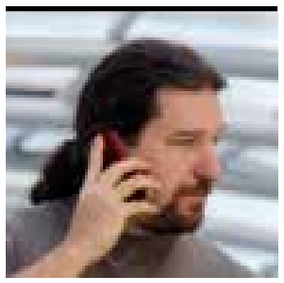

パート1

Oracle Berkeley DB Java Editionの Java API

TED NEWARD

BIO

Oracle Berkeley DB Java EditionのJava APIの動作を学習する

2005年夏、Microsoftのカンファレンスで、当時技術者の間で話題になっていたオブジェクト・リレーショナル・マッピング（ORM）ツールについて、見解を求められたことがあります。私の回答は、予想したとおり、注目を集めました。私は、「ORMはコンピュータ・サイエンスにとってのベトナム戦争だ」と答えたのです。

そのとき、私は間違いなく「流れに逆らって」いました。データの保存に適当な場所はリレーショナル・データベースしかなく、それも大規模なものが望ましいと「だれもが」考えていました。大きいほどよいというのは確かです。大容量のRAM、大容量のディスク領域、高速なI/O、より多い作業量を望まない人などいません。

私の意見は、1つには、たとえばJavaのオブジェクト・グラフとデータベースにおける矩形の表のように「形状」が異なるデータでは、どうしても不整合が生じるということでした。このような不整合は、どんなに洗練されたツールやライブラリをもってしても、解消することが非常に困難です。特に、階層構造を1つの表に押し込むことはとても難しく、Facebookプラットフォームに見られるような閉路グラフに至っては、表にすることはほとんど不可能です。不可能ではないとしても、このたった1つの概念を説くために1冊の書籍が必要となる（たとえばJoe Celko氏の『Trees and Hierarchies in SQL for Smarties』）ことを考えれば、必要となる作業量は現実的ではないでしょう。

その後、NoSQL運動が広まるにつれて、開発者は、データを格納するべき場所はリレーショナル・データベースだけではないかもしれないということを理解するようになりました。リレーショナル・データベースは、分析、レポート、変換などのための優れたツールがあるため、データが最終的に行き着く場所であるかもしれません。しかし、だからと言って、リレーショナル・データベースが唯一の利用可能なストレージ・システムであるということにはなりません。

リレーショナル・データベース以外のストレージ・システムは、数十年にわたって、目立たないながらも、適切に、効率的に、そしてコンパクトにデータを処理してきました。その1つであるBerkeley DBは、ここ数年、UNIXベースのシステムにとって重要な要素となっています。オラクルは、何年も前にBerkeley DBを買収し、Berkeley DBのJava APIを開発してきました。

Oracle Berkeley DB Java Editionはとても簡単に使用できます。その理由の1つは、Berkeley DBが表面上はリレーショナル・モデルと似ているためです。しかし、誤ってはなりません。Berkeley DBをリレーショナル・データベースと同じように使用しようとすれば、深刻な失敗に終わるでしょう。

RDBMSを超えて
NoSQL運動が広まるにつれて、開発者は、データを格納するべき場所はリレーショナル・データベースだけではないかもしれないということを理解するようになりました。

APIの要点

Oracle Berkeley DB Java EditionのJava APIには、多くのデータベースAPIと異なり、2つの「レベル」があります。

第1のレベルである基本APIは、本質的に、すべてのデータをキーと値のペアと見なす低レベルの読取り/書込み操作で構成されます。これらの操作で扱う値はバイト配列であり、Java開発者であればJavaの標準的なシリアライズかOracle Berkeley DB Java Editionの「バインド（bind）」APIを使用して作成と解析を行うことができるでしょう。

原則として、第1レベルのストレージ操作をあえて行う必要がある場合を除いては、このようなアクセス方法もあるということを認識した上で、第2レベルのAPIに目を向けることが有効です（第1レベルの操作が必要になる唯一と言ってもよい目的は、主キーの重複をサポートすることです）。

CベースのBerkeley DB APIを愛用してきた開発者は、おそらく基本APIを利用したいと考えるでしょう。しかし、注意が必要です。Oracle Berkeley DB Java Editionのマニュアルには、ファイル形式がC版で使用されている形式と同一ではないため、C版のAPIとJava EditionのAPIの間でファイルを移植することはできないと明記されています。ただし、Oracle Berkeley DB Java Editionのファイルを異なるJavaプラットフォーム間で移植することは可能です。

