

# 【kotlin】因為需要 < reified T > 取得泛型型別，所以從 inline functions 開始

參考：

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

[泛型基礎 \(三\) — Java 與 Kotlin 向下相容、Type Erasure 和 Reifiable](#)

[Inline functions - Kotlin Vocabulary](#)

[Kotlin 的 inline、noinline、crossline function](#)

## 從Type Erasure 開始

型別擦除 (Type Erasure) 是 JVM 泛型的一個特性，在執行時會移除泛型型別資訊。這意味著在編譯後，所有的泛型型別資訊都會被擦除，導致在運行時無法直接訪問或判斷泛型的具體型別。

## 型別擦除的原因

型別擦除的主要原因是為了保持向後兼容性。Java 在設計泛型時，需要確保編譯後的程式碼能夠在不支持泛型的 JVM 上運行。通過在編譯時檢查型別並在運行時擦除泛型資訊，Java 能夠確保新舊版本的兼容性。

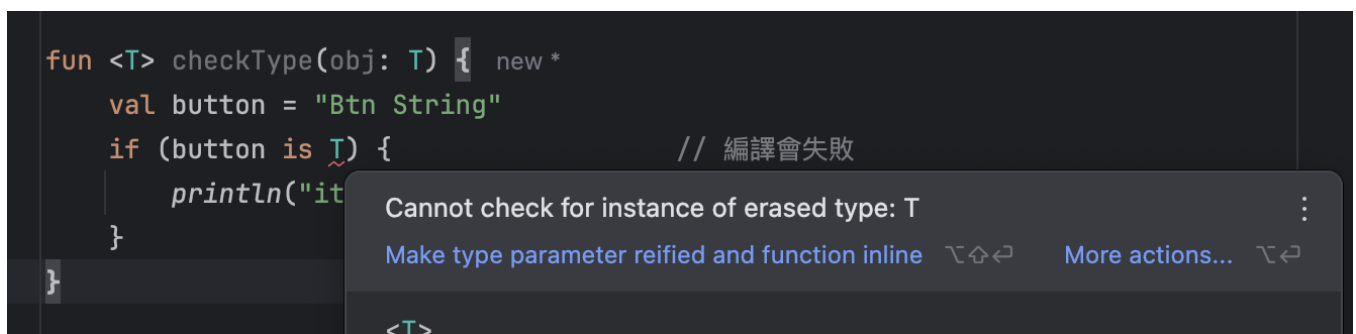
## 型別擦除的影響

型別擦除會導致一些限制和問題，包括：

### 1. 運行時無法獲取泛型資訊：

- 由於型別資訊在運行時被擦除，無法在運行時獲取泛型的具體型別。例如：

```
fun <T> checkType(obj: T) {  
    val button = "Btn String"  
    if (button is T) { // 編譯會失敗  
        println("it's String")  
    }  
}
```



### 2. 型別轉換的警告：

- 在編譯時會產生未經檢查的型別轉換警告，因為編譯器無法完全確保型別的安全性。

```
val rawList: List<*> = ArrayList<Any>()  
val stringList: List<String> = rawList as List<String> // 未經檢查的型別轉換警告
```

### 3. 創建泛型型別的實例：

- 由於運行時無法確定泛型的具體型別，無法直接創建泛型型別的實例。

```
class MyClass<T> {
```

```
fun createInstance(): T {
    return T() // 錯誤，無法創建泛型型別的實例
}
}
```

## 型別擦除的工作原理

型別擦除的工作原理如下：

**替換泛型參數：**

- 在編譯時，編譯器會用它們的邊界（上限）替換泛型參數。如果沒有明確的邊界，則使用 `Object` 作為默認邊界。例如，`List<T>` 會被替換為 `List<Object>`。

**插入型別檢查和轉換：**

- 在編譯後的程式碼中插入型別檢查和轉換，以確保在運行時的型別安全。例如：

```
List<String> stringList = new ArrayList<>();
stringList.add("Hello");
String str = stringList.get(0); // 編譯後，將插入型別轉換 (String)
```

