Sun Java System Application Server Enterprise Edition 8.2 パフォーマンスチューニングガイド

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 820–1613
2007 年 2 月

Copyright 2007 Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 U.S.A. All rights reserved.

本書で説明する製品で使用されている技術に関連した知的所有権は、Sun Microsystems, Inc. に帰属します。特に、制限を受けることなく、この知的所有権には、米国特許、および米国をはじめとする他の国々で申請中の特許が含まれています。

U.S. Government Rights – Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements.

本製品には、サードパーティーが開発した技術が含まれている場合があります。

本製品の一部は Berkeley BSD システムより派生したもので、カリフォルニア大学よりライセンスを受けています。UNIX は、X/Open Company, Ltd. が独占的にライセンスしている米国ならびにほかの国における登録商標です。

Sun、Sun Microsystems、Sun のロゴマーク、Solaris のロゴマーク、Java Coffee Cup のロゴマーク、docs.sun.com、Java、 Solaris は、米国およびその他の国における米国 Sun Microsystems, Inc. (以下、米国 Sun Microsystems 社とします) の商標もしくは登録商標です。Sun のロゴマークおよび Solaris は、米国 Sun Microsystems 社の登録商標です。すべての SPARC 商標は、米国 SPARC International, Inc. のライセンスを受けて使用している同社の米国およびその他の国における商標または登録商標です。SPARC 商標が付いた製品は、米国 Sun Microsystems 社が開発したアーキテクチャーに基づくものです。

OPEN LOOK および Sun™ Graphical User Interface は、米国 Sun Microsystems 社が自社のユーザーおよびライセンス実施権者向けに開発しました。米国 Sun Microsystems 社は、コンピュータ産業用のビジュアルまたはグラフィカルユーザーインタフェースの概念の研究開発における米国 Xerox 社の先駆者としての成果を認めるものです。米国 Sun Microsystems 社は米国 Xerox 社から Xerox Graphical User Interface の非独占的ライセンスを取得しており、このライセンスは、OPEN LOOK GUI を実装するか、または米国 Sun Microsystems 社の書面によるライセンス契約に従う米国 Sun Microsystems 社のライセンス実施権者にも適用されます。

この製品は、米国の輸出規制に関する法規の適用および管理下にあり、また、米国以外の国の輸出および輸入規制に関する法規の制限を受ける場合があります。核、ミサイル、生物化学兵器もしくは原子力船に関連した使用またはかかる使用者への提供は、直接的にも間接的にも、禁止されています。このソフトウェアを、米国の輸出禁止国へ輸出または再輸出すること、および米国輸出制限対象リスト (輸出が禁止されている個人リスト、特別に指定された国籍者リストを含む) に指定された、法人、または団体に輸出または再輸出することは一切禁止されています。

本書は、「現状のまま」をベースとして提供され、商品性、特定目的への適合性または第三者の権利の非侵害の黙示の保証を含みそれに限定されない、明示的であるか黙示的であるかを問わない、なんらの保証も行われないものとします。

070629@17466

目次

はじめに 13

1 **Application Server** パフォーマンスチューニングの概要 19
プロセスの概要 19
▼ パフォーマンスチューニングの流れ 20
処理要件の理解 20
アプリケーションアーキテクチャー 21
セキュリティー要件 22
ハードウェアリソース 23
管理 24
一般的なチューニングの概念 25
容量計画 26
▼ 容量を決定する 26
ユーザーの期待 27
詳細情報 28

2 アプリケーションのチューニング 29
Java プログラミング上の指針 29
直列化と直列化復元を避ける 29
JSP とサーブレットのチューニング 32
コーディング手法の提案 32
EJB のパフォーマンスチューニング 34
目標 34
EJB コンポーネントの監視 35
一般的な指針 37
ローカルインタフェースとリモートインタフェースの使用 39
EJB トランザクションのパフォーマンスの改善 41

特殊な手法の使用 43
特定タイプの EJB コンポーネントのチューニングのヒント 46
JDBC とデータベースアクセス 50
メッセージ駆動型 Beans のチューニング 51

3 Application Server のチューニング 53
配備の設定 53
自動配備を無効にする 54
プリコンパイルされた JavaServer Pages を使用する 54
アプリケーションの動的再読み込みを無効にする 54
ロガーの設定 55
一般的な設定 55
ログレベル 55
Web コンテナの設定 56
セッションプロパティー: セッションタイムアウト 56
マネージャープロパティー: 取得間隔 56
JSP の動的再読み込みを無効にする 57
EJB コンテナの設定 57
EJB コンテナの監視 57
EJB コンテナのチューニング 57
Java メッセージサービスの設定 63
トランザクションサービスの設定 64
トランザクションサービスの監視 64
トランザクションサービスのチューニング 65
HTTP サービスの設定 66
HTTP サービスの監視 66
HTTP サービスのチューニング 71
HTTP リスナー設定のチューニング 78
Version 7 からの移行 82
ORB の設定 83
概要 83
クライアントから ORB への接続方法 84
ORB の監視 84
ORB のチューニング 85
スレッドプールのサイジング 88

IIOP メッセージの検査 88
Java 直列化による ORB のパフォーマンス改善 89
▼ Java 直列化を有効にする 89
スレッドプールの設定 90
スレッドプールのチューニング (Unix /Linux のみ) 90
リソース 91
JDBC 接続プールの設定 91
コネクタ接続プールの設定 94

4 Java 実行システムのチューニング 97
Java 仮想マシンの設定 97
メモリーとガベージコレクションの管理 98
ガベージコレクタのチューニング 98
ガベージコレクションの追跡 100
ガベージコレクタのその他の設定 101
Java ヒープのチューニング 101
Windows での DLL ベースアドレスの再割り当て 104
▼ Application Server DLL のベースアドレスの再割り当てを行う 105
詳細情報 105

5 オペレーティングシステムのチューニング 107
サーバーのスケーリング 107
プロセッサ 107
メモリー 108
ディスク容量 108
ネットワーク関連機能 108
Solaris のチューニング 109
チューニングパラメータ 109
ファイル記述子の設定 111
代替スレッドの使用 111
Linux の設定 112
x86 上の Solaris のチューニング 113
セマフォーと共有メモリー 114
ファイル記述子 114
IP スタックの設定 114

Linux プラットフォームのチューニング 115
ファイル記述子 115
仮想メモリー 117
ネットワークインタフェース 117
ディスク入出力の設定 117
▼ SCSI 以外のディスクのディスク入出力パフォーマンスを調整する 117
TCP/IP 設定 118
▼ TCP/IP 設定を調整する 118

6 高可用性のチューニング 119
HADB のチューニング 119
ディスクの使用 119
メモリーの割り当て 121
パフォーマンス 122
オペレーティングシステム構成 129
高可用性に関する Application Server のチューニング 131
セッションの持続性の頻度のチューニング 131
セッションの持続性の範囲 132
セッションサイズ 133
ステートフルセッション Beans のチェックポイント 134
JDBC 接続プールの設定 134
ロードバランサの設定 135
診断プログラムの有効化 135

索引 137

図目次

図 1–1 J2EE アプリケーションモデル 21

表目次

表 1–1 パフォーマンスチューニングのロードマップ 19
表 1–2 パフォーマンスに影響する要因 25
表 3–1 Bean タイプ別のプール処理またはキャッシュ処理 58
表 3–2 EJB のキャッシュおよびプール設定 62
表 3–3 Grizzly プロパティーと PWC プロパティーとの対応関係 80
表 3–4 チューニング可能設定の、version 7 から version 8 へのマッピング 83
表 3–5 チューニング可能な ORB 設定 85
表 3–6 接続プールのサイジング 92
表 4–1 プロセスあたりの最大アドレス空間 102
表 5–1 Solaris のパラメータのチューニング 109

例目次

例 4–1 Solaris でのヒープ設定 103
例 4–2 Windows でのヒープ設定 105

はじめに

『パフォーマンスチューニングガイド』では、アプリケーションおよび Sun Java ™ System Application Server から、最大のパフォーマンス、スケーラビリティー、および信頼性を獲得する方法について説明します。このマニュアルは、サーバー管理者、J2EE アプリケーション開発者、ネットワーク管理者、および評価担当者による使用を想定して記述されています。

この「はじめに」では、Sun Java™ System Application Server のマニュアルセット全体についての情報および表記規則を示します。

Application Server のマニュアルセット

Application Server のマニュアルセットは、配備の計画とシステムのインストールについて説明しています。スタンドアロンの Application Server マニュアルの URL は `http://docs.sun.com/app/docs/coll/1310.4` です。Sun Java Enterprise System (Java ES) Application Server マニュアルの URL は `http://docs.sun.com/app/docs/coll/1659.1` です。Application Server についての理解を深めるには、次の表に示されている各マニュアルを、表中での順番どおりに参照してください。

表 P–1　Application Server のマニュアルセットの内容

マニュアル名	説明
『リリースノート』	ソフトウェアとマニュアルに関する最新情報。サポート対象のすべてのハードウェア、オペレーティングシステム、Java Development Kit (JDK™)、およびデータベースドライバを表形式で要約して示しています。
『Quick Start Guide』	Application Server 製品の使用を開始するための手順。
『Installation Guide』	ソフトウェアとそのコンポーネントのインストール。
『配備計画ガイド』	最適な方法で確実に Application Server を導入するための、システムニーズや企業ニーズの分析。サーバーを配備する際に注意すべき一般的な問題や懸案事項についても説明しています。

表 P–1 Application Server のマニュアルセットの内容 (続き)

マニュアル名	説明
『Developer's Guide』	J2EE コンポーネントおよび API 用のオープン Java 標準モデルに従い、Application Server 上で実行することを目的とする Java 2 Platform, Enterprise Edition (J2EE™ プラットフォーム) アプリケーションの作成と実装。開発ツール、セキュリティー、デバッグ、配備、ライフサイクルモジュールの作成などについての全般的な情報も提供します。
『J2EE 1.4 Tutorial』	J2EE アプリケーションの開発のための J2EE 1.4 プラットフォーム技術および API の使用。
『管理ガイド』	管理コンソールからの、Application Server サブシステムおよびコンポーネントの設定、管理、および配備。
『高可用性 (HA) 管理ガイド』	高可用性データベースのための、インストール後の設定と管理に関する解説。
『Administration Reference』	Application Server 設定ファイル `domain.xml` の編集。
『アップグレードと移行』	新しい Application Server プログラミングモデルへのアプリケーションの移行 (特に Application Server 6.x または 7 からの移行)。このマニュアルでは、製品仕様の非互換性をもたらす可能性のある、隣接した製品リリース間の相違点や設定オプションについても説明しています。
『パフォーマンスチューニングガイド』	パフォーマンスを向上させるための Application Server の調整。
『トラブルシューティングガイド』	Application Server の問題の解決。
『Error Message Reference』	Application Server のエラーメッセージの解決。
『Reference Manual』	Application Server で使用できるユーティリティーコマンド。マニュアルページのスタイルで記述されています。`asadmin` コマンド行インタフェースも含みます。

関連マニュアル

Application Server は、単体で購入することが可能です。あるいは、ネットワークまたはインターネット環境にわたって分散しているエンタープライズアプリケーションをサポートするソフトウェアインフラストラクチャーである Java ES のコンポーネントとして購入することもできます。Application Server を Java ES のコンポーネントとして購入した場合は、`http://docs.sun.com/coll/1657.1` にあるシステムマニュアルをよく読むことをお勧めします。Java ES およびそのコンポーネントに関するすべてのマニュアルの URL は `http://docs.sun.com/prod/entsys.5` です。

その他の Sun Java System サーバー製品のマニュアルについては、次を参照してください。

- Message Queue のマニュアル
- Directory Server のマニュアル
- Web Server のマニュアル

さらに、次のリソースが役立つことがあります。

- J2EE 1.4 仕様 (`http://java.sun.com/j2ee/1.4/docs/index.html`)
- J2EE 1.4 チュートリアル (`http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html`)
- J2EE Blueprints (`http://java.sun.com/reference/blueprints/index.html`)

デフォルトのパスとファイル名

次の表は、このマニュアルで使用するデフォルトのパスやファイル名について説明したものです。

表 P–2 デフォルトのパスとファイル名

プレースホルダ	説明	デフォルト値

install-dir	Application Server のベースインストールディレクトリを表します。	Solaris™ プラットフォーム上の Sun Java Enterprise System (Java ES) インストール:
		`/opt/SUNWappserver/appserver`
		Linux プラットフォーム上の Java ES インストール:
		`/opt/sun/appserver/`
		Solaris および Linux プラットフォームへのインストールで、ルートユーザーでない場合:
		ユーザーのホームディレクトリ `/SUNWappserver`
		Solaris および Linux プラットフォームへのインストールで、ルートユーザーである場合:
		`/opt/SUNWappserver`
		Windows のすべてのインストールの場合:
		SystemDrive`:\Sun\AppServer`

domain-root-dir	すべてのドメインを含むディレクトリを表します。	Solaris プラットフォーム上の Java ES インストール:
		`/var/opt/SUNWappserver/domains/`
		Linux プラットフォーム上の Java ES インストール:
		`/var/opt/sun/appserver/domains/`
		そのほかのすべてのインストールの場合:
		install-dir`/domains/`

表 P–2 デフォルトのパスとファイル名 (続き)

プレースホルダ	説明	デフォルト値
domain-dir	ドメインのディレクトリを表します。 設定ファイルには、次のように表される *domain-dir* があります。 `${com.sun.aas.instanceRoot}`	*domain-root-dir*/*domain-dir*
instance-dir	サーバーインスタンスのディレクトリを表します。	*domain-dir*/*instance-dir*

表記上の規則

このマニュアルでは、次のような字体や記号を特別な意味を持つものとして使用します。

表 P–3 表記上の規則

字体または記号	意味	例
`AaBbCc123`	コマンド名、ファイル名、ディレクトリ名、画面上のコンピュータ出力、コード例を示します。	`.login` ファイルを編集します。 `ls -a` を使用してすべてのファイルを表示します。 `machine_name% you have mail.`
`AaBbCc123`	ユーザーが入力する文字を、画面上のコンピュータ出力と区別して示します。	`machine_name%` **`su`** `Password:`
aabbcc123	変数を示します。実際に使用する特定の名前または値で置き換えます。	ファイルを削除するには、`rm` *filename* と入力します。
『 』	参照する書名を示します。	『コードマネージャー・ユーザーズガイド』を参照してください。
「 」	参照する章、節、ボタンやメニュー名、強調する単語を示します。	第 5 章「衝突の回避」を参照してください。 この操作ができるのは、「スーパーユーザー」だけです。
\	枠で囲まれたコード例で、テキストがページ行幅を超える場合に、継続を示します。	`sun%` **`grep '^#define \`** **`XV_VERSION_STRING'`**

コード例は次のように表示されます。

- C シェル

  ```
  machine_name% command y|n [filename]
  ```

- C シェルのスーパーユーザー

  ```
  machine_name# command y|n [filename]
  ```

- Bourne シェルおよび Korn シェル

  ```
  $ command y|n [filename]
  ```

- Bourne シェルおよび Korn シェルのスーパーユーザー

  ```
  # command y|n [filename]
  ```

[ ] は省略可能な項目を示します。上記の例は、*filename* は省略してもよいことを示しています。

| は区切り文字 (セパレータ) です。この文字で分割されている引数のうち 1 つだけを指定します。

キーボードのキー名は英文で、頭文字を大文字で示します (例: Shift キーを押します)。ただし、キーボードによっては Enter キーが Return キーの動作をします。

ダッシュ (-) は 2 つのキーを同時に押すことを示します。たとえば、Ctrl-D は Control キーを押したまま D キーを押すことを意味します。

記号の規則

この表は、このマニュアルで使用される記号について説明したものです。

表 P–4 記号の規則

記号	説明	例	意味
[ ]	省略可能な引数やコマンドオプションが含まれます。	ls [-l]	-l オプションは必須ではありません。
{ | }	必須のコマンドオプションの選択肢を囲みます。	-d {y|n}	-d オプションには y 引数か n 引数のいずれかを使用する必要があります。
${ }	変数参照を示します。	${com.sun.javaRoot}	com.sun.javaRoot 変数の値を参照します。

表 P–4 記号の規則 (続き)

記号	説明	例	意味
-	同時に押すキーを示します。	Control-A	Control キーを押しながら A キーを押します。
+	順番に押すキーを示します。	Ctrl+A+N	Control キーを押してから放し、それに続くキーを押します。
→	グラフィカルユーザーインタフェースでのメニュー項目の選択順序を示します。	「ファイル」→「新規」→「テンプレート」	「ファイル」メニューから「新規」を選択します。「新規」サブメニューから「テンプレート」を選択します。

マニュアル、サポート、およびトレーニング

Sun のサービス	URL	内容
マニュアル	http://jp.sun.com/documentation/	PDF 文書および HTML 文書をダウンロードできます。
サポートおよびトレーニング	http://jp.sun.com/supportraining/	技術サポート、パッチのダウンロード、および Sun のトレーニングコース情報を提供します。

◆ ◆ ◆ 第 1 章

Application Server パフォーマンスチューニングの概要

Sun Java System Application Server およびこの製品に配備されたアプリケーションのパフォーマンスは、いくつかの配備設定およびサーバー設定を調整することによって大きく改善することができます。ただし、環境およびパフォーマンスに関しての目標を理解することが重要です。本稼働環境で最適な設定と、開発環境で最適な設定は必ずしも一致しません。この章では、パフォーマンスチューニングのプロセスを紹介し、次のトピックについて説明します。

- 19 ページの「プロセスの概要」
- 20 ページの「処理要件の理解」
- 25 ページの「一般的なチューニングの概念」
- 28 ページの「詳細情報」

プロセスの概要

次の表は、管理プロセスの全体の概略と、一連の作業の中でパフォーマンスチューニングがどこに位置するかを示したものです。

表 1–1 パフォーマンスチューニングのロードマップ

ステップ	タスクの説明	参照マニュアル
1	設計: 高可用性トポロジを決定し、Application Server および高可用性データベース (HADB) システムを設定します。	『配備計画ガイド』
2	容量計画: 高いパフォーマンスを発揮するための十分なリソースがシステムに搭載されていることを確認します。	『配備計画ガイド』
3	インストール: Application Server ソフトウェアとともに、または単体で HADB ソフトウェアをインストールします。	『インストールガイド』

表 1–1 パフォーマンスチューニングのロードマップ (続き)

ステップ	タスクの説明	参照マニュアル
4	配備: アプリケーションをインストールして実行します。Application Server のサブシステムおよびコンポーネントの設定および管理方法について理解します。	『管理ガイド』
5	チューニング: アプリケーション、Java 実行時システム、オペレーティングシステム、HADB、および Application Server を調整します。	『パフォーマンスチューニングガイド』

▼ パフォーマンスチューニングの流れ

アプリケーション開発者は、本稼働環境での運用を開始する前にアプリケーションを調整する必要があります。アプリケーションのチューニングにより、パフォーマンスが劇的に改善されることがよくあります。システム管理者は、アプリケーションを調整したあと、またはアプリケーションチューニングを待機する必要があるがその間にもできるだけパフォーマンスを改善したい場合に、次のリストに示した残りのステップを実行します。

この一連のステップは、パフォーマンスチューニングを行なっているときに実行するのが理想的です。

1. 第 2 章の説明に従ってアプリケーションを調整します。
2. 第 2 章の説明に従ってサーバーを調整します。
3. 第 6 章の説明に従って、高可用性データベースを調整します。
4. 第 2 章の説明に従って、Java 実行時システムを調整します。
5. 第 5 章の説明に従って、オペレーティングシステムを調整します。

処理要件の理解

Application Server にアプリケーションを配備して調整する前に、運用環境を明確に定義することが重要です。運用環境は、次のような全体レベルの制約や要件によって決定されます。

- 21 ページの「アプリケーションアーキテクチャー」
- 22 ページの「セキュリティー要件」
- 23 ページの「ハードウェアリソース」

アプリケーションアーキテクチャー

J2EE アプリケーションモデルは、次の図に示すように非常に柔軟です。アプリケーションアーキテクトは、アプリケーションロジックを機能別に多数の層に分割できます。プレゼンテーション層は一般にサーブレット技術や JSP 技術を使用して実装され、Web コンテナ内で実行されます。

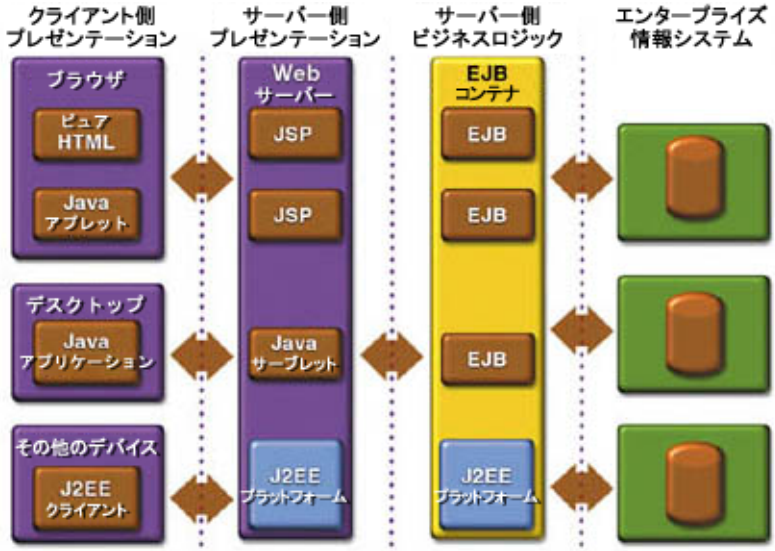

図 1–1　J2EE アプリケーションモデル

ある程度までの複雑さのエンタープライズアプリケーションは、全面的にサーブレット技術と JSP 技術を使用して開発できます。より複雑なビジネスアプリケーションでは、多くの場合 Enterprise JavaBeans (EJB) コンポーネントを使用します。Application Server では、Web コンテナと EJB コンテナを単一のプロセスに統合します。サーブレットから EJB コンポーネントへのローカルアクセスは非常に効率的です。ただし、一部のアプリケーション配備では、EJB コンポーネントを独立したプロセスで実行すること、および、EJB コンポーネントにサーブレットだけでなくスタンドアロンのクライアントアプリケーションからアクセスできることが必要な場合があります。サーバー管理者はアプリケーションアーキテクチャーに基づいて、Application Server を複数層に展開するか、あるいはプレゼンテーションロジックとビジネスロジックの両方を単一層に配置するかを選択できます。

新しい Application Server 配備を設計する前、また既存のアプリケーションサーバー配備に新しいビジネスアプリケーションを配備するときは、アプリケーションアーキテクチャーについて理解することが重要です。

セキュリティー要件

ほとんどのビジネスアプリケーションにはセキュリティーが必要です。この節では、セキュリティー関連の考慮事項および決定事項について説明します。

ユーザーの認証と承認

アプリケーションのユーザーは認証を受ける必要があります。Application Server ではユーザー認証の方法として、ファイルベース、LDAP、および Solaris の 3 つの選択肢が用意されています。

デフォルトのファイルベースセキュリティーレルムは、新しいアプリケーションの開発およびテストが行われる開発者環境に適しています。サーバー管理者は配備時に、Lightweight Directory Access Protocol (LDAP) または Solaris セキュリティーレルムを選択できます。多くの大企業では、LDAP ベースのディレクトリサーバーを使用して、従業員や顧客のプロファイルを管理しています。まだディレクトリサーバーを利用していない中小企業では、Solaris セキュリティーインフラストラクチャーへの投資を活用すると有利な場合があります。

セキュリティーレルムの詳細は、『Sun Java System Application Server Enterprise Edition 8.2 管理ガイド』の第 9 章「セキュリティーの設定」を参照してください。

選択した認証機構のタイプによっては、配備へのハードウェア追加が必要になる場合があります。通常、ディレクトリサーバーは独立したサーバー上で実行され、レプリケーションや高可用性のためのバックアップサーバーが必要な場合もあります。配備、サイジング、および可用性のガイドラインは、Sun Java System Directory Server のマニュアルを参照してください。

認証されたユーザーによるアプリケーション機能へのアクセスには、承認チェックも必要な場合があります。アプリケーションでロールに基づく J2EE 承認チェックを使用する場合、アプリケーションサーバーが何らかの追加チェックを実行し、これによって追加のオーバーヘッドが発生します。容量計画を実施するときは、この追加のオーバーヘッドを考慮する必要があります。

暗号化

セキュリティー上の理由から、機密性のあるユーザー入力およびアプリケーション出力を暗号化する必要があります。ほとんどのビジネス指向 Web アプリケーションでは、ブラウザと Application Server の間の通信フローの全部または一部を暗号化します。オンラインショッピングアプリケーションでは、ユーザーが購入を完了するとき、または個人データを入力するときにトラフィックを暗号化します。ニュースやメディアなどのポータルアプリケーションでは、通常は暗号化を行いません。SSL (Secure Sockets Layer) はもっとも一般的なセキュリティーフレームワークであり、多くのブラウザおよびアプリケーションサーバーでサポートされています。

Application Server は SSL 2.0 および 3.0 をサポートし、各種の暗号化方式群のソフトウェアサポートが含まれています。また、パフォーマンスをさらに高めるためにハードウェア暗号化カードの統合もサポートしています。セキュリティー上の考慮事項によって、統合ソフトウェアの暗号化を使用している場合に特に影響を受けるのは、ハードウェアのサイジングおよび容量計画です。

配備の暗号化ニーズを評価するときは、次のことを考慮します。

- セキュリティーの面から見たアプリケーションの性質はどのようなものか。アプリケーションの入力と出力をすべて暗号化するか、それとも一部のみか。セキュリティー保護された状態で転送する必要がある情報の割合はどの程度か。
- アプリケーションを配備しようとしているアプリケーションサーバーがインターネットに直接接続されているか。アプリケーションサーバー層およびバックエンドデータベースシステムから独立した非武装ゾーン (DMZ) に Web サーバーを配置する予定か。

  DMZ を使用する配備は、高いレベルのセキュリティーが必要な場合に推奨されます。また、アプリケーションに大量の静的テキストや画像コンテンツがある場合や、最も高いレベルでセキュリティー保護されているファイアウォールの内側に配置された Application Server 上で実行されるビジネスロジックがある場合にも便利です。Application Server では、一般的な Web サーバーとの統合を可能にするために、セキュリティー保護された逆プロキシプラグインが提供されています。Application Server は、DMZ 内の Web サーバーとしての配備および使用も可能です。
- DMZ 内の Web サーバーと、次の層のアプリケーションサーバーとの間で暗号化が必要か。Application Server で提供される逆プロキシプラグインは、Web サーバー層とアプリケーションサーバー層の間での SSL 暗号化をサポートします。SSL が有効な場合、ハードウェアの容量計画では暗号化のポリシーおよび機構を考慮に入れる必要があります。
- ソフトウェア暗号化を使用する予定の場合、次のことを考慮します。
  - 特定のセキュリティー要件のもとで、システムの各層で予測されるパフォーマンスのオーバーヘッドはどの程度か。
  - それぞれの選択肢のパフォーマンス特性およびスループット特性。

Web サーバーと Application Server の間で通信を暗号化する方法の詳細は、『Sun Java System Application Server Enterprise Edition 8.2 管理ガイド』の第 9 章「セキュリティーの設定」を参照してください。

ハードウェアリソース

使用可能なハードウェアリソースの種類と量は、パフォーマンスチューニングおよびサイト計画に大きな影響を及ぼします。

Application Server は、卓越した垂直方向のスケーラビリティーを提供します。ただ 1 つのアプリケーションサーバープロセスを使用しながら、複数の高性能 CPU を効率的に利用するような構成への拡張が可能です。アプリケーションサーバーインスタンスの数が少ないほど、保守が容易になり管理コストが低減されます。また、関連アプリケーションを配備するアプリケーションサーバーの数が少なければデータの局所性が高まってパフォーマンスが向上し、共存アプリケーション間でのキャッシュデータの再利用も活発化します。そのようなサーバーでは、負荷の増加に対応できる十分な量のメモリー、ディスク領域、およびネットワーク容量を備える必要もあります。

Application Server は、それほど高性能でない多数のハードウェアユニットで構成される大規模「ファーム」への配備も可能です。ビジネスアプリケーションを複数のサーバーインスタンス間でパーティション分割することができます。1 つ以上の外部ロードバランサを使用して、すべてのアプリケーションサーバーインスタンス間でユーザーアクセスを効率的に分散させることができます。水平型のスケーリングアプローチでは可用性の向上やハードウェアコストの低減が可能です。このアプローチは、一部の種類のアプリケーションに適しています。ただし、このアプローチでは、より多数のアプリケーションサーバーインスタンスやハードウェアノードの管理が必要になります。

管理

1 台のサーバーへの Application Server の単一インストール内で、複数のインスタンスを動作させることができます。単一の Application Server によって管理される 1 つ以上のインスタンスのグループを「ドメイン」と呼びます。サーバーインスタンスをドメインにグループ化することにより、役割の異なる人々がそれぞれ独立した形でグループを管理できます。

単一インスタンスドメインを使用して、特定の開発者および環境向けの「サンドボックス」を作成できます。このシナリオでは、個々の開発者はほかのアプリケーションサーバードメインに干渉することなく、自分自身のアプリケーションサーバーを管理します。小規模の開発グループでは、共同作業による開発を目的として、共有管理ドメイン内に複数のインスタンスを作成することを選択できます。

配備環境では、開発者はアプリケーションやビジネスの機能に基づいてドメインを作成できます。たとえば、社内の人事管理アプリケーションを 1 つの管理ドメイン内の 1 台または複数台のサーバーで運用し、外部の顧客アプリケーションをサーバーファーム内の複数の管理ドメインで運用することができます。

Application Server は、Web アプリケーション用の仮想サーバー機能をサポートします。たとえば、効率的な管理のために、サービスプロバイダのホストである Web アプリケーションが、単一の Application Server プロセス上で複数の異なる URL ドメインをホストできます。

管理の詳細は、『Sun Java System Application Server Enterprise Edition 8.2 管理ガイド』を参照してください。

一般的なチューニングの概念

パフォーマンスチューニングに影響する重要な概念には、次のものがあります。

- ユーザー負荷
- アプリケーションのスケーラビリティー
- 安全性のマージン

次の表で、これらの概念と実際の測定方法を説明します。左端の列では一般的な概念を示し、2 番目の列にはその概念を実行した場合の結果を示します。3 番目の列では測定値、右端の列では値のソースを説明します。

表 1–2 パフォーマンスに影響する要因

概念	実行した場合の結果	測定値	値のソース
ユーザー負荷	ピーク負荷時の同時セッション数	毎分トランザクション数 (TPM) 毎秒 Web インタラクション数 (WIPS)	(最大同時ユーザー数) * (予測応答時間) / (クリック間隔) 例: (100 ユーザー * 2 秒) / 10 秒 = 20
アプリケーションのスケーラビリティー	1 個の CPU 上で測定されるトランザクション率	TPM または WIPS	負荷ベンチマークから測定されます。各層で実行します。
垂直方向のスケーラビリティー	CPU の追加によるパフォーマンス上昇	追加 CPU あたりの上昇率	ベンチマークからの曲線の当てはめに基づく CPU 数を段階的に増やしながらテストを実行します。曲線の「転換点」を識別します。これは、CPU の追加によるパフォーマンス上昇効果がコストに見合わなくなるポイントです。このガイドの説明に従ったチューニングが必要です。各層で実行し、必要であれば反復します。この測定値がパフォーマンス要件を満たしていればここで中止します。
水平方向のスケーラビリティー	サーバー の追加によるパフォーマンス上昇	サーバープロセスやハードウェアノードの追加あたりの上昇率	前のステップと同様に、十分に調整された単一のアプリケーションサーバーインスタンスを使用します。サーバーインスタンスおよびハードウェアノードを 1 つ追加するたびにパフォーマンスがどれだけ改善されるかを測定します。
安全性マージン	高可用性要件	システムが障害に対処する必要がある場合、1 つ以上のアプリケーションサーバーインスタンスの機能停止を想定したパフォーマンス要件を満たすようにシステムのサイズを決定する	高可用性が必要な場合は、使用する等式が異なります。

