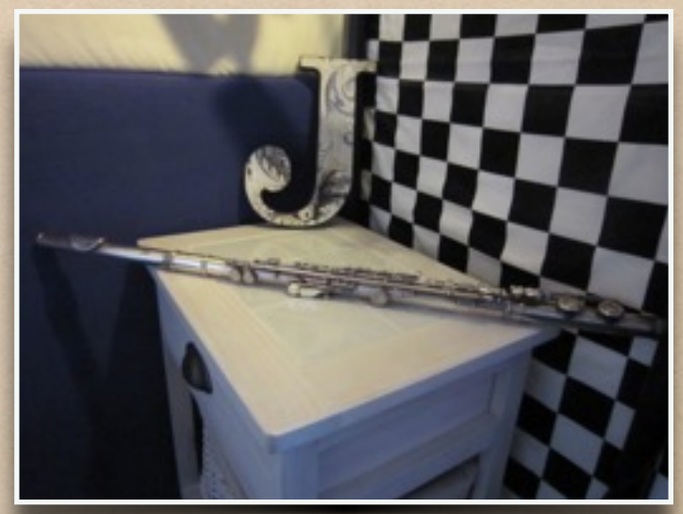
by



DBをリファクタリングしよう、DBとアプリの架け橋 DBFlute

久保雅彦(jflute)

自己紹介

わたしはだれ？

久保　雅彦 (jflute)
　オープンソースプログラマー

- DBFluteの作者
  - jfluteの日記　(http://d.hatena.ne.jp/jflute/)
  - Twitter: @jflute / facebook: dbflute

自己紹介

住んでるところ

Java, JDBC, フレームワーク,
DB設計, プログラマー教育

オープンソース

DBFluteとは？

- O/Rマッパー
- DB管理支援ツール

DBFluteの特徴は？

DB変更に強い

DBFluteのターゲット

- BtoCなどのサービス開発
  （事業会社）
  - リーン・スタートアップ
  - インクリメンタル開発
- DB設計と実装の同時開発

※ビジネスのための泥臭いツール

DBFluteとは？

・・・

ふーん

DBFluteは変えたい

DBサイドとアプリサイドの

ギャップ

ギャップって？

DBサイド：DB設計、インフラ
アプリサイド：アプリ開発

互いが互いに責め合う(T∀T)

ギャップその１

DBサイドとアプリサイドの ギャップ　その１

DB設計の意図が アプリ側に伝わらない

アプリあるある

- どこにIndex貼ってあるのかわからない
- NotNull制約の外れてる理由がわからない
- どことどこをjoinすればいいのか…
- この区分値、何が入るの？

アプリあるある

- 40.テーブル定義書フォルダに.xls発見、開く
- ウィルスチェック、開くまで待つ待つ
- 大量のシートで途方にくれる
- “エクセル シート移動 ショートカット” で検索
- 見たいテーブル見つけたー、…先輩通りすがる
- 「それ古いから見ない方がいいよ」

ギャップその１：DB設計の意図が伝わらない

というのから

卒業

DBFluteはDBを伝える！

- テーブル定義を自動生成(SchemaHTML)
  - メンテナンス不要（DBコメント重視）
  - 気楽に開けるHTML形式
- DB定義をJavaDocコメントに
- DBコメントをJavaDocコメントに
  - xlsは精神的距離が遠いが、JavaDocは近い

ギャップその１：DB設計の意図が伝わらない

SchemaHTML（テーブル一覧）

No.	Alias	Name	Type	ForeignTable	ReferrerTable	TableComment
1	会員	MEMBER	TABLE	MEMBER_STATUS, *MEMBER_ADDRESS(AsValid)*, *MEMBER_LOGIN(AsLatest)*	MEMBER_ADDRESS, MEMBER_FOLLOWING, MEMBER_FOLLOWING, MEMBER_LOGIN, MEMBER_SECURITY, MEMBER_SERVICE, MEMBER_WITHDRAWAL, PURCHASE	会員のプロフィールやアカウントなどの基本情報を保持する 基本的に物理削除はなく、退会したらステータスが退会会員 ライフサイクルやカテゴリの違う会員情報は、one-to-one
2	会員住所情報	MEMBER_ADDRESS	TABLE	MEMBER, REGION		会員の住所に関する情報で、同時に有効期間ごとに履歴管理 会員を基点に考えた場合、構造的には one-to-many だが、 なる。このような構造を「業務的one-to-one」と呼ぶ！ 有効期間は隙間なく埋められるが、ここでは住所情報のない な) "1：0..1" である。
3	会員フォローイング	MEMBER_FOLLOWING	TABLE	MEMBER, MEMBER		とある会員が他の会員をフォローできる。すると、フォロー
4	会員ログイン	MEMBER_LOGIN	TABLE	MEMBER_STATUS, MEMBER		ログインするたびに登録されるログイン履歴。 登録されたら更新されるも削除されることもない。さらには しているので、(紙面の都合上もあって)ここでは共通カラム
5	会員セキュリティ情報	MEMBER_SECURITY	TABLE	MEMBER		会員とは one-to-one で、会員一人につき必ず一つのセキ
6	会員サービス	MEMBER_SERVICE	TABLE	MEMBER, SERVICE_RANK		会員のサービス情報（ポイントサービスなど）。 テストケースのために、あえて統一性を崩してユニーク制約 る。
7	会員ステータス	MEMBER_STATUS	TABLE		MEMBER, MEMBER_LOGIN	会員のステータスを示す固定的なマスタテーブル。いわゆる 業務運用上で増えることはなく、増減するときはプログラム られる。 こういった固定的なマスタテーブルには、更新日時などの共 そういった情報を管理する必要性が低いという理由に加え、 が埋め尽くされてしまい見づらくなるというところで割り切 ることもある。

