【Kotlin】kotlinx.serialization Vs Gson

1. 簡介

- **kotlinx.serialization**：介紹它是 Kotlin 官方提供的序列化庫，支持 JSON 和其他格式。
- **Gson**：介紹它是 Google 開發的 Java 序列化庫，用於將 Java 對象轉換為 JSON，反之亦然。

- **範例程式碼：**

```
// kotlinx.serialization example
@Serializable
data class User(val name: String, val age: Int)
```

-

```
// Gson example
data class User(val name: String, val age: Int)
```

- **說明：** `kotlinx.serialization` 使用 `@Serializable` 註解來標記可序列化的類，而 Gson 則不需要特定註解（除非需要特殊處理）。

2. 設計理念

- **kotlinx.serialization**：設計為 Kotlin 專用，強調與 Kotlin 語言特性的緊密集成（例如 `data classes`、`sealed classes`）。
- **Gson**：設計為通用的 Java 庫，支持多種 Java 對象類型。

- **範例程式碼：**

```
// kotlinx.serialization
val jsonStr = Json.encodeToString(User("Alice", 25))
println(jsonStr)  // {"name":"Alice","age":25}


// Gson
val gson = Gson()
val gsonStr = gson.toJson(User("Alice", 25))
println(gsonStr)  // {"name":"Alice","age":25}
```

- **說明：** `kotlinx.serialization` 與 Kotlin 整合更緊密，直接使用 Kotlin 的標準庫方法，而 Gson 則是 Java 標準的物件-JSON 序列化工具。

3. 使用方式

- 介紹如何在兩者中實現基本的 JSON 序列化和反序列化。
- 範例程式碼：比較使用 `@Serializable` 註解和 Gson 的 `@SerializedName` 註解。
- 依賴管理：

-

```
object Kotlin {
 const val serializationJson  = "org.jetbrains.kotlinx:kotlinx-serialization-json:1.7.1"
}

dependencies {
 implementation(Kotlin.serializationJson)
}

plugins {
 kotlin("plugin.serialization") version "2.0.0"  // 添加序列化插件
}
```

-

```
object Libraries {
 const val gson   = "com.google.code.gson:gson:2.10.1"
}

dependencies {
```

```
 implementation(Libraries.gson)
}
```

- **範例程式碼：**

```
// kotlinx.serialization
@Serializable
data class Product(val id: Int, val name: String)

val jsonString = Json.encodeToString(Product(1, "Laptop"))
val product = Json.decodeFromString<Product>(jsonString)
```

- ```
  // Gson
  data class Product(val id: Int, val name: String)

  val gson = Gson()
  val jsonString = gson.toJson(Product(1, "Laptop"))
  val product = gson.fromJson(jsonString, Product::class.java)
  ```

- **簡短說明：** `kotlinx.serialization` 和 Gson 都可以輕鬆地進行物件與 JSON 之間的轉換，但 `kotlinx.serialization` 更加型別安全，避免了 Java 的類型擦除問題。

1. 類型擦除（Type Erasure）問題

- 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：

- ```
  @Serializable
  data class People<T>(var value: T)
  @Serializable
  data class Man(val name: String)

    val gson = Gson()
    val people = People(Man("John"))
    gson.toJson(people)

    gson.fromJson<People<Man>>(people, People::class.java) // 無法將 people.value 反序列化為 People
  ```

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

  url

  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 gson = Gson()
    val people = People(Man("John"))
    val peopleType = object : TypeToken<People<Man>>() {}.type
    val gsonString = gson.toJson(people, peopleType)
    println(gsonString) //{"value":{"name":"John"}}

    val people2 = gson.fromJson<People<Man>>(gsonString, peopleType)
    println(people2) // People(value=Man(name=John))
  ```

- ```
    val gson = Gson()
    val users = listOf(User("Alice",18),User("Sam",20))
    val gsonString = gson.toJson(users)
    // 序列化
    println(gsonString) // [{"name":"Alice","age":18},{"name":"Sam","age":20}]

