

【Kotlin】Serialization Chapter 5. JSON Features

JSON 特性

這是 Kotlin 序列化指南的第五章。本章將介紹 `Json` 類別中可用的 JSON 序列化功能。

目錄

- JSON 配置
- 美化打印 (Pretty Printing)
- 寬鬆解析 (Lenient Parsing)
- 忽略未知鍵 (Ignoring Unknown Keys)
- 替代的 JSON 名稱 (Alternative Json Names)
- 編碼預設值 (Encoding Defaults)
- 明確的 `null` (Explicit Nulls)
- 強制輸入值 (Coercing Input Values)
- 允許結構化的映射鍵 (Allowing Structured Map Keys)
- 允許特殊的浮點值 (Allowing Special Floating-Point Values)
- 用於多型性的類別區分符 (Class Discriminator for Polymorphism)
- 類別區分符輸出模式 (Class Discriminator Output Mode)
- 以不區分大小寫的方式解碼枚舉 (Decoding Enums in a Case-Insensitive Manner)
- 全域命名策略 (Global Naming Strategy)
- Base64
- JSON 元素
- 解析為 JSON 元素
- JSON 元素的類型
- JSON 元素建構器
- 解碼 JSON 元素
- 編碼字面值 JSON 內容 (實驗性功能)
- 序列化大型十進制數字
- 使用 `JsonUnquotedLiteral` 創建字面值不帶引號的 `null` 是被禁止的
- JSON 轉換
- 陣列包裝
- 陣列拆包
- 操作預設值
- 基於內容的多型性反序列化
- 底層實作 (實驗性功能)
- 維護自定義的 JSON 屬性

JSON 配置

默認的 `Json` 實作對無效的輸入非常嚴格。它強制執行 Kotlin 的類型安全性，並限制可以序列化的 Kotlin 值，以確保生成的 JSON 表示是標準的。通過創建自定義的 JSON 格式實例，可以支持許多非標準的 JSON 功能。

要使用自定義 JSON 格式配置，可以從現有的 `Json` 實例（例如默認的 `Json` 對象）中使用 `Json()` 建構函數來創建自己的 `Json` 類別實例。在括號中通過 `JsonBuilder` DSL 指定參數值。生成的 `Json` 格式實例是不可變且線程安全的；可以簡單地將其存儲在頂層屬性中。

出於性能原因，建議你存儲和重複使用自定義的格式實例，因為格式實作可能會快取有關其序列化的類別的特定於格式的附加資訊。

本章介紹了 `Json` 支持的配置功能。

美化打印

默認情況下，`Json` 輸出一行。你可以通過設置 `prettyPrint` 屬性為 `true` 來配置它以美化打印輸出（即添加縮排和換行以提高可讀性）：

```
val format = Json { prettyPrint = true }

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

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

這會給出如下優美的結果：

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

寬鬆解析

默認情況下，`Json` 解析器強制執行各種 JSON 限制，以盡可能符合規範（參見 RFC-4627）。特別是，鍵和字串文字必須加引號。這些限制可以通過設置 `isLenient` 屬性為 `true` 來放寬。當 `isLenient = true` 時，你可以解析格式相當自由的資料：

```
val format = Json { isLenient = true }

enum class Status { SUPPORTED }

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

fun main() {
    val data = format.decodeFromString<Project>("""
        {
            name : kotlinx.serialization,
            status : SUPPORTED,
            votes : "9000"
        }
        """)
    println(data)
}
```

即使源 JSON 的所有鍵、字串和枚舉值未加引號，而整數被加引號，仍然可以獲取對象：

```
Project(name=kotlinx.serialization, status=SUPPORTED, votes=9000)
```

忽略未知鍵

JSON 格式通常用於讀取第三方服務的輸出或其他動態環境中，在這些環境中，在 API 演進過程中可能會添加新屬性。默認情況下，反序列化期間遇到的未知鍵會產生錯誤。你可以通過設置 `ignoreUnknownKeys` 屬性為 `true` 來避免這種情況並僅忽略這些鍵：

```
val format = Json { ignoreUnknownKeys = true }

@Serializable
data class Project(val name: String)

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

即使 `Project` 類別沒有 `language` 屬性，它也會解碼對象：

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

替代的 JSON 名稱

當 JSON 字段因為模式版本更改而重新命名時，這並不罕見。你可以使用 `@SerialName` 註解來更改 JSON 字段的名稱，但這種重命名會阻止使用舊名稱解碼資料。為了支持一個 Kotlin 屬性具有多個 JSON 名稱，可以使用 `@JsonNames` 註解：

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

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

如你所見，`name` 和 `title` JSON 字段都對應於 `name` 屬性：

```
Project(name=kotlinx.serialization)
Project(name=kotlinx.coroutines)
```

