JSON Binding API 入門, 第 3 回: JSON-B でのカスタム・バインディング

アノテーションとランタイム構成を使用して JSON バインディングをカスタマイズする

Alex Theedom

2018年 8月 23日

JSON Binding API では、データのシリアライズとデシリアライズを簡単かつ直感的にカスタマイズできるようになっているため、開発者はかなりの力を手にすることができます。この記事では、アノテーションとランタイム構成をそれぞれ単独で、あるいは組み合わせて使用して、プロパティー、フィールド、日付形式と時刻形式などの要素のバインディングおよび表示を制御する方法を説明します。さらに、JSON-B での処理ロジックを変更するために、JSON-B のアダプターと下位レベルのシリアライザー/デシリアライザーをどのようにして導入するのかも説明します。

この連載の第 1 回では、JSON-B の概要を説明しました。その際に、アノテーションとランタイム構成を使用してバインディングの動作をカスタマイズする方法も紹介しました。今回の記事では、この 2 つのメカニズムについてさらに詳しく、実際的な使い方という点から紹介します。アノテーションとランタイム構成をそれぞれ単独で使用する方法、組み合わせて使用する方法、そしてカスタム・ソリューション内でのこの 2 つのメカニズムの相互作用を把握する方法を学んでください。さらにこの記事では、JSON-B での処理ロジックを変更するために、JSON-B のアダプターと下位レベルのシリアライザー/デシリアライザーをどのようにして導入するのかも紹介します。

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。

JSON-B をカスタマイズする 2 つの手段 (復習)

Java エンタープライズ・アプリケーション内での JSON バインディングをカスタマイズするには、アノテーションとランタイムを組み合わせて使用することも、それぞれ単独で使用することもできます。第 1 回をすでに読み終わっているとしたら、以下に記載するサンプル・コードで、アノテーション・モデルとランタイム構成モデルの仕組みを思い出してください。

コンパイル時のアノテーション

JSON-B には、数々の組み込みアノテーションが用意されています。これらのアノテーションを使用して、カスタム動作を有効にするフィールド、パラメーター、またはセッター/ゲッターをマークすることができます。リスト 1 に一例を記載します。

リスト 1. アノテーション・モデルの使用例

```
public class Book {
  @JsonbProperty("isbn")
  public String id;
}
```

上記の例では、シリアライズされた JSON ドキュメントに含まれる `id` フィールドの名前が `isbn` に変更されることになります。シリアライズされた JSON ドキュメントを `Book` インスタンスに正常にデシリアライズするには、JSON ドキュメントに `isbn` という名前のプロパティーが含まれていなければなりません。

ランタイム構成

アノテーションではなくランタイム構成を使用してカスタマイズしたいという場合は、構成ストラテジーを作成し、シリアライズとデシリアライズを実行するために使用する `Jsonb` インスタンスにその構成ストラテジーを渡します。リスト 2 に、この手法を示します。

リスト 2. ランタイム・モデルの使用例

```
JsonbConfig jsonbConfig = new JsonbConfig()
   .withPropertyNamingStrategy(
     PropertyNamingStrategy.LOWER_CASE_WITH_DASHES);

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

リスト 2 を見るとわかるように、プロパティーの命名ストラテジーを `JsonbConfig` インスタンス上に設定した後、このインスタンスをファクトリー・メソッドに渡します。これにより、`JsonBuilder` インスタンスが作成されます。このように作成された `Jsonb` インスタンスをシリアライズまたはデシリアライズに使用すると、このインスタンス上に設定されたプロパティーの命名ストラテジーが使用されます。この例の場合、プロパティー名が小文字に変換されて、名前を構成する各単語がダッシュで区切られます。

プロパティー名をカスタマイズする

プロパティーの名前は、そのプロパティーに対応する Java Bean のフィールド名によって決まります。シリアライズの際は、出力される JSON ドキュメント内でフィールド名がプロパティー名として使用されます。デシリアライズの際は、プロパティー名がフィールド名と照合されて、一致するフィールドのセッターが呼び出されるか、public フィールドが設定されます。

場合によっては、解決しようとしている問題にデフォルトの命名ストラテジーでは対応しきれないこともあります。例えば、JSON ドキュメントで使用している命名規則が企業で使用している命名規則とは異なる場合や、ビジネス目標の構造には一致しない JSON ドキュメント形式にデータをエクスポートしなければならない場合などです。

どのようなシナリオであれ、シリアライズとデシリアライズの柔軟性を向上させるには、プロパティー名をカスタマイズすることが 1 つの方法となります。プロパティー名をカスタマイズするには、1 つの方法として、フィールド、パラメーター、ゲッター、またはセッターに対して `@JsonbProperty` アノテーションを設定します。この場合、アノテーションに、カスタマイズした名前を値として渡します。以下の規則に注意してください。

- フィールド上にアノテーションを設定すると、カスタマイズした名前がシリアライズとデシリアライズの両方で使用されます。
- セッター・メソッド上にアノテーションを設定すると、カスタマイズした名前はデシリアライズの際にだけ使用されます。
- ゲッター・メソッド上にアノテーションを設定すると、カスタマイズした名前はシリアライズの際にだけ使用されます。

リスト 3 に、前回の記事で使用した `Book` クラスの `id`、`title`、`author` の各フィールドに対してプロパティー名をカスタマイズする例を記載します。

リスト 3. @JsonbProperty アノテーションを使用した Book クラス

