データベースシステム 113−21
（1997. 7. 14）

Javaのためのデータベース・エンジンの設計課題

渡辺 美樹* 早田 宏†
富士ゼロックス株式会社
ドキュメント工学研究所

要旨

本論文では，Java 言語のオブジェクトをデータベースに格納し，管理する機能（以下，永続化機能）の設計の概要を示す．

我々は，C++ 言語を対象としたオブジェクト指向データベース・エンジン *Earth* を実現しており，この研究で確立した技術を用いた Java 言語のための永続化機能の実現を目標とした研究を進めている．

永続化機能の設計の方針として，Java 言語に対する言語，実行環境の観点での高い適合性と，性能を重視する．また，Java 言語の永続化機能の設計空間を，永続直交性，永続透過性，永続識別性，キャッシュ管理，トランザクションとスレッドの 5 本の軸により定め，各軸における設計の選択肢について，特徴，課題を議論する．

さらに，我々が実現するデータベース・エンジン（*Earth* for Java）として，設計方針に従った，設計空間における設計の選択の概要を述べる．

現在，この設計に基づいて，*Earth* for Java の実現を進めている．

Database Engine Design Issues for Java

WATANABE, Yoshiki HAYATA, Hiroshi
Fuji Xerox Co., Ltd.
Document Engineering Lab.

Abstract

This paper describes a design overview of persistence functionality for Java language.

We have developed a database engine for C++, which is named *Earth*, and started development of persistence functionality for Java language based on the technology of *Earth*.

For the design of the functionality, we have two policies; high adaptability to Java language and Java execution environments, and high performance.

We define a design space of the Java persistence functionality, and discuss characteristics and issues on alternatives on five axes of the design space, according to the design policies. These axes are persistent orthogonality, persistent transparency, persistent identification, cache management, and transaction-thread relationship.

We also show a design of our database engine (*Earth* for Java) with using the above design space.

*yoshiki@rsl.crl.fujixerox.co.jp
†hayata@rsl.crl.fujixerox.co.jp

1 はじめに

我々は, 組み込み向けオブジェクト指向データベース・エンジンの実現を目的とした研究を進めてきた. この研究の成果として, C++ 言語を対象としたオブジェクト指向データベース・エンジン *Earth* を実現した [Hayata 95].

一方で, 組み込みシステム向のオブジェクト指向言語として, Sun Microsystems 社が開発した Java 言語が注目されるようになってきた. Java 言語で実用的なアプリケーションが記述されるようになるに伴い, データベースへのデータの格納, 取り出しのための機能の必要性が高まってきた.

この要求に対し, 我々は, *Earth* で確立した技術を用いた Java 言語のための永続化機能の実現を目標とした研究を開始した. 本論文では, 我々が実現しようとするシステムを *Earth* for Java と呼ぶ.

次節以降では, *Earth* for Java の設計指針, 設計上の選択肢, 課題とその解決方法について述べる.

2 設計指針

本節では, *Earth* for Java の設計における, Java への適合性, 性能の観点での設計の指針を示す.

2.1 Java への適合性

Java への適合性を言語に対する適合性と実行環境に対する適合性に分けて考える.

言語に対する適合性とは, 言語の構文や意味に対する変更の度合を表す. 変更が必要であり, 通常のアプリケーションの記述に対して与える影響の度合が大きいほど, 適合性が低い.

実行環境に対する適合性とは, 実行環境に対する変更の度合を表す. 変更が必要であり, その影響が大きいほど適合性が低い.

以下に, これら二つの観点で, 設計指針を示す.

言語 言語の構文, 意味に対する変更は一切行なわない. このことは, プリプロセッサの作成や, コンパイラの変更を必要としないことを意味する. また, 既存のクラス・ライブラリを, 変更することなく, 永続データの操作のために利用することができることを目標とする.

