情報処理学会論文誌　プログラミング　Vol. 4　No. 1　66–75 (Mar. 2011)

Javaにおける明示的メモリ管理効率化のための参照無効化手法

足立昌彦[†1]　小幡元樹[†1]　西山博泰[†1]
岡田浩一[†2]　長瀬卓真[†2]　中島恵[†2]

Javaにおける長時間GC発生回数の削減を目的として，明示的メモリ管理手法を提案している．本手法では，従来の世代別GCと同様に，Tenured領域内オブジェクトを簡易的にすべてliveと見なしている．そのためTenured領域から明示管理メモリへ参照がある場合，明示管理メモリ内のオブジェクトがliveと見なされ明示管理メモリの使用量が増加する．Tenured領域内オブジェクトの要否判定は長時間GCにより行われる．しかし，明示的メモリ管理を適用することにより長時間GCの発生間隔が長期化する．これにより明示的メモリ管理機能を適用すると本来deadなTenured領域内オブジェクトが長期にわたりliveと見なされ，そこから参照される明示管理メモリ領域内オブジェクトも長期間liveと見なされ明示管理メモリの使用効率が低下する．そこで本論文では，当該オブジェクトを早期に検出し，参照を無効化する手法を提案する．本手法はJavaヒープを小さなブロックに分割し，低オーバヘッドでTenured領域内オブジェクトの要否を判定し，deadなオブジェクトの参照を無効化するGC手法の一種である．本手法を適用した結果，Major GC時間の3%未満の処理時間で本来deadなTenured領域内オブジェクトの参照を無効化でき，明示管理メモリ領域の使用量を最大で従来の1/5に削減できることを確認した．

Reference Nullification for Efficient Management of Explicitly Managed Heap Memory

Masahiko Adachi,[†1] Motoki Obata,[†1]
Hiroyasu Nishiyama,[†1] Koichi Okada,[†2]
Takuma Nagase[†2] and Kei Nakajima[†2]

The explicitly managed heap memory is a kind of non-garbage collected heap memory aimed at reducing long waiting time caused by stop-the-world style garbage collection. Live references from tenured space to objects in explicitly managed heap memory increase because current implementation of explicitly managed heap memory treats objects in tenured area temporarily live as minor GC do. This delays reclamation of explicitly managed heap memory because of decreased collection of tenured space. To cope with this problem, we propose a low-overhead reference nullification method for efficient management of explicitly managed heap memory. Tenured space is divided into small chunks and dead objects in each area are recognized in turn. After detecting dead objects, references in the dead objects are cleared to ease reclamation of explicitly managed heap memory. Experimental evaluations of the proposed method show that references in dead objects can be cleared with only 3% overhead on major GC and usage of explicitly managed heap memory can be reduced to 1/5.

1. はじめに

近年，Java[1),⋆1]システムがサーバ分野における業務プログラムの基盤や，電子商取引に代表されるミッションクリティカル性の高い基幹システムなどで使われるようになってきている．これらの領域ではプログラムを実行するJava仮想マシン（Java VM）に対し，高い信頼性と実行性能が求められるため，アプリケーションの実行性能が高く利用実績の豊富なstop-the-world型の世代別ガベージコレクション（以下，GC）を採用することが多い．GCはヒープメモリの使用量が閾値を超えるときに発生するが，このタイミングを予見することは容易でない．GCによる停止時間は利用ヒープメモリ容量に応じて長くなる傾向にある．近年のハードウェアリソースの増大により，Javaが利用できるヒープメモリ容量は増加しており，GCによる停止時間の長期化は顕著になっている．世代別GCは処理時間の短いMinor GCと長時間の停止を要するMajor GCがあるが，特にMajor GCによって生じる長時間停止はミッションクリティカル性を要求するシステムにおいては許容し難いケースが多い．この問題を解決するため，我々はJavaプログラム上からメモリ領域の確保・解放を明示的に指示可能な明示的メモリ管理APIを提案している[2)]．これは，データ群を領域ごとに管理するAPIをプログラマが明示的に利用することにより，Major GCの発生頻度の抑止を可能にするものである．

明示的メモリ管理機能を様々なアプリケーションに適用し課題が明らかになってきた．そ

†1 株式会社日立製作所システム開発研究所
System Development Laboratory, Hitachi Ltd.
†2 株式会社日立製作所ソフトウェア事業部
Software Division, Hitachi Ltd.
⋆1 Javaは，Oracle Corporationおよびその子会社，関連会社の米国およびその他の国における登録商標です．

の 1 つにソースコードの修正がある．すなわち，明示的メモリ管理機能を適用するには明示管理メモリ領域（Explicit メモリ）の確保，解放の API を使用するようソースコードを改変する必要があった．これは既存のシステムへの適用の障壁となっていた．さらに，明示的メモリ管理機能を適用するために修正したソースコードは他の Java VM との可搬性を低下させる．これらの課題に対して，我々は明示的メモリ管理機能を半自動的に適用できる手法を提案している[3]．この手法では，ユーザが Explicit メモリを確保するプログラム点を外部ファイルで指定する．指定されたプログラム点を Java VM が実行する際に Explicit メモリを確保する API を挿入する．確保した Explicit メモリは Java VM が管理し，内部のオブジェクトが不要になった時点で一括して解放する．

