記事
Toshihiko Minamoto · 2021年9月30日 16m read

Caché 2016.2のドキュメントデータモデルの紹介

はじめに

Caché 2016.2のフィールドテストはかなり前から利用可能ですので、このバージョンで新しく追加されたドキュメントデータモデルという重要な機能に焦点を当てたいと思います。 このモデルは、オブジェクト、テーブル、および多次元配列など、データ処理をサポートするさまざまな方法として自然に追加されました。
プラットフォームがより柔軟になるため、さらに多くのユースケースに適したものになります。

いくつかのコンテキストから始めましょう。
NoSQLムーブメントの傘下にあるデータベースシステムを少なくとも1つは知っているかもしれません。
これにはかなりたくさんのデータベースがあり、いくつかのカテゴリにグループ化することができます。
Key/Valueは非常に単純なデータモデルです。
値をデータベースに格納し、それにキーを関連付けることができます。
値を取得する場合は、キーを介してそれにアクセスする必要があります。 適切なキーを選択によってソートが決まり、キーの一部であるものでグループ化する場合に単純な集計に使用できるようになるため、キーの選択が重要な鍵となります。 ただし、値は値にすぎません。
値内の特定のサブ要素にアクセスしたり、それらにインデックスを作成したりすることはできません。
値をさらに活用するには、アプリケーションロジックを書く必要があります。 Key/Valueは、大規模なデータセットと非常に単純な値を操作する必要がある場合に最適ですが、より複雑なレコードを扱う場合には価値が劣ります。

ドキュメントデータモデルはKey/Valueにとてもよく似ていますが、値はより複雑です。 値はキーに関連付けられたままになりますが、さらに、値のサブ要素にアクセスして特定の要素にインデックスを作成することができます。 つまり、いくつかのサブ要素が制限を満たす特定のドキュメントを検索することもできるということです。 明らかに、NoSQLの世界にはGraphのような他のモデルもさらに存在しますが、ここでは、ドキュメントに焦点を置くことにします。

そもそもドキュメントとは？

一部の読者を混乱させる傾向があるため、まず最初に、1つ明確にしておきましょう。この記事で「ドキュメント」と言った場合、PDFファイルやWordドキュメントといった物理的なドキュメントを指してはいません。
この文脈でのドキュメントとは、サブ値を特定のパスと関連付けることのできる構造を指しています。
ドキュメントを記述できるシリアル化形式には、JSONやXMLなどのよく知られたものが様々あります。
通常こういった形式には、共通して次のような構造とデータ型があります。

1. 順序付けされていないKey/Valueペアの構造
2. 順序付けされた値のリスト
3. スカラー値

1つ目は、XMLの属性要素とJSONのオブジェクトにマッピングされます。
2つ目の構造は、XMLのサブ要素とJSONの配列を使ったリストによって導入されています。
3つ目は、単に、文字列、数値、ブール値といったネイティブのデータ型を利用できるようしています。

JSONのようなシリアル化された形式でドキュメントを視覚化することは一般的ですが、これは、ドキュメントを表現できる一方法にすぎないことに注意してください。 この記事では、JSONを主なシリアル化形式として使用することにします。JSONサポートの改善機能をうまく利用できるでしょう。この改善についてまだ読んでいない方

はこちらをご覧ください。

ドキュメントはコレクションにグループ化されます。
セマンティックと潜在的に共通の構造を持つドキュメントは同じコレクションに保存する必要があります。
コレクションはその場で作成できるため、事前にスキーマ情報を用意しておく必要はありません。

コレクションにアクセスするには、データベースハンドルを最初に取得しておく必要があります。 データベースハンドルはサーバーへの接続として機能し、コレクションへの単純なアクセスを提供しますが、分散環境の場合にはさらに複雑なシナリオを処理することもできます。

基本

まず、Caché Object Scriptで単純なドキュメントを挿入する方法を見てみましょう。

```
USER>set db = ##class(%DataModel.Document.Database).$getDatabase()

USER>set superheroes = db.$getCollection("superheroes")

USER>set hero1 = {"name":"Superman","specialPower":"laser eyes"}

USER>set hero2 = {"name":"Hulk","specialPower":"super strong"}

USER>do superheroes.$insert(hero1)

USER>do superheroes.$insert(hero2)

USER>write superheroes.$size()
2
```

上記のコードサンプルでは、まずデータベースハンドルが取得されて、「superheroes」というコレクションが取得されます。 コレクションは明示的に作成されるため、事前に設定する必要はありません。 新しいコレクションにアクセスできるようになったら、ヒーローのSupermanとHulkを表す非常に単純なドキュメントを2つ作成します。
これらは$insert(<document>)
への呼び出しでコレクションに保存され、コレクションサイズの最終チェックで、2つのドキュメントを報告します。これは、以前にコレクションが存在していなかったためです。

