【Kotlin】Gson 使用指南

Gson 使用指南

概覽

Gson 是一個 Java 函式庫，可以用來將 Java 物件轉換成其 JSON 表示形式，也可以用來將 JSON 字串轉換成等效的 Java 物件。

Gson 可以處理任意的 Java 物件，包括您無法取得原始碼的既有物件。

Gson 的目標

- 提供簡單易用的機制，例如 `toString()` 和構造器（工廠方法），以在 Java 和 JSON 之間進行轉換。
- 允許既有的不可修改的物件轉換為 JSON，或從 JSON 轉換回來。
- 允許物件的自定義表示形式。
- 支援任意複雜的物件。
- 產生簡潔且可讀的 JSON 輸出。

Gson 的效能和擴展性

以下是我們在一台桌面電腦（雙 Opteron 處理器，8GB RAM，64 位元 Ubuntu 系統）上進行多項測試時取得的一些效能指標。您可以使用類別 `PerformanceTest` 來重新執行這些測試。

- 字串：反序列化超過 25MB 的字串沒有任何問題（參見 `PerformanceTest` 中的 `disabled_testStringDeserializationPerformance` 方法）
- 大型集合：
  - 序列化了一個包含 140 萬個物件的集合（參見 `PerformanceTest` 中的 `disabled_testLargeCollectionSerialization` 方法）
  - 反序列化了一個包含 87,000 個物件的集合（參見 `PerformanceTest` 中的 `disabled_testLargeCollectionDeserialization` 方法）

- Gson 1.4 將位元組陣列和集合的反序列化限制從 80KB 提升到超過 11MB。

注意：要執行這些測試，請刪除 `disabled_` 前綴。我們使用這個前綴來防止每次運行 JUnit 測試時都執行這些測試。

Gson 使用者

Gson 最初是為了在 Google 內部使用而創建的，現在它在 Google 的許多專案中都有使用。它現在也被許多公共專案和公司使用。

使用 Gson

主要使用的類別是 `Gson`，您只需通過呼叫 `new Gson()` 來創建它。此外，還有一個類別 `GsonBuilder`，可用於創建具有各種設定（例如版本控制等）的 Gson 實例。

在調用 JSON 操作時，Gson 實例不會維護任何狀態。因此，您可以自由地重複使用相同的物件進行多次 JSON 序列化和反序列化操作。

在 Gradle/Android 中使用 Gson

```
dependencies {
   implementation 'com.google.code.gson:gson:2.11.0'
}
```

在 Maven 中使用 Gson

要在 Maven2/3 中使用 Gson，您可以通過添加以下依賴項來使用 Maven 中央庫中提供的 Gson 版本：

```
<dependencies>
   <!--  Gson: Java to JSON conversion -->
   <dependency>
     <groupId>com.google.code.gson</groupId>
     <artifactId>gson</artifactId>
     <version>2.11.0</version>
     <scope>compile</scope>
   </dependency>
</dependencies>
```

這樣您的 Maven 專案就可以使用 Gson 了。

基本類型範例

```
// 序列化
val gson = Gson()
gson.toJson(1)          // ==> 1
gson.toJson("abcd")     // ==> "abcd"
gson.toJson(10L)        // ==> 10
val values = intArrayOf(1)
gson.toJson(values)     // ==> [1]

// 反序列化
val i = gson.fromJson("1", Int::class.java)
val intObj = gson.fromJson("1", Int::class.javaObjectType)
val longObj = gson.fromJson("1", Long::class.javaObjectType)
val boolObj = gson.fromJson("false", Boolean::class.javaObjectType)
val str = gson.fromJson("\"abc\"", String::class.java)
val strArray = gson.fromJson("[\"abc\"]", Array<String>::class.java)
```

物件範例

```
class BagOfPrimitives {
    private val value1 = 1
    private val value2 = "abc"
    private val value3 = 3 // 被標記為 transient 但不需要標記

    // 無參數構造器，Kotlin 默認會有
}

// 序列化
val obj = BagOfPrimitives()
val gson = Gson()
val json = gson.toJson(obj)
// ==> {"value1":1,"value2":"abc"}
```

注意，您無法序列化具有循環引用的物件，因為這會導致無限遞歸。

```
// 反序列化
val obj2 = gson.fromJson(json, BagOfPrimitives::class.java)
// ==> obj2 和 obj 一樣
```

物件的細節

- 使用 private 欄位是完全可以的（並且建議這樣做）。
- 不需要使用任何註解來指示欄位是否應

該包含在序列化和反序列化中。當前類別中的所有欄位（以及所有超類別中的欄位）默認都會被包含。

- 如果欄位被標記為 transient，（默認情況下）它將被忽略，不包含在 JSON 序列化或反序列化中。
- 這個實現正確地處理 null 值。
  - 在序列化時，null 欄位會從輸出中省略。
  - 在反序列化時，JSON 中缺少的條目會導致將對應的物件欄位設置為其默認值：對於物件類型為 null，對於數值類型為零，對於布林值為 false。
- 如果欄位是 *synthetic* 的，它將被忽略，不包含在 JSON 序列化或反序列化中。
- 內部類別中對應於外部類別的欄位將被忽略，不包含在序列化或反序列化中。
- 匿名和本地類別將被排除。它們將作為 JSON `null` 被序列化，而在反序列化時，它們的 JSON 值將被忽略，返回 `null`。要啟用序列化和反序列化，請將這些類別轉換為 `static` 的嵌套類別。

