リスク先制攻略で
失敗しないHibernateプロジェクト

JavaFesta in **札幌**2007

2007/11/02

株式会社 ビルドシステム
業務システム開発部
鈴木　裕太

ビルドシステム

ゴール

- **採用を検討している方々に、**
  Hibernateのリスクについて解説する事で採用の可否判定に役立ててもらう。

- **既に採用している方々に、**
  「これが出来ない」を伝える事で早期のリスク対策に役立ててもらう。

- **過去に断念or辛い思いをした方々に、**
  正のリスクも伝える事でリベンジを考えてもらう。

アジェンダ

- 一般的な事
- 永続化ライフサイクル
- トランザクション
- フェッチ
- クエリ
- キャッシュ
- その他
- まとめ

一般的な事　ー　正のリスク

- ＯＳＳ
- 高機能
- コンテナレス
- 実績が多そう
- それなりに情報もある

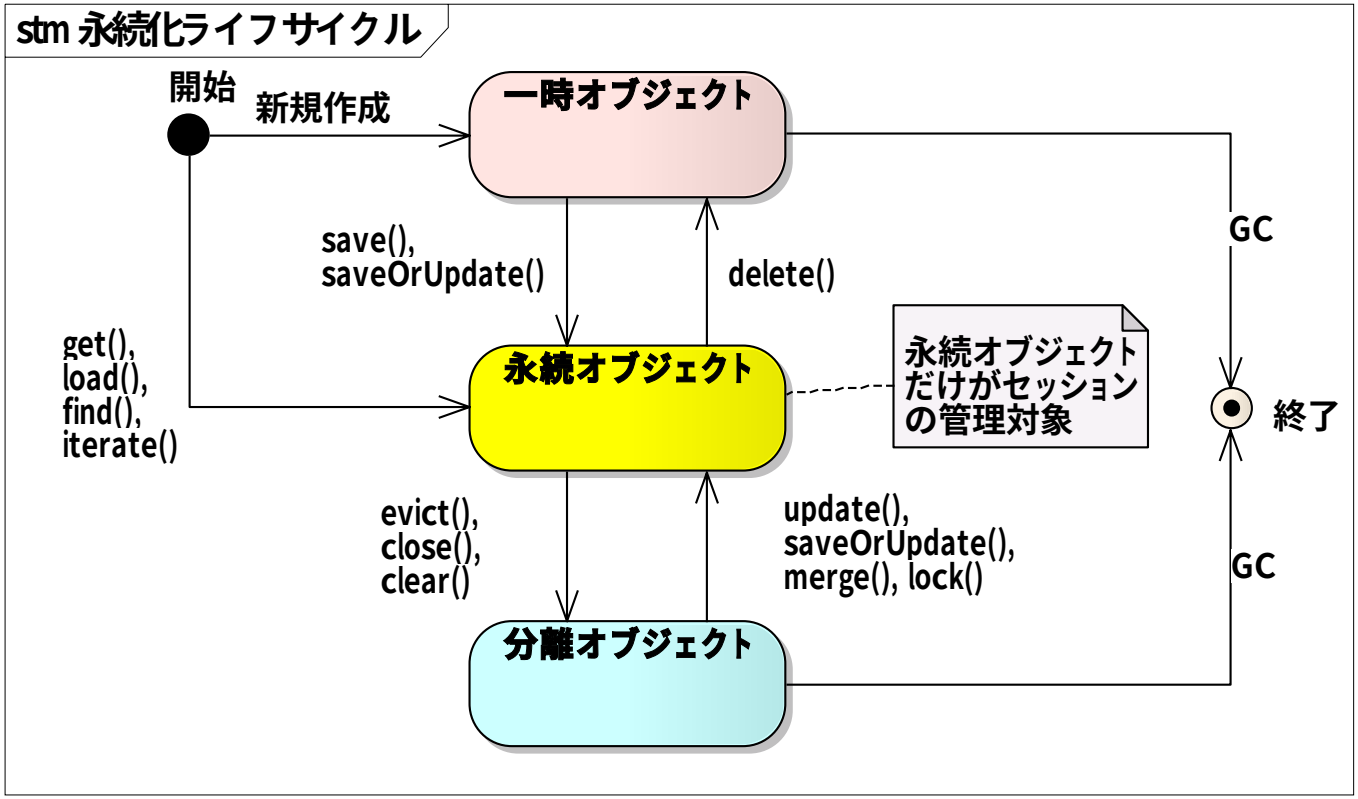
一般的な事　ー　負のリスク

- そもそも難しい
  - 思ったように動かない
  - 思ったより生産性が上がらない
  - チューニングが難しい
- それなりに情報はあるが、本当に欲しい情報は少なかったりする
- 抵抗勢力がいる

