JSON Binding API 入門, 第 4 回: JSON バインディング標準化への期待

Gson、Jackson、JSON-B を比較すると明らかになる、基本的な機能と動作における一貫性の欠如

Alex Theedom

2018年 8月 23日

Gson、Jackson、JSON-B の 3 つを比較すると、基本的な機能と動作において、これらのフレームワーク間で一貫性がないことが明らかになります。この比較は、新たな JSON バインディング標準の将来性を論証するものです。

この連載について: Java EE ではこれまで長い間 XML をサポートしてきましたが、JSON データの組み込みサポートは著しく欠けていました。Java EE 8 はこの事態を変えるべく、コアとなる Java エンタープライズ・プラットフォームに強力な JSON バインディング機能を導入しています。JSON-B を採用して、Java エンタープライズ・アプリケーション内で JSON ドキュメントを処理するための JSON Processing API やその他のテクノロジーと JSON-B がどのように結合するのかを学んでください。

JSON は 10 年近くデータ交換フォーマットとしてよく使われてきましたが、JSON バインディング標準の基礎にできるだけの確固とした仕様はこれまでありませんでした。けれども今や JSON Binding API がリリースされたことにより、ベンダーは JSON パーサー内でのデフォルト・バインディング処理とカスタム・バインディング処理の両方を標準化する機会を目の前にしています。開発者として、私たちはベンダーが JSON バインディングを標準化することを強く求めています。

3 つの JSON パーサーのシリアライズとデシリアライズの動作を比較すると、これらのツールが通常のバインディングを処理する仕組みという点で、差異と特異性が明白になります。さらに、多数の重要な機能に対するサポートに一貫性がなく、ツールによって特定の機能を提供していたり提供していなかったりすることも明らかになってきます。

特異性には価値があるものの、それが機能を犠牲にして成り立っていてはなりません。実装によってデフォルトの動作が大幅に異なっていては、第一の決定要因とすべきパフォーマンスの違いよりも、どのツールを選択するかのほうが重要になってしまいます。

JSON バインディング標準化の論拠

有効な標準を策定するには時間がかかります。標準はすでに確立されている使用ケースとベスト・プラクティスに基づいてこそ、最も効果を発揮します。つまり、優れた標準は、長年にわた

る実験、修正、革新に基づく専門知識の産物なのです。この点からすると、JSON パーサーを標準化するには、機が熟していると思います。

> **皆さんはどう思いますか？** JSON バインディング標準の採用は JSON が次に進むべきステップであり、その標準としては JSR 367 が最適であると私は確信していますが、皆さんはどう思いますか？この記事の最後に記載されているリンク先のページで、ご意見を聞かせてください。

JSON バインディングの処理と機能を標準化すれば、開発者は基礎となるエンジンの効率性とパフォーマンスに基づいてツールを比較できるようになります。実装者は、機能よりもパフォーマンスに大きな重点を置けるようになるでしょう。ツールの品質全般も向上するはずです。

標準化によって革新が抑制されることはありません。それどころか、標準化ではこれまでに生まれた革新を反映し、将来の革新と進歩に向けた堅固な基盤を築きます。このようにして確立された標準に組み込まれる候補となるのは、最良の機能です。

標準化することには確固たる論拠があり、JSON Binding API 仕様について言えば、これを標準にしない正当な理由はないと思います。以降のセクションで、コアとなる機能を 3 つの JSON バインディング・ツールの間で比較します。この比較は興味深いものですが、私が意図する要点は、それぞれのツールがバインディング処理と Java の型を扱う仕組みにおける違いを明らかにすることです。

JSON パーサーの比較

この記事では JSON Binding API に対し、よく使われている 2 つのフレームワークとして FastXML の Jackson Project と Google の Gson を比較します。

まずは、構造化 JSON ドキュメント内の主要なデータ型を、それぞれのフレームワークがデフォルトではどのように処理するのかを比較します。その後、Java 基本データ型から Java 8 に含まれる新しい日時形式まで、さまざまなデータ型のカスタマイズに各フレームワークがどのようにして対処するのかを見ていきます。

Java オブジェクトと JSON の間でのシリアライズとデシリアライズを処理する際の各 JSON フレームワークの動作を自分の目で確かめてください。私が目標とするのは科学的レビューやベンチマーク比較ではなく、これらのフレームワークの間での違いと一貫性を全体的に把握することです。そして最終的には、私が最も興味がある点として、JSON Binding 仕様を標準化するメリットを明らかにしたいと思います。