$insert()呼び出しは、成功すると、挿入されたドキュメントを返します。
このため、ドキュメントの操作を続行する場合に、自動的に割り当てられたIDを取得することができます。
また、チェーンメソッドも可能になります。

```
USER>set hero3 = {"name":"AntMan","specialPower":"can shrink and become super strong"
}

USER>write superheroes.$insert(hero3).$getDocumentID()
3
```

このコードスニペットは、別のヒーローオブジェクトを作成し、superheroesコレクションに永続させます。
今回は、メソッド呼び出しの$getDocumentID()を$insert()
呼び出しに連鎖させ、システムがこのドキュメントに割り当てたIDを取得します。 $insert()
は必ず自動的にIDを割り当てます。
独自のIDを割り当てる必要がある場合は、$insertAt(<User-ID>,<document>)呼び出しを利用できます。

特定のIDでドキュメントを取得する場合は、コ

レクションに対して$get(<ID>)メソッドを呼び出すことができます。

```
USER>set antMan = superHeroes.$get(3)

USER>write antMan.$toJSON()
{"name":"AntMan","specialPower":"can shrink and become super strong"}
```

Supermanとそのhometownを表すドキュメントを更新するとしましょう。
この場合、$upsert(<ID>,<document>)
呼び出しを使用して、既存のドキュメントを簡単に更新することができます。

```
USER>set hero1.hometown = "Metropolis"

USER>do superheroes.$upsert(1,hero1)

USER>write superheroes.$get(1).$toJSON()
{"name":"Superman","specialPower":"laser eyes","hometown":"Metropolis"}
```

$upsert()
は、IDがまだほかに使用されていない場合にドキュメントを挿入するか、そうでない場合に既存のドキュメントを更新します。
もちろん、$toJSON()
を呼び出すだけで、コレクションの全コンテンツをJSONにシリアル化することも可能です。

```
USER>write superheroes.$toJSON()
[
{"documentID":1,"documentVersion":4,"content":{"name":"Superman","specialPower":"laser eyes","hometown":"Metropolis"}},
{"documentID":2,"documentVersion":2,"content":{"name":"Hulk","specialPower":"super strong"}},
{"documentID":3,"documentVersion":3,"content":{"name":"AntMan","specialPower":"can shrink and become super strong"}}
]
```

コレクションがドキュメントの配列として表されていることがわかります。 各ドキュメントは、ドキュメントIDとドキュメントバージョンでラップされており、同時実行を適切に処理するために使用されます。
実際のドキュメントコンテンツは、ラッパーのプロパティコンテンツに格納されます。
これは、コレクションの完全なスナップショットを取得して移動できるようにするために必要な表現です。
また、これはモデルの非常に重要な側面をカバーしており、特殊プロパティを予約してドキュメントデータに挿入することはありません。
ドキュメントを適切に処理するためのエンジンが必要とする情報は、ドキュメントの外部に保存されます。
さらに、ドキュメントはオブジェクトまたは配列のいずれかです。
その他の多くのドキュメントストアは、オブジェクトを最上位の要素としてのみ許可しています。

コレクションでドキュメントを変更するためのAPI呼び出しには他にもたくさんあり、基本的な演算をいくつか見てきました。 ドキュメントの挿入と変更は楽しい作業ですが、実際にデータセットを分析したり、特定の制限を満たすドキュメントを取得したりすると、さらに興味深くなります。

クエリ

すべてのデータモデルが有用とみなされるには、何らかのクエリ機能が必要です。
動的スキーマを使用してドキュメントをクエリできるようにするには、2つの潜在的な方法があります。

1. 動的なドキュメントの性質に対処できる独自のクエリ言語を設計して実装する
2. 定着している構造化されたクエリ言語にクエリを統合する

この記事の後の方で説明するいくつかの理由により、コレクションをSQLエンジンに公開することにしました。 SQLの知識を引き続き活用できるというメリットがあります。また、クエリ方言の別のフレーバーを作成しているところでもあります。 実際、SQL
ANSI委員会は、JSONの標準拡張機能を提案しており、それに準拠しています。
まとめれば、これらの拡張機能には、JSON関数の2つのカテゴリが含まれています。

1. リレーションコンテンツからJSONコンテンツに公開するための関数セット
2. 動的JSONコンテンツをクエリするための関数セット

この記事の範囲では、2つ目のカテゴリである動的JSONコンテンツのクエリのみを取り上げ、結果をSQLで処理できるようにテーブルとして利用できるようにします。

動的コンテンツ（関連するスキーマのないコンテンツ）を公開し、事前に定義されたスキーマを使用してデータを操作するSQLで利用できるようにする魔法の関数は、JSON_TABLE です。
一般に、この関数は2つの引数を取ります。

