Cosminexus V11 アプリケーションサーバ システム設計ガイド

解説書

3021-3-J04-50

前書き

■ 対象製品

マニュアル「アプリケーションサーバ & BPM/ESB 基盤 概説」の前書きの対象製品の説明を参照してください。

■ 輸出時の注意

本製品を輸出される場合には、外国為替及び外国貿易法の規制並びに米国輸出管理規則など外国の輸出関連法規をご確認の上、必要な手続きをお取りください。

なお、不明な場合は、弊社担当営業にお問い合わせください。

■ 商標類

HITACHI，Cosminexus，HA モニタ，HiRDB，JP1，OpenTP1，TPBroker，uCosminexus，XDM は，株式会社 日立製作所の商標または登録商標です。

AIX は，世界の多くの国で登録された International Business Machines Corporation の商標です。

AMD は，Advanced Micro Devices, Inc.の商標です。

Intel は，Intel Corporation またはその子会社の商標です。

Linux は，Linus Torvalds 氏の米国およびその他の国における登録商標です。

Microsoft，SQL Server，Windows，Windows Server は，マイクロソフト 企業グループの商標です。

Oracle(R)，Java，MySQL 及び NetSuite は，Oracle，その子会社及び関連会社の米国及びその他の国における登録商標です。

Red Hat, and Red Hat Enterprise Linux are registered trademarks of Red Hat, Inc. in the United States and other countries. Linux(R) is the registered trademark of Linus Torvalds in the U.S. and other countries.

Red Hat，および Red Hat Enterprise Linux は，米国およびその他の国における Red Hat, Inc.の登録商標です。Linux(R)は，米国およびその他の国における Linus Torvalds 氏の登録商標です。

UNIX は，The Open Group の登録商標です。

その他記載の会社名，製品名などは，それぞれの会社の商標もしくは登録商標です。

This product includes software developed by the Apache Software Foundation (http://www.apache.org/).

■ 発行

2024 年 2 月 3021-3-J04-50

■ 著作権

All Rights Reserved. Copyright (C) 2020, 2024, Hitachi, Ltd.

変更内容

変更内容(3021-3-J04-50) uCosminexus Application Server 11-40，uCosminexus Client 11-40，uCosminexus Developer 11-40，uCosminexus Service Architect 11-40，uCosminexus Service Platform 11-40

追加・変更内容	変更個所
JDK17 のサポートに伴い，システム設計を始める前に決めておく項目に，インストールする JDK の検討を追加した。	2.1
ZGC を使用する場合のリソースの見積もりについて説明を追加した。	5.2.1(1)，5.2.5，5.3.1(1)，6.2.1(1)，6.3.1
Explicit ヒープのチューニングについての説明を変更した。	7.3.2(2)，7.11.1
選択できるメモリ管理方式に ZGC を追加した。これに伴い，ZGC についての説明を追加した。	7.1，7.17，7.18，7.19，7.20
Survivor 領域についての説明を変更した。	7.15.5
Concurrent Cleanup についての説明を削除した。	7.15.5，7.15.6，7.15.9
NewRatio および SurvivorRario オプションの説明を変更した。	7.16，7.16.3(3)
マニュアル訂正の内容を反映した。	-

単なる誤字・脱字などはお断りなく訂正しました。

はじめに

このマニュアルをお読みになる際の前提情報については，マニュアル「アプリケーションサーバ & BPM/ESB 基盤 概説」のはじめにの説明を参照してください。

目次

前書き　2
変更内容　4
はじめに　5

1　アプリケーションサーバのシステム設計の目的と流れ　15
1.1　アプリケーションサーバのシステム設計の目的　16
1.2　システム設計の流れ　17
1.2.1　オンライン処理を実行するアプリケーション（J2EE アプリケーション）の場合　17
1.2.2　バッチ処理を実行するアプリケーション（バッチアプリケーション）の場合　18

2　システム設計の準備　20
2.1　システム設計を始める前に決めておくこと　21
2.2　業務の種類を明確にする　22
2.3　使用する機能を検討する（オンライン処理を実行する場合）　23
2.3.1　プロセス構成　23
2.3.2　J2EE サーバの構成　25
2.3.3　J2EE アプリケーションと J2EE コンポーネント　26
2.3.4　J2EE コンテナ　29
2.3.5　J2EE サービス　30
2.3.6　J2EE リソース　31
2.3.7　JPA プロバイダ　32
2.3.8　コンテナ拡張ライブラリ　32
2.4　使用する機能を検討する（バッチ処理を実行する場合）　33
2.4.1　プロセス構成　33
2.4.2　バッチサーバの構成　35
2.4.3　バッチアプリケーション　37
2.4.4　バッチサービス　37
2.4.5　J2EE サービス　38
2.4.6　J2EE リソース　39
2.4.7　コンテナ拡張ライブラリ　39
2.5　システムの目的に応じたアプリケーションの構成を決める（オンライン処理を実行する業務の場合）　40
2.5.1　動作させる J2EE アプリケーションの検討　40
2.5.2　使用するプロセスの検討と必要なソフトウェアの準備　41
2.6　システムの目的に応じたアプリケーションの構成を決める（バッチ処理を実行する業務の場合）　46

2.6.1 動作させるバッチアプリケーションの検討 46
2.6.2 使用するプロセスの検討と必要なソフトウェアの準備 47
2.7 運用方法を検討する 50
2.7.1 JP1 と連携したシステムの運用 50
2.7.2 クラスタソフトウェアと連携したシステムの運用 50

3 システム構成の検討（J2EE アプリケーション実行基盤） 52
3.1 システム構成を検討するときに考慮すること 53
3.1.1 システムの目的と構成 53
3.1.2 システム構成の設計手順 54
3.1.3 システム構成の考え方 60
3.2 システム構成の説明について 64
3.2.1 この章のシステム構成に共通する留意点 64
3.2.2 システム構成図で使用する図の凡例 65
3.3 アプリケーションの構成を検討する 66
3.3.1 アプリケーションの構成とアクセスポイント 66
3.3.2 リソースの種類とリソースアダプタ 71
3.4 クライアントとサーバの構成を検討する 79
3.4.1 サーブレットと JSP をアクセスポイントに使用する構成（Web サーバ連携の場合） 79
3.4.2 サーブレットと JSP をアクセスポイントに使用する構成（NIO HTTP サーバに直接アクセスする場合） 81
3.4.3 Session Bean と Entity Bean をアクセスポイントに使用する構成 83
3.4.4 CTM を使用する場合に Stateless Session Bean をアクセスポイントに使用する構成 85
3.5 サーバ間での連携を検討する 88
3.5.1 Session Bean と Entity Bean を呼び出すサーバ間連携 88
3.5.2 CTM 経由で Stateless Session Bean を呼び出すサーバ間連携 91
3.6 トランザクションの種類を検討する 94
3.6.1 ローカルトランザクションを使用する場合の構成 94
3.6.2 グローバルトランザクションを使用する場合の構成 96
3.6.3 トランザクションコンテキストのプロパゲーションを使用する場合の構成 99
3.7 ロードバランスクラスタによる負荷分散方式を検討する 101
3.7.1 Web サーバ連携時の負荷分散機を利用した負荷分散（サーブレット／JSP の場合） 101
3.7.2 NIO HTTP サーバ使用時の負荷分散機を利用した負荷分散（Web サーバを経由しない場合） 103
3.7.3 CORBA ネーミングサービスを利用した負荷分散（Session Bean と Entity Bean の場合） 105
3.7.4 CTM を利用した負荷分散（Stateless Session Bean の場合） 107
3.8 サーバ間で非同期通信をする場合の構成を検討する 112
3.8.1 Message-driven Bean をアクセスポイントに使用する場合の構成（CJMS プロバイダを使用する場合） 112
3.8.2 Message-driven Bean をアクセスポイントに使用する場合の構成（TP1/Message Queue を使用する場合） 114

3.8.3 Message-driven Bean をアクセスポイントに使用する場合の構成（Reliable Messaging を使用する場合） 117
3.8.4 Message-driven Bean のインスタンスプールを利用した負荷分散（TP1/Message Queue を使用する場合） 119
3.9 運用管理プロセスの配置を検討する 122
3.9.1 運用管理サーバに Management Server を配置する構成 122
3.9.2 マシン単位に Management Server を配置する構成 124
3.9.3 コマンドで運用する場合の構成 126
3.10 セッション情報の引き継ぎを検討する 127
3.10.1 データベースを使用する構成（データベースセッションフェイルオーバ機能） 127
3.11 クラスタソフトウェアを使用した障害時の系切り替えを検討する 130
3.11.1 アプリケーションサーバの実行系と待機系を 1:1 にする構成（トランザクションサービスを使用しない場合） 131
3.11.2 アプリケーションサーバの実行系と待機系を 1:1 にする構成（トランザクションサービスを使用する場合） 134
3.11.3 運用管理サーバの実行系と待機系を 1:1 にする構成 136
3.11.4 アプリケーションサーバの実行系と待機系を相互スタンバイにする構成 138
3.11.5 リカバリ専用サーバを使用する場合の構成（N:1 リカバリシステム） 142
3.11.6 ホスト単位管理モデルの実行系と待機系を N：1 にする構成 147
3.12 性能解析トレースファイルを出力するプロセスを配置する 151
3.12.1 システム構成の特徴 151
3.12.2 それぞれのマシンに必要なソフトウェアと起動するプロセス 152
3.13 アプリケーションサーバ以外の製品との連携を検討する 153
3.13.1 JP1 を使用して運用する場合の構成 153
3.13.2 TP1 インバウンド連携機能を使用して OpenTP1 の SUP から Message-driven Bean を呼び出す場合の構成 153
3.13.3 CTM ゲートウェイ機能を利用して EJB クライアント以外から Stateless Session Bean を呼び出す構成 155
3.14 任意のプロセスを運用管理の対象にする 157
3.14.1 システム構成の特徴 157
3.14.2 それぞれのマシンで起動するプロセス 158
3.15 アプリケーションサーバのプロセスが使用する TCP/UDP のポート番号 160

4 システム構成の検討（バッチアプリケーション実行基盤） 168
4.1 システム構成を検討するときに考慮すること 169
4.1.1 システムの目的と構成 169
4.1.2 システム構成の設計手順 170
4.1.3 バッチアプリケーションを実行するシステムで使用する TCP/UDP のポートについての注意事項 172
4.2 バッチサーバを使用する場合のシステム構成 174
4.2.1 バッチアプリケーションのスケジューリング機能を使用しないシステムのシステム構成 174
4.2.2 バッチアプリケーションのスケジューリング機能を使用するシステムのシステム構成 176

5　使用するリソースの見積もり（J2EE アプリケーション実行基盤）　180
5.1　システム構成ごとに使用するリソース　181
5.1.1　Web サーバと J2EE サーバを同じマシンに配置する場合の使用リソース　181
5.1.2　Web サーバと J2EE サーバを別のマシンに配置する場合の使用リソース　183
5.1.3　データベースの使用リソース　186
5.1.4　運用管理サーバの使用リソース　187
5.1.5　CTM を使用する場合の使用リソース　188
5.2　プロセスごとに使用するリソース　193
5.2.1　J2EE サーバが使用するリソースの見積もり　193
5.2.2　運用管理エージェントが使用するリソースの見積もり　200
5.2.3　パフォーマンストレーサが使用するリソースの見積もり　202
5.2.4　CTM が使用するリソースの見積もり　205
5.2.5　cjclstartap プロセスの見積もり　210
5.3　プロセスごとに使用するメモリの見積もり　211
5.3.1　J2EE サーバが使用する仮想メモリの使用量の見積もり　211
5.3.2　CTM のデーモンプロセスが使用するメモリの使用量の見積もり　213

6　使用するリソースの見積もり（バッチアプリケーション実行基盤）　216
6.1　システム構成ごとに使用するリソース　217
6.1.1　バッチサーバを配置する場合の使用リソース　217
6.1.2　データベースの使用リソース　219
6.1.3　CTM を使用する場合の使用リソース　220
6.2　プロセスごとに使用するリソース　225
6.2.1　バッチサーバが使用するリソースの見積もり　225
6.2.2　運用管理エージェントが使用するリソースの見積もり　227
6.2.3　パフォーマンストレーサが使用するリソースの見積もり　227
6.2.4　CTM が使用するリソースの見積もり　227
6.3　仮想メモリの使用量の見積もり　228
6.3.1　仮想メモリの使用量の計算式　228
6.3.2　仮想メモリの使用量を計算する場合の注意事項　229

7　JavaVM のメモリチューニング　231
7.1　GC と JavaVM のメモリ管理の概要　232
7.2　SerialGC の仕組み　234
7.2.1　SerialGC の概要　234
7.2.2　オブジェクトの寿命と年齢の関係　236
7.2.3　CopyGC の仕組み　236
7.2.4　オブジェクトの退避　238
7.2.5　GC 対象外の領域（明示管理ヒープ機能を使用した Explicit ヒープ領域の利用）　239

7.2.6 SerialGC 使用時の JavaVM で使用するメモリ空間の構成と JavaVM オプション 240
7.2.7 GC の発生とメモリ空間の関係 246
7.3 FullGC 発生を抑止するためのチューニングの概要 248
7.3.1 チューニングの考え方 248
7.3.2 チューニング手順 249
7.4 Java ヒープのチューニング 254
7.4.1 Java ヒープのメモリサイズの見積もり 254
7.4.2 Java ヒープのメモリサイズの設定方法 255
7.4.3 Java ヒープのメモリサイズの使用状況の確認方法 256
7.5 Java ヒープ内の Tenured 領域のメモリサイズの見積もり 258
7.5.1 アプリケーションで必要なメモリサイズの算出 258
7.5.2 Java ヒープ内の New 領域のメモリサイズを追加する理由 258
7.6 Java ヒープ内の New 領域のメモリサイズの見積もり 261
7.6.1 Java ヒープ内の Survivor 領域のメモリサイズの見積もり 261
7.6.2 Java ヒープ内の Eden 領域のメモリサイズの見積もり 265
7.7 Java ヒープ内に一定期間存在するオブジェクトの扱いの検討 266
7.7.1 Java ヒープ内の New 領域に格納する方法 266
7.7.2 Java ヒープ内の Tenured 領域に格納する方法 267
7.7.3 Explicit ヒープに格納する方法 267
7.8 Java ヒープの最大サイズ／初期サイズの決定 268
7.9 Java ヒープ内の Metaspace 領域のメモリサイズの見積もり 269
7.10 拡張 verbosegc 情報を使用した FullGC の要因の分析方法 271
7.10.1 拡張 verbosegc 情報の出力形式の概要 271
7.10.2 FullGC 発生時の拡張 verbosegc 情報の出力例 271
7.11 Explicit ヒープのチューニング 277
7.11.1 Explicit ヒープのメモリサイズの見積もり（J2EE サーバが使用するメモリサイズの見積もり） 277
7.11.2 HTTP セッションに関するオブジェクトで利用するメモリサイズ 277
7.11.3 明示管理ヒープ機能利用によるメモリサイズの見積もりへの影響 280
7.11.4 稼働情報による見積もり 281
7.12 アプリケーションで明示管理ヒープ機能を使用する場合のメモリサイズの見積もり 287
7.12.1 アプリケーションで明示管理ヒープ機能を使用するかどうかの検討 287
7.12.2 見積もりの考え方 287
7.12.3 アプリケーションが使用するメモリサイズ 288
7.13 明示管理ヒープ機能の自動配置機能を使用した Explicit ヒープの利用の検討 291
7.13.1 アプリケーション内に Tenured 領域の増加原因のオブジェクトがある場合 291
7.13.2 Tenured 領域利用済みサイズの増加原因が不明な場合 292
7.14 明示管理ヒープ機能適用時に発生しやすい問題とその解決方法 295
7.14.1 Explicit ヒープのある時点での利用状況（スナップショット）の調査 295
7.14.2 利用状況の推移の調査 296

7.14.3 Explicit ヒープあふれが発生した場合の確認と対処 298
7.14.4 Explicit メモリブロックの初期化が失敗した場合の確認と対処 300
7.14.5 Explicit メモリブロック明示解放処理時に Java ヒープへのオブジェクト移動が発生した場合の確認と対処 302
7.14.6 Explicit メモリブロックの自動解放処理が長時間化した場合の確認と対処 304
7.15 G1GC の仕組み 309
7.15.1 G1GC の概要 309
7.15.2 オブジェクトの寿命と年齢の関係 311
7.15.3 New 領域を対象とした GC の仕組み 312
7.15.4 オブジェクトの退避 313
7.15.5 メモリ空間とリージョンの関係 314
7.15.6 リージョンの使われ方 316
7.15.7 G1GC で実行される GC 317
7.15.8 YoungGC 321
7.15.9 Concurrent Marking（CM） 324
7.15.10 MixedGC 327
7.15.11 FullGC 329
7.16 G1GC のチューニング 332
7.16.1 チューニングの流れ 333
7.16.2 初期検証 335
7.16.3 FullGC の発生を抑止するチューニング 337
7.16.4 最悪レスポンス時間を短くするチューニング 341
7.16.5 スループットを向上させるチューニング 343
7.17 ZGC の仕組み（JDK17 以降の場合） 345
7.17.1 ZGC の概要 345
7.17.2 ZGC の用語 345
7.17.3 JavaVM で使用するメモリ空間の構成 345
7.17.4 ZGC サイクル 346
7.17.5 Java ヒープ領域のメモリ管理方法 347
7.17.6 ZGC 独自の GC 発生要因 349
7.18 ZGC のチューニング（JDK17 以降の場合） 351
7.18.1 ZGC での GC の考え方 351
7.18.2 チューニングの流れ 351
7.18.3 GC 停止時間およびスループットの改善方法 352
7.18.4 OS のパラメタ設定（Linux の場合） 352
7.19 ZGC 使用時の他機能への影響（JDK17 以降の場合） 353
7.19.1 使用が制限される機能 353
7.19.2 出力内容が変更される機能 354
7.20 ZGC 使用時の注意事項（JDK17 以降の場合） 355

8　パフォーマンスチューニング（J2EE アプリケーション実行基盤）　356
8.1　パフォーマンスチューニングで考慮すること　357
8.1.1　パフォーマンスチューニングの観点　357
8.1.2　チューニング手順　359
8.1.3　アプリケーションの種類ごとにチューニングできる項目　360
8.2　チューニングの方法　362
8.2.1　J2EE サーバおよび Web サーバのチューニング　362
8.2.2　アプリケーションまたはリソースのチューニング　362
8.2.3　CTM の動作のチューニング　362
8.2.4　それ以外の項目のチューニング　362
8.3　同時実行数を最適化する　363
8.3.1　同時実行数制御および実行待ちキュー制御の考え方　363
8.3.2　最大同時実行数と実行待ちキューを求める手順　365
8.3.3　リクエスト処理スレッド数を制御する　366
8.3.4　Web アプリケーションの同時実行数を制御する　367
8.3.5　Enterprise Bean の同時実行数を制御する　370
8.3.6　CTM を使用して同時実行数を制御する　373
8.3.7　同時実行数を最適化するためのチューニングパラメタ　374
8.4　Enterprise Bean の呼び出し方法を最適化する　380
8.4.1　ローカルインタフェースを使用する　380
8.4.2　リモートインタフェースのローカル呼び出し最適化機能を使用する　381
8.4.3　リモートインタフェースの参照渡し機能を使用する　381
8.4.4　Enterprise Bean の呼び出し方法を最適化するためのチューニングパラメタ　382
8.5　データベースへのアクセス方法を最適化する　384
8.5.1　コネクションプーリングを使用する　384
8.5.2　ステートメントプーリングを使用する　388
8.5.3　データベースへのアクセス方法を最適化するためのチューニングパラメタ　390
8.6　タイムアウトを設定する　393
8.6.1　タイムアウトが設定できるポイント　393
8.6.2　Web フロントシステムでのタイムアウトを設定する　398
8.6.3　バックシステムでのタイムアウトを設定する　400
8.6.4　トランザクションタイムアウトを設定する　402
8.6.5　DB Connector でのタイムアウトを設定する　404
8.6.6　データベースでのタイムアウトを設定する　404
8.6.7　J2EE アプリケーションのメソッドタイムアウトを設定する　410
8.6.8　タイムアウトを設定するチューニングパラメタ　413
8.7　Web アプリケーションの動作を最適化する　422
8.7.1　静的コンテンツと Web アプリケーションの配置を切り分ける　422
8.7.2　静的コンテンツをキャッシュする　425

8.7.3　Web アプリケーションの動作を最適化するためのチューニングパラメタ　427
8.8　CTM の動作を最適化する　430
8.8.1　CTM ドメインマネジャおよび CTM デーモンの稼働状態の監視間隔をチューニングする　430
8.8.2　負荷状況監視間隔をチューニングする　431
8.8.3　CTM デーモンのタイムアウト閉塞を設定する　431
8.8.4　CTM で振り分けるリクエストの優先順位を設定する　432
8.8.5　CTM の動作を最適化するチューニングパラメタ　432

9　パフォーマンスチューニング（バッチアプリケーション実行基盤）　435

9.1　パフォーマンスチューニングで考慮すること　436
9.1.1　パフォーマンスチューニングの観点　436
9.1.2　チューニング手順　437
9.1.3　チューニング項目　437
9.2　チューニングの方法　439
9.2.1　バッチサーバのチューニング　439
9.2.2　リソースのチューニング　439
9.3　タイムアウトを設定する　440
9.3.1　タイムアウトが設定できるポイント　440
9.3.2　トランザクションタイムアウトを設定する　443
9.3.3　タイムアウトを設定するチューニングパラメタ　444
9.4　GC 制御で使用するしきい値を設定する　447
9.4.1　しきい値を設定する目的　447
9.4.2　しきい値設定の考え方　448
9.4.3　GC 制御で使用するしきい値を設定するためのチューニングパラメタ　450

付録　451

付録 A　HTTP セッションで利用する Explicit ヒープの効率的な利用　452
付録 A.1　HTTP セッションに格納するオブジェクトの寿命を考慮する　452
付録 A.2　HTTP セッションに格納するオブジェクトの更新頻度を考慮する　455
付録 A.3　HTTP セッションを作成するタイミングを考慮する　458
付録 B　Explicit ヒープに配置するオブジェクトの寿命による明示管理ヒープ機能への影響　460
付録 B.1　Explicit メモリブロックの自動解放処理への影響　460
付録 B.2　Explicit ヒープのメモリ使用量への影響　462
付録 B.3　Explicit ヒープに配置するオブジェクトの参照関係と寿命との関係　462
付録 C　推奨手順以外の方法でパフォーマンスチューニングをする場合のチューニングパラメタ　464
付録 C.1　同時実行数を最適化するためのチューニングパラメタ（推奨手順以外の方法）　464
付録 C.2　Enterprise Bean の呼び出し方法を最適化するためのチューニングパラメタ（推奨手順以外の方法）　467
付録 C.3　データベースへのアクセス方法を最適化するためのチューニングパラメタ（推奨手順以外の方法）468
付録 C.4　タイムアウトを設定するチューニングパラメタ（推奨手順以外の方法）　468

付録 C.5　Web アプリケーションの動作を最適化するためのチューニングパラメタ（推奨手順以外の場合）　470
付録 C.6　CTM の動作を最適化するチューニングパラメタ（推奨手順以外の方法）　471
付録 C.7　バッチサーバのフルガーベージコレクションを発生させるしきい値を設定するためのチューニングパラメタ（推奨手順以外の方法）　473
付録 D　用語解説　475

索引　476

1 アプリケーションサーバのシステム設計の目的と流れ

この章では，アプリケーションサーバのシステム設計の目的と，検討する必要がある項目について説明します。また，アプリケーションサーバのシステム設計の流れについても説明します。

1.1 アプリケーションサーバのシステム設計の目的

アプリケーションサーバは，Java や CORBA などの業界標準に準拠したアプリケーションの実行環境である，**アプリケーションサーバ**を構築する製品です。アプリケーションサーバは，業務システムの構築基盤として位置づけられます。

業務システムには，次のような要件が求められます。

- **信頼性と可用性の高いシステムの実現**
  業務システムを止めることなく安定稼働させるために，信頼性と可用性を確保したシステムであることが必要です。
- **セキュリティの確保**
  システムを管理または運用するユーザがシステムを構築・運用していく過程や，システムが提供するサービスをエンドユーザが利用していく過程で，システムにはセキュリティ上のさまざまな脅威が想定されます。脅威からシステムを守るには，システムを物理的に安全な構成に設計したり，作業者の運用ルールを定めたりするなどの対策を実施する必要があります。
- **優れた処理性能の実現**
  Web クライアントなどの多数のクライアントからの処理要求や，EJB クライアントなどの業務のバックシステムでのミッションクリティカルな処理要求などに，迅速かつ確実に対応するレスポンスが求められます。

これらの要件を満たすシステムを構築するためには，システム構築を始める前にシステムの目的や特徴を分析したり，システムで使用するリソースを見積もったりするなど，最適なシステム構成を検討する必要があります。また，システムの運用を開始する前に，セキュリティを確保するための手順を整備したり，実際に想定される運用時の状態で動作確認およびチューニングを実施したりする必要があります。

アプリケーションサーバのシステム設計は，これらの作業を通して，アプリケーションサーバ上で動作する業務システムを，最適な状態で運用できるようにすることを目的とします。このマニュアルでは，アプリケーションサーバのシステムを設計する場合に検討，考慮する必要がある項目として，次の項目について説明します。

- システム構成の検討方法
- セキュアなシステムの検討方法
- パフォーマンスチューニングの実施方法

1.2 システム設計の流れ

アプリケーションサーバのシステム設計の流れについて説明します。

システム設計で考慮することは，そのシステムで実行するアプリケーションが，オンライン処理を実行するアプリケーション（J2EE アプリケーション）か，バッチ処理を実行するアプリケーション（バッチアプリケーション）かによって異なります。

それぞれのシステム設計の流れを示します。

1.2.1 オンライン処理を実行するアプリケーション（J2EE アプリケーション）の場合

システム設計は，次の図に示す流れで実行します。

図 1-1 システムの設計の流れ（J2EE アプリケーションを実行する場合）

システム設計の流れ	実施する内容	参照先※
システム設計の準備	システムで使用するアプリケーションサーバの機能を決め，運用方法を決定します。	2章
システム構成の検討	使用する機能ごとに，プロセスを意識しながらシステム構成を検討します。	3章
セキュアなシステムの検討	セキュアなシステムを構築・運用するための考え方を基に，構築・運用手順，監査の方法などを検討します。	マニュアル「アプリケーションサーバ機能解説 セキュリティ管理機能編」
リソースの見積もり	決定したシステム構成で使用するリソースを見積もります。	5章
システムの構築	決定したシステム構成に従って，システムを構築します。	マニュアル「アプリケーションサーバシステム構築・運用ガイド」
パフォーマンスチューニング	実運用に近い負荷を掛けながらシステムを動作させて，パフォーマンスチューニングを実施します。JavaVMのメモリ空間のチューニングも実施します。	7章，8章
システムの運用開始	実運用を開始します。	マニュアル「アプリケーションサーバシステム構築・運用ガイド」

（凡例）
（色付き）：システム設計の工程で検討・実施する項目です。
（白抜き）：システム設計以外の工程で検討・実施する項目です。

注※

それぞれの参照先については，次の表に示した個所を参照してください。

システムの設計	参照先マニュアル	参照個所
システム設計の準備	このマニュアル	2 章
システム構成の検討		3 章
セキュアなシステムの検討	アプリケーションサーバ機能解説 セキュリティ管理機能編	4 章
リソースの見積もり	このマニュアル	5 章
システムの構築	アプリケーションサーバ システム構築・運用ガイド	
パフォーマンスチューニング	このマニュアル	7 章，8 章
システムの運用開始	アプリケーションサーバ システム構築・運用ガイド	

1.2.2 バッチ処理を実行するアプリケーション（バッチアプリケーション）の場合

システム設計は，次の図に示す流れで実行します。

図 1-2 システムの設計の流れ（バッチアプリケーションを実行する場合）

システム設計の流れ	実施する内容	参照先※
システム設計の準備	システムで使用するアプリケーションサーバの機能を決め，運用方法を決定します。	2章
システム構成の検討	使用する機能ごとに，プロセスを意識しながらシステム構成を検討します。	4章
リソースの見積もり	決定したシステム構成で使用するリソースを見積もります。	6章
システムの構築	決定したシステム構成に従って，システムを構築します。	マニュアル「アプリケーションサーバ システム構築・運用ガイド」
パフォーマンスチューニング	システムを動作させて，パフォーマンスチューニングを実施します。JavaVMのメモリ空間のチューニングも実施します。	7章，9章
システムの運用開始	実運用を開始します。	マニュアル「アプリケーションサーバ システム構築・運用ガイド」

（凡例）
：システム設計の工程で検討・実施する項目です。
：システム設計以外の工程で検討・実施する項目です。

注※

それぞれの参照先については，次の表に示した個所を参照してください。

システムの設計	参照先マニュアル	参照個所
システム設計の準備	このマニュアル	2 章
システム構成の検討		4 章
リソースの見積もり		6 章
システムの構築	アプリケーションサーバ システム構築・運用ガイド	4.6
パフォーマンスチューニング	このマニュアル	7 章，9 章
システムの運用開始	アプリケーションサーバ システム構築・運用ガイド	4.6.3

2 システム設計の準備

この章では，システム設計の準備として，システム設計を始める前に決めておくことについて説明します。

2.1 システム設計を始める前に決めておくこと

この節では，アプリケーションサーバのシステム設計を始める前に決めておくことについて説明します。

システム設計作業を始める前に，まず，次のことを明確にしてください。これらの検討結果を踏まえて，3 章以降で説明するシステム構成の検討やパフォーマンスチューニングを実施します。

- 業務の種類を明確にする

  システムで実現する業務の種類を明確にします。業務の種類によって，使用するアプリケーションの種類，構成，および必要なソフトウェアが決まります。

- システムの目的に応じたアプリケーションの構成を決める

  システムの目的に従って，使用する機能を検討した上で，動作させるアプリケーションの構成を明確にします。また，必要なソフトウェアを準備します。

- 運用方法を検討する

  アプリケーションサーバのシステムでは，Management Server という運用管理プロセスを使用して，複数のサーバプロセスを一括して運用します。

  運用方法の検討では，アプリケーションサーバのシステムのほかに，アプリケーションサーバ以外のプログラムを含めたシステム全体を，どのように運用するかを検討します。

- インストールする JDK を検討する

  JDK11 をインストールするか，JDK17 をインストールするかを検討します。

  インストールする JDK によって，アプリケーションサーバで使用できる機能が異なります。目的に合わせて使用する JDK を決定してください。また，インストール時にも次の違いがあります。

  - JDK11 の場合：旧バージョンからの更新インストールおよび新規インストールが可能
  - JDK17 の場合：新規インストールだけが可能

  機能の違いについては，マニュアル「アプリケーションサーバ 機能解説 保守／移行編」の「10.3.3 アプリケーションサーバ 11-00，11-10，11-20，または 11-30 から，11-40 までの仕様変更」を参照してください。

2.2　業務の種類を明確にする

システムでどのような業務を実現するのか，業務の種類を明確にします。

アプリケーションサーバのシステムでは，次の 2 種類の業務を実行できます。

- **オンライン業務**
  ネットワーク経由などで送信されるクライアントからのリクエストを処理する形式の業務です。
- **バッチ業務**
  定型的または定期的な作業をまとめて処理する形式の業務です。

業務の種類によって，実行するアプリケーションの形式や，アプリケーションを実行するサーバプロセスなどが異なります。業務の種類，アプリケーションの形式，およびアプリケーションを実行するサーバプロセスの対応を次の表に示します。

表 2-1　業務の種類，アプリケーションの形式，およびアプリケーションを実行するサーバプロセスの対応

業務の種類	アプリケーションの形式	アプリケーションを実行するサーバプロセス
オンライン業務	J2EE アプリケーション	J2EE サーバ
バッチ業務	バッチアプリケーション	バッチサーバ

ポイント

以降で実施するシステム設計の内容は，業務の種類によって異なります。業務の種類に応じて，必要なシステム設計を実施してください。業務の種類ごとに実行するシステム設計の内容と，このマニュアルでの参照先を次の表に示します。

表 2-2　業務の種類ごとに実行するシステム設計の内容とこのマニュアルでの参照先

システム設計の内容		業務の種類	
		オンライン業務	バッチ業務
システム設計の準備	アプリケーションの構成の決定	2.5	2.6
	運用方法の検討	2.7	
システム構成の検討		3 章	4 章
パフォーマンスチューニング		8 章	9 章
JavaVM のチューニング		7 章，7.4，7.11	

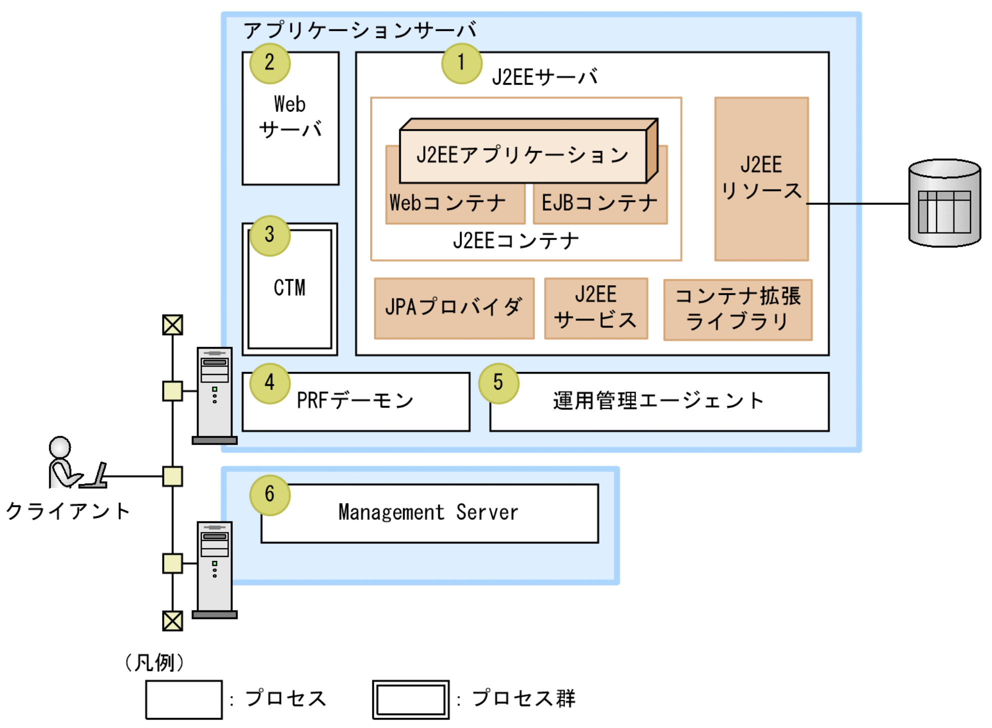
2.3 使用する機能を検討する（オンライン処理を実行する場合）

ここでは，J2EE アプリケーションを実行するアプリケーションサーバのプロセス構成と J2EE サーバの構成について説明します。

2.3.1 プロセス構成

J2EE アプリケーションを実行するアプリケーションサーバは，次の図に示すプロセスで構成されます。

図 2-1 J2EE アプリケーションを実行するアプリケーションサーバを構成するプロセス



> **参考**
>
> なお，システムを構築する場合は，これらのプロセスをシステムの要件に合わせて，システム内の各マシンに一つまたは複数配置します。

それぞれのプロセスについて説明します。なお，図中の番号は，(1)～(6)に対応します。

(1) J2EE サーバ

J2EE サーバは，J2EE アプリケーションの実行基盤となるプロセスです。J2EE サーバは，J2EE アプリケーション，J2EE コンテナ，J2EE サービス，J2EE リソースなど，複数のプログラムモジュールで構成されます。また，J2EE コンテナは，提供する機能によって，EJB コンテナと Web コンテナに分けられます。J2EE サーバを構成するプログラムモジュールについては，「2.3.2 J2EE サーバの構成」で説明します。

(2)　Web サーバ

Web サーバは，Web ブラウザからのリクエスト受信，および Web ブラウザへのデータ送信に関連する処理を実行するプロセスです。J2EE サーバ上で動作する J2EE アプリケーションに Web ブラウザからアクセスするシステムの場合に，Web サーバを使用する必要があります※。なお，Web ブラウザからアクセスできるのは，J2EE アプリケーションに含まれるサーブレット，JSP，または静的コンテンツです。

アプリケーションサーバでは，Web サーバとして，HTTP Server または Microsoft IIS を使用できます。HTTP Server は，アプリケーションサーバの構成ソフトウェアの一つです。HTTP Server の機能については，マニュアル「HTTP Server」を参照してください。

注※

Web サーバを経由しないで J2EE サーバの NIO HTTP サーバと Web ブラウザ間で直接リクエストを送受信する場合，Web サーバに相当するプロセスは不要です。詳細は，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(Web コンテナ)」の「7.　NIO HTTP サーバ」を参照してください。

(3)　CTM

CTM は，J2EE アプリケーション内の Session Bean に対するリクエストをスケジューリングするためのプロセス群です。CTM を使用することで，クライアントからのリクエストを適切に分散，スケジューリングできます。これによって，サーバの負荷を抑え，システムの可用性を高めて業務を滞りなく進めるようにできます。

CTM としての機能は，CTM デーモン，CTM レギュレータ，CTM ドメインマネジャなどの，複数のプロセスを使用して実現します。また，ネーミングサービスとして，CORBA ネーミングサービスを使用します。

CTM の機能の詳細については，マニュアル「アプリケーションサーバ 機能解説 拡張編」の「3.　CTM によるリクエストのスケジューリングと負荷分散」を参照してください。

ポイント

CTM は，構成ソフトウェアに Component Transaction Monitor を含む製品だけで利用できます。利用できる製品については，マニュアル「アプリケーションサーバ & BPM/ESB 基盤 概説」の「2.2　構成ソフトウェア」を参照してください。

(4)　PRF デーモン（パフォーマンストレーサ）

アプリケーションサーバは，リクエストを処理するときに，トレース情報をバッファに出力します。また，アプリケーションサーバだけでなく，トレースの対象を拡張して，アプリケーションでもトレース情報をバッファに出力できます。PRF デーモン（パフォーマンストレーサ）は，バッファに出力されたトレース情報をファイルに出力するための I／O プロセスです。

PRF デーモンが出力するトレース情報ファイルは，システムのボトルネックを検証したり，トラブルシュートの効率向上を図ったりするために役立ちます。

PRF デーモンの機能の詳細については，マニュアル「アプリケーションサーバ 機能解説 保守／移行編」の「7. 性能解析トレースを使用した性能解析」を参照してください。

(5) 運用管理エージェント

運用管理者の代わりに，それぞれのホスト上の論理サーバを起動したり，設定ファイルを更新したりするエージェント機能を持つプロセスです。なお，論理サーバとは，Management Server の運用管理の対象になる，サーバまたはクラスタです。

(6) Management Server

運用管理ドメイン内の各ホストに配置した運用管理エージェントに指示を出して，運用管理ドメイン全体の運用管理を実行するためのプロセスです。

(7) そのほかのプロセス

(1)～(6)で示したプロセス以外に，機能に応じて使用するプロセスとして，次のプロセスがあります。

- **ユーザサーバ**
  ユーザサーバとは，ユーザが定義する任意のサービスやプロセスです。ユーザサーバは論理サーバ（論理ユーザサーバ）として定義できます。論理ユーザサーバとして定義することで，特定のサービスやプロセスが Management Server の管理対象となります。これによって，ほかの論理サーバと同様に，Management Server で一括管理できるようになります。

2.3.2 J2EE サーバの構成

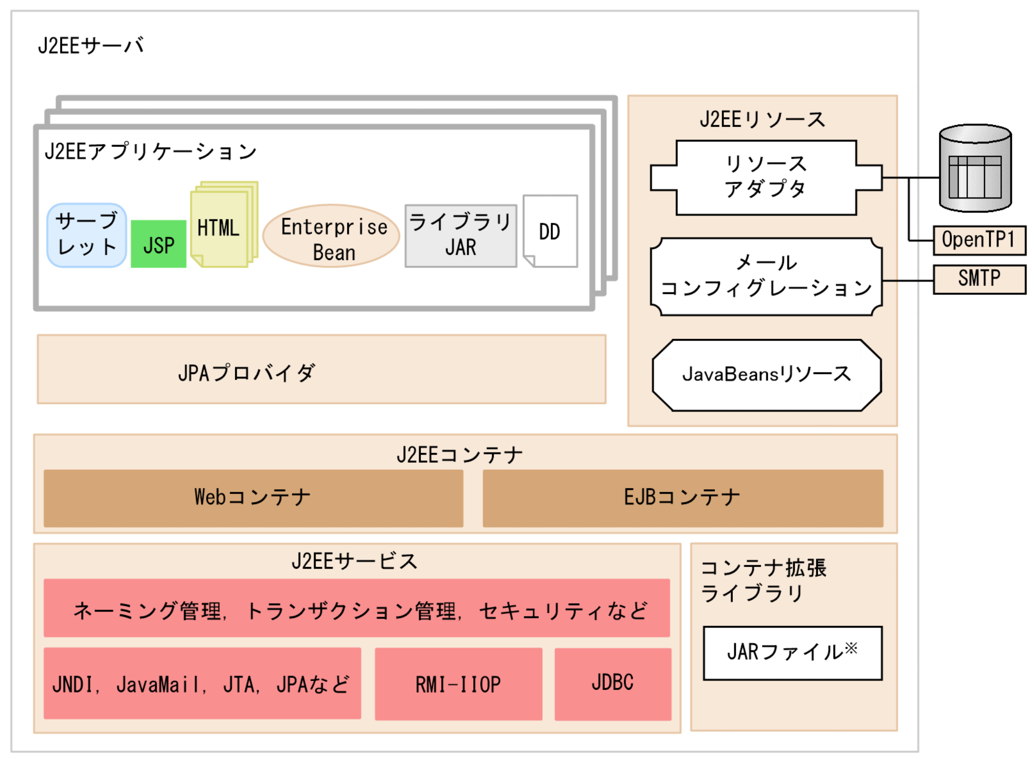
J2EE サーバとは，次に示す六つのプログラムモジュールを実行する Java アプリケーションです。