実行環境 Java の仮想機械 (VM) を変更しない. 既存の VM で実行可能であることを目標とする. また, JDK[SMC 97] などで提供されているクラス・ライブラリを変更せず, そのまま利用する.

2.2 性能

Earth for Java の実行時の性能に影響を与える要因として, 実行時に必要なメモリ資源の大きさと, Java オブジェクトの読み書きのためのオーバヘッドが考えられる.

永続データを操作するために必要となるメモリ資源の大きさを極力抑え, 永続データを扱わない場合と同程度のメモリ使用量で, 性能を損なうことなく実行可能となることを目指す.

また, Java オブジェクトの読み書きの処理を効率良く行なうため, データベースに格納されるオブジェクトは Java オブジェクトに限定する.

軸		選択肢
永続直交性	クラス定義の獲得	プリプロセッサ ポストプロセッサ リフレクション
	非 public 変数の参照	ネイティブ・メソッド リフレクション
永続透過性	変更の反映	自動検出 ポストプロセッサ
	ロックの確保	トランザクション・ベース 楽観的ロック
	オブジェクト・フォルトの検出	Upon dereference Upon discovery Never
永続識別性		到達可能性 明示
キャッシュ管理		weak pointer トランザクション境界
トランザクションとスレッド		Mutex object 入れ子の禁止

表 1: 設計空間の軸と選択肢

3 データベース・エンジンの設計空間

本節ではデータベース・エンジンの機能を軸とした設計空間を定め, 各軸上での設計の選択肢に対する特徴, 課題について議論する. 以下で述べる設計軸と各軸上での選択肢の概要を表 1 に示す.

3.1 永続直交性

永続直交性 (Persistence Orthogonality) は型と永続機能の独立性を示す [Atkinson 96].

オブジェクトに永続性を与える方法として, 永続化機能を持つクラスのサブクラスとする方法と, クラス階層とは独立に与える方法がある.

Earth や, ObjectDesign, Inc. の PSE for Java[ODI 96] は前者の方法を採用している. この方法は, 実現が容易であるが, 永続直行性がない. すなわち, 既存のプログラムで永続データをあつかうためには, プログラムの書き直しが必要である. このことは, オブジェクト指向の長所の一つであるプログラムの再利用性を損なうことを意味している. その

ため，最近のOODBMS，OODB エンジンでは，後者の方法が用いられ，高い永続直行性が得られている[Lamb 92][Singhal 92][Atkinson 96].

Java 言語で永続性をクラス階層と独立に与えるためには，以下の問題を解決する必要がある．

- クラス定義の獲得

  クラス定義をスキーマとしてデータベースに登録するためには，定義の内容を得る必要がある．その方法として，次の三種類が考えられる．

  プリプロセッサ ソースコードを読み込み，クラス定義を得る．この方法では，ソースコードのないクラスを永続化させることができないため，事実上，永続機能の直交性が損なわれる．

  ポストプロセッサ バイトコードを読み込み，クラス定義を得る．ソースコードが必要ないため，任意のクラスのオブジェクトを永続化できる．ただし，実行効率やメモリ効率の点で問題がある．

  リフレクション JDK 1.1 で採用されたリフレクションの機能を用いることで，実行時にクラス定義を得ることが可能である．Java への適合性の点で優れている．

- public でない変数の参照

  Java 言語では，高いセキュリティを確保するために，あるクラスの public ではない変数を他のクラスから参照することができない．クラス階層から独立に永続化の機能を実現するためには，このような変数を参照する手段が必要となる．

  ネイティブ・メソッド Java 言語では，C 言語等によるメソッドの実現を可能とするためにネイティブ・メソッドと呼ばれる機能が用意されている．JDK 1.0, JDK 1.1 のネイティブ・メソッドでは，すべてのクラスのすべての変数を自由に参照，変更できるため，上述の問題を回避することができる．この方法には実行環境に対する適合性を低下させる点で問題がある．

  リフレクション JDK 1.1 のリフレクションの機能では public でない変数を参照する事ができない．しかし，言語仕様上は可能であり，将来的には，実行環境への適合性を損なう事なく，public でない変数の値をデータベースに格納できるようになる可能性がある．

3.2 永続透過性

永続透過性 (Persistent Transparency) はプログラムと永続機能の独立性を示す [Atkinson 96].