この記事に記載するサンプル・コードには、JSON Processing 1.1 をプロバイダーとして使用した JSON Binding API バージョン 1.0 と、Gson バージョン 2.8.2、Jackson バージョン 2.9.2 を使用しています。GitHub リポジトリーを作成して、そこにサンプル・コードと単体テストを格納してあるので、これらのサンプル・コードを自分用に複製できます。

コードを入手する

バインディング・モデル

JSON パーサー内では、バインディング・モデルがシリアライズとデシリアライズのメカニズムとなります。ここで比較する 3 つすべてのフレームワークに、一般的なバインディング・シナリオ

でのデフォルトのバインディング動作が定義されています。さらに、これらの各フレームワークではカスタム・バインディング・ソリューションも使用できるようになっています。カスタム・バンディングの選択肢として、開発者はコンパイル時のアノテーションを使用することも、ランタイム構成を使用することもできます。このセクションでは、JSON-B、Jackson、Gson のそれぞれが、ランタイム構成を使用してどのようにカスタム・バインディングに対処するのかを説明します。カスタム・バインディングのモデルはフレームワークによって大幅に異なるため、比較の出発点として最適です。カスタマイズ機能については、後のほうのセクションで詳しく見ていきます。

JSON-B でのランタイム・バインディング

開発者は JSON Binding のランタイム構成にアクセスするために、`JsonConfig` クラスを使用します。このクラスの `with(...)` メソッドを呼び出して構成プロパティーを渡すことで、ランタイム構成をカスタマイズします。例えば、`new JsonConfig.withPropertyOrderStrategy(PropertyOrderStrategy.REVERSE)` といった具合に構成プロパティーを渡します。

これにより、`JsonConfig` インスタンスが `JsonBuilder` インスタンス上に設定されるので、このインスタンスに対して `toJson()` メソッドまたは `fromJson()` メソッドを呼び出します。リスト 1 に一例を記載します。

リスト 1. JSON Binding API でのランタイム構成

```
JsonbConfig jsonbConfig = new JsonbConfig()
    .with...(...);

String json = JsonbBuilder
    .create(jsonbConfig)
    .toJson(...);

AnObject anObj = JsonbBuilder
    .create(jsonbConfig)
    .fromJson("{JSON}", AnObject.class);
```

Jackson でのランタイム・バインディング

開発者がオブジェクト・マッパー・エンジンを直接構成するという点で、Jackson のランタイム・モデルは JSON-B のモデルとは異なります。構成されたオブジェクト・マッパー・エンジンが、ターゲット・オブジェクト上でシリアライズおよびデシリアライズを行うために使用されます。リスト 2 を見るとわかるように、`ObjectMapper` にはかなり幅広い構成メソッドが用意されています。

リスト 2. Jackson のランタイム・モデル

```
ObjectMapper objectMapper = new ObjectMapper()
    .set...(...)
    .configure(...)
    .addHandler(...)
    .disable(...)
    .enable(...)
    .registerModule(...)
    ...;

String json = objectMapper
    .writeValueAsString(...);

AnObject anObj = objectMapper
    .readValue("{JSON}", AnObject.class);
```

Gson でのランタイム・バインディング

Gson では `GsonBuilder` のインスタンスを構成し、このインスタンスから `Gson` インスタンスを作成します。Gson オブジェクトには、シリアライズおよびデシリアライズの処理を行うためのメソッドが用意されています。リスト 3 に、Gson ランタイム・モデルを記載します。

リスト 3. Gson ランタイム・モデル

```
GsonBuilder gsonBuilder = new GsonBuilder()
    .set...(...)
    .add....(...)
    .register...(...)
    .serialize...(...)
    .enable...(...)
    .disable...(...)
    ...;

String json = gsonBuilder
    .create()
    .toJson(...);

AnObject anObj = gsonBuilder
    .create()
    .fromJson("{JSON}", AnObject.class);
```

次は、これら 3 つのパーサーによる型の処理方法に目を向けます。まずは、Java 基本データ型から見ていきましょう。

Java 基本データ型

Java での**基本データ型**とは、`String` 型に加え、プリミティブ型とそれぞれに対応するラッパー・クラスのすべてを指します。これから、これらすべての型を JSON にシリアライズする単純な例を比較します。