```
public class Book {

  @JsonbProperty("isbn")
  private String id;
  private String title;
  private String author;

  @JsonbProperty("bookTitle")
  public String getTitle() {
      return title;
  }

  @JsonbProperty("authorName")
  public void setAuthor(String author) {
      this.author = author;
  }

  // other getters and setters omitted for brevity
}
```

`Book` インスタンスをシリアライズすると、リスト 4 に記載するような JSON ドキュメントになります。

リスト 4. JSON ドキュメントにシリアライズされた Book インスタンス

```
{
 "author": "Alex Theedom",
 "bookTitle": "Fun with JSON-B",
 "isbn": "ABC-123-XYZ"
}
```

リスト 4 の JSON ドキュメントでは、`author` フィールドの名前が変更されていないことに注意してください。これは、リスト 3 の `Book` クラスでは、セッター・メソッドにしかアノテーションが設定されていないためです。その結果、デシリアライズの際には `@JsonbProperty` とゲッター・メソッドしか呼び出されません。リスト 4 の JSON ドキュメントを `Book` インスタンスに正常にデシリアライズするためには、`author` フィールドを手作業で編集して `authorName` に変更する必要があります。

シリアライズとデシリアライズでそれぞれ異なるカスタム名を使用することや、同じ POJO 内にある複数のフィールド、ゲッター、セッターにそれぞれ異なるカスタム名を使用することも可能です。このように異なるカスタム名を使用する場合は、ゲッターとセッターに対するカスタマイズが、フィールドに対するカスタマイズよりも優先されることに注意してください。

プロパティーの命名ストラテジーをカスタマイズする

フィールド名がどのようにプロパティーに変換されるかは、プロパティーの命名ストラテジーによって決まります。デフォルトでは、フィールド名は変更されないままプロパティーに変換されますが、この動作はカスタマイズできます。例えば、プロパティー名の区切り文字に下線を使ったり、プロパティー名を小文字に変換したりできます。表 1 (以下を参照) に、すべてのプロパティーの命名ストラテジーがリストアップされています。

プロパティーの命名ストラテジーを構成するには、ランタイム・モデルを使用して、`JsonbConfig` インスタンス上に該当する値を設定します。リスト 5 に、その方法を示します。

リスト 5. プロパティーの命名ストラテジーをカスタマイズする

```
public class Magazine {
  private String title;
  private Author authorName;
  private String internalAuditCode;
  private String alternativetitle;
  // getters and setters omitted for brevity
}

Magazine magazine =
  new Magazine("Fun with JSON binding",
               new Author("Alex", "Theedom"),
               "ABC-123","Fun with JSON-B
               ");

JsonbConfig jsonbConfig = new JsonbConfig()
      .withPropertyNamingStrategy(
           PropertyNamingStrategy.LOWER_CASE_WITH_DASHES);

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

リスト 5 で使用しているストラテジーでは、すべてのフィールド名が小文字に変換され、名前を構成する各単語がダッシュで区切られます。リスト 6 に、その出力を記載します。

リスト 6. LOWER_CASE_WITH_DASHES シリアライズ・ストラテジー

```
{
  "alternativetitle": "Fun with JSON-B",
  "author-name": {
    "first-name": "Alex",
    "last-name": "Theedom"
  },
  "title": "Fun with JSON binding"
}
```