巢狀類別（包括內部類別）

Gson 可以輕鬆地序列化靜態巢狀類別。

Gson 也可以反序列化靜態巢狀類別。然而，Gson 無法自動反序列化 **純內部類別，因為它們的無參數構造器還需要一個參照包含物件的引用**，而在反序列化時這個引用並不存在。您可以通過將內部類別設為靜態類別，或為其提供一個自定義的 InstanceCreator 來解決這個問題。以下是範例：

```
class A {
    var a: String? = null

    inner class B {
        var b: String? = null

        constructor() {
            // 無參數構造器
        }
    }
}
```

注意：上述類別 B 無法（默認情況下）用 Gson 進行序列化。

Gson 無法將 `{"b":"abc"}` 反序列化為 B 的實例，因為 B 類別是內部類別。如果它被定義為靜態類別 B，那麼 Gson 就能夠反序列化該字串。另一個解決方案是為 B 撰寫自定義的實例創建器。

```
class InstanceCreatorForB(private val a: A) : InstanceCreator<A.B> {
    override fun createInstance(type: Type): A.B {
        return a.B()
    }
}
```

上述方法是可行的，但並不推薦使用。

陣列範例

```
val gson = Gson()
val ints = intArrayOf(1, 2, 3, 4, 5)
val strings = arrayOf("abc", "def", "ghi")

// 序列化
gson.toJson(ints)     // ==> [1,2,3,4,5]
gson.toJson(strings)  // ==> ["abc", "def", "ghi"]

// 反序列化
val ints2 = gson.fromJson("[1,2,3,4,5]", IntArray::class.java)
// ==> ints2 將和 ints 相同
```

我們還支援多維陣列，具有任意複雜的元素類型。

集合範例

```
val gson = Gson()
val ints: Collection<Int> = listOf(1, 2, 3, 4, 5)

// 序列化
val json = gson.toJson(ints)  // ==> [1,2,3,4,5]

// 反序列化
val collectionType = object : TypeToken<Collection<Int>>() {}.type
val ints2: Collection<Int> = gson.fromJson(json, collectionType)
// ==> ints2 和 ints 相同
```

這種方法比較麻煩：請注意我們如何定義集合的類型。 遺憾的是，在 Java 中無法避免這個問題。

集合的限制

Gson 可以序列化任意物件的集合，但無法從中反序列化，因為無法讓使用者指示結果物件的類型。相反，在反序列化時，集合必須是特定的泛型類型。 這是合理的，在遵循良好的 Java 編碼實踐時，這很少會是個問題。

Map 範例

Gson 預設將任何 `java.util.Map` 實作序列化為 JSON 物件。由於 JSON 物件僅支援字串作為成員名稱，Gson 通過呼叫 `toString()` 將 Map 的鍵轉換為字串，並對於 `null` 鍵使用 `"null"`：

```
val gson = Gson()
val stringMap = linkedMapOf("key" to "value", null to "null-entry")

// 序列化
val json = gson.toJson(stringMap) // ==> {"key":"value","null":"null-entry"}

val intMap = linkedMapOf(2 to 4, 3 to 6)

// 序列化
val json2 = gson.toJson(intMap) // ==> {"2":4,"3":6}
```

在反序列化時，Gson 使用為 Map 鍵類型註冊的 `TypeAdapter` 的 `read` 方法。與上面顯示的集合範例類似，反序列化時必須使用 `TypeToken` 來告訴 Gson Map 鍵和值的類型：

```
val gson = Gson()
val mapType = object : TypeToken<Map<String, String>>() {}.type
val json = "{\"key\": \"value\"}"

// 反序列化
val stringMap: Map<String, String> = gson.fromJson(json, mapType)
// ==> stringMap 是 {key=value}
```

Gson 還支援使用複雜類型作為 Map 鍵。這個功能可以通過 `GsonBuilder.enableComplexMapKeySerialization()` 啟用。如果啟用，Gson 使用為 Map 鍵類型註冊的 `TypeAdapter` 的 `write` 方法來序列化鍵，而不是使用 `toString()`。當任何鍵被適配器序列化為

