Morphia と MongoDB によるドメイン・モデルの永続化

Morphia を使用して、MongoDB にマッピングされた Java ドメイン・モデルを永続化、ロード、削除する方法、そしてこのモデルに対してクエリーを実行する方法

John D'Emic
Senior Software Engineer
IBM

2011年 1月 25日

Morphia は、オープンソースのドキュメント指向データベースである MongoDB のためのタイプ・セーフなオブジェクト・マッピング・ライブラリーです。この記事ではまず、ドキュメントとオブジェクトとの間でマッピングをすることによってもたらされるメリットを説明し、Morphia を使用してマッピングを行う方法を示します。その後、MongoDB にマッピングされた Java™ ドメイン・モデルを永続化、ロード、削除する方法、そしてこのモデルに対してクエリーを実行する方法をサンプル・コードで具体的に説明します。

MongoDB は、JSON (JavaScript Object Notation) 形式のドキュメントを保管および取得するためのドキュメント指向のデータベースです。索引付け、レプリケーション、そしてシャーディングの機能が増強された MongoDB は、堅牢でスケーラブルな NoSQL の有力候補として登場しました (「参考文献」を参照)。

MongoDB を操作するには、公式の Java ドライバーを使用することができます。このドライバーでは `BasicDBObject` という `Map` 実装で、データストア内のドキュメントを表現できるようになっています。`Map` 表現は便利ですが、ドキュメントを Java クラス階層としても表現できると便利です。これは JSON を対象としたシリアライズ/デシリアライズを行う場合に、特に言えることです。ドキュメントを Java ドメイン・モデルに、あるいは Java ドメイン・モデルをドキュメントにマッピングすると、MongoDB によるスキーマレスの開発がもたらすメリットを得られると同時に、例えば Java 層におけるタイプ・セーフを実現することができます。しかも、多くの Java フレームワークは POJO (Plain Old Java Object) を使用することを前提としているか、POJO の処理能力に優れています。

POJO を MongoDB にドキュメントとして永続化したり、MongoDB に保管された POJO に対して取得、削除といった操作やクエリーなどを実行したりするには、Apache ライセンスの下に提供される Google Code のプロジェクト、Morphia を使用することができます。Morphia はこの役割を果たすために、アノテーションのセットと、Mongo Java ドライバーのラッパーを提供します。概念的には、Morphia は JPA (Java Persistence API) 実装や JDO (Java Data Objects) 実装などのオブジェ

クト・リレーショナル・マッパーと同様です。この記事では、MongoDB にマッピングされた Java ドメイン・モデルで Morphia を使用する方法を紹介します。サンプル・コード一式を入手するには、「ダウンロード」を参照してください。

ドメイン・モデルの定義

この記事では、単純化したドメイン・モデルを使用して Morphia の機能を具体的に説明します。BandManager という仮想の Web アプリケーションは、音楽活動に関するデータとして、バンドのメンバー、事務所、バック・カタログ、ジャンル、その他の情報を提供します。このドメイン・モデルを表現するために、`Band`、`Song`、`Distributor`、`ContactInfo` という 4 つのクラスを定義します (図 1 を参照)。

図 1. BandManager のクラス

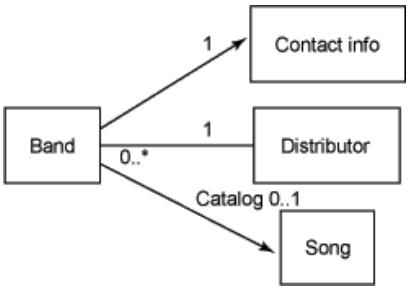

図 1 の UML (Unified Modeling Language) 図には、ドメイン・モデルのクラス階層が示されています。左側の四角形が表しているのは、`Band` クラスです。右側に縦に並んでいる四角形は、それぞれ `ContactInfo`、`Distributor`、`Song` クラスを表します。`Band` から `ContactInfo` に向かっている矢印の `ContactInfo` 側には「1」と示されていて、これはこの 2 つのクラスの間に 1 対 1 の関係があることを示しています。`Band` と `Distributor` を結ぶ線には、`Band` 側に「0..*」、`Distributor` 側に「1」と示されています。これが意味するのは、1 つの `Band` に関係する `Distributor` は 1 つだけであるのに対し、1 つの `Distributor` に関係する `Band` は複数あるということです。最後に、`Band` から `Song` に引かれた矢印には `Song` 側に「Catalog 0..1」と示されています。これは、`Band` は `Song` に対して 1 対多の関係を持ち、この関係には `Catalog` という名前が付けられていることを意味します。