まずはリスト 4 に、Java 言語のすべての基本データ型が含まれるクラスを記載します。

リスト 4. Java 基本データ型

```
public class AllBasicTypes {

    // Primitive types
    private byte bytePrimitive;
    private short shortPrimitive;
```

```
    private char charPrimitive;
    private int intPrimitive;
    private long longPrimitive;
    private float floatPrimitive;
    private double doublePrimitive;
    private boolean aBoolean;

    // Wrapper types
    private Byte byteWrapper = 0;
    private Short shortWrapper = 0;
    private Character charWrapper = 0;
    private Integer intWrapper = 0;
    private Long longWrapper = 0L;
    private Float floatWrapper = 0F;
    private Double doubleWrapper = 0D;
    private Boolean booleanWrapper = false;

    private String string = "Hello World";

    // Getters and setter omitted for brevity
}
```

3 つのフレームワークのそれぞれを使用してシリアライズすると、いずれもリスト 5 に記載する JSON ドキュメントが生成されます。この後わかるように、現在のところフレームワークの間で一貫している動作は、このシリアライズ動作だけです。この例では、JSON プロパティーをソース・クラスと対応させやすくするために、フィールドを自己記述的な名前にしていることに注意してください。

リスト 5. JSON ドキュメント

```
{
    "bytePrimitive": 0,
    "shortPrimitive": 0,
    "charPrimitive": "\u0000",
    "intPrimitive": 0,
    "longPrimitive": 0,
    "floatPrimitive": 0,
    "doublePrimitive": 0,
    "aBoolean": false,
    "byteWrapper": 0,
    "shortWrapper": 0,
    "charWrapper": "\u0000",
    "intWrapper": 0,
    "longWrapper": 0,
    "floatWrapper": 0,
    "doubleWrapper": 0,
    "booleanWrapper": false,
    "string": "Hello World"
}
```

3 つの JSON パーサーはいずれも、基本データ型の処理にデフォルトの構成を使用し、構成オプションはほとんど使用しません JSON-B で数値のフォーマットを設定する例については、「JSON Binding API 入門」の第 3 回を参照してください。

Java 固有のデータ型

いわゆる固有のデータ型には、`java.lang.Number` を継承する型など、より複雑な JDK データ型が含まれます。具体的には、`BigDecimal`、`AtomicInteger`、`LongAdder`、そして `OptionalInt` と `Optional<type>` を含む `Optional` 型や、`URL` および `URI` インスタンスなどがあります。

まずは、`Number` 型を比較しましょう。すべての JSON フレームワークは、単純に内部の数値をプロパティー値として抽出するという方法でシリアライズを行います。`BigDecimal`、`BigInteger`、`AtomicInteger`、および `LongInteger` の処理は、3 つのフレームワークの間で一貫しています。

`URI` および `URL` クラスについても、3 つすべてのフレームワークで同じように処理されます。つまり、シリアライズされた JSON ドキュメント内では、内部の値が `String` として表現されます。

一方、`Optional` 型の処理については、フレームワークの間で一貫していません。表 1 に、リスト 6 に示す 2 つのインスタンスをシリアライズした結果を記載します。

リスト 6. テスト用の Optional クラス

```
Optional<String> stringOptional = Optional.of("Hello World");
OptionalInt optionalInt = OptionalInt.of(10);
```

表 1 では、比較しやすいように 2 つのフィールドを分けて記載しています。

表 1. Optional 型

	Optional<String>	OptionalInt
JSON-B	`"stringOptional": "Hello World"`	`"optionalInt": 10`
Jackson	`"stringOptional": { "present": true }`	`"optionalInt": { "asInt": 10,"present": true }`
Gson	`"stringOptional": { "value": "Hello World" }`	`"optionalInt": { "isPresent": true, "value": 10 }`

ご覧のように、JSON-B は内部を覗いてそこにある値を取得することによって `Optional` 型の値を完全にサポートします。Jackson と Gson はいずれも、最初から `Optional` をサポートするようにはなっていません。`String Optional` のシリアライズに関しては、この 2 つは最も一貫性に欠けています。

Gson は `Optional` の構造を、`String` を格納した JSON オブジェクトにシリアライズする一方、Jackson は基礎となるインスタンスの値を含めることもせず、その値が存在することを記述するだけです。

`OptionalInt` と残りの `Optional` 型系列の数値は Gson でも Jackson でも十分にサポートされていませんが、少なくとも妥当な JSON オブジェクト、つまり基礎となる値とその値が存在することを表すブール値を格納した JSON オブジェクトにシリアライズされます。

日付、時刻、およびカレンダー型

従来の `java.util.Date` 型と `java.util.Calendar` 型や、`java.time` に含まれている新しい Java 8 の日付および時刻クラスを含め、日付、時刻、カレンダーの型は多数あります。

