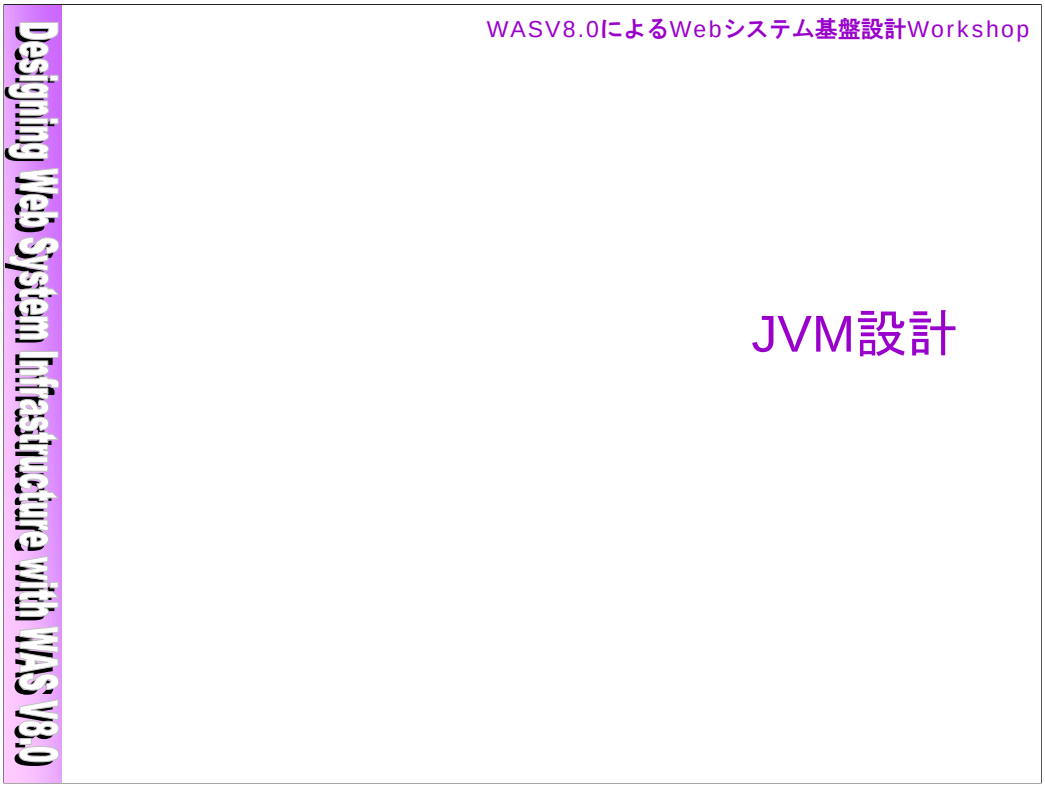

Designing Web System Infrastructure with WAS V8.0
WASV8.0によるWebシステム基盤設計Workshop
JVM設計

Designing Web System Infrastructure with WAS V8.0

当セッションの目的

- JVM設計に必要な知識の習得
  - JVM設計に影響するGCを理解する。
  - WAS32bit版とWAS64bit版の違いを理解する。
  - Dump Agentの構成について理解する。
  - その他，WAS V8.0のパフォーマンス向上機能を理解する。

2

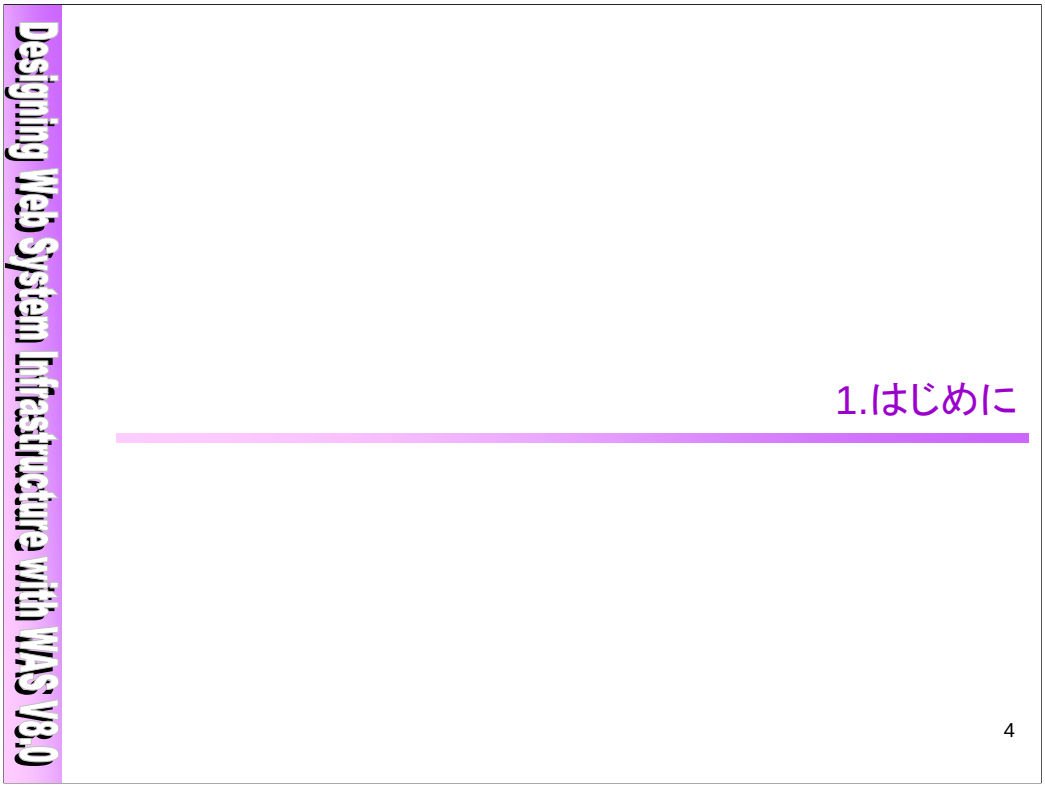
Designing Web System Infrastructure with WAS V8.0

Agenda

1. はじめに
2. GCの基本
3. GCポリシー
4. GCチューニング
5. WAS V8.0 64bit版
6. Dumpエージェントの構成
7. パフォーマンスの最適化

まとめ・参考文献

3

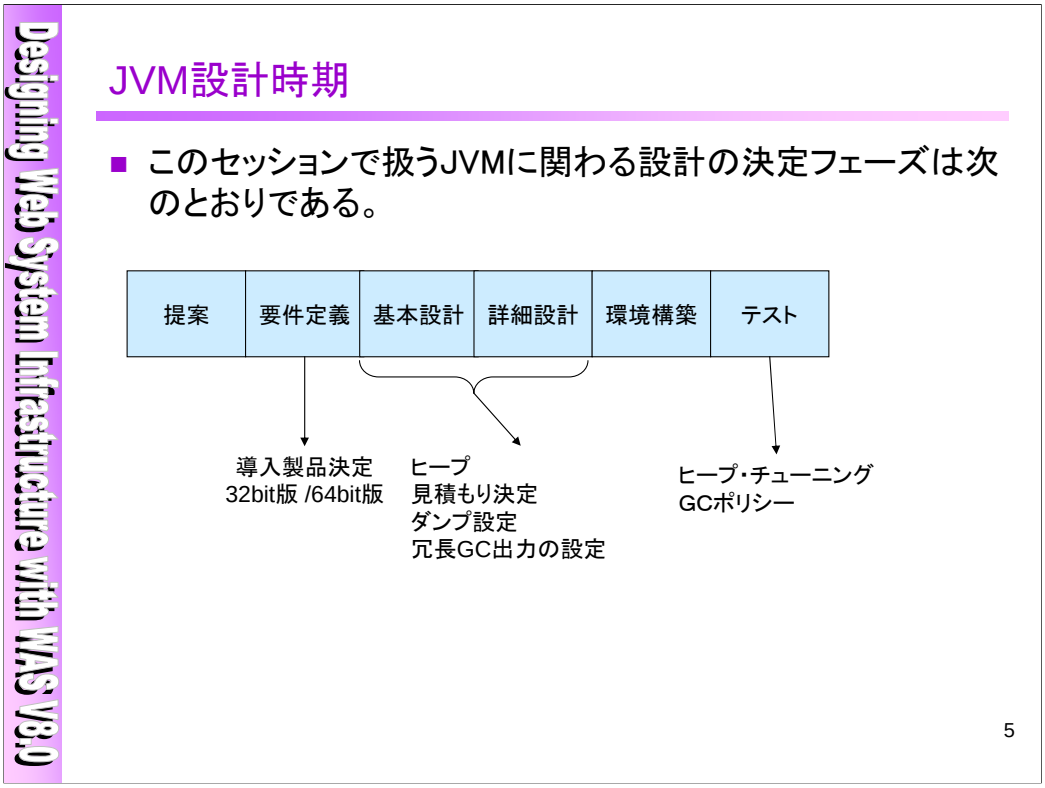


この章では，プロジェクトのどのフェーズまでにJVMに関係した何の事柄が決まっていれば良いかについてご説明致します。

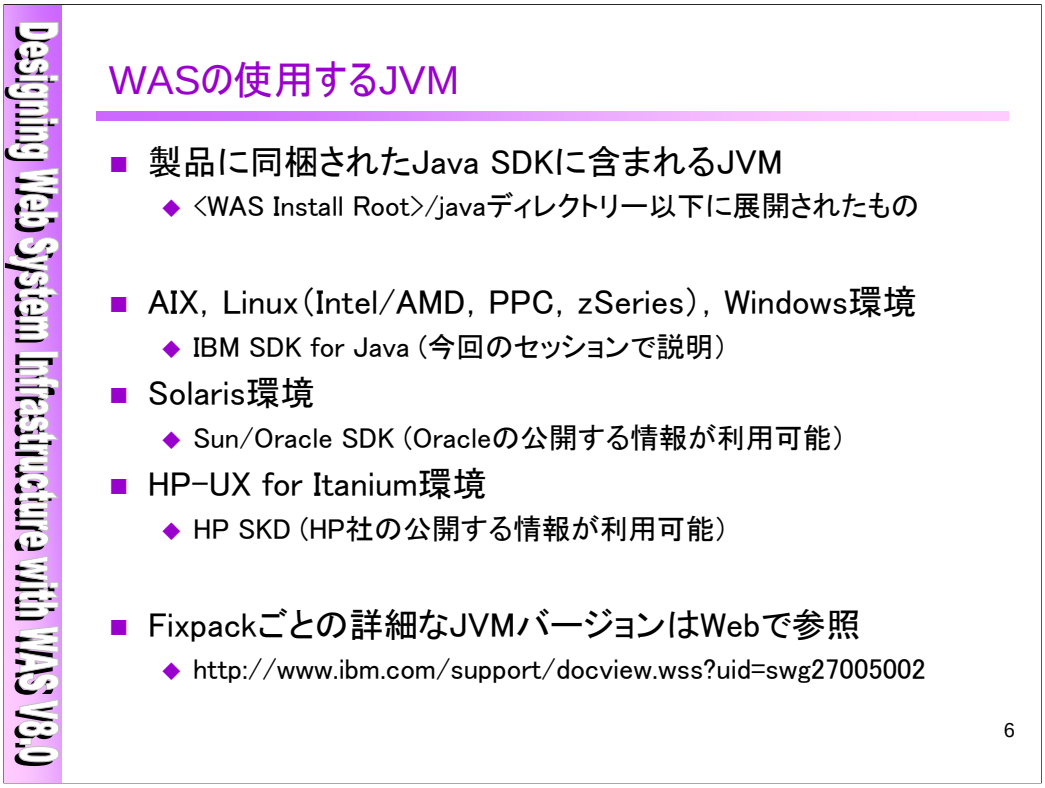
JVMとはJava仮想マシン(Java Virtual Machine)のことで，Javaバイトコードを実行するためのソフトウェアです。

Java言語で開発されたソフトウェアは，配布時にプラットフォームから独立した独自の形式（Javaバイトコード）になっているため，そのままでは実行することができません。そこでJavaプログラムを実行するためには，Javaのバイトコードを実行する仮想マシン・プログラムを各プラットフォーム上で作成され，そのうえで逐次解釈しながら実行されます。

また仮想マシンの実装によっては，頻繁に実行されるバイトコードを，そのプラットフォーム固有の形式（ネイティブコード）に動的に変換するコンパイラーを用意して，高速化を実現しているものもあります。このようなコンパイラーをJIT（Just in-Time）コンパイラーといいます。



このセッションで扱うJVMに関する設計の決定時期はチャートのとおりです。バージョン，エディション，32bit版/64bit版などの導入製品の選択は，設計が始まる前までに決定します。その後，環境構築までにだいたいのヒープ・サイズの見積もりをし，最後にテストフェーズで十分なパフォーマンステストを実施しGC頻度や長さから最適なヒープ・サイズを決定します。

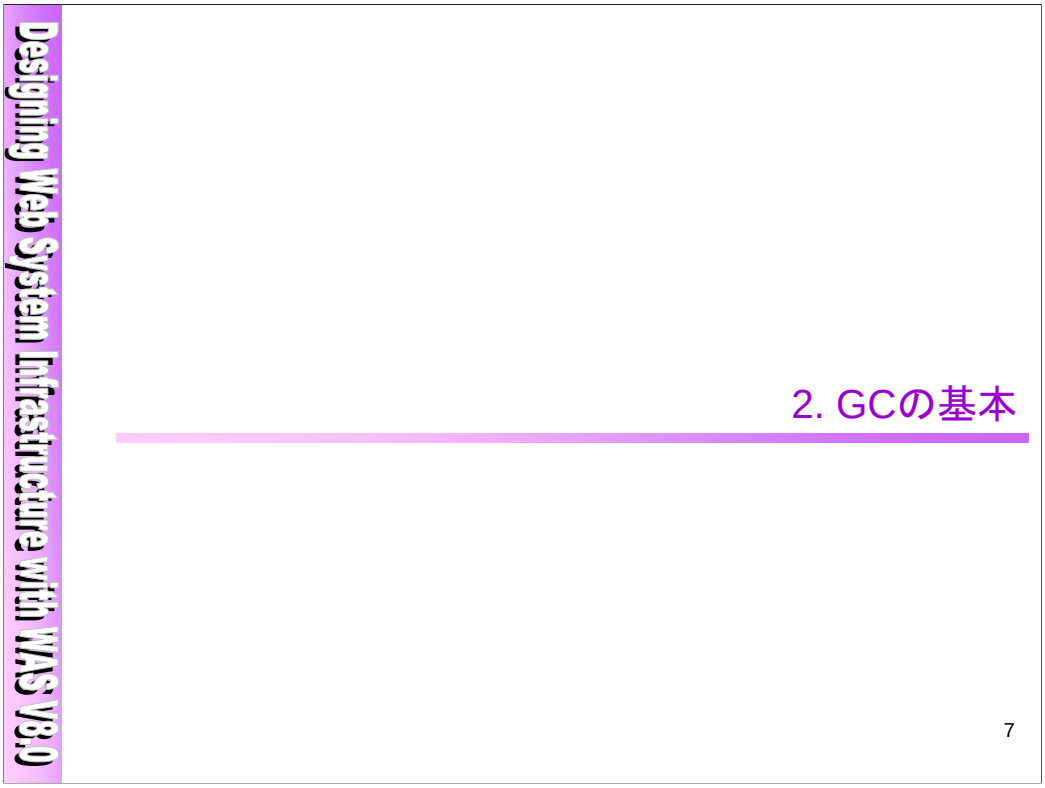


なお，WASの実行に使用するJVMは，製品に同梱されているものに限定されます。これらのJava実行環境は，ユーザーが独自に入れ替えることなどはできず，あくまでWASのFixpackなどの修正でのみ更新されます。

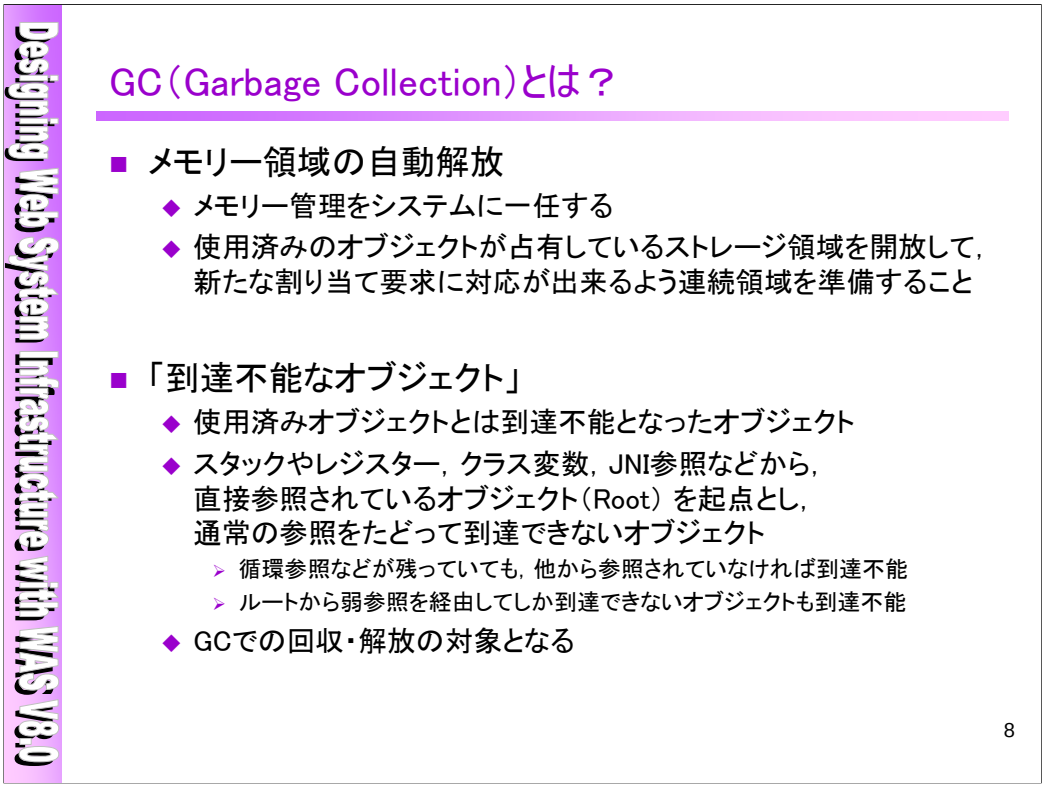
WASにどのようなJVM，Java SDKが同梱されているかは，プラットフォームによって異なります。AIX，Linux(Intel/AMD，PPC，zSeries)，WindowsプラットフォームのWASでは，IBMが独自に実装したJVMが使用されています。この独自JVMを含むIBM SDK for Javaでは，Sun/Oracleが提供している標準のJDKにはない多くの独自機能が利用できます。この資料では，これらのプラットフォームのIBM SDK for Javaについて解説します。

SolarisプラットフォームおよびHP-UXプラットフォームのWASでは，それぞれOracle社，HP社によって実装されたJVMが使用されています。これらのJVMについては，それぞれのベンダーが提供している問題判別資料やチューニング情報がそのまま利用できます。

WASのFixpackごとに使用されているJVMのバージョン番号は，Webサイトで確認できます。



この章では，基本的なGCの役割や動きを説明します。



Garbage Collection（通称GC）とはJVMによるメモリー管理機能です。GCによって不要なオブジェクトは削除され，メモリー領域が解放されます。それにより，新たなオブジェクト割り当て要求処理を行うことが可能になります。C言語のようにプログラマが明示的にヒープの確保と開放を行うのではなく，それらの作業をJVMに委譲することでより安全なプログラムの作成が行えるようになっています。

GCの際に不要と判断されるのは「到達不能なオブジェクト」です。Javaヒープの外部から直接参照されているオブジェクトをルートオブジェクトと言います。そこから直接・間接的に参照されているオブジェクトが到達可能なオブジェクト，それ以外のオブジェクトが到達不能オブジェクトです。

Javaアプリケーションの実行環境であるJVMが使用する仮想ストレージは一般にヒープと呼ばれています。IBM SDK for Javaの使用しているJVMはIBM SDK for Java 5.0から採用されたJ9 VMと呼ばれるJVMです(以後，特に断りのない限り，JVMとはWASに搭載されているJava仮想マシンを指し示すものとします)。また，JVMのサブコンポーネントでヒープの状態管理(メモリー管理)を担当しているのはSTコンポーネントと呼ばれます。

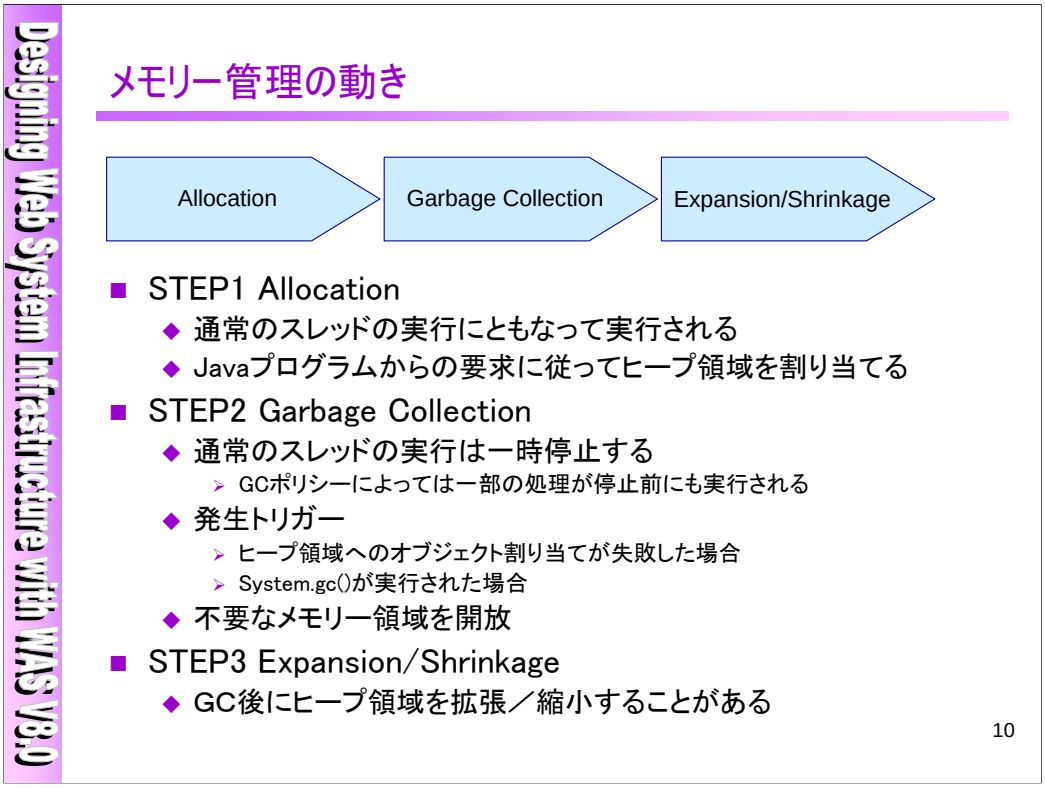
ヒープには，格納するオブジェクトの種類によって2種類のヒープが存在します。ひとつはリクエストを実施するときにアプリケーションによって生成されるオブジェクトを格納するためのJavaヒープであり，Garbage Collectionによるメモリー管理は通常このJavaヒープの状態を管理します。この領域は最小ヒープ・サイズ・最大ヒープ・サイズ(-Xms/-Xmx)の範囲で増減されます。Javaヒープには，一時オブジェクト(毎回のGCサイクルの中で消滅)，キャッシュ・オブジェクト(設定によりサイズが変動)，プールされたオブジェクト(トラフィック状況やタイムアウトによりサイズが変動)，プロセスの消滅まで存在し続けるオブジェクトなどが格納されます。

もうひとつはアプリケーションの稼働環境に必要なシステムや共有ライブラリーのクラスオブジェクトを格納するためのNative Heapです。Native Heapは，格納されるオブジェクトの性質上Garbage Collectionの対象になりません。また，ストレージコンポーネントは必要な分だけ領域を確保するような動きをとるため，Javaヒープのようにサイズの最大値，最小値を設定することはできません。

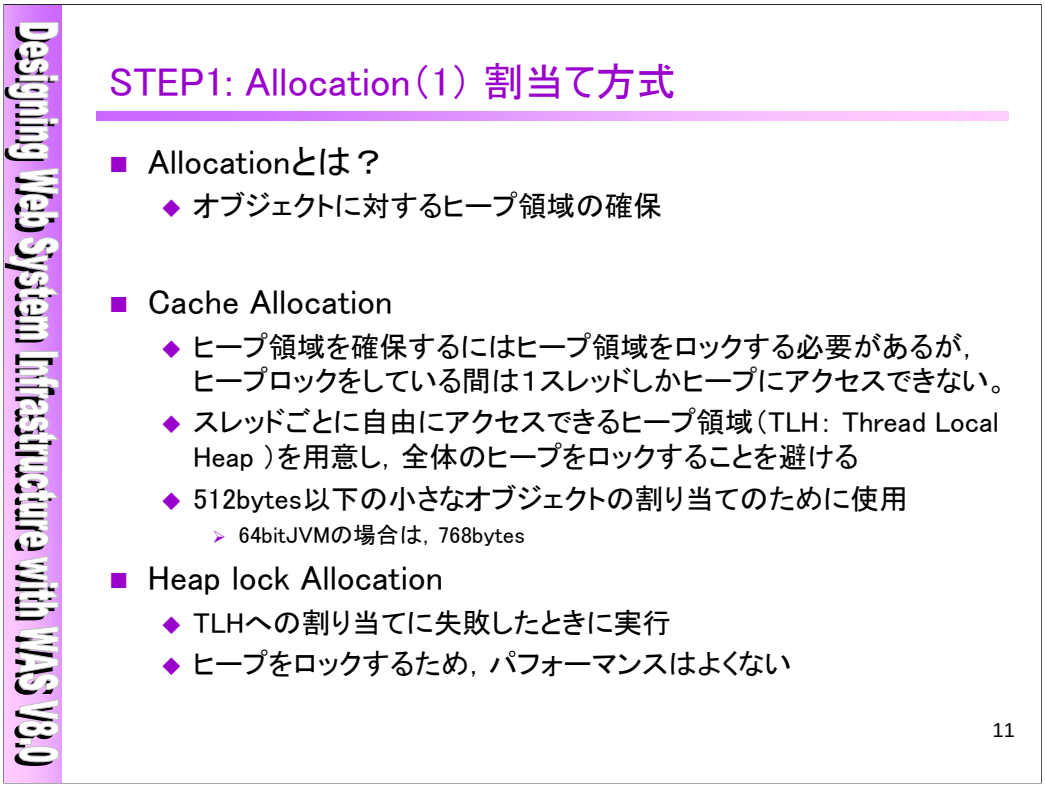
WASV6.1からはNative heapの初期値が，JVM引数-Xinitshで設定することができなくなっていますので，ご注意下さい。

PK66599: THE JAVA VIRTUAL MACHINE FOR WEBSPHERE APPLICATION SERVER VERSION 6.1 DOES NOT SUPPORT THE XINITSH ARGUMENT

http://www.ibm.com/support/docview.wss?rs=180&uid=swg1PK66599



メモリー管理は次の3つのフェーズから成り立っています。まず，オブジェクトにヒープ領域を割り当てるAllocationフェーズ，その割り当てに失敗した場合に発生するGCフェーズ，最後にGC実施後，十分な空き領域が確保できなかった場合にヒープの拡張や，十分すぎるメモリー領域が確保されていた場合はヒープの収縮が発生します。次ページ以降で詳しく説明します。

