平成16年度 学士論文

アプリケーション依存の
先読みが可能な
O/Rマッピングツール

東京工業大学 理学部 情報科学科
学籍番号 01-0017-3
青木 康博

指導教員
千葉 滋 助教授

平成17年2月14日

概要

リレーショナルデータベース (RDB) を利用して大量のデータを扱うアプリケーションの開発において、O/R マッピングフレームワークが広く利用されるようになった。O/R マッピングとは、RDB のレコードとオブジェクト指向言語のオブジェクトを対応づける操作を指し、O/R マッピングフレームワークによってオブジェクトへのデータ読み込みや RDB へのオブジェクト永続化がアプリケーションから透過的に行なえるようになる。さらに、RDB のテーブルとクラス間の煩雑なマッピング作業やトランザクション処理が自動化されユーザの手間が軽減される。

しかし、既存のフレームワークは効率の良い O/R マッピングを実現できていない。効率の良い O/R マッピングとは、少ない RDB アクセスによる無駄の無いマッピングである。例えば、今後必要になる可能性の高いデータを予め一括して RDB から取得 (先読み) してキャッシュしておくような技術がそれにあたる。一般に Web アプリケーションと RDB が別々のサーバに配置されるようなシステムでは、通信のオーバーヘッドが大であるため高度な先読み技術による最適な O/R マッピングが実行性能を飛躍的に向上させられる。ところが、先読みの性能はアプリケーションや実行環境に大きく依存するため、積極的な先読みによる不必要なデータ読み込みや消極的な先読みによる冗長な RDB アクセスを抑え、メモリ使用やネットワークインタラクションを最適化するのは非常に難しい。そのため、多くの既存フレームワークでは、不確定なデータに対する先読みをせず、テーブル間の静的な依存関係のみをユーザに XML で記述させる手動の先読みをサポートしている。しかし実際にはアプリケーションコンテキストに応じて、次にアクセスのあるテーブルを予測することは可能である。そのため、多くの熟練した開発者はプログラム中に直接 SQL を記述して性能向上を図り、O/R マッピングフレームワークの導入による可読性、拡張性、保守性の利得を消失させてしまっていた。

本研究では、アスペクト指向プログラミングに基づいた O/R マッピング (先読み) 記述を支援する Java 向けの O/R マッピングフレームワークを開発した。本フレームワークを利用することで、マッピング記述はアプリケーション内に散在することなく、かつ柔軟にアスペクトとしてプログラミングできるようになった。本フレームワークの実現にあたってはま

ず、既存の O/R マッピングフレームワークである Cayenne を改造してテーブルレコードのプロパティごとの取得を可能にした。また、レコード取得の際には必要最小限のプロパティのみを取得するように変更した。この変更により、必要なデータのみを取得することが出来る。しかし、これだけでは、プロパティにアクセスがあるたびに RDB へアクセスする必要があり、 RDB への過剰なアクセスを誘発してしまう。そこで、本フレームワークでは XPath と呼ばれる XML ドキュメント用の要素指定方法に基づいて先読みデータの指定とそれを実施するタイミングが、レコードアクセスの履歴をもとに指定可能な API を用意した。アスペクト内でこの API の記述を用いることによって、開発者はアプリケーションから分離されたモジュールから、テーブルのアクセスパターン、すなわちアプリケーションフローに応じた先読みを直接的かつ容易に記述可能になり、効率のよい O/R マッピングを実現可能になった。予備的な実験から、本フレームワークを用いて単純な先読みアルゴリズムを適用したアプリケーションで既存の O/R マッピングフレームワークの利用に比べて実行速度、メモリ消費、RDB アクセス回数の点で大きく性能を改善できることを確認した。

謝辞

本研究を進めるにあたり、研究の方向づけや論文の組立て方についての助言をしていただいた指導教官の千葉先生に感謝いたします。

東京工業大学の佐藤芳樹氏には、システムの設計・実装に始まり、論文の組み立て・実験方法まで多岐にわたって指導して頂きました。心より感謝いたします。

また、西沢無我氏には、多くの疑問を聞いて頂き、指導して頂きました。柳澤佳里氏にはデータベースに関する様々な助言を頂きました。須永豊氏、松沼正浩氏には、論文の構成について多くの助言を頂きました。感謝いたします。そして、本稿のスタイルファイルを作成ファイルを作成して頂き、論文の構成に関して多くの助言をして頂いた光来健一氏に感謝いたします。

最後に、卒業研究を行う上で励まして頂いた同研究室の皆様に感謝いたします。

目 次

第 1 章　はじめに　8

第 2 章　既存の O/R マッピング技術　11

2.1 O/R マッピング 11
2.1.1 JDBC 11
2.1.2 O/R マッピング 11
2.1.3 先読みの必要性 13
2.2 アプリケーション依存のマッピングの困難さ 14
2.3 既存の O/R マッピングフレームワーク 16
2.3.1 Entity Bean 16
2.3.2 Hibernate 18
2.3.3 Cayenne 23

第 3 章　アプリケーション依存の先読みが可能な O/R マッピングフレームワーク　27

3.1 特徴 27
3.2 先読み記述の分離 29
3.2.1 アスペクト指向プログラミング 29
3.2.2 AspectJ 30
3.3 XPath 32
3.4 仕様 35
3.5 記述例 38
3.5.1 テーブル・レコードの先読み 38
3.5.2 テーブル・カラムの先読み 39
3.5.3 Relationship の先読み 40

第 4 章　実装　42

4.1 O/R マッピングフレームワーク Cayenne の拡張 42
4.1.1 プロパティごとのマッピング 42
4.1.2 永続クラスの拡張 45
4.2 先読みを支援するライブラリの実装 47

第 5 章　実験 **49**
5.1　実験環境 49
5.2　取得したテーブルレコードの一部のプロパティしか利用しない場合が多いアプリケーション 49
5.3　取得したテーブルの一部の行 (レコード) しか使用しない場合が多いアプリケーション 50
5.4　ランダムにテーブルを辿り、不特定に取得したテーブルの全要素を利用する場合が多いアプリケーション 51
5.5　考察 52

第 6 章　まとめ **54**

図 目 次

2.1 永続クラスの例 13
2.2 Hibernate の構造 14
2.3 CMP Entity Bean での RDB とのマッピングの例 18
2.4 HQL の概要 19
2.5 Hibernate での RDB とのマッピングの例 21
2.6 Hibernate 内部の List の実装 22
2.7 Hibernate の構造 23
2.8 Expression を用いた RDB へのアクセス例 24
2.9 Cayenne Modeler によって生成された XML ドキュメント . 25

3.1 prefetch("./@*") による先読みの様子 39
3.2 prefetch("../Proceeding/@year") による先読みの様子 . . . 40
3.3 prefetch("./Proceeding/Paper/@*") による先読みの様子 . 41

4.1 Cayenne 内で Select 文が発行されるまでの流れ 43
4.2 拡張後の appendAttributes() メソッド 44
4.3 永続クラス 45
4.4 CayenneDataObject に付け加えた主な機能 47
4.5 先読み実行の流れ 48

5.1 マッピングの記述例 50
5.2 マッピングの記述例 51
5.3 マッピングの記述例 52

表 目 次

3.1 XPath で定義されているノード 33
3.2 XPath で定義されている軸 34
3.3 ノードテスト 34
3.4 ノードテスト 34
3.5 PrefetchAspect で定義されているポイントカット 35
3.6 PrefetchAspect で定義されているメソッド 35
3.7 XPath の仕様 37

5.1 一部のプロパティしか利用しない場合の比較 50
5.2 一部のレコードしか利用しない場合の比較 51
5.3 ランダムにデータにアクセスする場合の比較 52

第1章　はじめに

J2EE サーバなどの大規模な Java システムにおいてデータベースアクセスは必須の機能となっている。データベースには多くの管理方法があるが、その中でデータの蓄積量やパフォーマンス、コスト面でもっとも優れているとされているのがリレーショナルデータベース (RDB) である。

近年、 RDB を利用したアプリケーションの開発には O/R マッピングフレームワークが広く利用されるようになった。O/R マッピングとは、RDB のレコードとオブジェクト指向言語のオブジェクトを対応づける操作を指す。オブジェクト指向言語では、全ての事象をオブジェクトとして表す、一方 RDB は全てのデータを２次元のテーブルレコードとして表し、テーブルごとの関連づけによって事象を表す。そのため、RDB を用いたプログラミングにはオブジェクトと RDB のテーブルとの対応づけを常に意識しながらオブジェクトへのデータ読み込みや RDB へのオブジェクト永続化などをおこなう必要があった。O/R マッピングフレームワークはこれらの処理をアプリケーションから透過的に行うことを可能にする。さらに、RDB のテーブルとクラス間の煩雑なマッピング作業やトランザクション処理が自動化されユーザの手間が軽減される。

しかし、既存のフレームワークは効率の良い O/R マッピングを実現できているとは言えない。効率のよいマッピングとはできるだけ少ない RDB アクセスで必要なデータだけをマップすることである。効率のよい O/R マッピングを実現する技術の一つに先読みがある。先読みとは、今後必要になる可能性の高いデータを予め一括して RDB から取得 (先読み) してキャッシュしておく技術である。例えば、100 個のオブジェクトのリストへのデータ読み込みの際に、一つ一つのオブジェクトに対して RDB にアクセスしていたのでは全てのデータを読み込むためには RDB に 100 回アクセスしなければならない。このような場合、一回の RDB アクセスで 100 個のオブジェクトのデータを先読みすることによって効率のよいマッピングを実現することが出来る。

一般に Web アプリケーションと RDB が別々のサーバに配置されるようなシステムでは、通信のオーバーヘッドが大きいため高度な先読み技術による最適な O/R マッピングが実行性能を飛躍的に向上させられる。ところが、既存の O/R マッピングフレームワークでは、透過的な永続化を

行うために、固定的かつテーブル単位での一括した先読みしかサポートしていない。固定的と先読みというのは、あるテーブル A が取得される際にはテーブル B を先読みするといったような、テーブル間の静的な依存関係のみをユーザに XML で記述させる方法である。

先読みしたいデータはアプリケーションの文脈に応じて変化するものであると考えられる。そのため、テーブル間の静的な依存関係のみでは、効率のよいマッピングを行うことは困難である。そのため、取得するテーブルレコード数が膨大な場合や、レコードに大容量なデータが格納されている場合には、本来は不必要な大量のメモリを消費してしまい、アプリケーションのパフォーマンスを低下させてしまう危険性がある。

既存の O/R マッピングフレームワークがこのような先読みしかサポートしていない理由としては、限定的でない先読み、つまりはアプリケーションの文脈に依存した先読みを実行するためには、プログラム内のあちこちに先読みのコードが散在してしまうことになり、アプリケーションの保守性や拡張性を大きく阻害してしまうという理由がある。先読みは、アプリケーションのパフォーマンスを左右する重要な機能であるため、アプリケーションの完成後も幾度となくチューニングする可能性があると考えられる。このような場合、先読みコードがアプリケーション内に散在してしまっていると、非常に効率が低下してしまう。

本研究では、アスペクト指向プログラミングに基づいた O/R マッピング (先読み) 記述を支援する Java 向けの O/R マッピングフレームワークを開発した。本フレームワークを利用することで、マッピング記述はアプリケーション内に散在することなく、かつ柔軟にアスペクトとしてプログラミングできるようになった。また、先読みデータのも行・列単位という粒度の細かな単位でのマッピングを可能にした。

アスペクト指向とは、オブジェクト指向のみではモジュール化しきれない機能を分離するための技術である。本フレームワークでは、先読みの記述をアプリケーションから分離するために、Java 言語をアスペクト指向的に文法拡張した言語である AspectJ を利用した。さらに、本フレームワークで用意したデータベースアクセスの履歴を得る API を利用することによって、文脈に依存した先読みの指定が可能となった。

本フレームワークの実現にあたってはまず、プロパティごとに RDB のテーブルをマッピングするように、既存の O/R マッピングフレームワーク Cayenne を 改造した。従来、このような O/R マッピングは RDB への過剰な SQL 発行を誘発するため敬遠されていたが、先読みの細密なサポートによって現実的になる。また、本フレームワークでは、 Cayenne を改造して DB アクセスの履歴を辿ることが可能にした、さらに、 XPath と呼ばれる XML ドキュメント用の要素指定方法に基づいて、RDB アク

