◆ O-R マッピング

ページ数の関係で紙面に掲載できなかったパターンを、ここで解説します。

ユニットオブワーク(Unit of Work)

DB からデータを取り出す際は、その後、そのデータがどのように変更されたかを追跡することが重要です。さもなければ変更内容を DB に反映することはできません。新しく生成されたオブジェクトは挿入し、取り除かれたオブジェクトは削除しなければなりません。更新がある度に毎回 DB に反映させることも出来ますが、それでは細かな DB の呼び出しが発生してしまい、非常に効率が低下してしまいます。しかも、作業中ずっとトランザクションを開いておかなければならなくなります。これは複数のリクエストにまたがるビジネストランザクションでは現実的ではありません。読み込んだオブジェクトの不整合読み取りを防がなければならい場合は更に深刻となります。ユニットオブワークはビジネストランザクションの中で変更した内容(DB に書き戻さなければならないもの)を記録する仕組みです。これを使えば、作業完了時に DB に反映が必要な内容を知ることができます。

明らかに記録が必要な内容として更新が挙げられます。つまり新しいオブジェクトが生成された時と、既に存在するものが更新、削除された時です。読み込みに関してもユニットオブワークに伝える必要があります。これは不整合読み取りが起きていないか監視するためです。コミットの段階でユニットオブワークは、何をすべきかを判断します。システムトランザクションを開始し、悲観的、あるいは楽観的オフラインロックによる同時更新チェックを行い、実際の DB 更新を行います。アプリケーションプログラマが、直接の DB 更新処理を記述することはありません。これによりアプリケーションプログラマは、何が変更されたかを記録したり、参照保全(全ての外部キーが有効であることの保証)への影響を考えながら、更新の順序について悩む必要がなくなります。ユニットオブワークはバッチ更新を行う場所としても適しています。バッチ更新とは複数の SQL を一回で行ってリモート呼び出しの回数を減らすものです。これは大量の更新、挿入、削除が引き続いて発生するようなケースに有効です。ユニットオブワークはトランザクション更新可能なリソースならば、DB に限らず利用できます。メッセージキューやトランザクションモニタの制御にも使用できるでしょう。

アイデンティティマップ(Identity Map)

DB のロウはキーで識別します。同じキーを持つロウが 2 つ以上存在することはありません。オブジェクト指向の世界では、これは参照に相当します。同じ参照(C++で言えばオブジェクトのアドレス)を持つオブジェクトが 2 つ以上存在することはありません。ロウからオブジェクトに読み込む際には、同じキーを持つロウを同じオブジェクトにマッピングしなければなりません。これがアイデンティティマップの役目です。これと関連して、パフォーマンスの問題があります。もしも同じデータを 2 回以上読み込んでしまうと、リモート呼び出しの場合には高いコストについてしまいます。アイデンティティマップは 1 つのビジネストランザクションの中で DB から読み込まれたレコードを記録し、次にオブジェクトを読み込もうとしたら、まず先にアイデンティティマップを調べて既に読み込み済みであれば、それを再利用します。

オブジェクトがイミュータブル(変更不可)ならばアイデンティティマップは不要です。変更できないのであれば、更新について気にする必要はありません。たとえば値オブジェクト(Value Objects)はイミュータブルなのでアイデンティティマップで扱う必要はありません。また、後述の従属マッピング(Dependent Mapping)ではアイデンティティマップを使用する必要はありません。従属オブジェクトの永続性管理は、従属している親が管理するからです。

