【Kotlin】Collections (官方文件)

來源: https://kotlinlang.org/docs/collections-overview.html

Collections overview

Kotlin標準庫提供了一套全面的工具來管理集合——這些集合是問題解決中重要且經常操作的變數數量（可能為零）的項目群組。

集合是大多數編程語言中的常見概念，所以如果你熟悉例如Java或Python的集合，可以跳過這個介紹，直接進入詳細章節。

一個集合通常包含相同類型（及其子類型）的若干對象。集合中的對象稱為元素或項目。例如，一個系所的所有學生形成一個集合，可以用來計算他們的平均年齡。

以下是Kotlin中相關的集合類型：

- List是一個有序集合，可以通過索引——反映其位置的整數——訪問元素。元素在列表中可以出現多次。列表的一個例子是電話號碼：它是一組數字，順序很重要，而且可以重複。
- Set是一個唯一元素的集合。它反映了數學上的集合抽象：一組不重複的對象。一般來說，集合元素的順序沒有意義。例如，彩票上的號碼形成一個集合：它們是唯一的，順序並不重要。
- Map（或字典）是一組鍵值對。鍵是唯一的，每個鍵映射到一個確切的值。值可以重複。Map用於存儲對象之間的邏輯連接，例如員工的ID和他們的職位。

Kotlin允許你在操作集合時，不用考慮存儲在其中的對象的確切類型。換句話說，向字符串列表中添加字符串的方式與向整數列表或自定義類別列表中添加元素的方式相同。因此，Kotlin標準庫提供了泛型介面、類別和函數，用於創建、填充和管理任何類型的集合。

集合介面和相關函數位於kotlin.collections package中。我們來概覽一下其內容。

構建集合

從元素構建

創建集合最常見的方法是使用標準庫函數`listOf<T>()`、`setOf<T>()`、`mutableListOf<T>()`、`mutableSetOf<T>()`。如果你提供一個以逗號分隔的集合元素列表作為參數，編譯器會自動檢測元素類型。在創建空集合時，需要明確指定類型。

```
val numbersSet = setOf("one", "two", "three", "four")
val emptySet = mutableSetOf<String>()
```

同樣的功能也適用於map，使用函數`mapOf()`和`mutableMapOf()`。map的鍵和值是以Pair對象傳遞的（通常通過`to`中綴函數創建）。

```
val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key4" to 1)
```

需要注意的是，`to`表示法會創建一個短暫存在的Pair對象，所以建議僅在性能不是關鍵時使用它。為了避免過多的記憶體使用，可以使用其他方式。例如，可以創建一個可變map並使用寫操作來填充它。`apply()`函數可以幫助保持初始化的流暢性。

```
val numbersMap = mutableMapOf<String, String>().apply {
    this["one"] = "1"
    this["two"] = "2"
}
```

使用集合構建函數創建

另一種創建集合的方法是調用構建函數——`buildList()`、`buildSet()`或`buildMap()`。它們創建一個新的、可變的相應類型的集合，使用寫操作填充它，然後返回包含相同元素的只讀集合：

```
val map = buildMap {
    put("a", 1)
    put("b", 0)
    put("c", 4)
}

println(map) // {a=1, b=0, c=4}
```

空集合

也有一些函數可以創建沒有任何元素的集合：`emptyList()`、`emptySet()`和`emptyMap()`。在創建空集合時，應該明確指定集合將持有的元素類型。

```
val empty = emptyList<String>()
```

List的初始化函數

對於list，有一個類似構造函數的函數，它接受列表大小和基於索引定義元素值的初始化函數。

```
val doubled = List(3) { it * 2 }
println(doubled)
// [0, 2, 4]
```

具體類型構造函數

要創建具體類型的集合，例如`ArrayList`或`LinkedList`，可以使用這些類型的可用構造函數。對於`Set`和`Map`的實現也有類似的構造函數。

```
val linkedList = LinkedList<String>(listOf("one", "two", "three"))
val presizedSet = HashSet<Int>(32)
```

複製

要創建一個包含與現有集合相同元素的集合，可以使用複製函數。標準庫的集合複製函數會創建淺層複製集合，引用相同的元素。因此，對集合元素所做的更改會反映在所有其副本中。

集合複製函數，如`toList()`、`toMutableList()`、`toSet()`等，會在特定時刻創建集合的快照。其結果是一個包含相同元素的新集合。如果從原集合中添加或移除元素，這不會影響副本。副本可以獨立於源進行更改。

```
val alice = Person("Alice")
val sourceList = mutableListOf(alice, Person("Bob"))
val copyList = sourceList.toList()
sourceList.add(Person("Charles"))
alice.name = "Alicia"
println("First item's name is: ${sourceList[0].name} in source and ${copyList[0].name} in copy")
println("List size is: ${sourceList.size} in source and ${copyList.size} in copy")

// First item's name is: Alicia in source and Alicia in copy
// List size is: 3 in source and 2 in copy
```

這些函數也可以用來將集合轉換為其他類型，例如從列表構建集合，反之亦然。

```
val sourceList = mutableListOf(1, 2, 3)
val copySet = sourceList.toMutableSet()
copySet.add(3)
copySet.add(4)
println(copySet)
 // [1, 2, 3, 4]
```

另外，可以創建對同一集合實例的新引用。當你使用現有集合初始化一個集合變量時，就會創建新引用。因此，當通過某個引用修改集合實例時，這些更改會反映在所有引用中。

```
val sourceList = mutableListOf(1, 2, 3)
val referenceList = sourceList
referenceList.add(4)
println("Source size: ${sourceList.size}") // Source size: 4
```

集合初始化可以用來限制可變性。例如，如果你創建一個`List`引用到一個`MutableList`，編譯器會在你試圖通過此引用修改集合時產生錯誤。

```
val sourceList = mutableListOf(1, 2, 3)
val referenceList: List<Int> = sourceList
//referenceList.add(4)           //compilation error
sourceList.add(4)
println(referenceList) // shows the current state of sourceList
```

調用其他集合上的函數

集合可以作為對其他集合進行各種操作的結果來創建。例如，過濾列表會創建一個包含匹配過濾條件的元素的新列表：

```
val numbers = listOf("one", "two", "three", "four")
val longerThan3 = numbers.filter { it.length > 3 }
println(longerThan3)
// [three, four]
```

map 會根據轉換結果產生一個列表：

```
val numbers = setOf(1, 2, 3)
println(numbers.map { it * 3 })
println(numbers.mapIndexed { idx, value -> value * idx })
// [3, 6, 9]
// [0, 2, 6]
```

associate 產生maps：

```
val numbers = listOf("one", "two", "three", "four")
println(numbers.associateWith { it.length })

 // {one=3, two=3, three=5, four=4}
```

有關Kotlin中集合操作的更多信息，請參見集合操作概述。

迭代器(Iterator)

為了遍歷集合元素，Kotlin標準庫支持常用的迭代器機制——這些對象可以順序地訪問元素而不暴露集合的底層結構。當你需要一個一個地處理集合中的所有元素時，迭代器非常有用，例如，打印值或進行類似的更新。

迭代器可以通過調用`iterator()`函數從`Iterable<T>`介面的繼承者（包括Set和List）獲得。

一旦獲得迭代器，它會指向集合的第一個元素；調用`next()`函數返回該元素並將迭代器位置移動到下一個元素（如果存在的話）。

一旦迭代器遍歷了最後一個元素，它將不能再用於檢索元素；也不能將其重置到任何先前的位置。要再次遍歷集合，需要創建一個新的迭代器。

```
val numbers = listOf("one", "two", "three", "four")
val numbersIterator = numbers.iterator()
while (numbersIterator.hasNext()) {
    println(numbersIterator.next())
    // one
    // two
    // three
    // four
}
```

另一種遍歷`Iterable`集合的方法是眾所周知的`for`循環。當在集合上使用`for`時，你會隱式地獲得迭代器。因此，下面的代碼等同於上面的例子：

```
val numbers = listOf("one", "two", "three", "four")
for (item in numbers) {
    println(item)
    // one
    // two
    // three
    // four
}
```

最後，有一個有用的`forEach()`函數，讓你可以自動迭代集合並為每個元素執行給定的代碼。因此，相同的例子看起來如下：

```
val numbers = listOf("one", "two", "three", "four")
numbers.forEach {
    println(it)
    // one
    // two
```

```
    // three
    // four
}
```

列表迭代器

對於列表，有一個特殊的迭代器實現：`ListIterator`。它支持雙向迭代列表：向前和向後。

向後迭代通過`hasPrevious()`和`previous()`函數實現。此外，`ListIterator`還提供有關元素索引的信息，通過`nextIndex()`和`previousIndex()`函數。

```
val numbers = listOf("one", "two", "three", "four")
val listIterator = numbers.listIterator()
while (listIterator.hasNext()) listIterator.next()
println("Iterating backwards:")
// Iterating backwards:
while (listIterator.hasPrevious()) {
    print("Index: ${listIterator.previousIndex()}")
    println(", value: ${listIterator.previous()}")
    // Index: 3, value: four
    // Index: 2, value: three
    // Index: 1, value: two
    // Index: 0, value: one
}
```

能夠雙向迭代意味著`ListIterator`在到達最後一個元素後仍然可以使用。

可變迭代器

對於迭代可變集合，有`MutableIterator`，它擴展了`Iterator`並增加了元素移除函數`remove()`。因此，你可以在迭代集合時移除元素。

```
val numbers = mutableListOf("one", "two", "three", "four")
val mutableIterator = numbers.iterator()

mutableIterator.next()
mutableIterator.remove()
println("After removal: $numbers")
// After removal: [two, three, four]
```

