JPA (Java Persistence API) 2.0

1. JPAとは

JPAとはJava Persistence APIの略で、永続化に関する仕様です。JPA 1.0はEJB 3.0（JSR-317）の一部として策定され、JPA 2.0はJSR-317としてEJBから独立して策定されています。EJB2.1以前のEJBにはEntity Beanという永続化の機構がありました。Entity Beanを使用する開発では、複雑な規約に従う必要があったり、EJBコンテナが必要であったりしました。Entity Beanの問題点を解決するために、Hibernate等のO/Rマッピングツールからアイデアを取り入れ、JPAとして仕様が策定されました。JPAはEJBコンテナの中だけでなく、Java SE環境でも動作させることができます。

JPA 1.0の参照実装はOracleのTopLink Essentialsでした。TopLink EssentialsはEclipse Foundationに寄贈され、現在EclipseLinkという名称になっています。そして、JPA 2.0ではEclipseLinkが参照実装として採用されています。なお、JPAには参照実装以外に、準拠しているプロダクトとしてHibernate、Apache OpenJPAなどがあります。

なお、執筆時点（2009年4月30日）ではJPA 2.0の仕様は確定していませんが、参照実装はGlassFish v3（プレリュード版でない）に含まれています。サンプルもJPA 2.0に対応したものがダウンロードできます。

GlassFish v3
http://wiki.glassfish.java.net/Wiki.jsp?page=Here

Java EE 6サンプル
http://wiki.glassfish.java.net/Wiki.jsp?page=JavaEE6Samples

しかし、執筆時点のGlassFish v3では、JPA 2.0の参照実装が含まれているものの、未実装のクラスがあり、まだ動作させて確認することができませんでした。実際にJPA 2.0を試すにはもう少し待つ必要があるようです。

2. JPA と JDO

永続化を対象とする仕様として、JDO(Java Data Objects)があります。JPAの永続化のターゲットはDBですが、JDOはDBに限らずあらゆるストレージに永続化することができます。どちらもエンティティの永続化を行えますが、JPAはO/Rマッピングに関する仕様で、JDOはオブジェクトの永続化に関する仕様、という目的の違いがあります。どちらが優れているというより、用途に応じて使い分けることになります。また、DataNucleus のようにJDOの実装がJPAも実装しているプロダクトも存在します。

3. JPA 2.0で追加、変更された機能

大きく分けて以下の機能が追加、変更されました。

- **悲観的ロックのサポート**
- **コレクションのサポート強化**
- **組み込みオブジェクトのサポート強化**
- **キャッシュ仕様の追加**
- **Criteria APIの追加**
- JPQL(Java Persistence Query Language)の強化
- Bean Validation(JSR-303)のサポート

このレポートでは、JPQLの強化とBean Validationのサポートは取り上げません。

JPA 1.0からの変更箇所については、JPA 2.0のドキュメント『JSR 317: Java Persistence API, Version 2.0』の最後の「Appendix A Revision History」で見ることができます。このレポートに取り上げなかった変更もそこから読み取ることができます。

3.1. 悲観的ロックのサポート

JPA 1.0まではJPAでは楽観的ロック（Optimistic lock）だけがサポートされていました。JPA 2.0からは悲観的ロック（Pessimistic lock）をサポートすることになり、厳密なデータの整合性を保つ仕組みが提供されることになります。

JPA 2.0が悲観的ロックをサポートすることにより、長期間データの変更がないことを保証する必要がある処理でもJPAを安心して使用することができるようになります。しかし、悲観的ロックが行われている間は別の処理は待機状態になるので、パフォーマンスは低下する恐れがあります。