もう 1 つの課題は明示的メモリ管理機能における解放処理と世代別 GC に由来するものである．Explicit メモリを解放するとき，そこに含まれるオブジェクトで後のプログラム実行に必要なオブジェクトを解放しない領域へと退避する安全な領域解放機能を提供している．これによりプログラマはオブジェクトの要否を意識することなく明示的メモリ管理機能を利用できる．一方で，世代別 GC を採用している場合，旧世代領域（Tenured 領域）内のオブジェクトをすべて live オブジェクトと見なして処理を簡素化し高速化している．そのため，Tenured 領域から参照される他領域内のオブジェクトも live と見なされる．特に当該オブジェクトが Explicit メモリ内のオブジェクトを参照している場合，その被参照オブジェクトは後のプログラム実行に必要と判定するため，解放できない．すなわち，Explicit メモリの利用効率が低下する．

Tenured 領域内オブジェクトの正確な要否判定は Major GC 時に実施する．すなわち Major GC が発生するまで，Tenured 領域内のオブジェクトから参照される Explicit メモリ内のオブジェクトは live と見なされ続ける．

この課題に対して，本論文では Tenured 領域内のオブジェクトに対しての簡易的な要否判定を実施する方法を提案する．また，実アプリケーションを模擬したプログラムに提案手法を適用した結果について述べる．

2. 明示的メモリ管理

本章では，明示的メモリ管理機能，および，明示的メモリ管理機能の半自動適用技術について述べた後，世代別ヒープメモリ構造による明示的メモリの解放に関する課題を説明する．なお，以後の説明で述べる Java VM は HotSpot VM[4] を対象としている．

2.1 明示的メモリ管理の概要

世代別 GC では，Minor GC を複数回実施しても回収できなかったオブジェクトは Tenured 領域へ移動（promotion）する．この promotion により Tenured 領域の使用量が増加し，Tenured 領域の使用量が閾値を超えると Major GC が発生する．そのため，Tenured 領域へのオブジェクト移動量を抑えることができれば，Major GC の発生回数を削減することが期待できる．このような観点から，我々は，明示的メモリ管理を提案している．明示的メモリ管理とは，Java プログラムで利用するオブジェクト群を GC 対象外メモリ領域（Explicit メモリ）に配置し管理する仕組みである．

ユーザは明示的メモリ管理 API を通して Explicit メモリの生成，削除およびそのメモリへのオブジェクト配置などを行う．Explicit メモリは管理用 Java オブジェクトと関連付けられており，管理オブジェクトの生成により暗黙的に領域を確保する．Explicit メモリへのオブジェクト配置の際には，領域の自動拡張を行う．

Explicit メモリへのオブジェクト配置は Minor GC における promotion 時に実施する．すなわち，従来の promotion によるオブジェクトの移動先が Tenured 領域から Explicit メモリ領域に変更になるだけで，処理時間に大きな影響はなく Minor GC によるプログラム停止時間の劣化は少ない．Explicit メモリの解放は Java プログラムを停止し，その領域に配置された複数のオブジェクトを一括して削除する．ただし，生存（live）オブジェクトを削除するとプログラムの動作が不正になる可能性がある．このため，live である可能性のあるオブジェクトを Java VM が検出し，解放対象外のメモリ領域へ移動することで，後のプログラム実行に不正を発生させない安全な領域解放機能を提供している．

図 1 に安全な領域解放機能の概要を示す．Explicit メモリ 1 を解放する場合を考える．解放対象外のメモリ領域にある live オブジェクトからの参照を調査し Explicit メモリ 1 内のオブジェクトの要否判定を実施し，Explicit メモリ 1 の live オブジェクトを検出する（図 1 (a)）．当該オブジェクトを解放対象領域から退避するため，新たに Explicit メモリ 2 を確保する（図 1 (b)）．Explicit メモリ 1 の live オブジェクトを Explicit メモリ 2 へ移動し，Explicit メモリ 1 をその中に含まれる dead オブジェクトごとまとめて解放する（図 1 (c)）．

安全な開放処理において，オブジェクトの要否を高速に判定するため Card marking[5] を拡張した，ExplicitHeapReferenceArray（EHRA）を導入している[2]．

EHRA は Java ヒープ領域と Explicit メモリを一定サイズの区画に分割し，各区画の参照先を記憶する配列構造である（**図 2**）．たとえば，Java ヒープ内のオブジェクト（obj）が ID 番号 1 の Explicit メモリ内のオブジェクトを参照している場合，obj に対応する EHRA

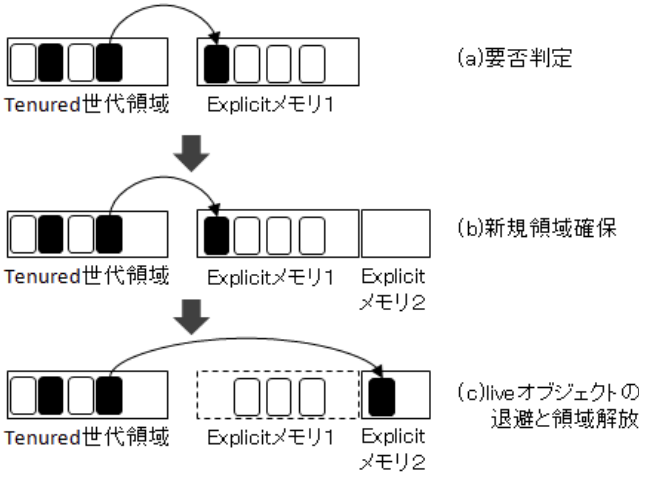


図 1　Explicit メモリの解放にともなう live オブジェクトの退避
Fig. 1　Live object migration from reclaimed ExplicityMemory to new one.

