

【kotlin】Serialization CH1-CH4

出處: <https://github.com/Kotlin/kotlinx.serialization/blob/master/docs/serialization-guide.md>

CH1 基本序列化

要將物件轉換成字串或位元組序列，必須經過兩個相互交織的過程。

第一步是序列化——將物件轉換成其組成的基本值的序列。這個過程對所有資料格式來說都是

共通的，結果取決於被序列化的物件。序列化的過程由序列化器控制。

第二步稱為編碼——這是將相應的基本值序列轉換成輸出格式表示的過程。編碼器控制這個過程。當區分不那麼重要時，編碼和序列化這兩個術語可以互換使用。

+-----+ 序列化 +-----+ 編碼 +-----+ | 物件 | -----> | 基本值 | -----> | 輸出格式 | +-----+ +-----+

```
+-----+ Serialization +-----+ Encoding +-----+
| Objects | -----> | Primitives | -----> | Output format |
+-----+           +-----+           +-----+
```

反向過程從解析輸入格式並解碼基本值開始，隨後將得到的流反序列化為物件。我們稍後會詳細探討這個過程。

現在，我們從 JSON 編碼開始。

JSON 編碼

將資料轉換成特定格式的整個過程稱為編碼。對於 JSON，我們使用 `Json.encodeToString` 擴展函數來編碼資料。它在底層將傳遞給它的物件序列化並編碼為 JSON 字串。

讓我們從描述專案的一個類別開始，並嘗試獲得其 JSON 表示。

```
class Project(val name: String, val language: String)

fun main() {
    val data = Project("kotlinx.serialization", "Kotlin")
    println(Json.encodeToString(data))
}
```

執行這段程式碼時，我們會得到一個例外。

```
Exception in thread "main" kotlinx.serialization.SerializationException: Serializer for class 'Project' is not found.
Please ensure that class is marked as '@Serializable' and that the serialization compiler plugin is applied.
```

可序列化的類別必須顯式標記。Kotlin Serialization 不使用反射，因此您無法意外反序列化不應該可序列化的類別。我們通過添加 `@Serializable` 註解來修正它。

```
@Serializable
class Project(val name: String, val language: String)

fun main() {
    val data = Project("kotlinx.serialization", "Kotlin")
    println(Json.encodeToString(data))
}
```

這樣，我們將會得到相應的 JSON 輸出。

```
{"name":"kotlinx.serialization","language":"Kotlin"}
```

JSON 解碼

反向過程稱為解碼。要將 JSON 字串解碼為物件，我們將使用 `Json.decodeFromString` 擴展函數。為了指定我們想要獲得的結果類型，我們將類型參數提供給這個函數。

如我們稍後所見，序列化可以處理不同種類的類別。這裡我們將 `Project` 類別標記為 `data class`，這不是因為它是必需的，而是因為我們想要打印其內容以驗證它是如何解碼的。

```
@Serializable
data class Project(val name: String, val language: String)

fun main() {
    val data = Json.decodeFromString<Project>("""
        {"name":"kotlin.serialization","language":"Kotlin"}
        """)
    println(data)
}
```

執行這段程式碼後，我們會得到物件：

```
Project(name=kotlin.serialization, language=Kotlin)
```

可序列化的類別

這部分將更詳細地說明如何處理不同的 `@Serializable` 類別。

序列化備援屬性

只有具有備援欄位的類別屬性會被序列化，因此沒有備援欄位的 getter/setter 屬性和委託屬性將不會被序列化，如以下範例所示。

```
@Serializable
class Project(
    var name: String // name 是具有備援欄位的屬性——會被序列化
) {
    var stars: Int = 0 // 具有備援欄位的屬性——會被序列化

    val path: String // 僅有 getter，沒有備援欄位——不會被序列化
    get() = "kotlin$name"

    var id by ::name // 委託屬性——不會被序列化
}

fun main() {
    val data = Project("kotlin.serialization").apply { stars = 9000 }
    println(Json.encodeToString(data))
}
```

我們可以清楚地看到，只有 `name` 和 `stars` 屬性出現在 JSON 輸出中。

```
{"name":"kotlin.serialization","stars":9000}
```

建構子屬性要求

如果我們想要定義一個 `Project` 類別，並讓它接受一個路徑字串，然後將其分解為相應的屬性，我們可能會傾向於寫出如下代碼。

```
@Serializable
class Project(path: String) {
    val owner: String = path.substringBefore('/')
    val name: String = path.substringAfter('/')
}
```

這個類別無法編譯，因為 `@Serializable` 註解要求類別主要建構子的所有參數必須是屬性。一個簡單的解決方案是定義一個具有屬性的私有主要建構子，並將我們想要的建構子變成次要建構子。

```
@Serializable
class Project private constructor(val owner: String, val name: String) {
    constructor(path: String) : this(
        owner = path.substringBefore('/'),
        name = path.substringAfter('/')
    )
}
```

```

    )

    val path: String
        get() = "$owner/$name"
}

```

序列化在具有私有主要建構子的情況下仍然能夠正常工作，並且仍然只會序列化備援欄位。

```

fun main() {
    println(Json.encodeToString(Project("kotlin/kotlinx.serialization")))
}

```

這個範例會產生預期的輸出。

```

{"owner":"kotlin","name":"kotlinx.serialization"}

```

資料驗證

當你希望在將值存儲到屬性之前進行驗證時，可能會想在主要建構子參數中引入一個沒有屬性的參數。為了使其可序列化，你應該將其替換為主要建構子中的屬性，並將驗證移至 `init { ... }` 區塊。

```

@Serializable
class Project(val name: String) {
    init {
        require(name.isNotEmpty()) { "name 不能為空" }
    }
}

```

反序列化過程與 Kotlin 中的常規建構子一樣，會調用所有的 `init` 區塊，確保你無法通過反序列化獲得一個無效的類別。讓我們試試看。

```

fun main() {
    val data = Json.decodeFromString<Project>("""
        {"name":""}
    """)
    println(data)
}

```

執行這段程式碼會產生一個例外：

```

Exception in thread "main" java.lang.IllegalArgumentException: name 不能為空

```

選擇性屬性

只有當輸入中存在所有屬性時，物件才能被反序列化。例如，執行以下程式碼。

```

@Serializable
data class Project(val name: String, val language: String)

fun main() {
    val data = Json.decodeFromString<Project>("""
        {"name":"kotlinx.serialization"}
    """)
    println(data)
}

```

這會產生一個例外：

```

Exception in thread "main" kotlinx.serialization.MissingFieldException: Field 'language' is required for type with serial name 'example.exampleClasses04.Project', but it was missing at path: $

```

這個問題可以通過為屬性添加一個默認值來解決，這會自動使其在序列化中成為選擇性屬性。

```

@Serializable
data class Project(val name: String, val language: String = "Kotlin")

fun main() {

```

```
val data = Json.decodeFromString<Project>("""
    {"name":"kotlinx.serialization"}
    """)
println(data)
}
```

這會產生以下輸出，其中 `language` 屬性使用默認值。

```
Project(name=kotlinx.serialization, language=Kotlin)
```

選擇性屬性初始化呼叫

當選擇性屬性存在於輸入中時，對應屬性的初始化器

甚至不會被調用。這是一個為了性能而設計的特性，所以要小心不要依賴於初始化器中的副作用。考慮以下範例。

```
fun computeLanguage(): String {
    println("計算中")
    return "Kotlin"
}

@Serializable
data class Project(val name: String, val language: String = computeLanguage())

fun main() {
    val data = Json.decodeFromString<Project>("""
        {"name":"kotlinx.serialization","language":"Kotlin"}
        """)
    println(data)
}
```

由於輸入中指定了 `language` 屬性，因此我們不會看到 "計算中" 的字串被打印出來。

```
Project(name=kotlinx.serialization, language=Kotlin)
```

必須的屬性

具有默認值的屬性可以通過 `@Required` 註解在序列格式中設置為必須的。我們將上一個範例的 `language` 屬性標記為 `@Required`。

```
@Serializable
data class Project(val name: String, @Required val language: String = "Kotlin")

fun main() {
    val data = Json.decodeFromString<Project>("""
        {"name":"kotlinx.serialization"}
        """)
    println(data)
}
```

我們會得到以下例外。

```
Exception in thread "main" kotlinx.serialization.MissingFieldException: Field 'language' is required for type with serial name
'example.exampleClasses07.Project', but it was missing at path: $
```

暫態屬性

屬性可以通過 `@Transient` 註解排除在序列化之外（不要將其與 `kotlin.jvm.Transient` 混淆）。暫態屬性必須有一個默認值。

```
@Serializable
data class Project(val name: String, @Transient val language: String = "Kotlin")

fun main() {
    val data = Json.decodeFromString<Project>("""
        {"name":"kotlinx.serialization","language":"Kotlin"}
        """)
    println(data)
}
```

```
}
```

即使在序列格式中明確指定了它的值，即使指定的值與默認值相等，也會產生以下例外。

```
Exception in thread "main" kotlinx.serialization.json.internal.JsonDecodingException: Unexpected JSON token at offset 42: Encountered an unknown key 'language' at path: $.name
Use 'ignoreUnknownKeys = true' in 'Json {}' builder to ignore unknown keys.
```

忽略未知鍵功能在 "忽略未知鍵" 一節中解釋。

默認值不會被編碼

默認情況下，默認值不會在 JSON 中編碼。這種行為的動機是，在大多數現實生活場景中，這種配置減少了視覺雜訊，並節省了序列化的資料量。

```
@Serializable
data class Project(val name: String, val language: String = "Kotlin")

fun main() {
    val data = Project("kotlinx.serialization")
    println(Json.encodeToString(data))
}
```

