【Kotlin】Scope functions (run , let, apply, also, let, takeIf and takeUnless)

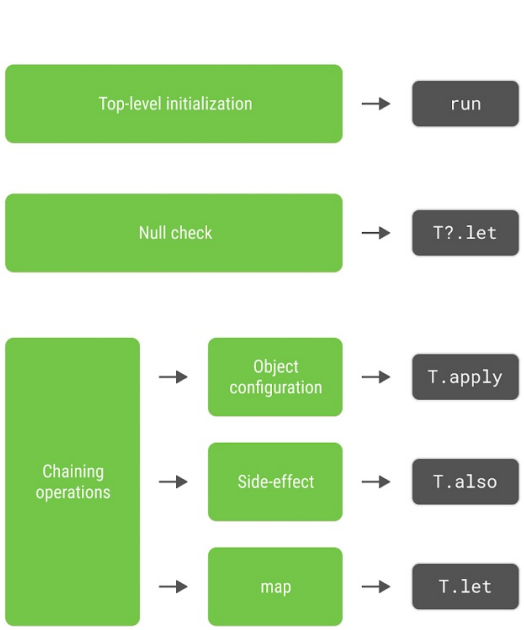
- 出處
- https://developer.android.com/codelabs/java-to-kotlin#0
- https://kotlinlang.org/docs/scope-functions.html

```
// 原寫法
        val mary = Person("mary")
        mary.age = 20
        mary.birthplace = "NewYork"
        println(mary) // name:mary, age:20, birthplace:NewYork

// 使用scope function 簡化語法加強語意
        val john = Person("John") .apply {
           age = 18
           birthplace = "Taipei"
        }
  println(john) // name:John, age:18, birthplace:Taipei
```

```
// Scope functions 可以做到一樣的事，只是寫法不同
// 印出 list 字元長度大於3 的資料
// let, also, run, apply, with
fun main() {

    val numbers = mutableListOf("one", "two", "three", "four", "five")
   // [3, 3, 5, 4, 4]

    // it代表自己
 numbers.map { it.length }.filter { it > 3 }.let {
     println(it)
    }
    // 命名(item)代表自己
    numbers.map { it.length }.filter { it > 3 }.let { item ->
     println(item)
    }

    // it代表自己
    numbers.map { it.length }.filter { it > 3 }.also {
     println(it)
    }
    // 命名(item)代表自己
    numbers.map { it.length }.filter { it > 3 }.also { item ->
     println(item)
    }

    // this代表自己
    numbers.map { it.length }.filter { it > 3 }.run {
     println(this)
    }

    // this代表自己
    numbers.map { it.length }.filter { it > 3 }.apply {
     println(this)
    }

    // this代表自己
 val gt3Numbers = numbers.map { it.length }.filter { it > 3 }
    with(gt3Numbers){
     println(this)
```

```
 }
}

// 結果都是 [5, 4, 4]
```

[官方]判斷如何使用何種function




```
val myObj = run {
    val generator = DateStringGenerator(2008, 9, 23)
    generator.locale = "US"
    generator.localizedDate
}
```

```
class MyClass {

    fun printExceptionMessage(exception: Exception?) {
        exception?.let {
            println(exception.message)
        }
    }
```

```
    val lastWeeksDate: String = Calendar.getInstance().apply {
        add(Calendar.DAY_OF_YEAR, -7)
    }
```

```
        .also {
            println("Created calendar of type: ${it.calendarType}")
        }
```

```
        .let { calendar ->
            val format = SimpleDateFormat("dd/M/yyyy hh:mm:ss", Locale.US)
            format.format(calendar.time)
        }
}
```


功能	物件引用	傳回值	是擴充函數	
let	it	Lambda result	Yes	使用這個物件，做以下操作，並返回最後一個操作 非null物件處理
run	this	Lambda result	Yes	lambda 同時初始化對象和計算返回值
run	-	Lambda result	否：在沒有上下文物件的情況下調用	運行代碼塊並計算結果(lambda)
with	this	Lambda result	否：將上下文物件作為參數。	使用這個物件，做以下操作
apply	this	Context object	Yes	將以下賦值應用到物件上
also	it	Context object	Yes	並且還可以對該物件執行以下操作

回傳值 \ 傳入參考物件	this	lambda(it)
this	apply	also
bock return (回傳最後一行結果)	run	let

下面的專門部分提供了有關這些功能的詳細資訊。
以下是根據預期目的選擇作用域函數的簡短指南：

- 對不可為 null 的物件執行 lambda：let
- 在局部範圍內引入表達式作為變數：let
- 物件配置：apply
- 物件配置和計算結果(初始化物件)：run
- 在需要表達式的地方運行語句(lambda)：非擴展run
- 附加效果：also
- 將物件的函數進行分組呼叫(Grouping function calls on an objec)：with

