# Java 8 习惯用语

第 1 部分: Java 中的一种更轻松的函数式编程途径

[第 2 部分: 函数组合与集合管道模式](http://www.ibm.com/developerworks/cn/java/j-java8idioms2/index.html?ca=drs-)

[第 3 部分: 传统 for 循环的函数式替代方案](http://www.ibm.com/developerworks/cn/java/j-java8idioms3/index.html?ca=drs-)

[第 4 部分: 提倡使用有帮助的编码](http://www.ibm.com/developerworks/cn/java/j-java8idioms4/index.html?ca=drs-)

[第 5 部分: 传递表达式（pass-through lambdas）的替代方案](http://www.ibm.com/developerworks/cn/java/j-java8idioms5/index.html?ca=drs-)

[第 6 部分: 为什么完美的 lambda 表达式只有一行](http://www.ibm.com/developerworks/cn/java/j-java8idioms6/index.html?ca=drs-)

# Java 中的一种更轻松的函数式编程途径

以声明方式思考如何在 Java 程序中采用函数方法

ava 开发人员早已习惯了命令式编程和面向对象的编程，因为 Java 语言从第一个版本开始就支持这些格式。在 Java 8 中，我们获得了一组强大的新的函数特性和语法。函数式编程已有数十年的历史，而且与面向对象的编程相比，函数式编程通常更简洁、更具表达力、更不容易出错，而且更容易并行化。所以在 Java 程序中引入函数特性是有充分理由的。尽管如此，函数式的编程需要对代码的设计方式进行一些改变。

我发现声明式思考（而不是命令式思考）可以简化向更加函数化编程的过渡。在新的 [*Java 8 习惯用语* 系列](https://www.ibm.com/developerworks/cn/views/global/libraryview.jsp?sort_by=&show_abstract=true&show_all=&search_flag=&contentarea_by=Jamp;ava+technology&search_by=Java+8++%E4%B9%A0%E6%83%AF%E7%94%A8%E8%AF%AD&topic_by=-1&type_by=%E6%89%80%E6%9C%89%E7%B1%BB%E5%88%AB&ibm-search=%E6%90%9C%E7%B4%A2) 的第一篇文章中，我将解释命令式、声明式和函数式编程之间的区别和共性。然后，将展示如何使用声明式思考方式逐步将函数方法集成到 Java 程序中。

## 命令式格式

经过针对命令式编程培训的开发人员已经习惯了告诉程序做什么和该如何做。这是一个简单示例：

清单 1. findNemo in the imperative style

|  |
| --- |
| import java.util.\*;    public class FindNemo {  public static void main(String[] args) {  List<String> names =  Arrays.asList("Dory", "Gill", "Bruce", "Nemo",  "Darla", "Marlin", "Jacques");      findNemo(names);  }    public static void findNemo(List<String> names) {    boolean found = false;    for(String name : names) {      if(name.equals("Nemo")) {       found = true;         break;      }    }        if(found)        System.out.println("Found Nemo");      else        System.out.println("Sorry, Nemo not found");    }  } |

findNemo() 方法首先初始化一个可变的*flag* 变量，也称为*垃圾（garbage）变量*。开发人员通常会随便给这些变量命名，比如 f、t 或 temp，这些名称表明了我们对这些变量不应存在的一般态度。在本例中，该变量名为 found。

接下来，该程序会在给定的 names 列表中循环，一次处理一个元素。它检查获得的名称是否与它寻找的值（在本例中为 Nemo）相等。如果值匹配，那么它会将 flag 变量设置为 true，并通知控制流“跳出”循环。

因为这是一个命令式编程的程序 — 许多 Java 开发人员最熟悉的格式 — 所以您需要定义程序的每一步：告诉它迭代每个元素，比较值，设置 flag 变量，然后跳出循环。命令式格式为您提供了完全的控制权，这有时是件好事。而另一方面，您需要执行所有工作。在许多情况下，可以减少工作量来提高效率。