プロパティーの命名ストラテジーには、全部で 6 つの選択肢があります。表 1 に、これらのストラテジーを要約します。

表 1. プロパティーの命名ストラテジー

ストラテジー	説明	Original	変換前
IDENTITY	The name is unchanged (default strategy).	authorName	authorName
LOWER_CASE_WITH_DASHES	名前は小文字に変換され、名前を構成する各単語がダッシュで区切られます。	authorName alternativetitle	author-name alternativetitle
LOWER_CASE_WITH_UNDERSCORES	名前は小文字に変換され、名前を構成する各単語が下線で区切られます。	authorName alternativetitle	author_name alternativetitle
UPPER_CAMEL_CASE	名前を構成する各単語の先頭文字が大文字に変換されます。	authorName alternativetitle	AuthorName Alternativetitle
UPPER_CAMEL_CASE_WITH_SPACES	単語の先頭文字が大文字にされて、名前を構成する各単語がスペースで区切られます。	authorName alternativetitle	Author Name Alternativetitle
CASE_INSENSITIVE	シリアライズには IDENTITY ストラテジーを使用しますが、デシリアライズではプロパティー名の大文字/小文字を区別しません。	AuthorNAME maps to authorName	

ここでも同じく注意する点として、`@JsonbProperty` アノテーションを使用して構成されたフィールドが、`JsonbConfig` 上に指定されたランタイム構成による命名ストラテジーよりも優先されます。

プロパティーの順序ストラテジーをカスタマイズする

JSON ドキュメント内でプロパティーが出現する順序としては、デフォルトでは辞書式順序が採用されます。表 2 に、この辞書式順序と他の 2 つのプロパティーの順序ストラテジーを記載します。

表 2. デフォルトおよびカスタムのプロパティーの順序ストラテジー

ストラテジー	説明
LEXICOGRAPHICAL	プロパティーは辞書式順序で並べられます (デフォルトのストラテジー)。
ANY	プロパティーに保証された順序はありません。
REVERSE	プロパティーは辞書式順序とは逆順に並べられます。

ランタイム・モデルを使用してプロパティーの順序ストラテジーを構成する場合は、適用するストラテジーを `JsonbConfig` インスタンス上に設定します。リスト 7 では、辞書式順序とは逆の順序にプロパティーを並べるストラテジーを設定しています。

リスト 7. プロパティーの順序ストラテジー REVERSE を使用したシリアライズ

```
Magazine magazine =
    new Magazine("Fun with JSON binding",
        new Author("Alex", "Theedom"),
        "Fun with JSON-B");

JsonbConfig jsonbConfig = new JsonbConfig()
      .withPropertyOrderStrategy(PropertyOrderStrategy.REVERSE);

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

リスト 8 に、このストラテジーが適用された JSON 出力を記載します。

リスト 8. プロパティーの順序ストラテジー REVERSE が適用された出力

```
{
 "title": "Fun with JSON binding",
 "authorName": {
   "lastName": "Theedom",
   "firstName": "Alex"
 },
 "alternativeTitle": "Fun with JSON-B"
}
```

別の方法としては、`@JsonbPropertyOrder` アノテーションを使用して、目的とするプロパティーの順序を明示的に指定することもできます。この場合、リスト 9 に示されているように、プロパティー名のリストを `@JsonbPropertyOrder` アノテーションに渡します。このリスト上に指定されていないプロパティーは、順序が明示的に指定されたプロパティーの後に、辞書式順序でリストアップされます。

リスト 9. @JsonbPropertyOrder を使用した Book インスタンスの出力

```
@JsonbPropertyOrder(value = {"id", "title", "author"})
public class Book {
  private String id;
  private String title;
  private String author;
  // getters and setters omitted for brevity
}
```

リスト 9 から、このカスタム順序ロジックが、リスト 3 に示されている名前付きプロパティーに適用されていることがわかります。`@JsonbPropertyOrder` によって指定された順序はデフォルトのプロパティーの順序ストラテジーをオーバーライドするだけでなく、あらゆるランタイム構成もオーバーライドします。新しい順序ストラテジーは、名前が変更された後のプロパティー名に適用されます。

フィールドの可視性をカスタマイズする

JSON Binding では、シリアライズとデシリアライズのプロセスにおいて、開発者がフィールドの可視性と包含/除外をきめ細かく制御できるようになっています。

一例として、フィールドに `@JsonbTransient` アノテーションを設定すると、そのフィールドはシリアライズされた JSON ドキュメントからは除外され、デシリアライズでは無視されます。このようにアノテーションを設定できるのは、フィールドとゲッター/セッター・メソッドだけです。クラス・レベルでのアノテーションの設定は有効ではなく、理にかなってもいません。

別の手段として、クラスに含まれるフィールドの可視性という特性を構成することもできます。デフォルトでシリアライズとデシリアライズに使用されるのは public フィールドとゲッター/セッター・メソッドだけですが、このようなアクセス制限を軽減または強化するようにカスタマイズすることもできます。

フィールドの可視性ストラテジーは、`PropertyVisibilityStrategy` インターフェースのカスタム実装内で定義します。このインターフェースでは、`isVisible(Field)` と `isVisibile(Method)` と

いう 2 つの多重定義されたメソッドが、それぞれフィールドの値、メソッドの値へのアクセスを制御します。リスト 10 に実装例を示します。

リスト 10. PropertyVisibilityStrategy の実装例