差異

由於作用域函數本質上很相似，因此了解它們之間的差異非常重要。每個作用域函數之間有兩個主要差異：

- 他們引用上下文物件(Context object)的方式。
- 他們的返回值。

在傳遞給作用域函數的 lambda 中，上下文物件可透過短引用而不是其實際名稱來取得。
每個作用域函數使用兩種方式之一來引用上下文物件：
作為 lambda接收器(`this`) 或作為 lambda 參數 (`it`)。
兩者都提供相同的功能，因此我們針對不同的用例描述了每種方法的優缺點，並提供了使用建議。

```
fun main() {
    val str = "Hello"
    // this
    str.run {
        println("The string's length: $length")
        //println("The string's length: ${this.length}") // does the same
    }

    // it
    str.let {
        println("The string's length is ${it.length}")
    }
}
```

this

run、with、 並透過關鍵字apply引用上下文物件作為 lambda接收器this。因此，在它們的 lambda 中，物件就像在普通類別函數中一樣可用。

大多數情況下，您可以this在存取接收者物件的成員時省略，從而使程式碼更短。另一方面，如果this省略，則很難區分接收者成員和外部物件或函數。
因此this，對於主要透過呼叫其函數或為屬性賦值來操作物件成員的 lambda，建議將上下文物件作為接收者 ()。

```
val adam = Person("Adam")
  .apply {
    age = 20 // same as this.age = 20
    city = "London"
  }
println(adam)
```

it

反過來說， `let` 和 `also` 會將上下文對象作為 lambda 參數引用。如果沒有指定參數名稱，則會使用默認名稱 `it` 來訪問對象。`it` 比 `this` 短，並且使用 `it` 的表達式通常更易於閱讀。

然而，當調用對象的函數或屬性時，你無法像使用 `this` 一樣隱式地獲取對象。因此，當對象主要用作函數調用中的參數時，通過 `it` 訪問上下文對象更好。如果在代碼塊中使用多個變量，使用 `it` 也是更好的選擇。

```
fun getRandomInt(): Int {
    return Random.nextInt(100).also {
        writeToLog("getRandomInt() generated value $it")
    }
}

val i = getRandomInt()
println(i)

// INFO: getRandomInt() generated value 78
// 78
```

以下範例展示了如何將上下文對象作為帶有參數名稱 `value` 的 lambda 參數引用：

```
fun getRandomInt(): Int {
    return Random.nextInt(100).also { value ->
        writeToLog("getRandomInt() generated value $value")
    }
}

val i = getRandomInt()
```

```
println(i)

// INFO: getRandomInt() generated value 4
// 4
```

傳回值

作用域函數因其傳回的結果而有所不同：
您應該根據您接下來想要在程式碼中執行的操作仔細考慮您想要的回傳值。這可以幫助您選擇要使用的最佳範圍函數。

上下文對象 (Context object)

`apply` 和 `also` 的返回值是上下文對象本身。因此，它們可以作為副步驟包含在調用鏈中：你可以在同一對象上連續調用函數。

```
val numberList = mutableListOf<Double>()
numberList.also { println("Populating the list") }
    .apply {
        add(2.71)
        add(3.14)
        add(1.0)
    }
    .also { println("Sorting the list") }
    .sort()
```

它們也可以用於返回上下文對象的函數的返回語句中。

```
fun getRandomInt(): Int {
    return Random.nextInt(100).also {
        writeToLog("getRandomInt() generated value $it")
    }
}

val i = getRandomInt()
```

Lambda result

`let`、`run` 和 `with` 返回的是 lambda 表達式的結果。因此，你可以在將結果賦值給變量、在結果上鏈式操作等情況下使用它們。

```
val numbers = mutableListOf("one", "two", "three")
val countEndsWithE = numbers.run {
    add("four")
    add("five")
    count { it.endsWith("e") }
}
println("There are $countEndsWithE elements that end with e.")

// There are 3 elements that end with e.
```

此外，你可以忽略返回值，使用作用域函數為局部變量創建一個臨時作用域。

```
val numbers = mutableListOf("one", "two", "three")
with(numbers) {
    val firstItem = first()
    val lastItem = last()
    println("First item: $firstItem, last item: $lastItem")
}

// First item: one, last item: three
```