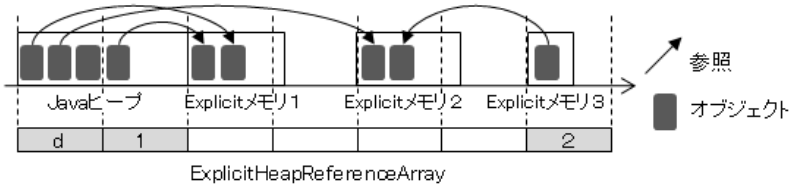


図 2　ExplicitHeapReferenceArray のイメージ
Fig. 2　Image of ExplicitHeapReferenceArray.

の要素の値が ID 番号の 1 となる．また，EHRA の 1 要素に対応する Java ヒープの区画内から異なる Explicit メモリ領域への参照がある場合，その EHRA の要素は d（dirty）となる．Explicit メモリ内オブジェクトの要否判定は，その ID 番号を持つ EHRA の要素に対応する領域内のオブジェクトの参照を調査するだけでよく，高速に実施できる[⋆1]．

これらの機能を実現する明示的メモリ管理 API を用いることで，これまで Tenured 領域

⋆1 Explicit メモリ内のオブジェクトを参照するオブジェクトを remembered set[5] に保存する方法でも要否判定を実現できる．Hotspot VM において，Copy GC の高速化のための実装が Card Marking であるため，明示的メモリ管理機能でもそれを拡張した EHRA を適用している．

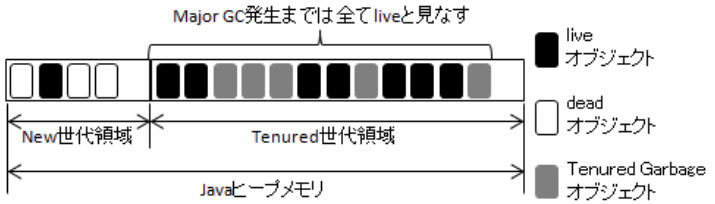


図 3　Java ヒープメモリ構造とオブジェクトの種類
Fig. 3　Object management in Java heap.

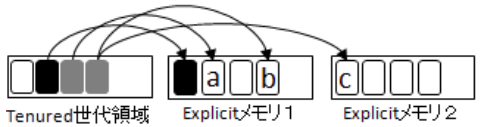


図 4　TG オブジェクトと Explicit メモリ内オブジェクトの参照関係の例
Fig. 4　References from tenured garbage objects to objects in explicitly managed memory.

に配置されていたオブジェクトを Explicit メモリに配置することが可能になり，Major GC による長時間停止を抑止できる．

2.2　世代別ヒープメモリ構造による Explicit メモリ解放の課題

Hotspot VM で採用している世代別 GC において，Minor GC では Tenured 領域内のオブジェクトをすべて live と見なし，New 領域内のオブジェクトの要否判定を行っている（**図 3**）．Minor GC を実施する際，Tenured 領域のオブジェクトはすべて live として扱い，そこから参照する New 領域のオブジェクトはすべて live と見なし回収しない．Tenured 領域内のオブジェクトの要否判定は Major GC の Mark 処理で実施する．そのため，Mark 処理で dead と判定できる Tenured 領域内のオブジェクトでも，Major GC が発生するまでは live オブジェクトとして処理する．このように正確な Mark 処理で dead と判定できるが簡易的に live と見なしている Tenured 領域内オブジェクトを Tenured Garbage オブジェクト（TG オブジェクト）と呼ぶ[6]．

TG オブジェクトが Explicit メモリ内のオブジェクトを参照している場合，Explicit メモリ内のオブジェクトも live と見なす．**図 4** の例では，Tenured 領域内の TG オブジェクトから参照する Explicit メモリのオブジェクト a，b，c は live と見なす．

TG オブジェクトは Major GC が発生するまで回収されないため，そこから参照する Explicit メモリ内のオブジェクトも回収できない．すなわち安全な領域解放を提供する明示

管理メモリ機能において，TG オブジェクトが回収されるまで Explicit メモリ内のオブジェクト a，b，c も回収できない．これは，Explicit メモリの一部の領域は Major GC が発生するまで利用できないことを意味し，Explicit メモリの利用効率が低下する．

この問題を回避するため，TG オブジェクトが発生しないように Java ヒープメモリの構造や promotion の閾値を調節する必要がある．しかし，Web システムなどではオブジェクトの生存期間が動的に変化するため，TG オブジェクトを発生させないように各種の設定を調整することは多くの試行錯誤を要する．

3. Tenured Garbage オブジェクトの参照無効化

本章では，TG オブジェクトからの参照を無効化する方法を説明する．まず TG オブジェクトを判定する方法を説明し，その後，TG オブジェクトからの参照を無効化する方法について説明する．

3.1 TG オブジェクトの判定方法

オブジェクトの要否判定には Mark 処理を実施するのが一般的であるが，Mark は Mark-Compact GC 処理の半分程度の長い時間を要することが予備実験により分かっている．そこで Tenured 領域を一定サイズの領域（Tenured ブロック）に分割し，その 1 つのブロックに存在するオブジェクトに対して簡易的に要否判定する手法を提案する．

Mark 処理が多くの時間を要するのは Java ヒープ全体を対象に，RootSet [*1] からたどれるすべてのオブジェクトを調査し，厳密にオブジェクトの生死を判定するためである．提案手法は，Tenured 領域を一定サイズのブロック（Tenured ブロック）に分割し，その 1 つの Tenured ブロックを調査対象とする．Tenured ブロックの大きさを小さく設定することで，そこに含まれる要否判定の対象となるオブジェクトの数を減らしオブジェクトの要否判定の処理時間を短縮できる．この要否判定処理ではオブジェクトの要否判定の対象にする Tenured ブロック以外の領域に含まれるオブジェクトを live と見なす必要があるため TG オブジェクトが残存することになる．そのため，オブジェクトの要否判定対象の Tenured ブロックを要否判定の実施ごとに変更し，TG オブジェクトを回収する．