Earth や PSE for Java では，永続オブジェクトに対する操作の前後に，オブジェクトの読み出し，書き込みのための処理を明示する必要がある．そのため，非永続オブジェクトを扱っていたプログラムを，変更せずに永続オブジェクトの操作に使用する事ができない．

この問題を解決するためには，一つのプログラムで永続，非永続の両オブジェクトを透過に扱えることが必要となる．これを実現するためには，以下の課題を解決しなければならない．

- キャッシュ内の変更の反映

  永続透過性を保証するためには，永続オブジェクトの変数を通常の代入文によって変更できなければならない．代入文で変更されたキャッシュ中の変数の値をデータベースに反映させる方法として，次の二種類が考えられる．

  自動検出 トランザクションごとに，そのトランザクションにおける処理によってキャッシュに読み込まれたオブジェクトをすべて記録しておく．それらのオブジェクトがキャッシュから追い出される時，またはトランザクションがコミットした時に，それらの変数の値を読み出し，データベースに反映させる．

  ポストプロセッサ バイトコードを読み込み，オブジェクトの変数の値を変更している部分を，データベースの値の更新を実行するコードに変更する．この方法は，厳密には，永続透過であるとはいえないが，再コンパイルが必要でないので，表面上は同等の効果が得られるが，実行効率，メモリ効率の点で問題がある．

- 書き込みのためのロックの確保

  オブジェクトをキャッシュに読み込む時点で，そのオブジェクトに対して，変数値の変更があるか否かを決定することはできない．そのため，その時点では排他的なロックを取得する必要があるか否かを決定できない．

  この問題を解決するために，以下の二種類の方法が考えられる．

  トランザクション・ベース トランザクション毎に，取得するロックの種類を指定する．すなわち，変数の値を変更する可能性のあるトランザクションには，それを開始する際に，あらかじめその旨を指定しておき，そのトランザクションの処理によって読み出されるオブジェクトにはすべて排他的なロックを確保する．

  この方法は，変更しないオブジェクトに対しても排他的なロックを確保してしまうため，本来必要のないロック待ちや，デッドロックを発生させる可能性がある．

  楽観的ロック 楽観的なロックによる排他制御を行

う．それにより，オブジェクトの読み出し時に変更の有無を指定する必要がなくなる．

この方法にはロック待ちがない利点があるが，変更が衝突した場合にはあるトランザクションのすべての処理を無効にしなければならないため，長いトランザクション処理により構成されるアプリケーションには適さない．

- オブジェクト・フォルトの検出

永続透過性を達成するためには，永続オブジェクトをキャッシュに読み込む際に，データベース内でのオブジェクト間の参照をキャッシュ内での参照(ポインタ) に置き換える必要がある．参照の利用時にこの置換が完了していない状態であった時，オブジェクト・フォルトが発生したという．

オブジェクト・フォルトを発見し，ポインタの置換を行なうには，参照のチェックを行う reservation check [Suzuki 94] を実現する必要がある．

reservation check の方法は，以下の三種類に分類できる．ここでは，説明のため，オブジェクト A がオブジェクト B を参照し，さらに B が C を参照している状態を考える．

Upon dereference キャッシュ内にオブジェクト A を読み込む際に，B への参照の値として，オブジェクト B のデータベース内での識別子を格納しておく．A から B への参照を利用する際に，参照先のオブジェクト B をキャッシュ内に読み込み，そのキャッシュ内の領域のアドレスを用いて，A から B への参照の値を書き換える．

