JSON Binding API 入門, 第 2 回: JSON-B でのデフォルト・マッピング

JSON Binding API を使用した日常的なシリアライズとデシリアライズ

Alex Theedom

2018年 8月 23日

新たな JSON Binding API は Java と JSON をシームレスにバインドし、多くの単純な使用ケースにはデフォルト・マッピングで対応できるようになっています。JSON-B のデフォルト構成はほとんどの開発者にとって直感的なものであり、シリアライズおよびデシリアライズの幅広い手段とシナリオを網羅しています。

この連載の第 1 回では、Java API for JSON Binding 1.0 (JSON-B) の概要を、カスタムとデフォルトのバインディング手段をまじえて紹介しました。今回の記事では、多くの Java データ型 (特殊なデータ型を含む) をシリアライズおよびデシリアライズする際の JSON-B のデフォルト・マッピングと自動動作について紹介します。

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。

デフォルト・マッピングと内部構造

JSON-B API ではフィールドとプロパティーのマッピングに、JSON バインダーを使った経験があれば簡単に理解できる、お馴染みの手法を採用しています。アノテーションやカスタム構成を使用しなくても、JSON-B のマッピングは一連のルールを適用します。これらのルールは最初から使えるようになっているため、開発者は JSON ドキュメントの構造どころか JSON データ交換フォーマット自体について理解していないとしても、すぐに JSON-B のマッピングを使い始めることができます。

JSON-B API にはエントリー・ポイント・インターフェースとして、`javax.json.bind.Jsonb` と `javax.json.bind.JsonBuilder` の 2 つが用意されています。`Jsonb` インターフェースでは、オーバーライドした `toJson()` メソッドと `fromJson()` メソッドにより、それぞれシリアライズ、デシリアライズの機能を提供します。`JsonbBuilder` インターフェースは、一連のオプション構成を使用して `Jsonb` インスタンスを作成し、そのインスタンスにクライアントがアクセスできるようにします。

JSON-B は JSON Processing API が提供する機能に大きく依存することも、注意すべき重要な点です。JSON-B は JSON-P のストリーミング・モデルを利用して、`javax.json.stream.JsonGenerator` で (`toJson()` メソッドを介して) JSON データを出力ソースに書き出せるようにしているだけでなく、`javax.json.stream.JsonParser` で (`fromJson()` メソッドを介して) JSON データへの順方向の読み取り専用アクセスを可能にしています。

Yasson をインストールする

JSON-B を使い始めるには、そのリファレンス実装である Eclipse Yasson が必要です。Eclipse Yasson は、このリンク先の Maven 中央リポジトリーから入手することができます。以下に示すように、Maven の JSON-B 用 POM 依存関係リストに、JSON Processing API を追加する必要があります。

リスト 1. Maven による調整

```
<dependency>
   <groupId>javax.json</groupId>
   <artifactId>javax.json-api</artifactId>
   <version>1.1</version>
</dependency>


<!-- JSON-P 1.1 RI -->
<dependency>
   <groupId>org.glassfish</groupId>
   <artifactId>javax.json</artifactId>
   <version>${javax.json.version}</version>
</dependency>


<!-- JSON-B 1.0 API -->
 <dependency>
     <groupId>javax.json.bind</groupId>
     <artifactId>javax.json.bind-api</artifactId>
     <version>1.0</version>
 </dependency>


<!-- JSON-B 1.0 RI -->
<dependency>
   <groupId>org.eclipse</groupId>
   <artifactId>yasson</artifactId>
   <version>1.0</version>
</dependency>
```

最も単純な例

早速、単純な例として、リスト 2 に記載する `Book` クラスのインスタンスを、シリアライズとデシリアライズの両方向で変換しましょう。

リスト 2. Book オブジェクト

```
public class Book {
   public String title;
}
```

シリアライズまたはデシリアライズを開始するには、`Jsonb` のインスタンスが必要です。それには、`JsonBuilder` インターフェースの static ファクトリー・メソッド `create()` を呼び出して Jsonb

インスタンスを作成します。このインスタンスがあれば、多重定義された `toJson()` メソッドまたは `fromJson()` メソッドのいずれか該当するほうを選択することで、シリアライズとデシリアライズを実行できます。