表 1–2 パフォーマンスに影響する要因 *(続き)*

概念	実行した場合の結果	測定値	値のソース
	想定を超えたピーク時の超過容量	ある程度の安全マージンを確保するために、ベンチマークで測定されたピークよりも低い負荷でサーバーを運用することが望ましい	ピーク負荷の 80% のシステム容量利用は、ほとんどのインストールで適切です。現実のピーク負荷およびシミュレートされたピーク負荷の各条件下で配備を測定します。

容量計画

これまでの説明では、配備アーキテクチャーの定義に向けた指針を示しました。ただし、配備の実際のサイズは、「容量計画」と呼ばれるプロセスによって決定します。容量計画では、次のことを予測します。

- 特定のハードウェア構成のパフォーマンス容量
- 指定されたアプリケーション負荷およびパフォーマンスを持続するために必要なハードウェアリソース

これらの値を予測するには、現実的なデータセットおよび作業負荷を用意したアプリケーションを使用して、入念なパフォーマンスベンチマーク測定を行います。

▼ 容量を決定する

1 1 個の CPU 上でのパフォーマンスを特定する。

まず、1 個のプロセッサで持続できる最大の負荷を特定します。この数値は、シングルプロセッサマシン上でアプリケーションのパフォーマンスを測定することによって入手できます。類似の処理特性を持つ既存のアプリケーションのパフォーマンス数値を利用するか、より理想的な方法として、テスト環境で実際のアプリケーションおよび作業負荷を使用します。この測定で使用するアプリケーション層およびデータソース層の構成は、最終的な配備と完全に一致するようにしてください。

2 垂直方向のスケーラビリティーを特定する。

プロセッサを追加したときにパフォーマンスがどの程度上昇するかを特定します。これは、特定の作業負荷がかかったときにサーバー上で発生する共有リソース競合の量を間接的に測定することになります。この情報は、マルチプロセッサシステム上でのアプリケーションの追加負荷テストに基づいて入手するか、あるいは、すでに負荷テストが完了している類似のアプリケーションから入手した既存の情報を利用します。

一般的には、1〜8 個の CPU 構成で一連のパフォーマンステストを段階的に実行することにより、システムの垂直方向スケーラビリティー特性の感覚が得られます。結

果を歪曲することがないように、必ず、アプリケーション、Application Server、バックエンドのデータベースリソース、およびオペレーティングシステムを適切に調整してください。

3 水平方向のスケーラビリティーを特定する。

十分に強力なハードウェアリソースを利用できる場合、単一のハードウェアノードでパフォーマンス要件を満たせる場合があります。それでも、2 台以上のシステムをクラスタ化することで、さらなる可用性の向上を実現できます。外部のロードバランサおよび作業負荷シミュレーションを利用して、ステップ (2) で特定したように、十分に調整された 1 つのアプリケーションサーバーノードをレプリケートすることによって得られるパフォーマンス向上を特定します。

ユーザーの期待

アプリケーションのエンドユーザーは一般に、パフォーマンスに関して何らかの期待を抱いています。多くの場合、そのような期待は数値的に定量化できます。顧客のニーズが満たされることを保証するために、これらの期待を明確に理解し、容量計画に反映する必要があります。

パフォーマンスに関する期待について、次の事項を検討します。

- アプリケーションとの各種の対話処理で、ユーザーが期待する平均応答時間はどの程度か。もっとも頻繁に行われる対話処理は何か。時間的要求がきわめて厳しい対話処理が存在するか。1 回の対話処理の長さは思考時間を含めてどの程度か。多くのケースで、精度の高い予測を得るためには、観察に基づいたユーザー調査の実施が必要です。
- 通常状態時およびピーク時のユーザー負荷はどの程度と予測されるか。一日、一週間、または一年のうち、負荷のピークが観測または予測される特定の時間帯または時期が存在するか、オンラインビジネスでは数百万の登録ユーザーが存在する場合もありますが、ある特定の時点で実際にログインしてビジネストランザクションを実行しているのはそのうちのごく一部です。容量計画の間の一般的な誤りの 1 つは、同時ユーザー数の平均値およびピーク値ではなく、顧客基盤の全体サイズを基準として使用することです。時間の経過とともに、同時ユーザー数に何らかのパターンが現れる場合もあります。
- 1 回の要求で転送される平均およびピークのデータ容量はどの程度か。この値はアプリケーション固有でもあります。コンテンツサイズの適切な予測とその他の使用状況パターンを組み合わせることで、ネットワーク容量のニーズを予測しやすくなります。
- 今後 1 年間で予測されるユーザー負荷の増加はどの程度か。将来を十分に見据えた計画を立てることにより、危機的な状況や、アップグレードのためのシステム停止期間を避けやすくなります。

詳細情報

- Java パフォーマンスの詳細は、「Java Performance Documentation」および「Java Performance BluePrints」を参照してください。
- EJBコンポーネントの最適化の詳細は、「Seven Rules for Optimizing Entity Beans」を参照してください。
- プロファイリングの詳細は、『Sun Java System Application Server Enterprise Edition 8.2 Developer's Guide』の「Profiling Tools」を参照してください。
- SNMP 監視の詳細は、『Sun Java System Application Server Enterprise Edition 8.2 管理ガイド』の第 16 章「コンポーネントとサービスの監視」を参照してください。
- `domain.xml` ファイルの詳細は、『Sun Java System Application Server Enterprise Edition 8.2 Administration Reference 』を参照してください。

◆ ◆ ◆ 第 2 章

アプリケーションのチューニング

この章では、最大のパフォーマンスを得るためのアプリケーションのチューニングに関する情報を提供します。パフォーマンスの高い Java および J2EE アプリケーションを記述するための完全なガイドを提供することは、この章の目的ではありません。

次の内容について説明します。

- 29 ページの「Java プログラミング上の指針」
- 32 ページの「JSP とサーブレットのチューニング」
- 34 ページの「EJB のパフォーマンスチューニング」

Java プログラミング上の指針

この節では、Java のコーディングおよびパフォーマンスに関連する問題を扱います。これらの指針は Application Server に固有のものではなく、多くの状況で役立つ一般則です。Java コーディングのベストプラクティスについての詳細な解説は、「Java Blueprints」を参照してください。

直列化と直列化復元を避ける

オブジェクトの直列化と直列化復元は、CPU 負荷の高い操作であり、アプリケーションの動作速度を低下させる可能性があります。`transient` キーワードを使用して、直列化されるデータの量を減らします。また、一部のクラスでは、カスタマイズした `readObject()` および `writeObject()` メソッドが役に立つ場合があります。

文字列の連結に StringBuffer を使用する

パフォーマンスを改善するには、文字列連結を使用する代わりに `StringBuffer.append()` を使用します。

文字列オブジェクトは不変であり、作成後に変化することはありません。以下のコードを例に説明します。

```
String str = "testing";
str = str + "abc";
```

コンパイラはこのコードを次のように変換します。

```
String str = "testing";
StringBuffer tmp = new StringBuffer(str);
tmp.append("abc");
str = tmp.toString();
```

この例が示すように、コピーは本質的に負荷が大きい処理であり、使用しすぎるとパフォーマンスを著しく低下させる可能性があります。

不要になった変数に NULL を代入する

不要になった変数に明示的に NULL を代入すると、安全に再生できるメモリー上の部分をガベージコレクタが識別しやすくなります。Java はメモリー管理機構を備えていますが、メモリーリークや過度のメモリー使用を防ぐわけではありません。

オブジェクト参照を解放しないことにより、アプリケーションがメモリーリークを誘発する場合があります。それにより、Java ガベージコレクタがそれらのオブジェクトを再生できなくなり、結果的に使用中のメモリーの量が増加します。使用後に変数への参照を明示的に null にすることにより、ガベージコレクタがメモリーを再生できるようになります。

メモリーリークを検出する 1 つの方法は、プロファイリングツールを利用し、毎回のトランザクション後にメモリーのスナップショットを取ることです。安定した状態でリークのないアプリケーションは、ガベージコレクション後に安定したアクティブヒープメモリーを示します。

必要な場合にのみメソッドを final として宣言する

最近の最適化動的コンパイラでは、Java メソッドが `final` と宣言されていない場合でも、インライン化やその他の手続き間最適化を実行できます。キーワード `final` は元々の意図どおりに、すなわち、プログラムアーキテクチャー上の理由および保守性のために使用します。

`final` キーワードは、メソッドをオーバーライドしてはならないという確証がある場合にのみ使用してください。

定数を static final として宣言する

定数を `static final` 変数として宣言すると、動的コンパイラは定数畳み込み最適化を容易に実行できます。

ファイナライザを避ける

コードにファイナライザを追加すると、ガベージコレクタの負荷が大きくなり、動作が予測不能になります。仮想マシンは、ファイナライザが実行されるタイミングを保証しません。プログラムが終了する前にファイナライザが必ず実行されるとは限りません。クリティカルなリソースを `finalize()` メソッドで解放すると、アプリケーションの予測不能な動作を引き起こす場合があります。

メソッド引数を **final** として宣言する

メソッド引数がメソッド内で変更されない場合、その変数を `final` として宣言します。一般に、すべての変数について、初期化または何らかの値に設定されたあとに変数が変更されない場合はその変数を `final` として宣言します。

必要なときにのみ同期を行う

同期が必要な場合を除き、コードブロックまたはメソッドを同期しないでください。スケーラビリティーのボトルネックを避けるために、同期されるブロックまたはメソッドはできるだけ短くします。同期されないデータ構造については、より負荷が大きい `java.util.HashTable` などの別の方法の代わりに Java Collections Framework を使用します。

SOAP 添付には **DataHandler** を使用する

SOAP 添付に `javax.activation.DataHandler` を使用すると、パフォーマンスが向上します。

JAX-RPC 仕様では、次のことが規定されています。

- 特定の MIME タイプから Java の型へのマッピング。
- 任意の MIME タイプが `javax.activation.DataHandler` にマップ可能であること。

結果として、Java の型マッピングを利用することによって、添付ファイル (`.gif` または XML ドキュメント) を SOAP 添付ファイルとして RPC スタイルの Web サービスに送信します。Web サービスの引数として (添付ファイルの MIME タイプにとって適切な) 必須の Java 型マッピングのいずれかを渡すとき、JAX-RPC ランタイムはこれらを SOAP 添付ファイルとして処理します。

たとえば、`image/gif` 添付ファイルを送信するには、`java.awt.Image` を使用するか、画像の DataHandler ラッパーを作成します。ラッパーを使用することの利点は次のとおりです。

- コーディングの削減: 添付ファイルに含まれるデータのコンテンツタイプが DataHandler によって自動的に特定されるため、汎用的な添付コードを再利用して添付ファイルを処理できます。 この機能は、ドキュメント形式のサービスを使用しているときに特に役立ちます。実行の時点でコンテンツが判明しているため、`attachment.setContent(stringContent, "image/gif")` などの呼び出しを行う必要がありません。

- パフォーマンスの向上: DataHandler ラッパーを使用すると、`image/gif` MIME タイプについてはスループットが約 2 倍に、`text/xml` タイプまたは `image/*` タイプの `java.awt.Image` についてはスループットが約 1.5 倍になることが非公式のテストによって示されています。

JSP とサーブレットのチューニング

Application Server 上で動作する多くのアプリケーションは、プレゼンテーション層でサーブレットまたは JavaServer Pages (JSP) 技術を使用します。この節では、コーディング手法、配備設定、およびその他の設定を通じて、そのようなアプリケーションのパフォーマンスを改善する方法について説明します。

コーディング手法の提案

この節では、サーブレットおよび JSP アプリケーションのパフォーマンスを改善するコーディング手法についてのヒントを示します。

一般的な指針

プレゼンテーション層のパフォーマンスを高めるには、次の一般的な指針に従います。

- サーブレット内での Java 同期を最小限にする。
- サーブレットでシングルスレッドモデルを使用しない。
- 負荷が大きい 1 回限りの初期化にはサーブレットの `init()` メソッドを使用する。
- `System.out.println()` 呼び出しの使用を避ける。

共有変更済みクラス変数を避ける

サーブレットのマルチスレッドモデル (デフォルト) では、個々のアプリケーションサーバーインスタンスに対して単一のサーブレットインスタンスが作成されます。そのアプリケーションインスタンス上のサーブレットに対するすべての要求は、同じサーブレットインスタンスを共有します。このことが原因で、サーブレットコードに同期ブロックが存在する場合にスレッド競合が発生する可能性があります。したがって、共有変更済みクラス変数は同期の必要性を作り出すため、使用を避けます。

HTTP セッション処理

HTTP セッションを使用するときは、次の指針に従います。

- セッションを作成しすぎないようにします。セッション作成には一定の負荷が伴います。不要な場合にはセッションを作成しないでください。

- セッションが不要になった時点で、`javax.servlet.http.HttpSession.invalidate()` を使用してセッションを解放します。
- 応答時間を短縮するために、セッションのサイズを小さく保ちます。可能であれば、セッションサイズを 7K バイト未満に保ちます。
- JSP ファイルで `<%page session="false"%>` 指令を使用して、セッションが必要でないときに Application Server が自動的にセッションを作成することを避ける。
- `HttpSession` で大きなオブジェクトグラフを避けます。大きなオブジェクトグラフは直列化を強制し、計算のオーバーヘッドを増加させます。一般則として、サイズの大きいオブジェクトを `HttpSession` 変数として格納しないでください。
- `HttpSession` にトランザクションデータをキャッシュしないでください。`HttpSession` 内のデータへのアクセスはトランザクション方式ではありません。このオブジェクトをトランザクションデータのキャッシュとして使用しないでください。トランザクションデータはデータベースに保持し、エンティティー Beans を使用してアクセスするほうが適切です。トランザクションは失敗時に元の状態にロールバックされます。ただし、古くなっていて正しくないデータが `HttpSession` オブジェクトに残る場合があります。Application Server では、読み取り専用データへのキャッシュを使用したアクセスのために、「読み取り専用」の Bean 管理による持続性エンティティー Beans が用意されています。

設定と配備のヒント

パフォーマンスを改善するには、設定に関する次のヒントに従います。これらのヒントは、開発環境ではなく本稼働環境を想定したものです。

- クラスのロード時間を短縮するには、サーバーの `CLASSPATH` に入れるディレクトリ数が多くなりすぎないようにします。アプリケーション関連クラスを JAR ファイルに置きます。
- 一般に、HTTP の応答時間は、キープアライブサブシステムおよび HTTP サーバーをどのように調整するかによって変化します。詳細は、66 ページの「HTTP サービスの設定」を参照してください。
- Solaris 8 を使用している場合、ヒープ空間への同時アクセスのための `malloc` ルーチンのコレクションを提供する `mtmalloc` ライブラリを使用することによって SSL を最適化します。`mtmalloc` を使用するには、次の手順に従います。
  - SunSolve Online から、パッチ 111308-03 を入手してインストールします。
  - ドメインの `bin/startserv` ディレクトリにある `startserv` スクリプトを編集し、LD_PRELOAD 環境変数の値を `/usr/lib/libmtmalloc.so` と定義します。

    環境変数を定義するための正確な構文は、使用するシェルによって異なります。
- 可能な場合はサーブレットの結果をキャッシュします。詳細は、『Sun Java System Application Server Enterprise Edition 8.2 Developer's Guide』の第 5 章「Developing Web Applications」を参照してください。

- アプリケーションに EJB コンポーネントが含まれない場合、アプリケーションを EAR ファイルでなく WAR ファイルとして配備します。

セキュリティーマネージャーの無効化

セキュリティーマネージャーは負荷が大きいアプリケーションです。これは、必要なリソースへの呼び出しで `doPrivileged()` メソッドを呼び出す必要があることに加えて、`server.policy` ファイルを使用してリソースをチェックする必要もあるためです。サーバー上で悪意のあるコードが実行されないという確証があり、アプリケーションの内部で認証を使用しない場合は、セキュリティーマネージャーを無効にすることができます。

`server.policy` ファイルの使用を無効にするには、管理コンソールを使用します。「設定」、「<設定名>」、「JVM 設定」(「JVM オプション」) の順に選択し、次のようなオプションを削除します。

```
-Djava.security.policy=${com.sun.aas.instanceRoot}/config/server.policy
```

EJB のパフォーマンスチューニング

Application Server の高性能な EJB コンテナには、パフォーマンスに影響を及ぼす多数のパラメータがあります。個別の EJB コンポーネントにも、パフォーマンスに影響を及ぼすパラメータがあります。個別 EJB コンポーネントのパラメータの値は、EJB コンテナの同じパラメータの値に優先します。デフォルト値はシングルプロセッサのコンピュータシステム用に設計されています。それ以外のシステム構成に合わせた最適化を行うには、パラメータを変更します。

この節の内容は、次のとおりです。

- 34 ページの「目標」
- 35 ページの「EJB コンポーネントの監視」
- 37 ページの「一般的な指針」
- 39 ページの「ローカルインタフェースとリモートインタフェースの使用」
- 41 ページの「EJB トランザクションのパフォーマンスの改善」
- 43 ページの「特殊な手法の使用」
- 46 ページの「特定タイプの EJB コンポーネントのチューニングのヒント」
- 50 ページの「JDBC とデータベースアクセス」
- 51 ページの「メッセージ駆動型 Beans のチューニング」

目標

EJB パフォーマンスチューニングの目標は次のとおりです。

- 速度の向上 - EJB キャッシュにできるだけ多くの Beans をキャッシュし、速度を向上させます (応答時間を短縮する)。キャッシュ処理により、CPU にかかる負荷の大きい操作が除去されます。ただし、メモリーは有限であるため、キャッシュサイズが増大するにつれて (ガベージコレクションを含む) キャッシュの管理に必要な時間も増大します。
- メモリー消費の低減 - プールまたはキャッシュ内の Beans は、Java 仮想マシンヒープからのメモリーを消費します。プールやキャッシュが肥大しすぎると、ガベージコレクションサイクルの時間が長くなるとともに頻度も高くなるため、パフォーマンスが低下します。
- 機能特性の改善 - ユーザーのタイムアウト、コミットオプション、セキュリティー、トランザクションオプションなどの機能特性は、ほとんどがアプリケーションの機能性および設定に関連します。一般に、これらの特性によってパフォーマンス関連の機能性が損なわれることはありません。場合によっては、機能性とパフォーマンスの間で「トレードオフ」の決定を下す必要が生じることがあります。この節では、そのような場合における提案を示します。

EJB コンポーネントの監視

EJB コンテナの監視を有効にした場合、Bean のプール設定とキャッシュ設定に基づいて、個別 Beans の統計を検証できます。

たとえば、次の監視コマンドにより、ステートフルセッション Bean の「Bean キャッシュ」を取得できます。

```
asadmin get --user admin --host e4800-241-a --port 4848
-m specjcmp.application.SPECjAppServer.ejb-module.
   supplier_jar.stateful-session-bean.BuyerSes.bean-cache.*
```

監視出力のサンプルを次に示します。

```
resize-quantity = -1
 cache-misses = 0
 idle-timeout-in-seconds = 0
 num-passivations = 0
 cache-hits = 59
 num-passivation-errors = 0
 total-beans-in-cache = 59
 num-expired-sessions-removed = 0
 max-beans-in-cache = 4096
 num-passivation-success = 0
```

次の監視コマンドは、エンティティー Bean の Bean プール統計を提供します。

```
asadmin get --user admin --host e4800-241-a --port 4848
-m specjcmp.application.SPECjAppServer.ejb-module.
```

```
   supplier_jar.stateful-entity-bean.ItemEnt.bean-pool.*
idle-timeout-in-seconds = 0
steady-pool-size = 0
total-beans-destroyed = 0
num-threads-waiting = 0
num-beans-in-pool = 54
max-pool-size = 2147483647
pool-resize-quantity = 0
total-beans-created = 255
```

次の監視コマンドは、ステートレス Bean の Bean プール統計を提供します。

```
asadmin get --user admin --host e4800-241-a --port 4848
-m test.application.testEjbMon.ejb-module.slsb.stateless-session-bean.slsb.bean-pool.*
idle-timeout-in-seconds = 200
steady-pool-size = 32
total-beans-destroyed = 12
num-threads-waiting = 0
num-beans-in-pool = 4
max-pool-size = 1024
pool-resize-quantity = 12
total-beans-created = 42
```

Bean のチューニングでは、一定の期間を区切って、対象となる Bean のキャッシュおよびプールの動作をグラフ化します。

あまりに多くのパッシベーションが発生しており、JVM ヒープは比較的小さいままである場合は、`max-cache-size` または `cache-idle-timeout-in-seconds` の値を大きくすることができます。ガベージコレクションがあまりにも頻繁に発生しており、プールサイズが増加しているが、キャッシュヒット率が低い場合、`pool-idle-timeout-in-seconds` の値を小さくしてインスタンスを破棄することができます。

注 – `max-pool-size` にゼロ (0) を指定すると、プールがアンバインドされます。プールされた Beans は、`pool-idle-timeout-in-seconds` に短い間隔を指定することによって削除される場合を除き、メモリー内に残ります。本番システムについては、アンバインドされたものとしてプールを指定することは推奨されません。

個別 EJB コンポーネントの監視

Bean 内のすべてのメソッドについてメソッド呼び出し統計を収集するには、次のコマンドを使用します。

```
asadmin get -m monitorableObject.*
```

monitorableObject は、次に示すような監視可能オブジェクトの階層からの完全指定 ID です。

serverInstance.application.*applicationName*.ejb-module.*moduleName*

moduleName は、モジュール x.jar に対しては x_jar です。

- .stateless-session-bean.beanName .bean-pool .bean-method.methodName
- .stateful-session-bean.beanName .bean-cache .bean-method.methodName
- .entity-bean.beanName .bean-cache .bean-pool .bean-method.methodName
- .message-driven-bean.beanName .bean-pool .bean-method.methodName (methodName = onMessage)

スタンドアロン Beans については、次のパターンを使用します。

serverInstance.application.*applicationName*.standalone-ejb-module.*moduleName*

指定可能な ID は、ejb-module と同じです。

たとえば、エンティティー Bean 内のメソッドについての統計を得るには、次のコマンドを使用します。

```
asadmin get -m serverInstance.application.appName.ejb-module.moduleName
.entity-bean.beanName.bean-method.methodName.*
```

監視可能なオブジェクト (アプリケーション、モジュール、Beans、およびメソッド) およびオブジェクト属性を検索するには、管理コンソールを使用します。詳細は、『Sun Java System Application Server Enterprise Edition 8.2 管理ガイド』の第 16 章「コンポーネントとサービスの監視」を参照してください。あるいは、asadmin list コマンドを使用します。詳細は、list(1) を参照してください。

ステートフルセッション Bean のパッシベーションについての統計を得るには、次のコマンドを使用します。

```
asadmin get -m serverInstance.application.appName.ejb-module.moduleName
.stateful-session-bean.beanName.bean-cache.*
```

返された属性値から、次のコマンドを使用します。

```
num-passivationsnum-passivation-errorsnum-passivation-success
```

一般的な指針

次の指針に従うことによって、EJB コンポーネントのパフォーマンスを改善できます。アプリケーションを多数の EJB コンポーネントに分解すると、オーバーヘッドが発生し、パフォーマンスが低下する可能性もあることに留意してください。EJB コンポーネントは単なる Java オブジェクトではありません。EJB コンポーネントは、プロパティーとメソッドに加えて、リモート呼び出しインタフェース、セキュリティー、およびトランザクションのセマンティクスを持つコンポーネントです。

パフォーマンスの高い Beans を使用する

アプリケーションの全体的なパフォーマンスを改善するには、できるだけパフォーマンスの高い Beans を使用します。詳細は、46 ページの「特定タイプの EJB コンポーネントのチューニングのヒント」を参照してください。

EJB コンポーネントのタイプの一覧を、もっともパフォーマンスの高いものから順に示すと次のようになります。

1. ステートレスセッション Beans およびメッセージ駆動型 Beans
2. ステートフルセッション Beans
3. 読み取り専用として設定されたコンテナ管理による持続性 (CMP) エンティティー Beans
4. 読み取り専用として設定された Bean 管理による持続性 (BMP) エンティティー Beans
5. CMP Beans
6. BMP Beans

キャッシュの使用

キャッシュを適切に使用することで、パフォーマンスを大きく改善できます。次に例を示します。

- **EJB** 参照のキャッシュ: 要求ごとに JNDI 検索が行われるのを避けるには、EJB 参照をサーブレットにキャッシュします。
- ホームインタフェースのキャッシュ: ホームインタフェースを繰り返し検索すると負荷が大きくなる可能性があるため、EJBHome への参照をサーブレットの `init()` メソッドにキャッシュします。
- **EJB** リソースのキャッシュ: `setSessionContext()` または `ejbCreate()` を使用して Bean リソースをキャッシュします。これは再び、可能な場合には Bean のライフサイクルメソッドを使用して、アプリケーションアクションを一度だけ実行する例です。取得したリソースを `ejbRemove()` メソッドで解放することを忘れないでください。

アプリケーションスタブの使用

EJB アプリケーションで必要なスタブクラスは、実行時に EJB クライアントで必要になったときに動的に生成されます。これは、リモート EJB コンポーネントを使用してアプリケーションを配備するときに、スタブを生成したりクライアントの JAR ファイルを取得したりする必要がないことを意味します。`--retrieve` オプションは配備を高速化しますが、アプリケーションを配備するときにこのオプションを指定する必要はなくなりました。

CosNaming サービスを直接使用する (これは推奨される構成ではない) 従来のリッチクライアントアプリケーションがある場合、RMIC を使用して、アプリケーションに

対してスタブを明示的に生成する必要があります。詳細は、『Sun Java System Application Server Enterprise Edition 8.2 トラブルシューティングガイド』を参照してください。

不必要なステートフルセッション Beans の削除

不要なステートフルセッション Bean を削除することにより、ディスク操作が必要な、その Beans のパッシベーションを回避できます。

キャッシュとプールのチューニングのヒント

EJB キャッシュとプールを使用してパフォーマンスを改善するには、次のヒントに従います。

- `remove()` を明示的に呼び出す: クライアントで `remove()` メソッドを明示的に呼び出すことにより、ステートフルセッション EJB コンポーネントをコンテナキャッシュから削除できます。
- エンティティー EJB コンポーネントのプールサイズを調整する: エンティティー Beans は、EJB プールとキャッシュ設定の両方を使用します。エンティティー EJB コンポーネントのプールサイズを調整して、Beans の作成と破棄を最小限にします。ゼロでない一定のサイズで事前にプールを埋めることは、初期要求に対する応答を向上させるために役立ちます。
- キャッシュ Bean 固有リソース: Bean 固有リソースをキャッシュするには、`setEntityContext()` メソッドを使用します。キャッシュしたリソースを解放するには、`unSetEntityContext()` メソッドを使用します。
- コンテナ管理による関係について、関連データを効率的にロードします。詳細は、48 ページの「コンテナ管理による関係 (CMR) Beans の先取り」を参照してください。
- 読み取り専用 Beans を識別する: 読み取り専用の操作に対しては、読み取り専用エンティティー Beans を設定します。詳細は、47 ページの「読み取り専用エンティティー Beans」を参照してください。

ローカルインタフェースとリモートインタフェースの使用

この節では、ローカルおよびリモートのクライアントによって EJB コンポーネントが使用されるときの考慮事項について説明します。

ローカルインタフェースを優先する

EJB コンポーネントは、リモートインタフェースとローカルインタフェースを持つことができます。Bean と同じアプリケーションサーバーインスタンスに配置されていないクライアント (リモートクライアント) は、リモートインタフェースを使用して

Bean にアクセスします。リモートインタフェースの呼び出しには、引数を整列化し、整列化したデータをネットワーク経由で転送し、引数を非整列化し、受信側の端で振り分ける一連の処理が必要です。そのため、リモートインタフェースの使用には大きなオーバーヘッドが伴います。

EJB コンポーネントがローカルインタフェースを持つ場合、同じアプリケーションサーバーインスタンス内のローカルクライアントは、そのインタフェースをリモートインタフェースの代わりに使用できます。引数の整列化、転送、および非整列化が不要になるため、ローカルインタフェースを使用するほうが効率的です。

Bean がローカルクライアントのみによって使用される予定の場合、ローカルインタフェースのみを提供する意味があります。一方、Bean を位置独立にする予定の場合は、リモートインタフェースとクライアントインタフェースの両方を提供することをお勧めします。これにより、リモートクライアントはリモートインタフェースを使用し、ローカルクライアントは効率性のためにローカルインタフェースを使用できるようになります。

参照渡しのセマンティクスの使用

Application Server のデフォルトでは、Bean のリモートインタフェースの呼び出しには、両者が同じ位置にある場合でも「値渡し」のセマンティクスが使用されます。値渡しのセマンティクスを使用するクライアントは、EJB コンポーネントに引数を渡す前に引数をコピーする必要があるため、これは負荷の大きい処理になる可能性があります。

ただし、ローカルクライアントは「参照渡し」セマンティクスを使用でき、その場合、渡されるオブジェクトをローカルインタフェースとリモートインタフェースが共有できます。しかしこれは、引数オブジェクトを共有可能にするためには、引数オブジェクトを適切に実装する必要があることを意味します。一般的には、可能な場合には参照渡しのセマンティクスを使用するほうが効率的です。

リモートインタフェースとローカルインタフェースを適切に使用することで、クライアントは EJB コンポーネントに効率的にアクセスできます。ローカルクライアントは参照渡しセマンティクスでローカルインタフェースを使用する一方で、リモートクライアントは値渡しセマンティクスでリモートインタフェースを使用します。

ただし、一部のインスタンスでは、次の例のような場合にローカルインタフェースを使用できない可能性があります。

- アプリケーションが EJB 2.0 仕様よりも前のものであり、ローカルインタフェースを一切使用せずに記述されている。
- Bean 間呼び出しが存在し、クライアント Beans の記述で、呼び出される Beans が同じ位置に存在するという可能性が一切想定されていない。

このようなケースのために、Application Server では、同じ位置に存在する EJB コンポーネントのリモートインタフェースに参照によって引数を渡すためにクライアントが使用できる参照渡しオプションが用意されています。

参照渡しオプションは、アプリケーション全体または単一の EJB コンポーネントを対象に指定できます。アプリケーションレベルで指定された場合、アプリケーション内のすべての Beans は、そのリモートインタフェースに引数を渡すときに参照渡しセマンティクスを使用します。Bean レベルで指定された場合、Bean のリモートインタフェースへのすべての呼び出しは参照渡しセマンティクスを使用します。参照渡しフラグの詳細は、『Sun Java System Application Server Enterprise Edition 8.2 Developer's Guide』の「Value Added Features」を参照してください。

EJB コンポーネントが参照渡しセマンティクスを使用することを指定するには、配備記述子 `sun-ejb-jar.xml` で次のタグを使用します。

```
<pass-by-reference>true</pass-by-reference>.
```

これにより、EJB コンポーネントのメソッドが呼び出されるときの引数のコピーと、メソッドが戻るときの結果のコピーが回避されます。ただし、呼び出しの間に別のソースによってデータが変更されると問題が発生します。

EJB トランザクションのパフォーマンスの改善

この節では、トランザクションを使用するときのパフォーマンスを改善するためのヒントを示します。

コンテナ管理によるトランザクションを使用する

コンテナ管理によるトランザクションは、整合性の面で望ましく、パフォーマンスにも優れています。

ユーザー入力時間を含めない

リソースが不必要に長期間保持されることを避けるために、トランザクションにはユーザーの入力または思考時間を含めないことをお勧めします。

非トランザクションメソッドを識別する

セッション EJB コンポーネントの非トランザクションメソッドを、トランザクション属性 `NotSupported` または `Never` を使用して宣言します。これらの属性は、配備記述子ファイル `ejb-jar.xml` にあります。トランザクションはデータベース行をロックするため、持続時間をできるだけ短くします。

長いトランザクション連鎖に TX_REQUIRED を使用する

非常に長いトランザクション連鎖については、トランザクション属性 `TX_REQUIRED` を使用します。呼び出し連鎖で EJB メソッドを保証するには、同じトランザクションを使用します。

もっともコストの低いデータベースロックを使用する

任意のトランザクションと整合性のあるデータベースから使用可能なロックのうち、もっともコストが低いものを使用します。各メソッド呼び出しのあとでなく、トランザクションが完了したあとにデータをコミットします。