比較対象の型の範囲があまりにも広いことから、ここでは次の 3 つに絞って取り上げることにします。

- ビルダーを使用して日付が設定される `Calendar` のインスタンス

- `String` からの日付を設定して `SimpleDateFormat` を適用する `Date` のインスタンス
- `parse()` メソッドを使用して日付が設定される Java 8 の `LocalDate` のインスタンス

Calendar 型

リスト 7 に示す `Calendar` インスタンスは、2017 年 12 月 25 日に日付を設定して構成されます。

リスト 7. Calendar 型

```
new Calendar.Builder().setDate(2017, 1, 25).build()
```

表 2 に、このインスタンスを 3 つのフレームワークのそれぞれを使用してシリアライズした結果を記載します。

表 2. シリアライズ後の Calendar インスタンス

	Calendar インスタンス
JSON-B	`2017-12-25T00:00:00Z[Europe/London]`
Jackson	`1514160000000`
Gson	`{ "year": 2017, "month": 11, "dayOfMonth": 25, "hourOfDay": 0, "minute": 0, "second": 0 }`

表 2 から明らかなように、これらのフレームワークが `Calenar` インスタンスをシリアライズするために実装している方法はそれぞれに異なります。JSON-B は時刻とタイムゾーンをゼロにして日付を出力します。Jackson はエポック (1970 年 1 月 1 日) からの経過時間 (ミリ秒数) を出力し、Gson は日付の数値の部分を格納した JSON オブジェクトを出力します。

このように、3 つのフレームワークのデフォルト動作には明確な違いがあります。これらのフォーマットをより読みやすいものに構成することは可能ですが、そもそも出力フォーマットにここまで大幅な違いがあってはなりません。

Date 型と LocalDate 型

次は、`Date` と `LocalDate` の処理方法を見ていきましょう。リスト 8 に、これら 2 つのインスタンスを作成するコードを記載し、それぞれのシリアライズ結果を表 3 に記載します。

リスト 8. LocalDate および Date インスタンス

```
new SimpleDateFormat("dd/MM/yyyy").parse("25/12/2017");
LocalDate.parse("2017-12-25");
```

表 3. シリアライズ後の LocalDate および Date インスタンス

	Date	LocalDate

JSON-B	`2017-12-25T00:00:00Z[UTC]`	`2017-12-25`
Jackson	`1514160000000`	`{ "year": 2017, "month": "DECEMBER", "chronology": { "id": "ISO", "calendarType": "iso8601" }, "dayOfMonth": 25, "dayOfWeek": "MONDAY", "dayOfYear": 359, "leapYear": false, "monthValue": 12, "era": "CE" }`
Gson	`Dec 25, 2017 12:00:00 AM`	`{ "year": 2017, "month": 12, "day": 25 }`

Date 型

JSON Binding は元の日付形式を無視してデフォルトのフォーマット設定スタイルを適用し、時刻とタイムゾーンの情報を追加します。Jackson はエポックからの経過時間 (ミリ秒数) を返し、Gson は `MEDIUM` 日付/時刻スタイルを適用します。

LocalDate 型

`LocalDate` インスタンスの処理方法もフレームワークによって異なり、JSON-B の観点からすると、LocalDate 型の扱いは `Data` 型の場合よりも少々論理的です。JSON-B は日付だけをデフォルトの日付形式で出力し、時刻やタイムゾーンの情報を追加することはしません。Jackson はインスタンス内のアクセサー・メソッドを呼び出し、`LocalDate` を POJO として効果的に処理することで JSON オブジェクトを生成します。Gson は、year、month、day の 3 つのプロパティーを使用してオブジェクトを作成します。

明らかに、`Date`、`LocalDate`、および `Calendar` インスタンスの処理方法は 3 つの JSON フレームワークの間で一貫していません。

日付形式と時刻形式を構成する

日付形式と時刻形式は、コンパイル時にカスタマイズすることも、ランタイム構成でカスタマイズすることもできます。特定のプロパティー、メソッド、またはクラスに日付形式と時刻形式を指定するアノテーションを使用できるのは、JSON-B と Jackson だけです (リスト 9 と 10 を参照)。

リスト 9. JSON Binding を使用してコンパイル時に日付形式を構成する

```
@JsonbDateFormat(value = "MM/dd/yyyy", locale = "Locale.ENGLISH")
```

リスト 10. Jackson を使用してコンパイル時に日付時刻を構成する

```
@JsonFormat(pattern = "MM/dd/yyyy", locale = "Locale.ENGLISH")
```