`@JsonNames` 註解的支持由 `JsonBuilder.useAlternativeNames` 標誌控制。與大多數配置標誌不同，這個是默認啟用的，不需要特別關注，除非你想進行一些微調。

編碼預設值

屬性的預設值默認不會被編碼，因為它們在解碼過程中缺失字段時會被賦值。詳情和範例參見 `Defaults are not encoded` 一節。這對於具有 `null` 預設值的可空屬性尤其有用，避免了編寫對應的 `null` 值。可以通過將 `encodeDefaults` 屬性設置為 `true` 來更改預設行為：

```
val format = Json { encodeDefaults = true }

@Serializable
class Project(
    val name: String,
    val language: String = "Kotlin",
    val website: String? = null
)

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

這會生成以下輸出，編碼了所有屬性值，包括預設值：

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

明確的 `null`

默認情況下，所有 `null` 值都被編碼為 JSON 字串，但在某些情況下，你可能希望省略它們。`null` 值的編碼可以通過 `explicitNulls` 屬性進行控制。

如果你將屬性設置為 `false`，即使屬性沒有預設 `null` 值，`null` 值的字段也不會編碼到 JSON 中。當解碼此類 JSON 時，對於沒有預設值的可空屬性，缺少屬性值會被視為 `null`。

```
val format = Json { explicitNulls = false }

@Serializable
data class Project(
    val name: String,
    val language: String,
    val version: String? = "1.2.2",
    val website: String?,
    val description: String? = null
)

fun main() {
    val data = Project("kotlinx.serialization", "Kotlin", null, null, null)
    val json = format.encodeToString(data)
    println(json)
    println(format.decodeFromString<Project>(json))
}
```

如你所見，`version`、`website` 和 `description` 字段在第一行的輸出 JSON 中並不存在。解碼後，沒有預設值的可空屬性 `website` 獲得了 `null` 值，而可空屬性 `version` 和 `description` 則使用了其預設值：

```
{"name":"kotlinx.serialization","language":"Kotlin"}
Project(name=kotlinx.serialization, language=Kotlin, version=1.2.2, website=null, description=null)
```

注意，`version` 在編碼前是 `null`，而在解碼後變為 `1.2.2`。如果 `explicitNulls` 設置為 `false`，則這種屬性（可空但有非 `null` 預設值）的編碼/解碼變得不對稱。

如果想讓解碼器將某些無效的輸入資料視為缺少字段來增強此標誌的功能，請參閱 `coerceInputValues` 下面的詳細信息。

`explicitNulls` 默認為 `true`，因為這是不同版本庫中的默認行為。

強制輸入值

來自第三方的 JSON 格式可以演變，有時會改變字段類型。這可能會在解碼過程中導致異常，因為實際值與預期值不匹配。默認的 `Json` 實作對輸入類型非常嚴格，如在 `Type safety is enforced` 部分中所展示的。你可以通過 `coerceInputValues` 屬性來放寬此限制。

此屬性僅影響解碼。它將一小部分無效的輸入值視為相應屬性缺失。當前支持的無效值清單包括：

- 對於不可空類型的 `null` 輸入
- 枚舉的未知值

如果缺少值，則會用預設屬性值（如果存在）替換，或者如果 `explicitNulls` 標誌設置為 `false` 且屬性為可空，則替換為 `null`（對於枚舉）。

此清單可能在未來擴展，因此配置此屬性的 `Json` 實例將對輸入中的無效值更加寬容，並用預設值或 `null` 進行替換。

參見 `Type safety is enforced` 部分的範例：

```
val format = Json { coerceInputValues = true }

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

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

`language` 屬性的無效 `null` 值被強制轉換為預設值：

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

以下範例展示了如何與 `explicitNulls` 旗標一起使用以強制無效的枚舉值：

```
enum class Color { BLACK, WHITE }

@Serializable
data class Brush(val foreground: Color = Color.BLACK, val background: Color?)

val json = Json {
    coerceInputValues = true
    explicitNulls = false
}

fun main() {
    val brush = json.decodeFromString<Brush>("""{"foreground":"pink", "background":"purple"}""")
    println(brush)
}
```

即使我們沒有 `Color.pink` 和 `Color.purple` 顏色，`decodeFromString` 函數仍成功返回：

```
Brush(foreground=BLACK, background=null)
```

`foreground` 屬性獲得了其預設值，而 `background` 屬性由於 `explicitNulls = false` 設置而獲得了 `null`。

允許結構化的映射鍵

JSON 格式本質上不支持具有結構化鍵的映射（Map）的概念。JSON 對象中的鍵是字串，並且默認只能用於表示基本類型或枚舉。你可以通過設置 `allowStructuredMapKeys` 屬性來啟用對結構化鍵的非標準支持。

以下是如何序列化具有使用者定義類別鍵的映射：

```
val format = Json { allowStructuredMapKeys = true }

@Serializable
data class Project(val name: String)

fun main() {
```

```
val map = mapOf(
    Project("kotlinx.serialization") to "Serialization",
    Project("kotlinx.coroutines") to "Coroutines"
)
println(format.encodeToString(map))
}
```