ギャップその１：DB設計の意図が伝わらない

SchemaHTML（テーブル詳細）

(会員)MEMBER (outsideSql=9)

会員のプロフィールやアカウントなどの基本情報を保持する。
基本的に物理削除はなく、退会したらステータスが退会会員になる。
ライフサイクルやカテゴリの違う会員情報は、one-to-oneなどの関連テーブルにて。

No.	PK	ID	UQ	IX	Not Null	Alias	Name	Type	Size	ForeignTable	ReferrerTable	Classification	Column
1	o	o			*	会員ID	MEMBER_ID	INTEGER	10	*MEMBER_ADDRESS(AsValid)*, *MEMBER_LOGIN(AsLatest)*	MEMBER_ADDRESS, MEMBER_FOLLOWING, MEMBER_FOLLOWING, MEMBER_LOGIN, MEMBER_SECURITY, MEMBER_SERVICE, MEMBER_WITHDRAWAL, PURCHASE		連番として自動採番される。会DBMS次第。
2				o	*	会員名称	MEMBER_NAME	VARCHAR	200				会員のフルネームの名称。 苗字と名前を分けて管理するこまとめ。
3			o		*	会員アカウント	MEMBER_ACCOUNT	VARCHAR	50				ログインIDとして利用する。 昨今メールアドレスをログイン見かけないかも!?
4				o	*	会員ステータスコード	MEMBER_STATUS_CODE	CHAR	3	MEMBER_STATUS		MemberStatus	会員ステータスを参照するコーステータスが変わるたびに、こ
5				o		正式会員日時	FORMALIZED_DATETIME	TIMESTAMP	23, 10				会員が正式に確定した(正式会員一度確定したらもう二度と更新
6						生年月日	BIRTHDATE	DATE	8				必須項目ではないので、このデ
7					*	登録日時	REGISTER_DATETIME	TIMESTAMP	23, 10				レコードが登録された日時。 会員が登録された日時とほぼ等を兼務させるのはあまり推奨さルには会員登録日時がない… 仕様はどのテーブルでも同じなーブルでしか書かない。
													レコードを登録したユーザ。

DBサイドとアプリサイドの
ギャップ　その2

アプリ側の都合に関係なく

DB変更される

アプリあるある

- 気付いたらDB変わってて落ちてる
- 影響範囲ありすぎでデグレまくる
- ローカル開発用DBがめっちゃ古い

DBFluteはDB変更につおい！

- タイプセーフAPI (ConditionBean) で
  DB変更の影響範囲検知

- 2WaySQLの外出しSQLを
  一括実行で検知

- 最新DB構造の横展開を自動化
  (ReplaceSchema)
  - バッチ一発で、サクッとDB作り直し
  - テストデータの一元管理

ConditionBean

```
ListResultBean<Member> memberList = memberBhv.selectList(cb -> {
    cb.setupSelect_MemberStatus();
    cb.query().setMemberName_LikeSearch("S", op -> op.likePrefix());
    cb.query().existsPurchase(purCB -> {
        purCB.query().setPurchasePrice_GreaterEqual(200);
    });
    cb.query().addOrderBy_Birthdate_Desc();
});
memberList.forEach(member -> {
    log(member.getMemberName(), member.getBirthdate());
});
```