形式をグローバルに設定するランタイム構成は、3 つすべてのパーサーで使用できるようになっています (リスト 11、12、13 を参照)。

リスト 11. JSON Binding を使用して実行時に日付形式を構成する

```
new JsonbConfig().withDateFormat("MM/dd/yyyy", Locale.ENGLISH)
```

リスト 12. Jackson を使用して実行時に日付形式を構成する

```
new ObjectMapper().setDateFormat(new SimpleDateFormat("MM/dd/yyyy"))
```

リスト 13. Gson を使用して実行時に日付形式を構成する

```
new GsonBuilder().setDateFormat("MM/dd/yyyy")
```

配列、コレクション、マッピング

配列、コレクション、マッピングの処理に関しては、意外にも 3 つのパーサーは一貫しています。その理由は、これらの構造のそれぞれは、同等の JSON 型に直接対応するためです。リスト 14 のコードは、どの JSON フレームワークを使用してシリアライズするとしても、リスト 15 に記載する JSON ドキュメントになります。

リスト 14. 配列、コレクション、マッピングの例

```
private int[] intArray = new int[]{ 1, 2, 3, 4 };
private String[] stringArray = new String[]{ "one", "two" };

Collection<Object> objectCollection = new ArrayList<Object>() {{
   add("one");
   add("two");
}};

private Map<String, Integer> stringIntegerMap = new HashMap<String, Integer>() {{
   put("one", 1);
   put("two", 2);
}};
```

上記の構造は、以下の JSON ドキュメントにシリアライズされます。

リスト 15. シリアライズ後の配列、コレクション、マッピング

```
{
    "intArray": [1, 2, 3, 4],
    "objectCollection": ["one", "two"],
    "stringArray": ["one", "two"],
    "stringIntegerMap": { "one": 1, "two": 2 }
}
```

null、コレクション内の null、Optional.empty()

null は JSON Binding や Gson ではシリアライズされませんが、Jackson では除外されることなく維持されます。デシリアライズの際は、JSON ドキュメント内で値が欠落していると、ターゲット・オブジェクト内の対応するセッター・メソッドが呼び出されません。ただし、値が null の場合は、通常の値として設定されます。

3 つすべてのフレームワークに共通して、配列、マッピング、コレクションに含まれる null はデフォルトで維持されます。

Optional.empty()

`Optional.empty()` の値は存在しない値を表すため、JSON Binding では null と同じように扱います。つまり、シリアライズでは JSON ドキュメントからそのプロパティーが除外されます。一方、Jackson と Gson はいずれも、この値を JSON オブジェクトにシリアライズしようと試みます。Jackson は `Optional.empty()` を POJO として扱い、Gson は空の JSON オブジェクトを生成します。

表 4 に、`Optional<Object> emptyOptional = Optional.empty()` のシリアライズを要約します。

表 4. Optional.Empty() インスタンスのシリアライズ

JSON-B	Jackson	Gson

(プロパティーは JSON ドキュメントから除外されます)	<pre>{
	  "emptyOptional": {
	  "present": false
	  }
	}</pre>
		<pre>{
		  "emptyOptional": {}
		}</pre>

null の構成手段

JSON Binding API では、null に対してランタイム構成とコンパイル時の構成の両方を使用できるようになっています。コンパイル時の構成で特定のフィールドに null を含めるには、`@JsonbProperty` アノテーションを使用して `nillable` フラグを `true` に設定します。あるいは、クラス・レベルまたはパッケージ・レベルで `@JsonbNillable` アノテーションを使用してグローバルに null を含めるよう設定するという方法もあります (リスト 16 を参照)。

リスト 16. JSON Binding でのコンパイル時の null の構成

```
@JsonbNillable
public class CompileTimeSampler {
   @JsonbProperty(nillable = true)
   private String nillable;
   // field level configuration overrides class and package level configuration settings.
}
```

ランタイム構成を使用する場合は、true に設定した `withNullValues()` メソッドを使用します (リスト 17 を参照)。

リスト 17. JSON Binding での null のランタイム構成

```
new JsonbConfig().withNullValues(true);
```

Jackson でも、実行時とコンパイル時の両方で null 値を除外する手段を使用できるようになっています。リスト 18 に、`@JsonInclude` アノテーションを使用して null 値を除外する方法を示します。

リスト 18. コンパイル時に null を除外する Jackson のアノテーション

```
@JsonInclude(JsonInclude.Include.NON_NULL)
```

ランタイム構成によって null 値を除外するには、Jackson では 2 つの手段を使用できます。一方の手段では null をグローバルに除外できます。もう一方の手段では、リスト 19 に示すように Map 内から null を除外できます (ただし、この手段は非推奨となっています)。