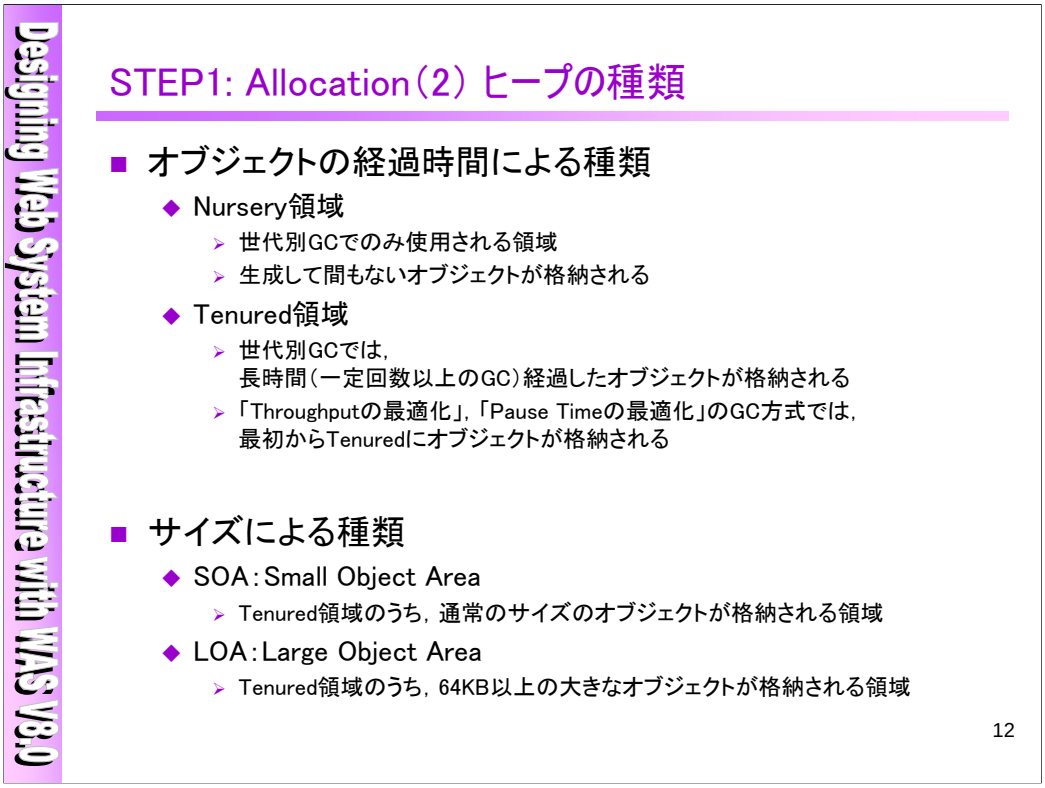


メモリー管理の最初のステップとして，まずAllocationがあります。

Allocationには二種類存在します。ひとつは他のスレッドによる割り当てと競合しないようにヒープにロックをかけて割り当てを行う方法で，もうひとつはヒープにロックをかけることなく割り当てを行う方法です。

ヒープをロックすることなく割り当てる方法をcache allocationと呼びます。実行スレッド毎に予め確保されているキャッシュ領域（Thread Local Heapと呼ぶ）を利用してそこにオブジェクトを割り当てます。基本的に512bytes以下（JVM64bitでは768bytes以下）の小さなサイズのオブジェクトの割り当てのために使用されますが，その時点で使用しているキャッシュ領域に空きがある場合は512bytesを超えるサイズのオブジェクトの割り当ても行われます。ヒープをロックすることなく割り当てが出来るため，小さなオブジェクトの割り当てが多い場合にパフォーマンスの向上が期待できます。TLHがいっぱいになって割り当てが出来なくなった場合に備えて，スレッドはTLH用の領域を40kBの大きさを上限としてJavaヒープから別途確保しており，小さなオブジェクトの割り当て要求が来た際にTLHが一杯だった場合はその新しい領域を使用します。

オブジェクトの割り当ての際に他のスレッドとの競合を避けるためにヒープにロックをかけて割り当てを行うのがHeap lock Allocationです。このAllocationは割り当てるオブジェクトのサイズが512bytesを超える場合で，かつTLHへの割り当てが出来ないときに発生します。

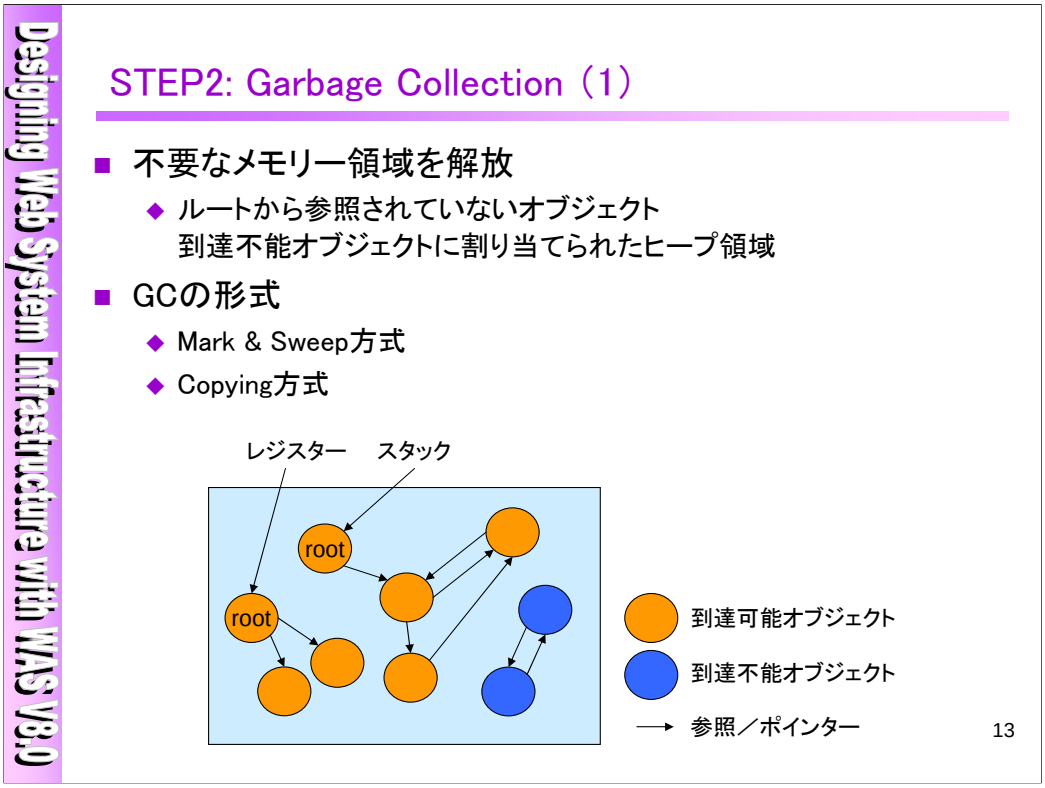


Allocationで取得されるヒープの種類にはいくつかの分類があります。

一つ目は，格納されるオブジェクトの寿命に注目した分類です。これは世代別GCを使用した場合のみ使用されます。Nursery領域は，新規に作成されたオブジェクト，およびまだ存続時間が短い（生き延びたGCサイクルの回数が少ない）オブジェクトが格納されます。一方，Tenured領域には，存続時間の長い（一定回数以上のGCサイクルを生き延びた）オブジェクトが格納されます。GCでは，作成されたばかりのオブジェクトが回収される可能性が高く（一時オブジェクト），長命のオブジェクトほど回収される可能性が低い，という性質に注目した分類です。通常のGC（Scavenger GC）では，Nursery領域のみを回収の対象にすることで，GCの時間を短くする効果があります。

世代別GCではない，「Throughputの最適化」「Pause Timeの最適化」のGCポリシーを使用している場合は，最初からTenured領域にオブジェクトが割り当てられます。

もう一つの分類がオブジェクトのサイズによるものです。大きなオブジェクトと小さなオブジェクトが混在しているとヒープの断片化を招きやすいため，IBM JVMではTenured領域をオブジェクトのサイズによって二つに分割しています。これらの領域の割合は自動調整されます。短命の大きなオブジェクトが多数使用されるアプリケーションでは，LOAのサイズを大きめに固定した方がパフォーマンスがよくなるケースがあります。



メモリー管理の2つ目のステップとして，GCがあります。

ヒープ内のオブジェクト管理方法は，「生きている」オブジェクトがどこにあるのかを管理し，「生きている」オブジェクトが参照していないオブジェクトをガーベッジと判断します。ガーベッジと判断されたオブジェクトの領域は不要とされ，メモリー解放されます。

ヒープ中から到達可能オブジェクトを選び出し，それ以外を解放する方式としては，Mark & Sweep方式とCopying方式の二つがあります。

Mark & Sweepには大きくわけて3つのフェーズが存在します。Markフェーズでは，RootおよびRootから到達可能なオブジェクトに対してマーキングされます。ここでマークされなかった，つまり到達できないオブジェクトについてはSweepフェーズでヒープ上から削除され，不要メモリーが解放されます。

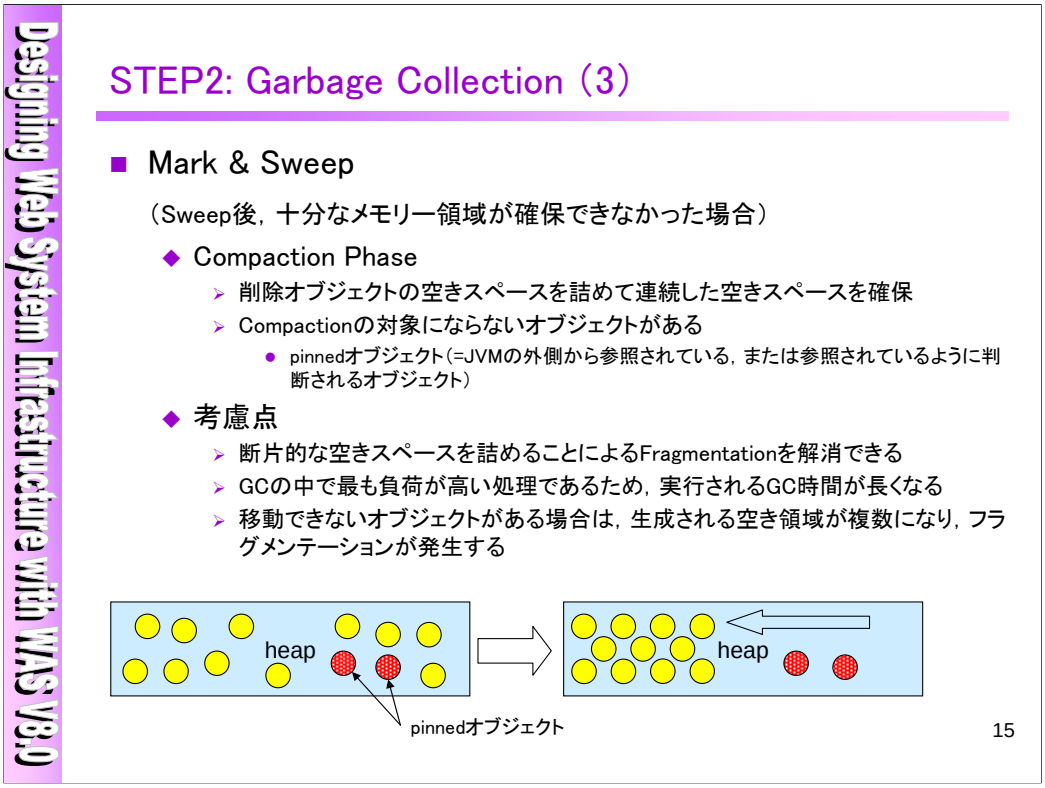
Mark & Sweepは，GCスレッドが稼動する際にすべてのアプリケーション・スレッドが停止するため，GCの負荷が1箇所に集中されます。

・Markフェーズ

まずRootを全走査し，Rootにしるし(Mark)をつけます。次にヒープ領域のすべての到達可能な「生きている」オブジェクトにしるし(Mark)をつけます。

・Sweepフェーズ

しるしのついていないオブジェクトを，ヒープ内から掃きとり(Sweep)ます。このとき，ヒープ領域内すべてのオブジェクトに対して走査が行われます。



Compactionフェーズでは，ヒープ上に存在しているすべてのオブジェクトをヒープのアドレス領域の一端にまとめる作業が行われます。MarkフェーズとSweepフェーズはそれぞれそれほど大きな負荷とならないため毎回のGCで実施されます。それに対してCompactionフェーズはオブジェクトの移動が伴うため個々のオブジェクトのコピーとメモリー・アドレス情報の再構成が行われます。そのため先の二つのフェーズと比べると負荷の大きな作業になるため，Compactionフェーズは発生条件が整ったときにのみ実施されます。例えばMark & Sweepが行われた後にヒープ・アロケーションを行えるだけの十分な空き領域が確保できなかった場合や，解放できた領域が少なくて全体の空き領域が一定以下だった場合です。また，実際にはCompactionの際に動かせない(Pinned)オブジェクトが存在します。その場合，空き領域が複数に分かれて存在することになります。

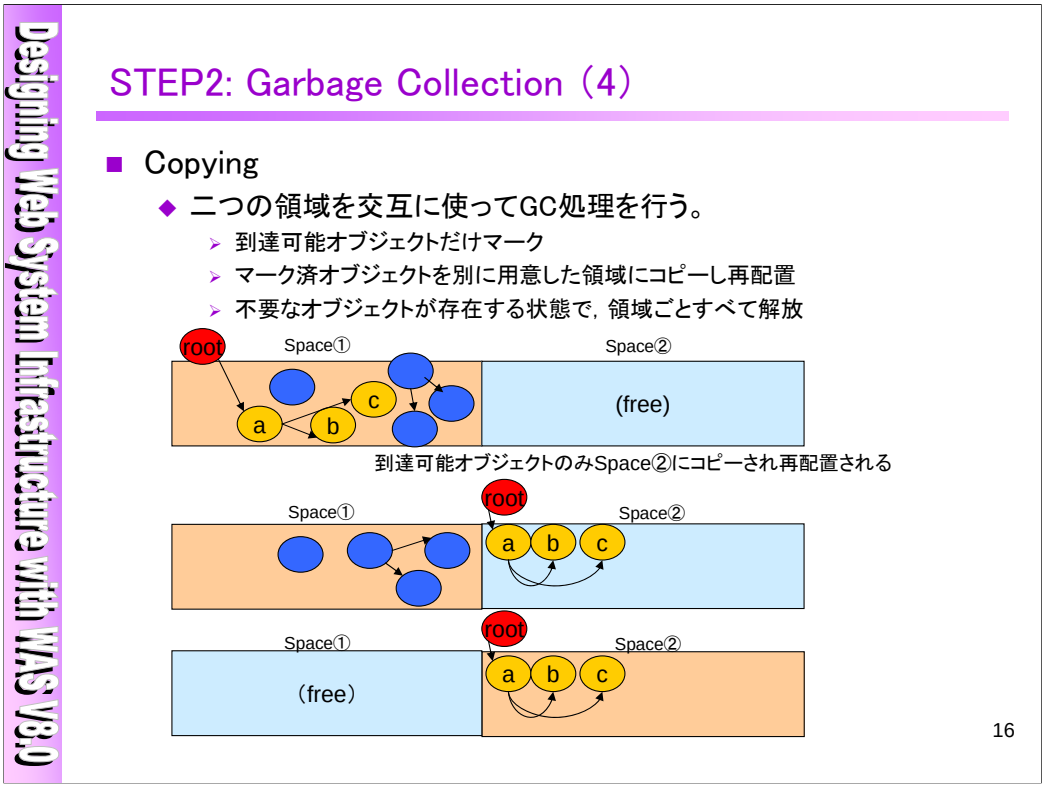
コンパクションを行わないオプション-Xnocompactgcも提供されていますが，以下の場合においては-Xnocompactgcは無視されます。

・明示的に-Xcompactgcが定義されている場合

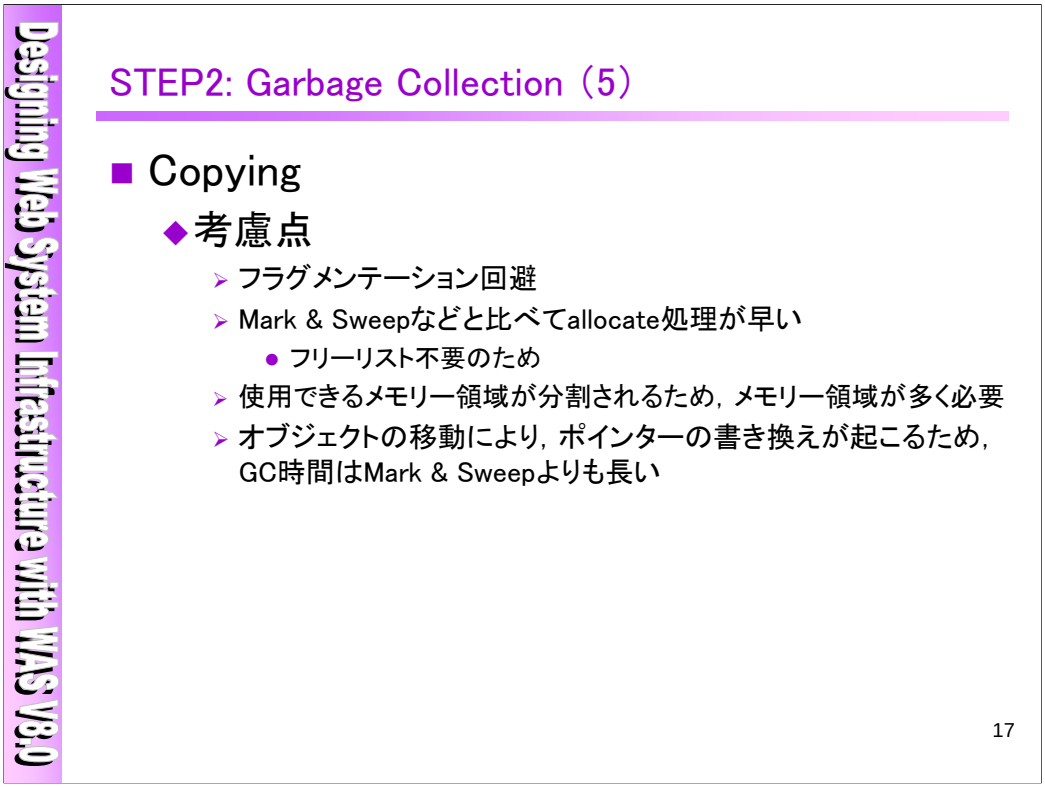
・Sweep phaseでリクエスト・オブジェクト割り当てに必要な空きスペースが無いと判断された場合

・System.gc()が実行され，かつ前回のGCでCompactionが行われなかった場合

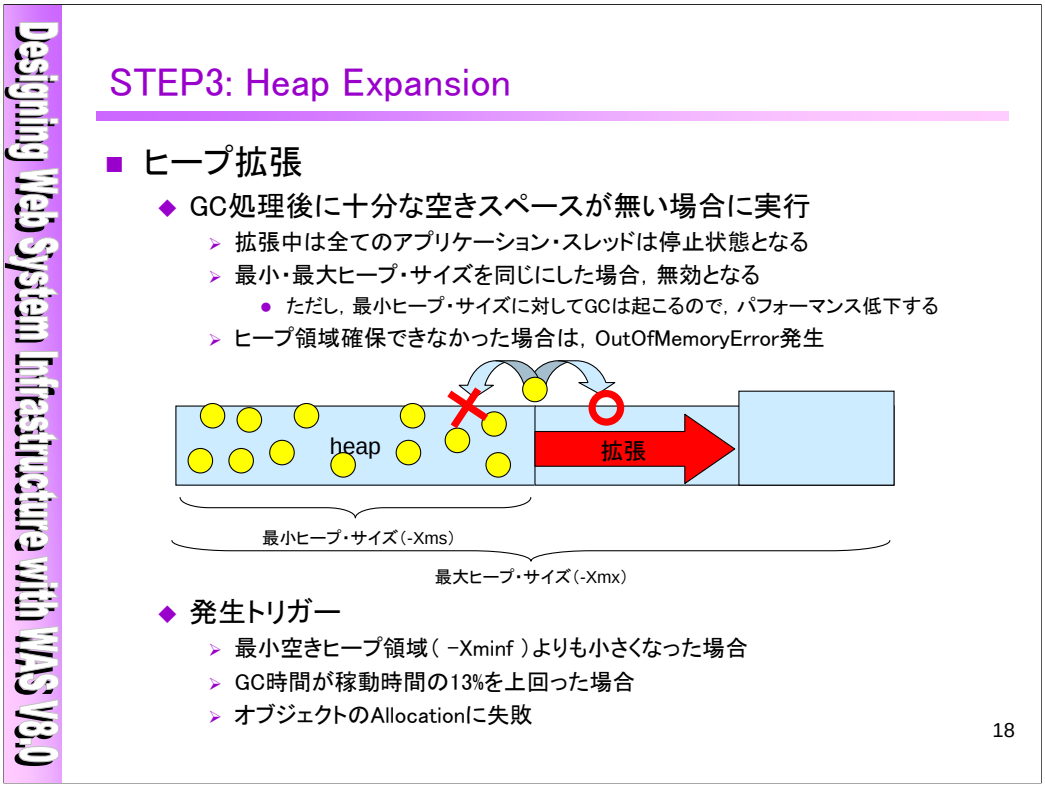
・Active heapの空き領域が5%未満もしくは128KB未満の場合



Copying方式では二つの領域を用意して，交互に領域の役割を与えGCを行います。まず，片方の領域でヒープ割り当てを行います。割り当てに失敗するとGCが開始され，使用中のオブジェクトにMarkします。このMarkするところまではMark & SweepのMarkフェーズと同様です。Copyingでは，このMarkフェーズ以降の動きが異なります。Markされたオブジェクトは未使用であったもう片方のヒープ領域にコピーされ，さらにもともと空領域だったためアドレスの端からオブジェクトを再配置します。すべての到達可能なオブジェクトがコピーされたところで，元のヒープ領域は不要オブジェクトのヒープ領域ごと廃棄され空領域となります。その後は，コピー先で割り当てに失敗するとGCが発生し，同じサイクルで各ヒープ領域の役割が交代します。



Copyingでは，メモリー解放後のヒープには常に連続した空き領域が確保されている状態になります。そのため，Mark & Sweepのように空き領域の検索作業が不要になります。つまり通常のリクエスト処理のパフォーマンスが向上します。また，ヒープ領域全体が空き領域になるのでフラグメンテーションも回避することができます。ただし，良いことばかりではありません。常に余分の空き領域を持っている分OSに対して要求するヒープ用のアドレス空間は大きくなります。また，オブジェクトのコピーとアドレスの付け替えがGCの度に起こるので，GC時間はMark & Sweepよりも長くなります。



GCの発生後，Allocationを行えるようにするためJVMはヒープ領域の拡張を試みることがあります。この際の拡張サイズは最大ヒープ領域（-Xmx）にて指定したヒープ・サイズを限度として行われます。拡張を実行しても十分な空き領域が得られなかった場合はOutOfMemoryErrorとなります。

拡張は負荷の高い処理であるため拡張を無効にすることも可能です。無効にする場合は最小ヒープ・サイズ（-Xms）と最大ヒープ・サイズ（-Xmx）を同じ値に設定します。しかし，実際に使用されるヒープが少ないにもかかわらず最小ヒープ・サイズを大きい値に設定すると，最初のGCに非常に長い時間がかかってしまいパフォーマンスが低下することから推奨はされていません。拡張を無効にする場合には，使用量に応じた最適なあたいに固定するようにします。

ヒープ拡張の発生トリガーは次のとおりです。

①Java heapの空き領域が最小空きヒープ領域よりも小さくなった場合

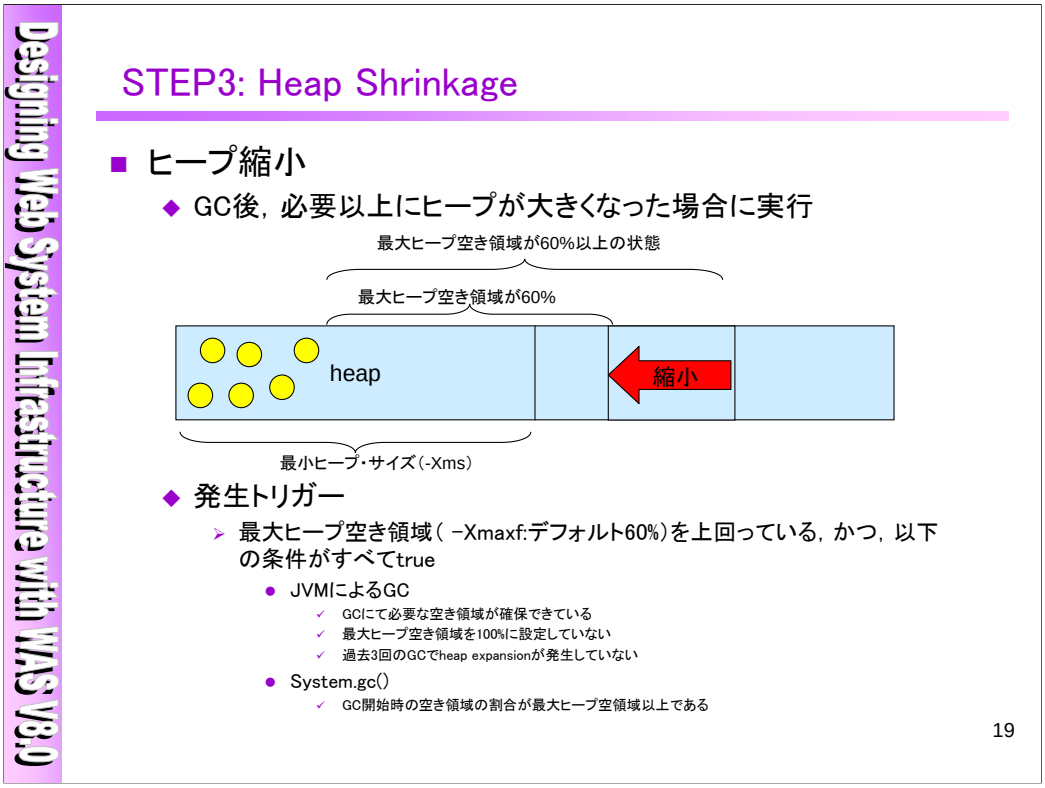
最小空きヒープ領域（-Xminf）のデフォルトは0.3であるため，Java heapの30%を空き領域として確保できるまで拡張は行われます。このとき，拡張しなければいけないサイズが最大拡張サイズ（-Xmaxe）よりも大きい場合には，最大拡張サイズ分だけ拡張します。最大拡張サイズのデフォルトは無制限です。一方，拡張しなければいけないサイズが最小拡張サイズ（-Xmine）より小さい場合には最小拡張サイズ分を拡張します。デフォルトは1MBです。

②GC時間が稼動時間の13%を上回った場合

GC時間がアプリケーション稼動時間の13%を上回った場合は，GC時間が長くなっているとJVMが判断し最小拡張サイズ分だけヒープを拡張します。拡張後の空き領域の割合は最大ヒープ空き領域を超えることはありません。最大ヒープ空き領域のデフォルトは60%です。