JSON 陣列或 JSON 物件時，Gson 將整個 Map 序列化為 JSON 陣列，由鍵值對（編碼為 JSON 陣列）組成。否則，如果沒有鍵被序列化為 JSON 陣列或 JSON 物件，Gson 將使用 JSON 物件來編碼 Map：

```
data class PersonName(val firstName: String, val lastName: String)

val gson = GsonBuilder().enableComplexMapKeySerialization().create()
val complexMap = linkedMapOf(PersonName("John", "Doe") to 30, PersonName("Jane", "Doe") to 35)

// 序列化；複雜的 Map 被序列化為一個 JSON 陣列，包含鍵值對（作為 JSON 陣列）
val json = gson.toJson(complexMap)
// ==> [[{"firstName":"John","lastName":"Doe"},30],[{"firstName":"Jane","lastName":"Doe"},35]]

val stringMap = linkedMapOf("key" to "value")
// 序列化；非複雜 Map 被序列化為一個常規 JSON 物件
val json2 = gson.toJson(stringMap) // ==> {"key":"value"}
```

重要提示： 因為 Gson 默認使用 `toString()` 來序列化 Map 鍵，這可能會導致鍵的編碼不正確或在序列化和反序列化之間產生不匹配，例如當 `toString()` 沒有正確實作時。可以使用 `enableComplexMapKeySerialization()` 作為解決方法，確保 `TypeAdapter` 註冊在 Map 鍵類型中被用於反序列化和序列化。如上例所示，當沒有鍵被適配器序列化為 JSON 陣列或 JSON 物件時，Map 將被序列化為常規 JSON 物件，這是期望的結果。

注意，當反序列化枚舉類型作為 Map 鍵時，如果 Gson 無法找到與相應 `name()` 值（或 `@SerializedName` 註解）匹配的枚舉常數，它會退回到通過 `toString()` 值查找枚

舉常數。這是為了解決上述問題，但僅適用於枚舉常數。

序列化和反序列化泛型類型

當您呼叫 `toJson(obj)` 時，Gson 呼叫 `obj.getClass()` 來獲取要序列化的欄位資訊。同樣，您通常可以在 `fromJson(json, MyClass.class)` 方法中傳入 `MyClass.class` 物件。這對於非泛型類型的物件來說效果很好。然而，如果物件是泛型類型，那麼由於 Java 類型擦除，泛型類型資訊就會丟失。以下是說明這一點的範例：

```
class Foo<T>(var value: T)

val gson = Gson()
val foo = Foo(Bar())
gson.toJson(foo) // 可能無法正確序列化 foo.value

gson.fromJson<Foo<Bar>>(json, foo::class.java) // 無法將 foo.value 反序列化為 Bar
```

上述代碼無法將 value 解釋為 Bar 類型，因為 Gson 調用 `foo::class.java` 來獲取其類別資訊，但此方法返回原始類別，`Foo::class.java`。這意味著 Gson 無法知道這是一個 `Foo<Bar>` 類型的物件，而不僅僅是普通的 `Foo`。

您可以通過為您的泛型類型指定正確的參數化類型來解決這個問題。您可以使用 `TypeToken` 類別來實現這一點。

```
val fooType = object : TypeToken<Foo<Bar>>() {}.type
gson.toJson(foo, fooType)

gson.fromJson<Foo<Bar>>(json, fooType)
```

用於獲取 `fooType` 的慣用方法實際上定義了一個匿名的本地內部類別，包含一個 `getType()` 方法，該方法返回完整的參數化類型。

序列化和反序列化包含任意類型物件的集合

有時候您會處理包含混合類型的 JSON 陣列。例如：`['hello',5,{name:'GREETINGS',source:'guest'}]`

等效的 `Collection` 包含以下內容：

```
val collection = mutableListOf<Any>("hello", 5, Event("GREETINGS", "guest"))
```

其中 `Event` 類別定義如下：

```
data class Event(val name: String, val source: String)
```

您可以使用 Gson 序列化這個集合而不需要做任何特殊處理：`toJson(collection)` 會輸出所需的結果。

然而，使用 `fromJson(json, Collection::class.java)` 進行反序列化是不會成功的，因為 Gson 無法知道如何將輸入映射到類型。Gson 要求

您在 `fromJson()` 中提供集合類型的泛型版本。因此，您有三個選擇：

1. 使用 Gson 的解析 API（低階串流解析器或 DOM 解析器 JsonParser）來解析陣列元素，然後對每個陣列元素使用 `Gson.fromJson()`。這是首選方法。這裡有一個範例 展示了如何做到這一點。
2. 為 `Collection::class.java` 註冊一個型別適配器，該適配器檢查每個陣列成員並將它們映射到適當的物件。這種方法的缺點是它會搞亂 Gson 中其他集合類型的反序列化。
3. 為 `MyCollectionMemberType` 註冊一個型別適配器，並使用 `fromJson()` 與 `Collection<MyCollectionMemberType>`。

此方法僅在陣列作為頂層元素出現時適用，或者您可以更改保存集合的欄位類型為 `Collection<MyCollectionMemberType>`。

內建的序列化器和反序列化器

Gson 內建了常用類別的序列化器和反序列化器，這些類別的默認表示方式可能不合適，例如：

- `java.net.URL` 以匹配 `"https://github.com/google/gson/"` 這樣的字串
- `java.net.URI` 以匹配 `"/google/gson/"` 這樣的字串