セスの履歴をもとに先読みのタイミングを記述できる API を用意した。アスペクト内でこの API を利用することで、開発者はテーブルのアクセスパターン、すなわちアプリケーションフローに応じた先読みが可能となった。先読みデータの指定も、XPath によって記述する API を用意した。指定したタイミングで、この API によって先読みデータを指定することで開発者は直感的かつ容易に、効率のよい O/R マッピングが実現可能になった。

以下、2 章では、既存の O/R マッピングフレームワークとその問題点について述べ、3 章では本研究で開発したフレームワークの仕様、利用例について述べる。4 章では、本フレームワークを開発するにあたって、改造した既存の O/R マッピングフレームワークである Cayenne の構造と、その改造点、用意した API の実装について述べる。 5 章では文脈に応じた先読みが必要な典型的なアプリケーションにおいて、Cayenne を用いてテーブル単位でのマッピングを行った場合、Cayenne を拡張してプロパティ単位でのマッピングを行った場合、本フレームワークを用いてプロパティ単位でのマッピングかつ文脈依存の先読みを行った場合の、実行時間、DB アクセス回数、メモリ使用量を比較した実験について述べる。最後に、6 章で本論文をまとめる。

第2章　既存のO/Rマッピング技術

2.1　O/Rマッピング

Web アプリケーションなどのサーバサイド Java システムからリレーショナルデータベースを利用するためにはオブジェクトとリレーショナルデータベースとの対応づけ (O/R マッピング) が必要となる。O/R マッピングとは、データベースからオブジェクトへのデータ読み込みや、オブジェクトからデータベースへの永続化作業を指す。

以下に O/R マッピングの手段である **JDBC** や既存の O/R マッピングフレームワーク、その問題点について述べる。

2.1.1　JDBC

JDBC とは Java プログラムからリレーショナルデータベースにアクセスするための API のことである。

リレーショナルデータベースにアクセスするための SQL 言語や、アクセスのための Java の API は各データベースベンダーごと異なっている。それゆえ、 Java プログラムからデータベースにアクセスするためには、使用するデータベース製品によって異なる API を用いなければならなかった。このためデータベース・アクセスのための標準 API として誕生したものが **JDBC** である。**JDBC** を用いることによって開発者は **JDBC** をサポートするどのデータベースに対しても統一された手順で操作を行うことが可能となった。

2.1.2　O/Rマッピング

O/R マッピングとは、リレーショナルデータベース (以下 RDB) と Java オブジェクトとの対応付けのことである。

Java と RDB とはそれぞれ全く異なったモデルに立脚したアーキテクチャーを採用している。Java はオブジェクト指向言語であり、その言語上で処理やデータなど全ての事象をオブジェクトとして表現する。それに対し、 RDB は二次元のテーブル形式でデータを表し、それらのテー

ブルの関連付けによって事象を表す。したがってデータベースアクセスの際には、データベースからオブジェクトへのデータ読み込みや、オブジェクトからデータベースへの永続化作業が必要となる。このオブジェクト、RDB 間の変換 (マッピング) を O/R マッピングという。

Java から RDB にアクセスするためには先ほど述べたように **JDBC** を用いるが、O/R マッピングには大きな問題点がある。一般にデータ変換処理には、処理速度の低下や情報劣化といった危険性が伴う。また、O/R マッピングをおこなうにはその作業自体に大変な手間がかかることは言うまでもない。さらに RDB にアクセスするためのクエリ言語 **SQL**(**Structure Query Language**) は非オブジェクト指向であるため、継承やカプセル化の概念がない。そのためオブジェクト指向の柔軟性が損なわれてしまう可能性がある。このようなモデルの違いによって生まれた、Java と RDB との大きな隔たりは「**インピーダンス・ミスマッチ**」と呼ばれている。

O/R マッピングのためには、この**インピーダンス・ミスマッチ**を埋める必要がある。また、**インピーダンスミスマッチ**以外にも **JDBC** を用いた RDB アクセスには多くの問題点がある。例えば、**JDBC** を用いる場合、データベース内のテーブルスキーマが変更されるとプログラム内に埋め込まれた SQL 文を逐一変更しなければならないという問題点や、DB 製品ごとの互換性が完全ではないという問題点があげられる。

このような様々な問題を解決するのが O/R マッピングフレームワークである。O/R マッピングフレームワークを利用した場合、永続化されるクラス (以下、永続クラス) とデータベース・テーブルとの対応付け、永続クラスのフィールドとデータベース・テーブルのカラムとの対応付けを XML ドキュメントとして記述することで透過的な O/R マッピングが行われ、**インピーダンス・ミスマッチ**が解消される。

O/R マッピングフレームワークは XML ドキュメントに定義された永続クラスとデータベース・テーブルとの対応付けを読み込むことによって、永続化される Java オブジェクト (以下、永続オブジェクト) とテーブル構造との同期を実現する。RDB へアクセスするための SQL 文やフレームワーク内部で自動的に発行されるので、 O/R マッピングフレームワークを用いたアプリケーションはデータベースを全く意識することなくシステムの開発を行うことが出来る。また、これによって特定のデータベースへの依存性も排除出来る。さらに、データベース・スキーマが変更された場合にも XML ドキュメントのみを変更することで対処することが可能である。

2.1.3　先読みの必要性

O/R マッピングフレームワークを用いることよって**インピーダンス・ミスマッチ**を考慮せずに Java アプリケーションから RDB にアクセスすることが可能になった。しかし、データベースへのアクセスは非常に時間のかかる処理である。これは、大規模な Web アプリケーションでは多くの場合アプリーションとデータベースは別々のサーバにあるためである。このような場合、データベースサーバへの SQL の送信・得られるデータの受信には、通信のためのオーバーヘッドが生じる。また、データベースにアクセスするためには、まず適当な SQL を O/R マッピングフレームワーク内で作成し、その SQL をデータベースに送信、得られたデータを永続オブジェクトに変換するという作業が必要となる。

以上の理由から RDB へのアクセスは出来るだけ効率よくおこなう必要がある。効率よくアクセスする一つの方法として先読み技術がある。先読みとは予め必要になるとわかっているデータ、または必要となる可能性が高いデータをデータベースから一括して取得する技術である。

例えば、図 2.1 のような永続クラスを考える。Proceeding オブジェクトが getPapers() メソッドによって Paper オブジェクトのリストを取得した場合、先読みを行わない場合には、取得したリストの要素である各 Paper オブジェクトにアクセスがあるたびごとに Select 文が発行され、データベースにアクセスすることになる。これでは非常に効率が悪い。予め全ての Paper オブジェクトが必要になることがわかっているのなら、全ての Paper オブジェクトを一括してデータベースから取得してキャッシュしておけば、 Paper オブジェクトが取得されるたびにデータベースにアクセスする必要性はなくなり、パフォーマンスを向上することが出来る。

```
class Proceeding {
    int conferece-name;
    int year;
    List papaerList;
    List getPapers() {
        return paperList;
    }
}

class Paper{
    int name;
    byte[] pdffile;
    byte[] psfile;
}
```

図 2.1: 永続クラスの例

2.2　アプリケーション依存のマッピングの困難さ

しかし、既存の O/R マッピングフレームワークの多くは、アプリケーションのクラスと RDB とテーブルとの自動的なマッピングをおこなうために、固定的なマッピングを行っている。固定的なマッピングとは、DB のあるテーブルのデータが必要となったとき、そのテーブル内の関連する全てのデータおよび、そのテーブルと関連付けられている他のテーブル内の全てのデータを一括して取得 (先読み) し、オブジェクトのマッピングするというものである。つまり、 SQL 発行のタイミングはマッピングのための XML ドキュメントのみに基づいてフレームワークが管理する。一部のマッピングを遅延させることも可能ではあるが、そのような遅延の指定も XML ドキュメントによるデータベース・テーブル間の静的な依存関係による指定にとどまっている。

このような固定的なマッピングでは、アプリケーションの文脈に応じてマッピングの変更をおこなうことが出来ないため、不必要な先読みによって計算機資源を浪費してしまう危険性がある。例えば、以下の例を考えてみる。

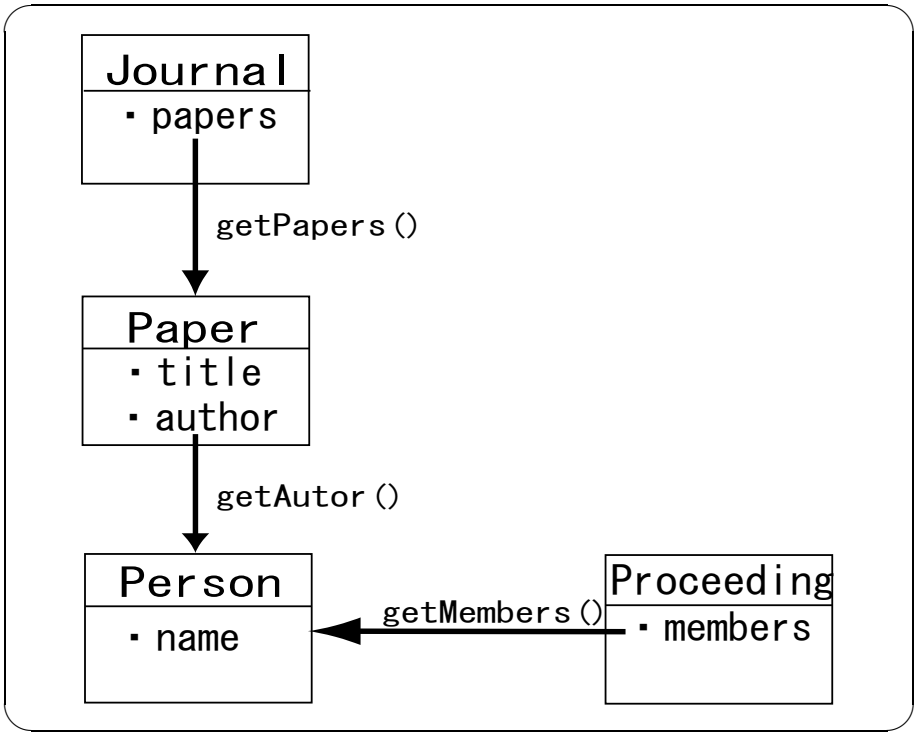

図 2.2: Hibernate の構造

図 2.2 は Journal 、 Paper 、 Person 、 Proceeding というデータベー

スに永続化されたクラスを表す。また、各クラスはそれぞれフィールドによって関係づけられている。これはデータベース内の各クラスに対応するテーブル同士がそれぞれ関連づけられていることを表す。今、Journal (論文誌) から Paper (論文) が取得され、その Paper から Person (その論文の著者) が取得された場合を考える。この場合、論文が取得され、その後、その論文の著者が取得されたのであるから、今後必要となるのは、その著者が書いた論文なのではないかと推測することが出来る。しかし、Proceeding (学会)　から Person (会員) が取得された場合には、今後必要となるのは、その会員が書いた論文ではなくて、その学会にに属する会員たちのデータだと推測できる。

このように、同じ Person のデータが必要となった場合でも、その後必要となるであろうデータは文脈によって異なってくる。そのため、固定的な先読みでは、不必要なデータまで先読みしてしまう危険性が避けられない。無駄なデータを先読みしないようにするためには、全てのデータを遅延するという方法も考えられる。しかし、そのような方法を取ってしまうと、今度は、データベースアクセスが過剰に増加してしまうという問題点もある。このような、先読みを多用してしまうと計算機資源の浪費につながり、先読みを行わないとデータベースアクセスの頻発を招くというトレードオフによって、既存のマッピングフレームワークでは効率のよい O/R マッピングを行えないという問題があった。

効率のよいマッピングを実現するために、O/R マッピングフレームワークを用いずに **JDBC** を用いて SQL を直書きするという方法も考えられる。しかし、このような方法を取ってしまうと、先に述べたインピーダンスミスマッチを開発者自身で解決しなければならないだけでなく、データベースアクセスのためのコードがアプリケーション内に散在してしまうという問題が生じてしまう。