## 声明式格式

声明式编程意味着，您仍会告诉程序要做什么，但将实现细节留给底层的函数库。让我们看看用声明式格式重新编写来自 [清单 1](https://www.ibm.com/developerworks/cn/java/j-java8idioms1/index.html?ca=drs-#清单 1) 的 findNemo 方法时，会发生什么：

清单 2. findNemo in the declarative style

|  |
| --- |
| public static void findNemo(List<String> names) {    if(names.contains("Nemo"))      System.out.println("Found Nemo");    else      System.out.println("Sorry, Nemo not found");  } |

首先请注意，这个版本中没有任何垃圾变量。您也没有将精力浪费在对集合的循环处理上，而是使用内置的 contains() 方法来完成工作。您仍然需要告诉程序要做什么 — 检查集合是否包含我们寻找的值 — 但将细节留给底层的方法。

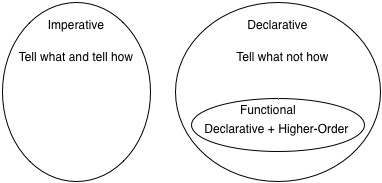
在命令式示例中，您控制着迭代，而且程序完全按照要求来操作。在声明式版本中，您无需关心工作如何完成，只要它完成即可。contains() 的实现可能不同，但只要结果符合预期，您可能就会很开心。花更少的精力获得同样的结果。

训练自己以声明式编程思考，这将大大简化向 Java 中的函数式编程的过渡。为什么呢？因为函数式编程是以声明式为基础而建立的。声明式思考试图逐步从命令式编程过渡到函数式编程。

## 函数式格式

尽管函数式格式的编程始终是声明式的，但简单地使用声明式编程并不等于函数式编程。这是因为函数式编程合并了声明式方法与高阶函数。图 1 直观地展示了命令式、声明式和函数式编程之间的关系。

图 1. 命令式、声明式和函数式编程的联系



**Java 中的高阶函数**

在 Java 中，要将对象传递给方法，在方法内创建对象，并从方法中返回对象。可以对函数执行同样的操作。也就是说，可以将函数传递给方法，在方法内创建函数，并从方法返回函数。

在此上下文中，*方法* 是类的一部分 — 静态或实例 — 但函数对于方法而言是本地函数，不能特意与类或实例关联。可以接收、创建或返回函数的函数或方法被视为*高阶函数*。

## 函数式编程示例

采用新编程的格式需要改变您思考程序的方式。可以通过简单的示例来实践这一过程，并逐步建立更复杂的程序。

清单 3. A Map in the imperative style

|  |
| --- |
| import java.util.\*;    public class UseMap {    public static void main(String[] args) {      Map<String, Integer> pageVisits = new HashMap<>();        String page = "https://agiledeveloper.com";        incrementPageVisit(pageVisits, page);      incrementPageVisit(pageVisits, page);        System.out.println(pageVisits.get(page));    }      public static void incrementPageVisit(  Map<String, Integer> pageVisits, String page) {      if(!pageVisits.containsKey(page)) {         pageVisits.put(page, 0);      }        pageVisits.put(page, pageVisits.get(page) + 1);    }  } |
|  |

在 [清单 3](https://www.ibm.com/developerworks/cn/java/j-java8idioms1/index.html?ca=drs-#清单 3) 中，main() 函数创建了一个 HashMap，其中包含一个网站的页面访问次数。同时，每次访问给定页面，incrementPageVisit() 方法都会增加计数。我们将重点查看此方法。

incrementPageVisit() 方法是使用命令式格式编写的：它的职责是递增给定页面的计数，将该计数存储在 Map 中。该方法不知道给定页面是否有计数，所以它首先会检查是否存在计数。如果不存在，那么它会插入一个 “0” 作为该页面的计数。然后获得该计数，递增它，并将新值存储在 Map 中。

声明式思考要求您将此方法的设计思路从 “如何做” 转变为 “做什么”。当调用方法 incrementPageVisit() 时，您希望将给定页面的计数初始化为 1 或将运行值递增 1。这就是*做什么*。