アジェンダ

- 一般的な事
- **永続化ライフサイクル**
- トランザクション
- フェッチ
- クエリ
- キャッシュ
- その他
- まとめ

永続化ライフサイクル　－　概要（その1）



ビルドシステム

永続化ライフサイクル　ー　概要(その2)

- 永続化ライフサイクルはTorqueやS2-DAOには無い概念。
- 永続オブジェクトであれば、
  - 関連する親や子の取得はgetterのみで出来る。
  - setterで設定した値は自動的にDBに反映される。
  - これがHibernateの大きな魅力。
- 反面、エンティテイが永続オブジェクトである限り、アクセスする際にはDBと同期している事を意識する必要がある。

永続化ライフサイクル　ー　リスク(その1)

- 永続オブジェクトの更新に関するリスク
  - 永続オブジェクトはsetterで値を設定するだけでDBに反映されてしまう。
    - 本来メリットなので「反映してくれる」なんですが・・・
  - session.update()しなければDBに反映されないと勘違いする人が多い。
    - session.update()は分離オブジェクトを永続オブジェクトにするメソッド。
    - 永続オブジェクトに対してsession.update()を使用するのはそもそも無意味。
  - オンメモリ特有のステータス等を使用する場合には注意が必要。
    - 永続オブジェクトを使用していると処理ステータス等でオンメモリ特有のはずの値がDBに反映されてしまうケースがある。
  - 対応策
    - 永続化ライフサイクルへの理解を徹底する。
    - オンメモリ特有の値はエンティティに設定しない。
    - 必要に応じて分離オブジェクトを利用する。
    - 他の処理系の実装を再利用する場合には必ず精査する。

永続化ライフサイクル　ー　リスク(その2)

- 分離オブジェクトを永続オブジェクトにする際のリスク
  - update()/saveOrUpdate()で分離オブジェクトを永続オブジェクトにすると全項目UpdateのSQLで更新される。
    - dynamic-update=trueでも全項目が更新されるので注意が必要。
  - 対応策
    - 出来るだけmerge()を使用する。
  - 詳しくは次ページの「表．マッピング定義と部分UPDATE可否の関係」を参照。

永続化ライフサイクル　－　リスク(その2)

- 分離オブジェクトを永続オブジェクトにする際のリスク

表．マッピング定義と部分UPDATE可否の関係

条件		更新方法		
dynamic-update	select-before-update	永続オブジェクトの変更	分離オブジェクトのupdate(※2)	分離オブジェクトのmerge(※2)
False	False	△	×	△(※1)
False	True	△	△(※1)	△(※1)
True	False	○	×	○(※1)
True	True	○	○(※1)	○(※1)

凡例	
×	全く変更が無くても常に主キー以外の全項目が更新される
△	どれかの項目が変更されていれば主キー以外の全項目が更新される
○	変更した項目と並行性制御用項目のみが更新される

※1　キャッシュミス時はselectが行われる

※2　分離オブジェクトを永続オブジェクトにする際にはcascade指定に従って関連する子も同様の処理が行われる(LAZYロードで未ロード状態の場合は対象外)

永続化ライフサイクル　ー　リスク(その3)

- session.merge()の追加による正のリスク
  - 分離オブジェクトを永続オブジェクト化する際の煩雑さはHibernate 3.0でmerge()が導入されてかなり改善された。
  - update()ではムダな更新が多くなるケースが目立つがmerge()なら最適化される。
  - update()ではセッション上にドッペルゲンガーがいると例外が発生してしまうがmerge()ならいいかんじに「マージ」してくれる。