    // 反序列化
    val userType = object : TypeToken<List<User>>() {}.type
  //   val users2 = gson.fromJson<List<User>>(gsonString,  List<User>::class.java)  // 無法反序列化
    val users2 = gson.fromJson<List<User>>(gsonString, userType)
    println(users2) // [User(name=Alice, age=18), User(name=Sam, age=20)]
  ```

- ```
  import com.google.gson.Gson
  import com.google.gson.GsonBuilder
  ```
  ```

```
import com.google.gson.reflect.TypeToken

val gson: Gson = GsonBuilder()
        .serializeNulls()
        .setLenient()
        .create()

    private val prettyGson: Gson = GsonBuilder()
        .setPrettyPrinting()
        .serializeNulls()
        .setLenient()
        .create()

    fun <T> fromJson(json: String, clazz: Class<T>): T {
        return gson.fromJson(json, clazz)
    }

    inline fun <reified T> fromJson(json: String): T {
        return gson.fromJson(json, object : TypeToken<T>() {}.type)
    }

    fun toJson(src: Any): String {
        return gson.toJson(src)
    }

    fun toPrettyJson(src: Any): String {
        return prettyGson.toJson(src)
    }
////////////////////////////////////////////////////////////////

    val map = listOf(User("Alice", 25) ,User("John", 18))
    val userList = GsonUtil.fromJson<List<User>>(GsonUtil.toJson(map))
    println(userList) // [User(name=Alice, age=25), User(name=John, age=18)]
```

2. kotlinx.serialization 的型別安全

`kotlinx.serialization` 是 Kotlin 原生的序列化庫，它能夠利用 Kotlin 的反射和內聯函數特性在編譯時和運行時保留類型信息，避免了類型擦除問題。這意味著你可以直接在編譯期獲取類型信息，從而在運行時保留完整的類型安全性。

範例：

```
@Serializable
data class People<T>(var value: T)
@Serializable
data class Man(val name: String)

    val people = People(Man("John"))
    val jsonString = Json.encodeToString(people)
    println(jsonString) // {"value":{"name":"John"}}
    val people2 = Json.decodeFromString<People<Man>>(jsonString)
    println(people2) // People(value=Man(name=John))
```

在這個例子中，`kotlinx.serialization` 使用 `decodeFromString<User>(jsonString)` 方法時，直接保留了 `User` 類型的信息，而無需額外提供類型提示。這是因為 Kotlin 的 `inline` 函數和 reified 泛型可以在運行時保留類型信息。

3. 優勢和好處

- **更簡潔的代碼**：由於不需要像 Java 中那樣使用 `TypeToken`，代碼更為簡潔。
- **避免運行時錯誤**：`kotlinx.serialization` 在編譯期就可以捕獲到類型錯誤，減少了在運行時發生類型錯誤的風險。
- **原生支持 Kotlin 特性**：`kotlinx.serialization` 支持 Kotlin 的數據類（data class）、密封類（sealed class）、型別別名（type alias）等，這些在序列化時都能得到更好的支持。

4. 特性比較

在 `kotlinx.serialization` 中，有多種設定可以控制 JSON 序列化和反序列化的行為，這些設定可以通過 `Json` 配置進行調整。以下是對這

些設定的詳細補充說明，以及如何混合使用它們：

1. prettyPrint

- **說明**：如果設置為 `true`，生成的 JSON 會以更具可讀性的格式進行縮排和換行。這對於需要易於閱讀和調試的 JSON 輸出非常有用。
- 預設值(false)
- **範例：**

```
val json = Json { prettyPrint = true }
val data = json.encodeToString(User("Alice", 25))
println(data) // 生成的 JSON 會換行和縮排

/*
{
   "name": "Alice",
   "age": 25
}
*/
```

- ```
  val gson = GsonBuilder().setPrettyPrinting().create()
  val jsonString = gson.toJson(User("Alice", 25))
  println(jsonString)

  /*
  {
     "name": "Alice",
     "age": 25
  }
  */
  ```