Functions

為了幫助你選擇適合你的使用場景的作用域函數，我們將詳細描述它們並提供使用建議。

let

- 上下文物件可用作參數 (it)。
- 傳回值是 lambda 結果。

在代碼中，`let` 可以理解為“使用這個物件，做以下操作，並返回最後一個操作”

let可用於在呼叫鏈的結果上呼叫一個或多個函數。例如，以下程式碼列印集合上兩個操作的結果：

```
val numbers = mutableListOf("one", "two", "three", "four", "five")
val resultList = numbers.map { it.length }.filter { it > 3 }
println(resultList)

// [5, 4, 4]
```

使用let，您可以重寫上面的範例，這樣您就不會將清單操作的結果指派給變數：

```
val numbers = mutableListOf("one", "two", "three", "four", "five")
numbers.map { it.length }.filter { it > 3 }.let {
    println(it)
    // and more function calls if needed
}

// [5, 4, 4]
```

如果傳遞給的程式碼區塊let包含單一函數作為it參數，則可以使用方法來引用 (::) 而不是 lambda 參數：

```
val numbers = mutableListOf("one", "two", "three", "four", "five")
numbers.map { it.length }.filter { it > 3 }.let(::println)

// [5, 4, 4]
```

`let` 通常用於執行包含非空值的代碼塊。要對非空對象執行操作，可以使用安全調用運算符 `?.`，並在其 lambda 中調用 `let` 來執行操作。

```
val str: String? = "Hello"
//processNonNullString(str)       // compilation error: str can be null
val length = str?.let {
    println("let() called on $it")
    processNonNullString(it)      // OK: 'it' is not null inside '?.let { }'
    it.length
}
```

你也可以使用 `let` 引入具有有限作用域的局部變量，以使你的代碼更易於閱讀。要為上下文對象定義一個新變量，可以將其名稱作為 lambda 參數提供，這樣它可以代替默認的 `it` 使用。

```
val numbers = listOf("one", "two", "three", "four")
val modifiedFirstItem = numbers.first().let { firstItem ->
    println("The first item of the list is '$firstItem'")
    if (firstItem.length >= 5) firstItem else "!" + firstItem + "!"
}.uppercase()
println("First item after modifications: '$modifiedFirstItem'")

// The first item of the list is 'one'
// First item after modifications: '!ONE!'
```

with

- 上下文物件可用作接收器 (this)。
- 傳回值是 lambda 結果。

由於 `with` 不是擴展函數：上下文對象作為參數傳遞，但在 lambda 內部，它可以作為接收者 (`this`) 使用。

我們建議在不需要返回結果時使用 `with` 來調用上下文對象上的函數。
在代碼中，`with` 可以理解為“使用這個對象，做以下操作。”

```
val numbers = mutableListOf("one", "two", "three")
with(numbers) {
    println("'with' is called with argument $this")
    println("It contains $size elements")
}

// 'with' is called with argument [one, two, three]
// It contains 3 elements
```

你也可以使用 `with` 引入一個輔助對象，其屬性或函數用於計算值。

```
val numbers = mutableListOf("one", "two", "three")
val firstAndLast = with(numbers) {
```

```
    "The first element is ${first()}," +
    " the last element is ${last()}"
}
println(firstAndLast)
```

run

- 上下文物件可用作接收器 (this)。
- 傳回值是 lambda 結果。

`run` 的功能與 `with` 相似，但它是作為擴展函數實現的。因此，像 `let` 一樣，你可以使用點符號在上下文對象上調用它。

`run` 當你的 lambda 同時初始化對象和計算返回值時非常有用。

```
val service = MultiportService("https://example.kotlinlang.org", 80)

val result = service.run {
    port = 8080
    query(prepareRequest() + " to port $port")
}

// the same code written with let() function:
val letResult = service.let {
    it.port = 8080
    it.query(it.prepareRequest() + " to port ${it.port}")
}

// Result for query 'Default request to port 8080'
// Result for query 'Default request to port 8080'
```