```
PropertyVisibilityStrategy vis = new PropertyVisibilityStrategy() {
  @Override
  public boolean isVisible(Field field) {
      return Modifier.isProtected(field.getModifiers());
  }

  @Override
  public boolean isVisible(Method method) {
      return !method.getName().contains("AlternativeTitle");
  }
};
```

`PropertyVisibilityStrategy` 型のインスタンスは、`JsonbConfig` インスタンス上に設定されます (リスト 11 を参照)。

リスト 11. PropertyVisibilityStrategy を使用するようにシリアライズを構成する

```
Magazine magazine =
    new Magazine("Fun with JSON binding",
    new Author("Alex", "Theedom"),
    "Fun with JSON-B");

JsonbConfig jsonbConfig = new JsonbConfig()
      .withPropertyVisibilityStrategy(vis);

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

isVisible() メソッド

シリアライズのプロセスでは、リフレクションを使用してオブジェクトがトラバースされ、各ゲッター・メソッド・オブジェクトが `isVisible()` メソッドに渡されます。このメソッド内に、所定のメソッドを呼び出すかどうかを決定するロジックを作成することができます。ロジックによってメソッドの呼び出しが決定された場合は、true を返すことで、呼び出されたメソッドからの戻り値を JSON ドキュメントの出力に含めるよう指示します。リスト 10 の `isVisible()` メソッドは、`getAlternativeTitle()` メソッドが渡されると false を返します。したがって、このメソッドに関連するフィールドは JSON 出力から除外されます。その他すべてのメソッドに対しては true を返すため、それらのプロパティー値が出力に含められることになります。

上記の例では、フィールドを受け入れる `isVisible()` が呼び出されることはありません。フィールドが渡された場合は、JavaBean メソッドを呼び出すようにシリアライズ処理が構成されているためです。Bean メソッドが存在していなければ、シリアライザーとデシリアライザーはインスタンスの public フィールドにアクセスしようと試みます。

その場合、`protected` 修飾子が設定されたフィールドだけを有効にするコードによって、`isVisible()` メソッドがオーバーライドされるようにしています。Java インスタンスに Bean メソッドが 1 つも含まれないというシナリオでは、`protected` 修飾子が設定されたフィールドだけにアクセスして、それらのフィールドが出力される JSON ドキュメントに含められることになります。

null の処理をカスタマイズする

デフォルトの動作では、シリアライズの際は null 値を格納するオブジェクト・フィールドが無視され、コレクションに含まれる null 値は維持されます。デシリアライズの際は、JSON プロパティーに null 値が格納されている場合、そのプロパティーに対応するオブジェクト・フィールドが null に設定されます。ターゲット・フィールドが `Optional` だとすると、その値は `Optional.empty()` に設定されます。このデフォルトの動作をカスタマイズするには 3 つの方法があります。

@JsonbNillable を使用する

最初の方法は、パッケージまたはクラスを `@JsonbNillable` アノテーションでマークすることです。このカスタマイズはパッケージまたはクラスに含まれるすべてのフィールドに適用されるため、シリアライズとデシリアライズの際にすべての null が維持されることになります。リスト 12 では、`title` フィールドに `@JsonbNillable` アノテーションを設定しています。

リスト 12. title フィールドに @JsonbNillable アノテーションが設定された Book クラス

```
public class Book {
  private String id;
  @JsonbNillable
  private String title;
  private String author;
  // getters and setters omitted for brevity
}
```

nillable フラグを使用する

上記と同じソリューションを実現するには、関連するフィールドまたは JavaBean プロパティーを `@JsonbProperty` アノテーションでマークして、このアノテーションの `nillable` フラグを `true` に設定するという方法もあります (リスト 13 を参照)。

リスト 13. authorName フィールドに @JsonbProperty アノテーションが設定された Magazine クラス

```
public class Magazine {
  private String title;
  @JsonbProperty(nillable = true)
  private Author authorName;
  private String alternativeTitle;
  // getters and setters omitted for brevity
}
```