- J2EE アプリケーション（サーブレット，JSP，Enterprise Bean など）
- J2EE コンテナ
- J2EE サービス
  JNDI，JavaMail，JTA，JPA，RMI-IIOP，JDBC，ネーミング管理，トランザクション管理，セキュリティなど
- J2EE リソース
- JPA プロバイダ
- コンテナ拡張ライブラリ

J2EE アプリケーションは，サーブレット，JSP，Enterprise Bean などによって構成されています。J2EE アプリケーションは，業務の内容に応じて，ユーザが開発します。なお，J2EE アプリケーション以外のプログラムモジュールは，アプリケーションサーバで提供されているモジュールです。

J2EE サーバの構造を次の図に示します。

図 2-2　J2EE サーバの構造



（凡例）

：Application Serverで提供されているプログラムモジュールの範囲です。

注※　JARファイルは使用する機能に応じてユーザが用意します。

以降の項で，J2EE サーバの各モジュールの概要を説明します。

2.3.3　J2EE アプリケーションと J2EE コンポーネント

J2EE アプリケーションは，一つ以上の J2EE コンポーネントで構成されています。ここでは，J2EE アプリケーションと J2EE コンポーネントについて説明します。

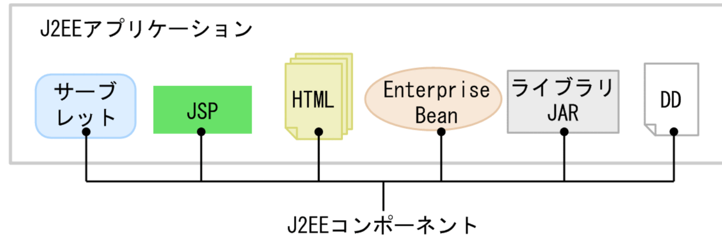
(1)　J2EE アプリケーションと J2EE コンポーネントの関係

J2EE アプリケーションは，サーブレット，JSP，Enterprise Bean などのユーザアプリケーションプログラムで構成されています。J2EE アプリケーションは，J2EE コンテナ上で動作します。

J2EE アプリケーションを構成する，サーブレット，JSP，Enterprise Bean などを J2EE コンポーネントといいます。

J2EE アプリケーションと J2EE コンポーネントの関係を次の図に示します。

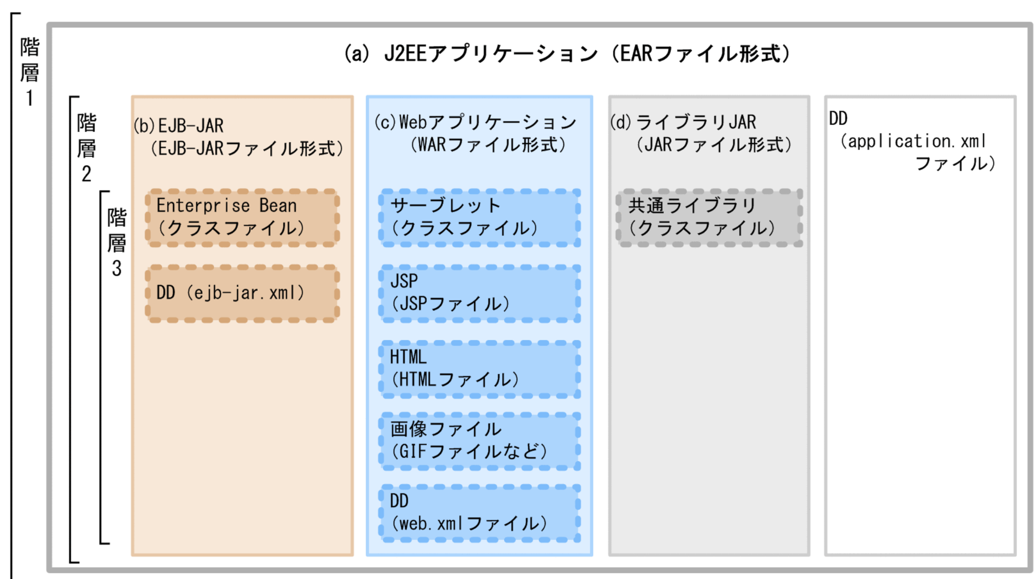
図 2-3　J2EE アプリケーションと J2EE コンポーネントとの関係



(2)　J2EE アプリケーションの構造

J2EE アプリケーションは，3 層の構造になっています。J2EE アプリケーションの構造を次の図に示します。

図 2-4　J2EE アプリケーションの構造



注　図中の項番は，本文中の項番と対応しています。

J2EE アプリケーションの最小単位は，階層 3 のファイル（図中，点線で囲まれたファイル）です。階層 3 のファイルには，クラスファイルや JSP ファイルなどがあります。

そして，階層 3 のファイルをパッケージしたものが階層 2 のファイルとなります。この図の場合，階層 2 の EJB-JAR ファイルは，階層 3 に属する Enterprise Bean と DD（ejb-jar.xml）をパッケージしたものとなります。

さらに，階層 2 のそれぞれのファイルをパッケージしたものが階層 1 の J2EE アプリケーションとなります。

ここでは，階層 1 および階層 2 のパッケージファイルについて説明します。なお，図中の項番は次の説明の項番と対応しています。

参考

それぞれの階層でファイル形式，および DD の DTD が規定されています。

DD とは，アプリケーションを運用環境に配置するときの定義情報を記述したファイルを指します。EJB-JAR の場合，DD は ejb-jar.xml，Web アプリケーションの場合，DD は web.xml，J2EE アプリケーションの場合，DD は application.xml となります。

なお，Enterprise Bean でアノテーションを使用する場合，ejb-jar.xml は不要です。

(a) J2EE アプリケーション

J2EE アプリケーションは，複数の，EJB-JAR，Web アプリケーション，ライブラリ JAR と，一つの DD（application.xml）で構成されます。

J2EE サーバで実行できる J2EE アプリケーションは，アーカイブ形式の J2EE アプリケーション，および展開ディレクトリ形式の J2EE アプリケーションです。

- **アーカイブ形式の J2EE アプリケーション**
  EJB やサーブレットなどのアプリケーションの実体を J2EE サーバの作業ディレクトリに持つ J2EE アプリケーションです。アーカイブ形式の J2EE アプリケーションを J2EE サーバ内にインポートしてクライアントから実行可能な状態にするためには，EAR 形式または ZIP 形式へのアセンブルと，デプロイが必要です。
- **展開ディレクトリ形式の J2EE アプリケーション**
  EJB やサーブレットなどのアプリケーションの実体を，J2EE サーバの外部にある一定のルールに従ったファイル／ディレクトリに持つ J2EE アプリケーションです。展開ディレクトリ形式の J2EE アプリケーションを J2EE サーバ内にインポートしてクライアントから実行可能な状態にするためには，デプロイが必要です。

参考

- **アセンブル**とは，単体では動作しない EJB-JAR をアプリケーションの 1 構成要素として位置づけるための組み立て作業のことです。アセンブルでは，EJB-JAR を 1 構成要素として取り込んだ J2EE アプリケーションを構築します。なお，J2EE アプリケーションの構成要素として，EJB-JAR のほかに WAR ファイル，ライブラリ JAR を含むことがあります。
  展開ディレクトリ形式の J2EE アプリケーションの場合には，EAR 形式または ZIP 形式へのアセンブルは不要です。
- **デプロイ**とは，J2EE アプリケーションをクライアントから実行可能な状態にすることです。

(b) EJB-JAR

EJB-JAR は，EJB-JAR ファイル形式でパッケージ化されています。複数の Enterprise Bean と一つの DD（ejb-jar.xml）で構成されます。なお，Enterprise Bean でアノテーションを使用している場合は，DD（ejb-jar.xml）は不要です。

(c) Web アプリケーション

Web アプリケーションは，WAR ファイル形式でパッケージ化されています。複数のサーブレット，JSP，HTML と一つの DD（web.xml）で構成されます。

(d) ライブラリ JAR

ライブラリ JAR は，JAR ファイル形式でパッケージ化されたものです。複数の共通ライブラリから構成されています。共通ライブラリは J2EE アプリケーション中の J2EE コンポーネントが共通で使用できるライブラリです。J2EE アプリケーションの DD（application.xml）の<module>タグ以下に定義されているファイル以外で，拡張子が小文字（.jar）の JAR ファイルがライブラリ JAR とみなされます。

(3) J2EE アプリケーションおよび J2EE コンポーネントの開発

J2EE アプリケーションでアプリケーションサーバが提供する実行基盤としての機能を使用するためには，機能に応じたアプリケーションの実装が必要です。なお，アプリケーションサーバでは，J2EE アプリケーションおよび J2EE コンポーネントを開発するための製品として，Developer を提供しています。

J2EE アプリケーションおよび J2EE コンポーネントの開発方法については，マニュアル「アプリケーションサーバ アプリケーション開発ガイド」を参照してください。

2.3.4 J2EE コンテナ

J2EE コンテナとは，J2EE アプリケーションを実行するためのサーバ基盤です。J2EE コンテナは，EJB コンテナと Web コンテナで構成されています。

J2EE コンポーネントは Web コンテナおよび EJB コンテナで提供する API を利用して，J2EE コンテナ上で動作します。アプリケーションサーバで提供している Web コンテナおよび EJB コンテナは，Java EE 6 に対応しています。これによって，Java EE に準拠する，本格的な基幹業務アプリケーションを迅速かつ容易に構築できます。

(1) Web コンテナ

Web コンテナは，サーブレットと JSP を実行するためのサーバ基盤です。Web クライアントからアクセスを受け取り，要求に応じたサービスを提供します。

Web コンテナでは，サーブレット，JSP の API を提供しています。

(2)　EJB コンテナ

EJB コンテナは，Enterprise Bean の実行を制御し，Enterprise Bean に各種のサービスを提供するサーバ基盤です。EJB コンテナでは，EJB の API を提供しています。

2.3.5　J2EE サービス

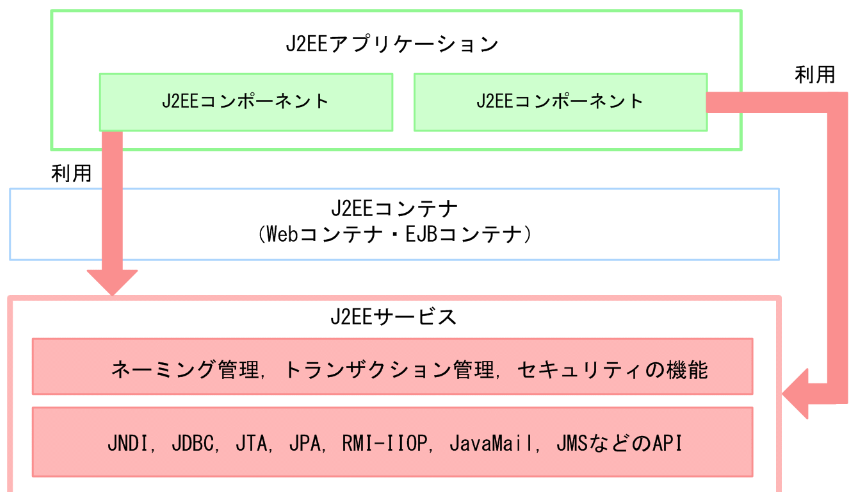
J2EE サービスでは，次に示す機能および API を提供しています。

1. トランザクション管理，セキュリティ管理，およびネーミング管理の機能
2. JNDI，JDBC，JTA，JPA，RMI-IIOP，JavaMail，JMS などの API

J2EE サービスは，J2EE コンテナの部品機能として利用され，J2EE コンポーネントである，サーブレット・JSP，および Enterprise Bean に，機能および API を提供します。J2EE サービスの API は，J2EE コンポーネントによって，直接，または J2EE コンテナ経由で利用されます。

J2EE サービスの位置づけを次の図に示します。

図 2-5　J2EE サービスの位置づけ



J2EE サービスでは，アプリケーションサーバの構成ソフトウェアの機能のほか，アプリケーションサーバ以外の製品の機能も使用します。J2EE サービスを実現する，ソフトウェア製品またはアプリケーションサーバの構成ソフトウェアを次の表に示します。

表 2-3　J2EE サービスを実現する，製品または構成ソフトウェア

分類		製品または構成ソフトウェア
サービス	ネーミング管理	Component Container※ TPBroker※
	トランザクション管理	
	セキュリティ	Component Container※

分類		製品または構成ソフトウェア
API	JNDI	
	JTA	
	JPA	
	JavaMail	
	RMI-IIOP	TPBroker※
	JDBC Standard Extension	HiRDB Type4 JDBC Driver Oracle JDBC Thin Driver SQL Server Driver for JDBC
	JDBC	
	JMS	Reliable Messaging※ TP1/Message Queue - Access

注※　アプリケーションサーバの構成ソフトウェアです。

2.3.6　J2EE リソース

J2EE サーバはリソースとして，データベース，OpenTP1，SMTP サーバ，および JavaBeans リソースを利用できます。J2EE リソースは，これらのリソースと接続するために使用します。

アプリケーションサーバで扱う J2EE リソースには，外部リソースとの接続に利用するリソースアダプタ，およびメールコンフィグレーションがあります。また，このほかに，内部リソースとして利用できる JavaBeans リソースがあります。

- リソースアダプタ
  接続するリソースの種類に応じて，次のリソースアダプタがあります。
  - DB Connector
    データベースとの接続に利用します。
  - DB Connector for Reliable Messaging および Reliable Messaging
    データベース上のキューとの接続に利用します。
  - TP1 Connector
    OpenTP1 の SPP との接続に利用します。
  - TP1/Message Queue - Access
    TP1/Message Queue との接続に利用します。
  - そのほかの Connector 1.0 または Connector 1.5※に準拠したリソースアダプタ
    任意のリソースとの接続に使用します。
- メールコンフィグレーション
  SMTP サーバとの接続に利用します。

- JavaBeans リソース
  内部のリソースとして利用できるリソースです。

J2EE リソースの詳細については，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(コンテナ共通機能)」の「3.　リソース接続とトランザクション管理」を参照してください。

注※　Outbound の通信モデルに対応したリソースアダプタを使用できます。詳細については，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(コンテナ共通機能)」の「3.16.8　Connector 1.5 仕様に準拠したリソースアダプタを使用する場合の設定」を参照してください。

2.3.7　JPA プロバイダ

アプリケーションサーバで提供している JPA 実装を JPA プロバイダといいます。JPA プロバイダを利用することによって，アプリケーションサーバで JPA アプリケーションを実行できます。

JPA プロバイダについては，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(コンテナ共通機能)」の「6.　JPA 2.2 の利用」を参照してください。

2.3.8　コンテナ拡張ライブラリ

Enterprise Bean，サーブレット，JSP が利用する共通のライブラリをコンテナ拡張ライブラリといいます。このライブラリを利用することによって，Enterprise Bean，サーブレット，JSP からユーザ作成の共通のライブラリを呼び出せるようになります。

コンテナ拡張ライブラリについては，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(コンテナ共通機能)」の「19.　コンテナ拡張ライブラリ」を参照してください。

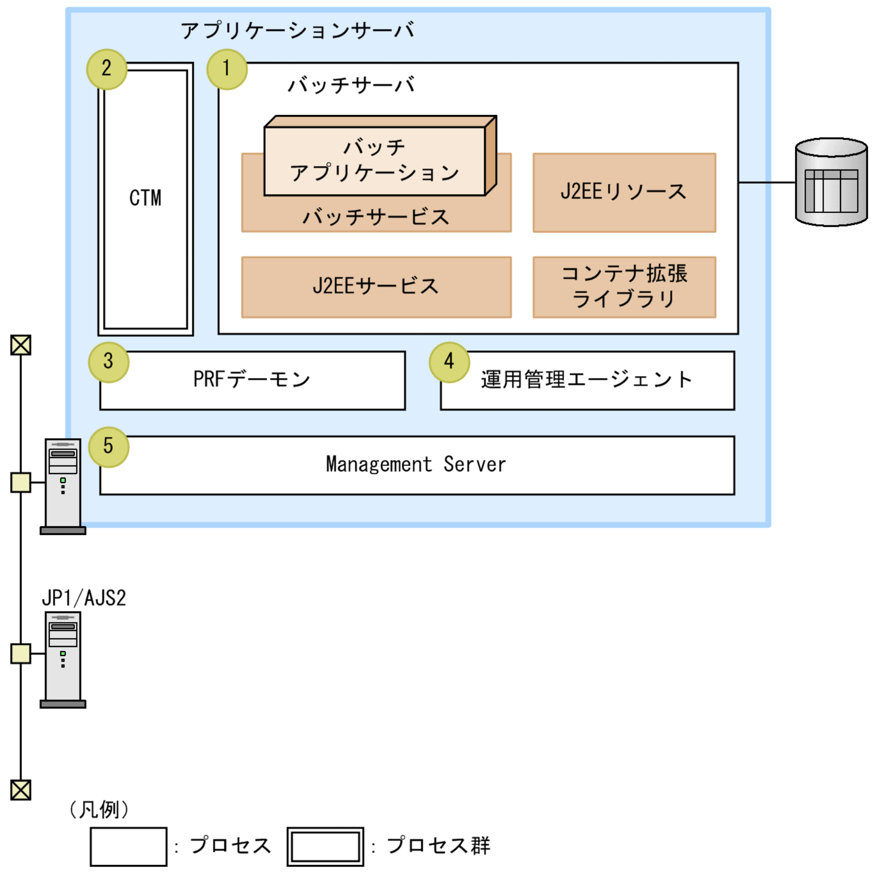
2.4　使用する機能を検討する（バッチ処理を実行する場合）

ここでは，バッチアプリケーションを実行するアプリケーションサーバのプロセス構成と，バッチサーバの構成について説明します。

2.4.1　プロセス構成

バッチアプリケーションを実行するアプリケーションサーバは，次の図に示すプロセスで構成されます。

図 2-6　バッチアプリケーションを実行するアプリケーションサーバを構成するプロセス



> **参考**
>
> システムを構築する場合は，これらのプロセスをシステムの要件に合わせて，システム内の各マシンに一つまたは複数配置します。

それぞれのプロセスについて説明します。なお，図中の番号は，(1)～(5)に対応します。

(1)　バッチサーバ

バッチサーバは，バッチアプリケーションの実行基盤となるプロセスです。バッチサーバは，バッチアプリケーション，バッチサービス，J2EE サービス，J2EE リソースなど，複数のプログラムモジュールで構成されます。バッチサーバを構成するプログラムモジュールについては，「2.4.2　バッチサーバの構成」で説明します。

(2)　CTM

CTM は，バッチアプリケーションの実行をスケジューリングするためのプロセス群です。CTM を使用することで，バッチアプリケーションの実行を適切に分散，スケジューリングできます。これによって，バッチサーバの数を意識することなく，複数のバッチアプリケーションを同時に実行できます。

CTM としての機能は，CTM デーモン，CTM レギュレータ，CTM ドメインマネジャなどの，複数のプロセスを使用して実現します。また，ネーミングサービスとして，CORBA ネーミングサービスを使用します。

CTM の機能の詳細については，マニュアル「アプリケーションサーバ 機能解説 拡張編」の「3.　CTM によるリクエストのスケジューリングと負荷分散」を参照してください。

ポイント

CTM は，構成ソフトウェアに Component Transaction Monitor を含む製品だけで利用できます。利用できる製品については，マニュアル「アプリケーションサーバ & BPM/ESB 基盤 概説」の「2.2　構成ソフトウェア」を参照してください。

(3)　PRF デーモン（パフォーマンストレーサ）

アプリケーションサーバは，トレース情報をバッファに出力します。また，トレースの対象を拡張して，アプリケーションでもトレース情報をバッファに出力できます。PRF デーモン（パフォーマンストレーサ）は，バッファに出力されたトレース情報をファイルに出力するための I／O プロセスです。PRF デーモンが出力するトレース情報ファイルは，システムのボトルネックを検証したり，トラブルシュートの効率向上を図ったりするために役立ちます。

PRF デーモンの機能の詳細については，マニュアル「アプリケーションサーバ 機能解説 保守／移行編」の「7.　性能解析トレースを使用した性能解析」を参照してください。

(4)　運用管理エージェント

運用管理者の代わりに，それぞれのホスト上の論理サーバを起動したり，設定ファイルを更新したりするエージェント機能を持つプロセスです。なお，論理サーバとは，Management Server の運用管理の対象になる，サーバまたはクラスタです。

(5) Management Server

運用管理ドメイン内の各ホストに配置した運用管理エージェントに指示を出して，運用管理ドメイン全体の運用管理を実行するためのプロセスです。

> **参考**
>
> バッチ処理を実行する場合，(1)～(5)で示したプロセス以外に，システムの目的に応じて**ユーザサーバ**というプロセスを使用できます。ユーザサーバとは，ユーザが定義する任意のサービスやプロセスです。ユーザサーバは論理サーバ（論理ユーザサーバ）として定義できます。論理ユーザサーバとして定義することで，特定のサービスやプロセスが Management Server の管理対象となります。これによって，ほかの論理サーバと同様に，Management Server で一括管理できるようになります。

2.4.2 バッチサーバの構成

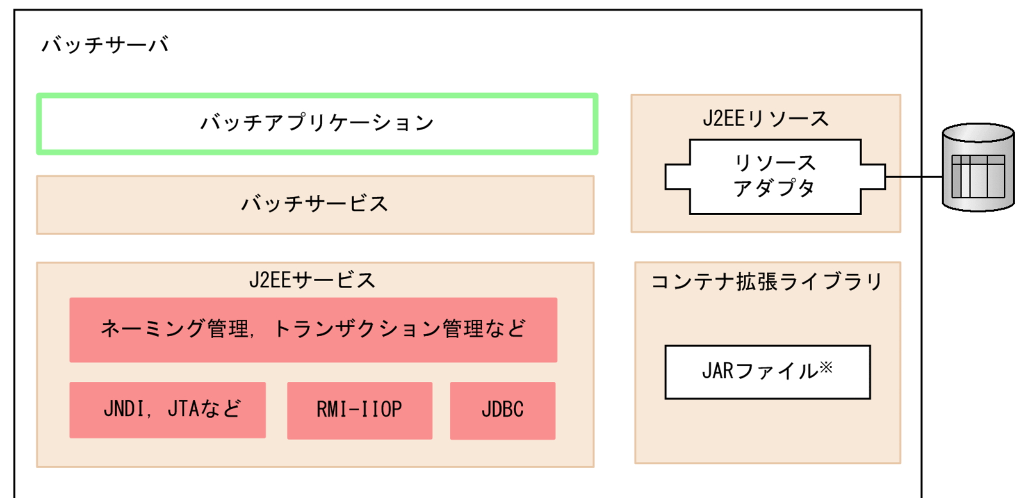
バッチサーバとは，次に示す五つのプログラムモジュールを実行する Java アプリケーションです。

- バッチアプリケーション
- バッチサービス
- J2EE サービス
  JNDI，JTA，RMI-IIOP，JDBC，ネーミング管理，トランザクション管理など
- J2EE リソース
- コンテナ拡張ライブラリ

バッチアプリケーションとは，バッチ処理を実装した Java アプリケーションです。バッチアプリケーションは，業務の内容に応じてユーザが開発します。なお，バッチアプリケーション以外のプログラムモジュールは，アプリケーションサーバで提供されているモジュールです。

バッチサーバの構造を次の図に示します。

図 2-7　バッチサーバの構造



（凡例）

：アプリケーションサーバで提供されているプログラムモジュールの範囲です。

注※　JARファイルは使用する機能に応じてユーザが用意します。

バッチサーバでは次に示す Java EE および J2EE の機能を使用できます。

- JDBC 2.0 コア/JDBC 2.0 オプションパッケージ
- JDBC 3.0※1
- JDBC 4.0※1，※2
- Connector 1.0（DB Connector）※3
- JTA 1.0.1（ただし，local だけ）※4

注※1

接続に使用する JDBC ドライバが，該当するバージョンの仕様で規定された機能をサポートしている必要があります。

注※2

接続できるドライバは，Oracle JDBC Thin Driver だけです。

注※3

トランザクションなし，またはローカルトランザクションの DB Connector を使用できます。

注※4

リソースアダプタの DD（ra.xml）の transaction-support で LocalTransaction を指定し，ビジネスロジック中にリモートで JavaVM の呼び出しをしない場合に，ローカルトランザクションが利用できます。

また，バッチサーバから次の EJB を呼び出せます。ただし，EJB の呼び出し方法はリモート呼び出しとなります。ローカル呼び出しはできません。

- EJB 2.0
- EJB 2.1
- EJB 3.0

以降の項で，バッチサーバの各モジュールの概要を説明します。

2.4.3 バッチアプリケーション

バッチアプリケーションとは，バッチ処理を実装した Java アプリケーションです。バッチアプリケーションは，一つのバッチサーバにつき一つ実行できます。

バッチアプリケーションを開始するには，アプリケーションサーバで提供しているバッチ実行コマンドを使用します。バッチサーバでは，バッチ実行コマンドによるバッチアプリケーションの実行リクエストを受けて，バッチアプリケーションを開始します。JP1/AJS と連携すると，バッチ実行コマンドを JP1 のジョブとして定義できるので，バッチアプリケーションの自動実行ができます。

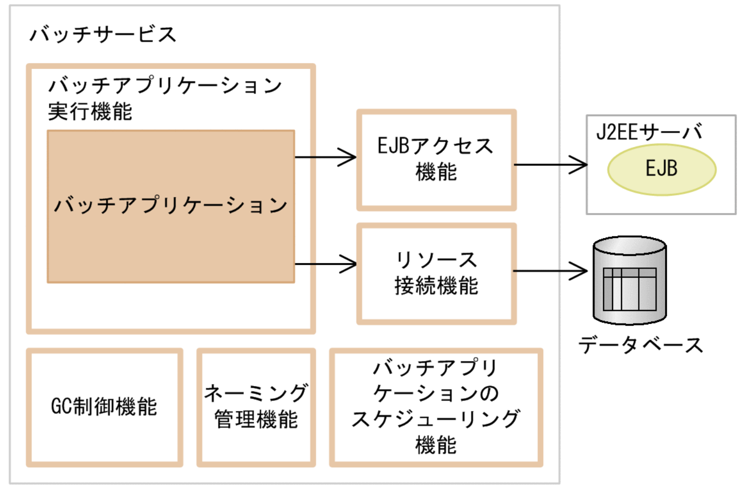
また，バッチアプリケーションのスケジューリング機能を使用すると，バッチアプリケーションの実行リクエストはスケジュールキューによって制御され，自動的にバッチサーバへ振り分けられます。複数のバッチアプリケーションを同時に開始したい場合に，バッチアプリケーションのスケジューリング機能を使用すると，バッチサーバの数や，どのバッチサーバで実行するかを意識する必要がありません。なお，バッチアプリケーションのスケジューリング機能を使用しない場合には，バッチアプリケーションごとにバッチサーバを用意してください。

バッチアプリケーションの詳細については，マニュアル「アプリケーションサーバ 機能解説 拡張編」の「2.　バッチサーバによるアプリケーションの実行」を参照してください。

2.4.4 バッチサービス

バッチアプリケーションを実行するアプリケーションサーバでは，バッチサービスを提供しています。バッチサービスとは，バッチアプリケーションを実行するための機能です。バッチサービスでは，次の図に示す機能を提供しています。

図 2-8　バッチサービスで提供している機能



- バッチアプリケーション実行機能

  バッチアプリケーションを開始したり，強制停止したりするための機能を提供しています。

- EJB アクセス機能

  バッチアプリケーションから J2EE サーバの EJB にアクセスするための機能を提供しています。EJB アクセス機能は J2EE サービスを使用します。

- リソース接続機能

  バッチアプリケーションからデータベースに接続するための機能を提供しています。リソース接続機能は，J2EE サービスおよび J2EE リソースを使用します。

- GC 制御機能

  バッチアプリケーションでリソース排他をしているときに，GC の実行を制御するための機能を提供しています。

- ネーミング管理機能

  EJB またはリソースを参照するときに，名前解決をするための機能を提供しています。ネーミング管理機能は J2EE サービスを使用します。

- バッチアプリケーションのスケジューリング機能

  CTM を使用して，バッチアプリケーションの実行をスケジューリングするための機能を提供しています。

バッチサービスで提供するそれぞれの機能については，マニュアル「アプリケーションサーバ 機能解説 拡張編」の「2.3　バッチアプリケーション実行機能」を参照してください。

2.4.5　J2EE サービス

J2EE サービスでは，次に示す機能および API を提供しています。

1. トランザクション管理，およびネーミング管理の機能

2. JNDI，JDBC，JTA，RMI-IIOP などの API

J2EE サービスは，バッチアプリケーションからデータベースに接続したり，EJB を呼び出したりするときに利用されます。

J2EE サービスでは，アプリケーションサーバの構成ソフトウェアの機能のほか，アプリケーションサーバ以外の製品の機能も使用します。J2EE サービスを実現する，ソフトウェア製品またはアプリケーションサーバの構成ソフトウェアを次の表に示します。

表 2-4　J2EE サービスを実現する，製品または構成ソフトウェア

<table>
<tr><th colspan="2">分類</th><th>製品または構成ソフトウェア</th></tr>
<tr><td rowspan="2">サービス</td><td>ネーミング管理</td><td rowspan="2">Component Container※
TPBroker※</td></tr>
<tr><td>トランザクション管理</td></tr>
<tr><td rowspan="5">API</td><td>JNDI</td><td rowspan="2">Component Container※</td></tr>
<tr><td>JTA</td></tr>
<tr><td>RMI-IIOP</td><td>TPBroker※</td></tr>
<tr><td>JDBC Standard Extension</td><td rowspan="2">HiRDB Type4 JDBC Driver
Oracle JDBC Thin Driver
SQL Server の JDBC ドライバ</td></tr>
<tr><td>JDBC</td></tr>
</table>

注※　アプリケーションサーバの構成ソフトウェアです。

2.4.6　J2EE リソース

J2EE リソースはリソースと接続するために使用します。バッチサーバではリソースとしてデータベースを利用できます。データベースに接続するには，アプリケーションサーバで扱う J2EE リソースのうちリソースアダプタを使用します。

J2EE リソースの詳細については，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(コンテナ共通機能)」の「3.　リソース接続とトランザクション管理」を参照してください。

2.4.7　コンテナ拡張ライブラリ

アプリケーションが利用する共通のライブラリをコンテナ拡張ライブラリといいます。このライブラリを利用することによって，バッチアプリケーションからユーザ作成の共通のライブラリを呼び出せるようになります。

コンテナ拡張ライブラリについては，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(コンテナ共通機能)」の「19.　コンテナ拡張ライブラリ」を参照してください。

2.5　システムの目的に応じたアプリケーションの構成を決める（オンライン処理を実行する業務の場合）

この節では，システムの目的に応じたアプリケーションの構成の検討について説明します。この節で説明するのは，J2EE アプリケーションの場合の構成です。また，必要となるソフトウェアについても説明します。

アプリケーションサーバで実現できるアプリケーションの機能については，次に示すマニュアルの機能の分類に関する説明を参照してください。

- マニュアル「アプリケーションサーバ 機能解説 基本・開発編(Web コンテナ)」の「1.1　機能の分類」
- マニュアル「アプリケーションサーバ 機能解説 基本・開発編(EJB コンテナ)」の「1.1　機能の分類」
- マニュアル「アプリケーションサーバ 機能解説 基本・開発編(コンテナ共通機能)」の「1.1　機能の分類」
- マニュアル「アプリケーションサーバ 機能解説 拡張編」の「1.1　機能の分類」
- マニュアル「アプリケーションサーバ 機能解説 セキュリティ管理機能編」の「1.1　機能の分類」
- マニュアル「アプリケーションサーバ 機能解説 運用／監視／連携編」の「1.1　機能の分類」
- マニュアル「アプリケーションサーバ 機能解説 保守／移行編」の「1.1　機能の分類」
- マニュアル「アプリケーションサーバ 機能解説 互換編」の「1.1　機能の分類」

2.5.1　動作させる J2EE アプリケーションの検討

システムで動作させる J2EE アプリケーションについて検討します。

システムの目的に応じたアプリケーションの構成を明確にすることで，システム構成の基本的な部分が決まります。例えば，クライアントに Web ブラウザを使用する場合には，アプリケーションはサーブレットや JSP で構成される Web アプリケーションにして，Web ブラウザからのリクエストを受け付けるようにします。また，必要に応じて，サーブレットや JSP から Enterprise Bean を呼び出すようなアプリケーションも考えられます。この場合は，システムは Web クライアントシステムにして，Web サーバの配置や，そこからアプリケーションサーバを呼び出す場合の連携方法を検討する必要があります。

また，業務システムの基幹部分を構成するシステムを構築する場合は，クライアントに EJB クライアントアプリケーションを使用する構成が考えられます。呼び出し先の Enterprise Bean の種類によっては，CTM を利用することも検討できます。

アプリケーションに含まれるコンポーネントの種類とシステム構成との関係については，「3.3　アプリケーションの構成を検討する」で詳しく説明します。そのほか，J2EE アプリケーションを実行する場合のシステム構成の検討方法については，「3.　システム構成の検討（J2EE アプリケーション実行基盤)」を参照してください。

システムの目的に応じて使用できる機能については，次に示すマニュアルのシステムの目的と機能の対応に関する説明を参照してください。

- マニュアル「アプリケーションサーバ 機能解説 基本・開発編(Web コンテナ)」の「1.2　システムの目的と機能の対応」
- マニュアル「アプリケーションサーバ 機能解説 基本・開発編(EJB コンテナ)」の「1.2　システムの目的と機能の対応」
- マニュアル「アプリケーションサーバ 機能解説 基本・開発編(コンテナ共通機能)」の「1.2　システムの目的と機能の対応」
- マニュアル「アプリケーションサーバ 機能解説 拡張編」の「1.2　システムの目的と機能の対応」
- マニュアル「アプリケーションサーバ 機能解説 運用／監視／連携編」の「1.2　システムの目的と機能の対応」
- マニュアル「アプリケーションサーバ 機能解説 保守／移行編」の「1.2　システムの目的と機能の対応」

また，アプリケーション開発の手順については，マニュアル「アプリケーションサーバ アプリケーション開発ガイド」を参照してください。

2.5.2　使用するプロセスの検討と必要なソフトウェアの準備

アプリケーションサーバのシステムでは，使用するプロセスの種類とその配置によってシステム構成が決まります。

アプリケーションサーバのシステムは Web フロントシステムとバックシステムで構成されます。Web フロントシステムは，クライアントとして Web ブラウザを使用するシステムです。バックシステムは，クライアントとして EJB クライアントを使用するシステムです。システムの分類については，「3.1.1　システムの目的と構成」で詳しく説明します。

ここでは，まず，システムの分類に応じて必要なプロセスとソフトウェアについて説明します。次に，使用する機能に応じて必要なプロセス，モジュールおよびソフトウェアについて説明します。なお，これらのプロセス，モジュール，ソフトウェアをどのようにシステムに配置するかについては，「3.　システム構成の検討（J2EE アプリケーション実行基盤）」で説明します。

(1)　システムの分類に応じて必要なプロセス

システムの分類に応じて必要なプロセスを次に示します。これらは，使用する機能に関係なく共通して必要なプロセスです。アプリケーションサーバによって提供されます。

Web フロントシステムの場合に必要なプロセス

Web フロントシステムの場合に必要なプロセスは次のとおりです。

- Web サーバ※
- J2EE サーバ

- PRF デーモン

なお，アプリケーションサーバに含まれる Web サーバは，HTTP Server です。クライアントには，Web ブラウザを使用します。

注※

Web サーバを経由しないで J2EE サーバの NIO HTTP サーバに直接アクセスする場合は，Web サーバのプロセスは不要です。

ポイント

Web サーバ選択の指針

Web クライアントシステムでは，Web クライアントからのリクエストを，次のどちらかの Web サーバを利用して処理できます。

- **リバースプロキシ機能を有効にした Web サーバ**
  プロキシモジュールを組み込んだ Web サーバと連携してリクエストを処理します。Web サーバが受信したリクエストは，プロキシモジュールを経由して，J2EE サーバの NIO HTTP サーバに送信されます。
- **NIO HTTP サーバ**
  Web コンテナ機能の一部として提供される，J2EE サーバのプロセス内で機能する HTTP サーバでリクエストを処理します。Web クライアントからのリクエストを J2EE サーバで直接受信できます。

なお，アプリケーションサーバでは，リバースプロキシ機能を有効にした Web サーバを利用することを推奨しています。また，デフォルトの設定で使用する場合は，リバースプロキシ機能を有効にした Web サーバが利用されます。特に性能を重視したシステムを構築したい場合に，NIO HTTP サーバへの直接アクセスを検討してください。

それぞれの Web サーバの特徴を次の表に示します。Web サーバを選択する場合の指針にしてください。

表 2-5　Web サーバ選択の指針

比較項目	リバースプロキシ機能を有効にした Web サーバ	NIO HTTP サーバ
Web サーバとして利用できる機能	○ HTTP Server（Apache の機能をベースにした Web サーバ）または Microsoft IIS が提供する多様な機能を利用できます。	△ サーブレット，JSP または HTML から構成される Web アプリケーションへのアクセスを目的にした最小限の機能だけが提供されています。※1
構築，運用の容易性	○ 構築時には，Web サーバの環境設定が必要です。運用時には，Web サーバの起動，	○

比較項目	リバースプロキシ機能を有効にした Web サーバ	NIO HTTP サーバ
	停止が必要です。ただし，Smart Composer 機能のコマンドで構築・運用できるため，煩雑な操作は不要です。	構築時の Web サーバの環境設定，および運用時の Web サーバの起動，停止の操作が不要です。
HTML，JPEG などの静的コンテンツに対するアクセス性能	○ 静的コンテンツを Web サーバ上に配置することによって，最適な性能を確保できます。 なお，Web コンテナ上に配置する場合は，リバースプロキシ経由のアクセスになるため，アクセス処理に時間が掛かります。	○ リバースプロキシを経由しないでアクセスできるため，最適な性能を確保できます。
サーブレット，JSP などの動的コンテンツに対するアクセス性能	△ リバースプロキシを経由するためのアクセス処理に時間が掛かります。	○ リバースプロキシを経由しないでアクセスできるため，最適な性能を確保できます。
注意事項	インターネットに接続する場合には，セキュリティ上の観点から，DMZ を確保する構成にして，リバースプロキシをフロントに配置することを推奨します。 また，リバースプロキシを配置しない場合は，リバースプロキシを組み込んだ Web サーバを DMZ に配置することで，同様の効果を得ることもできます。※2	インターネットに接続する場合には，セキュリティ上の観点から DMZ を確保する構成にして，必ずリバースプロキシをフロントに配置してください。※2

（凡例）○：優れている。 △：優れていない。

注※1　NIO HTTP サーバで使用できる機能の詳細については，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(Web コンテナ)」の「7.2.2　NIO HTTP サーバで使用できる機能」を参照してください。

注※2　DMZ への Web サーバの配置については，マニュアル「アプリケーションサーバ 機能解説 セキュリティ管理機能編」の「3.3　DMZ へのリバースプロキシの配置を検討する」を参照してください。

バックシステムの場合に必要なプロセス

バックシステムの場合に必要なプロセスは次のとおりです。

- J2EE サーバ
- PRF デーモン

バックシステムのクライアントには，EJB クライアントを使用します。**EJB クライアント**とは，Enterprise Bean を呼び出す，Servlet，JSP，ほかの Enterprise Bean，EJB クライアントアプリケーション，またはほかの業務システムのことです。

EJB クライアントとして EJB クライアントアプリケーションを使用するとき，Windows の場合はクライアントマシンを Client を使用して構築することもできます。アプリケーションサーバまたは Client のどちらのソフトウェアを使用した場合も，必要に応じて PRF デーモンを起動できます。

> **参考**
>
> CTM を使用したシステムの場合，クライアントとして TPBroker や TPBroker Object Transaction Monitor のクライアントなど，EJB クライアント以外のクライアントも使用できます。

(2) 使用する機能に応じて必要なプロセスおよびモジュール

使用する機能に応じて必要なプロセスおよびモジュールについて説明します。アプリケーションサーバによって提供されるものと，アプリケーションサーバ以外のソフトウェアによって提供されるものがあります。

使用する機能ごとに必要なプロセスのうち，アプリケーションサーバによって提供されるものを次の表に示します。これらのプロセスは，アプリケーションサーバをインストールしたマシンで起動できます。

表 2-6 機能ごとに必要なプロセスまたはモジュール（アプリケーションサーバによって提供されるもの）

機能	必要なプロセス
サーバ間連携で CTM を利用する／CTM を利用して負荷を分散する	CTM デーモン
	CTM レギュレータ
	CTM ドメインマネジャ
	グローバル CORBA ネーミングサービス
	スマートエージェント
Management Server を利用して運用管理する	Management Server
	運用管理エージェント
CORBA ネーミングサービスをアウトプロセスで起動する	CORBA ネーミングサービス

使用する機能ごとに必要なプロセスおよびモジュールのうち，アプリケーションサーバ以外の製品が提供するプロセスおよびモジュールを，表 2-7 および表 2-8 に示します。

表 2-7 機能ごとに必要なモジュール（アプリケーションサーバ以外によって提供されるもの）と提供するソフトウェア

機能	モジュール	提供するソフトウェア	備考
データベース（HiRDB）と接続する	HiRDB Type4 JDBC Driver	• HiRDB Server Version 10 • HiRDB/Run Time Version 10 • HiRDB/Developer's Kit Version 10 • HiRDB Developer's Suite Version 10 • HiRDB Server Version 9	JDBC ドライバとして HiRDB Type4 JDBC Driver を使用する場合に必要になります。