データベースアクセスのための SQL など特定の処理がいくつものモジュール間にまたがってしまうことを**関心事の横断**という。関心事が横断してしまうとプログラムの可読性が低下していしまうだけでなく、プログラムの保守性も損ねてしまうことになる。この問題はアプリケーションの規模が大きくなればなるほど深刻な問題である。例えば、 O/R マッピングの記述はアプリケーションのパフォーマンスを左右する重要な機能であるため、アプリケーションが完成した後でもパフォーマンス改善のために幾度となく、マッピングの変更が行われると考えられる。マッピングを変更するたびごとに、プログラム全体に散らばってしまっているマッピングのコードを探し出し、変更するのは大変効率が悪い。

2.3　既存の O/R マッピングフレームワーク

以下、既存の O/R マッピングフレームワークである **Entity Bean**[2]、**Hibetnate**[4]、**Cayenne**[5] について詳しく述べる。これらのマッピングフレームワークは、それぞれ優れた機能を提供しているが、いずれも固定的なマッピングを行っている。そのため、アプリケーションの文脈に応じた先読みの変更をサポートしていない。

2.3.1　Entity Bean

Entity Bean とは **EJB** [2] の機能の一部であり、永続オブジェクトと RDB とのマッピングを **EJB** コンテナによって行う機能である。**EJB** は **J2EE** の仕様の一部であり、アプリケーションのビジネスロジックを担う分散型のオブジェクト技術である。

まず、**J2EE** について簡単に述べる。**J2EE** は **Java 2 Plattform Enterprise Edition** の略称であり、**Sun Microsystems** が公開している抽象 Java プラットフォームのことである。**J2EE** には Java の標準プラットフォームである **Java 2 Standard Edition**(**J2SE**) にベースにエンタープライズ向けの各種サーバサイドシステムに必要な機能を提供する API の仕様やバージョン、それらの組み合わせを指定したものが追加されている。J2EE に含まれる主な要素としては、

Servlet クライアントからリクエストを受け取ると、データベースなどと連携して動的に HTML や XML ドキュメントを生成してクライアントに返信する仕組み。

JSP 静的 HTML ドキュメントに Java 言語で記述されたプログラムを埋め込み、動的なコンテンツ生成をための仕組み。

EJB コンテナ管理によるデータの永続性を実現する仕組み。

がある。

この **EJB** の機能の一部に **Entity Bean** がある。**Entity Bean** とはデータベース内のテーブル・レコードを Java オブジェクトとして扱えるようにするための技術でる。テーブル・レコードは Entity Bean としてデータベースから取り出され、また Entity Bean の値が変更されるとテーブルレコードの値も変更される。

Entity Bean には、**CMP** と **BMP** の 2 種類の Entity Bean が存在する。どちらの Entity Bean でもデータベースと Bean 間のデータのロード・セーブのタイミングやトランザクション管理は **EJB** コンテナが管理する。両者の違いは、**BMP** ではアプリケーション開発者が SQL 文を

JDBC を用いて実装するのに対して、**CMP** では **EJB** コンテナが SQL の発行まで自動的に行ってくれるので開発者は **JDBC** を一切意識しなくてよいという点である。

BMP では 2.1 節で述べたような**インピーダンス・ミスマッチ**の問題が生じてしまうことになる。以下 **CMP Entity Bean** について述べる。

CMP Entity Bean を用いることで永続オブジェクトと RDB とのマッピングを **EJB** コンテナが行ってくれる。**EJB** は基本的にローカルインターフェース、ローカルホームインターフェース、 Bean クラスという 3 つの Java クラスから構成される。永続オブジェクトと RDB とのマッピングなどの定義は Deployment Descriptor と呼ばれる XML ドキュメントに記述する。これらのクラスファイルと XML ドキュメントを **J2EE** サーバにデプロイすることによって **EJB** コンテナが RDB と永続クラス間のデータのロード・セーブのタイミングやトランザクション管理、 SQL の発行を行ってくれる。

RDB とのマッピング

先ほど述べたように永続オブジェクトと RDB とのマッピングは XML ドキュメントに定義する。XML ドキュメントには、複数の永続クラスを一つの **EJB** にまとめ、その **EJB** の特徴を宣言的に定義していく。Entity Bean を構成するクラスやその特徴を示す属性の宣言を記述する。<Entity> 要素に記述する。<local-home>、<local>、<ejb-class> の各要素にはそれぞれローカルホームインターフェース、ローカルインターフェース、Bean クラスを指定する。そして <cmp-field> に永続クラスのフィールドを定義していく。**Primary Key** に対応しているフィールドは <primkey-field> 要素に定義する。

永続クラス間の **Relationship** もこの XML ドキュメントに記述する。**Relationship** には一対一、多対一、多対多といったような多重度の指定と片方向、双方向といった方向性の指定をおこなう。**Relationship** の定義は <relationship> 要素内で定義する。

```
...
<ejb-jar>
  <enterprise-beans>
    <entity>
      <ejb-name>Proceeding</ejb-name>
      <local-home>entity.ejb.ProceedingLocalHome</local-home>
      <local>entity.ejb.ProceedingLocal</local>
      <ejb-class>entity.ejb.ProceedingBena</ejb-class>
      ...
      <abstract-schema-name>Proceeding</abstract-schema-name>
      <cmp-field><field-name>proceeding-id</field-name></cmp-field>
      <cmp-field><field-name>conference-name</field-name></cmp-field>
      <cmp-field><field-name>year</field-name></cmp-field>
      <primkey-field>proceeding-id</primkey-field>
    </entity>
  </enterprise-beans>

  <relationships>
    <ejb-relationship>
      <ejb-relation-name>Proceeding-to-Paper</ejb-relation-name>
      ...
      <multiplicity>Many</multiplicity>
      <relationship-role-source>
        <ejb-name>Paper<ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>cid</cmr-field-name>
      <cmr-field>
    </ejb-relationshp>
    ....
  </relationships>
</ejb-jar>
```

図 2.3: CMP Entity Bean での RDB とのマッピングの例

2.3.2　Hibernate

Hiberante[4] は Gavin King 氏を中心としたチームが開発している O/R マッピングフレームワークであり、現在最もよく利用されている O/R マッピングフレームワークの一つである。Hibernate の特徴としては以下のものがある。

- **HQL**(**Hibernate Query Language**) という SQL にオブジェクト指向的拡張を施したクエリ言語が容易されている。**HQL** は構文は SQL に非常に類似した形になっているが、完全なオブジェクト指向言語になっており、継承・ポリモーフィズム・関連といった概念に対応している。HQL を用いることによって DB に格納されているデータの柔軟な検索が可能である。以下にその概要を示す。

+ FROM clause
  指定されたオブジェクトをすべて取得する。
  - FROM <Cat>, <Dog>で複数のクラスの表の直積を
    得ることが出来る。

+ SELECT clause
  どのオブジェクトやプロパティを取得するかを指定出来る。

+ Joins
  HQL ではマッピングファイルで関連づけられた table 同士は
  自動的に結合されて取得されるが、Join 句を用いれば
  マッピングファイルに結合記述のないオブジェクト
  同士を結合することも可能。

+ Aggregate functions
  sum(...), min(...), max(...) などの関数を用いる
  ことが出来る。

+ Polymorphic クエリ
  From clause は指定されたクラスのサブクラスに対応する
  表のデータも取得する。

+ WHERE clause
  SQL のように WHERE 句を用いて取得するリストの範囲を
  限定することが出来る。
  WHERE 句の中には =, !=, and など SQL で使われる
  演算子を用いることが出来る。

+ ORDER By clause, GROUP BY clause
  SQL と同様に ORDER By, GROUP BY なども用いることが出来る。

図 2.4: HQL の概要

HQL を用いると

```
Session session = getSession();
List papers =
    session.find("From paper as Paper paper where paper.id < 100);
```

のように容易に RDB から取得したいデータを限定できる。

- Java のリフレクションと **CGLIB**[6] による実行時のバイトコード生成処理を行うことによって、永続オブジェクトとなる Java のク

ラスをそのまま利用することが出来る。そのためデバッキングなどが容易に行うことが出来る。

- 多様なキャッシュ機能が提供されている。

  Session Cache Session Cache とは、作成したアプリケーションにアクセスするユーザごとに作成されるキャッシュである。**Hibernate** が動作する上で不可欠なキャッシュであり、常に有効になっている。DB からロードしたオブジェクトの参照を保持する。

  Second Level Cache Second Level Cache とは、セッションファクトリ単位またはクラスタ環境でのキャッシュである。Second Level Cache はデフォルトでは有効になっていないが、頻繁にロードされる永続オブジェクトや永続コレクションに対し設定することによってパフォーマンスをよくすることが可能になる。また、read-only, readwrite などのタグをマッピングファイルに記述することによってどのようなオブジェクトに対して Second Level Cache を適用するかを指定することも可能である。

  Query Cache Query Cache とはクエリをキーとしてその結果を Second Level Cache に格納するためのキャッシュである。頻繁に実行されるクエリに対してこのキャッシュを適用することによってパフォーマンスを向上させることが出来る。デフォルトでは無効になっている。

RDB とのマッピング

永続オブジェクトと RDB との対応付けはユーザが記述した XML ドキュメントを読み込むことで行われる。マッピングはテーブル定義ではなく、 Java の永続オブジェクト定義に基づいて構築される。この XML ドキュメントは主に以下の要素から構成される。

class 永続クラスまたは永続インターフェースの Java クラス名、そのデータベーステーブルの名前、遅延初期化 (Lazy Initialization) など、永続オブジェクトの属性に関する情報を定義する。

id マッピングする永続クラスのデータベース・テーブルの Primary Key を定義する。

property 永続クラスが持つプロパティ名前、マッピングされたデータベース・テーブル・カラムの名前、プロパティの型の情報を定義する。

many-to-one, many-to-many, one-to-one 他の永続クラスへの関連を記述する。many-to-one は多対一、many-to-many は多対多、one-to-one は多対一の関連となる。関連するクラスに対するプロパティ名、クラス名などを定義する。

```
<?xml version="1.0" ?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping>
    <class name="entity.Proceeding" table="PROCEEDING" lazy="true">
        <id name="proceeding-id" column="PROCEEDING_ID" type="int" >
            <generator class="assigned" />
        </id>

        <property name="conference-name"
                type="string" column="CONFERENCE_NAME" />
        <property name="year" type="int" column="YEAR"/>

        <set name="papers" lazy="true">
            <key ><column name="proceeding-id" /></key>
            <one-to-many class="entity.Paper" />
        </set>
    </class>
</hibernate-mapping>
```

図 2.5: Hibernate での RDB とのマッピングの例

Lazy Load

Hibernate では永続オブジェクトが RDB からロードされる際、マッピングの定義を記述する XML ドキュメントでその永続オブジェクトに関連付けられた永続コレクションの先読みを実行する。しかし、マッピングを定義する際に lazy タグによって遅延初期化の設定がされている永続コレクションに関しては先読みは実行されない。これを遅延初期化という。Hibernate では遅延初期化実現するためにアプリケーションで用いられている java.util.Map、 java.util.Set、 java.util.List のといったコレクション型のインスタンスを Hiberante 内部で永続化機能を実装したインスタンスへ変換する処理を行っている。変換されたコレクション型のクラスはその要素にアクセスがあった時点でコレクション内の全ての要素を RDB からロードするように実装されている。

```
public class List extends ODMGCollection
       implements java.util.List, DList, DArray {
    private java.util.List list;

     public final void read() {
         // RDB からオブジェクトをロード
         initialize();
     }

     public int size() {
         read();
         return list.size();
     }

     public Object get(int index) {
         read();
         return list.get(index);
     }

     .....
}
```

図 2.6: Hibernate 内部の List の実装

Hiberante の構造

Hibernate 内では、 SessionFactory クラスが永続クラスとデータベースとのマッピングを管理する。Hibernate を用いてデータベースにアクセスするには、 SessionFactory クラスから Session オブジェクトを取得して、その Session オブジェクトを通してデータベースから永続オブジェクトを取得する。Session オブジェクトはフィールドとして CollectionEntry、EntityEntry クラスの集合を保持していて、この CollectionEntry、EntityEntry オブジェクトが Session オブジェクトによって取得された永続オブジェクトを管理している。

