链滴

Kotlin 基础 | 望文生义的 Kotlin 集合操作

作者：lzlyy
原文链接：https://ld246.com/article/1590457035382
来源网站：链滴

有没有那么一种代码，从头到尾读一遍就能清晰的明白语义？就好像在阅读英语文章一样。这篇文章试着用这样望文生义的代码来实现业务需求，剖析 kotlin 语言特性所带来的简洁及其背后原理。知识包括序列，集合操作，主构造方法，可变参数，默认参数，命名参数，for循环，数据类。本着实用主，不会面面俱到地展开知识点所有的细节（这样会很无趣），而是只讲述和实例有关的方面。

该系列每一篇例子用到的知识点会在上一篇的基础上扩充，若遇到不了解的语法也可以移步上一篇查。

业务需求如下：假设现在需要基于学生列表过滤出所有学生的选修课（课时数 < 70），输出时按课数升序排列，课时数相等的再按课程名字母序排列，并写课程名的第一个字母。

数据类

先得声明数据实体类，java的代码如下：

课程实体类

```
public class Course {
    private String name ;
    private int period ;
    private boolean isMust;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getPeriod() {
        return period;
    }

    public void setPeriod(int period) {
        this.period = period;
    }

    public boolean isMust() {
        return isMust;
    }

    public void setMust(boolean must) {
        isMust = must;
    }

    @Override
    public String toString() {
        return "Course{" +
                "name=  '" + name + '\'' +
                ", period=" + period +
                ", isMust=" + isMust +
                '}'  ;
```

```
    }
}
```

学生实体类

```
public class Student {
    private String name;
    private int age;
    private boolean isMale ;
    private List<Course> courses ;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public boolean isMale() {
        return isMale;
    }

    public void setMale(boolean male) {
        isMale = male;
    }

    public List<Course> getCourses() {
        return courses;
    }

    public void setCourses(List<Course> courses) {
        this.courses = courses;
    }

    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", isMale=" + isMale +
                ", courses=" + courses +
                '}'  ';
    }
}
```

代码略长，其实关键信息只有两个：类名+属性，其余部分都是模版代码，所以 kotlin 将数据类的定缩减成一行代码：

```
data class Course constructor(var name: String, var period: Int, var isMust: Boolean = false)

data class Student constructor(var name: String, var age: Int, var isMale: Boolean, var courses: ist<Course> = listOf())
```

- data是保留字，用于修饰一个类，表明该类只包含数据而不包含行为，即是 java 中的 Bean 类。
- 类的声明格式如下：

```
修饰词 class 类名 constructor(主构造函数参数列表){类体}
```

- class保留字用于声明类。
- constructor保留字用于声明类的**主构造方法**，它相当于把 java 中的类声明和构造函数声明合并到一行，下面的两段代码是完全等价的：

```
//java
class A(){
    private int i
    A(int i){
        this.i = i;
    }
}

//kotlin
class A constructor(var i: Int)
```

- **主构造方法**显示地声明了类的成员属性和其数据类型，这里包含的隐藏信息是，当构造 A的实例时传入构造方法的 Int 值会被赋值给成员 i。
- 除了简单地为成员赋值，主构造方法不包含其他任何逻辑。（当需要特殊的初始化逻辑时需要使用的方法，以后会讲到~）
- 当没有可见性修饰符修饰 **主构造方法**时，可以省去 constructor保留字，所以上面的数据类可以化成：

```
data class Course(var name: String, var period: Int, var isMust: Boolean = false)

data class Student(var name: String, var age: Int, var isMale: Boolean, var courses: List<Cours > = listOf())
```

这里还展示了一种在 java 中不支持的特性：**参数默认值**，Cours类的 isMust属性的默认值是 fals，减少了重载构造函数的数量，因为在 java 中只能通过重载来实现：

```
public Course{
    public Course(String name,int period,boolean isMust){
        this.name = name;
        this.period = period;
        this.isMust = isMust;
    }

    public Course(String name,int period){
        return Course(name,period,false);
    }
}
```

在简简单单的一句类声明的背后，编译器会自动为我们创建所有我们需要的方法，包括：

- setter() 和 getter()
- equals() 和 hashCode()
- toString()
- copy()

其中 copy()会基于对象现有属性值构建一个新对象。

构建集合

有了数据实体类后，就可以构建数据集合了，让我们来构建一个包含4个学生的列表，java 代码如下（**实直接跳过这段代码也是不错的选择，因为它很冗长而且可读性差**）：

```
Student student1 = new Student();
student1.setName("taylor");
student1.setAge(33);
student1.setMale(false);
List<Course> courses1 = new ArrayList<>();
Course course1 = new Course();
course1.setName("pysics");
course1.setPeriod(50);
course1.setMust(false);
Course course2 = new Course();
course2.setName("chemistry");
course2.setPeriod(78);
courses1.add(course1);
courses1.add(course2) ;
student1.setCourses(courses1);

Student student2 = new Student();
student2.setName("milo");
student2.setAge(20);
student2.setMale(false);
List<Course> courses2 = new ArrayList<>();
Course course3 = new Course();
course3.setName("computer");
course3.setPeriod(50);
course3.setMust(true);
student2.setCourses(courses2);

List<Student> students = new ArrayList<>();
students.add(student2);
students.add(student1);
...
```