這會產生以下輸出，因為 `language` 屬性的值等於默認值，所以它不會出現在 JSON 中。

```
{"name":"kotlinx.serialization"}
```

查看 JSON 的 "編碼默認值" 一節了解如何配置此行為。此外，這種行為可以在不考慮格式設置的情況下進行控制。為此，可以使用 `EncodeDefault` 註解：

```
@Serializable
data class Project(
    val name: String,
    @EncodeDefault val language: String = "Kotlin"
)
```

這個註解指示框架無論值或格式設置如何，都要始終序列化屬性。還可以使用 `EncodeDefault.Mode` 參數將其調整為相反的行為：

```
@Serializable
data class User(
    val name: String,
    @EncodeDefault(EncodeDefault.Mode.NEVER) val projects: List<Project> = emptyList()
)

fun main() {
    val userA = User("Alice", listOf(Project("kotlinx.serialization")))
    val userB = User("Bob")
    println(Json.encodeToString(userA))
    println(Json.encodeToString(userB))
}
```

如您所見，`language` 屬性被保留，而 `projects` 被忽略：

```
{"name":"Alice","projects":[{"name":"kotlinx.serialization","language":"Kotlin"}]}
{"name":"Bob"}
```

可空屬性

Kotlin Serialization 原生支持可空屬性。

```
@Serializable
class Project(val name: String, val renamedTo: String? = null)

fun main() {
    val data = Project("kotlinx.serialization")
}
```

```
println(Json.encodeToString(data))
}
```

由於默認值不會被編碼，因此此範例不會在 JSON 中編碼 null。

```
{"name":"kotlinx.serialization"}
```

強制型別安全

Kotlin Serialization 強制執行 Kotlin 編程語言的型別安全。特別是，讓我們嘗試將 JSON 對象中的 null 值解碼為非空的 Kotlin 屬性 `language`。

```
@Serializable
data class Project(val name: String, val language: String = "Kotlin")

fun main() {
    val data = Json.decodeFromString<Project>("""
        {"name":"kotlinx.serialization","language":null}
        """)
    println(data)
}
```

即使 `language` 屬性具有默認值，但將 null 值分配給它仍然是一個錯誤。

```
Exception in thread "main" kotlinx.serialization.json.internal.JsonDecodingException: Unexpected JSON token at offset 52: Expected
string literal but 'null' literal was found at path: $.language
Use 'coerceInputValues = true' in 'Json {}' builder to coerce nulls if property has a default value.
```

在解碼第三方 JSON 時，可能希望將 null 強制為默認值。相應的功能在 "強制輸入值" 一節中解釋。

引用物件

可序列化的類別可以在其可序列化屬性中引用其他類別。引用的類別也必須標記為 `@Serializable`。

```
@Serializable
class Project(val name: String, val owner: User)

@Serializable
class User(val name: String)

fun main() {
    val owner = User("kotlin")
    val data = Project("kotlinx.serialization", owner)
    println(Json.encodeToString(data))
}
```

當編碼為 JSON 時，它會生成一個嵌套的 JSON 物件。

```
{"name":"kotlinx.serialization","owner":{"name":"kotlin"}}
```

對於不可序列化的類別引用，可以將其標記為暫態屬性，或者為它們提供自定義序列化器，如 "序列化器" 一章所示。

不壓縮重複引用

Kotlin Serialization 專為編碼和解碼純資料而設計。它不支持重建具有重複物件引用的任意物件圖。例如，讓我們嘗試序列化一個兩次引用同一 `owner` 實例的物件。

```
@Serializable
class Project(val name: String, val owner: User, val maintainer: User)

@Serializable
class User(val name: String)

fun main() {
```

```
val owner = User("kotlin")
val data = Project("kotlinx.serialization", owner, owner)
println(Json.encodeToString(data))
}
```

我們會得到兩次編碼的 `owner` 值。

```
{"name":"kotlinx.serialization","owner":{"name":"kotlin"},"maintainer":{"name":"kotlin"}}
```

嘗試序列化循環結構會導致堆疊溢出。你可以使用暫態屬性來排除某些引用的序列化。

泛型類別

Kotlin 中的泛型類別提供了型別多態行為，這在編譯期間由 Kotlin Serialization 強制執行。例如，考慮一個泛型可序列化類別 `Box<T>`。

```
@Serializable
class Box<T>(val contents: T)
```

`Box<T>` 類別可以與內建型別如 `Int` 一起使用，也可以與使用者定義的型別如 `Project` 一起使用。

```
@Serializable
class Data(
    val a: Box<Int>,
    val b: Box<Project>
)

fun main() {
    val data = Data(Box(42), Box(Project("kotlinx.serialization", "Kotlin")))
    println(Json.encodeToString(data))
}
```

我們在 JSON 中獲得的實際型別取決於為 `Box` 指定的實際編譯時型別參數。

```
{"a":{"contents":42},"b":{"contents":{"name":"kotlinx.serialization","language":"K
otlin"}}
```

如果實際泛型類型不可序列化，則會產生編譯時錯誤。

序列欄位名稱

在編碼表示中使用的屬性名稱（例如我們的 JSON 示例中的屬性名稱）默認情況下與其在源代碼中的名稱相同。用於序列化的名稱稱為序列名稱，可以使用 `@SerializedName` 註解進行更改。例如，我們可以在源代碼中有一個 `language` 屬性，並將其序列名稱縮寫。

```
@Serializable
class Project(val name: String, @SerializedName("lang") val language: String)

fun main() {
    val data = Project("kotlinx.serialization", "Kotlin")
    println(Json.encodeToString(data))
}
```

現在我們看到在 JSON 輸出中使用了縮寫名稱 `lang`。

```
{"name":"kotlinx.serialization","lang":"Kotlin"}
```

下一章將介紹內建類別。

CH2 標準內建類別

這是《Kotlin 序列化指南》的第二章。除了所有的基本類型和字串外，Kotlin 標準函式庫中的一些類別（包括標準集合）的序列化功能也內建在 Kotlin 序列化中。本章將解釋這些功能的詳細內容。

目錄

- 基本類型
- 數字
- 長整數
- 作為字串的長整數
- 列舉類別
- 列舉項目的序列名稱
- 複合類型
- Pair 和 Triple
- 列表
- 集合和其他集合類型
- 反序列化集合
- 映射 (Map)
- 單例物件 (Unit) 和單例物件類別
- 持續時間 (Duration)
- Nothing

基本類型

Kotlin 序列化支援以下十種基本類型：Boolean、Byte、Short、Int、Long、Float、Double、Char、String 以及列舉類型。Kotlin 序列化中的其他類型都是由這些基本值組成的複合類型。

數字

所有類型的整數和浮點數 Kotlin 數字都可以序列化。

```
@Serializable
class Data(
    val answer: Int,
    val pi: Double
)

fun main() {
    val data = Data(42, PI)
    println(Json.encodeToString(data))
}
```

這些數字在 JSON 中會被表示為自然的形式。

```
{"answer":42,"pi":3.141592653589793}
```

長整數

長整數 (Long) 也可以被序列化。

```
@Serializable
class Data(val signature: Long)

fun main() {
    val data = Data(0x1CAFE2FEED0BABE0)
    println(Json.encodeToString(data))
}
```

預設情況下，它們會被序列化為 JSON 數字。

```
{"signature":2067120338512882656}
```

作為字串的長整數

上面的 JSON 輸出會被 Kotlin/JS 上運行的 Kotlin 序列化正常解碼。然而，如果我們嘗試使用原生的 JavaScript 方法解析這個 JSON，會得到這樣的截斷結果。


```
JSON.parse("{\"signature\":\"2067120338512882656\"}")
// ► {signature: 2067120338512882700}
```

Kotlin 的 `Long` 類型的完整範圍無法適配 JavaScript 的數字，因此在 JavaScript 中會丟失精度。常見的解決方案是使用 JSON 字串類型來表示具有完整精度的長整數。Kotlin 序列化通過 `LongAsStringSerializer` 選擇性地支持這種方法，可以使用 `@Serializable` 註解將其指定給特定的 `Long` 屬性：

```
@Serializable
class Data(
    @Serializable(with=LongAsStringSerializer::class)
    val signature: Long
)

fun main() {
    val data = Data(0x1CAFE2FEED0BABE0)
    println(Json.encodeToString(data))
}
```

這個 JSON 可以被 JavaScript 原生解析而不丟失精度。

```
{"signature":"2067120338512882656"}
```

關於如何為整個檔案中的所有屬性指定類似 `LongAsStringSerializer` 的序列化器，可以參考 "為檔案指定序列化器" 這一節。

列舉類別

所有的列舉類別都是可序列化的，無需將它們標記為 `@Serializable`，如下例所示。

```
// 列舉類別不需要 @Serializable 註解
enum class Status { SUPPORTED }

@Serializable
class Project(val name: String, val status: Status)

fun main() {
    val data = Project("kotlinx.serialization", Status.SUPPORTED)
    println(Json.encodeToString(data))
}
```

在 JSON 中，列舉會被編碼為字串。

```
{"name":"kotlinx.serialization","status":"SUPPORTED"}
```

注意：在 Kotlin/JS 和 Kotlin/Native 上，如果你想將列舉類別作為根物件使用（即使用 `encodeToString<Status>` (`Status.SUPPORTED`)），需要為列舉類別添加 `@Serializable` 註解。