継承マッピング

単一テーブル継承(Single Table Inheritance)については、紙面を参照下さい。

クラステーブル継承(Class Table Inheritance)は、継承ツリーの各クラスを 1 つのテーブルで表現します(図 1)。

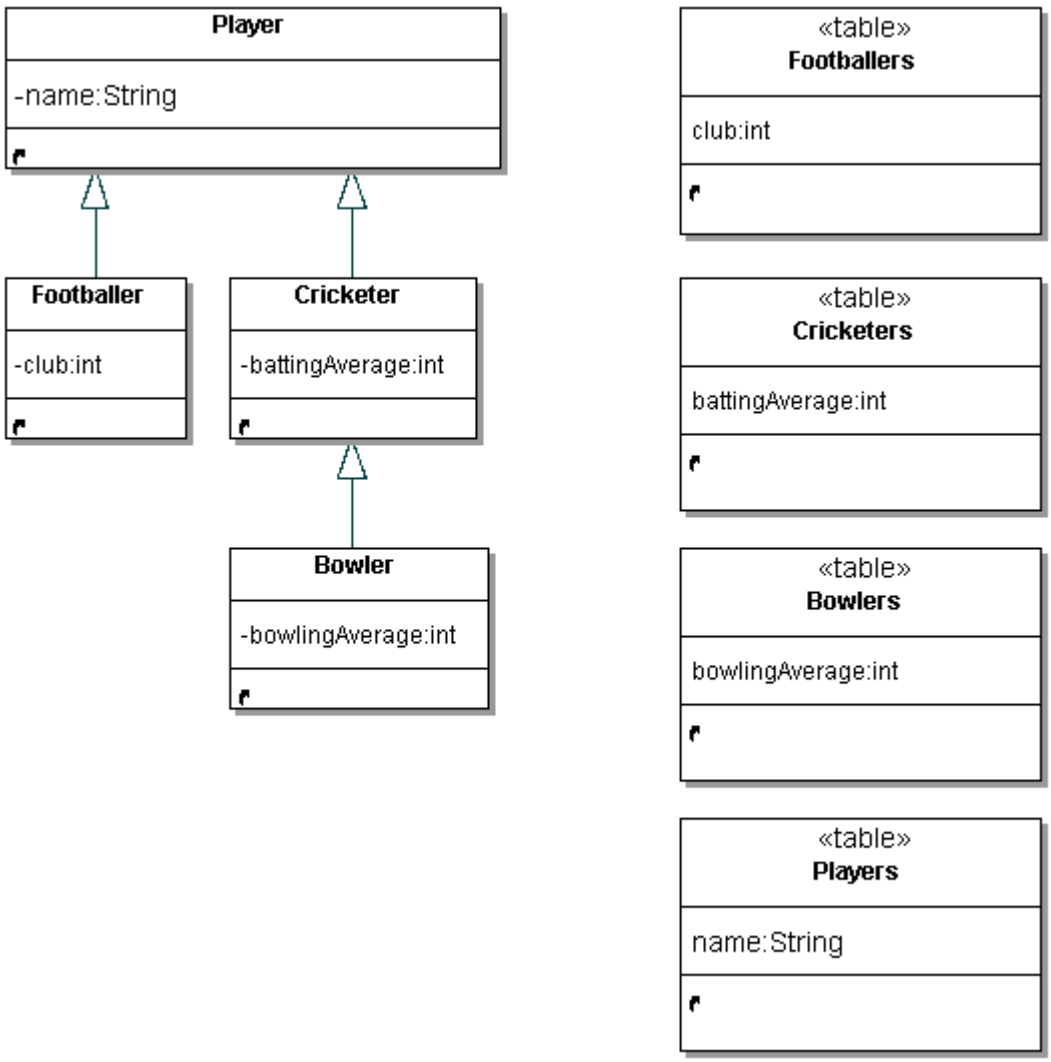

図 1 クラステーブル継承

クラスのフィールドがそのままテーブルのカラムにマッピングされます。1 つのクラスのデータが複数のテーブルに分散して格納されるので、どうやって複数のテーブルに散らばったデータを集めるかが問題となります。1 つの方法は、同じプライマリキーの値を使用するというものです。つまり Footballer テーブルのキー 101 のロウと、Player テーブルのキー 101 が同じオブジェクトを構成しているようにします。プライマリキーは継承ツリーのテーブル内で一意となります。もう 1 つの方法としては、プライマリキーには別のものを持たせて外部キーで関連付けるという方法が考えられます。

次に、いかに効率的に複数テーブルから 1 つのオブジェクトを読み出すかが問題となります。1 つ 1 つ別々にテーブルを読み出したのでは、複数の DB とのやりとりが必要になって、非効率です。join でテーブルを結合することもできますが 3 つも 4 つも結合すると、遅くなってしまいます。クエリによってはどのテーブルにアクセスすべきか明確にならない場合もあります。もしもフットボール選手を検索するのであれば、Footballer テーブルを検索すれば良いと分かりますが、特定条件に合った選手のグループを検索するようなケースなど、テーブルを特定できない場合があります。カラムが空かもしれないテーブルを join するには、outer join が必要になりますが、これは標準的な方法ではなく、実行速度も遅くなります。最初に階層の最上位のテーブルを読み、コードを調べてから読み出す方法もありますが、これでは複数のクエリが必要となってしまいます。