機能	モジュール	提供するソフトウェア	備考
		• HiRDB Server with Additional Function Version 9 • HiRDB/Run Time Version 9 • HiRDB/Developer's Kit Version 9 • HiRDB Developer's Suite Version 9	
データベース（Oracle）と接続する	Oracle JDBC Thin Driver	• Oracle JDBC Thin Driver	JDBC ドライバとして Oracle JDBC Thin Driver を使用する場合に必要になります。
データベース（SQL Server）と接続する	SQL Server JDBC Driver	• SQL Server JDBC Driver	JDBC ドライバとして SQL Server JDBC Driver を使用する場合に必要になります。
データベース（XDM/RD E2）と接続する	HiRDB Type4 JDBC Driver	• HiRDB Server Version 10 • HiRDB/Run Time Version 10 • HiRDB/Developer's Kit Version 10 • HiRDB Developer's Suite Version 10	JDBC ドライバとして HiRDB Type4 JDBC Driver を使用する場合に必要になります。
Message Queue サーバと接続する	TP1/Message Queue - Access	• TP1/Message Queue - Access	−
OpenTP1 の SPP と接続する	TP1 Connector	• TP1 Connector	−
	TP1/Client/J	• TP1/Client/J	−

（凡例）　−：該当しません。

注　モジュールは，J2EE サーバのプロセスに含まれて動作します。

表 2-8　機能ごとに必要なプロセス（アプリケーションサーバ以外によって提供されるもの）と提供するソフトウェア

機能	必要なプロセス	提供するソフトウェア
クラスタソフトウェアを使用して障害時に系を切り替える	Windows Server Failover Cluster	Windows Server Failover Cluster
	HA モニタ	HA モニタ

2.6　システムの目的に応じたアプリケーションの構成を決める（バッチ処理を実行する業務の場合）

この節では，システムの目的に応じたアプリケーションの構成の検討について説明します。この節で説明するのは，バッチアプリケーションの場合の構成です。また，必要となるソフトウェアについても説明します。

アプリケーションサーバで実現できるアプリケーションの機能については，次に示すマニュアルの機能の分類に関する説明を参照してください。

- マニュアル「アプリケーションサーバ 機能解説 基本・開発編(Web コンテナ)」の「1.1　機能の分類」
- マニュアル「アプリケーションサーバ 機能解説 基本・開発編(EJB コンテナ)」の「1.1　機能の分類」
- マニュアル「アプリケーションサーバ 機能解説 基本・開発編(コンテナ共通機能)」の「1.1　機能の分類」
- マニュアル「アプリケーションサーバ 機能解説 拡張編」の「1.1　機能の分類」
- マニュアル「アプリケーションサーバ 機能解説 セキュリティ管理機能編」の「1.1　機能の分類」
- マニュアル「アプリケーションサーバ 機能解説 運用／監視／連携編」の「1.1　機能の分類」
- マニュアル「アプリケーションサーバ 機能解説 保守／移行編」の「1.1　機能の分類」

2.6.1　動作させるバッチアプリケーションの検討

システムで動作させるバッチアプリケーションについて検討します。**バッチアプリケーション**とは，バッチ処理として実行する，定型的・定期的な処理を実装した Java アプリケーションのことです。

システム構成の基本的な部分は，バッチアプリケーションで実装した処理内容によって決まります。例えば，データベース上のデータを参照・更新するような処理を実装した場合は，トランザクションの管理方法や，リソースとの接続方法を検討する必要があります。また，ほかの J2EE サーバ上の業務処理プログラム（Enterprise Bean）を呼び出す処理を実装した場合は，サーバ間の連携方法について，検討する必要があります。

バッチアプリケーションを実行する場合のシステム構成の検討方法については，「4.　システム構成の検討(バッチアプリケーション実行基盤)」を参照してください。

システムの目的に応じて使用できる機能については，次に示すマニュアルのシステムの目的と機能の対応に関する説明を参照してください。

- マニュアル「アプリケーションサーバ 機能解説 基本・開発編(Web コンテナ)」の「1.2　システムの目的と機能の対応」
- マニュアル「アプリケーションサーバ 機能解説 基本・開発編(EJB コンテナ)」の「1.2　システムの目的と機能の対応」

- マニュアル「アプリケーションサーバ 機能解説 基本・開発編(コンテナ共通機能)」の「1.2　システムの目的と機能の対応」
- マニュアル「アプリケーションサーバ 機能解説 拡張編」の「1.2　システムの目的と機能の対応」
- マニュアル「アプリケーションサーバ 機能解説 運用／監視／連携編」の「1.2　システムの目的と機能の対応」
- マニュアル「アプリケーションサーバ 機能解説 保守／移行編」の「1.2　システムの目的と機能の対応」

2.6.2　使用するプロセスの検討と必要なソフトウェアの準備

アプリケーションサーバのシステムでは，使用するプロセスの種類とその配置によってシステム構成が決まります。

ここでは，まず，必要なプロセスとソフトウェアについて説明します。次に，使用する機能に応じて必要なプロセス，モジュールおよびソフトウェアについて説明します。なお，これらのプロセス，モジュール，ソフトウェアをどのようにシステムに配置するかについては，「4.　システム構成の検討（バッチアプリケーション実行基盤)」で説明します。

(1)　必要なプロセス

必要なプロセスを次に示します。これらは，使用する機能に関係なく共通して必要なプロセスです。アプリケーションサーバによって提供されます。

- バッチサーバ
- PRF デーモン

(2)　使用する機能に応じて必要なプロセスおよびモジュール

使用する機能に応じて必要なプロセスおよびモジュールについて説明します。アプリケーションサーバによって提供されるものと，アプリケーションサーバ以外のソフトウェアによって提供されるものがあります。

使用する機能ごとに必要なプロセスのうち，アプリケーションサーバによって提供されるものを次の表に示します。これらのプロセスは，アプリケーションサーバをインストールしたマシンで起動できます。

表 2-9　機能ごとに必要なプロセスまたはモジュール（アプリケーションサーバによって提供されるもの)

機能	必要なプロセス
Management Server を利用して運用管理する	Management Server
	運用管理エージェント
CTM を利用してバッチアプリケーションの実行をスケジューリングする	CTM デーモン
	CTM レギュレータ

機能	必要なプロセス
	CTM ドメインマネジャ
	グローバル CORBA ネーミングサービス
	スマートエージェント

使用する機能ごとに必要なプロセスおよびモジュールのうち，アプリケーションサーバ以外の製品が提供するプロセスおよびモジュールを，表 2-10 および表 2-11 に示します。

表 2-10　機能ごとに必要なモジュール（アプリケーションサーバ以外によって提供されるもの）と提供するソフトウェア

機能	モジュール	提供するソフトウェア	備考
データベース（HiRDB）と接続する	HiRDB Type4 JDBC Driver	• HiRDB Server Version 10 • HiRDB/Run Time Version 10 • HiRDB/Developer's Kit Version 10 • HiRDB Developer's Suite Version 10 • HiRDB Server Version 9 • HiRDB Server with Additional Function Version 9 • HiRDB/Run Time Version 9 • HiRDB/Developer's Kit Version 9 • HiRDB Developer's Suite Version 9	JDBC ドライバとして HiRDB Type4 JDBC Driver を使用する場合に必要になります。
データベース（Oracle）と接続する	Oracle JDBC Thin Driver	• Oracle JDBC Thin Driver	JDBC ドライバとして Oracle JDBC Thin Driver を使用する場合に必要になります。
データベース（SQL Server）と接続する	SQL Server JDBC Driver	• SQL Server JDBC Driver	JDBC ドライバとして SQL Server JDBC Driver を使用する場合に必要になります。
データベース（XDM/RD E2）と接続する	HiRDB Type4 JDBC Driver	• HiRDB Server Version 10 • HiRDB/Run Time Version 10 • HiRDB/Developer's Kit Version 10 • HiRDB Developer's Suite Version 10	JDBC ドライバとして HiRDB Type4 JDBC Driver を使用する場合に必要になります。

注　モジュールは，バッチサーバのプロセスに含まれて動作します。

表 2-11　機能ごとに必要なプロセス（アプリケーションサーバ以外によって提供されるもの）と提供するソフトウェア

機能	必要なプロセス	提供するソフトウェア
クラスタソフトウェアを使用して障害時に系を切り替える	Windows Server Failover Cluster	Windows Server Failover Cluster
	HA モニタ	HA モニタ

2.7　運用方法を検討する

この節では，アプリケーションサーバのシステムの運用方法を検討するときに考慮することについて説明します。

アプリケーションサーバのシステムの運用では，Management Server というプロセスを利用できます。Management Server を利用すると，アプリケーションサーバのシステムを構成する複数のプロセスを**論理サーバ**として扱い，一括して運用できるようになります。

なお，業務システムは一般的に，アプリケーションサーバだけではなくほかのプログラムで構築されるシステムと組み合わせて構築されています。運用方法の検討では，アプリケーションサーバのシステムのほかに，これらのプログラムも合わせてシステム全体をどのように運用するかを検討する必要があります。

2.7.1　JP1 と連携したシステムの運用

Management Server を利用して運用している場合，統合システム運用管理用ミドルウェア JP1 のプログラムと連携して，次のようなシステムの運用を実現できます。

- JP1/IM と連携したシステム全体の集中監視
- JP1/AJS と連携したシステム全体の自動運転

それぞれのプログラムとの連携によって実現できる機能の詳細については，マニュアル「アプリケーションサーバ 機能解説 運用／監視／連携編」を参照してください。

なお，Management Server を利用しないで運用している場合でも，JP1/AJS を利用したシステムの自動運転は実現できます。ただし，その場合，JP1/AJS のジョブとして，サーバ管理コマンドなどを一から定義する必要があります。

2.7.2　クラスタソフトウェアと連携したシステムの運用

クラスタソフトウェアと連携すると，障害発生時などに自動的に**系切り替え**ができます。

また，J2EE サーバをクラスタ構成にしている場合，N 個の J2EE サーバに対して 1 個のリカバリ専用サーバを配置することで，障害発生時にトランザクションを解決してリソースを解放できます。ただし，この機能は，バッチサーバには該当しません。

アプリケーションサーバのシステムでは，OS ごとに，次のクラスタソフトウェアを使用できます。

- Windows の場合
  Windows Server Failover Cluster
- AIX または Linux の場合
  HA モニタ

クラスタソフトウェアと連携するためには，システムを Management Server を利用して運用している必要があります。

クラスタソフトウェアと連携したシステムの運用をする場合のシステム構成については，「3.11　クラスタソフトウェアを使用した障害時の系切り替えを検討する」を参照してください。

3 システム構成の検討（J2EE アプリケーション実行基盤）

この章では，J2EE アプリケーション実行基盤を構築する場合のシステム構成について説明します。システムを設計する流れに沿って，それぞれの設計項目でのシステム構成の標準的なパターンを示します。また，それぞれのポイントで意識する必要があるコンポーネント，プロセスおよび処理の流れについて説明します。

バッチアプリケーション実行基盤のシステム構成を検討する場合は，「4.　システム構成の検討（バッチアプリケーション実行基盤）」を参照してください。

3.1　システム構成を検討するときに考慮すること

この節では，アプリケーションサーバを使用するシステムの構成を検討するときに考慮する必要があることについて説明します。

システム構成を検討する場合，J2EE アプリケーションで使用する機能に応じて，その機能を実現するために必要なプロセスを意識し，それぞれを各マシンに適切に配置することが必要です。このとき，アプリケーションサーバの要件である，信頼性，可用性なども十分に考慮する必要があります。

3.1.1　システムの目的と構成

アプリケーションサーバで構築するシステムは，目的とする業務および実行する J2EE アプリケーションの要件と特徴に応じて，次の 2 種類のシステムに分類できます。

- Web フロントシステム
- バックシステム

Web フロントシステムは，Web ベースのシステムの場合に，フロントエンドである Web ブラウザから送信されるリクエストを受け付けて，そのリクエストを処理するシステムです。このシステムでは，アプリケーションサーバ上の J2EE サーバで，サーブレット，JSP および Enterprise Bean が動作します。

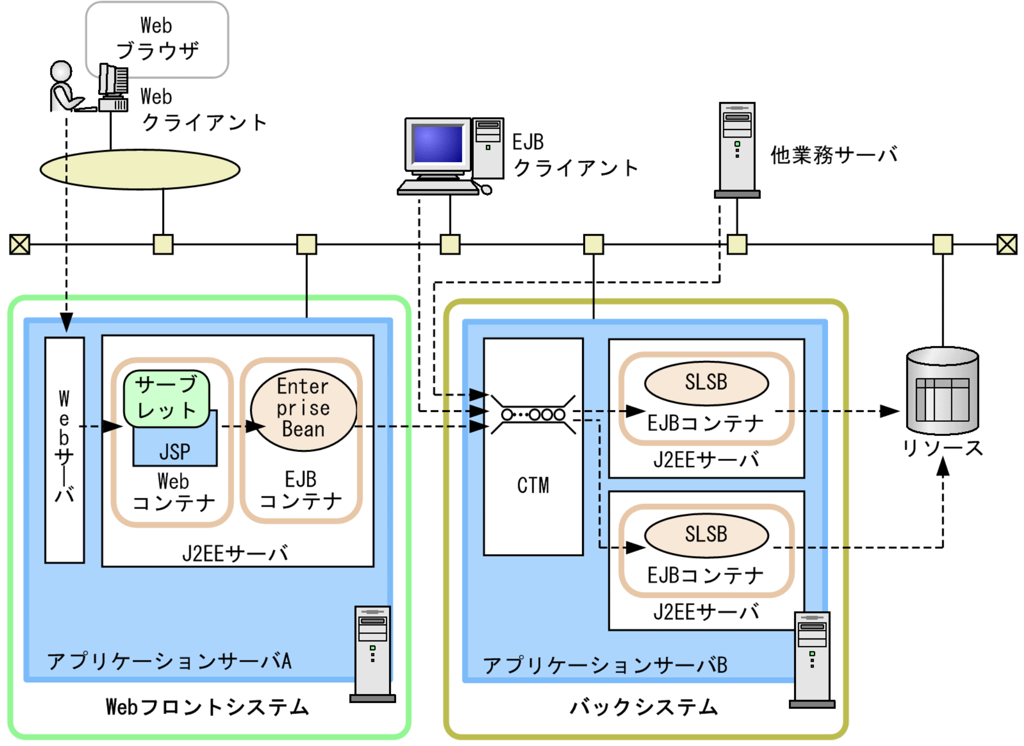
Web フロントシステムの背後で動作する，複数の業務システムに共通な業務サービスを実行するためのシステムが，**バックシステム**です。バックシステムにリクエストを送信するのは，次のようなコンポーネント，アプリケーションまたはシステムです。

- Web フロントシステム上で動作しているサーブレット，JSP または Enterprise Bean
- EJB クライアントマシンで動作している EJB クライアントアプリケーション
- ほかの業務システム

アプリケーションサーバのシステムは，目的や規模に応じて，Web フロントシステムとバックシステムを一つまたは複数組み合わせて構成されます。

Web フロントシステムとバックシステムの構成例を次の図に示します。

図 3-1　Web フロントシステムとバックシステムの構成例



（凡例）

---►：リクエストの流れ

注　SLSB：Stateless Session Bean

アプリケーションサーバのシステム構成を検討する場合，まず，システムの基本構成として，システム全体をどのようなシステムの組み合わせで構成するかを検討します。次に，構成要素であるそれぞれのシステムの目的と，クライアントからアクセスされるポイントを明確にします。それから，信頼性，性能，拡張性などアプリケーションサーバに共通して求められる要件や，EIS との接続や負荷分散の実現などそのシステム独自の要件を満たすためにはどのようにソフトウェアやプロセスを配置するのがよいのかを検討し，最適なシステム構成を設計していきます。

3.1.2　システム構成の設計手順

システム構成は，次の流れで設計します。

図 3-2　システム構成を設計する流れ（J2EE アプリケーション実行基盤の場合）

注　使用する機能に応じた構成を検討する順序は任意です。

(1)　アプリケーションの構成を検討する

各システム上で動作するアプリケーションで使用するコンポーネントの構成を明確にして，アプリケーションのどのコンポーネントをアクセスポイントにするかを決定します。詳細は，「3.3　アプリケーションの構成を検討する」を参照してください。

アクセスポイントとは，クライアントからリモートでアクセスされる場合に，アプリケーションのリクエスト受信窓口として動作するコンポーネントです。アクセスポイントになるコンポーネントの種類によって，実現できるシステム構成が決まります。

また，アプリケーションがデータベースなどのリソースに接続するかどうかを決めます。接続するリソースに対応して，使用するリソースアダプタが決まります。

なお，ここで検討したアクセスポイントになるコンポーネントの種類によって，それ以降のシステム構成の設計で検討が必要な項目が異なります。アクセスポイントになるコンポーネントの種類ごとに検討が必要な項目を，次の表に示します。

表 3-1 アクセスポイントになるコンポーネントの種類ごとに検討が必要な項目

設計項目	アクセスポイントになるコンポーネントの種類				参照先
	Web フロントシステム	バックシステム			
	サーブレット／JSP	Session Bean／Entity Bean	CTM を使用する場合の Stateless Session Bean	Message-driven Bean	
クライアントとサーバの構成	◎	◎	◎	－	3.4
サーバ間での連携方法	－	△	△	－	3.5
トランザクションの種類	◎	◎	◎	◎	3.6
ロードバランスクラスタによる負荷分散方式	△	△	△	－	3.7
サーバ間での非同期通信	－	－	－	◎	3.8
運用管理プロセスの配置	◎	◎	◎	◎	3.9
セッション情報を引き継ぐための構成	△	△	－	－	3.10
クラスタソフトウェアを使用した系切り替えを実現するための構成	△	△	△	△	3.11
ファイアウォールの配置	△	△	△	△	※2
DMZ へのリバースプロキシの配置	△※1	－	－	－	※3
性能解析トレースファイルを出力するプロセスの配置	◎	◎	◎	◎	3.12
アプリケーションサーバ以外の製品との連携	△	△	△	△	3.13
任意のプロセスの運用管理	△	△	△	△	3.14

（凡例）

◎：必ず検討する項目。

△：必要に応じて検討する項目。

－：検討する必要がない項目。

注※1　Web サーバを経由しないで直接 NIO HTTP サーバを使用する場合は必ず検討してください。

注※2　マニュアル「アプリケーションサーバ 機能解説 セキュリティ管理機能編」の「3.2　ファイアウォールを使用する構成を検討する」を参照してください。

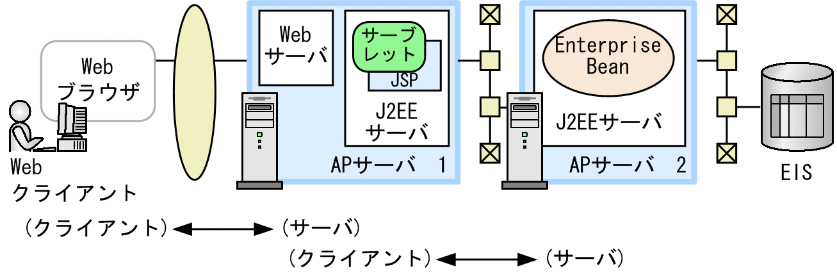
注※3　マニュアル「アプリケーションサーバ 機能解説 セキュリティ管理機能編」の「3.3　DMZ へのリバースプロキシの配置を検討する」を参照してください。

(2)　クライアントとサーバの構成を検討する

アクセスポイントになるコンポーネントに応じて，クライアントとサーバの対応を明確にします。システムに配置したアプリケーションサーバは，その役割に応じて，クライアントとして機能したり，サーバとして機能したりします。

クライアントとサーバの構成の考え方を次の図に示します。

図 3-3　クライアントとサーバの構成の考え方



注　APサーバ：アプリケーションサーバ

この構成の場合，AP サーバ 1 は Web クライアントに対するサーバに当たり，アクセスポイントはサーブレット／JSP になります。また，AP サーバ 1 は AP サーバ 2 に対してのクライアントになります。AP サーバ 2 は AP サーバ 1 に対するサーバに当たり，アクセスポイントは Enterprise Bean になります。

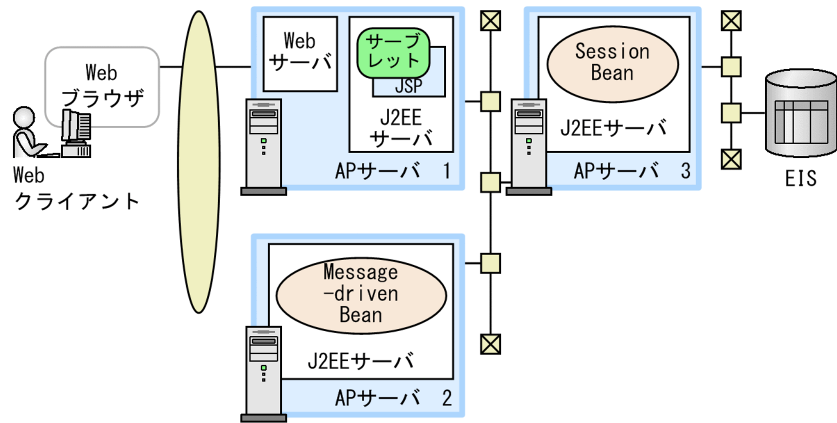
詳細は，「3.4　クライアントとサーバの構成を検討する」を参照してください。

(3)　サーバ間での連携方法を検討する

サーバが複数ある場合に，サーバ間で連携するかどうか，連携する場合はどのように連携するかを検討します。サーバ間連携とは，クライアントから見て垂直に並んだ複数のサーバから，ほかのサーバ上にあるアクセスポイントのコンポーネントを呼び出して処理を実行する連携方法です。

サーバ間連携の考え方を次の図に示します。

図 3-4　サーバ間連携の考え方



注　APサーバ：アプリケーションサーバ

この構成の場合，AP サーバ 1 と AP サーバ 2 は，AP サーバ 3 のクライアントになり，それぞれのサーブレット，JSP または Message-driven Bean から，AP サーバ 3 の Session Bean を呼び出します。

サーバ間連携と同時に，必要に応じて，アプリケーションを分割するなど，アプリケーションの形態も見直します。また，複数のシステム間で連携する場合も，それぞれのシステムに含まれるサーバ間の連携方法を検討する必要があります。

詳細は，「3.5　サーバ間での連携を検討する」を参照してください。

(4)　トランザクションの種類を検討する

データベースなどのリソースを使用するシステムの場合に，トランザクション管理の対象になるリソースの数に応じて，使用するトランザクションの種類を検討します。詳細は，「3.6　トランザクションの種類を検討する」を参照してください。

(5)　ロードバランスクラスタによる負荷分散方式を検討する

システムの可用性を高めるために，ロードバランスクラスタ構成によって負荷を分散するかどうか検討します。負荷を分散する場合は，アクセスポイントになるコンポーネントの種類に応じてどの方式で実現するかを検討します。詳細は，「3.7　ロードバランスクラスタによる負荷分散方式を検討する」を参照してください。

(6)　サーバ間の非同期通信をするための構成を検討する

Message-driven Bean を使用してアプリケーションサーバ間での非同期通信をする場合に，利用する製品に応じたシステムの構成を検討します。なお，Message-driven Bean による負荷分散についても，あわせて検討します。詳細は，「3.8　サーバ間で非同期通信をする場合の構成を検討する」を参照してください。

(7)　運用管理プロセスの配置を検討する

運用管理プロセス（Management Server）を利用する場合に，管理対象にする範囲によって，Management Server をどのサーバマシンに配置するかを検討します。詳細は，「3.9　運用管理プロセスの配置を検討する」を参照してください。

(8)　セッション情報の引き継ぎを検討する

Web フロントシステムで動作する J2EE アプリケーションまたは J2EE サーバに障害が発生した場合に，セッション情報をほかの J2EE サーバに引き継ぐための構成について検討します。詳細は，「3.10　セッション情報の引き継ぎを検討する」を参照してください。

(9)　クラスタソフトウェアを使用した障害時の系切り替えまたはリソース解放を実現するための構成を検討する

一つのアプリケーションサーバに障害が発生した場合およびシステムを保守する場合を考慮して，クラスタソフトウェアによって系切り替えを実行して，システムの運用を続行させるための構成を検討します。このとき，実行系と待機系がそれぞれ相互に切り替え対象になる構成（相互スタンバイ構成）も検討できます。また，トランザクションサービスを使用している場合は，障害発生時にリソースを解放するためのリカバリサーバを利用するための構成も検討します。詳細は，「3.11　クラスタソフトウェアを使用した障害時の系切り替えを検討する」を参照してください。

(10)　ファイアウォールの配置を検討する

アプリケーションサーバおよびリソースのセキュリティを確保するための，ファイアウォールの配置について検討します。ファイアウォールは，アクセスポイントになるコンポーネントの前に配置して，アクセスポイントに対する不正なアクセスを防止します。ファイアウォールを配置するポイントは，アクセスポイントによって決まります。

基本的なファイアウォールの配置については，マニュアル「アプリケーションサーバ 機能解説 セキュリティ管理機能編」の「3.2　ファイアウォールを使用する構成を検討する」を参照してください。

侵入検知システムも含めたセキュリティ構成の詳細については，マニュアル「アプリケーションサーバ 機能解説 セキュリティ管理機能編」の「4.11.2　ファイアウォールと侵入検知システムを配置する」を参照してください。

(11)　DMZ へのリバースプロキシの配置を検討する

インターネットと接続するシステムの場合などには，アプリケーションサーバおよびリソースのセキュリティを確保するために，DMZ へのリバースプロキシの配置を検討します。詳細は，マニュアル「アプリケーションサーバ 機能解説 セキュリティ管理機能編」の「3.3　DMZ へのリバースプロキシの配置を検討する」を参照してください。

(12)　性能解析トレースファイルを出力するプロセスを配置する

性能解析に使用するプロセスである PRF デーモン（パフォーマンストレーサ）の配置について検討します。詳細は，「3.12　性能解析トレースファイルを出力するプロセスを配置する」を参照してください。

(13)　アプリケーションサーバ以外の製品との連携を検討する

システムの集中監視，構成管理，自動運転などをしたい場合には，必要に応じて JP1 などのアプリケーションサーバ以外の製品との連携を検討します。詳細は，「3.13　アプリケーションサーバ以外の製品との連携を検討する」を参照してください。

(14)　任意のプロセスを運用管理の対象にする

ユーザが定義する任意のプロセスをユーザサーバとして Management Server による運用管理の対象にしたい場合には，ユーザサーバの配置について検討します。詳細は，「3.14　任意のプロセスを運用管理の対象にする」を参照してください。

(15)　そのほかの構成を検討する

(14)までで検討した以外の構成について検討します。また，必要に応じて，旧バージョンのアプリケーションサーバで構築しているシステムとの互換性を持つシステム構成も検討します。

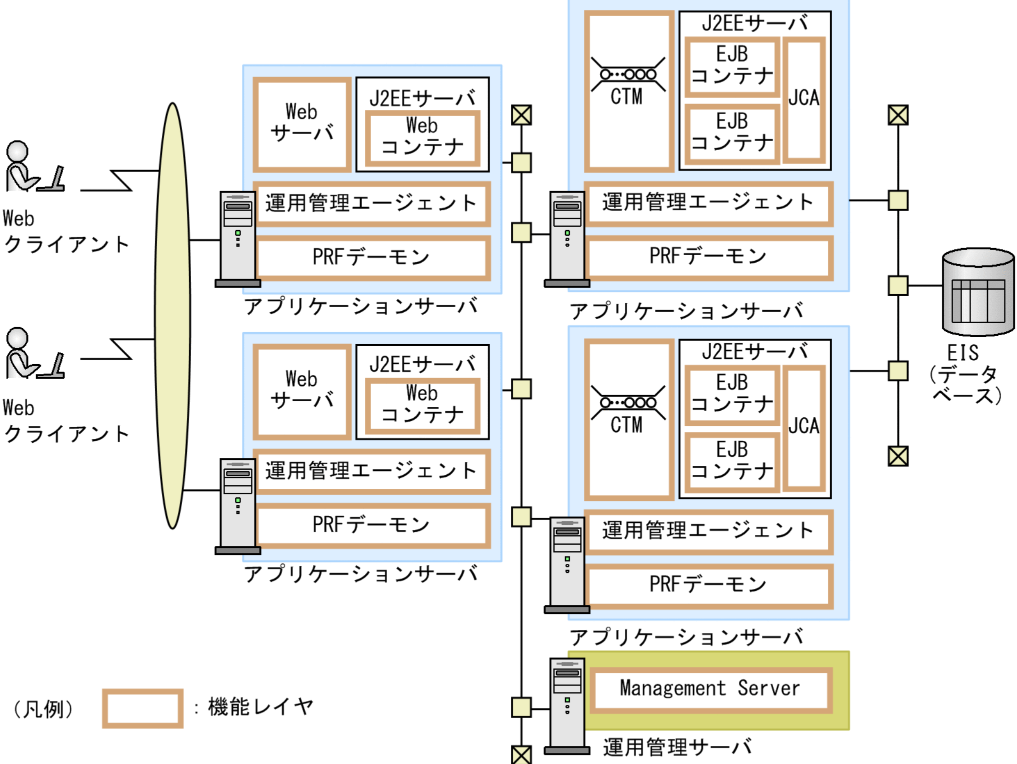
3.1.3　システム構成の考え方

ここでは，システム構成を検討するときの基本的な考え方について説明します。アプリケーションサーバで構築する実行環境は，複数のプロセスで構成されます。また，各プロセスには，提供する機能ごとに，複数のレイヤがあります。このレイヤを，**機能レイヤ**といいます。

実行環境では，各機能レイヤをそれぞれ異なるサーバに配置できます。配置は，システムの規模や目的に応じて検討する必要があります。

機能レイヤの配置例を次の図に示します。

図 3-5　機能レイヤの配置例



ここでは，次の機能レイヤを実行環境に配置する場合の構成の考え方について説明します。

- Web サーバ
- Web コンテナ
- CTM
- EJB コンテナ
- JCA
- EIS
- ネーミングサービス
- 運用管理サーバ

(1)　Web サーバと Web コンテナ（J2EE サーバ）の配置

Web サーバは，Web コンテナに対する HTTP リクエストのディスパッチ処理のほか，業務処理に含まれる静的コンテンツの処理を実行します。Web コンテナは，J2EE サーバの一部として動作して，サーブレットおよび JSP を実行するための基盤になる機能です。Web サーバと Web コンテナは同じホストまたは異なるホストに配置できます。なお，Web サーバを経由しないで J2EE サーバの NIO HTTP サーバに直接アクセスする場合は，Web サーバは不要です。

これを踏まえた上で，次の説明を参照して，システム構成を決定してください。

- 「3.4.1　サーブレットと JSP をアクセスポイントに使用する構成（Web サーバ連携の場合）」

- 「3.4.2　サーブレットと JSP をアクセスポイントに使用する構成（NIO HTTP サーバに直接アクセスする場合）」

(2)　CTM と EJB コンテナ（J2EE サーバ）の配置

CTM は，OLTP 技術を用いて，クライアントからのリクエストをスケジューリングする機能です。EJB コンテナは，J2EE サーバの一部として動作して，Enterprise Bean の実行および通信，トランザクション管理などのシステムレベルのサービスを提供するための基盤になる機能です。

CTM では，EJB コンテナに対して送信された IIOP リクエストのスケジュール処理と負荷分散処理を実行します。CTM は IIOP リクエストを処理するために特化された機能レイヤなので，EJB コンテナと別のホストに配置する必要はありません。IIOP リクエストの負荷分散は，CTM 間のデータ転送で実現されます。ただし，CTM による負荷分散の対象になるのは，Stateless Session Bean だけです。

CTM と EJB コンテナの関係は，1 対多にして，それぞれロードバランスクラスタ構成にすることをお勧めします。これによって，スループットが向上します。また，拡張性と性能が高いシステムを構築できます。また，クラスタ構成にすれば障害発生時の部分縮退や部分回復ができるようになるので，信頼性と可用性も向上します。

これを踏まえた上で，次の説明を参照して，システム構成を決定してください。

- 「3.4.4　CTM を使用する場合に Stateless Session Bean をアクセスポイントに使用する構成」
- 「3.7.4　CTM を利用した負荷分散（Stateless Session Bean の場合）」

(3)　JCA と EIS の配置

JCA は，既存システムやデータベースなどの EIS と接続するためのサポート機能を提供します。

EIS に対して，JCA ではコネクションプーリング処理をします。EIS を一つのサーバで構築している場合，JCA と EIS の関係は，多対 1 になります。EIS を複数のサーバで構築している場合は，JCA と EIS の関係は多対多になります。JCA では，EIS との接続で障害が発生した場合に，自動回復する機能を持っているので，信頼性と可用性が確保されています。

(4)　ネーミングサービスの配置

ネーミングサービスは，名前からオブジェクトを利用できるようにするネーミング管理機能を提供します。ネーミングサービスは，TPBroker によって提供されるサービスです。J2EE サーバのインプロセスで起動することをお勧めします。

インプロセスで起動することによって，アプリケーションサーバ内のプロセス数が削減できます。また，個別プロセスとしての起動処理や停止処理が不要になり，運用性が向上します。

(5) 運用管理プロセスの配置

Management Server は，アプリケーションサーバ全体を運用管理するための機能を集約したサーバです。システムの運用管理，監視の対象範囲となるドメイン内に一つ配置します。

ここでは，J2EE サーバなどのほかのプロセスとは別のホストに配置した構成を使用して説明していますが，同じホストに配置した構成にすることもできます。

運用管理，監視の対象になるプロセスを配置したホストには，それぞれ運用管理エージェントを起動します。運用管理エージェントは，運用管理サーバからの指示を受けて，各ホストで操作を実行します。

運用管理者は，Smart Composer 機能または Management Server のコマンドを使用して，運用管理を実行します。これらのコマンドは，Management Server を配置したホストで実行します。

参考

開発環境やテスト環境で運用管理を実行する場合には，必要に応じて次の機能も使用できます。

- **運用管理ポータル**
  Web ブラウザから Management Server を配置したホストにアクセスして実行します。

なお，運用管理ポータルを使用する場合は，必要に応じて管理クライアントマシンを用意してください。

運用管理プロセスを配置した構成については，「3.9　運用管理プロセスの配置を検討する」を参照してください。

3.2 システム構成の説明について

3 章では，システム構成を検討するための，さまざまなシステム構成パターンを示して，それぞれの特徴について説明します。この節では，システム構成の説明をお読みになる前に確認していただきたい，各システム構成パターンに共通する留意点について説明します。また，この章のシステム構成図で使用する図の凡例についてもあわせて説明します。これらをご確認の上，次節以降をお読みください。

参考

ここで説明する凡例は，「4.　システム構成の検討（バッチアプリケーション実行基盤）」およびマニュアル「アプリケーションサーバ 機能解説 セキュリティ管理機能編」の「4.11.2　ファイアウォールと侵入検知システムを配置する」にも当てはまります。

3.2.1　この章のシステム構成に共通する留意点

各システム構成に共通して，次のことに留意してください。

- この章では，Application Server を使用した場合のシステム構成について説明しています。必要に応じて，この章で使用している用語を，ご利用の製品に読み替えてください。

  ご利用の製品とこの章で使用している製品の対応を次の表に示します。

  表 3–2　ご利用の製品とこの章で使用している製品の対応

ご利用の製品	この章で記載している製品
Developer※	Application Server
Service Architect※	
Service Platform	

注※　テスト環境で使用している場合にだけ読み替えが必要です。

- システム構成の例では，主に，アプリケーションサーバを役割ごとに 1 台ずつ配置した例を示していますが，実際の構成では，スケーラビリティと可用性を考慮して負荷分散機などを使用したクラスタ構成にすることをお勧めします。これによって，トラブルが発生した場合やサーバのメンテナンスが必要な場合に，サービスが完全に止まることを防げます。負荷分散をする構成については，「3.7　ロードバランスクラスタによる負荷分散方式を検討する」を参照してください。
- この章では，Management Server を利用して運用することにした場合の例を示しています。Management Server を利用しないで運用する場合，次のプロセスは不要ですので，省略してお読みください。
  - Management Server
  - 運用管理エージェント

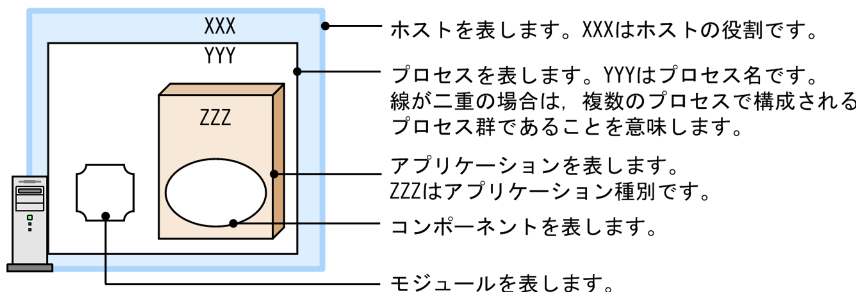
運用管理に必要なプロセスだけを起動するマシンを**運用管理サーバマシン**といいます。なお，Management Server を利用しないで運用する場合，運用管理サーバマシンは不要です。

- アクセスポイントがサーブレット／JSP のとき，この章では主に，Web サーバ連携の場合の構成例を示しています。Web サーバを経由しないで J2EE サーバの NIO HTTP サーバに直接アクセスする場合は，次のプロセスは不要ですので，省略してお読みください。
  - Web サーバ
- PRF デーモンは，性能解析トレースファイルを出力するためのプロセスです。アプリケーションサーバには必ず配置し，EJB クライアントには必要に応じて配置します。PRF デーモンの配置についての考え方については，「3.12　性能解析トレースファイルを出力するプロセスを配置する」を参照してください。

3.2.2　システム構成図で使用する図の凡例

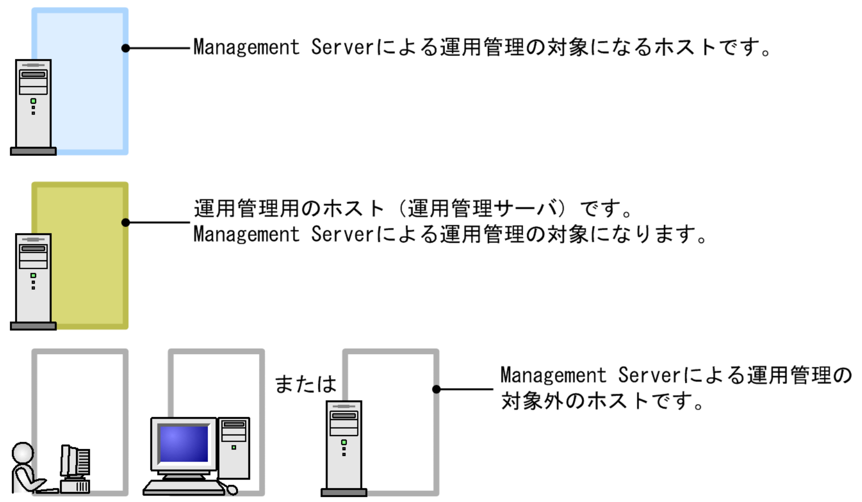
ここでは，3 章および 4 章のシステム構成図で使用する凡例を示します。

図 3-6　システム構成図で使用する図の凡例



また，ホストを表す色は，Management Server によって運用管理する場合のホストの分類を表しています。

図 3-7　システム構成図で使用する図の凡例（ホストの分類）



なお，ここで示した以外の凡例については，それぞれの図で示します。

3.3　アプリケーションの構成を検討する

この節では，アプリケーションの構成の検討方法について説明します。

アプリケーションは，構成するコンポーネントの種類によって分類できます。分類ごとに，クライアント側から見たアクセスポイントがあります。ここでは，アプリケーションを構成するコンポーネントの種類と，各アプリケーションでアクセスポイントになるコンポーネントについて説明します。また，接続するリソースの種類に応じたリソースアダプタの種類についても説明します。

3.3.1　アプリケーションの構成とアクセスポイント

ここでは，アプリケーションを構成するコンポーネントの種類と，それぞれの構成の場合のアクセスポイントについて説明します。

アプリケーションを構成するコンポーネントには，次の種類があります。

- サーブレットと JSP
- Session Bean と Entity Bean
- Message-driven Bean

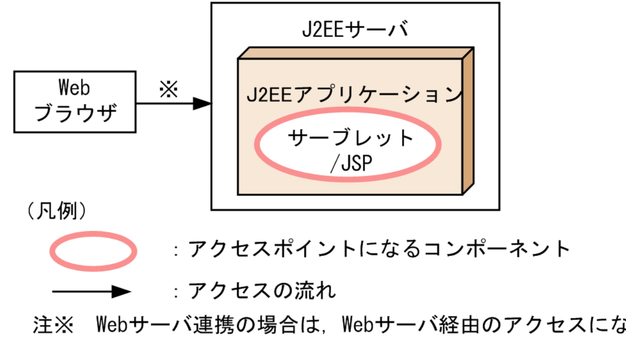
(1)　サーブレットと JSP で構成されるアプリケーション

サーブレットと JSP は，クライアントマシンの Web ブラウザに表示するプレゼンテーションを，動的に生成するためのコンポーネントです。クライアントである Web ブラウザから，HTTP または HTTPS によって Web サーバ経由でアクセスされます。

サーブレットと JSP で構成されるアプリケーションの場合，クライアントから見たアクセスポイントになるコンポーネントは，フロントに配置されたサーブレットまたは JSP になります。