③オブジェクト割り当て失敗

どのような状況にしても，Allocationに失敗した場合には，リクエスト・オブジェクトが格納できるサイズ分が拡張されます。



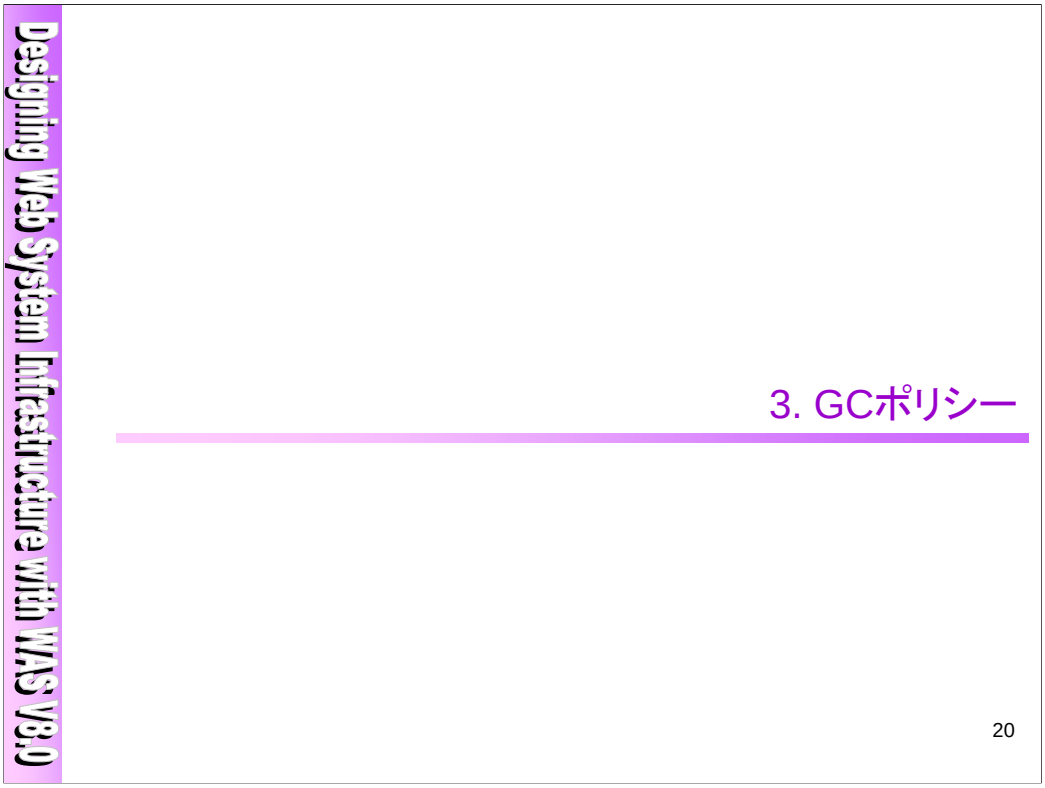
GCによる空き容量の確保の結果，ヒープが必要以上に大きくなった場合は最小空きヒープ領域(-Xms)で指定した最小値を限度としてヒープの収縮が行われます。ヒープ収縮が実施される条件は最大ヒープ空き領域(デフォルト60%)を上回っており，チャートの条件がすべてtrueになった時のみです。縮小されたヒープは最小ヒープ・サイズ(-Xms)よりも小さくはなりません。

また，ヒープ縮小処理前に以下の全ての条件がtrueの場合はCompactionが実施される場合があります。

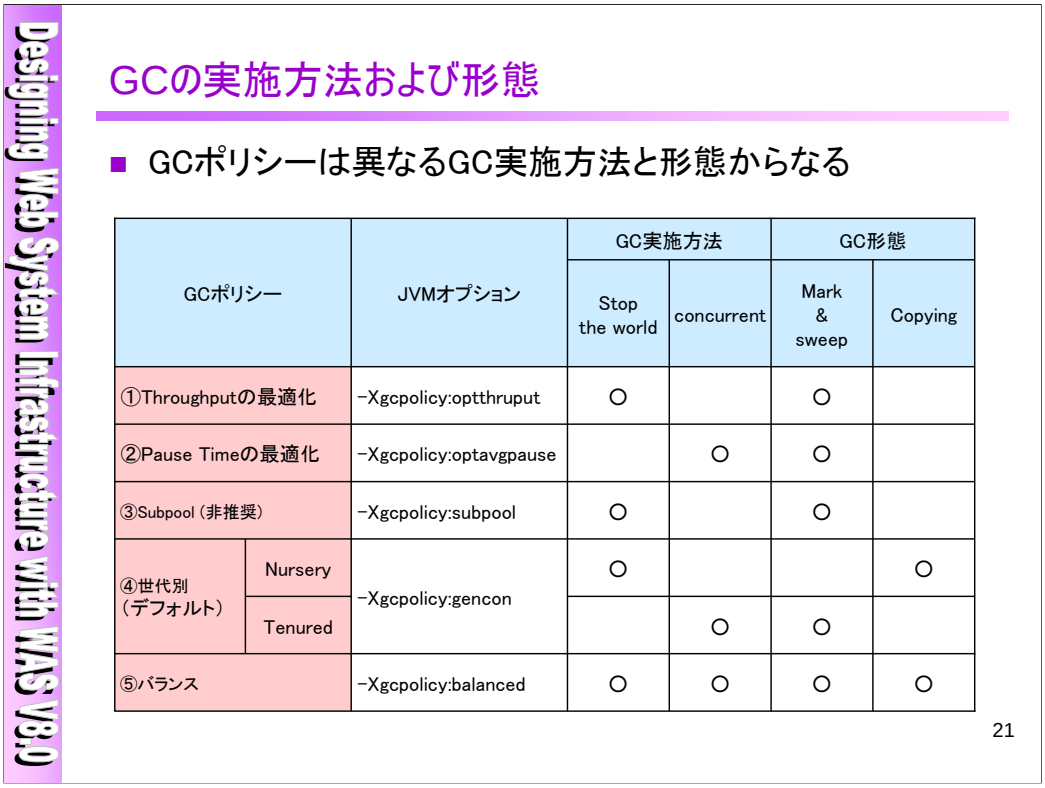
・直前のGCでCompactionが発生していない

・ヒープ後半に(断片化されている)空き領域が存在していない，もしくはヒープ後半に(断片化されている)空き領域の合計サイズが縮小するサイズの10%以下である

・前回のGC処理にて，Compactionも縮小処理も発生していない



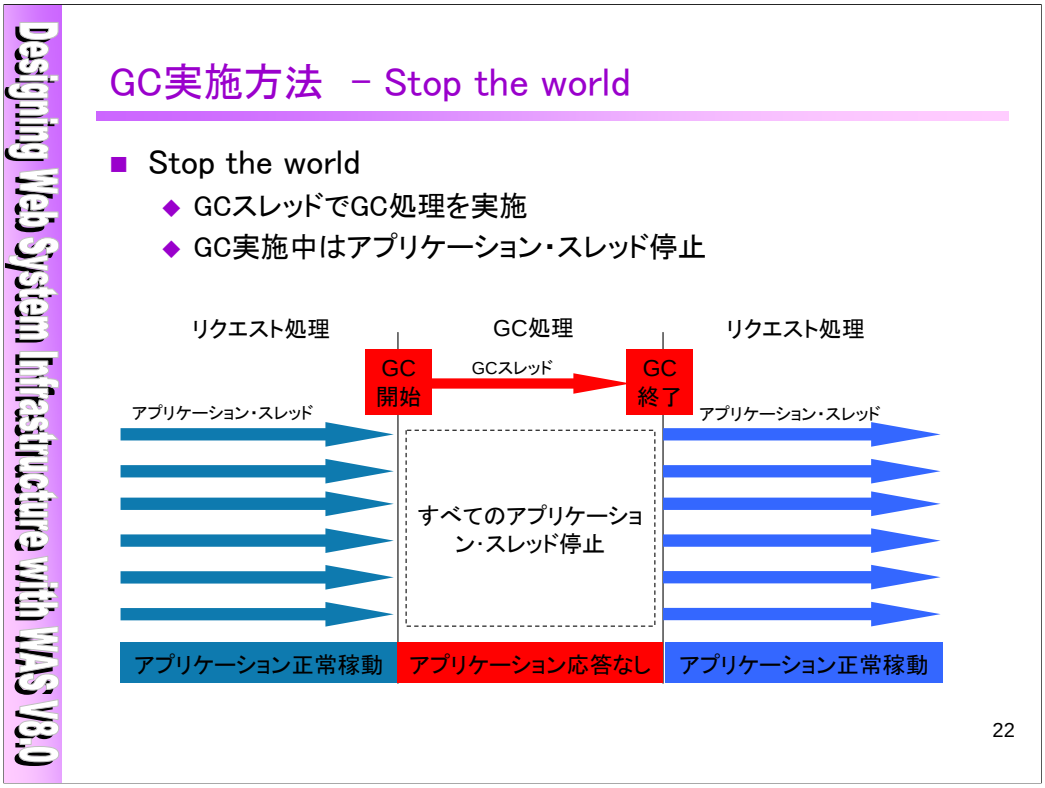
この章では，WASで設定可能なGCポリシーについて説明します。



GCポリシー		JVMオプション	GC実施方法		GC形態	
			Stop the world	concurrent	Mark & sweep	Copying
①Throughputの最適化		-Xgcpolicy:optthruput	○		○	
②Pause Timeの最適化		-Xgcpolicy:optavgpause		○	○	
③Subpool（非推奨）		-Xgcpolicy:subpool	○		○	
④世代別（デフォルト）	Nursery	-Xgcpolicy:gencon	○			○
	Tenured			○	○	
⑤バランス		-Xgcpolicy:balanced	○	○	○	○

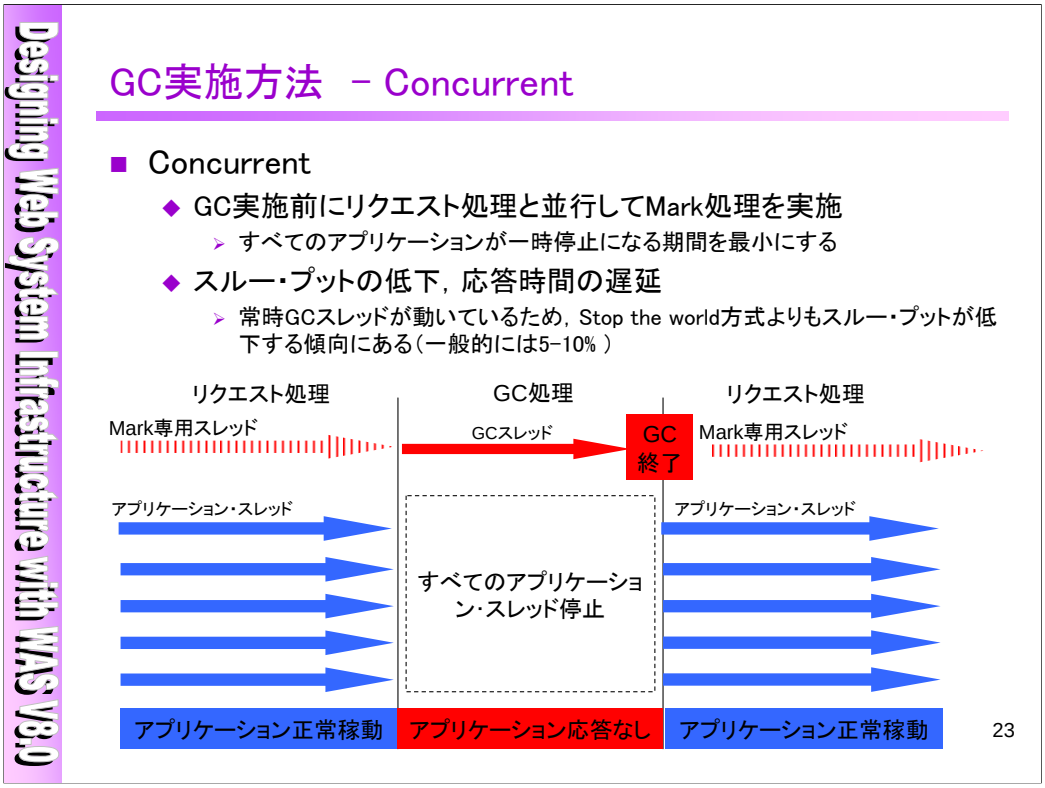
ヒープ割り当てやGCのアルゴリズムについてはこれまで沢山の研究が行われてきており，多種多様な手法がこれまで生み出されています。WAS V8.0で使用されているJ9 VMではチャートにあるように5つのGCポリシーが利用可能となっています。WAS V8.0で使用されるIBM JREでは，GCポリシーのデフォルトが従来のThroughputの最適化（optthruput）から世代別（gencon）に変更になりました。また，AIXプラットフォームで使用することができたSubpolが非推奨となり，かわりにバランスGC（balanced）が追加されています。

使用するGCポリシーの指定は，WASのJava仮想マシンに対する設定（汎用JVM引数）で行います。



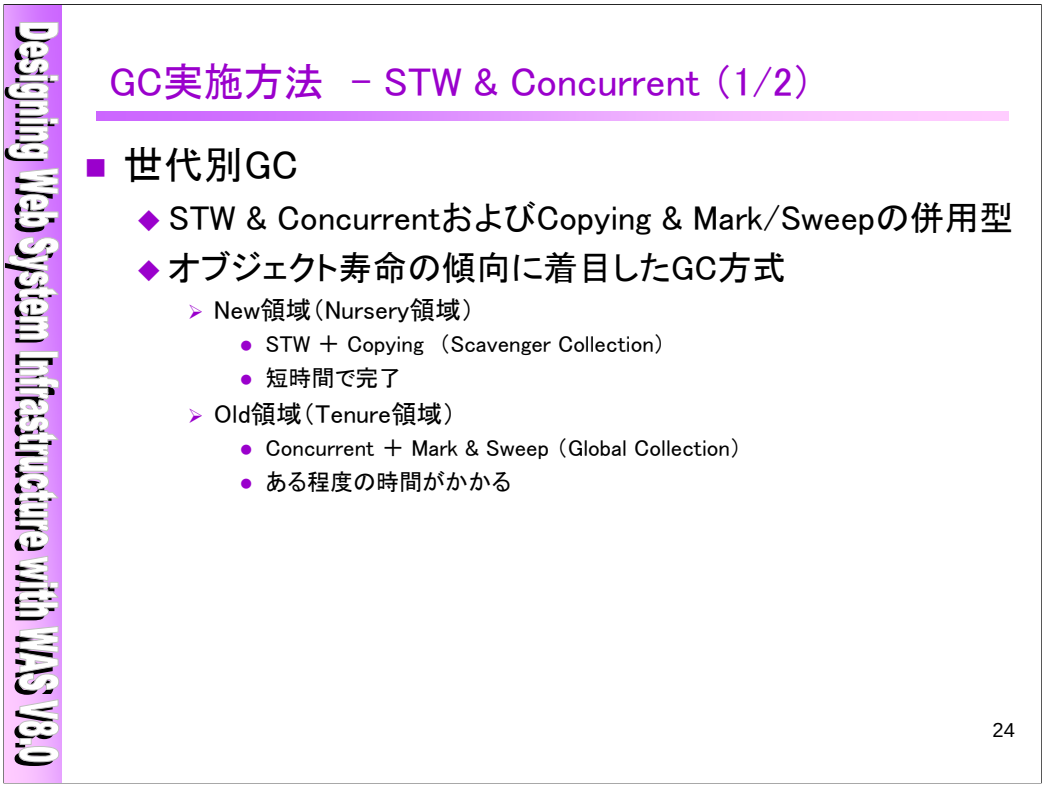
ここから「誰が何時GCを行うか」の説明を行います。まずはデフォルトのStop The World(STW)方式です。

GCはヒープ割り当てが失敗したときに発生します。その際，割り当てに失敗したJavaスレッドがGC作業を引き続き行うのですが，STW形式ではGCが開始された直後に，GCを担当するJavaスレッド以外のスレッド作業をすべて停止させます。GC処理が終了するまで他のスレッドは停止させられたままなので，GC作業を行うJavaスレッドの外では，言わば「世界が止まった状態」であるかのように見えるのでこのようなネーミングになっています。当然リクエストの処理もGCが終了するまでストップします。



Concurrent方式は，Stop the world で問題となっていたアプリケーション・スレッドの停止時間をできるだけ最小にした方式です。この方式を使用すると，Mark処理，Sweep処理，Compaction処理のうち，Mark処理をアプリケーション・スレッドと並行して行います。そうすることで，アプリケーションの停止時間を短くします。ただし，この方式を使用してもGCによりアプリケーション・スレッドが停止する時間は発生します。

具体的には，リクエストの処理中に優先度の低いスレッドをMark処理専用に別途用意し，CPUを利用できる時間を見つけてMark作業を先行して行います。GCの一部の作業を事前に行うことから，ConcurrentではSTW方式の時よりも，「世界を止めて」GC処理をする時間が理論上短くなります。ただし，これはあくまでも理論上の話で，実際はアプリケーションのデザインやシステムリソース，システムの利用状況などによって有効性は異なります。最悪の場合はSTWよりも効率が悪くなってしまう可能性もまったくないわけではありません。また，バックエンドとはいえ通常時にGC作業を先行して行う分，余剰CPU時間があまりない場合はリクエスト処理のスルー・プットに影響が及ぶ場合もあります（一般に5-10%の性能低下が見られると言われています）。

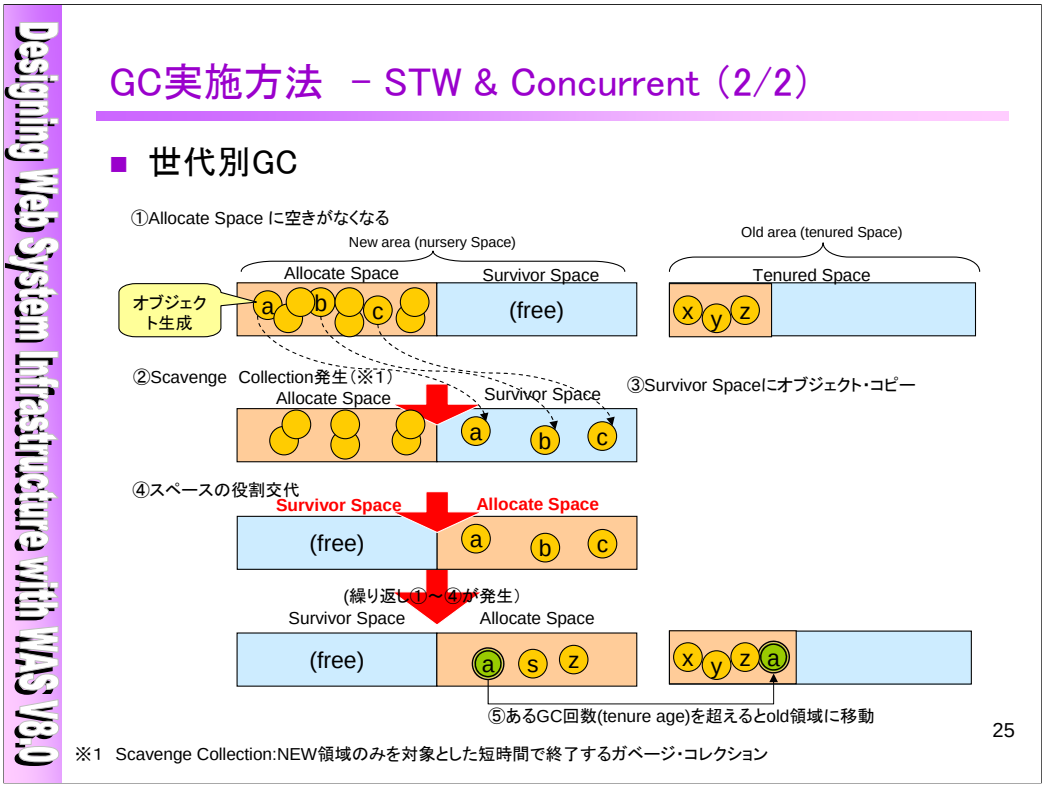


世代別GCのポイントはオブジェクトの生存期間に着目した点です。

リクエストの処理のため生成されるオブジェクトのように，一時的な目的のために生成されたオブジェクトは必要でなくなるまでの期間が短いという特徴があります。その一方でWASの運用そのものに必要なオブジェクトのようにWASプロセスの生存期間中ずっと必要となり続けるオブジェクトも存在します。

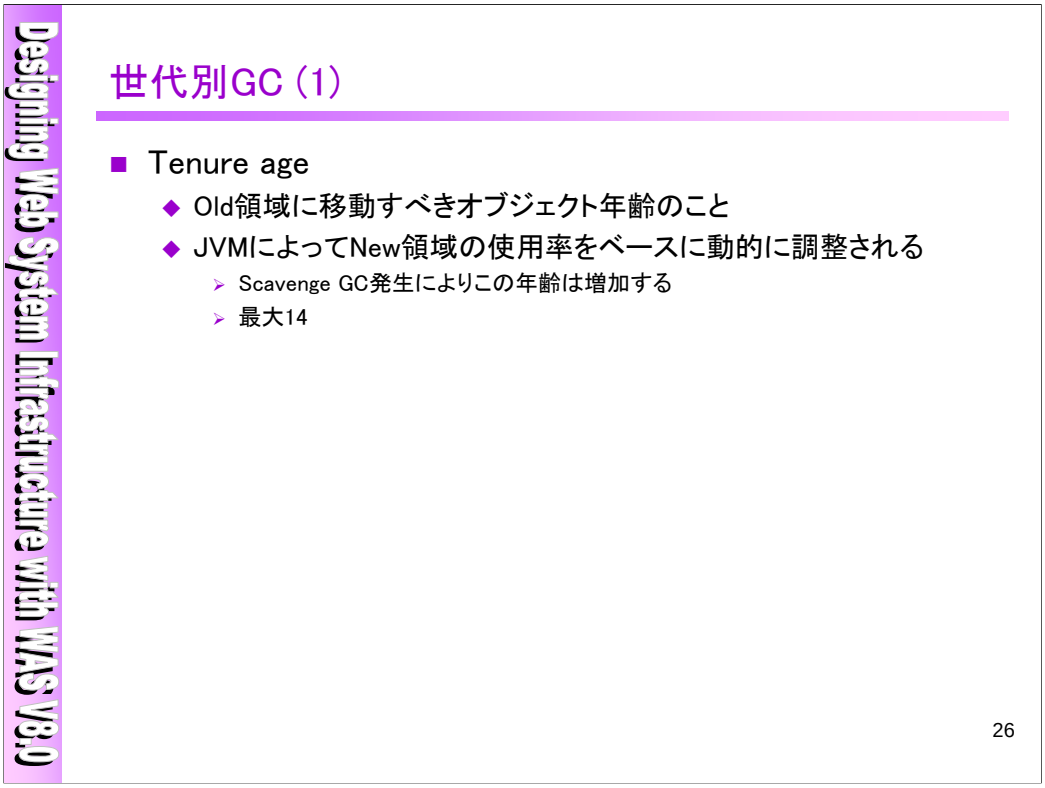
そこで，ヒープの割り当て領域を行える領域をある特定の領域（New領域）に絞り込むことでGCの発生頻度を相対的に上げつつ，その中である回数のGCの中を生き延びたオブジェクトはその後も長く必要となる可能性が高いと判断して，あまりGCをかけない領域（Old領域）に移動させるというのが世代別の方針です。

ただし，Old領域に移動させたオブジェクトがすべてWASの生存期間中に必要なオブジェクトとなる保障はないため，Old領域が一杯になったところでOld領域の中の使用されていないオブジェクトの廃棄がかかり，新たなOld領域のオブジェクト候補のためのスペースが確保されます。

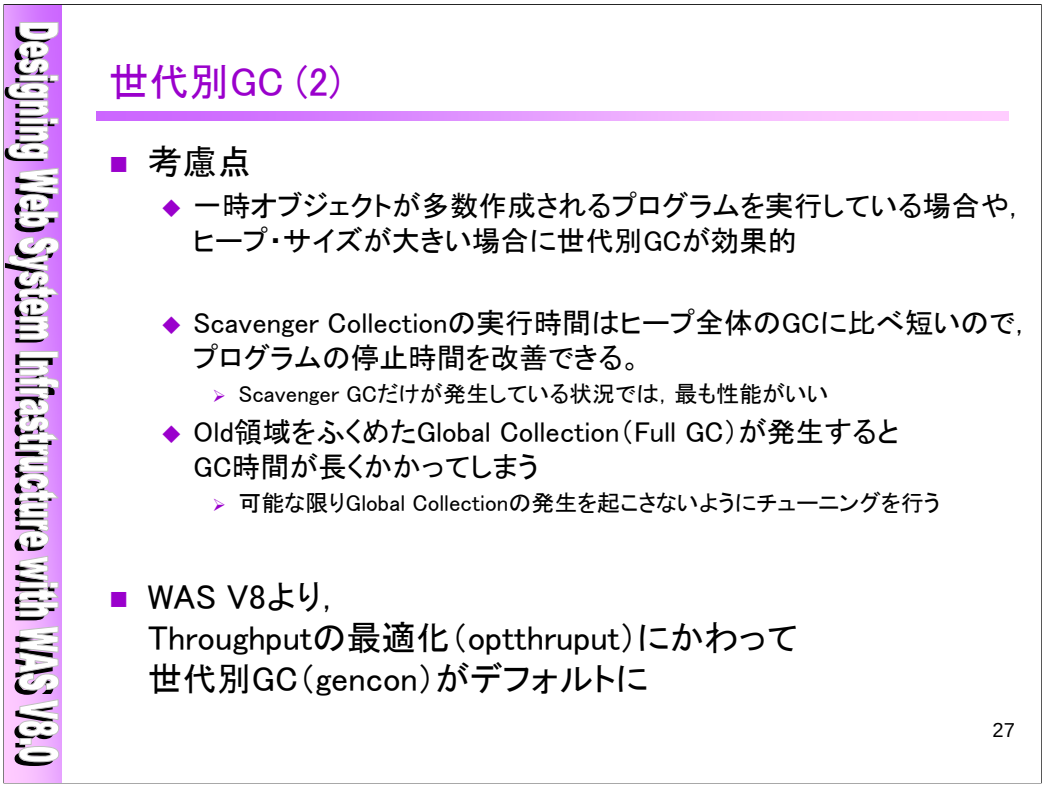


New領域は二つの領域に分かれ，一方をAllocate領域，もう一方をSurvivor領域と呼びます。生成されるオブジェクトはまずAllocate領域に格納されます。Allocate領域がいっぱいになると，ヒープ内の「生きた」オブジェクトのトレースが行われます。その際に，「生きた」オブジェクトがあるとそのオブジェクトをSurvivor領域にコピーをしていきます。これによって，Allocate領域の全体のチェックが終わると，「生きた」オブジェクトはすべてSurvivor領域にコピーされており，Survivor領域を使ってこれ以降のリクエスト処理を行えるようになっています。そこで，Survivor領域が新たにAllocate領域として使用され，元Allocate領域だったものはSurvivor領域となって次のガーベッジ・コレクションで再び役割が交代するまで放置されます。このNew領域を使用したガーベッジ・コレクションをScavenge Collectionと呼んでいます。Scavenger Collectionが行われている間は，Mark & Sweepの時と同じようにすべてのアプリケーション・スレッドは停止します。

ある設定された閾値を超えるGCをNew領域で生き延びたオブジェクトはSurvivor領域ではなくTenured領域にコピーされます。ある程度JVMが稼動し続けていると大抵はTenured領域もオブジェクトでいっぱいになります。その中には閾値を超えたGCを生き延びたものの，すでにガーベッジ（到達不能）となってしまっているオブジェクトも存在します。そこでTenured領域に対してもGCが実施されます。Tenured領域のGCは「Pause Timeの最適化」ポリシーと同じ方式がとられています。これによりTenured領域のGCにかかる時間を短縮します。GCポリシー名に「Concurrent」とついているのはこのTenured領域に採用されたGCの性質を表しているためです。



オブジェクトの「年齢」はどれだけの回数のGCを生き延びたかで判断されます。Scavenger Collectionが実施されるたびに，JVMは生き残ったオブジェクトにひとつずつ「年齢」を加算して管理を継続していきます。Tenured領域へは閾値の回数だけ生き延びたオブジェクトのみが移動することができますが，この閾値は最大でも14です(つまり，タイムアウトをある程度長く設定したHttpSessionオブジェクトやデータ・ソースのConnectionオブジェクトなどは最終的にTenured領域に格納される可能性が高くなります)。この閾値はNew領域内にとどまっているオブジェクトの割合によってJVMが調整を行っていて，生き延びているオブジェクトの占有率が10%以下の場合閾値は10に，その後占有率の上昇とともに閾値も上がっていき，占有率が30%を超えると最大値に到達します。



世代別GCは，リクエスト処理毎に使い捨てられる一時オブジェクトが多い場合や，ヒープのサイズが大きく全体のGCに時間がかかる場合に，パフォーマンスを向上する効果があります。