## 克服型別擦除的方法

在某些情況下，可以使用以下方法克服型別擦除的限制：

**傳遞型別參數：**

- 可以通過傳遞 `Class<T>` 或 `TypeToken<T>` 來保留型別資訊。

```
public <T> T createInstance(Class<T> clazz) throws InstantiationException, IllegalAccessException {
    return clazz.newInstance();
}
```

**使用反射：**

- 使用反射可以在運行時獲取一些泛型資訊，儘管這可能會變得複雜且難以維護。

```
import java.lang.reflect.ParameterizedType
import java.lang.reflect.Type

// 定義一個泛型類
class GenericClass<T>

// 繼承泛型類並指定具體的泛型型別
class StringGenericClass : GenericClass<String>()

fun main() {
    // 創建一個 StringGenericClass 的實例
    val instance = StringGenericClass()

    // 獲取泛型類的超類型（即 GenericClass<String>）
    val superclass: Type = instance::class.java.genericSuperclass

    if (superclass is ParameterizedType) {
        // 獲取泛型參數的型別（即 String）
        val actualTypeArgument = superclass.actualTypeArguments[0]
        println("泛型參數的型別是：${actualTypeArgument.typeName}")
    } else {
        println("無法獲取泛型參數的型別")
    }
}
```

**Kotlin 的 `reified` 型別參數：**

- 在 Kotlin 中，使用 `reified` 關鍵字來保留泛型型別資訊，以便在運行時可以訪問這些資訊。

# 使用 reified 必須使用 inline function

所以...

## inline function

在 Kotlin 中，使用 `inline` 修飾符可以將函式的內容插入到每一個函式呼叫的地方，而不是像普通函式那樣進行函式調用。這樣做可以減少函式調用帶來的開銷，特別是當函式體非常簡單且多次調用時，有助於提升程式的執行效能。

```
// 原本的function 在呼叫到hello() 會去hello 執行內部 println 動作
fun main(args: Array<String>) {
    hello()
}

fun hello() {
    println("Hello World !!")
}

/**
 * inline functino (加了inline)
 * 編譯時會將hello 內容插入呼叫位置
 */
fun main(args: Array<String>) {
    println("Hello World !!") // <- 將hello 內容插入
}

inline fun hello() {
    println("Hello World !!")
}
```

也可以使用 lambda

```
fun main(args: Array<String>) {
    todo{
        println("hello world")
    }
}

inline fun todo(doSomething:()->Unit) {
    doSomething()
}
```

以下是使用 `inline` 修飾符的一些特點和注意事項：

- 函式內容插入**：被標記為 `inline` 的函式在被調用時，其函式體內的程式碼會被直接插入到呼叫處，而不是進行常規的函式調用。
- 減少函式調用開銷**：這樣做可以減少函式調用帶來的堆疊和參數傳遞開銷，特別是對於一些簡單的函式，可以有效提升執行效能。
- 限制和注意事項**：由於內聯函式將函式體內容插入到每一個呼叫點，所以函式體內不能使用非公共 API 的聲明（如私有或內部聲明）。這是為了避免在模組之間的二進制不兼容性問題。

總結來說，Kotlin 中的內聯函式是一種提高執行效能的機制，特別適合於一些較短和簡單的函式，可以有效地減少函式調用的開銷。

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
    }
    // 沒有reified 會發出警告用下面這行消除
    @Suppress("UNCHECKED_CAST")
    return p as T?
}
```

```
// 所以會長得像這樣
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?
}
```

```
inline fun <reified T> example(value: T) {
    // 在函式內部可以使用 T 的真實類型信息
    println("Type of value is: ${T::class.simpleName}")
}

fun main() {
    example(42) // 調用時傳入 Int
    example("Hello") // 調用時傳入 String
}

// Type of value is: Int
// Type of value is: String
```

```
private inline fun <reified T : BaseEntity> gsonProcessAndSaveMongodb(
    value: String,
    repo: MongoRepository<T, *>
): T {
    val data: T = gson.fromJson(value, T::class.java)
    logger.info("{} data is {}", T::class.simpleName, data)
    repo.save(data)
    return data
}

////////////////////////////////////
val topic: String = record.topic()
val key: String = removeRandomKey(record.key())
val value: String = record.value()
when (key) {
    APP_LIVE_USER -> {
        gsonProcessAndSaveMongodb(value, appLiveUserRepo)
    }