アジェンダ

- 一般的な事
- 永続化ライフサイクル
- トランザクション
- フェッチ
- クエリ
- キャッシュ
- その他
- まとめ

トランザクション　ー　概要

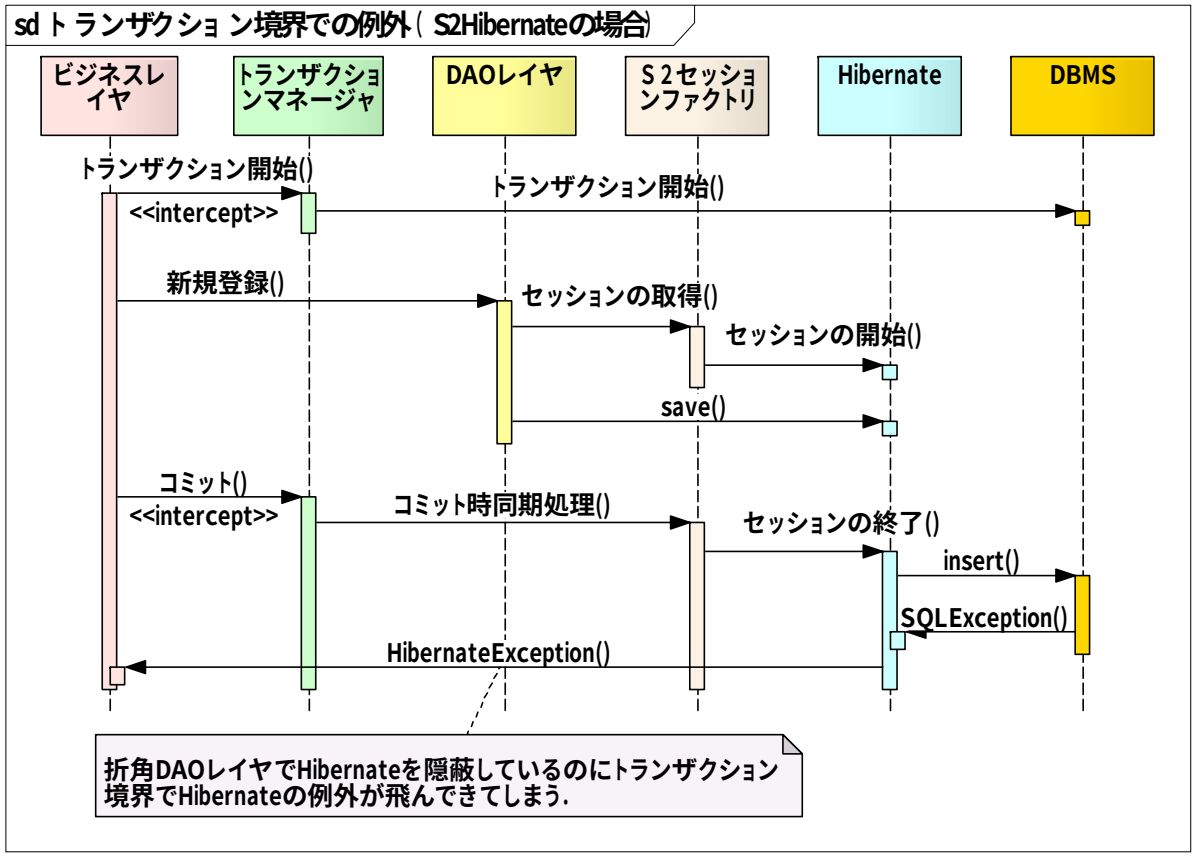
- DIコンテナ上でHibernateを使う場合にはセッションやトランザクションのライフサイクルはインターセプタによって処理するのが普通。
  - ここでのセッションとはHibernateのorg.hibernate.Session
- それはそれで楽ではあるがHibernate特有のリスクも存在する。

トランザクション　ー　リスク

- **書込みSQL発行タイミングに関するリスク**
  - Hibernate**が書込みSQLを発行するタイミングは**
    - **クエリで検索を行う時**
    - org.hibernate.Session.flush()**を呼び出した時**
    - org.hibernate.Transaction.commit()**を呼び出した時**
  - **そのため一意制約例外や並行性制御例外がトランザクション境界で発生してしまうケースがある。**

トランザクション　ー　リスク

- **実際の書込みSQL発行タイミングに関するリスク**



ビルドシステム

トランザクション　ー　リスク

- **実際の書込みSQL発行タイミングに関するリスク**
  - DB**アクセス(つまり**Hibernate**自体)を**DAO**レイヤに隠蔽しているのにトランザクション境界で**Hibernate**の例外が発生しては困ってしまう。**
  - **対応策**
    - **このような例外が発生する場合は**DAO**内で**Session.flush()**する。**
    - **トランザクション境界にインターセプタを追加して例外をラップする。**

アジェンダ

- 一般的な事
- 永続化ライフサイクル
- トランザクション
- フェッチ
- クエリ
- キャッシュ
- その他
- まとめ

フェッチ　ー　概要（その1）

- 関連のフェッチについて二つの直交する概念がある。
- 一つ目はどんな「方法」でフェッチするかという概念。
  - JOINフェッチ
    - OUTER JOINを使用して関連するインスタンスやコレクションを1回のSELECT でフェッチする。
  - SELECTフェッチ
    - 個別のSELECTで関連するエンティティやコレクションをフェッチする。キャッシュ利用の対象となるがヒットしない場合にはN＋1になってしまう。
  - BATCHフェッチ
    - IN句の列挙「where id in (?, ?, ? ･･･)」でエンティティのインスタンスやコレクションを一気にフェッチする。‘?’の数がバッチサイズ。
  - SUBSELECTフェッチ
    - IN句のサブクエリ「where id in (select id from hoge where ･･･)」でコレクションを一気にフェッチする. (Hibernate 3.0にて追加)

フェッチ　ー　概要(その2)

