記事
Toshihiko Minamoto · 2020年10月22日 7m read

JSONの機能強化

InterSystems IRIS 2019.1は公開されてからしばらく経ちますが、気づかれていない可能性のある、JSONの処理の強化機能について説明したいと思います。 最新のアプリケーションを構築する際、特にRESTエンドポイントを操作する際は、JSONをシリアル化形式として扱うことが重要です。

JSONの書式

まず、JSONに書式設定を適用すると、人の目で読みやすくなります。
コードをデバックする際に、特定のサイズでJSONコンテンツを確認する場合に非常に役立ちます。 構造が単純化されていれば、ざっと目を通すことが容易にはなりますが、ネストされている複数の要素に遭遇すると、あっという間に読みづらくなります。 以下に簡単な例を示します。

```
{"name":"Gobi","type":"desert","location":{"continent":"Asia","countries":["China","Mongolia"]},"dimensions":{"length":1500,"length_unit":"km","width":800,"width_unit":"km"}}
```

人の目でより読みやすい形式を適用すると、コンテンツの構造を調べやすくなります。
適切な改行やインデントを使用した同一のJSON構造を見てみましょう。

```
{
  "name":"Gobi",
  "type":"desert",
  "location":{
    "continent":"Asia",
    "countries":[
      "China",
      "Mongolia"
    ]
  },
  "dimensions":{
    "length":1500,
    "length_unit":"km",
    "width":800,
    "width_unit":"km"
  }
}
```

この単純な例だけでも、出力がかなり大きくなるため、多くのシステムでこの書式がデフォルト設定となっていない理由は明確でしょう。 ただし、この詳細な書式により、基礎構造を簡単に読み取れるようになり、何かが間違っているかどうかを見つけやすくなります。

InterSystems IRIS 2019.1では、%JSONという名前のパッケージが導入されました。 パッケージには、上記で示したとおり、動的なオブジェクトと配列、そしてJSON文字列をより読みやすくできる整形ツールなど、いくつかの便利なユーティリティがあります。 %JSON.Formatterは非常に単純なインターフェースを持つクラスです。
すべてのメソッドはインスタンスメソッドであるため、必ずインスタンスを取得するところから始めます。

```
USER>set formatter = ##class(%JSON.Formatter).%New()
```

この選択の背景には、インデント（空白またはタブなど）特定の文字や行末記号の特定の文字を使えるように整形ツールを1回構成し、それ以降、必要な個所に使用できるようになるという理由があります。

Format()メソッドは、動的なオブジェクト化配列またはJSON文字列を取ります。
では、動的なオブジェクトを使用した簡単な例を見てみましょう。

```
USER>do formatter.Format({"type":"string"})
{
  "type":"string"
}
```

そして、以下は、同じJSONコンテンツでJSON文字列を使った例です。

```
USER>do formatter.Format("{""type"":""string""}")
{
  "type":"string"
}
```

Format()
メソッドは、整形さ
れた文字列を現在のデバイスに出力しま
すが、直接変数に出力する場合のFormatToString()とFormatToStream() も表示されます。

変速を変える

上記は素晴らしい機能ですが、それだけでは記事にする価値はないかもしれません。 InterSystems IRIS 2019.1では、永続オブジェクトと一時オブジェクトをJSONとの間でシリアル化する便利な方法も導入されています。
ここで調べるクラスは %JSON.Adaptorです。
概念が%XML.Adaptorに非常に似ているため、その名前が付けられています。
JSONとの間でシリアル化するクラスは、%JSON.Adaptorをサブクラス化する必要があります。
クラスは、いくつかの便利なメソッドを継承します
が、中でも%JSONImport()と%JSONExport()の継承は非常に役立ちます。
これについては例で示すのが一番良いでしょう。 次のクラスがあったとします。

```
Class Model.Event Extends (%Persistent, %JSON.Adaptor)
{
 Property Name As %String;
 Property Location As Model.Location;
}
```

および

```
Class Model.Location Extends (%Persistent, %JSON.Adaptor)
{
 Property City As %String;
 Property Country As %String;
}
```

ご覧の通り、永続的なイベントクラスがあり、ロケーションにリンクしています。
両方のクラスは%JSON.Adaptorから継承されています。
このため、オブジェクトグラフを作成し、それをJSON文字列として直接エクスポートすることができます。

```
USER>set event = ##class(Model.Event).%New()

USER>set event.Name = "Global Summit"

USER>set location = ##class(Model.Location).%New()

USER>set location.City = "Boston"

USER>set location.Country = "United States of America"

USER>set event.Location = location

USER>do event.%JSONExport()
{"Name":"Global Summit","Location":{"City":"Boston","Country":"United States of Ameri
ca"}}
```

