

# 【kotlin】Inline functions (官方文件翻譯)

出處 <https://kotlinlang.org/docs/inline-functions.html>

使用高階函式會帶來某些執行時期的負擔：每個函式都是一個物件，而且會捕捉到閉包。閉包是可以在函式內部存取的變數範圍。記憶體配置（包括函式物件和類別的配置）以及虛擬呼叫都會引入執行時期的額外負擔。

但在許多情況下，這種額外負擔可以通過內聯化 lambda 表達式來消除。以下展示的函式就是這種情況的好例子。`lock()` 函式可以很容易地在呼叫點內聯化。請考慮以下情況：

```
lock(l) { foo() }
```

與其為參數創建一個函式物件並生成呼叫，編譯器可以發出以下程式碼：

```
l.lock()
try {
    foo()
} finally {
    l.unlock()
}
```

要讓編譯器這樣做，可以使用 `inline` 修飾符標記 `lock()` 函式：

```
inline fun <T> lock(lock: Lock, body: () -> T): T { ... }
```

`inline` 修飾符會影響函式本身和傳遞給它的 lambda 表達式：所有這些都會內聯到呼叫點。

內聯可能會導致生成的程式碼變大。然而，如果合理地進行內聯（避免內聯大型函式），在效能方面將會有顯著的提升，特別是在迴圈內的“多態性”呼叫點。

## noinline

如果你不希望傳遞給內聯函式的所有 lambda 表達式都被內聯，可以使用 `noinline` 修飾符標記部分函式參數：

```
inline fun foo(inlined: () -> Unit, noinline notInlined: () -> Unit) { ... }
```

可內聯的 lambda 表達式只能在內聯函式內被呼叫，或作為可內聯的參數傳遞。然而，使用 `noinline` 修飾符的 lambda 表達式則可以以任何方式操作，包括儲存在欄位中或傳遞。

如果一個內聯函式沒有可內聯的函式參數且沒有具現化的型別參數，編譯器會發出警告，因為內聯這樣的函式不太可能有任何好處（如果你確定需要內聯，可以使用 `@Suppress("NOTHING_TO_INLINE")` 註解來抑制這個警告）。

## non-local returns 非本地返回

在 Kotlin 中，你只能在命名函式或匿名函式中使用普通的未修飾返回來退出。要退出 lambda，則需要使用標籤。在 lambda 內部禁止使用裸的返回，因為 lambda 不能使包含它的函式返回。

```
fun foo() {
    ordinaryFunction {
        return // ERROR: cannot make `foo` return here
    }
}
```

但是，如果傳遞 lambda 的函式被內聯，則返回語句也會被內聯，因此這樣做是允許的：

```
fun foo() {
    inlined {
        return // OK: the lambda is inlined
    }
}
```

這種在 lambda 內部但退出包含函式的返回稱為非本地返回。這種構造通常出現在循環中，而內聯函式通常會包裹這樣的循環。

```
fun hasZeros(ints: List<Int>): Boolean {
    ints.forEach {
```

```

        if (it == 0) return true // returns from hasZeros
    }
    return false
}

```

注意，一些內聯函式可能會從不同的執行上下文（如局部物件或巢狀函式）而不是直接從函式主體內調用作為參數傳遞給它們的 lambda。在這種情況下，lambda 中也不允許非本地控制流。為了指示內聯函式的 lambda 參數不能使用非本地返回，可以使用 `crossinline` 修飾符來標記 lambda 參數：

```

inline fun f(crossinline body: () -> Unit) {
    val f = object: Runnable {
        override fun run() = body()
    }
    // ...
}

```

`break` 和 `continue` 在內聯 lambda 中目前還不可用，但我們計劃未來也支援它們。

## Reified type parameters 具現化（reified）型別參數

有時你需要訪問作為參數傳遞的型別：

```

fun <T> TreeNode.findParentOfType(clazz: Class<T>): T? {
    var p = parent
    while (p != null && !clazz.isInstance(p)) {
        p = p.parent
    }
    @Suppress("UNCHECKED_CAST")
    return p as T?
}

```

這裡，你遍歷一棵樹並使用反射來檢查節點是否具有某種類型。這一切都很好，但呼叫點並不十分美觀：

```

treeNode.findParentOfType(MyTreeNode::class.java)

```

更好的解決方案是直接將型別作為參數傳遞給這個函式。你可以這樣調用它：

```

treeNode.findParentOfType<MyTreeNode>()

```

為了實現這一點，內聯函式支持具現化（reified）型別參數，因此你可以這樣寫：

```

inline fun <reified T> TreeNode.findParentOfType(): T? {
    var p = parent
    while (p != null && p !is T) {
        p = p.parent
    }
    return p as T?
}

```

上述代碼使用 `reified` 修飾符來賦予型別參數在函式內部可訪問的能力，幾乎就像它是一個普通的類別一樣。由於函式被內聯，不需要使用反射，並且現在你可以使用像 `!is` 和 `as` 這樣的普通操作符。同時，你可以像上面展示的那樣調用這個函式：`myTree.findParentOfType<MyTreeNodeType>()`。

雖然在許多情況下不需要使用反射，但你仍然可以使用具現化（reified）型別參數來使用它：

```

inline fun <reified T> membersOf() = T::class.members

fun main(s: Array<String>) {
    println(membersOf<StringBuilder>().joinToString("\n"))
}

```

普通函式（未標記為內聯）不能具有具現化（reified）參數。沒有運行時表示（例如非具現化的型別參數或類似 `Nothing` 的虛構型別）的型別不能作為具現化型別參數的參數。

## Inline properties

在 Kotlin 中，`inline` 修飾符確實可以用於沒有後端字段的屬性的訪問器上。你可以為個別的屬性訪問器添加註解：

```
val foo: Foo
    inline get() = Foo()

var bar: Bar
    get() = ...
    inline set(v) { ... }
```

你也可以對整個屬性進行註解，這將使其所有的訪問器都標記為內聯：

```
inline var bar: Bar
    get() = ...
    set(v) { ... }
```

在呼叫處，內聯訪問器會像普通的內聯函式一樣被內聯。

## restrictions for public API inline functions 公共 API 的內聯函式限制

當內聯函式是公共或受保護的，但不屬於私有或內部聲明的一部分時，它被視為模組的公共 API。它可以在其他模組中被調用並且在這些調用點被內聯。然而，這會帶來二進制不兼容性的風險，如果調用模組在聲明模組變更後未重新編譯。

為了消除由模組的非公共 API 變更引入的這種不兼容性風險，公共 API 的內聯函式不允許在其內部使用非公共 API 的聲明（私有和內部）。

內部聲明可以使用 `@PublishedApi` 註解，這允許它在公共 API 的內聯函式中使用。當內部的內聯函式標記為 `@PublishedApi` 時，其函式體被視為公共，並且受到與公共函式相同的可見性限制和檢查。

---

🕒 修訂版本 #6

★ 由 treeman 建立於 18 🕒 2024 10:02:17

✍ 由 treeman 更新於 31 🕒 2024 10:36:13