# Android软件部Kotlin语言培训大纲

## 前言

Kotlin在2017年成为Android的官方第一语言，到现在已经三年了。随着时间的推移可以发现越来越多的库，还有Google原生的Demo中，都已经逐渐的放弃Java语言而改用Kotlin实现了，其目的很明显就是要扶植Kotlin。另外随着Kotlin语言的优化和升级，越来越多的新特性加入到了Demo和三方代码中，你会发现当带着Java的知识阅读Kotlin代码时越来越吃力。所以为了跟上Google的脚步，部门开始组织系统性的培训，帮助大家学习Kotlin。

## 目标

在全员80%掌握Kotlin后，我们会在未来的新项目中，转变为使用Kotlin进行实现。

因为在实际业务开发中，如果不使用新的语言一个是会遗忘，一个是在使用和设计的时候叫不准，这都极大的影响了我们的学习成果。

最后附带一些学习资料给大家：

Kotlin中文网：<https://www.kotlincn.net/>

Kotlin Jetpack 实战： <https://google.blog.csdn.net/article/details/113009301>

Kotlin 极简教程：

Kotlin项目推荐:Wan安卓

注：Kotlin 中文网是标准教程，更新及时，但存在翻译问题，如果有读不懂的知识点建议阅读Kotlin 极简教程。Kotlin 极简教程有一些内容可能更新不及时，或者缺少内容，此时以Kotlin 中文网为标准。

# 一、环境

在培训Kotlin语言期间使用IDEA作为代码展示和实操编译器。

其尽量使用2020.1.1版之后的版本。

关于版权，可以使用试用30天，或者淘宝买个认证。

培训期间Kotlin版本请尽量自动更新至最新，保证新特性可以被及时使用和展示。

在Kotlin语言常用方法，使用场景，常用语法糖等介绍完毕后，切换回Android Studio 进行实战培训展示。

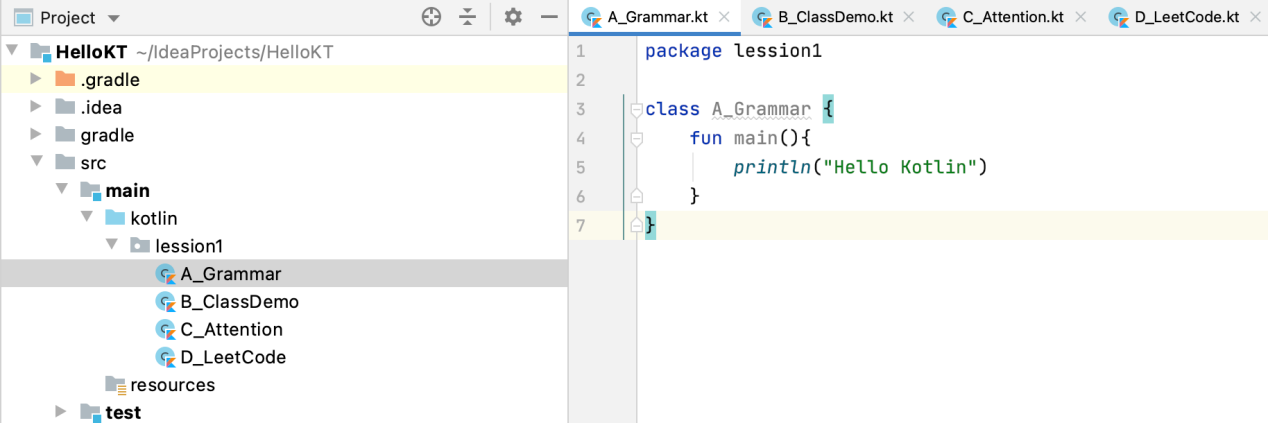
同样Android Studio的Kotlin 版本也请自动更新至最新。

# 二、培训建议

整个Kotlin的语法培训由于内容较多，所以会成多节课，多个方向进行培训。在培训期间，如果选定了一次课程的方向，为了更好的理解和展示语言特性，可以参考下面几个维度进行培训内容的展示。

1. 语法特性展示
2. 经典用法或案例
3. 需要注意点、易错点
4. 结合LeetCode 进行实战展示

关于培训的代码，请尽量标准。可以按照下面的模式进行分包。培训后的代码会按照课程顺序进行合并，作为后续参考手册。如果语法部分展示语法过多也可以创建语法的分包，然后语法按照顺序分类。其他单元类似。



# 三、课程大纲

## 3.1 环境与基本语法

### 3.1.1 IDEA部署Kotlin环境

### 3.1.2基本语法介绍

此部分作为Kotlin的大体语法展示单元，主要是展示Kotlin在常规编程中的使用方式。此部分讲解时需要体现特性和重点，部分细节可以放到后面单元专项中进行讲解。

此部分需要主要的知识点为：

空值与null检测

当某个变量的值可以为 null 的时候，必须在声明处的类型后添加 ? 来标识该引用可为空。

循环和区间

重点介绍when的使用。此部分语法较为灵活，需要展示运行结果。

### 3.1.3基本数据类型

此处注意应在注释中给出，数字类型的字节数与对应十进制范围。

另外需要注意说明数字类型间的转换-显式转换/隐式转换。

关于0/0与NaN的介绍。

## 3.2 习惯用法-类和继承

### 3.2.1 控制流与返回跳转

此部分作为上次培训的复习部分，即需要对上节课的内容进行回顾，也要做到一定的延伸。此部分需要完整的介绍控制流的相关内容。在介绍完控制流后进行返回和跳转的讲解。

### 3.2.2 习惯用法

此部分主要是介绍Kotlin的语法糖单元。

可以将用例粘贴到IDE中直接讲解代码。

关于单例的创建可以多展示几种模式。

### 3.2.3 类和继承

此处需要尽可能详尽的展示，类无父类时的构造形式。当有父类时的构造形式。另外多构造器时，构造方法引用构造方法也应详尽展示。

## 3.3 类与对象一

### 3.3.1 属性与字段

此部分应要着重介绍在Kotlin中非空属性和可空属性的区别。关于Setter和Getter部分除了展示标准的Getter Setter的写法外，还要讲述幕后字段和幕后属性的出现原因。

### 3.3.2 接口与函数式接口

着重讲解覆盖冲突即可。函数式接口，只需介绍是为后面Lambda表达式编程作准备即可。

### 3.3.3 可见性说明

这里要阐述和Java的不同，即类名和文件名不再有关系。方法也不是必须声明在某个类中。

另外顶层同名方法，可以通过修改报名来解决冲突。类似命名空间。

### 3.3.4 拓展函数

给出参考：<https://www.jianshu.com/p/ab1fb42ca85d> 这篇帖子写的较为详细且易读懂。

此部分为本次培训的重点单元，可以使用帖子中的素材作为展示案例。

同时需要提醒：拓展是静态解析的。

## 3.4 类与对象二

### 3.4.1数据类

主要展示数据类的toString()、 equals()、 hashCode() 以及 copy()等方法。

### 3.4.2 密封类

展示代码介绍清楚即可。

### 3.4.3 范型

此处可以简略介绍型变这个概念，重点突出范型的语法情况即可，如 星投影。

### 3.4.4 嵌套类与内部类

分别介绍嵌套类和内部类的写法后，需要阐述嵌套类和内部类的区别。

### 3.4.5 枚举类

正常讲述即可。

### 3.4.6 对象

此部分请重点阐述companion和object的作用。对于伴生对象部分，可以多给些实例。

最后输出对象表达式和对象声明的语义差异。

### 3.4.7 类型别名

简述即可，简单来说就是替换，短替换长。

### 3.4.8 内联类

简述即可，内联类的主要特性，它灵感来源于 “inline” 这个名称：类的数据被 “内联”到该类使用的地方。

一般来说，只要将内联类用作另一种类型，它们就会被装箱。

### 3.4.9 委托

注意区分委托方法委托属性即可。

### 3.4.10 委托属性

展示属性委托（想讲属性委托要求），延迟属性，可观察属性，属性委托给另外一个属性，将属性储存在映射中的代码即可。

在介绍完类后需要介绍 包与倒入

## 3.5函数与lambda表达式

### 3.5.1函数

默认参数这里要介绍和无默认参数方法的区别。如果在默认参数之后的最后一个参数是 [lambda 表达式](https://www.kotlincn.net/docs/reference/lambdas.html" \l "lambda-%E8%A1%A8%E8%BE%BE%E5%BC%8F%E8%AF%AD%E6%B3%95)，那么它既可以作为具名参数在括号内传入，也可以在[括号外](https://www.kotlincn.net/docs/reference/lambdas.html" \l "passing-a-lambda-to-the-last-parameter)传入,括号外传入可以省略参数的括号！

介绍Unit概念。

介绍varargs 的概念与原理

中缀表示法简单介绍即可 注意优先级

对于函数类型我们应该心中有数：

按作用域划分：局部函数 成员函数 顶层函数

函数分类

1. 单表达式函数
2. 范型函数
3. 内联函数
4. 扩展函数
5. 尾递归函数

### 3.5.2 高阶函数与Lambda表达式

介绍高阶函数的概念，主要是阐述高阶函数和普通函数的区别。

高阶函数是将函数用作参数或返回值的函数。

在介绍此部分之前先简述Lambda语法是怎么回事。另外先交代下方法引用的概念。

使用fold对集合进行自定义计算，以此为高阶函数概念的切入点。

下面对一下代码进行解析

fun <T, R> Collection<T>.fold(initial: R, combine: (acc: R, nextElement: T) -> R): R {

var accumulator: R = initial

for (element: T in this) {

accumulator = combine(accumulator, element)

}

return accumulator

}

1. 这是一个扩展函数，此扩展函数涉及两个范型,返回类型是范型R的类型。
2. 范型T是集合元素的参数化类型，R是输入的操作类型
3. 扩展的方法名为fold,参数有两个initial是输入，combine是一个函数参数
4. 这个函数参数参数的 模版为:(acc: R, nextElement: T) -> R

可以理解为，对此函数参数的规定，不按规定的传不进来

规定如下:此函数需要有两个参数一个类型是R,一个类型是T然后需要返回值，返回值是R

5.方法体内的代码就是正常的操作了，即先做一个赋值，然后遍历集合当前集合this。

然后进行操作，即调用我们传入的函数参数。遍历完成后返回结果。

另外此部分请尽量展示用例，并做出解释。

如：

如需将函数类型指定为[可空](https://www.kotlincn.net/docs/reference/null-safety.html" \l "%E5%8F%AF%E7%A9%BA%E7%B1%BB%E5%9E%8B%E4%B8%8E%E9%9D%9E%E7%A9%BA%E7%B1%BB%E5%9E%8B)，请使用圆括号：((Int, Int) -> Int)?。

函数类型可以使用圆括号进行接合：(Int) -> ((Int) -> Unit)

箭头表示法是右结合的，(Int) -> (Int) -> Unit 与前述示例等价，但不等于 ((Int) -> (Int)) -> Unit。

函数类型实例化

带与不带接收者的函数类型非字面值可以互换，其中接收者可以替代第一个参数，反之亦然。例如，(A, B) -> C 类型的值可以传给或赋值给期待 A.(B) -> C 的地方。

class HTML {

fun body() { …… }

}

fun html(init: HTML.() -> Unit): HTML {

val html = HTML() // 创建接收者对象

html.init() // 将该接收者对象传给该 lambda

return html

}

html { // 带接收者的 lambda 由此开始

body() // 调用该接收者对象的一个方法

}

### 3.5.3 内联函数

首先要阐述，函数也是对象，创建对象会产生性能消耗。所以我们使用内联函数的方式，直接通过编译将代码拿过来，这样就不创建对象了，降低了性能消耗。

可以内联的 lambda 表达式只能在内联函数内部调用或者作为可内联的参数传递， 但是 noinline 的可以以任何我们喜欢的方式操作：存储在字段中、传送它等等。

非局部返回，成员函数结合lambda或局部函数进行介绍。

具体化的类型参数 reified ，让我们的范型操作更优雅。

## 3.6集合

### 3.6.1 集合概述

#### 3.6.1.1 集合类是什么

在讲 Kotlin 的集合类之前，为了更加深刻理解为什么要有集合类，以及集合类到底是怎么一回事，让我们先来简单回顾一下编程的本质：

数据结构 + 算法 （信息的逻辑结构及其基本操作）

算法与数据的结构密切相关，算法无不依附于具体的数据结构，数据结构直接关系到算法的选择和效率。通常情况下，设计良好的数据结构可以大大简化算法的实现复杂度，同时可以提升存储效率。数据结构往往同高效的检索算法和索引技术相关。