オブジェクトを要否判定する手順を示す．

(1)　Tenured 領域をある大きさの領域（Tenured ブロック）に分割する．

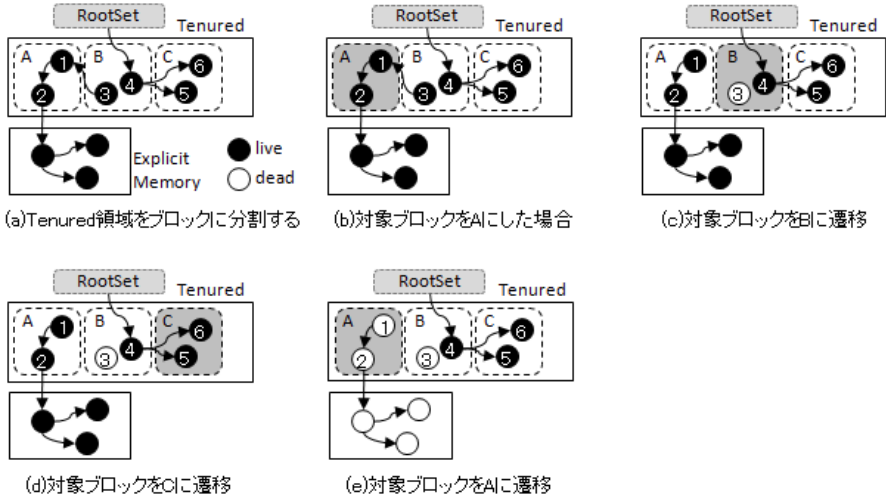


図 5　ブロックに分割した Tenured 領域ごとにオブジェクトの要否を判定する
Fig. 5　Division of tenured space into blocks and detection of necessity of objects in each block.

(2)　オブジェクトの要否判定を実施する Tenured ブロック（対象ブロック）を選択する．対象ブロック内に存在するオブジェクトを対象オブジェクトと呼ぶ．また，対象ブロック以外の Tenured ブロックに含まれるオブジェクトや New 領域や Explicit メモリに含まれるオブジェクトを対象外オブジェクトと呼ぶ．

(3)　RootSet および対象外オブジェクトが参照する対象オブジェクトを live と見なす．

(4)　ステップ（3）で求めた対象ブロック内の live オブジェクトから参照をたどれるすべての対象オブジェクトを live とする．

(5)　対象ブロック内で live でないオブジェクトを dead とする．

(6)　（2）に戻り，処理を繰り返す．

具体例を用いて上記手順を説明する．**図 5** (a) のように Tenured 内オブジェクトがすべて live と見なされているため，そのオブジェクトが参照する Explicit メモリ内のオブジェクトも live となりメモリ使用効率が低下しているケースを考える．

はじめに Tenured 領域をある大きさの領域のブロック（Tenured ブロック）に分割する（手順 1）．各ブロックを A，B，C とする．Tenured 領域に存在するオブジェクト（Obj1. . . Obj6）はすべて live と見なされている．

⋆1 Java ヒープ外からの参照．Java プログラムの実行に確実に必要になるオブジェクトを Java ヒープ外から参照している．RootSet から参照されるオブジェクトとして，実行中のスタックフレーム内で保持するオブジェクトやスレッドオブジェクト，例外オブジェクトなどがある．

まず，対象ブロックを A とする（手順 2）．対象外オブジェクトをすべて live として扱い，そこから対象オブジェクトへの参照をたどる．図 5 (b) では Obj3 はどこからも参照されていないため TG オブジェクトであるが，ここでは live として扱う．このため Obj3 が Obj1 を参照しているため Obj1 は live となる（手順 3）．Obj1 から参照する対象ブロック内のオブジェクト Obj2 も live となる（手順 4）．

次に，図 5 (c) に示すように対象ブロックを B へと遷移する（手順 6，2）．同様に，対象外オブジェクトを live として扱い，対象オブジェクトへの参照を調査する．RootSet から参照される Obj4 は live になる．Obj3 は参照されていないため dead と判定できる（手順 5）．ここで Obj3 を削除できるため Obj1 への参照も削除できる．

対象ブロックを C へと遷移する（図 5 (d)）．同様に対象外オブジェクト（ブロック C に含まれないオブジェクト）から参照をたどると，Obj4 から Obj5，Obj6 を参照しているため，これらのオブジェクトは live になる．

すべてのブロックを遷移したため，再び対象ブロックを A に戻す（図 5 (e)）．図 5 (b) とは異なり，対象外オブジェクトから参照をたどれない Obj1 は dead であると判定できる．ここで Obj1 を削除できるため Obj1 から Explicit メモリ内のオブジェクトへの参照も削除できる．これにより，Explicit メモリの利用効率を向上させることができる．