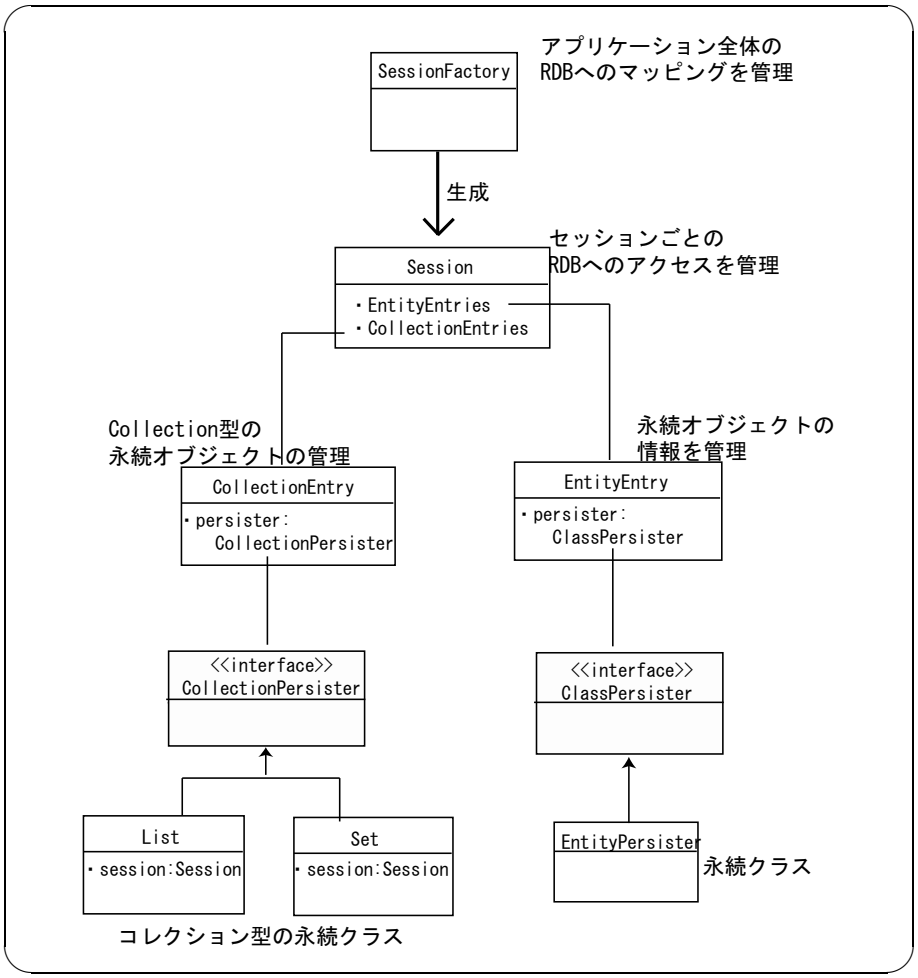

図 2.7: Hibernate の構造

2.3.3　Cayenne

Cayenne[5] は ObjectStyle グループによって提供されている O/R マッピングフレームワークである。豊富な O/R マッピング機能を持ち、高性能なパフォーマンスが得られる。Cayenne の特徴を以下に述べる。

- Cayenne Modeler という単一の GUI ベースのツールによって永続クラスの定義から DB スキーマの設計、マッピング定義を記述する XML ドキュメントの生成までの作業を行うことが可能である。
- Cayenne で使用される永続クラスは DataObject クラスを継承している。この DataObject クラスに RDB へのアクセスに関する処理を行う。

- Cayenne には先読みの機能が用意されていない。したがって、先読みを行いたいデータはアプリケーション内で指定してやる必要がある。これについては後で詳しく述べる。

- Cayenne ではオブジェクト指向ベースの RDB へのアクセスのための言語 Expression が提供されている。Expression は主に経路を表す Path Expression と演算を表す Conditional Expression に分類できる。Path Expression には永続オブジェクトのプロパティをによって RDB から取得するデータを限定する Object Path Expression と RDB のテーブル・カラムによって取得するデータを限定する Database Path Expression がある。また、 Cayenne では文字列を Expression に変換するパーサも提供している。

  Expression は ExpressionFacotry クラスから生成される。ExpressionFactory クラスには以下のようなメソッドが提供されている。

  - matchExp(String, Object)
    ”equal to” を表す Expression を生成する。
  - lessExp(String, Object)
    ”less than” を表す Expression を生成する。
  - greaterExp(String, Object)
    ”greater than” を表す Expression を生成する。
  - betweenExp(Steing, Object, Object)
    ”between” を表す Expression を生成する。

  また、生成された Expression を連結するメソッドも用意されている。

  andExp 二つの Expresion の論理積を表す Expression を返す。

  orExp 二つの Expresion の論理和を表す Expression を返す。

  joinExp 複数の Expression を論理和または論理積で結合した Expression を返す。

  記述は以下のようになる。

```
Expresion exp1 = ExpresionFactory.lessExp("id", 50);
Expresison exp2 =
        ExpresionFacory.lessExp("pdffile.size, 50);
exp = exp1.andExp(exp2);
Query query = new SelectQuery(Paper.class, exp);
context.performQuery(query);
```

図 2.8: Expression を用いた RDB へのアクセス例

RDB とのマッピング

Cayenne も他の O/R マッピングフレームと同様に永続オブジェクトと RDB・テーブルとの対応付け、永続オブジェクトのフィールドとテーブルのカラムとの対応付けはユーザが記述した XML ドキュメントを読み込むことによって行う。ただし、ユーザはこの XML ドキュメントを用意されている GUI ベースのツール Cayenne Modeler によって作成することが出来る。

```
<?xml version="1.0" encoding="utf-8"?>
<data-map project-version="1.1">
  <property name="defaultPackage" value="entity"/>
  <property name="defaultSchema" value="public"/>
  <db-entity name="proceeding" schema="public">
    <db-attribute name="conference_name" type="VARCHAR" length="30"/>
    <db-attribute name="year" type="INTEGER" length="10"/>
    <db-attribute name="proceedingid" type="INTEGER"
        isPrimaryKey="true" isMandatory="true" length="4"/>
  </db-entity>

  <db-entity name="paper" schema="public">
    ...
    <db-attribute name="cid" type="INTEGER" length="10"/>
    <db-attribute name="paperid" type="INTEGER"
        isPrimaryKey="true" isMandatory="true" length="4"/>
  </db-entity>
  <obj-entity name="Proceeding"
        className="entity.Proceeding" dbEntityName="proceeding">
    <obj-attribute name="conference-name"
         type="java.lang.String" db-attribute-path="conference_name"/>
    <obj-attribute name="year" type="java.lang.Integer"
                                          db-attribute-path="year"/>
    <obj-attribute name="proceeding-id"
         type="java.lang.Integer" db-attribute-path="proceedingid"/>
  </obj-entity>
  <obj-entity name="Paper" className="entity.Paper" dbEntityName="paper">
    ...
    <obj-attribute name="paperid"
         type="java.lang.Integer" db-attribute-path="paperid"/>
  </obj-entity>
  <db-relationship name="toPaper" source="proceeding"
                                        target="paper" toMany="true">
    <db-attribute-pair source="proceedingid" target="cid"/>
  </db-relationship>
  ...
</data-map>
```

図 2.9: Cayenne Modeler によって生成された XML ドキュメント

先読みの記述

Cayenne では先読みの記述をマッピングのための XML ドキュメントに記述するのではなく、アプリケーション内でクエリ言語によってデータベースにアクセスする際に指定することが出来る。例えば、

```
Context context = ...;
Query query = new SelectQuery(Proceeding.class);
query.addPrefetch("toPaper");
List ProceedingList = context.performQuery(query);
```

のような記述をおこなうと、 Cayenne 内部で Proceeding オブジェクトを取得するための Select 文と Paper オブジェクトを取得するための Select 文が投げられ、 RDB から取得された Paper オブジェクトは適当な Proceeding オブジェクトにマッピングされる。このような記述をおこなうことによって、 ProceedingList 内の要素である Proceeding オブジェクトが Paper オブジェクトを取得するたびごとに Select 文が投げられる N+1 問題を解決することが出来る。ただし、このような先読みの記述はアプリケーションから明示的にクエリを投げる場合にしか適用出来ない。

Cayenne の構造

Cayenne でも Hiberante と同様にフレームワーク内部で自動的に Select 文を発行できるように、アプリケーション内で用いられる List クラスを Cayenne 独自の実装のものに置き換えている。Caynenne の詳しい構造は実装の章で述べる。

第3章　アプリケーション依存の先読みが可能なO/Rマッピングフレームワーク

本研究では、**アスペクト指向**プログラミングに基づいた O/R マッピング (先読み) 記述を支援する Java 向けの O/R マッピングフレームワークの設計・実装を行った。以下、本 O/R マッピングフレームワークについて述べる。

3.1　特徴

この O/R マッピングフレームワークは**アスペクト指向**プログラミング (**AOP**) を用いることによって、アプリケーションと先読みの記述を分離しながら、アプリケーション依存の文脈に依存した粒度の細かな先読みを実現する O/R マッピングフレームワークである。本フレームワークは、既存の O/R マッピングフレームワークである **Cayenne**[5] を拡張することによって実装されている。

特徴としては以下の点が上げられる。

- アプリケーション内ではデータベースを意識したプログラミングをする必要がない。これは **EJB** の **CMP Entity Bean** などと同様である。
- **AOP** を用いることによってアプリケーションのモジュール化を阻害することなく、アプリケーション依存の先読みを可能にした。また、アスペクト内では、先読みを実行する位置のみを指定するのではなく、指定したオブジェクトがどのようなデータベースアクセスを辿ってきたかによって先読み指定の変更を行えるように設計されている。

  先読みの記述は永続オブジェクトが取得された経路に依存する場合が多いと考えられる。例えば、同じ Person オブジェクトのプロパティ name が取得された際にも、その Person オブジェクトが Proceeding オブジェクトから取得されたものなのか、

```
Session session = getSession();
Proceeding proceeding =
        session.load(Proceeding.class, new Integer(1);
List members = proceeding.getMembers();
Person memeber = (Person)members.get(2);
String name = member.getName();
...
```

その Person オブジェクトが Paper オブジェクトから取得されたものなのか、

```
Session session = getSession();
Journal journal =
    session.laod(Journal.class, new Integer(5);
List papers = journal.getPapers();
Paper paper = (Paper)papers.get(3);
Person author = paper.getAuthor();
String name = author.getName();
...
```

によって今後行われるであろう処理は異なる。このような状況の違いに対処出来るように、 this ポイントカットを拡張して永続オブジェクトが取得された経路の指定が出来るようにした。

- 先読みを実行するタイミングの指定、先読みの内容は**XPath**を用いて指定が可能を行う。SQL などのクエリ言語は、本来データベーステーブルからデータを取得することを目的として設計されている言語である。そのため、先読みなどの O/R マッピングの記述を SQL を用いて行おうとすると、テーブル名やカラム名の記述が必要となり、記述が非常に煩雑になってしまう場合がある。本フレームワークでは、**XPath**を用いることによって複雑な先読みを直感的かつ容易に記述することを可能にした。

- データベース・テーブルの行・列単位の細かなマッピング (先読み) が可能である。既存の O/R マッピングフレームワークでは、テーブル単位の大雑把な先読みしかサポートしていなかった。このような大雑把な先読みでは、テーブルのレコード数が膨大な場合、またはテーブルレコードに大容量のデータが格納されている場合には、本来は不必要なデータを大量に取得することによってメモリ効率が低下してしまい、しいてはアプリケーションのパフォーマンスの低下につながる恐れがある。しかし、単純にプロパティ単位のマッピングを行ったのでは、データベースへの過剰なアクセスを引き起こ

してしまう。本フレームワークでは、プロパティ単位でのマッピングを行い、かつアプリケーションの文脈に依存した適切な先読みを行えるライブラリを提供することによって、効率のよい　O/R マッピングをサポートしている。