1. JSONデータソース
2. 名前と型で列へのJSONパスのマッピングを指定する定義

骨に肉付けした例を見てみましょう。

```
SELECT name, power FROM JSON_TABLE(
  'superheroes',
  '$' COLUMNS(
    name VARCHAR(100) PATH '$.name',
    power VARCHAR(300) PATH '$.specialPower'
  )
)

name        power
---------   ------------------------------
Superman    laser eyes
Hulk        super strong
AntMan      can shrink and become super strong
```

JSON_TABLE 関数の最初の引数は、それが作成する仮想テーブルのソースを定義します。
この場合、コレクション「superheroes」をクエリします。
このコレクションのドキュメントごとに行が作られます。
2つ目の引数は、ドキュメントの特定の値をテーブルの列として公開することを忘れないでください。
この引数は2つ部分で構成されています。最初のステップとして、次の式のコンテキストを設定します。
ドル記号'$'には特別な意味があり、ドキュメントのルートを指しています。
それ以降のすべての式はこのコンテキストを基準としています。
後に続くのは、COLUMNS句で、カンマ区切りのCOLUMN式のリストです。
COLUMN式ごとに、仮想テーブルの列が作成されます。
「name」と「power」という2つの列をクエリで公開しています。
列「name」はVARCHAR(100)型で定義されていますが、列「power」は300文字に制限されています。
PATH式は、JPL（JSONパス言語）式を使用してドキュメントの特定の値を列に関連付けています。 キー「name」の値は列「name」に公開されますが、キー「specialPower」の値は列「power」にマッピングされます。
JPL式は非常に表現力が高く強力ですが、これについては別のトピックで説明することにします。
このサンプルで使用した式は非常に基本的な式です。

この構文が初めてであれば、理解するのに少し時間が掛かるかもしれませんが、 JSON_TABLE
関数を自然に読み取ると理解しやすいでしょう。 例として、上記のクエリを使います。

ここで表現しているのは、基本的に次のことです。

コレクション「superheroes」をクエリし、各ドキュメントのルートに式のコンテキストを設定します。
次の2つの列を公開します。

1. 列「name」を型VARCHAR(100)で公開し、キー「name」の値を挿入します。
2. 列「power」を型VARCHAR(300)で公開し、キー「specialPower」の値を挿入します。

前に述べた通り、JPL式は複雑になりがちであるか、たくさんの列を公開したいだけの場合もあります。
そのため、型定
義を参照できる標準への拡張を組み込みました。これは基本的に、事前定義済みのCOLUMNS句です。
このようにして、上記のCOLUMNS句を登録することができます。

```
do db.$createType("heropower",{"columns":[{"column":"name","type":"VARCHAR(100)","pat
h":"$.name"},{"column":"power","type":"VARCHAR(300)","path":"$.specialPower"}]})
```

型情報を登録したら、%TYPE式を使って、JSONTABLE 関数でそれを参照することができます。

```
SELECT name, power FROM JSON_TABLE(
  'superheroes',
  '$' %TYPE 'heropower'
)
```

これは明らかに、SQLクエリにドキュメントの一貫したビューを提供し、クエリそのものを大幅に簡略化する上で役立ちます。

高度な内容

ここまで説明したことのほぼすべてについて補足することはたくさんありますが、ここでは最も重要なことに焦点を当てたいと思います。
最後のセクションを読みながら、JSONTABLE
関数を非常に強力なポイントとしている手掛かりに気づいたかもしれません。

1. 仮想テーブルを作成する
2. JSONのようなデータをソースデータとして消費できる

最初の項目はそれだけで重
要なポイントです。コレクションを簡単にクエリして
、別のJSONTABLE 呼び出しまたは正にテーブルと結合することができるからです。 コレクションをテーブルと結合できることは大きなメリットです。要件に応じて、データに完璧なデータモデルを選択できるのです。
型安全、整合性チェックが必要であるのに、モデルがあまり進化していませんか？
リレーショナルを使いましょう。 他のソースのデータを処理してそのデータを消費する必要があり、モデルが必ず急速に変化するか、アプリケーションユーザーの影響を受ける可能性のあるモデルを保存することを検討していますか？ ドキュメントデータモデルを選択しましょう。
モデルはSQLで一緒にまとめることができるので安心です。

JSONTABLE 関数の2つ目のメリットは、実のところ基盤のデータモデルとは無関係です。
これまでに、JSONTABLE でコレクションのクエリを説明してきました。
最初の引数は任意の有効なJSON入力にすることもできます。 次の例を考察しましょう。

```
SELECT name, power FROM JSON_TABLE(
  '[
```

```
    {"name":"Thor","specialPower":"smashing hammer"},
    {"name":"Aquaman","specialPower":"can breathe underwater"}
  ]',
  '$' %TYPE 'heropowers'
)

name         power
---------    ------------------------------
Thor         smashing hammer
Aquaman      can breathe underwater
```