使用結構化鍵的映射被表示為 JSON 陣列，並包含以下項目：`[key1, value1, key2, value2,...]`。

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

允許特殊的浮點值

默認情況下，由於 JSON 規範禁止，特殊的浮點值如 `Double.NaN` 和無限大在 JSON 中不受支持。你可以通過 `allowSpecialFloatingPointValues` 屬性來啟用它們的編碼：

```
val format = Json { allowSpecialFloatingPointValues = true }

@Serializable
class Data(
    val value: Double
)

fun main() {
    val data = Data(Double.NaN)
    println(format.encodeToString(data))
}
```

此範例生成以下非標準 JSON 輸出，但這在 JVM 世界中是一種廣泛使用的編碼方式：

```
{"value":NaN}
```

用於多型性的類別區分符

當你有多型性資料時，可以在 `classDiscriminator` 屬性中指定一個鍵名稱來指定類型：

```
val format = Json { classDiscriminator = "#class" }

@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(format.encodeToString(data))
}
```

結合明確指定的類別 `SerialName`，你可以完全控制生成的 JSON 對象：

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

還可以為不同的繼承層次結構指定不同的類別區分符。可以在基礎可序列化類別上直接使用 `@JsonClassDiscriminator` 註解，而不是 `Json` 實例屬性：

```
@Serializable
@JsonClassDiscriminator("message_type")
sealed class Base
```

此註解是可繼承的，因此 `Base` 的所有子類別都將具有相同的區分符：

```
@Serializable // Class discriminator is inherited from Base
sealed class ErrorClass: Base()
```

要了解有關可繼承序列化註解的更多資訊，請參見 `InheritableSerialInfo` 的文檔。

請注意，無法在 `Base` 子類中明確指定不同的類別區分符。只有交集為空的繼承層次結構才能具有不同的區分符。

註解中指定的區分符優先於 `Json` 配置中的區分符：

```
val format = Json { classDiscriminator = "#class" }

fun main() {
    val data = Message(BaseMessage("not found"), GenericError(404))
    println(format.encodeToString(data))
}
```

如你所見，使用了 `Base` 類別的區分符：

```
{"message":{"message_type":"my.app.BaseMessage","message":"not found"},"error":
{"message_type":"my.app.GenericError","error_code":404}}
```

類別區分符輸出模式

類別區分符為序列化和反序列化多型性類別層次結構提供了資訊。如上所示，它默認僅添加於多型性類別。當你希望為各種第三方 API 編碼更多或更少的輸出類型資訊時，可以通過 `JsonBuilder.classDiscriminatorMode` 屬性來控制類別區分符的添加。

例如，`ClassDiscriminatorMode.NONE` 不會添加類別區分符，這適用於接收方不關心 Kotlin 類型的情況：

```
val format = Json { classDiscriminatorMode = ClassDiscriminatorMode.NONE }

@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(format.encodeToString(data))
}
```

請注意，無法使用 `kotlinx.serialization`

將此輸出反序列化回去。

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

另外兩個可用的值是 `ClassDiscriminatorMode.POLYMORPHIC`（默認行為）和 `ClassDiscriminatorMode.ALL_JSON_OBJECTS`（盡可能地添加區分符）。有關詳細信息，請參閱其文檔。

以不區分大小寫的方式解碼枚舉

Kotlin 的命名策略建議使用大寫下劃線分隔名稱或駝峰式名稱來命名枚舉值。`Json` 默認使用確切的 Kotlin 枚舉值名稱進行解碼。但是，有時候第三方 JSON 中的這些值是小寫的或混合大小寫的。在這種情況下，可以使用 `JsonBuilder.decodeEnumsCaseInsensitive` 屬性以不區分大小寫的方式解碼枚舉值：

```
val format = Json { decodeEnumsCaseInsensitive = true }

enum class Cases { VALUE_A, @JsonNames("Alternative") VALUE_B }

@Serializable
data class CasesList(val cases: List<Cases>)

fun main() {
    println(format.decodeFromString<CasesList>("""{"cases":["value_A", "alternative"]}"""))
}
```

這會影響序列名稱以及使用 `JsonNames` 註解指定的替代名稱，因此這兩個值都會成功解碼：

```
CasesList(cases=[VALUE_A, VALUE_B])
```