数据结构主要解决以下三个问题：

1.数据元素之间的逻辑关系。

这些逻辑关系有：**集合、线性结构、树形结构、图形结构等**。

数据的物理结构。

1. 数据的逻辑结构在计算机存储空间的存放形式。数据的物理结构是数据结构在计算机中的映射。

其具体实现的方法有： **顺序（Sequence）、链接（Link）、索引（Index）、散列（Hash）等形式。**

其中，顺序存储结构和链式存储结构是我们常用的两种存储结构。

顺序存储是使用元素在存储器中的相对位置来表示数据元素之间的逻辑关系；

链式存储使用指示元素存储位置的指针（pointer）来表示数据元素之间的逻辑关系。

3.数据的处理运算。

我们知道类的本质就是，对象及其关系的抽象（abstraction）。一个类通常有属性（数据结构）和行为（算法）。使用OO范式编程的大致过程为：

划分对象 → 抽象类 → 将类组织成为层次化结构(继承和合成) → 用类与实例进行设计和实现

#### 3.6.1.2 Kotlin 集合类简介

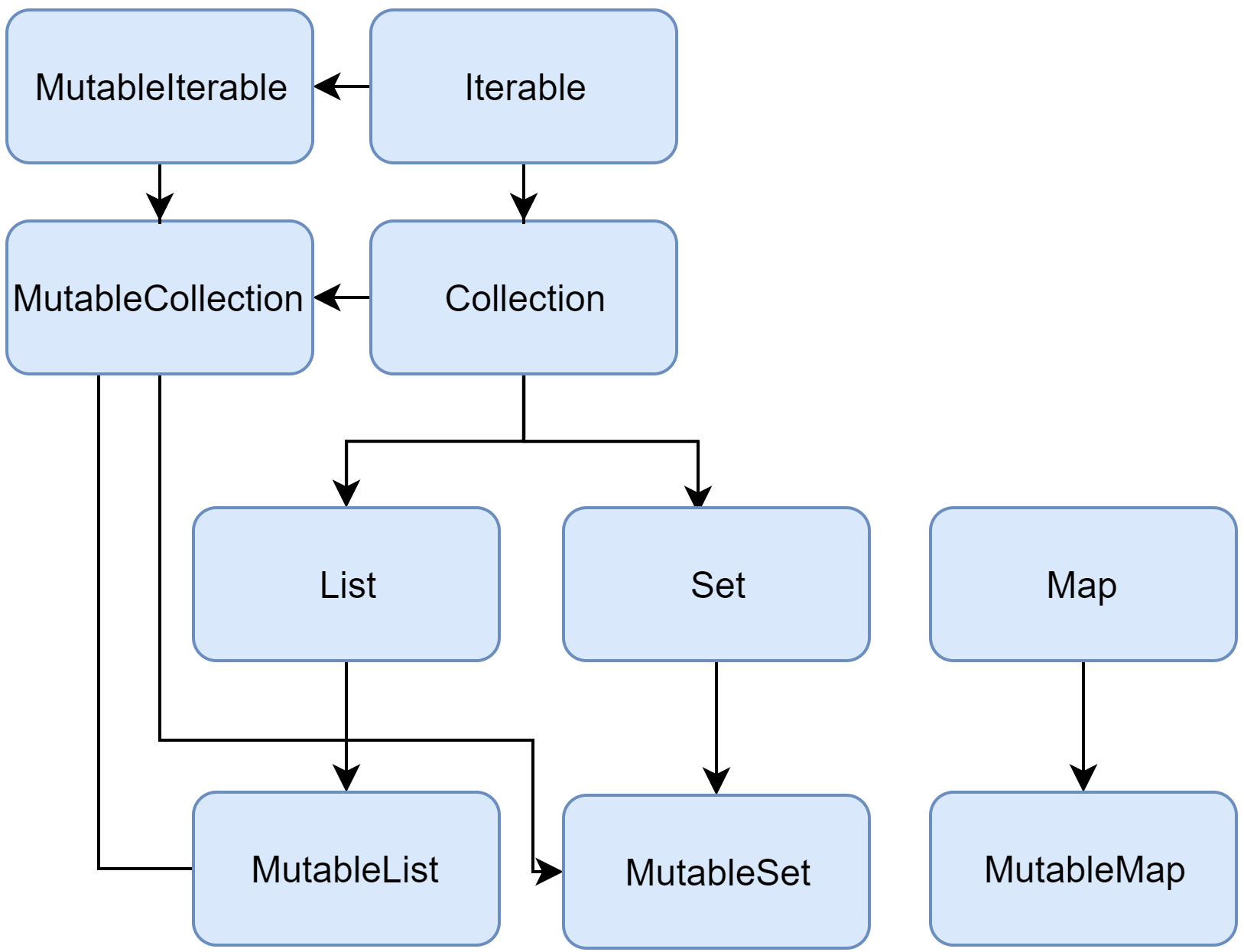
集合类存放的都是对象的引用，而非对象本身，我们通常说的集合中的对象指的是集合中对象的引用（reference)。

Kotlin的集合类分为：**可变集合类（Mutable）与不可变集合类（Immutable）**

一个 只读 接口，提供访问集合元素的操作。

一个 可变 接口，通过写操作扩展相应的只读接口：添加、删除和更新其元素。

集合类型主要有3种：**list(列表）、set(集）、和 map(映射)**



其中，Iterator是所有容器类Collection的迭代器。迭代器（Iterator）模式，又叫做游标（Cursor）模式。GOF给出的定义为：提供一种方法访问一个容器对象中各个元素，而又不需暴露该对象的内部细节。 从定义可见，迭代器模式是为容器而生。

(1)列表

列表的主要特征是其对象以线性方式存储，没有特定顺序，只有一个开头和一个结尾，当然，它与根本没有顺序的集是不同的。

列表在数据结构中可表现为：数组和向量、链表、堆栈、队列等。

(2)集

集（set）是最简单的一种集合，它的对象不按特定方式排序，只是简单的把对象加入集合中，就像往口袋里放东西。

对集中成员的访问和操作是通过集中对象的引用进行的，**所以集中不能有重复对象**。

集也有多种变体，可以实现排序等功能，如TreeSet，它把对象添加到集中的操作将变为按照某种比较规则将其插入到有序的对象序列中。它实现的是SortedSet接口，也就是加入了对象比较的方法。通过对集中的对象迭代，我们可以得到一个升序的对象集合。

(3)映射

映射与集或列表有明显区别，映射中每个项都是成对的。映射中存储的每个对象都有一个相关的关键字（Key）对象，关键字决定了 对象在映射中的存储位置，检索对象时必须提供相应的关键字，就像在字典中查单词一样。关键字应该是唯一的。

关键字本身并不能决定对象的存储位置，它需要对过一种散列(hashing)技术来处理，产生一个被称作散列码(hash code)的整数值，

散列码通常用作一个偏置量，该偏置量是相对于分配给映射的内存区域起始位置的，由此确定关键字/对象对的存储位置。理想情况 下，散列处理应该产生给定范围内均匀分布的值，而且每个关键字应得到不同的散列码。

List 是一个有序集合，可通过索引（反映元素位置的整数）访问元素。元素可以在 list 中出现多次。列表的一个示例是一句话：有一组字、这些字的顺序很重要并且字可以重复。

Set 是唯一元素的集合。它反映了集合（set）的数学抽象：一组无重复的对象。一般来说 set 中元素的顺序并不重要。例如，字母表是字母的集合（set）。

Map（或者字典）是一组键值对。键是唯一的，每个键都刚好映射到一个值。值可以重复。map 对于存储对象之间的逻辑连接非常有用，例如，员工的 ID 与员工的位置。

#### 3.6.1.3 Collection

[Collection<T>](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-collection/index.html) 是集合层次结构的根。此接口表示一个只读集合的共同行为：检索大小、检测是否为成员等等。 Collection 继承自 Iterable <T> 接口，它定义了迭代元素的操作。可以使用 Collection 作为适用于不同集合类型的函数的参数。对于更具体的情况，请使用 Collection 的继承者： [List](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-list/index.html) 与 [Set](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-set/index.html)。

[MutableCollection](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-collection/index.html) 是一个具有写操作的 Collection 接口，例如 add 以及 remove。

#### 3.6.1.4 List

List接口继承于Collection接口，元素以线性方式存储，集合中可以存放重复对象。Kotlin的List分为：不可变集合类List（ReadOnly, Immutable）和可变集合类MutableList（Read&Write, Mutable）。

[List<T>](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-list/index.html) 以指定的顺序存储元素，并提供使用索引访问元素的方法。索引从 0 开始 – 第一个元素的索引 – 直到 最后一个元素的索引 即 (list.size - 1)。

#### 3.6.1.5 Set

[Set<T>](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-set/index.html) 存储唯一的元素；它们的顺序通常是未定义的。null 元素也是唯一的：一个 Set 只能包含一个 null。当两个 set 具有相同的大小并且对于一个 set 中的每个元素都能在另一个 set 中存在相同元素，则两个 set 相等。

[MutableSet](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-set/index.html) 是一个带有来自 MutableCollection 的写操作接口的 Set。

Set的默认实现 - [LinkedHashSet](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-linked-hash-set/index.html) – 保留元素插入的顺序。 因此，依赖于顺序的函数，例如 first() 或 last()，会在这些 set 上返回可预测的结果。

#### 3.6.1.6 Map

[Map<K, V>](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-map/index.html) 不是 Collection 接口的继承者；但是它也是 Kotlin 的一种集合类型。 Map 存储 键-值 对（或 条目）；键是唯一的，但是不同的键可以与相同的值配对。Map 接口提供特定的函数进行通过键访问值、搜索键和值等操作。

无论键值对的顺序如何，包含相同键值对的两个 Map 是相等的。

[MutableMap](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-map/index.html) 是一个具有写操作的 Map 接口，可以使用该接口添加一个新的键值对或更新给定键的值。

#### 3.6.1.7 Pair

补充一下Pair的知识，Pair是Map的基本组成元素。

在后面的一些谓词操作时，涉及到一些Pair的隐士缩写，可能产生阅读上的疑惑或者困难。所以这里补充一下。

### 3.6.2 构造集合

创建集合的最常用方法是使用标准库函数 [listOf<T>()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/list-of.html)、[setOf<T>()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/set-of.html)、[mutableListOf<T>()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/mutable-list-of.html)、[mutableSetOf<T>()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/mutable-set-of.html)。

**如果以逗号分隔的集合元素列表作为参数，编译器会自动检测元素类型。创建空集合时，须明确指定类型。**

同样的，Map 也有这样的函数 [mapOf()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/map-of.html) 与 [mutableMapOf()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/mutable-map-of.html)。映射的键和值作为 Pair 对象传递（通常使用中缀函数 to 创建）。

注意，to 符号创建了一个短时存活的 Pair 对象，因此建议仅在性能不重要时才使用它。 为避免过多的内存使用，请使用其他方法。例如，可以创建可写 Map 并使用写入操作填充它。 [apply()](https://www.kotlincn.net/docs/reference/scope-functions.html" \l "apply) 函数可以帮助保持初始化流畅。

val numbersMap = mutableMapOf<String, String>()

.apply { this["one"] = "1"; this["two"] = "2" }

空集合：还有用于创建没有任何元素的集合的函数：[emptyList()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/empty-list.html)、[emptySet()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/empty-set.html) 与 [emptyMap()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/empty-map.html)。 创建空集合时，应指定集合将包含的元素类型。

val empty = emptyList<String>()

对于 List，有一个接受 List 的大小与初始化函数的构造函数，该初始化函数根据索引定义元素的值。

// 如果你想操作这个集合，应使用 MutableList

val doubled = List(3, { it \* 2 })

println(doubled)

**要创建具体类型的集合**，例如 ArrayList 或 LinkedList，可以使用这些类型的构造函数。 类似的构造函数对于 Set 与 Map 的各实现中均有提供。

val linkedList = LinkedList<String>(listOf("one", "two", "three"))

val presizedSet = HashSet<Int>(32)

要创建与现有集合具有相同元素的集合，可以使用复制操作。标准库中的集合复制操作创建了具有相同元素引用的 浅 复制集合。 因此，对集合元素所做的更改会反映在其所有副本中。

在特定时刻通过集合复制函数，**例如[toList()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/to-list.html)、[toMutableList()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/to-mutable-list.html)、[toSet()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/to-set.html) 等等。创建了集合的快照。** 结果是创建了一个具有相同元素的新集合 如果在源集合中添加或删除元素，则不会影响副本。副本也可以独立于源集合进行更改

### 3.6.3 迭代器

#### 3.6.3.1 常规迭代的写法

对于遍历集合元素， Kotlin 标准库支持 迭代器 的常用机制——**对象可按顺序提供对元素的访问权限，而不会暴露集合的底层结构**。 当需要逐个处理集合的所有元素（例如打印值或对其进行类似更新）时，迭代器非常有用。

[Iterable<T>](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-iterable/index.html) 接口的继承者（包括 Set 与 List）可以通过调用 [iterator()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-iterable/iterator.html) 函数获得迭代器。

**一旦获得迭代器它就指向集合的第一个元素；调用 [next()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-iterator/next.html) 函数将返回此元素，并将迭代器指向下一个元素（如果下一个元素存在）。 一旦迭代器通过了最后一个元素，它就不能再用于检索元素；也无法重新指向到以前的任何位置。要再次遍历集合，请创建一个新的迭代器。**

val numbers = listOf("one", "two", "three", "four")

val numbersIterator = numbers.iterator()

while (numbersIterator.hasNext()) {

println(numbersIterator.next())

}

遍历 Iterable 集合的另一种方法是众所周知的 for 循环。在集合中使用 for 循环时，将隐式获取迭代器。因此，以下代码与上面的示例等效：

val numbers = listOf("one", "two", "three", "four")

for (item in numbers) {

println(item)

}

最后，有一个好用的 forEach() 函数，可自动迭代集合并为每个元素执行给定的代码。因此，等效的示例如下所示：

val numbers = listOf("one", "two", "three", "four")

numbers.forEach {

println(it)

}

#### 3.6.3.2 ListIterator

对于列表，有一个特殊的迭代器实现： [ListIterator](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-list-iterator/index.html) 它支持列表双向迭代：正向与反向。 反向迭代由 [hasPrevious()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-list-iterator/has-previous.html) 和 [previous()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-list-iterator/previous.html) 函数实现。 此外， ListIterator 通过 [nextIndex()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-list-iterator/next-index.html) 与 [previousIndex()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-list-iterator/previous-index.html) 函数提供有关元素索引的信息。

具有双向迭代的能力意味着 ListIterator 在到达最后一个元素后仍可以使用。

#### 3.6.3.3 [MutableIterator](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-iterator/index.html)

为了迭代可变集合，于是有了 [MutableIterator](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-iterator/index.html) 来扩展 Iterator 使其具有元素删除函数 [remove()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-iterator/remove.html) 。因此，可以在迭代时从集合中删除元素。除了删除元素， [MutableListIterator](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-list-iterator/index.html) 还可以在迭代列表时插入和替换元素。

### 3.6.4 区间和数列

#### 3.6.4.1 区间

Kotlin 可通过调用 kotlin.ranges 包中的 [rangeTo()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.ranges/range-to.html) 函数及其操作符形式的 .. 轻松地创建两个值的区间。 通常，rangeTo() 会辅以 in 或 !in 函数。

if (i in 1..4) { // 等同于 1 <= i && i <= 4

print(i)

}

整数类型区间（[IntRange](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.ranges/-int-range/index.html)、[LongRange](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.ranges/-long-range/index.html)、[CharRange](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.ranges/-char-range/index.html)）还有一个拓展特性：可以对其进行迭代。 这些区间也是相应整数类型的[等差数列](https://zh.wikipedia.org/wiki/%E7%AD%89%E5%B7%AE%E6%95%B0%E5%88%97)。 这种区间通常用于 for 循环中的迭代。

for (i in 1..4) print(i)

要反向迭代数字，请使用 [downTo](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.ranges/down-to.html) 函数而不是 .. 。

for (i in 4 downTo 1) print(i)

也可以通过任意步长（不一定为 1 ）迭代数字。 这是通过 [step](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.ranges/step.html) 函数完成的。

for (i in 1..8 step 2) print(i)

println()

for (i in 8 downTo 1 step 2) print(i)

要迭代不包含其结束元素的数字区间，请使用 [until](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.ranges/until.html) 函数：

for (i in 1 until 10) { // i in [1, 10), 10被排除

print(i)

}

要为类创建一个区间，请在区间起始值上调用 rangeTo() 函数，并提供结束值作为参数。 rangeTo() 通常以操作符 .. 形式调用。

这里的重点是，我们要做类区间声明时，需要让该类继承**Comparable方法后重写compareTo！即告知区间如何进行比较。**

**class Version(val major: Int, val minor: Int): Comparable<Version> {**

**override fun compareTo(other: Version): Int {**

**if (this.major != other.major) {**

**return this.major - other.major**

**}**

**return this.minor - other.minor**

**}**

**}**

fun main() {

val versionRange = Version(1, 11)..Version(1, 30)

println(Version(0, 9) in versionRange)

println(Version(1, 20) in versionRange)

}

区间关键字

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| in | for(in) | If(in) | If(!in) | downTo | step | until |
| 区间 | 遍历包含首尾的区间 | 是否在区间 | 是否不在区间 | 降序 | 步长 | 不包含结尾 |