この方法を Java 言語で利用するためには，オブジェクト A からオブジェクト B を利用する処理の前に，その参照がデータベース内の識別子なのか，キャッシュ内の領域を指しているのかを判別する処理を実行する必要がある．これを実現するためには，VM を変更するか，プリプロセッサによるソースコードの変換か，ポストプロセッサによるバイトコードの変換を行う必要がある．

Upon discovery キャッシュにオブジェクト A を読み込む際に，キャッシュ内に参照先のオブジェクト B の領域を確保し，A から B への参照の値として，B のキャッシュ内のアドレスを格納する．

この方法を Java 言語で利用するためには，Upon dereference と同様に，オブジェクト A からオブジェクト B を利用する処理の前に，B の内容が読み込まれているか否かを判別する処理を実行する必要があり，VM の変更，または，プリ / ポストプロセッサが必要となる．

Never (Swizzling at page fault time) キャッシュにオブジェクト A が読み込まれる際に，B のキャッシュ内の領域のアドレスのみを予約しておき，A から B への参照の利用時に，OS 仮想記憶管理によるページ・フォルトによって起動される例外処理により B の内容を読み込む．Java 言語で利用するためには，VM の変更が必要となる．

3.3 永続識別性

永続識別性 (Persistence Identification) は，オブジェクトに対し，永続化するか否かをどのように指定するかを示す指標である [Atkinson 96].

永続性がクラス階層から独立に与えられている場合，オブジェクトの生成時に永続オブジェクトであることを指定する必要がなくなる [Moss 96]. この場合，オブジェクトは到達可能性 (**reachability**) により永続化されることが望ましい．

オブジェクトの生成時に永続，非永続を決定する方法もあり，*Earth*, ObjectStore[Lamb 92] がこの方法を採用している．

3.4 キャッシュ管理

Java 言語ではオブジェクトを明示的に削除することを許しておらず，ガーベジ・コレクタ (GC) による領域の回収を行っている．キャッシュからオブジェクトを取り除く方法としても GC を用いることが，言語との親和性の観点からも望ましい．

しかし，アプリケーション領域からキャッシュされたオブジェクトの参照が無くなっても，キャッシュ管理機能からの参照があるために，通常の GC ではキャッシュされたオブジェクトの領域が回収できなくなってしまう．この問題を解決する方法として，次の二つが考えられる．

weak pointer weak pointer とは，アプリケーションから見ると通常のポインタであるが，GC からはポインタとして扱わないポインタである．キャッシュ管理機能からキャッシュされたオブジェクトへの参照に weak pointer を用いることにより，アプリケーションからの参照がなくなり，管理機能からのみの参照になったときにキャッシュされたオブジェクトの領域を解放できる．

しかし，現在の Java 言語では weak pointer の機能が提供されていないため，VM を変更する等の方法が必要となり，適合性の点で問題となる．

トランザクションの境界による解放 アプリケーションがトランザクションを跨ってキャッシュ内のオブジェクトを参照することを禁止する．これにより，トランザクションの終了時に，管理機能から

キャッシュ内のオブジェクトへの参照をすべて破棄し, GC が回収できるようにする.

トランザクションが比較的短く, トランザクションを跨ってオブジェクトを利用することが多いようなアプリケーションではキャッシュの効果が低くなり, 性能が低下する恐れがある. ただし, トランザクションの範囲を越えて部分的にキャッシュを有効にするような機能を付加することで, 性能を改善することができる.

3.5 トランザクションとスレッド

複数のスレッドが同時に実行されている状況では, 永続オブジェクトに対してロックを確保する処理を排他的に実行しなければならない. Java 言語では, メソッドの排他的な実行のために synchronized method と呼ばれる機能が用意されている. しかしながら, synchronized method の呼び出しが入れ子になった状態で, 永続オブジェクトに対するロックを確保しようとし, ロック待ちの状態になると, そのオブジェクトに対するロックを保持しているスレッドがロックをはずす処理を実行できないため, デッドロックの状態となってしまう. この問題を回避する方法として, 以下の二つの方法が考えられる.