此屬性不會以任何方式影響編碼。

全域命名策略

如果 JSON 輸入中的屬性名稱與 Kotlin 不同，建議使用 `@SerialName` 註解顯式指定每個屬性的名稱。但是，有些情況下需要將轉換應用於每個序列名稱，例如從其他框架遷移或遺留代碼庫。在這些情況下，可以為 `Json` 實例指定 `namingStrategy`。kotlinx.serialization 提供了一種現成的策略實作，即 `JsonNamingStrategy.SnakeCase`：

```
@Serializable
data class Project(val projectName: String, val projectOwner: String)

val format = Json { namingStrategy = JsonNamingStrategy.SnakeCase }

fun main() {
    val project = format.decodeFromString<Project>("""{"project_name":"kotlinx.coroutines", "project_owner":"Kotlin"}""")
    println(format.encodeToString(project.copy(projectName = "kotlinx.serialization")))
}
```

如你所見，序列化和反序列化都如同所有序列名稱都從駝峰式轉換為蛇形：

```
{"project_name":"kotlinx.serialization","project_owner":"Kotlin"}
```

在處理 `JsonNamingStrategy` 時需要注意一些限制：

由於 `kotlinx.serialization` 框架的性質，命名策略轉換應用於所有屬性，不論其序列名稱是取自屬性名稱還是通過 `@SerialName` 註解提供。實際上，這意味著無法通過顯式指定序列名稱來避免轉換。要能夠反序列化未轉換的名稱，可以使用 `JsonNames` 註解。

轉換後的名稱與其他（轉換後的）屬性序列名稱或使用 `JsonNames` 指定的任何替代名稱發生衝突將導致反序列化異常。

全域命名策略非常隱晦：僅從類別定義無法確定其在序列化形式中的名稱。因此，命名策略對於 IDE 中的 `Find Usages` / `Rename` 操作、`grep` 全文搜索等操作來說不友好。對於它們而言，原始名稱和轉換後的名稱是兩個不同的東西；更改一個而不更改另一個可能會以多種意想不到的方式引入錯誤，並導致更大的代碼維護成本。

因此，在考慮向應用程序中添加全域命名策略之前，應仔細權衡利弊。

Base64

要編碼和解碼 Base64 格式，我們需要手動編寫一個序列化器。在這裡，我們將使用 Kotlin 的默認 Base64 編碼器實作。請注意，某些序列化器默認使用不同的 RFC 進行 Base64 編碼。例如，Jackson 默認使用 Base64 Mime 的變體。在 `kotlinx.serialization` 中可以使用 `Base64.Mime` 編碼器來實現相同的結果。Kotlin 的 Base64 文檔列出了其他可用的編碼器。

```
import kotlinx.serialization.encoding.Encoder
import kotlinx.serialization.encoding.Decoder
import kotlinx.serialization.descriptors.*
import kotlin.io.encoding.*

@OptIn(ExperimentalEncodingApi::class)
object ByteArrayAsBase64Serializer : KSerializer<ByteArray> {
    private val base64 = Base64.Default

    override val descriptor: SerialDescriptor
    get() = PrimitiveSerialDescriptor(
        "ByteArrayAsBase64Serializer",
        PrimitiveKind.STRING
    )

    override fun serialize(encoder: Encoder, value: ByteArray) {
        val base64Encoded = base64.encode(value)
        encoder.encodeString(base64Encoded)
    }

    override fun deserialize(decoder: Decoder): ByteArray {
        val base64Decoded = decoder.decodeString()
        return base64.decode(base64Decoded)
    }
}
```

有關如何創建自己的自定義序列化器的詳細信息，請參見 `custom serializers`。

然後我們可以像這樣使用它：

```
@Serializable
data class Value(
    @Serializable(with = ByteArrayAsBase64Serializer::class)
    val base64Input: ByteArray
) {
    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (javaClass != other?.javaClass) return false
        other as Value
        return base64Input.contentEquals(other.base64Input)
    }

    override fun hashCode(): Int {
        return base64Input.contentHashCode()
    }
}

fun main() {
    val string = "foo string"
    val value = Value(string.toByteArray())
    val encoded = Json.encodeToString(value)
    println(encoded)
    val decoded = Json.decodeFromString<Value>(encoded)
    println(decoded.base64Input.decodeToString())
}
```

```
{"base64Input":"Zm9vIHNOcmLuZW=="}
```

```
foo string
```

注意，我們編寫的序列化器並不依賴於 `Json` 格式，因此它可以在任何格式中使用。

對於在許多地方使用此序列化器的專案，可以通過使用 `typealias` 全域性地指定序列化器，以避免每次都指定序列化器。例如：

```
typealias Base64ByteArray = @Serializable(ByteArrayAsBase64Serializer::class) ByteArray
```

JSON 元素

除了字串與 JSON 對象之間的直接轉換外，Kotlin 序列化還提供了允許其他方式處理 JSON 的 API。例如，你可能需要在解析之前調整資料，或處理不容易適合 Kotlin 序列化的類型安全世界的非結構化資料。