入力は通常のJSON文字列で、オブジェクトの配列を表します。 構造がsuperheroesコレクションと一致するため、保存された型識別子「heropowers」を再利用することができます。
これにより、強力なユースケースが可能になります。
実際にメモリ内のJSONデータをディスクに永続させずにクエリすることができます。
REST呼び出しでJSONデータをリクエストし、クエリを実行してコレクションまたはテーブルを結合できます。
この機能を使用すると、Twitterタイムライン、GitHubリポジトリの統計、株式情報、または単に天気予報のフィードをクエリできます。
この機能は非常に便利であるため、これについては、後日、専用の記事で取り上げたいと思います。

REST対応

ドキュメントデータモデルでは、初期状態でREST対応のインターフェースが備わっています。
すべてのCRUD（作成、読み取り、更新、削除）とクエリ機能はHTTP経由で利用できます。 完全なAPIについては説明しませんが、ネームスペース「USER」内の「superheroes」コレクションのすべてのドキュメントを取得するサンプルcURLを以下に示します。

```
curl -X GET -H "Accept: application/json" -H "Cache-Control: no-
cache" http://localhost:57774/api/document/v1/user/superheroes
```

私のユースケースで使用できますか？

ドキュメントデータモデルは、InterSystemsのプラットフォームへの重要な追加機能です。
これは、オブジェクトとテーブルに続く、まったく新しいモデルです。
SQLとうまく統合できるため、既存のアプリケーションで簡単に利用することができます。 Caché 2016.1で導入された新しいJSON機能によって、CachéでのJSONの処理が楽しく簡単になります。

そうは言っても、これは新しいモデルです。 いつ、そしてなぜそれを使用するのかを理解する必要があります。
常に言っていることですが、特定のタスクにはそれに適したツールを選択してください。

このデータモデルは、動的データを処理する必要がある場合に優れています。
以下に、主な技術的メリットをまとめます。

- 柔軟性と使いやすさ

スキーマを予め定義する必要がないため、データの作業環境を素早くセットアップし、データ構造の変更に簡単に適応させることができます。

- スパース性

テーブルに300列があっても、各行が入力できるのはその内の15列であることを覚えていますか？
これはスパースなデータセットであり、リレーショナルシステムではそれらを最適に処理にできません。
ドキュメントは設計上スパースであり、効率的に保存と処理を行えるようになっています。

- 階層

配列やオブジェクトなどの構造化型は、任意の深さでネストすることができます。 つまり、ドキュメント内で関連するデータを保存することができるため、そのレコードにアクセスする必要がある場合の読み取りのI/Oを潜在的に縮小することができます。
データは非正規化して保存できますが、リレーショナルモデルではデータは正規化して保存されます。

- 動的な型

特定のキーには、列のように固定されたデータ型がありません。 名前は、あるドキュメントでは文字列であっても、別のドキュメントでは複雑なオブジェクトである可能性があります。 単純なものは単純にしましょう。
複雑になることはありますが、そうなった場合はもう一度単純化しましょう。

上記の項目はそれぞれ重要であり、適切なユースケースには、少なくとも1つが必要ではありますが、すべての項目が一致することは珍しいことではありません。

モバイルアプリケーションのバックエンドを構築しているとしましょう。 クライアント（エンドユーザー）は自由に更新できるため、同時に複数のバージョンのインターフェースをサポートする必要があります。 WebServicesを使用するなど、契約によって開発すると、データインターフェースを素早く適応させる能力が低下する可能性があります（ただし、安定性が増す可能性はあります）。 ドキュメントデータモデルには柔軟性が備わっているため、スキーマを素早く進化させ、特定のレコード型の複数のバージョンを処理し、それでもクエリで相関させることができます。

その他のリソース

この魅力的な新機能についてさらに詳しく知りたい方は、利用可能なフィールドテストバージョン2016.2または2016.3を入手してください。

『ドキュメントデータモデル学習パス』を必ず確認してください。
https://beta.learning.intersystems.com/course/view.php?id=9

今年のグローバルサミットのセッションをお見逃しなく。
「データモデリングリソースガイド」には、関連するすべてのセッションが集められています。
https://beta.learning.intersystems.com/course/view.php?id=106

最後に、開発者コミュニティに参加し、質問を投稿しましょう。また、フィードバックもお待ちしています。

#JSON #SQL #データモデル #ドキュメントデータモデル（NoSQL）

ソースURL:
https://jp.community.intersystems.com/post/cach%C3%A9-20162%E3%81%AE%E3%83%89%E3%82%AD%E3%83%A5%E3%83%A1%E3%83%B3%E3%83%88%E3%83%87%E3%83%BC%E3%82%BF%E3%83%A2%E3%83%87%E3%83%AB%E3%81%AE%E7%B4%B9%E4%BB%8B