XA 対応データソースを必要なときにのみ使用する

複数のデータベースリソース、コネクタリソース、または JMS リソースが 1 つのトランザクションに関係するとき、分散トランザクションまたはグローバルトランザクションを実行する必要があります。これには、XA 対応のリソースマネージャーおよびデータソースが必要です。XA 対応のデータソースは、2 つ以上のデータソースがトランザクションに関係する場合にのみ使用します。データベースがいくつかの分散トランザクションに参加するが、ほとんどの参加先がローカルトランザクションまたは単一データベーストランザクションである場合、2 つの独立した JDBC リソースを登録し、アプリケーション内で適切なリソースを使用することをお勧めします。

JDBC リソースを 1 フェーズコミットリソースとして設定する

複数のリソースが関係しているトランザクションのパフォーマンスを改善するために、Application Server では最終エージェント最適化 (LAO) を使用します。これにより、分散トランザクション内のリソースのうちの 1 つを、1 フェーズコミット (1PC) リソースとして設定することが可能になります。JDBC リソースに関しては、複数リソーストランザクションのオーバーヘッドのほうがメッセージキューよりもずっと大きいため、LAO により、1 つの JDBC リソースと 1 つ以上のメッセージキューが関係する分散トランザクションのパフォーマンスが大きく改善されます。LAO を利用するには、JDBC リソースを 1PC リソースとして設定します。JMS リソースを設定するために、特別なことを行う必要はありません。

複数の JDBC リソースが関係するグローバルトランザクションでも LAO によってパフォーマンスが改善されますが、改善の度合いは 1 つの JDBC リソースの場合には及びません。この状況では、JDBC リソースのうち 1 つを 1 PC として設定し、その他すべてを XA として設定することをお勧めします。

もっとも負荷の小さいトランザクション属性を使用する

EJB 配備記述子ファイル (`ejb-jar.xml`) で、次のトランザクション属性を設定します。オプションの一覧は、もっともパフォーマンスの高いものから低いものの順に並んでいます。パフォーマンスを改善するには、アプリケーションに必要な機能性を提供する属性の中でもっとも負荷の小さいものを選択します。

1. NEVER
2. TX_NOTSUPPORTED
3. TX_MANDATORY
4. TX_SUPPORTS

5. TX_REQUIRED
6. TX_REQUIRESNEW

特殊な手法の使用

この節では、特殊なパフォーマンス強化手法について説明します。ここで紹介する手法は、バージョン整合性と要求のパーティション分割の 2 つです。

バージョン整合性

データベース内のデータの完全性を保護しながらパフォーマンスを改善するには、「バージョン整合性」を使用します。アプリケーションサーバーは EJB コンポーネントの複数のコピーを同時に使用できるので、同時アクセスを通じて EJB コンポーネントが破壊された状態になる可能性があります。

破壊を防ぐ標準的な方法は、特定の Bean に関連付けられたデータベース行をロックすることです。これは、2 つの同時トランザクションによって Bean がアクセスされることを防ぎ、結果的にデータを保護します。ただし、この方法は実質的にすべての EJB アクセスを直列化するため、パフォーマンスも低下させます。

バージョン整合性は、EJB データの完全性を保護するためのもう 1 つのアプローチです。バージョン整合性を使用するには、バージョン番号として使用するデータベース内の列を指定します。その場合、EJB ライフサイクルの進行は次のようになります。

- 最初に Bean が使用されるとき、`ejbLoad()` メソッドが Bean を通常どおりにロードします。これには、データベースからのバージョン番号のロードが含まれます。
- `ejbStore()` メソッドが、データベース内のバージョン番号を、EJB コンポーネントがロードされたときの番号の値と比較してチェックします。
  - バージョン番号が変更されている場合、それは EJB コンポーネントへの同時アクセスがあったことを意味し、`ejbStore()` は `ConcurrentModificationException` をスローします。
  - そうでない場合、`ejbStore()` はデータを格納して通常どおりに完了します。

  `ejbStore()` メソッドは、Bean 内のデータが変更されたかどうかに関係なく、トランザクションの最後にこの検証を実行します。

2 回目以降の Bean の使用では、`ejbLoad()` メソッドがその初期データ (バージョン番号を含む) を内部キャッシュからロードすることを除いて、動作は同様です。これにより、データベースへのアクセスが省略されます。`ejbStore()` メソッドが呼び出されると、トランザクションで正しいデータが使用されたことを保証するためにバージョン番号がチェックされます。

バージョンが一致していると、2 つのトランザクションが同じ EJB コンポーネントを同時に使用できるため、めったに変更されない EJB コンポーネントがある場合には効果的な手法です。どちらのトランザクションもデータを変更しないため、両方の

トランザクションの終了時にバージョン番号は不変のままであり、トランザクションは両方とも成功します。ただし、この時点ではトランザクションの並列実行が可能です。2 つのトランザクションが同じ EJB コンポーネントを変更する場合、一方が成功し、もう一方は失敗して、新しい値を使用して再試行することができます。これは、再試行の頻度が一定以下であれば、EJB コンポーネントへのすべてのアクセスを直列化するよりも高速です (ただし、再試行操作を実行するための新しいアプリケーションロジックを準備する必要がある)。

バージョン整合性を使用するには、特定のテーブル用のデータベーススキーマに、バージョンを格納できる列を含める必要があります。その後、特定の Bean に対する配備記述子 `sun-cmp-mapping.xml` でそのテーブルを指定します。

```
<entity-mapping>
    <cmp-field-mapping>
        ...
    </cmp-field-mapping>
    <consistency>
        <check-version-of-accessed-instances>
            <column-name>OrderTable.VC_VERSION_NUMBER</column-name>
        </check-version-of-accessed-instances>
    </consistency>
</entity-mapping>
```

加えて、指定されたテーブル内のデータが変更されたときにバージョン列を自動的に更新するためのトリガーを、データベースに対して確立する必要があります。Application Server では、そのようなトリガーはバージョン整合性を使用する必要があります。そのようなトリガーを定義することにより、EJB データを変更する外部アプリケーションが、進行中の EJB トランザクションと競合しないことも保証されます。

たとえば、次の DDL は、Order テーブルのトリガーを作成する方法を例示します。

```
CREATE TRIGGER OrderTrigger
  BEFORE UPDATE ON OrderTable
  FOR EACH ROW
  WHEN (new.VC_VERSION_NUMBER = old.VC_VERSION_NUMBER)
  DECLARE
  BEGIN
    :NEW.VC_VERSION_NUMBER := :OLD.VC_VERSION_NUMBER + 1;
  END;
```

パーティション分割要求

「パーティション分割要求」では、要求の優先度を EJB コンポーネントに割り当てることができます。これにより、特定の EJB コンポーネントを、ほかよりも高い優先度で実行させる柔軟性が得られます。

要求優先度が割り当てられた EJB コンポーネントは、その要求 (サービス) を、割り当てられたスレッドプールの内部で実行させます。スレッドプールをその実行に割り当てることにより、EJB コンポーネントはほかの保留中の要求から独立して実行できます。要約すると、要求のパーティション分割により、サービスごとに異なるレベルの優先度が割り当てられたサービスレベルアグリーメントを満たすことが可能になります。

パーティション分割要求は、リモートインタフェースを実装するリモート EJB コンポーネントのみに適用されます。ローカル EJB コンポーネントは、その呼び出しスレッド内で実行されます (たとえば、サーブレットがローカル Bean を呼び出すと、サーブレットのスレッド上でローカル Bean 呼び出しが発生する)。

▼ パーティション分割要求を有効にする

1. 管理コンソールを使用して、**EJB** 実行用の追加スレッドプールを設定します。

2. **Application Server** の **ORB** に、追加スレッドプールの **ID** を追加します。

   これは、`domain.xml` ファイルを編集することによって、または管理コンソールを使用して行うことができます。

   たとえば、 `priority-1` および `priority-2` という名前のスレッドプールを有効にするには、`<orb>` 要素を次のように編集します。

   ```
   <orb max-connections="1024" message-fragment-size="1024"
       use-thread-pool-ids="thread-pool-1,priority-1,priority-2">
   ```

3. **EJB** コンポーネントの配備記述子 `sun-ejb-jar.xml` の `use-thread-pool-id` 要素に、スレッドプールの **ID** を含めます。

   たとえば、次に示す `sun-ejb-jar.xml` は、「TheGreeter」という名前の EJB コンポーネントの配備記述子であり、`priority-2` という名前のスレッドプールに割り当てられます。

   ```
   <sun-ejb-jar>
     <enterprise-beans>
       <unique-id>1</unique-id>
       <ejb>
         <ejb-name>TheGreeter</ejb-name>
         <jndi-name>greeter</jndi-name>
         <use-thread-pool-id>priority-1</use-thread-pool-id>
       </ejb>
     </enterprise-beans>
   </sun-ejb-jar>
   ```

4. **Application Server** を再起動します。

特定タイプの EJB コンポーネントのチューニングのヒント

この節では、特定のタイプの EJB コンポーネントのチューニングに関するヒントを示します。

- 46 ページの「エンティティー Beans」
- 46 ページの「ステートフルセッション Beans」
- 47 ページの「ステートレスセッション Beans」
- 47 ページの「読み取り専用エンティティー Beans」
- 48 ページの「コンテナ管理による関係 (CMR) Beans の先取り」

エンティティー Beans

特定のエンティティー Bean の使用状況に応じて、max-cache-size を調整します。これにより、使用頻度の低い Beans (たとえば、作成されたがトランザクションの終了後は使用されることのない注文) はあまりキャッシュされず、頻繁に使用される Beans (たとえば、きわめて頻繁に参照される在庫内の品目) は数多くキャッシュされるようにします。

ステートフルセッション Beans

ステートフルセッション Bean がユーザーを表すとき、Beans の妥当な `max-cache-size` は、アプリケーションサーバープロセス上の同時ユーザーの予測数です。この値が (ユーザーの安定負荷との比較で) 小さすぎると、Beans は頻繁に非活性化および活性化され、ディスク I/O に加えて CPU 負荷がかかる直列化および直列化復元によって応答時間への悪影響が生じます。

チューニングにとって重要なもう 1 つの変数は、`cache-idle-timeout-in-seconds` です。`cache-idle-timeout-in-seconds` の定期的な間隔で、`cache-idle-timeout-in-seconds` の時間よりも長くアクセスがなかったキャッシュ内のすべての Beans が非活性化されます。HTTP セッションのタイムアウトと同様に、`removal-timeout-in-seconds` に指定された時間だけアクセスがなかった Bean は削除されます。非活性化された Beans は、直列化された形式でディスク上に格納されます。非活性化された多数の Beans は、多数のファイルでディスクシステム上の領域を占有するだけでなく、呼び出しの前にセッション状態を直列化復元しなければならないことが原因で応答時間を低下させる可能性もあります。

必要時にのみチェックポイントを設定する

高可用性モードで、ステートフルセッション Beans を使用しているときは、Bean の状態を大きく変更するメソッドのみにチェックポイントを設定することを検討してください。これにより、Bean の状態を持続ストアにチェックポイント設定しなければならない回数が減少します。

ステートレスセッション Beans

ステートレスセッション Beans は、エンティティー Beans やステートフルセッション Beans よりも高速にプールされます。steady-pool-size、pool-resize-quantity 、および max-pool-size の有効な値は、これらのタイプの Beans に対して調整可能なパラメータのうちもっとも多く使用するものです。プールを事前生成したい場合は、steady-pool-size を 0 よりも大きい値に設定します。この設定により、コンテナが発生するとき、コンテナは Beans の steady-pool-size の数値を使用してプールを作成します。プールを事前生成することにより、メソッド呼び出しの間のオブジェクト作成時間を不要にすることができます。

steady-pool size をあまり大きい値に設定すると、メモリーの不要な肥大の原因となり、結果的にガベージコレクションの時間が長くなる可能性があります。pool-resize-quantity は、プールの伸長率と減衰率を決定します。減衰の動作は指数関数的な減衰に似たものになるため、このパラメータは小さい値に設定することをお勧めします。max-pool-size の設定値を小さくすると、現在のプールサイズが max-pool-size を超える場合にインスタンスがプールから破棄される過程で、過度のオブジェクト破棄 (および、その結果として過度のオブジェクト作成) の原因となる可能性があります。

読み取り専用エンティティー Beans

読み取り専用エンティティー Beans は、データベースからデータをキャッシュします。Application Server は、Bean 管理による持続性 (BMP) とコンテナ管理による持続性 (CMP) の両方を使用する読み取り専用 Beans をサポートします。2 つのタイプのうち、パフォーマンス面では CMP 読み取り専用 Beans のほうがずっと優れています。EJB ライフサイクルで、EJB コンテナは読み取り専用 Bean の ejbLoad() メソッドを 1 回呼び出します。コンテナはそのデータから EJB コンポーネントの複数のコピーを作成し、Beans はデータベースを更新しないため、コンテナが ejbStore() メソッドを呼び出すことはありません。これにより、これらの Beans に関係するデータベーストラフィックが大幅に減少します。

データベースを更新することのない Bean が存在する場合は、パフォーマンスを改善するために、その Bean の代わりに読み取り専用 Bean を使用してください。読み取り専用 Bean は次のいずれかの場合に適切です。

- Bean によって表されるデータベース行が変化しない。
- アプリケーションが、Bean に対して古くなった値の使用を許容できる。

たとえば、アプリケーションにおいて、ベストセラー書籍のリストを表現するために読み取り専用 Bean を使用できます。リストはデータベース内で (完全に別の Bean から) 変化する場合がありますが、アプリケーションでその変更をただちに反映する必要はありません。

読み取り専用 Bean の ejbLoad() メソッドの処理は、CMP Beans と BMP Beans で異なります。CMP Beans の場合、EJB コンテナはデータベースからデータをロードするために ejbLoad() を一度だけ呼び出します。Bean の 2 回目以降の使用ではそのデータを

コピーするだけです。BMP Beans の場合、EJB コンテナはトランザクションで Bean が最初に使用されたときに `ejbLoad()` を呼び出します。トランザクション内部で次回以降この Bean を使用する場合、同じ値が使用されます。Bean が使用されるたびに、コンテナはトランザクションの内部で実行されない BMP Bean の `ejbLoad()` を呼び出します。したがって、読み取り専用 BMP Bean は引き続き、数回のデータベースの呼び出しを行います。

読み取り専用 Bean を作成するには、EJB 配備記述子 `sun-ejb-jar.xml` に次の内容を追加します。

```
<is-read-only-bean>true</is-read-only-bean>
<refresh-period-in-seconds>600</refresh-period-in-seconds>
```

更新期間

読み取り専用 Beans のチューニングにとって重要なパラメータに更新期間があります。これは、配備記述子の `refresh-period-in-seconds` エンティティーによって表されます。CMP Beans では、Bean への初回アクセス時に Bean の状態がロードされます。更新期間が経過したあとの最初のアクセスで、データベースからデータを再ロードします。それ以降の Bean の使用ではすべて、(次の更新期間が経過するまでの間) 新しく更新されたデータを使用します。BMP Beans では、既存のトランザクション内部の `ejbLoad()` メソッドは、更新期間が経過しない限りキャッシュされたデータを使用します (経過した場合、コンテナは `ejbLoad()` を再び呼び出す)。

このパラメータでは、EJB コンポーネントが、それが表すデータベース値の「スナップショット」を定期的に更新することを有効にします。更新期間が 0 以下の場合、Bean がデータベースから更新されることはありません (更新期間を指定しない場合のデフォルトの動作)。

コンテナ管理による関係 (CMR) Beans の先取り

コンテナ管理による関係 (CMR) がアプリケーションに存在する場合、ある Bean をロードすると、それに関連するすべての Beans がロードされます。CMR の典型的な例は、注文と注文明細の関係です。1 つの「Order」EJB コンポーネントと、それに関連する複数の「OrderLine」EJB コンポーネントが存在します。Application Server の以前のリリースでは、これらすべての Beans を使用するためには、複数のデータベースクエリーが必要でした。「Order」Bean に対するクエリーと、関係内の各「OrderLine」Beans に対するクエリーです。

一般に、1 つの Bean に *n* 個の関係がある場合、Bean のすべてのデータを使用すると *n+1* 回のデータベースアクセスが必要になります。CMR 先取りを使用して、Bean とそれに関係するすべての Beans のすべてのデータを、1 回のデータベースアクセスで取得します。

たとえば、`ejb-jar.xml` ファイルで次の関係が定義されているとします。

```
<relationships>
    <ejb-relation>
        <description>Order-OrderLine</description>
        <ejb-relation-name>Order-OrderLine</ejb-relation-name>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                Order-has-N-OrderLines
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source>
                <ejb-name>OrderEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>orderLines</cmr-field-name>
                <cmr-field-type>java.util.Collection</cmr-field-type>
            </cmr-field>
        </ejb-relationship-role>
    </ejb-relation>
</relationships>
```

特定の Order がロードされるとき、アプリケーションの `sun-cmp-mapping.xml` ファイルに次の内容を追加することにより、それと関係を持つ OrderLine をロードできます。

```
<entity-mapping>
    <ejb-name>Order</ejb-name>
    <table-name>...</table-name>
    <cmp-field-mapping>...</cmp-field-mapping>
    <cmr-field-mapping>
        <cmr-field-name>orderLines</cmr-field-name>
        <column-pair>
            <column-name>OrderTable.OrderID</column-name>
            <column-name>OrderLineTable.OrderLine_OrderID</column-name>
        </column-pair>
        <fetched-with>
            <default>
        </fetched-with>
    </cmr-field-mapping>
</entity-mappping>
```

Order が取得されると、CMP エンジンは SQL を発行し、次の WHERE 句を持つ SELECT 文によって、すべての関係する OrderLine を取得します。

```
OrderTable.OrderID = OrderLineTable.OrderLine_OrderID
```

この句は外部結合を示します。これらの OrderLine は先取りされます。

一般的に、先取りを利用するとデータベースアクセスの数が減少するため、パフォーマンスが改善されます。ただし、ビジネスロジックでその OrderLine を参照す

ることなく Order を頻繁に使用する場合、先取りはパフォーマンス面でのペナルティーとなる可能性があります。具体的には、実際には必要でない OrderLine を先取りする労力をシステムが費やしたことになります。

特定の検索メソッドについては、先取りを避けます。これにより、しばしばそのペナルティーを回避できます。たとえば、「Order」Bean に 2 つの検索メソッドがあると仮定します。OrderLine を使用する `findByPrimaryKey` メソッドと、注文情報のみを返し OrderLine を使用しない `findByCustomerId` メソッドです。OrderLine に対して CMR 先取りを有効にした場合、両方の検索メソッドが OrderLine を先取りします。ただし、配備記述子 `sun-ejb-jar.xml` に次の情報を含めることにより、`findByCustomerId` メソッドでの先取りを無効にすることができます。

```
<ejb>
    <ejb-name>OrderBean</ejb-name>
    ...
    <cmp>
        <prefetch-disabled>
            <query-method>
                <method-name>findByCustomerId</method-name>
            </query-method>
        </prefetch-disabled>
     </cmp>
</ejb>
```

JDBC とデータベースアクセス

ここでは、データベースアクセスのパフォーマンスを改善するためのヒントを示します。

JDBC を直接使用する

大規模データベースの検索のように大量のデータを扱うときは、エンティティー EJB コンポーネントを使用する代わりに JDBC を直接使用します。

ビジネスロジックをエンティティー EJB コンポーネントにカプセル化する

ビジネスロジックを、そのロジックの処理に必要なデータを保持するエンティティー EJB コンポーネントと結合します。

接続をクローズする

接続がプールに戻されることを保証するために、使用後は必ず接続をクローズします。

データベーストランザクション遮断レベルを最小にする

別の遮断レベルでもアプリケーションが正しく動作し、高いパフォーマンスを発揮するという確証がある場合を除き、`setTransactionIsolationLevel()` を呼び出すことはせず、JDBC ドライバによって提供されるデフォルトの遮断レベルを使用します。

適切な状況では、データベーストランザクション遮断レベルを下げます。遮断レベルを下げると、データベース層での作業量が減少し、アプリケーションパフォーマンスの向上につながる場合があります。ただし、これはデータベーステーブルの使用パターンを入念に分析したあとで行う必要があります。

データベーストランザクション遮断レベルの設定は、管理コンソールで「リソース」、「JDBC」、「接続プール」の順に選択して「<プール名>」ページで行います。JDBC 接続プールのチューニングの詳細は、91 ページの「JDBC 接続プールの設定」を参照してください。

メッセージ駆動型 Beans のチューニング

この節では、JMS とメッセージ駆動型 Beans (MDB) を使用しているときにパフォーマンスを改善するためのヒントを示します。

getConnection() を使用する

JMS 接続は接続プールから供給されます。これは、Queue 接続ファクトリに対する `getConnection()` の呼び出しが高速であることを意味します。

注意 – 8.1 よりも前のバージョンでは、サーブレットまたは EJB コンポーネントとの接続の再利用が可能でした。つまり、サーブレットはその `init()` メソッド内で `getConnection()` を呼び出し、その後連続的に、各サーブレット呼び出しに対して `getSession()` を呼び出すことができました。グローバルトランザクションの内部で JMS を使用する場合、そのような呼び出しは機能しなくなりました。アプリケーションは各接続に対し `getSession()` を一度だけ呼び出すことができます。その後、接続をクローズする必要があります (これは実際には接続をクローズするのではなく、接続をプールに戻すだけである)。これは、移植性のある J2EE 1.4 アプリケーションの一般的な特徴です。Sun Java System Application Server では、以前の (J2EE 1.3 ベースの) アプリケーションサーバーでは施行されていなかったその制約を施行します。

メッセージ駆動型 Bean のプールサイズを調整する

メッセージ駆動型 Beans のコンテナは、エンティティー Beans およびセッション Beans のコンテナとは異なります。MDB コンテナでは、セッションおよびスレッドが MDB プール内の Beans に添付されます。この設計により、メッセージ駆動型要求をコンテナ内で実行するためのスレッドをプールできるようになっています。

メッセージ駆動型 Bean のプールサイズを調整し、メッセージの同時処理数を最適化します。サーバーのすべてのパラメータに基づいて、MDB プールのサイズを設定します (ほかのアプリケーションを考慮に入れる)。たとえば、500 を超える値は一般的には大きすぎます。

MDB プールの設定は、管理コンソールで「設定」、「<設定名>」、「EJB コンテナ」(「MDB 設定」) の順に選択して行うことができます。また、次のように `asadmin` を使用して設定することもできます。

```
asadmin set server.mdb-container.max-pool-size = value
```

Bean 固有リソースをキャッシュする

`setMesssageDrivenContext()` または `ejbCreate()` メソッドを使用して、Bean 固有リソースをキャッシュし、`ejbRemove()` メソッドでそれらのリソースを解放します。

JMS 接続の使用を制限する

JMS 接続を使用するアプリケーションを設計するときは必ず、接続をプールするか、または複数のセッションで同じ接続を使用することによって、使用する接続数を抑える方法論を使用します。

JMS 接続は 2 つのスレッドを使用し、セッションはそれぞれ 1 つのスレッドを使用します。これらのスレッドはプールから取得されたものではなく、結果のオブジェクトはプールされないため、利用負荷の高い期間中はメモリー不足が発生する可能性があります。

1 つの回避方法は、`createTopicConnection` をサーブレットの `init` に移動することです。

必ず、セッションを明示的にクローズしてください。クローズしないとセッションはオープン状態のままであり、リソースを解放しません。

◆ ◆ ◆ 第 3 章

Application Server のチューニング

この章では、Application Server をチューニングしてパフォーマンスを最適化する方法をいくつか説明します。具体的な内容は次のとおりです。

- 53 ページの「配備の設定」
- 55 ページの「ロガーの設定」
- 56 ページの「Web コンテナの設定」
- 57 ページの「EJB コンテナの設定」
- 63 ページの「Java メッセージサービスの設定」
- 64 ページの「トランザクションサービスの設定」
- 66 ページの「HTTP サービスの設定」
- 83 ページの「ORB の設定」
- 90 ページの「スレッドプールの設定」
- リソース:
  - 91 ページの「JDBC 接続プールの設定」
  - 94 ページの「コネクタ接続プールの設定」

配備の設定

配備の設定はパフォーマンスに大きな影響を及ぼします。パフォーマンスを最適化するために配備を設定する際には、次の各指針に従ってください。

- 54 ページの「自動配備を無効にする」
- 54 ページの「プリコンパイルされた JavaServer Pages を使用する」
- 54 ページの「アプリケーションの動的再読み込みを無効にする」

自動配備を無効にする

自動配備を有効にすることは、開発環境では便利ですが、配備に悪影響を及ぼします。本番システムでは、パフォーマンスを最適化するために自動配備を無効にしてください。自動配備が有効になっている場合、「再読込みのポーリング間隔」設定がパフォーマンスに大きな影響を及ぼす可能性があります。

管理コンソールで自動配備を無効にするには、「スタンドアロンインスタンス」>「server (管理サーバー)」の下にある「詳細/アプリケーション設定」タブを選択します。

プリコンパイルされた JavaServer Pages を使用する

JSP ファイルのコンパイルはリソース集約型であり、実行に長い時間がかかります。アプリケーションをサーバーに配備する前に JSP ファイルをプリコンパイルしておけば、アプリケーションのパフォーマンスが改善されます。そのようにした場合、結果として得られるサーブレットクラスファイルのみが配備されます。

管理コンソールまたは deploytool を使えば、アプリケーションを配備するときに JSP ファイルのプリコンパイルを指定できます。また、管理コンソールで「スタンドアロンインスタンス」>「server (管理サーバー)」の下にある「詳細/アプリケーション設定」タブを使えば、配備済みのアプリケーションに対する JSP ファイルのプリコンパイルを指定することもできます。

アプリケーションの動的再読み込みを無効にする

動的再読み込みを有効にすると、サーバーは配備されているアプリケーションの変更を定期的にチェックし、変更のあるアプリケーションを自動的に再読み込みします。動的再読み込みは開発環境向けのものであり、さらにセッション持続との互換性もありません。パフォーマンスを改善するには、クラスの動的再読み込みを無効にします。

管理コンソールで、すでに配備されているアプリケーションのクラスの動的再読み込みを無効にするには、「スタンドアロンインスタンス」>「server (管理サーバー)」の下にある「詳細/アプリケーション設定」タブを選択します。

ロガーの設定

Application Server は、インスタンスのログディレクトリ *appserver-root*/domains/domain-name/logs 内のログファイルに、ログメッセージと例外スタックトレース出力を書き込みます。当然ながら、ログアクティビティーのボリュームは、ベンチマークの状況下では特に、サーバーのパフォーマンスに影響を与える可能性があります。

一般的な設定

一般に、システムログに情報を書き込むとパフォーマンスが若干低下します。また、ログレベルを上げたりファイルローテーションの制限や時間制限を減らしたりすることでディスクアクセスが増大しても、アプリケーションのパフォーマンスが低下します。

また、任意のカスタムログハンドラがネットワークファイルシステムなどの低速デバイスにログを記録しないように十分に注意してください。なぜなら、それはパフォーマンスに悪影響を与える可能性があるからです。

ログレベル

管理コンソールでサーバーやそのサブシステムのログレベルを設定するには、「ロガーの設定」ページの「ログレベル」タブを選択します。このページでは、サーバー (「ルート」と表示) のデフォルトログレベル、EJB コンテナ、MDB コンテナ、Web コンテナ、クラスローダー、JNDI ネーミングシステム、セキュリティーなどの javax.enterprise.system サブシステム (「サーバー」と表示) のデフォルトログレベル、および個々のサブシステムごとのデフォルトログレベルを指定できます。

ログレベルは、最大のログ情報を提供する FINEST から、通常のプログラム実行の妨げとなるイベントのみをログに記録する SEVERE まで変化します。デフォルトのログレベルは INFO です。個々のサブシステムのログレベルは「サーバー」設定よりも優先されますが、その「サーバー」設定は「ルート」設定よりも優先されます。

たとえば、MDB コンテナは、サーバーのデフォルトとは異なるレベルでログメッセージを生成できます。より多くのデバッグメッセージを取得するには、ログレベルを FINE、FINER、または FINEST に設定します。通常の条件下で最高のパフォーマンスを得るには、ログレベルを WARNING に設定します。ベンチマークの条件下では通常、ログレベルを SEVERE に設定するのが適切です。

Web コンテナの設定

管理コンソールで Web コンテナのプロパティーを設定するには、「設定」>「*config-name*」>「Web コンテナ」を選択します。

- 56 ページの「セッションプロパティー: セッションタイムアウト」
- 56 ページの「マネージャープロパティー: 取得間隔」
- 57 ページの「JSP の動的再読み込みを無効にする」

セッションプロパティー: セッションタイムアウト

セッションタイムアウトは、ユーザーがセッションを明示的に無効にしなかった場合にサーバーがそのセッションをどれだけの期間維持するかを決定します。デフォルトは 30 分です。アプリケーションの要件に応じてこの値を調整してください。セッションタイムアウトに非常に大きな値を設定すると、サーバーがセッションストア内に維持すべきセッションの数が多くなりすぎ、パフォーマンスが劣化する可能性があります。一方、非常に小さな値を設定すると、サーバーがセッションを回収するタイミングが早くなりすぎる可能性があります。

マネージャープロパティー: 取得間隔

取得間隔を変更するとパフォーマンスが改善される可能性がありますが、使用するセッションやビジネスロジックの性質を考慮しないでこの値を設定すると、時間に基づく持続周期の場合は特に、データの不整合が発生する可能性があります。

たとえば、取得間隔を 60 秒に設定すると、セッションデータの値が 60 秒ごとに記録されます。このとき、クライアントが 20 秒ごとにサーブレットにアクセスして値を更新すると、不整合が発生します。

たとえば、次のようなオンラインオークションのシナリオを考えます。

- 入札が $5 から始まった場合、60 秒後に記録される値は $8 になります (20 秒間隔が 3 回)。
- 次の 40 秒で、クライアントが価格を増やし始めます。クライアントに表示される値は $10 になります。
- クライアントの 20 秒の休憩の間に、Application Server が 10 秒で停止および起動します。その結果、60 秒間隔で記録された最新値 ($8) が、セッション内に読み込まれます。
- クライアントは $11 が表示されるのを予期しながら再度クリックしますが、代わりに $9 が表示されます。これは間違っています。
- したがって、データの不整合を避けるには、取得間隔の調整時にアプリケーションの予期される動作を考慮してください。

JSP の動的再読み込みを無効にする

本番システムでは、Web コンテナのパフォーマンスを改善するために、JSP の動的再読み込みを無効にします。そうするには、インスタンスごとに `config` ディレクトリ内の `default-web.xml` ファイルを編集します。JSP ファイルのサーブレット定義を次のように変更してください。

```
<servlet>
    <servlet-name>jsp</servlet-name>
    <servlet-class>org.apache.jasper.servlet.JspServlet</servlet-class>
    ...
    <load-on-startup>3</load-on-startup>
</servlet>
```

EJB コンテナの設定

EJB コンテナには、パフォーマンスに影響を与える設定が数多くあります。EJB コンテナの実行やパフォーマンスを追跡するには、ほかの領域の場合と同じく、EJB コンテナの監視機能を使用します。

EJB コンテナの監視

EJB コンテナの監視はデフォルトで無効になっています。管理コンソールで監視を有効にするには、「設定」>「*config-name*」>「監視」を選択します。すべての配備済み EJB コンポーネント、EJB プール、および EJB キャッシュを監視するには、「EJB コンテナ」の監視レベルを「低」に設定します。EJB ビジネスメソッドも監視するには、この監視レベルを「高」に設定します。

EJB コンテナのチューニング

EJB コンテナは、EJB コンポーネントをキャッシュやプールに格納することでパフォーマンスの向上を図ります。キャッシュやプールのプロパティーを調整すれば、EJB コンテナのパフォーマンスを大幅に改善できます。EJB のキャッシュやプールの設定を行うには、管理コンソールの「設定」>「*config-name*」>「EJB コンテナ (EJB 設定)」を選択します。

プール設定がステートレスセッション Beans とエンティティー Beans で使用できるのに対し、キャッシュ設定はステートフルセッション Beans とエンティティー Beans で使用できます。

EJB のプール処理とキャッシュ処理の概要

ステートレスセッション Beans とエンティティー Beans はどちらも、サーバーのパフォーマンスを改善する目的でプール内に格納できます。さらに、ステートフルセッション Beans とエンティティー Beans はどちらも、パフォーマンスを改善する目的でキャッシュ内に格納できます。