此部分庫中的主要概念是 `JsonElement`。繼續閱讀以了解你可以用它做什麼。

解析為 JSON 元素

可以使用 `Json.parseToJsonElement` 函數將字串解析為 `JsonElement` 實例。這不叫解碼也不叫反序列化，因為在此過程中不會發生這些操作。它只是解析 JSON 並形成表示它的對象：

```
fun main() {
    val element = Json.parseToJsonElement("""
        {"name":"kotlinx.serialization","language":"Kotlin"}
        """)
    println(element)
}
```

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

JSON 元素的類型

`JsonElement` 類別有三個直接子類型，與 JSON 語法密切相關：

- `JsonPrimitive` 代表基本 JSON 元素，例如字串、數字、布林值和 `null`。每個基本元素都有一個簡單的字串內容。還有一個 `JsonPrimitive()` 構造函數，重載以接受各種基本 Kotlin 類型並將它們轉換為 `JsonPrimitive`。
- `JsonArray` 代表 JSON [...] 陣列。它是 Kotlin 列表，包含 `JsonElement` 項目。
- `JsonObject` 代表 JSON {...} 對象。它是從字串鍵到 `JsonElement` 值的 Kotlin 映射。

`JsonElement` 類別有擴展函數，將其轉換為相應的子類型：`jsonPrimitive`、`jsonArray`、`jsonObject`。 `JsonPrimitive` 類別依次提供轉換為 Kotlin 基本類型的轉換器：`int`、`intOrNull`、`long`、`longOrNull`，以及其他類型的類似轉換

器。你可以使用它們來處理已知結構的 JSON：

```
fun main() {
    val element = Json.parseToJsonElement("""
        {
            "name": "kotlinx.serialization",
            "forks": [{ "votes": 42 }, { "votes": 9000 }, { } ]
        }
    """)
    val sum = element
        .jsonObject["forks"]!!
        .jsonArray.sumOf { it.jsonObject["votes"]?.jsonPrimitive?.int ?: 0 }
    println(sum)
}
```

上面的範例會對 `forks` 陣列中所有對象的 `votes` 進行求和，忽略沒有 `votes` 的對象：

```
9042
```

請注意，如果資料的結構與預期不同，執行將會失敗。

JSON 元素建構器

你可以使用相應的建構器函數 `buildJsonArray` 和 `buildJsonObject` 構建 `JsonElement` 子類型的實例。它們提供了一種 DSL 來定義生成的 JSON 結構。這類似於 Kotlin 標準庫集合建構器，但具有 JSON 特定的便利功能，例如更多類型特定的重載和內部建構器函數。以下範例顯示了所有關鍵功能：

```
fun main() {
    val element = buildJsonObject {
        put("name", "kotlinx.serialization")
        putJsonObject("owner") {
            put("name", "kotlin")
        }
        putJsonArray("forks") {
            addJsonObject {
                put("votes", 42)
            }
            addJsonObject {
                put("votes", 9000)
            }
        }
    }
    println(element)
}
```

```
{"name":"kotlinx.serialization","owner":{"name":"kotlin"},"forks":[{"votes":42},{"votes":9000}]}
```

解碼 JSON 元素

`JsonElement` 類別的實例可以使用 `Json.decodeFromJsonElement` 函數解碼為可序列化對象：

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

fun main() {
    val element = buildJsonObject {
        put("name", "kotlinx.serialization")
        put("language", "Kotlin")
    }
    val data = Json.decodeFromJsonElement<Project>(element)
    println(data)
}
```

結果完全符合預期：

編碼字面值 JSON 內容（實驗性功能）

此功能是實驗性的，需要選擇加入 `Kotlinx Serialization API`。

在某些情況下，可能需要編碼任意未加引號的值。這可以通過 `JsonUnquotedLiteral` 實現。

序列化大型十進制數字

JSON 規範對數字的大小或精度沒有限制，但是無法使用 `JsonPrimitive()` 序列化任意大小或精度的數字。

如果使用 `Double`，則數字的精度受到限制，這意味著大數字會被截斷。在使用 Kotlin/JVM 時，可以改用 `BigDecimal`，但 `JsonPrimitive()` 將值編碼為字串，而不是數字。

```
import java.math.BigDecimal

val format = Json { prettyPrint = true }

fun main() {
    val pi = BigDecimal("3.141592653589793238462643383279")

    val piJsonDouble = JsonPrimitive(pi.toDouble())
    val piJsonString = JsonPrimitive(pi.toString())

    val piObject = buildJsonObject {
        put("pi_double", piJsonDouble)
        put("pi_string", piJsonString)
    }

    println(format.encodeToString(piObject))
}
```

儘管 `pi` 被定義為具有 30 位小數的數字，但生成的 JSON 並未反映這一點。`Double` 值被截斷為 15 位小數，而 `String` 被包裹在引號中，這不是 JSON 數字。

```
{
  "pi_double": 3.141592653589793,
  "pi_string": "3.141592653589793238462643383279"
}
```

為了避免精度損失，可以使用 `JsonUnquotedLiteral` 編碼 `pi` 的字串值。