1）ロックモードの変更

悲観的ロックの追加に伴い、ロックモードが表1から表2に変わっています。

表 1：JPA 1.0 のロックモード一覧

ロックモード	概要
READ	楽観的ロックによる読込
WRITE	楽観的ロックによる書込

表 2：JPA 2.0 のロックモード一覧

ロックモード	概要
READ	OPTIMISTIC のシノニム
WRITE	OPTIMISTIC_FORCE_INCREMENT のシノニム
OPTIMISTIC	楽観的ロックによる読込のモード
OPTIMISTIC_FORCE_INCREMENT	楽観的ロックによる書込と version アノテーションのカラムの自動インクリメントのモード
PESSIMISTIC_READ	悲観的ロックによる読込のモード
PESSIMISTIC_WRITE	悲観的ロックによる書込のモード
PESSIMISTIC_FORCE_INCREMENT	悲観的ロックによる書込と version アノテーションのカラムの自動インクリメントのモード
NONE	デフォルトのロックモード

2）悲観的ロックの使用上の注意点

JPAの悲観的ロックは悲観的ロックの取得方法については定めていません。そのため、悲観的の内容については、JPAの実装に依存しています。つまり、JPAの実装によってはDBの機構を使用するかもしれませんが、使わない可能性もあるということです。

また、悲観的ロックを使用する際に注意しなければならないのは、同一のテーブルに対して悲観的ロック、楽観的ロックの両方でアクセスする場合に不整合が発生する可能性があります。

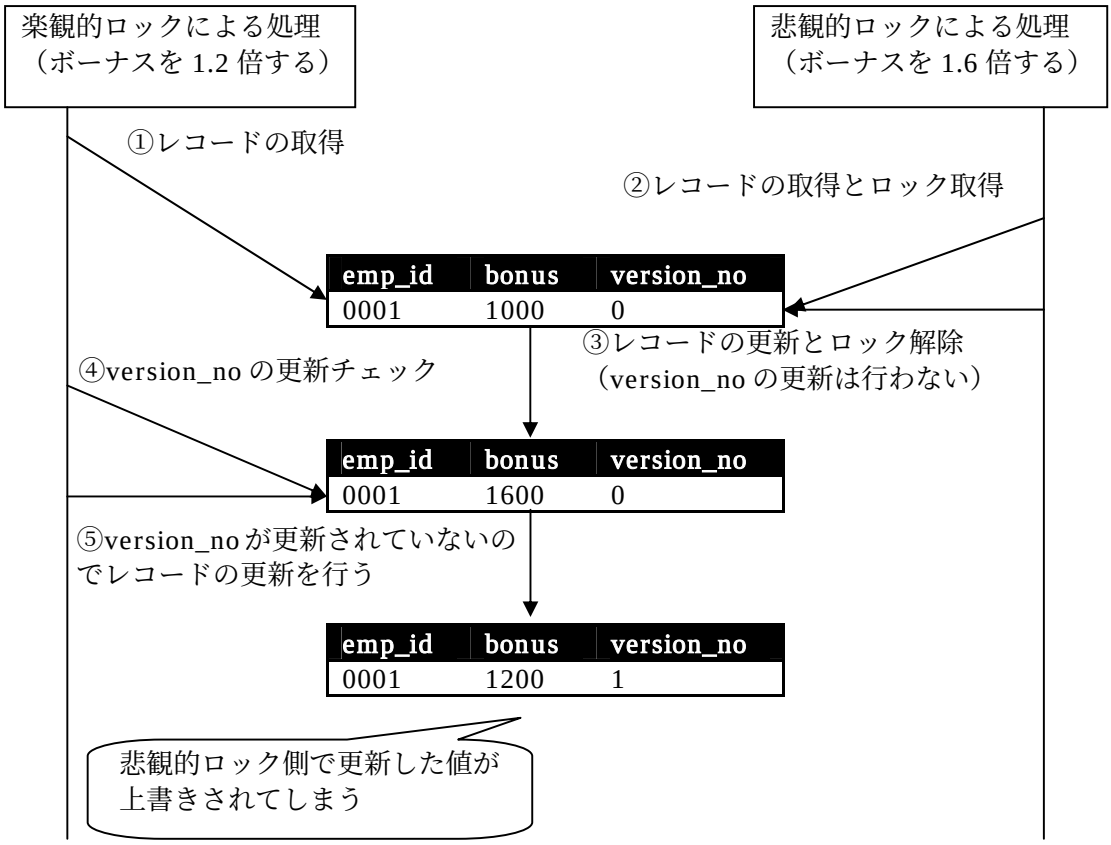
まず、楽観的ロックの仕組みですが、排他制御をversionアノテーション属性の値を見て判断します。読み込んだ時のversionアノテーション属性の値が、書き込み時に変更されていなければ書き込みを行い、変更されていれば書き込みを行わない、という排他制御です。