この方法の長所は以下の通りです、

- ロウ内の全てのカラムが関連していて理解が容易でありスペースの無駄がない。
- ドメインモデルと DB との関連がストレートである。

短所は以下のようになります。

- 読み出しのために複数のテーブルにアクセスしなければならない(join か複数クエリ)。
- リファクタリングでフィールドを継承クラス間で移動するとDB スキーマも変更となる。
- 継承ツリーの上位側のテーブルは頻繁にアクセスされることになり、パフォーマンス上のボトルネックとなるかもしれない。
- 高度に正規化されすぎていて、アドホックなクエリがやりにくくなるかもしれない。

具象テーブル継承(Concrete Table Inheritance)は、継承ツリーの中の具象クラスをテーブルに対応させる方法です(図 2)。

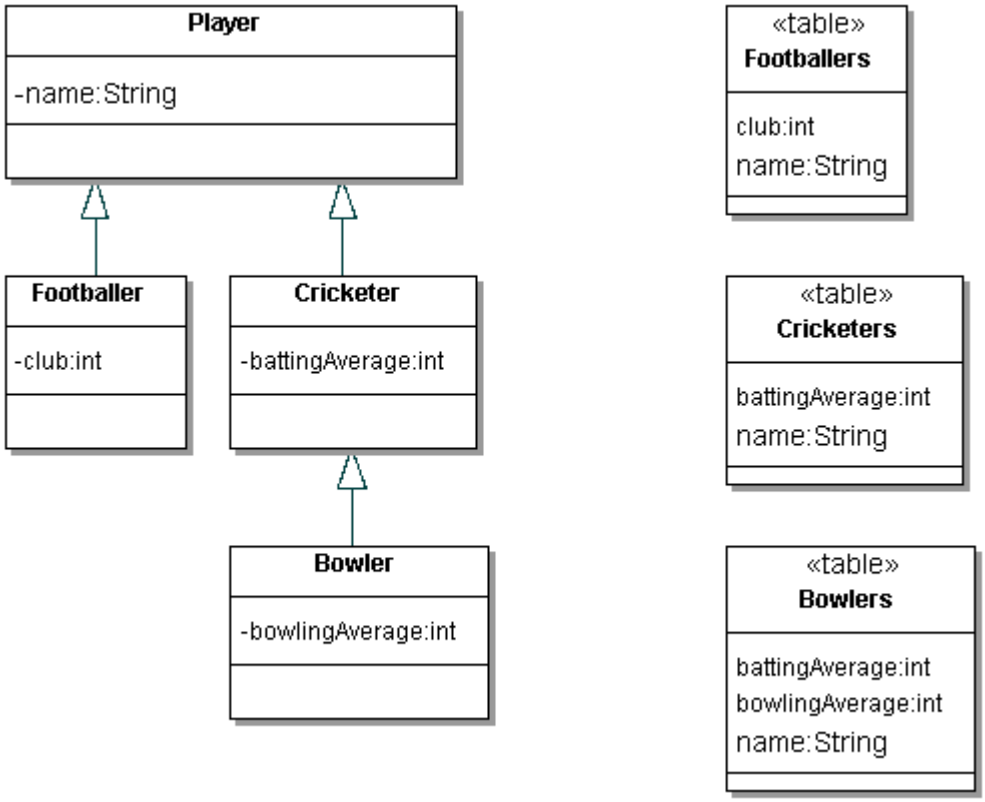

図 2 具象テーブル継承

それぞれのテーブルは該当クラスと、その先祖のクラス全てのフィールドを持ちます。このため親クラスのフィールドは重複して格納されることになります。クラステーブル継承と同様、あるプレーヤを select 文で検索したい場合、全てのテーブルを見て回らなければならなくなるので効率が悪くなります。
この方法の長所は以下の通りです、

- 各テーブルは自己完結であり関係のないフィールドを持たない。この構造はオブジェクト指向でない別のアプリケーションから見た時に自然である。
- 具象クラスのマッパからデータを読み出す時に join が必要ない。
- クラスごとに分離されているので、アクセス負荷が分散される。

短所は以下のようになります。

- プライマリキーの取り回しに難がある(多態性を実現するためにはキーが継承ツリー内で一意でなければならない)。

- 抽象クラスに対して DB の関連を求めることができない(抽象クラスは対応するテーブルを持たないので)。
- もしも親クラスのフィールドの定義が変更された時には、全ての子クラスのテーブルスキーマが影響を受ける。
- 親クラスのレベルで検索をしたい場合には、全ての子クラスのテーブルも検索しないといけない。これには outer join や複数のクエリを要する。