#### 3.6.4.2 数列

如上个示例所示，整数类型的区间（例如 Int、Long 与 Char）可视为[等差数列](https://zh.wikipedia.org/wiki/%E7%AD%89%E5%B7%AE%E6%95%B0%E5%88%97)。 在 Kotlin 中，这些数列由特殊类型定义：[IntProgression](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.ranges/-int-progression/index.html)、[LongProgression](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.ranges/-long-progression/index.html) 与 [CharProgression](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.ranges/-char-progression/index.html)。

等差数列具有三个基本属性：first 元素、last 元素和一个非零的 step。 首个元素为 first，后续元素是前一个元素加上一个 step。 以确定的步长在数列上进行迭代等效于 Java/JavaScript 中基于索引的 for 循环。

数列的 last 元素是这样计算的：

对于正步长：不大于结束值且满足 (last - first) % step == 0 的最大值。

对于负步长：不小于结束值且满足 (last - first) % step == 0 的最小值。

**因此，last 元素并非总与指定的结束值相同。**

### 3.6.5序列

除了集合之外，Kotlin 标准库还包含另一种容器类型——序列（[Sequence<T>](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.sequences/-sequence/index.html)）。 序列提供与 [Iterable](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-iterable/index.html) 相同的函数，但实现另一种方法来进行多步骤集合处理。

当 Iterable 的处理包含多个步骤时，它们会优先执行：每个处理步骤完成并返回其结果——中间集合。 在此集合上执行以下步骤。反过来，序列的多步处理在可能的情况下会延迟执行：仅当请求整个处理链的结果时才进行实际计算。

操作执行的顺序也不同：Sequence 对每个元素逐个执行所有处理步骤。 反过来，Iterable 完成整个集合的每个步骤，然后进行下一步。

因此，这些序列可避免生成中间步骤的结果，从而提高了整个集合处理链的性能。 但是，序列的延迟性质增加了一些开销，这些开销在处理较小的集合或进行更简单的计算时可能很重要。 因此，应该同时考虑使用 Sequence 与 Iterable，并确定在哪种情况更适合。

#### 3.6.5.1 构造

元素构造

要创建一个序列，请调用 [sequenceOf()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.sequences/sequence-of.html) 函数，列出元素作为其参数。

val numbersSequence = sequenceOf("four", "three", "two", "one")

由 Iterable

如果已经有一个 Iterable 对象（例如 List 或 Set），则可以通过调用 [asSequence()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/as-sequence.html) 从而创建一个序列。

val numbers = listOf("one", "two", "three", "four")

val numbersSequence = numbers.asSequence()

函数构造

创建序列的另一种方法是通过使用计算其元素的函数来构建序列。 要基于函数构建序列，请以该函数作为参数调用 [generateSequence()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.sequences/generate-sequence.html)。 （可选）可以将第一个元素指定为显式值或函数调用的结果。 当提供的函数返回 null 时，序列生成停止。因此，以下示例中的序列是无限的。

val oddNumbers = generateSequence(1) { it + 2 } // `it` 是上一个元素

println(oddNumbers.take(5).toList())

要使用 generateSequence() 创建有限序列，请提供一个函数，**该函数在需要的最后一个元素之后返回 null。**

val oddNumbersLessThan10 = generateSequence(1) { if (it + 2 < 10) it + 2 else null }

println(oddNumbersLessThan10.count())

#### 3.6.5.2 Iterable与Sequence 的区别

需要先展示代码，输出日志进行对比。

1. Iterable是以集合为运行单位的，即一个操作符，需要对集合里的每一个元素操作完成之后，才能进入下一个操作符。
2. Sequence是以元素为单位的，即一个元素完成当前操作符后可以直接进入下一个操作符，当前元素完成能完成的操作符后，进入下一个元素。

相当与Iterable是横向执行，而Sequence是纵向执行，对于相同的操作，当存在take时，即只取前几个元素时，Sequence更有优势，达到take的数量后Sequence不再执行。

### 3.6.6 操作概述

Kotlin 标准库提供了用于对集合执行操作的多种函数。这包括简单的操作，例如获取或添加元素，以及更复杂的操作，包括搜索、排序、过滤、转换等。

集合操作在标准库中以两种方式声明：**集合接口的[成员函数](https://www.kotlincn.net/docs/reference/classes.html" \l "%E7%B1%BB%E6%88%90%E5%91%98)和[扩展函数](https://www.kotlincn.net/docs/reference/extensions.html" \l "%E6%89%A9%E5%B1%95%E5%87%BD%E6%95%B0)。**

#### 3.6.6.1 公共操作

公共操作可用于[只读集合与可变集合](https://www.kotlincn.net/docs/reference/collections-overview.html" \l "%E9%9B%86%E5%90%88%E7%B1%BB%E5%9E%8B)。 常见操作分为以下几类：

[集合转换](https://www.kotlincn.net/docs/reference/collection-transformations.html)

[集合过滤](https://www.kotlincn.net/docs/reference/collection-filtering.html)

[plus 与 minus 操作符](https://www.kotlincn.net/docs/reference/collection-plus-minus.html)

[分组](https://www.kotlincn.net/docs/reference/collection-grouping.html)

[取集合的一部分](https://www.kotlincn.net/docs/reference/collection-parts.html)

[取单个元素](https://www.kotlincn.net/docs/reference/collection-elements.html)

[集合排序](https://www.kotlincn.net/docs/reference/collection-ordering.html)

[集合聚合操作](https://www.kotlincn.net/docs/reference/collection-aggregate.html)

**这些页面中描述的操作将返回其结果，而不会影响原始集合。**例如，一个过滤操作产生一个新集合，其中包含与过滤谓词匹配的所有元素。 此类操作的结果应存储在变量中，或以其他方式使用，例如，传到其他函数中。

对于某些集合操作，有一个选项可以指定 目标 对象。 目标是一个可变集合，该函数将其结果项附加到该可变对象中，而不是在新对象中返回它们。 对于执行带有目标的操作，有单独的函数，其名称中带有 To 后缀，例如，用 [filterTo()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/filter-to.html) 代替 [filter()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/filter.html) 以及用 [associateTo()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/associate-to.html) 代替 [associate()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/associate.html)。 这些函数将目标集合作为附加参数。

val numbers = listOf("one", "two", "three", "four")

val filterResults = mutableListOf<String>() // 目标对象

numbers.filterTo(filterResults) { it.length > 3 }

numbers.filterIndexedTo(filterResults) { index, \_ -> index == 0 }

println(filterResults) // 包含两个操作的结果

以上代码我们要注意两个事情：

1.numbers.filterIndexedTo(filterResults) { index, \_ -> index == 0 }

当我们使用lambda表达式时，有一个参数未使用时，可以使用”\_”代替。

如果使用了可以改为：

numbers.filterIndexedTo(filterResults) {

index, item -> index == 0 .also { print("$item") }

}

1. filterIndexedTo是继续往现有集合中添加，如果我们更改下条件numbers.filterIndexedTo(filterResults) { index, \_ -> index == 3 }

会得到结果：[three, four, four]

#### 3.6.6.2 写操作

对于可变集合，还存在可更改集合状态的 写操作 。这些操作包括添加、删除和更新元素。写操作在[集合写操作](https://www.kotlincn.net/docs/reference/collection-write.html)以及 [List 写操作](https://www.kotlincn.net/docs/reference/list-operations.html" \l "list-%E5%86%99%E6%93%8D%E4%BD%9C)与 [Map 写操作](https://www.kotlincn.net/docs/reference/map-operations.html" \l "map-%E5%86%99%E6%93%8D%E4%BD%9C)的相应部分中列出。

对于某些操作，有成对的函数可以执行相同的操作：一个函数就地应用该操作，另一个函数将结果作为单独的集合返回。 例如:

**[sort()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/sort.html) 就地对可变集合进行排序**，因此其状态发生了变化

**[sorted()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/sorted.html) 创建一个新集合**，该集合包含按排序顺序相同的元素。

#### 3.6.6.3 转换

Kotlin 标准库为集合 转换 **提供了一组扩展函数**。 这些函数根据提供的转换规则从现有集合中构建新集合。

映射两个点：

1. map&mapIndexed 不过滤空元素，可以使用mapNotNull和mapIndexedNotNull来过滤空元素。
2. 映射转换时，有两个选择：转换键，使值保持不变，反之亦然。

使用mapKeys转换key，mapValues转换values。

**这里面很厉害的一点是key和value的类型也是可变的。**

如我们将给的测试用例变化为：

val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)  
println(numbersMap.mapKeys { it.key.length })  
println(numbersMap.mapValues { "asd"+it.value + it.key.length })

依然是可以运行通过的！

#### 3.6.6.4 过滤

在Kotlin中，过滤条件由 谓词 定义——接受一个集合元素并且返回布尔值的 lambda 表达式：true 说明给定元素与谓词匹配，false 则表示不匹配。

对于 List 和 Set，过滤结果都是一个 List，对 Map 来说结果还是一个 Map。

//list的过滤是单个参数的可以直接隐藏，通过it调用

val numbers = listOf("one", "two", "three", "four")

val longerThan3 = numbers.filter { it.length > 3 }

println(longerThan3)

//map的过滤是2个参数的，需要写标准的lambda表达式。

val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)

val filteredMap = numbersMap.filter { (key, value) -> key.endsWith("1") && value > 10}

println(filteredMap)

#### 3.6.6.5 plus 与 minus 操作符(操作符重载)

在 Kotlin 中，为集合定义了 [plus](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/plus.html) (+) 和 [minus](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/minus.html) (-) 操作符。 它们把一个集合作为第一个操作数；第二个操作数可以是一个元素或者是另一个集合。 **返回值是一个新的只读集合**：

plus 的结果包含原始集合 和 第二个操作数中的元素。

minus 的结果包含原始集合中的元素，将第二个操作数中的元素 除外。

**如果第二个操作数是一个元素，那么 minus 移除其在原始集合中的 第一次 出现**；

**如果是一个集合，那么移除其元素在原始集合中的 所有 出现**。

#### 3.6.6.6 分组

Kotlin 标准库提供用于对集合元素进行分组的扩展函数。 基本函数 [groupBy()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/group-by.html) 使用一个 lambda 函数并返回一个 Map。 在此 Map 中，每个键都是 lambda 结果，而对应的值是返回此结果的元素 List。 例如，可以使用此函数将 String 列表按首字母分组。

还可以使用第二个 lambda 参数（值转换函数）调用 groupBy()。 在带有两个 lambda 的 groupBy() 结果 Map 中，由 keySelector 函数生成的键映射到值转换函数的结果，而不是原始元素。

val numbers = listOf("one", "two", "three", "four", "five")

println(numbers.groupBy { it.first().toUpperCase() })

println(numbers.groupBy(keySelector = { it.first() }, valueTransform = { it.toUpperCase() }))

此处的代码在理解上可能有一定难度，这里给出查阅底层实现后给出解释。

it.first().toUpperCase() 代表对每个元素的首字母做处理，将处理后的结果变成key

在得到key后将该元素存到对应的list中

one 的 首字母大写 O 此时得到 O 对应的集合 O=[one]

two的 首字母大写 T 此时得到 T 对应的集合 T=[two]

以此类推，最后得到 {O=[one], T=[two, three], F=[four, five]}

双参数的groupBy仅是增加了对值的处理{o=[ONE], t=[TWO, THREE], f=[FOUR, FIVE]}

如果要对元素进行分组，然后一次将操作应用于所有分组，请使用 [groupingBy()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/grouping-by.html) 函数。 它返回一个 [Grouping](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-grouping/index.html) 类型的实例。 通过 Grouping 实例，可以以一种惰性的方式将操作应用于所有组：这些分组实际上是刚好在执行操作前构建的。

val numbers = listOf("one", "two", "three", "four", "five", "six")

println(numbers.groupingBy { it.first() }.eachCount())

[Grouping](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-grouping/index.html) 操作简单来说就是在分组的基础上，对现有集合进行处理，上面处理后得到：

{o=1, t=2, f=2, s=1}

#### 3.6.6.7 取集合的一部分

Kotlin 标准库包含用于取集合的一部分的扩展函数。 这些函数提供了多种方法来选择结果集合的元素：显式列出其位置、指定结果大小等。

Slice

[slice()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/slice.html) 返回具有给定索引的集合元素列表。 索引既可以是作为[区间](https://www.kotlincn.net/docs/reference/ranges.html)传入的也可以是作为整数值的集合传入的。

slice(indices: IntRange)

slice(indices: Iterable<Int>)

Slice可以传入Range和集合，集合可以指定元素的位置。

val numbers = listOf("one", "two", "three", "four", "five", "six")

println(numbers.slice(1..3))

println(numbers.slice(0..4 step 2))

println(numbers.slice(setOf(3, 5, 0)))

Take 与 drop

要从头开始获取指定数量的元素，请使用 [take()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/take.html) 函数。 要从尾开始获取指定数量的元素，请使用 [takeLast()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/take-last.html)。 当调用的数字大于集合的大小时，两个函数都将返回整个集合。

要从头或从尾去除给定数量的元素，请调用 [drop()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/drop.html) 或 [dropLast()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/drop-last.html) 函数。

[takeWhile()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/take-while.html) 是带有谓词的 take()：它将不停获取元素直到排除与谓词匹配的首个元素。如果首个集合元素与谓词匹配，则结果为空。

[takeLastWhile()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/take-last-while.html) 与 takeLast() 类似：它从集合末尾获取与谓词匹配的元素区间。区间的首个元素是与谓词不匹配的最后一个元素右边的元素。如果最后一个集合元素与谓词匹配，则结果为空。

[dropWhile()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/drop-while.html) 与具有相同谓词的 takeWhile() 相反：它将首个与谓词不匹配的元素返回到末尾。

[dropLastWhile()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/drop-last-while.html) 与具有相同谓词的 takeLastWhile() 相反：它返回从开头到最后一个与谓词不匹配的元素。

Chunked

要将集合分解为给定大小的“块”，请使用 [chunked()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/chunked.html) 函数。 chunked() 采用一个参数（块的大小），并返回一个 List 其中包含给定大小的 List。 第一个块从第一个元素开始并包含 size 元素，第二个块包含下一个 size 元素，依此类推。 最后一个块的大小可能较小。还可以立即对返回的块应用转换。 为此，请在调用 chunked() 时将转换作为 lambda 函数提供。 lambda 参数是集合的一块。当通过转换调用 chunked() 时， 这些块是临时的 List，应立即在该 lambda 中使用。

Windowed

可以检索给定大小的集合元素中所有可能区间。 获取它们的函数称为 [windowed()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/windowed.html)：它返回一个元素区间列表，比如通过给定大小的滑动窗口查看集合，则会看到该区间。 与 chunked() 不同，windowed() 返回从每个集合元素开始的元素区间（窗口）。 所有窗口都作为单个 List 的元素返回。

val numbers = listOf("one", "two", "three", "four", "five")

println(numbers.windowed(3))

运行结果：[[one, two, three], [two, three, four], [three, four, five]]

windowed() 通过可选参数提供更大的灵活性：

step 定义两个相邻窗口的第一个元素之间的距离。默认情况下，该值为 1，因此结果包含从所有元素开始的窗口。如果将 step 增加到 2，将只收到以奇数元素开头的窗口：第一个、第三个等。

partialWindows 包含从集合末尾的元素开始的较小的窗口。例如，如果请求三个元素的窗口，就不能为最后两个元素构建它们。在本例中，启用 partialWindows 将包括两个大小为2与1的列表。