リスト 3 では、最も単純な形の `toJson()` メソッドを呼び出して `Book` オブジェクトを渡しています。

リスト 3. Book インスタンスのシリアライズ

```
Book book = new Book("Fun with Java"); // Sets the title field
String bookJson = JsonbBuilder.create().toJson(book);
```

このメソッドの値は、`toJson()` に渡されたオブジェクトの JSON データ表現を格納する `String` です (リスト 4 を参照)。

リスト 4. Book インスタンスの JSON 表現

```
{
  "title": "Fun with Java"
}
```

次は、デシリアライズの処理に目を向けましょう。デシリアライズの処理も、シリアライズの場合と同じく単純です。デシリアライズにも、`Jsonb` のインスタンスが必要になります。リスト 5 では、最も単純な形の `fromJson()` を呼び出して、このメソッドに、ターゲットの型と併せてデシリアライズ対象の JSON データを渡しています。

リスト 5. Book の JSON 表現のデシリアライズ

```
String json = "{\"title\":\"Fun with Java\"}";
Book book = JsonbBuilder.create().fromJson(json, Book.class);
```

以上の例で使用した `toJson()` と `fromJson()` は、`Jsonb` 上で使用できる最も単純な多重定義メソッドのうちの 2 つです。次は、このインターフェースに含まれる他のより複雑なメソッドのいくつかを見ていきましょう。

変換の前後での等価性
JSON-B 仕様では変換の前後で等価性を維持することを要件としていませんが、ほとんどの場合は、等価性が維持されることがわかるでしょう。この連載全体を通して、この通則に対する例外をいくつか特定し、それらの例外に対処する方法を紹介します。

多重定義されたメソッド

`Jsonb` インターフェースに用意されている、多重定義された `toJson()` メソッドと `fromJson()` は、それぞれ JSON-B のシリアライズ機能、デシリアライズ機能を実行します。表 1 に、これらの多重定義された 12 のメソッドと、それぞれのシグニチャーをリストアップします。

表 1. Jsonb インターフェース・メソッド

修飾子と型	メソッド
<T> T	fromJson(InputStream stream, Class<T> type)
<T> T	fromJson(InputStream stream, Type runtimeType)

<T> T	fromJson(Reader reader, Class<T> type)
<T> T	fromJson(Reader reader, Type runtimeType)
<T> T	fromJson(String string, Class<T> type)
<T> T	fromJson(String string, Type runtimeType)
String	toJson(Object object)
void	toJson(Object object, OutputStream stream)
void	toJson(Object object, Writer writer)
String	toJson(Object object, Type runtimeType)
void	toJson(Object object, Type runtimeType, OutputStream stream)
void	toJson(Object object, Type runtimeType, Writer writer)

上記の表を見るとわかるように、ほとんどの使用ケースのシナリオには、この幅広い一連のメソッドで対応できます。特に、`InputStream` インスタンスまたは `OutputStream` インスタンスに接続できるメソッドはとりわけ有用なソリューションになります。シリアライズ処理とデシリアライズ処理のどちらであるかに応じて、このメソッドは `Stream` インスタンスを受け入れて Java オブジェクトを出力するか、Java オブジェクトを `Stream` インスタンスに書き出します。

`InputStream` と `OutputStream` の組み合わせは、さまざまな独創的な方法で JSON データをストリーミングすることを可能にします。一例として、JSON データを格納し、シリアライズとデシリアライズの両方の処理を行うフラット・ファイルがあるとします。リスト 6 では `toJson(Object object, OutputStream stream)` メソッドを呼び出して、`Book` クラスのインスタンスをシリアライズし、テキスト・ファイル `book.json` に出力しています。

リスト 6. JSON シリアライズをテキスト・ファイル OutputStream に送信する

```
Jsonb jsonb = JsonbBuilder.create();
jsonb.toJson(book, new FileOutputStream("book.json"));
```

リスト 7 では、デシリアライズ処理を行うために、`fromJson(InputStream stream, Class<T> type)` メソッドを呼び出しています。

リスト 7. テキスト・ファイル内に格納された JSON データをデシリアライズする

```
Book book = jsonb.fromJson(new FileInputStream("book.json"), Book.class);
```