表 3–1　Bean タイプ別のプール処理またはキャッシュ処理

Bean 型	プール	キャッシュ
ステートレスセッション	可	不可
ステートフルセッション	不可	可
エンティティー	可	可

プール Bean とキャッシュ Bean の違いは、プール Beans はどれも等価であり、互いに区別できない、という点にあります。これに対し、キャッシュ Beans には、ステートフルセッション Beans の場合は対話状態が格納され、エンティティー Beans の場合は主キーが関連付けられます。エンティティー Beans は、`ejbActivate()` 呼び出し時にプールから削除されてキャッシュに追加され、`ejbPassivate()` 呼び出し時にキャッシュから削除されてプールに追加されます。`ejbActivate()` は、必要なエンティティー Bean がキャッシュ内に存在しない場合に、コンテナによって呼び出されます。`ejbPassivate()` は、キャッシュがその設定された上限を超えた場合に、コンテナによって呼び出されます。

注 – Sun Java Studio を使って EJB コンポーネントを開発および配備する場合、個々の Bean 記述子の設定を、Bean プールや Bean キャッシュ向けに編集する必要があります。これらの設定は、本番レベルの配備に適していない可能性があります。

EJB プールのチューニング

プール内の Bean は、EJB のライフサイクルにおけるプール状態を表します。つまり、その Bean は ID を持ちません。Bean をプール内に格納することの利点は、要求を処理するための Bean の作成時間を短縮できることにあります。このコンテナには、リクエストパスで Bean の作成時間を短縮できるよう、プールオブジェクトをバックグラウンドで作成するためのメカニズムが備わっています。

ステートレスセッション Beans とエンティティー Beans が EJB プールを使用します。ステートレスセッション Beans の使用方法やサーバーが処理するトラフィックの量に注意しながらプールサイズの調整を行い、Beans の過剰な作成や削除が発生しないようにしてください。

EJB のプール設定

ある特定の EJB コンポーネントで、EJB コンテナのキャッシュ設定を上書きする
キャッシュ設定を指定するには、その EJB コンポーネントの sun-ejb-jar.xml 配備記述子の <bean-pool> 要素を使用します。

EJB のプール設定は次のとおりです。

- 初期および最小プールサイズ: プール内に維持される Beans の初期および最小の数。有効な値は 0 から MAX_INTEGER までであり、デフォルト値は 8 です。EJB 配備記述子の対応する属性は、steady-pool-size です。

  中程度の負荷がかかるシステムでは、このプロパティーをゼロよりも大きい数値に設定します。値をゼロよりも大きくすれば、受信要求を処理するインスタンスが、プール内に常に存在するようになります。

- 最大プールサイズ: クライアント要求に応えるために作成できる接続の最大数。有効な値はゼロから MAX_INTEGER までです。デフォルトは 32 です。値ゼロは、プールのサイズが無制限であることを意味します。これは、JVM ヒープがプール内のオブジェクトでいっぱいになる可能性があることを意味します。EJB 配備記述子の対応する属性は、max-pool-size です。

  このプロパティーは、システムの予期される高負荷状態を表す値に設定してください。プールが大きすぎるとメモリーが無駄に消費され、システムのパフォーマンスが低下する可能性があります。プールが小さすぎても、多数の競合が発生するので非効率です。

- プールサイズ変更量: キャッシュがサーバーによって処理されているときに作成または削除される Beans の数。有効な値はゼロから MAX_INTEGER までであり、デフォルトは 16 です。EJB 配備記述子の対応する属性は、resize-quantity です。

  最大プールサイズを変更した場合はプールサイズ変更量も必ず調整し直し、バランスを維持できるようにしてください。一般に、最大プールサイズを大きくした場合には、プールサイズ変更量も大きくすべきです。

- プールアイドルタイムアウト: ステートレスセッション Bean、エンティティー Bean、またはメッセージ駆動型 Bean がプール内でアイドル状態でいられる時間の最大値。この時間を経過した Bean がステートレスセッション Bean、メッセージ駆動型 Bean のいずれかである場合、その Bean は破棄されます。これはサーバーに対するヒントです。デフォルト値は 600 秒です。EJB 配備記述子の対応する属性は、pool-idle-timeout-in-seconds です。

  プール内に存在している Beans が最大プールサイズよりも多い場合、そのプールのサイズは縮小されて初期および最小プールサイズに戻りますが、その処理は、プールアイドルタイムアウトに指定された間隔で、プールサイズ変更量の単位で行われます。サイズ変更量が小さすぎ、かつアイドルタイムアウトが大きいと、プールが縮小されて通常サイズに戻るまでに、非常に長い時間がかかります。

EJB キャッシュのチューニング

キャッシュ内の Bean は、EJB のライフサイクルにおける実行可能状態を表します。これは、その Bean に ID (主キーやセッション ID など) が関連付けられていることを意味します。

キャッシュの外側に移動する Beans は、EJB のライフサイクルに従って非活性化するか破棄する必要があります。非活性化された Bean をキャッシュ内に戻すには、その Bean を活性化する必要があります。エンティティー Beans は一般に、データベース内に格納され、何らかの形式のクエリー言語セマンティクスを使ってデータの読み込みや格納を行います。セッション Beans は、非活性化時にディスクまたはデータベースに格納するために直列化する必要があるほか、活性化時にも同様に直列化復元を行う必要があります。

こうしたキャッシュ内の「実行可能」Beans を使用する着信要求はすべて、作成、ID 設定、および場合によっては活性化のオーバーヘッドを回避することができます。したがって、理論上は、できるだけ多くの Beans をキャッシュに書き込むのが良いことになります。ところが、キャッシュへの書き込みには次のような欠点があります。

- すべての Beans によって消費されるメモリーが、仮想マシン内で利用可能なヒープに影響を与える。
- キャッシュ内のオブジェクトやメモリーを増やすと、ガベージコレクションの実行時間が長くなり、その実行頻度もおそらく多くなる。
- ヒープのチューニングをピーク負荷に基づいて注意深く行わないかぎり、アプリケーションサーバーのメモリーが不足する可能性がある。

アプリケーションによるステートフルセッション Beans やエンティティー Beans の使用方法やサーバーが処理するトラフィックの量に注意しながら EJB のキャッシュサイズおよびタイムアウト設定のチューニングを行い、活性化や非活性化の回数が最小限に抑えられるようにしてください。

EJB のキャッシュ設定

ある特定の EJB コンポーネントで、EJB コンテナのキャッシュ設定を上書きするキャッシュ設定を指定するには、その EJB コンポーネントの `sun-ejb-jar.xml` 配備記述子の `<bean-cache>` 要素を使用します。

EJB のキャッシュ設定は次のとおりです。

最大キャッシュサイズ	キャッシュ内の Beans の最大数。この設定は 1 より大きくしてください。デフォルト値は 512 です。値ゼロはキャッシュが無制限であることを示しますが、これは、キャッシュのサイズが「キャッシュアイドルタイムアウト」と「キャッシュのサイズ変更量」によって制御されることを意味します。EJB 配備記述子の対応する属性は、`max-cache-size` です。
キャッシュのサイズ変更量	キャッシュがサーバーによって処理されているときに作成または削除される Beans の数。有効な値はゼロから `MAX_INTEGER` までであり、デフォルトは 16 です。EJB 配備記述子の対応する属性は、`resize-quantity` です。
削除タイムアウト	ステートフルセッション Bean が非活性化状態 (バックアップストア内でのアイドル状態) でいられる時間。ある Bean へのアクセスがこの期間が過ぎたあとも発生しなかった場合、その Bean はバックアップストアから削除され、クライアントからアクセスできなくなります。デフォルトは 60 分です。EJB 配備記述子の対応する属性は、`removal-timeout-in-seconds` です。
選択内容の削除ポリシー	キャッシュからオブジェクトを削除するのに使用されるアルゴリズム。EJB 配備記述子の対応する属性は、`victim-selection-policy` です。選択肢は次のとおりです。 ■ NRU (最近使用されていない): これがデフォルトですが、これは実際には擬似乱数選択ポリシーです。 ■ FIFO (ファーストインファーストアウト) ■ LRU (最近の使用頻度がもっとも低い)
キャッシュアイドルタイムアウト	ステートフルセッション Bean またはエンティティー Bean がキャッシュ内でアイドル状態でいられる時間の最大値。この時間が経過した Bean は、非活性化されてバックアップストア内に格納されます。デフォルト値は 600 秒です。EJB 配備記述子の対応する属性は、`cache-idle-timeout-in-seconds` です。
更新間隔	読み取り専用 Bean がデータソースから更新される頻度。ゼロ (0) は、Bean が決して更新されないことを意味します。デフォルトは 600 秒です。EJB 配備記述子の対応する属性は、`refresh-period-in-seconds` です。注意: この設定に対応するカスタムフィールドは、管理コンソール内に存在しません。これを設定するには、「追加プロパティー」セクションの「プロパティーを追加」ボタンを使用してください。

個々の **EJB** コンポーネントのプールおよびキャッシュ設定

`sun-ejb-jar.xml` 配備記述子に含まれる個々の EJB のプールおよびキャッシュ設定は、EJB コンテナの設定よりも優先されます。次の表は、キャッシュおよびプール設定の一覧を、EJB コンポーネントのタイプ別に示したものです。

表 3–2 EJB のキャッシュおよびプール設定

	キャッシュ設定						プール設定			
Bean のタイプ	cache-resize-quantity	max-cache-size	cache-idle-timeout-in-seconds	removal-timeout- in-seconds	victim-selection-policy	refresh-period -in-seconds	steady-pool-size	pool-resize-quantity	max-pool-size	pool-idle-timeout -in- seconds
ステートフル セッション	X	X	X	X	X					
ステートレス セッション							X	X	X	X
エンティティー	X	X	X	X	X		X	X	X	X
エンティティー (読み取り専用)	X	X	X	X	X	X	X	X	X	X
メッセージ駆動型 Bean								X	X	X

コミットオプション

コミットオプションは、ある EJB コンポーネントがトランザクションを完了したときに EJB コンテナによって実行されるアクションを制御します。コミットオプションはパフォーマンスに大きな影響を及ぼします。

コミットオプションで使用可能な値には次の 2 つがあります。

- コミットオプション **B:** トランザクション完了時に、Bean はキャッシュ内に保存され、その ID を維持します。同じ主キーに対する次の呼び出しは、このキャッシュ内のインスタンスを使用できます。EJB コンテナは、データベースとの同期メソッドを呼び出す前に、Bean の `ejbLoad()` メソッドを呼び出します。
- コミットオプション **C:** トランザクション完了時に、EJB コンテナは Bean の `ejbPassivate()` メソッドを呼び出します。Bean はその主キーとの関連付けを解除され、未使用プールへと戻されます。同じ主キーに対する次の呼び出しは、プールから未使用 Bean を取得し、そのインスタンスに `PrimaryKey` を設定したあと、そのインスタンス上で `ejbActivate()` を呼び出します。この場合も、EJB コンテナは、データベースとの同期メソッドを呼び出す前に、Bean の `ejbLoad()` メソッドを呼び出します。

オプション B では、`ejbAcivate()` と `ejbPassivate()` の呼び出しを避けられます。このため、ほとんどの場合、オプション C の場合よりもパフォーマンスが良くなりま

す。なぜなら、プールからオブジェクトを取得したりオブジェクトを解放してプールに戻したりするオーバーヘッドの一部を回避できるからです。

ただし、オプション C のほうがパフォーマンスが良くなる場合もあります。キャッシュ内の Beans がほとんど再利用されず、Beans が継続的にキャッシュに追加されるようであれば、Beans をキャッシュに書き込む意味がありません。オプション C が使用される場合、コンテナは、メソッド呼び出し後やトランザクション完了時に、Beans をキャッシュに書き込む代わりにプールに戻します。このオプションでは、インスタンスの再利用がより効率的に行われるほか、JVM 内のライブオブジェクトの数が減少するのでガベージコレクションの高速化も図れます。

最適なコミットオプションの決定

コミットオプション B、コミットオプション C のどちらを使用すべきかを決定するには、まず、Bean の監視コマンドを使ってキャッシュヒット数を調べます。キャッシュヒット数がキャッシュミス数を大幅に上回っている場合は、オプション B が適切な選択肢となります。最良の結果を得るためには、さらに `max-cache-size` と `cache-resize-quantity` を変更する必要がある可能性もあります。

キャッシュヒット数が極めて低く、かつキャッシュミス数が非常に多い場合、アプリケーションが Bean インスタンスを再利用できていないので、`max-cache-size` を使ってキャッシュサイズを増やしても効果はありません (アクセスパターンが同じままであると仮定した場合)。この場合はコミットオプション C を使用することができます。キャッシュヒット数とキャッシュミス数にあまり大きな違いがない場合には、`max-cache-size` をチューニングし、場合によっては `cache-idle-timeout-in-seconds` もチューニングしてください。

Java メッセージサービスの設定

Java Message Service (JMS) がローカルシステム上、リモートシステム上のいずれに存在するかを決定する「型」属性は、パフォーマンスに影響を与えます。パフォーマンスは、ローカルの JMS のほうがリモートの JMS よりも高くなります。ただし、リモートクラスタはフェイルオーバー機能を備えており、管理も一括して行えます。したがって、リモートの JMS を使用することには別の利点が見いだせる可能性があります。JMS の使用方法の詳細については、『Sun Java System Application Server Enterprise Edition 8.2 管理ガイド』の第 4 章「Java Message Service (JMS) リソースの設定」を参照してください。

トランザクションサービスの設定

トランザクションマネージャーを使えば、分散トランザクションのコミットおよびロールバックを行えます。

分散トランザクションシステムは、トランザクションアクティビティーをトランザクションログに書き込み、あとで回復を実行できるようにします。ただし、トランザクションログへの書き込みを行うと、パフォーマンスがある程度低下します。

トランザクションサービスの監視

トランザクションマネージャーの監視はデフォルトで無効になっています。管理コンソールでトランザクションサービスの監視を有効にするには、「設定」>「*config-name*」>「監視」を選択します。

また、次のコマンドを使って監視を有効にすることもできます。

```
set serverInstance.transaction-service.monitoringEnabled=true
reconfig serverInstance
```

監視情報の表示

トランザクションサービスの監視を有効にした場合、その結果を表示するには次のようにします。

- 管理コンソールの「スタンドアロンインスタンス」>「*server-name*」(「監視」|「監視」) を選択する。「ビュー」ドロップダウンから「**transaction-service**」を選択する。
- 次のコマンドを使用する。

  ```
  asadmin get -m serverInstance.transaction-service.*
  ```

トランザクションサービスに関する次の統計が収集されます。

- `total-tx-completed` 完了したトランザクションの数。
- `total-tx-rolled-back` ロールバックされたトランザクションの合計数。
- `total-tx-inflight` 実行中の (アクティブな) トランザクションの合計数。
- `isFrozen` トランザクションシステムが凍結されているかどうか (true または false)
- `inflight-tx` 実行中の (アクティブな) トランザクションの一覧。

次に、`asadmin` を使用した場合の出力のサンプルを示します。

```
********** Stats for JTS ************
 total-tx-completed = 244283
 total-tx-rolled-back = 2640
```

```
total-tx-inflight = 702
isFrozen = False
inflight-tx =
Transaction Id , Status, ElapsedTime(msec)
000000000003C95A_00, Active, 999
```

トランザクションサービスのチューニング

回復よりもパフォーマンスが格段に重要であるような場合に、このプロパティーを使えば、トランザクションロギングを無効にすることができます。このプロパティーはデフォルトでは、サーバー設定内に存在しません。

分散トランザクションロギングの無効化

管理コンソールで分散トランザクションロギングを無効にするには、「設定」>「*config-name*」>「トランザクションサービス」を選択します。「プロパティーを追加」をクリックし、次のように指定します。

- 名前: `disable-distributed-transaction-logging`
- 値: true

  また、`asadmin` を使ってこのプロパティーを設定することもできます。次に例を示します。

  ```
  asadmin set
  server1.transaction-service.disable-distributed-transaction-logging=true
  ```

  この属性を true に設定すると、トランザクションロギングが無効になるため、パフォーマンスが改善される可能性があります。これを false (デフォルト) に設定すると、トランザクションサービスがトランザクションアクティビティーをトランザクションログに書き込むようになり、トランザクションの回復が可能となります。「再起動で回復」にチェックを入れると、このプロパティーは無視されます。

  このプロパティーを true に設定するのは、パフォーマンスがトランザクション回復よりも重要である場合だけにしてください。

再起動で回復 (自動回復)

管理コンソールで「再起動で回復」属性を設定するには、「設定」>「*config-name*」>「トランザクションサービス」を選択します。「回復」チェックボックスをクリックして true (チェックされた状態、デフォルト) または false (チェックが解除された状態) に設定します。

また、`asadmin` を使って自動解除を設定することもできます。次に例を示します。

```
asadmin set server1.transaction-service.automatic-recovery=false
```

「再起動で回復」が true の場合、「分散トランザクションロギングの無効化」属性の値に関係なく、サーバーは常にトランザクションロギングを実行します。

「再起動で回復」が false の場合は次のようになります。

- 「分散トランザクションロギングの無効化」が false (デフォルト) の場合、サーバーはトランザクションログへの書き込みを行います。
- 「分散トランザクションロギングの無効化」が true の場合、サーバーはトランザクションログへの書き込みを行いません。

  トランザクションログへの書き込みを行わない場合、パフォーマンスが約 20 パーセント改善されますが、その代わりに、任意の中断されたトランザクションからの回復ができなくなります。このパフォーマンス向上は、トランザクション集約型のテストを行なった場合のものです。実際のアプリケーションにおける向上率は、これより低くなる可能性があります。

キーポイント間隔

「キーポイント間隔」は、完了したトランザクションのエントリをログファイルから削除する頻度を決定します。キーポイント処理を使えば、プロセスログが無制限に増大するのを防ぐことができます。

頻繁なキーポイント処理はパフォーマンスに悪影響を及ぼします。キーポイント間隔のデフォルト値は 2048 ですが、ほとんどの場合はこれで十分です。

HTTP サービスの設定

クライアント要求を処理する HTTP サーバーインスタンスの監視とチューニングは、Application Server のピーク時のパフォーマンスを確保するうえで重要となります。

- 66 ページの「HTTP サービスの監視」
- 71 ページの「HTTP サービスのチューニング」
- 78 ページの「HTTP リスナー設定のチューニング」
- 82 ページの「Version 7 からの移行」

HTTP サービスの監視

HTTP サービスの監視統計を有効にするには、管理コンソール、`asadmin` のいずれかを使用します。管理コンソールの場合、監視レベル (「低」または「高」) は HTTP サービスの監視に何の効果も持ちません。

`asadmin` の場合、利用可能な監視パラメータを一覧表示するには、次のコマンドを使用します。

```
list --user admin --port 4848
 -m server-instance-name.http-service.*
```

ここで、*server-instance-name* はサーバーインスタンスの名前です。

値を取得するには、次のコマンドを使用します。

```
get --user admin --port 4848 -m server.http-service.parameter-name.*
```

ここで、*parameter-name* は監視するパラメータの名前です。

統計の収集はデフォルトで有効になっています。これを無効にするには、次のプロパティーを `domain.xml` に追加し、サーバーを再起動します。

```
<property name="statsProfilingEnabled" value="false" />
```

統計の収集を無効にするとパフォーマンスが向上します。

管理コンソールでも監視統計を表示できます。この情報は次のカテゴリに分けられます。

- 67 ページの「一般的な HTTP 統計 (http-service)」
- 68 ページの「DNS キャッシュ情報 (dns)」
- 69 ページの「接続キュー」
- 69 ページの「ファイルキャッシュ情報 (file-cache)」
- 70 ページの「キープアライブ (keep-alive)」
- 70 ページの「スレッドプール (pwc-thread-pool)」

一般的な HTTP 統計 (http-service)

管理コンソールが提供するパフォーマンス関連の HTTP 統計は、次のとおりです。

- 直前の 1 分間の平均負荷
- 仮想サーバーオーバーフローが有効かどうか
- HTTP サーバーのバージョン
- HTTP サーバーの ID
- バイトの受信速度
- スレッドの最大数
- HTTP サーバーの起動時刻
- 仮想サーバーの最大数
- プロファイリングが有効かどうか
- HTTP サービスの秒単位の稼働時間
- 直前の 15 分間の平均負荷
- 直前の 5 分間の平均負荷
- バイトの送信速度

DNS キャッシュ情報 (dns)

DNS キャッシュでは、IP アドレスと DNS 名がキャッシュされます。サーバーの DNS キャッシュはデフォルトで無効になっています。Web ベースの管理インタフェースの「監視」の下にある「プロセス ID すべての DNS 統計」ページでは、次の統計が表示されます。

Enabled

DNS キャッシュが無効になっていると、この節の残りの部分は表示されません。

デフォルトで、DNS キャッシュは無効になっています。管理コンソールで DNS キャッシュを有効にするには、DNS の値を「サーバーにアクセスするクライアントに対して DNS 検索を実行する」に設定します。

CacheEntries (CurrentCacheEntries / MaxCacheEntries)

キャッシュエントリの現在の数と最大数。単一のキャッシュエントリは、単一の IP アドレスまたは単一の DNS 名の検索を表します。キャッシュのサイズは、Web サイトに同時にアクセスするクライアントの最大数を格納できるだけの大きさにしてください。キャッシュサイズの設定値が大きすぎるとメモリーが無駄に消費され、パフォーマンスが劣化します。

DNS キャッシュの最大サイズを設定するには、「パフォーマンスチューニング」ページの「DNS キャッシュのサイズ」フィールドの値を入力または変更します。

HitRatio

ヒット率は、キャッシュヒット数をキャッシュ検索数で割ったものです。

この設定はチューニング不可能です。

注 – サーバー上で DNS 検索を無効にすると、ホスト名の制限が正常に機能しなくなるほか、ログファイル内でホスト名の代わりに IP アドレスが表示されます。

DNS エントリのキャッシュへの書き込み

DNS エントリをキャッシュに書き込むかどうかを指定することもできます。DNS キャッシュが有効になっていると、サーバーは、ホスト名の情報を受信したあと、その情報を格納することができます。その後、サーバー上でそのクライアントの情報が必要になった場合、その情報はキャッシュ内に存在しているため、再度クエリーを実行しなくてもその情報を利用できます。DNS キャッシュのサイズと DNS キャッシュエントリの有効期間を指定します。DNS キャッシュには 32 件から 32768 件までのエントリを格納できます。デフォルト値は 1024 です。キャッシュエントリが期限切れになるまでの時間の値は、1 秒から 1 年までの範囲で秒単位で指定できます。デフォルト値は 1200 秒 (20 分) です。

DNS 検索を非同期に制限する

DNS 検索はリソース集約型であるため、サーバープロセス内では使用しないでください。DNS 検索を含める必要がある場合には、検索を非同期にしてください。

Enabled

非同期 DNS が無効になっていると、この節の残りの部分は表示されません。

NameLookups

サーバー起動後に実行された名前検索 (DNS 名から IP アドレス) の数。この設定はチューニング不可能です。

AddrLookups

サーバー起動後に実行されたアドレス検索 (IP アドレスから DNS 名) の数。この設定はチューニング不可能です。

LookupsInProgress

現在実行中の検索の数。

接続キュー

- キューに入れられた接続の合計数: キューに入れられた接続の合計数とは、これまで接続がキューに入れられた回数の合計値のことです。これには、新しく受け付けられた接続と、キープアライブシステムからの接続が含まれます。
- 平均キュー遅延: 平均キュー遅延とは、接続が接続キュー内で費やす時間の平均値のことです。これは、ある要求の接続がサーバーによって受け付けられてから、要求処理スレッド (セッションとも呼ばれる) がその要求の処理を開始するまでの遅延を表します。

ファイルキャッシュ情報 (file-cache)

ファイルキャッシュには静的コンテンツが格納されるため、サーバーは静的コンテンツに対する要求をすばやく処理できます。file-cache セクションは、ファイルキャッシュの使用状況に関する統計を提供します。

ファイルキャッシュのチューニング方法の詳細については、77 ページの「HTTP ファイルキャッシュ」を参照してください。

- キャッシュファイルコンテンツのヒット数
- キャッシュエントリの数
- キャッシュファイル情報のヒット数

- キャッシュに使用されるヒープスペース
- キャッシュファイル情報のミス数
- キャッシュ検索のミス数
- キャッシュファイルコンテンツのミス数
- キャッシュエントリの最大有効期間: この最大有効期間には、有効なキャッシュエントリの最大有効期間が表示されます。
- キャッシュエントリの最大数
- オープンエントリの最大数
- ファイルキャッシュが有効かどうか: キャッシュが無効になっていると、その他の統計は表示されません。キャッシュはデフォルトで有効になっています。
- キャッシュに使用される最大メモリーマップ
- キャッシュに使用されるメモリーマップ
- キャッシュ検索のヒット数
- オープンキャッシュエントリ数: キャッシュエントリの現在の数と最大数がどちらも表示されます。単一のキャッシュエントリは単一の URI を表します。これはチューニング可能な設定です。
- キャッシュに使用される最大ヒープスペース

キープアライブ (keep-alive)

管理コンソールが提供するパフォーマンス関連のキープアライブ統計は、次のとおりです。

- クライアント接続タイムアウトに達したために終了した接続の数
- キープアライブでの最大許容接続数
- ヒット数
- キープアライブモードの接続の数
- 持続的接続が多すぎたためにキープアライブスレッドに渡されなかった接続の数
- アイドル接続が閉じられるまでの秒単位の時間
- 最大キープアライブ数を超過したために閉じられた接続の数

スレッドプール (pwc-thread-pool)

管理コンソールが提供するスレッドプール統計は、次のとおりです。

- Idle/Peak/Limit: Idle は、現在アイドル状態になっているスレッドの数を示します。Peak は、プール内のピーク数を示します。Limit は、スレッドプールで許可されるネイティブスレッドの最大数を示しており、`NativePoolMaxThreads` の設定によって決定されます。
- Work Queue Length /Peak /Limit: これらの数値は、プール内のネイティブスレッドを使用するために待機しているサーバー要求のキューに関するものです。

- Work Queue Length は、ネイティブスレッドを待機している要求の現在の数です。
- Peak は、ネイティブスレッドを使用するために同時にキューに入っていた要求の、サーバー起動後の最大数です。この値は、ネイティブスレッドを必要とする要求の最大並行性とみなすことができます。
- Limit は、ネイティブスレッドを待機するためにキュー内に一度に入れることのできる要求の最大数であり、`NativePoolQueueSize` の設定によって決定されます。

HTTP サービスのチューニング

管理コンソールでは、HTTP サービスの設定は次のカテゴリに分けられています。

- 71 ページの「アクセスログ」
- 71 ページの「要求処理」
- 73 ページの「キープアライブ」
- 75 ページの「接続プール」
- 76 ページの「HTTP プロトコル」
- 77 ページの「HTTP ファイルキャッシュ」

アクセスログ

ベンチマーク実行時にはアクセスロギングを無効にします。アクセスロギングはデフォルトで有効になっています。これを無効にするには、「HTTP サービス」で「プロパティーを追加」をクリックし、次のプロパティーを追加します。

- 名前: `accessLoggingEnabled`
- 値: false

設定可能なアクセスログのプロパティーは、次のとおりです。

- ローテーション (有効/無効)。ローテーションを有効にし、ログのディスク容量が不足しないようにします。
- ローテーションポリシー:時間ベースまたはサイズベース。サイズベースがデフォルトです。
- ローテーション間隔。

要求処理

「HTTP サービス」ページの「要求処理」タブで、次の HTTP 要求処理設定を調整します。

- スレッド数
- 初期スレッド数
- 要求タイムアウト

- バッファー長

スレッド数

「スレッド数」パラメータは、サーバーが処理できる同時要求の最大数を指定します。デフォルト値は 128 です。サーバーは、この要求スレッドの制限に達すると、アクティブ要求数がこの最大数を下回るまで、新しい要求の処理を遅らせます。この値を増やすと、HTTP 応答の待ち時間が短縮されます。

実際には、クライアントがサーバーに接続したあと、要求を完了しないことが頻繁に発生します。それらの場合、サーバーは、「要求タイムアウト」パラメータに指定された時間だけ待ちます。

また、一部のサイトでは、完了までに数分かかる高負荷のトランザクションが実行されます。これらの要因はどちらも、必要とされる最大同時要求数を押し上げます。多くの秒数のかかる要求を多数処理するサイトでは、最大同時要求数を増やさなければいけない可能性があります。

負荷と平均的な要求の処理時間に基づいて、スレッド数の値を調整します。一般に、アイドル CPU 時間と保留中の要求が存在している場合はこの数値を増やし、CPU が過負荷状態になっている場合はこの数値を減らします。HTTP 1.0 クライアント (または接続解除を頻繁に行う HTTP 1.1 クライアント) が多数存在している場合には、タイムアウト値を調整することで、接続を開いたままにしておく時間を短くします。

適切な要求スレッド数の値の範囲は、負荷によって 100 〜 500 になります。システムに余分な CPU サイクルが存在する場合には、スレッド数を段階的に増やしていき、増やすたびにパフォーマンスを監視します。パフォーマンスが飽和したら (改善しなくなったら)、スレッド数を増やすのをやめます。

初期スレッド数

「初期スレッド数」プロパティーは、サーバーが起動時に開始するスレッドの最小数を指定します。デフォルト値は 48 です。「初期スレッド数」は同時に実行可能なアクティブスレッドの最大数に対する強い制限値を表しますが、これがパフォーマンス上の障害になる可能性があります。

要求タイムアウト

「要求タイムアウト」プロパティーは、サーバーがあるクライアントへの接続を受け付けてからそのクライアントから情報を受け取るまでに待機する秒数を指定します。デフォルト設定は 30 秒です。ほとんどの状況下では、この設定を変更する必要はありません。これをデフォルトの 30 秒よりも小さい値に設定すると、スレッドをより早く解放できます。これをデフォルトの 30 秒よりも小さい値に設定すると、スレッドをより早く解放できる場合もあります。ただし、低速な接続のユーザーも切断してしまいます。

バッファー長

個々の要求処理スレッドがクライアントからの要求データを読み込むために使用するバッファーのサイズ (バイト)。

要求の実際のサイズに基づいてこの値を調整し、そのパフォーマンスへの影響を監視してください。ほとんどの場合はデフォルトで十分なはずです。要求のサイズが大きい場合はこのパラメータを増やしてください。

キープアライブ

HTTP 1.0 と HTTP 1.1 はどちらも、単一の HTTP セッション経由で複数の要求を送信する機能をサポートします。サーバーは、新しい HTTP 要求を 1 秒間に数百件受信できます。すべての要求の接続をいつまでも開いたままにできるとしたら、サーバーは多くの接続で過負荷状態になってしまう可能性があります。Unix/Linux システム上では、これにより、容易にファイルテーブルオーバーフローが発生する可能性があります。

Application Server のキープアライブシステムがこの問題を解決します。待機中の「キープアライブ」接続は、1 つ前の要求の処理を完了し、その同じ接続上で新しい要求が到着するのを待っています。サーバーは、待機中のキープアライブ接続の最大数に対するカウンタを維持します。サーバー上で最大待機接続数を超える接続が開いた状態で、新しい接続がキープアライブ要求を待機している場合、サーバーはもっとも古い接続を閉じます。このアルゴリズムにより、開いた状態で待機しているキープアライブ接続の数が制限されます。

システムに余分な CPU サイクルが存在する場合には、キープアライブ設定を段階的に増やしていき、増やすたびにパフォーマンスを監視します。パフォーマンスが飽和したら (改善しなくなったら)、設定を増やすのをやめます。

パフォーマンスに影響を与える HTTP キープアライブ設定は、次のとおりです。

- スレッド数
- 最大接続数
- タイムアウト
- キープアライブクエリー平均時間
- キープアライブクエリー最大スリープ時間

スレッド数

スレッド数は、キープアライブサブシステム内のスレッドの数を決定します。この設定は、システムのプロセッサ数の数倍になるように調整してください。たとえば、2 個の CPU を備えたシステムは、2 個または 4 個のキープアライブスレッドを持つことができます。

デフォルトは 1 です。ユーザー数や最大接続数が少ないサーバーでは、このデフォルトを変更しないでください。

最大接続数

最大接続数は、サーバーが維持するキープアライブ接続の最大数を制御します。指定可能な範囲はゼロから 32768 まであり、デフォルトは 256 です。

この設定は、サーバーが処理すると予想されるキープアライブ接続の数とサーバーの負荷に基づいて調整してください。なぜなら、この設定を増やすとリソース使用量も増え、待ち時間が長くなる可能性があるからです。