最后，可以立即对返回的区间应用转换。 为此，在调用 windowed() 时将转换作为 lambda 函数提供。

#### 3.6.6.8 取一个元素

Kotlin 集合提供了一套从集合中检索单个元素的函数。 此描述的函数适用于 list 和 set。

正如 [list 的定义](https://www.kotlincn.net/docs/reference/collections-overview.html)所言，list 是有序集合。 因此，list 中的每个元素都有其位置可供你引用。 除了此页面上描述的函数外，list 还提供了更广泛的一套方法去按索引检索和搜索元素。

反过来，从[定义](https://www.kotlincn.net/docs/reference/collections-overview.html)来看，set 并不是有序集合。 但是，**Kotlin 中的 Set 按某些顺序存储元素**。 这些可以是**插入顺序（在 LinkedHashSet 中）**、**自然排序顺序（在 SortedSet 中）**或者其他顺序。 一组元素的顺序也可以是未知的。 在这种情况下，元素仍会以某种顺序排序，因此，依赖元素位置的函数仍会返回其结果。 但是，除非调用者知道所使用的 Set 的具体实现，否则这些结果对于调用者是不可预测的。

按位置取

为了检索特定位置的元素，有一个函数 [elementAt()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/element-at.html)。 用一个整数作为参数来调用它，你会得到给定位置的集合元素。 第一个元素的位置是 0，最后一个元素的位置是 (size - 1)。

elementAt() 对于不提供索引访问或非静态已知提供索引访问的集合很有用。 在使用 List 的情况下，使用[索引访问操作符](https://www.kotlincn.net/docs/reference/list-operations.html" \l "%E6%8C%89%E7%B4%A2%E5%BC%95%E5%8F%96%E5%85%83%E7%B4%A0) （get() 或 []）更为习惯。

val numbers = **linkedSetOf**("one", "two", "three", "four", "five")

println(numbers.elementAt(3))

val numbersSortedSet = **sortedSetOf**("one", "two", "three", "four")

println(numbersSortedSet.elementAt(0)) // 元素以升序存储

运行结果：four four

还有一些有用的别名来检索集合的第一个和最后一个元素：[first()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/first.html) 和 [last()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/last.html)。

val numbers = listOf("one", "two", "three", "four", "five")

println(numbers.first())

println(numbers.last())

运行结果：one five

**为了避免在检索位置不存在的元素时出现异常，请使用 elementAt() 的安全变体：**

**当指定位置超出集合范围时，[elementAtOrNull()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/element-at-or-null.html) 返回 null。**

**[elementAtOrElse()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/element-at-or-else.html) 还接受一个 lambda 表达式，该表达式能将一个 Int 参数映射为一个集合元素类型的实例。** 当使用一个越界位置来调用时，elementAtOrElse() 返回对给定值调用该 lambda 表达式的结果。

按条件取

函数 [first()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/first.html) 和 [last()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/last.html) 还可以让你在集合中搜索与给定谓词匹配的元素。 当你使用测试集合元素的谓词调用 first() 时，你会得到对其调用谓词产生 true 的第一个元素。 反过来，带有一个谓词的 last() 返回与其匹配的最后一个元素。

val numbers = listOf("one", "two", "three", "four", "five", "six")

println(numbers.first { it.length > 3 })

println(numbers.last { it.startsWith("f") })

运行结果：three five

如果没有元素与谓词匹配，两个函数都会抛异常。 为了避免它们，请改用 [firstOrNull()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/first-or-null.html) 和 [lastOrNull()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/last-or-null.html)：如果找不到匹配的元素，它们将返回 null。

val numbers = listOf("one", "two", "three", "four", "five", "six")

println(numbers.firstOrNull { it.length > 6 })

运行结果：null

或者，如果别名更适合你的情况，那么可以使用别名：

使用 [find()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/find.html) 代替 firstOrNull()

使用 [findLast()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/find-last.html) 代替 lastOrNull()

val numbers = listOf(1, 2, 3, 4)

println(numbers.find { it % 2 == 0 })

println(numbers.findLast { it % 2 == 0 })

运行结果： 2 4

随机取元素

如果需要检索集合的一个随机元素，那么请调用 [random()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/random.html) 函数。 你可以不带参数或者使用一个 [Random](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.random/-random/index.html) 对象作为随机源来调用它。

val numbers = listOf(1, 2, 3, 4)

println(numbers.random())

运行结果： 结果随机

当集合为空的时候random()会抛出异常，请使用 [randomOrNull()](https://www.kotlincn.net/api/latest/jvm/stdlib/kotlin.collections/random-or-null.html)进行替换，为空的时候会返回null。

检测存在与否

如需检查集合中某个元素的存在，可以使用 [contains()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/contains.html) 函数。 如果存在一个集合元素等于（equals()）函数参数，那么它返回 true。 你可以使用 in 关键字以操作符的形式调用 contains()。

如需一次检查多个实例的存在，可以使用这些实例的集合作为参数调用 [containsAll()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/contains-all.html)。

val numbers = listOf("one", "two", "three", "four", "five", "six")

println(numbers.contains("four"))

println("zero" in numbers)

println(numbers.containsAll(listOf("four", "two")))

println(numbers.containsAll(listOf("one", "zero")))

运行结果：true false true false

此外，你可以通过调用 [isEmpty()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/is-empty.html) 和 [isNotEmpty()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/is-not-empty.html) 来检查集合中是否包含任何元素。

val numbers = listOf("one", "two", "three", "four", "five", "six")

println(numbers.isEmpty())

println(numbers.isNotEmpty())

val empty = emptyList<String>()

println(empty.isEmpty())

println(empty.isNotEmpty())

运行结果：false true true false

#### 3.6.6.9 集合排序

元素的顺序是某些集合类型的一个重要方面。 例如，如果拥有相同元素的两个列表的元素顺序不同，那么这两个列表也不相等。

在 Kotlin 中，可以通过多种方式定义对象的顺序。

首先，有 **自然顺序。它是为 [Comparable](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-comparable/index.html) 接口的继承者定义的。 当没有指定其他顺序时，使用自然顺序为它们排序。**

大多数内置类型是可比较的：

**数值类型使用传统的数值顺序**：1 大于 0； -3.4f 大于 -5f，以此类推。

**Char 和 String 使用[字典顺序](https://en.wikipedia.org/wiki/Lexicographical_order)**： b 大于 a； world 大于 hello。

**如需为用户定义的类型定义一个自然顺序，可以让这个类型继承 Comparable**。 这需要实现 compareTo() 函数。 compareTo() 必须将另一个具有相同类型的对象作为参数并返回一个整数值来显示哪个对象更大：

正值表明接收者对象更大。

负值表明它小于参数。

0 说明对象相等。

下面是一个类，可用于排序由主版本号和次版本号两部分组成的版本。

class Version(val major: Int, val minor: Int): Comparable<Version> {

override fun compareTo(other: Version): Int {

if (this.major != other.major) {

return this.major - other.major

} else if (this.minor != other.minor) {

return this.minor - other.minor

} else return 0

}

}

​

fun main() {

println(Version(1, 2) > Version(1, 3))

println(Version(2, 0) > Version(1, 5))

}

运行结果：false true

自定义 顺序让你可以按自己喜欢的方式对任何类型的实例进行排序。 特别是，你可以为不可比较类型定义顺序，或者为可比较类型定义非自然顺序。 如需为类型定义自定义顺序，可以为其创建一个**[Comparator](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-comparator/index.html)**。 Comparator 包含 compare() 函数：它接受一个类的两个实例并返回它们之间比较的整数结果。 如上所述，对结果的解释与 compareTo() 的结果相同。

补充代码：

val intComparable = Comparator { int1: Int, int2: Int -> int1 - int2}  
println(listOf(4, 5, 2, 1).sortedWith(intComparable))

运行结果：[1, 2, 4, 5]

此部分代码的意义在于明确对于**排序后集合是升序排列**。

val lengthComparator = Comparator { str1: String, str2: String -> str1.length - str2.length }

println(listOf("aaa", "bb", "c").sortedWith(lengthComparator))

运行结果：[c, bb, aaa] 由于升序 c的长度最短所以在第一位

有了 lengthComparator，你可以按照字符串的长度而不是默认的字典顺序来排列字符串。

定义一个 Comparator 的一种比较简短的方式是标准库中的 [compareBy()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.comparisons/compare-by.html) 函数。 compareBy() 接受一个 lambda 表达式，该表达式从一个实例产生一个 Comparable 值，并将自定义顺序定义为生成值的自然顺序。 使用 compareBy()，上面示例中的长度比较器如下所示：

​

println(listOf("aaa", "bb", "c").sortedWith(compareBy { it.length }))

运行结果：[c, bb, aaa]

Kotlin 集合包提供了用于按照自然顺序、自定义顺序甚至随机顺序对集合排序的函数。 在此页面上，我们将介绍适用于[只读](https://www.kotlincn.net/docs/reference/collections-overview.html" \l "%E9%9B%86%E5%90%88%E7%B1%BB%E5%9E%8B)集合的排序函数。 这些函数将它们的结果作为一个新集合返回，集合里包含了按照请求顺序排序的来自原始集合的元素。

自然顺序

基本的函数 [sorted()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/sorted.html) 和 [sortedDescending()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/sorted-descending.html) 返回集合的元素，这些元素按照其自然顺序升序和降序排序。 这些函数适用于 Comparable 元素的集合。

val numbers = listOf("one", "two", "three", "four")

println("Sorted ascending: ${numbers.sorted()}")

println("Sorted descending: ${numbers.sortedDescending()}")

运行结果：

Sorted ascending: [four, one, three, two]

Sorted descending: [two, three, one, four]

自定义顺序

为了按照自定义顺序排序或者对不可比较对象排序，可以使用函数 [sortedBy()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/sorted-by.html) 和 [sortedByDescending()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/sorted-by-descending.html)。 它们接受一个将集合元素映射为 Comparable 值的选择器函数，并以该值的自然顺序对集合排序。

val numbers = listOf("one", "two", "three", "four")

val sortedNumbers = numbers.sortedBy { it.length }

println("Sorted by length ascending: $sortedNumbers")

val sortedByLast = numbers.sortedByDescending { it.last() }

println("Sorted by the last letter descending: $sortedByLast")

运行结果：

Sorted by length ascending: [one, two, four, three]

Sorted by the last letter descending: [four, two, one, three]

倒序

你可以使用 [reversed()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/reversed.html) 函数以相反的顺序检索集合。

val numbers = listOf("one", "two", "three", "four")

println(numbers.reversed())

运行结果：[four, three, two, one]

**reversed() 返回带有元素副本的新集合**。 **因此，如果你之后改变了原始集合，这并不会影响先前获得的 reversed() 的结果。**

**另一个反向函数——[asReversed()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/as-reversed.html)——返回相同集合实例的一个反向视图，因此，如果原始列表不会发生变化，那么它会比 reversed() 更轻量，更合适。**

val numbers = listOf("one", "two", "three", "four")

val reversedNumbers = numbers.asReversed()

println(reversedNumbers)

运行结果： [four, three, two, one]

如果原始列表是可变的，那么其所有更改都会反映在其反向视图中，反之亦然。

val numbers = mutableListOf("one", "two", "three", "four")

val reversedNumbers = numbers.asReversed()

println(reversedNumbers)

numbers.add("five")

println(reversedNumbers)

运行结果:

[four, three, two, one]

[five, four, three, two, one]

但是，**如果列表的可变性未知或者源根本不是一个列表，那么 reversed() 更合适，因为其结果是一个未来不会更改的副本。**

随机顺序

最后，[shuffled()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/shuffled.html) 函数返回一个包含了以随机顺序排序的集合元素的新的 List。 你可以不带参数或者使用 [Random](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.random/-random/index.html) 对象来调用它。

val numbers = listOf("one", "two", "three", "four")

println(numbers.shuffled())

运行结果:产生新的随机集合

#### 3.6.6.10 集合聚合操作

Kotlin 集合包含用于常用的 聚合操作 （**基于集合内容返回单个值的操作**）的函数 。 其中大多数是众所周知的，并且其工作方式与在其他语言中相同。

[min()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/min.html) 与 [max()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/max.html) 分别返回最小和最大的元素；

[average()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/average.html) 返回数字集合中元素的平均值；

[sum()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/sum.html) 返回数字集合中元素的总和；

[count()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/count.html) 返回集合中元素的数量；

//此处代码需要注意的是min()和max()已经被maxOrNull()和minOrNull()取代。

//当最大值和最小值出现多个时，最大值最小值会产生，其值不变

val numbers = listOf(6, 6,42,42, 10, 4)  
  
println("Count: ${numbers.count()}")  
println("Max: ${numbers.maxOrNull()}")  
println("Min: ${numbers.minOrNull()}")  
println("Average: ${numbers.average()}")  
println("Sum: ${numbers.sum()}")

运行结果：Count: 6 Max: 42 Min: 4 Average: 18.333333333333332 Sum: 110

还有一些通过某些选择器函数或自定义 [Comparator](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-comparator/index.html) 来检索最小和最大元素的函数。

[maxBy()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/max-by.html)/[minBy()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/min-by.html) 接受一个选择器函数并返回使选择器返回最大或最小值的元素。

[maxWith()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/max-with.html)/[minWith()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/min-with.html) 接受一个 Comparator 对象并且根据此 Comparator 对象返回最大或最小元素。

val numbers = listOf(5, 42, 10, 4)  
val min3Remainder = numbers.minByOrNull { it % 3 }  
println(min3Remainder)  
  
val strings = listOf("one", "two", "three", "four")  
val longestString = strings.maxWithOrNull(compareBy { it.length })  
println(longestString)

运行结果：42 three

此外，有一些高级的求和函数，它们接受一个函数并返回对所有元素调用此函数的返回值的总和：

[sumBy()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/sum-by.html) 使用对集合元素调用返回 Int 值的函数。

[sumByDouble()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/sum-by-double.html) 与返回 Double 的函数一起使用。

//使用字符串集合可以更好的展示[sumBy()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/sum-by.html)和[sumByDouble()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/sum-by-double.html)函数

val strings = listOf("one", "two", "three", "four")  
println(strings.sumBy { it.length \* 2 })  
println(strings.sumByDouble { it.length.toDouble() / 2 })

运行结果：30 7.5

Fold 与 reduce

对于更特定的情况，有函数 [reduce()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/reduce.html) 和 [fold()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/fold.html)，它们依次将所提供的操作应用于集合元素并返回累积的结果。 操作有两个参数：先前的累积值和集合元素。

这两个函数的区别在于：

fold() 接受一个初始值并将其用作第一步的累积值

reduce() 的第一步则将第一个和第二个元素作为第一步的操作参数。

val numbers = listOf(5, 2, 10, 4)

val sum = numbers.reduce { sum, element -> sum + element }

println(sum)

val sumDoubled = numbers.fold(0) { sum, element -> sum + element \* 2 }

println(sumDoubled)