2. isLenient

- **說明**：如果設置為 `true`，反序列化時會允許 JSON 不符合嚴格的 JSON 標準。例如，允許非雙引號字符串、不轉義的控制字符等。
- **範例：**

```
val json = Json { isLenient = true }
val user = json.decodeFromString<User>("""{"name":'Alice', "age":25}""")
println(user)  // User(name=Alice, age=25)
```

- ```
    val gson = GsonBuilder()
              .setLenient()
              .create()
    val reader = StringReader("{'name': 'Alice', 'age': 25}")
    val user: User = gson.fromJson(reader, User::class.java)
    println(user) // User(name=Alice, age=25)
  ```

- **用途**：用於處理不符合標準的 JSON。

3. ignoreUnknownKeys

- **說明**：如果設置為 `true`，反序列化時會忽略 JSON 中多餘的、不在 Kotlin 資料類中定義的鍵。
- 預設值(false)，多餘的、不在 Kotlin 資料類中定義的鍵，會拋出錯誤
- **範例：**

```
val json = Json { ignoreUnknownKeys = true }
val data = json.decodeFromString<User>("""{"name":"Alice", "age":25, "gender":"female"}""")
// User(name=Alice, age=25)
```

- ```
  // Gson 只會輸出class有的屬性，不會出現錯誤
  val gson = Gson()
    val user: User = gson.fromJson("{\"name\": \"Alice\", \"age\": 25, \"gender\": \"female\"}", User::class.java)
    println(user) // User(name=Alice, age=25)
  ```

- **用途**：用於處理結構可能有變化或不完整的 JSON 資料。

4. @JsonNames

- **說明**：用於指定多個 JSON 欄位名可以對應於 Kotlin 類中的同一屬性。這對於處理不同命名風格或版本兼容性非常有用。
- **範例：**
  ```

```
@Serializable
data class User(
    @JsonNames("username", "user_name","uname")
    val name: String,
    val age: Int
)

val json = Json{}

val User = Json.encodeToString(User("mary", 20))
println(User) // {"name":"mary","age":20}

val jsonNamesData = json.decodeFromString<User>("""{"uname":"Alice", "age":25, "gender":"female"}""")
println(jsonNamesData) // User(name=Alice, age=25)
```

- ```
  import com.google.gson.annotations.SerializedName

  data class User(
      @SerializedName(value = "user_name", alternate = ["username", "userName"])
      val name: String,
      val age: Int
  )

  // 序列化
  val jsonString = """{"user_name": "Alice", "age": 25}"""
  val gson = Gson()
  val user: User = gson.fromJson(jsonString, User::class.java)
  println(user.name)  // Output: Alice

  // 反序列化
  val user = User(name = "Alice", age = 25)
  val gson = Gson()
  val jsonString = gson.toJson(user)
  println(jsonString)  // Output: {"user_name":"Alice","age":25}
  ```

- **用途**：用於不同 JSON 格式之間的兼容性。

5. `encodeDefaults`

- **說明**：如果設置為 `true`，則在序列化時會包含所有屬性，即使它們具有預設值；如果為 `false`，則省略具有預設值的屬性。預設輸出預設值(false)
- **範例**：

```
data class User(
    @JsonNames("username", "user_name", "uname")
    val name: String,
    val age: Int = 30
)

val json = Json { encodeDefaults = true }
val data = json.encodeToString(User("Alice"))

//    encodeDefaults = true
//   {
//      "name": "Alice",
//      "age": 30
//   }

//    encodeDefaults = false (default)
//   {
//      "name": "Alice"
//   }
```

- ```
  // 說明：Gson 不會使用 Kotlin 的預設參數值,
  // 想要預設值需要自定義序列化器
      val userSerializer = JsonSerializer<User> { src, _, _ ->
        val DEFAULT_AGE = 30
        val jsonObject = JsonObject()
        jsonObject.addProperty("name", src.name)
  ```

```
        if (src.age != DEFAULT_AGE) { // 檢查 age 是否為預設值
            jsonObject.addProperty("age", src.age)
        }
        jsonObject
    }

