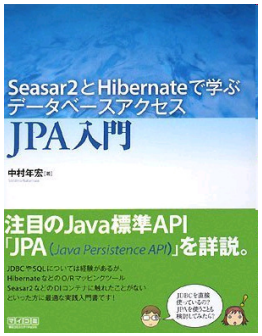
次世代Daoフレームワーク Doma

2009年9月12日
中村年宏
(http://d.hatena.ne.jp/taedium)

自己紹介

- 中村(taedium)
  - http://d.hatena.ne.jp/taedium
- S2Container、S2Dao、Slim3のコミッタ
  - S2JDBC-Gen
    - コードジェネレーション
    - データベースリファクタリング
  - S2Dao
    - 高速化対応（データベースのメタデータを不要に）
  - Slim3-Gen
    - aptとantによるコードジェネレーション
- JPAの入門書執筆
  - Seasar2とHibernateで学ぶJPA入門



アジェンダ

- Domaの概要
- apt
- Domaの戦略
- ロードマップとまとめ

アジェンダ

- Domaの概要
- apt
- Domaの戦略
- ロードマップとまとめ

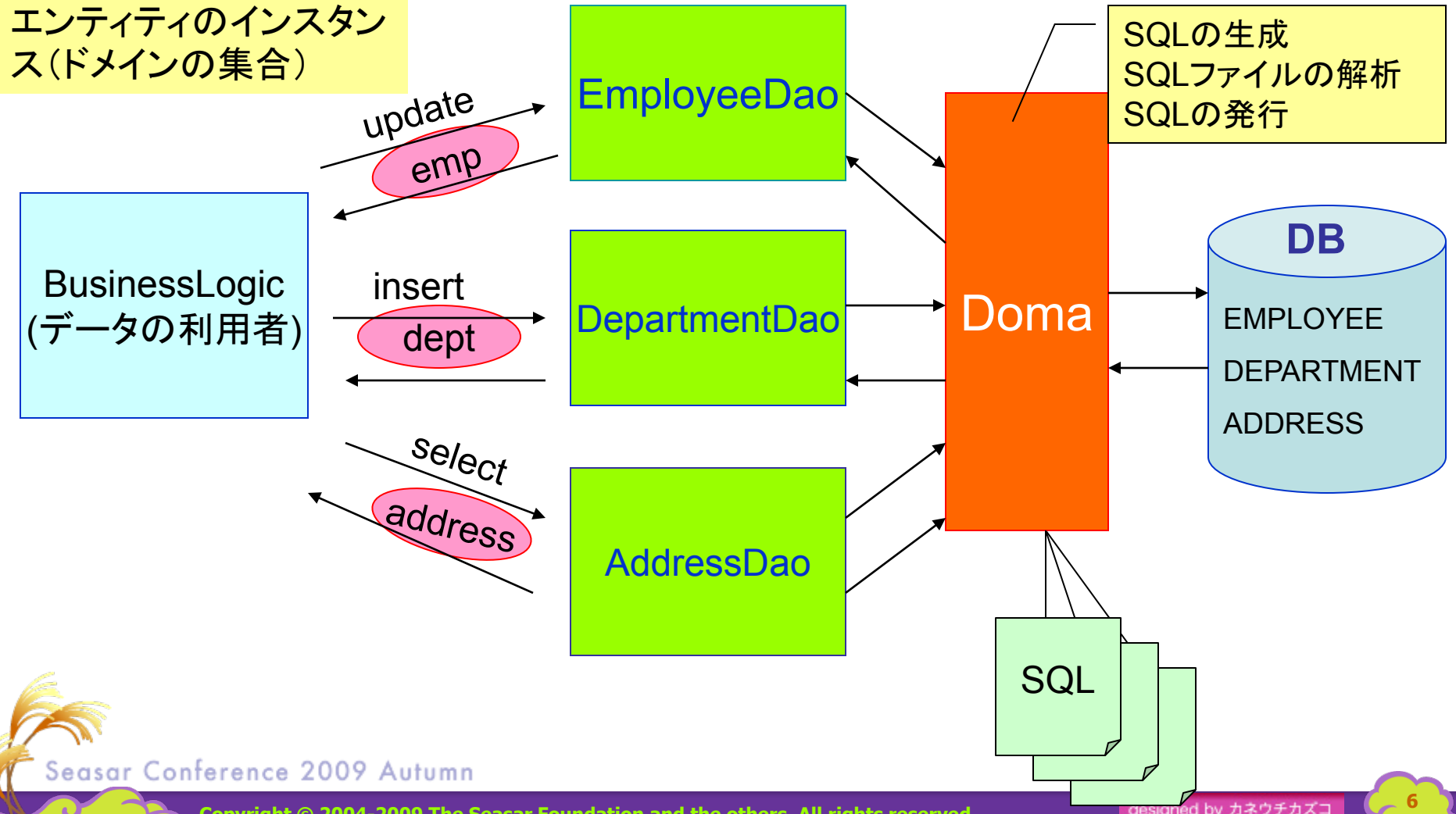
Domaとは？

- Domain Oriented MApping framework
  - ドメイン指向マッピングフレームワーク
  - ドメインとは？
    - 値の定義域（値がとりうる範囲）のこと、またはそれを表すクラス
      - 体重： 0から500の数値、単位はkg → Weightクラス
      - 都市名 :アルファベットの文字列 → CityNameクラス

- Daoフレームワーク
  - S2Dao、S2JDBC、Hibernateのよいところを取り入れたO/Rマッパー
  - データベースの型とドメインをマッピング、ドメインをStringやIntegerなど一般的な値型にマッピング
  - SQLファイルの実行と、SQLの自動生成の2パターンに対応
    - ただし、検索系はSQLファイルの実行のみ

Domaを利用した構成

emp、dept、addressは
エンティティのインスタンス（ドメインの集合）



ドメインの利点 – タイプセーフ

ドメインを使わない場合（値型を使用する場合）

```
public void execute(String orderNo, String customerName,
    String phoneNumber, String location, String description) {
    ...
}
```

ドメインを使う場合

```
public void execute(OrderNo orderNo, CustomerName customerName,
    PhoneNumber phoneNumber, Location location,
    Description description) {
    ...
}
```

ドメインの利点 – 振る舞い

ドメインを使わない場合（値型を使用する場合）

```
String phoneNumber = "03-1234-5678";
int areaCode = PhoneNumberUtil.getAreaCode(phoneNumber);