「最大接続数」に指定された接続数は、キープアライブスレッド間で均等に分割されます。最大接続数がスレッド数で割り切れない場合、サーバーは、最大接続数よりも若干多い数の同時キープアライブ接続を許可できます。

タイムアウト

タイムアウトは、サーバーが HTTP キープアライブ接続を開いたままで維持する最大時間 (秒) を決定します。クライアントはサーバーへの接続を開いたままに保つことができるため、1 つのサーバーへの複数の要求を、単一のネットワーク接続を使って処理できます。サーバーが処理できるオープン接続の数は限られているため、オープン接続の数が多すぎると、新しいクライアントが接続できなくなります。

タイムアウトのデフォルト値は 30 秒です。したがって、デフォルトでは、接続のアイドル状態が 30 秒を超えると、サーバーはその接続を閉じます。このパラメータの最大値は 300 秒 (5 分) です。

このパラメータの適切な値は、ある特定のクライアントからの各要求間の予想経過時間によって異なります。たとえば、クライアントからの要求頻度が多いことが予期される場合は、このパラメータを大きな値に設定します。同様に、クライアントからの要求頻度が少ないことが予期される場合は、このパラメータを小さな値に設定します。

キープアライブクエリー平均時間

キープアライブクエリー平均時間は、キープアライブ接続のポーリング間隔を指定します。このパラメータの値が *n* ミリ秒である場合、キープアライブ接続を要求したクライアントから見た応答時間は、0 ミリ秒から *n* ミリ秒までのオーバーヘッドを持ちます。

このパラメータのデフォルト値は 1 ミリ秒ですが、キープアライブ接続の予想同時負荷が 300 件未満である場合には、これで問題ありません。このデフォルト値を使用すると、同時負荷が高い場合にスケーラビリティーが極度に低下する恐れがあります。接続負荷の高いアプリケーションでは、このデフォルト値を増やしてください。

このパラメータを設定するには、`asadmin` を使用するか、あるいは管理コンソールの「HTTP サービス」ページで「プロパティーを追加」を選択し、次のように指定します。

- 名前: `keep-alive-query-mean-time`
- 値: *number of milliseconds*

キープアライブクエリー最大スリープ時間

キープアライブクエリー最大スリープ時間は、キープアライブ接続のポーリングによってさらなる要求の有無を確認したあとに待機する最大時間 (ミリ秒) を指定します。システムに余分な CPU サイクルが存在する場合には、このパラメータを段階的に増やしていき、増やすたびにパフォーマンスを監視します。パフォーマンスが飽和したら (改善しなくなったら)、設定を増やすのをやめます。

このパラメータを設定するには、`asadmin` を使用するか、あるいは管理コンソールの「HTTP サービス」ページで「プロパティーを追加」を選択し、次のように指定します。

- 名前: `keep-alive-query-max-sleep-time`
- 値: *number of milliseconds*

接続プール

接続キュー情報は、キュー内のセッション数や接続が受け付けられるまでの平均遅延時間などを示します。

システムに余分な CPU サイクルが存在する場合には、接続プール設定を段階的に増やしていき、増やすたびにパフォーマンスを監視します。パフォーマンスが飽和したら (改善しなくなったら)、設定を増やすのをやめます。

パフォーマンスに影響を与える接続プール設定は、次のとおりです。

- 最大保留カウント
- キューサイズ

最大保留カウント

最大保留カウントは、待機ソケット上で保留状態になっている接続の最大数を指定します。最大保留カウントを調整するのは、システムの負荷が非常に高い場合だけにしてください。低負荷から中程度の負荷の場合、デフォルトで問題ありません。

システムの動作を監視したあと、その結果に基づいて値を変更してください。そうでないと、サーバーは接続をドロップし始めます。バックログキューがいっぱいになった待機ソケット上で接続がタイムアウトすると、その接続は失敗します。最大保留カウントが制限に近づいた場合、高負荷下で接続のドロップが発生しないように、最大接続キューサイズを増やしてください。

キューサイズ

キューサイズは、サーバーが持つことのできる未処理の (まだ処理されていない) 接続の数を指定します。限られた数の要求処理スレッドを持つ、ユーザー数の多い高負荷システムでは、この設定を調整して大きな値にしてください。

注 – 接続キューサイズの設定を大きくしすぎると、サーバーのパフォーマンスが劣化する恐れがあります。その設計目的は、処理できない接続によってサーバーが過負荷状態に陥るのを防ぐことでした。サーバーが過負荷状態に陥っている場合に接続キューサイズを増やしても、要求処理の待ち時間が長くなり、接続キューが再度いっぱいになります。

送信バッファーサイズ

ソケットによって使用される送信バッファーのサイズ (バイト) を指定します。

受信バッファーサイズ

ソケットによって使用される受信バッファーのサイズ (バイト) を指定します。

送信バッファーサイズと受信バッファーサイズは、それぞれ出力バッファー、入力バッファー用として割り当てられるバッファーサイズのことです。これらのパラメータを調整するには、それらの値を規則的に増やし、そのパフォーマンスへの影響を監視します。パフォーマンスが飽和したら (あまり改善されなくなったら)、値を増やすのをやめます。

HTTP プロトコル

パフォーマンスに大きな影響を与える「HTTP プロトコル」属性は、「DNS 検索が有効」だけです。

DNS 検索が有効

この設定は、サーバーにアクセスするクライアントに対する DNS (ドメインネームサービス) 検索を、サーバーが実行するかどうかを指定します。DNS 検索が有効でない場合、あるクライアントが接続したときに、サーバーはそのクライアントの IP アドレスを認識しますが、そのホスト名は認識しません (たとえば、サーバーはクライアントを `www.xyz.com` としてではなく、198.95.251.30 として認識する)。DNS 検索が有効である場合、サーバーはクライアントの IP アドレスを解決してホスト名を得ますが、このホスト名は、アクセス制御、CGI (Common Gateway Interface) プログラム、エラー報告、アクセスロギングなどの処理で使用されます。

サーバーが 1 日に多数の要求への応答を行う場合には、DNS 検索を無効にすることで、DNS または NIS (Network Information System) サーバーへの負荷を減らしてください。DNS 検索を有効にすると待ち時間が長くなり、システムへの負荷が高まります。したがって、有効にする場合には注意が必要です。

HTTP ファイルキャッシュ

Application Server は、ファイルキャッシュを使用することで静的情報をより高速に提供します。ファイルキャッシュには、HTML ファイル、CSS ファイル、イメージファイル、テキストファイルなどの静的ファイルに関する情報が格納されます。HTTP ファイルキャッシュを有効にすると、静的ファイルを含むアプリケーションのパフォーマンスが改善されます。

管理コンソールでファイルキャッシュの属性を設定するには、「設定」>「*config-name*」>「HTTP サービス (HTTP ファイルキャッシュ)」を選択します。

最大ファイル数

最大ファイル数は、キャッシュ内のファイル数を決定します。この値が大きすぎると、必要性のほとんどないファイルまでサーバーがキャッシュに書き込むことになり、メモリーの無駄使いになります。この値が小さすぎると、キャッシュ処理の利点が失われます。この属性のさまざまな値を試し、特定のアプリケーションでの最適解を見つけてください。一般に、効果はそれほど大きくありません。

ハッシュ初期サイズ

ハッシュ初期サイズはメモリー使用量と検索時間に影響を与えますが、これによってパフォーマンスに対する測定可能な効果が得られることは、ほとんどありません。

最大有効期間

このパラメータは、ファイルがキャッシュに書き込まれたあと、そのキャッシュ情報がどのくらいの期間使用されるかを制御します。最大有効期間を経過したエントリは、同じファイルの新しいエントリで置き換えられます。

コンテンツの変更頻度が低い Web サイトでは、パフォーマンスを改善するためにこの値を増やしてください。最大有効期間を設定するには、Web ベースの管理コンソールで、HTTP サーバーノードに対する「HTTP ファイルキャッシュ」ページの「最大有効期間」フィールドで値を入力または変更し、「ファイルキャッシュを有効化」チェックボックスを選択します。

コンテンツを定期的に更新 (既存のファイルを変更) するかどうかに基づいて最大時間を設定します。たとえば、コンテンツが定期的に 1 日に 4 回更新される場合、最大時間を 21600 秒 (6 時間) に設定できます。それ以外の場合、最大時間には、ファイルが変更されたあと、以前のバージョンのコンテンツファイルを提供する最大時間を設定することを検討してください。

小/中のファイルサイズとファイルサイズ上限

キャッシュは小ファイル、中ファイル、大ファイルを異なる方法で処理します。中ファイルのコンテンツは、ファイルが仮想メモリー (Unix/Linux プラットフォーム) にマッピングされることにより、キャッシュされます。小ファイルのコンテンツは、ヒープスペースが割り当てられ、ファイルをそのスペース内に読み込むことにより、キャッシュされます。大ファイルのコンテンツはキャッシュされませんが、大ファイルの情報はキャッシュされます。

小ファイルと中ファイルを区別する利点は、小ファイルが多数ある場合に、仮想メモリーで多くのページが消費されるのを防ぐことにあります。このため、「Small File Size Limit」の値は、通常は VM ページサイズよりもやや低い値になります。

ファイル転送

ファイル転送が有効になっていると、サーバーはファイルのコンテンツではなく、ファイルのオープンファイル記述子をファイルキャッシュに書き込みます。また、キャッシュに書き込まれるのはオープンファイル記述子だけなので、通常行われる小、中、および大ファイルの区別は行われなくなります。

デフォルトでは、Windows では「ファイル転送」は有効に、UNIX では無効に設定されています。UNIX の場合、ファイル転送を有効にできるのは、必須のネイティブ OS サポートを含むプラットフォームだけです。それは HP-UX と AIX です。その他の UNIX/Linux プラットフォームではこれを有効にしないでください。

HTTP リスナー設定のチューニング

管理コンソールで HTTP リスナー設定を変更するには、「設定」>「*config-name*」>「HTTP サービス」>「HTTP リスナー」>「*listener-name*」を選択します。

ネットワークアドレス

ネットワークインタフェースカード (NIC) が 1 枚だけ装備されたマシンでは、ネットワークアドレスをそのマシンの IP アドレス (デフォルトの 0.0.0.0 ではなく 192.18.80.23 など) に設定してください。0.0.0.0 以外の IP アドレスを指定すると、サーバーが接続ごとに行うシステムコールの数が、1 つだけ少なくなります。実現可能な最高のパフォーマンスが得られるよう、0.0.0.0 以外の IP アドレスを指定してください。サーバーに複数の NIC カードが搭載されている場合には、NIC ごとに異なるリスナーを作成してください。

アクセプタスレッド

「アクセプタスレッド」設定は、1 つの待機ソケット上で常にいくつのスレッドを受け付けモードにしておく必要があるかを指定します。システムの CPU 数に等しいかそれより小さい値にこれを設定するのは、良い方法です。

Application Server では、HTTP リスナー上のアクセプタスレッドが接続を受け付け、それらを接続キューに入れます。次に、セッションスレッドがキューから接続を取り出し、その要求を処理します。サーバーは、要求の終了時に必要に応じてセッションスレッドを追加します。

新しいスレッドの追加ポリシーは、接続キューの状態に基づいています。

- 新しい接続が返されるたびに、キュー内で待機している接続の数 (接続のバックログ) が、すでに作成されたセッションスレッドの数と比較されます。その数がスレッド数よりも大きい場合、次回の要求完了時にスレッドを追加するスケジュールが作成されます。
- 直前のバックログが追跡され、次のいずれかが真になるまで *n* 個のスレッドが追加されます (*n* は「HTTP サービス」の「スレッドの増分」パラメータ)。
  - スレッド数が時間とともに増加する。
  - 増分が *n* より大きい。
  - セッションスレッド数からバックログを引いたものが、*n* より小さい。

  ベンチマーク負荷の開始時など、バックログが急激に増加する際にスレッドを作成しすぎないように、サーバーは、スレッドを追加する必要があるかどうかの判断を、すでに存在しているセッションスレッドの数に基づいて、16 個または 32 個の接続ごとに一度だけ下します。

Grizzly の設定

Grizzly は Java の NIO テクノロジを利用した HTTP リスナーであり、そのすべてが Java で実装されています。この再利用可能な NIO ベースのフレームワークを使えば、任意の HTTP 関連操作 (HTTP リスナー/コネクタ) だけでなく非 HTTP 操作も行えるため、スケーラビリティーの高い任意のタイプのマルチスレッドサーバーを作成できます。Grizzly HTTP リスナーは Java プラットフォームの NIO Selector クラス群に基づくキープアライブシステムを使用しますが、このクラス群は接続の監視をサポートしており、サービス拒否攻撃を防ぐのに役立ちます。サービス拒否システムは、リソースの枯渇または「氾濫」攻撃を予測できるよう、IP 検証、IP ごとの完了トランザクション数の追跡、アクティブでない接続の検出などに対する基本的なサポートを追加します。これらすべてのサービスが、キープアライブシステムと組み合わせて実行されます。Grizzly コネクタは静的リソース要求と動的リソース要求の両方をサーブレットコンテナに転送し、サーブレットコンテナはコンテナによって提供される専用のサーブレット (org.apache.catalina.servlets.DefaultServlet) を使って静的リソース要求を処理します。

Grizzly の詳細については、http://weblogs.java.net/blog/jfarcand/archive/2005/06/grizzly_an_http.html の Weblog を参照してください。

Grizzly は、Application Server Enterprise Edition 8.2 に含まれる NSAPI/WebCore の代替として利用可能になっています。Application Server には、Grizzly の設定をサポートするための特殊なプロパティーがいくつか用意されています。Grizzly を有効にするには、次のプロパティーを追加します。

`Dcom.sun.enterprise.web.httpservice.ee=false`。

Grizzly の実装は現在のところ、Application Server Enterprise Edition の、動的設定以外のすべての機能をサポートしています。

Grizzly の設定を制御するプロパティーは、次のとおりです。

- `-Dcom.sun.enterprise.web.connector.grizzly.keepAliveTimeoutInSeconds=30`
- `-Dcom.sun.enterprise.web.connector.grizzly.maxHttpHeaderSize=4096`
- `-Dcom.sun.enterprise.web.connector.grizzly.ssBackLog=4096`
- `-Dcom.sun.enterprise.web.connector.grizzly.queueSizeInBytes=-1`
- `-Dcom.sun.enterprise.web.connector.grizzly.maxKeepAliveRequests=250`
- `-Dcom.sun.enterprise.web.connector.grizzly.fileCache.isEnabled=false`
- `-Dcom.sun.enterprise.web.connector.grizzly.fileCache.maxEntrySize=1024`
- `-Dcom.sun.enterprise.web.connector.grizzly.fileCache.maxLargeFileCacheSize=10485760`

注 – Grizzly も、Sun Java System Application Server 8.2 のその他のすべての設定と同じく、非同期起動メカニズムを無効にするとパフォーマンスが向上します。これを無効にするには次のプロパティーを使用します。
`-Dcom.sun.enterprise.server.ss.ASQuickStartup=false`

次の表に、Grizzly プロパティーと production web container (PWC) プロパティーとの対応関係を示します。

表 3–3　Grizzly プロパティーと PWC プロパティーとの対応関係

Grizzly のプロパティー名	デフォルト値	説明	本番 Web コンテナの対応する設定
maxAcceptWorkerThread	0	OP_ACCEPT (socket.accept()) の処理に使用されるスレッドの数。	HTTP リスナーのアクセプタスレッドの数

表 3–3　Grizzly プロパティーと PWC プロパティーとの対応関係　　(続き)

Grizzly のプロパティー名	デフォルト値	説明	本番 Web コンテナの対応する設定
selector.timeout	60000	Selector.select() でタイムアウトが発生するまでの、ミリ秒単位の時間。	要求処理タイムアウト
minWorkerThreads	5	すべてのスレッドプールが作成時に使用する最小スレッド数。	要求処理の初期スレッド数
fileCache.isEnabled	false	ファイルキャッシュが有効かどうかを示します。	ファイルキャッシュが有効
fileCache.minEntrySize		小ファイルで可能な最小サイズ。	小ファイルのサイズ制限
fileCache.maxEntrySize	1024	中ファイルで可能な最大サイズ。	中ファイルのサイズ制限
fileCache.maxLargeFileCacheSize	10485760	中ファイル用のキャッシュ領域。	ファイルサイズ (中)
fileCache.maxSmallFileCacheSize		小ファイル用のキャッシュ領域。	ファイルサイズ (小)
fileCache.maxCacheEntries		キャッシュエントリの最大数。	最大ファイル数
keepAliveTimeoutInSeconds	30		キープアライブのタイムアウト
maxKeepAliveRequests	250		キープアライブの最大接続数
InitialRuleCount	128	Keep-Alive サブシステムによって作成される KeepAliveRule の初期数。	
useNioNonBlocking	true	NIO ブロックモードを使用するかどうかを示します。	
displayConfiguration	false	Grizzly の内部設定を表示するかどうかを示します。	

表 3–3　Grizzly プロパティーと PWC プロパティーとの対応関係　　(続き)

Grizzly のプロパティー名	デフォルト値	説明	本番 Web コンテナの対応する設定
useDirectByteBuffer	true	Grizzly バッファーの作成時に ByteBuffer.allocateDirect() を使用するかどうかを示します。	
pipelineClass	com.sun.enterprise.web.connector.grizzly.LinkedListPipeline	Grizzly がデフォルトで使用するパイプライン (スレッドプールラッパー)。	
maxSelectorReadThread	1	OP_READ 操作を処理するためのセレクタスレッドの数。	
useByteBufferView	false	1 つの大きな ByteBuffer を使用し、それを Grizzly バッファー間で分割するかどうかを指定します。	
algorithmClassName	com.sun.enterprise.web.connector.grizzly.algorithms.NoParsingAlgorithm	ByteBuffer からのバイトの読み取りに使用される、要求バイト解析アルゴリズム。	
buffersize	4096	Grizzly によって作成される ByteBuffer のサイズ。	
factoryTimeout	30	ソケット上での読み書き操作が失敗するまでの時間。	
maxReadWorkerThread	0	ソケットからの要求バイトの読み取りに使用されるスレッドの数。	

Version 7 からの移行

既存のインストールを Application Server version 7.x からversion 8 へ移行する場合には、次の表でチューニング可能パラメータのマッピングを確認してください。

表 3–4 チューニング可能設定の、version 7 から version 8 へのマッピング

version 7.x のチューニング可能設定	version 8.1 のチューニング可能設定
RqThrottle	スレッド数 (thread-count)
RqThrottleMin	初期スレッド数 (initial-thread-count)
ConnQueueSize	キューサイズ (queue-size-in-bytes)
KeepAliveThreads	スレッド数 (keep-alive-thread-count)
KeepAliveTimeout	タイムアウト (timeout-in-seconds)
MaxKeepAliveConnections	最大接続数 (max-connections)
KeepAliveQueryMeanTime	keep-alive-query-mean-time
KeepAliveQueryMaxSleepTime	keep-alive-query-max-sleep-time
ListenQ	スレッド数 (max-pending-count)
AcceptTimeout	要求タイムアウト
HeaderBufferSize	バッファー長

ORB の設定

Application Server には、高いパフォーマンスとスケーラビリティーを備えた CORBA ORB (Object Request Broker) が含まれています。ORB は、サーバー上の EJB コンテナの基盤となります。

概要

ORB は主に、次の経路で EJB コンポーネントによって使用されます。

- アプリケーションクライアントコンテナを使用するアプリケーションクライアント (またはリッチクライアント) からの RMI/IIOP パス。
- 別の Application Server インスタンスの ORB からの RMI/IIOP パス。
- 別のベンダーの ORB からの RMI/IIOP パス。
- Web コンテナまたは MDB (メッセージ駆動型 Beans) コンテナからのインプロセスパス。

あるサーバーインスタンスが別のサーバーインスタンスの ORB に接続する場合、その最初のインスタンスはクライアント ORB として機能します。SSL over IIOP では、暗号化アルゴリズムの高パフォーマンスなネイティブ実装を備えた、高速で最適化されたトランスポートが使用されます。

EJB のローカルインタフェースでは ORB が使用されない点に注意してください。ローカルインタフェースを使用する場合、引数はすべて参照として渡されるため、オブジェクトをコピーする必要はまったくありません。

クライアントから ORB への接続方法

あるリッチクライアント Java プログラムが新しい `initialContext()` 呼び出しを実行すると、クライアント側の ORB インスタンスが作成されます。次にこれが、Application Server IIOP ポートへのソケット接続を作成します。このクライアントからの IIOP 要求を処理するためのリーダースレッドが、サーバー ORB 上で起動されます。クライアントコードは `initialContext` を使って、サーバー上に配備された EJB の検索を行います。サーバー上に配備された EJB へのリモート参照である IOR が、クライアントに返されます。クライアントコードはこのオブジェクト参照を使って、EJB のリモートメソッドを呼び出します。

Bean の `InitialContext` 検索とメソッドの呼び出しでは、Java の整列化アプリケーション要求データが IIOP メッセージに変換され、そのメッセージが事前に作成されたソケット接続経由でサーバー ORB へと送信されます。するとサーバーは応答を作成し、それを同じ接続経由で送り返します。次に、応答内のそのデータがクライアント ORB によって非整列化され、クライアントコードに戻されて処理されます。リッチクライアントアプリケーションが終了すると、クライアント ORB は停止して接続を閉じます。

ORB の監視

ORB 統計はデフォルトで無効になっています。ORB 統計を収集するには、次の `asadmin` コマンドを使って監視を有効にします。

```
set serverInstance.iiop-service.orb.system.monitoringEnabled=true
reconfig serverInstance
```

接続の統計

ORB の接続に関して収集される統計は、次のとおりです。

- `total-inbound-connections` ORB へのインバウンド接続の合計。
- `total-outbound-connections` ORB からのアウトバウンド接続の合計。

ORB 接続の統計を取得するには、次のコマンドを使用します。

```
asadmin get --monitor
    serverInstance.iiop-service.orb.system.orb-connection.*
```

スレッドプール

ORB のスレッドプールに関して収集される統計は、次のとおりです。

- `thread-pool-size` ORB スレッドプール内のスレッドの数。
- `waiting-thread-count` 作業の到着を待っているスレッドプールスレッドの数。

ORB スレッドプールの統計を取得するには、次のコマンドを使用します。

```
asadmin get --monitor
    serverInstance.iiop-service.orb.system.orb-thread-pool.*
```

ORB のチューニング

ORB のパフォーマンスを調整するには、ORB のパラメータと ORB スレッドプールのパラメータを設定します。通常の場合、負荷分散、複数の共有接続、スレッドプール、およびメッセージ分割サイズを活用することで、応答時間を短縮できます。複数の ORB サーバー間でクライアント要求の負荷分散を行い、クライアントとサーバー間での接続数を調整することで、スケーラビリティーを改善できます。

次の表に、チューニング可能な ORB パラメータをまとめます。

表 3–5 チューニング可能な ORB 設定

パス	ORB モジュール	サーバー設定
アプリケーションクライアントからアプリケーションサーバーへの RMI/ IIOP	通信インフラストラクチャー、スレッドプール	steady-thread-pool-size、max-thread-pool-size、idle-thread-timeout-in-seconds
ORB から Application Server への RMI/ IIOP	通信インフラストラクチャー、スレッドプール	steady-thread-pool-size、max-thread-pool-size、idle-thread-timeout-in-seconds
ベンダー ORB からの RMI/ IIOP	通信インフラストラクチャーの一部、スレッドプール	steady-thread-pool-size、max-thread-pool-size、idle-thread-timeout-in-seconds
インプロセス	スレッドプール	steady-thread-pool-size、max-thread-pool-size、idle-thread-timeout-in-seconds

チューニング可能な ORB パラメータ

管理コンソールを使って次の ORB パラメータをチューニングします。

- スレッドプール ID: 使用するスレッドプールの名前。

- 最大メッセージ分割サイズ: このバイト数よりも大きいメッセージは断片化されます。CORBA GIOPv1.2 の Request、Reply、LocateRequest、および LocateReply メッセージは、複数のフラグメントに分割できます。最初のメッセージは、flags フィールド内でより多くのフラグメントビットが true に設定された、通常の Request または Reply メッセージになります。ORB 間のメッセージの大部分がデフォルトサイズ (1024 バイト) よりも大きい場合、フラグメントのサイズを増やしてネットワーク上の待ち時間を短縮してください。
- 総接続数: 任意のタイミングにおけるすべてのリスナー上の受信接続の最大数。有限の数の接続を許可することで、サーバーの状態を保護します。この値は、接続からアクティブに読み取るスレッドの最大数に等しくなります。
- IIOP クライアント認証が必要 (true/false)

ORB スレッドプールのパラメータ

ORB スレッドプールには、1 つのタスクキューと一連のスレッドが含まれます。タスクまたはジョブはこのタスクキューに挿入されますが、空き状態のスレッドがこのキューからタスクを取り出し、処理を実行します。タスクキューが常に空になるようなスレッドプールサイズを設定しないでください。大規模アプリケーションの場合、最大プールサイズが現在のタスクキューのサイズの 10 倍になるのが普通です。

Application Server は ORB スレッドプールを次の目的で使用します。

- すべての ORB 要求を実行する。
- EJB のプールおよびキャッシュを削除する。

したがって、ORB が RMI/ IIOP 経由のリモート呼び出し用として使用されない場合でも、EJB のプールおよびキャッシュのクリーンアップが円滑に進むように、このスレッドプールのサイズを設定してください。

ORB スレッドプールの属性を設定するには、「設定」>「*config-name*」>「スレッドプール」>「*thread-pool-ID*」を選択します。ここで、thread-pool-ID は ORB 用に選択されたスレッドプールの ID です。スレッドプールでパフォーマンスに影響を与える属性は、次のとおりです。

- 最小プールサイズ: ORB スレッドプール内のスレッドの最小数。通常 (RMI/ IIOP) 負荷時に必要となる平均スレッド数に設定します。
- 最大プールサイズ: ORB スレッドプール内のスレッドの最大数。
- アイドルタイムアウト: アイドルスレッドをプールから削除するまでの待機秒数。スレッドプールの収縮を可能にします。
- 作業キューの数

特に、最大プールサイズがパフォーマンス上、重要になります。詳細については、88 ページの「スレッドプールのサイジング」を参照してください。

クライアント ORB のプロパティー

クライアントプログラムを起動するときに、次のプロパティーをコマンド行引数として指定します。そうするには、Java VM の起動時に次の構文を使用します。

```
-Dproperty=value
```

クライアント ORB とサーバー ORB 間の接続の制御

クライアントでデフォルトの JDK ORB を使用する場合、クライアント ORB からアプリケーションサーバー ORB への接続が、初期コンテキストが作成されるたびに確立されます。同じプロセスからオープンされる接続のプーリングまたは共有を行うには、クライアント ORB の設定に次のコードを追加します。

```
-Djava.naming.factory.initial=com.sun.appserv.naming.S1ASCtxFactory
```

複数の接続の使用

注 – Sun Java System Application Server version 8.x では、プロパティー `com.sun.appserv.iiop.orbconnections` はサポートされていません。

コンテキストファクトリ (`com.sun.appserv.naming.S1ASCtxFactory`) を使用する場合、プロパティー `com.sun.appserv.iiop.orbconnections` を使用することで、クライアント ORB からサーバーに対してオープンする接続の数を指定できます。

デフォルト値は 1 です。複数の接続を使用すると、ネットワーク集約型アプリケーションのスループットが改善される可能性があります。クライアント ORB 上でこの設定変更を指定するには、次の jvm オプションを追加します。

```
-Djava.naming.factory.initial=com.sun.appserv.naming.S1ASCtxFactory
-Dcom.sun.appserv.iiop.orbconnections=value
```

負荷分散

クラスタ内の複数のアプリケーションサーバーインスタンス向けに RMI/IIOP を設定する方法については、『Sun Java System Application Server Enterprise Edition 8.2 高可用性 (HA) 管理ガイド』の第 11 章「RMI-IIOP 負荷分散とフェイルオーバー」を参照してください。

クライアント ORB の負荷分散や接続数を調整する場合、サーバー ORB 上でオープンされる接続の数を考慮してください。小さい接続数から始め、その数を増やしながら何らかのパフォーマンス改善が見られるか監視します。サーバーへの 1 つの接続は、その接続からアクティブに読み取る 1 つの ORB スレッドへと変換されます (これらのスレッドはプールには格納されず、接続の存続期間中に一時的に存在する)。

スレッドプールのサイジング

前述したようにインバウンド接続とアウトバウンド接続の数を確認したあと、スレッドプールのサイズのチューニングを適切に行います。これは、パフォーマンスと応答時間に大きな影響を及ぼす可能性があります。

サイズを計算する際には、同時に処理するクライアント要求の数、マシン上で利用可能なリソース (CPU 数やメモリー容量)、およびクライアント要求処理時に必要とされる応答時間を考慮します。

このサイズを非常に小さい値に設定すると、サーバーが要求を同時に処理する能力に悪影響が及び、その結果として要求がタスクキュー内にとどまる時間が長くなるため、応答時間にも悪影響が及ぶ可能性があります。これに対し、多数のワークスレッドを使って要求を処理させるのも、問題が生じる可能性があります。なぜなら、システムリソースが大量に消費され、並行性が増大するからです。これは、スレッドが EJB コンテナ内の共有構造を取得するまでの時間が長くなることを意味する可能性があり、その結果、応答時間にも悪影響が及ぶ可能性があります。

また、ワークスレッドプールは、プールやキャッシュの削除など、EJB コンテナの整理アクティビティー用としても使用されます。サイズを決定する際には、このアクティビティーのことも考慮する必要があります。ORB ワークスレッドが多すぎるとパフォーマンスに悪影響が及ぶのは、サーバーがそれらすべてのスレッドを維持する必要が生じるからです。アイドルスレッドタイムアウト期間が過ぎると、アイドルスレッドは破棄されます。

IIOP メッセージの検査

Application Server によって渡された IIOP メッセージを調査することが役立つ場合があります。サーバーが IIOP メッセージを `server.log` ファイルに保存するようにするには、JVM オプション `-Dcom.sun.CORBA.ORBDebug=giop` を設定します。クライアント ORB 上で同じオプションを使用します。

サーバーログに保存される IIOP メッセージの例を、次に示します。注意: 実際の出力では、各行の先頭に `[29/Aug/2002:22:41:43] INFO (27179): CORE3282: stdout` のようなタイムスタンプが付きます。

```
 ++++++++++++++++++++++++++++++++
 Message(Thread[ORB Client-side Reader, conn to 192.18.80.118:1050,5,main]):
createFromStream: type is 4 <
 MessageBase(Thread[ORB Client-side Reader, conn to 192.18.80.118:1050,5,main]):
Message GIOP version: 1.2
 MessageBase(Thread[ORB Client-side Reader, conn to 192.18.80.118:1050,5,main]):
ORB Max GIOP Version: 1.2
 Message(Thread[ORB Client-side Reader, conn to 192.18.80.118:1050,5,main]):
createFromStream: message construction complete.
```

```
 com.sun.corba.ee.internal.iiop.MessageMediator
(Thread[ORB Client-side Reader, conn to 192.18.80.118:1050,5,main]): Received message:
 ----- Input Buffer -----
 Current index: 0
 Total length : 340
 47 49 4f 50 01 02 00 04 0 0 00 01 48 00 00 00 05 GIOP.......H....
```

注 – フラグ `-Dcom.sun.CORBA.ORBdebug=giop` を指定すると、多数のデバッグメッセージがログ内に生成されます。これは、メッセージが断片化している疑いがある場合にのみ使用します。

このサンプル出力では、`createFromStream` のタイプが 4 と表示されています。これは、このメッセージがより大きなメッセージのフラグメントであることを意味しています。メッセージの断片化を防ぐには、フラグメントのサイズを増やします。フラグメントのサイズを大きくすると、メッセージがいくつかのフラグメントとしてではなく、1 つの単位として送信されるため、複数メッセージのオーバーヘッドが抑えられ、そうした複数のメッセージを受信側で結合する処理も必要なくなります。

アプリケーションで送信されるメッセージのほとんどが断片化されている場合、フラグメントサイズを増やせばおそらく効率が改善されます。これに対し、いくつかのメッセージだけが断片化されている場合、フラグメントサイズを小さくしたほうがメッセージの書き込みに必要とされるバッファーも小さくなり、より効率的である可能性があります。