以上のクラスにアノテーションを付けた後、Morphia の `Datastore` インターフェースを使用して 4 つのクラスをドキュメントとして MongoDB に保存します。

ドメイン・モデルにアノテーションを付ける

リスト 1 に、アノテーションを付けた `Band` クラスを記載します。

リスト 1. Band.java

```
@Entity("bands")
public class Band {

    @Id
    ObjectId id;

    String name;

    String genre;

    @Reference
```

```
    Distributor distributor;

    @Reference("catalog")
    List<Song> songs = new ArrayList<Song>();

    @Embedded
    List<String> members = new ArrayList<String>();

    @Embedded("info")
    ContactInfo info;
```

このクラスには、`@Entity` アノテーションが必要です。このアノテーションは、クラスを専用コレクション内にドキュメントとして永続化することを宣言します。`@Entity` アノテーションに指定されている “`bands`” という値は、コレクションの名前を定義しています。デフォルトでは、Morphia はクラス名を使用してコレクションに名前を付けるので、“`bands`” という値が省略されているとすると、データベース内のコレクション名は `Band` になります。

データ型

MongoDB がサポートするデータ型は Java 言語よりも少なく、具体的には `integer`、`long`、`double`、および `string` に限られます。Morphia は、Java の基本的な型 (`float` など) については自動的に変換してくれます。

`@Id` アノテーションは Morphia に対し、ドキュメントの ID として使用するフィールドを指示します。オブジェクトを永続化するときに、そのオブジェクトに `@Id` アノテーションが付けられたフィールドが無い場合には、Morphia が ID 値を自動生成します。

Morphia では、アノテーションが付けられていないフィールドは、`@Transient` アノテーションが付けられない限り永続化されます。上記の例では、`name` プロパティーと `genre` プロパティーが、それぞれ `name`、`genre` というキーの付いた `string` としてドキュメント内に保存されます。

`distributor`、`songs`、`members`、`info` の各プロパティーは他のオブジェクトを参照します。この後すぐにわかるように、メンバー・オブジェクトは、`@Reference` アノテーションが付いていない限り、オブジェクトの中に埋め込まれるオブジェクトであると見なされます。つまりコレクション内では、親ドキュメントの子になるということです。例えば、`members` という `List` を永続化すると、以下のような内容になります。

```
"members" : [ "Jim", "Joe", "Frank", "Tom"]
```

`info` プロパティーもオブジェクトの中に埋め込まれるオブジェクトですが、この場合、`@Embedded` アノテーションに “`info`” という値を明示的に設定しています。デフォルトでは、ドキュメント内の子であるこのプロパティーには `contactInfo` という名前が付けられますが、この明示的な設定によってデフォルトの名前が上書きされます。したがって、以下のようになります。

```
"info" : { "city" : "Brooklyn", "phoneNumber" : "718-555-5555" }
```

@Reference と DBRef

別のコレクションのオブジェクトを参照する場合、Morphia は内部で Mongo の `DBRef` を使用します。

オブジェクトが別のコレクションに含まれるドキュメントへの参照であることを示すには、`@Reference` アノテーションを使用します。Morphia はこのアノテーションが付けられたオブジェクトをMongo コレクションからロードするときに、これらの参照を辿ってオブジェクト・グラフを作成します。ドキュメントを永続化すると、そのドキュメント内で `distributor` プロパティーは以下のように表されます。

```
"distributor" : { "$ref" : "distributors", "$id" : ObjectId("4cf7ba6fd8d6daa68a510e8b") }
```