また、悲観的ロックではversionアノテーションを使用せず、実装に依存したロッ

ク機構で排他制御を行います。

そして、問題となるのは、両方のロックモードでのアクセスされるテーブルの場合、悲観的ロックによるアクセスでもversionアノテーションの属性を更新しないと楽観的ロック側で更新を感知できず、更新を行ってしまい、データの不整合が起きてしまいます。

図 1：悲観的ロックと楽観的ロックを併用する際の不整合の例



3）悲観的ロックのタイムアウト設定とヒント

悲観的ロックを使用する際に、タイムアウト時間を指定し、一定時間内に問い合わせに対する結果が取得できない場合、LockTimeoutExceptionがスローされます。タイムアウト時間の設定はpersistence.xmlのpropertyとして設定することができます。設定値の単位はミリ秒で、ゼロを設定すると既にロックが取得されている場合、即タイムアウトとなります。

リスト2：persistence.xmlでタイムアウト時間を設定する例

```
<?xml …
<persistence …
  <persistence-unit …
    <provider …
    <jta-data-source …
    <properties>
      <property name=”javax.persistence.lock.timeout” value=”3000” />
```

また、タイムアウト時間はpersistence.xmlで設定したものが全体に適用されますが、使用する際にEntityManagerやQueryのメソッドを通じて設定値をその時々に応じて変更することが可能です。つまり、persistence.xmlに設定されたタイムアウト値はデフォルト値としての役割を果たしています。

3.2. コレクションのサポート強化

1）基本データ型のコレクションサポート

JPA 2.0からエンティティで基本データ型のコレクションがサポートされます。また、エンティティが関連づくデータベース上のテーブルの子テーブルと関連付けられ、自動的に永続化されるアノテーションが追加されました。エンティティ内のコレクションにはElementCollectionアノテーションを設定します。

リスト2：ElementCollectionアノテーションの例

```
@Entity public class Person {
 @Id protected String ssn;
 protected String name;
 protected Address home;

 @ElementCollection
 protected Set<String> nickNames = new HashSet();

 ※メソッドは省略

}
```

エンティティ内のコレクションにElementCollectionアノテーションを設定した場合は、エンティティ名とコレクションのプロパティ名から自動的にテーブル名を連想し、コレクションの内容を永続化するためのテーブルを決定します。リスト2のPersonの属性nickNamesはPERSON_NICKNAMESというテーブルに永続化されます。PERSON_NICKNAMESテーブルは、PERSONテーブルと関連付けるPERSON_SSNと、ニックネームを表すNICKNAMESの2つのカラムを持ちます。

また、エンティティ内のコレクションをどのテーブルに永続化させるかを指定することも可能です。その場合、CollectionTableアノテーションで永続化させるテーブルを指定した上で、Columnアノテーションでどの列にデータを格納するかを指定することができます。

リスト3：CollectionTableアノテーションとColumnアノテーションの例

```
@Entity public class Person {
 @Id protected String ssn;
 protected String name;
 protected Address home;

 @ElementCollection
 @CollectionTable(name=”synonym”)
```

```
 @Column(name=”another_name”)
 protected Set<String> nickNames = new HashSet();

 ※メソッドは省略

}
```