Java 直列化による ORB のパフォーマンス改善

ネットワーク上でのトランスポート用データで、標準の CDR (Common Data Representation) の代わりに Java 直列化を使用すると、ORB のパフォーマンスを改善できます。この機能は、Java Serialization over GIOP (General Inter-ORB Protocol) または JSG と呼ばれます。

場合によっては、JSG は CDR よりもパフォーマンスやスループットが良くなる可能性があります。そのパフォーマンスの差異は、アプリケーションによって大きく異なります。クライアントとサーバー間で少量のデータが転送されるようなリモートオブジェクトを含むアプリケーションで JSG を使用すれば、パフォーマンスが改善される可能性が非常に高くなります。

▼ Java 直列化を有効にする

JSG を使用するすべてのサーバー上で、このプロパティーを設定する必要があります。

1 ツリーコンポーネントで、「設定」ノードを開きます。

2 目的のノードを展開します。

3 「**JVM** 設定」ノードを選択します。

4 「**JVM** 設定」ページで「**JVM** オプション」タブを選択します。

5 「**JVM** オプションを追加」をクリックし、次の値を入力します。

```
-Dcom.sun.CORBA.encoding.ORBEnableJavaSerialization=true
```

6 「保存」をクリックします。

7 **Application Server** を再起動します。

アプリケーションクライアントでの **JSG** の使用

アプリケーションがスタンドアロンの非 Web クライアント (アプリケーションクライアント) を使用する場合、JSG を使用するには、クライアントアプリケーションのシステムプロパティーも設定する必要があります。これを行うには通常、クライアントアプリケーションの起動に使用される Java コマンド行に、このプロパティーを追加します。次に例を示します。

```
java -Dcom.sun.CORBA.encoding.ORBEnableJavaSerialization=true
 -Dorg.omg.CORBA.ORBInitialHost=gollum
 -Dorg.omg.CORBA.ORBInitialPort=35309
MyClientProgram
```

スレッドプールの設定

管理コンソールを使えば、スレッドプール設定の監視とチューニングをどちらも行えます。管理コンソールで監視を設定するには、ページ「設定」>「*config-name*」>「監視」を開きます。管理コンソールで監視情報を表示するには、ページ「スタンドアロンインスタンス」>「*instance-name* (監視)」を開きます。

スレッドプールのチューニング (Unix /Linux のみ)

管理コンソールでスレッドプールの設定を行うには、「設定」>「*config-name*」>「スレッドプール」を選択します。

Unix/Linux のスレッドは常に、ユーザーによってスケジュールされる代わりにオペレーティングシステム (OS) によってスケジュールされるため、Unix/Linux ユーザーはネイティブスレッドプールを使用する必要がありません。したがって、Unix/Linux ユーザーインタフェースではこのオプションは提供されていません。ただし、必要であれば、OS によってスケジュールされるスレッドプールの編集や新しいスレッドプールの追加を、管理コンソールで行うことは可能です。

リソース

- 91 ページの「JDBC 接続プールの設定」
- 94 ページの「コネクタ接続プールの設定」

JDBC 接続プールの設定

データベース集約型アプリケーションで最適なパフォーマンスが得られるよう、Application Server によって管理される JDBC 接続プールの調整を行います。これらの接続プールは生存中のデータベース接続を多数維持しますが、そうした接続を再利用すれば、データベース接続のオープン/クローズによるオーバーヘッドの低減が図れます。この節では、JDBC 接続プールをチューニングしてパフォーマンスを改善する方法について説明します。

J2EE アプリケーションは、JDBC 接続プールによって維持されている接続を、JDBC リソース経由で取得します。複数の JDBC リソースが同一の JDBC 接続プールを参照してもかまいません。そのような場合、1 つの物理的な接続プールがすべてのリソースで共有されます。

JDBC 接続プールの監視

JDBC 接続プールの統計収集は、デフォルトで有効になっています。監視される属性は次のとおりです。

- numConnFailedValidation (カウント) 検証に失敗した接続の数。
- numConnUsed (範囲) 使用された接続の数。
- numConnFree (カウント) プール内の未使用接続の数。
- numConnTimedOut (制限付きの範囲) タイムアウトが発生したプール内の接続の数。

統計を取得するには、次のコマンドを使用します。

```
asadmin get --monitor=true
serverInstance.resources.jdbc-connection-pool.*asadmin get
--monitor=true serverInstance.resources.jdbc-connection-pool. poolName.* *
```

JDBC 接続プールのチューニング

管理コンソールで JDBC 接続プールの属性を設定するには、「リソース」>「JDBC」>「接続プール」>「*PoolName*」を選択します。パフォーマンスに影響を与える属性は、次のとおりです。

- 92 ページの「プールサイズの設定」
- 92 ページの「タイムアウトの設定」
- 93 ページの「遮断レベルの設定」

- 93 ページの「接続検証の設定」

プールサイズの設定

接続プールのサイズを制御する設定は、次のとおりです。

初期および最小プールサイズ　作成時のプールのサイズおよびその最小許容サイズ。

最大プールサイズ　プールのサイズの上限。

プールサイズ変更量　アイドルタイムアウト発生時に削除される接続の数。タイムアウトよりも長い間アイドル状態になっていた接続が、削除対象の候補となります。プールサイズが初期および最小プールサイズに達すると、接続の削除が停止します。

次の表に、接続プールのサイジング時に考慮すべき利点と欠点をまとめます。

表 3–6　接続プールのサイジング

接続プール	利点	欠点
小さい接続プール	接続テーブルへのアクセスがより高速になります。	要求の処理に必要な接続を十分に確保できない可能性があります。 要求がキュー内で費やす時間がより長くなります。
大きい接続プール	要求を処理するための接続がより多く存在します。 要求がキュー内で費やす時間が、より少なくなるか、まったくなくなります	接続テーブルへのアクセスがより低速になります。

タイムアウトの設定

タイムアウト設定には次の 2 つがあります。

- 最大待ち時間: 接続タイムアウトになる呼び出し元 (接続を要求するコード) が待つ時間。デフォルト値は 60 秒です。値ゼロを指定すると、呼び出し元が無制限に待機します。

  パフォーマンスを改善するため、「最大待ち時間」をゼロ (0) に設定してください。この場合は基本的に、接続が利用可能になるまで呼び出し元のスレッドがブロックされます。また、この場合、要求ごとに経過した待機時間を追跡する作業の一部からサーバーが解放されるため、パフォーマンスが向上します。
- アイドルタイムアウト: プールで接続がアイドル状態のままでいられる最長時間を指定します。この時間が経過すると、プールはその接続を閉じることができます。このプロパティーは、データベースサーバーの接続タイムアウトを制御しません。

使用不可能な接続が Application Server 内に蓄積されないよう、このタイムアウトがデータベースサーバーのタイムアウトよりも短くなるようにしてください (データベース上でそのようなタイムアウトが設定されている場合)。

最高のパフォーマンスを得るには、アイドル接続が削除されないように、アイドルタイムアウトをゼロ (0) 秒に設定します。これにより、通常は新しい接続の作成に伴う不利益がなくなり、アイドルスレッドの監視が無効化されます。ただし、長期間使用されていない接続をデータベースサーバーがリセットしてしまう危険性があります。

遮断レベルの設定

データベースサーバー上での接続プールのトランザクション遮断レベルを制御する設定には、次の 2 つがあります。

- トランザクション遮断レベル: プール内のデータベース接続のトランザクション遮断レベルを指定します。このパラメータが指定されなかった場合、プールは、JDBC ドライバが提供するデフォルトの遮断レベルを使用します。
- 遮断レベルを保証: プールから取得されたすべての接続が、「トランザクション遮断レベル」パラメータに指定された遮断レベルを持つことを保証します。トランザクション遮断レベルが指定された場合にのみ適用されます。デフォルト値は true です。

  この設定は、一部の JDBC ドライバのパフォーマンスに、ある程度の影響を与える可能性があります。アプリケーションが接続を返す前にその遮断レベルを変更しないことが確実である場合には、false に設定します。

トランザクション遮断レベルを指定しないようにしてください。それが不可能な場合は、「遮断レベルを保証」を false に設定することを検討し、アプリケーションがプログラミングによって接続の遮断レベルを変更しないことを確認してください。

遮断レベルを指定する必要がある場合には、できるだけ高いパフォーマンスが得られるレベルを指定してください。次に、遮断レベルの一覧をパフォーマンスの高い順に示します。

1. READ_UNCOMMITTED
2. READ_COMMITTED
3. REPEATABLE_READ
4. SERIALIZABLE

最高のパフォーマンスが得られ、かつ並行性や整合性に関するアプリケーションのニーズも満たせるような遮断レベルを選択してください。

接続検証の設定

次の各設定によって、プールが接続検証を実行するかどうか、およびその実行方法が決まります。

接続検証	Required がチェックされている場合、プールは、接続の検証 (接続が使用可能かどうかのチェック) を行なったあとで、その接続をアプリケーションに渡します。 可能であれば、デフォルト値の false のままにしてください。接続検証を要求すると、プールが接続を返すたびにサーバーが検証アルゴリズムを適用しなければいけなくなるため、`getConnection()` の待ち時間のオーバーヘッドが増大します。データベースの接続が信頼できる場合には、検証を省略できます。
検証方法	実行する接続検証のタイプ。必ず次のどれかになります。 ■ auto-commit: 接続上で自動コミットの実行を試みます。 ■ metadata: 接続からのメタデータの取得を試みます。 ■ table (指定されたテーブルに対してクエリーを実行する):「表名」も設定する必要があります。JDBC ドライバが `setAutoCommit()` や `getMetaData()` への呼び出しをキャッシュに書き込む場合、この方法を使用しなければいけない可能性があります。
表名	検証方法が「table」の場合にクエリー対象となる表の名前。
「すべての障害ですべての接続を閉じる」チェックボックス	ある単一の検証チェックが失敗した場合にプール内のすべての接続を閉じるかどうか。デフォルトはチェックなしです。失敗した接続を再確立する試行は、1 回行われます。

コネクタ接続プールの設定

パフォーマンスの観点から見ると、コネクタ接続プールは JDBC 接続プールに似ています。1 つ前の節である、91 ページの「JDBC 接続プールのチューニング」で説明した推奨事項に従ってください。

トランザクションサポート

各コネクタ接続プールで指定されたデフォルトのトランザクションサポートを上書きすることで、パフォーマンスを改善できる可能性があります。

たとえば、あるエンタープライズ情報システム (EIS) の接続ファクトリが、グローバルトランザクションよりも高いパフォーマンスを示すローカルトランザクションをサポートしているような場合を考えます。この EIS からのリソースを別のリソースマネージャーからのリソースと混在させる必要がある場合、デフォルト動作では XA トランザクションの使用が強制されるため、パフォーマンスが低下します。しかしながら、この EIS のコネクタ接続プールが LocalTransaction トランザクションサポートを使用するように変更し、前述した最終エージェント最適化機能を活用すれば、より高いパフォーマンスを示す EIS の LocalTransaction 実装を活用できます。LAO の詳細については、42 ページの「JDBC リソースを 1 フェーズコミットリソースとして設定する」を参照してください

管理コンソールでは、「リソース」>「コネクタ」>「コネクタ接続プール」で新しいコネクタ接続プールを作成したりコネクタ接続プールを編集したりするときに、トランザクションサポートを指定します。

また、asadmin を使ってトランザクションサポートを設定することもできます。たとえば、次の asadmin コマンドを使えば、トランザクションサポートが「LOCAL」のコネクタ接続プール「TESTPOOL」を作成できます。

```
asadmin> create-connector-connection-pool --raname jdbcra
--connectiondefinition javax.sql.DataSource
-transactionsupport LocalTransaction
TESTPOOL
```

◆ ◆ ◆ 第 4 章

Java 実行システムのチューニング

この章の内容は次のとおりです。

- 97 ページの「Java 仮想マシンの設定」
- 98 ページの「メモリーとガベージコレクションの管理」
- 105 ページの「詳細情報」

Java 仮想マシンの設定

J2SE 5.0 では、HotSpot Java 仮想マシン (JVM) の 2 つの実装が提供されています。

- クライアント VM は、起動時間を短縮し、メモリーの専有領域を縮小するように調整されています。`-client` JVM コマンド行オプションを使用して呼び出します。
- サーバー VM は、プログラム実行速度が最大になるように設計されています。`-server` JVM コマンド行オプションを使用して呼び出します。

デフォルトでは、Application Server の用途に適した JVM 設定が使用されます。

- アプリケーション開発者向けの Platform Edition では、起動時のパフォーマンスを最適化し、メモリーリソースを節約するために、`-client` JVM フラグが使用されます。
- 本稼働配備向けの Enterprise Edition では、デフォルトの JVM 起動モードが使用されます。デフォルトでは、クライアント Hotspot VM が使用されます。ただし、サーバー VM が必要な場合は、「-server」という名前の <jvm-option> を作成することにより指定できます。

管理コンソールの「設定」>「*config-name*」>「JVM 設定」(「JVM オプション」) で JVM 設定を変更することにより、デフォルト値をオーバーライドできます。

J2SE 5.0 でのサーバークラスマシン検出の詳細については、「サーバークラスマシンの検出」を参照してください。

JVM の詳細については、「JavaTM 仮想マシン」を参照してください。

メモリーとガベージコレクションの管理

アプリケーションの効率は、メモリーとガベージコレクションをどれだけ適切に管理するかによって左右されます。次の各節では、メモリー機能と割り当て機能の最適化について説明しています。

- 34 ページの「目標」
- 100 ページの「ガベージコレクションの追跡」
- 101 ページの「ガベージコレクタのその他の設定」
- 101 ページの「Java ヒープのチューニング」
- 104 ページの「Windows での DLL ベースアドレスの再割り当て」
- 105 ページの「詳細情報」

ガベージコレクタのチューニング

ガベージコレクション (GC) では、以前にオブジェクトに割り当てられ、現在は不要になったヒープスペースが再生されます。デッドオブジェクトを見つけて削除するプロセスによって、アプリケーションが停止状態になったり、25% ものスループットが消費されたりすることがあります。

ほとんどすべての Java 実行時環境には、世代別オブジェクトメモリーシステムと高性能の GC アルゴリズムが備わっています。世代別メモリーシステムでは、ヒープが、入念にサイジングされた「世代」と呼ばれる複数のパーティションに区切られます。世代別メモリーシステムの効率は、ほとんどのオブジェクトの生存期間が短いという観測に基づいています。それらのオブジェクトが蓄積されてメモリー不足の状態になると、GC が強制的に実行されます。

ヒープスペースは、古い世代と新しい世代に分けられます。新しい世代は、新しいオブジェクトの領域 (Eden) と、2 つの Survivor 領域で構成されます。JVM では、新しいオブジェクトは Eden 領域に割り当てられ、生存期間が長いオブジェクトは新しい世代から古い世代に移動されます。

若い世代では、Eden と 2 つの下位領域 (Survivor 領域) を用いる高速コピーガベージコレクタが使用され、生存しているオブジェクトが一方の Survivor 領域からもう一方の Survivor 領域にコピーされます。若い世代の領域で複数回のコレクションを経ても生存しているオブジェクトは、永続化されます。つまり寿命の長い (tenured) 世代にコピーされます。寿命の長い (tenured) 世代はサイズが大きいため、すぐにいっぱいになることはありません。そのため、ガベージコレクションはそれほど頻繁に行われませんが、1 回のコレクションにかかる時間は若い世代の領域のみのコレクションよりも長くなります。寿命の長い (tenured) 世代の領域のコレクションは、全世代のコレクションとも呼ばれます。

頻繁に行われる若い世代の領域のガベージコレクションは短時間 (数ミリ秒) ですが、全世代のコレクションはもっと時間がかかります (ヒープサイズに応じて数 10 ミリ秒から数秒)。

CMS (Concurrent Mark Sweep) など、その他の GC アルゴリズムは増分的で、フル GC はいくつかの増分区分に分けられます。このため、短時間の中断が生じる可能性が高くなります。このプロセスは、オーバーヘッドを伴うため、企業向けの Web アプリケーションでは使用されません。

新しい世代がいっぱいになると、生存しているオブジェクトを古い世代に移動するマイナーコレクションがトリガーされます。古い世代がいっぱいになると、オブジェクトヒープ全体を対象とするメジャーコレクションがトリガーされます。

HotSpot と Solaris JDK はどちらも、スレッドローカルなオブジェクト割り当てプールを使用して、ロックされない高速でスケーラブルなオブジェクト割り当てを行います。そのため、ほとんどの場合、カスタムオブジェクトプールは必要ありません。オブジェクト構築コストが高く、実行プロファイルに大きく影響する場合にのみ、プールを検討します。

ガベージコレクションアルゴリズムの選択

フル GC 中の中断時間が 4 秒を超えると、HADB でのセッションデータの持続に断続的に失敗する可能性があります。

GC が進行している間、Application Server は動作していません。中断が長すぎると、HADB で既存の接続がタイムアウトになります。その後、アプリケーションサーバーが動作を再開したときに、それらの接続を使用してセッションデータを持続しようとすると、HADB によるエラーが生成されます。生成されるエラーは、「セッションデータの保存に失敗しました」、「トランザクションが中止されました」、「HADB サーバーへの接続に失敗しました」などです。

この問題が発生しないようにするには、GC アルゴリズムとして CMS コレクタを使用します。このコレクタはほとんど絶え間なく実行されているので、使用率の高いシステムではスループットが低下する可能性があります。ただし、ガベージコレクタが低頻度で実行される場合に生じる可能性がある長い中断は回避できます。

▼ CMS コレクタを使用する

1. システムの **CPU** 使用率が **100%** ではないことを確認します。
2. 管理ガイドの説明に従って、**HADB** タイムアウトを設定します。
3. サーバーインスタンスで **CMS** コレクタを設定します。
   これを行うには、次の JVM オプションを追加します。
   - -XX:+UseConcMarkSweepGC

- -XX:SoftRefLRUPolicyMSPerMB=1

補足情報

HotSpot ガベージコレクションを監視するには、`jvmstat` ユーティリティーを使用します。105 ページの「詳細情報」を参照してください。

ガベージコレクタのチューニングの詳細については、『Tuning Garbage Collection with the 5.0 Java Virtual Machine』を参照してください。

ガベージコレクションの追跡

ガベージコレクションのパフォーマンスは、主にスループットとポーズによって測定されます。スループットは、GC 以外のアクティビティーに費やされた合計時間の割合です。ポーズは、GC が原因でアプリケーションが応答していないように見える時間です。

そのほかにも、フットプリントと即応性の 2 つの考慮事項があります。フットプリントは JVM プロセスの作業サイズで、ページとキャッシュ行で測定されます。即応性は、オブジェクトがデッド状態になってからメモリーが利用可能になるまでの時間です。分散システムでは、これは重要な考慮事項です。

各世代のサイズを決定するときは、これら 4 つのメトリック間のトレードオフを考慮します。たとえば、若い世代を大きくすれば、スループットを最大にできるかもしれませんが、フットプリントと即応性は犠牲にされます。逆に、若い世代を小さくして 増分 GC を使用すれば、ポーズを最小にできるので即応性は増しますが、スループットは低下します。

JVM 診断の出力に、ガベージコレクションによるポーズに関する情報が表示されます。サーバーを冗長モード (`asadmin start-domain --verbose` *domain* コマンドを使用) で起動する場合は、コマンド行引数の `-verbose:gc` により、コレクションのたびに情報が出力されます。この JVM フラグで生成される情報の出力例を次に示します。

```
[GC 50650K->21808K(76868K), 0.0478645 secs]
 [GC 51197K->22305K(76868K), 0.0478645 secs]
 [GC 52293K->23867K(76868K), 0.0478645 secs]
 [Full GC 52970K->1690K(76868K), 0.54789968 secs]
```

各行の最初の数字は、GC 前のライブオブジェクトの合計サイズ、2 番目の数字は GC 後のライブオブジェクトのサイズ、括弧内の数字は合計空き容量 (ヒープの合計容量から一方の Survivor 領域を差し引いたもの) を示します。最後の数字は、GC に要した時間です。この例は、3 回のマイナーコレクションと 1 回のメジャーコレクションが行われたことを示しています。最初の GC では、コレクションの前は 50650K バイトのオブジェクトが存在していましたが、コレクションのあとは 21808K バイトにな

りました。つまり、28842K バイトのオブジェクトがデッド状態で収集されたことになります。ヒープの合計サイズは 76868K バイトです。コレクションプロセスには 0.0478645 秒を要しました。

その他の有用な監視オプションを次に示します。

- より詳細なログ情報を得るための `-XX:+PrintGCDetails`
- ログファイルに情報を保存するための `-Xloggc:file`

ガベージコレクタのその他の設定

クラスの動的な生成および読み込みを行わないアプリケーションでは、永続的な世代のサイズは GC のパフォーマンスに影響しません。クラスの動的な生成および読み込みを行うアプリケーション (JSP アプリケーションなど) では、永続的な世代がいっぱいになるとフル GC がトリガーされるので、永続的な世代のサイズが GC のパフォーマンスに影響します。永続的な世代の最大サイズを調整するには、`-XX:MaxPermSize` オプションを使用します。

`System.gc()` メソッドを使用してアプリケーションから GC を明示的に呼び出せますが、メジャーコレクションが強制され、大規模なシステムではスケーラビリティーが抑制されるので、この方法はお勧めできません。`-XX:+DisableExplicitGC` フラグを使用して明示的な GC を無効にすることをお勧めします。

Application Server では、管理モジュールの RMI を使用して監視が行われます。RMI ベースの分散アプリケーションのガベージコレクションでは、時々ローカルコレクションを行う必要があるため、RMI では定期的なフルコレクションが強制されます。これらのコレクションの頻度は、`-sun.rmi.dgc.client.gcInterval` プロパティーによって制御します。たとえば、`- java -Dsun.rmi.dgc.client.gcInterval=3600000` と指定すると、デフォルトの 1 分ごとではなく、1 時間ごとに明示的なコレクションが行われます。

Java 仮想マシンの属性を指定するには、管理コンソールを使用して、「*config-name*」>「JVM 設定」(「JVM オプション」) でプロパティーを設定します。

Java ヒープのチューニング

ここでは、パフォーマンスを向上させるための Java ヒープのチューニングに関する項目について説明します。

- 101 ページの「Java ヒープのサイジングのガイドライン」
- 102 ページの「ヒープのチューニングパラメータ」

Java ヒープのサイジングのガイドライン

最大ヒープサイズは、プロセスあたりの最大アドレス空間によって決まります。次の表に、各種プラットフォームのプロセスあたりの最大アドレス値を示します。

表 4–1 プロセスあたりの最大アドレス空間

オペレーティングシステム	プロセスあたりの最大アドレス空間
Redhat Linux 32 bit	2G バイト
Redhat Linux 64 bit	3G バイト
Windows 98/2000/NT/Me/XP	2G バイト
Solaris x86 (32 bit)	4G バイト
Solaris 32 bit	4G バイト
Solaris 64 bit	テラバイト

プロセスにもスタックやライブラリなどのための領域が必要なため、最大ヒープスペースは常に、プロセスあたりの最大アドレス空間よりも小さくなります。割り当て可能な最大ヒープスペースを決定するには、プロファイリングツールを使用して、メモリーがどのように使用されているかを調べます。プロセスで使用される最大スタック空間と、ライブラリやその他のメモリー構造で使用されるメモリー量を測定します。最大アドレス空間とこれらの値の合計の差が、ヒープに割り当て可能なメモリー量になります。

ヒープサイズを大きくしたり、別のガベージコレクタを使用したりすることで、パフォーマンスを改善できます。一般に、長時間実行されるサーバーアプリケーションでは、複数のプロセッサを持つマシンで J2SE スループットコレクタを使用し (`-XX:+AggressiveHeap`)、ヒープサイズはマシンの空きメモリーをできるだけ多く使用した大きさにします。

ヒープのチューニングパラメータ

次の JVM パラメータでヒープサイズを制御できます。

- `-Xms`*value*
- `-Xmx`*value*
- `-XX:MinHeapFreeRatio=`*minimum*
- `-XX:MaxHeapFreeRatio=`*maximum*
- `-XX:NewRatio=`*ratio*
- `-XX:NewSize=`*size*
- `-XX:MaxNewSize=`*size*
- `-XX:+AggressiveHeap`

`-Xms` パラメータと `-Xmx` パラメータはそれぞれ、最小ヒープサイズと最大ヒープサイズを定義します。世代がいっぱいになると GC が行われるので、スループットは使用可能メモリーの量に反比例します。デフォルトでは、JVM はコレクションのたびにヒープサイズを拡大または縮小して、ライブオブジェクトに対する空き容量の割合

を一定の範囲内に維持しようとします。この範囲は、`-XX:MinHeapFreeRatio=`*minimum* パラメータと `-XX:MaxHeapFreeRatio=`*maximum* パラメータでパーセント値として設定され、合計サイズは `-Xms` と `-Xmx` によって制限されます。

ヒープサイズを固定する場合は、`-Xms` と `-Xmx` に同じ値を設定します。ヒープサイズが拡大または縮小されると、JVM では、事前に定義された `NewRatio` を維持するために、古い世代と新しい世代のサイズが再計算されます。

`NewSize` パラメータと `MaxNewSize` パラメータは、新しい世代の最小サイズと最大サイズを制御します。新しい世代のサイズを固定するには、これらのパラメータに同じ値を設定します。若い世代を大きくするほど、マイナーコレクションの発生回数は少なくなります。古い世代に対する若い世代のサイズの比率は、`NewRatio` で制御されます。たとえば、`-XX:NewRatio=3` と設定すると、古い世代と若い世代の比率は 1:3 になり、Eden 領域と Survivor 領域の合計サイズはヒープサイズの 4 分の 1 になります。

デフォルトでは、Application Server は Java HotSpot Server JVM によって起動されます。Server JVM のデフォルトの `NewRatio` は 2 で、古い世代がヒープの 3 分の 2、新しい世代が 3 分の 1 を占めます。新しい世代を大きくするほど、生存期間の短いオブジェクトを数多く収容できるので、時間のかかるメジャーコレクションの回数が減ります。同時に、古い世代も、生存期間の長いオブジェクトを多数保持するのに十分な大きさがあります。

Java ヒープのサイズを決定するには、次の手順に従います。

- JVM に割り当てることができるメモリーの総量を決定します。それに応じて、若い世代のサイズに対する独自のパフォーマンスメトリックをグラフにして、最適な設定を求めます。
- 若い世代に十分なメモリーを割り当てます。デフォルト値は、`NewRatio` と `-Xmx` の設定から計算されます。
- Eden 領域または若い世代の領域を大きくするほど、フル GC の実行間隔は長くなります。ただし、それに比例して若い世代でのコレクションに時間がかかる可能性があります。一般に、Eden のサイズは最大ヒープサイズの 4 分の 1 から 3 分の 1 に設定します。古い世代は新しい世代よりも大きくしてください。

最新のデフォルト設定については、「Java HotSpot VM Options」を参照してください。

例 4–1　Solaris でのヒープ設定

Solaris 上の Application Server で大きなアプリケーションに対して使用されるヒープ設定の例を次に示します。

```
-Xms3584m
 -Xmx3584m
 -verbose:gc
```

例 4–1 Solaris でのヒープ設定 (続き)

```
-Dsun.rmi.dgc.client.gcInterval=3600000
```

Survivor 領域の比率の設定

`SurvivorRatio` パラメータは、2 つの Survivor 領域のサイズを制御します。たとえば `-XX:SurvivorRatio=6` と設定すると、各 Survivor 領域と Eden 領域の比率は 1:6 になり、それぞれの Survivor 領域のサイズは若い世代全体の 8 分の 1 になります。Solaris のデフォルトは 32 です。Survivor 領域が小さすぎると、コピーコレクションがオーバーフローし、その分が直ちに古い世代に送られてしまいます。Survivor 領域が大きすぎると、無意味になります。GC のたびに、JVM は、寿命の長い (tenured) 世代に送るまでにオブジェクトをコピーできる回数 (tenure しきい値と呼ばれる) を判断します。このしきい値は、Survivor 領域の使用量が半分に維持されるように選択されます。

このしきい値と、新しい世代のオブジェクトのコピー回数を表示するには、`-XX:+PrintTenuringDistribution` オプションを使用します。これは、アプリケーションの生存期間の分布を観測するのに役立ちます。

Windows での DLL ベースアドレスの再割り当て

JVM は、初期化時に、 `-Xms` 設定を使用してヒープを割り当てようとします。Application Server DLL のベースアドレスによって、利用可能な連続アドレス空間の量が制限され、JVM の初期化に失敗することがあります。Java メモリーで利用可能な連続アドレス空間の量は、DLL に割り当てられたベースアドレスによって異なります。Application Server DLL のベースアドレスの再割り当てを行うことにより、利用可能な連続アドレス空間の量を増やせます。

読み込みアドレスの衝突を避けるには、Visual Studio および Platform SDK に付属する rebase ユーティリティーを使用して、優先ベースアドレスを設定します。rebase ユーティリティーを使用して Application Server DLL のベースアドレスを割り当て直すことで、読み込み時の再配置を避け、Java ヒープで利用可能なプロセスメモリーを増やします。

衝突の原因となる可能性があるデフォルト以外のベースアドレスを持つ Application Server DLL がいくつかあります。次に例を示します。

- `nspr` ライブラリは、0x30000000 の優先アドレスを持ちます。
- `icu` ライブラリは、0x4A?00000 のアドレスを持ちます。

利用可能な最大連続アドレス空間を実質的に増やすには、これらのライブラリをシステム DLL (`msvcrt.dll` は `0x78000000`) の近くに移動します。すべての DLL でベースアドレスの再割り当てを行えるので、Application Server をインストールしたら、DLL のベースアドレスの再割り当てを実行してください。

▼ Application Server DLL のベースアドレスの再割り当てを行う

始める前に　ベースアドレスの再割り当てを実行するには、次のものが必要です。

- Windows 2000
- Visual Studio および Microsoft Framework SDK の rebase ユーティリティー

1. 次のコマンドを使用して、*install_dir*\bin をデフォルトディレクトリにします。

```
cd install_dir\bin
```

2. 次のコマンドを入力します。

```
rebase -b 0x6000000 *.dll
```

3. dependencywalker ユーティリティーを使用して、**DLL** のベースアドレスの再割り当てが正しく行われたことを確認します。

   詳細については、Dependency Walker の Web サイトを参照してください。

4. **Java** ヒープのサイズを大きくし、管理コンソールの「**JVM** 設定」ページで、それに応じた **JVM** オプションを設定します。

5. **Application Server** を再起動します。

例 4–2 Windows でのヒープ設定

次に、Windows 上の Sun ONE Application Server で負荷の大きいサーバー中心型のアプリケーションに対して使用されるヒープ設定の例を示します。これは、server.xml ファイルに設定されます。

```
<jvm-options> -Xms1400m </jvm-options>
<jvm-options> -Xmx1400m </jvm-options>
```

参照　ベースアドレスの再割り当ての詳細については、MSDN の rebase ユーティリティーに関するページを参照してください。

詳細情報

JVM のチューニングの詳細については、次の Web ページを参照してください。

- 『Java HotSpot VM Options』
- 『Frequently Asked Questions About the Java HotSpot Virtual Machine』
- 『Performance Documentation for the Java HotSpot VM』
- Java のパフォーマンスに関する Web ページ
- 『Java プラットフォームの監視と管理 (J2SE 5.0)』
- jvmstat 監視ユーティリティーに関する Web ページ

◆ ◆ ◆ 第 5 章

5 オペレーティングシステムのチューニング

この章では、最適なパフォーマンスを得るための、オペレーティングシステム (OS) のチューニングについて説明します。この章の内容は次のとおりです。

- 107 ページの「サーバーのスケーリング」
- 109 ページの「Solaris のチューニング」
- 113 ページの「x86 上の Solaris のチューニング」
- 115 ページの「Linux プラットフォームのチューニング」

サーバーのスケーリング

ここでは、次のサーバーサブシステムに関するサーバーのスケーリングで最適なパフォーマンスを得るための推奨事項について説明します。

- 107 ページの「プロセッサ」
- 108 ページの「メモリー」
- 108 ページの「ディスク容量」
- 108 ページの「ネットワーク関連機能」

プロセッサ

Application Server では、自動的に複数の CPU が利用されます。一般に、複数の CPU による効率はオペレーティングシステムと作業負荷によって異なりますが、通常は、プロセッサが多いほど動的コンテンツのパフォーマンスは向上します。