リスト 19. null を除外する Jackson のランタイム構成

```
new ObjectMapper()
    .setSerializationInclusion(JsonInclude.Include.NON_NULL)
    .configure(SerializationFeature.WRITE_NULL_MAP_VALUES, false);
```

Gson での null のランタイム構成は、null を維持するシリアライズをグローバルに有効にします (リスト 20 を参照)。

リスト 20. null を除外する Gson のランタイム構成

```
new GsonBuilder().serializeNulls();
```

型のシリアライズについても、3 つのフレームワークでの処理方法には一貫性がありません。null の処理に一貫性がないだけでなく、構成手段さえも異なります。

フィールドの可視性

フィールドの可視性によって、そのフィールドが JSON ドキュメントにシリアライズされるかどうか、そしてターゲット・インスタンスにデシリアライズされるかどうかが決まります。しかも、それぞれのフレームワークが可視性を解釈する方法も異なります。

リスト 21 に記載するコードは、3 つすべてのフレームワークを使用してシリアライズされたものです。フィールドは自己記述的な名前になっています。一部のフィールドには対応するセッター/ゲッター・メソッドが定義されていますが (これはフィールド名を見ればわかります)、残りのフィールドはそうではありません。さらに、1 つの仮想フィールドには対応するフィールドがありません。このフィールドは、ゲッター・メソッドによってのみ表現されています。

リスト 21. フィールドの可視性オプションを指定したクラス

```
public class FieldsVisibility {

    // Without getter and setters
    private String privateString = "";
    String defaultString = "";
    protected String protectedString = "";
    public String publicString = "";
    final private String finalPrivateString = "";
    final public String finalPublicString = "";
    static public String STATIC_PUBLIC_STRING = "";

    // With getter and setters. Omitted for brevity
    private String privateStringWithSetterGetter = "";
    String defaultStringWithSetterGetter = "";
    protected String protectedStringWithSetterGetter = "";
    public String publicStringWithSetterGetter = "";
    final private String finalPrivateStringWithSetterGetter = "";
    final public String finalPublicStringWithSetterGetter = "";
    static public String STATIC_PUBLIC_STRING_WITH_SETTER_GETTER = "";
```

```
    public String getVirtualField() {
        return "";
    }
}
```

それぞれのフレームワークがフィールドの可視性にどのように対処するかを比較できるよう、表 5 にリスト 21 のコードをシリアライズしてコンパイルした結果を示します。

表 5. 各フレームワークのデフォルトによるフィールドの可視性の処理方法まとめ

アクセス手段	フィールド修飾子	JSON-B	Jackson	Gson
フィールド	`private`	✗	✗	✓
	`default`	✗	✗	✓
	`protected`	✓	✓	✓
	`public`	✓	✓	✓
	`final public`	✗	✗	✓
	`final private`	✗	✗	✗
	`static public`	✓	✓	✓
public ゲッター	`private`	✓	✓	✓
	`default`	✓	✓	✓
	`protected`	✓	✓	✓
	`public`	✓	✓	✓
	`final private`	✓	✓	✓
	`final public`	✓	✓	✓
	`static public`	✗	✗	✗
	`virtual field`	✓	✓	✗

3 つすべてのフレームワークで唯一一貫しているのは、public の非 static アクセス修飾子に従うという点です。さらに、これらのフレームワークのうち、public static フィールドをシリアライズするものは 1 つもありません。このことは、フィールドに public ゲッター・メソッドが定義されているとしても当てはまります。

明らかに、Gson はフィールドのアクセス修飾子をまったく無視し、指定されている修飾子が何であろうとシリアライズにすべてのフィールドを含めます (ただし、public static フィールドは例外です)。また、Gson は仮想フィールドも除外しますが、JSON-B と Jackson はそうではありません。

JSON-B と Jackson は、フィールドの可視性に対する手法という点ではまったく同じです。Gson は final private フィールドをシリアライズに含め、仮想フィールドを除外するという点で異なります。

可視性の構成

可視性の構成手段は極めて幅広く、また、複雑になりかねません。それぞれのフレームワークが可視性をどのように構成するのかを見ていきましょう。

JSON Binding

JSON-B では、単純な `@JsonbTransient` アノテーションを任意のフィールドに設定することで、そのフィールドをシリアライズとデシリアライズから除外できます。

さらに、JSON-B ではより高度な方法でフィールドの可視性を制御することもできます。それは、`PropertyVisibilityStrategy` インターフェースを実装し、それをランタイム・プロパティーとして `JsonbConfig` インスタンス上に設定して、カスタム可視性ストラテジーを作成するという方法です。リスト 22 に一例を記載します (`PropertyVisibilityStrategy` クラスの実装例については、第 3 回を参照してください)。