サーブレットと JSP で構成されるアプリケーションを次の図に示します。

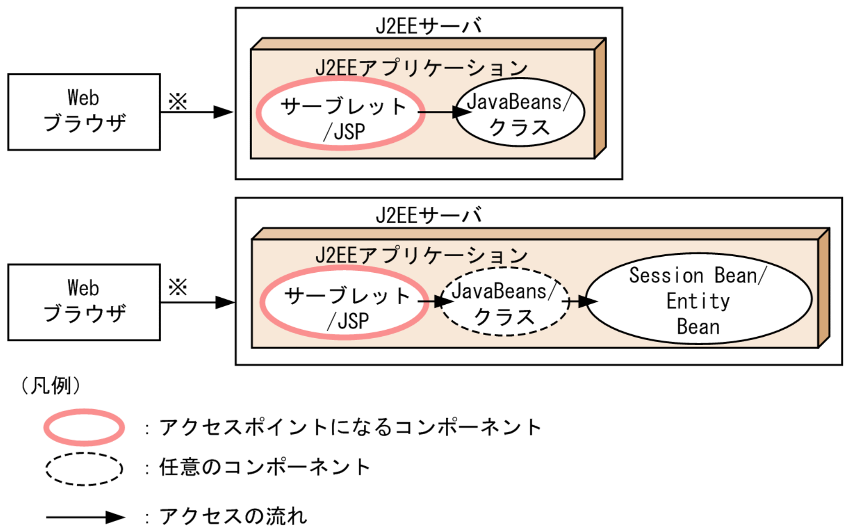
図 3-8　サーブレットと JSP で構成されるアプリケーション



注　これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

サーブレットまたは JSP からは，JavaBeans や Java クラスなど，ほかのコンポーネントを呼び出せます。また，Session Bean や Entity Bean を呼び出すこともできます。この場合も，クライアントから見たアクセスポイントとなるコンポーネントは，フロントに配置したサーブレットまたは JSP になります。

図 3-9　サーブレットと JSP からほかのコンポーネントを呼び出す場合のアクセスポイント



注※　Webサーバ連携の場合は，Webサーバ経由のアクセスになります。

注　これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

このアプリケーションは，主に Web フロントシステムで動作します。

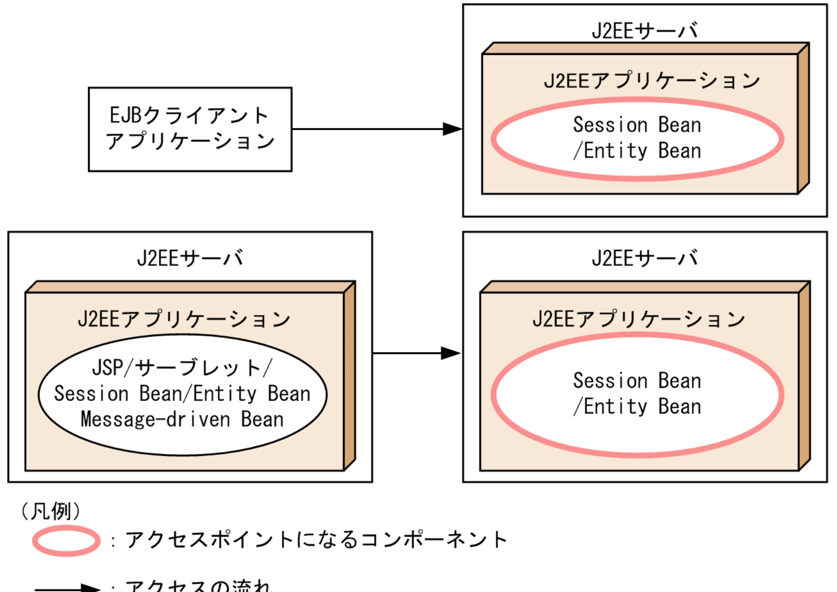
(2)　Session Bean と Entity Bean で構成されるアプリケーション

Session Bean と Entity Bean は，ビジネスロジックを実装するためのコンポーネントです。EJB クライアントから，RMI-IIOP によってアクセスされます。なお，**EJB クライアント**とは，Enterprise Bean を呼び出すコンポーネントの総称です。クライアントマシンで動作する EJB クライアントアプリケーション，ほかの J2EE サーバで動作しているサーブレット，JSP，Session Bean，Entity Bean または Message-driven Bean が該当します。

Session Bean と Entity Bean で構成されるアプリケーションの場合，アクセスポイントになるコンポーネントは，フロントに配置された Session Bean または Entity Bean になります。

Session Bean と Entity Bean で構成されるアプリケーションを次の図に示します。

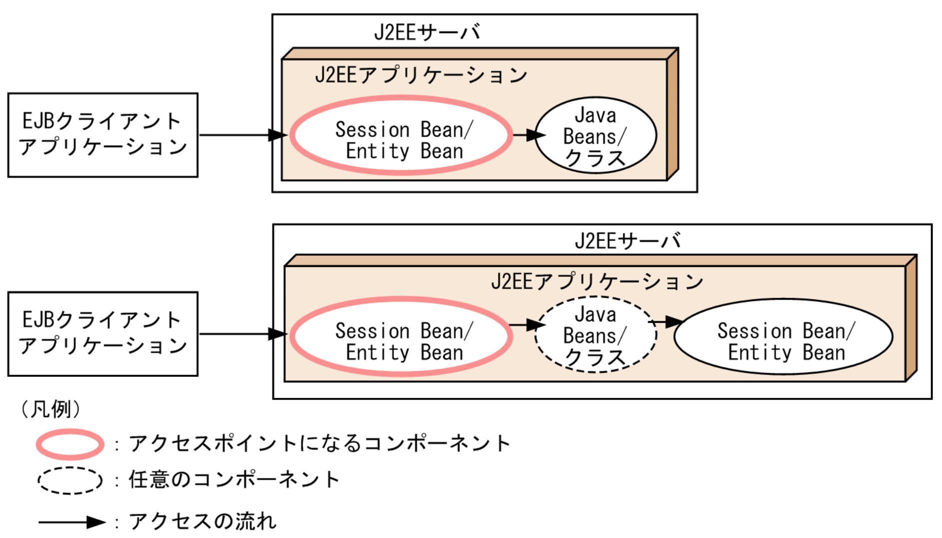
図 3-10　Session Bean と Entity Bean で構成されるアプリケーション



注　これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

Session Bean または Entity Bean からは，JavaBeans や Java クラスなど，ほかのコンポーネントを呼び出せます。また，ほかの Session Bean や Entity Bean を呼び出すこともできます。ただし，この場合も，クライアントから見たアクセスポイントとなるコンポーネントは，フロントに配置した Session Bean または Entity Bean になります。

図 3-11　Session Bean または Entity Bean からほかのコンポーネントを呼び出す場合のアクセスポイント



注　これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

このアプリケーションは，主にバックシステムで動作します。

(3)　Message-driven Bean で構成されるアプリケーション

Message-driven Bean は，メッセージ駆動型のシステムでビジネスロジックを実装するためのコンポーネントです。次のどれかの方法でアクセスされます。

- CJMS プロバイダ経由のアクセス
- TP1/Message Queue および TP1/Message Queue - Access 経由でのアクセス
- データベース（HiRDB または Oracle）および Reliable Messaging 経由でのアクセス
- TP1 インバウンド連携でのアクセス

なお，Message-driven Bean をアクセスポイントとする構成の場合，リソースアダプタとして，次のどれかが必要です。

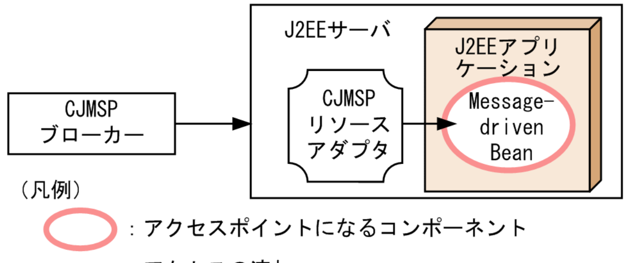
- CJMSP リソースアダプタ※
- TP1/Message Queue - Access
- Reliable Messaging※
  Reliable Messaging は，DB Connector for Reliable Messaging※と組み合わせて使用します。
- TP1 インバウンドアダプタ※

注※　アプリケーションサーバが提供するリソースアダプタです。

Message-driven Bean で構成されるアプリケーションの場合，アクセスポイントになるコンポーネントは，Message-driven Bean になります。

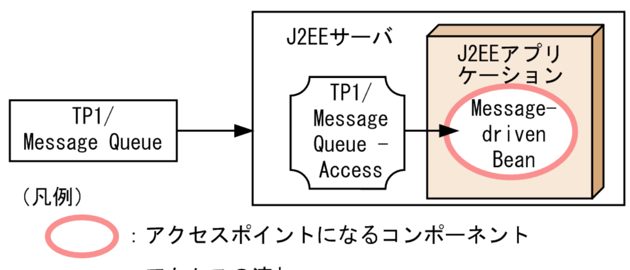
Message-driven Bean で構成されるアプリケーションを次の図に示します。

図 3-12　Message-driven Bean で構成されるアプリケーション（CJMS プロバイダ経由の場合）



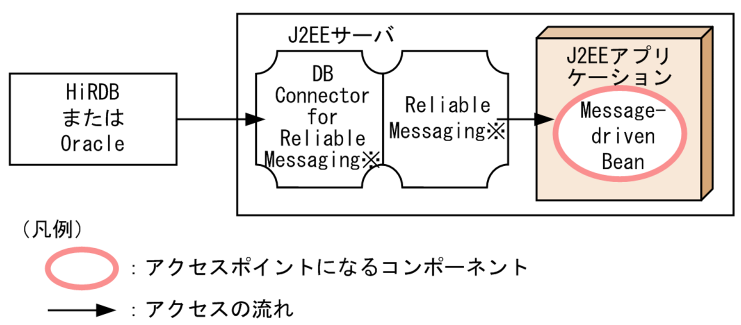
注　これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

図 3-13　Message-driven Bean で構成されるアプリケーション（TP1/Message Queue - Access 経由の場合）



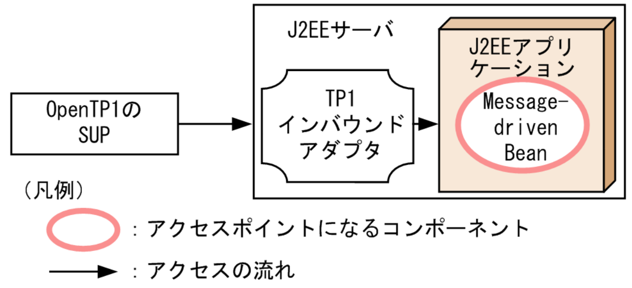
注　これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

図 3-14　Message-driven Bean で構成されるアプリケーション（Reliable Messaging 経由の場合）



注　これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

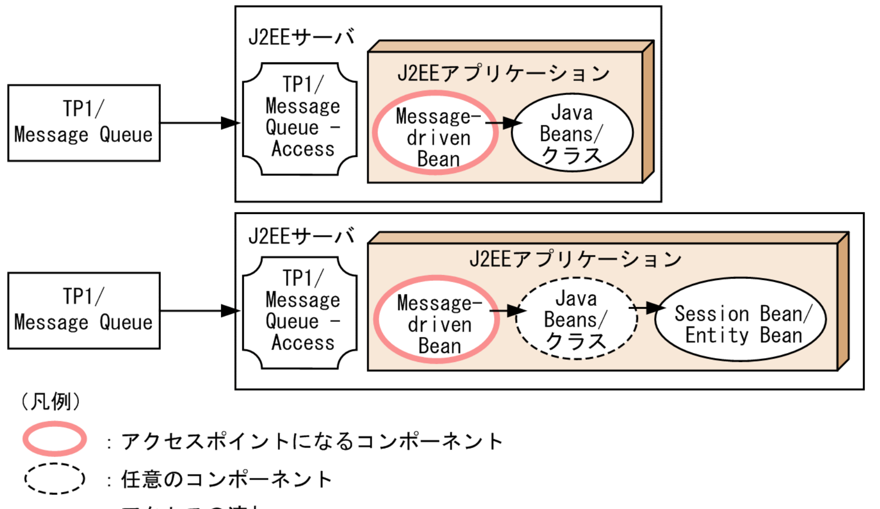
図 3-15　Message-driven Bean で構成されるアプリケーション（TP1 インバウンド連携の場合）



注　これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

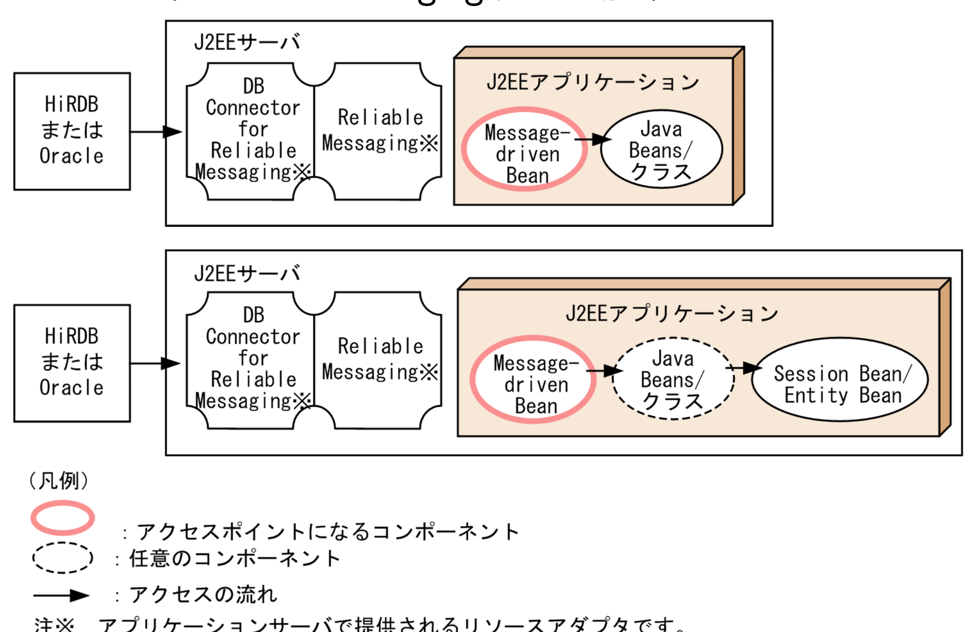
Message-driven Bean からは，JavaBeans や Java クラスなど，ほかのコンポーネントを呼び出せます。また，Session Bean や Entity Bean を呼び出すこともできます。ただし，この場合も，クライアントから見たアクセスポイントとなるコンポーネントは，Message-driven Bean になります。例を示します。

図 3-16　Message-driven Bean からほかのコンポーネントを呼び出す場合のアクセスポイント（TP1/Message Queue - Access 経由の場合）



注　これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

図 3-17　Message-driven Bean からほかのコンポーネントを呼び出す場合のアクセスポイント（Reliable Messaging 経由の場合）



注　これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

このアプリケーションは，主にバックシステムで動作します。

3.3.2　リソースの種類とリソースアダプタ

アプリケーションサーバのシステムでは，次のリソースと接続できます。

- データベース（HiRDB，Oracle，SQL Server または XDM/RD E2）
- OpenTP1 の TP1/Message Queue
- OpenTP1 の SPP
- OpenTP1 の SUP
- CJMS プロバイダの CJMSP ブローカー
- メールコンフィグレーション
- JavaBeans リソース

OpenTP1 の SUP とは，Inbound で接続します。なお，メールコンフィグレーションまたは JavaBeans リソースを使用する場合，リソースアダプタは不要です。

ここでは，アプリケーションが接続するリソースの種類ごとに，使用するリソースアダプタについて説明します。

ポイント

アプリケーションサーバでは，Connector 1.0 または Connector 1.5 仕様に準拠したリソースアダプタを利用できます。

ここで説明する以外のリソースアダプタについては，ご使用のリソースアダプタの説明をご確認ください。

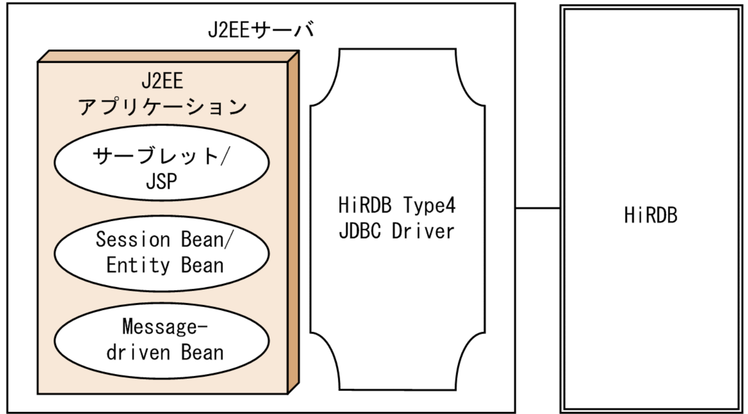
(1) データベースと接続するためのリソースアダプタ（JDBC インタフェースを使用する場合）

JDBC インタフェースを使用してデータベースと接続する場合，リソースアダプタとして DB Connector を使用します。DB Connector を使用すると，サーブレット，JSP，Session Bean，Entity Bean または Message-driven Bean から，JDBC インタフェースを使用してデータベースにアクセスできます。

なお，アプリケーションサーバがリソースアダプタを使用してアクセスできるデータベースは，HiRDB，Oracle，SQL Server または XDM/RD E2 です。リソースアダプタを使用してデータベースにアクセスする場合の構成を，データベースの種類ごとに図に示します。

- HiRDB にアクセスする場合の構成
  HiRDB に接続する場合，J2EE サーバと同じマシンに，HiRDB Type4 JDBC Driver が必要となります。
  HiRDB にアクセスする場合の構成を次の図に示します。

図 3-18　リソースアダプタを使用して HiRDB にアクセスする場合の構成



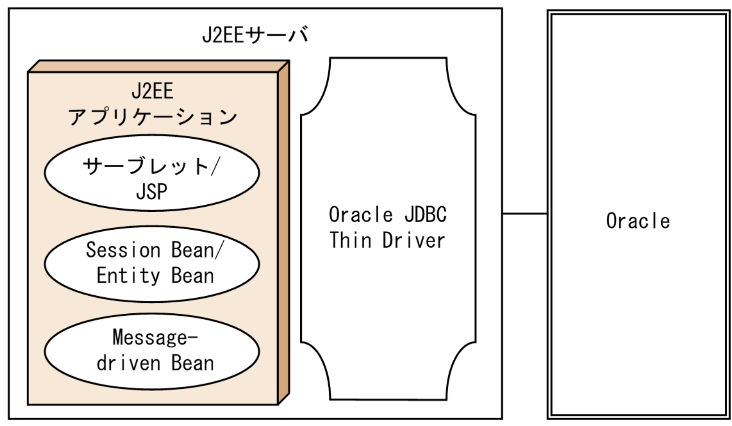
注　凡例については，「3.2　システム構成の説明について」を参照してください。

- Oracle にアクセスする場合の構成

Oracle に接続する場合，J2EE サーバと同じマシンに，Oracle JDBC Thin Driver が必要となります。

Oracle にアクセスする場合の構成を次の図に示します。

図 3-19　リソースアダプタを使用して Oracle にアクセスする場合の構成



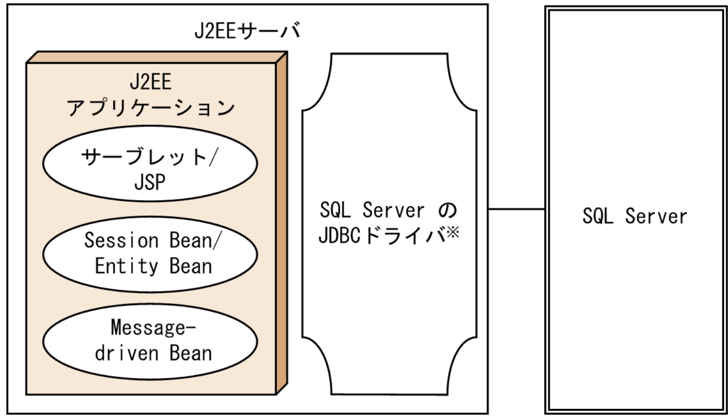
注　凡例については，「3.2　システム構成の説明について」を参照してください。

- SQL Server にアクセスする場合の構成

SQL Server に接続する場合，J2EE サーバと同じマシンに，SQL Server の JDBC ドライバが必要となります。

SQL Server にアクセスする場合の構成を次の図に示します。

図 3-20　リソースアダプタを使用して SQL Server にアクセスする場合の構成



注※　SQL Server 2005に接続する場合は，SQL Server 2005 JDBC Driverになります。

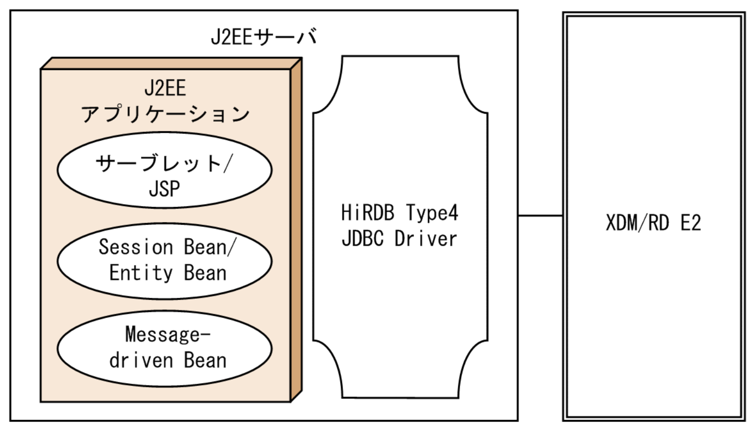
注　凡例については，「3.2　システム構成の説明について」を参照してください。

- XDM/RD E2 にアクセスする場合の構成

XDM/RD E2 に接続する場合，J2EE サーバと同じマシンに，HiRDB Type4 JDBC Driver が必要となります。

XDM/RD E2 にアクセスする場合の構成を次の図に示します。

図 3-21　リソースアダプタを使用して XDM/RD E2 にアクセスする場合の構成



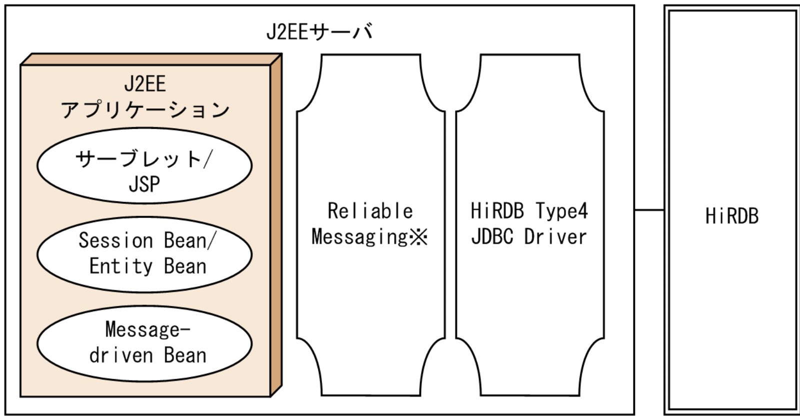
注　凡例については，「3.2　システム構成の説明について」を参照してください。

(2)　データベースと接続するためのリソースアダプタ（JMS インタフェースを使用する場合）

JMS インタフェースを使用してデータベースと接続する場合，リソースアダプタとして DB Connector for Reliable Messaging を使用します。DB Connector for Reliable Messaging は，Reliable Messaging と連携するためのリソースアダプタです。Reliable Messaging を使用すると，サーブレット，JSP，Session Bean，Entity Bean または Message-driven Bean から，JMS インタフェースを使用してデータベース上に実現したキューにアクセスできます。この場合，J2EE サーバ上では，Reliable Messaging のライブラリが動作して，キューのデータはデータベースに保存されます。なお，Reliable Messaging はアプリケーションサーバの構成ソフトウェアです。

Reliable Messaging を使用してデータベースにアクセスする場合の構成を，HiRDB にアクセスする場合と，Oracle にアクセスする場合に分けて，図に示します。

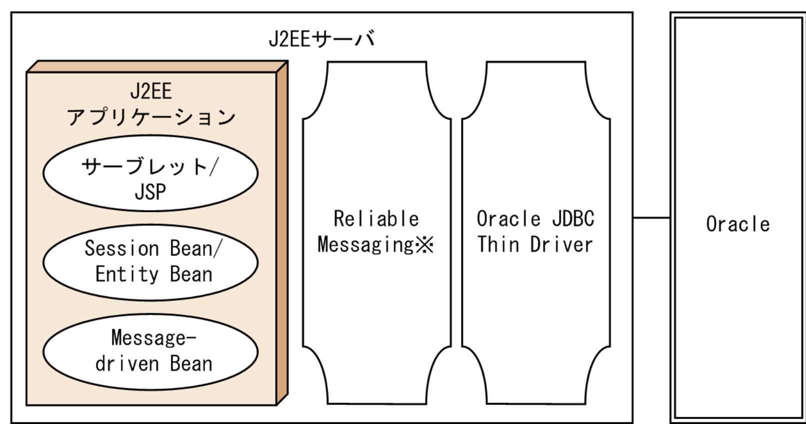
図 3-22　Reliable Messaging を使用して HiRDB にアクセスする場合の構成



注※　アプリケーションサーバで提供されるリソースアダプタです。

注　凡例については，「3.2　システム構成の説明について」を参照してください。

図 3-23　Reliable Messaging を使用して Oracle にアクセスする場合の構成



注※　アプリケーションサーバで提供されるリソースアダプタです。

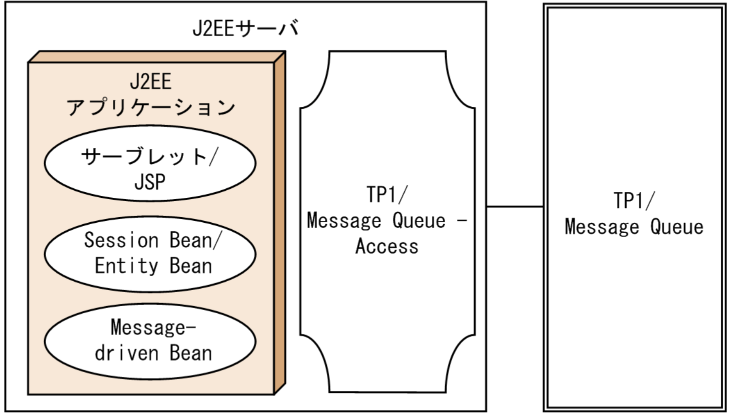
注　凡例については，「3.2　システム構成の説明について」を参照してください。

(3)　OpenTP1 の TP1/Message Queue に接続するためのリソースアダプタ

OpenTP1 のメッセージキューイング機能である TP1/Message Queue と接続する場合，リソースアダプタとして **TP1/Message Queue - Access** を使用します。TP1/Message Queue - Access を使用すると，サーブレット，JSP，Session Bean，Entity Bean または Message-driven Bean から，JMS インタフェースを使用して TP1/Message Queue にアクセスできます。この場合，J2EE サーバと同じマシンに，TP1/Message Queue - Access が必要となります。

リソースアダプタを使用して TP1/Message Queue にアクセスする場合の構成を，次の図に示します。

図 3-24　リソースアダプタを使用して TP1/Message Queue にアクセスする場合の構成



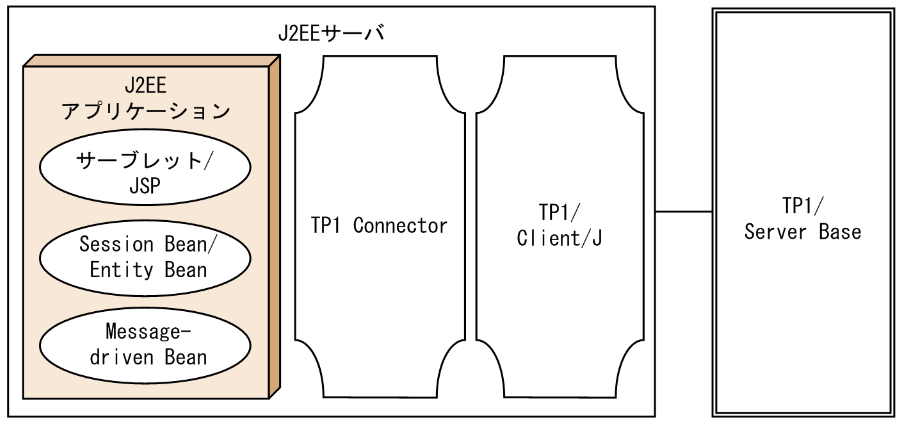
注　凡例については，「3.2　システム構成の説明について」を参照してください。

(4)　OpenTP1 の SPP と接続するためのリソースアダプタ

OpenTP1 の SPP（サービス提供プログラム：Service Providing Program）と接続する場合，リソースアダプタとして **TP1 Connector** および **TP1/Client/J** を使用します。TP1 Connector および TP1/Client/J を使用すると，サーブレット，JSP，Session Bean，Entity Bean または Message-driven Bean から，OpenTP1 の SPP にアクセスできます。このとき，J2EE サーバと同じマシンに，TP1 Connector および TP1/Client/J が必要となります。

リソースアダプタを使用して OpenTP1 の SPP にアクセスする場合の構成を，次の図に示します。

図 3-25　リソースアダプタを使用して OpenTP1 の SPP にアクセスする場合の構成



注　凡例については，「3.2　システム構成の説明について」を参照してください。

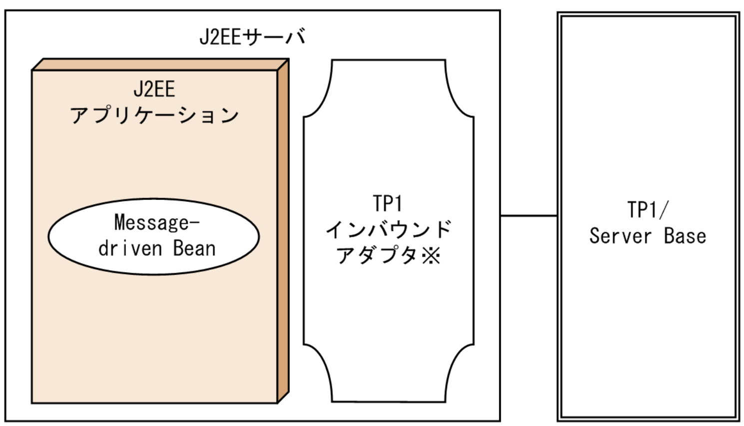
(5)　OpenTP1 の SUP と接続するためのリソースアダプタ

TP1 インバウンド連携機能を利用して OpenTP1 の SUP（サービス利用プログラム：Service Using Program）と接続する場合，リソースアダプタとして **TP1 インバウンドアダプタ**を使用します。TP1 インバウンド連携機能を利用すると，OpenTP1 の SUP から，J2EE サーバ上で動作する Message-driven

Bean に Inbound でアクセスできます。このとき，J2EE サーバと同じマシンに，TP1 インバウンドアダプタが必要となります。

リソースアダプタを使用して OpenTP1 の SUP から Inbound でアクセスする場合の構成を，次の図に示します。

図 3-26　リソースアダプタを使用して OpenTP1 の SUP から Inbound でアクセスする場合の構成



注※　アプリケーションサーバで提供されるリソースアダプタです。

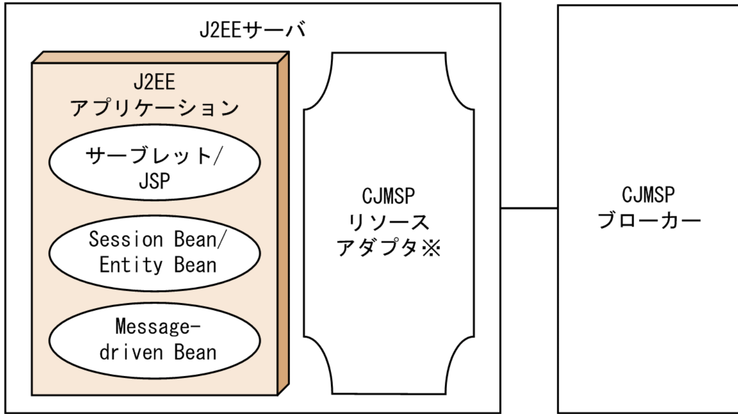
注　凡例については，「3.2　システム構成の説明について」を参照してください。

(6)　CJMS プロバイダの CJMSP ブローカーと接続するためのリソースアダプタ

CJMS プロバイダの CJMSP ブローカーと接続する場合，リソースアダプタとして **CJMSP リソースアダプタ**を使用します。CJMSP リソースアダプタを使用すると，サーブレット，JSP，Session Bean，Entity Bean または Message-driven Bean から，JMS インタフェースを使用して CJMSP ブローカーにアクセスできます。このとき，J2EE サーバと同じマシンに，CJMSP リソースアダプタが必要となります。

リソースアダプタを使用して CJMSP ブローカーにアクセスする場合の構成を，次の図に示します。

図 3-27　リソースアダプタを使用して CJMSP ブローカーにアクセスする場合の構成



注※　アプリケーションサーバで提供されるリソースアダプタです。

注　凡例については，「3.2　システム構成の説明について」を参照してください。

3.4 クライアントとサーバの構成を検討する

この節では，クライアントとサーバの構成の種類と，それぞれの場合に各マシンに配置するプロセスについて説明します。また，それぞれの構成の特徴についても説明します。

クライアントとサーバの構成を検討するためには，処理を呼び出す側であるクライアントと呼び出される側であるサーバの対応を決めて，サーバ側で動作するアプリケーションのアクセスポイントを明確にする必要があります。

ここでは，次のコンポーネントをアクセスポイントとするクライアントとサーバの構成について説明します。なお，サーブレットと JSP をアクセスポイントにする構成については，Web サーバ連携の場合と Web サーバを経由しないで J2EE サーバの NIO HTTP サーバに直接アクセスする場合に分けて説明します。

- サーブレットと JSP
- Session Bean と Entity Bean
- CTM を使用する場合の Stateless Session Bean

以降，アプリケーションサーバを配置したマシンを**アプリケーションサーバマシン**，クライアントとして使用するマシンを**クライアントマシン**といいます。

3.4.1 サーブレットと JSP をアクセスポイントに使用する構成（Web サーバ連携の場合）

サーブレットと JSP がアクセスポイントになる，Web ベースのシステム構成について説明します。この構成を **Web クライアント構成**といいます。Web フロントシステムで使用できる構成です。ここでは，Web サーバ連携の場合の構成について説明します。

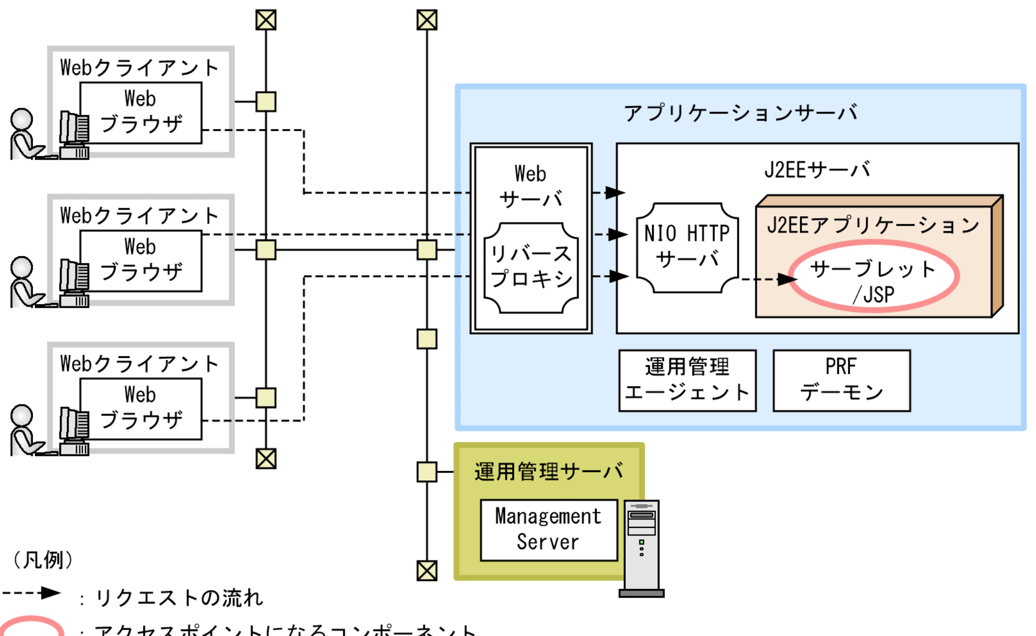
また，この構成でインターネットに接続する場合，セキュリティの観点から，アプリケーションサーバと同じマシンに配置する Web サーバとは別に，DMZ にリバースプロキシの機能を持つ Web サーバを配置することをお勧めします。詳細は，マニュアル「アプリケーションサーバ 機能解説 セキュリティ管理機能編」の「3.3　DMZ へのリバースプロキシの配置を検討する」を参照してください。

(1) システム構成の特徴

Web フロントシステムで，Web ブラウザから送信されたリクエストをアプリケーションサーバで処理する場合に適用されるシステム構成です。

最も基本的な Web クライアント構成は，クライアントマシンと 1 台のアプリケーションサーバマシンによって構築できます。1 台のアプリケーションサーバマシンで構築する Web クライアント構成の例を次の図に示します。

図 3-28　1 台のアプリケーションサーバマシンで構築する Web クライアント構成の例（Web サーバ連携の場合）



これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

特徴

クライアントで使用するソフトウェアは Web ブラウザだけです。なお，NIO HTTP サーバにアクセスする場合，次の点に留意してください。

- NIO HTTP サーバでは，HTTPS はサポートされていません。このため，SSL を使用する場合には，プロキシサーバをフロントに配置してプロキシモジュールを組み込んだ Web サーバの SSL 機能を使用するか，または SSL アクセラレータをフロントに配置してください。

リクエストの流れ

アクセスポイントであるサーブレットと JSP は，J2EE サーバ上で動作します。これらのコンポーネントは，Web ブラウザから Web サーバ経由でアクセスされます。

(2)　それぞれのマシンに必要なプロセスとソフトウェア

それぞれのマシンに必要なソフトウェアとプロセスについて説明します。なお，リソースと接続するために必要なプロセスについては，「3.6　トランザクションの種類を検討する」を参照してください。

(a)　アプリケーションサーバマシン

アプリケーションサーバマシンには，Application Server をインストールする必要があります。

必要なプロセスは次のとおりです。

- Web サーバ
- J2EE サーバ

- 運用管理エージェント
- PRF デーモン

なお，Application Server には，Web サーバである HTTP Server が含まれています。

(b) 運用管理サーバマシン

運用管理サーバマシンには，Application Server をインストールする必要があります。

起動するプロセスは次のとおりです。

- Management Server

(c) Web クライアントマシン

Web クライアントマシンには，Web ブラウザが必要です。

3.4.2 サーブレットと JSP をアクセスポイントに使用する構成（NIO HTTP サーバに直接アクセスする場合）

サーブレットと JSP がアクセスポイントになる，Web ベースのシステム構成について説明します。ここでは，NIO HTTP サーバに直接アクセスする場合の構成について説明します。

なお，この構成でインターネットに接続する場合，セキュリティの観点から，DMZ にリバースプロキシの機能を持つ Web サーバを配置してください。詳細は，マニュアル「アプリケーションサーバ 機能解説 セキュリティ管理機能編」の「3.3　DMZ へのリバースプロキシの配置を検討する」を参照してください。

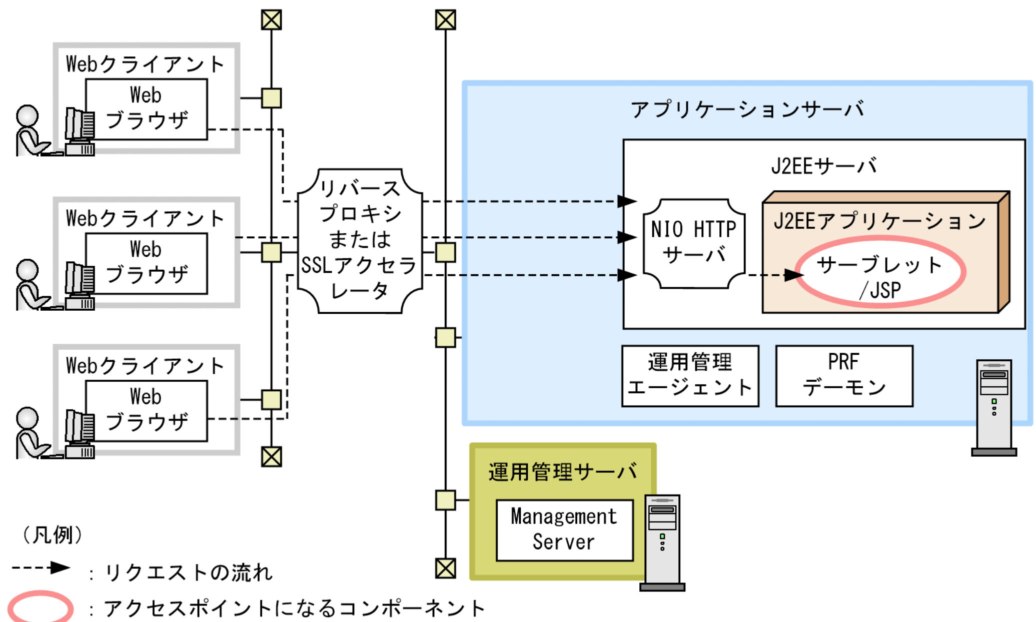
(1) システム構成の特徴

Web フロントシステムで，Web ブラウザから送信されたリクエストをアプリケーションサーバで処理する場合に適用されるシステム構成です。

最も基本的な Web クライアント構成は，クライアントマシンと 1 台のアプリケーションサーバマシンによって構築できます。NIO HTTP サーバに直接アクセスする場合，J2EE サーバのフロントに Web サーバを起動する必要がありません。J2EE サーバ上の HTTP サーバの機能を使用します。