    // 創建 Gson 並註冊自定義序列化器
    val gson = GsonBuilder()
        .registerTypeAdapter(User::class.java, userSerializer)
        .create()

    val alice = User("Alice")
    val jsonString = gson.toJson(alice)
    println(jsonString) // Output: {"name":"Alice"}

    val tom = User("Tom",18)
    val jsonString2 = gson.toJson(tom)
    println(jsonString2) // Output: {"name":"Tom","age":18}
```

- **直接隱藏屬性**

```
import com.google.gson.Gson
import com.google.gson.GsonBuilder
import java.lang.reflect.Modifier
import kotlinx.serialization.Transient

// Gson 或是使用 @Transient 註解 + excludeFieldsWithModifiers 排除屬性
// kotlinx.serialization 也可以用此註解，效果一樣
data class User(
    val name: String,
    @Transient val age: Int = 30 // 被 transient 修飾符修飾
) {
    companion object {
        const val COMPANY = "ExampleCompany"  // 靜態屬性，將被排除
    }
}

fun main() {
    val gson: Gson = GsonBuilder()
        .excludeFieldsWithModifiers(Modifier.TRANSIENT, Modifier.STATIC) // 排除 transient 和 static 修飾符的屬性
        .create()

    val user = User(name = "Alice")
    val jsonString = gson.toJson(user)

    println(jsonString) // {"name":"Alice"}

    // kotlinx.serialization 也可以用此註解，效果一樣
    println(Json.encodeToString(user)) // Output: {"name":"Alice"}
}
```

- **用途**：控制 JSON 中輸出的詳細程度。

6. `explicitNulls`

- **說明**：如果設置為 `false`，序列化時將省略值為 `null` 的屬性；如果設置為 `true`，則會顯示 `null` 值。
  預設(false)不輸出null
- **範例：**

```
val json = Json { explicitNulls = true }
val data = json.encodeToString(User("Alice", null))

//   {
//     "name": "Alice",
//     "age": 25,
//     "nickName": null
//   }
```

```
data class User2(
```

```
//   @JsonNames("username", "user_name", "uname")
  val name: String,
  val age: Int? = null,
  val nickName: String? = null
)

val gsonIncludeNulls = GsonBuilder().serializeNulls().create() // 包含 null
val gsonExcludeNulls = GsonBuilder().create() // 排除 null

println(gsonIncludeNulls.toJson(User("Alice", 12, null)))
 // {"name":"Alice","age":12,"nickName":null}

println(gsonExcludeNulls.toJson(User("Alice", 13, null)))
 // {"name":"Alice","age":13}
```

- **用途**：控制如何處理 `null` 值，特別是在需要減少 JSON 大小時。

7. `allowStructuredMapKeys`

- **說明**：如果設置為 `true`，允許使用複雜類型（例如，資料類）作為 Map 的鍵；否則僅允許基本類型作為鍵。
- **範例**：

```
val json = Json { allowStructuredMapKeys = true }
val data = json.encodeToString(mapOf(User("Alice", 25) to "Developer"))

/*
[
  {
    "name": "Alice",
    "age": 25
  },
  "Developer"
]
*/
```

- ```
  // Gson 不直接支持複雜類型作為 Map 的鍵。通常需要將 Map 鍵轉換為字串形式：
  data class User(val name: String, val age: Int)
  val gson = GsonBuilder().create()
  val map = mutableMapOf(gson.toJson(User("Alice", 25)) to "Developer")
  val jsonString = gson.toJson(map)
   // {"{\"name\":\"Alice\",\"age\":25}":"Developer"}
  ```

- ```
  // 使用 enableComplexMapKeySerialization 會有以意料之外結果
    val gson = GsonBuilder()
      .enableComplexMapKeySerialization()
      .create()

    val map = mutableMapOf(User("Alice", 25) to "Developer")
    println(map) // {User(name=Alice, age=25)=Developer}
    println(gson.toJson(map)) // [[{"name":"Alice","age":25},"Developer"]]
  /*

  當你使用 enableComplexMapKeySerialization() 時，
  Gson 允許序列化包含複雜對象（例如，自定義的 User 類型）作為鍵的 Map。默認情況下，Gson 只能處理字符串等基本類型作為鍵。
  Gson 序列化 Map 時，會將鍵和值分別序列化。當使用複雜對象作為鍵時，Gson 會將這些鍵作為 JSON 的單獨條目來處理。