因为您在执行声明式编程，所以下一步是扫描 JDK 库，查找 Map 接口中可以完成您的目标的方法 — 也就是说，寻找一个知道*如何* 完成给定任务的内置方法。

事实证明，merge() 方法能完美实现您的目标。清单 4 使用新的声明式方法修改来自 [清单 3](https://www.ibm.com/developerworks/cn/java/j-java8idioms1/index.html?ca=drs-#清单 3) 的 incrementPageVisit() 方法。但是，在本例中，并不仅仅通过选择更智能的方法采用更加声明式的格式编程；因为 merge() 是一个高阶函数，所以新代码实际上是一个不错的函数式编程示例：

清单 4. A Map in the functional style

|  |
| --- |
| public static void incrementPageVisit(  Map<String, Integer> pageVisits, String page) {  pageVisits.merge(  page, 1, (oldValue, value) -> oldValue + value);  } |

在清单 4 中，page 作为第一个参数传递给 merge()：该键的值应该更新。第二个参数是给该键分配的初始值，*如果*Map 中不存在该键（在本例中，该值为 “1”）。第三个参数（一个拉姆达表达式）接收 map 中该键对应的值作为其参数，并且将该值作为变量传递给 merge 方法中的第二个参数。 这个拉姆达表达式返回的是它的参数的和，实际上就是递增计数。

比较[清单 4](https://www.ibm.com/developerworks/cn/java/j-java8idioms1/index.html?ca=drs-#清单 4) 中的 incrementPageVisit() 方法中的一行代码与 [清单 3](https://www.ibm.com/developerworks/cn/java/j-java8idioms1/index.html?ca=drs-#清单 3) 中的多行代码。[清单 4](https://www.ibm.com/developerworks/cn/java/j-java8idioms1/index.html?ca=drs-#清单 4) 中的程序是一个函数式编程示例，由此可见声明式考虑问题有助于我们的跳跃性思维。

## 结束语

在 Java 程序中采用函数式方法和语法有许多好处：代码简洁，更富于表达，不易出错，更容易并行化，而且通常比面向对象的代码更容易理解。我们面临的挑战在于将思维方式从命令式编程 — 绝大多数开发人员都熟悉它 — 转变为声明式思考。

尽管函数式编程不那么容易或直观，但您可以通过学习关注您想要程序实现的*目的*而不是关注您希望它执行的*方式*，从而实现思维上的巨大飞跃。通过允许底层函数库管理执行代码，您将逐步直观地了解高阶函数，它们是函数式编程的构建基块。

# 函数组合与集合管道模式

迭代 Java 中集合的函数模式

在开始采用函数式编程风格时，您的程序中会自然而然地出现一些函数设计模式，但您仍需要努力掌握它们。本文将介绍函数组合和集合管道，您可以结合使用这两种函数式模式来迭代代码中的集合。了解这些模式的结构有助于您搭建自己的 Java™ 程序，从而充分利用高阶函数和拉姆达表达式。

## 语句与表达式

如果在代码库中快速查找 for，您可能会惊奇地发现，您的代码中对 for 循环的使用非常频繁。我将这种情形称为 for*重复(hammer)*：只要我们需要循环，似乎就会用到 for。

在 Java 中，for 和 while 都是语句。*语句*执行一个操作，但不会生成任何结果。就本质而言，任何执行有用的操作的语句都会导致数据变化。这是语句表达其效果的唯一方式。而*表达式*则相反：它们可以得出结果而不会导致变化。

在代码中使用语句就像是合作处理一部分工作，但无法在团队成员之间直接转交工作结果。分享结果的唯一方法是将它放在桌面上或架子上，让另一位团队成员可以获得它。表达式的工作更像一条链：当某个人完成一项任务时，他将结果转交给链中的下一个人。

表达式帮助实现了集合管道模式，Martin Fowler 将该模式描述为[运算序列](https://martinfowler.com/articles/collection-pipeline)，会将从一次运算收集的输出提供给下一次运算。尽管面向对象的编程中使用了集合管道模式（您可能在使用对象构建器的代码中看到过它），但它在函数编程中更常见。