FKラインを辿る旅
DBが綺麗であればあるほど
実装しやすくなる というインセンティブ

ギャップその２：DB変更される

一括テスト実行できる
2WaySQL (OutsideSql)

```
select mb.MEMBER_ID
     , mb.MEMBER_NAME
     , mb.BIRTHDATE
     , stat.MEMBER_STATUS_NAME
  from MEMBER mb
    left outer join MEMBER_STATUS stat
      on mb.MEMBER_STATUS_CODE = stat.MEMBER_STATUS_CODE
/*BEGIN*/
where
  /*IF pmb.memberId != null*/
  mb.MEMBER_ID = /*pmb.memberId*/3
  /*END*/
  /*IF pmb.memberName != null*/
  and mb.MEMBER_NAME like /*pmb.memberName*/'S%' -- // keyword for pref
  /*END*/
  /*IF pmb.birthdate != null*/
  and mb.BIRTHDATE = /*pmb.birthdate*/'1966-09-15' -- // used as equal
  /*END*/
/*END*/
order by mb.BIRTHDATE desc, mb.MEMBER_ID asc
```

DBAの人も

DB変更の影響規模を探りやすい

ローカルで試しに変えてみて
バッチを叩けばOK

ギャップその3

DBサイドとアプリサイドの ギャップ　その3

というか…

DB変更の内容が アプリ側に伝わらない

アプリあるある

- 何が変わったのかがわからない
- 直すべきところわからず放置
- というかDBAも細かく伝えるの面倒

DBFluteはDB変更を伝える！

- HistoryHTMLで
  履歴ドキュメントを自動生成
  （自然と作られていくところがポイント）

- もちろん、
  JavaDocコメントや
  SchemaHTMLも大活躍

ギャップその３：DB変更が伝わらない

HistoryHTML

Diff Date: 2010/06/12 21:03:57

Change Table

- MEMBER

  Delete Column

  - STEP_UP_FLG

- MEMBER_ALERT

  Add Column

  - MEMBER_ID

  Change Column

  - MEMBER_ALERT_NAME

Size	300 -> 200
Not Null	true -> false

  Delete Column

  - MEMBER_ALERT_ID

プロシージャだって

Change Procedure

- exampledb.SP_IN_OUT_PARAMETER

SourceLine

5 -> 4

SourceSize

100 -> 70

SourceHash

3069f26a -> 27652c66

ギャップその4

DBサイドとアプリサイドの ギャップ　その4

アプリが ぐるぐる回す

(ぶーんぶーんばばーん!)

チューニングあるある

- 一回のリクエストで300回のSQL
- 一個一個は速いので、頼まれてもDB側の調整ではどうにもならない（jflute経験済み）
- 夜間バッチだと…ひぃぃぃー

フレームワークあるある

getMemberStatus() された時に検索

```
List<Member> memberList = …(最初の検索)
for (Member member : memberList) {
    member.getMemberStatus() // ひぃぃぃー
}
```

getされた時に関連テーブルのデータを検索
いわゆる LazyLoad 機能

DBFluteは明文主義

- データ取得したい関連テーブル明示
- getメソッドでLazyLoadしない
- SQLの発行回数を数えるための拡張ポイント(CallbackContext)

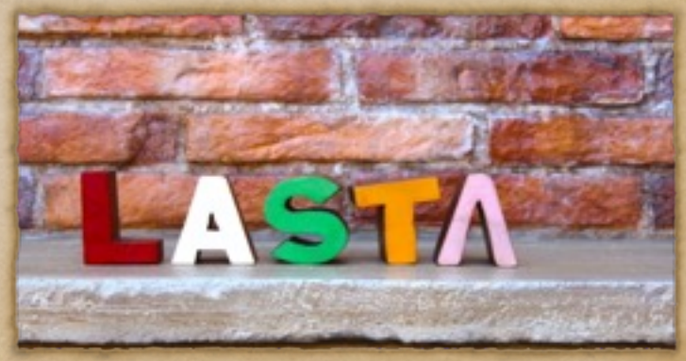
SQLの発行回数をログに

…(デバッグログの中)

```
[request] lastaflute.dbflute.SQL_COUNT=
{total=2, selectCB=2, entityUpdate=0
, queryUpdate=0, outsideSql=0, procedure=0}
…
```

※DBFluteと連携したWebフレームワーク

「LastaFlute」にて



発行しすぎ警告ログ

…(警告ログの中)

```
*Too many SQL executions:
{total=81, selectCB=3, entityUpdate=78,
queryUpdate=0, outsideSql=0, procedure=0}
  in ProductListAction.search()
…
```