```

ドメインを使う場合

```
PhoneNumber phoneNumber = new PhoneNumber("03-1234-5678");
int areaCode = phoneNumber.getAreaCode();
```

振る舞いはエンティティにもたせればいいという考え方があるが、エンティティよりもプロパティにもたせた方がコードの重複が少ない。

同じドメインは複数のエンティティにまたがって使われるため。

Employee
id phoneNumber ...

Department
id phoneNumber ...

ドメインの定義

```
public class PhoneNumber extends StringDomain<PhoneNumber> {
    private static final long serialVersionUID = 1L;
    public PhoneNumber() {
    }
    public PhoneNumber(String value) {
        super(value);
    }
    public int getAreaCode() {
        ...
    }
}
```

値型に対応する抽象ドメインクラスを継承して作成。

publicなデフォルトコンストラクタが必要。

任意のメソッドをもてる。

必要に応じて定義域をバリデーション。

ドメインのインタフェース

```
public interface Domain<V, D extends Domain<V, D>> {
    V get();
    void set(V value);
    void setDomain(D other);
    boolean isNull();
    boolean isChanged();
    void setChanged(boolean changed);
    Class<V> getValueClass();
    <R, P, TH extends Throwable> R accept(DomainVisitor<R, P, TH> visitor, P p)
            throws TH, DomaNullPointerException;
}
```

```
PhoneNumber phoneNumber = new PhoneNumber();
phoneNumber.set("03-1234-5678");
String value = phoneNumber.get();
PhoneNumber copy = new PhoneNumber();
copy.setDomain(phoneNumber);
```

利用例

ジェネリクスを活用しタイプセーフを実現。

エンティティの定義

```
@Entity
public interface Employee {

    @Id
    Identity id();

    Name name();

    Salary salary();

    @Version
    VersionNo version();
}
```

