From JPA to S2Persistence

JPAの問題点

- Lazy loading
- 永続コンテキストが複雑
- SQLをDTOにマッピングできない
- JPQLが微妙
- 初期化に時間がかかる
- 学習コストが高い
- パフォーマンスが悪い
- SQLがいつのまにか発行される
- 生産性が悪い

JPAの問題点

- 典型的なメタボリック
  - いろんなフレームワークをつまみ食いして仕様がfat
  - でもいろんなしがらみから痒いところに手が届かない
  - 動作が遅い

JPAの問題点

- 健康体に戻るには
  - 80%のニーズを満たせるくらいまで仕様を絞り込む
    - JPAは100%をめざしてfatになった

S2Persistenceの解決策

- Lazy loading
  - Lazy loadingはしない
  - 必要なデータは一回のSELECT文でJOINで持ってくる。

S2Persistenceの解決策

- 永続コンテキストが複雑
  - Entityはライフサイクルを持たない
  - 常にdetachされた状態
  - detachされた状態でいきなり更新ができる

S2Persistenceの解決策

- SQLをDTOにマッピングできない
  - ResultSetとDTO,Mapを自動マッピング

S2Persistenceの解決策

- JPQLが微妙
  - SQLがそのままつかえる
  - もちろんS2Daoの2way SQL

S2Persistenceの解決策

- 初期化に時間がかかる
  - 初期化はオンデマンド

S2Persistenceの解決策

- 学習コストが高い
  - 仕様は8割の要求を満たす程度にとどめる

S2Persistenceの解決策

- パフォーマンスが悪い
  - ダーティチェックを必要とする永続コンテキストを持たないので高速

S2Persistenceの解決策

- SQLがいつのまにか発行される
  - メモリとデータベースの同期を取るためのflushをしない
  - Lazy loadingをしない

S2Persistenceの解決策

- 生産性が悪い
  - JPQLは動かさないと結果がわからないので生産性が悪い
  - 凝ったSQLをJPQLでやろうとすると大変
  - SQLをDTOにマッピングできないので大変
- 解決策
  - 8割のSQLは自動生成
  - 残り2割はSQLファイルでDTOにマッピング。

S2Persistence(案)

- Daoいらず
  - 高水準なPersistenceManager API
- ネストしたManyToOne, OneToOneサポート
- OneToManyサポート
- RDBMSを生かすPaging処理
- パフォーマンスの向上
  - 完全なHOT deploy対応。
  - データベースのメタデータを使わずデフォルトのルールとアノテーションを使う。
  - PreparedStatementをキャッシュする。

Entity

- マッピング
  - デフォルトのルールに一致している場合は自動マッピング
  - そうでない場合はアノテーションで指定
- publicフィールド

Entity

- テーブル名
  - テーブル名とクラス名が一致していれば省略可能
  - テーブル名から_を除いた部分がクラス名と一致していれば省略可能
  - _を使う使わないは設定で選択
  - そうでない場合はアノテーションで指定

    @Table(“dept”)

Entity

- カラム名
  - カラム名とフィールド名が一致していれば省略可能
  - カラム名から_を除いた部分がフィールド名と一致していれば省略可能
  - _を使う使わないは設定で選択
  - そうでない場合はアノテーションで指定

```
@Column(“hoge”)
public String foo;
```

Entity

- プライマリーキー
  - カラム名が特定のフィールド名であれば省略可能
    - 特定のフィールド名のデフォルトはid
  - そうでない場合はアノテーションで指定

```
@Id
public String key;
```

  - 複合プライマリーキーは@Idを複数指定

```
@Id
public String key;
@Id
public String key2;
```

- プライマリーキーの自動採番
  - 採番テーブル方式
  - insert時にプライマリーキーに値が入っていない場合は自動採番

- バージョンによる楽観的排他制御
  - カラム名が特定のフィールド名であれば省略可能
    - 特定のフィールド名のデフォルトはversion
  - そうでない場合はアノテーションで指定

```
@Version
public Integer versionNo;
```

- ManyToOne
  - FK用のカラム名がManyToOneのフィールド名+_+プライマリーキーのカラム名の場合省略可能。
    - 例えば、public Dept dept;が宣言されている場合に、dept_idのフィールドが存在していれば省略可能。
    - S2PersistenceはLazy loadingを行なわないのでFKのフィールドもEntityに実在する。
  - そうでない場合はアノテーションで指定

```
@JoinColumn(name="dept_id",
  referencedColumnName="id")
public Dept department;
```