NIO HTTP サーバに直接アクセスする場合の Web クライアント構成の例を次の図に示します。

図 3-29　1 台のアプリケーションサーバマシンで構築する Web クライアント構成の例（NIO HTTP サーバに直接アクセスする場合）



これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

特徴

クライアントで使用するソフトウェアは Web ブラウザだけです。

Web ブラウザから Web サーバを経由しないで直接 J2EE サーバにアクセスできるため，性能上のメリットがあります。また，Web サーバを起動する必要がないため，運用が簡易になります。

なお，NIO HTTP サーバに直接アクセスする場合，次の点に留意してください。

- NIO HTTP サーバでは，Web サーバとして動作するための最小限の機能だけがサポートされています。このため，Web サーバのさまざまな機能を必要とする場合は，この構成は適していません。Web サーバと連携する構成を選択してください。NIO HTTP サーバで使用できる機能については，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(Web コンテナ)」の「7.　NIO HTTP サーバ」を参照してください。
- NIO HTTP サーバでは，HTTPS はサポートされていません。このため，SSL を使用する場合には，プロキシサーバをフロントに配置してプロキシモジュールを組み込んだ Web サーバの SSL 機能を使用するか，または，SSL アクセラレータをフロントに配置してください。
- インターネットに接続するシステムでは，セキュリティ上の観点から，必ずリバースプロキシをフロントに配置してください。リバースプロキシを配置する構成については，マニュアル「アプリケーションサーバ 機能解説 セキュリティ管理機能編」の「3.3　DMZ へのリバースプロキシの配置を検討する」を参照してください。

リクエストの流れ

アクセスポイントであるサーブレットと JSP は，J2EE サーバ上で動作します。これらのコンポーネントは，Web ブラウザから直接アクセスされます。

(2)　それぞれのマシンに必要なプロセスとソフトウェア

それぞれのマシンに必要なソフトウェアとプロセスについて説明します。なお，リソースと接続するために必要なプロセスについては，「3.6　トランザクションの種類を検討する」を参照してください。

(a)　アプリケーションサーバマシン

アプリケーションサーバマシンには，Application Server をインストールする必要があります。

必要なプロセスは次のとおりです。

- J2EE サーバ
- 運用管理エージェント
- PRF デーモン

(b)　運用管理サーバマシン

運用管理サーバマシンには，Application Server をインストールする必要があります。

起動するプロセスは次のとおりです。

- Management Server

(c)　Web クライアントマシン

Web クライアントマシンには，Web ブラウザが必要です。

3.4.3　Session Bean と Entity Bean をアクセスポイントに使用する構成

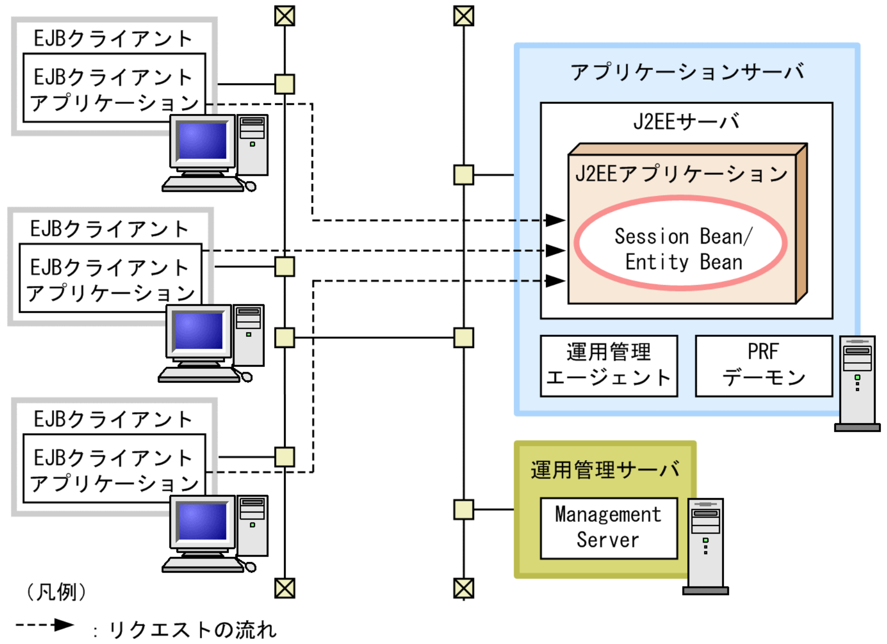
Session Bean と Entity Bean がアクセスポイントになり，クライアントで EJB クライアントアプリケーションを動作させるシステム構成について説明します。この構成を，**EJB クライアント構成**といいます。

(1)　システム構成の特徴

最も基本的な EJB クライアント構成は，クライアントマシンと 1 台のアプリケーションサーバマシンによって構築できます。

1 台のアプリケーションサーバマシンで構築する EJB クライアント構成の例を次の図に示します。

図 3-30　1 台のアプリケーションサーバマシンで構築する EJB クライアント構成の例



これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

特徴

Windows の場合は，Client を使用してクライアントマシンの環境を構築できます。

リクエストの流れ

アクセスポイントである Session Bean と Entity Bean は，J2EE サーバ上で動作します。

EJB クライアントアプリケーションからのリクエストは，RMI-IIOP によってアクセスポイントに送られ，Session Bean と Entity Bean を呼び出します。このとき，EJB クライアントアプリケーションは，J2EE サーバ内で動作しているネーミングサービスから名前を検索（ルックアップ）して，Session Bean と Entity Bean にアクセスします。

(2)　それぞれのマシンに必要なソフトウェアと起動するプロセス

それぞれのマシンに必要なソフトウェアと起動するプロセスについて説明します。なお，リソースに接続するためのプロセスについては，「3.6　トランザクションの種類を検討する」を参照してください。

(a)　アプリケーションサーバマシン

アプリケーションサーバマシンには，Application Server をインストールする必要があります。

起動するプロセスは次のとおりです。

- J2EE サーバ
- 運用管理エージェント
- PRF デーモン

(b)　運用管理サーバマシン

運用管理サーバマシンには，Application Server をインストールする必要があります。

起動するプロセスは次のとおりです。

- Management Server

(c)　EJB クライアントマシン

EJB クライアントマシンには，Application Server または Client（Windows の場合）をインストールする必要があります。

起動するプロセスは，EJB クライアントアプリケーションのプロセスです。

3.4.4　CTM を使用する場合に Stateless Session Bean をアクセスポイントに使用する構成

CTM を使用する場合の，Stateless Session Bean をアクセスポイントとするシステム構成について説明します。

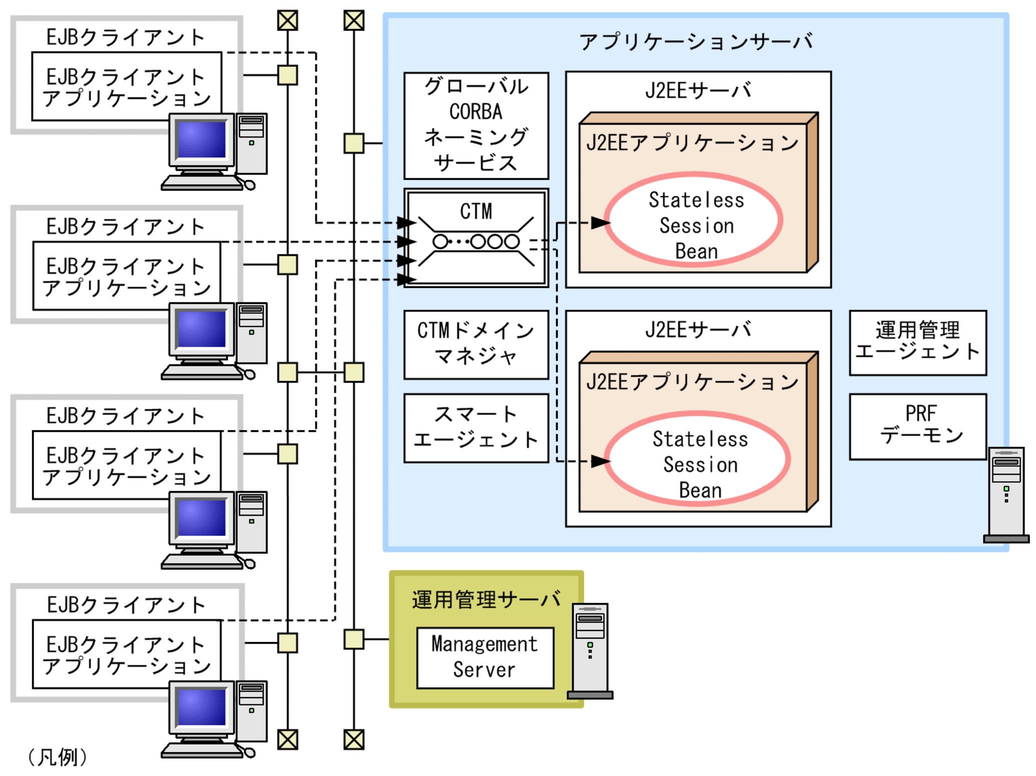
> **参考**
>
> ここでは，クライアントとして EJB クライアントアプリケーションを使用する EJB クライアント構成の例を示します。このほか，CTM ゲートウェイ機能を使用すると，EJB クライアント以外のクライアントアプリケーションから J2EE サーバ上の EJB アプリケーションを直接呼び出す構成も実現できます。これらの構成については，「3.13.3　CTM ゲートウェイ機能を利用して EJB クライアント以外から Stateless Session Bean を呼び出す構成」を参照してください。

(1)　システム構成の特徴

EJB クライアント構成の一つです。アクセスポイントである Stateless Session Bean に対して，CTM によってスケジューリングされたリクエストが送信されます。

CTM を使用する場合の EJB クライアント構成の例を次の図に示します。

図 3-31　CTM を使用する場合の EJB クライアント構成の例



これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

特徴

- CTM によってリクエストをスケジューリングすることで，サービス閉塞を実行したり同時実行数を制御したりするシステム運用ができるようになります。
- J2EE サーバを二つ以上起動することで，一つの J2EE サーバにトラブルが発生した場合に，縮退運転をしてシステムの稼働を続けられます。
- Windows の場合は，Client を使用してクライアントマシンの環境を構築できます。

リクエストの流れ

アクセスポイントである Stateless Session Bean は，J2EE サーバ上で動作します。

EJB クライアントアプリケーションからのリクエストは，CTM 経由で送られ，Stateless Session Bean を呼び出します。このとき，EJB クライアントアプリケーションは，グローバル CORBA ネーミングサービスから名前をルックアップして，Stateless Session Bean にアクセスします。

(2)　それぞれのマシンに必要なソフトウェアと起動するプロセス

それぞれのマシンに必要なソフトウェアと起動するプロセスについて説明します。なお，リソースに接続するためのプロセスについては，「3.6　トランザクションの種類を検討する」を参照してください。

(a)　アプリケーションサーバマシン

アプリケーションサーバマシンには，Application Server をインストールする必要があります。

起動するプロセスは次のとおりです。

- J2EE サーバ
- 運用管理エージェント
- PRF デーモン
- グローバル CORBA ネーミングサービス
- CTM のプロセス群（CTM デーモンおよび CTM レギュレータ）
- CTM ドメインマネジャ
- スマートエージェント

(b)　運用管理サーバマシン

運用管理サーバマシンには，Application Server をインストールする必要があります。

起動するプロセスは次のとおりです。

- Management Server

(c)　EJB クライアントマシン

EJB クライアントマシンには，Application Server または Client（Windows の場合）をインストールする必要があります。

起動するプロセスは EJB クライアントアプリケーションのプロセスです。

3.5 サーバ間での連携を検討する

この節では，複数のアプリケーションサーバ上の J2EE サーバで動作するアプリケーションを連携させる構成の種類と，それぞれの場合に各マシンに配置するプロセスについて説明します。また，それぞれの構成の特徴についても説明します。

なお，複数のアプリケーションサーバで構成するシステムの場合，呼び出し元になるアプリケーションが動作しているアプリケーションサーバを**クライアント側のアプリケーションサーバ**，呼び出されるアプリケーションが動作しているアプリケーションサーバを**サーバ側のアプリケーションサーバ**といいます。

ここでは，次に示す 2 種類のサーバ間連携の構成について説明します。

- Session Bean と Entity Bean を呼び出すサーバ間連携
- CTM 経由で Stateless Session Bean を呼び出すサーバ間連携

3.5.1 Session Bean と Entity Bean を呼び出すサーバ間連携

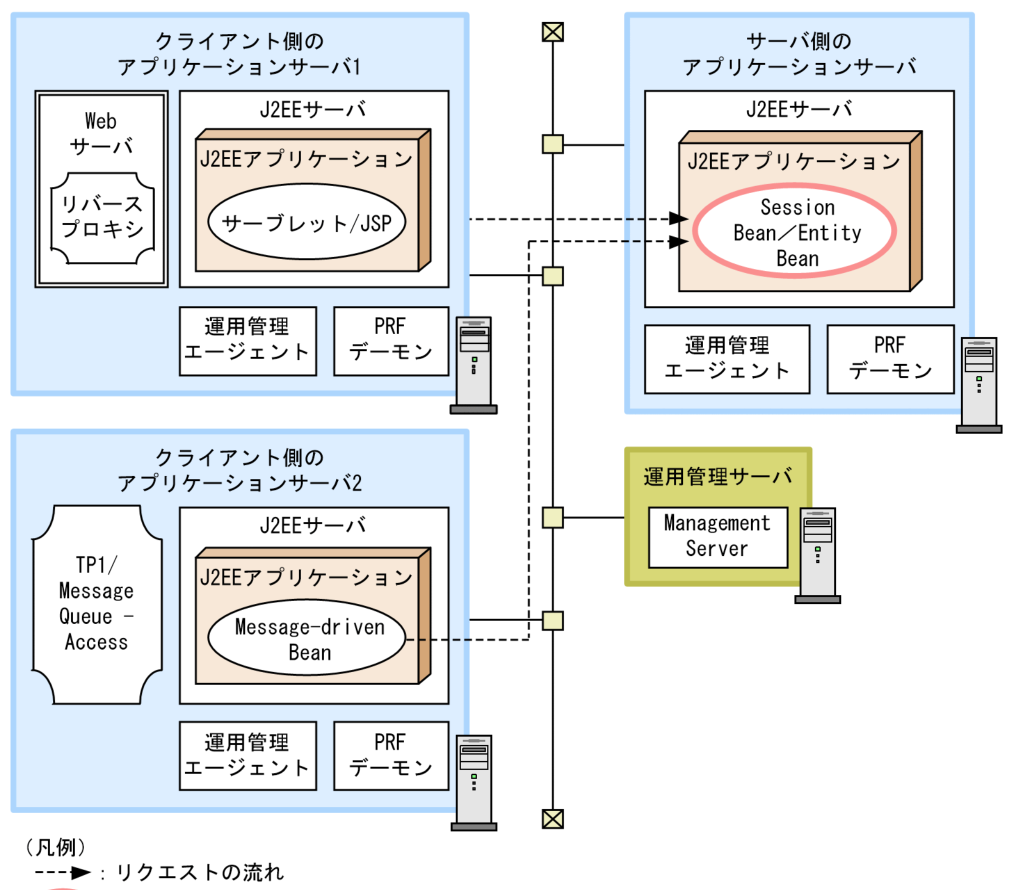
ほかのアプリケーションサーバ上の J2EE サーバから Session Bean または Entity Bean を呼び出す場合の構成について説明します。

(1) システム構成の特徴

クライアント側のアプリケーションサーバから，サーバ側のアプリケーションサーバを呼び出す構成です。クライアント側のアプリケーションサーバでは，サーブレット，JSP，Entity Bean，Session Bean または Message-driven Bean で構成されるアプリケーションが動作します。サーバ側のアクセスポイントになるコンポーネントは，Session Bean または Entity Bean です。クライアント側のアプリケーションからの呼び出しは，RMI-IIOP で実行されます。

Session Bean と Entity Bean を呼び出すサーバ間連携の構成の例を，次の図に示します。

図 3-32　Session Bean と Entity Bean を呼び出すサーバ間連携の構成の例



これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

特徴

- アプリケーション間連携で使用できる構成です。複数のクライアントアプリケーションから，サーバアプリケーションを呼び出す場合などに適用できます。
- システム間連携の場合にも使用できます。すでに構築されたクライアント側システムとサーバ側システムをこのシステム構成にすることで，アプリケーションをシステム間で呼び出せます。

リクエストの流れ

サーバ側のアクセスポイントである Session Bean または Entity Bean へのリクエストは，クライアント側の J2EE サーバから送られます。

このとき，クライアント側では，サーバ側のアプリケーションサーバの J2EE サーバ内でインプロセスで起動されている CORBA ネーミングサービスから名前をルックアップして，Session Bean または Entity Bean にアクセスします。

(2)　それぞれのマシンに必要なソフトウェアと起動するプロセス

Session Bean または Entity Bean を呼び出すサーバ間連携の場合に，それぞれのマシンに必要なソフトウェアと起動するプロセスについて説明します。なお，リソースと接続するために必要なプロセスについては，「3.6　トランザクションの種類を検討する」を参照してください。

(a)　クライアント側のアプリケーションサーバマシン（サーブレット／JSP の実行環境）

サーブレットおよび JSP を動作させるクライアント側のアプリケーションサーバマシンには，Application Server をインストールする必要があります。

起動するプロセスは次のとおりです。

- Web サーバ
- J2EE サーバ
- 運用管理エージェント
- PRF デーモン

(b)　クライアント側のアプリケーションサーバマシン（Message-driven Bean の実行環境）

Message-driven Bean を動作させるクライアント側のアプリケーションサーバマシンには，Application Server をインストールする必要があります。また，リソースアダプタとして TP1/Message Queue - Access を使用する場合は，TP1/Message Queue - Access をインストールする必要があります。なお，Reliable Messaging を使用する場合は，アプリケーションサーバに含まれているため，個別にインストールする必要はありません。

起動するプロセスは次のとおりです。

- J2EE サーバ
- 運用管理エージェント
- PRF デーモン

(c)　サーバ側のアプリケーションサーバマシン

サーバ側のアプリケーションサーバマシンには，Application Server をインストールする必要があります。

起動するプロセスは次のとおりです。

- J2EE サーバ
- 運用管理エージェント
- PRF デーモン

(d)　運用管理サーバマシン

運用管理サーバマシンには，Application Server をインストールする必要があります。

起動するプロセスは次のとおりです。

- Management Server

3.5.2　CTM 経由で Stateless Session Bean を呼び出すサーバ間連携

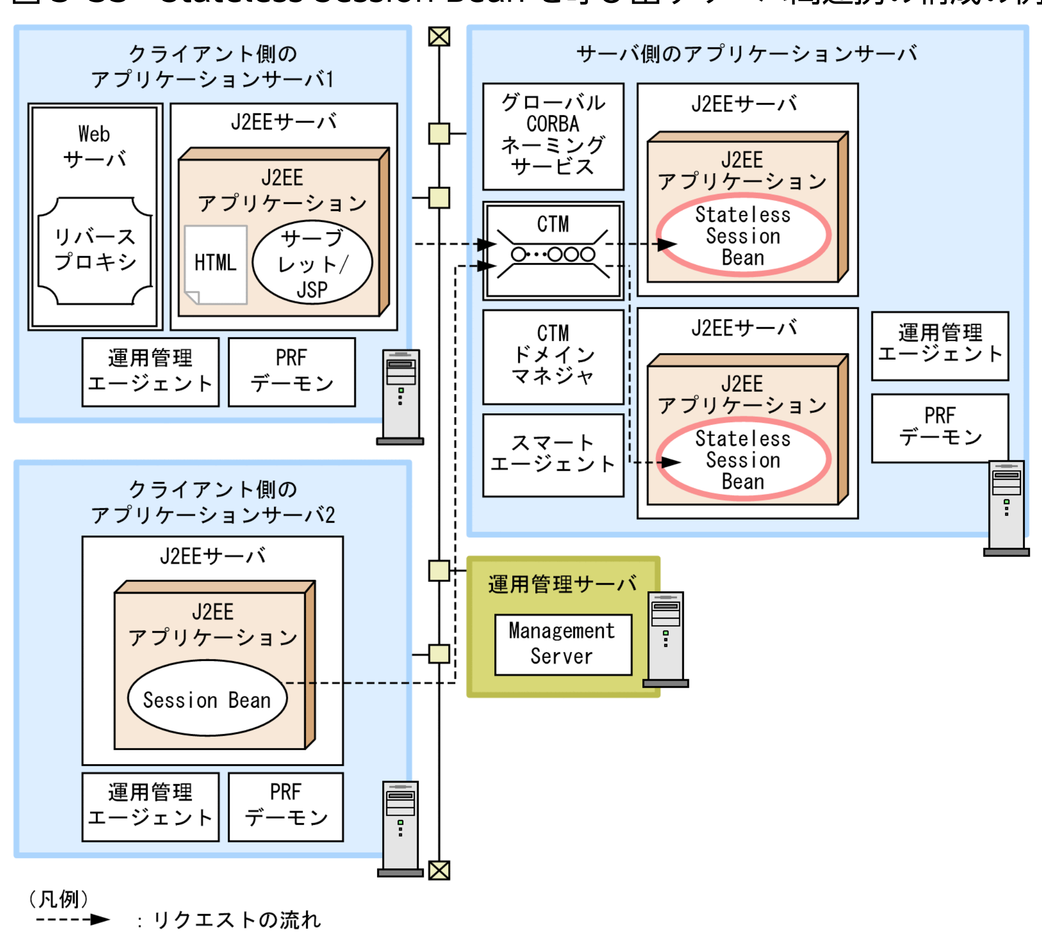
CTM を使用する場合に，ほかのアプリケーションサーバ上の J2EE サーバから CTM 経由で Stateless Session Bean を呼び出すときの構成について説明します。

(1)　システム構成の特徴

クライアント側のアプリケーションサーバから，CTM 経由でサーバ側のアプリケーションサーバを呼び出す構成です。クライアント側のアプリケーションサーバでは，サーブレット，JSP，Entity Bean，Session Bean または Message-driven Bean で構成されるアプリケーションが動作します。サーバ側のアクセスポイントになるコンポーネントは，Stateless Session Bean です。クライアント側のアプリケーションからの呼び出しは，RMI-IIOP で実行されます。

CTM 経由で Stateless Session Bean を呼び出すサーバ間連携の構成の例を次の図に示します。

図 3-33　Stateless Session Bean を呼び出すサーバ間連携の構成の例



これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

特徴

- アプリケーション間連携で使用できる構成です。複数のクライアントアプリケーションから，サーバアプリケーションを呼び出す場合などに適用できます。
- システム間連携の場合にも使用できます。すでに構築されたクライアント側システムとサーバ側システムをこのシステム構成にすることで，アプリケーションをシステム間で呼び出せます。

リクエストの流れ

サーバ側のアクセスポイントである Stateless Session Bean へのリクエストは，クライアント側の J2EE サーバから CTM 経由で送られます。

このとき，クライアント側では，サーバ側のアプリケーションサーバのグローバル CORBA ネーミングサービスから名前をルックアップして，CTM 経由で Stateless Session Bean にアクセスします。

CTM を経由したリクエストは，CTM によって J2EE サーバ上で動作する Stateless Session Bean に適切に振り分けられます。

(2)　それぞれのマシンに必要なソフトウェアと起動するプロセス

CTM 経由で Stateless Session Bean を呼び出すサーバ間連携の場合に，それぞれのマシンに必要なソフトウェアと起動するプロセスについて説明します。なお，リソースと接続するために必要なプロセスについては，「3.6　トランザクションの種類を検討する」を参照してください。

(a)　クライアント側のアプリケーションサーバマシン（サーブレット／JSP の実行環境）

サーブレットおよび JSP を動作させるクライアント側のアプリケーションサーバマシンには，Application Server をインストールする必要があります。

起動するプロセスは次のとおりです。

- Web サーバ
- J2EE サーバ
- 運用管理エージェント
- PRF デーモン

(b)　クライアント側のアプリケーションサーバマシン（Session Bean の実行環境）

Session Bean を動作させるクライアント側のアプリケーションサーバマシンには，Application Server をインストールする必要があります。

起動するプロセスは次のとおりです。

- J2EE サーバ
- 運用管理エージェント
- PRF デーモン

(c)　サーバ側のアプリケーションサーバマシン

サーバ側のアプリケーションサーバマシンには，Application Server をインストールする必要があります。

起動するプロセスは次のとおりです。

- J2EE サーバ
- 運用管理エージェント
- PRF デーモン
- グローバル CORBA ネーミングサービス
- CTM のプロセス群（CTM デーモンおよび CTM レギュレータ）
- CTM ドメインマネジャ
- スマートエージェント

(d)　運用管理サーバマシン

運用管理サーバマシンには，Application Server をインストールする必要があります。

起動するプロセスは次のとおりです。

- Management Server

3.6 トランザクションの種類を検討する

この節では，トランザクションの種類ごとのシステム構成について説明します。

トランザクションに参加するリソースが一つの場合は，ローカルトランザクションを使用します。トランザクションに参加するリソースが複数の場合は，グローバルトランザクションを使用します。また，サーバ間で連携する構成の場合に，それぞれの J2EE サーバが異なるリソースに接続するときには，トランザクションコンテキストのプロパゲーションも使用できます。

なお，接続するリソースとリソースアダプタの対応については，「3.3.2　リソースの種類とリソースアダプタ」を参照してください。

> **参考**
>
> この節では，アプリケーションサーバによって開始されるトランザクションの種類ごとの構成について説明しますが，EJB クライアントアプリケーションでもトランザクションを開始できます。なお，EJB クライアントアプリケーションでトランザクションを開始する場合は，EJB クライアントマシンに Application Server が必要です。

3.6.1 ローカルトランザクションを使用する場合の構成

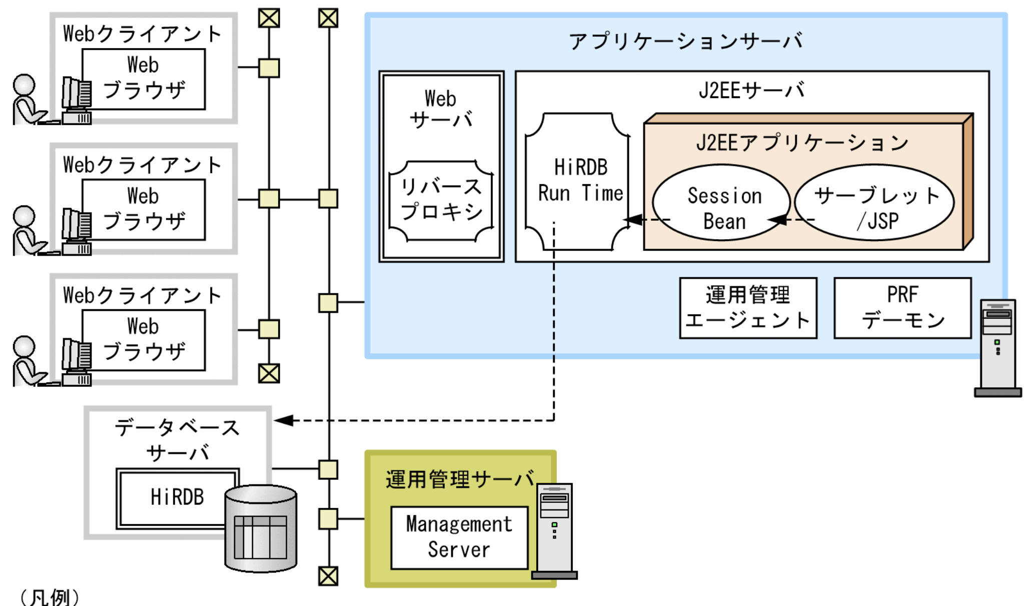
トランザクションに参加するリソースが一つの場合の構成について説明します。

(1) システム構成の特徴

サーブレット，JSP および Session Bean で構成される一つの J2EE アプリケーションから，リソースアダプタ経由で一つのリソースマネジャにアクセスする場合の構成です。アプリケーションからリソースマネジャへのアクセスで使用するトランザクションは，リソースマネジャ側で管理されます。トランザクションの種類は XA インタフェースを使用しない，ローカルトランザクションになります。

ローカルトランザクションを使用する場合の構成の例を，次の図に示します。

図 3-34　ローカルトランザクションを使用する場合の構成の例



これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

特徴

アプリケーションから一つのリソースマネジャにアクセスします。2 フェーズコミットは必要ありません。

アプリケーションからリソースアダプタ経由でリソースマネジャにアクセスする流れ

Web ブラウザから Web サーバ経由でアクセスされたサーブレットおよび JSP は，同じアプリケーション内の Session Bean をローカルで呼び出します。Session Bean は，リソースアダプタを経由してリソースマネジャ（図の場合は HiRDB）にアクセスします。

(2)　それぞれのマシンに必要なソフトウェアと起動するプロセス

ローカルトランザクションを使用する場合に，それぞれのマシンに必要なソフトウェアと起動するプロセスについて説明します。

(a)　アプリケーションサーバマシン

アプリケーションサーバマシンには，Application Server をインストールする必要があります。

また，リソースマネジャと接続するために，次のソフトウェアが必要です。

接続するリソースマネジャ	必要なソフトウェア
HiRDB	HiRDB Run Time または HiRDB Type4 JDBC Driver
Oracle	Oracle Client または Oracle JDBC Thin Driver
SQL Server	SQL Server の JDBC ドライバ

接続するリソースマネジャ	必要なソフトウェア
XDM/RD E2	HiRDB Run Time または HiRDB Type4 JDBC Driver
TP1/Message Queue	TP1/Message Queue - Access
OpenTP1 の SPP	TP1 Connector TP1/Client/J

起動するプロセスは次のとおりです。

- Web サーバ
- J2EE サーバ
- 運用管理エージェント
- PRF デーモン

(b) リソースマネジャが動作するマシン

リソースマネジャが動作するマシンには，次に示すソフトウェアのどれかをインストールしてください。

- HiRDB（HiRDB と接続する場合）
- Oracle（Oracle と接続する場合）
- SQL Server（SQL Server と接続する場合）
- XDM/RD E2（XDM/RD E2 と接続する場合）
- TP1/Message Queue（TP1/Message Queue と接続する場合）
- TP1/Server Base（OpenTP1 の SPP と接続する場合）

また，それぞれのリソースマネジャで必要なプロセスを起動してください。

(c) 運用管理サーバマシン

運用管理サーバマシンには，Application Server をインストールする必要があります。

起動するプロセスは次のとおりです。

- Management Server

(d) Web クライアントマシン

Web クライアントマシンには，Web ブラウザが必要です。

3.6.2 グローバルトランザクションを使用する場合の構成

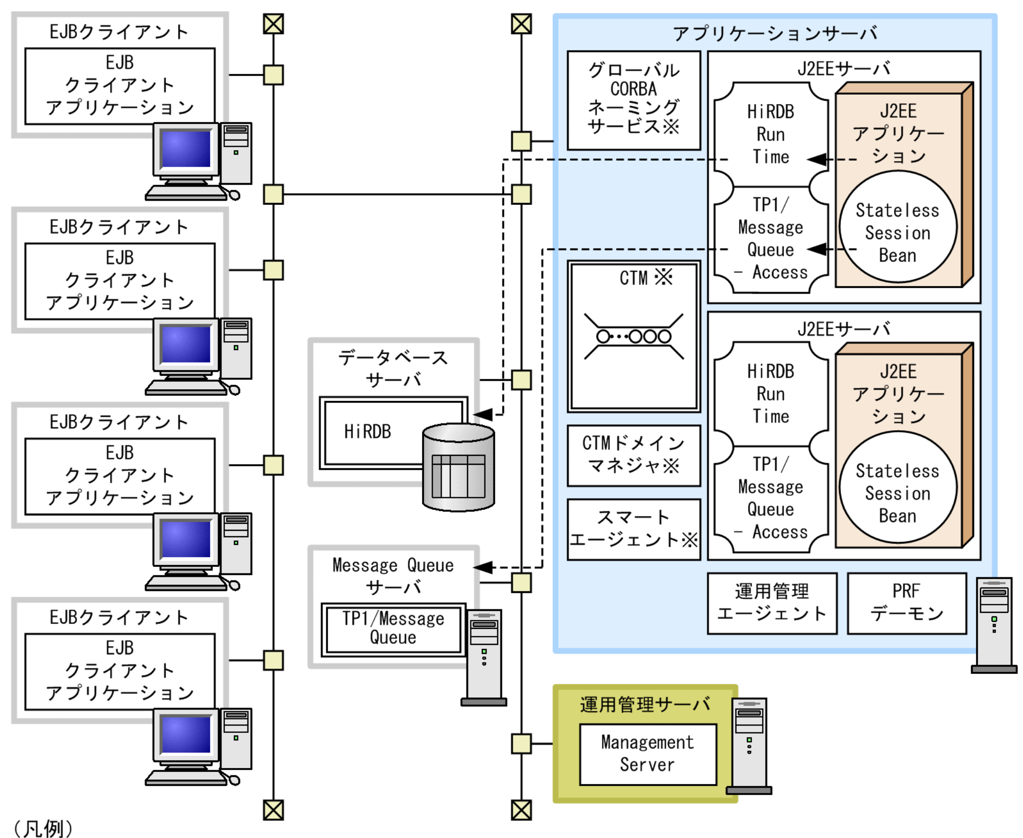
トランザクションに参加するリソースが複数の場合の構成について説明します。

(1)　システム構成の特徴

Session Bean を構成要素とする一つの J2EE アプリケーションから，リソースアダプタ経由で複数のリソースマネジャにアクセスする構成です。アプリケーションからリソースマネジャへのアクセスで使用するトランザクションは，J2EE サーバ側で管理します。トランザクションの種類は，XA インタフェースを使用するグローバルトランザクションになります。

グローバルトランザクションを使用する場合の構成の例を，次の図に示します。この例は，CTM を使用するシステムの場合の例です。

図 3-35　グローバルトランザクションを使用する場合の構成の例



（凡例）

---►：アプリケーションからリソースアダプタ経由でリソースマネジャにアクセスする流れ

注※　CTMを使用する場合に必要となります。

これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

特徴

- アプリケーションから複数のリソースマネジャにアクセスします。2 フェーズコミットが必要になります。
- 一つのリソースマネジャにアクセスするアプリケーションと，複数のリソースマネジャにアクセスするアプリケーションを混在させるシステムの場合も，この構成にする必要があります。

アプリケーションからリソースアダプタ経由でリソースマネジャにアクセスする流れ

EJB クライアントアプリケーションからアクセスされた Stateless Session Bean は，リソースアダプタを経由してリソースマネジャにアクセスします。

(2) それぞれのマシンに必要なソフトウェアと起動するプロセス

グローバルトランザクションを使用する場合に，それぞれのマシンに必要なソフトウェアと起動するプロセスについて説明します。

(a) アプリケーションサーバマシン

アプリケーションサーバマシンには，Application Server をインストールする必要があります。

また，リソースマネジャと接続するために，次のソフトウェアをインストールする必要があります。

接続するリソースマネジャ	必要なソフトウェア
HiRDB	HiRDB Run Time または HiRDB Type4 JDBC Driver
Oracle	Oracle Client または Oracle JDBC Thin Driver
TP1/Message Queue	TP1/Message Queue - Access
OpenTP1 の SPP	TP1 Connector TP1/Client/J

注 SQL Server および XDM/RD E2 はグローバルトランザクションでは使用できません。

起動するプロセスは次のとおりです。

- J2EE サーバ
- 運用管理エージェント
- PRF デーモン

また，CTM を使用する構成の場合は，Application Server をインストールして，上記のプロセスのほかに，グローバル CORBA ネーミングサービス，CTM のプロセス群，CTM ドメインマネジャおよびスマートエージェントも起動する必要があります。詳細は，「3.5.2 CTM 経由で Stateless Session Bean を呼び出すサーバ間連携」を参照してください。

(b) リソースマネジャが動作するマシン

リソースマネジャが動作するマシンには，次に示すソフトウェアのどれかをインストールしてください。なお，SQL Server および XDM/RD E2 はグローバルトランザクションでは使用できません。

- HiRDB（HiRDB と接続する場合）
- Oracle（Oracle と接続する場合）
- TP1/Message Queue（TP1/Message Queue と接続する場合）

- TP1/Server Base（OpenTP1 の SPP と接続する場合）

また，それぞれのリソースマネジャで必要なプロセスを起動してください。

(c) 運用管理サーバマシン

運用管理サーバマシンには，Application Server をインストールする必要があります。

起動するプロセスは次のとおりです。

- Management Server

(d) EJB クライアントマシン

EJB クライアントマシンには，Application Server または Client（Windows の場合）をインストールする必要があります。

起動するプロセスは，EJB クライアントアプリケーションのプロセスです。

3.6.3 トランザクションコンテキストのプロパゲーションを使用する場合の構成

サーバ間で連携する構成の場合に，それぞれの J2EE サーバが異なるリソースに接続する構成について説明します。この構成では，トランザクションコンテキストのプロパゲーションを使用します。なお，この構成の場合，CTM は使用できません。

(1) システム構成の特徴

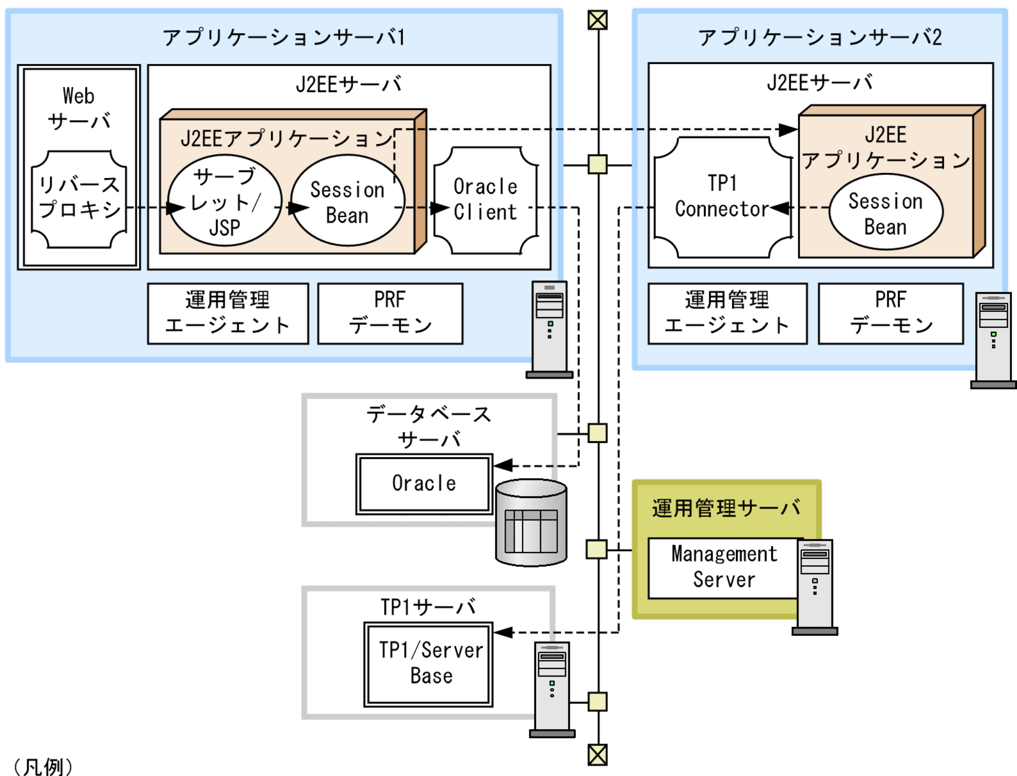
複数の J2EE アプリケーションでサーバ間連携をする構成です。ここでは，サーブレット，JSP，および Session Bean で構成される J2EE アプリケーションと，Session Bean だけで構成される J2EE アプリケーションでサーバ間連携をする例について説明します。サーブレット，JSP および Session Bean で構成されるアプリケーションは，クライアント側のアプリケーションサーバで動作し，Session Bean で構成される J2EE アプリケーションは，サーバ側のアプリケーションサーバで動作します。

それぞれの J2EE アプリケーションは，リソースアダプタ経由で異なるリソースマネジャにアクセスします。この場合，リソースマネジャへのアクセスで使用するトランザクションは，J2EE サーバ側で管理します。このときのトランザクションの種類は，XA インタフェースを使用するグローバルトランザクションになります。

なお，CTM 経由で Stateless Session Bean を呼び出す構成では，トランザクションコンテキストのプロパゲーションは使用できません。

トランザクションコンテキストのプロパゲーションを使用する場合の構成の例を次の図に示します。この例では，J2EE アプリケーションは，それぞれ，データベースと OpenTP1 の SPP にアクセスします。

図 3-36　トランザクションコンテキストのプロパゲーションを使用する場合の構成の例



これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

特徴

- サーバ間連携をするアプリケーションから複数のリソースマネジャにアクセスする場合に適用できます。
- サーバ間連携をする場合に，一つのリソースマネジャにアクセスするアプリケーションと，複数のリソースマネジャにアクセスするアプリケーションを混在させるシステムの場合は，この構成にする必要があります。

アプリケーションからリソースアダプタ経由でリソースマネジャにアクセスする流れ

Web ブラウザから Web サーバ経由でアクセスされたサーブレットおよび JSP は，同じアプリケーション内の Session Bean をローカルで呼び出します。Session Bean は，トランザクションを開始して，データベースにアクセスしてから，サーバ側のアプリケーションサーバ上の J2EE サーバで動作している Session Bean を呼び出します。呼び出されたサーバ側のアプリケーションサーバ上の Session Bean は，OpenTP1 の SPP にアクセスします。クライアント側のアプリケーションサーバ上の Session Bean は，サーバ側の Session Bean から制御が戻ると，トランザクションをコミットします。

(2)　それぞれのマシンに必要なソフトウェアと起動するプロセス

トランザクションコンテキストのプロパゲーションを使用するときに必要なソフトウェアと起動するプロセスは，「3.6.2　グローバルトランザクションを使用する場合の構成」と同様です。ただし，コンテキストのトランザクションプロパゲーションを使用する場合，CTM は使用できません。