構成の競合が発生した場合は、スコープが最も小さいアノテーションが優先されます。したがって、クラスに適用された `@JsonbNillable` アノテーションは、`nillable = false` が設定された `@JsonbProperty` アノテーションによってオーバーライドされます。

withNullValues() を使用する

null の処理をカスタマイズする 3 つ目の方法は、`JsonbConfig` インスタンス上の `withNullValues()` メソッドに `true` または `false` を渡すことです。リスト 14 に、null 値を `true` に設定した `JsonbConfig` インスタンスの構成を示します。

リスト 14. null 値を true に設定して JsonbConfig インスタンスを構成する

```
JsonbConfig jsonbConfig = new JsonbConfig().withNullValues(true);
Booklet booklet = new Booklet("Fun with Java EE", null, null);
String json = JsonbBuilder.create(jsonbConfig).toJson(booklet);
```

リスト 15 に、上記の `Booklet` インスタンスから出力された JSON ドキュメントを記載します。

リスト 15. null 値が維持されている JSON ドキュメント

```
{
 "author": {
   "firstName": null,
   "lastName": null
 },
 "title": "Fun with Java EE"
}
```

オブジェクトの作成をカスタマイズする

JSON-B では、すべてのクラスに引数なしの public コンストラクターが定義されることになっています。デシリアライズの際は、このコンストラクターを使用してターゲット・クラスがインスタンス化されます。インスタンスが作成されると、そのインスタンスに JSON ドキュメントからデータを取り込むために、該当するセッター・メソッドが呼び出されるか、public フィールドが直接設定されます。

この手法はほとんどの単純なシナリオに有効ですが、複雑なインスタンスを作成する場合には必要を満たさないこともあります。そのような場合は、カスタム・コンストラクターまたは static ファクトリー・メソッドを実装する必要があります。リスト 16 に、カスタム・コンストラクターの例を記載します。

リスト 16. カスタム・コンストラクター

```
public class Magazine {
  private String title;
  private Author authorName;

  @JsonbCreator
  public Magazine(@JsonbProperty("bookTitle") String title,
                  @JsonbProperty("firstName") String firstName,
                  @JsonbProperty("surname") String lastName) {
      this.title = title;
      this.authorName = new Author(firstName, lastName);
  }
  // getters and setters omitted for brevity
}
```

JSON プロパティー名をコンストラクターのパラメーター・リストに含まれるパラメーターにマッピングするために、`@JsonbProperty` アノテーションを使用していることに注目してください。リスト 17 に記載する JSON ドキュメントは、リスト 16 の `Magazine` クラスに正常にデシリアライズされます。

リスト 17. Magazine JSON ドキュメント

```
{
 "firstName": "Alex",
 "surname": "Theedom",
 "bookTitle": "Fun with JSON-B"
}
```

JSON-B アダプターについて説明するセクションで、さらに高度なシリアライズおよびデシリアライズ・プロセスのカスタマイズ方法を紹介します。

日付形式と時刻形式をカスタマイズする

日付と時刻の処理は、どのような処理であっても面倒な問題になることがあります。ありがたいことに、新しい JSON Binding API には日付形式と時刻形式を簡単にカスタマイズするための機能が揃っています。

`@JsonbDateFormat` アノテーションを使用すると、パッケージからフィールドに至るまでのあらゆるレベルで日付形式と時刻形式を構成することができます。リスト 18 では、セッター・メソッドだけをアノテーションでマークして、デシリアライズの際にだけ、指定の日付形式が適用されるようにしています。ゲッター・メソッドだけをアノテーションでマークするとしたら、指定の日付形式はシリアライズの際にだけ適用されることになります。

リスト 18. デシリアライズの場合にだけカスタマイズされる日付形式

```
public class Magazine {
  private String title;
  private Author authorName;
  private LocalDate localDate;

  @JsonbDateFormat(value = "MM/dd/yyyy", locale = "Locale.ENGLISH")
  public void setLocalDate(LocalDate localDate) {
      this.localDate = localDate;
  }
  // other getter and setters omitted for brevity
}
```

ランタイム・モデルを使用して日付形式をカスタマイズすることもできます (リスト 19 を参照)。この場合、`withDateFormat()` メソッドを使用して、`JsonbConfig` インスタンス上に日付形式とロケールを設定します。

リスト 19. JsonbConfig インスタンス上に日付形式を設定する

```
String json = "{\"published\":\"10/10/2018\",
               \"author\":\"Alex Theedom\",
               \"title\":\"Fun with JSON-B\"}";

JsonbConfig jsonbConfig = new JsonbConfig()
    .withDateFormat("MM/dd/yyyy", Locale.ENGLISH);

JsonbBuilder.create(jsonbConfig).fromJson(json, Book.class);
```