  由於 User("Alice", 25) 是一個非基本類型對象，Gson 將其序列化為 {"name":"Alice","age":25}，
  而 Map 的結果被序列化為包含鍵值對的數組。

  當你打印 gson.toJson(map) 時，結果是 [[{"name":"Alice","age":25},"Developer"]]，
  這意味著 JSON 的輸出是一個數組，數組中包含了鍵和值的另一個數組。
  這是因為 Gson 使用 List（一種數組結構）來表示 Map 的鍵值對，特別是在鍵是非基本類型的情況下。

  */

    val map = mutableMapOf(User("Alice", 25) to "Developer",User("John", 18) to "Student")
    println(map)
     // {User(name=Alice, age=25)=Developer, User(name=John, age=18)=Student}
    println(gson.toJson(map))
  ```

```
// [[{"name":"Alice","age":25},"Developer"],[{"name":"John","age":18},"Student"]]
```

- **用途**：用於需要使用複雜鍵的 JSON 結構。

8. `allowSpecialFloatingPointValues`

- **說明**：如果設置為 `true`，允許特殊的浮點數值（例如，NaN、Infinity）在 JSON 中表示；如果為 `false`，則會拋出錯誤。
- **範例**：

```
val json = Json { allowSpecialFloatingPointValues = true }
val data = json.encodeToString(Double.POSITIVE_INFINITY)

// Infinity
```

```
val gson = GsonBuilder()
    .registerTypeAdapter(Double::class.java, JsonSerializer<Double> { src, _, _ ->
        when {
            src.isNaN() || src.isInfinite() -> JsonPrimitive(src.toString())
            else -> JsonPrimitive(src)
        }
    })
    .registerTypeAdapter(Double::class.java, JsonDeserializer { json, _, _ ->
        when (val value = json.asString) {
            "NaN" -> Double.NaN
            "Infinity" -> Double.POSITIVE_INFINITY
            "-Infinity" -> Double.NEGATIVE_INFINITY
            else -> value.toDouble()
        }
    })
    .create()
```

- **用途**：用於需要表示特殊浮點值的 JSON 序列化和反序列化。

如何混合使用這些配置

你可以將這些配置項混合使用來滿足特定的序列化和反序列化需求。

例如，假設你有一個情況需要反序列化非標準的 JSON 資料，並且需要忽略未知的鍵且保留結構化的 Map 鍵：

```
val json = Json {
    prettyPrint = true
    isLenient = true
    ignoreUnknownKeys = true
    encodeDefaults = false
    explicitNulls = false
    coerceInputValues = true
    allowStructuredMapKeys = true
    allowSpecialFloatingPointValues = true
}

val data = json.decodeFromString<User>("""{"name":"Alice", "unknownField":"value"}""")
println(data)
// User(name=Alice, age=30)
```

```
val gson = GsonBuilder()
    .setPrettyPrinting()
    .setLenient()
    .serializeNulls()  // Include nulls
    .registerTypeAdapter(User::class.java, JsonDeserializer { json, _, _ ->
        val jsonObject = json.asJsonObject
        val name = if (jsonObject.has("name")) jsonObject["name"].asString else "DefaultName"
        val age = if (jsonObject.has("age")) jsonObject["age"].asInt else 30 // Default value
        User(name, age)
    })
    .create()

val jsonString = """{"name":"Alice","unknownField":"value"}"""
val user: User = gson.fromJson(jsonString, User::class.java)
println(user) //User(name=Alice, age=30)
```

```
println(gson.toJson(User("Alice")))
 /*
 {
   "name": "Alice",
   "age": 30
 }
 */
```

在這個例子中，我們結合了多種配置來實現靈活且容錯的 JSON 解析和輸出，確保代碼能夠應對多種 JSON 格式和數據問題。這樣的設置可以讓你的應用更加健壯和靈活。

json default value:

```
/**
 * Configuration of the current [Json] instance available through [Json.configuration]
 * and configured with [JsonBuilder] constructor.
 *
 * Can be used for debug purposes and for custom Json-specific serializers
 * via [JsonEncoder] and [JsonDecoder].
 *
 * Standalone configuration object is meaningless and can nor be used outside the
 * [Json], neither new [Json] instance can be created from it.
 *
 * Detailed description of each property is available in [JsonBuilder] class.
 */
public class JsonConfiguration @OptIn(ExperimentalSerializationApi::class) internal constructor(
    public val encodeDefaults: Boolean = false,
    public val ignoreUnknownKeys: Boolean = false,
    public val isLenient: Boolean = false,
    public val allowStructuredMapKeys: Boolean = false,
    public val prettyPrint: Boolean = false,
    public val explicitNulls: Boolean = true,
    @ExperimentalSerializationApi
    public val prettyPrintIndent: String = "    ",
    public val coerceInputValues: Boolean = false,
    public val useArrayPolymorphism: Boolean = false,
    public val classDiscriminator: String = "type",
    public val allowSpecialFloatingPointValues: Boolean = false,
    public val useAlternativeNames: Boolean = true,
    @ExperimentalSerializationApi
    public val namingStrategy: JsonNamingStrategy? = null,
    @ExperimentalSerializationApi
    public val decodeEnumsCaseInsensitive: Boolean = false,
    @ExperimentalSerializationApi
    public val allowTrailingComma: Boolean = false,
    @ExperimentalSerializationApi
    public val allowComments: Boolean = false,
    @ExperimentalSerializationApi
    @set:Deprecated(
        "JsonConfiguration is not meant to be mutable, and will be made read-only in a future release. " +
            "The `Json(from = ...) {}` copy builder should be used instead.",
        level = DeprecationLevel.ERROR
    )
    public var classDiscriminatorMode: ClassDiscriminatorMode = ClassDiscriminatorMode.POLYMORPHIC,
) {

    /** @suppress Dokka **/
    @OptIn(ExperimentalSerializationApi::class)
    override fun toString(): String {
        return "JsonConfiguration(encodeDefaults=$encodeDefaults, ignoreUnknownKeys=$ignoreUnknownKeys, isLenient=$isLenient, "
+
                "allowStructuredMapKeys=$allowStructuredMapKeys, prettyPrint=$prettyPrint, explicitNulls=$explicitNulls, " +
                "prettyPrintIndent='$prettyPrintIndent', coerceInputValues=$coerceInputValues,
useArrayPolymorphism=$useArrayPolymorphism, " +
                "classDiscriminator='$classDiscriminator', allowSpecialFloatingPointValues=$allowSpecialFloatingPointValues, " +
                "useAlternativeNames=$useAlternativeNames, namingStrategy=$namingStrategy,
decodeEnumsCaseInsensitive=$decodeEnumsCaseInsensitive, " +
                "allowTrailingComma=$allowTrailingComma, allowComments=$allowComments,
classDiscriminatorMode=$classDiscriminatorMode)"
```

```
    }
}
```

---

5. 擴展性和客製化

在 JSON 序列化和反序列化過程中，`kotlinx.serialization` 和 Gson 都提供了強大的擴展性和客製化能力。這些能力允許開發者根據需要定義自定義的序列化和反序列化邏輯，以處理特殊格式的數據或滿足特定需求。

kotlinx.serialization 的擴展性和客製化

`kotlinx.serialization` 提供了一個簡潔的方式來自定義序列化和反序列化邏輯，通過實作 `KSerializer` 介面，你可以完全控制對象的序列化和反序列化過程。

如何實作自定義序列化器

1. **定義資料類：**

```
@Serializable
data class Event(val name: String,
                 @Serializable(with = DateSerializer::class) val date: Date)
```

   在這個例子中，`Event` 類的 `date` 屬性使用了自定義序列化器 `DateSerializer` 來處理 `Date` 類型。

2. **實作 `KSerializer` 介面：**

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

    override fun serialize(encoder: Encoder, value: Date) {
        // 自定義的序列化邏輯，將 Date 轉換為 String
        val dateFormat = SimpleDateFormat("yyyy-MM-dd")
        val dateString = dateFormat.format(value)
        encoder.encodeString(dateString)
    }

    override fun deserialize(decoder: Decoder): Date {
        // 自定義的反序列化邏輯，將 String 轉換為 Date
        val dateFormat = SimpleDateFormat("yyyy-MM-dd")
        val dateString = decoder.decodeString()
        return dateFormat.parse(dateString)
    }
}
```