リスト 22. JSON-B でカスタム可視性ストラテジーを設定する

```
new JsonbConfig()
    .withPropertyVisibilityStrategy(new CustomPropertyVisibilityStrategy());
```

Jackson

Jackson ではコンパイル時のアノテーションを使用する方法として、`@JsonIgnore` アノテーションを設定したフィールドを無視できるほか、無視するフィールドを明示的に指定したリストをクラス・レベルのアノテーション `@JsonIgnoreProperties({"aField"})` に渡すという方法で可視性を制御することができます。メソッドとフィールドの可視性は、`@JsonAutoDetect` を使用して構成できます (リスト 23 を参照)。

リスト 23. Jackson のアノテーションを使用してフィールドの可視性を設定する

```
@JsonAutoDetect(fieldVisibility = JsonAutoDetect.Visibility.PROTECTED_AND_PUBLIC)
public class CompileTimeSampler {
    // fields and methods omitted for brevity
}
```

Jackson では、ランタイム構成を使用して、フィールドのアクセス修飾子に応じて明示的にフィールドを含めることもできます。リスト 24 に記載するサンプル・コードに、シリアライズの際に public フィールドと protected フィールドを含めるよう指定する方法を示します。

リスト 24. public フィールドおよび protected フィールドを含めるように構成する

```
new ObjectMapper()
  .setDefaultVisibility(
    JsonAutoDetect.Value.construct(PropertyAccessor.FIELD,
      JsonAutoDetect.Visibility.PROTECTED_AND_PUBLIC));
```

Gson

Gson では、コンパイル時のアノテーションを使用して、特定のフィールドをシリアライズおよびデシリアライズ処理に含めるように指定できます。この場合、各フィールドを `@Expose` アノテー

ションでマークして、`serialize` フラグまたは `deserialize` フラグ、あるいはその両方を `true` または `false` に設定します (リスト 25 を参照)。

リスト 25. @Expose アノテーションを使用する

```
@Expose(serialize = false, deserialize = false)
private String aField;
```

@Expose アノテーションを使用するには、このアノテーションを有効にするために、GsonBuilder インスタンスに対して `excludeFieldsWithoutExposeAnnotation()` メソッドを呼び出す必要があります (リスト 26 を参照)。

リスト 26. @Expose アノテーションを有効にする

```
new GsonBuilder().excludeFieldsWithoutExposeAnnotation();
```

別の方法として、フィールドのアクセス修飾子に基づいて明示的にフィールドを除外することもできます (リスト 27 を参照)。

リスト 27. 修飾子に基づいてフィールドを除外する

```
new GsonBuilder().excludeFieldsWithModifiers(Modifier.PROTECTED);
```

プロパティーの順序を構成する

JSON データ交換フォーマット仕様の IETF RFC 7159 では、JSON オブジェクトを「ゼロ以上の名前と値のペアからなる、順不同のコレクション」として定義しています。けれども場合によっては、プロパティーを特定の順序で表示しなければならないこともあります。デフォルトでは、JSON-B は辞書式順序でプロパティーを配置する一方、Jackson と Gson ではクラス内でフィールドが出現する順序を適用します。

プロパティーの順序を指定するには、3 つすべてのフレームワークで共通して、明示的にフィールド名をリストアップするという方法、または順序ストラテジーを指定するという方法を使えます。

JSON Binding

JSON-B ではプロパティーの順序を指定するために、ランタイム・メカニズムと実行時のメカニズムの両方を使用できるようになっています。プロパティーの順序を実行時に指定するには、フィールドを並べたリストを `@JsonbPropertyOrder` アノテーションに渡します (リスト 28 を参照)。

リスト 28. JSON-B でフィールドの順序を明示的に指定する

```
@JsonbPropertyOrder({"firstName", "title", "author"})
```

ランタイム構成を使用する場合は、`JsonbConfig` のインスタンス上に目的の順序ストラテジーを指定するという方法で、**グローバル順序ストラテジー**を設定します。

リスト 29. JSON-B で辞書式順序とは逆の順序を指定する

```
new JsonbConfig()
    .withPropertyOrderStrategy(PropertyOrderStrategy.REVERSE);
```

Jackson

Jackson にもフィールドの順序を明示的に指定する手段があります。さらに、順序をアルファベット順として指定することもできます (リスト 30 を参照)。

リスト 30.Jackson でフィールドの順序を指定する 2 つの方法

```
@JsonPropertyOrder(value = {"firstName", "title", "author"}, alphabetic = true)

@JsonPropertyOrder(alphabetic = true)
```