    APP_LIVE_PAGE_VIEW -> {
        gsonProcessAndSaveMongodb(value, appLivePageViewRepo)
    }
    .....
}
```

## 特點和優點

使用 `reified` 型別參數有幾個主要優點：

- **運行時類型資訊**：通過 `reified`，函式內部可以獲取到泛型型別的實際類型資訊，例如使用 `T::class` 可以獲取到 `T` 的 `KClass` 對象。
- **簡化語法**：不需要額外的類型轉換或者反射操作，使得程式碼更加清晰和簡潔。
- **效能**：因為函式是內聯的，且使用了 `reified` 型別，編譯器可以在編譯時期進行更多的優化，而無需再在運行時進行額外的操作。

## 注意事項

使用 `reified` 型別參數有一些限制和注意事項：

- 只能用於**內聯函式**中，因為它需要編譯器能夠直接插入泛型實體化的程式碼。
- 只能在函式內部使用 `T::class` 或類似的語法來獲取泛型型別的類型資訊，**無法在函式外部**進行操作。
- `reified` 型別參數**僅限於函式內部**，不影響函式的參數或返回類型的宣告。

總結來說，`reified` 型別參數是 Kotlin 中一個強大的功能，用於在泛型程式碼中取得運行時類型資訊，提升了程式碼的靈活性和效能。

## noinline

如果有多個參數，不想被inline 可以使用關鍵字 `noinline`

```
fun main(args: Array<String>) {
    todo({
        println("hello world")
    }, {
        println("hello world2")
    })
}

inline fun todo(
    doSomething: ()->Unit,
    noinline doSomething2: ()->Unit
) {
    doSomething()
    doSomething2()
}
```

## crossinline

`inline fun` 使用 `return` 會有意想不到的狀況，可使用 `crossinline` 強迫使用 `return@XXX` 退出

```
fun main(args: Array<String>) {
    todo({
        println("hello world")
        return // 因為使用inline，return 後以下不會執行
    }, {
        println("hello world2")
    })
}

inline fun todo(
    doSomething: ()->Unit,
    doSomething2: ()->Unit
) {
    doSomething()
    doSomething2()
}

// hello world
```

```
fun main(args: Array<String>) {
    todo({
        println("hello world")
        return@todo // 使用crossinline 會強迫使用 return@{function name} 跳出
    }, {
        println("hello world2")
    })
}

inline fun todo(
    crossinline doSomething: ()->Unit,
    doSomething2: ()->Unit
) {
    doSomething()
}
```

```
doSomething2()
}
// hello world
// hello world2
```

# Scope function 就是使用 inline function

@kotlin.internal.InlineOnly

表示這個函式是一個內聯函式，但它只能在 Kotlin 標準函式庫內部使用。這個註解通常用於指示編譯器只將這個函式內聯化，而不生成實際的函式調用。

## let

```
val str: String? = "Hello"
//processNonNullString(str)    // compilation error: str can be null
val length = str?.let {
    println("let() called on $it")
    it.length
}
println(length) // 5
```

**inline fun <T, R> T.let(block: (T) -> R): R** 函式定義：

- `inline fun`：這表示 `let` 函式是一個內聯函式，其函式體會被內聯到每一個函式調用處。
- `<T, R>`：這是泛型參數列表。`T` 表示函式調用的對象的類型，而 `R` 表示 `block` 函式的返回類型。
- `T.let(block: (T) -> R)`：這是函式的簽名。它擁有一個 `block` 參數，這個 `block` 是一個接受 `T` 類型參數並返回 `R` 類型的函數。
- `: R`：函式的返回類型是 `R`，即 `block` 函式的返回值類型。

```
/**
 * Calls the specified function [block] with `this` value as its argument and returns its result.
 *
 * For detailed usage information see the documentation for [scope functions](https://kotlinlang.org/docs/reference/scope-functions.html#let).
 */
@kotlin.internal.InlineOnly
public inline fun <T, R> T.let(block: (T) -> R): R {
    contract {
        callsInPlace(block, InvocationKind.EXACTLY_ONCE)
    }
    return block(this)
}
```

## apply