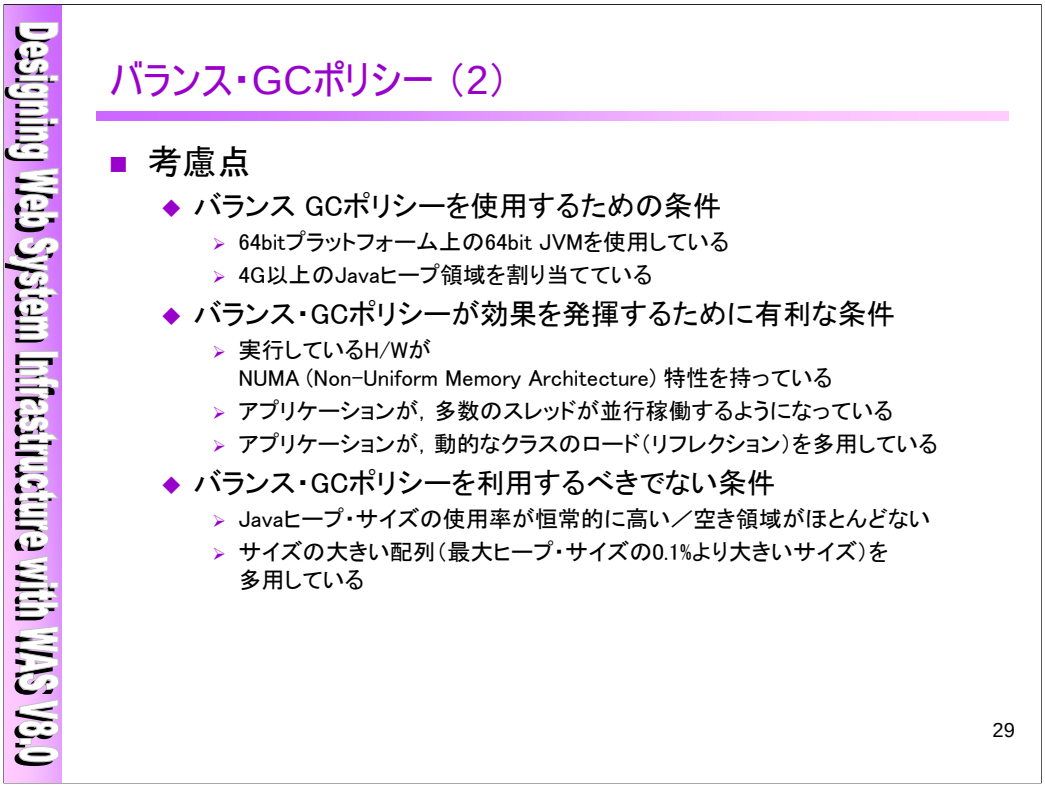
New領域（Nursery領域）を対象としたScavenger Collectionの実行時間は短く，このタイプのGCだけが起こっている間は，他のGCポリシーと比較して最も性能がよくなります。ただし，世代別GCの弱点として，Old領域（Tenured領域）を含めたGlobal Collection（Full GC）が発生してしまうと，停止時間が極めて大きくなってしまう点が上げられます。そのため，世代別GCを使用する場合には，可能な限りGlobal Collectionを起こさないようにチューニングを行う必要があります。

昨今は，WASのアプリケーション・サーバーに割り当てられるJavaヒープのサイズが大きくなる傾向があり，また一時オブジェクトを多用するフレームワークが増加したこともあり，世代別GCがデフォルトのGCポリシーになりました。

WAS V8.0のIBM JVMから，バランス・GCポリシーが新たに追加されました。

このGCポリシーでは，小さなサイズ（最大Javaヒープ・サイズで指定されたサイズの1024分の1の値を，2の階乗に切り捨てたサイズ）の領域に分割し，独立して管理することによってGCの割合を低減させます。各スレッドは，個別に空の領域を割り当てられ，新規に作成されたオブジェクトにAllocateします。領域がいっぱいになると，その領域内でCopyingによるGCを実行します。必要に応じて複数領域や全体のGCをおこないます。

この方式は，割り当てたJavaヒープ・サイズが極めて大きく（4G以上），New領域に限定したGCであっても停止時間が無視できなくなってきた場合に使用を検討します。H/Wの特性やアプリケーションの条件によっては，GCの時間を短縮することができる場合があります。

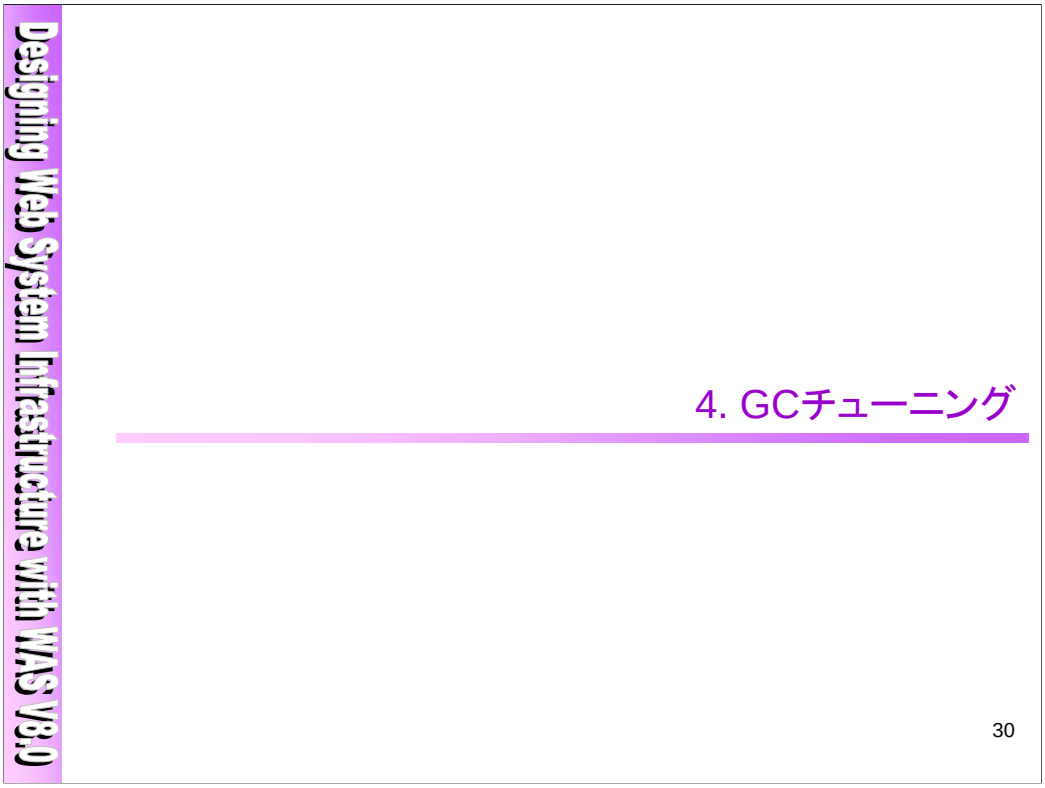


バランスGCポリシーを使用するにあたって必須の条件が二つあります。一つは64bit JVMを使用していること，4G以上のJavaヒープ領域を割り当てていることです。32bit版のJVMや，4G未満のJavaヒープ領域しか割り当てていない場合には，バランスGCポリシーは使用できません。

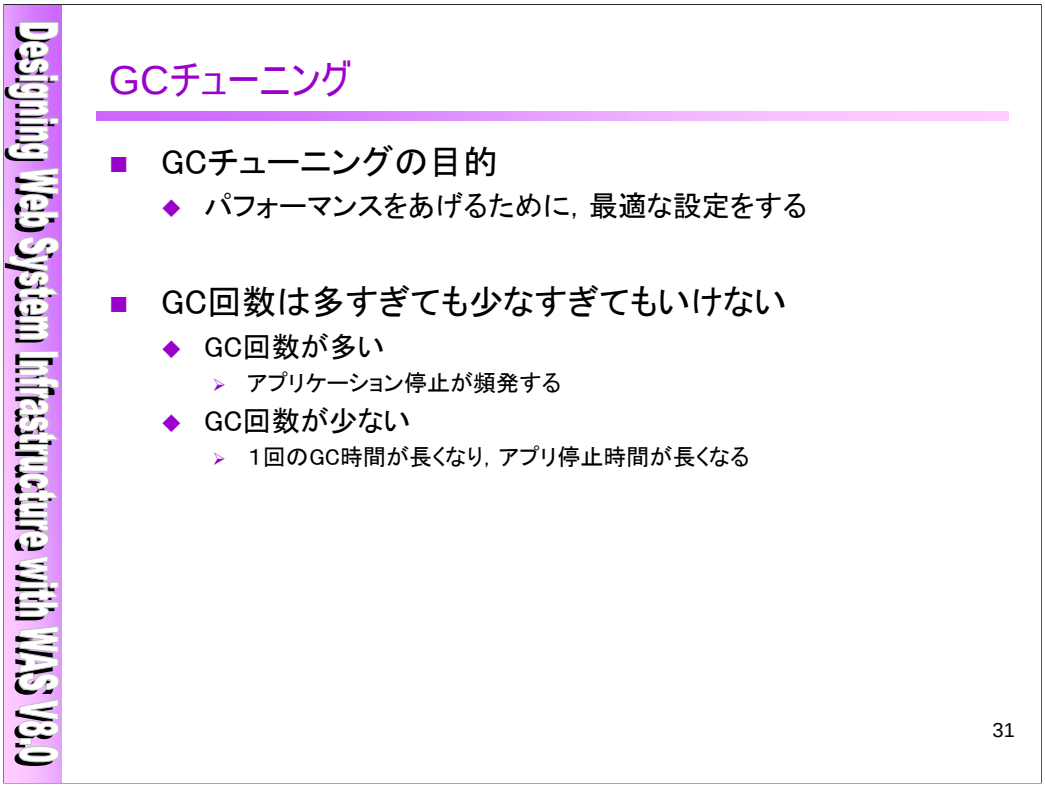
バランスGCポリシーは，H/WがNUMA特性を持っており，多数のCPUが並行して稼動している環境で特に効果が大きくなります。また，アプリケーションも多くのスレッドを使用している場合に有利となります。また，アプリケーションがリフレクションを多用している場合にも効果が期待できます。リフレクションを多用すると，メソッドやフィールドにアクセスするためのアクセッサクラスが一時オブジェクトとして多数作成されます。通常の世代別GCでは，Scavenger Collectionではクラス・オブジェクトがGCの対象とならないため，これらのクラスオブジェクトが蓄積してFull GCをひきおこすことがあります。バランスGCポリシーでは，クラスオブジェクトも通常のGCの対象となります。

一方，バランスGCポリシーを使用すべきでない状況もいくつかあります。このGCポリシーでは，空き領域が十分にないと効率が悪くなるため，Javaヒープの使用率が恒常的に高い場合にはパフォーマンスが悪化します。また，一つの領域に入りきらない大きなサイズのオブジェクトがある場合にも効率が悪くなります。

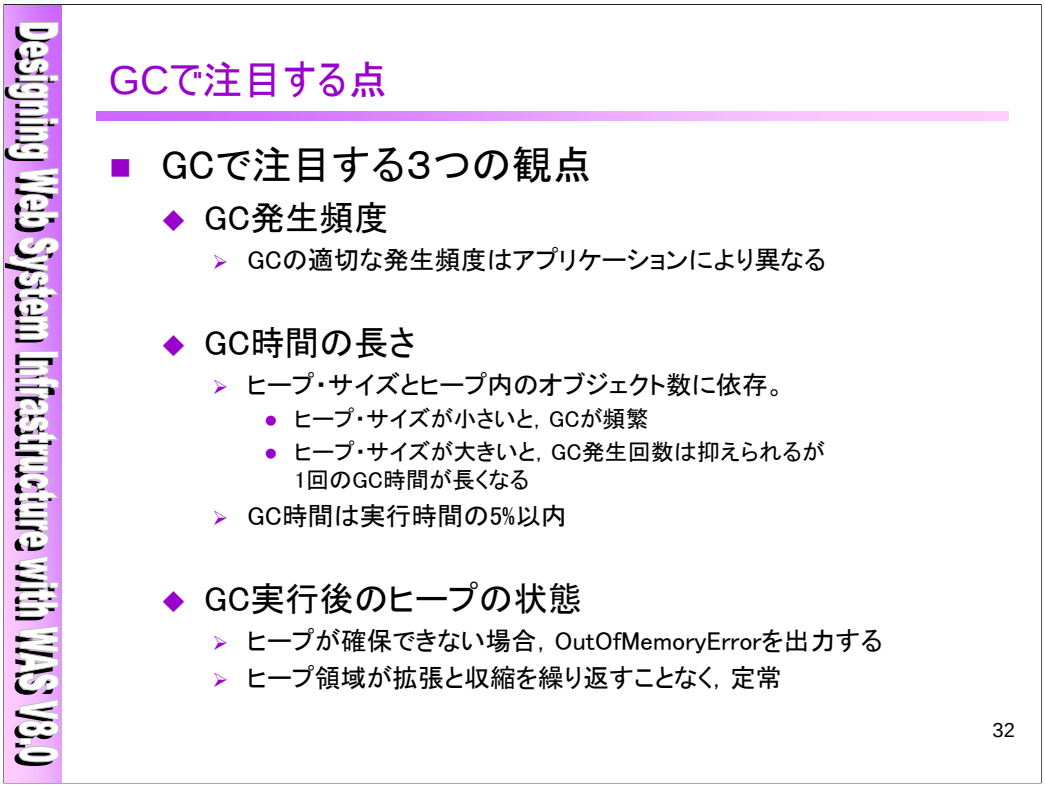
このように使いどころが難しいGC方式ではあります。ですが，8Gや12Gなどの巨大ヒープを割り当てたアプリケーション・サーバーなどの事例も出てきており，長大なGC時間を何とか短くするというチューニングが実施される機会も増えています。そのような場合には，このGCポリシーの使用も検討して下さい。



ここでは，GCおよびJavaヒープのチューニングについて説明します。



GCは，Javaにおけるメモリー関連のパフォーマンス・ボトルネックの大きな要因といえます。そのため，GCが多すぎるとアプリケーションの停止が頻発し，GCが少なすぎると1回のGC時間が長くなります。



GCで注目する観点は三つあります。1つめがGCの発生頻度，これはGC前後のタイム スタンプを比較することで確認することができます。

2つめがGC時間の長さです。このとき，ヒープ・サイズは小さすぎても大きすぎても最適なパフォーマンスは得られません。GC時間は実行時間の5%以内になるようチューニングします。GC時間が5%以上で頻発している状態の場合は，Javaヒープ・サイズを増やす必要があります。

・計算方法

GC実施時間÷最後のオブジェクト割り当て（AF）以降の時間×100 ＝GC時間

最後にGC後のメモリー領域に着目します。ヒープの状態によっては，ヒープの拡張，縮小が発生しますが，これらの処理の間，アプリケーションは停止してしまいますので，できるだけ拡張，縮小はおさえたほうがよいです。また，拡張しても十分なヒープが得られない場合はOutOfMemoryErrorが発生してしまうので，注意する必要があります。

Designing Web System Infrastructure with WAS V8.0

冗長ガーベッジ・コレクション

- GCの詳細をログにレポートする機能
  - メモリー関連のトラブル発生時や，ヒープのチューニングに必須の資料
- 全てのアプリケーション・サーバー管理プロセスで有効にしておくことを推奨
  - 管理コンソールでサーバーの「プロセス定義」→「Java仮想マシン」で「冗長ガーベッジ・コレクション」にチェックを入れる
- 出力は，プロセスログのSTDERRファイル（デフォルトではnative_stderr.log）に出力
  - 汎用JVM引数の「-Xverbosegclog:<file_name>」で別ファイルに出力させることが可能
  - 「-Xverbosegclog:<file_name>,X,Y」とファイル名に続けて数字を指定する事でログ・ローテートの指定が可能。Y回のGCの実行ごとにファイルをわけ，X世代のファイルを残すようになる
  - 独立したファイルに出力することで，解析ツールなどを使用した分析を安全・確実に行うことができる

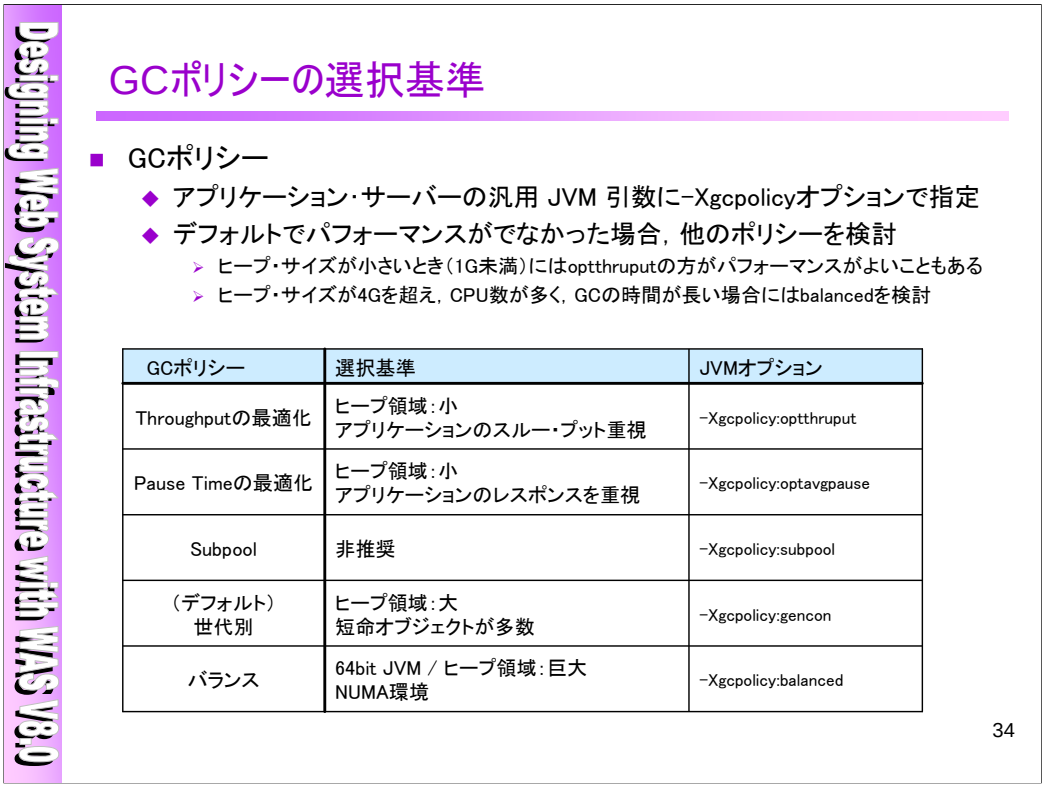
33

GCのチューニングをするにあたって必須の設定が冗長ガーベッジ・コレクションの有効化です。

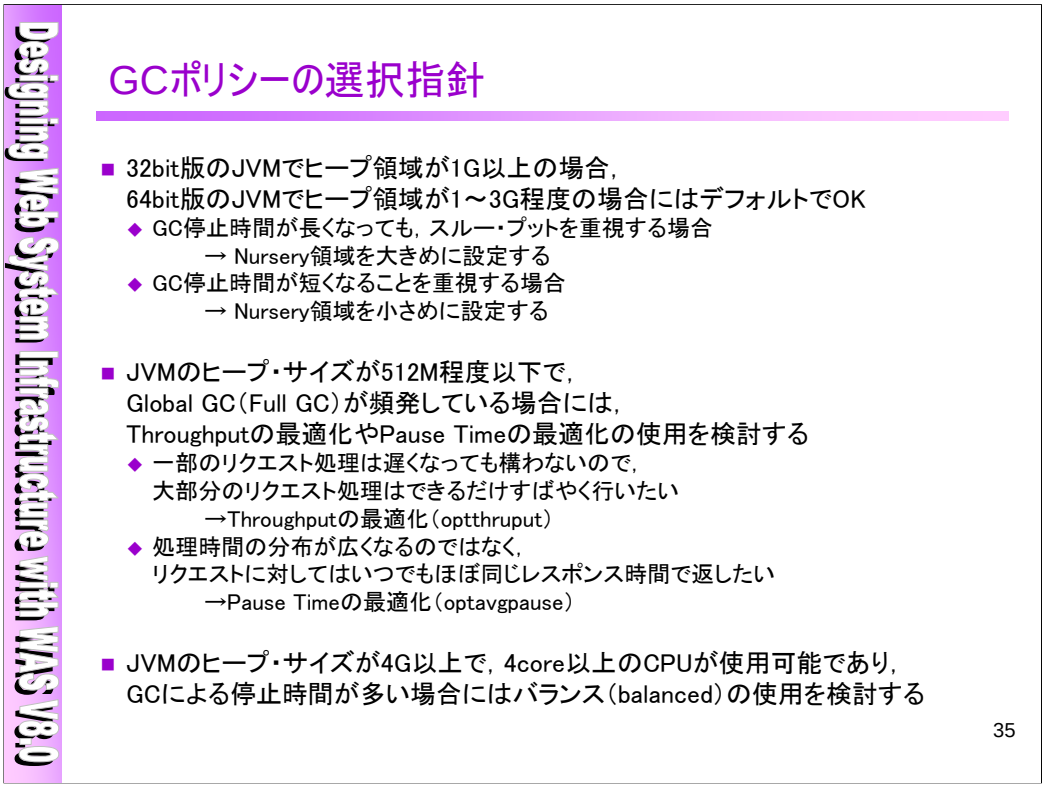
IBM JVMでは，-verbose:gcを指定したときに，他社のJVMにくらべて詳細なレポートが出力されます。WAS V8.0で使用されているJVMでは，GCにかかった時間，トリガーとなったAllocation，GC前後のメモリーの詳細な使用量などの情報を，XML形式で出力します。

ヒープのチューニングやOutOfMemoryErrorの問題解析など，Javaヒープに関わるあらゆる解析で必須となる情報ですので，かならず有効にしておきます。管理コンソールから「冗長ガーベッジ・コレクション」にチェックを入れることで有効にできます。アプリケーション・サーバーのJVMだけでなく，NodeAgentやDeploymentManagerなどの管理プロセスについても，同様に有効にしておきましょう。

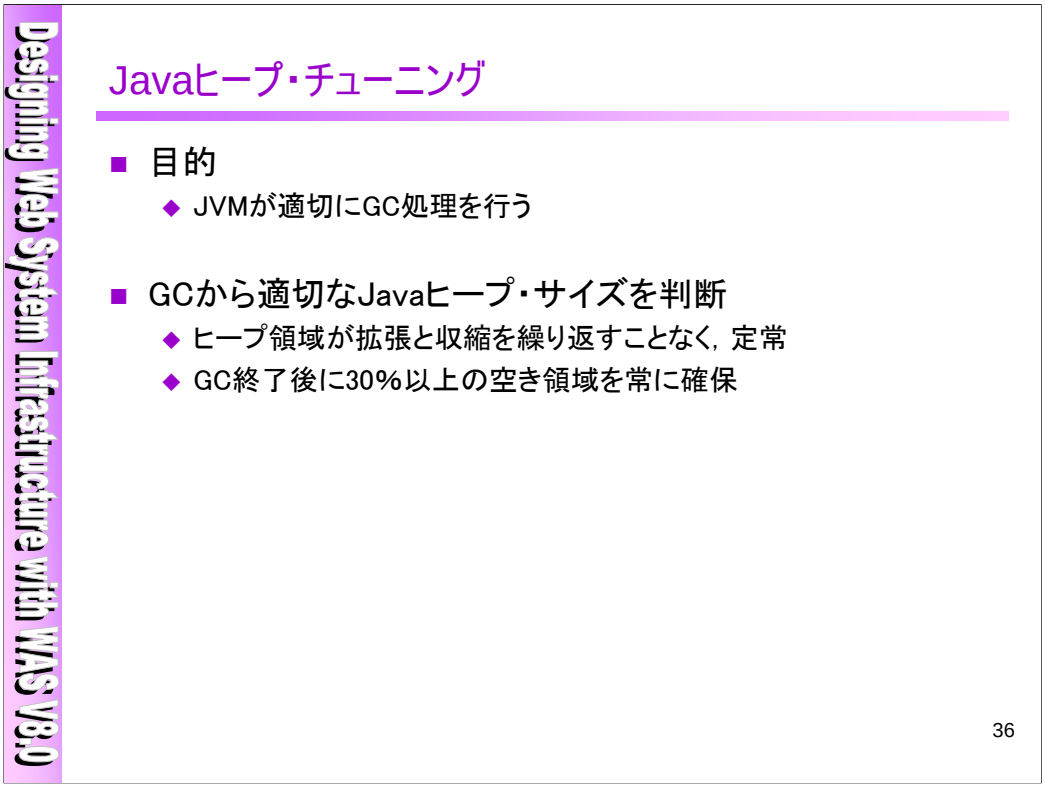
レポートはJVMプロセスの標準エラー出力に印字されますが，ここには冗長ガーベッジ・コレクション以外の情報も出力されることがありますので，解析を容易にするために別ファイル分けることも有効です。汎用JVM引数を指定する事で，指定したファイルに出力したり，またローテートログの構成を行うことができます。



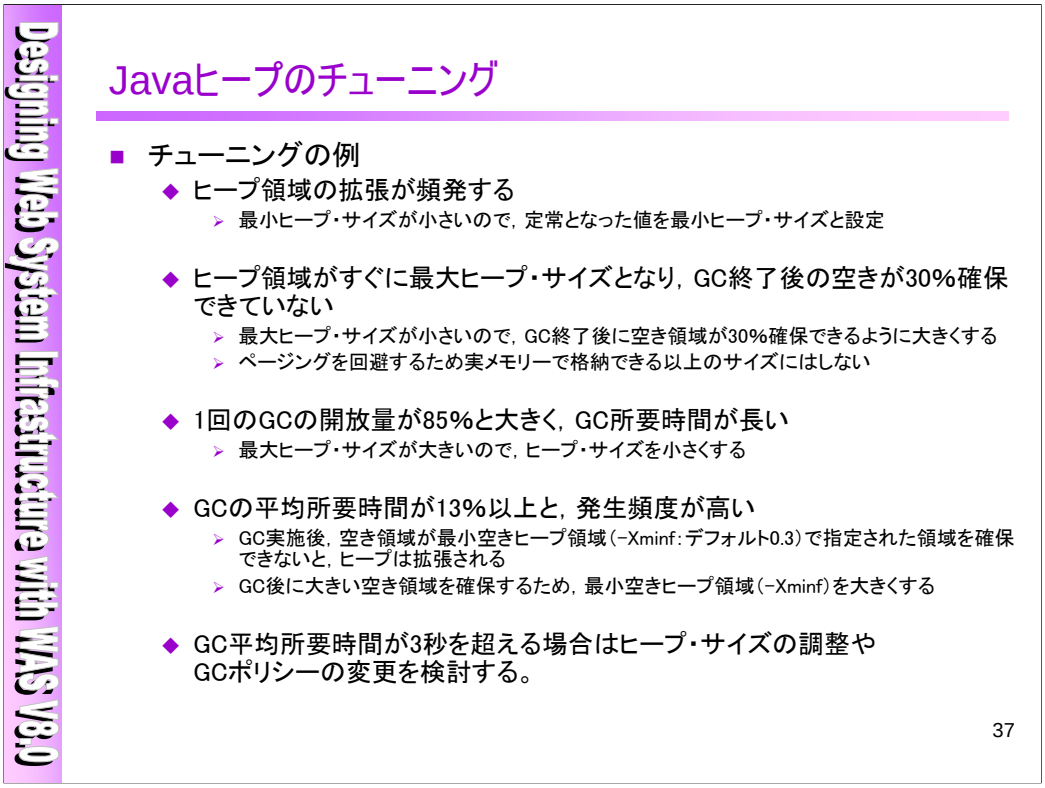
これまでの説明事項をひとつの表にまとめてみました。それぞれのGCポリシーを使用したい場合にJVMへ渡す引数も一緒に乗せてあります。WAS V8.0で使用されるIBM JREでのデフォルトは世代別GCです。これらの指定は，WASのJava仮想マシンに対する設定（汎用JVM引数）で行います。