2）組み込みオブジェクトのコレクションサポート

「3.3. 組み込みオブジェクトのサポート強化」の「1）組み込みオブジェクトのコレクションサポート」で説明します。

3）常にソートされるリストのサポート

OrderColumnアノテーションにより実現されるソート機能です。JPA 1.0にも似た機能でOrderByアノテーションが存在しましたが、これは取得時にだけ対象のコレクションがソートされます。

OrderColumnアノテーションに指定できるのは、1対多の関係、多対多の関係、コレクションに対してソートを設定することができます。OrderColumnアノテーションを指定すると、データベースに追加、更新、削除の処理が行われるとともに、ソートが実行されます。

なお、OrderByアノテーションとOrderColumnアノテーションの両方が同じコレクションに指定されている場合、OrderColumnアノテーションの設定が優先されます。

リスト4：OrderColumnアノテーションの例

```
@Entity public class Person {
 @Id protected String ssn;
 protected String name;
 protected Address home;

 @ElementCollection
 @OrderColumn
 protected Set<String> nickNames = new HashSet();

 ※メソッドは省略

}
```

3.3. 組み込みオブジェクトのサポート強化

1）組み込みオブジェクトのコレクションサポート

組み込みオブジェクトをコレクションに持つ事がサポートされます。

リスト5：組み込みオブジェクトをコレクションに持つ例

```
@Entity public class Person {
  @Id protected String ssn;
  ...
  @ElementCollection
  public Set<Phone> phones;
  ...
}
```

```
@Embeddable public class Phone {
  protected String phoneNumber;
  protected String status;
}
```

リスト5ではPersonが複数のPhoneを持つ事を表現しています。永続化のされ方はElementCollectionを指定する属性にEntityを設定した場合と同様で、リスト5のPersonの属性のphonesはPERSON_PHONESテーブルに永続化されます。

2）組み込みオブジェクトの多段ネストサポート

組み込みオブジェクトの中に組み込みオブジェクトを入れることが可能になりました。

リスト6：組み込みオブジェクトのネストの例

```
@Embeddable public class Address {
 protected String street;
 protected String city;
 protected String state;
 @Embedded protected Zipcode zipcode;
}
```

```
@Embeddable public class ZipCode {
 protected String zip;
 protected String plusFour;
}
```

リスト6では、組み込みオブジェクト"Address"の中に組み込みオブジェクト"ZipCode"が含まれていることを表しています。

3.4. キャッシュ仕様の追加

キャッシュの仕様が新たに追加されました。実装側では必須ではありませんので、実装によってはこの機能は存在しないかもしれません。キャッシュはデータベース上のデータを一時的に保持し、データベースへのアクセスを効率化するものです。ただし、このキャッシュの仕様は開発者向けのインターフェースを定義しているもので、キャッシュの動作については実装側に任されています。

キャッシュを使用するにはpersistence.xmlにCachingエレメントを記述します。Cachingエレメントの値は表3の中の値から選択します。

表 3：Caching エレメントに設定する値

値	概要
ALL	全てのエンティティをキャッシュする
NONE	一切のエンティティをキャッシュしない
ENABLE_SELECTIVE	選択したものをキャッシュする
DISABLE_SELECTIVE	選択したものをキャッシュしない

Cachingエレメントにデフォルト値は存在しませんので、必ずいずれかの値を設定します。ただし、Cachingエレメント自体がpersistence.xmlに記述されない場合の動作はJPA実装に依存します。

Cachingエレメントに"ENABLE_SELECTIVE"または"DISABLE_SELECTIVE"を設定した場合、Cacheableアノテーションを一緒に使用することになります。キャッシュモードとCacheableアノテーションの組み合わせでキャッシュするエンティティを区別します。キャッシュの対象となるかどうかは表4にある通りです。

表 4：Caching エレメントとキャッシュ設定の対応表

	ENABLE_SELECTIVE	DISABLE_SELECTIVE
Cacheable(true)	キャッシュする	キャッシュする
Cacheable(false)	キャッシュしない	キャッシュしない
Cacheable を設定しない	キャッシュしない	キャッシュする