- 先読みを非同期行うことが可能である。大規模な Web アプリケーションではデータベースが利用されている可能性が高い。このような場合、アプリケーション実行時にクライアントの Think Time が発生すると考えられる。この Think Time 時に先読みの実行を可能にした。

3.2　先読み記述の分離

アスペクト指向言語 **AspectJ**[8] を拡張することによって先読みの記述をアプリケーションと分離した。本フレームワークでは、マッピングの記述をアスペクトとして記述し、そのアスペクトをフレームワークに合成 (**weave**) することによって、文脈に応じたマッピングを行いながら、オブジェクトの透過的な永続化をサポートしている。ここではアスペクト指向プログラミングについての説明と、このフレームワークで用いた **AspectJ** について述べる。

3.2.1　アスペクト指向プログラミング

アスペクト指向プログラミング (**AOP**) とは、アプリケーション内の各モジュールにまたがってしまう処理、横断的関心事を分離するためのプログラミング技法のことをいう。

ソフトウェアを開発する際には、開発するシステムを小さな機能 (モジュール) ごとに分解し、各モジュールを組み合わせることで開発を行っていく。例えば、Java 言語では開発するシステムの機能をオブジェクトという単位で分解する。システム内の様々な機能は各オブジェクトが担当する。開発者は作成されたオブジェクトを組み合わせることによってシステム全体を構成していく。システム内の各機能は継承や抽象などのオブジェクト指向の概念を用いるによってカプセル化されるため、開発者は各オブジェクトの細かな設計を考慮することなく、保守性、拡張性、可読性の高いプログラムを記述することが出来るとされている。

しかし、オブジェクトによって全ての機能をカプセル化できるわけではない。場合によっては、ある種の機能のための処理群が複数のオブジェクト間に散らばってしまう可能性がある。このような処理群は**横断的関心事**と呼ばれ、単一のオブジェクトにモジュール化できないことが問題視され

ている。機能がモジュール間にまたがってしまうことによって、可読性が損なわれることは言うまでもない。さらに、横断的関心事を修正するためにはプログラム中の多くの箇所を修正する必要が生じるという問題点や新たなオブジェクトを作成する際に横断的関心事となっている機能について考慮しながら作成しなければならないという問題点がある。

AOP はこのような**横断的関心事**の問題を解決するための技法と言える。**AOP** では複数のモジュール間にまたがってしまう機能、つまり横断的関心事をオブジェクトとは別の側面から考慮し、それをアスペクトとしてモジュール化して扱えることが出来る。よくアスペクトとして扱われる処理としてはログ処理、同期制御、トランザクション処理などがある。

AOP には、このように基本モジュールであるオブジェクトと、オブジェクトによって分離しきれない機能を記述するアスペクトの二種類のモジュールが存在する。これら二つのモジュールを合成する作業を **weave** という。オブジェクト指向によって記述された元々のコードに、アスペクトで記述されたコードを **weave** することによってアスペクト内に分離して記述されていた処理を、複数のオブジェクトに同時に埋め込むことが出来る。

3.2.2　AspectJ

AspectJ とは汎用 **AOP** の一つであり、**Java** 言語を **AOP** の概念の取り入れて拡張した言語である。**AspectJ** は Java 言語に取って代わる言語という訳ではなく、Java 言語に付け加えることによってオブジェクト指向だけでは分離しきれない処理をモジュール化することを可能にするための言語である。

Java 言語ではクラス、インターフェースを定義していくことによってプログラムを構成してゆく。**AspectJ** ではクラス、インターフェースにアスペクトという概念が付け加えられる。**AspectJ** でのプログラミングではこのアスペクトにオブジェクト指向では分離しきれない処理群を記述する。アスペクト内に記述された処理群はコンパイル時に **weave** される。

以下では、**AsepctJ** を理解するのに必要な概念である。**Joinpoint**、**Pointcut**、**Advice**、**Aspect** について述べる。

Joinpoint

joinpoint とは Java プログラム内の演算でアスペクト内のコードを合成しうる個所のことである。アスペクト内に記述された処理群はプログラム内の任意の個所に合成できる訳ではなく、**AspectJ** で **joinpoint** とし

て定義されている個所にのみ合成することが出来る。

AspectJ で定義されている **joinpoint** は主に、メソッド呼び出し時点、メソッド実行時点、コンストラクタ呼び出し時点、コンストラクタ実行時点、フィールド参照時点、フィールド代入時点、インスタンス初期化時点などがある。

Pointcut

pointcut とは条件を指定することにより、プログラム内の全ての **joinpoint** 集合から必要な **jointpoint** の集合を抽出する作業のことをいう。**pointcut** によって抽出された **joinpoint** にアスペクト内で記述された処理を埋め込むことが出来る。

例えば、任意のクラスで定義されている、戻り値がvoid型、引数が(int, int)型であるようなメソッドdraw()が実行される時点を抽出したい場合には、次のように記述することが出来る。

$$execution(void * .draw(int, int)$$

ここでexecutionはメソッドの実行時点を抽出する **pointcut** である。

以下に、**AspectJ** によって定義されている主な **pointcut** を示す。

call メソッド、コンストラクタの呼び出し時点を抽出

execution メソッドの実行時点を抽出

get クラスフィールドまたはインスタンスフィールドの参照時点を抽出

set クラスフィールドまたはインスタンスフィールドの代入時点を抽出

initialization クラスのインスタンス生成時点を抽出

within 指定したクラス、インターフェース内の全てのjoinpointを抽出

this 指定したオブジェクトが処理の主体である全てのjoinpointを抽出

target 指定したオブジェクトが処理の対象である全てのjoinpointを抽出

args 指定した引数で実行されるメソッド、コンストラクタ内の全てのjoinpoint

cflow 指定したpointcutによって抽出される全てのjoinpointについて、各joinpointの開始と終了の間に発生する全てのjoinpontを抽出

if 指定した条件を満たす全てのjointpointを抽出

これらのpointcutを組み合わせて新たなpointcutを定義することも可能である。

例えば、Pointクラスのint型のフィールドxが参照された時点、または代入された時点を抽出するpointcutを定義したい場合、以下のように定義出来る。

$$pointcutaccess() : get(Point.x)||set(Point.y);$$

Advice

adviceとは**pointcut**によって抽出された**joinpoint**の集合において、実行したい処理を指示する機能のことである。

adviceには**before**、**after**、**aroud**の3種類がある。**before**は抽出された**joinpoint**集合が実行される直前に実行したい処理を埋め込み、**after**は**joinpoint**集合が実行された直後に処理を埋め込む、**around**は**joinpoint**を実行する代わりに記述された処理を実行する。

実行したい処理は{ }によって囲まれた**advice**のボディに記述する。例えば、

```
before call(void Point.draw(int, int) {
  System.out.println("before call Point.draw()");
}
```

のように**advice**を記述すると。Pointクラスの戻り値がvoid型であるメソッドdraw(int, int)が呼ばれる直前には必ず、

```
System.out.println("before call Point.draw()");
```

というコードが埋め込まれる。

Aspect

Aspectとは**pointcut**と**advice**との組み合わせを指定するモジュール単位のことである。**advice**として横断的関心事となっている処理群を記述し、**pointcut**によってその処理群をプログラム内のどの位置に埋め込むかを指定することが出来る。

3.3　XPath

本フレームワークでは、先読みの記述および先読を実行するタイミングを**XPath**(**XML Path Language**)[7]によって指定する。

XPath とは、**W3C** で勧告された XML ドキュメントの一部を参照するためのアドレッシング言語であり、現在は **XPath**1.0 が勧告として公開されている。**XPath** では、 XML ドキュメントを幾つかの種類のノードから構成される木とみなす。この XML ドキュメントを表す木構造をたどって文書内のあらゆる要素や属性にアクセスする手段を提供する。以下、**XPath** の簡単な仕様を述べる。

XPath では XML ドキュメントを以下のノードとしてモデル化する。

表 3.1: XPath で定義されているノード

ノード	内容
ルートノード	最上位ノードを表す。によって表現される。
要素ノード	XML の要素を表す。
テキストノード	開始タグと終了タグで挟まれた文字列データを表す。
属性ノード	名前空間を表す。
処理命令ノード	処理命令を表す。
コメントノード	コメントを表す。

XPath ではロケーションパスを用いてこれらのノードを区切り、軸とノードテスト、述語によってそのノードが何であるかを表現する。軸、ノードテスト、述語をまとめてロケーションステップという。ロケーションステップは

軸 :: ノードテスト [式]

または、

軸 :: ノードテスト

の形で表現される。

ロケーションパスとはロケーションステップとロケーションステップを / によって区切っていく記述方法で URL や UNIX のディレクトリ構造を表現する際によく利用される記方法である。ロケーションパスには、絶対表記と相対表記がある。絶対表記ではルートノードからの位置関係を記述し、相対表記ではカレントノードからの位置関係を記述する。

軸とはノードの方向を示すものである。主に以下のような軸がある。

表 3.2: XPath で定義されている軸

軸	内容
self	コンテキストノード自身の集合
child	コンテキストノードの子ノードの集合
parent	コンテキストノードの親ノードの集合
attribute	コンテキストノードの属性の集合

ノードテストとは選択するノードの型と名前を指定するものである。軸によって選択されたノードの集合にさらに条件をつける。例えば、「child::Proceeding」という記述はコンテキストノードの子ノードである Proceeding という名前のノードを指定したことになる。直接ノード名で指定する他に、以下のようなノードテストを使用することが出来る。

表 3.3: ノードテスト

ノードテスト	内容
text()	テキストノードを表す
node()	要素や属性を表す
*	軸の子ノードすべてを表す
comment()	コメントノードを表す
attribute	コンテキストノードの属性の集合

軸とノードテストは通常略記で示される。以下、主な略記を示す。

表 3.4: ノードテスト

ノードテスト	内容
.	self::node()
..	parent::node()
@	attribute::
軸を指定しない	child::node()

述語とは選択するノードの集合を、任意の式を使用して選別するものである。=、!=、>、>=、<、<=、or、and といった比較演算子や +、−、*、div といった数値演算子や XPath で定義されている関数を用いて、ノードを絞り込むことが出来る。

例えば、「Proceeding[@year = 2005]/Paper/@name」という記述はyearという属性の値が2005 である Proceding というノードの子ノードPaperの属性である name を指定たことになる。

3.4　仕様

本研究で実装したフレームワークでも既存のO/Rマッピングフレームワークと同様に永続クラスとデータベース・テーブルとの対応づけはXMLドキュメントを用いて行う。

そして、先読みの記述は**Aspect**内でおこなう。用意されているPrefetchAspectクラスを継承することで用意に先読みが記述できるようになっている。PrefetcAsepct クラスには以下のポイントカットとメソッドが定義されている。

表 3.5: PrefetchAspect で定義されているポイントカット

ポイントカット	内容
createQury	RDB にアクセスするためにクエリが作成される時点
load()	永続オブジェクトのプロパティを RDB からロードする時点
init()	永続オブジェクトのプロパティを RDB またはキャッシュから初期化する時点

表 3.6: PrefetchAspect で定義されているメソッド

メソッド	内容
setAllPropertyLoad()	RDB から永続オブジェクトをロードする際に、 その永続オブジェクトの全てのプロパティを ロードするように設定
setLoadProperty()	RDB から永続オブジェクトをロードする 際に、ロードするプロパティを設定
prefetch()	先読みの実行
prefetchAync()	先読みを非同期に実行

先読みを実行するタイミングはポイントカットによって行う。しかし、

AspectJ のポイントカットでは、先読みのコードを実行する位置しか指定出来ない。そのため、本フレームワークでは、this ポイントカットを拡張して、 DB アクセスの履歴を考慮した先読みのタイミングの指定を可能にした。本フレームワークでの this ポイントカットの仕様は以下のようになる。

- 記述方式
  this(< クラスまたはインターフェースのタイプ >)[XPath 文字列]
  または、
  this(< タイプ識別子 >)[XPath 文字列]