静的コンテンツには、CPU アクティビティーよりも、主に入出力 (I/O) アクティビティーが関係します。サーバーが適切に調整されている場合、主メモリーを増やせばコンテンツのキャッシュ量も増え、その結果、CPU アクティビティーに使用される時間に対して I/O アクティビティーに使用される時間の相対量が増えます。CPU 数を 2 倍にすると、サーブレットのパフォーマンスが 50 ～ 80% 向上するということが、調査により判明しています。

メモリー

サポートされているオペレーティングシステム別のメモリーに関する推奨事項については、『Sun Java System Application Server Enterprise Edition 8.2 リリースノート (UNIX 版)』の「ハードウェアとソフトウェアの要件」を参照してください。

ディスク容量

OS、ドキュメントツリー、およびログファイルに十分な ディスク容量を用意することをお勧めします。ほとんどの場合は、合計 2G バイトで十分です。

OS、スワップファイルとページングファイル、Application Server ログ、およびドキュメントツリーを、それぞれ別個のハードドライブに配置します。このようにすると、ログドライブがログファイルでいっぱいになっても、OS に影響しません。また、OS のページングファイルによるドライブアクティビティーが発生しているかどうかを見分けることなども容易です。

OS ベンダーは、通常、スワップ空間やページング空間の割り当て量の推奨値を提示しています。Sun のテストによると、Application Server は、スワップ空間が RAM と同じ容量の場合に最適に動作し、ドキュメント ツリーのマッピングにも十分に対応できます。

ネットワーク関連機能

アプリケーションに必要な帯域幅を決定するには、次の値を調べます。

- サーバーで処理する必要があるピーク時の同時ユーザー数 (N_{peak})。
- サイトの平均要求サイズ (*r*)。平均要求には、複数のドキュメントが含まれる場合があります。不確かな場合は、ホームページと、関連付けられたすべてのファイルとグラフィックスを使用します。
- ピーク使用時に平均的なユーザーがドキュメントを快く待っていられる時間 (*t*) を決めます。

次に、必要な帯域幅を次のように計算します。

$N_{peak} r / t$

たとえば、ピーク時に 50 人のユーザーをサポートし、平均ドキュメントサイズが 24K バイトで、各ドキュメントを平均 5 秒で送信する場合は、240K バイト (1920K ビット/秒) 必要です。そのため、このサイトには 2 本の T1 回線 (それぞれが 1544K ビット/秒) が必要です。この帯域幅だと、オーバーヘッドがいくらか増えても対応できます。

サーバーのネットワークインタフェースカードは、接続される WAN 以上をサポートする必要があります。たとえば、最大で 3 本の T1 回線を使用する場合は、10BaseT インタフェースで対応できます。1 本の T3 回線 (45M ビット/秒) までは、100BaseT を使用できます。ただし、WAN の帯域幅が 50M ビット/秒を超える場合は、複数の 100BaseT インタフェースを設定するか、ギガビット Ethernet 技術を検討してください。

Solaris のチューニング

- 109 ページの「チューニングパラメータ」
- 111 ページの「ファイル記述子の設定」
- 111 ページの「代替スレッドの使用」

チューニングパラメータ

Solaris の TCP/IP 設定のチューニングは、多数のソケットを開閉するプログラムで役立ちます。Application Server は少数の固定された接続のセットで動作するので、パフォーマンスがそれほど顕著に向上しない場合もあります。

次の表に、パフォーマンスとスケーラビリティーのベンチマークに影響する Solaris のチューニングパラメータを示します。これらの値は、最適なパフォーマンスを得るためのシステムの調整例を示しています。

表 5–1 Solaris のパラメータのチューニング

パラメータ	スコープ	デフォルト	調整値	コメント
rlim_fd_max	/etc/system	1024	8192	プロセスが開けるファイル記述子の制限。予想される負荷 (存在する場合は、関連付けられたソケット、ファイル、パイプなど) を考慮して設定します。
rlim_fd_cur	/etc/system	1024	8192	
sq_max_size	/etc/system	2	0	ストリームドライバのキューサイズを制限します。0 に設定すると無限になり、バッファーの容量不足がパフォーマンスに影響しなくなります。クライアント上でも設定します。
tcp_close_wait_interval	ndd /dev/tcp	240000	60000	クライアント上でも設定します。

表 5–1 Solaris のパラメータのチューニング (続き)

パラメータ	スコープ	デフォルト	調整値	コメント
tcp_time_wait_interval	ndd /dev/tcp	240000	60000	
tcp_conn_req_max_q	ndd /dev/tcp	128	1024	
tcp_conn_req_max_q0	ndd /dev/tcp	1024	4096	
tcp_ip_abort_interval	ndd /dev/tcp	480000	60000	
tcp_keepalive_interval	ndd /dev/tcp	7200000	900000	トラフィックの多い Web サイトでは、この値を小さくします。
tcp_rexmit_interval_initial	ndd /dev/tcp	3000	3000	再転送率が 30 〜 40% を超える場合は、この値を大きくします。
tcp_rexmit_interval_max	ndd /dev/tcp	240000	10000	
tcp_rexmit_interval_min	ndd /dev/tcp	200	3000	
tcp_smallest_anon_port	ndd /dev/tcp	32768	1024	クライアント上でも設定します。
tcp_slow_start_initial	ndd /dev/tcp	1	2	データが少量であればやや高速に転送します。
tcp_xmit_hiwat	ndd /dev/tcp	8129	32768	送信バッファーのサイズ。
tcp_recv_hiwat	ndd /dev/tcp	8129	32768	送信バッファーのサイズ。
tcp_recv_hiwat	ndd /dev/tcp	8129	32768	受信バッファーのサイズ。
tcp_conn_hash_size	ndd /dev/tcp	512	8192	接続ハッシュテーブルのサイズ。110 ページの「接続ハッシュテーブルのサイジング」を参照してください。

接続ハッシュテーブルのサイジング

接続ハッシュテーブルには、アクティブな TCP 接続に関するすべての情報が保持されます。接続ハッシュテーブルのサイズを取得するには、次のコマンドを使用します。

```
ndd -get /dev/tcp tcp_conn_hash
```

この値によって接続数が制限されることはありませんが、接続のハッシュにかかる時間が長くなることがあります。デフォルトサイズは 512 です。

ルックアップを効率化するには、サーバーで予想される同時 TCP 接続数の半分の値を設定します。この値は、`/etc/system` でのみ設定可能で、起動時に有効になります。

現在の TCP 接続数を取得するには、次のコマンドを使用します。

```
netstat -nP tcp|wc -l
```

ファイル記述子の設定

Solaris では、ulimit プロパティーによる開けるファイルの最大数の設定は、サポートできる RMI/IIOP クライアントの最大数に大きく影響します。

強い制限値を引き上げるには、/etc/system に次のコマンドを追加して 1 回再起動します。

```
set rlim_fd_max = 8192
```

次のコマンドを使用して、この強い制限値を確認します。

```
ulimit -a -H
```

強い制限値を設定したら、次のコマンドを使用して、このプロパティーの値を明示的に (この制限値まで) 増やします。

```
ulimit -n 8192
```

次のコマンドを使用して、この制限値を確認します。

```
ulimit -a
```

たとえば、ulimit のデフォルト値の 64 では、簡単なテストドライバでサポートできる同時クライアントの数は 25 ですが、ulimit を 8192 に設定すると、同じテストドライバで 120 の同時クライアントをサポートできます。このテストドライバは複数のスレッドを生成し、各スレッドは JNDI 検索を実行して同じビジネスメソッドを繰り返し呼び出します。ビジネスメソッドの呼び出し間の思考 (遅延) 時間は 500 ミリ秒で、約 100K バイトのデータがやりとりされます。

これらの設定は、Solaris 上の RMI/IIOP クライアントに適用されます。ファイル記述子の制限設定の詳細については、Solaris のオンラインマニュアルを参照してください。

代替スレッドの使用

Solaris オペレーティング環境では、デフォルトで 2 レベルのスレッドモデルがサポートされます (Solaris 8 まで)。アプリケーションレベルの Java スレッドは、ユーザーレベルの Solaris スレッドにマッピングされ、制限のある軽量プロセス (LWP) プール上で多重化されます。カーネルリソースを維持し、システムの効率を最大にするに

は、システムのプロセッサと同じ数の LWP があれば十分です。これは、ユーザーレベルのスレッドが何百もある場合に役立ちます。JVM に応じて、複数のスレッドモデルか、モデル内での複数の同期メソッドのどちらかを選択できます。

Solaris 8 で、LD_LIBRARY_PATH の /usr/lib の前に /usr/lib/lwp を挿入して、/usr/lib/lwp/ にある代替スレッド libthread.so を読み込んでみてください。一部のアプリケーションでは、特に、使っているスレッドが少ない場合は、スループットとシステム利用率が向上します。

デフォルトでは、Application Server は /usr/lib/lwp を使用します。デフォルトの設定を変更して LWP を使わないようにするには、startserv スクリプトの LD_LIBRARY_PATH から /usr/lib/lwp を削除します。ただし、本当に必要な場合以外は変更しないでください。

多数のスレッドを使うアプリケーションには、/usr/lib/libthread.so ライブラリの使用をお勧めします。これにより、1.4 のデフォルトである LWP ベースの同期が有効になるだけでなく、TLABS (スレッドローカル割り当てバッファー) が無効になるため、ヒープがすぐに使い果たされてガベージコレクションが頻繁に行われる可能性があります。多数のスレッドを使用するアプリケーションでは -Xconcurrentio の使用方法も参照してください。

詳細情報

Solaris のスレッドに関する問題の詳細については、「Solaris and Java Threading」を参照してください。

HotSpot JVM のチューニングの詳細については、105 ページの「詳細情報」を参照してください。

Linux の設定

システムの起動時に実行される /etc/rc.d/rc.local ファイルに、次のパラメータを追加してください。

```
<-- begin
#最大ファイル数は 4M バイトあたり 256 記述子までに更新されました。
システムの RAM 容量に基づいて、ファイル記述子の数を指定します。
echo "6553" > /proc/sys/fs/file-max
#inode-max は file-max の 3 〜 4 倍です
#ファイルが存在しません。
#echo"262144" > /proc/sys/fs/inode-max
#利用できるローカルポートを増やします
echo 1024 25000 > /proc/sys/net/ipv4/ip_local_port_range
#ソケットバッファーで利用できるメモリーを増やします
echo 2621143 > /proc/sys/net/core/rmem_max
```

```
echo 262143 > /proc/sys/net/core/rmem_default
#2.4.X のカーネルでは次のように設定します
echo 4096 131072 262143 > /proc/sys/net/ipv4/tcp_rmem
echo 4096 13107262143 > /proc/sys/net/ipv4/tcp_wmem
#"RFC2018 TCP 選択確認応答" と "RFC1323 TCP タイムスタンプ" を無効にします
echo 0 > /proc/sys/net/ipv4/tcp_sack
echo 0 > /proc/sys/net/ipv4/tcp_timestamps
#実行時に shm に割り当てられるメモリーの最大量を 2 倍にします
echo "67108864" > /proc/sys/kernel/shmmax
#Linux の仮想メモリー VM サブシステムを改善します
echo "100 1200 128 512 15 5000 500 1884 2" > /proc/sys/vm/bdflush
#sysctl も実行します
sysctl -p /etc/sysctl.conf
-- end -->
```

さらに、/etc/sysctl.conf ファイルを作成し、次の値を追加します。

```
<-- begin
#パケット転送を無効にします
net.ipv4.ip_forward = 0
#ソースルートの検証を有効にします
net.ipv4.conf.default.rp_filter = 1
#magic-sysrq キーを無効にします
kernel.sysrq = 0
fs.file-max=65536
vm.bdflush = 100 1200 128 512 15 5000 500 1884 2
net.ipv4.ip_local_port_range = 1024 65000
net.core.rmem_max= 262143
net.core.rmem_default = 262143
net.ipv4.tcp_rmem = 4096 131072 262143
net.ipv4.tcp_wmem = 4096 131072 262143
net.ipv4.tcp_sack = 0
net.ipv4.tcp_timestamps = 0
```

kernel.shmmax = 67108864

Solaris システムのチューニングの詳細については、『Solaris Tunable Parameters Reference Manual』を参照してください。

x86 上の Solaris のチューニング

x86 上の Solaris を Application Server および HADB 用に調整するときは、次のオプションについて検討します。

- 114 ページの「IP スタックの設定」
- 115 ページの「ファイル記述子」
- 117 ページの「仮想メモリー」

いくつかの値は、利用可能なシステムリソースによって異なります。/etc/system に変更を加えた場合は、マシンを再起動してください。

セマフォーと共有メモリー

/etc/system ファイルに次の行を追加 (または編集) します。

```
set shmsys:shminfo_shmmax=0xffffffff
set shmsys:shminfo_shmseg=128
set semsys:seminfo_semmnu=1024
set semsys:seminfo_semmap=128
set semsys:seminfo_semmni=400
set semsys:seminfo_semmns=1024
```

これらの設定は、セマフォーの数と共有メモリーに影響します。これらは、Application Server よりも、HADB サーバーを実行しているマシンのほうに深く関係します。

ファイル記述子

/etc/system ファイルに次の行を追加 (または編集) します。

```
set rlim_fd_max=65536
set rlim_fd_cur=65536
set sq_max_size=0
set tcp:tcp_conn_hash_size=8192
set autoup=60
set pcisch:pci_stream_buf_enable=0
```

これらの設定は、ファイル記述子に影響します。

IP スタックの設定

/etc/system ファイルに次の行を追加 (または編集) します。

```
set ip:tcp_squeue_wput=1
set ip:tcp_squeue_close=1
set ip:ip_squeue_bind=1
set ip:ip_squeue_worker_wait=10
set ip:ip_squeue_profile=0
```

これらの設定は、IP スタックを調整します。

このファイルに加えた変更がシステムの再起動時に常に維持されるようにするには、システムの再起動時に実行される起動スクリプトのデフォルトの TCP 変数を次のように変更します。

```
ndd -set /dev/tcp tcp_time_wait_interval 60000
ndd -set /dev/tcp tcp_conn_req_max_q 16384
ndd -set /dev/tcp tcp_conn_req_max_q0 16384
ndd -set /dev/tcp tcp_ip_abort_interval 60000
ndd -set /dev/tcp tcp_keepalive_interval 7200000
ndd -set /dev/tcp tcp_rexmit_interval_initial 4000
ndd -set /dev/tcp tcp_rexmit_interval_min 3000
ndd -set /dev/tcp tcp_rexmit_interval_max 10000
ndd -set /dev/tcp tcp_smallest_anon_port 32768
ndd -set /dev/tcp tcp_slow_start_initial 2
ndd -set /dev/tcp tcp_xmit_hiwat 32768
ndd -set /dev/tcp tcp_recv_hiwat 32768
```

Linux プラットフォームのチューニング

Linux で最大のパフォーマンスが得られるように調整するには、次の調整を行う必要があります。

- 115 ページの「ファイル記述子」
- 117 ページの「仮想メモリー」
- 117 ページの「ネットワークインタフェース」
- 117 ページの「ディスク入出力の設定」
- 118 ページの「TCP/IP 設定」

ファイル記述子

ファイル記述子の数をデフォルト値よりも増やす必要がある場合があります。ファイル記述子の数を増やすと、サーバーが高負荷時でもソケットを開くことができるようになり、クライアントから受け取る要求の処理が中止されなくなります。

まず、次のコマンドを使用して、ファイル記述子のシステム制限を確認します。

```
cat /proc/sys/fs/file-max
8192
```

現在の制限は 8192 と示されています。これを 65535 に増やすには、次のコマンドを root として実行します。

```
echo "65535" > /proc/sys/fs/file-max
```

システムを再起動してもこの値を維持するには、これを `/etc/sysctl.conf` に追加し、許可されるオープンファイルの最大数を指定します。

```
fs.file-max = 65535
```

注意: このパラメータは proc.sys.fs.file-max ではありませんので、間違えないようにしてください。

sysctl を使用して変更できる使用可能パラメータを一覧表示するには、次のコマンドを使用します。

```
sysctl -a
```

sysctl.conf ファイルから新しい値を読み込むには、次のコマンドを使用します。

```
sysctl -p /etc/sysctl.conf
```

シェルあたりの制限値を確認および変更するには、次のコマンドを使用します。

```
limit
```

次のように出力されます。

```
cputime         unlimited
filesize        unlimited
datasize        unlimited
stacksize       8192 kbytes
coredumpsize    0 kbytes
memoryuse       unlimited
descriptors     1024
memorylocked    unlimited
maxproc         8146
openfiles       1024
```

openfiles と descriptors には、1024 の制限値が示されています。全ユーザーを対象にこの制限値を 65535 に増やすには、root として/etc/security/limits.conf を編集し、nofile 設定 (ファイル数) エントリを変更または追加します。

```
*         soft    nofile                    65535
*         hard    nofile                    65535
```

「*」の文字は、すべてのユーザーを示すワイルドカードです。代わりに特定のユーザー ID を指定することもできます。

次に、/etc/pam.d/login を編集し、次の行を追加します。

```
session required /lib/security/pam_limits.so
```

Red Hat ではさらに、/etc/pam.d/sshd を編集し、次の行を追加する必要があります。

```
session required /lib/security/pam_limits.so
```

多くのシステムでは、ここまでの手順で十分です。残りの手順を実行する前に、通常のユーザーとしてログインし、結果を試してください。プラグイン可能認証モジュール (PAM) およびセキュアシェル (SSH) の設定方法によっては、残りの手順は不要な場合があります。

仮想メモリー

仮想メモリーの設定を変更するには、`/etc/rc.local` に次のエントリを追加します。

```
echo 100 1200 128 512 15 5000 500 1884 2 > /proc/sys/vm/bdflush
```

詳細については、`bdflush` のマニュアルページを参照してください。

HADB の設定については、第 6 章を参照してください。

ネットワークインタフェース

ネットワークインタフェースを確実に全二重モードで動作させるには、`/etc/rc.local` に次のエントリを追加します。

```
mii-tool -F 100baseTx-FD eth0
```

eth0 は、ネットワークインタフェースカード (NIC) の名前です。

ディスク入出力の設定

▼ SCSI 以外のディスクのディスク入出力パフォーマンスを調整する

1. ディスクの速度をテストします。
   次のコマンドを使用します。
   ```
   /sbin/hdparm -t /dev/hdX
   ```
2. ダイレクトメモリーアクセス (DMA) を有効にします。
   次のコマンドを使用します。
   ```
   /sbin/hdparm -d1 /dev/hdX
   ```
3. `hdparm` コマンドを使用して速度をもう一度確認します。
   DMA がデフォルトで有効にされていなければ、転送速度が大幅に向上している可能性があります。再起動するたびにこの設定が行われるようにするには、`/etc/conf.d/local.start` や `/etc/init.d/rc.local` など、起動スクリプトを呼び出すすべてのファイルに、`/sbin/hdparm -d1 /dev/hdX` という行を追加します。

SCSI ディスクについては、『System Tuning Info for Linux Servers』の「SCSI Tuning」を参照してください。

TCP/IP 設定

▼ TCP/IP 設定を調整する

1 /etc/rc.local に次のエントリを追加します。

```
echo 30 > /proc/sys/net/ipv4/tcp_fin_timeout
echo 60000 > /proc/sys/net/ipv4/tcp_keepalive_time
echo 15000 > /proc/sys/net/ipv4/tcp_keepalive_intvl
echo 0 > /proc/sys/net/ipv4/tcp_window_scaling
```

2 /etc/sysctl.conf に次のエントリを追加します。

```
# Disables packet forwarding
net.ipv4.ip_forward = 0
# Enables source route verification
net.ipv4.conf.default.rp_filter = 1
# Disables the magic-sysrq key
kernel.sysrq = 0
net.ipv4.ip_local_port_range = 1204 65000
net.core.rmem_max = 262140
net.core.rmem_default = 262140
net.ipv4.tcp_rmem = 4096 131072 262140
net.ipv4.tcp_wmem = 4096 131072 262140
net.ipv4.tcp_sack = 0
net.ipv4.tcp_timestamps = 0
net.ipv4.tcp_window_scaling = 0
net.ipv4.tcp_keepalive_time = 60000
net.ipv4.tcp_keepalive_intvl = 15000
net.ipv4.tcp_fin_timeout = 30
```

3 /etc/rc.local の最後のエントリとして次の行を追加します。

```
sysctl -p /etc/sysctl.conf
```

4 システムを再起動します。

5 次のコマンドを使用して、送信バッファーのサイズを大きくします。

```
tcp_recv_hiwat ndd /dev/tcp 8129 32768
```

◆ ◆ ◆ 第 6 章

高可用性のチューニング

次の内容について説明します。

- 119 ページの「HADB のチューニング」
- 131 ページの「高可用性に関する Application Server のチューニング」
- 135 ページの「ロードバランサの設定」

HADB のチューニング

Application Server は、高可用性データベース (HADB) を使用して持続セッション状態データを保存します。パフォーマンスを最適化するには、Application Server の負荷に応じて HADB を調整します。データの量、トランザクションの頻度、および各トランザクションのサイズが HADB のパフォーマンスに影響を及ぼす場合があります。そのため、Application Server のパフォーマンスにもその影響が及ぶ場合があります。

この節の内容は次のとおりです。

- 119 ページの「ディスクの使用」
- 121 ページの「メモリーの割り当て」
- 122 ページの「パフォーマンス」
- 129 ページの「オペレーティングシステム構成」

ディスクの使用

ここでは、HADB データデバイスのサイズの計算方法と、複数のデータデバイスでの個別ディスクの使用について説明します。

HADB データデバイスのサイズの計算

HADB データベースを作成するときに、データデバイスの数と、各データデバイスのサイズを指定します。これらのデバイスは、すべてのユーザーデータを格納でき

る十分な容量にします。さらに、あとの節で説明されている、内部オーバーヘッドに対応するための追加スペースも確保します。

データベースがデバイス空間を使い果たすと、HADB から Application Server にエラーコード 4593 または 4592 が返されます。

注 – これらのエラーメッセージの詳細については、『Sun Java System Application Server Enterprise Edition 8.2 Error Message Reference』を参照してください。

また、HADBによってこれらのメッセージは履歴ファイルにも書き込まれます。この場合、HADB では、データを挿入または更新するクライアント要求はすべてブロックされます。ただし、削除操作は受け付けられます。

HADB では、セッション状態がバイナリデータとして保存されます。セッション状態は直列化され、BLOB (バイナリラージオブジェクト) として保存されます。各 BLOB は、約 7K バイトのチャンクに分割され、各チャンクは、16K バイトのページにデータベース行 (コンテキスト行はタプルまたはレコードと同義) として保存されます。

各行で小さなメモリーオーバーヘッド (約 30 バイト) が生じます。もっともコンパクトな行割り当て (BLOB チャンク) では、1 ページに 2 行が保存されます。内部断片化により、各ページに1 行しか格納されないこともあります。平均では、各ページの 50% にユーザーデータが格納されます。

ノード障害が発生した場合でも利用できるように、HADB では常に、ユーザーデータがレプリケートされます。HADB ノードには、そのノード自体のデータと、ミラーノードからのコピーデータが保存されます。このように、すべてのデータは 2 回保存されます。平均ではノード容量の 50% がユーザーデータで、各ノードはミラーリングされるので、データデバイスの容量は、ユーザーデータの量の 4 倍以上にしてください。

データの再断片化については、HADB では、再断片化操作の実行中に古いバージョンのテーブルと新しいバージョンのテーブルの両方が保持されます。すべてのアプリケーション要求は古いテーブルで実行されますが、同時に新しいテーブルが作成されています。このため、データベースが主に、セッション状態の BLOB データを格納する 1 つの大きなテーブルとして使用される場合は、デバイス空間に必要な容量をさらに 2 倍にします。したがって、稼働中のデータベースにノードを追加し、すべてのノードを使用するためにデータを再断片化する場合は、利用可能なユーザーデータの量の 8 倍の容量が必要になります。

さらに、HADB の内部使用のために予約されているデバイス空間 (`LogBufferSize` の値の 4 倍) も考慮に入れます。HADB では、このディスク領域は高負荷時のログバッファーの一時的な記憶域として使用されます。

データデバイスのサイズの調整

HADB データデバイスのサイズを大きくするには、次のコマンドを使用します。

```
hadbm set TotalDatadeviceSizePerNode
```

このコマンドを実行すると、変更を適用するためにすべてのノードが 1 つずつ再起動されます。このコマンドの使用の詳細については、『Sun Java System Application Server Enterprise Edition 8.2 高可用性 (HA) 管理ガイド』の「HADB の設定」を参照してください。

注 – `hadbm` では、稼働中のデータベースインスタンスにデータデバイスは追加されません。

物理ディスクへの **HADB** ファイルの配置

パフォーマンスを最適にするには、各データデバイスを個別の物理ディスクに割り当てるべきです。これは、複数のデータデバイスを持つノードがある場合、または同じホストに複数のノードがある場合に当てはまります。

異なるノードに属するデバイスは、異なるデバイスに配置します。Red Hat AS 2.1 では、複数のノードに属するデバイスに同じディスクが使用されていると、非同期の入出力を待つために HADB ノードが常に監視されるので、特にこの措置が重要になります。

HADB ノードでは、出力デバイスで通常行われるように、非同期ではなく同期的に情報、警告、およびエラーが履歴ファイルに書き込まれます。そのため、HADB の動作やパフォーマンスが、履歴ファイルへの書き込み時にディスクの待ち時間による影響を受ける可能性があります。この状況が発生した場合は、履歴ファイル内で次のメッセージによって示されます。

```
BEWARE - last flush/fputs took too long
```

この問題を回避するには、HADB 実行可能ファイルと履歴ファイルを、データデバイスとは異なる物理ディスクに保存します。

メモリーの割り当て

ほかのプロセスと共存する場合は特に、HADB に十分なメモリーを割り当てることが不可欠です。

HADB ノードスーパーバイザープロセス (NSUP) では、最後に監視が実行されてから経過した時間が追跡されます。その時間が、指定された最大値 (デフォルトでは 2500 ミリ秒) を超えると、NSUP によってノードが再起動されます。メモリーの獲得を競い合うほかのプロセスがシステムに存在し、スワッピングや複数のページフォルト

が発生した場合に、この状況になる可能性があります。ブロックされたノードが再起動されると、そのノード上のすべてのアクティブトランザクションが中止されます。

Application Server のスループットが低下し、中止またはタイムアウトが要求された場合は、スワッピングが原因ではないことを確認します。Unix システムでスワッピングのアクティビティーを監視するには、次のコマンドを使用します。

```
vmstat -S
```

さらに、HADB の履歴ファイルに次のメッセージがないか探します。このメッセージは HADB ノードが再起動されたときに書き込まれ、M は N より大きい値になります。

```
Process blocked for .M. sec, max block time is .N. sec
```

中止されたトランザクションの存在は、次のエラーメッセージによって示されます。

```
HADB00224: Transaction timed out または HADB00208: Transaction aborted.
```

パフォーマンス

最適なパフォーマンスを得るには、すべての HADB プロセス (`clu_xxx_srv`) が物理メモリーに収まるようにします。ページングやスワッピングが起こらないようにしてください。同じことが、使用される共有メモリーセグメントにも当てはまります。

共有メモリーセグメントのいくつかはサイズを設定できます。それらのセグメントが小さすぎると、パフォーマンスが低下し、ユーザートランザクションが遅延したり、さらには中止されたりします。セグメントが大きすぎると、物理メモリーが無駄になります。

次のパラメータを設定できます。

- 122 ページの「DataBufferPoolSize」
- 124 ページの「LogBufferSize」
- 125 ページの「InternalLogbufferSize」
- 126 ページの「NumberOfLocks」
- 128 ページの「タイムアウト」

DataBufferPoolSize

HADB のデータは、ディスク上に割り当てられたデータデバイスに保存されます。データが処理されるには、そのデータが主メモリーにある必要があります。HADB ノードでは、この用途に共有メモリーの一部が割り当てられます。割り当てられたデータベースバッファーが、処理されるデータに比べて小さいと、多くの処理容量

がディスク入出力によって浪費されます。書き込み集約型の操作を行う (たとえば、頻繁に更新されるセッション状態がある) システムでは、ディスク入出力に使用される処理容量が多すぎて要求処理が阻害されないように、データベースバッファーを十分な大きさにしてください。

データベースバッファーは、ファイルシステムのキャッシュに似ています。パフォーマンスを向上させるには、ディスク読み取り操作を待つ必要がないように、できるだけ容量の大きいキャッシュを使用します。データベース全体の内容がデータベースバッファーに収まるようにすれば、最高のパフォーマンスが得られます。しかし、ほとんどの場合、これは実現不可能です。クライアントアプリケーションの作業セットがバッファーに収まることを目標にします。

また、ディスク入出力も調べます。HADB で多数のディスク読み取り操作が実行される場合は、データベースのバッファー容量が不足していることになります。データベースバッファーは、ディスクで使用されるブロックサイズと同じ 16K バイトのブロックに分割されます。HADB では、1 回の入出力操作で読み取り用と書き込み用に複数のブロックがスケジュールされます。

ディスクの使用状況を調べるには、`hadbm deviceinfo` コマンドを使用します。たとえば、**`hadbm deviceinfo --details`** というコマンドを実行すると、次のような出力が表示されます。

```
NodeNo   TotalSize       FreeSize        Usage
0        512             504             1%
1        512             504             1%
```

出力の各列の表記の意味は次のとおりです。

- TotalSize: デバイスのサイズ (M バイト単位)。
- FreeSize: 空きサイズ (M バイト単位)。
- Usage: 使用率 (%)

  リソースの使用率を調べるには、`hadbm resourceinfo` コマンドを使用します。たとえば、次のコマンドでは、データバッファープールの情報が表示されます。

  ```
  %hadbm resourceinfo --databuf
  NodeNo   Avail     Free     Access          Misses          Copy-on-write
  0        32        0        205910260       8342738         400330
  1        32        0        218908192       8642222         403466
  ```

  出力の各列の表記の意味は次のとおりです。

- Avail: バッファーのサイズ (M バイト単位)。
- Free: 空きサイズ。この出力例は、データのボリュームがバッファーよりも大きいことを示しています。つまり、常にバッファー全体が使用されています。
- Access: バッファー内でブロックがアクセスされた回数。

- Misses: キャッシュに見つからなかった (ユーザーがディスク読み取りを待たされた) ブロック要求の数。
- Copy-on-write: ディスクへの書き込み中にブロックが変更された回数。

  システムを適切に調整するには、ミス回数 (それによる読み取り回数) が、書き込み回数に比べてごく少数になるようにします。この例では、ミスの割合が約 4% である (2 億回のアクセスに対して 800 万回のミスがあった) ことが示されています。これらの数値の許容範囲は、クライアントアプリケーションの要件によって異なります。

DataBufferPoolSize のチューニング

データベースバッファーのサイズを変更するには、次のコマンドを使用します。

```
hadbm set DataBufferPoolSize
```

このコマンドを実行すると、変更を有効にするためにすべてのノードが 1 つずつ再起動されます。このコマンドの使用の詳細については、『Sun Java System Application Server Enterprise Edition 8.2 高可用性 (HA) 管理ガイド』の「HADB の設定」を参照してください。

LogBufferSize

HADB では、データの挿入、削除、更新、読み取りなど、データベースを変更するすべての操作が、実行前にログに記録されます。それらの操作が記述されたログレコードは、(タプル) ログバッファーと呼ばれる共有メモリー部分に配置されます。HADB では、これらのログを使用して、トランザクションが中止されたときの取り消し操作、ノードクラッシュ時の復元、およびミラーノード間のレプリケーションが行われます。

ログレコードは、ローカルで処理されてミラーノードに送られるまでは、バッファー内に残っています。ログレコードは、トランザクションの結果 (コミットまたは中止) が確定するまで保持されます。HADB ノードでタプルログの残りが少なくなると、ユーザートランザクションが遅延し、場合によってはタイムアウトになります。

LogBufferSize のチューニング

デフォルト値から開始します。履歴ファイルで `HIGH LOAD` 情報メッセージを探します。関連するメッセージには必ず、`tuple log` または `log` と、発生した内部リソース競合の説明が含まれます。

通常の操作では、ログ領域の使用率は 70 ～ 80% と報告されます。領域再生が「低速」と言われるのはこのためです。HADB では、ノードクラッシュが発生時した場合、復旧するためにできるだけ多くのログデータを必要とします。