   在這個例子中：
   - `serialize` 方法將 `Date` 對象序列化為 `String` 格式。
   - `deserialize` 方法將 `String` 反序列化為 `Date` 對象。

3. **使用自定義序列化器：**

```
val json = Json { }
val event = Event("Conference", Date())
val jsonString = json.encodeToString(event)
println(jsonString) // {"name":"Conference","date":"2024-08-29"}

val deserializedEvent = json.decodeFromString<Event>(jsonString)
println(deserializedEvent)
// Event(name=Conference, date=Thu Aug 29 00:00:00 CST 2024)
```

   自定義序列化器 `DateSerializer` 會自動在 `encodeToString` 和 `decodeFromString` 操作中被使用，處理 `Event` 類中的 `date` 屬性。

Gson 的擴展性和客製化

在 Gson 中，客製化序列化和反序列化是通過實作 `JsonSerializer` 和 `JsonDeserializer` 介面來實現的。這提供了類似的靈活性，可以完全控制對象的序列化和反序列化過程。

如何實作自定義序列化器和反序列化器

1. **定義資料類：**

```
data class Event(val name: String, val date: Date)
```

與 `kotlinx.serialization` 不同，Gson 並不需要在資料類上添加任何特別的註解。

2. **實作 `JsonSerializer` 和 `JsonDeserializer` 介面：**

```
val dateSerializer = JsonSerializer<Date> { src, _, _ ->
    // 自定義的序列化邏輯，將 Date 轉換為 String
    val dateFormat = SimpleDateFormat("yyyy-MM-dd")
    JsonPrimitive(dateFormat.format(src))
}

val dateDeserializer = JsonDeserializer<Date> { json, _, _ ->
    // 自定義的反序列化邏輯，將 String 轉換為 Date
    val dateFormat = SimpleDateFormat("yyyy-MM-dd")
    dateFormat.parse(json.asString)
}
```

在這個例子中：
- `dateSerializer` 是一個 `JsonSerializer<Date>`，負責將 `Date` 序列化為 JSON 字符串。
- `dateDeserializer` 是一個 `JsonDeserializer<Date>`，負責將 JSON 字符串反序列化為 `Date` 對象。

3. **使用自定義序列化器和反序列化器：**

```
val gson = GsonBuilder()
    .registerTypeAdapter(Date::class.java, dateSerializer)
    .registerTypeAdapter(Date::class.java, dateDeserializer)
    .create()

val event = Event("Conference", Date())
val jsonString = gson.toJson(event)
println(jsonString)
  // {"name":"Conference","date":"2024-08-29"}

val deserializedEvent = gson.fromJson(jsonString, Event::class.java)
println(deserializedEvent)
 // Event(name=Conference, date=Thu Aug 29 00:00:00 CST 2024)
```

在這個例子中，`registerTypeAdapter` 方法用來註冊自定義的序列化器和反序列化器。Gson 會在序列化和反序列化 `Date` 類型時使用這些自定義的邏輯。

總結

- **`kotlinx.serialization` 的擴展性和客製化**：使用 `KSerializer` 介面來定義自定義的序列化和反序列化邏輯。這種方法能夠充分利用 Kotlin 的語言特性，如型別安全和編譯期檢查，確保序列化和反序列化過程中不會發生類型錯誤。
- **Gson 的擴展性和客製化**：使用 `JsonSerializer` 和 `JsonDeserializer` 介面來實現自定義的序列化和反序列化邏輯。Gson 的這種設計讓開發者可以非常靈活地處理各種數據格式和需求，特別是在 Java 環境中使用時。

這兩者都提供了足夠的靈活性來滿足大多數序列化和反序列化需求，不過 `kotlinx.serialization` 更加貼合 Kotlin 語言的特性和風格，而 Gson 則更加通用且適用於 Java 開發環境。