列舉項目的序列名稱

列舉項目的序列名稱可以像 "序列欄位名稱" 一節所示一樣，通過 `SerialName` 註解來自訂。然而，在這種情況下，整個列舉類別必須標記為 `@Serializable`。

```
@Serializable // 因為使用了 @SerialName，所以需要此註解
enum class Status { @SerialName("maintained") SUPPORTED }

@Serializable
class Project(val name: String, val status: Status)

fun main() {
    val data = Project("kotlinx.serialization", Status.SUPPORTED)
    println(Json.encodeToString(data))
}
```

我們可以看到，在結果 JSON 中使用了指定的序列名稱。

```
{"name":"kotlinx.serialization","status":"maintained"}
```

複合類型

Kotlin 序列化支援標準函式庫中的一些複合類型。

Pair 和 Triple

Kotlin 標準函式庫中的簡單資料類別 `Pair` 和 `Triple` 是可序列化的。

```
@Serializable
class Project(val name: String)

fun main() {
    val pair = 1 to Project("kotlinx.serialization")
    println(Json.encodeToString(pair))
}
```

```
{"first":1,"second":{"name":"kotlinx.serialization"}}
```

並不是所有的 Kotlin 標準函式庫中的類別都是可序列化的，特別是範圍 (`Range`) 和正則表達式 (`Regex`) 類別目前不可序列化。未來可能會添加它們的序列化支持。

列表

可以序列化一個可序列化類別的列表。

```
@Serializable
class Project(val name: String)

fun main() {
    val list = listOf(
        Project("kotlinx.serialization"),
        Project("kotlinx.coroutines")
    )
    println(Json.encodeToString(list))
}
```

結果在 JSON 中表示為列表。

```
[{"name":"kotlinx.serialization"}, {"name":"kotlinx.coroutines"}]
```

集合和其他集合類型

其他集合類型，如集合 (`Set`)，也可以被序列化。

```
@Serializable
class Project(val name: String)

fun main() {
    val set = setOf(
        Project("kotlinx.serialization"),
        Project("kotlinx.coroutines")
    )
    println(Json.encodeToString(set))
}
```

`Set` 也會像其他所有集合一樣，在 JSON 中表示為列表。

```
[{"name":"kotlinx.serialization"}, {"name":"kotlinx.coroutines"}]
```

反序列化集合

在反序列化過程中，結果物件的類型由源代碼中指定的靜態類型決定——無論是屬性的類型還是解碼函數的類型參數。以下範例展示了相同的 JSON 整數列表如何被反序列化為兩個不同 Kotlin 類型的屬性。

```
@Serializable
data class Data(
    val a: List<Int>,

```

```

    val b: Set<Int>
)

fun main() {
    val data = Json.decodeFromString<Data>("""
        {
            "a": [42, 42],
            "b": [42, 42]
        }
        """)
    println(data)
}

```

由於 `data.b` 屬性是 `Set`，因此其中的重複值消失了。

```
Data(a=[42, 42], b=[42])
```

映射 (`Map`)

具有基本類型或列舉鍵和任意可序列化值的 `Map` 可以被序列化。

```

@Serializable
class Project(val name: String)

fun main() {
    val map = mapOf(
        1 to Project("kotlinx.serialization"),
        2 to Project("kotlinx.coroutines")
    )
    println(Json.encodeToString(map))
}

```

Kotlin 的映射在 JSON 中表示為物件。在 JSON 中，物件的鍵總是字串，所以即使在 Kotlin 中它們是數字，也會被編碼為字串，如下所示。

```
{ "1": { "name": "kotlinx.serialization" }, "2": { "name": "kotlinx.coroutines" } }
```

這是 JSON 的

一個特定限制，即鍵不能是複合的。可以通過參考 "允許結構化的映射鍵" 這一節來解決這個限制。

單例物件 (`Unit`) 和單例物件類別

Kotlin 內建的 `Unit` 類型也是可序列化的。`Unit` 是 Kotlin 的一個單例物件，並且與其他 Kotlin 物件一樣處理。

從概念上講，單例物件是一個只有一個實例的類別，這意味著狀態不定義物件，而是物件定義其狀態。在 JSON 中，物件會被序列化為空結構。

```

@Serializable
object SerializationVersion {
    val libraryVersion: String = "1.0.0"
}

fun main() {
    println(Json.encodeToString(SerializationVersion))
    println(Json.encodeToString(Unit))
}

```

雖然這在表面上看似無用，但在密封類別序列化中這很有用，這在 "多型性" 章節的 "物件" 部分中進行了解釋。

```

{}
{}

```

物件的序列化是格式特定的。其他格式可能會以不同方式表示物件，例如使用它們的全名。

持續時間 (Duration)

自 Kotlin 1.7.20 以來，`Duration` 類別已經成為可序列化的。

```
fun main() {  
    val duration = 1000.toDuration(DurationUnit.SECONDS)  
    println(Json.encodeToString(duration))  
}
```

`Duration` 會被序列化為 ISO-8601-2 格式的字串。

```
"PT16M40S"
```

Nothing

預設情況下，`Nothing` 是一個可序列化的類別。然而，由於該類別沒有實例，因此不可能對其值進行編碼或解碼——任何嘗試都會導致異常。

當語法上需要某種類型，但實際上在序列化中不使用時，會使用這個序列化器。例如，在使用參數化多型基類時：

```
@Serializable  
sealed class ParametrizedParent<out R> {  
    @Serializable  
    data class ChildWithoutParameter(val value: Int) : ParametrizedParent<Nothing>()  
}  
  
fun main() {  
    println(Json.encodeToString(ParametrizedParent.ChildWithoutParameter(42)))  
}
```

在編碼過程中，`Nothing` 的序列化器未被使用。

```
{"value":42}
```

下一章將介紹序列化器 (`Serializers`)。

CH3 序列化器

這是《Kotlin 序列化指南》的第三章。本章將更詳細地介紹序列化器，並展示如何編寫自定義序列化器。

目錄

- 序列化器介紹
- 插件生成的序列化器
- 插件生成的泛型序列化器
- 內建的基本序列化器
- 構建集合序列化器
- 使用頂級序列化器函數
- 自定義序列化器
 - 基本序列化器
 - 委派序列化器
 - 通過代理實現複合序列化器
 - 手寫的複合序列化器
 - 順序解碼協議（實驗性）
- 序列化第三方類別
 - 手動傳遞序列化器
 - 為屬性指定序列化器
 - 為特定類型指定序列化器
 - 為檔案指定序列化器
 - 使用型別別名全局指定序列化器
- 泛型類型的自定義序列化器
- 特定格式的序列化器
- 上下文序列化
- 序列化器模組
- 上下文序列化與泛型類別

- 為其他 Kotlin 類別衍生外部序列化器（實驗性）
- 外部序列化使用屬性

序列化器介紹

像 JSON 這樣的格式控制著物件編碼成特定的輸出位元組，但如何將物件分解為其組成屬性則由序列化器控制。到目前為止，我們已經使用了通過 `@Serializable` 註解自動生成的序列化器，如 "可序列化的類別" 章節中所解釋的，或使用了在 "內建類別" 章節中展示的內建序列化器。

作為一個激勵性的例子，讓我們來看一下下面這個 `Color` 類別，它使用一個整數值來存儲其 RGB 位元組。

```
@Serializable
class Color(val rgb: Int)

fun main() {
    val green = Color(0x00ff00)
    println(Json.encodeToString(green))
}
```

預設情況下，這個類別會將其 `rgb` 屬性序列化為 JSON。

```
{"rgb":65280}
```

插件生成的序列化器

每個標記有 `@Serializable` 註解的類別（如上一個例子中的 `Color` 類別）都會由 Kotlin 序列化編譯器插件自動生成一個 `KSerializer` 介面的實例。我們可以使用類別的伴生對象上的 `.serializer()` 函數來檢索此實例。

我們可以檢查其 `descriptor` 屬性，它描述了序列化類別的結構。我們會在後續章節中詳細了解這一點。

```
fun main() {
    val colorSerializer: KSerializer<Color> = Color.serializer()
    println(colorSerializer.descriptor)
}
```

輸出：

```
Color(rgb: kotlin.Int)
```

當 `Color` 類別本身被序列化時，或當它被用作其他類別的屬性時，Kotlin 序列化框架會自動檢索並使用此序列化器。

你無法在可序列化類別的伴生對象上自定義自己的 `serializer()` 函數。

插件生成的泛型序列化器

對於泛型類別，如 "泛型類別" 章節中展示的 `Box` 類別，自動生成的 `.serializer()` 函數接受的參數數量與對應類別中的類型參數數量相同。這些參數的類型是 `KSerializer`，因此在構建泛型類別的序列化器實例時，必須提供實際類型參數的序列化器。

```
@Serializable
@SerialName("Box")
class Box<T>(val contents: T)

fun main() {
    val boxedColorSerializer = Box.serializer(Color.serializer())
    println(boxedColorSerializer.descriptor)
}
```

我們可以看到，已經實例化了一個序列化器來序列化具體的 `Box<Color>`。

```
Box(contents: Color)
```

內建的基本序列化器

內建類別的基本序列化器可以通過 `.serializer()` 擴展函數來檢索。

```
fun main() {
    val intSerializer: KSerializer<Int> = Int.serializer()
    println(intSerializer.descriptor)
}
```

構建集合序列化器

當需要時，內建集合的序列化器必須通過對應的函數如 `ListSerializer()`、`SetSerializer()`、`MapSerializer()` 等顯式構建。這些類別是泛型的，因此要實例化它們的序列化器，我們必須為其類型參數提供相應的序列化器。例如，我們可以如下生成一個 `List<String>` 的序列化器。

```
fun main() {
    val stringListSerializer: KSerializer<List<String>> = ListSerializer(String.serializer())
    println(stringListSerializer.descriptor)
}
```