本手法では対象外オブジェクトを live と見なすことで，それらのオブジェクトの要否判定が不要になるため，対象外領域から参照されていないオブジェクトを軽量に要否判定できる[⋆1]．また，1 回の要否判定処理で 1 つの Tenured ブロックのみ要否判定対象にすることで対象となるオブジェクトの数を少なくでき要否判定処理時間を Major GC に対して低減させることができる．この要否判定処理を実施するタイミングとして，Minor GC や Explicit メモリの使用率が閾値を超えたときなどが考えられる．本論文では Minor GC ごとに 1 つの Tenured ブロックを要否判定の対象としている．すなわち，Minor GC ごとに 1 つの Tenured ブロックの要否判定処理を行う．これにより，Tenured 領域のオブジェクトを軽量に要否判定できる．さらに，従来 Major GC においてのみ回収可能であった TG オブジェクトが，早い段階で検出できるようになるため，不要なオブジェクトが長期にわたりメモリ領域を占有する状態を回避できる．

⋆1 対象外オブジェクトを live と見なすことで TG オブジェクトが残存する可能性があるが，Tenured ブロックの要否判定処理ごとに対象ブロックを遷移させることで，徐々に TG オブジェクトを回収できる．

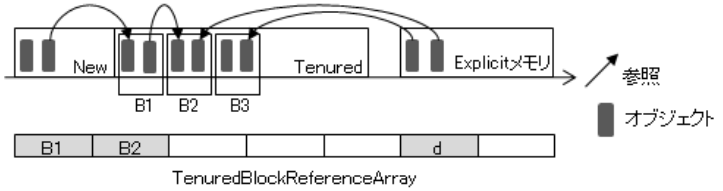


図 6　TenuredBlockReferenceArray の構造
Fig. 6　Structure of TenuredBlockReferenceArray.

3.2　要否判定の高速化

オブジェクトの要否判定は多くの時間を要する．そのため，オブジェクトの要否判定を高速化するため，Tenured Block Reference Array（TBRA）というブロック間の参照関係の特定を容易化する配列形式の補助構造を導入する．TBRA は Java ヒープ領域および Explicit メモリの全領域を一定サイズの領域に分割し，各領域から参照する Tenured ブロックの ID 番号[⋆2]を配列の要素で管理する．

図 6 の例では，New 領域のオブジェクトから Tenured 領域への参照がある．参照先のオブジェクトが属する Tenured ブロックの ID 番号が B1 であることから，参照元の領域に対応する TBRA の要素には B1 を格納する．同様に，Tenured ブロック B1 に属するオブジェクトは Tenured ブロック B2 のオブジェクトを参照しているため，参照元の領域に対応する TBRA の要素には B2 を格納する．Explicit メモリのオブジェクトはそれぞれ B2 と B3 の Tenured ブロック内のオブジェクトを参照している．参照元のオブジェクトが属する領域に対応する TBRA の要素が同一であるので，その要素には d（dirty）を格納する．このようにして参照先の Tenured ブロックを管理することでオブジェクトの要否判定を実施する必要のある領域を限定でき，要否判定を高速化できる．

3.3　循 環 参 照

提案する軽量な要否判定方法には，ブロックを跨ぐ循環参照関係にあるオブジェクトを回収できないという問題がある．すなわち，オブジェクト間の参照でブロックをまたいだ循環参照が存在する場合，それらを dead と判定することができない．たとえばブロック A，B に存在するオブジェクト Obj1，Obj3 が循環参照になっている場合を考える（**図 7**）．

この場合，対象ブロックが A のとき，Obj3 から Obj1 を参照しているため Obj1 は live

⋆2 Tenured ブロックはブロックを一意に特定できる ID 番号を持つ．

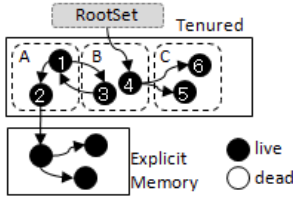


図 7　相互参照により Dead 判定ができない場合
Fig. 7　The case that dead object detection is not possible because of a circular reference.

となる．対象ブロックが B のときは Obj1 が Obj3 を参照しているため Obj3 は live となる．このように，ブロックをまたいで循環参照が存在する場合，対象ブロックを遷移させても TG オブジェクト Obj1，Obj3 を dead と判定できない．

この問題に対して，循環参照関係になっているオブジェクトの要否を判定する一般的な方法は Mark 処理であるが，前述のとおり，この処理は多くの時間を要する．別の解決方法として，複数ブロックを要否判定の対象とする方法が考えられる．たとえば Obj1 と Obj3 が相互参照の関係にあることを解析し，それぞれが属するブロック A とブロック B を要否判定対象ブロックとする．これによりブロックをまたがない相互参照にすることができるため，TG オブジェクトである Obj1 と Obj3 を回収できる．

我々が対象としているプログラムについて，ブロックをまたぐ循環参照数を調査したところ，ブロックをまたぐ参照のうち 0.1%未満程度であった．このことから，循環参照に対する上記の対策の効果が小さいと考え，本論文では循環参照に関する対策を実施していない．

3.4　参照無効化手法

TG オブジェクトを回収するために，3.1 節で特定した dead オブジェクトからの参照を無効にする必要がある．dead オブジェクトは後の実行に必要ないオブジェクトであり他のフィールドが持つ値を維持する必要はない．そのため，参照を持たないオブジェクト（ダミーオブジェクト）で dead オブジェクトを上書きする方法を採用する．ダミーオブジェクトは int 配列オブジェクトなどの要素数を変更することでオブジェクトの大きさを柔軟に変更できる配列オブジェクトを適用する[⋆1]．

参照無効化した領域は Java VM 内で管理し再利用可能である．参照無効化した領域を再利用することで Tenured 領域のメモリ利用効率が向上することが考えられる．我々が対象とするシステムに明示的メモリ管理を導入することで Major GC の発生頻度を低減できていることから[2),3)]，参照無効化領域を再利用する利点は少ないと考える．そのため，本論文では参照無効化領域の再利用を導入していない．