パフォーマンスと再利用

Jsonb インターフェースに関する Java ドキュメントに、この記事で取り上げるメソッドの処理について、他の例とともに詳細な説明が記載されています。さまざまなアドバイスのうち、このドキュメントでは特に、`JsonbBuilder` と `Jsonb` のインスタンスを再利用することが最適な使用法であり、通常の使用ケースではアプリケーションごとに 1 つの `Jsonb` インスタンスだけを使用すればよいとアドバイスしています。1 つの `Jsonb` インスタンスだけを使用できる理由は、Jsonb インターフェースのメソッドがスレッドセーフであることです。

Java 基本データ型に対するデフォルト

Java 基本データ型のシリアライズとデシリアライズに対して JSON-B が適用するルールは単純なものです。Java 基本データ型には、`String` 型、`Boolean` 型、そして `Character` や `AtomicInteger` などのすべての `Number` 型とそれぞれに対応するプリミティブ型が含まれます。

プリミティブ型とそのラッパーとの等価性を維持するために、JSON-B は `toString()` メソッドを使用して、JSON の `Number` に変換される `String` を生成します。`toString()` メソッドを呼び出す `Number` 型の中には、`String` と同等の要素を生成しないものがあります (`java.util.concurrent.atomic.LongAdder` など)。そのような場合は、`doubleValue()` メソッドを呼び出すことで、double プリミティブ型を生成できます。

Number 型の例外処理

符号化方式
`toJson()` のシリアライズ処理で使用されるデフォルトの符号化方式は UTF-8 です。`fromJson()` によるデシリアライズでは、符号化方式は JSON データから自働的に検出され、それが RFC 7159 で定義されている有効な文字符号化方式であると見なされます。必要に応じて符号化をカスタマイズすることも可能です。

JSON-B 仕様では、ラッパー・タイプを除くすべての `Number` 型については `BigDecimal` をインスタンス化することを要件としています。デシリアライズの際は、デシリアライズ処理で `BigDecimal` のインスタンスを作成するために、JSON の数値を `String` としてコンストラクターに渡します (例えば、`new BigDecimal("10")` のようにします)。けれども、ターゲットの型が `BigDecimal` ではなく、他の多数の `Number` 型のうちの 1 つ (`AtomicInteger` など) である場合は、この仕組みが問題になります。

ほとんどの基本型については、デシリアライズ処理によって、該当する `parse$Type` メソッドが自動的に呼び出されます (例えば、(`Integer.parseInt("10")` や `Boolean.parseBoolean("true")` といった具合です)。ラッパー・タイプではない `Number` 型では `parse$Type` メソッドが使われないこともあります。この場合、デシリアライザーが `parse$Type` メソッドを呼び出そうとすると、例外がスローされます。

この仕組みを確認するには、まずはリスト 8 に記載する単純化した `Book` クラスを見てください。

リスト 8. AtomicInteger を使用した SimpleBook クラス

```
public class SimpleBook {
   private AtomicInteger bookVersion;
   // Getters and setters omitted for brevity
}
```

リスト 9 に示されているように、この JSON データには `Integer` 型の数値が 1 つだけ含まれます。

リスト 9. SimpleBook の JSON 表現

```
{
    "bookVersion":10
}
```

JSON-B がこの `SimpleBook` の JSON 表現を再び `SimpleBook` に変換しようとするときに、デシリアライズ処理ではターゲット・フィールドをその名前 (`bookVersion`) で識別します。したがって、`bookVersion` はラッパー・タイプのグループには分類されない `Number` 型であると判別し、`BigDecimal` のインスタンスを作成します。

`BigDecimal` 型には `AtomicInteger` 型との互換性がありません。そのため、デシリアライズ処理によって「`IllegalArgumentException` (引数の型不一致)」というメッセージとともに

`IllegalArgumentException` がスローされることになります。この例外を発生させるのが、リスト 10 のコードです。

リスト 10. IllegalArgumentException のスロー

```
Jsonb jsonb = JsonbBuilder.create();
jsonb.fromJson(
       jsonb.toJson(new SimpleBook(new AtomicInteger(10))),
       SimpleBook.class);
```

要するに、変換前後での等価性は、プリミティブ型とそのラッパーに相当する要素だけでなく、`BigDecimal` と `BigInteger` についても維持されるということです。他の `Number` 型は、そのままでは、変換前後での等価性をサポートすることはありません。この問題は、JSON-B アダプターを使用することによって解決できます。その方法については、第 3 回で紹介します。