```
val john = Person("John") .apply {
    age = 18
    birthplace = "Taipei"
}
println(john) // name:John, age:18, birthplace:Taipei
```

**inline fun <T> T.apply(block: T.() -> Unit): T** 函式定義：

- `inline fun`：這表示 `apply` 函式是一個內聯函式，其函式體會被內聯到每一個函式調用處。
- `<T>`：這是一個泛型參數，表示函式可以操作任意類型 `T` 的對象。
- `T.apply(block: T.() -> Unit)`：這是函式的簽名。它接受一個名為 `block` 的參數，這個 `block` 是一個以 `T` 類型的接收者 (receiver) 為上下文的函數型參數，沒有任何輸入，並且沒有返回值 (`Unit`)。
- `: T`：函式的返回類型是 `T`，也就是函式調用的對象本身。

```
/**
 * Calls the specified function [block] with `this` value as its receiver and returns `this` value.
 *
```

```

    * For detailed usage information see the documentation for [scope functions](https://kotlinlang.org/docs/reference/scope-functions.html#apply).
    */
    @kotlin.internal.InlineOnly
    public inline fun <T> T.apply(block: T.() -> Unit): T {
        contract {
            callsInPlace(block, InvocationKind.EXACTLY_ONCE)
        }
        block()
        return this
    }
}

```

```

/**
 * Calls the specified function [block] and returns its result.
 *
 * For detailed usage information see the documentation for [scope functions](https://kotlinlang.org/docs/reference/scope-functions.html#run).
 */
@kotlin.internal.InlineOnly
public inline fun <R> run(block: () -> R): R {
    contract {
        callsInPlace(block, InvocationKind.EXACTLY_ONCE)
    }
    return block()
}

/**
 * Calls the specified function [block] with `this` value as its receiver and returns its result.
 *
 * For detailed usage information see the documentation for [scope functions](https://kotlinlang.org/docs/reference/scope-functions.html#run).
 */
@kotlin.internal.InlineOnly
public inline fun <T, R> T.run(block: T.() -> R): R {
    contract {
        callsInPlace(block, InvocationKind.EXACTLY_ONCE)
    }
    return block()
}

/**
 * Calls the specified function [block] with the given [receiver] as its receiver and returns its result.
 *
 * For detailed usage information see the documentation for [scope functions](https://kotlinlang.org/docs/reference/scope-functions.html#with).
 */
@kotlin.internal.InlineOnly
public inline fun <T, R> with(receiver: T, block: T.() -> R): R {
    contract {
        callsInPlace(block, InvocationKind.EXACTLY_ONCE)
    }
    return receiver.block()
}

/**
 * Calls the specified function [block] with `this` value as its receiver and returns `this` value.
 *
 * For detailed usage information see the documentation for [scope functions](https://kotlinlang.org/docs/reference/scope-functions.html#apply).
 */
@kotlin.internal.InlineOnly
public inline fun <T> T.apply(block: T.() -> Unit): T {
    contract {
        callsInPlace(block, InvocationKind.EXACTLY_ONCE)
    }
}

```

```

    block()
    return this
}

/**
 * Calls the specified function [block] with `this` value as its argument and returns `this` value.
 *
 * For detailed usage information see the documentation for [scope functions](https://kotlinlang.org/docs/reference/scope-functions.html#also).
 */
@kotlin.internal.InlineOnly
@SinceKotlin("1.1")
public inline fun <T> T.also(block: (T) -> Unit): T {
    contract {
        callsInPlace(block, InvocationKind.EXACTLY_ONCE)
    }
    block(this)
    return this
}

/**
 * Calls the specified function [block] with `this` value as its argument and returns its result.
 *
 * For detailed usage information see the documentation for [scope functions](https://kotlinlang.org/docs/reference/scope-functions.html#let).
 */
@kotlin.internal.InlineOnly
public inline fun <T, R> T.let(block: (T) -> R): R {
    contract {
        callsInPlace(block, InvocationKind.EXACTLY_ONCE)
    }
    return block(this)
}

```

🔄 修訂版本 #29

★ 由 treeman 建立於 18 🕒 2024 11:12:27

✍ 由 treeman 更新於 31 🕒 2024 10:36:13