运行结果:21 42

上面的实例展示了区别：fold() 用于计算加倍的元素之和。 如果将相同的函数传给 reduce()，那么它会返回另一个结果，因为在第一步中它将列表的第一个和第二个元素作为参数，所以第一个元素不会被加倍。

如需将函数以相反的顺序应用于元素，可以使用函数 [reduceRight()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/reduce-right.html) 和 [foldRight()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/fold-right.html) 它们的工作方式类似于 fold() 和 reduce()，但从最后一个元素开始，然后再继续到前一个元素。 记住，在使用 foldRight 或 reduceRight 时，操作参数会更改其顺序：第一个参数变为元素，然后第二个参数变为累积值。

//**此处要注意的是reduce 与reduceRight 的传参顺序和运算结果都是不一样的**

val numbers = listOf(5, 2, 10, 4)  
val sumDoubledRight = numbers.foldRight(0) { element, sum -> sum + element \* 2 }  
println(sumDoubledRight)  
  
println("reduce:" + numbers.reduce { sum, element -> sum - element })  
println("reduceRight:" + numbers.reduceRight { element, sum -> sum - element })

运行结果：42 reduce:-11 reduceRight:-13

你还可以使用将元素索引作为参数的操作。 为此，使用函数 [reduceIndexed()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/reduce-indexed.html) 和 [foldIndexed()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/fold-indexed.html) 传递元素索引作为操作的第一个参数。

最后，还有将这些操作从右到左应用于集合元素的函数[reduceRightIndexed()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/reduce-right-indexed.html) 与 [foldRightIndexed()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/fold-right-indexed.html)。

val numbers = listOf(5, 2, 10, 4)

val sumEven = numbers.foldIndexed(0) { idx, sum, element -> if (idx % 2 == 0) sum + element else sum }

println(sumEven)

​

val sumEvenRight = numbers.foldRightIndexed(0) { idx, element, sum -> if (idx % 2 == 0) sum + element else sum }

println(sumEvenRight)

运行结果:15 15

所有的reduce 操作当空的时候都会抛出异常，推荐使用加OrNull()方法的。

[reduceOrNull()](https://www.kotlincn.net/api/latest/jvm/stdlib/kotlin.collections/reduce-or-null.html)

[reduceRightOrNull()](https://www.kotlincn.net/api/latest/jvm/stdlib/kotlin.collections/reduce-right-or-null.html)

[reduceIndexedOrNull()](https://www.kotlincn.net/api/latest/jvm/stdlib/kotlin.collections/reduce-indexed-or-null.html)

[reduceRightIndexedOrNull()](https://www.kotlincn.net/api/latest/jvm/stdlib/kotlin.collections/reduce-right-indexed-or-null.html)

#### 3.6.6.11 集合写操作

**[可变集合](https://www.kotlincn.net/docs/reference/collections-overview.html" \l "%E9%9B%86%E5%90%88%E7%B1%BB%E5%9E%8B)支持更改集合内容的操作**，**例如添加或删除元素**。 在此页面上，我们将描述实现 MutableCollection 的所有写操作。

添加元素

要将单个元素添加到列表或集合，请使用 [add()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-list/add.html) 函数。指定的对象将添加到集合的末尾。

val numbers = mutableListOf(1, 2, 3, 4)

numbers.add(5)

println(numbers)

运行结果：[1, 2, 3, 4, 5]

[addAll()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/add-all.html) 将参数对象的每个元素添加到列表或集合中。参数可以是 Iterable、Sequence 或 Array。 接收者的类型和参数可能不同，例如，你可以将所有内容从 Set 添加到 List。

当在列表上调用时，addAll() 会按照在参数中出现的顺序添加各个新元素。 你也可以调用 addAll() 时指定一个元素位置作为第一参数。 参数集合的第一个元素会被插入到这个位置。 其他元素将跟随在它后面，将接收者元素移到末尾。

val numbers = mutableListOf(1, 2, 5, 6)

numbers.addAll(arrayOf(7, 8))

println(numbers)

numbers.addAll(2, setOf(3, 4))

println(numbers)

运行结果：

[1, 2, 5, 6, 7, 8]

[1, 2, 3, 4, 5, 6, 7, 8]

你还可以使用 [plus 运算符](https://www.kotlincn.net/docs/reference/collection-plus-minus.html) - [plusAssign](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/plus-assign.html) (+=) 添加元素。 当应用于可变集合时，+= 将第二个操作数(一个元素或另一个集合)追加到集合的末尾。

val numbers = mutableListOf("one", "two")

numbers += "three"

println(numbers)

numbers += listOf("four", "five")

println(numbers)

运行结果：

[one, two, three]

[one, two, three, four, five]

删除元素

若要从可变集合中移除元素，请使用 [remove()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/remove.html) 函数。 remove() 接受元素值，并删除该值的一个匹配项。

val numbers = mutableListOf(1, 2, 3, 4, 3)

numbers.remove(3) // 删除了第一个 `3`

println(numbers)

numbers.remove(5) // 什么都没删除

println(numbers)

运行结果:

[1, 2, 4, 3]

[1, 2, 4, 3]

**要一次删除多个元素**，有以下函数：

[removeAll()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/remove-all.html) 移除参数集合中存在的所有元素。 或者，**你可以用谓词作为参数来调用它；在这种情况下，函数移除谓词产生 true 的所有元素**。

[retainAll()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/retain-all.html) 与 removeAll() 相反：**它移除除参数集合中的元素之外的所有元素**。 当与谓词一起使用时，它只留下与之匹配的元素。

[clear()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-list/clear.html) 从列表中移除所有元素并将其置空。

val numbers = mutableListOf(1, 2, 3,3,3, 4)  
println(numbers)  
numbers.retainAll { it >= 3 }  
println(numbers)  
numbers.clear()  
println(numbers)  
  
val numbersSet = mutableSetOf("one", "one", "one", "two", "three", "four")  
numbersSet.removeAll(setOf("one", "two"))  
println(numbersSet)

运行结果：

[1, 2, 3, 3, 3, 4]

[3, 3, 3, 4]

[]

[three, four]

从集合中移除元素的另一种方法是使用 [minusAssign](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/minus-assign.html) (-=) ——原地修改版的 [minus](https://www.kotlincn.net/docs/reference/collection-plus-minus.html) 操作符。 [minus](https://www.kotlincn.net/docs/reference/collection-plus-minus.html) 操作符。 第二个参数可以是元素类型的单个实例或另一个集合。 **右边是单个元素时，-= 会移除它的第一个匹配项。 反过来，如果它是一个集合，那么它的所有元素的每次出现都会删除。** 例如，如果列表包含重复的元素，它们将被同时删除。 第二个操作数可以包含集合中不存在的元素。这些元素不会影响操作的执行。

val numbers = mutableListOf("one", "two", "three", "three", "four")

numbers -= "three"

println(numbers)

numbers -= listOf("four", "five")

//numbers -= listOf("four") // 与上述相同

println(numbers)

运行结果：

[one, two, three, four]

[one, two, three]

更新元素

list 和 map 还提供更新元素的操作，**对于 set 来说，更新没有意义，因为它实际上是移除一个元素并添加另一个元素。**

#### 3.6.6.12 List相关操作

[List](https://www.kotlincn.net/docs/reference/collections-overview.html" \l "list) 是 Kotlin 标准库中最受欢迎的集合类型。对列表元素的索引访问为 List 提供了一组强大的操作。

按索引取元素

List 支持按索引取元素的所有常用操作： elementAt() 、 first() 、 last() 与[取单个元素](https://www.kotlincn.net/docs/reference/collection-elements.html)中列出的其他操作。 List 的特点是能通过索引访问特定元素，因此读取元素的最简单方法是按索引检索它。 这是通过 [get()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-list/get.html) 函数或简写语法 [index] 来传递索引参数完成的。

如果 List 长度小于指定的索引，则抛出异常。 另外，还有两个函数能避免此类异常：

[getOrElse()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/get-or-else.html) 提供用于计算默认值的函数，如果集合中不存在索引，则返回默认值。

[getOrNull()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/get-or-null.html) 返回 null 作为默认值。

val numbers = listOf(1, 2, 3, 4)

println(numbers.get(0))

println(numbers[0])

//numbers.get(5) // exception!

println(numbers.getOrNull(5)) // null

println(numbers.getOrElse(5, {it})) // 5

运行结果：1 1 null 5

取列表的一部分

除了[取集合的一部分](https://www.kotlincn.net/docs/reference/collection-parts.html)中常用的操作， List 还提供 [subList()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-list/sub-list.html) 该函数将指定元素范围的视图作为列表返回。 因此，如果原始集合的元素发生变化，则它在先前创建的子列表中也会发生变化，反之亦然。

val numbers = (0..13).toList()  
println(numbers)  
println(numbers.subList(3, 6))

运行结果：

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]

[3, 4, 5]

查找元素位置

线性查找

在任何列表中，都可以使用 [indexOf()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/index-of.html) 或 [lastIndexOf()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/last-index-of.html) 函数找到元素的位置。 它们返回与列表中给定参数相等的元素的第一个或最后一个位置。 **如果没有这样的元素，则两个函数均返回 -1。**

val numbers = listOf(1, 2, 3, 4, 2, 5)

println(numbers.indexOf(2))

println(numbers.lastIndexOf(2))

运行结果：1 4

还有一对函数接受谓词并搜索与之匹配的元素：

[indexOfFirst()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/index-of-first.html) 返回与谓词匹配的第一个元素的索引，如果没有此类元素，则返回 -1。

[indexOfLast()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/index-of-last.html) **返回与谓词匹配的最后一个元素的索引**，如果没有此类元素，则返回 -1。

val numbers = mutableListOf(1, 2, 3, 4)

println(numbers.indexOfFirst { it > 2})

println(numbers.indexOfLast { it % 2 == 1})

运行结果：2 2

在有序列表中二分查找

还有另一种搜索列表中元素的方法——[二分查找算法](https://zh.wikipedia.org/wiki/%E4%BA%8C%E5%88%86%E6%90%9C%E5%B0%8B%E6%BC%94%E7%AE%97%E6%B3%95)。 它的工作速度明显快于其他内置搜索功能，但要求该列表按照一定的顺序（自然排序或函数参数中提供的另一种排序）按升序[排序过](https://www.kotlincn.net/docs/reference/collection-ordering.html)。 否则，结果是不确定的。

要搜索已排序列表中的元素，请调用 [binarySearch()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/binary-search.html) 函数，并将该值作为参数传递。 如果存在这样的元素，则函数返回其索引；否则，将返回 (-insertionPoint - 1)，其中 insertionPoint 为应插入此元素的索引，以便列表保持排序。 如果有多个具有给定值的元素，搜索则可以返回其任何索引。

还可以指定要搜索的索引区间：在这种情况下，该函数仅在两个提供的索引之间搜索。

val numbers = mutableListOf("one", "two", "three", "four")

**numbers.sort() //使用二分查找的前提是有序**

println(numbers)

println(numbers.binarySearch("two")) // 3

println(numbers.binarySearch("z")) // -5

println(numbers.binarySearch("two", 0, 2)) // -3

运行结果：3 -5 -3

Comparator 二分搜索

**如果列表元素不是 Comparable，则应提供一个用于二分搜索的 [Comparator](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-comparator.html)。 该列表必须根据此 Comparator 以升序排序。**来看一个例子：

data class Product(val name: String, val price: Double)

val productList = listOf(

Product("WebStorm", 49.0),

Product("AppCode", 99.0),

Product("DotTrace", 129.0),

Product("ReSharper", 149.0))

println(productList.binarySearch(Product("AppCode", 99.0), compareBy<Product> { it.price }.thenBy { it.name }))

运行结果：1

这是一个不可排序的 Product 实例列表，以及一个定义排序的 Comparator：如果 p1 的价格小于 p2 的价格，则产品 p1 在产品 p2 之前。 因此，按照此顺序对列表进行升序排序后，使用 binarySearch() 查找指定的 Product的索引。

当列表使用与自然排序不同的顺序时（例如，对 String 元素不区分大小写的顺序），自定义 Comparator 也很方便。

//先声明一个无序的 listOf

val colors = listOf( "Red","Blue", "green", "ORANGE", "yellow")

//对现有集合进行排序，返回新的集合  
val nColors=colors.sortedWith(String.CASE\_INSENSITIVE\_ORDER)  
println(nColors) //输出新的集合

//在有序集合能查找  
println(nColors.binarySearch("RED", String.CASE\_INSENSITIVE\_ORDER))

运行结果：[Blue, green, ORANGE, Red, yellow] 3

比较函数二分搜索

使用 比较 函数的二分搜索无需提供明确的搜索值即可查找元素。 取而代之的是，**它使用一个比较函数将元素映射到 Int 值**，并搜索函数返回 0 的元素。 **该列表必须根据提供的函数以升序排序；换句话说，比较的返回值必须从一个列表元素增长到下一个列表元素**。

data class Product(val name: String, val price: Double)  
//将我们的数据对象映射到整形上  
fun priceComparison(product: Product, price: Double) = sign(product.price - price).toInt()  
  
val productList = listOf(  
 Product("WebStorm", 49.0),  
 Product("AppCode", 99.0),  
 Product("DotTrace", 129.0),  
 Product("ReSharper", 149.0))  
//这样我们就可以类似模糊搜索了 被搜索的字段需要有序！  
println(productList.binarySearch { priceComparison(it, 99.0) })

运行结果：1

**Comparator 与比较函数二分搜索都可以针对列表区间执行。**

List 写操作

除了[集合写操作](https://www.kotlincn.net/docs/reference/collection-write.html)中描述的集合修改操作之外，[可变](https://www.kotlincn.net/docs/reference/collections-overview.html" \l "%E9%9B%86%E5%90%88%E7%B1%BB%E5%9E%8B)列表还支持特定的写操作。 这些操作使用索引来访问元素以扩展列表修改功能。

添加

要将元素添加到列表中的特定位置，请使用 [add()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-list/add.html) 或 [addAll()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/add-all.html) 并提供元素插入的位置作为附加参数。 位置之后的所有元素都将向右移动。

val numbers = mutableListOf("one", "five", "six")

numbers.add(1, "two")

numbers.addAll(2, listOf("three", "four"))

println(numbers)

运行结果：[one, two, three, four, five, six]

更新

列表还提供了在指定位置替换元素的函数——[set()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-list/set.html) 及其操作符形式 []。set() 不会更改其他元素的索引。

val numbers = mutableListOf("one", "five", "three")

numbers[1] = "two"

println(numbers)

运行结果：[one, two, three]

[fill()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/fill.html) **简单地将所有集合元素的值替换为指定值**。

val numbers = mutableListOf(1, 2, 3, 4)

numbers.fill(3)

println(numbers)

运行结果：[3, 3, 3, 3]

删除

要从列表中删除指定位置的元素，请使用 [removeAt()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-list/remove-at.html) 函数，并将位置作为参数。 在元素被删除之后出现的所有元素索引将减 1。

val numbers = mutableListOf(1, 2, 3, 4, 3)

numbers.removeAt(1)

println(numbers)