`@Embedded` アノテーションの場合と同じく、`@Reference` には値を指定することで、デフォルトの名前を上書きすることができます。この例では、`songs` の `List` はドキュメント内で `catalog` という名前になります。

次は、`Song`、`Distributor`、`ContactInfo` のクラス定義を見てください。まず、リスト 2 に `Song` のクラス定義を記載します。

リスト 2. Song.java

```
@Entity("songs")
public class Song {

    @Id
    ObjectId id;

    String name;
```

リスト 3 は `Distributor` のクラス定義です。

リスト 3. Distributor.java

```
@Entity("distributors")
public class Distributor {

    @Id
    ObjectId id;

    String name;

    @Reference
    List<Band> bands = new ArrayList<Band>();
```

`ContactInfo` のクラス定義はリスト 4 のとおりです。

リスト 4. ContactInfo.java

```
public class ContactInfo {


    public ContactInfo() {
    }

    String city;

    String phoneNumber;
```

`ContactInfo` クラスには `@Entity` アノテーションがありませんが、`ContactInfo` には専用コレクションが必要ないため、意図的にアノテーションを付けていません。したがって、このクラスのインスタンスは常に `band` ドキュメントに組み込まれることになります。

ドメイン・モデルを定義し、アノテーションを付けたところで、ここからは Morphia の `Datastore` を使用してエンティティーを保存、ロード、削除する方法を説明します。

`Datastore` の使用方法

依存性注入 (DI)

`Datastore` と Mongo は、どちらも DI フレンドリーなので、この 2 つを例えば Spring や Guice で関連付ける作業に手を焼くことはありません。可能であれば、このそれぞれをシングルトンとして構成し、互いに連携する Bean の間で共有してください。

MongoDB 内のエンティティーを管理するには、`Datastore` インターフェース (Mongo Java ドライバーのラッパー) を使用します。`Datastore` をインスタンス化するには Mongo インスタンスが必要になります。それには、既存の Mongo インスタンスを再利用することも、環境に合わせたインスタンスを構成することもできます。以下は、ローカル Mongo インスタンスに接続する `Datastore` をインスタンス化する例です。

```
Mongo mongo = new Mongo("localhost");
Datastore datastore = new Morphia().createDatastore(mongo, "bandmanager");
```

次に、`Band` のインスタンスを作成します。

```
Band band = new Band();
band.setName("Love Burger");
band.getMembers().add("Jim");
band.getMembers().add("Joe");
band.getMembers().add("Frank");
band.getMembers().add("Tom");
band.setGenre("Rock");
```

`Band` のインスタンス (band) が用意できたので、`datastore` を使ってこのインスタンスを永続化することができます。

```
datastore.save(band);
```

これで、`band` は `bandmanager` データベース内の `band` という名前のコレクションに保存されました。Mongo のコマンドライン・インターフェース・クライアントを使用すれば、コレクション内に保存されたインスタンスを確認することができます (これ以降のサンプル・コードは、ページ幅に収まるように改行が挿入されています)。

```
> db.bands.find();
{ "_id" : ObjectId("4cf7cbf9e4b3ae2526d72587"), "className" :
"com.bandmanager.model.Band", "name" : "Love Burger", "genre" : "Rock",
"members" : [ "Jim", "Joe", "Frank", "Tom" ] }
```

上出来です。インスタンスはしっかり保存されています。すべては期待通りの内容になっているように見えますが、`className` フィールドはその限りではありません。このフィールドは、Morphia が MongoDB 内のオブジェクトの型を記録するために自動的に作成します。その主

な用途は、(例えば、コレクションからさまざまな型が混在したオブジェクトをロードするときなど) コンパイル時に必ずしも既知になっているわけではないオブジェクトの型を判断するためです。この機能が厄介で、必要ではないことがわかっている場合には、`className` を無効にして永続化されないようにすることもできます。それには、以下のように `@Entity` アノテーションに `noClassnameStored` という値を追加してください。

```
@Entity(value="bands",noClassnameStored=true)
```