除了移除元素外，`MutableListIterator`還可以在迭代列表時插入和替換元素，通過使用`add()`和`set()`函數。

```
val numbers = mutableListOf("one", "four", "four")
val mutableListIterator = numbers.listIterator()

mutableListIterator.next()
mutableListIterator.add("two")
println(numbers)
// [one, two, four, four]
mutableListIterator.next()
mutableListIterator.set("three")
println(numbers)
// [one, two, three, four]
```

範圍與級數

Kotlin允許你使用`kotlin.ranges`包中的`.rangeTo()`和`.rangeUntil()`函數輕鬆創建值的範圍。

要創建：

- 閉合範圍，使用`..`運算符調用`.rangeTo()`函數。
- 開放範圍，使用`..<`運算符調用`.rangeUntil()`函數。

例如：

```
// 閉合範圍
println(4 in 1..4)  // true
```

```
// 開放範圍
println(4 in 1..<4)  // false
```

範圍特別適用於在`for`循環中進行迭代：

```
for (i in 1..4) print(i)
// 1234
```

要反向迭代數字，可以使用`downTo`函數代替`..`：

```
for (i in 4 downTo 1) print(i)
// 4321
```

也可以使用`step`函數以任意步長（不一定是1）迭代數字：

```
for (i in 0..8 step 2) print(i)
println()
// 02468

for (i in 0..<8 step 2) print(i)
println()
// 0246

for (i in 8 downTo 0 step 2) print(i)
// 86420
```

級數

整數類型（如Int、Long和Char）的範圍可以視為算術級數。在Kotlin中，這些級數由特定類型定義：`IntProgression`、`LongProgression`和`CharProgression`。

級數有三個基本屬性：第一個元素、最後一個元素和非零步長。第一個元素是`first`，隨後的元素是前一個元素加上步長。帶有正步長的級數迭代等同於Java/JavaScript中的索引`for`循環。

```
for (int i = first; i <= last; i += step) {
  // ...
}
```

當你通過迭代範圍隱式地創建級數時，這個級數的第一個和最後一個元素是範圍的端點，步長為1。

```
for (i in 1..10) print(i)
// 12345678910
```

要定義自訂的級數步長，可以在範圍上使用`step`函數。

```
for (i in 1..8 step 2) print(i)
// 1357
```

級數的最後一個元素計算方法如下：

- 對於正步長：不大於結束值的最大值，使得`(last - first) % step == 0`。
- 對於負步長：不小於結束值的最小值，使得`(last - first) % step == 0`。

因此，最後一個元素不一定是指定的結束值。

```
for (i in 1..9 step 3) print(i) // 最後一個元素是7
// 147
```

級數實現了`Iterable<N>`，其中`N`分別是`Int`、`Long`或`Char`，所以你可以在各種集合函數中使用它們，如`map`、`filter`等。

```
println((1..10).filter { it % 2 == 0 })
// [2, 4, 6, 8, 10]
```

序列（Sequence）

除了集合，Kotlin標準庫還包含另一種類型——序列（Sequence）。與集合不同，序列不包含元素，而是在迭代時生成它們。序列提供與`Iterable`相同的函數，但實現了另一種多步集合處理的方法。

當對`Iterable`進行多步處理時，這些步驟是急切執行的：每個處理步驟完成並返回其結果——一個中間集合。下一步在這個集合上執行。相反，序列的多步處理在可能的情況下是懶惰執行的：實際計算僅在請求整個處理鏈的結果時發生。

操作執行的順序也不同：序列對每個元素依次執行所有處理步驟。相反，`Iterable`完成整個集合的每一步，然後進入下一步。

因此，序列使你可以避免構建中間步驟的結果，從而提高整個集合處理鏈的性能。然而，序列的懶惰性質增加了一些開銷，這在處理較小的集合或進行簡單計算時可能是顯著的。因此，你應該同時考慮序列和`Iterable`，並決定哪個更適合你的情況。

創建

從元素創建

要創建序列，請調用`sequenceOf()`函數並列出其參數中的元素。

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

從`Iterable`創建

如果你已經有一個`Iterable`對象（例如List或Set），可以通過調用`asSequence()`從中創建序列。

```
val numbers = listOf("one", "two", "three", "four")
val numbersSequence = numbers.asSequence()
```

從函數創建(from chunks)

另一種創建序列的方法是使用計算其元素的函數來構建它。要基於函數創建序列，請調用`generateSequence()`並將此函數作為參數。可以選擇性地將第一個元素指定為明確的值或函數調用的結果。當提供的函數返回`null`時，序列生成停止。因此，下面示例中的序列是無限的。

```
val oddNumbers = generateSequence(1) { it + 2 } // `it` 是前一個元素
println(oddNumbers.take(5).toList())
// println(oddNumbers.count())     // 錯誤：序列是無限的
```

要使用`generateSequence()`創建有限序列，請提供在最後一個元素之後返回`null`的函數。

```
val oddNumbersLessThan10 = generateSequence(1) { if (it < 8) it + 2 else null }
println(oddNumbersLessThan10.count())
```

從塊創建

最後，有一個函數允許你逐個或按任意大小的塊生成序列元素——`sequence()`函數。該函數接受包含`yield()`和`yieldAll()`函數調用的lambda表達式。它們將元素返回給序列消費者，並掛起`sequence()`的執行，直到消費者請求下一個元素。`yield()`以單個元素作為參數；`yieldAll()`可以接受`Iterable`對象、迭代器或另一個序列。`yieldAll()`的序列參數可以是無限的。然而，這樣的調用必須是最後一個：所有後續調用將永遠不會執行。

```
val oddNumbers = sequence {
    yield(1)
    yieldAll(listOf(3, 5))
    yieldAll(generateSequence(7) { it + 2 })
}
println(oddNumbers.take(5).toList())
```

序列操作

序列操作根據其狀態要求分為以下幾類：

- 無狀態操作：不需要狀態並獨立處理每個元素，例如`map()`或`filter()`。無狀態操作也可以需要少量常數狀態來處理元素，例如`take()`或`drop()`。
- 有狀態操作：需要大量狀態，通常與序列中元素的數量成正比。

如果序列操作返回另一個序列，該序列是懶惰生成的，則稱為中間操作。否則，該操作是終端操作。例如，終端操作包括`toList()`或`sum()`。序列元素只能通過終端操作檢索。

序列可以多次迭代；但是某些序列實現可能會限制自己只能迭代一次。在它們的文檔中會特別提到這一點。

序列處理示例

讓我們通過一個例子來看看`Iterable`和序列之間的區別。

Iterable

假設你有一個單詞列表。下面的代碼過濾出長度超過三個字符的單詞，並打印出前四個這樣的單詞的長度。

```
val words = "The quick brown fox jumps over the lazy dog".split(" ")
val lengthsList = words.filter { println("filter: $it"); it.length > 3 }
    .map { println("length: ${it.length}"); it.length }
    .take(4)

println("Lengths of first 4 words longer than 3 chars:")
println(lengthsList)

/*
filter: The
filter: quick
filter: brown
filter: fox
filter: jumps
filter: over
filter: the
filter: lazy
filter: dog
length: 5
length: 5
length: 5
length: 4
length: 4
Lengths of first 4 words longer than 3 chars:
[5, 5, 5, 4]
*/
```

運行此代碼時，你會看到`filter()`和`map()`函數按代碼中出現的順序執行。首先，你會看到所有元素的`filter:`，然後是過濾後的元素的`length:`，最後是最後兩行的輸出。

這就是列表處理的方式：

列表處理

序列

現在用序列寫相同的代碼：

```
val words = "The quick brown fox jumps over the lazy dog".split(" ")
// 將List轉換為Sequence
val wordsSequence = words.asSequence()

val lengthsSequence = wordsSequence.filter { println("filter: $it"); it.length > 3 }
    .map { println("length: ${it.length}"); it.length }
    .take(4)

println("Lengths of first 4 words longer than 3 chars:")
// 終端操作：將結果作為List獲取
println(lengthsSequence.toList())
/**
Lengths of first 4 words longer than 3 chars:
filter: The
filter: quick
length: 5
filter: brown
length: 5
filter: fox
filter: jumps
length: 5
filter: over
length: 4
[5, 5, 5, 4]
*/
```

此代碼的輸出顯示，`filter()`和`map()`函數僅在構建結果列表時調用。因此，你首先看到文本行"Lengths of..."，然後序列處理開始。注意，對於過濾後的元素，`map`在過濾下一個元素之前執行。當結果大小達到4時，處理停止，因為這是`take(4)`可以返回的最大大小。

序列處理如下：

序列處理

在此示例中，序列處理用了18步，而使用列表的處理用了23步。

希望這些範例有助於你在開發Kotlin程式時瞭解序列與集合的使用差異及其優劣勢，特別是在處理多步集合操作時。

集合操作概述

Kotlin標準庫提供了多種函數，用於對集合進行操作。這些操作包括簡單的操作（如獲取或添加元素）以及更複雜的操作（如搜索、排序、過濾、轉換等）。

擴展函數與成員函數

集合操作在標準庫中以兩種方式聲明：集合介面的成員函數和擴展函數。

成員函數定義了集合類型所必需的操作。例如，`Collection`包含檢查其是否為空的函數`isEmpty()`；`List`包含用於索引訪問元素的函數`get()`等。

當你創建自己的集合介面實現時，必須實現其成員函數。為了使新實現的創建更容易，可以使用標準庫中的集合介面的骨架實現：`AbstractCollection`、`AbstractList`、`AbstractSet`、`AbstractMap`及其可變對應物。