更多資訊，請參見內部類別 `TypeAdapters`。

您也可以在此頁面找到一些常用類別（如 JodaTime）的原始碼。

自定義序列化和反序列化

有時候默認的表示方式不是您想要的。這種情況通常發生在處理庫類別（例如 DateTime 等）時。 Gson 允許您註冊自己的自定義序列化器和反序列化器。這是通過定義兩個部分來完成的：

- JSON 序列化器：需要為物件定義自定義序列化
- JSON 反序列化器：需要為類型定義自定義反序列化
- 實例創建器：如果有無參數構造器可用或註冊了一個反序列化器，則不需要

```
val gsonBuilder = GsonBuilder()
gsonBuilder.registerTypeAdapter(MyType2::class.java, MyTypeAdapter())
gsonBuilder.registerTypeAdapter(MyType::class.java, MySerializer())
gsonBuilder.registerTypeAdapter(MyType::class.java, MyDeserializer())
gsonBuilder.registerTypeAdapter(MyType::class.java, MyInstanceCreator())
```

`registerTypeAdapter` 呼叫檢查：

1. 如果型別適配器實作了這些介面中的多個，則會為所有這些介面註冊該適配器。
2. 如果型別適配器適用於 Object 類別或 JsonElement 或其任何子類別，則會拋出 IllegalArgumentException，因為不支援覆蓋這些類型的內建適配器。

撰寫序列化器

以下是一個撰寫 JodaTime `DateTime` 類別自定義序列化器的範例。

```
private class DateTimeSerializer : JsonSerializer<DateTime> {
   override fun serialize(src: DateTime?, typeOfSrc: Type, context: JsonSerializationContext): JsonElement {
      return JsonPrimitive(src.toString())
   }
}
```

Gson 在序列化期間遇到 `DateTime` 物件時會呼叫 `serialize()`。

撰寫反序列化器

以下是一個撰寫 JodaTime DateTime 類別自定義反序列化器的範例。

```
private class DateTimeDeserializer : JsonDeserializer<DateTime> {
   override fun deserialize(json: JsonElement, typeOfT: Type, context: JsonDeserializationContext): DateTime {
      return DateTime(json.asJsonPrimitive.asString)
   }
}
```

Gson 在需要將 JSON 字串片段反序列化為 DateTime 物件時會呼叫 `deserialize`。

關於序列化器和反序列化器的細節

通常您希望為所有對應於原始類型的泛型類型註冊一個處理程序

- 例如，假設您有一個 `Id` 類別，用於 ID 的表示/轉換（即內部 vs. 外部表示）。
- `Id<T>` 類型對所有泛型類型具有相同的序列化
  - 實質上是寫出 ID 值
- 反序列化非常相似，但不完全相同
  - 需要呼叫 `new Id(Class<T>, String)`，它返回一個 `Id<T>` 的實例

Gson 支援為此註冊一個處理程序。您還可以為特定泛型類型（例如需要特殊處理的 `Id<RequiresSpecialHandling>`）註冊一個特定的處理程序。 `toJson()` 和 `fromJson()` 的 `Type` 參數包含泛型類型資訊，幫助您為所有對應的泛型類型撰寫單個處理程序。

撰寫實例創建器

在反序列化物件時，Gson 需要創建類別的預設實例。 行為良好的類別（用於序列化和反序列化的）應該有一個無參數構造器。

- 無論是 public 還是 private 都無所謂

通常，當您處理的庫類別沒有定義無參數構造器時，需要實例創建器。

實例創建器範例

```
private class MoneyInstanceCreator : InstanceCreator<Money> {
  override fun createInstance(type: Type): Money {
    return Money("1000000", CurrencyCode.USD)
  }
}
```

Type 可能是相應泛型類型的。

- 非常有用，可以呼叫需要特定泛型類型資訊的構造器
- 例如，如果 `Id` 類別存儲了創建 ID 的類別

Parameterized 類型的 InstanceCreator

有時候您要實例化的類型是參數化類型。一般來說，這不是問題，因為實際的實例是原始類型。以下是一個範例：

```
class MyList<T> : ArrayList<T>()

class MyListInstanceCreator : InstanceCreator<MyList<*>> {
  override fun createInstance(type: Type): MyList<*> {
    // 不需要使用參數化列表，因為實際實例無論如何都將具有原始類型。
    return MyList<Any?>()
  }
}
```

然而，有時候您確實需要根據實際的參數化類型創建實例。在這種情況下，您可以使用傳遞給 `createInstance` 方法的類型參數。以下是一個範例：

```
class Id<T>(private val classOfId: Class<T>, private val value: Long)

class IdInstanceCreator : InstanceCreator<Id<*>> {
  override fun createInstance(type: Type): Id<*> {
    val typeParameters = (type as ParameterizedType).actualTypeArguments
    val idType = typeParameters[0] as Class<*> // Id 只有一個參數化類型 T
    return Id(idType, 0L)
  }
}
```