ランタイム構成を使用して、プロパティーの順序をグローバルに指定することも可能です (リスト 31 を参照)。

リスト 31. Jackson でプロパティーの順序をグローバルに指定する

```
new ObjectMapper().configure(MapperFeature.SORT_PROPERTIES_ALPHABETICALLY, true);
```

Gson

Gson には、プロパティーを簡単に順序付けられる構成設定はありません。したがって、フィールドの順序付けロジックをカプセル化したカスタム・シリアライザー/デシリアライザーを作成しなければなりません。

継承

JSON-B では継承に対処するために、最初に親クラスを配置し、その後に子クラスを続けるという方法を使用します。リスト 32 に `Parent` クラスのコードを記載します。この `Parent` クラスを継承する `Child` クラスのコードをリスト 33 に記載します。

リスト 32. Parent クラス

```
public class Parent {
   private String parentName = "Parent";
   // Getters/setter omitted
}
```

リスト 33. Parent クラスを継承する Child クラス

```
public class Child extends Parent {
   private String child = "Child";
   // Getters/setter omitted
}
```

Jackson でも継承に配慮して、子の前に親を配置しますが、Gson ではこのような順序に従いません。表 6 に、3 つすべてのフレームワークが `Child` クラスのシリアライズを処理する方法を示します。

表 6. 各フレームワークのデフォルトによるフィールドの可視性の処理方法まとめ

JSON Binding	Jackson	Gson
`{ "parentName": "Parent", "child": "Child" }`	`{ "parentName": "Parent", "child": "Child" }`	`{ "child": "Child", "parentName": "Parent" }`

JSON Processing の型に対するサポート

JSON Processing の型 (JSR 374) に対するサポートを提供しているのは、現在のところ JSON Binding API だけです。ただし、Jackson には JSON Processing の型を処理できる拡張モジュールが用意されています。第 1 回で説明したように、JSON Processing の型は JSON 構造をモデル化していることを思い出してください。例えば、`JsonObject` クラスは JSON のオブジェクトをモデル化し、`JsonArray` は JSON の配列をモデル化しています。JSON Binding API にも、JSON 構造を生成するための 2 つのモデルがあります。Jackson と Gson は両方とも、基礎となる Map 実装から直接 JSON Processing インスタンスをシリアライズしようと試みます。

表 7. JSONObject のシリアライズ

JSON Binding	Jackson	Gson
`"jsonObject": { "firstName": "Alex", "lastName": "Theedom" }`	`"jsonObject": { "firstName": { "chars": "Alex", "string": "Alex", "valueType": "STRING" }, "lastName": { "chars": "Theedom", "string": "Theedom", "valueType": "STRING" } }`	`"jsonObject": { "firstName": { "value": "Alex" }, "lastName": { "value": "Theedom" } }`

まとめ

特にカスタマイズを目的とした幅広い機能を提供している Gson と Jackson は、JSON フレームワークとして定着しており、非常によく使われています。けれどもこの記事で明らかにしたように、単純なシナリオを処理する方法という点では、これらのフレームワークに一貫性はほとんどありません。最も単純な null の処理でさえも、フレームワークによって方法は異なります。

型の処理において一貫性があるとしたら、それは他の選択肢がないからでしょう。基本データ型のみ、その実際の値として実際にシリアライズすることができる一方、マッピングとコレクションは、それぞれに対応する JSON オブジェクトに直接変換されます (その逆も当てはまります)。最も明確に一貫性が欠けているのは、日付と時刻の処理です。これらの型を同じようにシリアライズするフレームワークは 1 つとしてありません。形式を制御する構成手段はあるとはいえ、ここまで大きな矛盾があるということは、標準の必要性を示唆しています。

開発者が機能と引き換えにパフォーマンスに妥協せざるを得ないようであってはなりません。そのような状況は、JSON バインディング標準によって解決されます。JSON バインディング標準の採用は JSON が次に進むべきステップであり、その標準としては JSR 367 が最適であると私は確

信していますが、皆さんはどう思いますか？この記事の最後に記載されているリンク先のページで、ご意見を聞かせてください。

関連トピック

- JSON Binding のホームページ
- JSON Binding 仕様のコードベース
- JSR 367: Java API for JSON Binding (JSON-B)
- Yasson: JSON-B のリファレンス実装
- Documentation for Jackson
- User guide for Gson
- Java EE 8 の新機能