第2のレベルであるDirect Persistence Layer（DPL）APIは、

写真：
PHIL SALTONSTALL

よりオブジェクト中心の方法でデータ・ストレージを操作するため、Java Persistence API（JPA）などのORMツールに精通している開発者にとって利用しやすいAPIです。ただし、Oracle Berkeley DB Java EditionはORMではないという点に注意することが重要です。Oracle Berkeley DB Java Editionでは、ストレージ・ファイルに実際の表がなく、オブジェクトはデータベースに直接保存されます。そのため、オブジェクトを矩形の表に押し込むために時間を費やすことはありません。

探索的テスト

新しいAPIを使用する際に、私が好んで利用する手法は、探索的テストと呼ばれるものです。基本的にはいくつかの単体テストを作成しますが、その目的は、対象製品のテストではありません。探索的テストの目的には、以下があります。

- 1つの単純なフレームワークに沿ってさまざまな操作を試すこと
- 段階的に少しずつ調査を進めながら探索を続けること
- 最初にいくつかのアサーションを記述し、それらのアサーションが満たされるか検証すること
- もっとも重要な目的として、製品の新バージョンが登場した際、新バージョンに対してテストを実行し、変更点を把握すること

前提となる知識

一般に、プログラムでデータベースのデータを格納または取得するためには、まずデータベースに接続する、すなわちデータベースを「開く（open）」必要があります。従来のクライアント/サーバー型RDBMSの場合は、データベースを開くために、ネットワークの情報と認証情報を指定し、（通常は）データベースに対するTCP/IPソケットを開いて認証します。一方、Oracle Berkeley DB Java Editionのような組込みデータベースの場合は、データベースを開くために、通常、ディスク上のデータベースの場所と、データベースが存在しない場合に新規作成するかどうかをAPIに指定します。

Oracle Berkeley DB Java EditionのAPIでは、データベース環境を表す**Environment**オブジェクトを作成することでデータベースを開きます。この処理には、Javaの**File**オブジェクトと**EnvironmentConfig**オブジェクトが必要です。Fileオブジェクトは、データ・ファイルの保存先となるディレクトリを示します。このディレクトリは事前に作成する必要があります。EnvironmentConfigオブジェクトは、Oracle Berkeley DB Java Editionに対し、特定の条件でどのように動作するかを指示します。

たとえば、探索的テストを簡潔に保つため、実行のたびにデータベースの作成と破棄を行うとします。その場合、リスト1に示すように、**EnvironmentConfig**の**allowCreate**プロパティを**true**に設定する必要があります。次に、**リスト1**からわかるように、データベース環境を閉じるために**close()**を呼び出します。

探索的テストを実行するディレクトリ内に、**someSubdir**というディレクトリが存在するとします。その中にはファイルが1つではなく、いくつか格納されています。それらのファイルの中で特に重要なファイルは、拡張子が.jdbのファイルです。.jdbファイルには1ずつ増加する連番の名前が付けられます。最初のファイルの名前は00000000.jdbになります。このファイルには、既存のデータに追加される形でデータが次々と書き込まれます。

Oracle Berkeley DB Java Editionのマニュアルには、このようなファイルをバックアップする際、0から昇順にファイルがコピーされる場合であれば、データベースを閉じる必要はないと注記されています。

EntityStore

DPL APIを使用するためには、データベース環境を開いた後、**Environment**の上で**EntityStore**も開く必要があります（DPLは、前述のように、基盤となるキー/バイト配列値ペアのストアの上位に位置する抽象レベルです）。

Environmentの設定オプションを表すためには**EnvironmentConfig**を使用しますが、これと同様に、**EntityStore**の作成時には**EntityStoreConfig**を使用します。また、**EntityStoreConfig**では、データベース・ファイルが存在しない場合は新規作成するように指定する必要がありますが、**EntityStoreConfig**でも**allowCreate**プロパティを使用して同じ設定を行います。

これまで説明したすべてのAPI（およびDPL APIに関係するほぼすべての要素）は、com.sleepycat.persistパッケージにあります。

今後のテストのために、これまでのコー

リスト1

```
@Test public void openAndCloseADatabase()
  throws DatabaseException
{
  EnvironmentConfig config = new EnvironmentConfig();
  config.setAllowCreate(true);
  Environment dbEnv =
    new Environment(new File("./someSubdir"), config);

  dbEnv.close();
}
```

全てのリストをテキストで表示

ドを、**@Before**アノテーションを付加した**openDatabase**メソッドと、@Afterアノテーションを付加した**closeDatabase** メソッドに移します（**リスト2**参照）。

このコードでは、テスト後にデータ・ファイルが残らないようにするため、意図的に、テストのたびにディレクトリの削除と再作成を行っています。当然ながら、本番システムでは異なるロジックを使用します。