其他集合操作則聲明為擴展函數。這些包括過濾、轉換、排序和其他集合處理函數。

常用操作

常用操作可用於只讀和可變集合。常用操作分為以下幾組：

- 轉換
- 過濾
- 加號和減號操作符
- 分組
- 檢索集合部分
- 檢索單個元素
- 排序
- 聚合操作

這些頁面上描述的操作返回其結果而不影響原始集合。例如，過濾操作生成一個包含所有匹配過濾條件的元素的新集合。這些操作的結果應該存儲在變量中，或以其他方式使用，例如，傳遞給其他函數。

```
val numbers = listOf("one", "two", "three", "four")
numbers.filter { it.length > 3 }  // numbers沒有任何變化，結果丟失
println("numbers are still $numbers")
val longerThan3 = numbers.filter { it.length > 3 } // 結果存儲在longerThan3中
println("numbers longer than 3 chars are $longerThan3")
```

對於某些集合操作，可以指定目標對象。目標對象是可變集合，函數將其結果項附加到該集合中，而不是返回新的對象。對於具有目標的操作，有帶有`To`後綴的單獨函數名稱，例如`filterTo()`代替`filter()`或`associateTo()`代替`associate()`。這些函數將目標集合作為附加參數。

```
val numbers = listOf("one", "two", "three", "four")
val filterResults = mutableListOf<String>()  // 目標對象
numbers.filterTo(filterResults) { it.length > 3 }
numbers.filterIndexedTo(filterResults) { index, _ -> index == 0 }
println(filterResults) // 包含兩次操作的結果
```

為了方便起見，這些函數返回目標集合，因此你可以在函數調用的相應參數中直接創建它：

```
// 將數字過濾到新的哈希集，從而消除結果中的重複項
val result = numbers.mapTo(HashSet()) { it.length }
println("distinct item lengths are $result")
```

具有目標的函數適用於過濾、關聯、分組、展平和其他操作。完整的目標操作列表請參見Kotlin集合參考。

寫操作

對於可變集合，還有改變集合狀態的寫操作。這些操作包括添加、刪除和更新元素。寫操作列在寫操作以及List特定操作和Map特定操作的對應部分中。

對於某些操作，有成對的函數執行相同的操作：一個在原地應用操作，另一個返回結果作為單獨的集合。例如，`sort()`對可變集合進行原地排序，因此其狀態改變；`sorted()`創建一個包含相同元素的新集合，按排序順序排列。

```
val numbers = mutableListOf("one", "two", "three", "four")
val sortedNumbers = numbers.sorted()
println(numbers == sortedNumbers)  // false
numbers.sort()
println(numbers == sortedNumbers)  // true
```

集合轉換操作

Kotlin標準庫提供了一組用於集合轉換的擴展函數。這些函數根據提供的轉換規則從現有集合構建新集合。以下是可用的集合轉換函數概述。

映射 (Map)

映射轉換通過對另一個集合的元素應用函數來創建集合。基本的映射函數是`map()`。它將給定的lambda函數應用於每個元素，並返回lambda結果的列表。結果的順序與原始元素的順序相同。要應用同時使用元素索引作為參數的轉換，請使用`mapIndexed()`。

```
val numbers = setOf(1, 2, 3)
println(numbers.map { it * 3 })
println(numbers.mapIndexed { idx, value -> value * idx })
```

如果轉換在某些元素上產生null，可以通過調用`mapNotNull()`代替`map()`，或`mapIndexedNotNull()`代替`mapIndexed()`來過濾結果集合中的null。

```
val numbers = setOf(1, 2, 3)
println(numbers.mapNotNull { if (it == 2) null else it * 3 })
println(numbers.mapIndexedNotNull { idx, value -> if (idx == 0) null else value * idx })
```

在轉換地圖時，你有兩個選擇：變換鍵而保持值不變，反之亦然。要對鍵應用給定的轉換，使用`mapKeys()`；反過來，`mapValues()`轉換值。這兩個函數都使用將map條目作為參數的轉換，因此你可以同時操作其鍵和值。

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

拉鍊 (Zip)

拉鍊轉換是從兩個集合中相同位置的元素構建對。在Kotlin標準庫中，這是通過`zip()`擴展函數完成的。

當在一個集合或數組上調用並將另一個集合（或數組）作為參數時，`zip()`返回一個`Pair`對象的列表。接收集合的元素是這些對中的第一個元素。

如果集合大小不同，`zip()`的結果是較小的大小；較大集合的最後元素不包括在結果中。

`zip()`也可以以中綴形式調用，即`a zip b`。

```
val colors = listOf("red", "brown", "grey")
val animals = listOf("fox", "bear", "wolf")
println(colors zip animals)

val twoAnimals = listOf("fox", "bear")
println(colors.zip(twoAnimals))
```

你也可以用一個帶有兩個參數的轉換函數調用`zip()`：接收元素和參數元素。在這種情況下，結果列表包含對接收元素和參數元素的對調用轉換函數的返回值。

```
val colors = listOf("red", "brown", "grey")
val animals = listOf("fox", "bear", "wolf")
```

```
println(colors.zip(animals) { color, animal -> "The ${animal.replaceFirstChar { it.uppercase() }} is $color"})
```

當你有一個`Pair`列表時，可以進行反向轉換——解壓縮，從這些對中構建兩個列表：

第一個列表包含原始列表中每個`Pair`的第一個元素。

第二個列表包含第二個元素。

要解壓縮一個`Pair`列表，調用`unzip()`。

```
val numberPairs = listOf("one" to 1, "two" to 2, "three" to 3, "four" to 4)
println(numberPairs.unzip())
```

關聯 (Associate)

關聯轉換允許從集合元素及其相關的某些值構建map。在不同的關聯類型中，元素可以是關聯map中的鍵或值。

基本的關聯函數`associateWith()`創建一個map，其中原始集合的元素是鍵，值是由給定的轉換函數生成的。如果兩個元素相等，只有最後一個會保留在map中。

```
val numbers = listOf("one", "two", "three", "four")
println(numbers.associateWith { it.length })
```

要構建map，其集合元素作為值，有`associateBy()`函數。它接收一個函數，該函數根據元素的值返回鍵。如果兩個元素的鍵相等，只有最後一個會保留在map中。

`associateBy()`也可以與值轉換函數一起調用。

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

println(numbers.associateBy { it.first().uppercaseChar() })
println(numbers.associateBy(keySelector = { it.first().uppercaseChar() }, valueTransform = { it.length }))
```

另一種方法是使用`associate()`構建map，其中鍵和值都從集合元素中生成。它接收一個lambda函數，該函數返回一個`Pair`：對應的map條目的鍵和值。

需要注意的是，`associate()`會產生短暫存在的`Pair`對象，這可能會影響性能。因此，當性能不是關鍵或比其他選項更可取時，應使用`associate()`。

一個示例是當鍵和值是一起從元素生成時。

```
val names = listOf("Alice Adams", "Brian Brown", "Clara Campbell")
println(names.associate { name -> parseFullName(name).let { it.lastName to it.firstName } })
```

在這裡，我們首先對元素調用轉換函數，然後從該函數的結果屬性構建一個對。

展平 (Flatten)

如果你操作嵌套集合，你可能會發現標準庫提供的平展訪問嵌套集合元素的函數很有用。

第一個函數是`flatten()`。你可以在集合的集合上調用它，例如，一個`List`的`Set`。該函數返回一個包含所有嵌套集合元素的單個列表。

```
val numberSets = listOf(setOf(1, 2, 3), setOf(4, 5, 6), setOf(1, 2))
println(numberSets.flatten())
```

另一個函數`flatMap()`提供了一種靈活的方式來處理嵌套集合。它接收一個函數，該函數將集合元素映射到另一個集合。結果，`flatMap()`返回其對所有元素的返回值的單個列表。因此，`flatMap()`的行為類似於隨後調用`map()`（以集合作為映射結果）和`flatten()`。

```
val containers = listOf(
    StringContainer(listOf("one", "two", "three")),
    StringContainer(listOf("four", "five", "six")),
    StringContainer(listOf("seven", "eight"))
)
println(containers.flatMap { it.values })
```

字符串表示 (String representation)

如果你需要以可讀格式檢索集合內容，請使用將集合轉換為字符串的函數：`joinToString()`和`joinTo()`。

`joinToString()`根據提供的參數構建單個字符串。`joinTo()`做同樣的事，但將結果附加到給定的`Appendable`對象。

當使用默認參數調用時，這些函數返回的結果類似於對集合調用`toString()`：元素的字符串表示形式以逗號和空格分隔的字符串。

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

println(numbers)
println(numbers.joinToString())

val listString = StringBuffer("The list of numbers: ")
numbers.joinTo(listString)
println(listString)
```

要構建自定義字符串表示，可以在函數參數中指定其參數：分隔符、前綴和後綴。結果字符串將以前綴開始，以後綴結束。分隔符將在每個元素之後出現，除了最後一個元素。

```
val numbers = listOf("one", "two", "three", "four")
println(numbers.joinToString(separator = " | ", prefix = "start: ", postfix = ": end"))
```

對於較大的集合，你可能希望指定限制——將包含在結果中的元素數量。如果集合大小超過限制，所有其他元素將被替換為單個截斷參數值。

```
val numbers = listOf("one", "two", "three", "four")
println(numbers.joinToString { "Element: ${it.uppercase()}"})
```

過濾集合

過濾是集合處理中最常見的任務之一。在Kotlin中，過濾條件由謂詞定義——這些lambda函數接收一個集合元素並返回一個布爾值：`true`表示給定元素符合謂詞，`false`表示相反。

標準庫包含一組擴展函數，允許你在一次調用中過濾集合。這些函數不會更改原始集合，因此它們可用於可變和只讀集合。要操作過濾結果，應將其賦值給變量或在過濾後鏈接函數。

根據謂詞過濾

基本的過濾函數是`filter()`。當用謂詞調用時，`filter()`返回符合條件的集合元素。對於List和Set，結果集合是List，對於Map，結果也是Map。

```
val numbers = listOf("one", "two", "three", "four")
val longerThan3 = numbers.filter { it.length > 3 }
println(longerThan3)

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)
```