使用頂級序列化器函數

當不確定時，你可以隨時使用頂級泛型 `serializer<T>()` 函數來檢索源代碼中任意 Kotlin 類型的序列化器。

```
@Serializable
@SerialName("Color")
class Color(val rgb: Int)

fun main() {
    val stringToColorMapSerializer: KSerializer<Map<String, Color>> = serializer()
    println(stringToColorMapSerializer.descriptor)
}
```

自定義序列化器

插件生成的序列化器非常方便，但對於像 `Color` 這樣的類別，它可能無法生成我們想要的 JSON。讓我們來研究一些替代方案。

基本序列化器

我們想將 `Color` 類別序列化為一個十六進制字串，其中綠色將被表示為 `"00ff00"`。為了實現這一點，我們需要編寫一個實現 `KSerializer` 介面的物件來處理 `Color` 類別。

```
object ColorAsStringSerializer : KSerializer<Color> {
    override val descriptor: SerialDescriptor = PrimitiveSerialDescriptor("Color", PrimitiveKind.STRING)

    override fun serialize(encoder: Encoder, value: Color) {
        val string = value.rgb.toString(16).padStart(6, '0')
        encoder.encodeString(string)
    }

    override fun deserialize(decoder: Decoder): Color {
        val string = decoder.decodeString()
        return Color(string.toInt(16))
    }
}
```

序列化器包含三個必須的部分：

1. `serialize` 函數實現了 `SerializationStrategy`。它接收一個 `Encoder` 的實例和一個要序列化的值。它使用 `Encoder` 的 `encodeXxx` 函數來將值表示為一系列基本類型。在我們的例子中，使用了 `encodeString`。
2. `deserialize` 函數實現了 `DeserializationStrategy`。它接收一個 `Decoder` 的實例並返回一個反序列化的值。它使用 `Decoder` 的 `decodeXxx` 函數來解碼對應的值。在我們的例子中，使用了 `decodeString`。
3. `descriptor` 屬性必須準確描述 `encodeXxx` 和 `decodeXxx` 函數的作用，以便格式實現能夠提前知道它們將調用哪些編碼/解碼方法。對於基本序列化，必須使用 `PrimitiveSerialDescriptor` 函數並為正在序列化的類型提供唯一的名稱。`PrimitiveKind` 描述了實現中使用的特定 `encodeXxx/decodeXxx` 方法。

當 `descriptor` 與編碼/解碼方法不對應時，結果代碼的行為是未定義的，可能會在未來的更新中隨機更改。

下一步是將序列化器綁定到類別。這可以通過在 `@Serializable` 註解中添加 `with` 屬性來實現。

```
@Serializable(with = ColorAsStringSerializer::class)
class Color(val rgb: Int)
```

現在我們可以像以前一樣序列化 `Color` 類別了。

```
fun main() {
    val green = Color(0x00ff00)
    println(Json.encodeToString(green))
}
```

我們得到了我們想要的十六進制字串

的序列表示。

```
"00ff00"
```

反序列化也很簡單，因為我們實現了 `deserialize` 方法。

```
@Serializable(with = ColorAsStringSerializer::class)
class Color(val rgb: Int)

fun main() {
    val color = Json.decodeFromString<Color>("{\"00ff00\"}")
    println(color.rgb) // prints 65280
}
```

它也適用於我們序列化或反序列化具有 `Color` 屬性的不同類別。

```
@Serializable(with = ColorAsStringSerializer::class)
data class Color(val rgb: Int)

@Serializable
data class Settings(val background: Color, val foreground: Color)

fun main() {
    val data = Settings(Color(0xffffffff), Color(0))
    val string = Json.encodeToString(data)
    println(string)
    require(Json.decodeFromString<Settings>(string) == data)
}
```

兩個 `Color` 屬性都被序列化為字串。

```
{"background":"ffffff","foreground":"000000"}
```

委派序列化器

在前面的例子中，我們將 `Color` 類別表示為字串。字串被認為是一種基本類型，因此我們使用了 `PrimitiveClassDescriptor` 和專門的 `encodeString` 方法。現在讓我們看看如果我們需要將 `Color` 序列化為另一種非基本類型（例如 `IntArray`），我們的操作會是什麼。

`KSerializer` 的實現將在 `Color` 和 `IntArray` 之間進行轉換，但實際的序列化邏輯將委派給 `IntArraySerializer`，使用 `encodeSerializableValue` 和 `decodeSerializableValue`。

```
import kotlinx.serialization.builtins.IntArraySerializer

class ColorIntArraySerializer : KSerializer<Color> {
    private val delegateSerializer = IntArraySerializer()
    override val descriptor = SerialDescriptor("Color", delegateSerializer.descriptor)

    override fun serialize(encoder: Encoder, value: Color) {
        val data = intArrayOf(
            (value.rgb shr 16) and 0xFF,
            (value.rgb shr 8) and 0xFF,
            value.rgb and 0xFF
        )
        encoder.encodeSerializableValue(delegateSerializer, data)
    }
}
```

```

override fun deserialize(decoder: Decoder): Color {
    val array = decoder.decodeSerializableValue(delegateSerializer)
    return Color((array[0] shl 16) or (array[1] shl 8) or array[2])
}
}

```

注意，這裡我們不能使用預設的 `Color.serializer().descriptor`，因為依賴於架構的格式可能會認為我們會調用 `encodeInt` 而不是 `encodeSerializableValue`。同樣，我們也不能直接使用 `IntArraySerializer().descriptor`，否則處理整數數組的格式將無法區分它是 `IntArray` 還是 `Color`。不用擔心，當序列化實際的底層整數數組時，這個優化仍然會起作用。

我們現在可以使用這個序列化器：

```

@Serializable(with = ColorIntArraySerializer::class)
class Color(val rgb: Int)

fun main() {
    val green = Color(0x00ff00)
    println(Json.encodeToString(green))
}

```

正如您所見，這種數組表示在 JSON 中不是很有用，但在與 `ByteArray` 和二進制格式一起使用時可能會節省一些空間。

```
[0,255,0]
```

通過代理實現複合序列化器

現在我們的挑戰是讓 `Color` 被序列化，這樣它在 JSON 中被表示為一個具有三個屬性（`r`、`g` 和 `b`）的類別，從而讓 JSON 將其編碼為物件。實現這一目標的最簡單方法是定義一個模擬 `Color` 序列化形式的代理類別，然後將其用作 `Color` 的序列化器。我們還可以將這個代理類別的 `SerialName` 設置為 `Color`。然後，如果任何格式使用此名稱，代理看起來就像是一個 `Color` 類別。代理類別可以是私有的，並且可以在其 `init` 區塊中強制執行所有關於類別序列表示的約束。

```

@Serializable
@SerialName("Color")
private class ColorSurrogate(val r: Int, val g: Int, val b: Int) {
    init {
        require(r in 0..255 && g in 0..255 && b in 0..255)
    }
}

```

現在我們可以使用 `ColorSurrogate.serializer()` 函數來檢索為代理類別自動生成的序列化器。

我們可以像在委派序列化器中那樣使用相同的方法，但這次，我們完全重用了為代理類別自動生成的 `SerialDescriptor`，因為它應該與原始類別無法區分。

```

object ColorSerializer : KSerializer<Color> {
    override val descriptor: SerialDescriptor = ColorSurrogate.serializer().descriptor

    override fun serialize(encoder: Encoder, value: Color) {
        val surrogate = ColorSurrogate((value.rgb shr 16) and 0xff, (value.rgb shr 8) and 0xff, value.rgb and 0xff)
        encoder.encodeSerializableValue(ColorSurrogate.serializer(), surrogate)
    }

    override fun deserialize(decoder: Decoder): Color {
        val surrogate = decoder.decodeSerializableValue(ColorSurrogate.serializer())
        return Color((surrogate.r shl 16) or (surrogate.g shl 8) or surrogate.b)
    }
}

```

我們將 `ColorSerializer` 序列化器綁定到 `Color` 類別。

```

@Serializable(with = ColorSerializer::class)
class Color(val rgb: Int)

```

現在我們可以享受 `Color` 類別序列化的結果了。

```
{"r":0,"g":255,"b":0}
```

手寫的複合序列化器

有些情況下，代理解決方案並不適用。可能我們想避免額外分配的性能影響，或者我們希望為最終的序列化表示提供一組可配置/動態屬性。在這些情況下，我們需要手動編寫一個模仿生成的序列化器行為的類別序列化器。

```
object ColorAsObjectSerializer : KSerializer<Color> {
```

讓我們逐步介紹它。首先，使用 `buildClassSerialDescriptor` 構建器定義一個 `descriptor`。在構建器的 DSL 中，`element` 函數會根據其類型自動檢索對應字段的序列化器。元素的順序很重要。它們從零開始編號。

```
override val descriptor: SerialDescriptor =
    buildClassSerialDescriptor("Color") {
        element<Int>("r")
        element<Int>("g")
        element<Int>("b")
    }
```

這裡的 "element" 是一個通用術語。`descriptor` 的元素取決於其 `SerialKind`。類別描述符的元素是其屬性，枚舉描述符的元素是其案例，等等。

接著，我們使用 `encodeStructure` DSL 編寫 `serialize` 函數，它在其區塊中提供對 `CompositeEncoder` 的訪問。`Encoder` 和 `CompositeEncoder` 之間的區別在於後者具有對應於前者的 `encodeXxx` 函數的 `encodeXxxElement` 函數。它們必須按與 `descriptor` 中相同的順序調用。

```
override fun serialize(encoder: Encoder, value: Color) =
    encoder.encodeStructure(descriptor) {
        encodeIntElement(descriptor, 0, (value.rgb shr 16) and 0xff)
        encodeIntElement(descriptor, 1, (value.rgb shr 8) and 0xff)
        encodeIntElement(descriptor, 2, value.rgb and 0xff)
    }
```