你也可以以非擴展函數的形式調用 `run`。非擴展變體的 `run` 沒有上下文對象，但仍然返回 lambda 的結果。非擴展 `run` 允許你在需要表達式的地方執行多個語句的代碼塊。在代碼中，非擴展 `run` 可以理解為“運行代碼塊並計算結果。”

```
val hexNumberRegex = run {
    val digits = "0-9"
    val hexDigits = "A-Fa-f"
    val sign = "+-"

    Regex("[$sign]?[$digits$hexDigits]+")
}

for (match in hexNumberRegex.findAll("+123 -FFFF !%*& 88 XYZ")) {
    println(match.value)
}

// +123
// -FFFF
// 88
```

```
val getHexNumberRegex: () -> Regex = {
    val digits = "0-9"
    val hexDigits = "A-Fa-f"
    val sign = "+-"

    return Regex("[$sign]?[$digits$hexDigits]+")
}
```

apply

- 上下文物件可用作接收器 (this)。
- 傳回值是物件本身。

`apply` 返回上下文對象本身，因此建議在不返回值的代碼塊中使用它，並且主要操作接收者對象的成員。`apply` 的最常見用例是對對象進行配置。這類調用可以理解為“將以下賦值應用到物件上。”

```
val adam = Person("Adam").apply {
    age = 32
    city = "London"
}
```

```
println(adam)

 // Person(name=Adam, age=32, city=London)
```

另一個 `apply` 的使用場景是將其包含在多個調用鏈中以進行更複雜的處理。

```
data class Address(var street: String = "", var city: String = "")
data class Person(var name: String = "", var address: Address = Address())

val person = Person().apply {
    name = "Alice"
    address = Address().apply {
        street = "123 Main St"
        city = "Wonderland"
    }
}

println(person) // Output: Person(name=Alice, address=Address(street=123 Main St, city=Wonderland))
```

在這個例子中，我們使用 `apply` 配置 `Person` 對象的屬性，並在內部使用另一個 `apply` 來配置 `Address` 對象。這樣的鏈式調用使得代碼結構更清晰，易於維護。

also

- 上下文物件可用作參數 (it)。
- 傳回值是物件本身。

`also` 對於執行一些需要上下文對象作為參數的操作非常有用。使用 `also` 用於那些需要對象引用而不是其屬性和函數的情況，或者當你不想影響外部作用域的 `this` 參考時。
當您also在程式碼中看到時，您可以將其讀作"並且還可以對該物件執行以下操作。" (or 插入log ??)

```
val numbers = mutableListOf("one", "two", "three")
numbers
    .also { println("The list elements before adding new one: $it") }
    .add("four")


// The list elements before adding new one: [one, two, three]
```

takeIf and takeUnless

`takeIf` 和 `takeUnless` 是 Kotlin 標準庫中的函數，允許你在調用鏈中嵌入對對象狀態的檢查。

當對一個對象調用 `takeIf` 並傳遞一個條件時，如果該對象滿足該條件，則返回這個對象；否則返回 `null`。因此，`takeIf` 是針對單個對象的過濾函數。

```
val number = Random.nextInt(100)

val evenOrNull = number.takeIf { it % 2 == 0 }
val oddOrNull = number.takeUnless { it % 2 == 0 }
println("even: $evenOrNull, odd: $oddOrNull")

// even: 62, odd: null
```

在使用 `takeIf` 和 `takeUnless` 之後鏈式調用其他函數時，請不要忘記執行空值檢查或使用安全調用運算符 `?.`，因為它們的返回值是可為空的。

```
val str = "Hello"
val caps = str.takeIf { it.isNotEmpty() }?.uppercase()
//val caps = str.takeIf { it.isNotEmpty() }.uppercase() //compilation error
println(caps)

// HELLO
```

`takeIf` 和 `takeUnless` 與作用域函數結合使用時特別有用。例如，你可以將 `takeIf` 和 `takeUnless` 與 `let` 鏈式調用，以便在符合給定條件的對象上執行代碼塊。具體來說，先對對象調用 `takeIf`，然後使用安全調用（`?.`）調用 `let`。對於不符合條件的對象，`takeIf` 會返回 `null`，因此 `let` 不會被調用。

```
fun displaySubstringPosition(input: String, sub: String) {
    input.indexOf(sub).takeIf { it >= 0 }?.let {
        println("The substring $sub is found in $input.")
        println("Its start position is $it.")
    }
}

displaySubstringPosition("010000011", "11")
displaySubstringPosition("010000011", "12")

// The substring 11 is found in 010000011.
// Its start position is 7.
```

以下是一個示例，展示如何在不使用 `takeIf` 或作用域函數的情況下編寫相同的功能：

```
fun displaySubstringPosition(input: String, sub: String) {
    val index = input.indexOf(sub)
    if (index >= 0) {
        println("The substring $sub is found in $input.")
        println("Its start position is $index.")
    }
}

displaySubstringPosition("010000011", "11")
displaySubstringPosition("010000011", "12")

// The substring 11 is found in 010000011.
// Its start position is 7.
```

⏲修訂版本 #31
★由 treeman 建立於 17 C 2024 10:21:21
✎由 treeman 更新於 31 C 2024 10:36:13