`filter()`中的謂詞只能檢查元素的值。如果你想在過濾中使用元素的位置，請使用`filterIndexed()`。它接收一個帶有兩個參數的謂詞：索引和元素的值。

要根據否定條件過濾集合，使用`filterNot()`。它返回一個對謂詞結果為`false`的元素列表。

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

val filteredIdx = numbers.filterIndexed { index, s -> (index != 0) && (s.length < 5) }
val filteredNot = numbers.filterNot { it.length <= 3 }

println(filteredIdx)
println(filteredNot)
```

還有一些函數可以通過過濾給定類型的元素來縮小元素類型：

- `filterIsInstance()`返回給定類型的集合元素。對`List<Any>`調用`filterIsInstance<T>()`返回`List<T>`，從而允許你對其項目調用T類型的函數。

```
val numbers = listOf(null, 1, "two", 3.0, "four")
```

```
println("All String elements in upper case:")
numbers.filterIsInstance<String>().forEach {
    println(it.uppercase())
}
```

- `filterNotNull()` 返回所有非空元素。對 `List<T?>` 調用 `filterNotNull()` 返回 `List<T: Any>`，從而允許你將元素作為非空對象處理。

```
val numbers = listOf(null, "one", "two", null)
numbers.filterNotNull().forEach {
    println(it.length)   // length is unavailable for nullable Strings
}
```

分區 (Partition)

另一個過濾函數——`partition()`——根據謂詞過濾集合，並將不符合條件的元素保存在單獨的列表中。因此，你會得到一對列表作為返回值：第一個列表包含符合謂詞的元素，第二個列表包含原始集合中的其他所有元素。

```
val numbers = listOf("one", "two", "three", "four")
val (match, rest) = numbers.partition { it.length > 3 }

println(match)
println(rest)
```

測試謂詞 (Test predicates)

最後，有一些函數只是將謂詞測試集合元素：

- `any()`：如果至少有一個元素符合給定謂詞，則返回 `true`。
- `none()`：如果沒有元素符合給定謂詞，則返回 `true`。
- `all()`：如果所有元素都符合給定謂詞，則返回 `true`。注意，對空集合調用任何有效謂詞時，`all()` 返回 `true`。這種行為在邏輯中稱為真空真理。

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

println(numbers.any { it.endsWith("e") })
println(numbers.none { it.endsWith("a") })
println(numbers.all { it.endsWith("e") })

println(emptyList<Int>().all { it > 5 })   // vacuous truth
```

`any()` 和 `none()` 也可以不帶謂詞使用：在這種情況下，它們只是檢查集合是否為空。`any()` 返回 `true` 如果有元素，否則返回 `false`；`none()` 做相反的事。

```
val numbers = listOf("one", "two", "three", "four")
val empty = emptyList<String>()

println(numbers.any())
println(empty.any())

println(numbers.none())
println(empty.none())
```

加號和減號操作符

在Kotlin中，加號（+）和減號（-）操作符是為集合定義的。它們將集合作為第一個操作數；第二個操作數可以是元素或另一個集合。返回值是一個新的只讀集合：

- `plus` 的結果包含原始集合中的元素以及第二個操作數中的元素。
- `minus` 的結果包含原始集合中的元素，但去除了第二個操作數中的元素。如果第二個操作數是元素，`minus` 會移除它的首次出現；如果是集合，則移除其所有元素的出現。

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

val plusList = numbers + "five"
val minusList = numbers - listOf("three", "four")
println(plusList)  // [one, two, three, four, five]
println(minusList) // [one, two]
```

對映射的加號和減號操作符

關於映射（Map）的加號和減號操作符的詳細信息，請參見映射特定操作。

增強賦值操作符

增強賦值操作符 `plusAssign`（+=）和 `minusAssign`（-=）也為集合定義。然而，對於只讀集合，它們實際上使用 `plus` 或 `minus` 操作符並嘗試將結果賦值給相同的變量。因此，它們僅適用於 `var` 只讀集合。對於可變集合，如果是 `val`，它們會修改集合。更多詳細信息請參見集合寫操作。

分組

Kotlin標準庫提供了一組用於對集合元素進行分組的擴展函數。基本的分組函數 `groupBy()` 接受一個lambda函數並返回一個Map。在這個Map中，每個鍵是lambda結果，對應的值是返回該結果的元素列表。此函數可以用來，例如，按字符串的首字母對字符串列表進行分組。

你還可以使用第二個lambda參數來調用 `groupBy()`，這是一個值轉換函數。在使用兩個lambda的 `groupBy()` 的結果Map中，`keySelector` 函數生成的鍵被映射到值轉換函數的結果，而不是原始元素。

以下示例展示了如何使用 `groupBy()` 函數按字符串的首字母對字符串進行分組，使用 `for` 運算符迭代結果Map中的組，然後使用 `keySelector` 函數將值轉換為大寫：

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

// 使用 groupBy() 按首字母對字符串進行分組
val groupedByFirstLetter = numbers.groupBy { it.first().uppercase() }
println(groupedByFirstLetter)
// {O=[one], T=[two, three], F=[four, five]}

// 迭代每個組並打印鍵及其相關的值
for ((key, value) in groupedByFirstLetter) {
    println("Key: $key, Values: $value")
}
// Key: O, Values: [one]
// Key: T, Values: [two, three]
// Key: F, Values: [four, five]

// 按首字母對字符串進行分組並將值轉換為大寫
val groupedAndTransformed = numbers.groupBy(keySelector = { it.first() }, valueTransform = { it.uppercase() })
println(groupedAndTransformed)
// {o=[ONE], t=[TWO, THREE], f=[FOUR, FIVE]}
```

如果你想對元素進行分組，然後一次性對所有組應用操作，請使用 `groupingBy()` 函數。它返回一個 `Grouping` 類型的實例。`Grouping` 實例允許你以懶惰的方式對所有組應用操作：這些組實際上是在操作執行之前構建的。

具體而言，`Grouping` 支持以下操作：

- `eachCount()` 計算每個組中的元素數量。
- `fold()` 和 `reduce()` 對每個組作為單獨的集合執行fold和reduce操作，並返回結果。
- `aggregate()` 將給定操作依次應用於每個組中的所有元素，並返回結果。這是對 `Grouping` 執行任何操作的通用方法。當 `fold` 或 `reduce` 不夠時，使用它來實現自定義操作。

你可以在結果Map上使用 `for` 運算符來迭代由 `groupingBy()` 函數創建的組。這允許你訪問每個鍵和與該鍵相關的元素計數。

以下示例展示了如何使用 `groupingBy()` 函數按字符串的首字母對字符串進行分組，計算每個組中的元素數量，然後迭代每個組以打印鍵和元素的數量：

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

// 使用 groupingBy() 按首字母對字符串進行分組並計算每個組中的元素數量
val grouped = numbers.groupingBy { it.first() }.eachCount()

// 迭代每個組並打印鍵及其相關的值
for ((key, count) in grouped) {
    println("Key: $key, Count: $count")
    // Key: o, Count: 1
    // Key: t, Count: 2
    // Key: f, Count: 2
```

```
}
```

這樣，你可以根據特定的條件對集合元素進行分組，並對分組結果進行進一步的操作。

檢索集合部分

Kotlin標準庫包含一組擴展函數，用於檢索集合的部分內容。這些函數提供多種方式來選擇結果集合的元素：明確列出它們的位置、指定結果大小等。

切片 (Slice)

`slice()` 返回具有給定索引的集合元素列表。索引可以作為範圍或整數值的集合傳遞。

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.slice(1..3))              // [two, three, four]
println(numbers.slice(0..4 step 2))       // [one, three, five]
println(numbers.slice(setOf(3, 5, 0)))    // [four, six, one]
```

取出和丟棄 (Take and drop)

要從第一個元素開始取出指定數量的元素，使用`take()`函數。要取出最後的元素，使用`takeLast()`。當傳遞的數量大於集合大小時，這兩個函數都返回整個集合。

要取出除了給定數量的第一個或最後一個元素之外的所有元素，分別調用`drop()`和`dropLast()`函數。

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.take(3))                // [one, two, three]
println(numbers.takeLast(3))            // [four, five, six]
println(numbers.drop(1))                // [two, three, four, five, six]
println(numbers.dropLast(5))            // [one]
```

你還可以使用謂詞來定義取出或丟棄的元素數量。有四個與上面描述的函數類似的函數：

- `takeWhile()` 是帶有謂詞(predicates)的`take()`：它取出直到不匹配謂詞的第一個元素為止（不包括該元素）的元素。如果第一個集合元素不匹配謂詞，結果為空。
- `takeLastWhile()` 類似於`takeLast()`：它從集合末尾取出匹配謂詞的元素範圍。範圍的第一個元素是不匹配謂詞的最後一個元素的下一個元素。如果最後的集合元素不匹配謂詞，結果為空。
- `dropWhile()` 與帶有相同謂詞的`takeWhile()`相反：它返回從不匹配謂詞的第一個元素到結尾的元素。
- `dropLastWhile()` 與帶有相同謂詞的`takeLastWhile()`相反：它返回從開始到不匹配謂詞的最後一個元素的元素。

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.takeWhile { !it.startsWith('f') })     // [one, two, three, four]
println(numbers.takeLastWhile { it != "three" })       // [four, five, six]
println(numbers.dropWhile { it.length == 3 })          // [four, five, six]
println(numbers.dropLastWhile { it.contains('i') })    // [one, two, three]
```

分塊 (Chunked)

要將集合拆分為給定大小的部分，使用`chunked()`函數。`chunked()`接受一個參數——塊的大小，並返回一個包含給定大小的列表的列表。第一個塊從第一個元素開始並包含大小元素，第二個塊包含接下來的大小元素，依此類推。最後一個塊可能大小較小。

```
val numbers = (0..13).toList()
println(numbers.chunked(3))               // [[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10, 11], [12, 13]]
```