- 二つ目はどんなタイミングでフェッチするかという概念。
  - EAGERフェッチ
    - 当該オブジェクトがロードされたときに関連やコレクションを即時にフェッチする。
  - LAZYコレクションフェッチ
    - アプリケーションがコレクションに対して操作を行ったときにコレクション全体をフェッチする。
  - EXTRA-LAZYコレクションフェッチ
    - コレクションの要素毎に必要なときにフェッチする。(Hibernate 3.1にて追加)
  - PROXYフェッチ
    - 対1関連で関連オブジェクトの主キーのgetter以外のメソッドが呼び出された時にフェッチする。
  - NO-PROXYフェッチ
    - 対1関連で関連オブジェクトのインスタンス変数にアクセスした時にフェッチする。ビルド時のバイトコード組み込みが必要。(Hibernate 3.1にて追加)
  - LAZYプロパティフェッチ
    - プロパティにアクセスしたときにそのプロパティをフェッチする。ビルド時のバイトコード組み込みが必要。(Hibernate 3.0にて追加)

フェッチ　ー　リスク(その1)

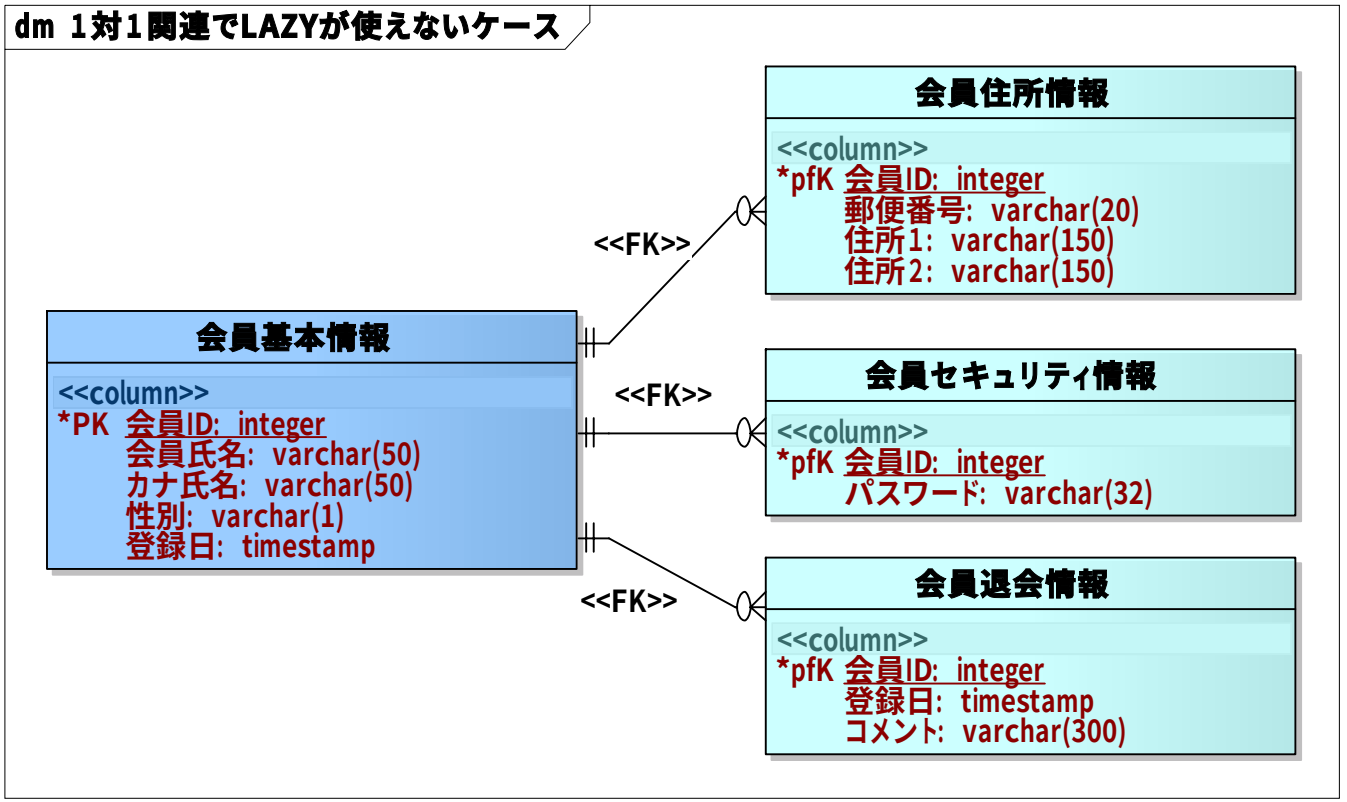
- HQLがマッピングファイル中のJOINフェッチ指定(fetch="join")を無視するリスク
  - HQLでは明示的にJOINフェッチを指定しないとSELECTフェッチになってしまう。
    - 確かに、コレクションをJOINフェッチする場合には大抵はResultTransformerでDistinctしないといけないので無視してくれた方が安全かもしれない。
    - でも対1関連についてはマッピングファイルに従って欲しい時も多い。
  - そのためマッピング定義で指定してもHQLでは個別にJOINフェッチを指定する必要がある。
    - HQL実装者がその点を理解しなければいけないので面倒。
  - Criteriaはマッピングファイル中のJOINフェッチ指定に従う。
    - これもこれでResultTransformerでDistinctしないとマッピング定義のコレクションのJOINフェッチ有無で検索結果が変わってしまうのでリスクではある。
  - 対応策
    - HQLとCriteriaの一方しか使わない。(無理か？)
    - 慣れる。