3.7 ロードバランスクラスタによる負荷分散方式を検討する

この節では，ロードバランスクラスタによって負荷を分散するための構成について説明します。

負荷分散は次の方法で実現します。アクセスポイントになるコンポーネントの種類によって，実現できるものが異なります。

- 負荷分散機を利用する（サーブレット／JSP の場合）
- ネーミングサービスを利用する（Session Bean／Entity Bean の場合）
- CTM を利用する（Stateless Session Bean の場合）

負荷分散機を利用する構成については，Web サーバ連携の場合と Web サーバを経由しないで J2EE サーバの NIO HTTP サーバに直接アクセスする場合に分けて説明します。なお，アクセスポイントが Message-driven Bean の場合の負荷分散については，「3.8.4　Message-driven Bean のインスタンスプールを利用した負荷分散（TP1/Message Queue を使用する場合）」を参照してください。

3.7.1 Web サーバ連携時の負荷分散機を利用した負荷分散（サーブレット／JSP の場合）

負荷分散機（レイヤ 5 スイッチ）によって負荷を分散する構成について説明します。この方法は，サーブレットまたは JSP がアクセスポイントの場合に使用できます。

ここでは，Web サーバと連携する場合について説明します。

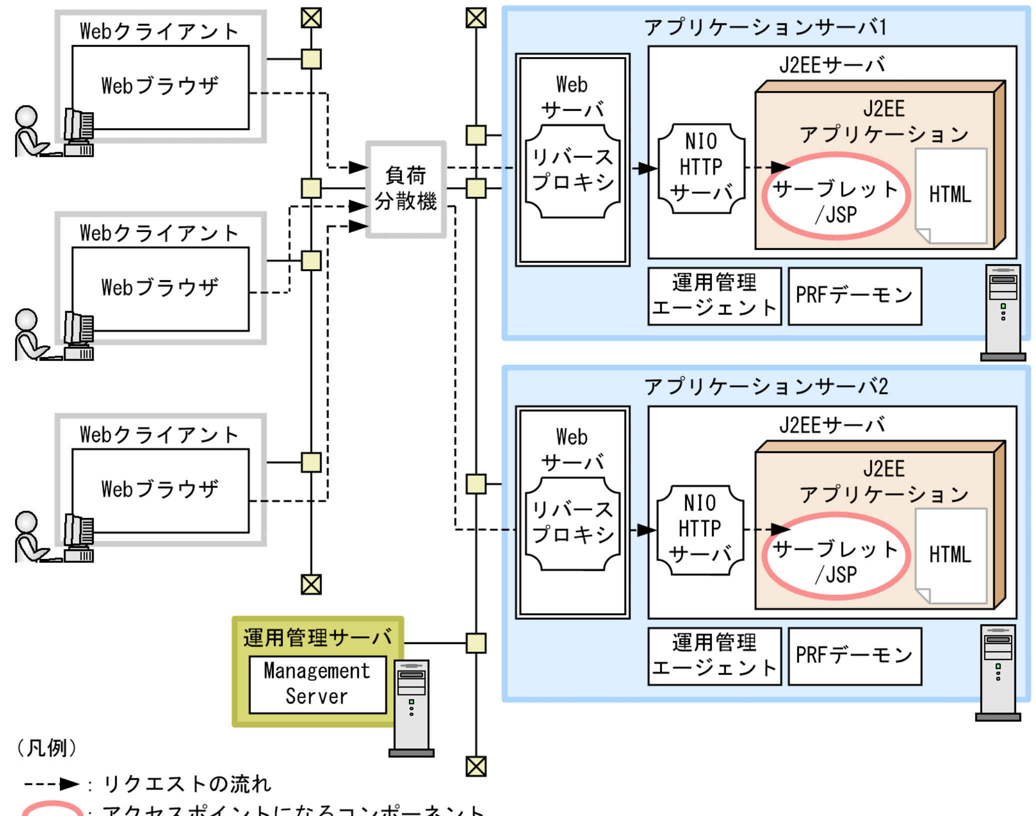
(1) システム構成の特徴

J2EE サーバ上で動作するアプリケーションのアクセスポイントが，サーブレットまたは JSP の場合に使用できる構成です。

負荷分散は，負荷分散機に対象となるアプリケーションサーバを登録することで実現できます。アプリケーションサーバを複数登録することで，クライアントからのアクセスによる負荷を分散できます。

負荷分散機を利用した負荷分散の構成の例を次の図に示します。

図 3-37　負荷分散機を利用した負荷分散の構成の例（Web サーバと連携する場合）



これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

特徴

- サーブレットと JSP のスケーラビリティと可用性を確保できます。
- 特定のアプリケーションサーバでトラブルが発生した場合，またはメンテナンスが必要な場合に，負荷分散機から該当するアプリケーションサーバへのアクセスを停止することで，システムの縮退運転ができます。

リクエストの流れ

アクセスポイントである J2EE サーバ上のサーブレットと JSP へのリクエストは，Web ブラウザから，負荷分散機および Web サーバ経由で送られます。その際，負荷分散機では，Web ブラウザからのアクセスを，それぞれのアプリケーションサーバ上で動作している Web サーバに対して振り分けます。なお，負荷分散機では，HTTP セッションのスティッキー（Sticky）やアフィニティ（Afinity）の関連づけも制御します。

参考

- HTTPS を使用する場合に，負荷分散機でのリクエストの振り分けにリクエストの中身（ヘッダなど）を使用するときは，負荷分散機のフロントに SSL アクセラレータを配置する必要があります。

- Smart Composer 機能を使用する場合は，負荷分散機を冗長化構成で配置することもできます。

(2)　それぞれのマシンに必要なソフトウェアと起動するプロセス

負荷分散機を使用する場合にそれぞれのマシンに必要なソフトウェアと起動するプロセスは，サーブレットと JSP をアクセスポイントにする構成と同じです。「3.4.1　サーブレットと JSP をアクセスポイントに使用する構成（Web サーバ連携の場合）」を参照してください。

3.7.2　NIO HTTP サーバ使用時の負荷分散機を利用した負荷分散（Web サーバを経由しない場合）

負荷分散機（**レイヤ 5 スイッチ**）によって負荷を分散する構成について説明します。この方法は，サーブレットまたは JSP がアクセスポイントの場合に使用できます。

ここでは，Web サーバを経由しないで J2EE サーバの NIO HTTP サーバに直接アクセスする場合について説明します。

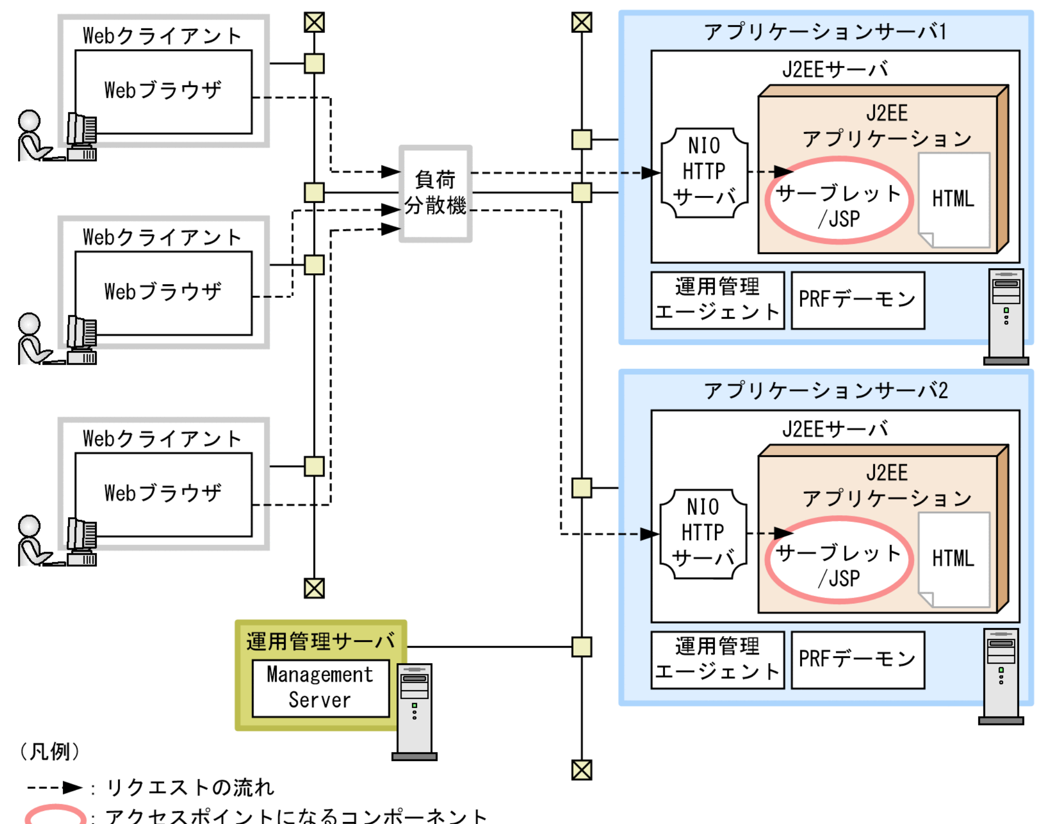
(1)　システム構成の特徴

J2EE サーバ上で動作するアプリケーションのアクセスポイントが，サーブレットまたは JSP の場合に使用できる構成です。

負荷分散は，負荷分散機に対象になるアプリケーションサーバを登録することで実現できます。アプリケーションサーバを複数登録することで，クライアントからのアクセスによる負荷を分散できます。

負荷分散機を利用した負荷分散の構成の例を次の図に示します。

図 3-38　負荷分散機を利用した負荷分散の構成の例（NIO HTTP サーバに直接アクセスする場合）



これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

特徴

- サーブレットと JSP のスケーラビリティと可用性を確保できます。
- 特定のアプリケーションサーバでトラブルが発生した場合，またはメンテナンスが必要な場合に，負荷分散機から該当するアプリケーションサーバへのアクセスを停止することで，システムの縮退運転ができます。
- Web ブラウザから Web サーバを経由しないで直接 J2EE サーバにアクセスできるため，性能上のメリットがあります。また，Web サーバを起動する必要がないため，運用が簡易になります。ただし，使用できる機能や構成について，留意する必要があります。また，インターネットに接続する場合は，必ずリバースプロキシを組み込んだ Web サーバをフロントに配置してください。詳細は，「3.4.2　サーブレットと JSP をアクセスポイントに使用する構成（NIO HTTP サーバに直接アクセスする場合）」を参照してください。

リクエストの流れ

アクセスポイントである J2EE サーバ上のサーブレットと JSP へのリクエストは，Web ブラウザから，負荷分散機経由で送られます。その際，負荷分散機では，Web ブラウザからのアクセスを，それぞれのアプリケーションサーバ上で動作している J2EE サーバに対して振り分けます。なお，負荷分散機では，HTTP セッションのスティッキー（Sticky）やアフィニティ（Afinity）の関連づけも制御します。

> **参考**
>
> HTTPS を使用する場合は，負荷分散機のフロントに SSL アクセラレータを配置する必要があります。

(2) それぞれのマシンに必要なソフトウェアと起動するプロセス

負荷分散機を使用する場合にそれぞれのマシンに必要なソフトウェアと起動するプロセスは，サーブレットと JSP をアクセスポイントにする構成と同じです。「3.4.2 サーブレットと JSP をアクセスポイントに使用する構成（NIO HTTP サーバに直接アクセスする場合）」を参照してください。

3.7.3 CORBA ネーミングサービスを利用した負荷分散（Session Bean と Entity Bean の場合）

アクセスポイントになるコンポーネントが Session Bean または Entity Bean の場合に，J2EE サーバ内（インプロセス）の CORBA ネーミングサービスのラウンドロビン検索機能を使用して負荷を分散する構成について説明します。

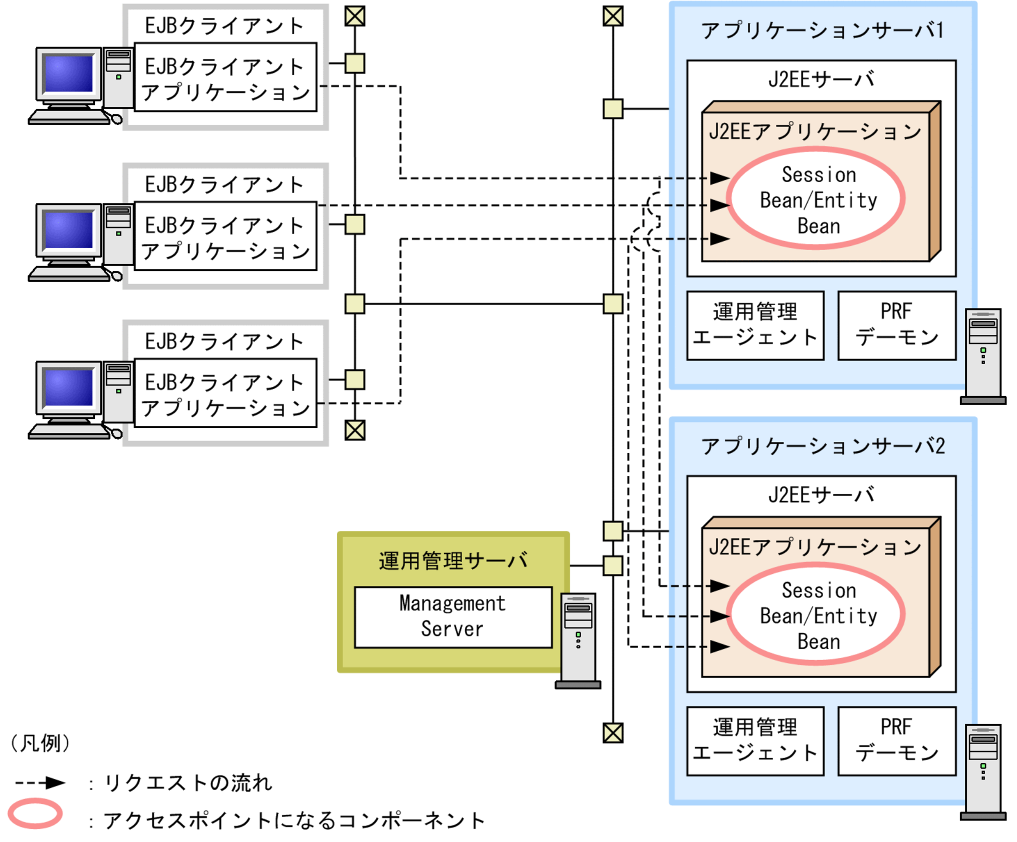
(1) システム構成の特徴

J2EE サーバ上で動作するアプリケーションのアクセスポイントが，Session Bean または Entity Bean の場合に使用できる構成です。クライアントは，EJB クライアントアプリケーションです。EJB クライアントアプリケーションは，システムプロパティに登録した論理ネーミングサービスから，オブジェクトリファレンスをラウンドロビン方式でルックアップすることで，リクエストの振り分け先を分散します。ただし，それぞれの J2EE サーバでは同じ名前（同じ別名）で Session Bean および Entity Bean を開始しておく必要があります。

なお，EJB クライアントアプリケーションは，ラウンドロビン方式で J2EE サーバ内のネーミングサービスをルックアップしたあと再度ネーミングサービスをルックアップするまでの間は，同じ J2EE サーバにアクセスします。

Session Bean と Entity Bean を対象にした負荷分散の構成の例を次の図に示します。

図 3-39　Session Bean と Entity Bean を対象にした負荷分散の構成の例



これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

特徴

- Session Bean と Entity Bean のスケーラビリティと可用性を確保できます。
- アプリケーションサーバを複数用意して，EJB クライアントからラウンドロビン方式で CORBA ネーミングサービスを選択してアクセスすることで，負荷を分散できます。
- 特定のアプリケーションサーバでトラブルが発生した場合，またはメンテナンスが必要な場合，EJB クライアントアプリケーションでは，J2EE サーバの終了を検出したあと，該当するアプリケーションサーバにアクセスしません。このため，システムの縮退運転ができます。

リクエストの流れ

アクセスポイントである J2EE サーバ上の Session Bean と Entity Bean へのリクエストは，EJB クライアントアプリケーションから送られます。その際，EJB クライアントアプリケーションでは，J2EE サーバ内のネーミングサービスをラウンドロビン方式で選択して，オブジェクトリファレンスをルックアップします。

(2)　それぞれのマシンに必要なソフトウェアと起動するプロセス

CORBA ネーミングサービスを使用して負荷を分散する場合にそれぞれのマシンに必要なソフトウェアと起動するプロセスは，Session Bean と Entity Bean をアクセスポイントにする構成と同じです。「3.4.3 Session Bean と Entity Bean をアクセスポイントに使用する構成」を参照してください。

3.7.4　CTM を利用した負荷分散（Stateless Session Bean の場合）

アクセスポイントになるコンポーネントが Stateless Session Bean の場合に，CTM によって負荷を分散する構成です。

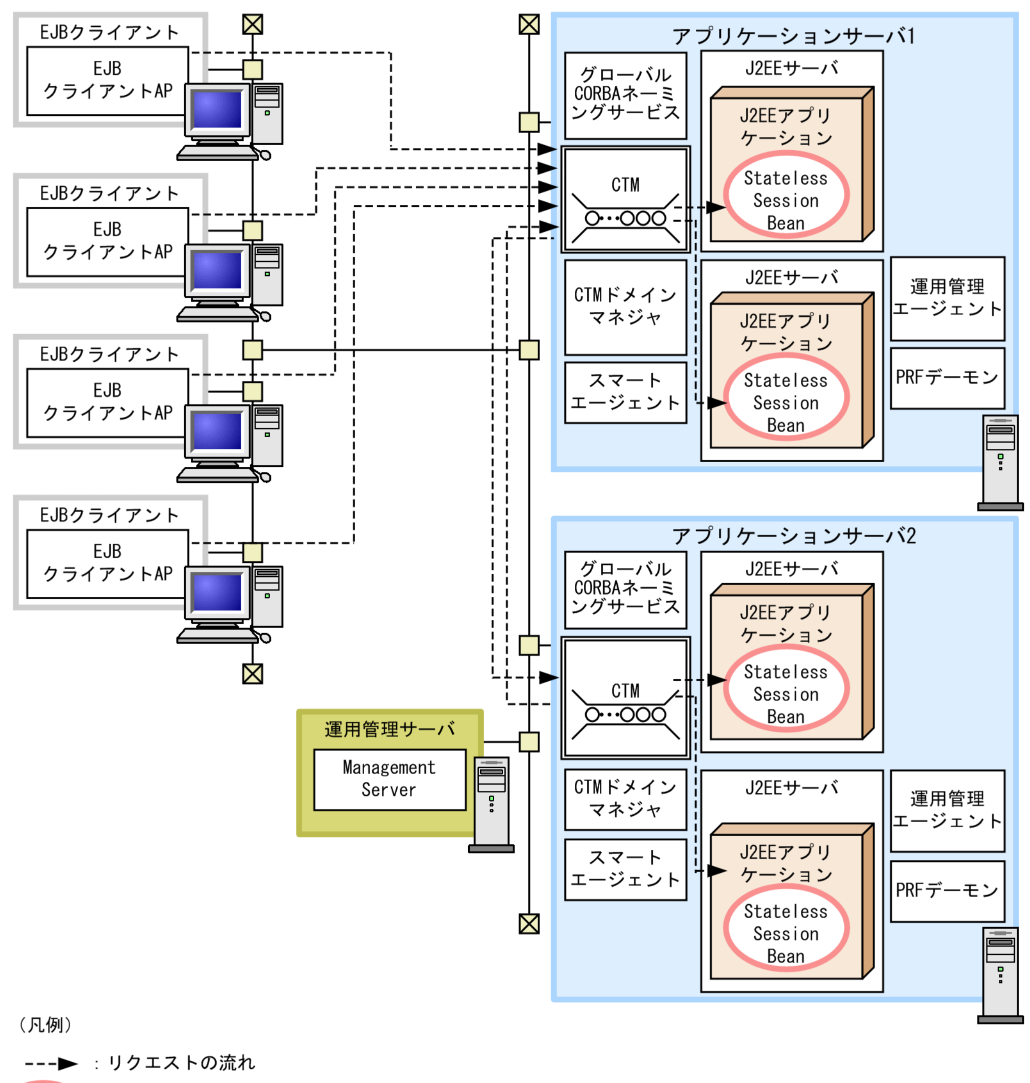
(1)　システム構成の特徴

J2EE サーバ上で動作するアプリケーションのアクセスポイントが Stateless Session Bean の場合に，CTM を使用して実現する構成です。ここでは，クライアントが EJB クライアントアプリケーションの場合について説明します。

EJB クライアントアプリケーションは，システムプロパティに登録したグローバル CORBA ネーミングサービスからオブジェクトリファレンスをラウンドロビン方式でルックアップすることで，リクエストの振り分け先を分散します。ただし，それぞれの J2EE サーバでは同じ名前（同じ別名）で Stateless Session Bean を開始しておく必要があります。

Stateless Session Bean を対象にした CTM による負荷分散の構成の例を次の図に示します。

図 3-40　Stateless Session Bean を対象にした CTM による負荷分散の構成の例



これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

特徴

- Stateless Session Bean の負荷の均衡を実現して，高い可用性を確保できます。
- EJB クライアントからのラウンドロビン方式によるグローバル CORBA ネーミングサービスのルックアップと，CTM によるリクエストの振り分けによって，J2EE サーバ間の負荷均衡が実現できます。
- Stateless Session Bean のスケールアウトが容易に実現できるので，アプリケーションサーバの稼働率を向上できます。
- 特定の J2EE サーバにトラブルが発生した場合，CTM によってほかの J2EE サーバにリクエストをスケジューリングすることで，システムの縮退運転ができます。また，EJB クライアントアプリケーションからは，トラブルを検知した J2EE サーバにはアクセスされません。

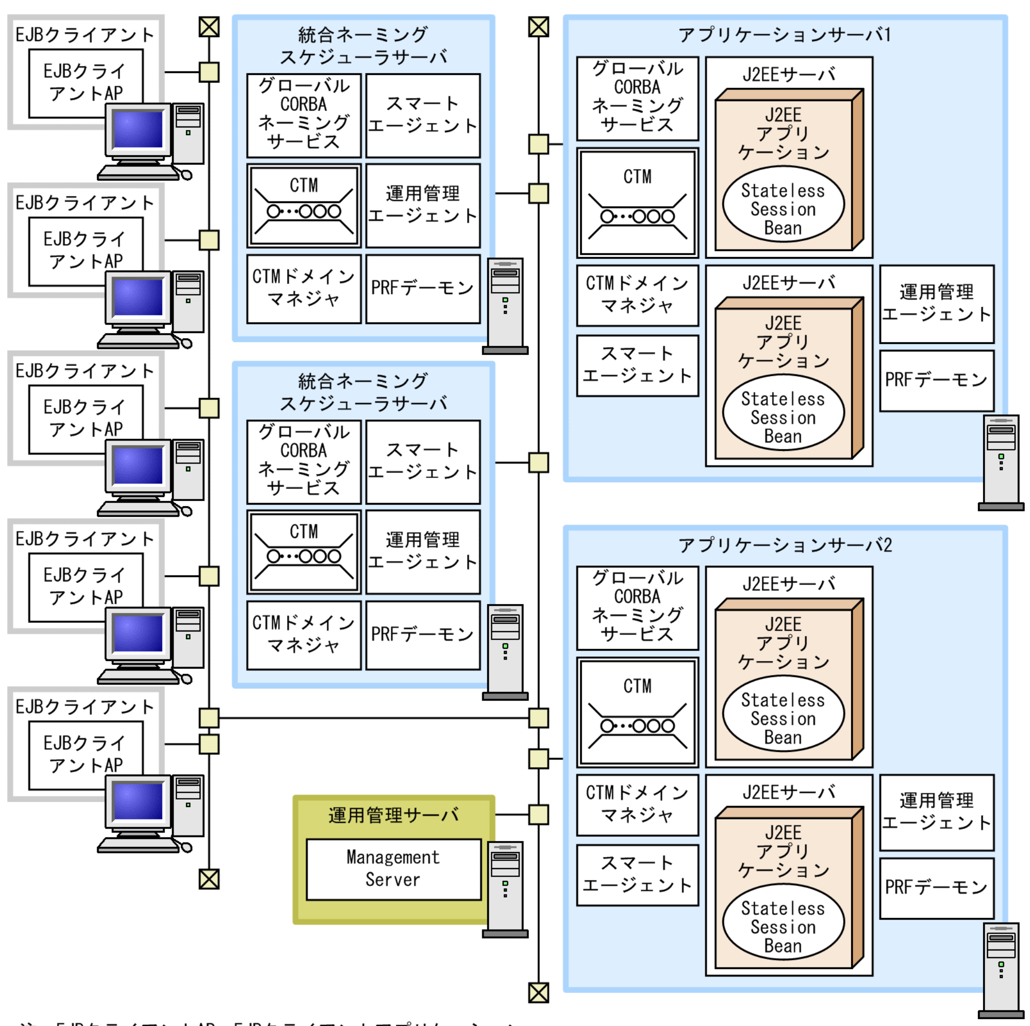
リクエストの流れ

アクセスポイントである J2EE サーバ上の Stateless Session Bean へのリクエストは，EJB クライアントから CTM 経由で送られます。その際，EJB クライアントは，グローバル CORBA ネーミングサービスから Stateless Session Bean の EJBHome オブジェクトリファレンスの名前をルックアップします。そのあと，リクエストは CTM によって，適切なアプリケーションサーバ上の J2EE サーバに振り分けられます。

CTM を使用して負荷を分散する場合，グローバル CORBA ネーミングサービスを独立したマシンに配置することもできます。このとき，グローバル CORBA ネーミングサービスを配置したマシンを，**統合ネーミングスケジューラサーバ**といいます。

統合ネーミングスケジューラサーバを配置したシステムの構成を次の図に示します。

図 3-41　Stateless Session Bean を対象にした CTM による負荷分散の構成の例（統合ネーミングスケジューラサーバを配置した場合）



凡例については，「3.2　システム構成の説明について」を参照してください。

特徴

- Stateless Session Bean の負荷の均衡を実現して，高い可用性を確保できます。
- 統合ネーミングスケジューラサーバのレプリカを作成して複数配置することで，ネーミングサービスの可用性を確保できます。
- EJB クライアントからのラウンドロビン方式によるグローバル CORBA ネーミングサービスのルックアップと，CTM によるリクエストの振り分けによって，J2EE サーバ間の負荷均衡が実現できます。
- Stateless Session Bean のスケールアウトが容易に実現できるので，アプリケーションサーバの稼働率を向上できます。スケールアウト時に，EJB クライアントで定義したグローバル CORBA ネーミングサービスのリストを変更する必要はありません。
- 特定の J2EE サーバにトラブルが発生した場合，CTM によってほかの J2EE サーバにリクエストをスケジューリングすることで，システムの縮退運転ができます。EJB クライアントアプリケーションからは，トラブルを検知した J2EE サーバにはアクセスされません。
- EJB クライアントは，統合ネーミングスケジューラサーバ上のグローバル CORBA ネーミングサービスから Stateless Session Bean の EJBHome オブジェクトリファレンスの名前をルックアップします。そのあと，アプリケーションサーバの CTM デーモンに処理が振り分けられます。

(2) それぞれのマシンに必要なソフトウェアと起動するプロセス

CTM を使用して負荷を分散する場合に，それぞれのマシンに必要なソフトウェアと起動するプロセスについて説明します。

(a) アプリケーションサーバマシン

アプリケーションサーバマシンには，Application Server をインストールする必要があります。

起動するプロセスは次のとおりです。

- J2EE サーバ
- 運用管理エージェント
- PRF デーモン
- グローバル CORBA ネーミングサービス
- CTM のプロセス群（CTM デーモンおよび CTM レギュレータ）
- CTM ドメインマネジャ
- スマートエージェント

(b) 統合ネーミングスケジューラサーバマシン

統合ネーミングスケジューラサーバを配置するシステム構成の場合，統合ネーミングスケジューラサーバマシンには，Application Server をインストールする必要があります。

起動するプロセスは次のとおりです。J2EE サーバを起動する必要はありません。

- グローバル CORBA ネーミングサービス
- CTM のプロセス群（CTM デーモン）
- CTM ドメインマネジャ
- スマートエージェント
- 運用管理エージェント
- PRF デーモン

(c) 運用管理サーバマシン

運用管理サーバマシンには，Application Server をインストールする必要があります。

起動するプロセスは次のとおりです。

- Management Server

(d) EJB クライアントマシン

EJB クライアントマシンには，Application Server または Client（Windows の場合）をインストールする必要があります。

起動するプロセスは EJB クライアントアプリケーションのプロセスです。

3.8　サーバ間で非同期通信をする場合の構成を検討する

この節では，サーバ間で Message-driven Bean を使用して非同期通信をする場合のシステム構成について説明します。アクセスポイントは Message-driven Bean になります。

Message-driven Bean を使って非同期通信をするシステム構成としては，CJMS プロバイダを使用するシステム構成，TP1/Message Queue を使用するシステム構成，および Reliable Messaging を使用するシステム構成があります。JMS プロバイダと Reliable Messaging は，アプリケーションサーバの構成ソフトウェアです。なお，Reliable Messaging を使用したシステムでは，Message-driven Bean のインスタンスプールを使用した負荷分散はできません。

3.8.1　Message-driven Bean をアクセスポイントに使用する場合の構成（CJMS プロバイダを使用する場合）

ここでは，CJMS プロバイダを使用して J2EE サーバ上の Message-driven Bean を呼び出す構成について説明します。

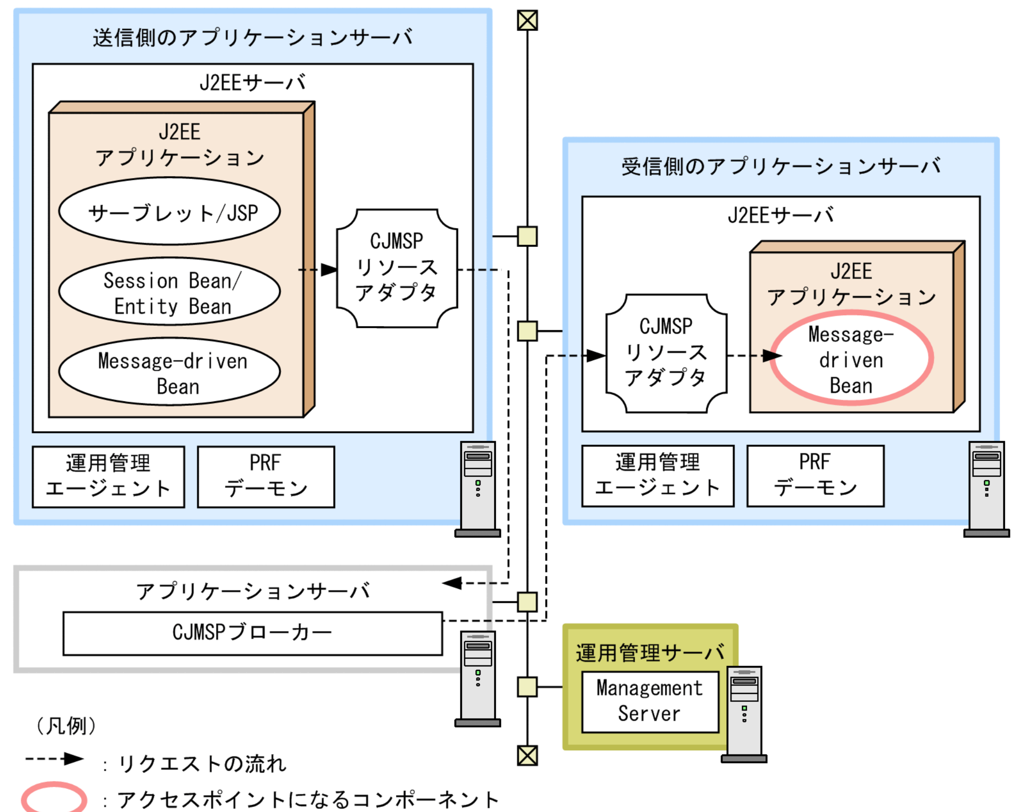
メッセージの送信側のアプリケーションサーバから，CJMSP ブローカーを経由して受信側のアプリケーションサーバを呼び出す構成です。送信側のアプリケーションサーバでは，サーブレット，JSP，Entity Bean，Session Bean または Message-driven Bean で構成されるアプリケーションが動作します。受信側のアクセスポイントになるコンポーネントは，Message-driven Bean です。

(1)　システム構成の特徴

最も基本的なメッセージ駆動型のシステムの一つです。

CJMS プロバイダを使用する場合のメッセージ駆動型のシステム構成の例を次の図に示します。

図 3-42　メッセージ駆動型のシステム構成の例（CJMS プロバイダを使用する場合）



これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

特徴

送信側のアプリケーションサーバでは，メッセージを送信するために JMS インタフェースを使用する J2EE クライアントアプリケーションと CJMSP リソースアダプタを使用します。

リクエストの流れ

アクセスポイントである Message-driven Bean は，受信側のアプリケーションサーバの J2EE サーバ上で動作します。リソースアダプタである CJMSP リソースアダプタのライブラリは，送信側のアプリケーションサーバの J2EE サーバ，および受信側のアプリケーションサーバの J2EE サーバ上で動作します。

送信側のアプリケーションサーバの J2EE アプリケーションからのリクエスト（メッセージ）は，CJMSP ブローカー経由で送られ，受信側のアプリケーションサーバ上の Message-driven Bean を呼び出します。

なお，送信側のアプリケーションサーバ，受信側のアプリケーションサーバおよび CJMSP ブローカーは，同じマシンに配置することもできます。

(2)　それぞれのマシンに必要なソフトウェアと起動するプロセス

それぞれのマシンに必要なソフトウェアと起動するプロセスについて説明します。

(a)　アプリケーションサーバマシン（送信側のアプリケーションサーバマシン）

アプリケーションサーバマシン（サーバ側のアプリケーションサーバマシン）には，Application Server をインストールする必要があります。

起動するプロセスは次のとおりです。

- J2EE サーバ
- 運用管理エージェント
- PRF デーモン

(b)　アプリケーションサーバマシン（CJMSP ブローカーを配置するマシン）

CJMSP ブローカーを配置するマシンには，Application Server をインストールする必要があります。

起動する必要があるプロセスは，CJMSP ブローカーです。なお，CJMSP ブローカーは，Management Server による運用管理の対象になりません。

(c)　アプリケーションサーバマシン（受信側のアプリケーションサーバマシン）

クライアントマシン（クライアント側のアプリケーションサーバマシン）には，Application Server をインストールする必要があります。

起動するプロセスは次のとおりです。

- J2EE サーバ
- 運用管理エージェント
- PRF デーモン

(d)　運用管理サーバマシン

運用管理サーバマシンには，Application Server をインストールする必要があります。

起動するプロセスは次のとおりです。

- Management Server

3.8.2　Message-driven Bean をアクセスポイントに使用する場合の構成（TP1/Message Queue を使用する場合）

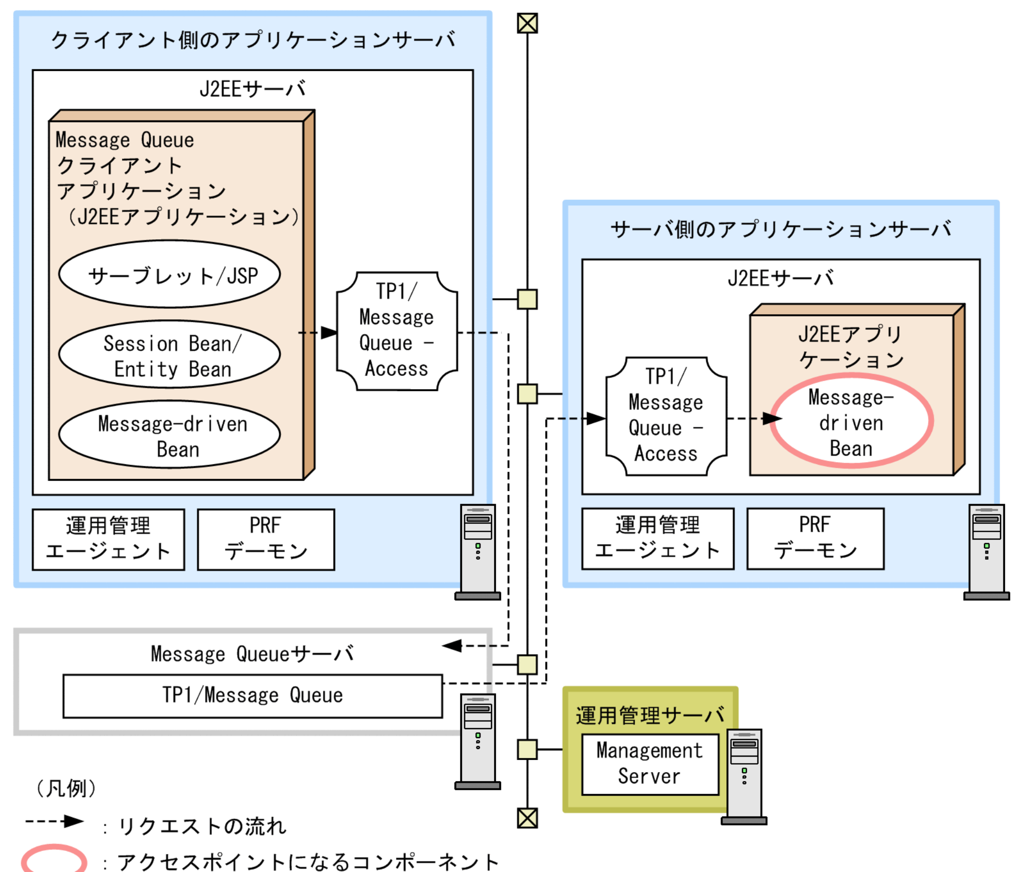
ここでは，クライアントとして，TP1/Message Queue が動作している Message Queue サーバを使用する場合の構成について説明します。

(1) システム構成の特徴

最も基本的なメッセージ駆動型のシステムの一つです。

TP1/Message Queue を使用する場合のメッセージ駆動型のシステム構成の例を次の図に示します。

図 3-43 メッセージ駆動型のシステム構成の例（TP1/Message Queue を使用する場合）



これ以外の凡例については，「3.2 システム構成の説明について」を参照してください。

特徴

クライアントに，TP1/Message Queue にメッセージを送信する Message Queue クライアントアプリケーションと TP1/Message Queue を使用します。なお，Message Queue クライアントアプリケーションとは，JMS インタフェースを使用する J2EE アプリケーションです。

リクエストの流れ

アクセスポイントである Message-driven Bean は，サーバ側のアプリケーションサーバの J2EE サーバ上で動作します。リソースアダプタである TP1/Message Queue - Access のライブラリは，サーバ側のアプリケーションサーバの J2EE サーバ，およびクライアント側のアプリケーションサーバの J2EE サーバ上で動作します。

クライアント側のアプリケーションサーバの J2EE アプリケーションからのリクエスト（メッセージ）は，TP1/Message Queue 経由で送られ，サーバ側のアプリケーションサーバ上の Message-driven Bean を呼び出します。

(2)　それぞれのマシンに必要なソフトウェアと起動するプロセス

それぞれのマシンに必要なソフトウェアと起動するプロセスについて説明します。なお，リソースに接続するためのプロセスについては，「3.6　トランザクションの種類を検討する」を参照してください。

(a)　アプリケーションサーバマシン（サーバ側のアプリケーションサーバマシン）

アプリケーションサーバマシン（サーバ側のアプリケーションサーバマシン）には，Application Server および TP1/Message Queue - Access をインストールする必要があります。

起動するプロセスは次のとおりです。

- J2EE サーバ
- 運用管理エージェント
- PRF デーモン

(b)　Message Queue サーバマシン

Message Queue サーバマシンには，TP1/Message Queue をインストールする必要があります。

起動する必要があるプロセスは，TP1/Message Queue のプロセスです。

(c)　アプリケーションサーバマシン（クライアント側のアプリケーションサーバマシン）

クライアントマシン（クライアント側のアプリケーションサーバマシン）には，Application Server および TP1/Message Queue - Access をインストールする必要があります。