最複雜的部分是 `deserialize` 函數。它必須支持像 JSON 這樣可以以任意順序解碼屬性的格式。它以調用 `decodeStructure` 開始，從而獲得 `CompositeDecoder` 的訪問權限。在內部，我們編寫了一個循環，該循環反覆調用 `decodeElementIndex` 來解碼下一個元素的索引，然後使用我們的示例中的 `decodeIntElement` 解碼相應的元素，最後當遇到 `CompositeDecoder.DECODE_DONE` 時終止循環。

```
override fun deserialize(decoder: Decoder): Color =
    decoder.decodeStructure(descriptor) {
        var r = -1
        var g = -1
        var b =

-1
        while (true) {
            when (val index = decodeElementIndex(descriptor)) {
                0 -> r = decodeIntElement(descriptor, 0)
                1 -> g = decodeIntElement(descriptor, 1)
                2 -> b = decodeIntElement(descriptor, 2)
                CompositeDecoder.DECODE_DONE -> break
                else -> error("Unexpected index: $index")
            }
        }
        require(r in 0..255 && g in 0..255 && b in 0..255)
        Color((r shl 16) or (g shl 8) or b)
    }
```

現在，我們將最終的序列化器綁定到 `Color` 類別並測試其序列化/反序列化。

```
@Serializable(with = ColorAsObjectSerializer::class)
data class Color(val rgb: Int)

fun main() {
    val color = Color(0x00ff00)
    val string = Json.encodeToString(color)
    println(string)
    require(Json.decodeFromString<Color>(string) == color)
}
```

與之前一樣，我們得到了以具有三個鍵的 JSON 物件表示的 `Color` 類別：

```
{"r":0,"g":255,"b":0}
```

順序解碼協議（實驗性）

前一部分中的 `deserialize` 函數的實現適用於任何格式。然而，有些格式無論何時都會按順序存儲所有複雜數據，或者有時這樣做（例如，JSON 始終按順序存儲集合）。對於這些格式來說，在循環中調用 `decodeElementIndex` 的複雜協議並不是必需的，如果 `CompositeDecoder.decodeSequentially` 函數返回 `true`，則可以使用更快的實現。插件生成的序列化器實際上在概念上類似於以下代碼。

```
override fun deserialize(decoder: Decoder): Color =
    decoder.decodeStructure(descriptor) {
        var r = -1
        var g = -1
        var b = -1
        if (decodeSequentially()) { // sequential decoding protocol
            r = decodeIntElement(descriptor, 0)
            g = decodeIntElement(descriptor, 1)
            b = decodeIntElement(descriptor, 2)
        } else while (true) {
            when (val index = decodeElementIndex(descriptor)) {
                0 -> r = decodeIntElement(descriptor, 0)
                1 -> g = decodeIntElement(descriptor, 1)
                2 -> b = decodeIntElement(descriptor, 2)
                CompositeDecoder.DECODE_DONE -> break
                else -> error("Unexpected index: $index")
            }
        }
        require(r in 0..255 && g in 0..255 && b in 0..255)
        Color((r shl 16) or (g shl 8) or b)
    }
```

序列化第三方類別

有時應用程式必須處理一個不可序列化的外部類型。我們以 `java.util.Date` 為例。如前所述，我們從為該類別編寫 `KSerializer` 的實現開始。我們的目標是將 `Date` 序列化為一個長整數，表示自 Unix 紀元以來的毫秒數，這與 "基本序列化器" 章節中的方法類似。

在接下來的章節中，任何類型的 `Date` 序列化器都可以工作。例如，如果我們希望將 `Date` 序列化為一個物件，我們可以使用 "通過代理實現複合序列化器" 章節中的方法。如果您需要序列化一個本應可序列化但實際不可序列化的第三方 Kotlin 類別，請參閱 "為另一個 Kotlin 類別衍生外部序列化器（實驗性）"。

```
object DateAsLongSerializer : KSerializer<Date> {
    override val descriptor: SerialDescriptor = PrimitiveSerialDescriptor("Date", PrimitiveKind.LONG)
    override fun serialize(encoder: Encoder, value: Date) = encoder.encodeLong(value.time)
    override fun deserialize(decoder: Decoder): Date = Date(decoder.decodeLong())
}
```

我們無法通過 `@Serializable` 註解將 `DateAsLongSerializer` 序列化器綁定到 `Date` 類別，因為我們無法控制 `Date` 的源代碼。這裡有幾種解決方法。

手動傳遞序列化器

所有的 `encodeToXxx` 和 `decodeFromXxx` 函數都有一個帶有第一個序列化器參數的重載版本。當一個不可序列化的類別（如 `Date`）是正在序列化的頂級類別時，我們可以使用這些重載版本。

```
fun main() {
    val kotlin10ReleaseDate = SimpleDateFormat("yyyy-MM-ddX").parse("2016-02-15+00")
    println(Json.encodeToString(DateAsLongSerializer, kotlin10ReleaseDate))
}
```

輸出：

```
1455494400000
```

為屬性指定序列化器

當不可序列化類別（如 `Date`）的屬性作為可序列化類別的一部分被序列化時，我們必須為其提供序列化器，否則代碼將無法編譯。這可以通過在屬性上使用 `@Serializable` 註解來實現。

```
@Serializable
class ProgrammingLanguage{
```

```

val name: String,
@Serializable(with = DateAsLongSerializer::class)
val stableReleaseDate: Date
)

fun main() {
    val data = ProgrammingLanguage("Kotlin", SimpleDateFormat("yyyy-MM-ddX").parse("2016-02-15+00"))
    println(Json.encodeToString(data))
}

```

`stableReleaseDate` 屬性將使用我們為其指定的序列化策略進行序列化：

```

{"name":"Kotlin","stableReleaseDate":1455494400000}

```

為特定類型指定序列化器

`@Serializable` 註解也可以直接應用於類型。當需要為像 `Date` 這樣的類型提供自定義序列化器時，這非常方便。最常見的用例是當你有一個日期列表時：

```

@Serializable
class ProgrammingLanguage(
    val name: String,
    val releaseDates: List<@Serializable(DateAsLongSerializer::class) Date>
)

fun main() {
    val df = SimpleDateFormat("yyyy-MM-ddX")
    val data = ProgrammingLanguage("Kotlin", listOf(df.parse("2023-07-06+00"), df.parse("2023-04-25+00"), df.parse("2022-12-28+00")))
    println(Json.encodeToString(data))
}

```

輸出：

```

{"name":"Kotlin","releaseDates":[1688601600000,1682380800000,1672185600000]}

```

為檔案指定序列化器

可以通過在檔案開頭使用檔案級別的 `UseSerializers` 註解來為整個檔案中的特定類型（如 `Date`）指定序列化器。

```

@file:UseSerializers(DateAsLongSerializer::class)

```

現在可以在可序列化類別中使用 `Date` 屬性，而不需要額外的註解。

```

@Serializable
class ProgrammingLanguage(val name: String, val stableReleaseDate: Date)

fun main() {
    val data = ProgrammingLanguage("Kotlin", SimpleDateFormat("yyyy-MM-ddX").parse("2016-02-15+00"))
    println(Json.encodeToString(data))
}

```

輸出：

```

{"name":"Kotlin","stableReleaseDate":1455494400000}

```

使用型別別名全局指定序列化器

在處理序列化策略時，`kotlinx.serialization` 傾向於成為始終顯式的框架：通常，應在 `@Serializable` 註解中明確提到它們。因此，我們不提供任何類型的全局序列化器配置（除非後面提到的上下文序列化器）。

然而，在有大量檔案和類別的專案中，每次指定 `@file:UseSerializers` 可能太過繁瑣，尤其是對於像 `Date` 或 `Instant` 這樣在專案中有固定序列化策略的類別。對於這些情況，可以使用型別別名指定序列化器，因為它們會保留註解，包括與序列化相關的註解：

```

typealias DateAsLong = @Serializable(DateAsLongSerializer::class) Date

typealias

```

```
DateAsText = @Serializable(DateAsSimpleTextSerializer::class) Date
```

使用這些新的不同類型，可以在沒有額外註解的情況下序列化 `Date`：

```
@Serializable
class ProgrammingLanguage(val stableReleaseDate: DateAsText, val lastReleaseTimestamp: DateAsLong)

fun main() {
    val format = SimpleDateFormat("yyyy-MM-ddX")
    val data = ProgrammingLanguage(format.parse("2016-02-15+00"), format.parse("2022-07-07+00"))
    println(Json.encodeToString(data))
}
```

輸出：

```
{"stableReleaseDate":"2016-02-15","lastReleaseTimestamp":1657152000000}
```

泛型類型的自定義序列化器

讓我們看一下泛型 `Box<T>` 類別的例子。我們計劃為其編寫一個自定義序列化策略，因此將其標記為 `@Serializable(with = BoxSerializer::class)`。

```
@Serializable(with = BoxSerializer::class)
data class Box<T>(val contents: T)
```