フェッチ　ー　リスク(その2)

- 外部キーが無い1対0..1関連にはLAZYフェッチは使えないリスク
  - 外部キーが無い場合はプロキシの有無を外部キーがnullか否かで制御出来ないためで、その場合EAGERフェッチされる。
  - サイズの大きいオプションデータを1:0..1で追い出しておくようなケースでEAGERにロードされてメモリを大量に消費したりパフォーマンスが悪化したりする場合がある。
  - さらにHQLでの検索の際にはJOINフェッチを明示的に指定しないとN+1になってしまう。
    - EAGERフェッチ＋SELECTフェッチになってしまうためN+1になる。
    - しかもone-to-oneマッピングではBATCHフェッチやSUBSELECTフェッチは出来ない。
  - 対応策
    - 1対0..1関連は主キー関連ではなくて外部キー関連で実装する.
    - 1対0..1関連は1対0..N関連としてマッピング定義する.

フェッチ　ー　リスク(その2)

- 外部キーが無い1対0..1関連にはLAZYフェッチは使えないリスク



ビルドシステム

フェッチ　ー　リスク(その3)

- 一つのクエリでは一つのコレクションしかJOINフェッチができないリスク
  - 親-子-孫の関係でも一つのクエリでは一つのコレクションしかJOINフェッチ出来ない。
  - そのため親-子-孫の関連をフェッチする際にはn＋1問題が発生する。
  - 対応策
    - BATCHフェッチを利用する。
    - SUBSELECTフェッチを利用する。
  - この件については、3.Xのリファレンスの「関連と結合」の章には出来るような事が書いてあるのですが、確認したところ例外が発生しました。もしかしたら特定のケースで制限が緩和されているのかもしません。

フェッチ　ー　リスク(その4)

- 参照整合性が無いN対1関連もマッピング出来るようになった正のリスク
  - Hibernate 2.1では参照整合性制約をつけていなくて外部キーの参照先が無いデータが存在するN対1関連はフェッチの際に例外が発生してしまった。
    - レガシーなＤＢではこのようなケースが結構あるので辛かった。
  - Hibernate 3.0よりmany-to-oneマッピングで not-found=“ignore”を指定すれば外部キー＝nullと同様の扱いをしてくれるようになった。

フェッチ　ー　リスク(その5)

- フェッチ戦略はそもそも難しいというリスク

  - フェッチに影響するパラメタが多くてしかも関連しあっている。
    - 他のパラメタや検索方法の変更で意図せずにフェッチ形態が変わってしまう事がある。

  - 対応策
    - 十分に時間をかけて取り組む。
    - 詳しい人をプロジェクトに巻き込む。

アジェンダ

- 一般的な事
- 永続化ライフサイクル
- トランザクション
- フェッチ
- クエリ
- キャッシュ
- その他
- まとめ

クエリ　ー　概要

- Hibernateのクエリは3種類
  - HQL
    - Hibernateのメインのクエリ機能
    - SQLのような文字列で検索条件を指定する
  - Criteria
    - 検索条件を文字列ではなくてメソッド呼び出しで組み立てる
  - ネイティブSQL
    - ＤＢＭＳ特有のSQLで検索条件を指定する

クエリ　ー　リスク(その1)

- HQLでの開発はサクサク感が無いというリスク
  - SQLのように簡単に実行できない。
    - Eclipse上でHibernateTools等からは実行できる。
    - 「単体テストを書くのだからそこで試行すれば良い」と実装者を納得させるのが大変。
    - S2DAOで2WAY-SQLを使っている人でなくても文句を言う。
  - SQLと似ていて違うので余計に混乱する。
    - 出来ない事が出来ないと気づくまでに時間がかかって腹が立ったりする。
  - コミュニケーションが面倒になる。
    - SQLは共通語だがHQLはそうじゃない。
    - 実装者がDBAに複雑な検索について相談するとHQLを知らないDBAは当然SQLで検索方法を答えるので実装者は自分でHQLに翻訳しなければならない。
  - 対応策
    - HQLを使わない。(無理か？)
    - 慣れる。(実装者だけでなくDBAも)

クエリ　ー　リスク(その2)