数値のフォーマットをカスタマイズする

数値プロパティーのフォーマットは、明示的に設定することができます。それには、特殊な数値パターン文字からなるストリングを `@JsonbNumberFormat` アノテーションに渡します。リスト 20 に、「`#.00`」というパターンを使用して数値をフォーマット設定する例を示します。

リスト 20. 数値フィールドの数値フォーマットを定義する

```
public class Booklet {
  private String title;
  private Author author;

  @JsonbNumberFormat("#.00")
  private double price;
}
```

`@JsonbNumberFormat` アノテーションでは、パッケージ、クラス、ゲッター/セッター、パラメーター、フィールドをマークすることができます。

数値パターン・フォーマットの詳細については、java.text.DecimalFormat クラスに関する Javadoc を参照してください。

ランタイムのグローバル・カスタマイズ

これまでに説明したカスタマイズの多くは、フィールドやメソッドに対してローカルに適用することも、実行するすべての処理に対してグローバルに適用することもできます。以降のセクションでは、グローバルな要素だけからなる残りのランタイム構成について見ていきます。

バイナリー・データの処理をカスタマイズする

バイナリー・データの処理ストラテジーには、`BYTE`、`BYTE_64`、`BYTE_64_URL` の 3 つがあります。バイナリー・データの処理ストラテジーをカスタマイズするには、`withBinaryDataStrategy()` メソッドに 3 つの静的定数のうちのいずれかを渡せばよいだけです (リスト 21 を参照)。

リスト 21. バイナリー・トラテジー

```
new JsonbConfig()
  .withBinaryDataStrategy(BinaryDataStrategy.BASE_64_URL);
```

I-JSON を構成する

I-JSON は、相互運用性を最大限にするよう意図された、制約のある JSON プロファイルです。デフォルトでは、JSON-B はこのプロファイルに緩やかに従います。このプロファイルに厳格に準拠させる必要がある場合は、`JsonbConfig` インスタンス上で `StrictIJSON` フラグを true に設定することで、厳格な準拠を有効にすることができます。リスト 22 に、このように構成する方法を示します。

リスト 22. 厳格な I-JSON を設定する

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

文字符号化をカスタマイズする

バインディング処理では、以下のプロパティーによって指定された符号化タイプを使用して、JSON データを符号化します。リスト 23 に一例を記載します。

リスト 23. デフォルトの符号化を設定する

```
JsonbConfig jsonbConfig = new JsonbConfig().withEncoding("UTF-32");
```

有効な文字符号化は、RFC 7159 のセクション 8 で定義されています。

デフォルトのロケールをカスタマイズする

ロケールは、リスト 24 に示すように設定できます。

リスト 24. デフォルトの `Locale` を設定する

```
JsonbConfig jsonbConfig = new JsonbConfig()
  .withLocale(Locale.CANADA);
```

整形された JSON 出力をカスタマイズする

シリアライズ・プロセスによる出力に、適切な改行とインデントを使用して整形された JSON が使用されるよう設定することができます。リスト 25 に、そのための構成を記載します。

リスト 25. 整形式 JSON を出力する

```
JsonbConfig jsonbConfig = new JsonbConfig().withFormatting(true);
```

シリアライズとデシリアライズをカスタマイズする

これまで紹介した単純なアノテーションと構成フラグを使用してできることには限りがあります。いずれは、より堅牢で適応性に優れたソリューションが必要になるでしょう。まさにそのようなシナリオのために導入されているのが、JSON-B のアダプターと下位レベルのシリアライズ/デシリアライズ機能です。

JSON B アダプター

アダプターは、`JsonBAdapter` インターフェースのメソッドを実装してカスタマイズしたオブジェクトの作成とシリアライズを構成します。JSON-B アダプターを使用すれば、`adaptToJson()` および `adaptFromJson()` メソッドを、シリアライズおよびデシリアライズに対する新しいロジックで上書きすることができます。

リスト 26 は、`Booklet` オブジェクトを `JsonObject` に変換するコードを使用した `adaptToJson()` メソッドの実装です。`adaptFromJson()` メソッドは、JSON Processing API に用意されている JSON Object Builder を使用して、`JsonObject` インスタンスから `Booklet` オブジェクトを作成します。