為常規類型編寫 `KSerializer` 的實現就像我們在本章中的 `Color` 類型例子中所見那樣，作為一個 `object` 來處理。而對於泛型類別的序列化器，它需要一個構造函數，該構造函數接受與類型具有的泛型參數數量相同的 `KSerializer` 參數。讓我們編寫一個 `Box<T>` 序列化器，在序列化過程中擦除自身，並將所有工作委派給其 `data` 屬性的底層序列化器。

```
class BoxSerializer<T>(private val dataSerializer: KSerializer<T>) : KSerializer<Box<T>> {
    override val descriptor: SerialDescriptor = dataSerializer.descriptor
    override fun serialize(encoder: Encoder, value: Box<T>) = dataSerializer.serialize(encoder, value.contents)
    override fun deserialize(decoder: Decoder) = Box(dataSerializer.deserialize(decoder))
}
```

現在我們可以序列化和反序列化 `Box<Project>`。

```
@Serializable
data class Project(val name: String)

fun main() {
    val box = Box(Project("kotlinx.serialization"))
    val string = Json.encodeToString(box)
    println(string)
    println(Json.decodeFromString<Box<Project>>(string))
}
```

生成的 JSON 看起來像是直接序列化了 `Project` 類別。

```
{"name":"kotlinx.serialization"}
```

```
Box(contents=Project(name=kotlinx.serialization))
```

特定格式的序列化器

上述自定義序列化器對每種格式的工作方式相同。然而，可能存在格式特定的功能，序列化器實現可能希望利用這些功能。

"JSON 轉換" 章節提供了利用 JSON 特定功能的序列化器範例。

格式實現可以有針對某種類型的格式特定表示，如 "替代和自定義格式（實驗性）" 章節中的 "格式特定類型" 所解釋的。

本章接下來將介紹根據上下文調整序列化策略的通用方法。

上下文序列化

之前所有的自定義序列化策略都是靜態的，即在編譯時完全定義。例外情況是 "手動傳遞序列化器" 方法，但它僅適用於頂級物件。你可能需要在執行時更改深層物件樹中的物件的序列化策略，策略的選擇是基於上下文的。例如，你可能希望根據序列化資料的協議版本在 JSON 格式中將 `java.util.Date` 表示為 ISO 8601 字串或長整數。這就是所謂的上下文序列化，並且它由內建的 `ContextualSerializer` 類別支持。通常我們不需要顯式使用這個序列化器類別——可以使用 `@Contextual` 註解作為 `@Serializable(with = ContextualSerializer::class)` 註解的快捷方式，或者可以像 `UseSerializers` 註解那樣在檔案級別使用 `UseContextualSerialization` 註解。讓我們看一個利用前者的範例。

```
@Serializable
class ProgrammingLanguage(
    val name: String,
    @Contextual
    val stableReleaseDate: Date
)
```

要實際序列化這個類別，我們必須在調用 `encodeToXxx/decodeFromXxx` 函數時提供對應的上下文。否則，我們將得到 "Serializer for class 'Date' is not found" 的異常。

序列化器模組

要提供上下文，我們需要定義一個 `SerializersModule` 實例，該實例描述了在執行時應該使用哪些序列化器來序列化哪些上下文可序列化類別。這可以使用 `SerializersModule {}` 構建函數完成，該函數提供了 `SerializersModuleBuilder` DSL 來註冊序列化器。在下面的範例中，我們使用了帶有序列化器的 `contextual` 函數。對應的類別將通過內聯類型參數自動獲取該序列化器。

```
private val module = SerializersModule {
    contextual(DateAsLongSerializer)
}
```

接下來，我們使用 `json {}` 構建函數和 `serializersModule` 屬性創建一個帶有這個模組的 JSON 格式實例。

關於自定義 JSON 配置的詳細信息可以在 "JSON 配置" 章節中找到。

```
val format = json { serializersModule = module }
```

現在我們可以使用這個格式序列化我們的資料。

```
fun main() {
    val data = ProgrammingLanguage("Kotlin", SimpleDateFormat("yyyy-MM-ddX").parse("2016-02-15+00"))
    println(format.encodeToString(data))
}
```

輸出：

```
{"name":"Kotlin","stableReleaseDate":1455494400000}
```

上下文序列化與泛型類別

在上一節中，我們看到可以在模組中註冊序列化器實例，以便上下文中序列化我們想要的類別。我們還知道泛型類別的序列化器具有構造函數參數——類型參數序列化器。這意味著我們不能為類別使用一個序列化器實例，如果這個類別是泛型的：

```
val incorrectModule = SerializersModule {
    // 只能序列化 Box<Int>，而不能序列化 Box<String> 或其他類型
    contextual(BoxSerializer(Int.serializer()))
}
```

當我們想要上下文序列化泛型類別時，可以在模組中註冊提供者：

```
val correctModule = SerializersModule {
    // args[0] 包含 Int.serializer() 或 String.serializer()，具體取決於使用情況
    contextual(Box::class) { args -> BoxSerializer(args[0]) }
}
```

關於序列化模組的額外細節可以在 "多型性" 章節的 "合併庫序列化模組" 部分中找到。

為其他 Kotlin 類別衍生外部序列化器（實驗性）

如果要序列化的第三方類別是具有僅包含屬性的主要構造函數的 Kotlin 類別（這種類別本可以標記為 `@Serializable`），那麼您可以使用 `Serializer` 註解並在物件上設置 `forClass` 屬性來為其生成一個外部序列化器。

```
// NOT @Serializable
class Project(val name: String, val language: String)

@Serializer(forClass = Project::class)
object ProjectSerializer
```

您必須使用本章中解釋的一種方法將此序列化器綁定到類別。我們將按照 "手動傳遞序列化器" 方法進行此示例。

```
fun main() {
    val data = Project("kotlinx.serialization", "Kotlin")
    println(Json.encodeToString(ProjectSerializer, data))
}
```

這將所有的 `Project` 屬性序列化：

```
{"name":"kotlinx.serialization","language":"Kotlin"}
```

外部序列化使用屬性

正如我們之前看到的，常規 `@Serializable` 註解會創建一個序列化器，以便序列化備援欄位。使用 `Serializer(forClass = ...)` 的外部序列化無法訪問備援欄位，因此其工作方式不同。它僅

序列化具有 `setter` 的可訪問屬性或屬於主要構造函數的屬性。以下範例展示了這一點。

```
// NOT @Serializable, will use external serializer
class Project(
    // val in a primary constructor -- serialized
    val name: String
) {
    var stars: Int = 0 // property with getter & setter -- serialized

    val path: String // getter only -- not serialized
    get() = "kotlin/$name"

    private var locked: Boolean = false // private, not accessible -- not serialized
}

@Serializer(forClass = Project::class)
object ProjectSerializer

fun main() {
    val data = Project("kotlinx.serialization").apply { stars = 9000 }
    println(Json.encodeToString(ProjectSerializer, data))
}
```

輸出如下：

```
{"name":"kotlinx.serialization","stars":9000}
```

下一章將介紹多型性。

CH4 多型性

這是《Kotlin 序列化指南》的第四章。本章將介紹 Kotlin 序列化如何處理多型性類別層次結構。

目錄

- 閉合多型性
 - 靜態類型
 - 設計可序列化的層次結構

- 密封類別
- 自訂子類別序列名稱
- 基類中的具體屬性
- 物件
- 開放多型性
 - 註冊子類別
 - 序列化介面
 - 介面類型的屬性
 - 用於多型性的靜態父類型查找
 - 明確標記多型性類別屬性
 - 註冊多個超類別
 - 多型性與泛型類別
 - 合併庫序列化模組
 - 用於反序列化的預設多型性類型處理程序
 - 用於序列化的預設多型性類型處理程序

閉合多型性

讓我們從多型性的基本介紹開始。

靜態類型

Kotlin 序列化在預設情況下是完全靜態的。編碼物件的結構由物件的編譯時期類型決定。讓我們更詳細地探討這一點，並學習如何序列化多型性資料結構，其中資料的類型在運行時期確定。

為了展示 Kotlin 序列化的靜態性質，讓我們進行以下設置。一個開放類別 `Project` 只有 `name` 屬性，而其派生類別 `OwnedProject` 則增加了一個 `owner` 屬性。在下面的例子中，我們將序列化一個靜態類型為 `Project` 的變數 `data`，該變數在運行時期被初始化為 `OwnedProject` 的實例。

```
@Serializable
open class Project(val name: String)

class OwnedProject(name: String, val owner: String) : Project(name)

fun main() {
    val data: Project = OwnedProject("kotlinx.coroutines", "kotlin")
    println(Json.encodeToString(data))
}
```

儘管運行時期的類型是 `OwnedProject`，但只有 `Project` 類別的屬性被序列化。

```
{"name":"kotlinx.coroutines"}
```

讓我們將 `data` 的編譯時期類型更改為 `OwnedProject`。

```
@Serializable
open class Project(val name: String)

class OwnedProject(name: String, val owner: String) : Project(name)

fun main() {
    val data = OwnedProject("kotlinx.coroutines", "kotlin")
    println(Json.encodeToString(data))
}
```

我們會得到一個錯誤，因為 `OwnedProject` 類別不可序列化。

```
Exception in thread "main" kotlinx.serialization.SerializationException: Serializer for class 'OwnedProject' is not found.
Please ensure that class is marked as '@Serializable' and that the serialization compiler plugin is applied.
```

設計可序列化的層次結構

我們不能簡單地將前面例子中的 `OwnedProject` 標記為 `@Serializable`。這樣會導致編譯失敗，因為不符合構造函數屬性的要求。要使類別的層次結構可序列化，父類別中的屬性必須標記為抽象，使 `Project` 類別也成為抽象類別。

```
@Serializable
abstract class Project {
    abstract val name: String
}
```

```
class OwnedProject(override val name: String, val owner: String) : Project()

fun main() {
    val data: Project = OwnedProject("kotlinx.coroutines", "kotlin")
    println(Json.encodeToString(data))
}
```