- HQLで動的なクエリがサポートされていないリスク
  - S2DAOのIFコメントのような動的な条件分岐がサポートされていない。
  - プログラムで組み立てるとHQLがコード上にバラバラに入ってしまい保守しにくい。
  - Criteriaでやろうにも複雑な事は出来なかったり結局保守しにくかったりする。
  - 対応策
    - 自力で動的にクエリの未使用条件を消し込む機能を実現する。
    - 素直にプログラム上で動的に組み立てる。

クエリ　ー　リスク(その3)

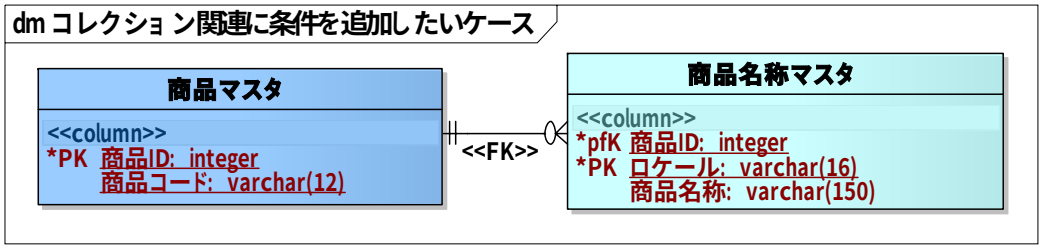
- HQLのfrom節にサブクエリを書けないリスク
  - 最近のDBMSは皆from節にサブクエリを書けるのでそれに慣れていると結構つらい。
  - select節とwhere節には書ける。
  - Criteriaでも出来ない。
  - 対応策
    - 可能ならfrom節にサブクエリを書かないで同じ結果を得る方法に変更する。
    - ネイティブSQLクエリを使う。

クエリ　ー　リスク(その4)

- HQL,Criteriaクエリで、マッピング定義に関連を記述していない条件ではJOIN出来ないリスク
  - HQLのJOIN節やCriteriaのFetchMode.JOINは結合条件を指定するのではなくてJOINフェッチを指定するものと考えた方が良い。
    - ただし、コレクションでFetch指定なしの場合と対1結合の場合にはHibernate 3.1で追加されたwith節で条件の追加はできる。
      - from Owner as owner left join owner.employees as employee with employee.waist >= 85.0
      - でもコレクションの場合Fetch指定なしだと結果がObjectの配列になっちゃうのでいまいち。Fetch指定ありでwithが使いたい・・・
  - 対応策
    - 参照整合性があるならマッピング定義に関連を記述する。
    - ネイティブSQLクエリで管理外エンティティを使用する。
    - 等価結合で良ければ以下のように記述可能。ただし結果はObjectの配列になる。
      - 「from Hoge hoge, Foo foo where hoge.id = foo.id」

クエリ　ー　リスク(その5)

- HQL,Criteriaクエリでコレクションの関連条件を動的に追加するのが面倒なリスク
  - 静的条件で良いならマッピング定義のwhere属性で追加出来るが実行時にオーバーライド出来ない。また、with節を使うと結果がObjectの配列になってしまう。
    これは以下のケース等で不便。
    - 多言語対応での読み込む言語の切り替え
    - 論理削除レコード読み込み有無の切り替え
  - LAZYでしかもコードをいちいち書けるならsession.createFilter()で対応出来なくもないが面倒。
  - 対応策
    - コレクションのサイズが小さいならモデル変換部分等に自前のフィルタロジック書いて吸収する。
    - コレクションのサイズが大きい場合はsession.createFilter()で対応する。
    - with節を使用して検索し自力で階層構造に復元する。



ビルドシステム

クエリ　ー　リスク(その6)

- Criteriaは列指定も条件指定もタイプセーフになってないリスク
  - テーブル名や列名, 列型の変更でコンパイルエラーにならない。
  - IDEの補完機能が使えずサクサク感が無い。
  - 当然であるがこの点はソース自動生成タイプのTorqueやDBFluteに負けてる。
  - 対応策
    - 自分で自動生成コードでラップしてタイプセーフにする処理系を作る。
    - 中途半端なタイプセーフのCriteriaは使わない。
    - 素直に受容する。

クエリ　ー　リスク(その7)

- パラメタをバインド変数ではなくリテラルに出来ないリスク
  - バインド変数だとインデックスを使用してくれないケースがあるDBMSではチューニングが面倒。
    - あるＤＢＭＳでは、外部結合したテーブルの項目に対するwhere節の条件指定がバインド変数だとインデックスを使ってくれない.
      - リテラルに出来ないと条件有りの時だけ内部結合に変更しなければならない。
      - このケースに限ればCriteriaなら自動的に内部結合にしてくれるがHQLはダメ。
    - このようなケースに柔軟に対応する事が出来ない。
  - S2DAOの外だしSQLのようにパラメタ毎に簡単にリテラル化が出来るようにしてほしい。
  - 対応策
    - 受容する。
    - 自分でパラメタをリテラルにしたHQLを組み立てる。
    - そんなタイプのDBMSの場合にはHibernateを断念する。(さすがにこれは無いか)