明示的にCacheableアノテーションが設定されているものは、アノテーションの設定に準じたエンティティのキャッシュを行います。また、Cacheableアノテーションが未定義のエンティティについては、Cachingエレメントの設定に従いキャッシュされるかどうかが決定されます。

キャッシュの動作モードをEntityManagerやQuerysetPropertiesメソッドに設定し、キャッシュの使用方法を指定することもできます。

キャッシュのモードを設定するクラスは表5の2つです。

表 5：キャッシュモードの設定

キャッシュモードクラス	概要
javax.persistence.CacheGetMode	データ取得に関するキャッシュモード
javax.persistence.CachePutMode	データ更新に関するキャッシュモード

キャッシュモードを設定するそれぞれのクラスで取りうる値と、動作の説明は表6と表7の通りです。

表 6：キャッシュモード CacheGetMode の設定値

値	概要
USE	キャッシュを使用しデータを取得する（デフォルト）
BYPASS	キャッシュを DB から直接取得する

表 7：キャッシュモード CachePutMode の設定値

CachePutMode の値	概要
USE	キャッシュに更新をする。DB にはコミット時に書き込まれる。（デフォルト）
BYPASS	キャッシュには更新を反映せず、DB に直接書き込まれる。
REFRESH	強制的にキャッシュを更新する

3.5. Criteria API の追加

Criteria APIはオブジェクトの操作を通じ、DBへの問い合わせを作成するためのAPI群です。Criteria APIはタイプセーフ（データ型に対する安全性）な問い合わせを作成するための方法があります。

Criteria APIによる問い合わせの作成はMetamodel APIを使用した方法と、文字列ベースの方法の2種類があります。

1） Metamodel APIでの問い合わせの作成

Metamodel APIを使用した方法で、DB上のカラムに対してJavaのどのデータ型に対応するのかをMetamodel APIにより定義し、タイプセーフを実現するアプローチです。Metamodel APIを使用するアプローチ方法には以下の3つがあります。

- エンティティクラスに対するメタクラスを手作りする方法
- アノテーションによりメタクラスを生成する方法
- "javax.persistence.metamodel.Metamodel"インターフェースから動的に作成する方法

（A）メタクラスを作成し、使用する方法

メタクラスを自動生成する場合も、手作りする場合も、作成されるメタクラスはルールに従う必要があります。メタクラス作成のルールは以下の通りです。

- メタクラスは対応するエンティティクラスと同じパッケージに作成します。
- メタクラス名は対応するエンティティクラス名の末尾に"_"を付けます。
- メタクラスには"Generated"と"TypesafeMetamodel"の2つのアノテーションを付けます。
- 継承関係にあるエンティティ同士のメタクラス同士も継承関係にします。
- クラスXに宣言されるコレクション型でない属性のデータ型をY、属性名をyとした場合、メタモデルでの属性はリスト7の様に宣言します。

リスト7：コレクションでない属性のメタモデルの属性宣言

```
public static volatile Attribute<X, Y> y;
```

- クラスXに宣言されるコレクション型の属性のデータ型をZ、属性名をzとした場合、メタモデルではコレクションクラスに応じてリスト8～リスト12の様に宣言します。

リスト8：Zがjava.util.Collectionである場合のメタモデルの属性宣言

```
…
import javax.persistence.Collection;
…
public static volatile Collection<X, Z> z;
```

リスト9：Zがjava.util.Setである場合のメタモデルの属性宣言

```
…
import javax.persistence.Set;
…
public static volatile Set<X, Z> z;
```

リスト10：Zがjava.util.Listである場合のメタモデルの属性宣言

```
…
import javax.persistence.List;
…
public static volatile List<X, Z> z;
```