- 選択されるジョインポイント
  処理の主体側インスタンスのタイプが < ... > で指定されるクラスまたはインターフェースであり、かつ、**XPath** での記述とマッチしている全てのジョインポイント。

this ポイントカットの特定に用いる **XPath** の文法について述べる。ここで用いる文法は **W3C** によって勧告されている **XPath** の文法のサブセットである。

まず、ロケーションパスは永続クラス間の Relationship を区切っていくものとする。また、ノードは永続クラスまたは、永続クラスのプロパティであることにする。

元来 **XPath** では [ ] を用いて

$$//Proceeding[@year]/Paper/@name$$

のような記述が可能である。この記述は「属性 year をもつ Proceeding ノードの子ノードである Paper ノードの属性 name」という意味になるが、これを本研究での永続クラス間の Relationship にという観点から翻訳すると「 year というプロパティをもつ Proceeding オブジェクトに関連付けられている Paper オブジェクトの name プロパティ」という無意味な記述になってしまう。text() や last() 、position() のような関数も同様に意味を成さない。

このことは、ノードを永続クラスまたは永続クラスのプロパティに限定したためにおきる矛盾だと考えられる。このため、this ポイントカットを指定するする際に **XPath** の述語を用いる場合には、比較演算子または述語演算子を用いた限定のみ有効であることにした。

$$//Biblography/Proceeding[@year < 2005]/Paper/@name$$

のような記述は有効なものである。

以下、永続クラス間の関係を指定する **XPath** の仕様を示す。

表 3.7: XPath の仕様

記述	用途
//	間に任意個の永続クラスを含む永続クラス間の Relationship
.	現在指している永続オブジェクトを表す
..	現在指している永続オブジェクトの親オブジェクトを表す
@	永続クラスのプロパティを表す
*	任意の永続クラスまたは永続クラスのプロパティを表す
[]	永続クラスに関する条件の指定に用いる

この **XPath** の文法を用いると以下のように this ポイントカットの指定が出来る。this ポイントカットの記述例は以下のようになる。

- this[//Proceeding/@*]
  Proceeding オブジェクトの任意のフィールドがアクセスされた時点。AspectJ の this ポイントカットと等しい。

- this[//Biblography/Proceeding/@year]
  Biblography オブジェクトによって取得された Proceeding オブジェクトの year フィールドがアクセスされた時点

- this[//Biblography/Proceeding[year = 2005]/@conference-name]
  Biblography オブジェクトによって取得された year フィールドの値が 1 である Proceeding オブジェクトの conferece-name フィールドがアクセスされた時点

Prefetch アスペクトクラスに定義されている。 prefetch() メソッド、prefetchAsync() メソッドも同様にこの **XPath** の文法を用いて先読みの指定を行う。prefetch() メソッド、 pefetchAsync() メソッドは XPath 形式の文字列で指定されたデータを RDB から取得し、その後取得したデータを適当な永続オブジェクトにマップする。

この **XPath** を用いた先読みデータの指定を行うことで、オブジェクトレベルでの先読みの指定が可能となる。そのため、 SQL や **HQL** を用いる場合に比べて直感的かつ容易に先読みの記述が出来る。例えば、

$$prefetch(\text{"}./Proceeding/Paper/@pdffile)$$

を SQL を用いて記述しようとすると

```
SELECT pdffile FROM paper t0, proceeding t1, biblography t2
       WHERE t0.cid = t1.conferenceid AND t1.bib-id = t2.id
       AND (t2.bib-id like ?)
```

のようなデータベースのテーブル名、カラム名を意識した煩雑は記述をしなければならなくなる。

以下、簡単な prefetch() メソッドの記述例を示す。

- prefetch(”./@*”);
  thisJointPoint が表す永続オブジェクトの全てのプロパティを先読みする。

- prefetch(”../Paper/@*[self::pdffile and self::psfile]”);
  thisJoinPoint が表す永続オブジェクトを取得したオブジェクトに関連付けられている Paper オブジェクトの pdffile と psfile プロパティを取得する。

3.5　記述例

先読みの記述例とその記述による先読みの様子を以下に示す。

3.5.1　テーブル・レコードの先読み

Bibliography オブジェクトから取得された Proceeding オブジェクトの year フィールドがロードされる時点でその Proceeding オブジェクトの全てのフィールドを先読みをする例を示す。

```
after(Proceeding proceeding) :
        load() &&
        this(proceeding)[//Biblography/Proceeding/@year] {
    prefetch("./@*");
}
```

この記述は以下のような先読みを行う。

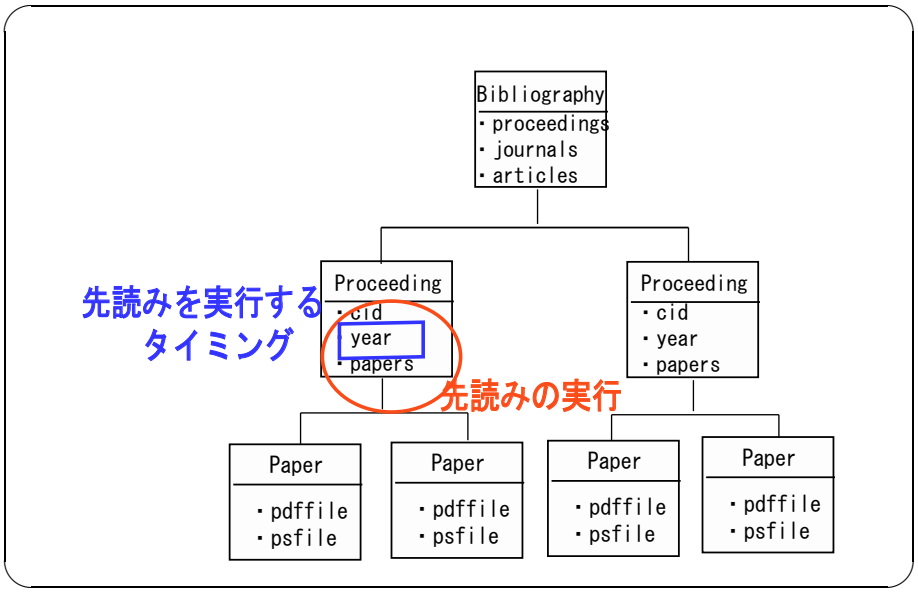

図 3.1: prefetch(”./@*”) による先読みの様子

3.5.2　テーブル・カラムの先読み

Bibliography オブジェクトから取得された Proceeding オブジェクトの year フィールドがロードされる時点でその Proceeding オブジェクトに関連付けられている全ての Paper オブジェクトの pdffile フィールドを先読みする例を示す。

```
after(Proceeding proceeding) :
        load() &&
        this(proceeding)[//Biblography/Proceeding/@year] {
    prefetch("./Paper/@pdffile");
}
```

この記述は以下のような先読みを行う。

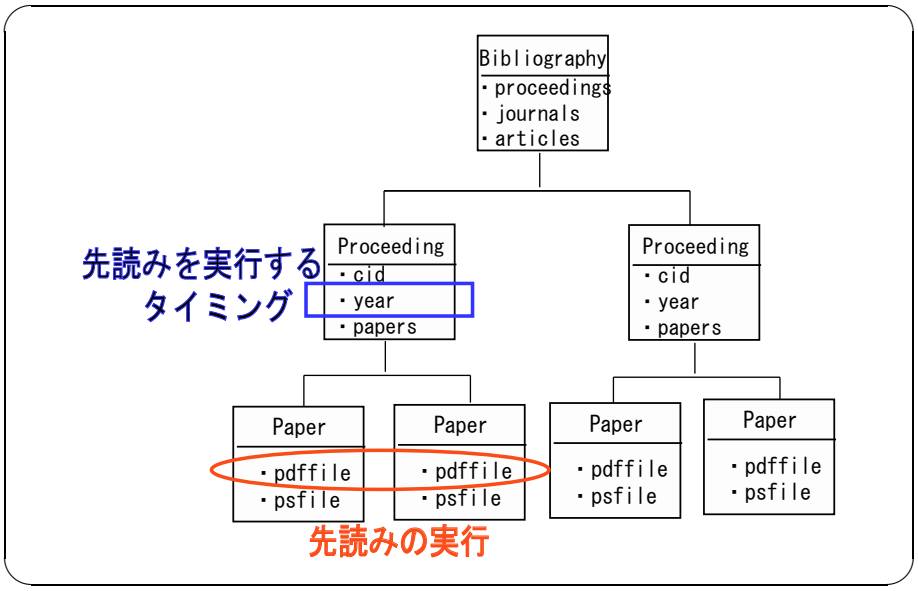

図 3.2: prefetch("../Proceeding/@year") による先読みの様子

3.5.3　Relationship の先読み

Bibliography オブジェクトの任意のフィールドがロードされる時点でその Bibliography オブジェクトに関連付けられている全ての Proceeding オブジェクト、その Proceeding オブジェクトに関連付けられている全ての Paper オブジェクトの任意のフィールドを先読みする例を示す。

```
after(Biblography bib) :
        load() &&
        this(proceeding)[//Biblography/@*}] {
    prefetch("./Proceeding/Paper/@*");
}
```

この記述は以下のような先読みを行う。

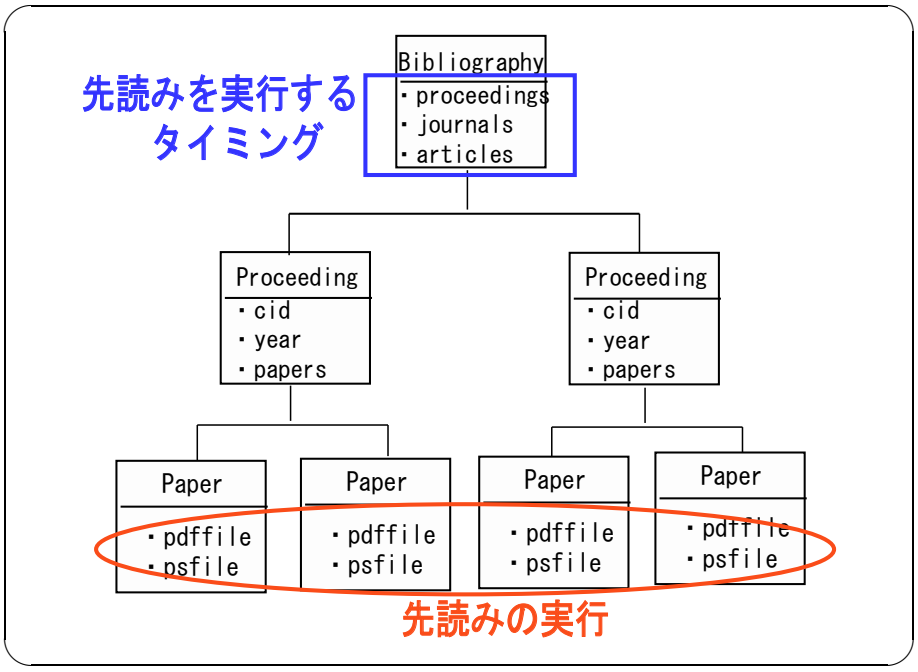

図 3.3: prefetch(”./Proceeding/Paper/@*”) による先読みの様子

第4章　実装

4.1　O/R マッピングフレームワーク Cayenne の拡張

アプリケーション依存の先読みが可能な O/R マッピングフレームワークの実装には第 2 章でのべた、既存の O/R マッピングフレームワークである **Cayenne**[5] を拡張することで実装を行った。以下、**Cayenne** の改造について述べる。

4.1.1　プロパティごとのマッピング

もともとの **Cayenne** の実装では、データベース・テーブルからオブジェクトへのマッピングを行う際には、そのテーブル内の関連する全てのプロパティをオブジェクトにマッピングするよう実装されていた。しかし、実際にはマップされた永続オブジェクトの全てのプロパティがアプリケーションで使われるかどうは、アプリケーションの文脈に依存すると考えられる。そのため、テーブルレコードに容量の大きなデータが格納されている場合、不必要なデータを大量に読み込んでしまう結果、アプリケーションのパフォーマンスを低下させてしまう危険性があった。そのため、まずデータベースからの不必要なデータ読み込みを出来るだけ防ぐため、データベース・テーブルのプロパティごとのマッピングを行うよう変更した。具体的には、**Cayenne** 内の Select 文を発行しているクラスを拡張してして、データベース・テーブルからオブジェクトへデータを読み込む際には、そのテーブルの 主キー (Primary key) とオブジェクト間の関連付けのために必要となるプロパティのみを取得するように変更した。