6. 依賴管理與體積

- **範例程式碼：**

```
// kotlinx.serialization
implementation "org.jetbrains.kotlinx:kotlinx-serialization-json:1.5.0"

// Gson
implementation 'com.google.code.gson:gson:2.9.0'
```

- ## kotlinx.serialization
  - **依賴體積**：`kotlinx.serialization` 是 Kotlin 官方提供的庫，專為 Kotlin 設計。這意味著它在體積上相對輕量，因為它不需要額外的依賴來運行 Kotlin 特性。它也不需要引入 Java 標準庫以外的額外依賴。
  - **依賴管理**：`kotlinx.serialization` 的版本更新和依賴管理與 Kotlin 生態系統的版本緊密相關。它由 JetBrains 官方維護，更新相對頻繁，且通常會與 Kotlin 編譯器的更新一起釋出，這意味著使用者需要隨時注意 Kotlin 版本的變化。
  - **優勢**：
    - **與 Kotlin 無縫集成**：`kotlinx.serialization` 是一個 Kotlin 原生庫，它與 Kotlin 語言特性（如 data class、sealed class）無縫集成。
    - **輕量且效能優化**：由於其緊密結合的 Kotlin 庫體積較小，並且效能優化專門針對 Kotlin 特性，這使得它在 Kotlin 開發中非常高效。
    - **易於在多平台項目中使用**：`kotlinx.serialization` 支持 Kotlin Multiplatform，因此它可以在同一套代碼中使用於

Android、iOS 和其他平台。
- **劣勢**：
  - **僅適用於 Kotlin 項目**：由於其與 Kotlin 的深度集成，如果你的項目是基於 Java 的，將不適合使用 `kotlinx.serialization`。

Gson

- **依賴體積**：Gson 是一個通用的 Java 庫，設計用來支持各種 Java 對象的序列化和反序列化。由於它的廣泛兼容性和全面功能，Gson 的依賴體積相對較大，包含許多標準庫來處理各種情境。
- **依賴管理**：Gson 是由 Google 開發和維護的開源項目，但它的更新頻率比 `kotlinx.serialization` 要低。Gson 的版本變更通常是為了修復錯誤或增強功能，而不是頻繁的版本更新。
- **優勢**：
  - **廣泛的兼容性**：作為一個 Java 庫，Gson 可以在任何 Java 環境中使用，包括 Kotlin 開發中。它可以處理 Java 的各種類型和情況，如泛型、嵌套類型等。
  - **功能全面且穩定**：Gson 在處理複雜 JSON 結構時非常靈活，並且提供了豐富的自定義擴展選項（如自定義序列化和反序列化）。
  - **廣泛的社群支持**：由於它的廣泛使用和成熟度，Gson 擁有大量的使用者和豐富的社群資源，許多常見問題和挑戰都有現成的解決方案。
- **劣勢**：
  - **較大的依賴體積**：相比 `kotlinx.serialization`，Gson 的依賴更大，這可能會對應用的 APK 體積有影響，特別是在 Android 開發中。
  - **較少的 Kotlin 特性支持**：雖然 Gson 可以用於 Kotlin 項目，但它並沒有 `kotlinx.serialization` 對 Kotlin 特性的原生支持，如 `sealed class` 和 `data class`。

7. 社群與支援

- **簡短說明：** `kotlinx.serialization` 有官方支持且定期更新，更適合與 Kotlin 生態系統一起使用；Gson 有更長的歷史和廣泛的使用者社群，但更新頻率較低。

8. 總結與建議

- **簡短說明：** 根據項目的需要選擇適合的工具。對於需要與 Kotlin 深度集成的應用，建議使用 `kotlinx.serialization`；如果需要支持更廣泛的 Java 類型或已有 Gson 的基礎設施，可以選擇 Gson。

修訂版本 #1
★由 treeman 建立於 2 ﷿E﷿﷿ 2024 09:54:14
🖉由 treeman 更新於 2 ﷿E﷿﷿ 2024 09:54:26