你還可以立即對返回的塊應用轉換。為此，調用`chunked()`時提供轉換函數。lambda參數是集合的一個塊。當`chunked()`與轉換一起調用時，這些塊是短暫的列表，應該在該lambda中消費。

```
val numbers = (0..13).toList()
println(numbers.chunked(3) { it.sum() })   // [3, 12, 21, 30, 25]
```

窗口 (Windowed)

你可以檢索集合元素的所有可能的給定大小的範圍。用於獲取它們的函數稱為`windowed()`：它返回元素範圍的列表，就像你通過滑動窗口查看集合一樣。與`chunked()`不同，`windowed()`返回從每個集合元素開始的元素範圍。所有窗口都作為單個列表的元素返回。

```
val numbers = listOf("one", "two", "three", "four", "five")
println(numbers.windowed(3))              // [[one, two, three], [two, three, four], [three, four, five]]
```

`windowed()` 提供了可選參數，使其更具靈活性：

- `step` 定義了兩個相鄰窗口的第一個元素之間的距離。默認值為1，因此結果包含從所有元素開始的窗口。如果將步長增加到2，你將只會獲得從奇數元素開始的窗口：第一、第三，依此類推。
- `partialWindows` 包含從集合末尾的元素開始的較小大小的窗口。例如，如果你請求三個元素的窗口，你無法為最後兩個元素構建它們。在這種情況下，啟用 `partialWindows` 將包含大小為2和1的兩個列表。

最後，你可以立即對返回的範圍應用轉換。為此，調用 `windowed()` 時提供轉換函數。

```
val numbers = (1..10).toList()
println(numbers.windowed(3, step = 2, partialWindows = true))  // [[1, 2, 3], [3, 4, 5], [5, 6, 7], [7, 8, 9], [9, 10]]
println(numbers.windowed(3) { it.sum() })                      // [6, 9, 12, 15, 18, 21, 24, 27]
```

要構建兩元素的窗口，有一個單獨的函數——`zipWithNext()`。它創建接收集合的相鄰元素對。請注意，`zipWithNext()` 不會將集合分成對；它為每個元素（除了最後一個）創建一個 `Pair`，因此其結果對於[1, 2, 3, 4]是[[1, 2], [2, 3], [3, 4]]，而不是[[1, 2], [3, 4]]。`zipWithNext()` 也可以與轉換函數一起調用；該轉換函數應接受接收集合的兩個元素作為參數。

```
val numbers = listOf("one", "two", "three", "four", "five")
println(numbers.zipWithNext())                            // [(one, two), (two, three), (three, four), (four, five)]
println(numbers.zipWithNext { s1, s2 -> s1.length > s2.length }) // [false, false, false, true]
```

這些擴展函數使你能夠靈活地檢索和處理集合的部分內容。

檢索單個元素

Kotlin集合提供了一組函數，用於從集合中檢索單個元素。本頁描述的函數適用於列表和集合。

按位置檢索

要檢索特定位置的元素，可以使用 `elementAt()` 函數。調用該函數並傳入一個整數作為參數，你將獲得給定位置的集合元素。第一個元素的位置是0，最後一個元素的位置是 `size - 1`。

`elementAt()` 對於不提供索引訪問的集合或靜態上未知提供索引訪問的集合很有用。在列表的情況下，更符合習慣的是使用索引訪問操作符（`get()` 或 `[]`）。

```
val numbers = linkedSetOf("one", "two", "three", "four", "five")
println(numbers.elementAt(3))  // four

val numbersSortedSet = sortedSetOf("one", "two", "three", "four")
println(numbersSortedSet.elementAt(0))  // one (元素按升序存儲)
```

還有一些用於檢索集合的第一個和最後一個元素的有用別名：`first()` 和 `last()`。

```
val numbers = listOf("one", "two", "three", "four", "five")
println(numbers.first())  // one
println(numbers.last())   // five
```

為了避免在檢索不存在位置的元素時拋出異常，可以使用 `elementAt()` 的安全變體：

- `elementAtOrNull()` 當指定位置超出集合範圍時返回 `null`。
- `elementAtOrElse()` 另外接受一個lambda函數，該函數將一個整數參數映射為集合元素類型的實例。當調用超出範圍的位置時，`elementAtOrElse()` 返回該lambda對給定值的結果。

```
val numbers = listOf("one", "two", "three", "four", "five")
println(numbers.elementAtOrNull(5))  // null
println(numbers.elementAtOrElse(5) { index -> "The value for index $index is undefined" })
// The value for index 5 is undefined
```

按條件檢索

函數 `first()` 和 `last()` 也讓你可以搜索集合中符合給定謂詞的元素。當你調用帶有測試集合元素的謂詞的 `first()` 時，你將獲得謂詞為 `true` 的第一個元素。反過來，帶有謂詞的 `last()` 返回匹配謂詞的最後一個元素。

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.first { it.length > 3 })  // three
println(numbers.last { it.startsWith("f") })  // five
```

如果沒有元素匹配謂詞，這兩個函數會拋出異常。為了避免這種情況，使用 `firstOrNull()` 和 `lastOrNull()`：如果沒有找到匹配的元素，它們返回 `null`。

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.firstOrNull { it.length > 6 })  // null
```

如果它們的名字更適合你的情況，可以使用別名：

- `find()` 代替 `firstOrNull()`
- `findLast()` 代替 `lastOrNull()`

```
val numbers = listOf(1, 2, 3, 4)
println(numbers.find { it % 2 == 0 })  // 2
println(numbers.findLast { it % 2 == 0 })  // 4
```

帶選擇器的檢索

如果你需要在檢索元素之前對集合進行映射，可以使用 `firstNotNullOf()` 函數。它結合了兩個操作：

1. 使用選擇器函數映射集合
2. 返回結果中的第一個非空值

如果結果集合中沒有非空元素，`firstNotNullOf()` 會拋出 `NoSuchElementException`。使用對應的 `firstNotNullOfOrNull()` 來在這種情況下返回 `null`。

```
val list = listOf<Any>(0, "true", false)
// 將每個元素轉換為字符串並返回具有所需長度的第一個
val longEnough = list.firstNotNullOf { item -> item.toString().takeIf { it.length >= 4 } }
println(longEnough)  // true
```

隨機元素

如果你需要檢索集合的任意元素，調用 `random()` 函數。可以不帶參數調用它，或者將 `Random` 對象作為隨機性的來源。

```
val numbers = listOf(1, 2, 3, 4)
println(numbers.random())
```

在空集合上，`random()` 會拋出異常。要接收 `null`，請使用 `randomOrNull()`。

檢查元素存在

要檢查集合中是否存在某個元素，使用 `contains()` 函數。如果有一個等於函數參數的集合元素，它返回 `true`。可以使用 `in` 關鍵字以操作符形式調用 `contains()`。

要一次檢查多個實例的存在，調用 `containsAll()` 並將這些實例的集合作為參數。

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.contains("four"))  // true
println("zero" in numbers)         // false

println(numbers.containsAll(listOf("four", "two")))  // true
println(numbers.containsAll(listOf("one", "zero")))  // false
```

此外，可以通過調用 `isEmpty()` 或 `isNotEmpty()` 檢查集合中是否包含任何元素。

```
val numbers = listOf("one", "two", "three", "four", "five", "six")
println(numbers.isEmpty())       // false
println(numbers.isNotEmpty())    // true

val empty = emptyList<String>()
println(empty.isEmpty())         // true
println(empty.isNotEmpty())      // false
```

這些擴展函數使你能夠靈活地檢索和處理集合中的單個元素。

排序

元素的順序是某些集合類型的重要方面。例如，兩個具有相同元素的列表，如果它們的元素順序不同，它們就不相等。

在Kotlin中，可以通過幾種方式定義對象的順序。

首先是自然順序。它是為實現 Comparable 介面的類型定義的。當沒有指定其他順序時，會使用自然順序來對其進行排序。

大多數內建類型是可比較的：

- 數字類型使用傳統的數值順序：1大於0；-3.4f大於-5f，依此類推。
- 字符和字符串使用詞典順序：b大於a；world大於hello。

要為自定義類型定義自然順序，請使該類型成為 Comparable 的實現者。這需要實現 compareTo() 函數。 compareTo() 必須接受同類型的另一個對象作為參數，並返回一個整數值，表示哪個對象更大：

- 正值表示接收對象更大。
- 負值表示它小於參數。
- 零表示兩個對象相等。

以下是一個排序版本的類，該版本由主要部分和次要部分組成。

```
class Version(val major: Int, val minor: Int): Comparable<Version> {
    override fun compareTo(other: Version): Int = when {
        this.major != other.major -> this.major compareTo other.major // compareTo() 以中綴形式
        this.minor != other.minor -> this.minor compareTo other.minor
        else -> 0
    }
}

fun main() {
    println(Version(1, 2) > Version(1, 3))  // false
    println(Version(2, 0) > Version(1, 5))  // true
}
```

自定義順序允許你以所需的方式對任何類型的實例進行排序。特別是，你可以為不可比較的對象定義順序，或為可比較類型定義自然順序以外的順序。要為某種類型定義自定義順序，請為其創建 Comparator 。 Comparator 包含 compare() 函數：它接受一個類的兩個實例，並返回它們之間比較的整數結果。結果的解釋方式與上述 compareTo() 結果相同。