Entity

DPL APIを使用する場合、JPAなどのオブジェクトベースのストレージ・システムと同様、オブジェクトのインスタンスの格納と取得を行います。

Oracle Berkeley DB Java Editionでは、クラスに**@Entity**アノテーションを付加することで、オブジェクトを永続化することを示します。さらに、クラスの1つ以上のフィールドに@PrimaryKeyアノテーションを付加する必要もあります。**@PrimaryKey**アノテーションの目的は名前のとおり主キーの指定のように思われますが、実際の使用法は他のORMシステムと異なります。

本記事では、例として、ブログ・エンジン・システムを扱います（**リスト3**参照）。

新規ブログ記事を**EntityStore**に保存することは、**EntityStore**のある種の「保存」メソッドを使用すれば可能ではないかと思いがちですが、それほど単純ではありません。Oracle Berkeley DB Java Editionエンジンでは、ストア内で定義されるインデックスに細心の注意を払う必要があります。インデックスとなることがもっとも明らかなものは、**@PrimaryKey**が付加された特定の型のフィールドです。

主キーを使用してオブジェクトの格納や取得を行うには、まず、EntityStoreから**PrimaryIndex<K,V>** 型のインデックス（**K**は**@PrimaryKey**アノテーションが付加されたフィールドの型、**V**はそのフィールドが定義されている、**@Entity**アノテーションが付加されたクラスの型）を取得する必要があります。

これにより、**PrimaryIndex**のputメソッドで、オブジェクトを**EntityStore**に格納できます。また、**PrimaryKey**の**get**メソッドで、主キーを使用してオブジェクトを**EntityStore**から取得できます（**リスト4**参照）。

SQL文はまったく使用しません。これは大きなメリットですが、前提として、オブジェクトを検索する際に対象のオブジェクト（または少なくともその主キーの値）が正確にわかっている必要があります。しかしながら、ブログ記事の最新10件のリストを取得する場合など、オブジェクトのリスト全体を取得しなければならない場合があります。リスト全体を取得する場合には、エンティティに対するイテレータが必要となります。このイテレータも、**PrimaryIndex<>**オブジェクトから取得します（**リスト5参照**）。

繰り返し処理の完了時に、カーソルに対して**close()**を呼び出している点に注意してください。close()を呼び出さない場合、データベース自体を閉じる際にデータベースのランタイムで例外が発生します（**close(**は**finally**ブロック内に置くことが最適ですが、このコードは探索的テスト用であるため、このままにします）。

主キーを人工キー（単調増加する数値シーケンスなど）にする必要がある場合、Oracle Berkeley DB Java EditionのAPIでは、**@PrimaryKey** アノテーションを変更して「sequence」を使用することで、キーを自動的に生成できます。たとえば次のように記述します。

```
private @PrimaryKey(sequence=
"Sequence_Namespace") int id;
```

こうして生成された主キーは、GUID（グローバル一意識別子）と同等です。つまり、キーには意味がなく、キーの値とオブジェクトの実際の内容には関連がありません。そのため、主キーを直接使用してオブジェクトの参照や取得を行うことができなくなります。一方で、主キーを変更可能にする場合に発生する面倒な問題を避けることができます。

リスト2　リスト3　リスト4　リスト5

```
public class BDBTest
{
  // ...

  File dbDir = new File("./data");
  Environment dbEnv;
  EntityStore dbStore;
  @Before public void openDatabase()
  {
    if (!dbDir.exists())
      dbDir.mkdir();

    EnvironmentConfig config = new EnvironmentConfig();
    config.setAllowCreate(true);
    dbEnv = new Environment(dbDir, config);

    StoreConfig storeConfig = new StoreConfig();
    storeConfig.setAllowCreate(true);
    dbStore = new EntityStore(dbEnv, "EntityStore", storeConfig);
  }
  @After public void closeDatabase()
  {
    dbStore.close();

    dbEnv.close();

    dbDir.delete();
  }
}
```

全てのリストをテキストで表示

セカンダリ・キー

ここまで、主キーによる検索には問題はありませんが、主キー以外の条件でオブジェクトを取得しなければならないこともあります。たとえばブログ・システムでは、特定のタイトルのエントリではなく、特定の日付のエントリを表示する場合が少なくありません。そのような場合、データベースでは、主キー以外の検索スキームを使用してオブジェクトを取得する必要があります。

Oracle Berkeley DB Java Editionでは、フィールドに**@SecondaryKey**アノテーショ

ンを付加することで、セカンダリ・キーによる検索が可能になります。検索するには、**リスト6**に示すように、まず**SecondaryIndex**インスタンスを取得し、**EntityStore**から必要なオブジェクトを取得します。