這接近於設計可序列化類別層次結構的最佳方案，但運行時會產生以下錯誤：

```
Exception in thread "main" kotlinx.serialization.SerializationException: Serializer for subclass 'OwnedProject' is not found in the
polymorphic scope of 'Project'.
Check if class with serial name 'OwnedProject' exists and serializer is registered in a corresponding SerializersModule.
To be registered automatically, class 'OwnedProject' has to be '@Serializable', and the base class 'Project' has to be sealed and
'@Serializable'.
```

密封類別

使用多型性層次結構進行序列化的最簡單方法是將基類標記為密封類別。密封類別的所有子類別必須顯式標記為 `@Serializable`。

```
@Serializable
sealed class Project {
    abstract val name: String
}

@Serializable
class OwnedProject(override val name: String, val owner: String) : Project()

fun main() {
    val data: Project = OwnedProject("kotlinx.coroutines", "kotlin")
    println(Json.encodeToString(data)) // Serializing data of compile-time type Project
}
```

現在，我們可以看到在 JSON 中表示多型性的預設方式。類型鍵作為識別符被添加到生成的 JSON 物件中。

```
{"type":"example.examplePoly04.OwnedProject","name":"kotlinx.coroutines","owner":"kotlin"}
```

注意上例中與靜態類型相關的一個小但非常重要的細節：`val data` 屬性的編譯時期類型是 `Project`，儘管其運行時期類型是 `OwnedProject`。在序列化多型性類別層次結構時，你必須確保序列化物件的編譯時期類型是多型性的，而不是具體的。

讓我們看看如果將範例稍微更改，使被序列化物件的編譯時期類型為 `OwnedProject`（與其運行時期類型相同）會發生什麼。

```
@Serializable
sealed class Project {
    abstract val name: String
}

@Serializable
class OwnedProject(override val name: String, val owner: String) : Project()

fun main() {
    val data = OwnedProject("kotlinx.coroutines", "kotlin") // data: OwnedProject here
    println(Json.encodeToString(data)) // Serializing data of compile-time type OwnedProject
}
```

`OwnedProject` 的類型是具體的而不是多型性的，因此類型識別符屬性不會被寫入生成的 JSON。

```
{"name":"kotlinx.coroutines","owner":"kotlin"}
```

一般來說，Kotlin 序列化設計為僅在序列化期間使用的編譯時期類型與反序列化期間使用的編譯時期類型相同時，才能正常工作。你可以在調用序列化函數時始終顯式指定類型。可以通過調用 `Json.encodeToString<Project>(data)` 修正前面的範例以使用 `Project` 類型進行序列化。

自訂子類別序列名稱

類型鍵的值默認是完整的類別名稱。我們可以將 `SerialName` 註解放在對應的類別上來更改它。

```
@Serializable
sealed class Project {
```



```

    abstract val name: String
}

@Serializable
@SerialName("owned")
class OwnedProject(override val name: String, val owner: String) : Project()

fun main() {
    val data: Project = OwnedProject("kotlinx.coroutines", "kotlin")
    println(Json.encodeToString(data))
}

```

這樣，我們就可以擁有一個穩定的序列名稱，不會受源代碼中類別名稱的影響。

```

{"type":"owned","name":"kotlinx.coroutines","owner":"kotlin"}

```

此外，JSON 可以配置為使用不同的鍵名稱作為類別識別符。在 "用於多型性的類別識別符" 章節中可以找到一個範例。

基類中的具體屬性

密封層次結構中的基類可以具有具有備援字段的屬性。

```

@Serializable
sealed class Project {
    abstract val name: String
    var status = "open"
}

@Serializable
@SerialName("owned")
class OwnedProject(override val name: String, val owner: String) : Project()

fun main() {
    val json = Json { encodeDefaults = true } // "status" will be skipped otherwise
    val data: Project = OwnedProject("kotlinx.coroutines", "kotlin")
    println(json.encodeToString(data))
}

```

超類別的屬性在子類別的屬性之前被序列化。

```

{"type":"owned","status":"open","name":"kotlinx.coroutines","owner":"kotlin"}

```

####

物件

密封層次結構可以將物件作為其子類別，它們也需要標記為 `@Serializable`。讓我們來看一個不同的例子，其中有一個 `Response` 類別的層次結構。

```

@Serializable
sealed class Response

@Serializable
object EmptyResponse : Response()

@Serializable
class TextResponse(val text: String) : Response()

```

讓我們序列化一個包含不同響應的列表。

```

fun main() {
    val list = listOf(EmptyResponse, TextResponse("OK"))
    println(Json.encodeToString(list))
}

```

物件被序列化為一個空類別，預設也使用其完整類別名稱作為類型。

```

[{"type":"example.examplePoly08.EmptyResponse"}, {"type":"example.examplePoly08.TextResponse","text":"OK"}]

```

即使物件有屬性，它們也不會被序列化。

開放多型性

序列化可以處理任意開放的類別或抽象類別。然而，由於這種多型性是開放的，因此子類別可能在源代碼中的任何地方定義，甚至在其他模組中，序列化的子類別列表無法在編譯時期確定，必須在運行時期顯式註冊。

註冊子類別

讓我們從 "設計可序列化的層次結構" 章節的代碼開始。要使其在不將其標記為密封類別的情況下正常工作，我們必須使用 `SerializersModule {}` 構建函數定義一個 `SerializersModule`。在模組中，基類在 `polymorphic` 構建器中指定，並使用 `subclass` 函數註冊每個子類別。現在，可以使用這個模組實例化一個自定義的 JSON 配置並用於序列化。

```
val module = SerializersModule {
    polymorphic(Project::class) {
        subclass(OwnedProject::class)
    }
}

val format = Json { serializersModule = module }

@Serializable
abstract class Project {
    abstract val name: String
}

@Serializable
@SerialName("owned")
class OwnedProject(override val name: String, val owner: String) : Project()

fun main() {
    val data: Project = OwnedProject("kotlinx.coroutines", "kotlin")
    println(format.encodeToString(data))
}
```

這個附加的配置使我們的代碼像在 "密封類別" 章節中那樣工作，但這裡的子類別可以隨意地分佈在代碼中。

```
{"type":"owned","name":"kotlinx.coroutines","owner":"kotlin"}
```

請注意，這個例子僅在 JVM 上有效，因為序列化函數有一些限制。對於 JS 和 Native，應使用顯式序列化器：`format.encodeToString(PolymorphicSerializer(Project::class), data)`。你可以在[此處](#)跟蹤這個問題。

序列化介面

我們可以更新前面的範例，將 `Project` 超類別更改為介面。然而，我們不能將介面本身標記為 `@Serializable`。沒關係，介面不能自己擁有實例。介面只能由其派生類別的實例表示。介面在 Kotlin 語言中用於實現多型性，因此所有介面在默認情況下被視為隱式可序列化的，並且使用 `PolymorphicSerializer` 策略。我們只需要標記其實現類別為 `@Serializable` 並註冊它們。

```
interface Project {
    val name: String
}

@Serializable
@SerialName("owned")
class OwnedProject(override val name: String, val owner: String) : Project
```

現在，如果我們將 `data` 聲明為 `Project` 類型，就可以像以前一樣簡單地調用 `format.encodeToString`。

```
fun main() {
    val data: Project = OwnedProject("kotlinx.coroutines", "kotlin")
    println(format.encodeToString(data))
}
```

```
{"type":"owned","name":"kotlinx.coroutines","owner":"kotlin"}
```

注意：在 Kotlin/Native 上，由於反射能力有限，應使用 `format.encodeToString(PolymorphicSerializer(Project::class), data)`。

介面類型的屬性

繼續前面的範例，讓我們看看如果在某個其他可序列化類別中使用 `Project` 介面作為屬性會發生什麼。介面是隱式多型的，因此我們可以只聲明一個介面類型的屬性。

```
@Serializable
class Data(val project: Project) // Project is an interface

fun main() {
    val data = Data(OwnedProject("kotlinx.coroutines", "kotlin"))
    println(format.encodeToString(data))
}
```

只要我們在格式的 `SerializersModule` 中註冊了實際序列化的介面子類型，它就能在運行時正常工作。

```
{"project":{"type":"owned","name":"kotlinx.coroutines","owner":"kotlin"}}
```

用於多型性的靜態父類型查找

在序列化多型性類別時，多型性層次結構的根類型（在我們的例子中是 `Project`）是靜態確定的。讓我們舉一個具有可序列化的抽象類別 `Project` 的例子，但將 `main` 函數中的 `data` 聲明為 `Any` 類型：

```
fun main() {
    val data: Any = OwnedProject("kotlinx.coroutines", "kotlin")
    println(format.encodeToString(data))
}
```

我們會得到異常：

```
Exception in thread "main" kotlinx.serialization.SerializationException: Serializer for class 'Any' is not found.
Please ensure that class is marked as '@Serializable' and that the serialization compiler plugin is applied.
```

我們必須根據我們在源代碼中使用的對應靜態類型來註冊類別以進行多型性序列化。首先，我們更改模組以註冊 `Any` 的子類別：

```
val module = SerializersModule {
    polymorphic(Any::class) {
        subclass(OwnedProject::class)
    }
}
```

然後我們可以嘗試序列化 `Any` 類型的變數：

```
fun main() {
    val data: Any = OwnedProject("kotlinx.coroutines", "kotlin")
    println(format.encodeToString(data))
}
```

然而，`Any` 是一個類別，它是不可序列化的：

```
Exception in thread "main" kotlinx.serialization.SerializationException: Serializer for class 'Any' is not found.
Please ensure that class is marked as '@Serializable' and that the serialization compiler plugin is applied.
```