在上述範例中，如果不傳入參數化類型的實際類型，就無法創建 Id 類別的實例。我們通過使用傳遞的參數 `type` 解決了這個問題。在這個例子中，`type` 物件是 `Id<Foo>` 的 Java 參數化類型表示，其中實際的實例應綁定為 `Id<Foo>`。由於 `Id` 類別只有一個參數化類型參數 `T`，我們使用 `getActualTypeArgument()` 返回的類型數組的第零個元素，它在這種情況下將持有 `Foo.class`。

JSON 輸出格式的壓縮與漂亮印出

Gson 提供的預設 JSON 輸出格式是一種緊湊的 JSON 格式。這意味著在輸出 JSON 結構中不會有任何空白。因此，在 JSON 輸出中，欄位名稱與其值、物件欄位以及陣列中的物件之間不會有空白。另外，"null" 欄位將在輸出中被忽略（注意：在物件集合/陣列中，null 值仍然會被包括）。請參見 Null 物件支援 部分，了解如何配置 Gson 以輸出所有 null 值。

如果您想使用漂亮印出功能，必須使用 `GsonBuilder` 來配置您的 `Gson` 實例。`JsonFormatter` 並未通過我們的公開 API 曝露，因此用戶

端無法配置 JSON 輸出的默認打印設定/邊距。目前，我們僅提供一個默認的 `JsonPrintFormatter`，該格式器具有 80 字元的默認行長、2 字元的縮排和 4 字元的右邊距。

以下範例顯示了如何配置 `Gson` 實例以使用默認的 `JsonPrintFormatter` 而非 `JsonCompactFormatter`：

```
val gson = GsonBuilder().setPrettyPrinting().create()
val jsonOutput = gson.toJson(someObject)
```

Null 物件支援

Gson 實作的預設行為是忽略 `null` 物件欄位。這允許更緊湊的輸出格式；然而，當 JSON 格式轉換回其 Java 形式時，用戶端必須為這些欄位定義一個默認值。

以下是如何配置 `Gson` 實例以輸出 null 的範例：

```
val gson = GsonBuilder().serializeNulls().create()
```

注意：使用 Gson 序列化 `null` 時，它會向 `JsonElement` 結構添加一個 `JsonNull` 元素。因此，這個物件可以在自定義序列化/反序列化中使用。

以下是一個範例：

```
data class Foo(val s: String? = null, val i: Int = 5)

val gson = GsonBuilder().serializeNulls().create()
val foo = Foo()
var json = gson.toJson(foo)
println(json)

json = gson.toJson(null)
println(json)
```

輸出是：

```
{"s":null,"i":5}
null
```

版本支援

可以使用 @Since 註解來維護同一物件的多個版本。此註解可以用於類別、欄位，未來版本中也可用於方法。為了利用此功能，您必須配置您的 `Gson` 實例以忽略任何高於某個版本號的欄位/物件。如果在 `Gson` 實例上未設定版本，則它將序列化和反序列化所有欄位和類別，而不考慮版本。

```
data class VersionedClass(
    @Since(1.1) val newerField: String = "newer",
    @Since(1.0) val newField: String = "new",
    val field: String = "old"
)

val versionedObject = VersionedClass()
var gson = GsonBuilder().setVersion(1.0).create()
var jsonOutput = gson.toJson(versionedObject)
println(jsonOutput)
println()

gson = Gson()
jsonOutput = gson.toJson(versionedObject)
println(jsonOutput)
```

輸出是：

```
{"newField":"new","field":"old"}

{"newerField":"newer","newField":"new","field":"old"}
```

從序列化和反序列化中排除欄位

Gson 支援多種機制來排除頂層類別、欄位和欄位類型。下面是可插入的機制，允許排除欄位和類別。如果下列機制無法滿足您的需求，您還可以使用自定義序列化器和反序列化器。

Java 修飾符排除

默認情況下，如果您將欄位標記為 `transient`，則該欄位將被排除。同樣，如果欄位標記為 `static`，則默認情況下也會被排除。如果您想包括某些 transient 欄位，可以按以下方式進行操作：

```
import java.lang.reflect.Modifier

val gson = GsonBuilder()
    .excludeFieldsWithModifiers(Modifier.STATIC)
    .create()
```

注意：您可以向 `excludeFieldsWithModifiers` 方法提供任意數量的 `Modifier` 常數。例如：

```
val gson = GsonBuilder()
    .excludeFieldsWithModifiers(Modifier.STATIC, Modifier.TRANSIENT, Modifier.VOLATILE)
    .create()
```