- インタフェース！
- @Entityが必須。
- プロパティはフィールドではなくメソッドで定義。getter/setterが不要で簡潔。
- プロパティはもちろんドメインクラス。
- アノテーションの名前はJPAにあわせているが、すべてorg.seasar.domaのパッケージに属する。

Daoの定義

```
@Dao(config = AppConfig.class)
public interface EmployeeDao {
   @Select
   Employee selectById(Identity id);
   @Select
   List<Employee> selectByNames
      (List<Name> names);
   @Insert
   int insert(Employee employee);
   @Update
   int update(Employee employee);
   @Delete
   int delete(Employee employee);
}
```

- やっぱりインタフェース！
- @Daoが必須。
- データソースやデータベースの方言の指定はconfigに指定するクラスの中で設定。
- クエリの種類はすべてアノテーションで明示。
- メソッド名に制約はなし。

サンプルコード – 追加

Javaコード

```
Employee employee = new Employee_();
employee.id().set(99);
employee.name().set("test");
employee.salary().set(300);
EmployeeDao dao = new EmployeeDao_();
dao.insert(employee);
```

エンティティもDaoも実装クラスをnewすればOK。

SQLのログ

```
insert into EMPLOYEE (ID, NAME, SALARY, VERSION) values
(99, 'test', 300, 1)
```

サンプルコード – 検索 & 更新

Javaコード

```
EmployeeDao dao = new EmployeeDao_();

Employee employee = dao.selectById(new Identity(1));

employee.name().set("hoge");

dao.update(employee);
```

Daoは実装クラスを
newすればOK。

selectByIdに対応するSQLファイル

```
select * from employee where id = /*id*/0
```

SQLのログ

```
select * from employee where id = 1

update EMPLOYEE set NAME = 'hoge', VERSION = 1 + 1 where
ID = 1 and VERSION = 1
```

アジェンダ

- Domaの概要
- apt
- Domaの戦略
- ロードマップとまとめ

次世代とは？

- 旧世代
  - 黒魔術に頼りすぎてしまった時代
    - 限定的に使用すれば少ないコードで効力を発揮するが、一歩誤ると生産性や保守性を損なう技術。アプリケーションで管理されたソースコードをみただけでは挙動がわからない。わかりにくいエラーメッセージが出ることも。
      - AOP（バイトコードエンハンス）
      - 命名規約
- 次世代
  - 脱黒魔術
    - ソースコードのとおりに動く
      - バイトコードエンハンスや命名規約を使った場合と同等のソースコードを生成。
      - デバッグしやすい
    - 規約は事前にチェックする
      - コンパイルエラーとして扱えば単なる命名規約ではない
    - エラーが起きてもエラーメッセージがわかりやすい

脱黒魔術のためのツール：apt

- Pluggable Annotation Processing API
  - Java6から導入されたアノテーション処理のためのAPI
    - javax.annotation.processing
    - javax.lang.model
    - javax.lang.model.element
    - javax.lang.model.type
    - javax.lang.model.util
  - コンパイル時に次のことが可能
    - ソースコードの生成
    - ソースコードの検証
    - リソースの参照
    - リスースの生成

aptの魅力

- コンパイル時に自動で実行される
  - コンパイルするだけでよい
    - コンパイル以外で別の処理（antタスクやプラグインなどで）を実行したり、実行時にオプション指定（javaの-javaagent）をする必要がない
  - 開発環境（Eclipse）でもjavacでも動作する
- エラーを早期に検出できる
  - 実行時ではなくコンパイル時に検出
  - 動かさなくても間違いに気づけるため開発効率が高い
- 実行時の負荷を軽減できる
  - 多くのフレームワークは実行時にクラスのメタ情報を解析しキャッシュするが、aptを使えば、メタ情報をコンパイル時にあらかじめソースコードにして出力。解析やキャッシュの必要性がない。

DomaにおけるAptの利用

- エンティティの検証・実装クラスの生成
- Daoの検証・実装クラスの生成
- Dto（JavaBeans）の生成
- SQLファイルの検証
  - バインド変数
  - 条件式