我們必須顯式傳遞 `PolymorphicSerializer` 的實例（基類為 `Any`）作為 `encodeToString` 函數的第一個參數。

```
fun main() {
    val data: Any = OwnedProject("kotlinx.coroutines", "kotlin")
    println(format.encodeToString(PolymorphicSerializer(Any::class), data))
}
```

使用顯式序列化器，它像以前一樣工作。

```
{"type":"owned","name":"kotlinx.coroutines","owner":"kotlin"}
```

明確標記多型性類別屬性

介面類型的屬性被隱式認為是多型性的，因為介面是關於運行時期多型性的。然而，Kotlin 序列化不會編譯具有不可序列化類別類型屬性的可序列化類別。如果我們有一個 `Any` 類別或其他不可序列化類別的屬性，那麼我們必須顯式提供其序列化策略，如我們在 "為屬性

指定序列化器" 章節中所見。要指定屬性的多型性序列化策略，使用特殊的 `@Polymorphic` 註解。

```
@Serializable
class Data {
    @Polymorphic // the code does not compile without it
    val project: Any
}

fun main() {
    val data = Data(OwnedProject("kotlinx.coroutines", "kotlin"))
    println(format.encodeToString(data))
}
```

註冊多個超類別

當同一類別作為不同超類別列表中的屬性的值被序列化時，我們必須在 `SerializersModule` 中分別為每個超類別註冊它。可以方便地將所有子類別的註冊提

取到一個單獨的函數中，並將其用於每個超類別。你可以使用以下模板來編寫它。

```
val module = SerializersModule {
    fun PolymorphicModuleBuilder<Project>.registerProjectSubclasses() {
        subclass(OwnedProject::class)
    }
    polymorphic(Any::class) { registerProjectSubclasses() }
    polymorphic(Project::class) { registerProjectSubclasses() }
}
```

多型性與泛型類別

對於可序列化類別的泛型子類型需要特殊處理。考慮以下層次結構：

```
@Serializable
abstract class Response<out T>

@Serializable
@SerialName("OkResponse")
data class OkResponse<out T>(val data: T) : Response<T>()
```

Kotlin 序列化沒有內建的策略來表示序列化多型性類型 `OkResponse<T>` 的屬性時實際提供的參數類型 `T`。我們必須在定義 `Response` 的序列化器模組時顯式提供此策略。在下面的範例中，我們使用 `OkResponse.serializer(...)` 來檢索 `OkResponse` 類別的插件生成的泛型序列化器，並使用 `PolymorphicSerializer` 的實例（基類為 `Any`）進行實例化。通過這種方式，我們可以序列化 `OkResponse` 的實例，該實例具有任何已作為 `Any` 的子類型多型性註冊的 `data` 屬性。

```
val responseModule = SerializersModule {
    polymorphic(Response::class) {
        subclass(OkResponse.serializer(PolymorphicSerializer(Any::class)))
    }
}
```

合併庫序列化模組

當應用程式規模增長並拆分為源代碼模組時，將所有類別層次結構存儲在一個序列化器模組中可能會變得不方便。讓我們將 "Project" 層次結構的庫添加到前一節的代碼中。

```
val projectModule = SerializersModule {
    fun PolymorphicModuleBuilder<Project>.registerProjectSubclasses() {
        subclass(OwnedProject::class)
    }
    polymorphic(Any::class) { registerProjectSubclasses() }
    polymorphic(Project::class) { registerProjectSubclasses() }
}
```

我們可以使用 `plus` 運算符將這兩個模組組合在一起，將它們合併，以便在同一個 `Json` 格式實例中使用它們。

你也可以在 `SerializersModule {}` DSL 中使用 `include` 函數。

```
val format = Json { serializersModule = projectModule + responseModule }
```

現在，這兩個層次結構中的類別可以一起序列化和反序列化。

```
fun main() {
    // both Response and Project are abstract and their concrete subtypes are being serialized
    val data: Response<Project> = OkResponse(OwnedProject("kotlinx.serialization", "kotlin"))
    val string = format.encodeToString(data)
    println(string)
    println(format.decodeFromString<Response<Project>>(string))
}
```

生成的 JSON 是深度多型性的。

```
{"type":"OkResponse","data":{"type":"OwnedProject","name":"kotlinx.serialization","owner":"kotlin"}}
```

```
OkResponse(data=OwnedProject(name=kotlinx.serialization, owner=kotlin))
```

如果你正在編寫一個帶有抽象類別及其一些實現的庫或共享模組，可以為客戶端公開你自己的序列化器模組，這樣客戶端就可以將你的模組與他們的模組結合起來使用。

用於反序列化的預設多型性類型處理程序

當我們反序列化未註冊的子類別時會發生什麼？

```
fun main() {
    println(format.decodeFromString<Project>("""
        {"type":"unknown","name":"example"}
        """))
}
```

我們會得到以下異常：

```
Exception in thread "main" kotlinx.serialization.json.internal.JsonDecodingException: Unexpected JSON token at offset 0: Serializer for subclass 'unknown' is not found in the polymorphic scope of 'Project' at path: $
Check if class with serial name 'unknown' exists and serializer is registered in a corresponding SerializersModule.
```

當讀取靈活的輸入時，我們可能希望在這種情況下提供一些預設行為。例如，我們可以有一個 `BasicProject` 子類型來表示所有未知的 `Project` 子類型。

```
@Serializable
abstract class Project {
    abstract val name: String
}

@Serializable
data class BasicProject(override val name: String, val type: String): Project()

@Serializable
@SerialName("OwnedProject")
data class OwnedProject(override val name: String, val owner: String) : Project()
```

我們使用 `polymorphic { ... }` DSL 中的 `defaultDeserializer` 函數註冊一個預設的反序列化器處理程序，該函數定義了一個策略，該策略將輸入中的類型字串映射到反序列化策略。在下面的範例中，我們沒有使用類型，而是始終返回 `BasicProject` 類別的插件生成的序列化器。

```
val module = SerializersModule {
    polymorphic(Project::class) {
        subclass(OwnedProject::class)
        defaultDeserializer { BasicProject.serializer() }
    }
}
```

使用這個模組，我們現在可以反序列化已註冊的 `OwnedProject` 實例和任何未註冊的實例。

```
val format = Json { serializersModule = module }

fun main() {
    println(format.decodeFromString<List<Project>>("""
[
```

```

        {"type":"unknown","name":"example"},
        {"type":"OwnedProject","name":"kotlinx.serialization","owner":"kotlin"}
    ]
    """))
}

```

注意，`BasicProject` 還捕獲了其 `type` 屬性中的指定類型鍵。

```
[BasicProject(name=example, type=unknown), OwnedProject(name=kotlinx.serialization, owner=kotlin)]
```

我們使用了一個插件生成的序列化器作為預設序列化器，這意味著 "unknown" 資料的結構是事先知道的。在現實世界的 API 中，這種情況很少見。為此需要一個自定義的、結構化較少的序列化器。你將在未來的 "維護自定義 JSON 屬性" 章節中看到此類序列化器的範例。

用於序列化的預設多型性類型處理程序

有時你需要根據實例動態選擇用於多型性類型的序列化器，例如，如果你無法訪問完整的類型層次結構，或者它經常發生變化。對於這種情況，你可以註冊一個預設序列化器。

```

interface Animal {
}

interface Cat : Animal {
    val catType: String
}

interface Dog : Animal {
    val dogType: String
}

private class CatImpl : Cat {
    override val catType: String = "Tabby"
}

private class DogImpl : Dog {
    override val dogType: String = "Husky"
}

object AnimalProvider {
    fun createCat(): Cat = CatImpl()
    fun createDog(): Dog = DogImpl()
}

```

我們使用 `SerializersModule { ... }` DSL 中的 `polymorphicDefaultSerializer` 函數註冊一個預設序列化器處理程序，該函數定義了一個策略，該策略接受基類的實例並提供一個序列化策略。在下面的範例中，我們使用 `when` 區塊來檢查實例的類型，而無需引用私有實現類別。

```

val module = SerializersModule {
    polymorphicDefaultSerializer(Animal::class) { instance ->
        @Suppress("UNCHECKED_CAST")
        when (instance) {
            is Cat -> CatSerializer as SerializationStrategy<Animal>
            is Dog -> DogSerializer as SerializationStrategy<Animal>
            else -> null
        }
    }
}

object CatSerializer : SerializationStrategy<Cat> {
    override val descriptor = buildClassSerialDescriptor("Cat") {
        element<String>("catType")
    }

    override fun serialize(encoder: Encoder, value: Cat) {
        encoder.encodeStructure(descriptor) {
            encodeStringElement(descriptor, 0, value.catType)
        }
    }
}

```

```
object DogSerializer : SerializationStrategy<Dog> {
    override val descriptor = buildClassSerialDescriptor("Dog") {
        element<String>("dogType")
    }

    override fun serialize(encoder: Encoder, value: Dog) {
        encoder.encodeStructure(descriptor) {
            encodeStringElement(descriptor, 0
, value.dogType)
        }
    }
}
```

使用這個模組，我們現在可以序列化 `Cat` 和 `Dog` 的實例。

```
val format = Json { serializersModule = module }

fun main() {
    println(format.encodeToString<Animal>(AnimalProvider.createCat()))
}
```

輸出：

```
{"type":"Cat","catType":"Tabby"}
```

下一章將介紹 JSON 特性。

🕒 修訂版本 #1

★ 由 treeman 建立於 2 🕒 2024 09:52:17

✍ 由 treeman 更新於 2 🕒 2024 09:53:04