リスト11：Zがjava.util.Mapである場合のメタモデルの属性宣言

```
…
import javax.persistence.Map;
…
public static volatile Set<X, K, Z> z;
```

※KはZのキー値の型

エンティティに対するメタクラスを見てみましょう。リスト12がエンティティクラスで、リスト13がメタクラスです。

リスト12：メタクラスを作成する対象となるエンティティクラス

```
package com.example;

※インポートは省略

@Entity public class Order {
   @Id Integer orderId;
   @ManyToOne Customer customer;
   @OneToMany Set<Item> lineitems;
   Address shippingAddress;
   BigDecimal totalCost;

   ※メソッドは省略
}
```

リスト13：Orderエンティティのメタクラス

```
package com.example;

※インポートは省略

@Generated
@TypeSafeMetamodel
public class Order_ {
   public static volatile Attribute<Order, Integer> orderId;
   public static volatile Attribute<Order, Customer> customer;
   public static volatile Set<Order, Item> lineitems;
   public static volatile Attribute<Order, Address> shippingAddress;
   public static volatile Attribute<Order, BigDecimal> totalCost;
}
```

メタクラスには対応するエンティティの属性や関係について記述します。そして、それらの属性がどのデータ型に属するのかを記述しています。

（B）"javax.persistence.metamodel.Metamodel"インターフェースから動的に作成する方法

この方法の場合はまず、EntityManagerから取得したMetamodelを使用してエンティティのMetamodelオブジェクトを作成します。具体的にはリスト14のようなソースになります。

リスト14："javax.persistence.metamodel.Metamodel"インターフェースから動的に作成する例

```
Metamodel metamodel = entityManager.getMetamodel();
Entity<Order> order_ = metamodel.entity(Order.class);

Root<Customer> customer = criteriaQuery.from(Customer.class);
Join<Order, Item> item = customer.join(order_.lineitems);
```

2） 文字列ベースでの問い合わせの作成

文字ベースの問い合わせの作成は、テーブルの結合や列指定で使用するメソッドの引数として文字列を指定する方法になります。Criteria APIの操作自体はMetamodelを使用する場合と同様です。ただし、引数をリテラルで指定するため、

データ型に対する安全性がMetamodel APIを使用した場合と同じレベルで保障はされません。文字列ベースでリスト14と同等の事を記述するとリスト15の様になります。

リスト15：文字ベースでjoinを行う例

```
Root<Customer> customer = criteriaQuery.from(Customer.class);
Join<Order, Item> item = customer.join("orders").join("lineitems");
```

両方Customerエンティティに記述されているlineitems属性にitemが結合されるように記述しています。記述方法の違いからは機能面の差はありません。Metamodel APIの場合、joinのジェネリクスに指定される型が正しいか判断することができます。しかし、文字ベースの場合、joinのジェネリクスに指定される型が正しいか判断しません。この差異を指して、同レベルでの型の安全性が保障されないとJPA 2.0のドキュメントで定義されています。

4. まとめ

JPA 2.0の機能強化はDBへの問い合わせの方法や、エンティティの定義方法、データ型に対する配慮など、広く機能を充実させています。その中でも特に、O/Rマッピングとしての柔軟性を持たせるような側面に力を入れています。JPA 2.0の実装が正式にリリースされれば、Javaにおけるシステム開発の生産性と保守性をより高める事に寄与できると思います。

5. 参考文献

（1）『JSR 317: Java Persistence API, Version 2.0』

http://jcp.org/en/jsr/detail?id=317

（2）『【コラム】Java API、使ってますか? (51) EJBから独立したJava Persistence 2.0』

http://journal.mycom.co.jp/column/jsr/051/index.html

（3）『EclipseLink/Development/JPA - Eclipsepedia』

http://wiki.eclipse.org/EclipseLink/Development/JPA_2.0

（4）『Seasar2とHibernateで学ぶデータベースアクセス JPA入門』

著者：中村　年宏　出版社：毎日コミュニケーションズ

開発部
多田　丈晃