我只写了2个学生构建代码，不想再写下去了。。。你能不能一眼看出它到底在构建啥吗？

还是看看 kotlin 是怎么玩的吧：

```
val students = listOf(
    Student("taylor", 33, false, listOf(Course("physics", 50), Course("chemistry", 78))),
```

```
    Student("milo", 20, false, listOf(Course("computer", 50, true))),
    Student("lili", 40, true, listOf(Course("chemistry", 78), Course("science", 50))),
    Student("meto", 10, false, listOf(Course("mathematics", 48), Course("computer", 50, true)))
)
```

就算是第一次接触 kotlin ，一定也看懂这是在干嘛。

- 得益于 **参数默认值**，对于同一个 Cours构造函数，可传入2个参数 Course("physics", 50)，也可入3个参数 Course("computer", 50, true)
- listOf()是 kotlin 标准库中的方法，这个方法极大简化了构建集合的代码，看下它的源码：

```
public fun <T> listOf(vararg elements: T): List<T> = if (elements.size > 0) elements.asList() els
 emptyList()
```

- vararg保留字用于修饰**可变参数**，表示这个该函数可以接收任意数量的该类参数。
- listOf()的返回值是 kotlin 中的 List类型。

一眼看去，我们就能知道这段代码构建了一个列表，列表中构建了4个学生实例，在构建学生实例的时构建了一系列课程实例。

但是构建学生时，传入的布尔值是什么语义？猜测可能是年龄，在 IDE 跳转功能的帮助下，可以方便到 Student定义处确认一下。但如果在网页端进行 Code Review 时就没有这么好的条件了。

有什么办法在方法调用处就指明参数的语义？

命名参数功能就是为此而生，上面的代码还可以这样写：

```
val students = listOf(
    Student("taylor", 33, isMale = false, courses = listOf(Course("physics", 50), Course("chemist
y", 78))),
    Student("milo", 20, isMale = false, courses = listOf(Course("computer", 50, true))),
    Student("lili", 40, isMale = true, courses = listOf(Course("chemistry", 78), Course("science",
0))),
    Student("meto", 10, isMale = false, courses = listOf(Course("mathematics", 48), Course("co
puter", 50, true)))
)
```

可以在参数前通过加 变量名 =的方式来显示指明参数语义，同时这对变量的命名也提出了更高的要求。

作为程序员的我们，绝大部分时间不是在写而是在读别人或自己的代码。就好像语文阅卷老师要读大作文一样，如果字迹潦草，段落不清晰，就是在给自己给老师添麻烦。同样的，达意的命名，一致的进，语义清晰的调用，让自己和同事赏心悦目。（这也是 kotlin 为啥能提高产生效率的原因，因为它简洁，更可读）

操纵集合

下一个步骤是操纵集合，直接上 kotlin ：

```
val friends = students
        .flatMap { it.courses }
        .toSet()
        .filter { it.period < 70 && !it.isMust }
        .map {
            it.apply {
```

```
            name = name.replace(name.first(), name.first().toUpperCase())
        }
    }
    .sortedWith(compareBy({ it.period }, { it.name }))
```

扫了一遍，在很多陌生函数里面有一个上篇讲解过的 apply()，它做的事情是将集合中的每个元素中的am属性的第一个字符换成大写。

在 java 中（8.0以前），为了操纵集合元素，必然要用 for循环遍历集合。但在上面的代码中，没有现类似的遍历操作，那 kotlin 是如何获取集合中元素的?

map()

kotlin 标准库中预定了很多集合操纵方法，上面用到的 map()就是其中一个，它的源码如下：

```
public inline fun <T, R> Iterable<T>.map(transform: (T) -> R): List<R> {
    return mapTo(ArrayList<R>(collectionSizeOrDefault(10)), transform)
}
```

map()是一个 Iterabl类的扩展函数，这个类表示一个可以被迭代的对象，Collection和 List都是继承它：

```
/**
 * Classes that inherit from this interface can be represented as a sequence of elements that c
n
 * be iterated over.
 * @param T the type of element being iterated over. The iterator is covariant on its element t
pe.
 */
public interface Iterable<out T> {
    /**
     * Returns an iterator over the elements of this object.
     */
    public operator fun iterator(): Iterator<T>
}

public interface Collection<out E> : Iterable<E> {
    ...
}

public interface List<out E> : Collection<E> {
    ...
}
```

map()内会新建一个 ArrayList类型的集合（它是一个中间临时集合）并传给 mapTo()

```
public inline fun <T, R, C : MutableCollection<in R>> Iterable<T>.mapTo(destination: C, trans
orm: (T) -> R): C {
    for (item in this)
        destination.add(transform(item))
    return destination
}
```

这里出现了一个熟悉的保留字 for，它 in搭配后和 java 中的 for-each语义类似。

原来 map()内部使用了 for循环遍历源集合，并在每个元素上应用了 transform这个变换，最后将变后的元素加入临时集合中并将其返回。

所以 map()函数的语义是：**在集合的每一个元素上应用一个自定义的变换**

filter()

