream.iterator(final T, final UnaryOperator)`，直接创建 Stream。

`Stream.generate(Supplier)` 通过参数 `Supplier` 获取 Stream 序列的新元素

```
  //
  // 生成指定数量的商品并输出
  //
  void generate(int number) {

  Stream.generate(Good::new).limit(number).forEach(System.out::print
  ln);
  }
```


`Stream.iterator(final T, final UnaryOperator)` 提供了一种“迭代”的形式：第一个元素，以及第 n 个元素到第 n + 1 个元素的生成方式 `UnaryOperator`。

```
  //
  // 生成指定数量的序列 1, 2, 4, 8, 16 ... 并输出
  //
  void generateSequence(int number) {
```

filter 是“中间操作”，以 `Predicate`

anyMatch / allMatch / noneMatch

anyMatch、allMatch、noneMatch，都是“终止操作”，与 filter 接收相同的参数，其功能顾名思义，例如：

```
//
// 检查商品集合是否包含指定名称的商品
//
boolean hasGoodWithName(List goods, String name) {
  return goods.stream().anyMatch(c -> name.equals(c.getName()));
}
```

findAny / findFirst

findAny、findFirst，都是“终止操作”，分别获取 Stream 元素序列的任意元素和第一个元素：


```
//
// 获取商品集合中任意名称为指定名称的商品
//
Optional findAnyGoodWithName(List goods, String name) {
  return goods.stream().filter(c ->
name.equals(c.getName())).findAny();
}
```

findAny、findFirst 的返回值都是 `Optional` 类型，避免了 Stream 序列为空时返回 null。关于 `Optional` 类型，不属于本文的范围，请参阅 Java doc

```
//
// 获取高于指定价格的商品数量
//
long countGoodsOverPrice(List goods, long price) {
  return goods.stream().filter(c -> c.getPrice() > price).count();
}
```

min / max，以 Comparator

并行

通过“并行 Stream”即可获得 Stream API 的并行能力，例如：

```
//
// 获取最高的商品价格
//
OptionalLong maxGoodPrice(List goods) {
  return
goods.stream().parallel().mapToLong(Good::getPrice).max();
}
```


代码所示，通过 Collection 接口的 parallelStream()、BaseStream 接口的 parallel() 方法，都能够获得“并行 Stream”。

并行 Stream 内部是基于 ForkJoinPool 模型获得并行能力，其默认线程数量即为通过 Runtime.getRuntime().availableProcessors() 获得的线程数。

不过，关于并行，两件事必须注意：一方面，正确性，避免 Stream 处理过程中共享可变状态；另一方面，务必记住，并行未必能够提高性能，通常适用于 Stream 元素数量大

```
true; }).limit(1).collect(Collectors.toList());
```

2. 分别运用 Stream API 的 reduce、collect 方法实现以下方法：

```
long getTotalSalesAmount(List goods) {
  //
  // 获取 goods 的销售总金额
  //
}
```

3. 通过 Stream API 实现以下方法：

```
void printFibonacciSequence(int n) {
  //
  // 输出斐波那契数列的前 n 个数
  //
}
```


4. 通过 Stream API 改造以下代码：（提示，需要了解 collect 方法的参数类型 `Collector`