Mutex object 排他制御を行うためのオブジェクト (Mutex object と呼ぶ) を用意し, Java 言語の構文による排他制御とは独立に, 自由に排他制御を行う機能を実現する.

デッドロックや不必要な実行待ちを避けるために, Mutex object の利用には細心の注意が必要であるが, 排他制御の範囲を自由に設定できるため, プログラムの並列性を高めることができ, 性能向上にも貢献できる.

入れ子の禁止 回避できないデッドロックが発生するのは synchronized method が入れ子状態で呼び出される場合である. データベース側のメソッドを synchronized method が入れ子にならないように実現し, アプリケーションが synchronized method を利用しないようにすれば, デッドロックは避けられる.

しかし, この方法で並行処理プログラムを実現することは困難である. また, 仮に実現できたとしても, 広い範囲の処理を排他的に実行しなければならなくなるため, 実行待ち状態のスレッドが増え, プログラムの並列性を低くしてしまう.

4 設計

本節では, 前節で定めた設計空間において, *Earth* for Java の設計として, どのような設計を選択するかとその理由を述べる. 選択の一覧を表 2 に示す.

軸		選択
永続直交性	クラス定義の獲得	リフレクション
	非 public 変数の参照	ネイティブ・メソッド + リフレクション
永続透過性	変更の反映	自動検出
	ロックの確保	トランザクション・ベース
	オブジェクト・フォルトの検出	Upon discovery
永続識別性		到達可能性
キャッシュ管理		トランザクション境界
トランザクションとスレッド		Mutex object

表 2: *Earth* for Java の設計

4.1 永続直交性

クラス定義を得る方法としてリフレクションの機能を用いる. また, public でない変数を参照する方法として, ネイティブ・メソッドを利用する. これにより, 高い永続直交性を達成することが可能となる. プリプロセッサ, ポストプロセッサを用いる方法は永続直交性を損なうため採用しない.

4.2 永続透過性

キャッシュ内の変更をデータベースに反映させる方法として, 自動検出の方法を用いる. ポストプロセッサを用いる方法は, 永続直交性を損なうため, 用いない.

また, 書き込みのロックの取得方法としては, 実装が容易であるトランザクション・ベースの方法を用いる. 楽観的ロックによる方法については, 将来の課題とする.

オブジェクト・フォルトの発見方法としては, **Upon discovery** に基づくもので, *Earth* と同じくキャッシュ内にデータベース内の識別子への参照を持たずに, 変換時にデータベースの内容を利用するコピー変換方式 [Moss 92] を採用する. 実装を容易にするため, 環境に対する適合性が低下するが, アプリケーションが明示的にチェック・コードを呼び出す方法を採用する. 将来的は VM からの支援が得られることを期待する.

4.3 永続識別性

到達可能性による永続化を実現する. そのために, 永続化の起点となる永続ハッシュ表を実現し, 文字列をキーとしオブジェクトを値として格納できるようにする. このハッシュ表に格納されているオブジェ

クトとそこからたどれるすべてのオブジェクトが永続化される.

4.4 キャッシュ管理

キャッシュからオブジェクトを取り除く方法としては, トランザクションの境界による解放の方法を採用する. また, トランザクションを跨って利用したいオブジェクトをキャッシュに残せるように, オブジェクトごとに解放の可,不可を指定できるようにする. 将来的に Java 言語で weak pointer が提供されるようになった時点で, weak pointer によるキャッシュ管理を実現する.

4.5 トランザクションとスレッド

排他制御の方式として **Mutex object** による同期の方法を採用する.

5 関連研究

以下に本研究に関連する研究, 製品を紹介する.