ログバッファーのサイズと使用状況に関する情報を表示するには、次のコマンドを使用します。

```
hadbm resourceinfo --logbuf
```

たとえば、次のような出力が表示されます。

```
Node No.     Avail         Free Size
0            44            42
1            44            42
```

出力の各列の表記の意味は次のとおりです。

- Node No.: ノード番号。
- Avail: バッファーのサイズ (M バイト単位)。
- Free Size: 空きサイズ (M バイト単位)。データのボリュームがバッファーよりも大きい場合は、常にバッファー全体が使用されます。

  ログバッファーのサイズを変更するには、次のコマンドを使用します。

  ```
  hadbm set LogbufferSize
  ```

  このコマンドを実行すると、変更を有効にするためにすべてのノードが 1 つずつ再起動されます。このコマンドの使用の詳細については、『Sun Java System Application Server Enterprise Edition 8.2 高可用性 (HA) 管理ガイド』の「HADB の設定」を参照してください。

InternalLogbufferSize

ノード内部ログ (nilog) には、ローカルノードの物理的 (論理的の反対、行レベル) 操作に関する情報が記録されます。たとえば、ディスクブロックの割り当てと解放、および B ツリーブロックの分割があるかどうかに関する情報が提供されます。このバッファーは、共有メモリーに保持され、定期的にディスク (個別のログデバイス) との照合も行われます。このバッファーのページサイズおよび関連付けられているデータデバイスのサイズは 4096 バイトです。

大きい BLOB には、必然的に多くのディスクブロックが割り当てられるので、ノード内部ログの負荷が大きくなります。nilog の各エントリは小さいため、通常はこれが問題になることはありません。

InternalLogbufferSize のチューニング

デフォルト値から開始します。履歴ファイルで HIGH LOAD 情報メッセージを探します。関連するメッセージには、nilog と、発生した内部リソース競合の説明が含まれています。

ノード内部ログバッファーの情報を表示するには、次のコマンドを使用します。

```
hadbm resourceinfo --nilogbuf
```

たとえば、次のような出力が表示されることがあります。

```
Node No.     Avail         Free Size
0            11            11
1            11            11
```

nilog バッファーのサイズを変更するには、次のコマンドを使用します。

```
hadbm set InternalLogbufferSize
```

hadbm を実行すると、変更を有効にするためにすべてのノードが 1 つずつ再起動されます。このコマンドの使用の詳細については、『Sun Java System Application Server Enterprise Edition 8.2 高可用性 (HA) 管理ガイド』の「HADB の設定」を参照してください。

注 – nilog バッファーのサイズが変更されると、関連付けられているログデバイス (データデバイスと同じディレクトリにある) のサイズも変更されます。内部ログバッファーのサイズは、内部ログデバイスと同じサイズにします。hadbm set InternalLogBufferSize コマンドによって、この要件を確実に満たします。このコマンドを実行すると、ノードが停止され、InternalLogBufferSize の値が増やされ、内部ログデバイスが再初期化され、ノードが再起動されます。この一連の操作は、すべてのノードに対して実行されます。

NumberOfLocks

行レベル操作ごとに、データベースのロックが必要です。ロックは、トランザクションがコミットまたはロールバックされるまで保持されます。ロックは行 (BLOB チャンク) レベルで設定されます。つまり、大きなセッション状態には多数のロックが必要になります。ロックは、主ノードとミラーノードの両方の操作に必要です。したがって、1 回のBLOB 操作で、2 つの HADB ノードに同じ数のロックが割り当てられます。

テーブルの再断片化が実行される場合、HADB には追加のロックリソースが必要です。そのため、通常のユーザートランザクションが獲得できるのは、割り当てられたロックのうち半分だけです。

利用可能なロックオブジェクトが HADB ノードにない場合は、ログファイルにエラーが書き込まれます。詳細については、『Sun Java System Application Server Enterprise Edition 8.2 Error Message Reference』の第 14 章「HADB Error Messages」を参照してください。

ロック数の計算

必要なロック数を計算するには、次のパラメータを見積もります。

- セッションデータを要求して HADB に保存する同時ユーザーの数 (ユーザーあたり 1 セッションレコード)
- BLOB セッションの最大サイズ
- 持続性の範囲 (session または modified-session の場合はセッションデータの最大サイズ、modified-attribute の場合は属性の最大数)。セッションデータが変更されるたびに、`setAttribute()` を呼び出す必要があります。

  次のように仮定します。
- *x* を同時ユーザーの最大数とします。つまり、*x* 個のセッションデータレコードが HADB に存在するとします。
- *y* をセッションサイズ (session または modified-session の場合) または属性サイズ (modified-attribute の場合) とします。

  この場合、HADB に書き込まれるレコード数は次のようになります。

  $xy/7000 + 2x$

  挿入、削除、更新、読み取りなどのレコード操作では、1 つのレコードに 1 つのロックが使用されます。

注 – ロックは、主レコードとホットスタンバイレコードの両方で保持されます。したがって、挿入、更新、および削除の操作では、1 回のトランザクションに、レコード数の 2 倍のロック数が必要です。読み取り操作では、主レコードのロックのみが必要です。再断片化と二次インデックスの作成が行われるときには、関係するテーブルのログレコードも、作成されるフラグメントレプリカに送信されます。その場合、1 回のトランザクションに、関係するレコード数の 4 倍のロック数が必要です (すべてのクエリーが、その影響を受けるテーブルに対するものと想定した場合)。

サマリー

再断片化が実行される場合、設定するロック数は次のようになります。

$$N_{locks} = 4x\,(y/7000 + 2) = 2xy/3500 + 2x$$

それ以外の場合、設定するロック数は次のようになります。

$$N_{locks} = 2x\,(y/7000 + 2) = xy/3500 + 4x$$

NumberOfLocks のチューニング

デフォルト値から開始します。Application Server のログファイルで、指示されたエラーコードを持つ例外を探します。再断片化が行われていない通常の操作では、クライアントアプリケーションはロック数の半分しか使用しないこともあります。

割り当てられたロック数と使用中のロック数に関する情報を得るには、次のコマンドを使用します。

```
hadbm resourceinfo --locks
```

たとえば、このコマンドを実行したときに、次のような出力が表示されることがあります。

```
Node No.     Avail            Free           Waits
0            50000            50000          na
1            50000            50000          na
```

- Avail: 利用可能なロックの数。
- Free: 使用中のロックの数。
- Waits: ロックを待っていたトランザクションの数。すべてのロックが利用可能な場合は「na」 (適用外) が示されます。

  ロック数を変更するには、次のコマンドを使用します。

  ```
  hadbm set NumberOfLocks
  ```

  hadbm を実行すると、変更を有効にするためにすべてのノードが 1 つずつ再起動されます。このコマンドの使用の詳細については、『Sun Java System Application Server Enterprise Edition 8.2 高可用性 (HA) 管理ガイド』の「HADB の設定」を参照してください。

タイムアウト

ここでは、パフォーマンスに影響するいくつかのタイムアウト値について説明します。

JDBC 接続プールのタイムアウト

これらの値は、サーバーがタイムアウトになるまでにプールからの接続を待機する時間を制御します。ほとんどの場合はデフォルト値で十分です。チューニングの詳細については、91 ページの「JDBC 接続プールのチューニング」を参照してください。

ロードバランサのタイムアウト

次に、パフォーマンスに影響する可能性がある値のいくつかを示します。

- response-timeout-in-seconds - ロードバランサプラグインが応答を待機する時間。この時間が経過すると、インスタンスのデッド状態が宣言され、クラスタ内の次のインスタンスに処理が継続されます。この値は、最悪 (高負荷) の状況下でのサーバーインスタンスからの要求の最大処理時間に合わせた十分な長さにします。

- 診断プログラム: interval-in-seconds - インスタンスが正常であるかどうかを確認するためにロードバランサが Ping を実行する頻度を指定します。デフォルト値は 30 秒です。response-timeout-in-seconds が最適に調整されていて、サーバーのトラフィックがそれほど多くない場合は、デフォルト値で十分です。
- 診断プログラム: timeout-in-seconds - ロードバランサがインスタンスに対して Ping を実行したあとに処理を待つ時間。デフォルト値は 100 秒です。

  診断プログラムの interval-in-seconds と timeout-in-seconds の値の組み合わせによって、ロードバランサプラグインからサーバーインスタンスに行く追加トラフィックの量が決まります。

ロードバランサプラグインの設定の詳細については、『Sun Java System Application Server Enterprise Edition 8.2 高可用性 (HA) 管理ガイド』の「ロードバランサの設定」を参照してください。

HADB のタイムアウト

sql_client タイムアウト値が、パフォーマンスに影響する場合があります。

オペレーティングシステム構成

次に、オペレーティングシステムの構成について説明します。

セマフォー

セマフォーの数が少なすぎると、HADB に障害が発生し、次のエラーメッセージが表示されることがあります。

```
No space left on device
```

これは、データベースの起動時または実行中に発生することがあります。セマフォーはオペレーティングシステムによってグローバルリソースとして提供されるので、その構成は、HADB だけでなく、ホストで実行されるすべてのプロセスによって決まります。Solaris では、`/etc/system` ファイルを編集することによってセマフォーの設定を構成します。

ノード、NNODES (`--hosts` オプションによって暗黙的に HADB に送信されるノードの数)、ホストあたりの NCONNS 接続 (HADB 構成パラメータの `NumberOfSessions`、デフォルト値は 100) を実行するには、次のセマフォー設定を使用します。

```
set semsys:seminfo_semmap = <default=10> + NNODES
set semsys:seminfo_semmni = <default=10> + NNODES
set semsys:seminfo_semmns = <default=60> + (NNODES * 8)
set semsys:seminfo_semmnu = <default=30> + NNODES + NCONNS
```

1 つのホストで複数のノードを実行する予定の場合は、必ず `semmap = NNODES` を設定します。設定を調べるには、`sysinfo` コマンドと `sysdef` コマンドを使用します。

共有メモリー

共有メモリーの最大サイズを、物理 RAM の合計容量に設定します。さらに、プロセスあたりの共有メモリーセグメントの最大数を、HADB プロセスに対応できるように 6 以上に設定します。ホストで実行されるノードの数に基づいて、システム全体の共有メモリー識別子の数を設定します。

Solaris

Solaris 9 では、カーネルが変更されたため、`hmsys:shminfo_shmseg` 変数は使用されなくなっています。Solaris 8 では、`/etc/system` ファイルに次の設定を追加します。

```
set shmsys:shminfo_shmmax = 0xffffffff
set shmsys:shminfo_shmseg = <default=6>
set shmsys:shminfo_shmmni = <default=100> + (6 * NNODES)
```

デフォルト値は Solaris 8 の値です。デフォルト値かどうかに関係なく、以前の変数値に HADB リソース要件を追加します。

注 – これらの設定を変更したら、ホストを再起動してください。

Linux

共有メモリーを 512M バイトに増やすには、次のコマンドを実行します。

```
echo 536870912 > /proc/sys/kernel/shmmax
echo 536870912 > /proc/sys/kernel/shmall
```

`shmmax` ファイルには 1 つの共有メモリーセグメントの最大サイズが含まれており、`shmall` には利用可能にする共有メモリーの合計サイズが含まれています。

この値は、デフォルト値を使用する標準的な HADB ノードに十分な大きさです。デフォルト値を変更する場合は、これらの値の変更も検討してください。

変更後の設定を永続的なものにするには、Linux マシンの `/etc/rc.local` にこれらの行を追加します。RedHat Linux では、カーネルパラメータを設定する `sysctl.conf` も変更できます。

高可用性に関する Application Server のチューニング

ここでは、Application Server の高可用性機能の設定方法について説明します。この節の内容は次のとおりです。

- 131 ページの「セッションの持続性の頻度のチューニング」
- 132 ページの「セッションの持続性の範囲」
- 133 ページの「セッションサイズ」
- 134 ページの「ステートフルセッション Beans のチェックポイント」
- 134 ページの「JDBC 接続プールの設定」
- Web アプリケーションの記述子設定

高可用性 Web アプリケーションに対して持続セッションデータを保証するために、高可用性データベース (HADB) には、HTTP セッションデータを保存するためのバックエンドストアが用意されています。ただし、HADB のデータの保存と読み取りに関連するオーバーヘッドが生じます。セッション持続の各種スキーマと、それらがパフォーマンスや可用性に与える影響について理解すると、高可用性に関する Application Server の構成時の意思決定に役立ちます。

通常は、アプリケーションサーバーインスタンスの 2 倍の数の HADB ノードを保持します。アプリケーションサーバーインスタンスごとに 2 つの HADB ノードが必要です。

セッションの持続性の頻度のチューニング

Application Server では、セッションデータを HADB に書き込むことによって HTTP セッションの持続性とフェイルオーバーの機能が提供されます。持続性の頻度を指定することにより、サーバーが HADB に書き込む頻度を制御できます。

持続性の頻度は、管理コンソールの「設定」>「*config-name*」>「可用性サービス」(「Web コンテナの可用性」) で指定します。

持続性の頻度は、次のいずれかに設定できます。

- web-method
- time-based

ほかの条件がすべて同じであれば、持続性の頻度に time-based 指定すると、web-method を指定した場合よりもパフォーマンスは向上しますが、可用性は劣ります。これは、セッション状態が、取得間隔 (デフォルトは 60 秒) で指定された間隔で持続性ストア (HADB) に書き込まれるためです。その間隔内にサーバーインスタンスで障害が起きると、セッション情報が HADB に最後に書き込まれたあとにセッション状態に発生したすべての更新が失われます。

Web-method

持続性の頻度に web-method を指定すると、サーバーは、各クライアント要求に応答する前に、HTTP セッション状態を HADB に書き込みます。このため、持続されるデータのサイズに基づく応答時間に影響を及ぼす可能性があります。可用性が重要で、ある程度のパフォーマンス低下は許容できるアプリケーションでは、このモードの持続性の頻度を使用します。

web-method 持続性の頻度の詳細については、『Sun Java System Application Server Enterprise Edition 8.2 高可用性 (HA) 管理ガイド』の「Web コンテナの可用性の設定」を参照してください。

Time-based

持続性の頻度に time-based を指定すると、サーバーは、取得間隔と呼ばれる一定の間隔で持続性ストアにセッション情報を保存します。取得間隔は、「設定」 > 「*config-name* 」 > 「Web コンテナ」 (「マネージャープロパティー」) で指定します。*config-name* は、設定の名前です。デフォルトの取得間隔は 60 秒です。取得間隔が経過するたびに、専用スレッドが起動されて、メモリー内のすべてのセッションの繰り返し処理が行われ、セッションデータが保存されます。

一般に、持続性の頻度に time-based を指定すると、web-method を指定した場合よりも高いパフォーマンスが得られます。これは、クライアントに対するサーバーの応答が、HADB へのセッション情報の保存操作による影響を受けないためです。可用性よりもパフォーマンスが重要な場合は、このモードの持続性の頻度を使用します。

セッションの持続性の範囲

管理コンソールの「設定」 > 「*config-name*」 > 「可用性サービス」 (「Web コンテナの可用性」) で持続性の頻度を指定できますが、同じページで持続性の範囲も指定できます。

各持続性の範囲の詳細な説明については、『Sun Java System Application Server Enterprise Edition 8.2 高可用性 (HA) 管理ガイド』の第 9 章「高可用性 (HA) セッション持続性とフェイルオーバーの設定」を参照してください。

持続性の範囲は、次のいずれかに設定できます。

- session
- modifed-session
- modified-attribute

session

持続性の範囲に session を指定すると、サーバーは、変更されたかどうかに関係なく、セッション全体のデータを HADB に書き込みます。このモードでは、要求ごとにすべてのセッションデータが持続されるので、バックエンドストアには常に最新のセッションデータが保持されますが、パフォーマンスは低くなります。

modified-session

持続性の範囲に modified-session を指定すると、サーバーは、HTTP セッションの状態を調べます。サーバーは、データが変更されている場合にかぎり、セッションデータを HADB に保存します。このモードでは、セッションが変更されている場合にのみ HADB を呼び出してデータを持続するため、session モードよりも高いパフォーマンスが得られます。

modified-attribute

持続性の範囲に modified-attribute を指定すると、属性の相互参照は行われず、アプリケーションは `setAttribute()` と `getAttribute()` を使用して HTTP セッションデータを操作します。この方法で書き込むアプリケーションでは、このセッション範囲の性質を活かして高いパフォーマンスを得ることができます。

セッションサイズ

HTTP セッションサイズがパフォーマンスに与える影響を知っておくことは重要です。パフォーマンスは、持続する必要があるセッションデータのサイズに反比例します。セッションデータは、直列化された形式で HADB に格納されます。そのため、データの直列化、BLOB としての挿入、さらに取得時の直列化復元によるオーバーヘッドが生じます。

24K バイトまでのセッションサイズでは、パフォーマンスは変わらないままであることがテストで判明しています。セッションサイズが 100K バイトを超え、同じバックエンドストアが同じ数の接続で使用される場合は、スループットが 90% 低下します。

HTTP セッションサイズの決定に注意を払うことが重要です。サイズの大きい HTTP セッションオブジェクトを作成する場合は、119 ページの「HADB のチューニング」で説明されているように HADB ノードを計算します。

ステートフルセッション Beans のチェックポイント

チェックポイントでは、ステートフルセッション Bean (SFSB) の状態が HADB に保存されます。これにより、サーバーインスタンスに障害が発生した場合に、SFSB の処理がクラスタ内の別のインスタンスに継続され、Bean の状態が復元されます。チェックポイント操作が行われるデータのサイズと、チェックポイント操作の実行頻度によって、特定のクライアントとの対話での応答時間における追加のオーバーヘッドが決まります。

SFSB のチェックポイントは、さまざまなレベルで有効にできます。

- サーバーインスタンスまたは EJB コンテナ全体
- アプリケーション全体
- 特定の EJB モジュール
- 個々の EJB モジュール内のメソッド単位

  最適なパフォーマンスを得るには、`sun-ejb-jar.xml` ファイルに `<checkpointed-methods>` タグを追加して、Bean の状態を大きく変えるメソッドのみにチェックポイントを指定します。

  詳細については、『Sun Java System Application Server Enterprise Edition 8.2 Developer's Guide』の「Using Session Beans」を参照してください。

JDBC 接続プールの設定

Application Server では、JDBC を使用して HADB データが保存および取得されます。最適なパフォーマンスを得るには、もっとも高速と予測される HADB 読み取り/書き込み操作に対して JDBC 接続プールを設定してください。

JDBC 接続プールは、管理コンソールの「リソース」>「JDBC」>「接続プール」>「*pool-name*」で設定します。接続プールの設定には次の項目があります。

- 初期および最小プールサイズ: プール内で維持される初期および最小接続数 (デフォルトは 8)
- 最大プールサイズ: クライアント要求に応じるために作成できる接続の最大数 (デフォルトは 32)
- プールサイズ変更量: アイドルタイムアウト時間が経過したときに削除する接続数
- アイドルタイムアウト: プールで接続がアイドル状態のままでいられる最長時間 (秒単位)。デフォルトは 300
- 最大待ち時間: 接続がタイムアウトになるまでに、呼び出し側が待つ時間 (ミリ秒単位)

最適なパフォーマンスを得るには、ノードあたりの接続数が 8 ～ 16 のプールを使用します。たとえば、4 つのノードが設定されている場合は、通常プールサイズを 32 に、最大プールサイズを 64 に設定します。「アイドルタイムアウト」と「プールサイズ変更量」の値は、監視統計に基づいて調整します。

最適なパフォーマンスを得るには、次の設定を使用します。

- 接続検証: Required
- 検証方法: meta-data
- トランザクション遮断レベル: repeatable-read

標準属性に加えて、次の 2 つのプロパティーを追加します。

- cacheDatabaseMetaData: **false**
- eliminateRedundantEndTransaction: **true**

プロパティーを追加するには、「プロパティーを追加」ボタンをクリックし、プロパティー名と値を指定して、「保存」をクリックします。

JDBC 接続プールの設定の詳細については、91 ページの「JDBC 接続プールのチューニング」を参照してください。

ロードバランサの設定

Application Server には、クラスタに属する複数のインスタンスに要求の負荷を分散できるロードバランサプラグインが用意されています。ロードバランサの設定の詳細については、『Sun Java System Application Server Enterprise Edition 8.2 高可用性 (HA) 管理ガイド』の「ロードバランサの設定」を参照してください。

注 – 次の各節では、サーバーが、サービス着信要求を効果的に処理できるように調整されていることを前提としています。

診断プログラムの有効化

ロードバランサは、`loadbalancer.xml` ファイル内の `health-checker` 要素で指定された値に基づいて、不健全とマークされたすべての構成済み Application Server インスタンスを定期的にチェックします。診断プログラムを有効にするかどうかは任意です。診断プログラムが有効にされていない場合、不健全なインスタンスの定期的な診断は実行されません。

ロードバランサの診断プログラムメカニズムは、HTTP を使用してアプリケーションサーバーインスタンスと通信します。診断プログラムは、指定された URL に HTTP 要求を送信し、応答を待ちます。インスタンスが正常であるとみなされるためには、HTTP 応答ヘッダー内の状態コードが 100 から 500 の間でなければなりません。

診断プログラムを有効にするには、次のプロパティーを編集します。

- **url**: ロードバランサが健康状態を判断するためにチェックするリスナーの URL を指定します。
- **interval-in-seconds**: インスタンスの診断プログラムを実行する間隔を指定します。デフォルトは 30 秒です。
- **timeout-in-seconds**: 正常とみなされるリスナーが応答を受け取るまでのタイムアウト間隔を指定します。デフォルトは 10 秒です。

サーバーからの標準的な応答に *n* 秒かかり、ピーク負荷時には *m* 秒かかる場合は、次のように、`timeout-in-seconds` プロパティーを *m* + *n* に設定します。

```
<health-checker
url="http://hostname.domain:port"
interval-in-seconds="n"
timeout-in-seconds="m+n"/>
```

詳細については、『Sun Java System Application Server Enterprise Edition 8.2 高可用性 (HA) 管理ガイド』の「ロードバランサの設定」を参照してください。

索引

A
AddrLookups，69

B
B コミットオプション，62

C
cacheDatabaseMetaData，135
CacheEntries，68
CDR (Common Data Representation)，89
CMS コレクタ，99
C コミットオプション，62

D
DataBufferPoolSize，122-124
DNS キャッシュ，68-69
DNS 検索，69, 76

E
EJB コンテナ，57-63
　監視，35, 57
　キャッシュ処理とプール処理，58
　キャッシュ設定，60-61
　チューニング，34, 57-63
　プール設定，59
EJB コンポーネント
　キャッシュのチューニング，38, 39, 60-61
　個別監視，36-37
　コミットオプション，62-63
　スタブ、使用，38-39
　タイプ別のパフォーマンス，38
　トランザクション，41-43
　プールのチューニング，39, 58-59
EJB コンポーネントの先取り，48
eliminateRedundantEndTransaction，135

F
final、メソッド，30

H
HADB，119
　JDBC 接続プール，134
　タイムアウト，128
　データデバイスのサイズ，119
　データベースバッファー，123
　メモリー，121
　履歴ファイル，120
　ロック，126
HitRatio，68
HotSpot，99, 100
HTTP サービス，66
　監視，66
　キープアライブの設定，73
　初期スレッド数，72

HTTP サービス (続き)
スレッド数，72
接続プール，75-76
チューニング，71
バッファー長，73
要求タイムアウト，72
HTTP セッション，33
HTTP ファイルキャッシュ，77-78
最大ファイル数，77
最大有効期間，77
小/中のファイルサイズ，78
ハッシュ初期サイズ，77
HTTP プロトコル，76
HTTP リスナー設定，78-82

I

Idle/Peak/Limit，70
IIOP クライアント認証が必要，86
IIOP メッセージ，88-89
InternalLogbufferSize，125-126
IP スタック，114

J

Java 仮想マシン (JVM)，97
Java コーディングの指針，29-32
Java 直列化，89-90
Java ヒープ，101-104
JAX-RPC，31
JDBC
ヒント，50-51
リソース，42
JDBC 接続プール，91
HADB，134
検証方法，94
「すべての障害ですべての接続を閉じる」チェックボックス，94
接続検証が必要，94
接続検証の設定，93-94
表名，94
JMS
接続，52
JMS (続き)
ヒント，51-52
ローカル vs リモートサービス，63
JSP ファイル，32
再読み込み，57
チューニング，32-34
プリコンパイル，54
jvmstat ユーティリティー，100

L

Lightweight Directory Access Protocol (LDAP)，22
Linux，115, 130
LogBufferSize，120, 124-125
LookupsInProgress，69
LWP，111

M

max-cache-size，61
max-pool-size，59
MaxNewSize，103

N

NameLookups，69
NewRatio，103
NewSize，103
NULL、代入，30
NumberOfLocks，126-128

O

ORB，83-90
IIOP クライアント認証が必要，86
監視，84-85
クライアントのプロパティー，87
最大メッセージ分割サイズ，86
スレッドプール，85
スレッドプール ID，85
総接続数，86

ORB *(続き)*
チューニング，85

R
rlim_fd_cur，109
rlim_fd_max，109

S
Secure Sockets Layer，22
SOAP 添付，31-32
Solaris，130
JDK，99
TCP/IP 設定，109
version 8，33
sq_max_size，109
SSL，22
StringBuffer，29-30
-sun.rmi.dgc.client.gcInterval，101
Survivor 領域の比率の設定，104
System.gc()，101

T
tcp_close_wait_interval，109
tcp_conn_hash_size，110
tcp_conn_req_max_q，110
tcp_conn_req_max_q0，110
tcp_ip_abort_interval，110
TCP/IP 設定，109, 118
tcp_keepalive_interval，110
tcp_recv_hiwat，110
tcp_rexmit_interval_initial，110
tcp_rexmit_interval_max，110
tcp_rexmit_interval_min，110
tcp_slow_start_initial，110
tcp_smallest_anon_port，110
tcp_time_wait_interval，110
tcp_xmit_hiwat，110

U
ulimit，111

V
victim-selection-policy，61

W
Web コンテナ，56
Work Queue Length，70

X
x86，113
XA 対応データソース，42
-Xms，102
-Xmx，102
-XX
+DisableExplicitGC，101
MaxHeapFreeRatio，103
MaxPermSize，101
MinHeapFreeRatio，103

あ
アイドルタイムアウト
EJB キャッシュ，61
EJB プール，59
アクセスログ，71
アクセプタスレッド，78-79
値渡し，40
アプリケーション
アーキテクチャー，21
スケーラビリティー，25
チューニング，29
暗号化，22-23
安全性マージン，25

い
移行、パフォーマンス上の考慮点，82

え
エンティティー Beans，46

お
オープンファイル，111, 116
オペレーティングシステム、チューニング，107-118

か
仮想メモリー，117
ガベージコレクタ，98-100
監視
　EJB コンテナ，35
　HTTP サービス，66
　JDBC 接続プール，91
　ORB，84-85
　トランザクションサービス，64
　ファイルキャッシュ，69-70

き
キープアライブ
　クエリー最大スリープ時間，75
　クエリー平均時間，74-75
　最大接続数，74
　スレッド数，73
　設定，73
　タイムアウト，74
　統計，70
期待，27
キャッシュ
　サーブレットの結果，33
　メッセージ駆動型 Beans，52
キャッシュ処理, EJB コンポーネント，58
キューサイズ、接続プール，76
キューに入れられた接続の合計数，69
共有メモリー，114, 130

く
クライアント ORB のプロパティー，87
クラス変数、共有，32

け
軽量プロセス，111
検証方法、JDBC 接続プール，94

こ
高可用性データベース，119
更新間隔, 読み取り専用 Beans，61
更新期間, 読み取り専用 Beans，48
コーディングの指針，29-32
コードの同期，31
コネクタ接続プール，94
個別ディスク，119, 121
　複数のデータデバイス，119
コミットオプション，62-63
コンテキストファクトリ，87
コンテナ管理による関係，48
コンテナ管理によるトランザクション，41

さ
サーバーのチューニング，53
サーブレット，32
　結果のキャッシュ，33
　チューニング，32-34
再起動回復，65-66
再起動で回復，65-66
最終エージェント最適化 (LAO)，42
サイズ変更量
　EJB キャッシュ，61
　EJB プール，59
最大ファイル数、HTTP ファイルキャッシュ，77

最大保留カウント、接続プール，75
最大メッセージ分割サイズ、ORB，86
最大有効期間、HTTP ファイルキャッシュ，77
削除タイムアウト，61
参照渡し，40-41

し
持続性の範囲，132
持続性の頻度，131
持続セッション状態の保存，119
自動回復，65-66
受信バッファーサイズ、接続プール，76
取得間隔，56
小/中のファイルサイズ、HTTP ファイル
　キャッシュ，78
承認，22
初期スレッド数、HTTP サービス，72
処理要件，20-24
診断プログラム，135

す
ステートフルセッション Beans，46, 134
ステートレスセッション Beans，47
「すべての障害ですべての接続を閉じる」
　チェックボックス、JDBC 接続プール，94
スループット，100
スレッド数、HTTP サービス，72
スレッドプール
　サイジング，88
　チューニング，90
　統計，70, 85
スレッドプール ID、ORB，85
スレッドモデル，111

せ
セキュリティーの考慮事項，22
セキュリティーマネージャー，34
世代別オブジェクトメモリー，98
セッション
　サイズ，133
　持続性の範囲，132
　持続性の頻度，131
　状態、保存，119
　タイムアウト，56
接続検証が必要、JDBC 接続プール，94
接続検証の設定、JDBC 接続プール，93-94
接続ハッシュテーブル，110
接続プール、HTTP サービス
　キューサイズ，76
　最大保留カウント，75
　受信バッファーサイズ，76
　送信バッファーサイズ，76
設定のヒント，33-34
セマフォー，114, 129
選択内容の削除ポリシー，61

そ
送信バッファーサイズ、接続プール，76
総接続数、ORB，86
即応性，100

た
帯域幅，108
タイムアウト、HADB，128
断片化されたメッセージ，89

ち
チェックポイント，134
チェックポイント設定，46
チューニング
　EJB キャッシュ，60-61
　EJB プール，58-59
　JDBC 接続プール，91-94
　Solaris の TCP/IP 設定，109
　アプリケーション，29
　サーバー，53
　スレッドプール，90

直列化，29-32, 89-90
直列化復元，29-32

て
定数，30
ディスク入出力パフォーマンス，117-118
ディスク容量，108
データデバイスのサイズ，119
データベースバッファー，123

と
動的再読み込み、無効，54
トランザクション
　CMT 用の管理，93
　EJB コンポーネント，41-43
　EJB トランザクション属性，42-43
　監視，64
　コネクタ接続プール，94
　遮断レベル，51
　チューニング，65

に
認証，22

ね
ネットワークアドレス，78
ネットワークインタフェース，117

の
ノードスーパーバイザープロセス (NSUP)，121

は
ハードウェアリソース，23
配備
　設定，53
　ヒント，33-34
配列，29
ハッシュ初期サイズ、HTTP ファイル
　キャッシュ，77
ハッシュテーブル、接続，110
バッファー長、HTTP サービス，73

ひ
非武装ゾーン (DMZ)，23
表名、JDBC 接続プール，94

ふ
ファイナライザ、回避，31
ファイル記述子，114, 115
ファイルキャッシュ，69-70, 77-78
ファイルサイズ上限、HTTP ファイルキャッシュ，
　ファイルサイズ上限，78
ファイル転送、HTTP ファイルキャッシュ, ファイ
　ル転送，78
プールサイズ、メッセージ駆動型 Bean，51
フットプリント，100
プリコンパイルされた JSP ファイル，54
プログラミング上の指針，29-32
プロセッサ，107
分散トランザクションロギング、無効，65

へ
平均キュー遅延，69
ベストプラクティス，29
変数、NULL を代入，30

ほ
ポーズ，100

め
メッセージ駆動型 Beans，51
メモリー，108, 121

も
文字列，29-30

ゆ
ユーザー負荷，25

よ
要求処理の設定，71
要求タイムアウト、HTTP サービス，72
容量計画，26
読み取り専用 Beans，47-48
　更新間隔，61
　更新期間，48

り
リモートインタフェースとローカルインタフェース，39-40
履歴ファイル、HADB，120

ろ
ロードバランサ，135
ロガーの設定，55
ログレベル，55
ロック、HADB，126