4. 評　　価

本章ではこれまでに説明した明示的メモリ管理機能の半自動適用技術を適用した Java VM を用いて，TG オブジェクトの参照無効化について評価を行う．

まず，提案手法によって TG オブジェクトが参照無効化できるか否かを検証する．その後，TG オブジェクトの参照無効化と Explicit メモリの使用効率との関係を評価する．

4.1　実 験 環 境

実験で用いるマシン環境は，CPU：Xeon（1.6 GHz，Dual Core × 2），メモリ：2 GB，OS：Linux（CentOS4.8）である．評価プログラムとして O/R マッピングフレームワークの Hibernate を用いる．Hibernate を利用したユーザプログラムはデータベースを検索する処理とした．データベースアクセス処理は一定時間継続しているため，関係するオブジェクトが Tenured 領域へと配置される．データベースアクセス処理が終了するとそれらのオブジェクトは不要になる．評価で用いる Java VM のメモリ設定は Java ヒープを 768 MB，Explicit メモリを 256 MB とする．

評価プログラムに対し明示的メモリ管理 API を適用する．適用点はデータベースにアクセスするたびに生成されるフレームワーク内のデータ構造[⋆2]である．

4.2　評 価 結 果

対象ブロックのサイズを 1 MB，5 MB，10 MB にした場合における参照無効化した領域サイズの累積と参照無効化したオブジェクト数の累積を**図 8** に示す．グラフの横軸（時間軸）に対して参照を無効化できているオブジェクトが増加していることから，提案手法を用いて TG オブジェクトの参照を無効化できていることが分かる．また，対象ブロックのサイズが大きいほど TG オブジェクトの参照無効化が促進されている．

表 1 に対象ブロックへの参照の平均数と対象ブロックを遷移し Tenured 領域を一巡した回数を示す．対象ブロックのサイズが大きいほど対象ブロックへの参照数の平均が多い．こ

⋆1 Explicit メモリを参照するオブジェクトを remembered set で管理する場合（2.1 節），TG オブジェクトの参照を無効化する代わりに，remembered set から TG オブジェクトを削除する．

⋆2 ThreadLocalSessionContext.doBind() メソッドと Configuration.buildSessionFactory() メソッドのそれぞれで生成される HashMap オブジェクトと SessionFactoryImpl オブジェクトを Explicit メモリへ配置する．

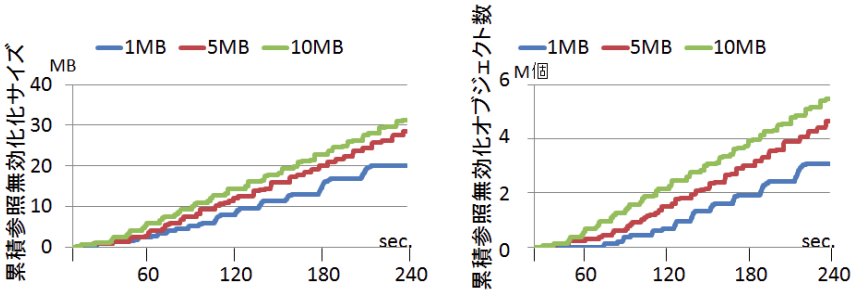


図 8　参照無効化オブジェクトの累積数
Fig. 8　The cumulative number of nullified objects.

表 1　対象ブロック間の参照数と一巡回数
Table 1　The number of references among target blocks and the number of whole block iterations.

対象ブロックサイズ	1 Mb	5 MB	10 MB
平均ブロック間参照数 [単位領域の参照数]	1,501[1,501.0]	5,908[1,181.6]	10,821[1,082.1]
一巡した回数	45	154	226

れは対象ブロックに含まれるオブジェクトが多いためである．ただし，単位領域（1 MB）あたりの参照数に換算すると，対象ブロックサイズが大きいほど参照数が少なくなる．これはブロックサイズを大きくすることでブロックをまたがない参照が増えることを意味する．

また，対象ブロックサイズが大きいほど対象ブロックを遷移し Tenured 領域を一巡する回数が多くなっている[⋆1]．これはブロックサイズが大きいと Tenured 領域を対象ブロックが遷移し一巡するのが早いためである．

以上のことから，対象ブロックが大きいほど単位領域あたりの対象外領域からの参照が少なく，また，ブロックをまたがった参照が存在する場合も Tenured 領域を一巡する回数が多いため，Tenured 領域の参照無効化が促進されると考えられる．

表 2 に 1 つのブロックを参照無効化する平均時間と Minor GC 時間および Major GC 時間に対する割合を示す．

⋆1 一巡する回数がブロックサイズに比例しないのは使用している Tenured 領域の端点とブロックの端点が一致しないからである．たとえば，使用中の Tenured 領域が 7 MB でブロックサイズが 1 MB なら Minor GC が 7 回実施されれば一巡する．しかし，ブロックサイズが 5 MB の場合，Tenured 領域を一巡するためには Minor GC を 2 回実施する必要がある．

表 2　平均参照無効化時間と GC 時間
Table 2　Average times of the nullification and the garbage collection.