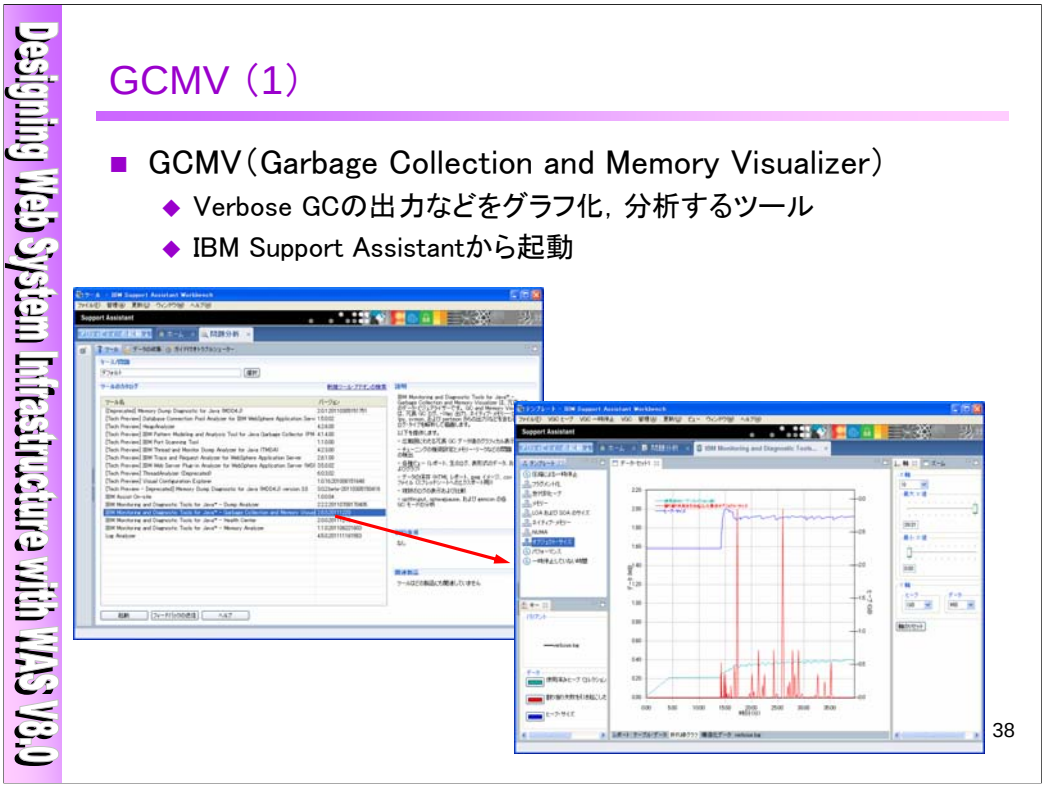
最適なGCポリシーはアプリケーションのデザイン，システム構成，および利用状況によって異なります。大抵のケースではデフォルトのポリシーで十分要件が満たされる場合が多いといわれています。もしデフォルトポリシーでうまくいかなかった場合，このチャートをチューニングで試すポリシーの検討材料にしてみてください。



Javaヒープのチューニングは GCの状態を確認しながら決定します。本来，ヒープは拡張および縮小を繰り返すことなく一定の状態であることが望ましく，ヒープ占有率は70%前後が好ましいです。そのような状態にあるときにGC発生頻度やGC時間が最も安定します。ヒープ状況をモニターするには冗長ガーベッジコレクションを使用します。

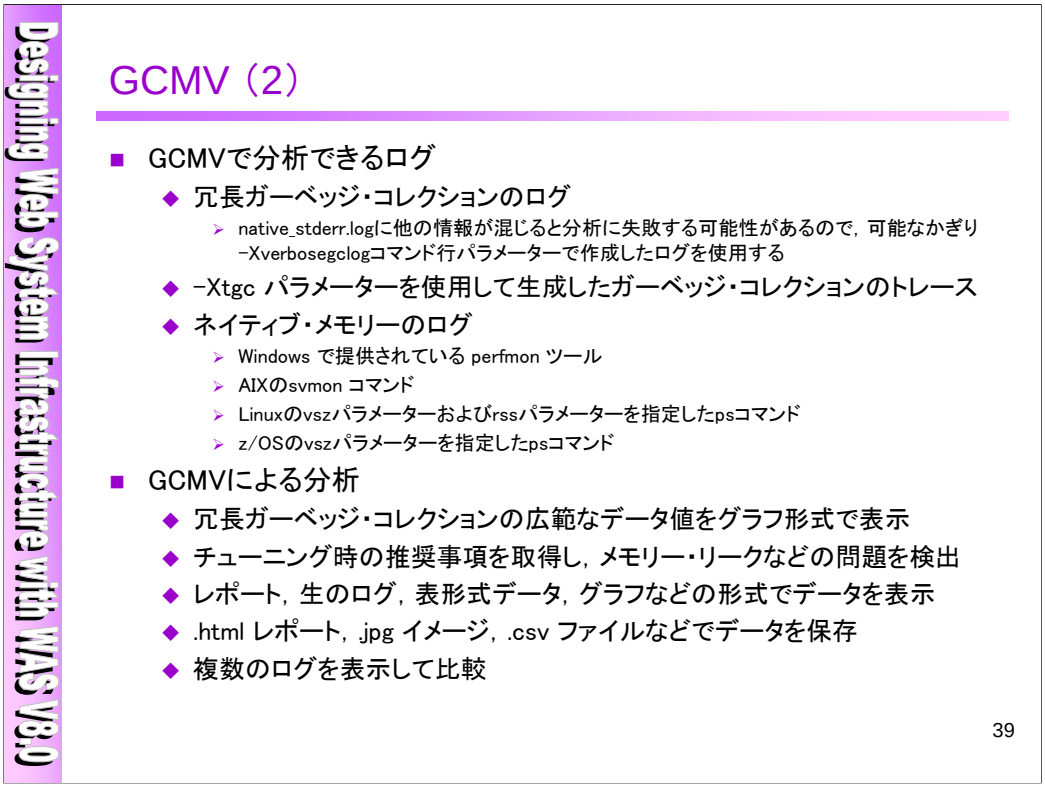


ヒープ・サイズの設定を行うときの一般的な設定指針を参考として記載します。システムのサービスインを迎える前までに，この指針に沿って決定した値を出発点としてチューニングを行い，個々のシステムに適切な値を決定してください。また，GCの平均所要時間は，GCの平均所要時間％＝(GC平均所要時間)/(GCの平均所要時間＋GC呼び出し平均間隔)×100，で見積もることが可能です。

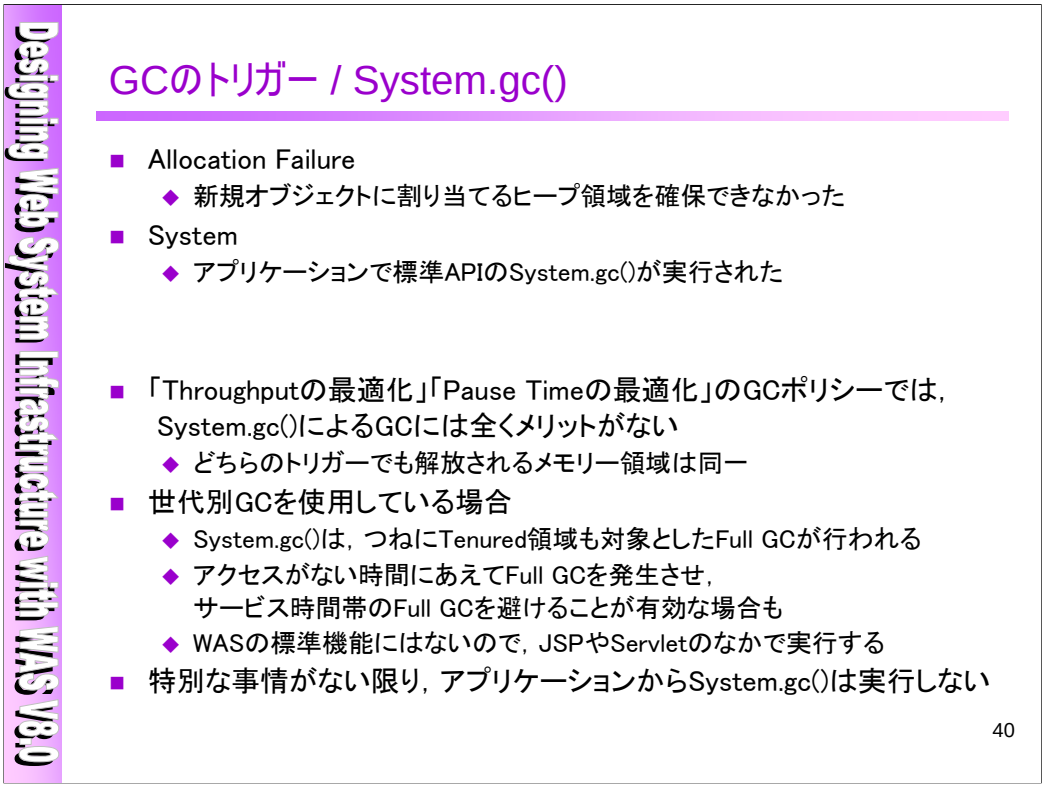


記録された冗長ガーベッジコレクションは，テキスト形式（XML形式）ですので直接読むことができます。ただ，全体的な傾向をつかむためにはデータをグラフ化して視覚的に分析することが有効です。IBMからは，冗長ガーベッジコレクションの出力をグラフで分析するツールとして，GCMV（Garbage Collection and Memory Visualizer）が提供されています。

GCVMは，無償でダウンロードできるツールで，ISA（IBM Support Assistant）から導入・起動ができます。



GCVMでは，冗長ガーベッジ・コレクションに含まれる様々な情報をグラフ化することができるほか，GCトレースで取得された情報，OSで取得されたネイティブ・メモリーの情報もグラフ化することができます。また，結果を自動的に分析し，問題点をレポートする機能も持っています。画像ファイルを出力してレポートのデータとして使用したり，また他のツールで解析を行うためにCVS形式でデータを保存することもできます。

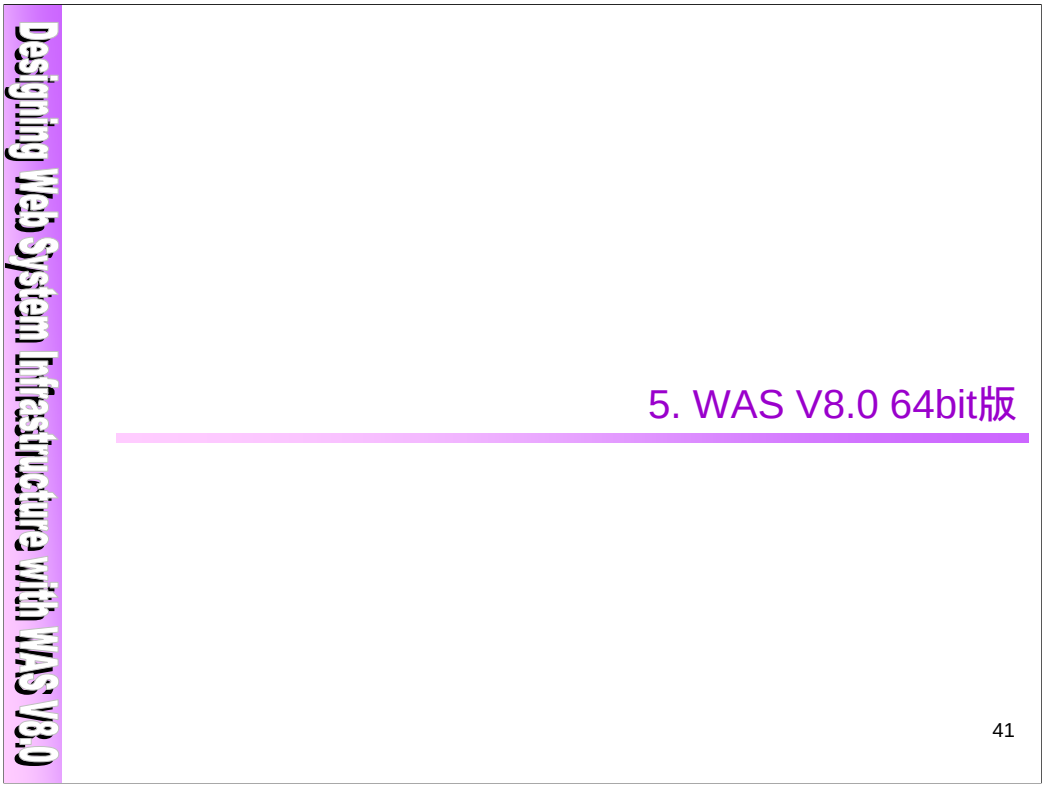


GCが開始されるトリガーには二つあります。一つは新規に作成されるオブジェクトにAllocateする(割り当てる)ヒープ領域が確保できなかったときで，これはAllocation Failure(AF)とよばれます。もう一つのトリガーが，Javaのアプリケーションから System.gc()を実行することによるGC開始の指示です。

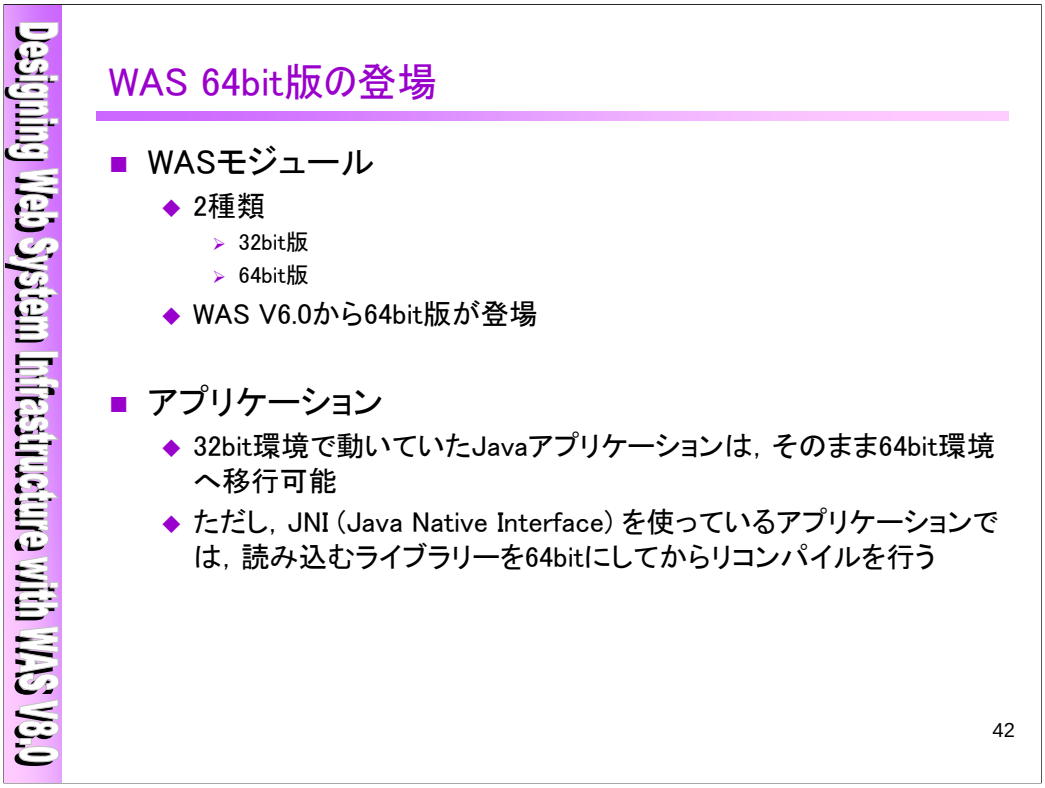
一般にGCのチューニングは，GCの発生回数を可能な限り減らすことを目的に実施されますので，わざわざGCをおこすSystem.gc()の実行にはメリットがありません。アプリケーション中での System.gc()実行は避けるべきです。

ですが，限定的な状況でSystem.gc()の実行が有効な場合があります。それは，世代別GCを利用している状況で，Full GCが数日に一回発生しているというような状況です。業務サービスを提供している時間帯にFull GCが発生すると，その間のレスポンスが低下します。これを避けるために業務時間外にあえてFull GCを発生させ，業務時間中のFull GC発生を抑止することが有効な場合があります。

このような特別な事情がない限りは，アプリケーション中からSystem.gc()は実行しないようにして下さい。



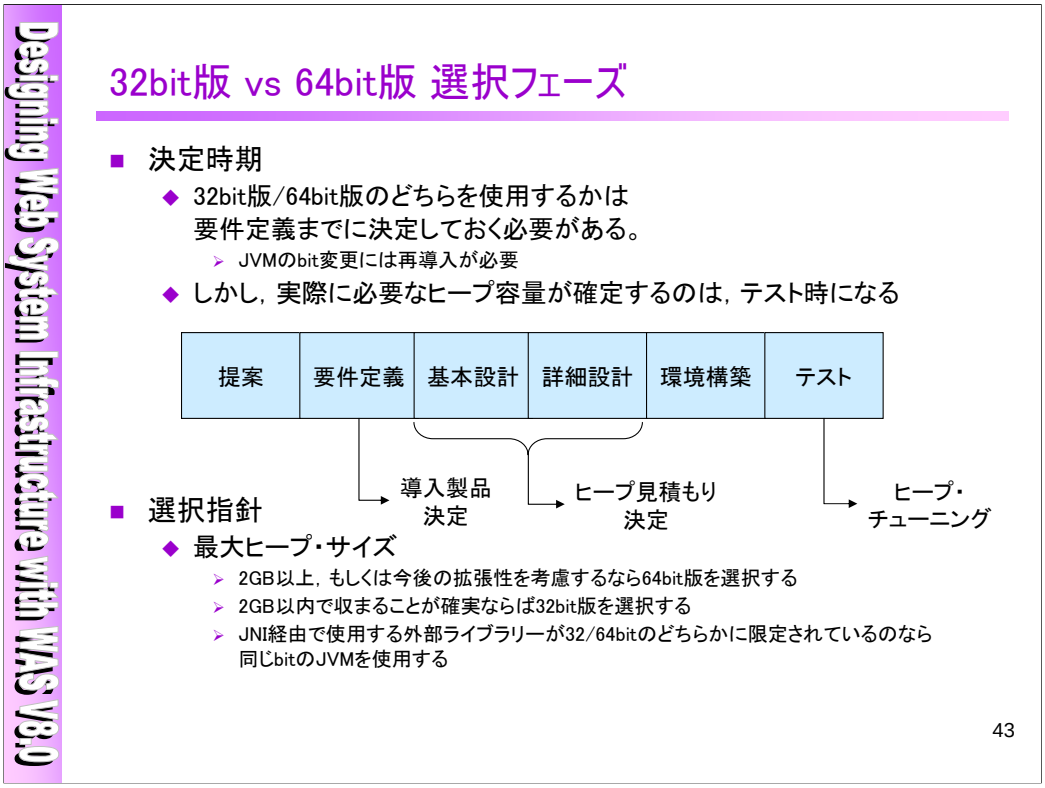
この章では，WASの64bit版と32bit版について説明します。



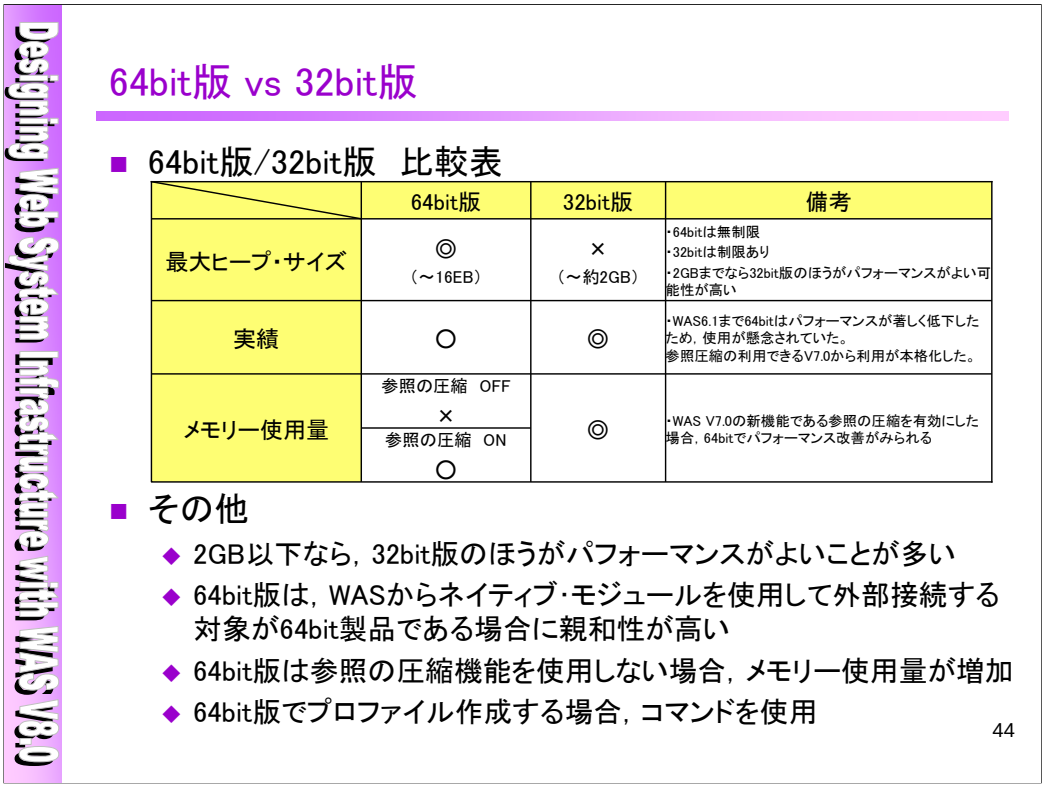
WASモジュールには，32bit版と64bit版の2種類が存在します。

2012年2月時点で最新のWASバージョンは8.0であり，64bit版モジュールはその3つ前のメジャーバージョンである6.0から提供されています。64bit版モジュールを導入するためには64bit OS環境が必要となります。また，32bit版，64bit版モジュールは別モジュールになるため，32bit版から64bit版に，もしくは64bit版から32bit版に変更する場合は，再インストールをおこなう必要があります。

WASの上で動くJava EEアプリケーションについては，WAS環境の違いにかかわらず，同じものを利用することができます。



実際のヒープ・サイズがわかるのはテストフェーズになってからですが，導入製品の決定はもっと早いフェーズで実施されます。64bit版モジュールを選択する際の判断基準として，最大ヒープ・サイズが挙げられます。32bit版モジュールの場合，最大ヒープ・サイズを2GB程度(この値はOSによって若干異なる)までしか指定することができません。そのため，お客様要件より1つのJVMでヒープ・サイズを2GB以上必要とすることが確定している場合は，64bit版モジュールを選択してください。今後の拡張によりヒープ・サイズが2GB以上になる可能性がある，または今後の技術動向や拡張性を考慮するのであれば64bit版をお勧めしております。ただし，ヒープ・サイズはGCの観点から小さい程パフォーマンスは良いので，必要以上にヒープを大きくすることは推奨しません。



下記32bit版，64bit版のメリット・デメリットを参考に使用するモジュールを選択ください。

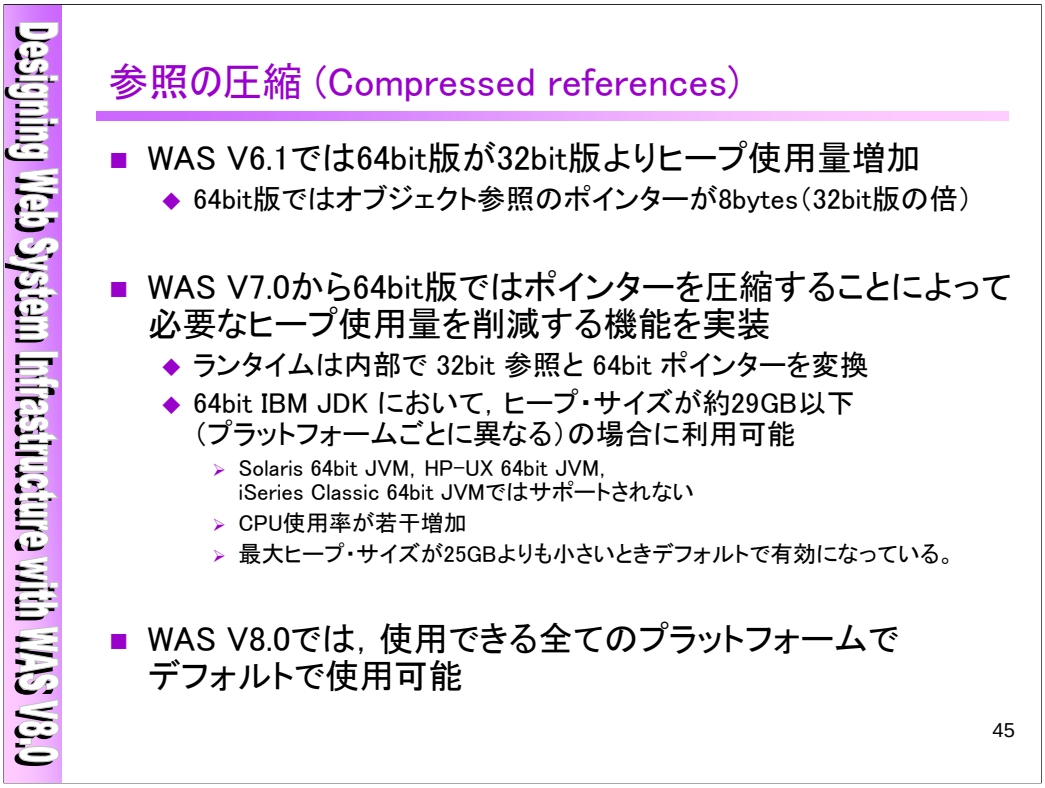
■32bit版
–メリット
実績が多い。
必要なHeapサイズが2GB以下である場合，64bit版と比較してパフォーマンスがよい可能性が高い。

・デメリット
最大ヒープ・サイズを2GB程度までしか指定できない。

■64bit版
・メリット
最大ヒープ・サイズを2GB以上(理論上16EBまで)指定できる。
WASからネイティブ・モジュールを使用して外部接続する対象が64bit製品である場合，親和性が高い(後述の[その他考慮点]を参照)

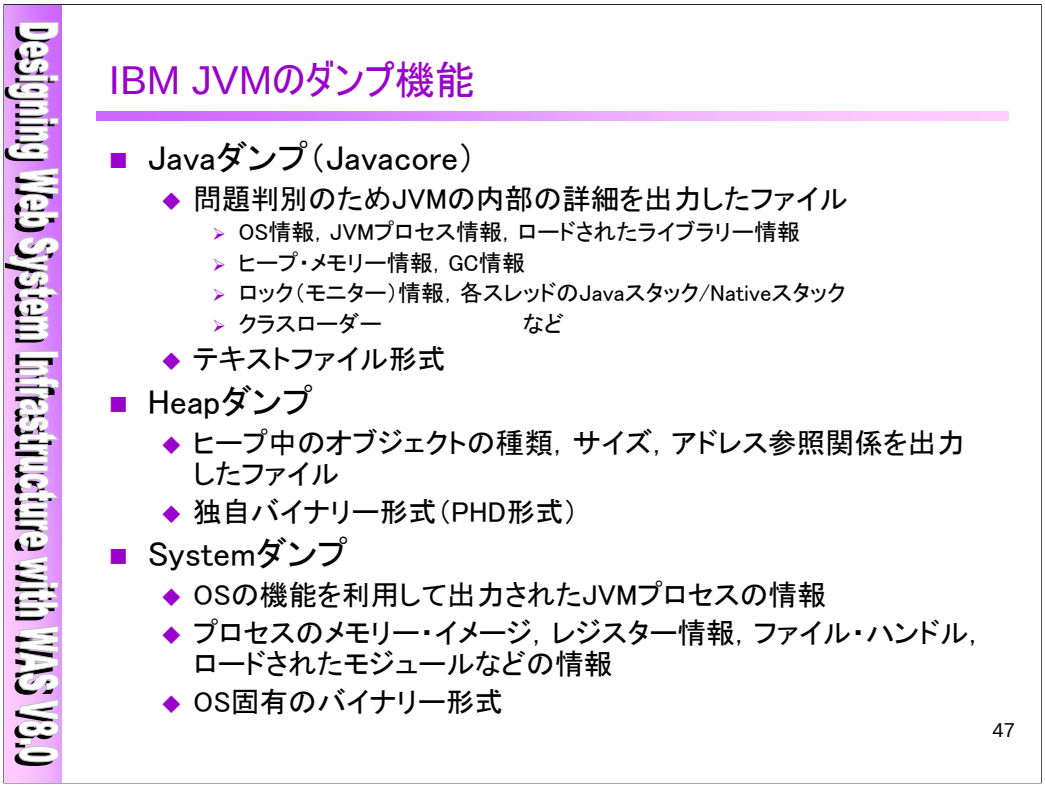
・デメリット
実績は32bit版と比較すると少ない。(ただし，64bit版の使用実績は着実に増えている。)
メモリー・アドレス参照に32bit版は4byte使用するのに対し，64bit版は8byte必要となるため，64bit版の方が最大で60%程度メモリー使用量が増加する可能性がある。また，この理由からプロセッサーキャッシュを有効利用できず，32bit版と比較してパフォーマンスがダウンする可能性もある。(ただし，メモリー使用量の増加については，WAS V7.0の新機能である参照の圧縮を使用することにより改善可能。)
プロファイル作成時にプロファイル管理ツール(PMT)を使用することができないため，manageprofilesコマンドでプロファイルを作成する必要がある。