```
val lengthComparator = Comparator { str1: String, str2: String -> str1.length - str2.length }
println(listOf("aaa", "bb", "c").sortedWith(lengthComparator))  // [c, bb, aaa]
```

擁有 lengthComparator ，你可以按字符串長度排列，而不是按默認的詞典順序。

定義 Comparator 的一種更簡便的方法是使用標準庫中的 compareBy() 函數。 compareBy() 接受一個lambda函數，該函數從實例生成一個 Comparable 值，並將自定義順序定義為生成值的自然順序。

使用 compareBy() ，上面示例中的長度比較器如下所示：

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

Kotlin集合包提供了用於按自然順序、自定義順序甚至隨機順序排序集合的函數。在本頁中，我們將描述適用於只讀集合的排序函數。這些函數返回其結果作為包含原始集合元素的新集合，並按請求的順序排列。要了解有關原地排序可變集合的函數，請參見列表特定操作。

自然順序

基本函數 sorted() 和 sortedDescending() 根據其自然順序返回升序和降序排序的集合元素。這些函數適用於 Comparable 元素的集合。

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

println("Sorted ascending: ${numbers.sorted()}")        // Sorted ascending: [four, one, three, two]
println("Sorted descending: ${numbers.sortedDescending()}")  // Sorted descending: [two, three, one, four]
```

自定義順序

對於自定義順序排序或不可比較對象排序，有 sortedBy() 和 sortedByDescending() 函數。它們接受一個選擇器函數，該函數將集合元素映射為 Comparable 值，並按這些值的自然順序排序集合。

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

val sortedNumbers = numbers.sortedBy { it.length }
println("Sorted by length ascending: $sortedNumbers")  // Sorted by length ascending: [one, two, four, three]
val sortedByLast = numbers.sortedByDescending { it.last() }
```

```
println("Sorted by the last letter descending: $sortedByLast")  // Sorted by the last letter descending: [four, two, one, three]
```

要為集合排序定義自定義順序，可以提供自己的`Comparator`。為此，調用`sortedWith()`函數並傳入你的`Comparator`。使用此函數，按字符串長度排序看起來如下所示：

```
val numbers = listOf("one", "two", "three", "four")
println("Sorted by length ascending: ${numbers.sortedWith(compareBy { it.length })}")  // Sorted by length ascending: [one, two, four, three]
```

反向順序

可以使用`reversed()`函數檢索反向順序的集合。

```
val numbers = listOf("one", "two", "three", "four")
println(numbers.reversed())  // [four, three, two, one]
```

`reversed()`返回包含元素副本的新集合。因此，如果你稍後更改原始集合，這不會影響以前獲得的`reversed()`結果。

另一個反轉函數 - `asReversed()`

返回相同集合實例的反轉視圖，因此如果原始列表不會更改，它可能比`reversed()`更輕量且更可取。

```
val numbers = listOf("one", "two", "three", "four")
    val reversedNumbers = numbers.asReversed()
    println(numbers)     // [one, two, three, four]
    println(numbers.reversed()) // [four, three, two, one]
    println(numbers)   // [one, two, three, four]
    println(reversedNumbers)   // [four, three, two, one]
```

如果原始列表是可變的，它的所有更改都會反映在其反轉視圖中，反之亦然。

```
val numbers = mutableListOf("one", "two", "three", "four")
val reversedNumbers = numbers.asReversed()
println(reversedNumbers)  // [four, three, two, one]
numbers.add("five")
println(reversedNumbers)  // [five, four, three, two, one]
```

然而，如果列表的可變性未知或源根本不是列表，`reversed()`更可取，因為它的結果是未來不會更改的副本。

隨機順序

最後，有一個返回包含隨機順序的集合元素的新列表的函數——`shuffled()`。你可以不帶參數調用它，或將`Random`對象作為參數。

```
val numbers = listOf("one", "two", "three", "four")
println(numbers.shuffled())
```

這些函數使你能夠靈活地排序集合，並根據需要檢索其元素。

聚合操作

Kotlin集合包含一些常用的聚合操作函數——這些操作根據集合的內容返回單個值。大多數這些操作都是眾所周知的，並且在其他語言中也以相同方式工作：

- `minOrNull()` 和 `maxOrNull()` 分別返回最小和最大的元素。對於空集合，它們返回`null`。
- `average()` 返回數字集合中元素的平均值。
- `sum()` 返回數字集合中元素的總和。
- `count()` 返回集合中的元素數量。

```
fun main() {
    val numbers = listOf(6, 42, 10, 4)

    println("Count: ${numbers.count()}")         // Count: 4
    println("Max: ${numbers.maxOrNull()}")       // Max: 42
    println("Min: ${numbers.minOrNull()}")       // Min: 4
    println("Average: ${numbers.average()}")     // Average: 15.5
    println("Sum: ${numbers.sum()}")             // Sum: 62
}
```

還有一些函數可以通過特定的選擇器函數或自定義的 Comparator 來檢索最小和最大的元素：

- maxByOrNull() 和 minByOrNull() 接受一個選擇器函數，並返回該函數返回值最大的或最小的元素。
- maxWithOrNull() 和 minWithOrNull() 接受一個 Comparator 對象，並根據該 Comparator 返回最大或最小的元素。
- maxOfOrNull() 和 minOfOrNull() 接受一個選擇器函數，並返回該選擇器函數的最大或最小返回值。
- maxOfWithOrNull() 和 minOfWithOrNull() 接受一個 Comparator 對象，並根據該 Comparator 返回選擇器函數的最大或最小返回值。

這些函數在空集合上返回 null 。還有一些替代函數——maxOf 、minOf 、maxOfWith 和 minOfWith ——它們與對應的函數作用相同，但在空集合上會拋出 NoSuchElementException 。

```
val numbers = listOf(5, 42, 10, 4)
val min3Remainder = numbers.minByOrNull { it % 3 }
println(min3Remainder)  // 4

val strings = listOf("one", "two", "three", "four")
val longestString = strings.maxWithOrNull(compareBy { it.length })
println(longestString)  // three
```

除了常規的 sum() ，還有一個高級求和函數 sumOf() ，它接受一個選擇器函數並返回該函數應用於所有集合元素的總和。選擇器可以返回不同的數字類型： Int 、 Long 、 Double 、 UInt 和 ULong （在JVM上還包括 BigInteger 和 BigDecimal ）。

```
val numbers = listOf(5, 42, 10, 4)
println(numbers.sumOf { it * 2 })         // 122
println(numbers.sumOf { it.toDouble() / 2 }) // 30.5
```

Fold和Reduce

對於更具體的情況，有 reduce() 和 fold() 函數，它們將提供的操作依次應用於集合元素並返回累積結果。該操作接受兩個參數：先前的累積值和集合元素。

這兩個函數的區別在於 fold() 接受一個初始值，並在第一步中將其作為累積值，而 reduce() 的第一步使用第一個和第二個元素作為操作參數。

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

val simpleSum = numbers.reduce { sum, element -> sum + element } // 5 + 2 +10 + 4
println(simpleSum)  // 21
val sumDoubled = numbers.fold(0) { sum, element -> sum + element * 2 } // 0 + 5*2 + 2*2 + 10*2 + 4*2
println(sumDoubled)  // 42

// 錯誤：第一個元素未在結果中加倍
// val sumDoubledReduce = numbers.reduce { sum, element -> sum + element * 2 }
// println(sumDoubledReduce)
```

上述示例顯示了區別： fold() 用於計算加倍元素的總和。如果將相同的函數傳遞給 reduce() ，它將返回另一個結果，因為它在第一步中使用列表的第一個和第二個元素作為參數，因此第一個元素不會被加倍。

要對元素按相反順序應用函數，請使用 reduceRight() 和 foldRight() 函數。它們的工作方式類似於 fold() 和 reduce() ，但從最後一個元素開始，然後繼續到前一個。請注意，當從右到左進行折疊或歸約時，操作參數會改變順序：首先是元素，然後是累積值。

```
val numbers = listOf(5, 2, 10, 4)
val sumDoubledRight = numbers.foldRight(0) { element, sum -> sum + element * 2 } // 0 + 4*2 + 10*2 + 2*2 + 5*2
println(sumDoubledRight)  // 42
```

你還可以應用將元素索引作為參數的操作。為此，使用 reduceIndexed() 和 foldIndexed() 函數，將元素索引作為操作的第一個參數。

最後，有一些函數將此類操作應用於集合元素，從右到左——reduceRightIndexed() 和 foldRightIndexed() 。

```
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)  // 15

val sumEvenRight = numbers.foldRightIndexed(0) { idx, element, sum -> if (idx % 2 == 0) sum + element else sum }
println(sumEvenRight)  // 14
```

所有的歸約操作在空集合上會拋出異常。要接收 null ，請使用它們的 *OrNull() 對應函數：

- reduceOrNull()

- reduceRightOrNull()
- reduceIndexedOrNull()
- reduceRightIndexedOrNull()

如果你希望保存中間累加值，請使用runningFold()（或其同義詞scan()）和runningReduce()函數。

```
val numbers = listOf(0, 1, 2, 3, 4, 5)
val runningReduceSum = numbers.runningReduce { sum, item -> sum + item }
val runningFoldSum = numbers.runningFold(10) { sum, item -> sum + item }

println(runningReduceSum)  // [0, 1, 3, 6, 10, 15]
println(runningFoldSum)    // [10, 10, 11, 13, 16, 20, 25]
```

如果你需要在操作參數中使用索引，請使用runningFoldIndexed()或runningReduceIndexed()。

集合寫操作

可變集合支持更改集合內容的操作，例如添加或移除元素。本頁將描述所有可變集合實現可用的寫操作。關於列表和映射的特定操作，請分別參見列表特定操作和映射特定操作。

添加元素

要向列表或集合添加單個元素，使用add()函數。指定的對象會附加到集合的末尾。

```
val numbers = mutableListOf(1, 2, 3, 4)
numbers.add(5)
println(numbers)  // [1, 2, 3, 4, 5]
```

addAll()將參數對象的每個元素添加到列表或集合。參數可以是Iterable、Sequence或Array。接收者和參數的類型可以不同，例如，可以將集合中的所有項目添加到列表中。