JSON-B での null 値

null 値を持つフィールドをシリアライズすると、そのフィールドのプロパティーは、出力される JSON ドキュメントから除外されることに注意してください。デシリアライズの際は、プロパティーが存在しないためにターゲット・プロパティーの値が null に設定されることはありません。また、プロパティーが存在しなければ、セッター・メソッド (あるいは public フィールドのセット) も呼び出されません。プロパティーの値は変更されずに維持されて、ターゲット・クラス内ではデフォルト値を設定できます。

標準的な Java データ型に対するデフォルト

JSON-B では、`BigDecimal`、`BigInteger`、`URL`、`URI`、`Optional`、`OptionalInt`、`OptionalLong`、`OptionalDouble` を含め、標準的な Java SE データ型をサポートしています。

前に説明したように、`Number` 型をシリアライズするには `toString()` メソッドを呼び出し、デシリアライズするには該当するコンストラクターまたはファクトリー・メソッドを使用します。URL 型と URI 型のシリアライズの動作も、同じように `toString()` を呼び出すことによって行われます。デシリアライズを行うには、該当するコンストラクターまたは static ファクトリー・メソッドを使用します。

`Optional` の値をシリアライズするには、その内部インスタンスを取得して、その型に適した方法 (通常は、`toString()` を呼び出すという方法) でインスタンスを JSON に変換します。リスト 11 では、`Integer` の `toString()` を呼び出すことによって、`Integer` の `Optional` をシリアライズします。

リスト 11. Optional<Integer> インスタンスのシリアライズ

```
public class OptionalExample {
   public Optional<Integer> value;
   // Constructors, getters and setters omitted for brevity
}


JsonBuilder.create().toJson(new OptionalExample(10))
```

リスト 12 に、上記のコードによって出力された JSON の数値を記載します。

リスト 12. OptionalExample の JSON 表現

```
{
  "value": 10
}
```

JSON の値は、ターゲット・フィールドの型に応じて `Optional<T>` または `OptionalInt/Long/Double` のいずれか該当する値にデシリアライズされます。

リスト 13 に、JSON の数値を `OptionalInt` インスタンスにデシリアライズする例を記載します。

リスト 13. JSON 数値の OptionalInt へのデシリアライズ

```
JsonBuilder.create().fromJson("{\"value\":10}", OptionalIntExample.class)
```

JSON データは `OptionalIntExample` クラス (リスト 14 を参照) にデシリアライズされます。このクラスには、value という名前の単一の `OptionalInt` フィールドがあります。デシリアライズ後の value フィールドには値 `OptionalInt.of(10)` が格納されます。

リスト 14. OptionalIntExample クラス

```
public class OptionalIntExample {
   public OptionalInt value;
   // Constructors, getters and setters omitted for brevity
}
```

以上の説明からわかるように、シリアライズでもデシリアライズでも、`Optional` クラスは完全にサポートされています。

空の `Optional` のシリアライズは、基本型をシリアライズする際に null を処理する方法と同じように処理されます。つまり、デフォルトでは空の `Optional` は維持されません。デシリアライズでは、null 値は `Optional.empty()` (または `Optional/Int/Long/Double.empty()`) インスタンスとして表現されますが、それとは反対に、配列およびマップ・データ構造内に格納された空の `Optional` は、出力される JSON 配列内で null としてシリアライズされます。

特殊なデータ型に対するデフォルト

次は、特殊なデータ型に対するデフォルトの動作を見ていきましょう (第 3 回で、`@JsonbNillable` マッピング・アノテーションを使用して null の処理をカスタマイズする方法を説明します)。

列挙型

列挙型の値のシリアライズでは、`name()` メソッドを呼び出して、このメソッドから列挙型定数の `String` 表現を取得します。このことは、特にデシリアライズ処理では `valueOf()` メソッドを呼び出してこのプロパティー値を渡すことから重要になります。列挙型では、変換前後での等価性が維持されます。

Date 型

JSON Binding API は、(JDK 1.1 で導入された) `java.util` パッケージに含まれる古い `Date` クラスから生成される日時インスタンスも、新しい Java 8 の `java.time` パッケージに含まれる日付クラスと時刻クラスから生成される日時インスタンスもサポートしています。