[その他考慮点]
DB2のType2ドライバーを使用する場合や，MQのBinding接続をおこなう際，ネイティブ・モジュールが必要になる場合があります。このような場合は，WASの環境に合わせたネイティブ・モジュールを使用するように構成してください。(例: WAS 32bit版であれば，接続するDB2インスタンスが64bitであってもDB2 CLI 32bit版を指定する。)



WAS V6.1の64bit版は32bit版と比較してヒープを多く使用する傾向にありました。これはオブジェクト参照のポインター・サイズが32bit版が4bytesであるのに対して64bit版では8bytesであるのが原因でした。WAS V7.0から64bit版のメモリー使用量を抑える機能として，64bit版の8bytesの参照を4bytesに圧縮する参照の圧縮(Compressed references)機能が導入されています。本機能は64bit版の機能となります。参照の圧縮はメモリー使用量の削減に効果があり，2GB以下のヒープにおけるパフォーマンスも32bit版に近い値を実現することができます。WAS V8.0の64bit版モジュールでは，この機能がデフォルトで有効になっています。

この章では，問題判別の資料収集に役立つDumpエージェントの構成についてご説明します。



IBM JVMでは，問題判別の役にたつダンプ機能が各種実装されています。これらは，取得された時点のJava実行環境の各種情報を含んでいます。

Javaダンプは，JVMの異常終了やハング，スローダウンなどの解析に利用されます。Javaアプリケーションレベルの詳細な情報が記録されています。

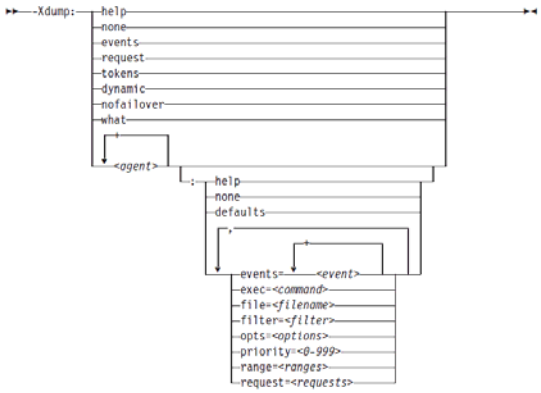
Heapダンプは，ヒープ上にAllocateされているオブジェクトの詳細な情報を記録しています。OutOfMemoryErrorやメモリー・リークなどの問題を解析する際の有用な資料となります。

Systemダンプは，OSの機能を使用してJVMプロセスの全ての情報をファイルに出力したものです。Javaよりも下層のレベル，JITの問題やOSの機能に関わる問題を調査する際に使用されます。

Designing Web System Infrastructure with WAS V8.0

Dumpエージェント

- IBM JVMで使用できるダンプを統合的に扱うエージェント
  - 出力トリガー，出力ダンプ，出力先を構成
- 汎用JVM引数 -Xdumpで構成
  - 他のコマンド行オプション・環境変数による指定は，互換性のために残っているが，可能な限り-Xdumpを使用する



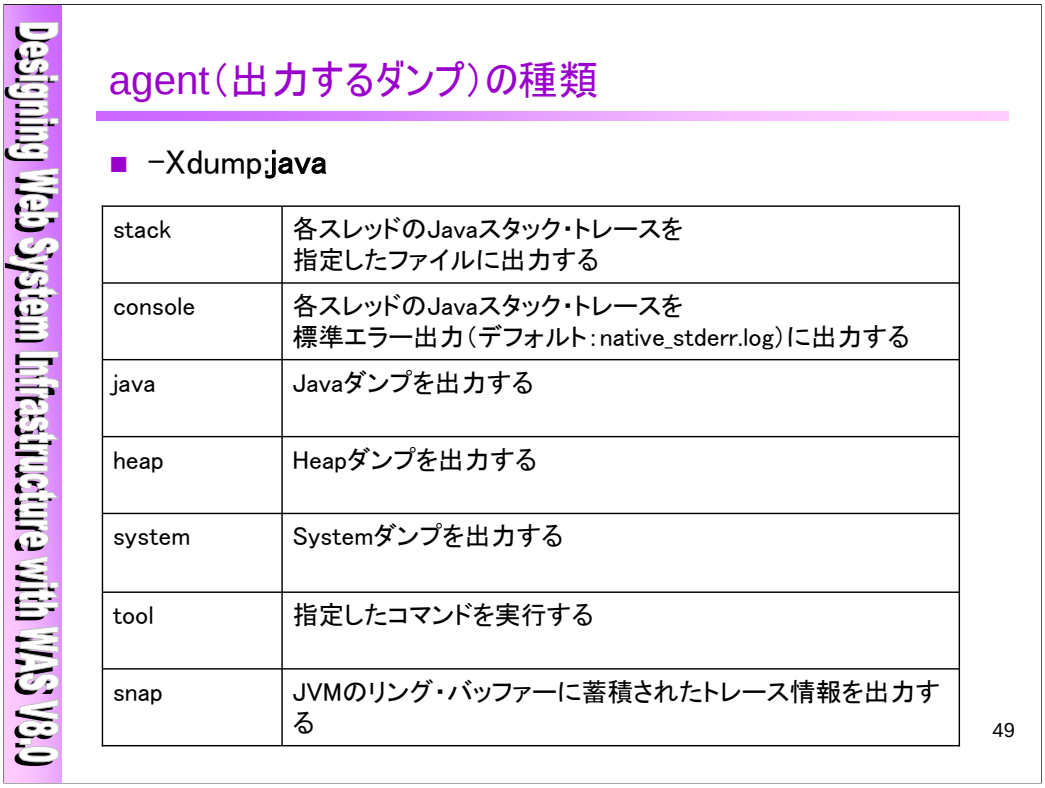
48

これらのダンプは，デフォルトで必要に応じて適切なものが自動的に出力されるようになっています。

管理者がその出力タイミングや出力するダンプの種類，出力先などを変更したい場合には，Dumpエージェントを構成します。Dumpエージェントは，ダンプ出力やトレース機能の出力などを統合的に管理するエージェントです。汎用JVM引数の-Xdumpを使用して構成します。

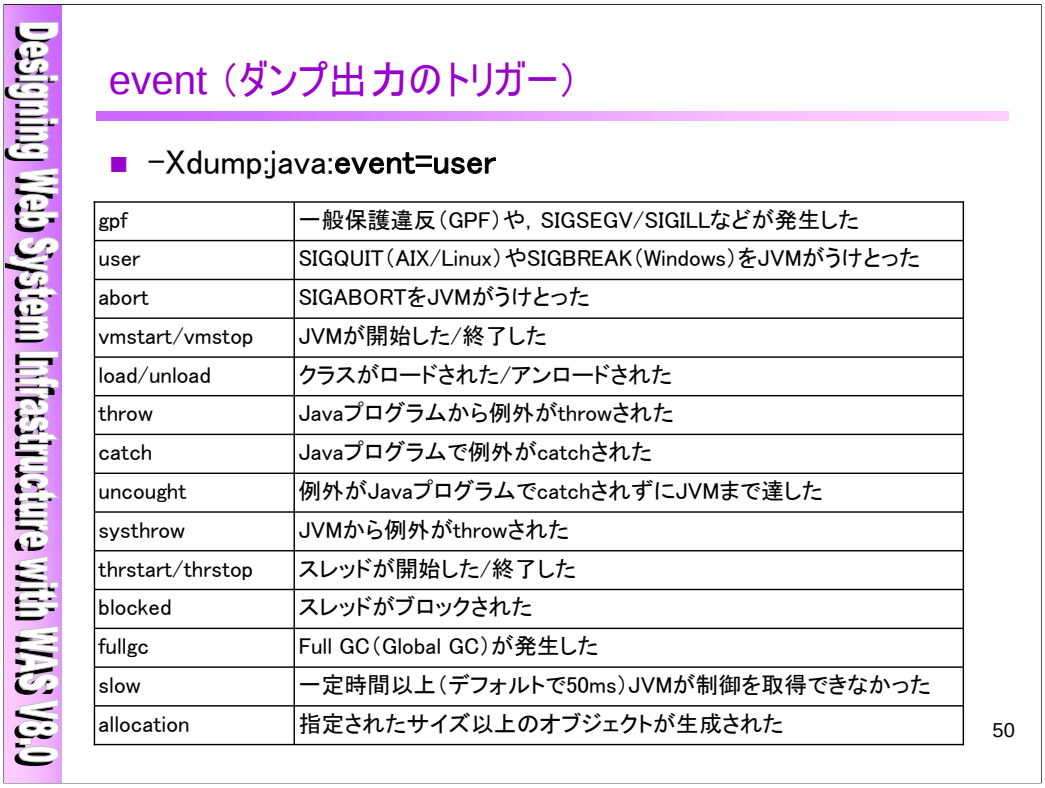
ダンプ出力の制御や出力先の指定などは，以前は個別の引数や環境変数で行っていました。現在のJVMでも，それらの設定は互換性のため有効になってはいますが，今後は可能な限り-Xdump引数を使用して制御するようにして下さい。

-Xdumpは，出力するダンプの種類，出力するトリガー，各種条件を順に指定します。-Xdumpは複数指定することもできます。複数の-Xdumpのトリガーに合致した場合，priorityの順に全てが実行されます。

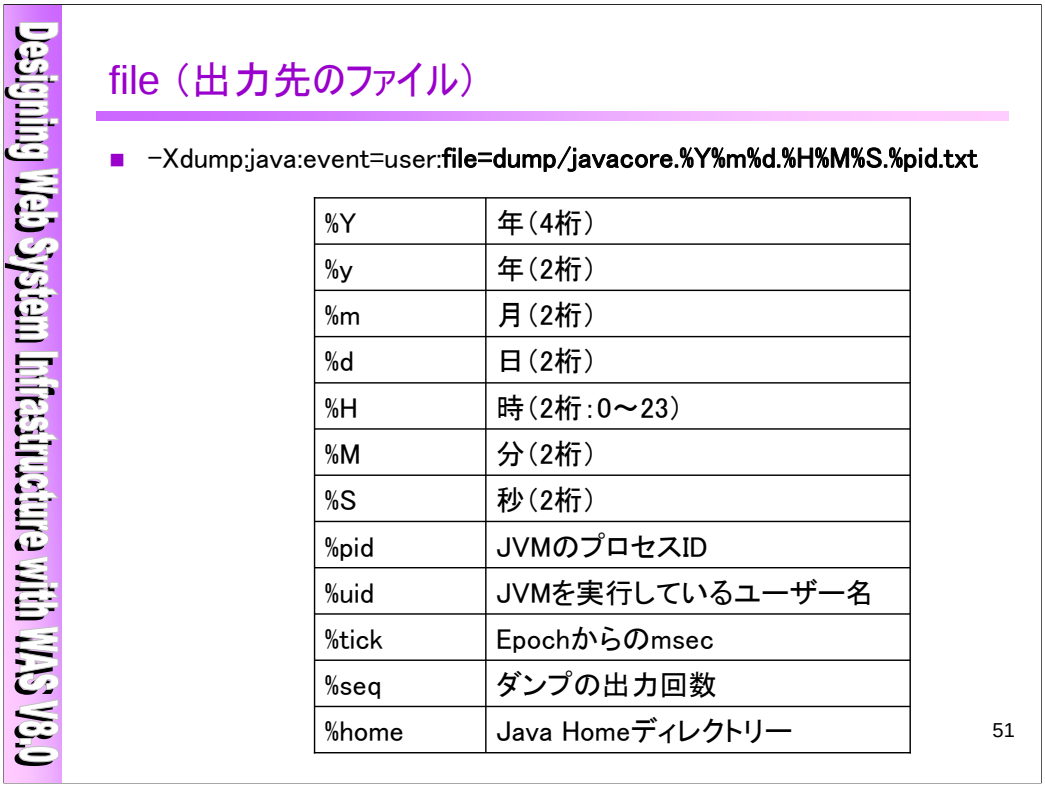


-Xdumpには，コロンで区切って出力するダンプの種類を指定します。使用できるagentとしては，前述の三つのダンプの他，スタック・トレースの出力やローカルのコマンドの実行なども指定する事ができます。

コマンドの実行を指定する場合は，実行にかかる時間に注意して下さい。コマンドの実行している間はJVMの動作は停止します。コマンドの実行が完了するまでJVMの動作は再開しませんので，あまり長い時間がかかるコマンドは指定しないで下さい。



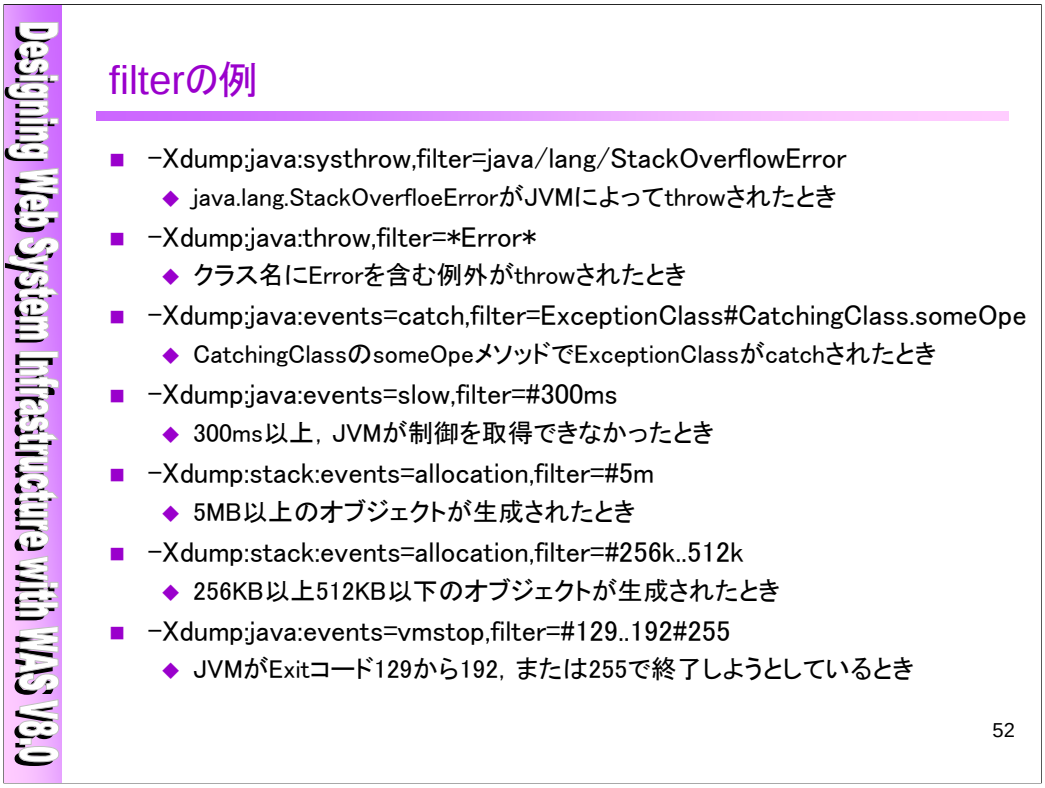
続いてダンプを出力するトリガーを指定します。外部やOSからのシグナルの他，例外の処理，ヒープのアロケートやスローダウンなどをトリガーとすることもできます。



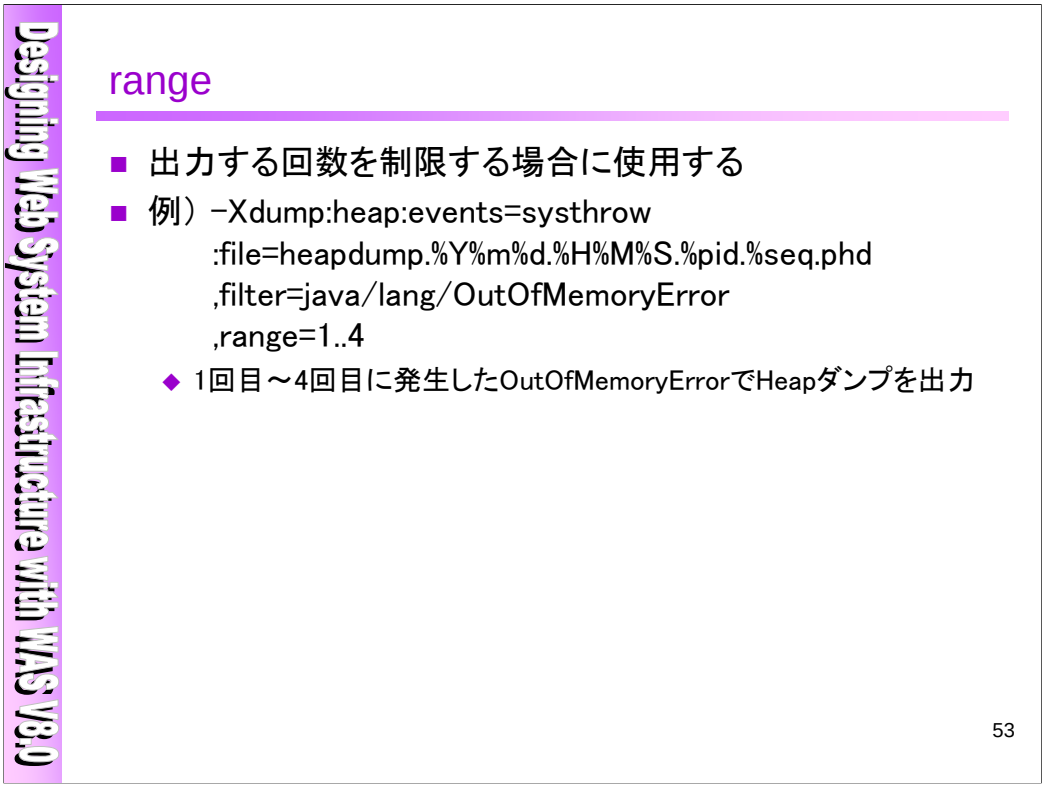
出力するファイル名には，各種の置き換えトークンが使用できます。

ファイル名を相対パスで記述した場合は，JVMのカレントディレクトリー（WASの管理コンソールで指定します。デフォルトはプロファイルのディレクトリー）からの相対パスとして扱われます。絶対パスを記述することもできます。

ダンプのうち，HeapダンプおよびSystemダンプは出力サイズが巨大になりがちです。これらのダンプの出力先を，十分な空き容量を持ったディスク領域に指定しておくことは，設計フェーズで検討するべき項目の一つとなります。

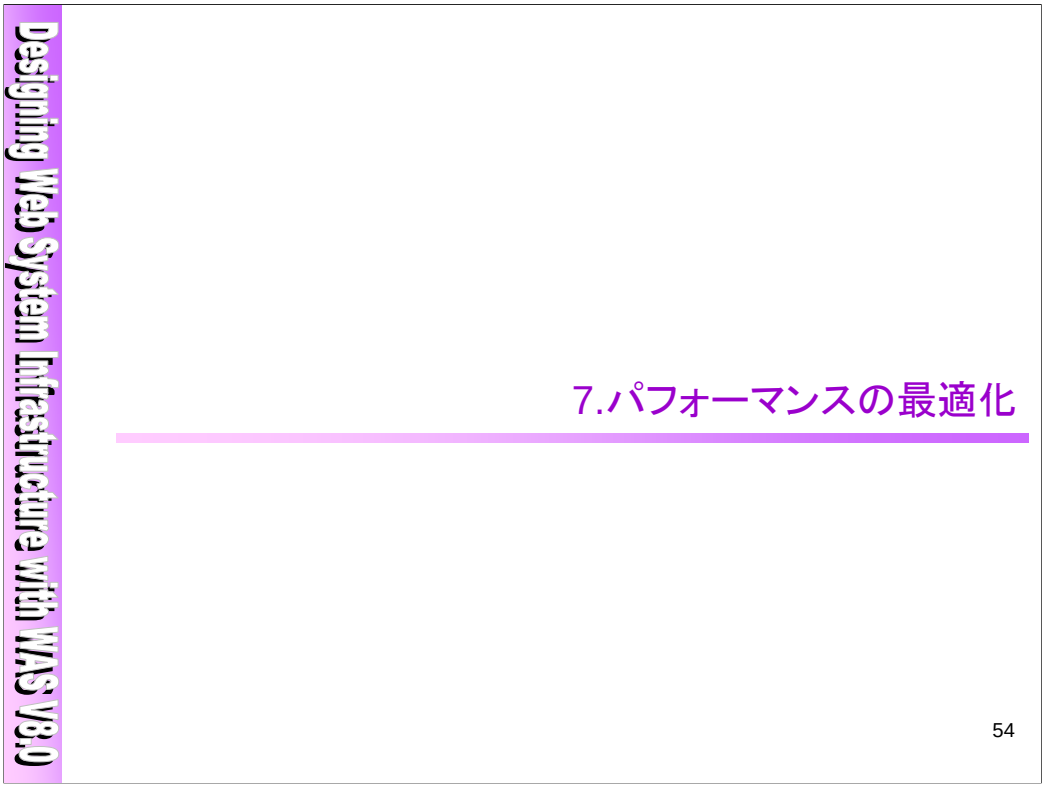


Dumpエージェントの指定には，filterによって各種の条件をつけることができます。特定の問題についての解析を行う場合には，それに応じた処理を記述します。

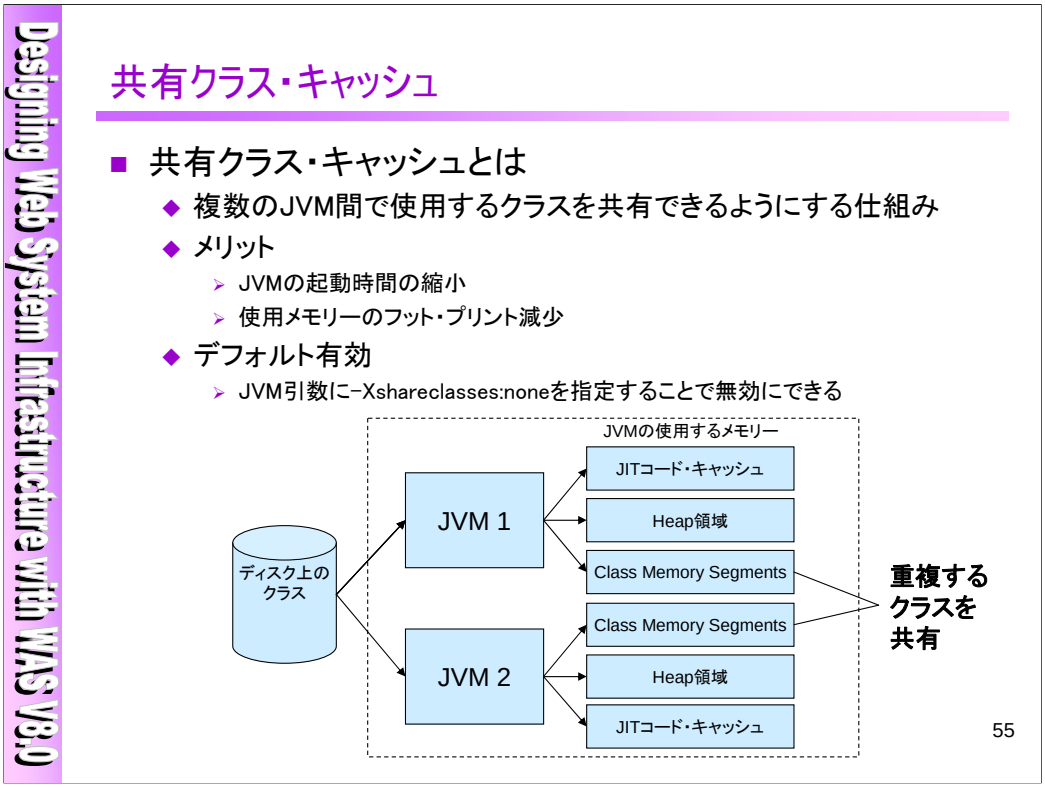


また，出力条件として使用できる項目としてrange指定があります。これはダンプの出力回数（JVMが起動してからの回数）を一定回数に制限する機能です。OutOfMemoryErrorで出力されるHeapダンプは出力が大きいため，無制限に出力しているとディスク領域があふれる可能性があります。このようなダンプについて回数を制限しておくことは有用です。

ただ，回数を制限した場合は，実際にダンプが出力されたときに，すみやかに解析のためのダンプを退避してJVMを再起動することが必要です。そうしないと，続けて問題が発生した場合に，必要な情報が取れないことになってしまいます。



この章では，パフォーマンス向上が期待できるWAS V8.0の機能を紹介します。



JVMのパフォーマンス向上の仕組みのひとつとして共有クラス・キャッシュがあります。これはIBM JDK V5で導入されたもので，複数のJVM間で使用するクラスを共有する仕組みです。この機能を用いることによりJVMの起動時間の短縮と使用メモリーのフット・プリントを減少させることができます。

デフォルトで有効になっており，無効化するにはJVM汎用引数に-Xshareclasses:noneを指定します。

WAS V8 InfoCenter – Java virtual machine settings

http://publib.boulder.ibm.com/infocenter/wasinfo/v8r0/topic/com.ibm.websphere.nd.doc/info/ae/ae/urun_rconfproc_jvm.html

Designing Web System Infrastructure with WAS V8.0

共有クラス・キャッシュ – JDK6での改良点

- JDK6ではOS再起動をまたいで共有クラス・キャッシュの情報を保持することが可能
  - JDK5ではOS再起動の際にキャッシュ情報は消去されていた
  - JDK6ではキャッシュ情報をファイルに書き出すことによりOS再起動をまたいで情報を保持することを可能にしている
  - JVM汎用引数で設定
    - 有効　-Xshareclasses:persistent
    - 無効　-Xshareclasses:nonpersistent

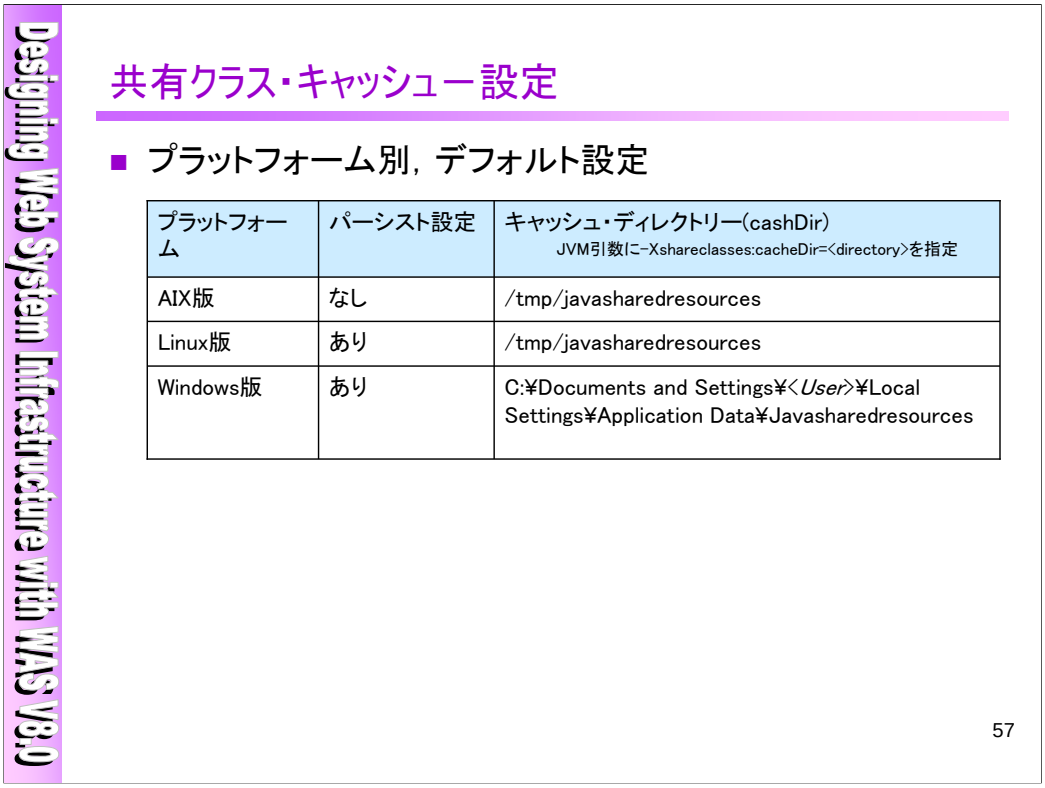
56

共有クラス・キャッシュはIBM JDK 5で導入されましたが，JDK6でいくつかの改良がされています。JDK5ではOSを再起動させるとキャッシュ情報はクリアされてしまいましたが，JDK6ではファイルに書き出すことによってOS再起動をまたいでキャッシュ情報を保持させることが可能になっています。これによりOS再起動後の最初のJVM起動時間の短縮がはかれます。