PJava[Atkinson 96] PJava は, Glasgow 大学と Sun Microsystems Laboratories との共同ですすめられている研究であり, 高い永続直交性を達成することを目標としている. 既存のコンパイラをそのまま利用でき, ポストプロセッサも必要としていないため, 既存のバイトコードをそのまま利用できる. しかし, PJava 用の特殊な VM を必要としており, 実行環境への適合性の点で問題がある.

PSE for Java[ODI 96] PSE for Java は Object Design 社が開発した製品であり, API が簡便であること, 少いメモリ資源で動作可能であることを特徴としている. 永続化の機能を提供するクラスのサブクラスのインスタンスのみが永続化可能であるため, 永続直交性がない. しかし, ポストプロセッサを用いたバイトコードの変換の機能により, 外見上, 永続直交性を達成している. 変換されたバイトコードは, どの VM でも動作可能であり, 実行環境への適合性は高い. しかし, 実行効率や実行時のメモリ効率の点で問題がある.

6 まとめ

Java のためのデータベース・エンジンの設計空間の軸として, 永続直交性, 永続透過性, 永続識別性, キャッシュ管理, トランザクションとスレッドの 5 軸を定め, 各軸における設計の選択肢と課題について議論した. また, 我々が実現するデータベース・エンジン *Earth* for Java の設計にあたり, その方針と設計空間における機能の選択を示した.

現在, この設計に基づいて, JDK 1.1.x を用いて *Earth* for Java の実現を進めている.

今後の課題としては, 本報告書で述べた課題のほかに, Java 言語の特徴の一つであるネットワーク・ローダブルである点を活かしたデータ管理の方法を, *Earth* の拡張可能性の特徴を活かし, さらに押し進める形で確立することがあげられる.

謝辞

設計の議論に加わっていただいた, 弊社ドキュメント工学研究所の掘切和典, 奥村洋, システム実験研究所の安松一樹, 関島章文の各氏に感謝する.

参考文献

[Atkinson 96] Atkinson, M. P., Jordan, M. J., Daynès, L., and Spence, S.: Design issues for Persistent Java: a type-safe, object-oriented, orthogonally persistent system, In *Proceedings of the Seventh International Workshop on Persistent Object Systems*, May, 1996, `http://www.dcs.gla.ac.uk/pjava`

[Hayata 95] 早田宏, 渡辺美樹, 田中圭, 山崎伸宏: 組み込みに適した拡張可能なオブジェクト指向データベース・エンジン Earth の実現, 富士ゼロックステクニカルレポート *No.10*, pp. 98–107, 1995.

[Lamb 92] Lamb, C., Landis, G., Orenstein, J., and Weinreb, D.: The ObjectStote Database System, *Comm. ACM*, Vol. 34, No. 10, pp. 50–63, 1992.

[Moss 92] Moss, J. E. B.: Working with Persistent Objects: To Swizzle or Not to Swizzle, *IEEE Trans. Softw. Eng.*, Vol. 18, No. 8, pp. 657–673, 1992.

[Moss 96] Moss, J. E. B. and Hosking, A. L.: Approaches to Adding Persistence to Java, In `http://www.dcs.gla.ac.uk/~carol/Workshops/PJ1Papers.html`, 1996.

[ODI 96] Object Design, Inc.: Object Design ObjectStore for Java, `http://www.odi.com/products/pse/doc/index.html`, 1996.

[Singhal 92] Singhal, V., Kakkad, S. V., and Wilson, P. R.: Texas: An Efficient, Protable Persistent Store, In *Proc. Fifth Int'l. Workshop on Persistent Object Systems*, 1992.

[SMC 97] Sun Microsystems, Inc.: Java Development Kit Version 1.1.1, `http://www.javasoft.com/products/jdk/1.1/index.html`, 1997.

[Suzuki 94] Suzuki, S., Kitsuregawa, M., and Takagi, M.: Dimensions and Mechanisms of Persistent Object Faulting, In *ADTI '94*, pp. 145–151, 1994.