Gson 的 `@Expose`

此功能提供了一種方法，您可以標記物件的某些欄位，以排除它們不被考慮進行序列化和反序列化為 JSON。要使用此註解，您必須通過使用 `new GsonBuilder().excludeFieldsWithoutExposeAnnotation().create()` 創建 Gson。創建的 Gson 實例將排除類別中未標記 `@Expose` 註解的所有欄位。

使用者定義的排除策略

如果上述排除欄位和類別類型的機制對您不起作用，則可以撰寫自己的排除策略並將其插入到 Gson 中。詳細資訊請參見 `ExclusionStrategy` JavaDoc。

以下範例展示了如何排除具有特定 `@Foo` 註解的欄位，並排除類別 `String` 的頂層類型（或聲明的欄位類型）。

```
@Retention(AnnotationRetention.RUNTIME)
@Target(AnnotationTarget.FIELD)
annotation class Foo

data class SampleObjectForTest(
    @Foo val annotatedField:

 Int = 5,
    val stringField: String = "someDefaultValue",
    val longField: Long = 1234L
)

class MyExclusionStrategy(private val typeToSkip: Class<*>) : ExclusionStrategy {
    override fun shouldSkipClass(clazz: Class<*>): Boolean {
        return clazz == typeToSkip
    }

    override fun shouldSkipField(f: FieldAttributes): Boolean {
        return f.getAnnotation(Foo::class.java) != null
    }
}

fun main() {
    val gson = GsonBuilder()
        .setExclusionStrategies(MyExclusionStrategy(String::class.java))
        .serializeNulls()
        .create()
    val src = SampleObjectForTest()
    val json = gson.toJson(src)
    println(json)
}
```

輸出是：

```
{"longField":1234}
```

JSON 欄位命名支援

Gson 支援一些預定義的欄位命名策略，用於將標準 Java 欄位名稱（即，以小寫字母開頭的駝峰式命名，例如 `sampleFieldNameInJava`）轉換為 JSON 欄位名稱（例如，`sample_field_name_in_java` 或 `SampleFieldNameInJava`）。有關預定義命名策略的資訊，請參見 FieldNamingPolicy 類別。

它還有基於註解的策略，允許客戶端根據每個欄位定義自定義名稱。請注意，基於註解的策略具有欄位名稱驗證功能，如果提供的欄位名稱無效，將引發「執行時」異常。

以下是如何使用 Gson 命名策略功能的範例：

```
data class SomeObject(
    @SerializedName("custom_naming") val someField: String,
    val someOtherField: String
)

val someObject = SomeObject("first", "second")
val gson = GsonBuilder().setFieldNamingPolicy(FieldNamingPolicy.UPPER_CAMEL_CASE).create()
val jsonRepresentation = gson.toJson(someObject)
println(jsonRepresentation)
```

輸出是：

```
{"custom_naming":"first","SomeOtherField":"second"}
```

如果您需要自定義命名策略（請參見此討論），您可以使用 @SerializedName 註解。

在自定義序列化器和反序列化器之間共享狀態

有時候您需要在自定義序列化器/反序列化器之間共享狀態（請參見此討論）。您可以使用以下三種策略來實現：

1. 將共享狀態存儲在靜態欄位中
2. 將序列化器/反序列化器聲明為父類型的內部類別，並使用父類型的實例欄位來存儲共享狀態
3. 使用 Java `ThreadLocal`