またJDK6ではキャッシュできるデータタイプも増えています。

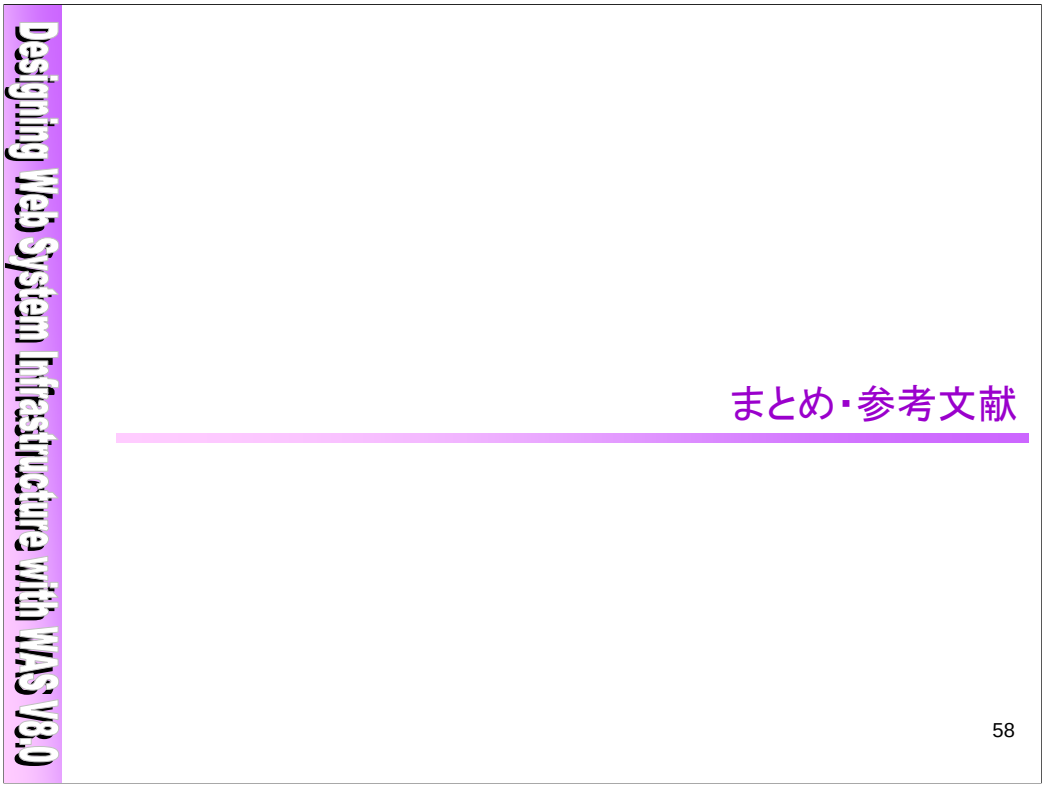
Java Diagnostics Guide 6 – Class data sharing command-line options

http://publib.boulder.ibm.com/infocenter/javasdk/v6r0/topic/com.ibm.java.doc.user.aix32.60/user/sharedclassesxoptions.html



オプションにcacheDir=<directory> を設定することで，JVM キャッシュ・ファイルの場所を設定変更することができます。

Windows版のキャッシュ・ディレクトリーで，Windowサービスによる起動では，<User>はデフォルトでSYSTEM ユーザーとなります。


Designing Web System Infrastructure with WAS V8.0
まとめ・参考文献
58

Designing Web System Infrastructure with WAS V8.0

まとめ

- GCポリシーを選択し，
  チューニングすることが可能
- 参照の圧縮により，
  WAS64bit版もスルー・プットが改善されている
- ダンプの出力を設定しておく
- その他，パフォーマンス向上のための機能として
  共有クラス・キャッシュとランタイムが提供されている

59

Designing Web System Infrastructure with WAS V8.0

参考文献

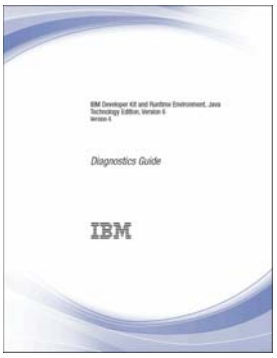
- Information Center
  - WebSphere Application Server V8.0
    - http://publib.boulder.ibm.com/infocenter/wasinfo/v8r0/index.jsp
    - http://publib.boulder.ibm.com/infocenter/wasinfo/v8r0/topic/com.ibm.java.doc.60_26/homepage/plugin-homepage-java626.html
  - IBM SDK and Runtime Environment Java Version 6
    - http://publib.boulder.ibm.com/infocenter/javasdk/v6r0/index.jsp
- ワークショップ資料
  - WebSphere Application Server V8 アナウンスメント・ワークショップ
    - http://www.ibm.com/developerworks/jp/websphere/library/was/was8_ws/
  - V7.0によるWebシステム基盤設計ワークショップ資料
    - http://www.ibm.com/developerworks/jp/websphere/library/was/was7_guide/

60

Designing Web System Infrastructure with WAS V8.0

IBM SKD for Java Diagnosis documentation

- IBM JVMの内部動作，設定，問題判別手法について詳細に解説したドキュメント
  - メモリー管理，クラスロード，クラス共有，JIT/AOTコンパイラー，RMI/ORB，JNI
  - プラットフォームごとの問題判別手法（AIX，Linux，Windows，z/OS）
  - Garbage Collectionの詳細と問題判別手法
  - JVMのトレース，ダンプの構成方法と記録される情報
  - JVMで使用可能なコマンド・ライン・オプション



- WebからPDF形式でダウンロード可能
  - http://www.ibm.com/developerworks/java/jdk/diagnosis/
- 年に数回更新されているので，常に最新版を手元においておきましょう

61

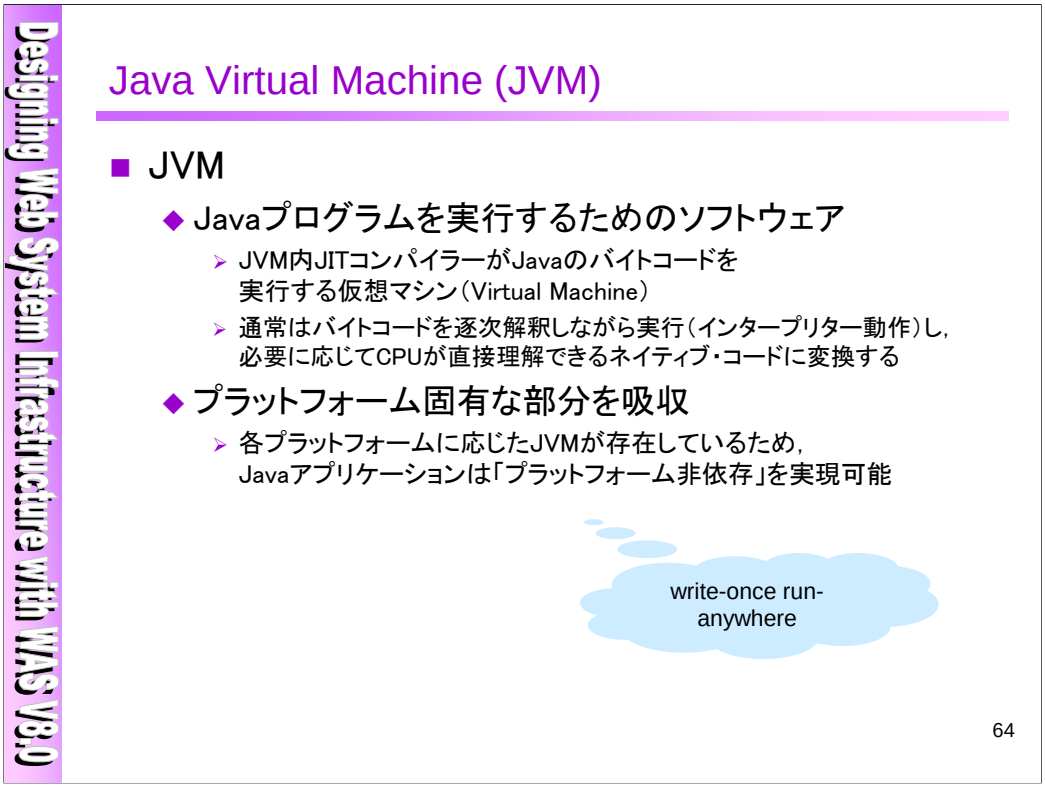

Designing Web System Infrastructure with WAS V8.0
WASV8.0によるWebシステム基盤設計Workshop
JVM設計
ー 参考 ー

Designing Web System Infrastructure with WAS V8.0

2. GCの基本

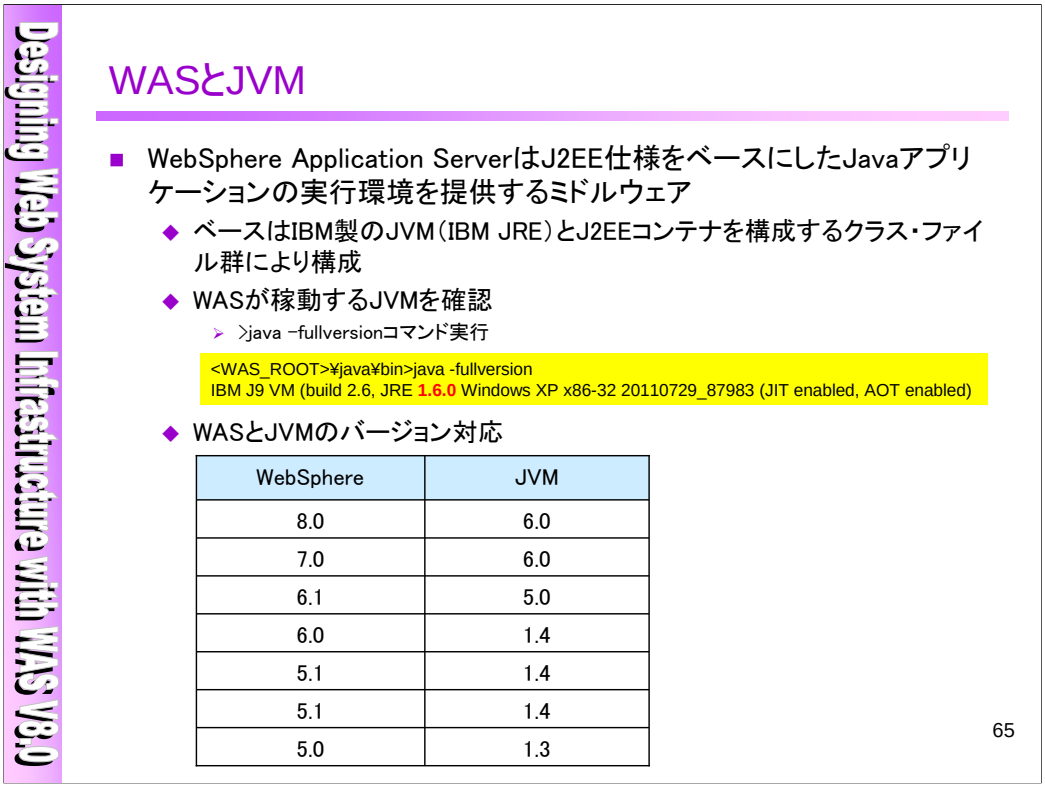
・GCの基本
・GCポリシー
・GCチューニング
・WAS8.0 64bit
・パフォーマンスの最適化

63

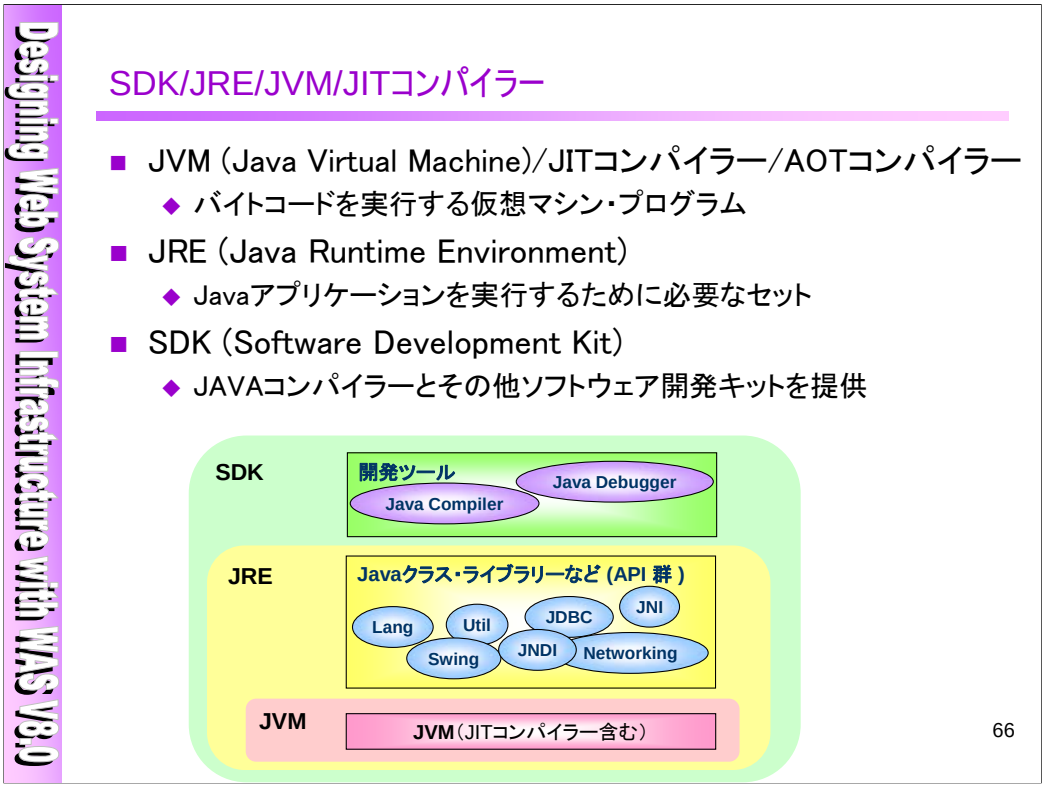


Java仮想マシン（通称JVM）とは，Javaプログラムを実行するためのソフトウェアです。JVMによってOSやハードウェアの違いを吸収することで，同一のバイトコードを複数のプラットフォームで実行することを可能にしています。つまり，異なるプラットフォーム間でもソースの互換性が保証されています。これが，「Write Once, Run Anywhere」（一度プログラムを作ると，どこでも実行させることができる。環境に合わせてプログラムを変更する必要がない）と呼ばれるJavaの特徴です。

JVMはクラスファイル（バイトコード）のフォーマットを正しく読み取り，そこに指定されている操作を実行することのみが仕様として規定されています。そのため，メモリー管理の仕組みなど，内部の仕組みをどのような形で実装するかはJVMの開発ベンダーに任せられており，ベンダー毎に実装方法が異なっていることもあれば，同一ベンダーによる実装でもバージョンによって実装方法がことなっていることもあります。



WebSphere Application ServerはJavaのエンタープライズ向けの仕様であるJ2EEに対応したアプリケーションの実行環境を提供するミドルウェアです。当セッションが対象としているWAS V8.0ではJ2EE1.6をサポートしています。



JVMはJavaのクラスファイルに格納されたバイトコードの実行環境です。コンパイルされたJavaソースコードはバイトコードという中間コードに変換されますが，これはそのままではCPUで実行することができません。そのため，中間コードを実行する仮想マシンが必要となります。

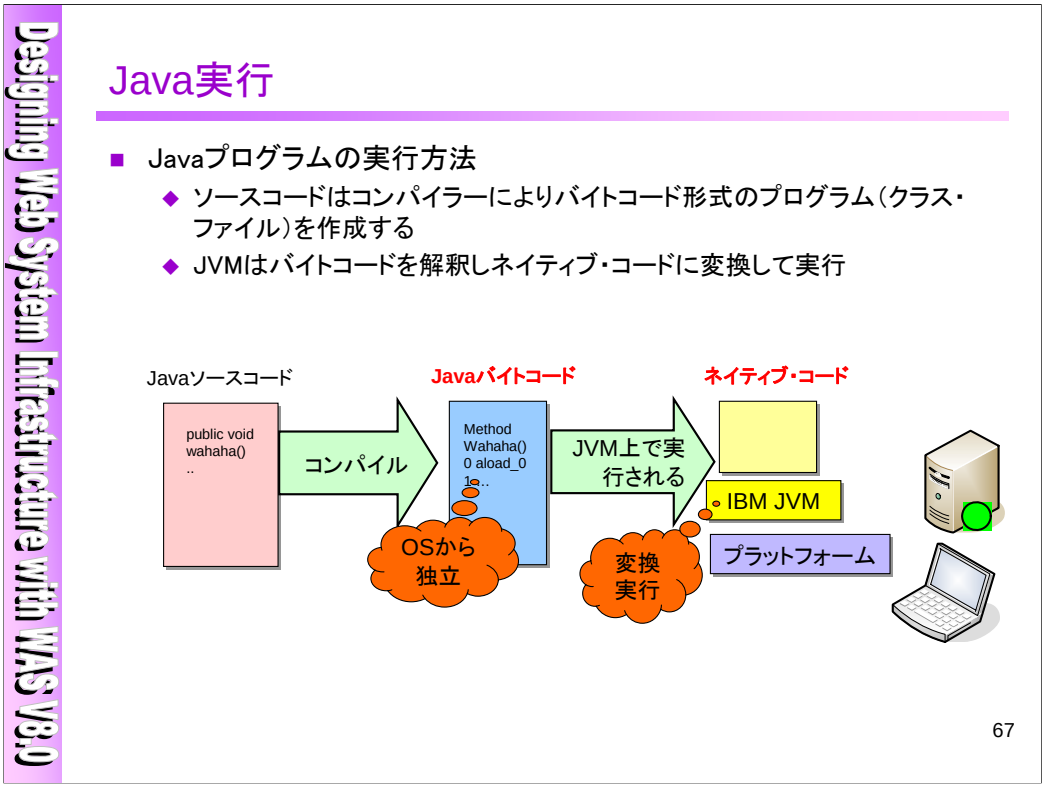
JIT（Just-In-Time）コンパイラーは高速化のための仕組みです。バイトコードを逐次翻訳・実行しているインタプリター動作では，プラットフォームの持つパフォーマンスを十分に引き出すことができません。そこで，頻繁に実行されるバイトコードを中心に，動的にバイトコードを（プラットフォームのCPUで直接実行できる）Nativeのコードにコンパイルする機能が実装されています。これがJITコンパイラーです。IBM JVMのJITコンパイラーは，コンパイルされたNativeコードを，実行時に取得された統計情報を元にさらに最適化する機能も備えています。

JITコンパイラーは，Javaプログラムの実行中にコンパイル動作を行うため，実行の中断が発生してしまうという問題があります。これを解消するため，プログラムの実行前に積極的にバイトコードをNativeコードに変換する仕組みがAOT（Ahead-Of-Time）コンパイラーです。AOTコンパイラーの生成するNativeコードは，実行時の統計情報が利用できない分だけ，JITコンパイラーが生成するコードよりも最適化のレベルが落ち，パフォーマンスが出ないという欠点があります。

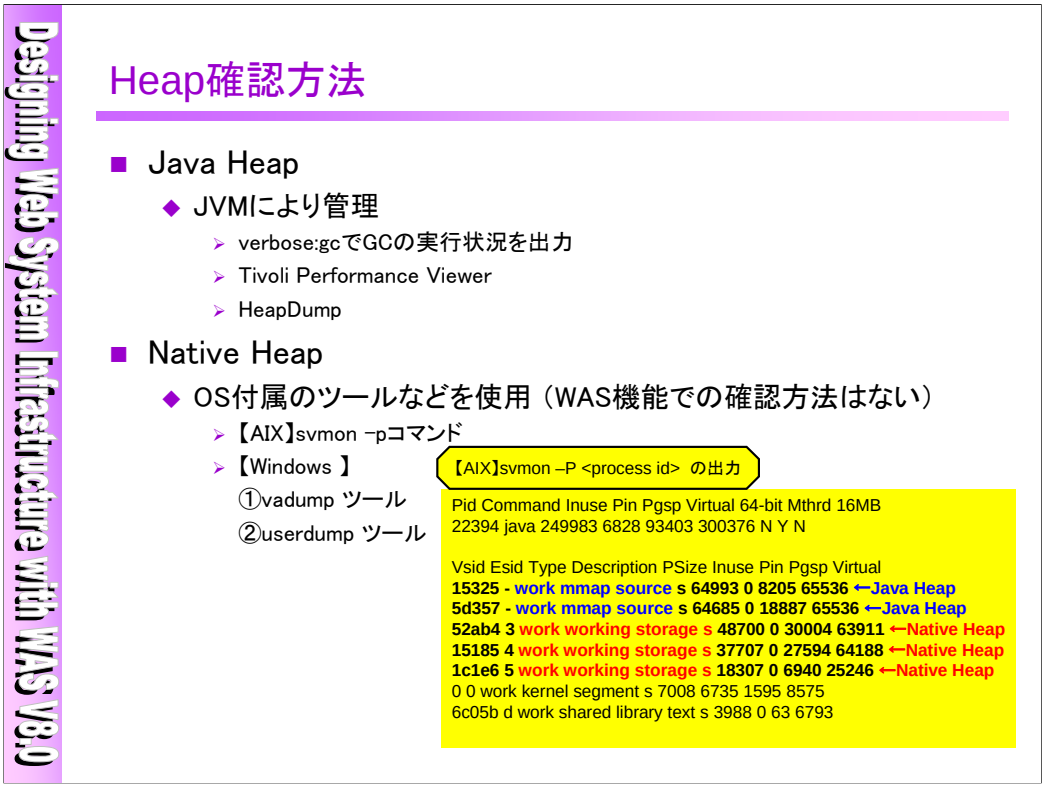
WAS V8で使用されるIBM JVMは，AOTとJITを必要に応じて組み合わせ，AOTが生成されたコードも繰り返し最適化することによってパフォーマンスを向上させています。

JREはJavaアプリケーションを実行するために必要なソフトウェアのセットであり，その実態はJVMと実行に必要な標準APIクラスファイルのライブラリー群，実行に必要なプログラム群からなります。JREはJVMを含みます。

SDKはJREの機能セットに加えて，Javaアプリケーションの開発の際に必要な，ソースファイルからクラスファイルを生成するためのコンパイラーなどを追加した開発キットです。SDKはJREを含みます（つまりJVMも含みます）。



Javaはコンパイル時にソースコードからプラットフォームに固有のマシンコードを直接生成するのではなく，バイトコードを生成することが大きな特徴です。各プラットフォームごとにバイトコードの実行機能（Java仮想マシン）を用意すれば，一度ソースを作成すれば様々なプラットフォーム・OSで実行できるようになっています。バイトコードはJVM上で実行されるのであり，OS上で実行されるわけではありません。つまり，バイトコードはOSに依存せず実行可能です。



Java heapをモニターもしくはダンプ取得するには，verbose:GC，TPV，HeapDumpなどが提供されています。
Native heapはOSに管理されているため，OS付属のツールをご使用ください。WASとして出力する機能は有しておりません。OS付属のツールを使用する場合の注意点として，使用するコマンド，ツールによってメモリーの算出方法が異なり，出力される結果に差異がある場合があります。

AIXではsvmonコマンドを実行することで取得可能です。Native Heap領域はworking storageとしてDescriptionに記載されます。
Windowsではvadumpと呼ばれるツール等を使用することで，JVMのアドレス空間の概略をわかりやすい形で表示する事が可能です。ツールに関してのより詳細な情報が必要な場合は，マイクロソフト社へお問い合わせください。
■<< developerWorks:Don't forget about memory – How to monitor your Java applications' Windows memory usage >>
http://www.ibm.com/developerworks/library/j-memusage/

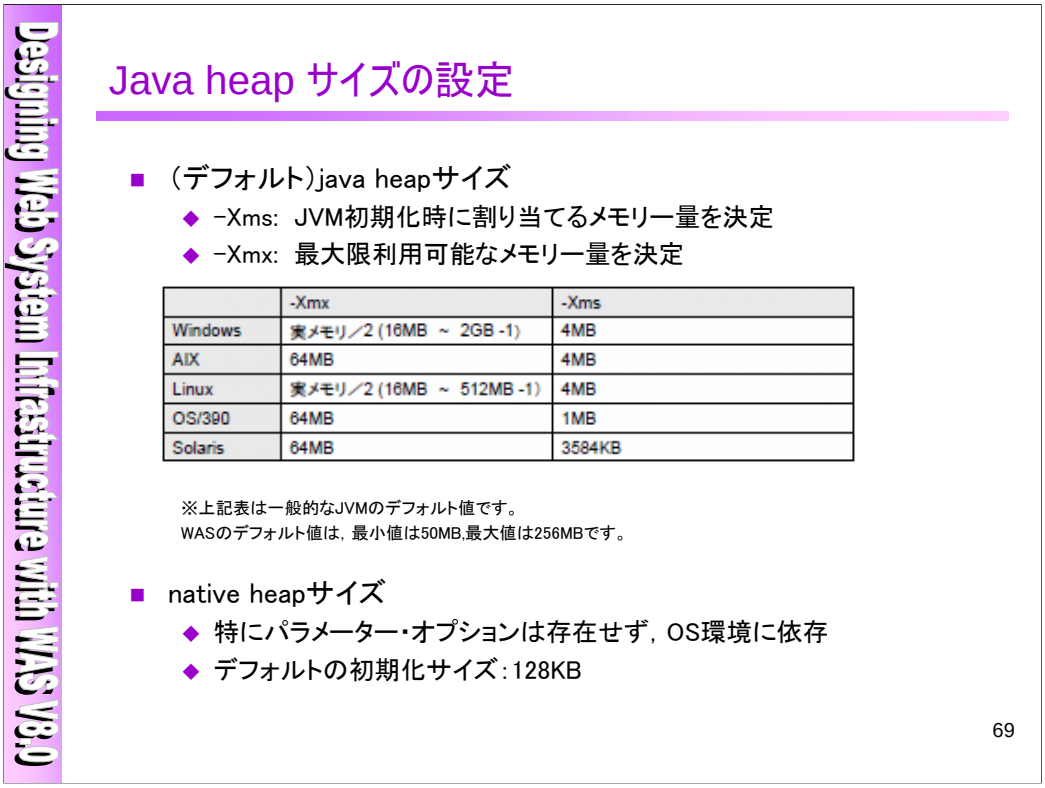
また，userdumpツールによりダンプファイルを作成し，jdmpviewコマンドを用いて解析する方法もあります。
Vadumpツール同様に，より詳細な情報が必要な場合は，マイクロソフト社へお問い合わせください。