リスト 26. JsonbAdapter インターフェースの実装

```
public class BookletAdapter implements JsonbAdapter<Booklet, JsonObject> {

  @Override
  public JsonObject adaptToJson(Booklet booklet) {
      return Json.createObjectBuilder()
              .add("title", booklet.getTitle())
              .add("firstName", booklet.getAuthor().getFirstName())
              .add("lastName", booklet.getAuthor().getLastName())
              .build();
  }

  @Override
  public Booklet adaptFromJson(JsonObject json) {
```

```
      Booklet booklet = new Booklet(
              json.getString("title"),
              new Author(json.getString("firstName"),
   json.getString("lastName")));
      return booklet;
  }
}
```

リスト 28 を見るとわかるように、`adaptToJson()` メソッドによって `Author` オブジェクトは 2 つのプロパティー (`firstName` と `lastName`) に「フラット化」されます。リスト 27 では、`Booklet` クラスのインスタンスを基に、`adaptFromJson()` が `Author` オブジェクトを再作成して `Booklet` インスタンスを出力します。

リスト 27. Booklet と Author クラス

```
public class Booklet {
   private String title;
   private Author author;
    // getters and setters omitted for brevity
}

public class Author {
  private String firstName;
  private String lastName;
  // getters and setters omitted for brevity
}
```

リスト 28 に、出力された `Booklet` インスタンスを記載します。

リスト 28. シリアライズされた Booklet オブジェクト

```
{
 "title": "Fun with JSON-B",
 "firstName": "Alex",
 "lastName": "Theedom"
}
```

フィールド・レベルの JsonbAdapter を使用する

`JsonbAdapter` は極めて柔軟性に優れていて、オブジェクト全体に対してだけでなく、個々のオブジェクトに対してもシリアライズとデシリアライズをカスタマイズするために使用できます。それには、カスタマイズした動作を適用する対象のフィールド、メソッド、またはクラスを `@JsonbTypeAdapter` アノテーションでマークし、使用する `JsonbAdapter` のクラス名をこのアノテーションに渡します。

リスト 29 では、`firstName` フィールドを `@JsonbTypeAdapter` アノテーションでマークし、`FirstNameAdapter` クラスをアダプターとして指定しています。

リスト 29. firstName フィールドに対して FirstNameAdapter を指定する

```
public class Author {
  private String lastName;
  @JsonbTypeAdapter(FirstNameAdapter.class)
  private String firstName;
  // getters and setters omitted for brevity
}
```

リスト 30 に、`FirstNameAdapter` クラスの実装を記載します。

リスト 30. FirstNameAdapter アダプター

```
public class FirstNameAdapter
implements JsonbAdapter<String, JsonValue> {

  @Override
  public JsonValue adaptToJson(String firstName) {
      return Json.createValue(firstName.subSequence(0,1).toString());
  }

  @Override
  public String adaptFromJson(JsonValue json) {
      return json.toString();
  }
}
```

アダプターの型は、そのアダプターがオブジェクトまたはフィールドのどちらを適応させるかによって決まります。オブジェクトを適応させる場合は `JsonObject` 型が使用され、フィールドを適応させる場合は `JsonValue` 型が使用されます。フィールドの型が配列であれば、`JsonArray` 型が使用されます。

下位レベルのシリアライザー/デシリアライザーによるカスタマイズ

JSON-B のシリアライザーとデシリアライザーは、最下位レベルで利用できるカスタマイズ手段です。この 2 つを使用することで、JSON Processing API 内にあるパーサーやジェネレーターにアクセスできるようになります。

カスタム・シリアライザーでは `JsonbSerializer` インターフェースを実装して、`serialize()` メソッドのロジックを定義する必要があります。リスト 31 に、`book` オブジェクトのシリアライズをカスタマイズする例を記載します。

リスト 31. カスタマイズされた Book クラスのシリアライズ

```
public class BookSerializer implements JsonbSerializer<Book> {
  @Override
  public void serialize(Book book, JsonGenerator generator, SerializationContext ctx) {
      generator.writeStartObject();
      generator.write("id", "QWE-123-RTS");
      generator.write("title", book.getTitle());
      generator.write("firstName", book.getAuthor().split(" ")[0]);
      generator.write("lastName", book.getAuthor().split(" ")[1]);
      generator.writeEnd();
  }
}
```

この例では、`JsonGenerator` を使用して、プロパティーごとに JSON ドキュメントを作成していきます。`id` プロパティーの値は固定され、作成者の名前が `firstName` と `lastName` の 2 つのプロパティーにフラット化されます。

デシリアライズ処理は、`JsonbDeserializer` インターフェースを実装して `deserialize()` メソッドのロジックを定義することによってカスタマイズします。リスト 32 の例では、JSON ドキュメントから `Book` の `id` だけが抽出されます。