當在列表上調用時，addAll()按參數中的順序添加新元素。你也可以在調用addAll()時指定元素位置作為第一個參數。參數集合的第一個元素將插入到此位置。參數集合的其他元素將跟隨它，將接收者元素移到末尾。

```
val numbers = mutableListOf(1, 2, 5, 6)
numbers.addAll(arrayOf(7, 8))
println(numbers)  // [1, 2, 5, 6, 7, 8]
numbers.addAll(2, setOf(3, 4))
println(numbers)  // [1, 2, 3, 4, 5, 6, 7, 8]
```

你還可以使用原地版本的加號操作符plusAssign (+=)添加元素。當應用於可變集合時，+=將第二個操作數（元素或另一個集合）附加到集合的末尾。

```
val numbers = mutableListOf("one", "two")
numbers += "three"
println(numbers)  // [one, two, three]
numbers += listOf("four", "five")
println(numbers)  // [one, two, three, four, five]
```

移除元素

要從可變集合中移除元素，使用remove()函數。remove()接受元素值並移除此值的第一個出現。

```
val numbers = mutableListOf(1, 2, 3, 4, 3)
numbers.remove(3)  // 移除第一個`3`
println(numbers)  // [1, 2, 4, 3]
numbers.remove(5)  // 不移除任何東西
println(numbers)  // [1, 2, 4, 3]
```

要一次移除多個元素，有以下函數：

- removeAll() 移除參數集合中的所有元素。或者，你可以將它與謂詞作為參數一起調用；在這種情況下，該函數會移除謂詞為 true 的所有元素。
- retainAll() 是removeAll()的反向操作：它移除除參數集合之外的所有元素。當與謂詞一起使用時，它僅保留匹配謂詞的元素。
- clear() 移除列表中的所有元素，並使其變空。

```
val numbers = mutableListOf(1, 2, 3, 4)
println(numbers)  // [1, 2, 3, 4]
numbers.retainAll { it >= 3 }
println(numbers)  // [3, 4]
numbers.clear()
println(numbers)  // []

val numbersSet = mutableSetOf("one", "two", "three", "four")
numbersSet.removeAll(setOf("one", "two"))
println(numbersSet)  // [three, four]
```

另一種從集合中移除元素的方法是使用減號操作符 `minusAssign (-=)` ——原地版本的 `minus`。第二個參數可以是元素類型的單個實例或另一個集合。當右側是單個元素時，`-=` 移除它的第一次出現。反之，如果它是一個集合，則會移除其元素的所有出現。例如，如果列表包含重複的元素，它們會一次性移除。第二個操作數可以包含集合中不存在的元素。這些元素不會影響操作的執行。

```
val numbers = mutableListOf("one", "two", "three", "three", "four")
numbers -= "three"
println(numbers)  // [one, two, three, four]
numbers -= listOf("four", "five")
//numbers -= listOf("four")    // 與上面相同
println(numbers)  // [one, two, three]
```

更新元素

列表和映射還提供了用於更新元素的操作。它們在列表特定操作和映射特定操作中描述。對於集合，更新沒有意義，因為它實際上是移除一個元素並添加另一個元素。

這些操作使你能夠靈活地管理可變集合的內容，添加、移除和更新元素以滿足應用程序的需求。

List特定操作

列表是Kotlin中最受歡迎的內建集合類型。對列表元素的索引訪問提供了一套強大的操作。

按索引檢索元素

列表支持所有常見的元素檢索操作：`elementAt()`、`first()`、`last()` 等。在列表中特定的是對元素的索引訪問，因此最簡單的讀取元素方式是通過索引檢索。這可以通過帶有索引作為參數的 `get()` 函數或簡寫的 `[index]` 語法來完成。

如果列表大小小於指定的索引，會拋出異常。有兩個其他函數可以幫助你避免這些異常：

- `getOrElse()` 允許你提供計算默認值的函數，以便在索引不存在時返回該值。
- `getOrNull()` 返回 `null` 作為默認值。

```
val numbers = listOf(1, 2, 3, 4)
println(numbers.get(0))             // 1
println(numbers[0])                 // 1
//numbers.get(5)                    // exception!
println(numbers.getOrNull(5))       // null
println(numbers.getOrElse(5) { it })  // 5
```

檢索列表部分

除了檢索集合部分的常見操作外，列表還提供了 `subList()` 函數，該函數返回指定元素範圍的視圖作為列表。因此，如果原始集合的元素發生變化，它也會在先前創建的子列表中變化，反之亦然。

```
val numbers = (0..13).toList()
println(numbers.subList(3, 6))  // [3, 4, 5]
```

查找元素位置

線性搜索

在任何列表中，你可以使用 `indexOf()` 和 `lastIndexOf()` 函數查找元素的位置。它們返回列表中等於給定參數的元素的第一個和最後一個位置。如果沒有這樣的元素，這兩個函數返回-1。

```
val numbers = listOf(1, 2, 3, 4, 2, 5)
println(numbers.indexOf(2))       // 1
```

```
println(numbers.lastIndexOf(2))    // 4
```

還有一對函數，它們接受謂詞並搜索匹配的元素：

- `indexOfFirst()` 返回匹配謂詞的第一個元素的索引，如果沒有這樣的元素則返回-1。
- `indexOfLast()` 返回匹配謂詞的最後一個元素的索引，如果沒有這樣的元素則返回-1。

```
val numbers = mutableListOf(1, 2, 3, 4)
println(numbers.indexOfFirst { it > 2})   // 2
println(numbers.indexOfLast { it % 2 == 1})  // 2
```

有序列表中的二分搜索

還有一種在列表中搜索元素的方法——二分搜索。它比其他內建搜索函數快得多，但要求列表按某個順序升序排序：自然順序或函數參數中提供的其他順序。否則，結果是未定義的。

要在排序列表中搜索元素，調用 `binarySearch()` 函數並傳遞值作為參數。如果存在這樣的元素，該函數返回其索引；否則，它返回 `(-insertionPoint - 1)`，其中 `insertionPoint` 是應插入此元素以保持列表排序的索引。如果有多個具有給定值的元素，搜索可以返回其中任何一個的索引。

你還可以指定一個索引範圍進行搜索：在這種情況下，該函數僅在提供的兩個索引之間搜索。

```
val numbers = mutableListOf("one", "two", "three", "four")
numbers.sort()
println(numbers)  // [four, one, three, two]
println(numbers.binarySearch("two"))  // 3
println(numbers.binarySearch("z")) // -5
println(numbers.binarySearch("two", 0, 2))  // -3
```

使用Comparator的二分搜索

當列表元素不可比較時，你應該提供一個 `Comparator` 來使用二分搜索。列表必須根據此 `Comparator` 按升序排序。讓我們看一個例子：

```
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 }))
```

這裡有一個 `Product` 實例的列表，這些實例不可比較，還有一個定義順序的 `Comparator`：如果 `p1` 的價格小於 `p2` 的價格，則 `p1` 在 `p2` 之前。因此，擁有按此順序升序排列的列表，我們使用 `binarySearch()` 來查找指定 `Product` 的索引。

當列表使用不同於自然順序的順序時，自定義比較器也很方便，例如，對字符串元素使用不區分大小寫的順序。

```
val colors = listOf("Blue", "green", "ORANGE", "Red", "yellow")
println(colors.binarySearch("RED", String.CASE_INSENSITIVE_ORDER)) // 3
```

使用比較函數的二分搜索

使用比較函數進行二分搜索可以讓你在不提供顯式搜索值的情況下查找元素。相反，它接受一個將元素映射為 `Int` 值的比較函數，並搜索該函數返回零的元素。列表必須按所提供函數的升序排序；換句話說，從一個列表元素到下一個列表元素，比較返回值必須遞增。

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

fun priceComparison(product: Product, price: Double) = sign(product.price - price).toInt()

fun main() {
    val productList = listOf(
        Product("WebStorm", 49.0),
        Product("AppCode", 99.0),
        Product("DotTrace", 129.0),
        Product("ReSharper", 149.0))

    println(productList.binarySearch { priceComparison(it, 99.0) })  // 1
}
```

比較器和比較函數的二分搜索也可以在列表範圍內執行。

列表寫操作

除了在集合寫操作中描述的集合修改操作外，可變列表還支持特定的寫操作。這些操作使用索引訪問元素，以擴展列表的修改功能。

添加

要將元素添加到列表的特定位置，使用`add()`和`addAll()`並提供元素插入位置作為附加參數。所有位於該位置之後的元素將右移。

```
val numbers = mutableListOf("one", "five", "six")
numbers.add(1, "two")
numbers.addAll(2, listOf("three", "four"))
println(numbers)  // [one, two, three, four, five, six]
```

更新

列表還提供了替換給定位置的元素的函數——`set()`及其操作符形式`[]`。`set()`不會改變其他元素的索引。

```
val numbers = mutableListOf("one", "five", "three")
numbers[1] =  "two"
println(numbers)  // [one, two, three]
```

`fill()`簡單地將所有集合元素替換為指定值。

```
val numbers = mutableListOf(1, 2, 3, 4)
numbers.fill(3)
println(numbers)  // [3, 3, 3, 3]
```

移除

要從列表中移除特定位置的元素，使用`removeAt()`函數並提供位置作為參數。被移除元素之後的所有元素索引將減少一。