Cayenne 内部では、アプリケーション内で明示的に **RDB** にアクセスがされた場合、または永続オブジェクトから暗示的に **RDB** にアクセスする場合の流れは以下のようになっている。

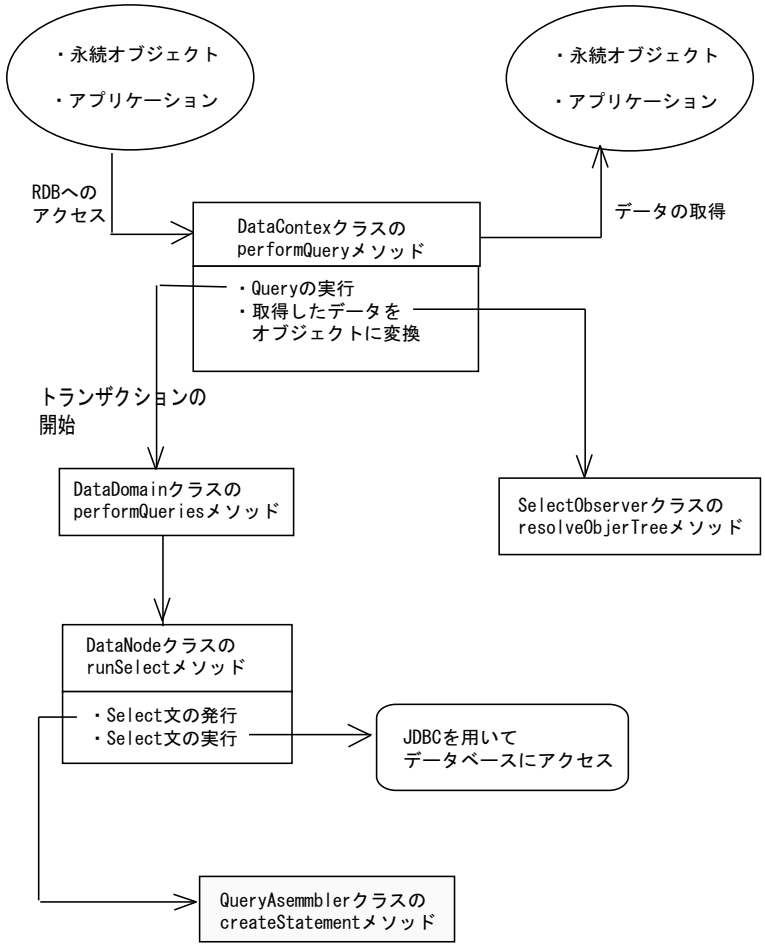

図 4.1: Cayenne 内で Select 文が発行されるまでの流れ

図 4.1 からわかるように SQL 文の作成は QueryAssembler クラスの createStatement() メソッドで行われる。 createStatement() メソッドは **RDB** にアクセスするためのクエリの文字列を作成するが、そのクエリに付け加えるプロパティを取得するメソッドが appendAttributes() メソッドである。まずはこのメソッドを変更することによって、指定したプロパティのみを取得する SQL を発行するように変更した。

```
private void appendAttributes(String[] properties) {
        DbEntity dbe = getRootDbEntity();
        SelectQuery q = getSelectQuery();
        .....
        EntiyInheritanceTree tree = getRootInheritanceTree();
        if(properties != null) {
          if(tree == null) {
               for(int i = 0; i < properties.length; i++) {
                 ObjAttribute oa = (ObjAttribute)
                     oe.getAttribute(properties[i]);
                 Iterator dbPathIterator =
                     oa.getDbPathIterator();
                 while (dbPathIterator.hasNext()) {
                   Object pathPart = dbPathIterator.next();
                   ...
                   if (pathPart instanceof DbAttribute) {
                     DbAttribute dbAttr =
                             (DbAttribute) pathPart;
                     columnList.add(dbAttr);
                   }
                 }
               }
          }
          else {
            Iterator attrs = tree.allAttributes().iterator();
            while (attrs.hasNext()) {
              ObjAttribute oa = (ObjAttribute) attrs.next();
             // properties を満たしている Attribute が存在すれば、
              // それらを columnList に加える。
            }
          }
       }
       else {
         // 全てのプロパティを ColumnList に加える。
       }

       // primary key, relationship keys の取得
       ...
    }
}
```

図 4.2: 拡張後の appendAttributes() メソッド

どのプロパティを取得するかの情報は org.objectSytle.cayenne.query.Query クラスに String 型の配列 properties を持たせることによって実装した。この properties フィールドが QueryAssembler クラスの createStatement() メソッドに渡されることによって必要なプロパティのみをデータベースか

ら取得する。

4.1.2　永続クラスの拡張

Cayenne において永続クラスは CayenneDataObject を継承する仕様になっている。永続クラスにはアプリケーション内のビジネスロジックが記述され、CayenneDataObject にはデータベースへのアクセスや **Relationship** のある永続オブジェクトへのアクセス方法などが記述されている。

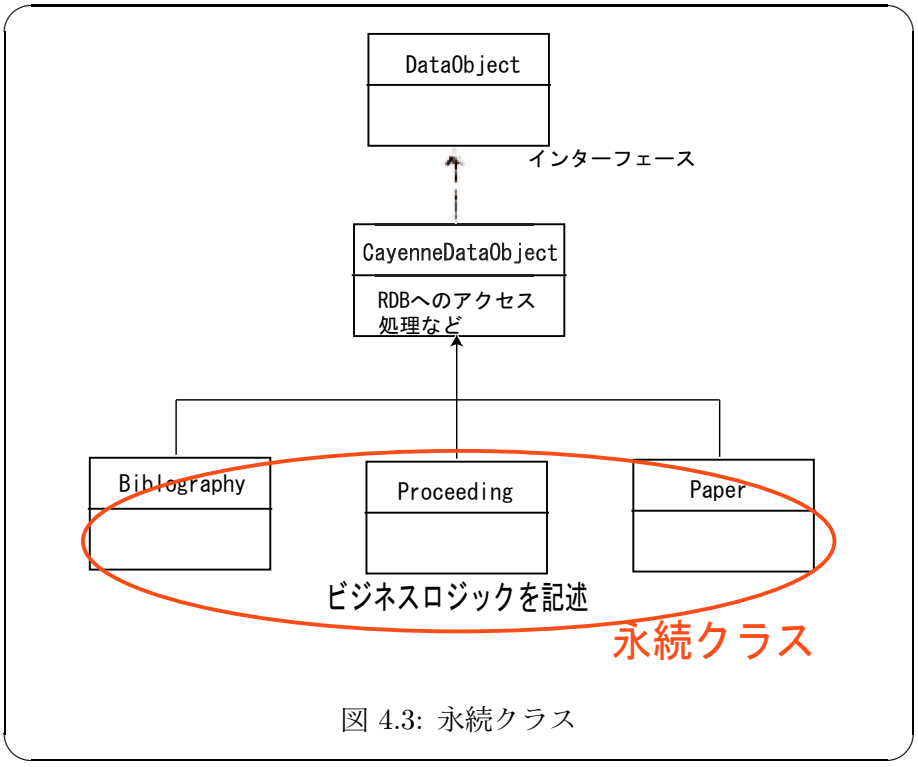

図 4.3: 永続クラス

プロパティごとのデータ取得

永続クラスを読み込む際に指定したプロパティのみをロードするように変更したことに伴い、永続クラス自身の変更も行った。もともと **Cayenne** では永続クラスがロードされる際にはそのクラスの全てのプロパティをロードする実装になっていたため、永続オブジェクトが自身のプロパティに関してデータベースにアクセスする必要はなかった。

しかし、プロパティごとのマッピングを可能にするとそれぞれの永続オブジェクトがプロパティが初期化されていない場合にはデータベースから

必要なプロパティを取得して、適切なオブジェクトにマップしてやらなければならない。よって、永続クラスのプロパティにアクセスがあった場合には、そのプロパティが初期化されているのかを判断して、初期化されていない場合にはまず、キャッシュにそのプロパティがのっていればキャッシュからそのプロパティを取得し、キャッシュにも乗っていないのであればデータベースに SQL を投げてそのプロパティを取得し、マッピングする実装に変更した。

どのオブジェクトから呼ばれたかの履歴を保持

永続オブジェクトがどの永続オブジェクトから取得されたものかがわかるように CayenneDataObject に parent フィールドを付け加えた。また、java.util.List クラス、 java.util.Iterator クラスなどのコレクション型のクラスの実装を独自に変更することによって、永続オブジェクトがコレクション型のオブジェクトの要素として取得された場合にも正しく履歴をたどることが可能なように変更した。parent フィールドをたどることによって永続オブジェクトがどのような経緯でデータベースからからロードされたかがわかるようになる。この情報は、先読みを実行するタイミングの指定に用いるほか、先読みによって取得したデータを適切なオブジェクトにマッピングする際に利用する。

先読みを実行する際の同期

さらに CayenneDataObject には先読みを実行する際に同期をとるためのメソッドを付け加えた。非同期に先読みを実行している場合、先読み用のスレッドがプロパティをロードしている最中に main スレッドでもそのプロパティをロードしてしまっては先読みの意味がない。そのため、非同期に先読みを実行する際には、先読みするプロパティに対してフラグをたて、main スレッド内で永続クラスのプロパティにアクセスする際には、まずそのフラグを見て、そのプロパティが先読み中であった場合には、先読みが完了するまで待機するという仕様にした。こうすることで二重に永続クラスのプロパティをロードロードしてしまう恐れがなくなる。

```
public class CayenneDataObject implements DataObject {
    //初期化されたプロパティが格納
    private Map values = new HashMap();

    synchronized void beforePrefetch(String[] propNames) {
      // 非同期な先読みのための前処理
    }

    synchronized void afterPrefetch(String propName) {
     // 非同期な先読みのための後処理
    }

    CayenneDataObject getParent() {
       // この永続オブジェクトを取得した親オブジェクトを返す
    }

    Object readProperty(String propName) {
      // 先読みが完了するまで待機
      while(...) {
        wait();
      }
      Object object = readPropertyDirectly(propName);
      ...
      // プロパティが初期化されていない場合の処理。
      if(object == null &&
          getPersistenceState() == PersistenceState.PARTIAL) {
        resolvePartial(propName);
        object = readPropertyDirectly(propName);
      }
    return object;
    }
}
```

図 4.4: CayenneDataObject に付け加えた主な機能

4.2　先読みを支援するライブラリの実装

先読みを実行するためのメソッドは引数に **XPath** 形式の文字列をとり、この文字列を読み込んで先読みを実行するように実装した。prefetch() メソッドでは引数として受け取った **XPath** 文字列を Lex クラスによって字句解析した後、Node クラスを用いて構文解析を行って構文解析で得られた PrefetchNode クラス、 RelationshipNode クラスを元に先読みを実行する。先読みの実行とは、Node クラスのデータを元にデータベースから指定されたデータを取得し、取得したデータを、適切なオブジェクトへ