- OneToMany
  - アノテーションでペアとなるEntityのフィールドを指定

```
@MappedBy(“dept”)
public List<Emp> emps;
```

- OneToOne(FKを持っているほう)
  - FK用のカラム名がOneToOneのフィールド名+_+プライマリーキーのカラム名の場合省略可能。
    - 例えば、public Address addr;が宣言されている場合に、addr_idのフィールドが存在していれば省略可能。
    - S2PersistenceはLazy loadingを行なわないのでFKのフィールドもEntityに実在する。
  - そうでない場合はアノテーションで指定

```
@JoinColumn(name="addr_id",
  referencedColumnName="id")
public Address address;
```

- OneToOne(FKを持っていないほう)
  - アノテーションでペアとなるEntityのフィールドを指定

```
@MappedBy(“address”)
public Emp emp;
```

- ManyToMany
  - サポートなし
  - 交差エンティティを作ってManyToOneとOneToManyを組み合わせる

PersistenceManager API(案)

- キーで検索

```
Employee emp = pm.find(Employee.class, 1);
```

PersistenceManager API(案)

- =検索

```
Map where = new HashMap();
where.put(“job”, JobType.MANAGER);
where.put(“department$departmentName”,
  “RESEARCH”);
List<Employee> employees =
  pm.from(Employee.class).where(where).asList();
```

PersistenceManager API(案)

- =以外の検索

```
Map where = new HashMap();
where.put(“salary_GE”, 1000);
where.put(“salary_LE”, 3000);
List<Employee> employees =
   pm.from(Employee.class)
   .where(where).asList();
```

PersistenceManager API(案)

- Paging

```
List<Employee> employees =
    pm.from(Employee.class)
    .offset(100).limit(10).asList();
```

PersistenceManager API(案)

- Fetch Join

```
List<Employee> employees =
  pm.from(Employee.class)
  .leftOuterFetchJoin("department").asList();
```

PersistenceManager API(案)

- NamedQuery

  examples/sql/aaa.sql

  select …where …hoge = /*hoge*/1

```
List<EmployeeDto> dtoList =
    pm.getNamedQuery("examples/sql/aaa.sql")
    .setParameter("hoge", 2)
    .asList(EmployeeDto.class);
```

PersistenceManager API(案)

- DynamicQuery

```
StrinbBuilder sb = new StringBuilder();
…
List<EmployeeDto> list =
   pm.createQuery(sb.toString())
   .setParameter(“hoge”, 2)
   .asList(EmployeeDto.class);
```

PersistenceManager API(案)

- Query As Map

```
StrinbBuilder sb = new StringBuilder();
…
List<Map> list =
   pm.createQuery(sb.toString())
   .setParameter(“hoge”, 2)
   .asMap();
```

PersistenceManager API(案)

- INSERT

```
Employee e = new Employee();
e.employeeName = “SCOTT”;
pm.insert(e);
```

PersistenceManager API(案)

- 配列INSERT

```
List<Employee> employees = new
   ArrayList<Employee>();
…
pm.insert(employees);
```

PersistenceManager API(案)

- Bulk INSERT

  examples/sql/bbb.sql

  insert into … select … where hoge = /*hoge*/1

  pm.getNamedQuery(“exmples/sql/bbb.sql”)
  .setParameter(“hoge”, 2)
  .executeUpdate();

PersistenceManager API(案)

- UPDATE

```
Employee e = pm.find(Employee.class, 1);
e.employeeName = “SCOTT”;
pm.update(e);
```

PersistenceManager API(案)

- 配列UPDATE

```
List<Employee> employees =
    pm.from(Employee.class).asList();
...
pm.update(employees);
```

PersistenceManager API(案)

- Bulk UPDATE

  examples/sql/bbb.sql

  update set … where hoge = /*hoge*/1

  pm.getNamedQuery(“exmples/sql/bbb.sql”)
  .setParameter(“hoge”, 2)
  .executeUpdate();

PersistenceManager API(案)

- DELETE

```
Employee e = pm.find(Employee.class, 1);
pm.delete(e);
```

PersistenceManager API(案)

- 配列DELETE

```
List<Employee> employees =
    pm.select(Employee.class).asList();
…
pm.delete(employees);
```

PersistenceManager API(案)

- Bulk DELETE

  examples/sql/bbb.sql

  delete from … where hoge = /*hoge*/1

  ```
  pm.getNamedQuery(“exmples/sql/bbb.sql”)
    .setParameter(“hoge”, 2)
    .executeUpdate();
  ```