在 map()函数前调用了 filter()，源码如下：

```
public inline fun <T> Iterable<T>.filter(predicate: (T) -> Boolean): List<T> {
    return filterTo(ArrayList<T>(), predicate)
}

public inline fun <T, C : MutableCollection<in T>> Iterable<T>.filterTo(destination: C, predica
e: (T) -> Boolean): C {
    for (element in this) if (predicate(element)) destination.add(element)
    return destination
}
```

类似的，它也会构建一个临时集合来暂存运算的中间结果，在遍历源集合的同时应用条件判断 predicat当满足条件时才将源集合元素加入到临时集合。

所以 filter()的语义是：**只保留满足条件的集合元素**

toSet()

filter()之前调用是 toSet()：

```
public fun <T> Iterable<T>.toSet(): Set<T> {
    if (this is Collection) {
        return when (size) {
            0 -> emptySet()
            1 -> setOf(if (this is List) this[0] else iterator().next())
            else -> toCollection(LinkedHashSet<T>(mapCapacity(size)))
        }
    }
    return toCollection(LinkedHashSet<T>()).optimizeReadOnlySet()
}

public fun <T, C : MutableCollection<in T>> Iterable<T>.toCollection(destination: C): C {
    for (item in this) {
        //重复元素会添加失败
        destination.add(item)
    }
    return destination
}
```

遍历源集合的同时借助 LinkedHashSet来实现元素的唯一性。

所以 toSet()的语义是：**将集合元素去重**

flatMap()

在调用链的最开始，调用的是 flatMap()：

```
public inline fun <T, R> Iterable<T>.flatMap(transform: (T) -> Iterable<R>): List<R> {
    return flatMapTo(ArrayList<R>(), transform)
}

public inline fun <T, R, C : MutableCollection<in R>> Iterable<T>.flatMapTo(destination: C, tr
nsform: (T) -> Iterable<R>): C {
    for (element in this) {
        val list = transform(element)
        destination.addAll(list)
    }
    return destination
}
```

flatMap()的源码和 map()非常相似，唯一的区别是，transform这个变换的结果是一个集合类型，然会把该集合整个加入到临时集合。

flatMap()做了两件事情：先对源集合中每个元素做变换（变换结果是另一个集合），然后把多个集合并成一个集合。这样的操作非常适用于集合中套集合的数据结构，就好像本例中的学生实例存放在生列表中，而每个学生实例中包含课程列表。通过先变换后平铺的操作可以方便地将学生列表中的所课程平铺开来。

所以 flatMap()的语义是：**将嵌套集合中的内层集合铺开**

asSequence()

因为每个操纵集合的函数都会新建一个临时集合以存放中间结果。

为了更好的性能，有没有什么办法去掉临时集合的创建?

序列就是为此而生的，用序列改写上面的代码：

```
val friends = students.asSequence()
        .flatMap { it.courses.asSequence() }
        .filter { it.period < 70 && !it.isMust }
        .map {
            it.apply {
                name = name.replace(name.first(), name.first().toUpperCase())
            }
        }
        .sortedWith(compareBy({ it.period }, { it.name }))
        .toSet()
```

通过调用 asSequence()将原本的集合转化成一个序列，序列将对集合元素的操作分为两类：

1. 中间操作

2. 末端操作

从返回值上看，中间操作返回的另一个序列，而末端操作返回的是一个集合（toSet()就是末端操作）。

从执行时机上看，中间操作都是惰性的，也就说中间操作都会被推迟执行。而末端操作触发执行了所被推迟的中间操作。所以将 toSet()移动到了末尾。

序列还会改变中间操作的执行顺序，如果不用序列，n 个中间操作就需要遍历集合 n 遍，每一遍应用个操作，使用序列之后，只需要遍历集合 1 遍，在每个元素上一下子应用所有的中间操作。

如果用 java 实现上述集合操作的话，需要定义一个不是太简单的算法，定神分析一番才能明白业务求，而 kotlin 的代码就好像把需求翻译成了英语，顺着读完代码就能明白语义。这种 **“望文生义”** 效果，真是 java 不能比拟的。

知识点总结

- 通过 data关键词配合主构造函数，kotlin 可以用一行代码声明数据类。
- 主构造方法是一个用于为类属性赋初始值的构造方法。它通过 constructor保留字和类头声明在一行。
- 保留字 vararg用于声明可变参数，带有可变参数的方法可以接收任意个数的参数。
- 可以通过 =在声明方法时为参数设置默认值，以减少重载函数。
- 可以通过 变量名 =语法在方法调用的时候添加命名参数，增加方法调用的可读性。
- kotlin 标准库预定义了很多处理集合的方法，其中
  - filter()的语义是：只保留满足条件的集合元素
  - toSet()的语义是：将集合元素去重
  - flatMap()的语义是：将嵌套集合中的内层集合铺开
  - map()函数的语义是：在集合的每一个元素上应用一个自定义的变换
  - asSequence()用于将一连串集合操作变成序列，以提升集合操作性能。