SecondaryIndex<>型は**PrimaryIndex<>**型と直接結び付いています。第1の総称型パラメータは@**SecondaryKey**アノテーションが付加されたフィールドの型、第2の総称型パラメータは**@PrimaryKey**アノテーションが付加されたフィールドの型、そして第3の型パラメータは**Entity**の型そのものです。コンストラクタではパラメータとして、インデックス、@**SecondaryKey**アノテーションが付加されたフィールドの型、フィールド名を使用します。

さらに、**SecondaryIndex**に基づいて特定の「範囲」にあるオブジェクトを取得する必要がある場合は、**entities()**を呼び出す際に、範囲の境界を示す2つの値（1つが「開始」、もう1つが「終了」）を指定します。それぞれの値には、その値を範囲に含めるかどうかを示すBoolean型の値を指定します（**true**は「含める」、**false**は「含めない」）。**リスト7**を参照してください。

ところで、**SecondaryKey**インスタンスとその値との関係には、さまざまな可能性があります。この関係は、アノテーションの記述部で**relate**パラメータを使用して表します。詳細についてはJavadocに記載されていますが、基本的には、どのようなデータベース・システムにも見られる、1:1、1:N、N:1、M:Nの関係のいずれかです。ブログの場合は複数のブログ記事を1日に投稿できるため、**BlogPost**型の**postingDate**フィールドに付加するべき適切なアノテーションは、次のようになります。

```
private @SecondaryKey(relate=
MANY_TO_ONE) Date postingDate;
```

MANY_TO_ONEは**Relationship**の列挙値であるため、上記のコードが動作するためには、**Relationship**型を静的にインポートするか、**Relationship.MANY_TO_ONE**と記述する必要があります。

パート1のまとめ

パート2では、より複雑なオブジェクトを格納する方法について説明しますが、Oracle Berkeley DB Java EditionのDPL APIの動作については、本記事で十分理解できます。Oracle Berkeley DB Java Editionは、本質的には永続オブジェクト・ストアであり、永続オブジェクト・リレーショナル・ストアではありません。そのため、オブジェクトを永続化するための事務的な作業が極めて少なくなります。

実際、ここまでの作業は、リレーショナル・スキーマと比較すればほとんどゼロです。データ定義言語もなければ、立ち上げておくべきサーバー・インスタンスもありません。認証情報を設定する必要もなく、認証ルールもありません。ただディレクトリを1つ作成するだけで、データベースを利用できます。

ただし、1つ注意点があります。将来、**BlogPost**型の機能を拡張する（たとえば、WordPressのようなマルチテナントのブログ・システムにするため、ブログ記事の著者を表す**Author**型を追加する）場合、Oracle Berkeley DB Java EditionではBlogPost型における変更内容をそのまま反映しようとします。そのため、追加的なリファクタリング（フィールドの追加など、追加のみを伴うリファクタリング）は、コードを変更せずに実施できます。しかし、**postingDate** フィールドに@**SecondaryKey**アノテーションを追加するといった変更の場合は、既存のデータ・ストアを拡張するための詳細な作業が必要になります。

これは、探索的テストではそれほど大きな問題になりません。特に、本記事では、テストのたびにデータベースを完全に破棄しているため、考慮する必要はありません。しかし、実際のデータベースでは、バージョン1とバージョン2の間で、このような拡張の問題が発生する可能性があります。Oracle Berkeley DB Java Editionでは、このような拡張を「変異（mutation）」と呼んでいます。リファクタリングで変異が発生する場合、Berkeley DBで対応できる変異もありますが、複雑な変異については、データベースを移行するためのユーティリティの開発が必要になる場合もあります。

最後に、私は、現存するあらゆるリレーショナル・データベースを完全に破棄しOracle Berkeley DB Java EditionやApache CouchDBに置き換えるべきだと主張するつもりはありません。

より複雑な事例を取り上げるパート2をご期待ください。</article>

リスト6　リスト7

```
@Test public void storeAndRetrieveOneByDate()
{
  BlogPost newPosting =
    new BlogPost("The Vietnam of Computer Science");

  PrimaryIndex<String,BlogPost> primaryIndex =
    dbStore.getPrimaryIndex(String.class, BlogPost.class);
  primaryIndex.put(newPosting);

  SecondaryIndex<Date, String, BlogPost> dateIndex =
    dbStore.getSecondaryIndex(primaryIndex, Date.class,
      "postingDate");

  EntityCursor<BlogPost> postCursor =
    dateIndex.entities();
  for (BlogPost post : postCursor)
  {
    assertTrue(post.getTitle().contains("Computer Science"));
  }
  postCursor.close();
}
```

全てのリストをテキストで表示