クエリ　ー　リスク(その8)

- HQL,Criteriaクエリで**UNION**がサポートされていないリスク
  - **DB設計の方針等により**UNION**を多用するプロジェクトでは厳しい。**
  - **対応策**
    - **可能なら**UNION**を使わない方法に変更する。**
    - **VIEWを使用する。**
    - **ネイティブ**SQL**クエリを使う。**

クエリ　ー　リスク (その9)

- ネイティブSQLクエリは実はあまりネイティブではないというリスク
  - 書き方が素のSQLではないケースがある。
    - 例えば名前の衝突を避ける場合は、
      “SELECT **{cat.*}, {mother.*}** FROM CATS c, CATS m”
      と書くので、SQLとしてそのまま実行出来ない。
  - 永続化ライフサイクルの管理下である。
    - マッピング定義されたエンティティは結局永続化ライフサイクルの管理下なので難解なフェッチ戦略等からは開放されない。
  - 対応策
    - 受容する。
    - 出来るだけ管理外のエンティティ (Hibernate 3.2からの新機能) を利用する。

クエリ　ー　リスク(その11)

- ネイティブSQLクエリで管理外エンティティが使えるようになった正のリスク
  - Hibernate 3.2からは管理外のエンティティも使えるようになった。
    - これによってテーブルに対応しないエンティティでの受け取りが簡単になった。
    - ちなみにHQLでは「select new MyEntity(hoge, bar, ...) from ...」構文でHibernate2.1(それ以前？)から管理外のエンティティに対応していた。
  - 永続化ライフサイクルの管理外なのでフェッチ戦略を気にしなくて良い。
  - 欠点もある
    - まっ平らなエンティティで受けるので階層構造には自分で変換するしかない。
    - DBメタ情報のままの項目名称でsetterを呼ぶのでsetHogeFoo()じゃなくてsetHOGE_FOO()やsethoge_foo()といったsetterが無いとダメなのは中途半端。(独自のAliasToBeanResultTransformerを実装すれば回避可能)

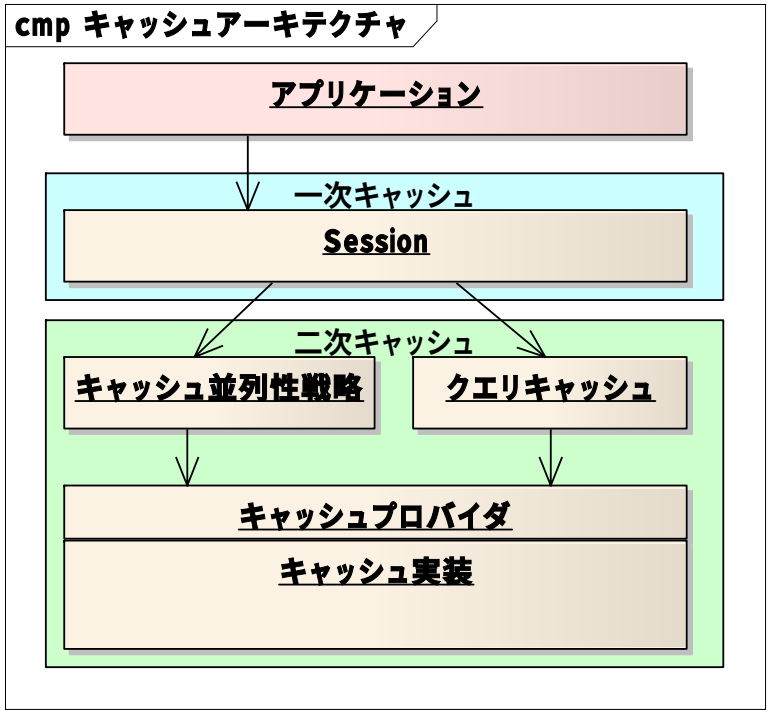
クエリ　ー　リスク(その12)

- クエリはHibernate 2.1のころと比べると色々な機能が強化されているという正のリスク
  - 3.0でHQLのパーサ実装が変更されて機能が強化された。
    - 例えばcase節等もここで追加された。
  - 3.0でinsert,update,deleteのためのカスタムSQLを定義してHibernateの処理をオーバーライド出来るようになった。
  - 3.0でCriteriaがリファクタリングされて機能強化された。
    - サブクエリやグループ化がサポートされた。
  - 3.1でHQLにwith節が追加されてjoinでの結合条件の追加が可能となった。
  - 3.2でネイティブSQLクエリで管理外エンティティが使えるようになった。

クエリ　ー　リスク(その13)