今度は `Band` をロードして、このクラスが先ほど永続化した `band` インスタンスと同じであることをアサートします。

```
assert(band.equals(datastore.get(Band.class, band.getId())));
```

`Datastore` の `get()` メソッドでは、エンティティーをロードするために、エンティティーの ID を使用することができます。コレクションを指定する必要も、オブジェクトをロードするためのクエリー・ストリングを定義する必要もありません。`Datastore` に対し、ロードしたいクラスとその ID を指定するだけで十分です。あとはすべて、Morphia が引き受けてくれます。

次は、`Band` の連携オブジェクトについて調べてみましょう。まずは、`Song` のインスタンスをいくつか定義するところから取り掛かり、その定義したオブジェクトを先ほど作成した `Band` インスタンスに追加します。

```
Song song1 = new Song("Stairway");
Song song2 = new Song("Free Bird");

datastore.save(song1);
datastore.save(song2);
```

ここで Mongo 内の `songs` コレクションを確認してみると、以下のような内容が表示されるはずです。

```
> db.songs.find();
{ "_id" : ObjectId("4cf7d249c25eae25028ae5be"), "className" :
"com.bandmanager.model.Song", "name" : "Stairway" }
{ "_id" : ObjectId("4cf7d249c25eae25038ae5be"), "className" :
"com. bandmanager.model.Song", "name" : "Free Bird" }
```

この時点では、`Song` は `band` から参照されていないことに注意してください。そこで、先ほど定義した Song のインスタンスを `band` に追加して、どうなるかを見てみます。

```
band.getSongs().add(song1);
band.getSongs().add(song2);

datastore.save(band);
```

これで `bands` コレクションを照会してみると、以下の内容が表示されるはずです。

```
{ "_id" : ObjectId("4cf7d249c25eae25018ae5be"), "name" : "Love Burger", "genre" : "Rock",
   "catalog" : [
   {
      "$ref" : "songs",
      "$id" : ObjectId("4cf7d249c25eae25028ae5be")
   },
   {
      "$ref" : "songs",
      "$id" : ObjectId("4cf7d249c25eae25038ae5be")
   }
], "members" : [ "Jim", "Joe", "Frank", "Tom"] }
```

トランザクション

重要な点として、MongoDB は大抵のリレーショナル・データベース管理システムとは異なり、トランザクションをサポートしないことに注意してください。コレクションの書き込み/読み取りを行う複数のスレッドを調整しなければならないアプリケーションの場合には、Java 言語のシリアライズおよび並行性機能を利用する必要があります。

`songs` コレクションは、`catalog` という配列に 2 つの `DBRef` として保存されていることに注目してください。

現行の制約として、参照先のオブジェクトを保存してからでないと、他のオブジェクトがそのオブジェクトを参照することはできません。そのため、`song1` と `song2` を保存してから `band` に追加したのです。

ここで、`song2` を削除します。

```
datastore.delete(song2);
```

`songs` コレクションを照会すると、`song2` が見つからないと示されます。けれども `band` を調べてみると、そこにはまだ song があります。さらに悪いことに、`band` エンティティーをロードしようとすると例外が発生します。

```
Caused by: com.google.code.morphia.mapping.MappingException: The
reference({ "$ref" : "songs", "$id" : "4cf7d249c25eae25038ae5be" }) could not be
fetched for com.bandmanager.model.Band.songs
```

とりあえず、このエラーを回避するには、song を削除する前に、song への参照をなくすしかありません。

クエリー

今のところはまだ、エンティティーをその ID でロードできるようになっただけです。最終的には、Mongo に対して目的のエンティティーを照会するクエリーを実行できるようにならなければなりません。

`band` をその ID でロードする代わりに、今度はその名前を使って band を照会するクエリーを実行します。それには、`Query` オブジェクトを作成し、目的の結果に絞り込むためのフィルターを指定します。

```
Query query = datastore.createQuery(Band.class).filter("name = ","Love Burger");
```