运行结果：[1, 3, 4, 3]

可以使用[removeFirst()](https://www.kotlincn.net/api/latest/jvm/stdlib/kotlin.collections/-mutable-list/remove-first.html) and [removeLast()](https://www.kotlincn.net/api/latest/jvm/stdlib/kotlin.collections/-mutable-list/remove-last.html)去移除队列的第一个和最后一个元素，当队列为空的时候会抛出异常，推荐使用[removeFirstOrNull()](https://www.kotlincn.net/api/latest/jvm/stdlib/kotlin.collections/-mutable-list/remove-first-or-null.html) and [removeLastOrNull()](https://www.kotlincn.net/api/latest/jvm/stdlib/kotlin.collections/-mutable-list/remove-last-or-null.html)

val numbers = mutableListOf(1, 2, 3, 4, 3)

numbers.removeFirst()

numbers.removeLast()

println(numbers)

​

val empty = mutableListOf<Int>()

// empty.removeFirst() // NoSuchElementException: List is empty.

empty.removeFirstOrNull() //null

运行结果：[2, 3, 4]

排序

在[集合排序](https://www.kotlincn.net/docs/reference/collection-ordering.html)中，描述了按特定顺序检索集合元素的操作。 对于可变列表，标准库中提供了类似的扩展函数，这些扩展函数可以执行相同的排序操作。 将此类操作应用于列表实例时，它将更改指定实例中元素的顺序。

就地排序函数的名称与应用于只读列表的函数的名称相似，但没有 ed/d 后缀：

sort\* 在所有排序函数的名称中代替 sorted\*：[sort()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/sort.html)、[sortDescending()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/sort-descending.html)、[sortBy()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/sort-by.html) 等等。

[shuffle()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/shuffle.html) 代替 shuffled()。

[reverse()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/reverse.html) 代替 reversed()。

[asReversed()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/as-reversed.html) 在可变列表上调用会返回另一个可变列表，该列表是原始列表的反向视图。在该视图中的更改将反映在原始列表中。 以下示例展示了可变列表的排序函数：

val numbers = mutableListOf("one", "two", "three", "four")

​

numbers.sort()

println("Sort into ascending: $numbers")

numbers.sortDescending()

println("Sort into descending: $numbers")

​

numbers.sortBy { it.length }

println("Sort into ascending by length: $numbers")

numbers.sortByDescending { it.last() }

println("Sort into descending by the last letter: $numbers")

​

numbers.sortWith(compareBy<String> { it.length }.thenBy { it })

println("Sort by Comparator: $numbers")

​

numbers.shuffle()

println("Shuffle: $numbers")

​

numbers.reverse()

println("Reverse: $numbers")

运行结果：

Sort into ascending: [four, one, three, two]

Sort into descending: [two, three, one, four]

Sort into ascending by length: [two, one, four, three]

Sort into descending by the last letter: [four, two, one, three]

Sort by Comparator: [one, two, four, three]

Shuffle: [three, four, one, two]

Reverse: [two, one, four, three]

#### 3.6.6.13 Set相关操作

Kotlin 集合包中包含 set 常用操作的扩展函数：**查找交集、并集或差集**。

**要将两个集合合并为一个（并集），可使用 [union()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/union.html) 函数。**

**也能以中缀形式使用 a union b。**

注意，对于有序集合，操作数的顺序很重要：在结果集合中，左侧操作数在前。

**要查找两个集合中都存在的元素（交集），请使用 [intersect()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/intersect.html) 。**

**要查找另一个集合中不存在的集合元素（差集），请使用 [subtract()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/subtract.html) 。**

**差集 a subtract b 只会返回 b中不存在的 a的元素！**

**这两个函数也能以中缀形式调用，例如， a intersect b 。**

val numbers = setOf("one", "two", "three")  
  
println(numbers union setOf("four", "five"))  
println(setOf("four", "five") union numbers)  
  
println(numbers intersect setOf("two", "one"))  
println(numbers subtract setOf("three", "four"))  
**println(numbers subtract setOf("four", "three")) //比较用例**println(numbers subtract setOf("four", "three"))  
  
**println(setOf("four", "three") subtract numbers)//补充用例**

运行结果：

[one, two, three, four, five]

[four, five, one, two, three]

[one, two]

[one, two]

[one, two]

[one, two]

[four]

注意， List 也支持 Set 操作。 但是，对 List 进行 Set 操作的结果仍然是 Set ，因此将删除所有重复的元素。

#### 3.6.6.14 Map 相关操作

在 [map](https://www.kotlincn.net/docs/reference/collections-overview.html" \l "map) 中，键和值的类型都是用户定义的。 对基于键的访问启用了各种特定于 map 的处理函数，从键获取值到对键和值进行单独过滤。 在此页面上，我们提供了来自标准库的 map 处理功能的描述。

取键与值

要从 Map 中检索值，必须提供其键作为 [get()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-map/get.html) 函数的参数。 还支持简写 [key] 语法。 如果找不到给定的键，则返回 null 。 **还有一个函数 [getValue()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/get-value.html) ，它的行为略有不同：如果在 Map 中找不到键，则抛出异常**。 此外，还有两个选项可以解决键缺失的问题：

[getOrElse()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/get-or-else.html) 与 list 的工作方式相同：对于不存在的键，其值由给定的 lambda 表达式返回。

[getOrDefault()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/get-or-default.html) 如果找不到键，则返回指定的默认值。

val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)

println(numbersMap.get("one"))

println(numbersMap["one"])

println(numbersMap.getOrDefault("four", 10))

println(numbersMap["five"]) // null

//numbersMap.getValue("six")

运行结果：1 1 10 null

要对 map 的所有键或所有值执行操作，可以从属性 keys 和 values 中相应地检索它们。 keys 是 Map 中所有键的集合， values 是 Map 中所有值的集合。

val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)

println(numbersMap.keys)

println(numbersMap.values)

运行结果：

[one, two, three]

[1, 2, 3]

过滤

可以使用 [filter()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/filter.html) 函数来[过滤](https://www.kotlincn.net/docs/reference/collection-filtering.html) map 或其他集合。 **对 map 使用 filter() 函数时， Pair 将作为参数的谓词传递给它**。 它将使用谓词同时过滤其中的键和值。

val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)

val filteredMap = numbersMap.filter { (key, value) -> key.endsWith("1") && value > 10} //返回的是一个map集合

println(filteredMap)

运行结果：{key11=11}

还有两种用于过滤 map 的特定函数：按键或按值。 这两种方式，都有对应的函数： [filterKeys()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/filter-keys.html) 和 [filterValues()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/filter-values.html) 。 两者都将返回一个新 Map ，其中包含与给定谓词相匹配的条目。 filterKeys() 的谓词仅检查元素键， filterValues() 的谓词仅检查值。

val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key11" to 11)

val filteredKeysMap = numbersMap.filterKeys { it.endsWith("1") }

val filteredValuesMap = numbersMap.filterValues { it < 10 }

println(filteredKeysMap)

println(filteredValuesMap)

运行结果：

{key1=1, key11=11}

{key1=1, key2=2, key3=3}

plus 与 minus 操作

由于需要访问元素的键，[plus](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/plus.html)（+）与 [minus](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/minus.html)（-）运算符对 map 的作用与其他集合不同。 plus 返回包含两个操作数元素的 Map ：左侧的 Map 与右侧的 Pair 或另一个 Map 。 当右侧操作数中有左侧 Map 中已存在的键时，该条目将使用右侧的值。

val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)  
println(numbersMap + Pair("four", 4))  
println(numbersMap + Pair("one", 10))  
println(numbersMap + mapOf("five" to 5, "one" to 11))

运行结果：

{one=1, two=2, three=3, four=4}

{one=10, two=2, three=3}

{one=11, two=2, three=3, five=5}

minus 将根据左侧 Map 条目创建一个新 Map ，右侧操作数带有键的条目将被剔除。 因此，右侧操作数可以是单个键或键的集合： list 、 set 等。

val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)  
println(numbersMap - "one")  
println(numbersMap - listOf("two", "four"))

运行结果：{two=2, three=3} {one=1, three=3}

Map 写操作

[Mutable](https://www.kotlincn.net/docs/reference/collections-overview.html" \l "%E9%9B%86%E5%90%88%E7%B1%BB%E5%9E%8B) Map （可变 Map ）提供特定的 Map 写操作。 这些操作使你可以使用键来访问或更改 Map 值。

Map 写操作的一些规则：

**值可以更新。 反过来，键也永远不会改变：添加条目后，键是不变的。**

每个键都有一个与之关联的值。也可以添加和删除整个条目。

下面是对**可变 Map** 中可用写操作的标准库函数的描述。

添加与更新条目

要将新的键值对添加到可变 Map ，请使用 [put()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-map/put.html) 。 将新条目放入 LinkedHashMap （Map的默认实现）后，会添加该条目，以便在 Map 迭代时排在最后。 在 Map 类中，新元素的位置由其键顺序定义。

val numbersMap = mutableMapOf("one" to 1, "two" to 2)

numbersMap.put("three", 3)

println(numbersMap)

运行结果：{one=1, two=2, three=3}

要一次添加多个条目，请使用 [putAll()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/put-all.html) 。它的参数可以是 Map 或一组 Pair ： Iterable 、 Sequence 或 Array 。

val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3)

numbersMap.putAll(setOf("four" to 4, "five" to 5))

println(numbersMap)

运行结果：{one=1, two=2, three=3, four=4, five=5}

如果给定键已存在于 Map 中，则 put() 与 putAll() 都将覆盖值。 因此，可以使用它们来更新 Map 条目的值。

val numbersMap = mutableMapOf("one" to 1, "two" to 2)

**val previousValue = numbersMap.put("one", 11) //返回了修改前的值**

println("value associated with 'one', before: $previousValue, after: ${numbersMap["one"]}")

println(numbersMap)

运行结果：

value associated with 'one', before: 1, after: 11

{one=11, two=2}

还可以使用快速操作符将新条目添加到 Map 。 有两种方式：

[plusAssign](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/plus-assign.html) （+=） 操作符。

[] 操作符为 set() 的别名。

val numbersMap = mutableMapOf("one" to 1, "two" to 2)

numbersMap["three"] = 3 // 调用 numbersMap.set("three", 3) 直接添加

numbersMap += mapOf("four" to 4, "five" to 5)

println(numbersMap)

运行结果：{one=1, two=2, three=3, four=4, five=5}

**使用 Map 中存在的键进行操作时，将覆盖相应条目的值。**

删除条目

要从可变 Map 中删除条目，请使用 [remove()](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-mutable-map/remove.html) 函数。 调用 remove() 时，可以传递键或整个键值对。 如果同时指定键和值，则仅当键值都匹配时，才会删除此的元素。

val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3)

numbersMap.remove("one")

println(numbersMap)

numbersMap.remove("three", 4) //不会删除任何条目

println(numbersMap)

运行结果：{one=1, two=2, three=3, four=4, five=5}

还可以通过键或值从可变 Map 中删除条目。 在 Map 的 .keys 或 .values 中调用 remove() 并提供键或值来删除条目。 在 .values 中调用时， remove() 仅删除给定值匹配到的的第一个条目。

val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3, "threeAgain" to 3)

numbersMap.keys.remove("one")

println(numbersMap)

numbersMap.values.remove(3)

println(numbersMap)

运行结果：

{two=2, three=3, threeAgain=3}

{two=2, threeAgain=3}

[minusAssign](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/minus-assign.html) （-=） 操作符也可用于可变 Map 。

val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3)

numbersMap -= "two"

println(numbersMap)

numbersMap -= "five" //不会删除任何条目

println(numbersMap)