対象ブロックサイズ	1 Mb	5 MB	10 MB
平均参照無効化時間 (msec)	3.5	6.1	9.2
Minor GC に対する割合 (%)	17.2	28.0	37.7
Major GC に対する割合 (%)	0.9	1.6	2.4

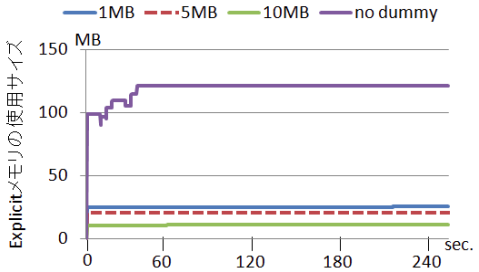


図 9　Explicit メモリの使用量
Fig. 9　Usage of explicitly managed heap memory.

参照無効化オブジェクト数に比例して参照無効化の処理時間を要する．参照無効化時間は Major GC 時間の 3%未満であることから実用的であると考えられる．なお，TBRA の全要素に占める dirty の割合は，いずれの対象ブロックサイズにおいても，0.1%未満であり，TBRA が高速化に有効であることを確認している．

Minor GC に対する参照無効化時間の割合は最大で 40%弱と大きくなっている．本論文では対象ブロックの大きさを固定としたが，参照の無効化時間はブロックサイズに比例することから，対象ブロックの大きさを参照の無効化の進んだ領域と無効化の進んでいない領域に鑑みて，可変長にするなどで性能を向上させることが考えられる．これに関しては今後の課題である．ブロックサイズが異なる場合における Explicit メモリの使用量の推移を**図 9** に示す．なお，no dummy は参照無効化を適用しない場合の Explicit メモリの使用量である．

対象ブロックサイズが大きいほど Explicit メモリを効率的に使用できている．最も使用効率の悪い対象ブロックが 1 MB の場合においても，参照無効化を用いない場合に比べてメモリ使用量が約 1/5 になっている．これはブロックサイズが大きいほど Tenured 領域の参

照無効化が促進され，そこからの参照が無効になるため，Explicit メモリに配置されたオブジェクトの要否判定精度が向上し，回収効率が向上していると考えられる．

また，同様の評価を Web アプリケーションフレームワークの Spring[7] および Seasar2[8] に対して実施した⋆1．その結果，同様の傾向が得られ，メモリ使用量をそれぞれ約 60%削減でき，提案手法の有効性を確認した．

5. 関 連 研 究

GC によって生じる停止時間を短縮する手法として Concurrent Mark-Sweep GC（CMS）[9] がある．これはミューテータと並行してコレクタを実行するため，プログラムの実行性能が大幅に低下するという問題がある．また，CMS では Tenured 領域の断片化が進むと，Major GC が発生し長時間の停止を免れない．Major GC を発生させにくくした Garbage First Garbage Collection（G1GC）[10] も提案されているが，実行性能が低下してしまうという問題は残っている．明示的メモリ管理機能に関しては，Explicit メモリの解放を高速化するため EHRA を導入している．この補助構造の修正はミューテータによるオブジェクトの参照変更時に実施する．その実行時オーバヘッドは 0.1%以下と十分小さい．G1GC と明示的メモリ管理機能は領域の開放について，Live オブジェクトの少ない領域を優先して解放している．Live オブジェクト数について，G1GC では Mark 処理により正確に調査し，明示的メモリ管理機能では EHRA の要素から Live オブジェクト数を概算している．このため，明示的メモリ管理機能の方が算出負荷が小さい．しかし算出の誤差が大きい場合，領域解放にともなう Live オブジェクトの退避が多くなる可能性があり，処理時間が大きくなる．G1GC は他の GC と同じくヒープメモリの使用量が一定値を超えたときに，GC により不要なオブジェクトを回収する．我々が提案している明示的メモリ管理機能では，オブジェクトが不要になったときにユーザが（もしくは Java VM が自動的に）解放するため，不要なオブジェクトが長期にわたりメモリ領域を占有し続けることはない．このため，明示的メモリ管理機能は Java システム全体のメモリ利用量を削減できるという利点がある．ただし，明示的メモリ管理機能はユーザが Explicit メモリを確保する点を指定する手間が必要である．

明示的メモリ管理や提案する Tenured 領域のブロック分割に関して，複数のメモリ領域を利用することから，多世代型の GC[5),11)] に似ている．多世代型 GC では Java ヒープを世代という概念で複数の領域に分割する．オブジェクトは GC 経過回数を age として記録し，その age に応じて，対応する世代領域に移動し管理される．すなわち，ガベージコレクタはオブジェクトの age を基準に管理している．それに対し，我々が提案する明示的メモリ管理は，ユーザ指定によって関連したデータごとに領域を分割し管理する．明示的メモリ管理はデータが不要になった領域を優先して解放するため，age を基準に領域解放する手法に比べて，メモリ利用効率が高い．また，提案する Tenured 領域をブロックに分割してオブジェクトの参照無効化する手法は，age で管理する必要はなく，一定サイズで領域を分割している点で異なる．

Tenured 領域内オブジェクトの要否判定を細切れに実施するという点で提案手法は Incremental GC[5),12)] の一手法に似ている．たとえば Mark-Sweep GC を Incremental に実施する GC の場合，Mark 処理を 1 度に実施せず，プログラム実行と同時に少しずつ実施する．すべての Mark 処理が終了した後，Sweep 処理もプログラム実行と同時に少しずつ実施する．これにより一括して Mark-Sweep GC を実施する場合に比べて停止時間を削減できる．しかし，Mark-Sweep GC の Incremental 実行で Tenured Garbage オブジェクトを検出するためには，最も時間の短いケースでも Tenured 領域内のすべてのオブジェクトに対して Mark 処理を完了する必要がある．最も時間を要するケースでは Tenured 領域内のすべてのオブジェクトに対して Mark-Sweep 処理が完了している必要がある．それに比べて提案手法では，対象ブロック内のオブジェクトに対する Mark 処理と dead オブジェクトの参照無効化処理が完了しているだけでよく，Tenured Garbage オブジェクトを早期に検出できる点で異なる．