1 和 2 都不是線程安全的選項，但 3 是。

串流

除了 Gson 的物件模型和數據綁定外，您還可以使用 Gson 來讀取和寫入串流。您還可以結合串流和物件模型訪問，以獲得兩種方法的最佳效果。

設計 Gson 時遇到的問題

請參見 Gson 設計文檔，了解我們在設計 Gson 時面臨的問題討論。它還包括 Gson 與其他可用於 JSON 轉換的 Java 函式庫的比較。

Gson 的未來增強

欲了解最新的增強提議列表，或如果您想建議新功能，請參見項目網站下的 Issues 部分。

```
/*
 * Copyright (C) 2008 Google Inc.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
```

```
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package com.google.gson.metrics;

import static com.google.common.truth.Truth.assertThat;

import com.google.gson.Gson;
import com.google.gson.JsonParseException;
import com.google.gson.annotations.Expose;
import com.google.gson.reflect.TypeToken;
import java.io.StringWriter;
import java.lang.reflect.Type;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import org.junit.Before;
import org.junit.Ignore;
import org.junit.Test;

/**
 * Tests to measure performance for Gson. All tests in this file will be disabled in code. To run
 * them remove the {@code @Ignore} annotation from the tests.
 *
 * @author Inderjeet Singh
 * @author Joel Leitch
 */
@SuppressWarnings("SystemOut") // allow System.out because test is for manual execution anyway
public class PerformanceTest {
  private static final int COLLECTION_SIZE = 5000;

  private static final int NUM_ITERATIONS = 100;

  private Gson gson;

  @Before
  public void setUp() throws Exception {
    gson = new Gson();
  }

  @Test
  public void testDummy() {
    // This is here to prevent Junit for complaining when we disable all tests.
  }

  @Test
  @Ignore
  public void testStringDeserialization() {
    StringBuilder sb = new StringBuilder(8096);
    sb.append("Error Yippie");

    while (true) {
      try {
        String stackTrace = sb.toString();
        sb.append(stackTrace);
        String json = "{\"message\":\"Error message.\"," + "\"stackTrace\":\"" + stackTrace + "\"}";
        parseLongJson(json);
        System.out.println("Gson could handle a string of size: " + stackTrace.length());
      } catch (JsonParseException expected) {
        break;
      }
    }
  }
```

```
private void parseLongJson(String json) throws JsonParseException {
  ExceptionHolder target = gson.fromJson(json, ExceptionHolder.class);
  assertThat(target.message).contains("Error");
  assertThat(target.stackTrace).contains("Yippie");
}

private static class ExceptionHolder {
  public final String message;
  public final String stackTrace;

  // For use by Gson
  @SuppressWarnings("unused")
  private ExceptionHolder() {
    this("", "");
  }

  public ExceptionHolder(String message, String stackTrace) {
    this.message = message;
    this.stackTrace = stackTrace;
  }
}

@SuppressWarnings("unused")
private static class CollectionEntry {
  final String name;
  final String value;

  // For use by Gson
  private CollectionEntry() {
    this(null, null);
  }

  CollectionEntry(String name, String value) {
    this.name = name;
    this.value = value;
  }
}

/** Created in response to http://code.google.com/p/google-gson/issues/detail?id=96 */
@Test
@Ignore
public void testLargeCollectionSerialization() {
  int count = 1400000;
  List<CollectionEntry> list = new ArrayList<>(count);
  for (int i = 0; i < count; ++i) {
    list.add(new CollectionEntry("name" + i, "value" + i));
  }
  String unused = gson.toJson(list);
}

/** Created in response to http://code.google.com/p/google-gson/issues/detail?id=96 */
@Test
@Ignore
public void testLargeCollectionDeserialization() {
  StringBuilder sb = new StringBuilder();
  int count = 87000;
  boolean first = true;
  sb.append('[');
  for (int i = 0; i < count; ++i) {
    if (first) {
      first = false;
    } else {
      sb.append(',');
    }
    sb.append("{name:'name").append(i).append("',value:'value").append(i).append("'}");
  }
  sb.append(']');
```

```
  String json = sb.toString();
  Type collectionType = new TypeToken<ArrayList<CollectionEntry>>() {}.getType();
  List<CollectionEntry> list = gson.fromJson(json, collectionType);
  assertThat(list).hasSize(count);
 }

 /** Created in response to http://code.google.com/p/google-gson/issues/detail?id=96 */
 // Last I tested, Gson was able to serialize upto 14MB byte array
 @Test
 @Ignore
 public void testByteArraySerialization() {
  for (int size = 4145152; true; size += 1036288) {
    byte[] ba = new byte[size];
    for (int i = 0; i < size; ++i) {
      ba[i] = 0x05;
    }
    String unused = gson.toJson(ba);
    System.out.printf("Gson could serialize a byte array of size: %d\n", size);
  }
 }

 /** Created in response to http://code.google.com/p/google-gson/issues/detail?id=96 */
 // Last I tested, Gson was able to deserialize a byte array of 11MB
 @Test
 @Ignore
 public void testByteArrayDeserialization() {
  for (int numElements = 10639296; true; numElements += 16384) {
    StringBuilder sb = new StringBuilder(numElements * 2);
    sb.append("[");
    boolean first = true;
    for (int i = 0; i < numElements; ++i) {
      if (first) {
        first = false;
      } else {
        sb.append(",");
      }
      sb.append("5");
    }
    sb.append("]");
    String json = sb.toString();
    byte[] ba = gson.fromJson(json, byte[].class);