■Userdump.exe ツールを使用してダンプ ファイルを作成する方法
①userdumpツールによりダンプ取得
%windir%¥system32¥kktools¥userdump.exe 〈JVMのプロセスID〉〈出力先〉
②jdmpviewコマンドを用いて解析
〈WAS_root〉java¥bin¥jdmpview.exe 〈①で取得したファイル〉

【User Mode Process Dumper Version 8.1】
http://www.microsoft.com/downloads/details.aspx?FamilyID=E089CA41-6A87-40C8-BF69-28AC08570B7E&displaylang=en

【Java Diagnostics Guide 5.0 – Using system dumps and the dump viewer】

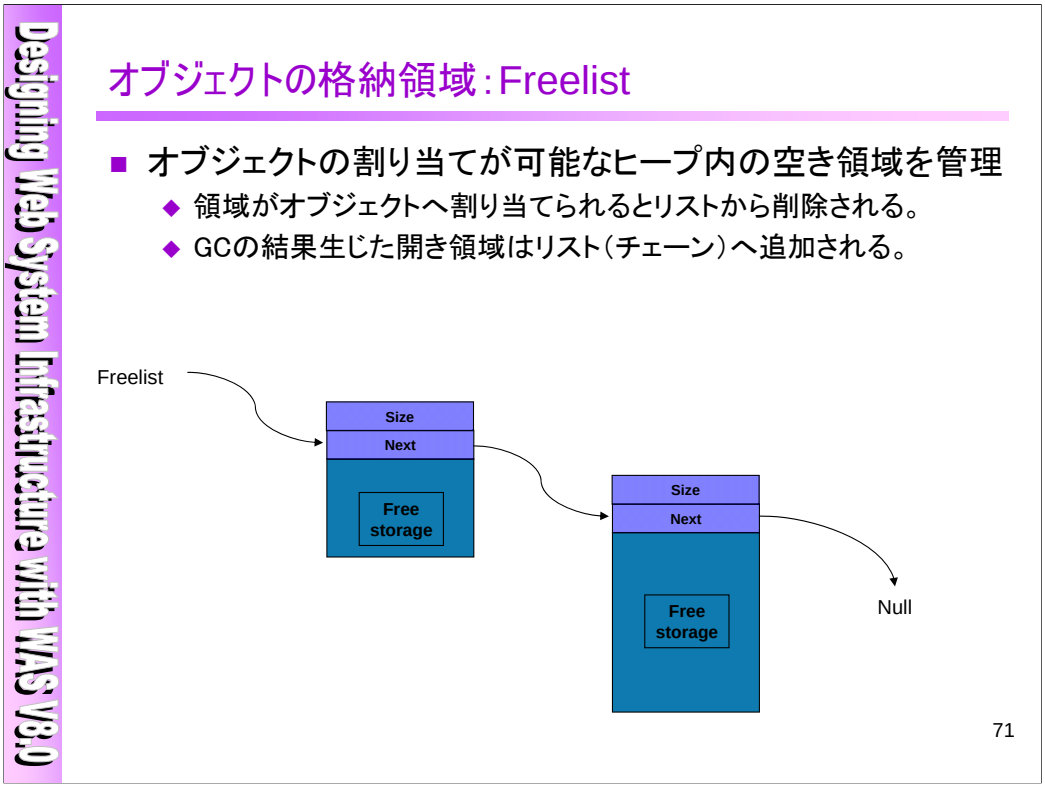
http://publib.boulder.ibm.com/infocenter/javasdk/v6r0/topic/com.ibm.java.doc.diagnostics.60/diag/tools/dump_viewer.html



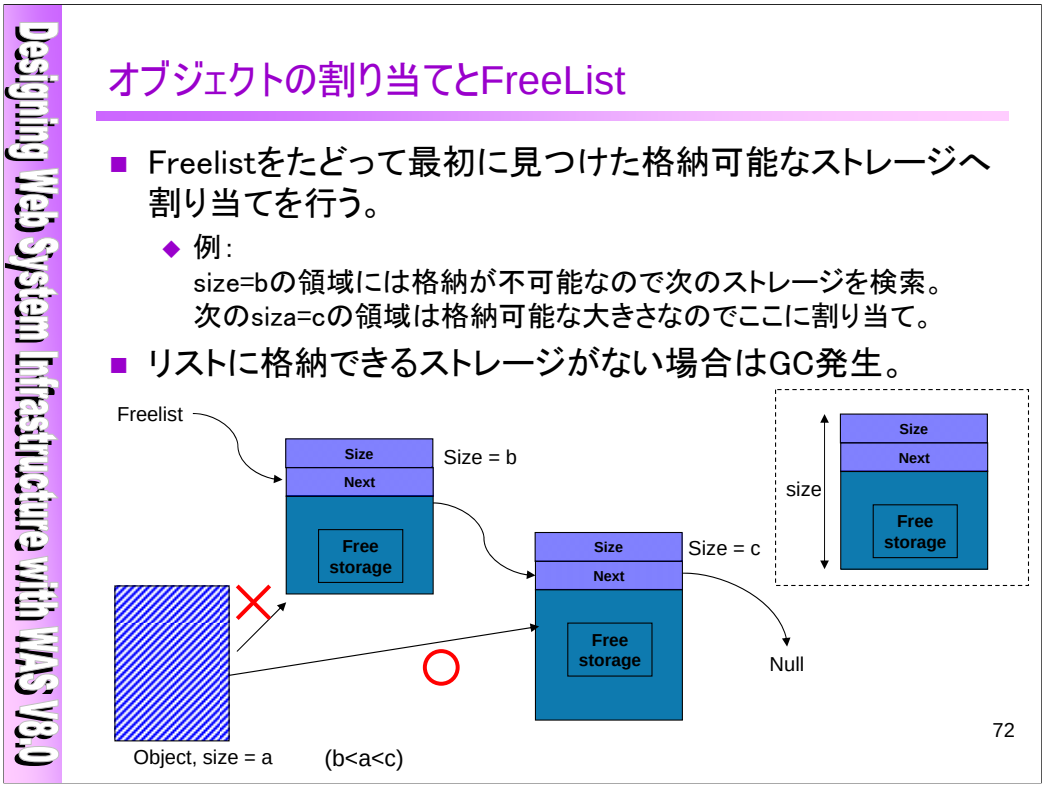
ヒープ・サイズの設定はチャートにあるようにJVM起動時に引数を与えることによって行うことができます。デフォルトでの設定値はチャートにあるようにプラットフォームによって異なります。

なお，この値はJVM一般の値であり，WebSphere Application Serverでは，デフォルトの最小値は50MB,最大値は256MBとなっています(分散プラットフォームの場合)。WebSphere Application Serverでは，管理コンソール上の各サーバの[仮想マシン]設定の欄に引数を渡すか，または最小・最大ヒープ・サイズの入力欄を使用することでヒープ・サイズの設定を行います。


Designing Web System Infrastructure with WAS V8.0
3. GCポリシー
・GCの基本
・GCポリシー
・GCチューニング
・WAS7.0 64bit
・パフォーマンスの最適化
70



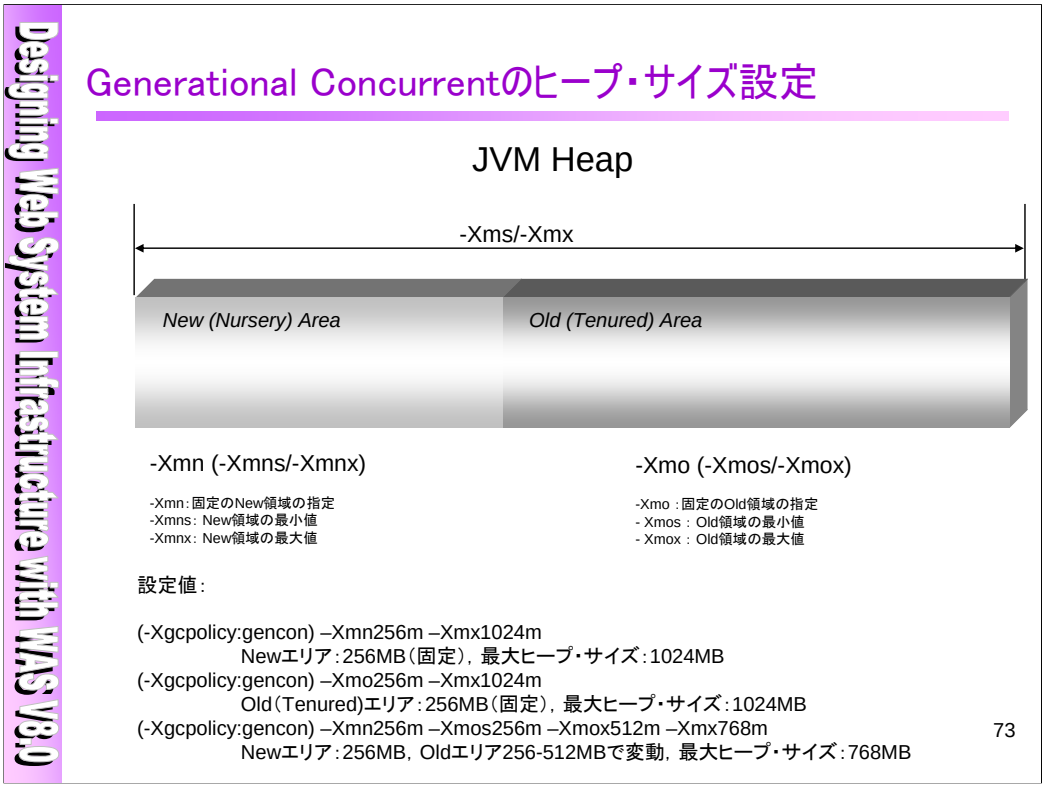
JVMが初期化された直後はオブジェクトの割り当てが行われていないために空き領域は連続したひとつの領域となっていますが，GCを繰り返すうちに領域は不連続となります。それら不連続な領域の集合である空き領域の管理を行うために，ストレージコンポーネントが空き領域情報をFreeListと呼ばれるリストとして所有しています。GCによってある領域が開放されると，その領域（チャンクと呼ばれる）はFreeListの末端に加えられ，Allocationのために使用されるとそのチャンクはFreeListから外されます。各チャンクにはヘッダ領域があり，チャンクのサイズと次のチャンクへの参照情報が格納されています。



Allocationを行う際には，オブジェクトが入る大きさのチャンクをFreeListの最初から検索して引き渡します。この作業は場合によってFreeList内のすべてのチャンクを検索することになり，パフォーマンスに大きな影響が生じる可能性があります。そこで，前回Allocationを実施した際に検索した結果から以下の情報を保持し，その情報を次のAllocation時のチャンクの検索に利用することで毎回Listの最初から検索しなくても済むようにしています。

〇前回検索した際にチェックしたチャンクの数

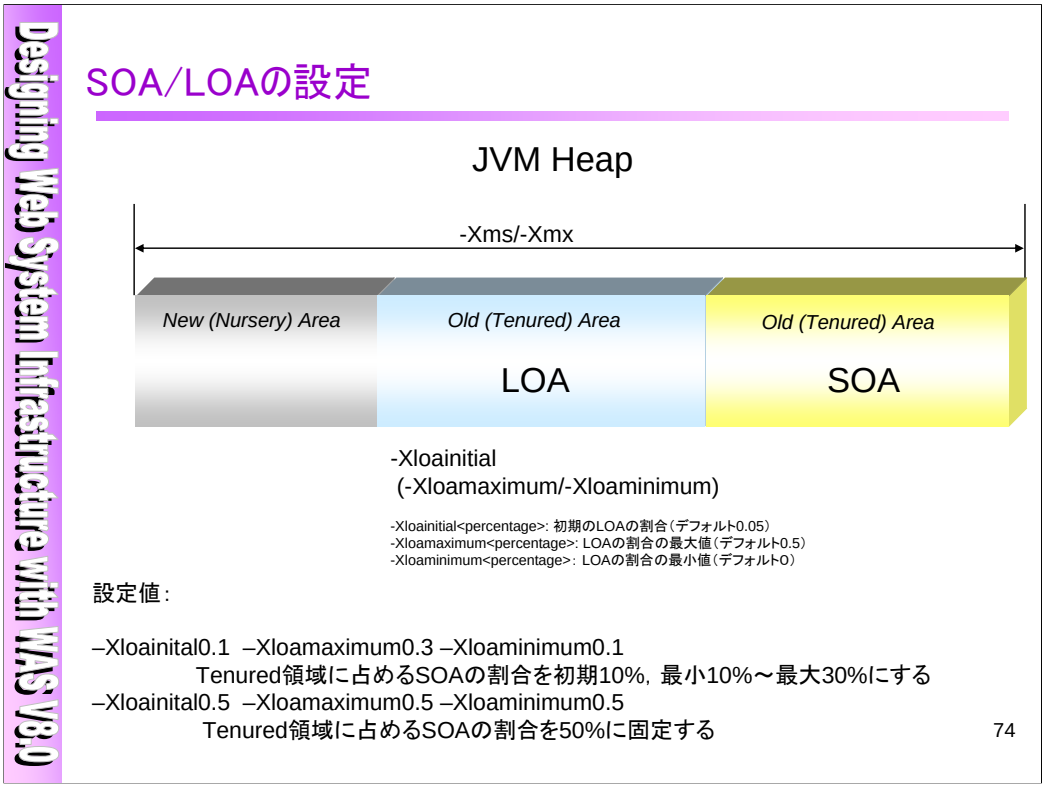
〇検索した中で一番大きなチャンクのサイズ



IBM JREにおけるGenerational Concurrentのヒープ・サイズ設定方法をご紹介します。この方式ではNew領域とOld領域をWASにデプロイするアプリケーションの性質に合わせて調整する必要があります。

他のポリシーと異なり，デプロイされているアプリケーションやシステム利用状況を考慮した決め細やかやパフォーマンスの調整が必須なため敷居は高くなりますが，逆に決め細やかに行うことができるようになったことで，VMのデフォルト挙動に頼らずに最適な設定を行うことができるようになります。

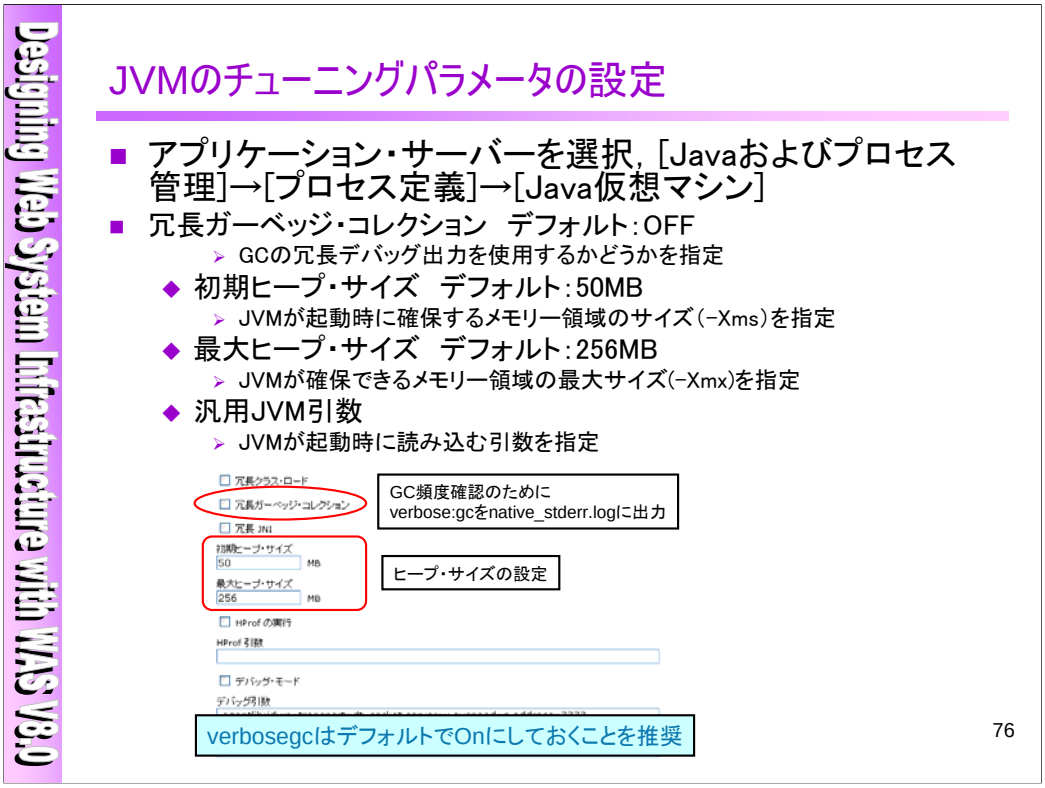
Javaヒープ全体の初期・最大サイズの設定についてはこれまでと同様です。New領域とOld領域の設定は，それぞれ値を固定する方法とある程度変動幅を持たせる方法の二通りを選択でき，組み合わせて使用することも可能です。



Tenured領域に占めるLOAの割合は，初期は5%で，0%～50%の間で自動的に調整されます。何らかの理由でこの割合を変更したい場合は，-Xloainitial/-Xloamaximum/-XloaminimumのJVM汎用引数で指定します。

アプリケーションが非常にサイズの大きいオブジェクトを一時オブジェクトとして多用している場合，LOA領域が枯渇してGCが頻発しているにもかかわらず，LOA領域が広がらないという現象が発生することがあります。このような場合は，手動でLOA領域を広げておきます。

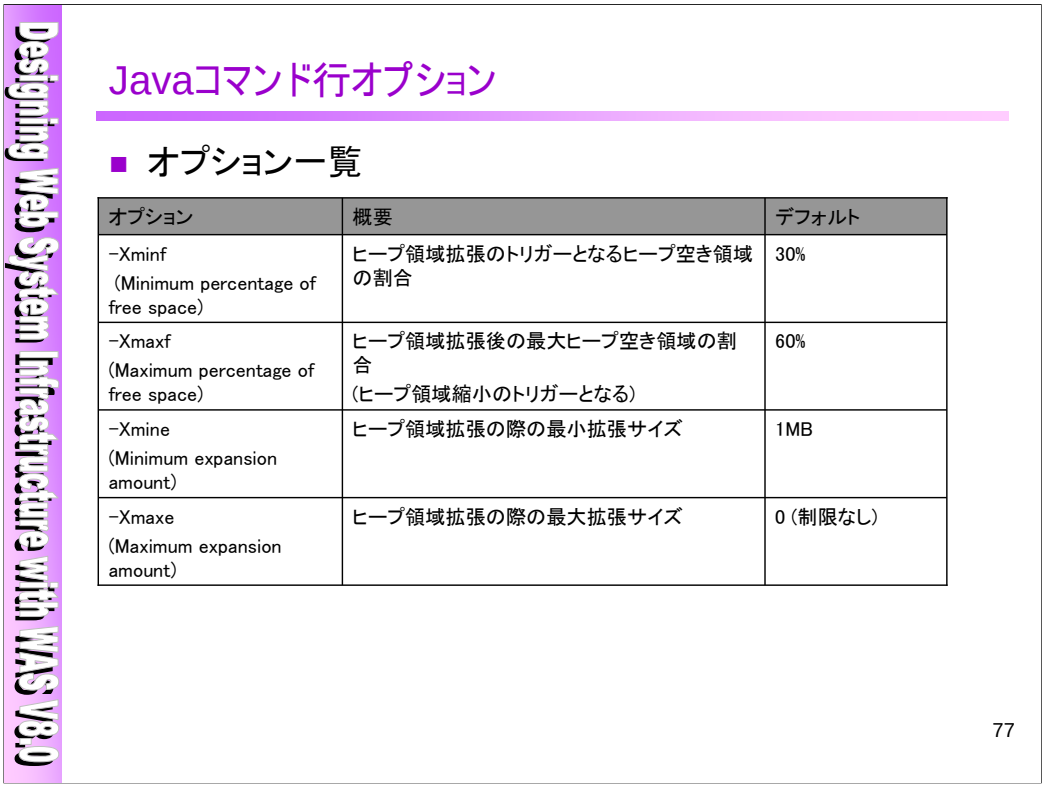

Designing Web System Infrastructure with WAS V8.0
4. GCチューニング
・GCの基本
・GCポリシー
・GCチューニング
・WAS7.0 64bit
・パフォーマンスの最適化
75



Javaヒープ・サイズの設定はアプリケーション・サーバーごとに行います。管理コンソールからアプリケーション・サーバーを選択し，[Javaおよびプロセス管理]→[プロセス定義]→[Java仮想マシン]を選択すると，設定画面になります。また，JVMに関してその他の引数を指定したい場合は，汎用JVM引数の欄に指定します。verbose:gcを出力させたい場合は，冗長ガーベッジコレクションのチェックをONにします。verbose:gcはデフォルトではOFFになっていますので出力させたい場合は，手動で設定が必要です。

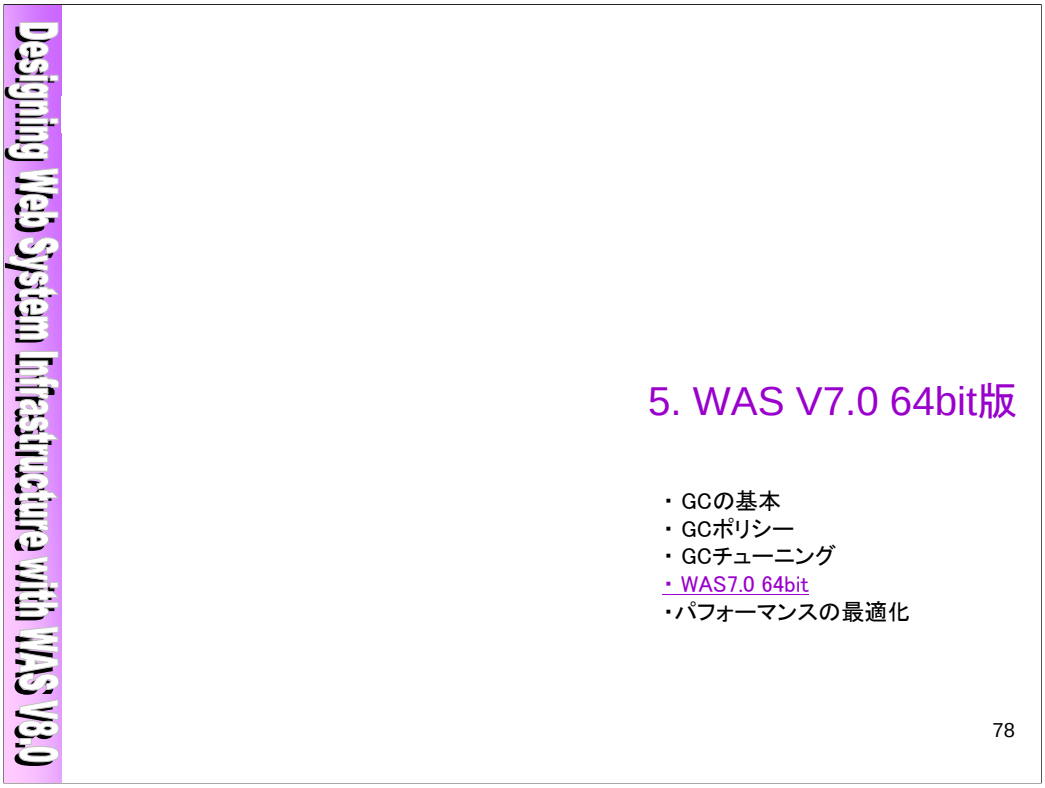
なおverbose:gcは本番稼動時もONにするかどうかの判断ですが，verbose:GCの出力によるパフォーマンスへの影響は通常は特にないと考えて構いません。

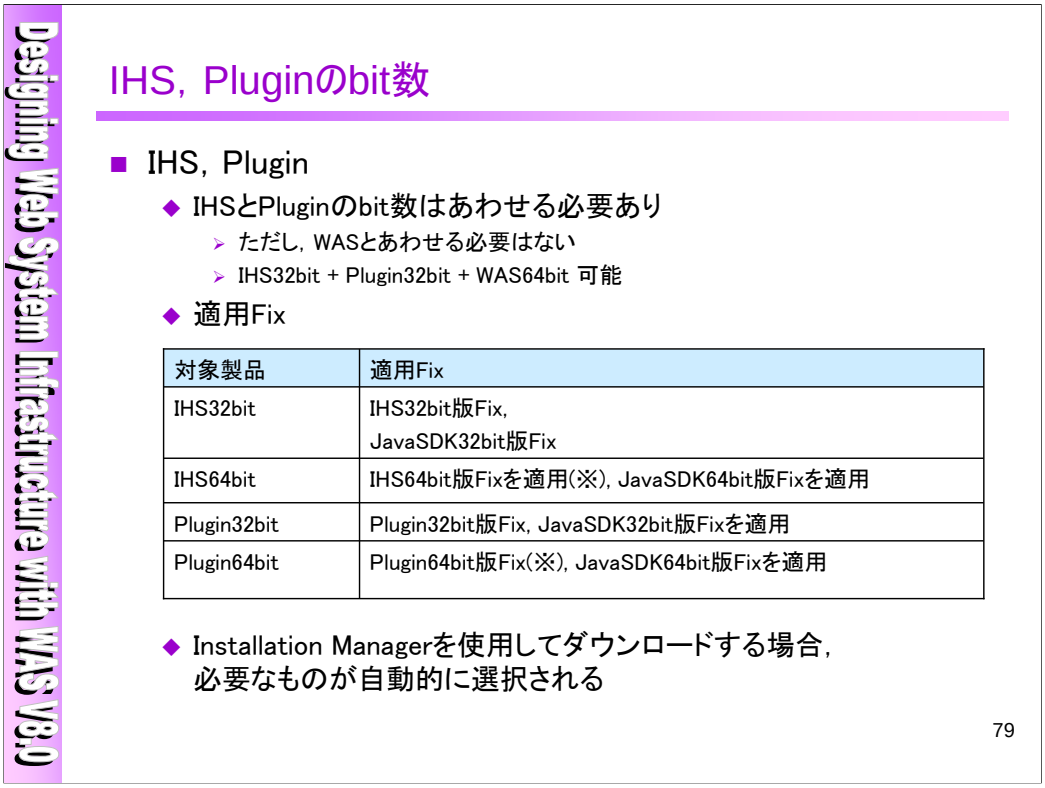
デフォルトではnative_stderr.logログに結果が出力されますので，適宜別のファイルに出力を変更するか，ログの管理を適切に運用してください。



オプション	概要	デフォルト
-Xminf (Minimum percentage of free space)	ヒープ領域拡張のトリガーとなるヒープ空き領域の割合	30%
-Xmaxf (Maximum percentage of free space)	ヒープ領域拡張後の最大ヒープ空き領域の割合 (ヒープ領域縮小のトリガーとなる)	60%
-Xmine (Minimum expansion amount)	ヒープ領域拡張の際の最小拡張サイズ	1MB
-Xmaxe (Maximum expansion amount)	ヒープ領域拡張の際の最大拡張サイズ	0 (制限なし)

Javaヒープ領域の拡張・縮小にかかわるJavaコマンド行オプション一覧です。


Designing Web System Infrastructure with WAS V8.0
5. WAS V7.0 64bit版
・GCの基本
・GCポリシー
・GCチューニング
・WAS7.0 64bit
・パフォーマンスの最適化
78



IHSおよびWebサーバー・プラグインにも32bit版，64bit版モジュールが存在します。

IHSおよびWebサーバー・プラグインとWASは互いに独立したモジュールであるため，IHSおよびWebサーバー・プラグインとWASのbit数を合わせる必要はありません。例えば，WASは64bit版モジュールを使用していても，IHSおよびWebサーバー・プラグインは32bit版でも問題ありません。ただし，IHSとWebサーバー・プラグインのbit数は合わせる必要があります。

適用するFixは上記のとおりになっています。通常は，Installation Managerを使用して，導入済み製品に適用するフィックスを自動ダウンロードします。

Designing Web System Infrastructure with WAS V8.0

6.パフォーマンスの最適化

- GCの基本
- GCポリシー
- GCチューニング
- WAS7.0 64bit
- パフォーマンスの最適化

80

共有クラス・キャッシュ（-Xshareclasses）の組み込まれる有効なオプションとその概要をまとめた一覧表です。

Designing Web System Infrastructure with WAS V8.0

82