DomaにおけるAptの利用 – デモ

- 設定
- バリデーション
- コードジェネレーション

アジェンダ

- Domaの概要
- apt
- **Domaの戦略**
- ロードマップとまとめ

Domaの戦略

- 依存ライブラリ
- Dao
- SQLファイル
- キャッシュ
- プラガブルな構成
- JavaBeans（DTO）への自動変換

依存ライブラリ

- 依存ライブラリはなし
  - Seasar2にも依存していない
    - DIも必須でない
    - newすればいいだけ
  - Domaのjar（doma-x.x.x.jar）が1つがあればいい
  - 他のフレームワークと特定のライブラリが競合するといった問題が発生しない

Dao

- 構成のわかりやすさ
  - メソッド呼び出しとSQLの発行が1対1で対応しているのがわかりやすい。
- メンテナンスのしやすさ
  - CRUDがどこで行われているのか、すぐに調べられるのが重要。
- テストのしやすさ
  - Mockが作りやすい。Mockライブラリとの相性。

SQLファイル

- 検索系SQL
  - 意図的に自動生成機能はなし。すべてSQLファイルを利用。動的なSQLは、SQLコメントによる条件式で生成。Javaコードに埋め込むのはメンテナンスを難しくするので認めない。
- 更新系SQL
  - 基本自動生成。SQLファイルの利用も可。

SQLファイル

```
select * from employee where /*%if id != null*/id = /*id*/99/*%end*/
```

実行されるSQL　id != null　が成立する場合

```
select * from employee where id = ?
```

実行されるSQL　id != null　が成立しない場合

```
select * from employee
```

キャッシュ

- メモリを圧迫する可能性あり
- キャッシュはできるだけ避けるべき
  - Domaではaptでコンパイル時にメタ情報のクラスを生成するため、キャッシュの必要性がない
  - デフォルトでキャッシュするのはSQLの解析結果のみ（キャッシュの実装は差し替え可能）

プラガブルな構成

- 差し替えられるところを明確化している
  - ネーミング規約
  - データベースの方言
  - SQLファイルのキャッシュ戦略
  - JavaBeansへのアクセス方法
  - ドメインとデータベースの型のマッピング
  - SQLのログ出力とフォーマット方法

JavaBeans（DTO）への変換

JavaBeansへの変換は容易（他のフレームワークとの連携もOK）

```
@Entity
public interface Employee {
    @Id
    Identity id();
    Name name();
    Salary salary();
    @Version
    VersionNo versionNo();
}
```

自動生成

```
public class EmployeeDto
        implements java.io.Serializable {
    private static final long
         serialVersionUID = 1L;
    private java.lang.Integer id;
    private java.lang.String name;
    private java.lang.Long salary;
    private java.lang.Integer versionNo;

    // getter/setter
```

相互変換

```
Employee employee = dao.selectById(new Identity(1));
EmployeeDto dto = new EmployeeDto();
CopyUtil.copy(employee, dto);
```

アジェンダ

- Domaの概要
- apt
- Domaの戦略
- ロードマップとまとめ

ロードマップ

- Eclipseのプラグインの作成
  - DaoとSQLファイルの相互遷移
  - SQLの保存と同時にDaoを再ビルド
    - aptをキック
  - 「Create Domain Class」ダイアログ
- データベースからエンティティとDaoの自動生成
- テストのサポート
  - S2Unitを進化させたもの？
- aptを使ったSQLファイルの検証をさらに賢く
  - 型チェック
    - 異なる型同士の比較をエラーとする。
      - /*%if name == 1*/　という式でnameがStringDomainの場合

まとめ

- 値型よりもリッチなクラスにメリットがある
- aptを使って脱黒魔術
  - コードジェネレーションやバリデーションがコンパイル時に実行できる
- Domaは、aptを活用したDaoフレームワーク
  - ドメインというリッチなクラスとデータベースの型をマッピング
  - プラガブルな構成なため、カスタマイズが容易
  - 必要なところではJavaBeansに変換し、他のフレームワークとの互換性も確保

- サイト
  - http://doma.sandbox.seasar.org/
- よくある質問
  - http://doma.sandbox.seasar.org/faq.html
- ソースリポジトリ
  - http://doma.sandbox.seasar.org/source-repository.html

ご清聴ありがとうございました。