デフォルトで適用されるタイム・ゾーンは GMT グリニッジ標準時です。オフセットは UTC グリニッジとして指定されます。日時形式はオフセットなしの ISO 8601 です。これらのデフォルトは、第 3 回で紹介するカスタマイズ方法を使用してオーバーライドできます。

非推奨となった 3 文字のタイム・ゾーン ID を除き、タイム・ゾーン・インスタンスは完全にサポートされています。リスト 15 と 16 に、日時をシリアライズする 2 つの例を記載します。

リスト 15. 従来の Date インスタンスのシリアライズ

```
SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
Date date = sdf.parse("25/12/2018");
String json = jsonb.toJson(date);


Serialize to JSON: "2018-12-25T00:00:00Z[UTC]"
```

リスト 16. Java 8 の Duration インスタンスのシリアライズ

```
Duration.ofHours(4).plusMinutes(3).plusSeconds(2)


Serialize to JSON: "PT4H3M2S"
```

それぞれの型で使用される形式の構成要素については、JSON-B 仕様のドキュメント (セクション 3.5) を参照してください。

配列とコレクション型

コレクションと配列のすべての型はサポートされていて JSON 配列にシリアライズされる一方、`Map` 型は JSON のマッピングにシリアライズされます。リスト 17 に、`Map` をシリアライズする例を示します。この例でシリアライズされた JSON 表現を、リスト 18 に記載します。

リスト 17. String と Integer のマッピング

```
new HashMap<String, Integer>() {{
   put("John", 10);
   put("Jane", 10);
}};
```

リスト 18. Map の JSON 表現

```
{
  "John": 10,
  "Jane": 10
}
```

`Collection` または `Map` がクラスのメンバーになっている場合、生成される JSON オブジェクトでは、そのフィールド名がプロパティー名として使用されます (リスト 19 とリスト 20 を参照)。

リスト 19. クラスのメンバーとしての Map

```
public class ScoreBoard {
   public Map<String, Integer> scores =
new HashMap<String, Integer>() {{
       put("John", 12);
       put("Jane", 34);
   }};


}
```

リスト 20. JSON 表現

```
{
 "scores": {
   "John": 12,
   "Jane": 34
 }
}
```

JSON-B では多次元プリミティブ型と Java 型の配列もサポートしています。

コレクションとマッピングでの null 値

配列およびマッピング構造に含まれる null 値については、JSON-B は慣例に反して、シリアライズとデシリアライズの両方で null を維持します。つまり、空の要素は null にシリアライズされ、null は空の要素にデシリアライズされます。空の `Optional` 値は、出力される JSON 配列内で null としてシリアライズされ、`Optional.empty()` インスタンスとしてデシリアライズされます。

型付けされていないマッピング

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。

これまでに見てきたように、JSON プロパティー・データはターゲット・クラス内の対応するフィールドの Java 型にデシリアライズされますが、出力の型が指定されていない場合、あるいは `Object` として指定されているときは、JSON の値は対応するデフォルトの Java の型にデシリアライズされます。表 2 に、このような場合にデフォルトでデシリアライズされる Java データ型を記載します。

表 2. デフォルトでデシリアライズされる型

JSON の値	Java データ型
オブジェクト	java.util.Map<String, Object>
配列	java.util.List<Object>
ストリング	java.lang.String
数値	java.math.BigDecimal
true、false	java.lang.Boolean
null	null

リスト 21 に、実際にこれがどのような意味であるのかを説明するために、JSON オブジェクトが Java の `Map` インスタンスにデシリアライズされる例を示します。

リスト 21. Java の Map にデシリアライズされる JSON オブジェクト

```
String json = "{
\"title\":\"Fun with Java\",
\"price\":24.99,
\"issue\":null}";
Map<String, Object>() map =
JsonbBuilder.create().fromJson(json, Map.class);
```

Java クラスのシリアライズとデシリアライズ

デシリアライズ処理では、ターゲット・クラスに引数なしの public または protected コンストラクターがあること、あるいはターゲット・クラスがカスタム・オブジェクトを作成する際に使用するメソッドを明示的に指定していることが要件となります。オブジェクトの作成に使用するメソッドがなければ、`javax.json.bind.JsonbException` がスローされることになります。シリアライズについては、このような要件はありません。