※DBFluteと連携したWebフレームワーク

「LastaFlute」にて

こういう視点も

DBの問題はDBだけでは
解決できない

↓

Webフレームワーク
との連携も大切

ちなみに

SQLをべたっと書けばぐるぐるが消える
というのはまちがい
↓↓↓
プログラム上でSQLを組み立てづらい環境だからこそ面倒になってぐるぐるする
↓↓↓
LazyLoadしなくて、
SQLを組み立てやすいツール
の方が防げきやすい
(無論100%防げるわけではないが比較的)

ギャップその5

DBサイドとアプリサイドの ギャップ　その5

本番と結合と開発で、

スキーマ構造が違う！

アプリ，DBAあるある

落ちた
↓
調べる
↓
本番DB違う

Alter書いた
↓
Index書き忘れた
↓
実行し忘れた

スタートアップあるある

まず、ズレる

(毎週のようにDB変更しますから…)

DBFluteは差分大好き

- HistoryHTMLでDB変更の歴史
- AlterCheckで
  Alter文の整合性チェック
- SchemaSyncCheckで
  二つのDBの差分チェック

DB差分御三家を呼ぶ

AlterCheckの方程式

前のDB ＋ 差分DDL

＝ 最新のフルDDL

ギャップその5：本番とスキーマちっがーう

AlterCheckの流れ

1. 前のDB（フルDDL）を保存 by DBFlute
2. 普通にDB変更 by ERD
3. Alter文を書く by 人類
4. フルDDLを吐き出して… by ERD
5. 方程式と合わせる by DBFlute

AlterCheckの結果

ダメ
ぜったら…

Diff Date: 2010/06/12 21:03:57

Change Table

- MEMBER
  - **Delete Column**
    - STEP_UP_FLG
- MEMBER_ALERT
  - **Add Column**
    - MEMBER_ID
  - **Change Column**
    - MEMBER_ALERT_NAME

Size	300 -> 200
Not Null	true -> false

  - **Delete Column**
    - MEMBER_ALERT_ID

差分がなくなるまで差し戻し

DBサイドとアプリサイドの ギャップ　その6

アプリ屋さんよぅ…

パフォーマンス考慮お願い

DBFluteはSQLを大切に１

発行されるSQLは徹底フォーマット

(プログラマーがすぐにログ確認。実行はバインド変数)

```
select dfloc.MEMBER_ID as MEMBER_ID, dfloc.MEMBER_NAME as
     , dfrel_0.MEMBER_STATUS_CODE as MEMBER_STATUS_CODE_0,
  from MEMBER dfloc
    inner join MEMBER_STATUS dfrel_0 on dfloc.MEMBER_STATU
 where dfloc.MEMBER_NAME like 'S%' escape '|'
   and exists (select sub1loc.MEMBER_ID
                 from PURCHASE sub1loc
                where sub1loc.MEMBER_ID = dfloc.MEMBER_ID
                  and sub1loc.PURCHASE_PRICE >= 200
       )
 order by dfloc.BIRTHDATE desc
```

DBFluteはSQLを大切に２

発行されたSQLの実行時間をログに
(プログラマーに普段から意識してもらう)

```
.MEMBER_ID = dfloc.MEMBER_ID
.PURCHASE_PRICE >= 200



]-DEBUG (XLog#log():43) - ============/ [00m00s007ms (13)
```

※結果の件数もね

DBFluteはSQLを大切に３

どこから呼ばれたSQL？をログに
（プログラマーをSQLに振り向かせる）

```
/=========================================================================
                                                    ProductBhv.selectList()
                                                    ======================/
ProductListAction.index():54 -> ProductListAction.selectProductPage():69 -> ...
```

LastaFluteと連携でSQLにアプリクラス埋め込みも
（DB側でSQLを抽出しても、どこのSQLかすぐにわかる）

```
where dfloc.PRODUCT_STATUS_CODE = 'ONS'
order by dfloc.PRODUCT_NAME asc, dfloc.PRODUCT_ID asc
-- org.docksidestage.app.web.product.ProductListAction#index(): (HBR)
```

DBFluteはSQLを大切に４

EntityのSetter呼び出し情報でupdate文を構築
(無駄な事前検索や問答無用全カラム更新しない)

```
Member mb = new Member();
mb.setMemberId(3);
mb.setMemberName( “jflute” );
memberBhv.update(mb);
          ↓ ↓ ↓
update MEMBER
    set MEMBER_NAME =  ‘jflute’
 where MEMBER_ID = 3
```

まとめ

DBをリファクタリングするために

RDBを隠蔽するのではなく、

RDBを強く意識させる

O/Rマッパー
DBFlute

まとめ

DBサイドとアプリサイド

どっちもWinWinになってこそ、

お客様も最高のシステムに
出会えるはず

まとめ

架け橋

そのためのツール、

選んでみませんか？