6. ま と め

本論文では，低オーバヘッドで Tenured 領域内オブジェクトの要否を判定し，dead なオブジェクトからの参照を無効化する手法を提案した．

提案手法では，Tenured 領域を複数のブロックに分割し，ブロック間の参照のみに着目し，簡易的に要否判定を実施する．解放対象ブロックを変更することで，複数のブロックにまたがる参照構造を持つオブジェクトも要否判定可能である．要否判定により dead と判定できたオブジェクトを無効化するため，dead オブジェクトの領域を int 配列オブジェクトで上書きする．これにより容易に参照を無効化する．

提案手法をオープンソースのミドルウェアに適用した結果，Major GC 処理時間の 3%未満

⋆1 Spring で Explcit メモリを適用するのは Spring JDBC の RowMapperResultSetExtractor.extractData() メソッドで生成する ArrayList オブジェクトである．Seasar2 で Explcit メモリを適用するのは S2JDBC の BeanListResultSetHandler.handle() メソッドで生成する ArrayList オブジェクトである．

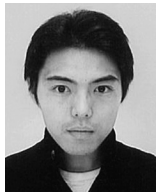
で Tenured Garbage オブジェクトを検出し，無効化できることを確認した．また，Tenured Garbage オブジェクトを無効化することにより，従来に比べて Explicit メモリ使用量を最大 1/5 に削減でき，提案手法の有用性を確認した．

ただし，Minor GC に対する参照無効化の処理時間は最大で 40%弱と大きい．参照無効化の処理時間は参照無効化の対象ブロックの大きさに比例する本論文の結果を参考に，対象ブロックの大きさを参照の無効化の進んだ領域と無効化の進んでいない領域に鑑みて，可変長にするなどで性能を向上させることが考えられる．これに関しては今後の課題である．

参　考　文　献

1) Gosling, J., Joy, B., Steele, G.L. and Bracha, G.: *The Java Language Specification*, Addison Wesley (2005).
2) 小幡元樹，西山博泰，足立昌彦，岡田浩一，長瀬卓真，中島　恵：Java における明示的メモリ管理，情報処理学会論文誌，Vol.50, No.7, pp.1693–1715 (2009).
3) 足立昌彦，小幡元樹，西山博泰，岡田浩一，長瀬卓真，中島　恵：Java における明示的メモリ管理機能の半自動適用技術，情報処理学会論文誌　プログラミング（PRO），Vol.3, No.2, pp.26–35 (2010).
4) Bak, L., Duimovich, J. and Fang, J.: The New Crop of Java Virtual Machines, *Proc. 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, Vol.33, pp.179–182 (1998).
5) Jones, R. and Lins, R.: *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, Wiley (1996).
6) Ungar, D. and Jackson, F.: An adaptive tenuring policy for generation scavengers, *ACM Trans. Prog. Lang. Syst.*, Vol.14, pp.1–27 (1992).
7) Spring (2010). http://www.springsource.org
8) Seasar2 (2010). http://www.seasar.org
9) Printezis, T. and Detlefs, D.: A generational mostly-concurrent garbage collector, *Proc. 2nd International Symposium on Memory Management*, pp.143–154 (2000).
10) Detlefs, D., Flood, C., Heller, S. and Printezis, T.: Garbage-First Garbage Collection, *Proc. 4th International Symposium on Memory Management*, pp.37–48 (2004).
11) Stefanovic, D., Hertz, M., Blackburn, S.M., McKinley, K.S. and Moss, J.E.B.: Older-first Garbage Collection in Practice: Evaluation in a Java Virtual Machine, *Proc. 2002 workshop on Memory system performance*, pp.25–36 (2002).
12) Wilson, P.R.: Uniprocessor Garbage Collection Techniques, *Proc. International Workshop on Memory Management*, pp.1–42 (1992).

(平成 22 年 7 月 5 日受付)
(平成 22 年 11 月 7 日採録)



足立　昌彦

2006 年神戸大学大学院修士課程修了．同年（株）日立製作所入社．同社システム開発研究所にて Java に関する研究開発に従事．

小幡　元樹（正会員）

1973 年生．2003 年早稲田大学大学院理工学研究科博士課程修了．工学博士．（株）日立製作所システム開発研究所主任研究員．Java に関する研究開発に従事．

西山　博泰（正会員）

1993 年筑波大学大学院工学研究科博士課程修了．工学博士．（株）日立製作所システム開発研究所主任研究員．最適化コンパイラ，Java 実行環境等言語処理系の研究に従事．ACM，IEEE 各会員．

岡田　浩一

1979 年生．2003 年信州大学大学院工学系研究科修士課程修了．同年（株）日立製作所入社．ソフトウェア事業部にて Java 仮想マシンに関する製品開発に従事．

長瀬　卓真

1981 年生．2006 年法政大学情報科学研究科修士課程修了．同年（株）日立製作所入社．ソフトウェア事業部にて Java 仮想マシンに関する製品開発に従事．

中島　　恵

1966 年生．東海大学工学部情報数理学科卒業．（株）日立製作所ソフトウェア事業部担当部長．1989 年入社以来，C/C++，Fortran コンパイラの製品設計，開発，および Java 仮想マシンの製品計画，開発を経て，2007 年よりアプリケーションサーバの製品計画，開発とりまとめに従事．