マッピングしていく作業を指す。

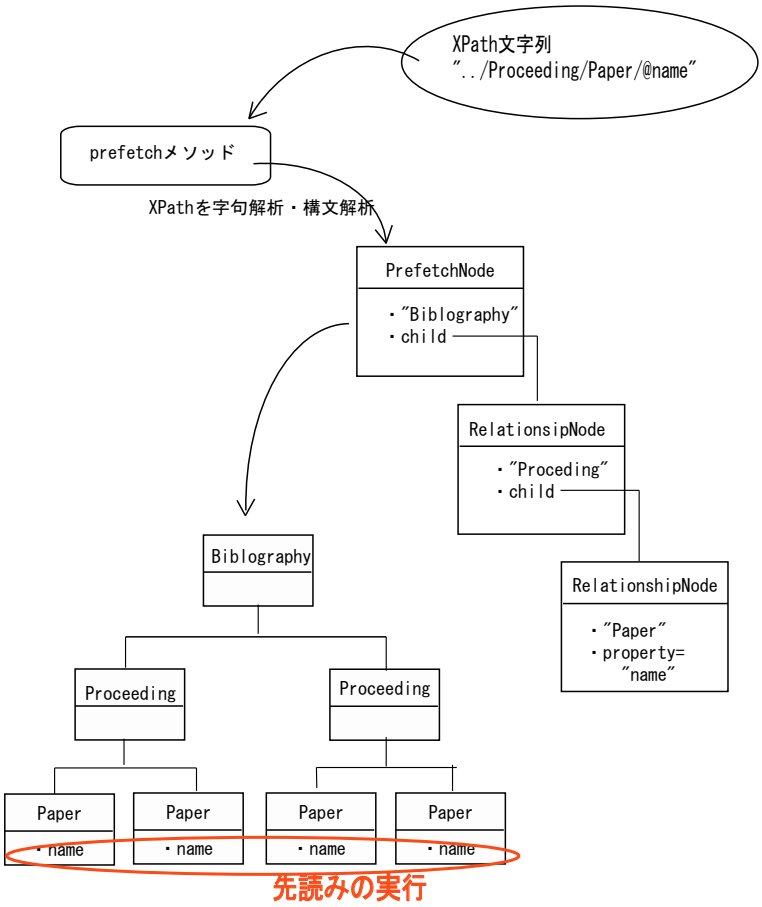

図 4.5: 先読み実行の流れ

第5章　実験

本フレームワークの有効性を検証するために、我々は実験を行った。実験内容としては、データベースを用いる単純なアプリケーションに対し、実行速度とメモリ消費量、データベースへのアクセス回数の比較を以下の先読みのポリシーで行った。

- テーブル単位でのデータの読み込みを行う
- プロパティ単位でのデータの読み込みを行い、先読みをしない
- プロパティ単位でのデータの読み込みを行い、かつアプリケーションの文脈ごとに適切な先読み行う

テーブル単位でデータを取得する O/R マッピングには既存の O/R マッピングフレームワークである **Cayeenne** を用いた。また、先読みを行わない O/R マッピング、文脈に依存した O/R マッピングには本研究で **Cayenne** を改良した O/R マッピングフレームワークを用いた。

実験は、以下の 3 つのアプリケーションに対して行った。

5.1　実験環境

データベース用のサーバとしては Sun Fire V60x Server (CPU Intel(R) Xeon(TM) CPU 3.06Hz × 2, 2GB, Linux 2.6.7) を用い、データベースには PostgreSQL 7.4.2 を用いた。クライアントとしては、(Pentium(R) 4 CPU 2.80GHz, 1.00GB, Microsoft Windows XP) から行った。LAN は 1000 BaseTX である。

5.2　取得したテーブルレコードの一部のプロパティしか利用しない場合が多いアプリケーション

実際のアプリケーションでは取得したテーブルレコードの全てのプロパティが利用されるとは限らない。そのため、レコードに容量の大きなデータが格納されている場合には、レコードを取得する際に、全てのプロパ

ティを取得していてはパフォーマンスを低下させる原因となる可能性がある。

この実験ではテーブルレコードの一部しか利用しなかった場合の、先読みポリシーの違いから来る性能の比較を行った。この実験に用いた本フレームワークのマッピングのポリシーは文脈に応じて必要となるであろう列だけをを読み込むというものである。記述例は以下のようになる。

```
after(Paper paper) :
            load() &&
            this(paper)[//Paper/@*] {
    prefetch(paper, "//Paper[@paperid < 1000]/
                    @*[self::title and self::psfile]");
}
```

図 5.1: マッピングの記述例

取得するテーブルレコードは 8byte のデータが格納されているカラム 5 つと約 50byte のデータが格納されているカラム、約 6000byte のデータが格納されているカラム 2 つの計 8 カラムからなる。実験では、サイズ 1000 の永続オブジェクトのリストを取得し、それらすべてのオブジェクトに対して 6000byte のカラム 1 つを除いた 7 つのデータを取得する場合の、実行時間、RDB へのアクセス回数、メモリ使用量を比較した。結果は以下の通りである。

表 5.1: 一部のプロパティしか利用しない場合の比較

ポリシー	実行時間 (ms)	アクセス回数 (回)	メモリ量 (MB)
テーブル単位	1925	1	6200
先読みなし	13538	2000	3210
先読みあり	1872	3	3210

5.3　取得したテーブルの一部の行 (レコード) しか使用しない場合が多いアプリケーション

Google などの検索システムでは、検索条件を満たす大量のデータの中から順々にページに収まる単位づつデータを表示していく。このような場合、全てのテーブルレコードが利用されることは稀である。そのため、一度に全てのデータを取得するのは効率的とはいえない。

この実験では、テーブルの一部のレコードしか利用しなかった場合の、先読みのポリシーの違いから来る性能の比較を行った。この実験に用いた本フレームワークのマッピングのポリシーはテーブルの行を 100 行ずつに区切りオンデマンドにアクセスがあったときのみデータを読み込むというものである。記述例は以下のようになる。

```
after(Paper paper) :
            load() &&
            this(paper)[//Paper/@*] {
    int id = getInt("./@paperid");
    int min = 0;
    while(min + 100 <= id)
        min += 100;
    prefetch(paper, "//Paper[@paperid >= " + min +
                    " and @paperid < " + (min + 100) + "]/@*");
}
```

図 5.2: マッピングの記述例

取得する 1 カラムの大きさは約 12MB とし、サイズ 2000 の永続オブジェクトのリストを取得し、そのうち約 1/4 までのデータを取得する場合の、実行時間、RDB へのアクセス回数、メモリ使用量を比較した。結果は以下の通りである。

表 5.2: 一部のレコードしか利用しない場合の比較

ポリシー	実行時間 (ms)	アクセス回数 (回)	メモリ量 (MB)
テーブル単位	12866	1	124000
先読みなし	10480	1500	3380
先読みあり	1773	11	24000

5.4　ランダムにテーブルを辿り、不特定に取得したテーブルの全要素を利用する場合が多いアプリケーション

Web アプリケーションなどでは、ランダムにいくつものデータにアクセスし、自分の興味のあるデータに関してはそれらを逐一取得するという場合が考えられる。このような場合、どのデータを必要としているかの判断が難しいために、既存の O/R マッピングフレームワークでは、先読み

が難しかった。

この実験では、RDB のデータにランダムにアクセスし、そのうちのいくつかに関してはそのデータを全て取得する場合の、先読みのポリシーの違いから来る性能の比較を行った。この実験に用いた本フレームワークのマッピングのポリシーは、あるオブジェクトからテーブルに複数回アクセスがアクセスがあった場合に、そのテーブル内のそのオブジェクトに関するデータを全て読み込むというものである。記述例は以下のようになる。

```
int id;
after(Paper paper) :
            load() &&
            this(paper)[//Author/Paper/@*"] {
    int i = getInt("../@authorid");
    if(i == id) {
        prefetch(paper, "../Paper/@*");
    } else {
        prefetch(paper, "./@*");
        id = i;
    }
}
```

図 5.3: マッピングの記述例

使用するテーブルは、1 カラムのメモリ量が約 12MB のもの、約 0.5MB のもの、約 0.7MB のものとし、実行時間、RDB へのアクセス回数、メモリ使用量を比較した。結果は以下の通りである。

表 5.3: ランダムにデータにアクセスする場合の比較

ポリシー	実行時間 (ms)	アクセス回数 (回)	メモリ量 (MB)
テーブル単位	10500	100	10000
先読みなし	13100	564	980
先読みあり	6050	174	1200

5.5　考察

これらの実験からからテーブル単位の取得には大量のメモリが消費されていることがわかる。扱うレコード数が多い場合、または 1 カラムのデータ量が大きい場合ひは、メモリの消費量が原因でアプリケーションのパフォーマンスを大きく低下させてしまう可能性がある。また、先読み

を用いないと、RDB へのアクセスが著しく増加していることがわかる。RDB との通信速度に問題がある場合、または RDB への負荷が大きい場合には、RDB への過剰なアクセスによってアプリケーションのパフォーマンスを大きく低下させてしまう可能性がある。

一方、本フレームワークを用いて適切な先読みを行った場合には、実行速度、RDB へのアクセス回数、メモリ使用量において、優れたパフォーマンスを得られることが確認された。

第6章　まとめ

本研究では、アスペクト指向プログラミングに基づいた O/R マッピング (先読み) 記述を支援する Java 向けの O/R マッピングフレームワークの提案、実装を行った。本フレームワークを利用することで、マッピング記述はアプリケーション内に散在することなく、かつ柔軟にプログラミングできるようになった。

先読み記述のアプリケーションからの分離には Java 言語を拡張したアスペクト指向言語である AspectJ を利用した。しかし、AspectJ では、プログラム中の先読みの位置を指定することしか出来ない。そのため、XPath による記述に基づいて DB アクセスの履歴を辿る API を用意し、DB アクセスの履歴を考慮した先読みの指定を可能にした。

本フレームワークは、細かな先読み記述が可能になるように、既存の O/R マッピングフレームワークである Cayenne を改造したものである。改造に際しては、まず、プロパティごとにデータベース・テーブルをマッピングするよう変更し、テーブルからデータの取得の際には主キー (Primary Key) のみを取得するように変更した。このような O/R マッピングはデータベースへの過剰な SQL 発行を誘発するため敬遠されていたが、XPath に基づいた行・列単位の粒度の細かく柔軟な先読みが可能なライブラリを提供することで、効率のよい O/R マッピングをサポートした。また、先読みのタイミング、先読みデータの指定に XPath を用いることで、オブジェクトのレベルでの直感的かつ容易な先読みの記述が可能になった。

実験では、本フレームワークを用いて単純な先読みアルゴリズムを適用したアプリケーションで既存の O/R マッピングフレームワークの利用に比べて実行速度、メモリ消費、RDB アクセス回数の点で大きく性能を改善できることを確認した。

今後の課題

この後の課題としては以下の事柄があげられる。

- **現実的なアプリケーションでの実験**
  本論分では、非常に単純なアプリケーションでの実験を行った。しかし、フレームワークの性能を確かめるためには、より現実的なア

プリケーションでの実験が必要であると考えている。

- **より細かな先読みのタイミングの指定**
  本フレームワークでは、先読み記述とアプリケーションとの分離のために AspectJ を利用した。DB アクセスの履歴を辿ることによって、文脈に依存した先読みの指定を可能にした。しかし、効率のよりマッピングには、より細かな先読みの指定が必要であると考える。例えば、 java.util.Iterator からあるテーブルが取得された場合、 取得されたテーブルはすべて同等とプロパティを必要とすると考えられる。そのような状況での指定がより容易に行えるようにする必要がある。

- **文脈に応じたアプリケーション全体でのキャッシュのポリシーの変更**
  データベースを用いたアプリケーションのパフォーマンスをあげるためには、文脈に応じて、セッションごとのマッピングの指定だけではなく、アプリケーション全体のキャッシュのポリシーも変更できるほうがよりパフォーマンスを向上できるのではないかと考えている。例えば、多くのセッションで同様のデータが取得されることがわかっているのであれば、アプリケーションとしてそのデータをキャッシュしておくことで、アプリケーション全体での DB アクセスを抑制できるのではないかと思われるので、そのような、指定が出来るようにしたいと考えている。

参考文献

[1] Java 2 Platform Enterprise Edition. http://java.sun.com/j2ee/.

[2] **EJB** http://java.sun.com/products/ejb/.

[3] **Jboss.** http://www.Jboss.org/.

[4] **HIBERNATE.** http://www.hibernate.org/.

[5] **ObjectStyle: Cayenne.** http://www.objectstyle.org/cayenne/.

[6] **cglib.** http://cglib.sourceforge.net/.

[7] **World wide Web Consortium: XML Path Language.** http://www.w3.org/TR/xpath/.

[8] **Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jerey Palm, and William G. Griswold. An Overview of AspectJ. In** European Conference on Object Oriented Programming **(ECOOP), pages 327 - 353, June 2001.**