```
import java.math.BigDecimal

val format = Json { prettyPrint = true }

fun main() {
    val pi = BigDecimal("3.141592653589793238462643383279")

    // 使用 JsonUnquotedLiteral 編碼原始 JSON 內容
    val piJsonLiteral = JsonUnquotedLiteral(pi.toString())

    val piJsonDouble = JsonPrimitive(pi.toDouble())
    val piJsonString = JsonPrimitive(pi.toString())

    val piObject = buildJsonObject {
        put("pi_literal", piJsonLiteral)
        put("pi_double", piJsonDouble)
        put("pi_string", piJsonString)
    }

    println(format.encodeToString(piObject))
}
```

`pi_literal` 現在準確地匹配定義的值。

```
{
```

```
"pi_literal": 3.141592653589793238462643383279,
"pi_double": 3.141592653589793,
"pi_string": "3.141592653589793238462643383279"
}
```

要將 `pi` 解碼回 `BigDecimal`，可以使用 `JsonPrimitive` 的字串內容。

(此演示使用 `JsonPrimitive` 為了簡便。要獲得更可重用的處理序列化的方法，請參見 `Json Transformations` 下面的部分。)

```
import java.math.BigDecimal

fun main() {
    val piObjectJson = """
        {
            "pi_literal": 3.141592653589793238462643383279
        }
    """.trimIndent()

    val piObject: JsonObject = Json.decodeFromString(piObjectJson)

    val piJsonLiteral = piObject["pi_literal"]!!.jsonPrimitive.content

    val pi = BigDecimal(piJsonLiteral)

    println(pi)
}
```

```
3.141592653589793238462643383279
```

使用 `JsonUnquotedLiteral` 創建字面值不帶引號的 `null` 是被禁止的

為了避免創建不一致的狀態，編碼等於 `"null"` 的字串是被禁止的。請改用 `JsonNull` 或 `JsonPrimitive`。

```
fun main() {
    // 注意：使用 JsonUnquotedLiteral 創建 null 將導致異常！
    JsonUnquotedLiteral("null")
}
```

```
Exception in thread "main" kotlinx.serialization.json.internal.JsonEncodingException: Creating a literal unquoted value of 'null' is forbidden. If you want to create JSON null literal, use JsonNull object, otherwise, use JsonPrimitive
```

JSON 轉換

要影響序列化後的 JSON 輸出的形狀和內容，或適應反序列化的輸入，可以編寫自定義序列化器。但是，特別是對於相對較小且簡單的任務來說，仔細遵循 `Encoder` 和 `Decoder` 調用約定可能很不方便。為此，Kotlin 序列化提供了 API，可以將實作自定義序列化器的負擔減輕為操作 JSON 元素樹的問題。

我們建議你熟悉 `Serializers` 章節：其中解釋了如何將自定義序列化器綁定到類別。

轉換功能由 `JsonTransformingSerializer` 抽象類別提供，它實作了 `KSerializer`。此類別不直接與 `Encoder` 或 `Decoder` 交互，而是要求你通過 `transformSerialize` 和 `transformDeserialize` 方法為表示為 `JsonElement` 類別的 JSON 樹提供轉換。我們來看看範例。

陣列包裝

第一個範例是為列表實作 JSON 陣列包裝。

考慮一個 REST API，它返回一個 `User` 對象的 JSON 陣列，如果結果中只有一個元素，則返回單個對象（未包裝到陣列中）。

在資料模型中，使用 `@Serializable` 註解指定 `users: List<User>` 屬性使用自定義序列化器。

```
@Serializable
data class Project(
    val name: String,
    @Serializable(with = UserListSerializer::class)
    val users: List<User>
)

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

由於此範例僅涵蓋反序列化情況，因此你可以實作 `UserListSerializer` 並僅重寫 `transformDeserialize` 函數。 `JsonTransformingSerializer` 構造函數將原始序列化器作為參數（此方法在 `Constructing collection serializers` 部分中顯示）：

```
object UserListSerializer : JsonTransforming

Serializer<List<User>>(ListSerializer(User.serializer())) {
    // 如果響應不是一個陣列，那麼它是一個應該被包裝到陣列中的單個對象
    override fun transformDeserialize(element: JsonElement): JsonElement =
        if (element !is JsonArray) JsonArray(listOf(element)) else element
}
```

現在你可以用 JSON 陣列或單個 JSON 對象作為輸入來測試代碼。

```
fun main() {
    println(Json.decodeFromString<Project>("""
        {"name":"kotlin.serialization","users":{"name":"kotlin"}}
        """))
    println(Json.decodeFromString<Project>("""
        {"name":"kotlin.serialization","users":[{"name":"kotlin"}, {"name":"jetbrains"}]}
        """))
}
```