运行结果：

{one=1, three=3}

{one=1, three=3}

### 3.6.7 集合拓展方法速查表

基本集合：List Set Map Sequence

基于：\_Collections.kt的拓展方法速查表

|  |  |  |  |
| --- | --- | --- | --- |
| 方法名称 | 参数 | 返回值 | 说明 |
| contains | (element: E) | Boolean | 集合是否包含传参元素 |
| containsAll | (elements: Collection<E>) | Boolean | 集合是否包含传参集合 |
| elementAt | (index: Int) | T | Iterable的子类获取指定位置的元素 |
| first | () | T | Iterable的子类取第一个元素 |
| first | (predicate: (T) -> Boolean) | T | Iterable的子类按条件从前往后，取第一个满足条件元素，如果存在返回元素，不存在报异常 |
| firstOrNull/find  互为别名方法 | (predicate: (T) -> Boolean) | T | 逻辑和上面相同，不存在时会返回null |
| last | () | T | Iterable的子类取最后一个元素 |
| last | (predicate: (T) -> Boolean) | T | Iterable的子类按条件从后往前取第一个满足条件的元素，如果存在返回元素，不存在报异常 |
| [lastOrNull](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/last-or-null.html)/findLast  互为别名方法 | (predicate: (T) -> Boolean) |  | 逻辑和上面相同，不存在时会返回null |
| [elementAtOrNull](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/element-at-or-null.html) | (index: Int) | T | Iterable的子类获取指定位置的元素,不存在时返回null。**推荐使用！** |
| elementAtOrElse | (index: Int,  defaultValue: (Int) -> T) | T | Iterable的子类获取指定位置的元素,不存在时走方法参数。**推荐使用！** |
| filter | 过滤 |  |  |
| filterIndex | 有索引的过滤 |  | 过滤条件Index和value |
| filterNot | 使用否定条件来过滤集合 |  |  |
| filterIsInstance | filterIsInstance<T>() |  | 返回一个 List<T> |
| [partition](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/partition.html) | (predicate: (T) -> Boolean) | Pair<List<T>, List<T>> | 根据条件，将一个集合分成两个集合，返回一个含有两个集合的Pair |
| map | 映射 |  | 不过滤空，mapIndex也不过滤空 |
| mapIndex | 有索引映射 |  | 映射回调两个参数index,value |
| [mapNotNull](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/map-not-null.html) | 过滤空的映射 |  |  |
| [mapIndexedNotNull](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/map-indexed-not-null.html) | 有索引的过滤空映射 |  |  |
| assoicate | 关联，List 关联后形成Map  (transform: (T) -> Pair<K, V>) |  | Assoicate开头的核心就是形成map |
| associateWith | (valueSelector: (K) -> V) |  | 此时元素作为value |
| associateBy | (keySelector: (T) -> K) |  | 此时元素作为key,**如果两个元素相等，则仅最后一个保留在 Map 中** |
| associateBy | (keySelector: (T) -> K, valueTransform: (T) -> V) |  | 双参数时，是一个返回key，一个返回value |
| Iterator | 游标 |  | 仅可以下移，删除。用完不可在用。 |
| ListIterator | 双向游标 |  | 可前可后，可增删改，用完可以继续用。 |
| forEach | 遍历 |  |  |
| take | (n: Int) | List<T> | 取出前n个元素 |
| takeLast | (n: Int) | List<T> | 从后往前取n个元素 |
| drop | (n: Int) | List<T> | 移除前n个元素 |
| dropLast | (n: Int) | List<T> | 从后往前移除n个元素 |
| takeWhile | (predicate: (T) -> Boolean) | List<T> | 按条件从前往后取元素  此处仅以takeWhile举例  还有takeLastWhile、dropWhile、dropLastWhile意思相近 |
| chunked | (size: Int) | List<List<T>> | 按给的大小分组，最后一个组可能小于给定的大小 |
| chunked | (size: Int, transform: (List<T>) -> R) | List<R> | 按给定的大小分组，并对组进行操作 |
| windowed | (size: Int, step: Int = 1, partialWindows: Boolean = false) | List<List<T>> | size代表分组的大小，step代表每个组的第一个元素直接间隔的元素，partialWindows代表不满size时是否继续取元素 |
| zip 二元运算符 | 合拢，**返回Pair 对象的列表**。 |  | 根据两个集合中具有相同位置的元素构建配对。 |
| Zip(i:Iterable<T>) | 合拢，返回Iterable<V> |  | 同时操作相同位置的元素对 |
| unzip | Pair<List<T>, List<R>> |  |  |
| flatten | 打平，操作嵌套的集合 |  | 该函数返回嵌套集合中的所有元素的一个 List |
| flatMap | 返回值必须是Iterable |  | 可以理解为集合合并，多个集合转换为一个新集合 |
| any | (predicate: (T) -> Boolean) |  | 如果至少有一个元素匹配给定谓词 |
| none | (predicate: (T) -> Boolean) |  | 如果没有元素与给定谓词匹配 |
| all | (predicate: (T) -> Boolean) |  | 如果所有元素都匹配给定谓词 |
| any()  none() | 不带谓词使用 |  | 如果集合中有元素，any() 返回 true，否则返回 false；none() 则相反 |
| [joinToString](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/join-to-string.html) | 将集合转为字符串 |  | 自定义字符串表示形式可选参数：  Separator：分隔符  Prefix：开始  Postfix：结束  Limit:  限制范围  Truncated:超过范围后的显示内容  transform: ((T) -> CharSequence)? = null |
| joinTo | 和[joinToString](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/join-to-string.html)相同的操作 |  | 将结果附加到给定的 [Appendable](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.text/-appendable/index.html) 对象 |
| groupBy | (keySelector: (T) -> K, valueTransform: (T) -> V) |  | 分组第一个条件是key的处理，第二是值的处理，返回Map<K, List<V>> |
| groupingBy | (keySelector: (T) -> K) |  | 一次将操作应用于所有分组 |
| Slice | (indices: IntRange) |  | 按给定的range取出部分元素 |
| Slice | (indices: Iterable<Int>) |  | 按给定的集合取出元素 |
| sortedWith | (comparator: Comparator<in T>) | List<T> | 接收一个自定义Comparator  对象，按规则进行排序 |
| sorted | () | List<T> | 自然升序，用于实现了排序功能的元素 |
| sortedDescending | () | List<T> | 自然降序，用于实现了排序功能的元素 |
| sortedBy | (selector: (T) -> R?) | List<T> | 按方法参数返回的条件进行升序 |
| sortedByDescending | (selector: (T) -> R?) | List<T> | 按方法参数返回的条件进行降序 |
| reversed | () | List<T> | 将现有集合逆序，**用于不可变集合** |
| asReversed | () | List<T> | 将现有集合逆序，用于可变结合，原集合的改变会影响逆序后的集合 |
| shuffled | () | List<T> | 将现有顺序集合随机排序  **会产生相同集合** |
| Collection<T>.count | () | Int | 集合求和 |
| Iterable<T>.maxOrNull | () | T | 找到集合最大值，元素需要可比较 |
| Iterable<T>.minOrNull | () | T | 找到集合最小值，元素需要可比较 |
| Iterable<Int>.average | () | T | 整形集合求平均数 |
| Iterable<Int>.sum | () | Int | 整形集合求和 |
| minByOrNull | (selector: (T) -> R) | T | 按输入条件找到集合最小值 |
| maxWithOrNull | (c: Comparator<in T>) | T | 按给定的比较条件找到最大值 |
| sumBy | (selector: (T) -> Int) | Int | 按给定的条件求和 |
| sumByDouble | (selector: (T) -> Double) | Double | 按给定的条件求 double类型的和 |
| reduce | o: (acc: S, T) -> S | S | 按给定的条件  从集合的第一个元素和第二个元素开始进行操作产生结果，此结果和第三个元素作为条件的第二次入参 |
| fold | (initial: R,  o: (acc: R, T) -> R) | R | 逻辑和reduce一样，区别在于initial为给定的第一个元素，和集合的第一个元素进行运算，产生结果 |
| foldRight | 同fold | R | 和fold相逆的逻辑 |
| reduceRight | 同reduce | S | 和reduce相逆的逻辑 |
| foldIndexed | (initial: R, o: (index: Int, acc: R, T) -> R) | R | 逻辑和Fold一样只是表达式增加了index |
| foldRightIndexed | 同foldIndexed | R | 和foldIndexed逻辑相逆 |
| add | (element: E) | Boolean | 给集合添加一个元素 |
| addAll | (elements: Array<out T>) | Boolean | 给集合添加多个元素 |
| plusAssign | (element: T)  (elements: Iterable<T>)  (elements: Array<T>) | MutableCollection<in T> | 给集合添加一个或多个元素  可以使用 +/+= 代替 plusAssign |
| remove | (element: E) | Boolean | 移除操作 |
| removeAt | (index: Int) | E | 按位置移除第一个出现的元素 |
| removeFirstOrNull | () | (可变集合方法) | 移除集合第一个元素 |
| removeLastOrNull | () | (可变集合方法) | 移除集合最后一个元素 |
| clear | () | Unit | 清空集合操作 |
| removeAll | (elements: Collection<E>) | Boolean | 将现有集合中和传入集合中相同的元素全部删掉 |
| retainAll | (elements: Collection<E>) | Boolean | 仅保留和传入集合相同的元素 |
| minusAssign | (element: T)  (elements: Iterable<T>)  (elements: Array<T>) | MutableCollection<in T> | 去掉现有集合中和传入集合相同的所有元素，可以传入一个或多个 |
| binarySearch | (element: T?,  fromIndex: Int = 0, toIndex: Int = size) | Int | 可以按给定的条件对有序集合进行二分查找 |
| MutableList<T>.fill | (value: T) | (可变集合方法) | 将集合的值变成给定值 |
| Iterable<T>.union | (other: Iterable<T>) | Set<T> | 合集方法可以使用中缀方式 union |
| Iterable<T>.intersect | (other: Iterable<T>) | Set<T> | 交集方法可以使用中缀方式 intersect |
| Iterable<T>.subtract | (other: Iterable<T>) | Set<T> | 差集方法可以使用中缀方式 subtract |
|  |  |  |  |

## 3.7协程

本质上，协程是轻量级的线程。

在协程部分，需要我们整理常用协程函数速查表，和常用协程操作符速查表。

漫画理解协程：

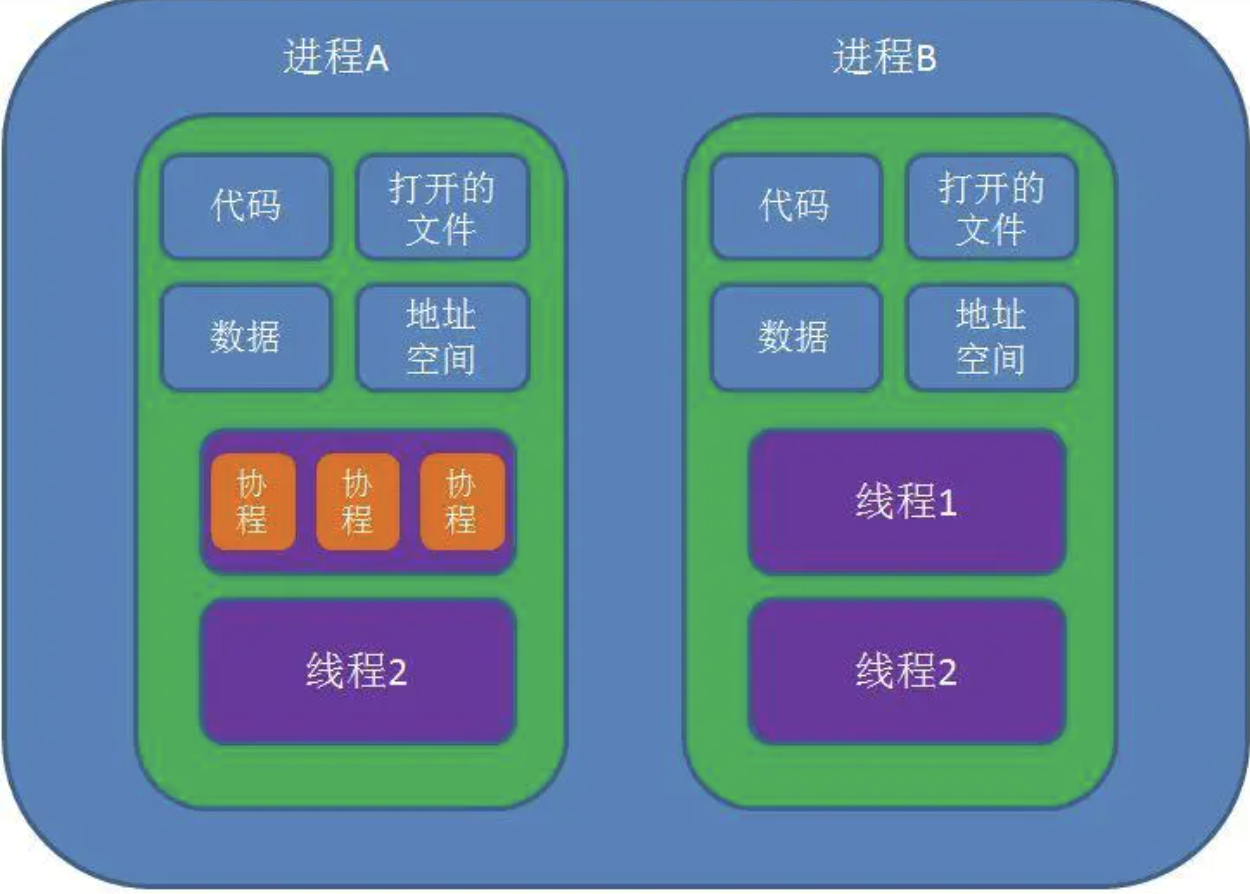
https://www.sohu.com/a/236536167\_684445

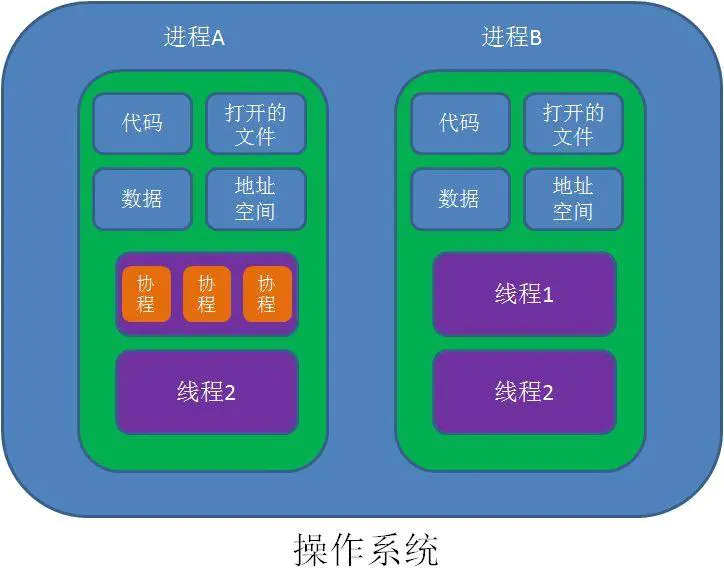
先对协程进行概念性的理解。直接先说区别，Coroutine是编译器级的，Process和Thread是操作系统级的。

线程之间的切换需要深入到内核级别，因此线程的切换代价比较大，表现在：  
\* 线程对象的创建和初始化  
\* 线程上下文切换  
\* 线程状态的切换由系统内核完成  
\* 对变量的操作需要加锁

因为在同一个线程里，协程之间的切换不涉及线程上下文的切换和线程状态的改变，不存在资源、数据并发，所以不用加锁，只需要判断状态就OK，所以执行效率比多线程高很多

· 协程是非阻塞式的(也有阻塞API)，一个协程在进入阻塞后不会阻塞当前线程，当前线程会去执行其他协程任务





理性看待协程，虽然从协程的定义来看非常适合Android开发，因为协程是在线程当中的，又不阻塞线程，那么我们就可以在主线程中启动协程，当协程返回结果时直接刷新ui。

如果这样实现远比我们使用第三方框架然后切换线程要方便的多。

如果将程序分为IO密集型应用和CPU密集型应用，二者的发展历程大致如下：

IO密集型应用: 多进程->多线程->事件驱动->协程  
CPU密集型应用:多进程->多线程

协程主要是让原来要使用“异步+回调方式”写出来的复杂代码, 简化成可以用看似同步的方式写出来（对线程的操作进一步抽象）。这样我们就可以按串行的思维模型去组织原本分散在不同上下文中的代码逻辑，而不需要去处理复杂的状态同步问题。

#### 3.6.0 准备工作

### 搭建协程代码工程

首先，我们来新建一个Kotlin Gradle工程。生成标准gradle工程后，在配置文件build.gradle中，配置kotlinx-coroutines-core依赖：

添加 dependencies :

1. compile 'org.jetbrains.kotlinx:kotlinx-coroutines-core:0.16'

kotlinx-coroutines还提供了下面的模块：

compile group: 'org.jetbrains.kotlinx', name: 'kotlinx-coroutines-jdk8', version: '0.16'

compile group: 'org.jetbrains.kotlinx', name: 'kotlinx-coroutines-nio', version: '0.16'

compile group: 'org.jetbrains.kotlinx', name: 'kotlinx-coroutines-reactive', version: '0.16'

我们使用Kotlin最新的1.1.3-2 版本:

1. buildscript {
2. ext.kotlin\_version = '1.1.3-2'
3. ...
4. dependencies {
5. classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin\_version"
6. }
7. }

其中，kotlin-gradle-plugin是Kotlin集成Gradle的插件。

另外，配置一下JCenter 的仓库:

1. repositories {
2. jcenter()
3. }

### 9.1.2 简单协程示例

下面我们先来看一个简单的协程示例。

运行下面的代码：

1. fun firstCoroutineDemo0() {
2. launch(CommonPool) {
3. delay(3000L, TimeUnit.MILLISECONDS)
4. println("Hello,")
5. }
6. println("World!")
7. Thread.sleep(5000L)
8. }

#### 3.6.1桥接阻塞与非阻塞的世界

Kotlin中提供了runBlocking函数来实现类似主协程的功能：

runBlocking函数来实现类似主协程的功能。 runBlocking函数不是用来当作普通协程函数使用的，它的设计主要是用来桥接普通阻塞代码和挂起风格的（suspending style）的非阻塞代码的