JSON-B では、public クラス、protected クラス、protected-static クラスをデシリアライズできますが、匿名クラスはサポートされていません。匿名クラスをシリアライズすると、JSON オブジェクトが生成されます。前述のインターフェースを除き、デシリアライズ時にサポートされるインターフェースはありません。シリアライズの際は、ランタイム型が使用されます。

public アクセスおよび protected アクセスが設定されたクラスに対しては、シリアライズとデシリアライズの両方で、ネストされたクラスと静的にネストされたクラスがサポートされます。

JSON-P 型のサポート

JSON-B は JSON Processing API に基づいて作成されているため、JSON-P のすべての型をサポートしています。これらの型には、`javax.json` パッケージに含まれている型と、`JsonValue` のすべてのサブ型が含まれます。リスト 22 に `JsonArray` のインスタンスをシリアライズする例を示し、この例でシリアライズされた JSON 配列をリスト 23 に記載します。

リスト 22. JSON Value のシリアライズ

```
JsonArray value = Json.createArrayBuilder()
       .add(Json.createValue("John"))
       .add(JsonValue.NULL)
       .build();
```

リスト 23. JSON Value インスタンスの JSON 表現

```
[
  "John",
  null
]
```

プロパティーの順序

デフォルトでは、プロパティーは辞書式順序でシリアライズされます。リスト 24 に記載するクラスのインスタンスは、リスト 25 に記載する JSON ドキュメントとしてシリアライズされます。

リスト 24. 辞書式順序でフィールドが配列されていないクラス

```
public class LexicographicalOrder {
   public String dog = "Labradoodle";
   public String animal = "Cat";
   public String bread = "Chiapata";
   public String car = "Ford";
}
```

リスト 25. 辞書式順序で配列されたプロパティーを示しているシリアライズ

```
{
  "animal": "Cat",
  "bread": "Chiapata",
  "car": "Ford",
  "dog": "Labradoodle"
}
```

JSON-B でのプロパティーの順序ストラテジーは、カスタマイズ手段を使って構成することができます。これについては、第 3 回で詳しく説明します。

デフォルトのアクセス・ストラテジー

第 1 回で簡単に説明したように、JSON-B でのデフォルトのフィールド・アクセス・ストラテジーでは、メソッドまたはフィールドが public アクセス修飾子を指定してパブリック・アクセスを許可することが要件となっています。シリアライズの際は、フィールドの値を取得するために、そのフィールドの public ゲッター・メソッドが呼び出されます。そのようなゲッター・メソッドが存在しない場合 (あるいは、パブリック・アクセスが許可されていない場合) は、フィールドに直接アクセスすることになります。ただし、この動作が行われるのは、アクセス修飾子が public の場合のみです。

同様に、デシリアライズでプロパティーの値を設定するには、セッター・メソッドへのパブリック・アクセスが可能でなければなりません。セッター・メソッドが存在しないか public でない場合は、public フィールドが直接設定されます。

JSON-B のフィールド・アクセス・ストラテジーを構成して、より制約的なものにすることもできます。それには、関連するカスタマイズ手段を微調整します。フィールド・アクセス・ストラテジーの構成やその他のカスタマイズについては、第 3 回で詳しく説明します。

まとめ

単純な使用ケースのほとんどのシナリオには、JSON Binding API をそのままの形で使用して、賢明なデフォルトを設定できます。他の何らかの JSON バインディング・テクノロジーを使い慣れている開発者にとって、JSON-B は直感的に使用できます。また、JSON-B は使いやすいため、JSON を使用するのが初めての開発者でも簡単に理解できます。

この連載のこれまでの記事では、JSON-B を実際に使って、その主要なデフォルトの特長と機能を見てきました。JSON Binding の真の力を活用するためには、これらのデフォルトの処理をカスタマイズしなければならないこともあります。第 3 回では、JSON-B のカスタマイズ・モデルと機能を取り上げ、コンパイル時のアノテーションとランタイム構成を使用してこの API をカスタマイ

ズする方法を説明します。JSON-B に用意されているカスタマイズ手段を使用すれば、下位レベルのシリアライズとデシリアライズ処理をはじめ、この API のほぼあらゆる側面をカスタマイズすることができます。

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
- Java EE 8 の新機能