    System.out.printf("Gson could deserialize a byte array of size: %d\n", ba.length);
  }
 }

 // The tests to measure serialization and deserialization performance of Gson
 // Based on the discussion at
 // http://groups.google.com/group/google-gson/browse_thread/thread/7a50b17a390dfaeb
 // Test results: 10/19/2009
 // Serialize classes avg time: 60 ms
 // Deserialized classes avg time: 70 ms
 // Serialize exposed classes avg time: 159 ms
 // Deserialized exposed classes avg time: 173 ms

 @Test
 @Ignore
 public void testSerializeClasses() {
  ClassWithList c = new ClassWithList("str");
  for (int i = 0; i < COLLECTION_SIZE; ++i) {
    c.list.add(new ClassWithField("element-" + i));
  }
  StringWriter w = new StringWriter();
  long t1 = System.currentTimeMillis();
  for (int i = 0; i < NUM_ITERATIONS; ++i) {
    gson.toJson(c, w);
  }
  long t2 = System.currentTimeMillis();
  long avg = (t2 - t1) / NUM_ITERATIONS;
  System.out.printf("Serialize classes avg time: %d ms\n", avg);
```

```
}

@Test
@Ignore
public void testDeserializeClasses() {
  String json = buildJsonForClassWithList();
  ClassWithList[] target = new ClassWithList[NUM_ITERATIONS];
  long t1 = System.currentTimeMillis();
  for (int i = 0; i < NUM_ITERATIONS; ++i) {
    target[i] = gson.fromJson(json, ClassWithList.class);
  }
  long t2 = System.currentTimeMillis();
  long avg = (t2 - t1) / NUM_ITERATIONS;
  System.out.printf("Deserialize classes avg time: %d ms\n", avg);
}

@Test
@Ignore
public void testLargeObjectSerializationAndDeserialization() {
  Map<String, Long> largeObject = new HashMap<>();
  for (long l = 0; l < 100000; l++) {
    largeObject.put("field" + l, l);
  }

  long t1 = System.currentTimeMillis();
  String json = gson.toJson(largeObject);
  long t2 = System.currentTimeMillis();
  System.out.printf("Large object serialized in: %d ms\n", (t2 - t1));

  t1 = System.currentTimeMillis();
  Map<String, Long> unused = gson.fromJson(json, new TypeToken<Map<String, Long>>() {}.getType());
  t2 = System.currentTimeMillis();
  System.out.printf("Large object deserialized in: %d ms\n", (t2 - t1));
}

@Test
@Ignore
public void testSerializeExposedClasses() {
  ClassWithListOfObjects c1 = new ClassWithListOfObjects("str");
  for (int i1 = 0; i1 < COLLECTION_SIZE; ++i1) {
    c1.list.add(new ClassWithExposedField("element-" + i1));
  }
  ClassWithListOfObjects c = c1;
  StringWriter w = new StringWriter();
  long t1 = System.currentTimeMillis();
  for (int i = 0; i < NUM_ITERATIONS; ++i) {
    gson.toJson(c, w);
  }
  long t2 = System.currentTimeMillis();
  long avg = (t2 - t1) / NUM_ITERATIONS;
  System.out.printf("Serialize exposed classes avg time: %d ms\n", avg);
}

@Test
@Ignore
public void testDeserializeExposedClasses() {
  String json = buildJsonForClassWithList();
  ClassWithListOfObjects[] target = new ClassWithListOfObjects[NUM_ITERATIONS];
  long t1 = System.currentTimeMillis();
  for (int i = 0; i < NUM_ITERATIONS; ++i) {
    target[i] = gson.fromJson(json, ClassWithListOfObjects.class);
  }
  long t2 = System.currentTimeMillis();
  long avg = (t2 - t1) / NUM_ITERATIONS;
  System.out.printf("Deserialize exposed classes avg time: %d ms\n", avg);
}

@Test
```

```
@Ignore
public void testLargeGsonMapRoundTrip() throws Exception {
  Map<Long, Long> original = new HashMap<>();
  for (long i = 0; i < 1000000; i++) {
    original.put(i, i + 1);
  }

  Gson gson = new Gson();
  String json = gson.toJson(original);
  Type longToLong = new TypeToken<Map<Long, Long>>() {}.getType();
  Map<Long, Long> unused = gson.fromJson(json, longToLong);
}

private static String buildJsonForClassWithList() {
  StringBuilder sb = new StringBuilder("{");
  sb.append("field:").append("'str',");
  sb.append("list:[");
  boolean first = true;
  for (int i = 0; i < COLLECTION_SIZE; ++i) {
    if (first) {
      first = false;
    } else {
      sb.append(',');
    }
    sb.append("{field:'element-" + i + "'}");
  }
  sb.append(']');
  sb.append('}');
  String json = sb.toString();
  return json;
}

@SuppressWarnings("unused")
private static final class ClassWithList {
  final String field;
  final List<ClassWithField> list = new ArrayList<>(COLLECTION_SIZE);

  ClassWithList() {
    this(null);
  }

  ClassWithList(String field) {
    this.field = field;
  }
}

@SuppressWarnings("unused")
private static final class ClassWithField {
  final String field;

  ClassWithField() {
    this("");
  }

  public ClassWithField(String field) {
    this.field = field;
  }
}

@SuppressWarnings("unused")
private static final class ClassWithListOfObjects {
  @Expose final String field;
  @Expose final List<ClassWithExposedField> list = new ArrayList<>(COLLECTION_SIZE);

  ClassWithListOfObjects() {
    this(null);
  }

  ClassWithListOfObjects(String field) {
```

```
      this.field = field;
    }
  }

  @SuppressWarnings("unused")
  private static final class ClassWithExposedField {
    @Expose final String field;

    ClassWithExposedField() {
      this("");
    }

    ClassWithExposedField(String field) {
      this.field = field;
    }
  }
}
```

修訂版本 #1
★由 treeman 建立於 26 ﷺ 2024 18:22:57
✎由 treeman 更新於 26 ﷺ 2024 18:23:39