`createQuery()` メソッドには、照会したいクラス (`Band`) とフィルターを指定します。クエリーを定義した後は、以下のように `asList()` メソッドを使用して、結果にアクセスできるようになります。

```
Band band = (Band) query.asList().get(0);
```

Morphia のフィルター演算子は、MongoDB クエリーで使用されているクエリー演算子と密接に対応しています。例えば、上記のクエリーで使用した `=` 演算子は、MongoDB の `$eq` 演算子と同様です。フィルター演算子に関する詳細は、Morphia のオンライン・マニュアル (「参考文献」を参照) で調べてください。

フィルター・クエリーの代わりの手段として、Morphia にはクエリーを作成するための流れるようなインターフェースが用意されています。以下に一例として記載する、流れるようなインターフェースのクエリーは、上記のフィルター・クエリーとまったく同じです。

```
Query query = datastore.createQuery(Band.class).field("name").equal("Love Burger");
```

組み込みオブジェクトのクエリーには、「ドット表記」を使用することができます。以下は、ドット表記と流れるようなインターフェースを使用して、Brooklyn を拠点とするすべてのバンドを選択するクエリーです。

```
Query query = datastore.createQuery(Band.class).field("info.city").equal("Brooklyn");
```

クエリーの結果セットを絞り込んで定義することもできます。上記のクエリーを変更して、バンドを名前でソートして結果を 100 に絞り込むとすると、以下のクエリーになります。

```
Query query =
datastore.createQuery(Band.class).field("info.city").equal
("Brooklyn").order("name").limit(100);
```

索引付け

コレクションが大きくなるにつれ、クエリーのパフォーマンスが劣化していくことに気付くはずです。クエリーのパフォーマンスをある程度維持するには、リレーショナル・データベースのテーブルと同じように、Mongo のコレクションにも適切に索引を付ける必要があります。

プロパティーに `@Indexed` でアノテーションを付けるということは、フィールドに索引を付けるということです。ここでは `Band` の `genre` プロパティーに対して `genreName` という昇順の索引を作成します。

```
@Indexed(value = IndexDirection.ASC, name = "genreName")
String genre;
```

索引を適用するには、Morphia はどのクラスがマッピングされるのかを把握している必要があります。索引が適用されることを確実にするとなると、Morphia をインスタンス化する方法が少し異なってきます。その方法は、以下のとおりです。

```
Morphia morphia = new Morphia();
morphia.mapPackage("com.bandmanager.model");
datastore = morphia.createDatastore(mongo, "bandmanager");
datastore.ensureIndexes();
```

最後の `ensureIndexes()` の呼び出しによって、データベースは (既存の索引がない場合には) 必要な索引を作成します。

索引は、重複するエンティティーがコレクションに挿入されないようにするためにも使用することができます。例えば、`band` の名前の `@Indexed` アノテーションに `unique` プロパティーを設定することによって、特定の名前の `band` がコレクション内に 1 つしか存在しないことを確実にすることができます。

```
@Indexed(value = IndexDirection.ASC, name = "bandName", unique = true)
String name;
```

これ以降、同じ名前を持つ他の `band` は破棄されることになります。

まとめ

Morphia は、MongoDB を操作するための強力なツールです。Morphia によって、MongoDB のドキュメントへのタイプ・セーフなイディオムのようなアクセスが可能になります。この記事では、Morphia の主な操作方法を説明しましたが、取り上げていない機能もあります。是非、Morphia Google Code プロジェクトを調べて、その DAO (Data Access Object: データ・アクセス・オブジェクト) のサポート、検証、そして手動マッピング機能について学んでください。

ダウンロード

内容	ファイル名	サイズ
Sample code for this article	j-morphia.zip	17.2KB

著者について

John D'Emic

John D'Emic は、IBM Global Services のシニア・ソフトウェア・エンジニアです。この 1 年の間、MongoDB をさまざまな開発コンテキストで使用してきました。彼は、『Mule in Action』(Manning Publications、2009年) の共著者です (もう 1 人の著者は David Dossot です)。