[runBlocking](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/run-blocking.html) 与 [coroutineScope](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/coroutine-scope.html) 可能看起来很类似，因为它们都会等待其协程体以及所有子协程结束。 主要区别在于，[runBlocking](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/run-blocking.html) 方法会阻塞当前线程来等待， 而 [coroutineScope](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/coroutine-scope.html) 只是挂起，会释放底层线程用于其他用途。 由于存在这点差异，[runBlocking](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/run-blocking.html) 是常规函数，而 [coroutineScope](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/coroutine-scope.html) 是挂起函数。

#### 3.6.2 取消与超时

在Java中有两类线程：用户线程 (User Thread)、守护线程 (Daemon Thread)。

所谓守护线程，是指在程序运行的时候在后台提供一种通用服务的线程，比如垃圾回收线程就是一个很称职的守护者，并且这种线程并不属于程序中不可或缺的部分。因此，当所有的非守护线程结束时，程序也就终止了，同时会杀死进程中的所有守护线程。

我们可以看出，活动的协程不会使进程保持活动状态。它们的行为就像守护程序线程。

Join()方法的介绍

kotlin 协程的所有suspend 函数都是可以取消的。我们可以通过job的isActive状态来判断协程的状态，或者检查是否有抛出 CancellationException 时取消。

协程正工作在循环计算中，并且不检查协程当前的状态, 那么调用cancel来取消协程将无法停止协程的运行,我们可以看出，即使我们调用了cancel函数，当前的job状态isAlive是false了，但是协程的代码依然一直在运行，并没有停止。

### 计算代码协程的有效取消

#### 方法一： 显式检查取消状态isActive

此部分可以先介绍 try finally 语法 和 isActive 字段。此部分可以从后往前说，否则会产生疑惑。结合文件操作的案例解释较好。

注意展示cancel 与 cancelAndJoin的区别。

#### 3.6.3 组合挂起函数

此部分可以先介绍measureTimeMillis 方法，避免大家产生理解上的障碍。

在这部分，首先要介绍 suspend 方法和普通方法的区别。

之后介绍同步执行和异步执行（async）。最后介绍一下取消就可以了。

代码块中的delay(3000L, TimeUnit.MILLISECONDS)函数，是一个用suspend关键字修饰的函数，我们称之为挂起函数。挂起函数只能从协程代码内部调用，普通的非协程的代码不能调用。

此处注意介绍 suspend,async（await）;在完成两个方法的介绍后，介绍launch与async的区别。以及返回对象job和De

#### 3.6.4 协程上下文与调度器

此部分较为重要，需要介绍清楚上下文的概念。

协程上下文包含一个 协程调度器 （参见 [CoroutineDispatcher](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-dispatcher/index.html)）它确定了相关的协程在哪个线程或哪些线程上执行。

协程调度器主要是控制我们的协程在哪个线程中运行，那么当我们的协程在非ui线程运行的时候，回调ui线程依然要做线程通信。

带有协程上下文参数的方法，均可以指定协程上下文。

关于非受限调度器和受限调度器这里面要展示Demo体现出，非受限调度器，耗时任务执行前后都在统一线程中，而受限的默认调度器耗时任务执行前后不在统一线程中。

协程的调试正常使用示例展示即可。

在不同线程间跳转，讲完之后要提醒大家：注意，在这个例子中，当我们不再需要某个在 [newSingleThreadContext](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/new-single-thread-context.html) 中创建的线程的时候， 它使用了 Kotlin 标准库中的 use 函数来释放该线程。

#### 3.6.5 子协程父协程与作用域

子协程

当一个协程被其它协程在 [CoroutineScope](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-scope/index.html) 中启动的时候， 它将通过 [CoroutineScope.coroutineContext](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-coroutine-scope/coroutine-context.html) 来承袭上下文，并且这个新协程的 [Job](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-job/index.html) 将会成为父协程作业的 子 作业。**当一个父协程被取消的时候，所有它的子协程也会被递归的取消**。**然而，当使用 [GlobalScope](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-global-scope/index.html) 来启动一个协程时，则新协程的作业没有父作业。 因此它与这个启动的作用域无关且独立运作。**

**父协程**

一个父协程总是等待所有的子协程执行结束。父协程并不显式的跟踪所有子协程的启动，并且不必使用 [Job.join](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-job/join.html) 在最后的时候等待它们。

协程命名-此处可以展示协程的常用方法。也可介绍下运算符重载的概念。

### 3.7 异步流

挂起函数可以异步的返回单个值，但是该如何异步返回多个计算好的值呢？这正是 Kotlin 流（Flow）的用武之地。

注意使用 [Flow](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/-flow/index.html) 的代码与先前示例的下述区别：

* 名为 [flow](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/-flow/index.html) 的 [Flow](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/flow.html) 类型构建器函数。
* flow { ... } 构建块中的代码可以挂起。
* 函数 simple 不再标有 suspend 修饰符。
* **流使用 [emit](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/-flow-collector/emit.html) 函数 发射 值。**
* **流使用 [collect](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/collect.html) 函数 收集 值。**

流式操作，式kotlin中较为优雅的处理复杂遍历的方式。

如我们对一组数据进行遍历，可以看成数据是一个一个像流一样被处理，那么一般我们要做的就是看看是否符合要求（判断逻辑）使用map；对符合要求的数据再进行处理.collect逻辑。

flowOn 操作符 结束后，涉及到生产者和消费者速度协调处理的问题，kotlin封装了我们常见的业务逻辑，其实都是有迹可循的，我们可以结合实际用例，方便大家对这些操作服的理解。

缓冲(可以理解为下载功能)

合并conflate(可以理解为视频流抛弃)

collectLatest 输入都会进入到处理函数，但是新的输入会导致旧的未完成的输入结束云南行，并开始新的输入的运行。停止处理当前的任务 立即开始新任务的处理

合并 可根据 代码实现后的命令行输出 比较zip和combine的区别。

简单来说zip是等两个结果都符合的时候再继续进行合并处理。

而combine类似与缓存了结果，当有新结果时和当前的老结果进行合并处理。

展平流 一个流程的结束是下一个流程的开始。

一个流的结束可以看成是一个流返回返回结果了，另外一个流接收到此返回结果进行他的流程处理。当然他也可能是一个流。这种反而更贴近现实的业务模型。

### 3.8通道-生产者消费者-管道

#### **3.8.1通道**

延迟对象提供了一种在协程之间传输单个值的方法。而通道（Channel）提供了一种传输数据流的方法。通道是使用 SendChannel 和使用 ReceiveChannel 之间的非阻塞通信。

**通道跟阻塞队列一个关键的区别是：通道有挂起的操作, 而不是阻塞的, 同时它可以关闭。**

**这里可以先说明通道的模型是生产者与消费者模型。**

**通道是较为容易理解的。可以一句话概括为，缓存为一的非阻塞生产者消费者模型。**

关闭-和队列不同，**一个通道可以通过被关闭来表明没有更多的元素将会进入通道**。 **在接收者中可以定期的使用 for 循环来从通道中接收元素**。

#### f3.8.2构建通道生产者

使用协程生成元素序列的模式非常常见。这是在并发代码中经常有的生产者-消费者模式。

#### 3.8.3 管道

管道（Pipeline）是一种模式, **我们可以用一个协程生产无限序列**。

可以理解为，通道是封装好入口和出口的一个异步队列。

而管道是我们可以串联起来进行数据流转的一个工具。

此部分目前示例代码资料较少，介绍即可。

这里面有一个核心逻辑即管道串联后，最后一个管道调用receive的时候整条管道会从管道入口开始接收一遍。

这也就是给出的例子中，参与到循环的数字，逐步递增的原因。

#### 3.8.4 扇出与扇入

扇出与扇入的意思是一个管道可以介入多个输出，一个管道同时可以作为多个输入。

### 3.9协程方法速查表

|  |  |  |
| --- | --- | --- |
| 协程方法名 | 是否阻塞 | 描述 |
| launch | 否 | launch函数它以非阻塞（non-blocking）当前线程的方式，启动一个新的协程后台任务，并返回一个Job类型的对象作为当前协程的引用。 |
| runBlocking | 否 | runBlocking函数来实现类似主协程的功能。 runBlocking函数不是用来当作普通协程函数使用的，它的设计主要是用来桥接普通阻塞代码和挂起风格的（suspending style）的非阻塞代码的 |
| withTimeout | 否 | 设置超时的函数，方法体里面实现耗时操作。 |
| Suspend 修饰方法 | 是 | 这个执行可以看成线性执行。 |
| measureTimeMillis | 是 | 和内容有关，包裹代码块。计算方法体内整体的运行时间。需要方法体内运行结束。 |
| async | 否 | 方法体，将同步方法变成异步执行.  通过返回对象的await() 等待执行完毕 拿到运行结果 |

|  |  |
| --- | --- |
| 参数 | 说明 |
| context | 协程上下文 |
| Start | 协程启动选项 |
| Block | 协程真正要执行的代码块，必须是suspend修饰的刮起函数 |
| Job(返回类型) | 这个launch函数返回一个Job类型，Job是协程创建的后台任务的概念，它持有该协程的引用。Job接口实际上继承自CoroutineContext类型。一个Job有如下三种状态  截屏2021-01-23 下午4.37.14 |
| CommonPool：共享线程池 | CommonPool是代表共享线程池，它的主要作用是用来调度计算密集型任务的协程的执行。这个CommonPool是代表共享线程池，它的主要作用是用来调度计算密集型任务的协程的执行。它的实现使用的是java.util.concurrent包下面的API。它首先尝试创建一个java.util.concurrent.ForkJoinPool（ForkJoinPool是一个可以执行ForkJoinTask的ExcuteService，它采用了work-stealing模式：所有在池中的线程尝试去执行其他线程创建的子任务，这样很少有线程处于空闲状态，更加高效）；如果不可用，就使用java.util.concurrent.Executors来创建一个普通的线程池：Executors.newFixedThreadPool。 |
| Deferred | public interface Deferred<out T> : Job  截屏2021-01-27 下午2.57.11  截屏2021-01-27 下午2.57.23 |

## 3.10 协程与线程比较

直接先说区别，**协程是编译器级的，而线程是操作系统级的**。

协程通常是由编译器来实现的机制。线程看起来也在语言层次，但是内在原理却是操作系统先有这个东西，然后通过一定的API暴露给用户使用，两者在这里有不同。

协程就是用户空间下的线程。用协程来做的东西，用线程或进程通常也是一样可以做的，但往往多了许多加锁和通信的操作。

**线程是抢占式，而协程是非抢占式的，所以需要用户自己释放使用权来切换到其他协程，因此同一时间其实只有一个协程拥有运行权，相当于单线程的能力。**

协程并不是取代线程, 而是抽象于线程之上, 线程是被分割的CPU资源, 协程是组织好的代码流程, 协程需要线程来承载运行, 线程是协程的资源, 但协程不会直接使用线程, 协程直接利用的是执行器(Interceptor), 执行器可以关联任意线程或线程池, 可以使当前线程, UI线程, 或新建新程.。

线程是协程的资源。协程通过Interceptor来间接使用线程这个资源。

## 9.14 协程的好处

与多线程、多进程等并发模型不同，**协程依靠user-space调度，而线程、进程则是依靠kernel来进行调度**。**线程、进程间切换都需要从用户态进入内核态，而协程的切换完全是在用户态完成，且不像线程进行抢占式调度，协程是非抢占式的调度。**

通常多个运行在同一调度器中的协程运行在一个线程内，这也消除掉了多线程同步等带来的编程复杂性。同一时刻同一调度器中的协程只有一个会处于运行状态。

我们使用协程，程序只在用户空间内切换上下文，不再陷入内核来做线程切换，这样可以避免大量的用户空间和内核空间之间的数据拷贝，降低了CPU的消耗，从而大大减缓高并发场景时CPU瓶颈的窘境。

另外，使用协程，我们不再需要像异步编程时写那么一堆callback函数，代码结构不再支离破碎，整个代码逻辑上看上去和同步代码没什么区别，简单，易理解，优雅。

我们使用协程，我们可以很简单地实现一个可以随时中断随时恢复的函数。

**一些 API 启动长时间运行的操作(例如网络 IO、文件 IO、CPU 或 GPU 密集型任务等)，并要求调用者阻塞直到它们完成。协程提供了一种避免阻塞线程并用更廉价、更可控的操作替代线程阻塞的方法：协程挂起。**

协程通过将复杂性放入库来简化异步编程。程序的逻辑可以在协程中顺序地表达，而底层库会为我们解决其异步性。该库可以将用户代码的相关部分包装为回调、订阅相关事件、在不同线程(甚至不同机器)上调度执行，而代码则保持如同顺序执行一样简单。

### 3.11 阻塞 vs 挂起

协程可以被挂起而无需阻塞线程。而线程阻塞的代价通常是昂贵的，尤其在高负载时，阻塞其中一个会导致一些重要的任务被延迟。

另外，协程挂起几乎是无代价的。不需要上下文切换或者 OS 的任何其他干预。

最重要的是，挂起可以在很大程度上由用户来控制，我们可以决定挂起时做些，并根据需求优化、记日志、拦截处理等。

## 9.15 协程的内部机制

### 9.15.1 基本原理

协程完全通过编译技术实现(不需要来自 VM 或 OS 端的支持)，挂起机制是通过状态机来实现，其中的状态对应于挂起调用。

在挂起时，对应的协程状态与局部变量等一起被存储在编译器生成的类的字段中。在恢复该协程时，恢复局部变量并且状态机从挂起点接着后面的状态往后执行。

挂起的协程，是作为Continuation对象来存储和传递，Continuation中持有协程挂起状态与局部变量。

关于协程工作原理的更多细节可以在这个[设计文档](https://github.com/Kotlin/kotlin-coroutines/blob/master/kotlin-coroutines-informal.md)中找到。

### 9.15.2 标准 API

协程有三个主要组成部分：

* 语言支持(即如上所述的挂起功能)，
* Kotlin 标准库中的底层核心 API，
* 可以直接在用户代码中使用的高级 API。
* 底层 API：kotlin.coroutines

底层 API 相对较小，并且除了创建更高级的库之外，不应该使用它。 它由两个主要包组成：

kotlin.coroutines.experimental 带有主要类型与下述原语：

* createCoroutine()
* startCoroutine()
* suspendCoroutine()

kotlin.coroutines.experimental.intrinsics 带有甚至更底层的内在函数如 ：

* suspendCoroutineOrReturn()

大多数基于协程的应用程序级API都作为单独的库发布：kotlinx.coroutines。这个库主要包括下面几大模块：

* 使用 kotlinx-coroutines-core 的平台无关异步编程
* 基于 JDK 8 中的 CompletableFuture 的 API：kotlinx-coroutines-jdk8
* 基于 JDK 7 及更高版本 API 的非阻塞 IO(NIO)：kotlinx-coroutines-nio
* 支持 Swing (kotlinx-coroutines-swing) 和 JavaFx (kotlinx-coroutines-javafx)
* 支持 RxJava：kotlinx-coroutines-rx

这些库既作为使通用任务易用的便利的 API，也作为如何构建基于协程的库的端到端示例。关于这些 API 用法的更多细节可以参考相关文档。