もちろん、%JSONImport()を使用して、逆の方向にエクスポートすることもできます。

```
USER>set jsonEvent = {"Name":"Global Summit","Location":{"City":"Boston","Country":"U
nited States of America"}}

USER>set event = ##class(Model.Event).%New()

USER>do event.%JSONImport(jsonEvent)

USER>write event.Name
Global Summit
USER>write event.Location.City
Boston
```

インポートメソッドとエクスポートメソッドは、任意のネストされた構造で機能します。
%XML.Adaptor
と同様に、対応するパラメーターを設定して、個々のプロパティのマッピングロジックを指定できます。
Model.Eventクラスを次の定義に変更してみましょう。

```
Class Model.Event Extends (%Persistent, %JSON.Adaptor)
{
 Property Name As %String(%JSONFIELDNAME = "eventName");
 Property Location As Model.Location(%JSONINCLUDE = "INPUTONLY");
}
```

上記の例とオブジェクト構造が
変数eventに割り当てられていると仮定した場合、%JSONExport()を呼び出すと、次の結果が返されます。

```
USER>do event.%JSONExport()
{"eventName":"Global Summit"}
```

Nameプロパティは、eventNameフィールド名にマッピングされ、Locationプロパティは
%JSONExport()呼び出しから除外されますが、存在する場合は%JSONImport()
呼び出し中にJSONコンテンツに入力されます。
マッピングを調整するには、次のようなパラメーターを使用できます。

- %JSONFIELDNAME: JSONコンテンツのフィールド名に対応します。
- %JSONIGNORENULL:
  開発者が文字列プロパティの空の文字列のデフォルト処理をオーバーライドできるようにします。

- %JSONINCLUDE : このプロパティがJSON出力/入力に含まれるかどうかを制御します。
- %JSONNULL: これがtrue（=1）である場合、未指定のプロパティはnull値としてエクスポートされます。
  そうでない場合は、プロパティに対応するフィールドはエクスポート中に省略されます。
- %JSONREFERENCE: オブジェクト参照の処理方法を指定します。
  デフォルトは「OBJECT
  」で、参照先クラスのプロパティが参照先オブジェクトを表すために使用されることを示します。
  その他のオプションは「ID」、「OID」、および「GUID」です。

これにより高度な制御が可能になり、非常に便利です。
オブジェクトを手動でJSONにマッピングする時代は終わりました。

あともう一つ

マッピングパラメーターをプロパティレベルで設定する代わりに、XDataブロックにJSONマッピングを定義することも可能です。
次に示すOnlyLowercaseTopLevel
という名前のXDataブロックには、上記のeventクラスと同じ設定が行われています。

```
Class Model.Event Extends (%Persistent, %JSON.Adaptor)
{
 Property Name As %String;
 Property Location As Model.Location;
 XData OnlyLowercaseTopLevel
 {
 <Mapping xmlns="http://www.intersystems.com/jsonmapping">
     <Property Name="Name" FieldName="eventName"/>
     <Property Name="Location" Include="INPUTONLY"/>
 </Mapping>
 }
}
```

重要な違いが1つあります。それは、XDataブロックのJSONマッピングはデフォルトの動作を変更しないが、対応する%JSONImport()と%JSONExport()
の呼び出しで最後の引数として参照する必要があるということです。例を示します。

```
USER>do event.%JSONExport("OnlyLowercaseTopLevel")
{"eventName":"Global Summit"}
```

指定された名前のXDataブロックが存在しない場合、デフォルトのマッピングが使用されます。 このアプローチを使用すると、複数のマッピングを構成し、各呼び出しに必要なマッピングを個別に参照することができます。そのため、マッピングをより柔軟で再利用可能にしながら、より高い制御性を得ることができます。

これらの機能強化によって作業が楽になることを願っています。皆さんからのフィードバックを楽しみしています。 ぜひコメントを残してください。

#JSON #REST API #XML #オブジェクトデータモデル #InterSystems IRIS

ソースURL:
https://jp.community.intersystems.com/post/json%E3%81%AE%E6%A9%9F%E8%83%BD%E5%BC%B7%E5%8C%96