リスト 32. カスタマイズされた Book クラスのデシリアライズ

```
public class BookDeserializer implements JsonbDeserializer<String> {
  @Override
  public String deserialize(JsonParser parser, DeserializationContext ctx, Type rtType) {
      while (parser.hasNext()) {
          JsonParser.Event event = parser.next();
          if (event == JsonParser.Event.KEY_NAME) {
              String keyName = parser.getString();
              if (keyName.equals("id")) {
                  return ctx.deserialize(String.class, parser);
              }
          }
          parser.next();
      }
      return "";
  }
}
```

シリアライザーとデシリアライザーのインスタンスを `JsonbConfig` に登録するには、`withDeserializers()` または `withSerializers()` のいずれか該当するメソッドを使用します (リスト 33 を参照)。

リスト 33. シリアライザーとデシリアライザーの両方を登録する

```
JsonbConfig config = new JsonbConfig()
      .withDeserializers(new BookDeserializer())
      .withSerializers(new BookSerializer());
```

あるいは、型に `@JsonbTypeSerializer` または `@JsonbTypeDeserializer` アノテーションを設定し、該当するカスタマイズ・クラスのクラス名を渡すという方法もあります。

まとめ

JSON-B の設計者たちは、開発者が Java オブジェクトを JSON ドキュメントに、または JSON ドキュメントを Java オブジェクトに変換する方法を標準化するための作業に着手しました。この連載でこれまでに紹介した JSON-B の API と機能が馴染み深いものに感じられたとしたら、それは偶然ではありません。つまり、この API には、ほとんどの Java 開発者が使い慣れているはずのシリアライズ手法とデシリアライズ手法を標準化することが意図されています。

ランタイム構成またはコンパイル時のアノテーションを使用するという 2 つのモデルでは、極めて柔軟かつ直感的に、この API を微調整できるようになっています。さらに、自己記述的なアノテーションと構成メソッドは、開発サイクルにおける生産性と作業の軽減に貢献します。これらのモデルを使用すれば、下位レベルのシリアライズとデシリアライズを簡単に操作できるだけでなく、高度なカスタマイズを行うこともできます。

JSON-B では `Optional` などの Java SE 8 の機能と新しい Date/Time API を巧みに扱えることから、これらの言語構造の処理に四苦八苦する他の主要な JSON パーサーよりも JSON-B は先んじています。この点について納得できるよう、この連載の最終回となる次回の記事では、JSON-B を他のよく使われている Jackson と GSON という 2 つの JSON パーサーと比較します。

学んだ知識をテストしてください

1. 次のメカニズムのうち、デフォルトの動作をカスタマイズするために使用できるのはどれですか？
   a. パッケージ、クラス、フィールド、ゲッター/セッター・メソッドをアノテーションでマークする
   b. XML 構成ファイル
   c. `JsonbConfig` インスタンスを構成して `Jsonb` インスタンス上に設定する
   d. アプリケーションの起動時にブートストラップするプロパティー・ファイル
   e. 構成シングルトン
2. 次のアノテーションのうち、JSON Binding のアノテーションはどれですか？
   a. `@JsonbProperty`
   b. `@JsonbIgnore`
   c. `@JsonbNotNullable`
   d. `@JsonbPropertyOrder`
   e. `@JsonbNumberFormat`
3. 次のうち、JSON Binding のストラテジーはどれですか？
   a. `PropertyVisibilityStrategy`
   b. `PropertyOrderStrategy`
   c. `PropertyNamingStrategy`
   d. `DateAndTimeStrategy`
   e. `NullPreservationStrategy`
4. カスタム・アダプターを作成するには、どのクラスを実装する必要がありますか？
   a. `JsonbCustomAdapter`
   b. `JsonbSerializationAdapter`
   c. `JsonbDeserializationAdapter`
   d. `JsonAdapter`
   e. どれも当てはまらない
5. JSON シリアライザーを実装して使用するには、どうすればよいですか？
   a. `JsonbSerializer #####`
   b. シリアライズするオブジェクトを、`JsonbSerializer` インスタンスの `serialize()` メソッドに渡す
   c. `JsonbSerializer` のインスタンスを使用して `JsonbConfig` インスタンスを構成する
   d. `JsonbSerializer` のインスタンスを `@JsonbTypeSerializer` に渡す
   e. どれも当てはまらない

このリンク先のページで正解を確認してください。

関連トピック

- Java EE 8 の新機能
- この記事のコードを GitHub で入手してください。
- JSON Binding のホームページを確認してください。
- JSR 367: Java API for JSON Binding (JSON-B)
- JSON-B の公式ユーザー・ガイド
- Yasson: JSON-B のリファレンス実装