輸出顯示了兩種情況都正確地反序列化為 Kotlin 列表。

```
Project(name=kotlin.serialization, users=[User(name=kotlin)])
Project(name=kotlin.serialization, users=[User(name=kotlin), User(name=jetbrains)])
```

陣列拆包

你還可以實作 `transformSerialize` 函數，在序列化期間將單個元素列表拆包為單個 JSON 對象：

```
override fun transformSerialize(element: JsonElement): JsonElement {
    require(element is JsonArray) // 此序列化器僅用於列表
    return element.singleOrNull() ?: element
}
```

現在，如果你從 Kotlin 序列化單個元素列表對象：

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

你最終會得到一個單個 JSON 對象，而不是一個包含一個元素的陣列：

```
{"name":"kotlin.serialization","users":{"name":"kotlin"}}
```

操作預設值

另一種有用的轉換是從輸出 JSON 中省略特定值，例如，如果它被用作缺少時的預設值或出於其他原因。

假設你由於某些原因無法為 `Project` 資料模型中的 `language` 屬性指定預設值，但你需要在它等於 Kotlin 時將其從 JSON 中省略（我們都同意 Kotlin 應該是預設值）。你可以通過基於 `Project` 類別的 `Plugin` 生成的序列化器來編寫特別的 `ProjectSerializer`。

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

object ProjectSerializer : JsonTransformingSerializer<Project>(Project.serializer()) {
    override fun transformSerialize(element: JsonElement): JsonElement =
        // 過濾掉鍵為 "language" 且值為 "Kotlin" 的頂層鍵值對
        JsonObject(element.jsonObject.filterNot {
            (k, v) -> k == "language" && v.jsonPrimitive.content == "Kotlin"
        })
}
```

在下面的範例中，我們在頂層序列化 `Project` 類別，因此我們如 `Passing a serializer manually` 部分中所示，顯式將上

述 `ProjectSerializer` 傳遞給 `Json.encodeToString` 函數：

```
fun main() {
    val data = Project("kotlinx.serialization", "Kotlin")
    println(Json.encodeToString(data)) // 使用插件生成的序列化器
    println(Json.encodeToString(ProjectSerializer, data)) // 使用自定義序列化器
}
```

查看自定義序列化器的效果：

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

基於內容的多型性反序列化

通常，多型性序列化需要在輸入的 JSON 對象中有一個專用的 "type" 鍵（也稱為類別區分符）來確定應該使用的實際序列化器來反序列化 Kotlin 類別。

但是，有時候輸入中可能沒有類型屬性。在這種情況下，你需要通過 JSON 的形狀來猜測實際類型，例如通過某個特定鍵的存在。

`JsonContentPolymorphicSerializer` 提供了此類策略的骨架實作。要使用它，請重寫其 `selectDeserializer` 方法。我們先從以下類別繼承層次結構開始。

請注意，這不必像在 `Sealed classes` 部分中推薦的那樣是密封的，因為我們不打算利用自動選擇相應子類別的插件生成代碼，而是打算手動實作此代碼。

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

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

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

你可以通過 JSON 對象中是否存在 `owner` 鍵來區分 `BasicProject` 和 `OwnedProject` 子類。

```
object ProjectSerializer : JsonContentPolymorphicSerializer<Project>(Project::class) {
    override fun selectDeserializer(element: JsonElement) = when {
        "owner" in element.jsonObject -> OwnedProject.serializer()
        else -> BasicProject.serializer()
    }
}
```

當你使用此序列化器來序列化資料時，會在運行時為實際類型選擇已註冊的或預設的序列化器：

```
fun main() {
    val data = listOf(
        OwnedProject("kotlinx.serialization", "kotlin"),
        BasicProject("example")
    )
    val string = Json.encodeToString(ListSerializer(ProjectSerializer), data)
    println(string)
    println(Json.decodeFromString(ListSerializer(ProjectSerializer), string))
}
```

在 JSON 輸出中不會添加類別區分符：

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

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

底層實作（實驗性功能）

儘管上述抽象序列化器可以涵蓋大多數情況，但可以僅使用 `KSerializer` 類別手動實作類似的機制。如果修

改 `transformSerialize` / `transformDeserialize` / `selectDeserializer` 抽象方法還不夠，那麼可以更改 `serialize` / `deserialize` 方法。

以下是一些有關使用 `Json` 的自定義序列化器的有用信息：

- 如果當前格式是 `Json`，則可以將 `Encoder` 和 `Decoder` 分別轉換為 `JsonEncoder` 和 `JsonDecoder`。
- `JsonDecoder` 有 `decodeJsonElement` 方法，而 `JsonEncoder` 有 `encodeJsonElement` 方法，這些方法基本上將元素從流中的當前位置檢索或插入元素。
- `JsonDecoder` 和 `JsonEncoder` 都有 `json` 屬性，它返回當前使用的 `Json` 實例及其所有設置。
- `Json` 有 `encodeToJsonElement` 和 `decodeFromJsonElement` 方法。

有了這些，就可以實作兩階段轉換 `Decoder -> JsonElement -> value` 或 `value -> JsonElement -> Encoder`。例如，你可以為以下 `Response` 類別實作完全自定義的序列化器，這樣它的 `Ok` 子類可以直接表示，而 `Error` 子類則通過帶有錯誤訊息的對象表示：