```
val numbers = mutableListOf(1, 2, 3, 4, 3)
numbers.removeAt(1)
println(numbers)  // [1, 3, 4, 3]
```

排序

在集合排序中，我們描述了按特定順序檢索集合元素的操作。對於可變列表，標準庫提供了類似的擴展函數，這些函數執行相同的排序操作。當你將這樣的操作應用於列表實例時，它會改變該實例中元素的順序。

原地排序函數的名稱與適用於只讀列表的函數名稱相似，但沒有ed/d後

綴：

- 所有排序函數的名稱中用`sort*`代替`sorted*`：`sort()`、`sortDescending()`、`sortBy()`等。
- `shuffle()`代替`shuffled()`。
- `reverse()`代替`reversed()`。

調用`asReversed()`於可變列表時，返回的另一個可變列表是原列表的反轉視圖。該視圖中的變更會反映在原始列表中。以下示例顯示了可變列表的排序函數：

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

numbers.sort()
println("Sort into ascending: $numbers")  // [four, one, three, two]
numbers.sortDescending()
println("Sort into descending: $numbers")  // [two, three, one, four]

numbers.sortBy { it.length }
println("Sort into ascending by length: $numbers")  // [one, two, four, three]
numbers.sortByDescending { it.last() }
println("Sort into descending by the last letter: $numbers")  // [four, two, one, three]

numbers.sortWith(compareBy<String> { it.length }.thenBy { it })
println("Sort by Comparator: $numbers")  // [one, two, four, three]

numbers.shuffle()
println("Shuffle: $numbers")  // [three, one, two, four]
```

```
numbers.reverse()
println("Reverse: $numbers")  // [four, two, one, three]
```

這些操作使你能夠靈活地管理和修改列表中的元素，以滿足各種需求。

Set特定操作

Kotlin集合包中包含了一些常見操作的擴展函數，如查找交集、合併或從另一個集合中減去集合。

合併集合

要將兩個集合合併為一個集合，使用`union()`函數。它可以以中綴形式使用，即`a union b`。注意，對於有序集合，操作數的順序很重要。在結果集合中，第一個操作數的元素排在第二個操作數的元素之前：

```
val numbers = setOf("one", "two", "three")

// 按順序輸出
println(numbers union setOf("four", "five"))
// [one, two, three, four, five]
println(setOf("four", "five") union numbers)
// [four, five, one, two, three]
```

查找交集和差集

要查找兩個集合之間的交集（即同時存在於兩個集合中的元素），使用`intersect()`函數。要查找不在另一個集合中的集合元素，使用`subtract()`函數。這兩個函數也可以以中綴形式調用，例如`a intersect b`：

```
val numbers = setOf("one", "two", "three")

// 相同輸出
println(numbers intersect setOf("two", "one"))
// [one, two]
println(numbers subtract setOf("three", "four"))
// [one, two]
println(numbers subtract setOf("four", "three"))
// [one, two]
```

查找對稱差

要查找存在於兩個集合中的任一集合但不在其交集中存在的元素（即對稱差），你可以使用`union()`函數。對於這種操作，計算兩個集合之間的差異並合併結果：

```
val numbers = setOf("one", "two", "three")
val numbers2 = setOf("three", "four")

// 合併差異
println((numbers - numbers2) union (numbers2 - numbers))
// [one, two, four]
```

列表的集合操作

你也可以將`union()`、`intersect()`和`subtract()`函數應用於列表。不過，它們的結果總是set。在這個結果中，所有的重複元素都會合併為一個，且無法進行索引訪問：

```
val list1 = listOf(1, 1, 2, 3, 5, 8, -1)
val list2 = listOf(1, 1, 2, 2, 3, 5)

// 兩個列表相交的結果是一個集合
println(list1 intersect list2)
// [1, 2, 3, 5]

// 相同元素合併為一個
println(list1 union list2)
// [1, 2, 3, 5, 8, -1]
```

這些操作使你能夠靈活地管理和操作set，以滿足不同的需求。

Map 特定操作

在 Map 中，鍵和值的類型是用戶定義的。基於鍵的訪問使得 Map 特定的處理功能變得可能，從通過鍵獲取值到分別過濾鍵和值。在本頁中，我們將介紹標準庫中的 Map 處理函數。

檢索鍵和值

要從 Map 中檢索值，必須提供鍵作為 `get()` 函數的參數。也支持簡寫的 `[key]` 語法。如果找不到給定的鍵，則返回 `null`。還有一個 `getValue()` 函數，其行為略有不同：如果在 Map 中找不到鍵，則會拋出異常。此外，還有兩個選項來處理鍵的缺失：

- `getOrElse()` 的工作方式與列表相同：不存在鍵的值從給定的 lambda 函數返回。
- `getOrDefault()` 在找不到鍵時返回指定的默認值。

```
val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)
println(numbersMap.get("one"))         // 1
println(numbersMap["one"])             // 1
println(numbersMap.getOrDefault("four", 10))  // 10
println(numbersMap["five"])            // null
//numbersMap.getValue("six")           // exception!
```

要對 Map 的所有鍵或所有值執行操作，可以分別從 `keys` 和 `values` 屬性中檢索它們。`keys` 是一組所有 Map 鍵，`values` 是所有 Map 值的集合。

```
val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)
println(numbersMap.keys)               // [one, two, three]
println(numbersMap.values)             // [1, 2, 3]
```

過濾

你可以使用 `filter()` 函數來過濾 Map，與其他集合一樣。當對 Map 調用 `filter()` 時，將一個帶有 `Pair` 作為參數的謂詞(predicate)傳遞給它。這使得你可以在過濾謂詞中同時使用鍵和值。

```
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}
println(filteredMap)  // {key11=11}
```

還有兩種特定的過濾 Map 的方法：按鍵和按值。對於每種方法，都有一個函數：`filterKeys()` 和 `filterValues()`。這兩個函數都返回一個新的 Map，其中包含匹配給定謂詞的 entries。`filterKeys()` 的謂詞只檢查元素的鍵，`filterValues()` 的謂詞只檢查值。

```
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)  // {key1=1, key11=11}
println(filteredValuesMap)  // {key1=1, key2=2, key3=3}
```

加號和減號操作符

由於基於鍵訪問元素，加號 `(+)` 和減號 `(-)` 操作符在 Map 中的工作方式與其他集合不同。加號返回一個包含其兩個操作數元素的 Map：左側的 Map 和右側的 `Pair` 或另一個 Map。當右側操作數包含鍵存在於左側 Map 中的 entries 時，結果 Map 包含來自右側的 entries。

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

減號從左側 Map 中的 entries 創建一個 Map，除了右側操作數中的鍵。所以，右側操作數可以是單個鍵或鍵的集合：列表、集合等。

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

有關在可變 Map 上使用加號賦值 `(+=)` 和減號賦值 `(-=)` 操作符的詳細信息，請參見下面的 Map 寫操作。

Map 寫操作

可變 Map(Mutable maps) 提供 Map 特定的寫操作。這些操作允許你使用基於鍵的訪問來更改 Map 內容。

有些規則定義了 Map 上的寫操作：

- 值可以更新。反之，鍵永遠不變：一旦添加 entry，其鍵是固定的。
- 對於每個鍵，總是有一個與之關聯的單個值。你可以添加和移除整個 entry。

以下是標準庫中可變 Map 上可用的寫操作函數的描述。

添加和更新 entry

要向可變 Map 添加新的鍵值對，使用 `put()`。當將新 entry 放入 `LinkedHashMap`（默認 Map 實現）時，它會被添加，使其在迭代 Map 時出現在最後。在排序 Map 中，新元素的位置由其鍵的順序定義。

```
val numbersMap = mutableMapOf("one" to 1, "two" to 2)
numbersMap.put("three", 3)
println(numbersMap)  // {one=1, two=2, three=3}
```

要一次添加多個 entry，使用 `putAll()`。其參數可以是 `Map` 或一組 `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)  // {one=1, two=2, three=3, four=4, five=5}
```

`put()` 和 `putAll()` 都會覆蓋已存在鍵的值。因此，你可以使用它們來更新 Map entries 的值。

```
val numbersMap = mutableMapOf("one" to 1, "two" to 2)
val previousValue = numbersMap.put("one", 11)
println("value associated with 'one', before: $previousValue, after: ${numbersMap["one"]}")
println(numbersMap)  // {one=11, two=2}
```

你也可以使用簡寫操作符形式向 Map 添加新 entry。有兩種方式：

- `plusAssign (+=)` 操作符。
- `[]` 操作符的 `set()` 別名。

```
val numbersMap = mutableMapOf("one" to 1, "two" to 2)
numbersMap["three"] = 3     // 調用 numbersMap.put("three", 3)
numbersMap += mapOf("four" to 4, "five" to 5)
println(numbersMap)  // {one=1, two=2, three=3, four=4, five=5}
```

當鍵存在於 Map 中時，操作符會覆蓋相應 entry 的值。

移除 entry

要從可變 Map 中移除 entry，使用 `remove()` 函數。調用 `remove()` 時，可以傳遞鍵或整個鍵值對。如果同時指定了鍵和值，則只有當值與第二個參數匹配時，該鍵的元素才會被移除。

```
val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3)
numbersMap.remove("one")
println(numbersMap)  // {two=2, three=3}
numbersMap.remove("three", 4)  // 不移除任何東西
println(numbersMap)  // {two=2, three=3}
```

你也可以根據鍵或值從可變 Map 中移除 entries。要做到這一點，調用 `remove()` 在 Map 的 `

keys`或`values`中提供鍵或值。如果調用`values`，`remove()` 只會移除第一個與給定值匹配的 entry。

```
val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3, "threeAgain" to 3)
numbersMap.keys.remove("one")
println(numbersMap)  // {two=2, three=3, threeAgain=3}
numbersMap.values.remove(3)
println(numbersMap)  // {two=2, threeAgain=3}
```

`minusAssign (-=)` 操作符同樣適用於可變 Map。

```
val numbersMap = mutableMapOf("one" to 1, "two" to 2, "three" to 3)
numbersMap -= "two"
println(numbersMap)  // {one=1, three=3}
numbersMap -= "five"  // 不移除任何東西
println(numbersMap)  // {one=1, three=3}
```

修訂版本 #2
★由 treeman 建立於 30 C 2024 18:34:06
由 treeman 更新於 31 C 2024 10:36:13