- どのクエリを選択すれば良いのか迷うというリスク
  - 一般的にはHQLを使うのが無難。
    - 中途半端なタイプセーフのCriteriaより良いとも思う。
  - しかしHibernateのクエリはどれも一癖あって決定力に欠ける。
    - ネイティブSQLクエリをもっと強化してほしい。
    - S2DAOの2WAY-SQLのような機能を実現してほしい。
    - そうすればS2DAOユーザの何割かを取り込めるかも。
  - ネイティブSQLクエリが強化されればHQLレスの開発も可能かもしれない
    - 簡単な検索は自力でタイプセーフにラップしたCriteriaを使い、
    - 難しい検索は名前付きのネイティブSQLクエリで解決する。
    - HQLは使わない。
      という方法にチャレンジ出来るかも？(それなら素直にDBFlute使った方が・・・)

キャッシュ　ー　概要



- 二次キャッシュ利用時の並列性戦略はテーブル毎に以下の何れかを選択できる。
  - read only 戦略
  - read/write 戦略
  - nonstrict-read/write 戦略
  - transactional 戦略

キャッシュ　ー　リスク(その1)

- 二次キャッシュはおまけ程度と考えた方が良い
  - 他の処理系からも書き込みを行うテーブルには二次キャッシュは適用しにくい。
    - そのため二次キャッシュ無しでは性能が出ない可能性がある場合はHibernateの採用は止めた方がよい。
  - 区分値等の定数テーブルには非常に有効。
    - いちいちJOINフェッチ指定しなくてもSELECTフェッチでキャッシュヒットする。
  - 基本的には以下のように適用を検討する。
    - 二次キャッシュ適用を積極的に検討すべきテーブル
      - 区分値等の定数テーブル
    - 二次キャッシュ適用を検討すべきテーブル
      - 更新が少ないマスタ系テーブル
    - 二次キャッシュ適用を避けるべきテーブル
      - トランザクション系テーブル

キャッシュ　ー　リスク(その2)

- 一次キャッシュの使用量に関するリスク
  - 一次キャッシュはOFFに出来ない。
    - 上限サイズの指定も出来ない。
  - セッション内で使用された永続オブジェクトは全て保管される。
    - クエリ結果の永続オブジェクトも全て保管される。
    - 明示的にevict(),clear()するかセッション終了まで破棄されない。
  - そのため大量のデータを扱う時には適宜evict(),clear()しないとOutOfMemoryErrorが発生する。
  - 対応策
    - 大量の永続オブジェクトを扱う場合には適宜evict(),clear()する。
    - また、大量の検索結果を扱う場合には必ずScrollableResultsを利用する。

キャッシュ　ー　リスク(その3)

- S2Hibernateを使用するとread-write戦略で二次キャッシュが効かなくなるリスク
  - S2HibernateはConnectionProviderを使用せずにSessionFactory.openSession(Connection connection) でSessionにコネクションを設定している。
  - そのため二次キャッシュのリピータブルリードを制御するためのトランザクション開始時刻にLong.MIN_VALUEが設定されてしまい全てのキャッシュエントリが無視される。
  - 対応策
    - S2SessionFactoryImplをConnectionProviderを使用出来るように修正する。
    - リピータブルリードでないnonstrict-read-write戦略でキャッシュを使用する。
    - transactional戦略キャッシュなら大丈夫かも？(未確認です)

その他のリスク

- ボトムアップ開発ではマッピングファイルの微調整が面倒。
  - モデリングツールにDDL吐かせてボトムアップで開発する場合にマッピングファイルの微調整が面倒。
  - 自動生成したマッピングファイルに対して別ファイルでオーバーライドするような機能が欲しい。
- ログでバインドパラメータを見るのが大変。
  - PreparedStatementのバインドパラメータを見ようとするとResultSetの内容も出力されてしまうのでうるさい。(自分でフィルターを書けば良いんですけど)
  - S2DAOみたいにそのままコピペでSQLとして実行出来るように'?'の部分をバインドパラメタで置換したログを出すオプションも欲しい。
- 正直ソースはかなり読みにくい。
  - そのためソースを追いかけるのは相当のスキルが無いと大変。
  - これはOSSにとっては大きな弱点でそのために離れていく人も多いと思う。
- 抵抗勢力の存在
  - このリスクの対応方法はセッションのスコープ外なので割愛します。

まとめ

- バージョンが上がる毎に機能追加されて、2.1の頃に比べると出来ない事は減っている。
- しかし使い方が簡単になったわけではない。
- やはり短納期のプロジェクトで初めてHibernateを使うというのはリスクが大きい。
  - 特に高度なSQL記述への依存度が高いシステムや、そういった開発手法を好むチームへの適用は慎重に検討した方が良い。
- しかし使い慣れてしまえば大抵のシステムに適用可能なO/Rマッパーである。
- 今後も利用者の意見を取り入れて更に使いやすいものになって行くでしょう。

- ご静聴ありがとうございました。