函数组合和集合管道模式是两种可协同工作的模式。下一节将使用熟悉的 for 语句解决一个问题。然后将介绍如何使用这两种模式更高效地解决同一个问题。

## 使用语句进行迭代和排序

首先创建一个 Car 类，它包含属性 make、model 和 year：

清单 1. 一个 Car 类

|  |
| --- |
| package agiledeveloper;    public class Car {    private String make;    private String model;    private int year;      public Car(String theMake, String theModel, int yearOfMake) {      make = theMake;      model = theModel;      year = yearOfMake;    }      public String getMake() { return make; }    public String getModel() { return model; }    public int getYear() { return year; }  } |

我们可以添加一个 Car 实例集合，比如：

清单 2. 一个 Car 实例集合

|  |
| --- |
| package agiledeveloper;    import java.util.Collections;  import java.util.Comparator;  import java.util.Arrays;  import java.util.ArrayList;  import java.util.List;  import static java.util.Comparator.comparing;  import static java.util.stream.Collectors.toList;    public class Iterating {    public static List<Car> createCars() {      return Arrays.asList(        new Car("Jeep", "Wrangler", 2011),        new Car("Jeep", "Comanche", 1990),        new Car("Dodge", "Avenger", 2010),        new Car("Buick", "Cascada", 2016),        new Car("Ford", "Focus", 2012),        new Car("Chevrolet", "Geo Metro", 1992)      );    } |

在清单 3 中，我们使用命令式编程来迭代该列表，并获取 2000 年后制造的汽车的名称。然后按年份对这些型号进行升序排序。

清单 3. 使用 ‘for’ 按年份对各种型号进行排序

|  |
| --- |
| public static List<String> getModelsAfter2000UsingFor(  List<Car> cars) {      List<Car> carsSortedByYear = new ArrayList<>();        for(Car car : cars) {        if(car.getYear() > 2000) {          carsSortedByYear.add(car);        }      }        Collections.sort(carsSortedByYear,  new Comparator<Car>() {        public int compare(Car car1, Car car2) {          return new Integer(  car1.getYear()).compareTo(car2.getYear());        }      });        List<String> models = new ArrayList<>();      for(Car car : carsSortedByYear) {        models.add(car.getModel());      }        return models;    } |

如您所见，此代码中有许多循环。首先，getModelsAfter2000UsingFor 方法接受一个汽车列表作为参数。它提取或过滤出 2000 年后制造的汽车，将它们放在一个名为 carsSortedByYear 的新列表中。接下来，按照制造年份对该列表进行升序排序。最后，循环处理列表 carsSortedByYear，以获取型号名称，并在一个列表中返回它们。

能否提高这段代码的效率？我们可以通过一次性对所有汽车排序来消除一个 for 循环，但随后需要对一个更大的对象列表进行排序。

这是汽车列表示例的输出：

清单 4. getModelsAfter2000UsingFor 的输出

|  |
| --- |
| Avenger, Wrangler, Focus, Cascada |

这个排序示例演示了我所称的*语句的效果*。尽管函数和方法通常可用作表达式，但 Collectionssort 方法没有返回结果。因为它被用作语句，所以它改变了作为参数提供的列表。两个 for 循环在迭代时都改变了相应的列表。作为语句，这是这些元素的工作方式。因此，该代码包含不必要的垃圾变量，如图 1 所示。

图 1. 语句的效果



## 使用集合管道进行迭代和排序

在函数编程中，通常会通过一系列更小的模块化函数或运算来对复杂运算进行排序。该系列被称为函数组合（composition of functions, or a function composition）。当一个数据集合流经一个函数组合时，它就变成一个集合管道。函数组合和集合管道是函数式编程中常用的两种设计模式。

无需使用一个很大的 for 重复（ hammer ），依据手头的问题，我们能使用多种专业工具。无需像命令式编程中常见的那样对所有运算都使用语句，函数式编程鼓励使用表达式。表达式没有语句所拥有的改变对象的副作用。像 filter 或 map 这样的表达式也返回一个结果，我们可以将它传递给另一个函数，我们就是通过这种方式创建集合管道的。考虑清单 5 中的代码。