```
@Serializable(with = ResponseSerializer::class)
sealed class Response<out T> {
    data class Ok<out T>(val data: T) : Response<T>()
    data class Error(val message: String) : Response<Nothing>()
}

class ResponseSerializer<T>(private val dataSerializer: KSerializer<T>) : KSerializer<Response<T>> {
    override val descriptor: SerialDescriptor = buildSerialDescriptor("Response", PolymorphicKind.SEALED) {
        element("Ok", dataSerializer.descriptor)
        element("Error", buildClassSerialDescriptor("Error") {
            element<String>("message")
        })
    }

    override fun deserialize(decoder: Decoder): Response<T> {
        // Decoder -> JsonDecoder
        require(decoder is JsonDecoder) // 此類別只能由 JSON 解碼
        // JsonDecoder -> JsonElement
        val element = decoder.decodeJsonElement()
        // JsonElement -> value
        if (element is JsonObject && "error" in element)
            return Response.Error(element["error"]!!.jsonPrimitive.content)
        return Response.Ok(decoder.json.decodeFromJsonElement(dataSerializer, element))
    }

    override fun serialize(encoder: Encoder, value: Response<T>) {
        // Encoder -> JsonEncoder
        require(encoder is JsonEncoder) // 此類別只能由 JSON 編碼
        // value -> JsonElement
        val element = when (value) {
            is Response.Ok -> encoder.json.encodeToJsonElement(dataSerializer, value.data)
            is
                Response.Error -> buildJsonObject { put("error", value.message) }
        }
        // JsonElement -> JsonEncoder
        encoder.encodeJsonElement(element)
    }
}
```

有了這個可序列化的 `Response` 實作，你可以為其資料使用任何可序列化的有效載荷，並序列化或反序列化相應的響應：

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

fun main() {
    val responses = listOf(
        Response.Ok(Project("kotlinx.serialization")),
        Response.Error("Not found")
    )
    val string = Json.encodeToString(responses)
    println(string)
    println(Json.decodeFromString<List<Response<Project>>>(string))
}
```

這樣可以精細地控制 `Response` 類別在 JSON 輸出中的表示形式：

```
[{"name":"kotlinx.serialization"}, {"error":"Not found"}]
```

```
[Ok(data=Project(name=kotlinx.serialization)), Error(message=Not found)]
```

維護自定義的 JSON 屬性

一個很好的自定義 JSON 特定序列化器範例是一個解包所有未知 JSON 屬性到 `JsonObject` 類別的專用字段中的反序列化器。

讓我們添加 `UnknownProject` – 一個具有 `name` 屬性和任意細節的類別，將它們展平到相同的對象中：

```
data class UnknownProject(val name: String, val details: JsonObject)
```

但是，默認的插件生成的序列化器要求 `details` 是一個單獨的 JSON 對象，而這不是我們想要的。

為了緩解這個問題，編寫一個自己的序列化器，它使用事實來處理 JSON 格式：

```
object UnknownProjectSerializer : KSerializer<UnknownProject> {
    override val descriptor: SerialDescriptor = buildClassSerialDescriptor("UnknownProject") {
        element<String>("name")
        element<JsonElement>("details")
    }

    override fun deserialize(decoder: Decoder): UnknownProject {
        // 轉換為特定於 JSON 的介面
        val jsonInput = decoder as? JsonDecoder ?: error("Can be deserialized only by JSON")
        // 將整個內容讀取為 JSON
        val json = jsonInput.decodeJsonElement().jsonObject
        // 提取並移除名稱屬性
        val name = json.getValue("name").jsonPrimitive.content
        val details = json.toMutableMap()
        details.remove("name")
        return UnknownProject(name, JsonObject(details))
    }

    override fun serialize(encoder: Encoder, value: UnknownProject) {
        error("Serialization is not supported")
    }
}
```

現在可以用來讀取展平的 JSON 詳細資料作為 `UnknownProject`：

```
fun main() {
    println(Json.decodeFromString(UnknownProjectSerializer, ""
    {"type":"unknown","name":"example","maintainer":"Unknown","license":"Apache 2.0"}""))
}
```

```
UnknownProject(name=example, details={"type":"unknown","maintainer":"Unknown","license":"Apache 2.0"})
```

下一章將涵蓋替代和自定義格式（實驗性功能）。

🔄 修訂版本 #1

★ 由 treeman 建立於 26 🕒 2024 18:24:14

✍ 由 treeman 更新於 26 🕒 2024 18:24:39