**清**单 5. 集合管道模式中的函数组合

|  |
| --- |
| public static List<String>  getModelsAfter2000UsingPipeline(      List<Car> cars) {      return        cars.stream()            .filter(car -> car.getYear() > 2000)            .sorted(Comparator.comparing(Car::getYear))            .map(Car::getModel)            .collect(toList());   } |

方法 getModelsAfter2000UsingPipeline 生成的结果与 [清单 3](https://www.ibm.com/developerworks/cn/java/j-java8idioms2/index.html?ca=drs-#清单 3) 中的方法 getModelsAfter2000UsingFor 相同，但要注意代码中的不同之处：

函数式代码比命令式代码更简洁。

函数式代码不会表现出明显的易变性，而且使用了更少的垃圾变量。

第二个方法中使用的函数/方法都是有返回值的表达式。将此方法与 [清单 3](https://www.ibm.com/developerworks/cn/java/j-java8idioms2/index.html?ca=drs-#清单 3) 中的 Collections.sort 方法进行对比。

getModelsAfter2000UsingPipeline 使用了集合管道模式，而且非常富于表达。

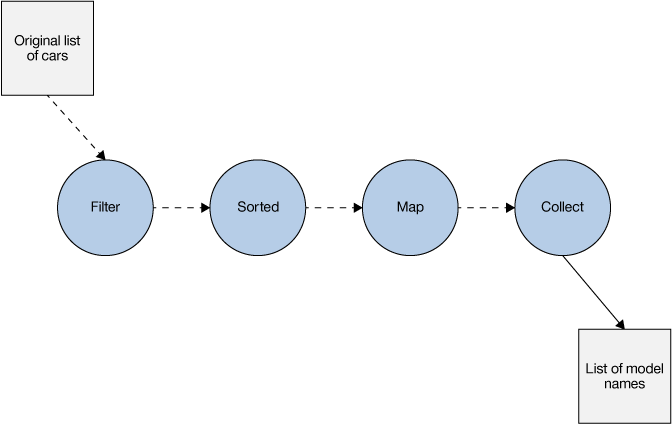
只用了短短几行代码，代码的意图就很明显 — 给定一个汽车集合，过滤或提取仅在 2000 年或以后制造的汽车；然后按年份进行排序，将这些对象映射或转换为它们的型号，最后将结果收集到一个列表中。

[清单 5](https://www.ibm.com/developerworks/cn/java/j-java8idioms2/index.html?ca=drs-#清单 5) 中的代码如此简洁且富于表达，部分原因是使用了方法引用。将一个拉姆达表达式传递给 filter 很有用，因为它可以获取给定对象的 year 属性并将其与 year 2000 进行比较。但是，将拉姆达表达式传递给 map 就没有这么有效。传递给 map 方法的表达式为 car -> car.getModel()，该表达式非常繁琐。该拉姆达表达式仅返回给定对象的某个属性，不执行任何实际计算或运算。我们最好将它替换为一个方法引用。

我们将方法引用 Car::getModel 传递给 map 方法，而不传递拉姆达表达式。类似地，我们将方法引用 Car::getYear 传递给 comparing 方法，而不传递拉姆达表达式 car -> car.getYear()。方法引用简短、简洁且富于表达。最好尽可能地使用它们。

图 2 给出了清单 5 中的集合管道。

图 2. 集合管道的魅力



查看图 2，可以看到 getModelsAfter2000UsingPipeline 函数如何执行集合管道，通过一系列函数转换给定输入。随着数据流经各个函数，Java 8 的惰性计算和函数融合功能（参阅 2014 年的 [*Java 中的函数编程*](https://pragprog.com/book/vsjava8/functional-programming-in-java)）可帮助避免在某些情况下创建中间对象。数据在管道中传输时，函数不会使中间对象变得可见或可用。

## 结束语

在命令式编程中，对于大部分数据处理，通常都会使用 for 和 while 循环。在本文中，您了解了一种在函数式编程中非常流行的替代方法。函数组合是一项简单技术，有助于对模块化函数进行排序，从而创建更复杂的运算。按该序列处理数据时，您就有了一个集合管道。结合使用函数组合和集合管道模式，可以创建复杂的程序，让数据从上游流到下游，并经历一系列转换。