起動するプロセスは次のとおりです。

- J2EE サーバ
- 運用管理エージェント
- PRF デーモン

(d)　運用管理サーバマシン

運用管理サーバマシンには，Application Server をインストールする必要があります。

起動するプロセスは次のとおりです。

- Management Server

3.8.3　Message-driven Bean をアクセスポイントに使用する場合の構成（Reliable Messaging を使用する場合）

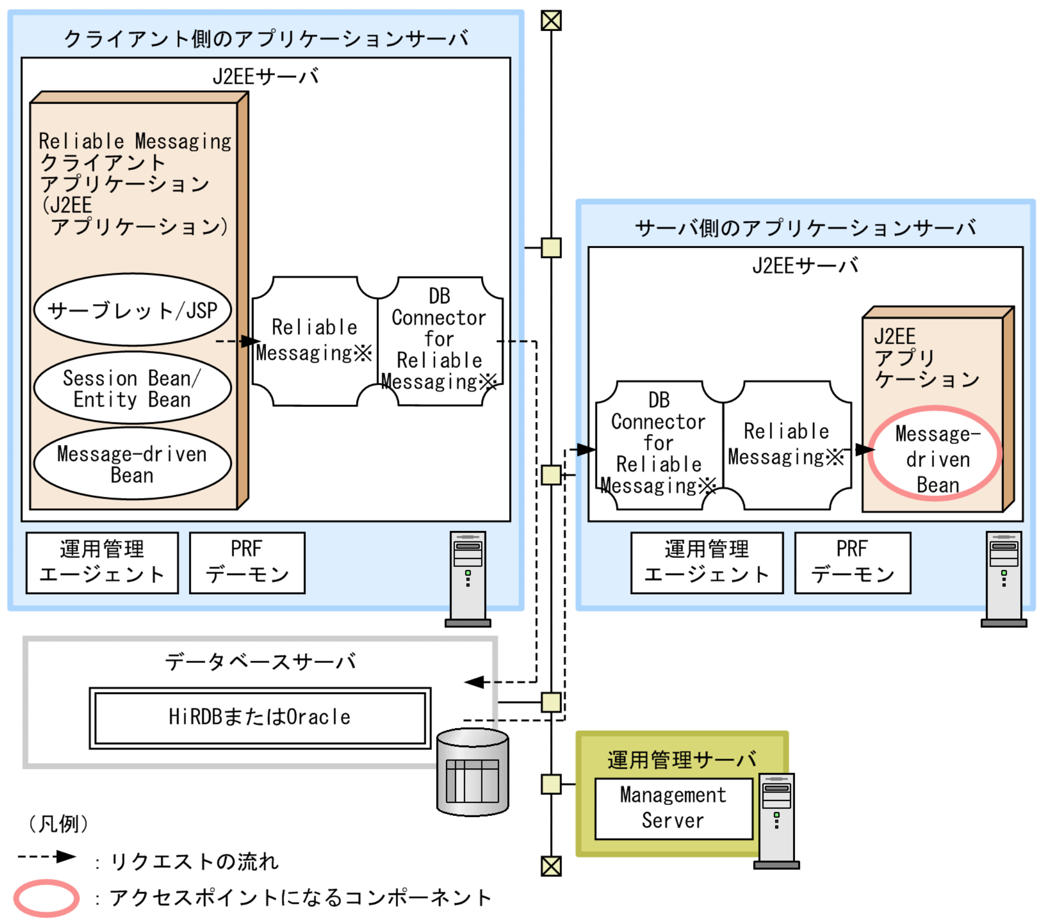
ここでは，クライアントとして，Reliable Messaging と連携しているデータベース（HiRDB または Oracle）を使用する場合の構成について説明します。

(1)　システム構成の特徴

最も基本的なメッセージ駆動型のシステムの一つです。

Reliable Messaging を使用する場合のメッセージ駆動型のシステム構成の例を次の図に示します。

図 3-44　メッセージ駆動型のシステム構成の例（Reliable Messaging を使用する場合）



これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

特徴

クライアントに，Reliable Messaging にメッセージを送信する Reliable Messaging のクライアントアプリケーションとデータベースサーバを使用します。なお，Reliable Messaging クライアントアプリケーションとは，JMS インタフェースを使用する J2EE アプリケーションです。

リクエストの流れ

アクセスポイントである Message-driven Bean は，サーバ側のアプリケーションサーバの J2EE サーバ上で動作します。リソースアダプタである Reliable Messaging のライブラリは，サーバ側のアプリケーションサーバの J2EE サーバ上，およびクライアント側のアプリケーションサーバの J2EE サーバ上で動作します。

クライアント側のアプリケーションサーバの J2EE アプリケーションからのリクエスト（メッセージ）はデータベース上に実現されたキュー経由で送られ，サーバ側のアプリケーションサーバ上の Message-driven Bean を呼び出します。

(2)　それぞれのマシンに必要なソフトウェアと起動するプロセス

それぞれのマシンに必要なソフトウェアと起動するプロセスについて説明します。なお，リソースに接続するためのプロセスについては，「3.6　トランザクションの種類を検討する」を参照してください。

(a)　サーバ側のアプリケーションサーバマシン

サーバ側のアプリケーションサーバマシンには，Application Server をインストールする必要があります。

起動するプロセスは次のとおりです。

- J2EE サーバ
- 運用管理エージェント
- PRF デーモン

(b)　データベースサーバマシン

データベースサーバマシンには，HiRDB または Oracle をインストールする必要があります。

起動する必要があるプロセスは，HiRDB または Oracle のプロセスです。

(c)　クライアントマシン（クライアント側のアプリケーションサーバマシン）

クライアントマシン（クライアント側のアプリケーションサーバマシン）には，Application Server をインストールする必要があります。

起動するプロセスは次のとおりです。

- J2EE サーバ
- 運用管理エージェント
- PRF デーモン

(d)　運用管理サーバマシン

運用管理サーバマシンには，Application Server をインストールする必要があります。

起動するプロセスは次のとおりです。

- Management Server

3.8.4 Message-driven Bean のインスタンスプールを利用した負荷分散（TP1/Message Queue を使用する場合）

アクセスポイントになるコンポーネントが Message-driven Bean の場合に，J2EE サーバごとの Message-driven Bean のインスタンスのプール数に応じて負荷を分散する構成について説明します。

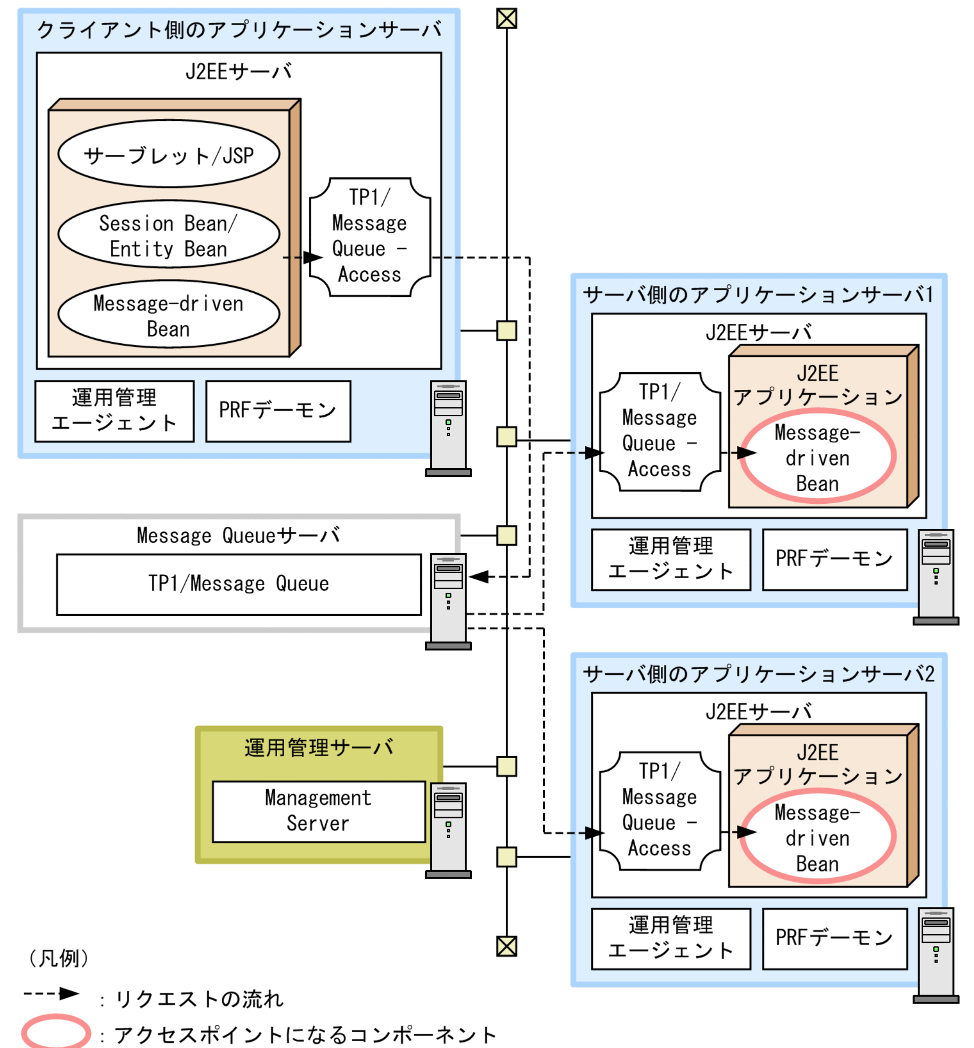
なお，この構成によって負荷を分散できるのは，TP1/Message Queue を使用して Message-driven Bean にアクセスする場合だけです。Reliable Messaging を使用して Message-driven Bean にアクセスする場合は，Message-driven Bean のインスタンスプールを利用した負荷分散はできません。

(1) システム構成の特徴

J2EE サーバ上で動作するアプリケーションのアクセスポイントが Message-driven Bean である，メッセージ駆動型のシステムで使用できる構成です。Message Queue クライアントからのメッセージが Message Queue サーバ上の TP1/Message Queue のキューに到着すると，J2EE サーバ上でプールされている Message-driven Bean のインスタンス数に応じて，Message-driven Bean が呼び出されます。

Message-driven Bean を対象にした負荷分散の構成の例を次の図に示します。

図 3-45　Message-driven Bean を対象にした負荷分散の構成の例



これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

特徴

- Message-driven Bean の負荷分散を実現して，高い可用性を確保できます。
- アプリケーションサーバを複数用意することで，アプリケーションサーバの負荷を分散できます。
- サーバ側のアプリケーションサーバで障害が発生した場合やアプリケーションサーバのメンテナンスが必要な場合，TP1/Message Queue から該当するアプリケーションサーバ上の J2EE サーバにはアクセスしません。このため，縮退運転ができます。

リクエストの流れ

アクセスポイントであるサーバ側のアプリケーションサーバの J2EE サーバ上の Message-driven Bean へのリクエスト（メッセージ）は，クライアント側のアプリケーションサーバの J2EE サーバ上で動作する J2EE アプリケーションから，TP1/Message Queue 経由で送られます。TP1/Message Queue にメッセージが到着すると，サーバ側のアプリケーションサーバの J2EE サーバ上の Message-driven Bean が呼び出されます。その際，Message-driven Bean のインスタンスプールのプール数に応じて，リクエストが振り分けられます。

(2)　それぞれのマシンに必要なソフトウェアと起動するプロセス

Message-driven Bean のインスタンスプールを使用して負荷を分散する場合に必要なソフトウェアと起動するプロセスは，Message-driven Bean をアクセスポイントにする構成と同じです。「3.8.2　Message-driven Bean をアクセスポイントに使用する場合の構成（TP1/Message Queue を使用する場合）」を参照してください。

3.9　運用管理プロセスの配置を検討する

この節では，運用管理に使用するプロセスである，Management Server の配置について説明します。Management Server は，複数のホスト上に構築されているアプリケーションサーバのシステム全体を一括管理，および一括運用するプロセスです。Management Server を使用することで，各ホスト上のサーバの環境設定をまとめて実施したり，サーバを一括起動したりできます。また，システム全体の状態を把握できます。

ここでは，次の構成を説明します。

- 運用管理サーバに Management Server を配置する構成（運用管理サーバモデル）
- マシン単位に Management Server を配置する構成（ホスト単位管理モデル）
- コマンドを使用して運用する構成

なお，運用管理サーバモデルで Management Server を配置したマシンを，**運用管理サーバ**といいます。

参考

- Management Server の機能を使用して，J2EE サーバ，またはバッチサーバの稼働情報を監視・収集する場合，運用監視エージェントというエージェントプログラムを使用します。監視対象にするホスト上で，監視対象にする J2EE サーバ，またはバッチサーバに一つずつ含まれます。また，クラスタ構成の場合は，クラスタに含まれる J2EE サーバに一つずつ含まれます。
- 業務用のサーバを配置した LAN と管理用のサーバを配置した LAN に分けている場合，運用管理サーバを管理用のサーバを配置した LAN に置くこともできます。LAN を分け，一つのマシンを複数のネットワークセグメントに接続する場合，環境設定に注意が必要です。詳細については，マニュアル「アプリケーションサーバ 運用管理ポータル操作ガイド」の「付録 D　一つのマシンを複数のネットワークセグメントに接続する場合の環境設定での注意」を参照してください。

3.9.1　運用管理サーバに Management Server を配置する構成

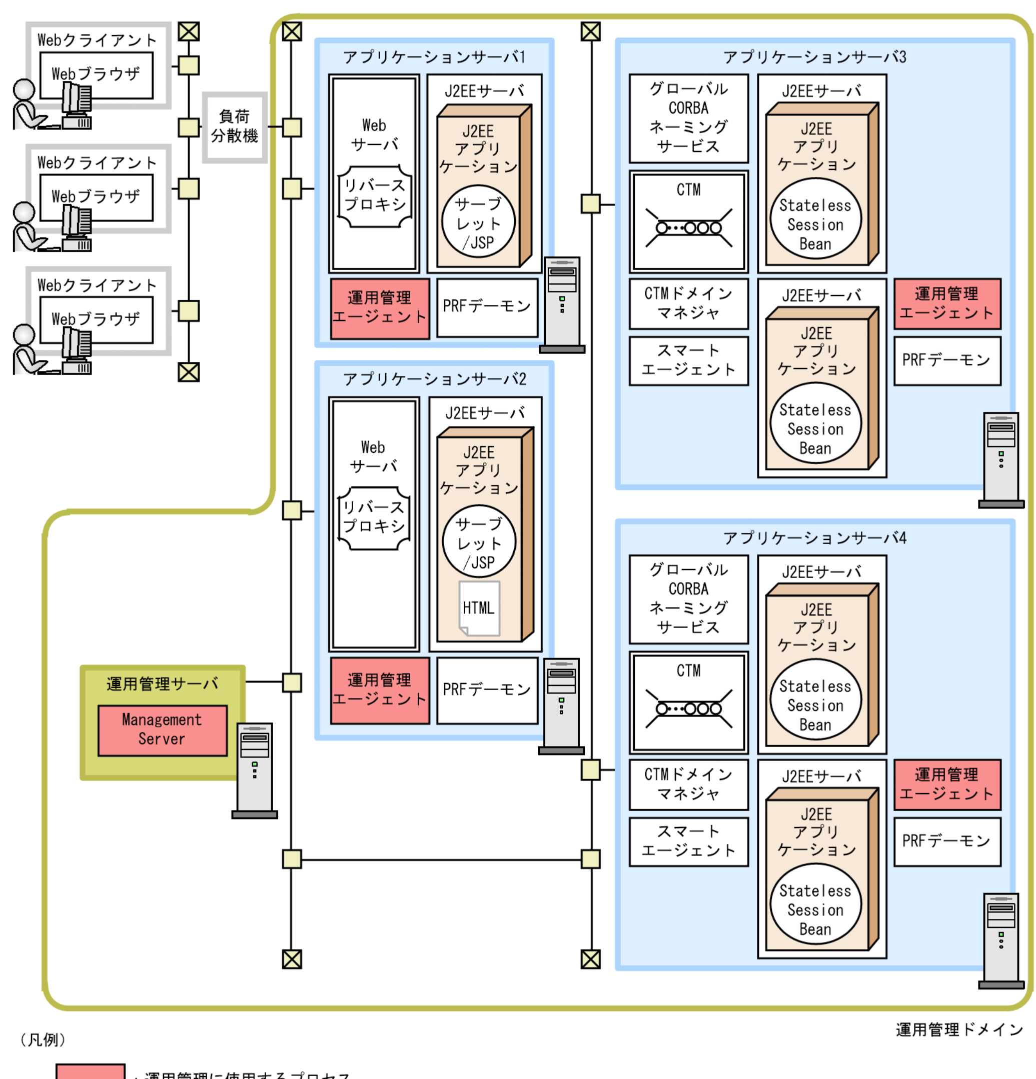
Management Server を利用して運用する場合に，ドメイン内に一つ運用管理サーバを配置する構成について説明します。

(1)　システム構成の特徴

複数のアプリケーションサーバで構成されるシステムで，ドメイン内に運用管理サーバマシンを配置する構成です。Management Server であらかじめ定義したドメイン（**運用管理ドメイン**）内の運用管理および監視が実現できます。運用管理サーバマシンの Management Server は，それぞれのアプリケーションサーバに配置した運用管理エージェントと協調して，運用管理操作を実行します。

運用管理サーバに Management Server を配置する構成の例を次の図に示します。なお，この例では，接続するリソースマネジャについては省略しています。

図 3-46　運用管理サーバに Management Server を配置する構成の例



これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

特徴

- システムの一括運用に適しています。
- 独立した運用管理サーバ用のマシンを用意することをお勧めします。ただし，システム内の一つのアプリケーションサーバ内に Management Server を配置することもできます。

(2)　それぞれのマシンで起動するプロセス

運用管理サーバに Management Server を配置する構成の場合に，それぞれのマシンに必要なソフトウェアと起動するプロセスについて説明します。

(a)　運用管理サーバマシン

運用管理サーバマシンには，Application Server をインストールする必要があります。

起動するプロセスは次のとおりです。

- Management Server

> **ポイント**
>
> 運用管理サーバマシンを用意できないときは，ドメイン内のどれかのアプリケーションサーバが運用管理サーバを兼ねるシステム構成にもできます。
>
> この場合は，運用管理サーバにするアプリケーションサーバマシンに，Management Server のプロセスを配置してください。

(b)　アプリケーションサーバマシン

運用管理サーバで運用管理するために，それぞれのアプリケーションサーバマシンで起動するプロセスは次のとおりです。

- 運用管理エージェント

これ以外にアプリケーションサーバマシンに必要なソフトウェアと起動するプロセスは，使用する機能に応じたシステム構成ごとに異なります。使用する機能に応じて必要なソフトウェアとプロセスを配置してください。

3.9.2　マシン単位に Management Server を配置する構成

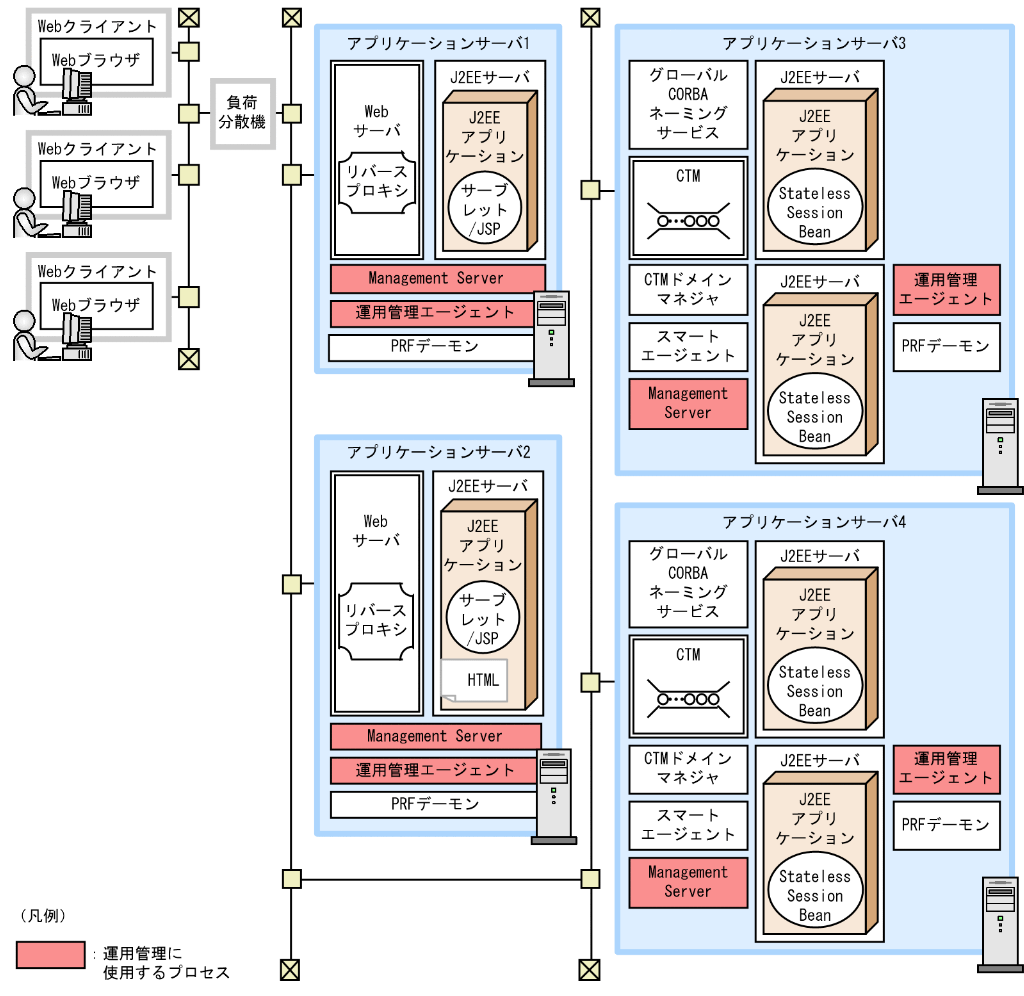
Management Server を利用して運用する場合に，それぞれのアプリケーションサーバマシンに Management Server を配置する構成について説明します。

(1)　システム構成の特徴

複数のアプリケーションサーバで構成されるシステムで，Management Server をそれぞれのアプリケーションサーバマシンに配置する構成です。アプリケーションサーバ単位の運用管理および監視が実行できます。Management Server は，運用管理エージェントと協調して，運用管理操作を実行します。

マシン単位に Management Server を配置する構成の例を次の図に示します。なお，この例では，接続するリソースマネジャについては省略しています。

図 3-47　マシン単位に Management Server を配置する構成の例



これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

特徴

- アプリケーションサーバ単位の運用に適しています。

(2)　それぞれのマシンに必要なソフトウェアと起動するプロセス

マシン単位に Management Server を配置する構成の場合に，それぞれのマシンに必要なソフトウェアと起動するプロセスについて説明します。

(a)　アプリケーションサーバマシン

マシン単位の運用管理をするためにそれぞれのアプリケーションサーバマシンで起動するプロセスは次のとおりです。

- Management Server
- 運用管理エージェント

これ以外にアプリケーションサーバマシンに必要なソフトウェアと起動するプロセスは，使用する機能に応じたシステム構成ごとに異なります。使用する機能に応じて必要なソフトウェアとプロセスを配置してください。

3.9.3　コマンドで運用する場合の構成

Management Server を利用しないで運用する場合の構成について説明します。

Management Server を利用しない場合は，定義ファイルの編集や，サーバ管理コマンドの実行などによって，アプリケーションサーバを運用します。このとき，それぞれのアプリケーションサーバに，運用管理エージェントおよび Management Server は不要です。アプリケーションサーバ以外の運用支援ソフトウェアなどを使用してシステムを管理する場合に，この構成を検討してください。

3.10　セッション情報の引き継ぎを検討する

この節では，セッション情報の引き継ぎを実現するためのシステム構成について説明します。この構成は，J2EE サーバが負荷分散機によって冗長化されていることが前提です。

セッション情報を引き継ぐ構成を次に示します。

- データベースを使用する構成（データベースセッションフェイルオーバ機能を使用する場合）

3.10.1　データベースを使用する構成（データベースセッションフェイルオーバ機能）

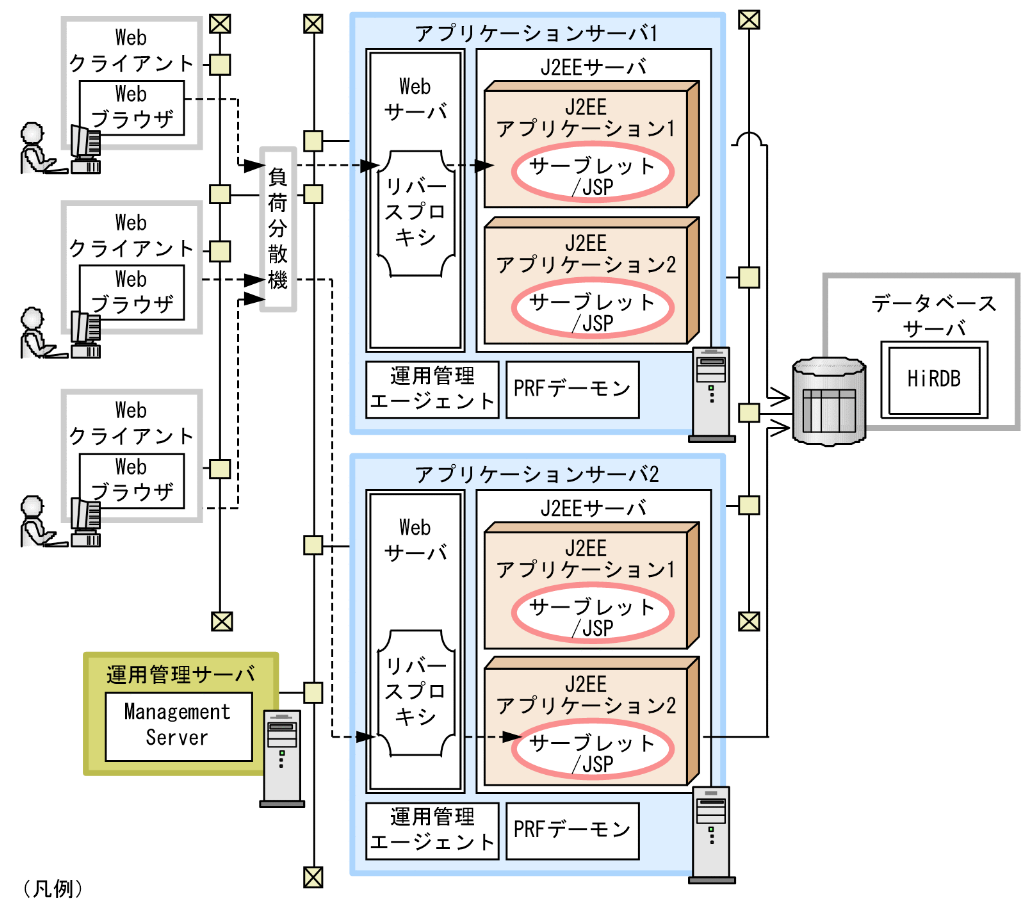
データベースを使用してセッション情報を引き継ぐ構成について説明します。

(1)　システム構成の特徴

セッション情報を格納するためのデータベースを用意します。業務情報を格納するために使用しているデータベースがある場合は，同じデータベースを使用できます。

データベースを使用してセッション情報を引き継ぐ場合の構成の例を次の図に示します。

図 3-48　データベースを使用してセッション情報を引き継ぐ構成の例



これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

特徴

- 特定の J2EE サーバでトラブルが発生した場合に，セッション情報をほかの J2EE サーバに引き継げます。
- データベースサーバにトラブルが発生した場合も，データベースに格納したセッション情報は残ります。データベースサーバ再起動後に利用できます。

リクエストの流れ

データベースでは，J2EE サーバ上のグローバルセッション情報が冗長化されて管理されています。

J2EE サーバがリクエストを受け付けてセッションが確立された時点で，データベースに格納されたグローバルセッション情報にロックが掛けられます。J2EE アプリケーションの処理が完了すると，J2EE サーバ上のグローバルセッション情報の内容に合わせてデータベースのグローバルセッション情報が更新されます。そのあとで，データベースのロックが解除されます。

J2EE サーバにトラブルが発生した場合は，データベースに格納したグローバルセッション情報が別な J2EE サーバから取得され，セッション情報が引き継がれます。

(2) それぞれのマシンで起動するプロセス

それぞれのマシンに必要なソフトウェアとプロセスについて説明します。

(a) アプリケーションサーバマシン

データベースセッションフェイルオーバ機能を使用する場合，データベースに接続するためのソフトウェアをインストールする必要があります。必要なソフトウェアを次の表に示します。

使用するデータベース	必要なソフトウェア
HiRDB	HiRDB Type4 JDBC Driver
Oracle	Oracle JDBC Thin Driver

これ以外に必要なソフトウェアと起動するプロセスは，使用する機能に応じたシステム構成ごとに異なります。使用する機能に応じて必要なソフトウェアとプロセスを配置してください。

(b) データベースサーバマシン

データベースが動作するマシンには，次に示すソフトウェアのどちらかをインストールしてください。

- HiRDB（HiRDB と接続する場合）
- Oracle（Oracle と接続する場合）

また，それぞれのデータベースで必要なプロセスを起動してください。

(c) 運用管理サーバマシン

運用管理サーバマシンには，Application Server をインストールする必要があります。

起動するプロセスは次のとおりです。

- Management Server

(d) Web クライアントマシン

Web クライアントマシンには，Web ブラウザが必要です。

3.11 クラスタソフトウェアを使用した障害時の系切り替えを検討する

この節では，クラスタソフトウェアを使用した障害時の系切り替えを実現するためのシステム構成について説明します。

アプリケーションサーバのシステムでは，OS ごとに，次のクラスタソフトウェアを使用できます。

- Windows の場合
  Windows Server Failover Cluster
- AIX または Linux の場合
  HA モニタ

アプリケーションサーバで構築するシステムでは，クラスタソフトウェアを使用して，次に示す構成を実現できます。なお，これらの構成は，Management Server を利用して運用することが前提になります。

- アプリケーションサーバの実行系と待機系を 1:1 にする構成
  実行系のアプリケーションサーバマシン 1 台に対して待機系のアプリケーションサーバマシンを 1 台用意しておく構成です。実行系のアプリケーションサーバマシンにトラブルが発生してマシンが終了した場合，または運用管理エージェントが終了した場合に，クラスタソフトウェアによって待機系のアプリケーションサーバを起動して処理を切り替えます。トランザクションサービスを使用しない場合と使用する場合で，共有ディスク装置の要否が異なります。なお，Windows Server Failover Cluster を使用する場合は，トランザクションサービスを使用しない場合も，共有ディスク装置が必要です。
- 運用管理サーバの実行系と待機系を 1:1 にする構成
  実行系の運用管理サーバマシン 1 台に対して待機系の運用管理サーバマシンを 1 台用意しておく構成です。実行系の運用管理サーバマシンにトラブルが発生してマシンが終了した場合，または Management Server のプロセスが終了した場合に，クラスタソフトウェアによって待機系の運用管理サーバを起動して処理を切り替えます。
- アプリケーションサーバの実行系と待機系を相互スタンバイにする構成
  アプリケーションサーバの実行系と待機系を 1：1 にする構成の一つです。それぞれのアプリケーションサーバに同じ種類の J2EE サーバを配置して，異なる J2EE サーバを起動しておくことで，それぞれのアプリケーションサーバが実行系として動作しながらお互いの待機系として機能します。どちらか一方の系に障害が発生すると，もう一方の系に切り替えられます。これによって，少ない台数のアプリケーションサーバマシンで，むだの少ない運用が可能になります。

  この構成では，それぞれのアプリケーションサーバマシンで Management Server を起動して，それぞれに異なる運用管理ドメインを管理します。
- N 台の実行系に対して 1 台の待機系を用意する構成（リカバリ専用サーバを使用する構成）
  この構成は，J2EE サーバを冗長化した構成でグローバルトランザクションを使用する場合に，特定の J2EE サーバにトラブルが発生したときにトランザクションを決着するためのシステム構成です。

  N 台の J2EE サーバを冗長化するためには，負荷分散機などを使用します。
- ホスト単位管理モデルの実行系と待機系を N：1 にする構成

アプリケーションサーバマシン（ホスト）の実行系複数（1～N 台）と待機系 1 台を配置し，それぞれに Management Server および運用管理エージェントを配置するシステムです。実行系のアプリケーションサーバマシンに障害が発生したときに待機系のアプリケーションサーバマシンに系を切り替えて，業務を続行できます。

3.11.1 アプリケーションサーバの実行系と待機系を 1:1 にする構成（トランザクションサービスを使用しない場合）

アプリケーションサーバの実行系と待機系を 1:1 にする構成の場合に，トランザクションサービスを使用しないときの構成について説明します。

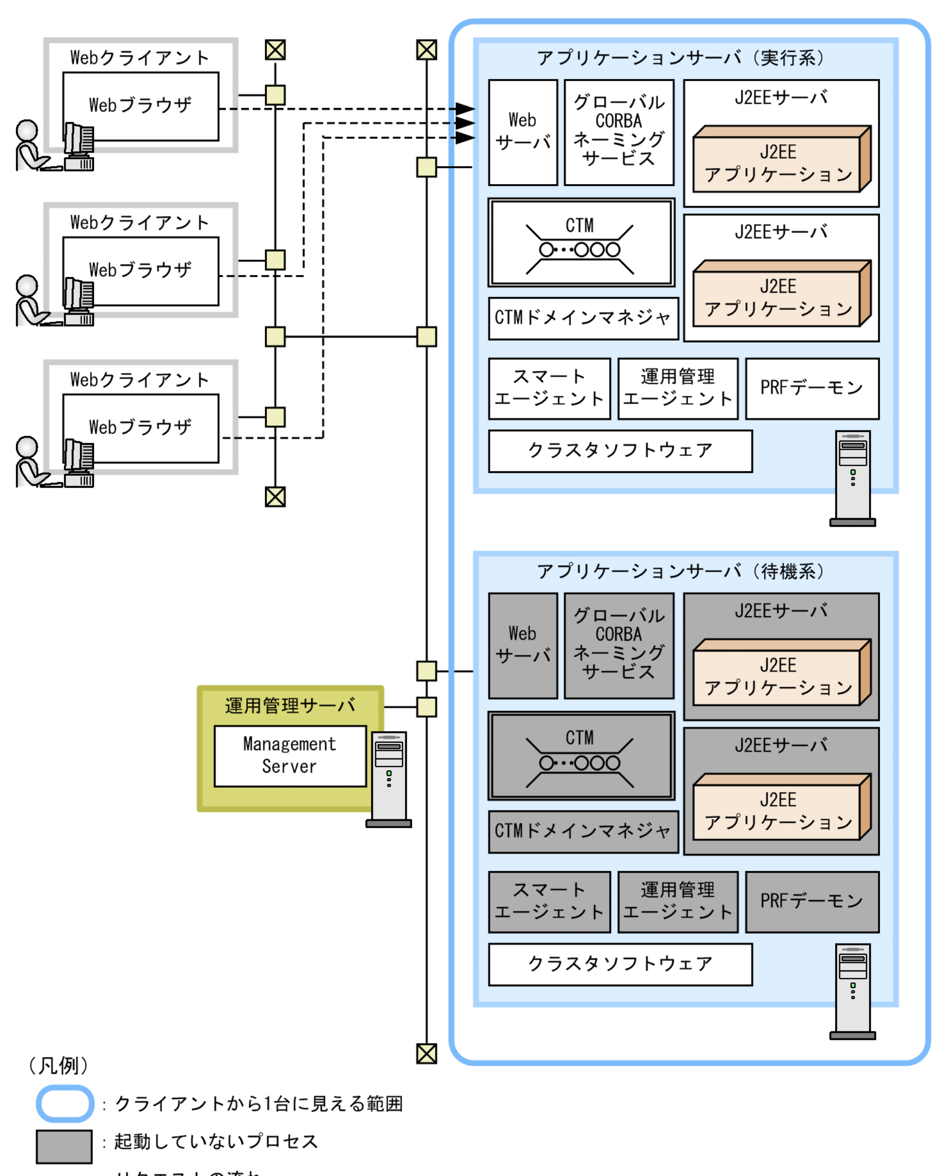
(1) システム構成の特徴

ローカルトランザクションを使用する場合など，トランザクションサービスを使用しない場合のシステム構成です。

なお，ここでは，アプリケーションサーバを系切り替えする場合の構成を示します。運用管理サーバを系切り替えする場合の構成については，「3.11.3 運用管理サーバの実行系と待機系を 1:1 にする構成」を参照してください。

トランザクションサービスを使用しない場合の構成の例を次の図に示します。

図 3-49　クラスタソフトウェアを使用して実行系と待機系を 1:1 にする場合のシステム構成の例（トランザクションサービスを使用しない場合）



これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

特徴

- 実行系と待機系は 1：1 で構成します。
- HA モニタを使用する場合は，共有ディスク装置は不要です。Windows Server Failover Cluster を使用する場合は，共有ディスク装置が必要です。図は，HA モニタの場合の構成です。
- データベースがクラスタ構成になっている場合でも，アプリケーションサーバでは仮想アドレス（論理アドレス）だけを認識していれば，接続できます。
- Management Server をアプリケーションサーバと同じマシンに配置する場合については，「3.11.6 ホスト単位管理モデルの実行系と待機系を N：1 にする構成」を参照してください。

リクエストの流れ

クライアントからは，実行系，待機系のアプリケーションサーバは，合わせて 1 台として認識されます。リクエストは，すべて実行系のアプリケーションサーバに送信されます。

系切り替えの流れ

実行系のアプリケーションサーバマシンに障害が発生すると，系切り替えが実行されます。系切り替えは，OS，または運用管理エージェントがダウンした場合に実行されます。なお，このほか，明示的に系切り替えを実行するコマンドを実行した場合や，これ以外にクラスタソフトウェアが系切り替えの対象にしている事象が発生した場合も，系切り替えが実行されます。

系切り替えが実行されると，待機系のアプリケーションサーバの運用管理エージェント，およびそれまで停止していたプロセスが起動されます。

(2) それぞれのマシンで起動するプロセス

それぞれのマシンに必要なソフトウェアとプロセスについて説明します。

(a) アプリケーションサーバマシン（実行系）

クラスタソフトウェアを使用する場合に必ず起動するプロセスは次のとおりです。

- 運用管理エージェント

これ以外にアプリケーションサーバに必要なソフトウェアと起動するプロセスは，使用する機能に応じたシステム構成ごとに異なります。使用する機能に応じて必要なソフトウェアとプロセスを配置してください。

(b) アプリケーションサーバマシン（待機系）

待機系のアプリケーションサーバマシンにインストールするソフトウェアの構成は，実行系と一致させてください。ただし，系切り替えが発生するまで，プロセスは起動しません。

(c) 運用管理サーバマシン

運用管理サーバマシンには，Application Server をインストールする必要があります。

起動するプロセスは次のとおりです。

- Management Server

(d) クライアントマシン

クライアントマシンには，次のソフトウェアをインストールしてください。

Web クライアント構成の場合

Web ブラウザ

EJB クライアント構成の場合

Client（Windows の場合），Application Server

3.11.2　アプリケーションサーバの実行系と待機系を 1:1 にする構成（トランザクションサービスを使用する場合）

アプリケーションサーバの実行系と待機系を 1:1 にする構成の場合に，トランザクションサービスを使用するときの構成について説明します。

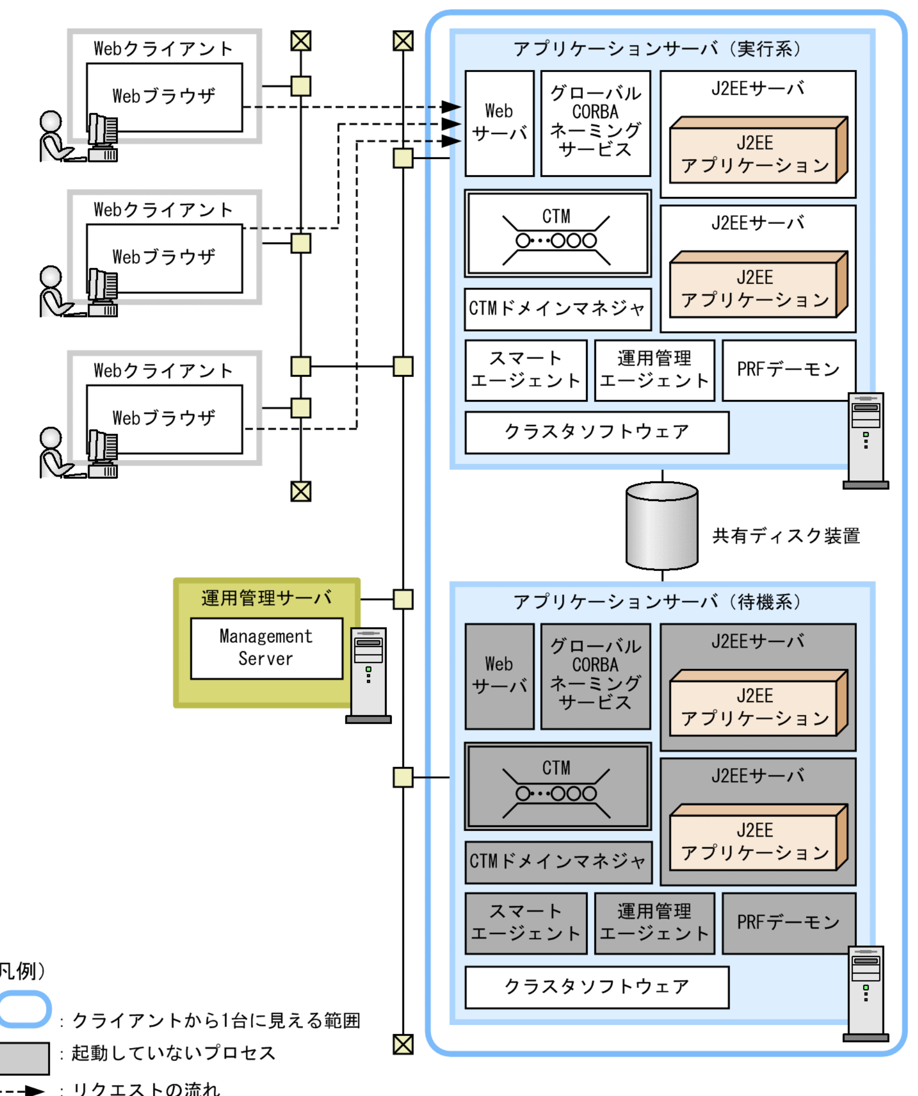
(1)　システム構成の特徴

グローバルトランザクションを使用する場合など，トランザクションサービスを使用する場合のシステム構成です。この場合，共有ディスク装置が必要です。

なお，ここでは，アプリケーションサーバを系切り替えする場合の構成を示します。運用管理サーバを系切り替えする場合の構成については，「3.11.3　運用管理サーバの実行系と待機系を 1:1 にする構成」を参照してください。

トランザクションサービスを使用する場合の構成の例を次の図に示します。

図 3-50　クラスタソフトウェアを使用してアプリケーションサーバの実行系と待機系を 1:1 にする場合のシステム構成の例（トランザクションサービスを使用する場合）



これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

特徴

- 実行系と待機系は 1：1 で構成します。
- 共有ディスク装置が必要です。
- データベースがクラスタ構成になっている場合でも，アプリケーションサーバでは仮想アドレス（論理アドレス）だけを認識していれば，接続できます。
- Management Server をアプリケーションサーバと同じマシンに配置する場合については，「3.11.3　運用管理サーバの実行系と待機系を 1:1 にする構成」を参照してください。

リクエストの流れ

クライアントからは，実行系，待機系のアプリケーションサーバは，合わせて 1 台として認識されます。リクエストは，すべて実行系のアプリケーションサーバに送信されます。

系切り替えの流れ

実行系のアプリケーションサーバマシンに障害が発生すると，系切り替えが実行されます。系切り替えは，OS，または運用管理エージェントがダウンした場合に実行されます。なお，このほか，明示的に系切り替えを実行するコマンドを実行した場合や，これ以外にクラスタソフトウェアが系切り替えの対象にしている事象が発生した場合も，系切り替えが実行されます。

系切り替えが実行されると，待機系のアプリケーションサーバの運用管理エージェント，およびそれまで停止していたプロセスが起動されます。なお，トランザクション情報については，共有ディスクに格納されていた情報が引き継がれます。

(2)　それぞれのマシンで起動するプロセス

それぞれのマシンに必要なソフトウェアとプロセスは，トランザクションサービスを利用しない場合と同じです。

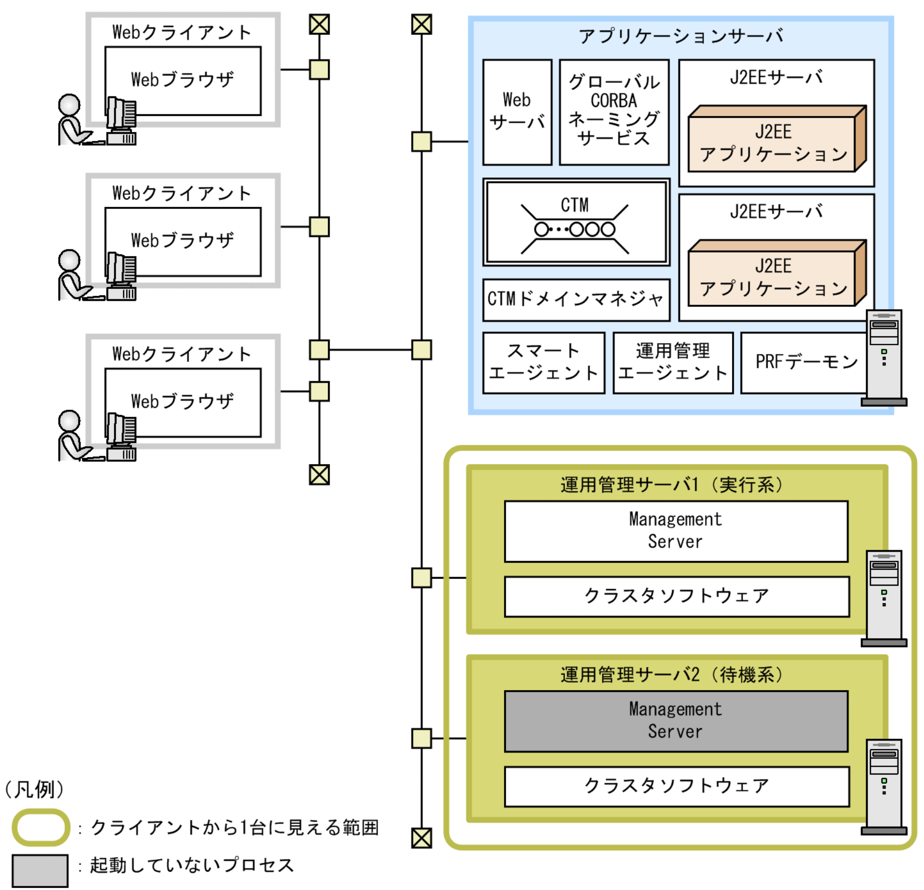
3.11.3　運用管理サーバの実行系と待機系を 1:1 にする構成

運用管理サーバを系切り替えの対象にする場合の構成について説明します。運用管理サーバの実行系と待機系を 1：1 にします。

(1)　システム構成の特徴

運用管理サーバの実行系と待機系を 1：1 にする場合の構成の例を次に示します。

図 3-51　クラスタソフトウェアを使用して運用管理サーバの実行系と待機系を 1:1 にする場合のシステム構成の例



これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

特徴

- 実行系と待機系は 1：1 で構成します。それぞれのマシンに Management Server を配置します。
- Windows Server Failover Cluster を使用する場合は，共有ディスク装置が必要です。

系切り替えの流れ

実行系の運用管理サーバマシンに障害が発生すると，系切り替えが実行されます。系切り替えは，OS，または Management Server がダウンした場合に実行されます。なお，このほか，明示的に系切り替えを実行するコマンドを実行した場合や，これ以外にクラスタソフトウェアが系切り替えの対象にしている事象が発生した場合も，系切り替えが実行されます。

系切り替えが実行されると，待機系の運用管理サーバの Management Server のプロセスが起動されます。

(2)　それぞれのマシンで起動するプロセス

それぞれのマシンに必要なソフトウェアとプロセスについて説明します。

(a)　アプリケーションサーバマシン

アプリケーションサーバに必要なソフトウェアと起動するプロセスは，使用する機能に応じたシステム構成ごとに異なります。使用する機能に応じて必要なソフトウェアとプロセスを配置してください。

(b)　運用管理サーバマシン（実行系）

運用管理サーバマシンには，Application Server をインストールする必要があります。

起動するプロセスは次のとおりです。

- Management Server

(c)　運用管理サーバマシン（待機系）

待機系のアプリケーションサーバマシンにインストールするソフトウェアの構成は，実行系と一致させてください。ただし，系切り替えが発生するまで，プロセスは起動しません。

(d)　クライアントマシン

クライアントマシンには，次のソフトウェアをインストールしてください。

Web クライアント構成の場合
　Web ブラウザ

EJB クライアント構成の場合
　Client（Windows の場合），Application Server

3.11.4　アプリケーションサーバの実行系と待機系を相互スタンバイにする構成

アプリケーションサーバの実行系と待機系を 1：1 にする構成で，それぞれのアプリケーションサーバを実行系として稼働させながら，お互いの待機系にする構成について説明します。この構成を，**相互スタンバイ構成**といいます。また，この構成で構築されたシステムを，**相互系切り替えシステム**といいます。

(1)　システム構成の特徴

この構成は，実行系のアプリケーションサーバマシン 1 台に対して待機系のアプリケーションサーバマシンを 1 台用意しておく構成の一つです。ただし，待機系のアプリケーションサーバを停止させておくのではなく，実行系とは異なる J2EE サーバを動作させておきます。これを，**相互スタンバイ**といいます。

それぞれのアプリケーションサーバに同じ種類の J2EE サーバを配置して，異なる J2EE サーバを起動しておくことで，それぞれのアプリケーションサーバが現用系として動作しながらお互いの予備系として機能します。どちらかのマシンに障害が発生した場合は，系が切り替えられ，もう片方のアプリケーションサーバマシンで両方の J2EE サーバの処理が実行されるようになります。

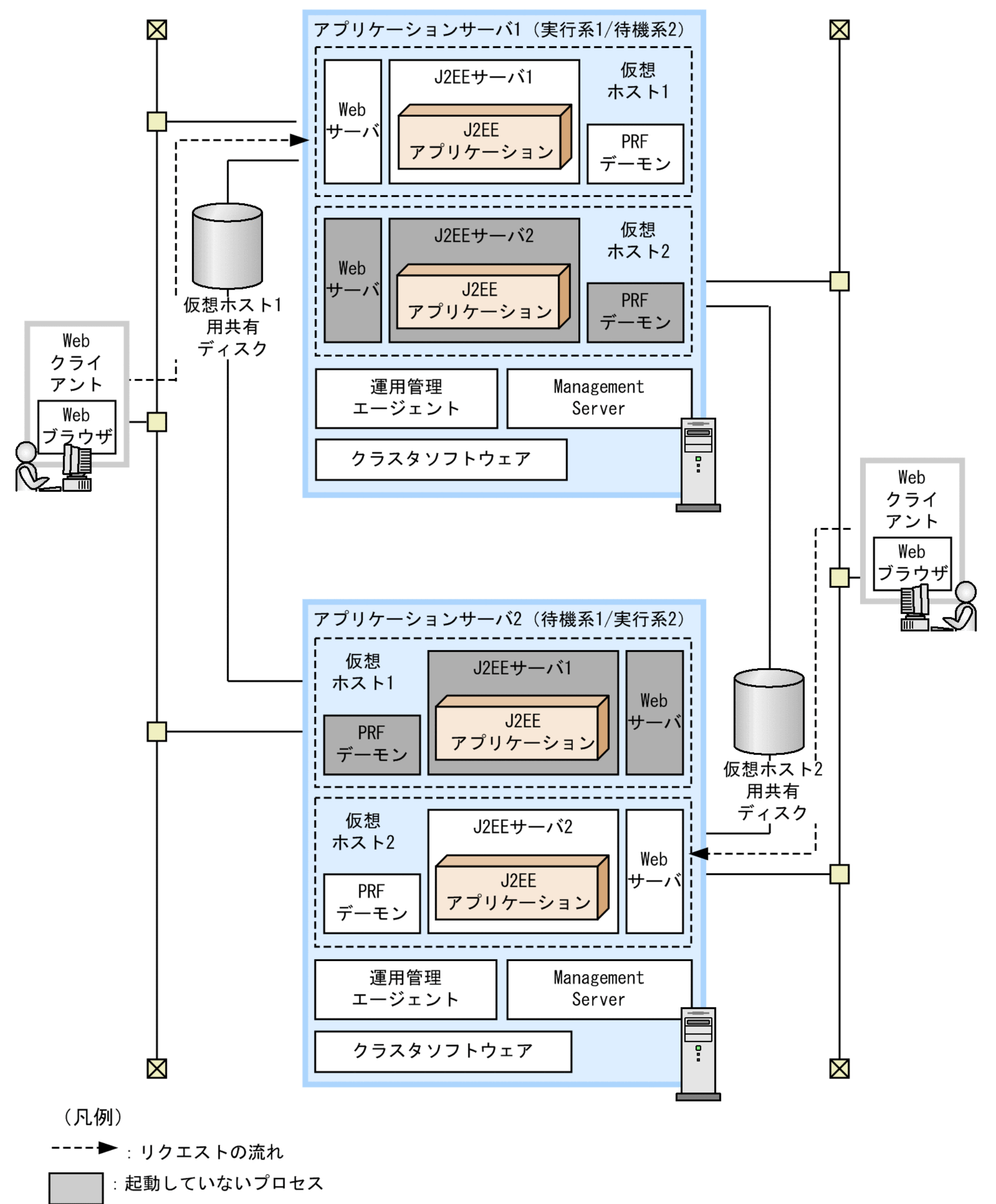
この構成は，次のことが前提になります。

- CORBA ネーミングサービスはインプロセスで起動されていること
- グローバルトランザクションを使用する場合，トランザクションのステータスファイルは共有ディスクに格納されていること
- Management Server を利用して運用していること

Management Server は，アプリケーションサーバと同じマシンに，一つずつ配置して，両方のホストで起動します。それぞれの Management Server は，別の運用管理ドメインを管理します。それぞれの運用管理ドメインの範囲は，Management Server が配置されているアプリケーションサーバマシン内です。

相互系切り替えシステムの構成の例を次の図に示します。この例は，トランザクションサービスを使用する例です。共有ディスクは，トランザクションサービスを使用する場合にだけ必要です。

図 3-52　相互系切り替えシステムの構成の例（トランザクションサービスを使用する場合）



これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

特徴

- 実行系と待機系は 1：1 で構成します。それぞれのマシンに Management Server を配置します。
- 一つの運用管理ドメイン（一つのマシン）内に，二つの仮想ホストを定義します。それぞれの仮想ホストを，実行系のアプリケーションサーバのホスト，待機系のアプリケーションサーバのホストとして，システムを構築します。
- 運用管理ドメイン内のそれぞれの仮想ホストは，一つの運用管理エージェントで起動・停止を制御されます。ただし，それぞれの仮想ホストには実際に運用に使用される IP アドレスとは異なる IP アドレスが割り当てられているため，見かけ上異なるホスト上で動作しているように定義されています。

- 各アプリケーションサーバの運用に使用する IP アドレスは，クラスタソフトウェアによって動的に割り当てられる IP アドレスを使用します。Management Server から運用管理エージェントに対するリクエストの送信には，系切り替えによって他系へ移動しない IP アドレスを使用します。
- グローバルトランザクションを使用する場合は，共有ディスク装置が必要です。共有ディスクは，仮想ホストごとに必要です。
- 図の場合，仮想ホスト単位に LAN を分けていますが，必須ではありません。

系切り替えの流れ

どちらかのアプリケーションサーバマシンに障害が発生すると，系切り替えが実行されます。系切り替えは，Management Server，運用管理エージェント，またはクラスタソフトウェアなどに障害が発生した場合に実行されます。なお，このほか，明示的に系切り替えを実行するコマンドを実行した場合や，これ以外にクラスタソフトウェアが系切り替えの対象にしている事象が発生した場合も，系切り替えが実行されます。

系切り替えが実行されると，待機系だった側のアプリケーションサーバマシンの運用管理エージェントによって，それまで停止していた論理サーバのプロセスが起動されます。

例えば，図の場合に，アプリケーションサーバ 1 に障害が発生した場合は，クラスタソフトウェアによってアプリケーションサーバ 2 への系切り替えが実行され，アプリケーションサーバ 2 の運用管理エージェントによって，アプリケーションサーバ 2 上の J2EE サーバ 1 およびそれ以外の論理サーバのプロセスが起動されます。

(2) それぞれのマシンで起動するプロセス

それぞれのマシンに必要なソフトウェアとプロセスについて説明します。

(a) アプリケーションサーバマシン（実行系 1／待機系 2）

アプリケーションサーバマシンには，Application Server をインストールする必要があります。

相互系切り替えシステムのアプリケーションサーバマシンで必ず起動するプロセスは次のとおりです。

- 運用管理エージェント
- Management Server

これ以外にアプリケーションサーバに必要なソフトウェアおよびプロセスは，使用する機能に応じたシステム構成ごとに異なります。使用する機能に応じて必要なソフトウェアとプロセスを配置してください。

なお，系切り替えの対象になるプロセスは，仮想ホスト 1 のプロセスだけを起動してください。

(b) アプリケーションサーバマシン（待機系 1／実行系 2）

待機系のアプリケーションサーバマシンにインストールするソフトウェアおよびプロセスの構成は，実行系 1／待機系 2 と一致させてください。

なお，系切り替えの対象になるプロセスは，仮想ホスト 2 のプロセスだけを起動してください。

(c) クライアントマシン

クライアントマシンには，次のソフトウェアをインストールしてください。

Web クライアント構成の場合
　Web ブラウザ

EJB クライアント構成の場合
　Client（Windows の場合），Application Server

3.11.5 リカバリ専用サーバを使用する場合の構成（N:1 リカバリシステム）

N 台の実行系に対して 1 台の待機系（**リカバリ専用サーバ**）を用意する構成について説明します。この構成を，**N:1 リカバリシステム**といいます。

(1) システム構成の特徴

この構成は，J2EE サーバを冗長化した構成でグローバルトランザクションを使用する場合に，特定の J2EE サーバにトラブルが発生したときにトランザクションを解決してリソースを解放するためのシステム構成です。N 台の J2EE サーバを冗長化するためには，負荷分散機などを使用します。

次のことが前提になります。

- グローバルトランザクションを使用するシステムで利用されること
- CORBA ネーミングサービスはインプロセスで起動されていること
- トランザクションのステータスファイルは共有ディスクに格納されていること
- Management Server を利用して運用していること

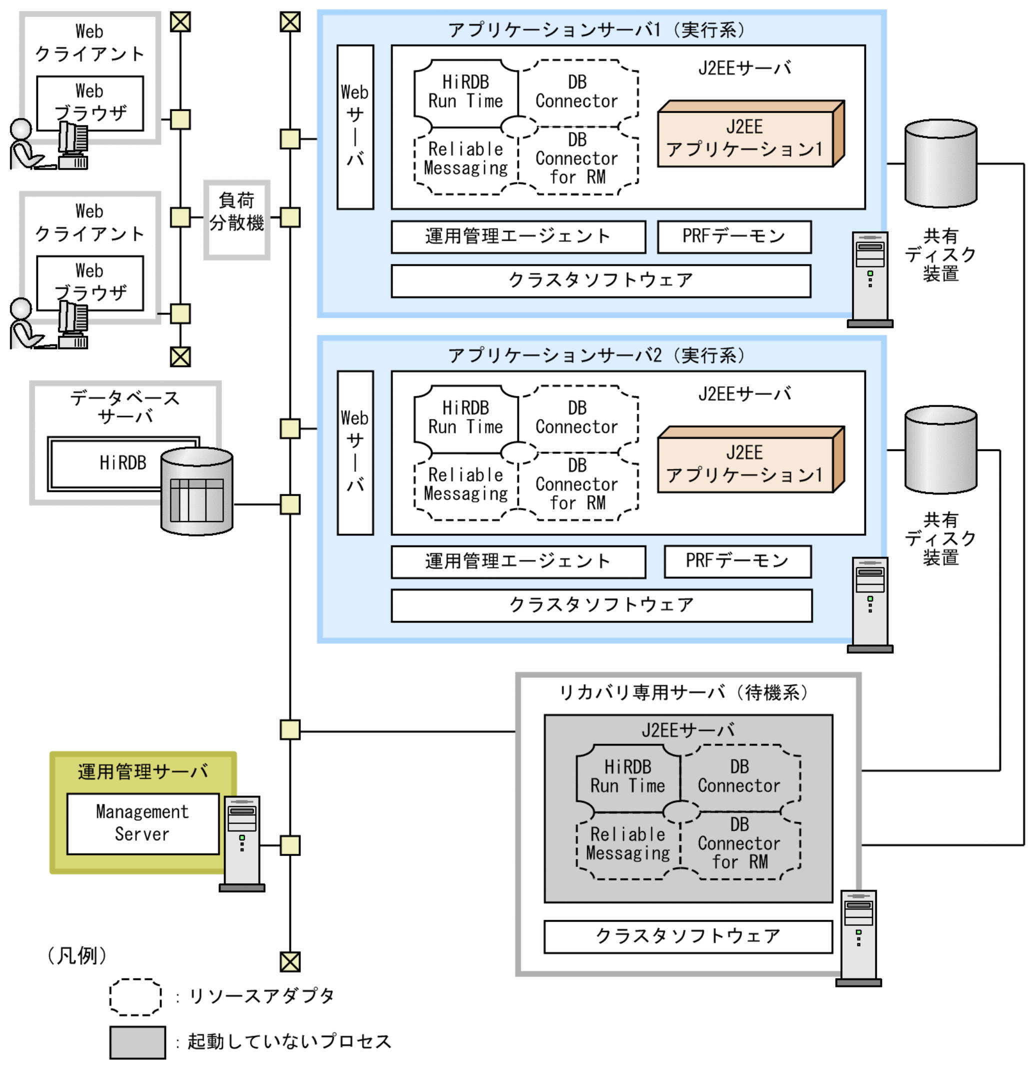
なお，N:1 リカバリシステムで系切り替えの対象になるのは，アプリケーションサーバです。

また，N:1 リカバリシステムを構成する場合，実行系で使用するリソースアダプタが，待機系にも必要です。ここでは，リソースアダプタの配置とシステム構成の関係について，次の 3 種類の構成を例として説明します。

- N 台の実行系マシンで，J2EE アプリケーションとリソースアダプタの構成がすべて同じ場合
- N 台の実行系マシンで，J2EE アプリケーションの構成は異なり，リソースアダプタはすべて同一の場合
- N 台の実行系マシンで，J2EE アプリケーションとリソースアダプタの構成がすべて異なる場合

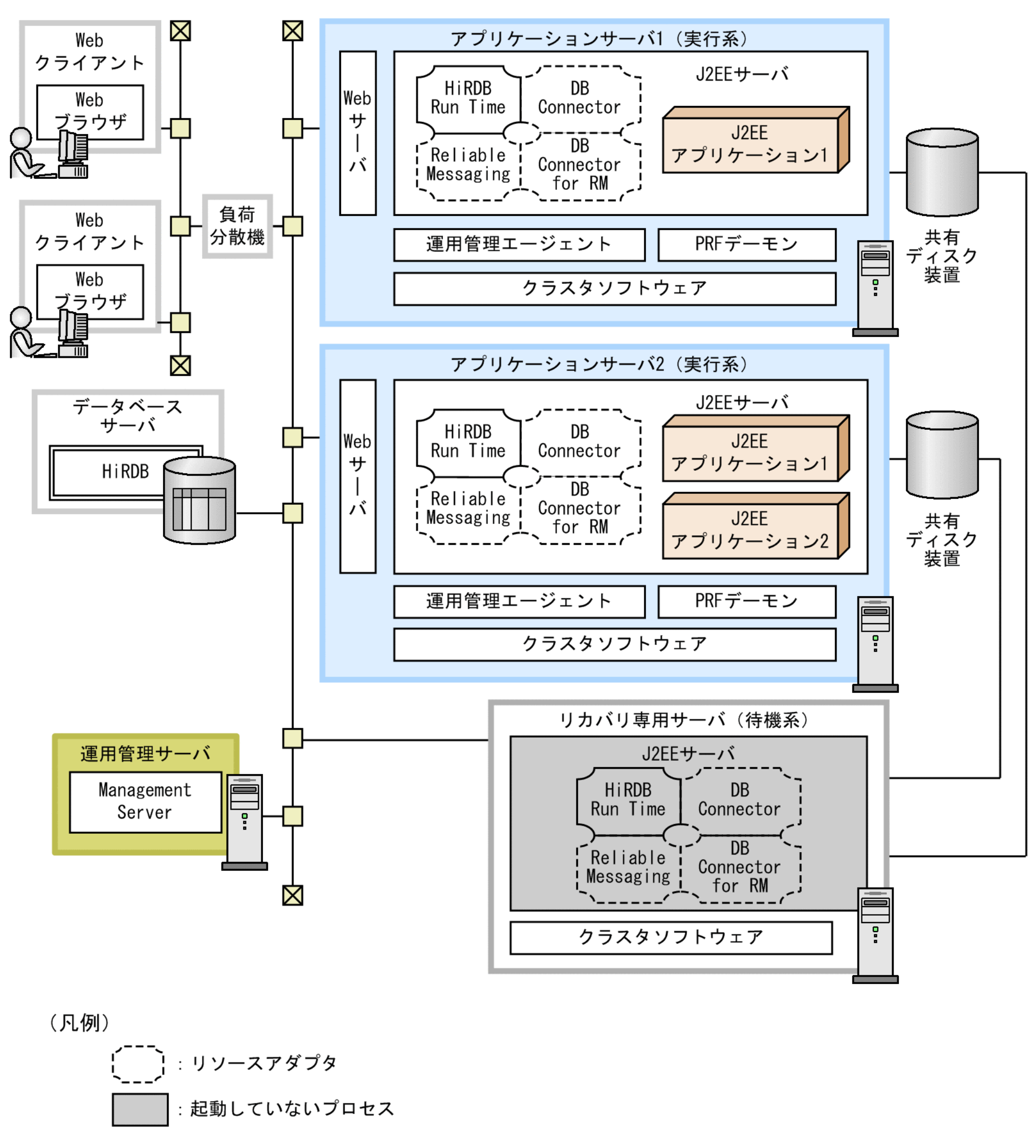
次にそれぞれの構成の例を示します。

図 3-53　N:1 リカバリシステムのシステム構成の例（N 台の実行系マシンで，J2EE アプリケーションとリソースアダプタの構成がすべて同じ場合）



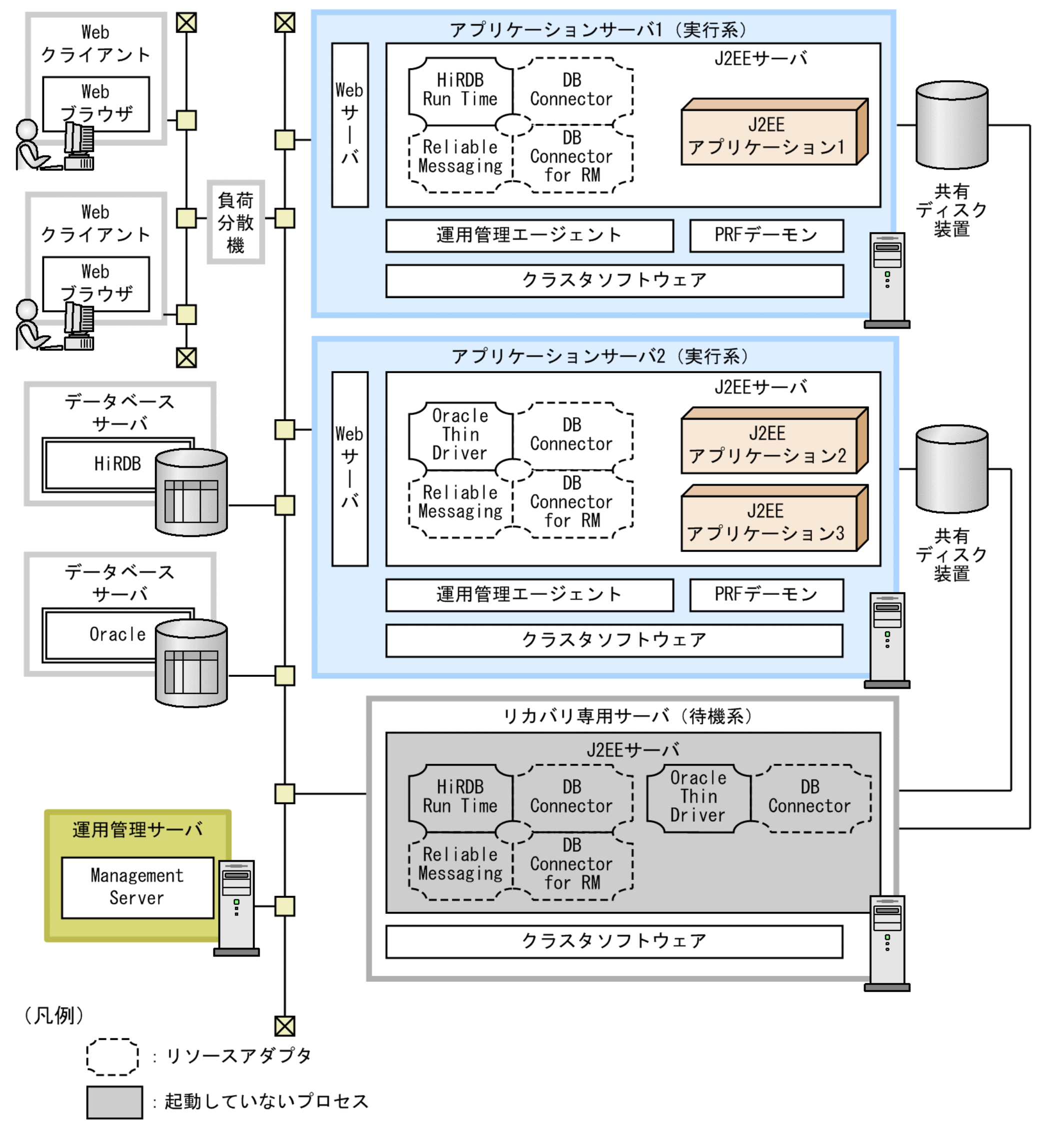
これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

図 3-54　N:1 リカバリシステムのシステム構成の例（N 台の実行系マシンで，J2EE アプリケーションの構成は異なり，リソースアダプタはすべて同一の場合）



これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

図 3-55　N:1 リカバリシステムのシステム構成の例（N 台の実行系マシンで，J2EE アプリケーションとリソースアダプタの構成がすべて異なる場合）



これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

特徴

- 実行系と待機系は N：1 で構成します。
- 共有ディスク装置が必要です。共有ディスク装置には実行系の個数（N 個）分のボリュームグループが必要です。
- CORBA ネーミングサービスはインプロセスで起動します。
- 待機系には，実行系と同じリソースアダプタが必要です。実行系の J2EE サーバごとに異なるリソースアダプタがインポートされている場合，待機系の J2EE サーバにはそれらすべてをインポートする必要があります。
  なお，待機系の J2EE サーバに J2EE アプリケーションは不要です。

- データベースがクラスタ構成になっている場合でも，アプリケーションサーバでは仮想アドレス（論理アドレス）だけを認識していれば，接続できます。

系切り替えの流れ

実行系のアプリケーションサーバで起動されている J2EE サーバのうち，どれかの J2EE サーバにトラブルが発生すると，クラスタソフトウェアによってリカバリ専用サーバの J2EE サーバが起動され，トラブルが発生した J2EE サーバが使用していたトランザクションが決着されます。そのあとで，トラブルが発生した J2EE サーバマシンのクラスタソフトウェアと，それに対応する待機系のクラスタサービスが停止されます。

(2)　それぞれのマシンで起動するプロセス

それぞれのマシンに必要なソフトウェアとプロセスについて説明します。

(a)　アプリケーションサーバマシン（実行系）

実行系のアプリケーションサーバマシンには，Application Server をインストールする必要があります。

クラスタソフトウェアを使用する場合に必ず起動するプロセスは次のとおりです。

- 運用管理エージェント

これ以外にアプリケーションサーバに必要なソフトウェアと起動するプロセスは，使用する機能に応じたシステム構成ごとに異なります。使用する機能に応じて必要なソフトウェアとプロセスを配置してください。

(b)　アプリケーションサーバマシン（待機系）

待機系のアプリケーションサーバマシンには，Application Server をインストールする必要があります。

また，実行系のアプリケーションサーバマシン上の J2EE サーバにインポートされているすべてのリソースアダプタをインポートしてください。

なお，待機系のアプリケーションサーバマシンは，コールドスタンバイの状態になります。

系切り替えが発生するまで，プロセスは起動しません。系切り替えが発生すると，J2EE サーバのプロセスが，リカバリモードで起動されます。

(c)　運用管理サーバマシン

運用管理サーバマシンには，Application Server をインストールする必要があります。

起動するプロセスは次のとおりです。

- Management Server

(d)　クライアントマシン

クライアントマシンには，次のソフトウェアをインストールしてください。

Web クライアント構成の場合

Web ブラウザ

EJB クライアント構成の場合

Client（Windows の場合），Application Server

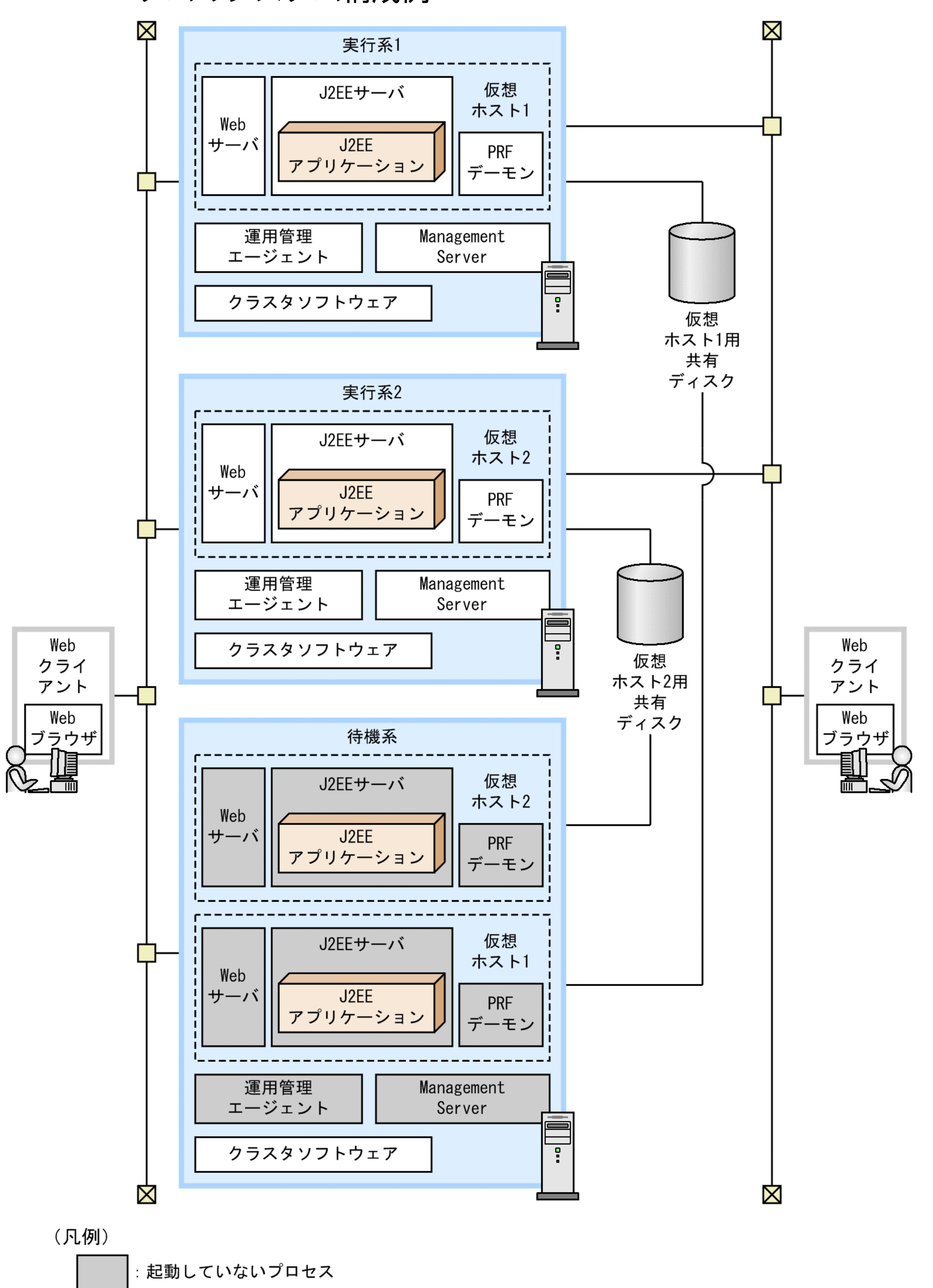
3.11.6　ホスト単位管理モデルの実行系と待機系を N：1 にする構成

ホスト単位管理モデルを系切り替えの対象にする場合の構成について説明します。

(1)　システム構成の特徴

ホスト単位管理モデルを N：1 で構成します。システム構成の例を次に示します。

図 3-56　クラスタソフトウェアを使用してホスト単位管理モデルを対象にした系切り替えシステムのシステム構成例



これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

特徴

- アプリケーションサーバの実行系 N 台と待機系 1 台を配置し，それぞれに Management Server と運用管理エージェントを配置している構成です。
- 論理サーバは別々のホスト内に配置されているが，外部からは同じ論理サーバとして動作するのに対し，Management Server は別々のホスト内で動作し，それぞれ運用管理ドメインを持ちます。

一つの運用管理ドメイン内に一つの仮想ホストを定義し，それぞれを現用系アプリケーションサーバのホスト，予備系アプリケーションサーバのホストとして構築します。

- 各アプリケーションサーバの運用に使用する IP アドレスは，クラスタソフトウェアによって割り当てられる仮想 IP アドレスを使用し，Management Server，運用管理エージェントの IP アドレスには実 IP アドレスを使用します。
- グローバルトランザクションを使用する場合は，共有ディスク装置が必要です。共有ディスクは，仮想ホストごとに必要です。

> **参考**
>
> 仮想ホストとは，運用管理エージェントによってアプリケーションサーバの起動，停止を制御しますが，運用に使用する IP アドレスと同じ IP アドレスを割り当て，見かけ上同じホストであるかのように定義したものです。

系切り替えの流れ

実行系のアプリケーションサーバマシンに障害が発生すると，系切り替えが実行されます。系切り替えは，Management Server，運用管理エージェント，またはクラスタソフトウェアなどに障害が発生した場合に実行されます。

系切り替えが実行されると，待機系のアプリケーションサーバマシンのクラスタソフトウェアによって運用管理エージェントと Management Server が起動されます。起動された運用管理エージェントによって，それまで停止していた論理サーバのプロセスが起動されます。

(2)　それぞれのマシンで起動するプロセス

それぞれのマシンに必要なソフトウェアとプロセスについて説明します。

(a)　アプリケーションサーバマシン（実行系／待機系）

アプリケーションサーバマシンには，Application Server をインストールする必要があります。

ホスト単位管理モデルを対象とした系切り替えシステムのアプリケーションサーバマシンで必ず起動するプロセスは次のとおりです。

- 運用管理エージェント
- Management Server

これ以外にアプリケーションサーバに必要なソフトウェアおよびプロセスは，使用する機能に応じたシステム構成ごとに異なります。使用する機能に応じて必要なソフトウェアとプロセスを配置してください。

(b)　クライアントマシン

クライアントマシンには，次のソフトウェアをインストールしてください。

Web クライアント構成の場合

　Web ブラウザ

EJB クライアント構成の場合

Client（Windows の場合），Application Server

3.12　性能解析トレースファイルを出力するプロセスを配置する

この節では，性能解析トレースファイルを出力するプログラムの配置について説明します。このプロセスは，アプリケーションサーバのシステムには必ず配置するプロセスです。アプリケーションサーバマシンには必ず配置してください。EJB クライアントマシンへの配置は任意です。

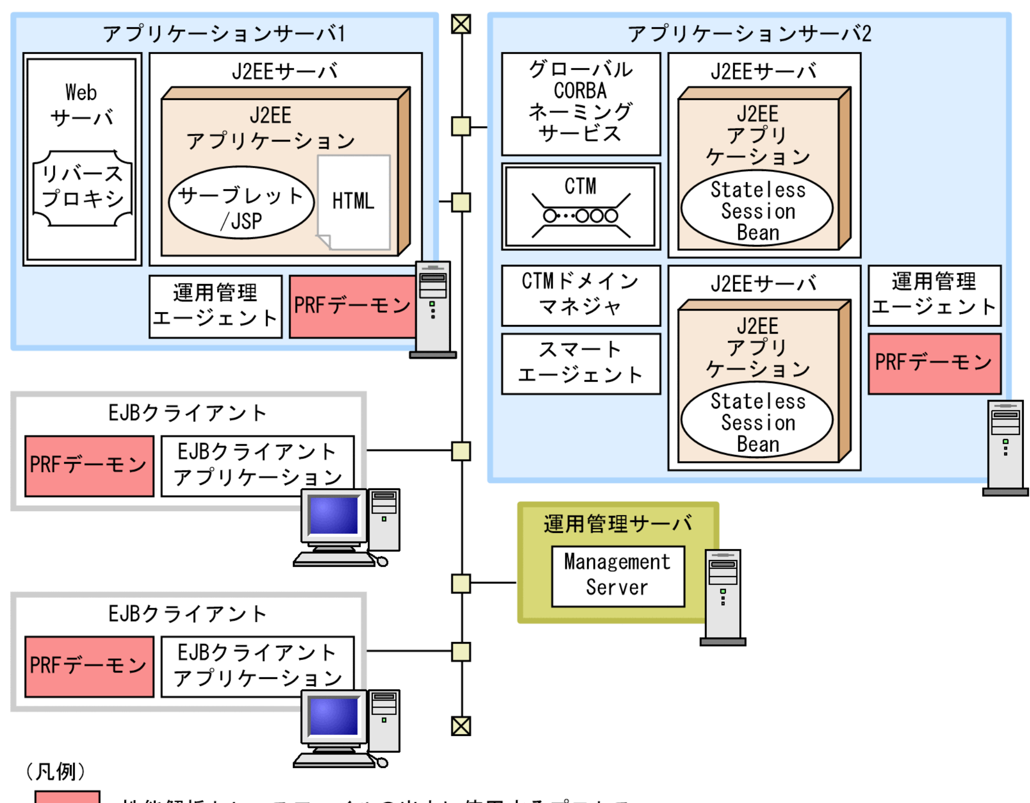
性能解析トレースファイルは，PRF デーモン（パフォーマンストレーサ）によって出力されます。

3.12.1　システム構成の特徴

PRF デーモンは，アプリケーションサーバおよび EJB クライアントアプリケーションを実行するマシンに配置します。

PRF デーモンを配置する例を次の図に示します。

図 3-57　PRF デーモンを配置する例



これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

特徴

- システム性能を解析するためのトレースファイルを出力できます。
- トレースファイルは，トラブルが発生した場合のエラー個所の特定に利用できます。

性能解析トレースを出力する仕組み

クライアントとサーバ間でリクエストを処理するときに，アプリケーションサーバまたは EJB クライアントアプリケーションの各機能レイヤは，性能解析情報をバッファに出力します。PRF デーモンは，アプリケーションサーバまたは EJB クライアントアプリケーションの各機能レイヤがバッファに出力したトレース情報を，各マシン内のファイルに出力します。ファイル出力されたトレース情報は，Management Server を利用して収集できます。

3.12.2 それぞれのマシンに必要なソフトウェアと起動するプロセス

性能解析トレースファイルを出力するために起動するプロセスは次のとおりです。

- PRF デーモン

これ以外にアプリケーションサーバに必要なソフトウェアと起動するプロセスは，使用する機能に応じたシステム構成ごとに異なります。使用する機能に応じて必要なソフトウェアとプロセスを配置してください。

3.13　アプリケーションサーバ以外の製品との連携を検討する

この節では，アプリケーションサーバ以外の製品と連携する場合のシステム構成について説明します。

ここでは，次の構成について説明します。

- JP1 を使用して運用する場合の構成
  運用管理プログラムとして JP1 を使用する場合の構成です。
- TP1 インバウンド連携機能を使用して OpenTP1 の SUP から Message-driven Bean を呼び出す場合の構成
  TP1 インバウンド連携機能を使用して OpenTP1 の SUP からアプリケーションサーバ上の Message-driven Bean を呼び出す場合の構成です。
- CTM ゲートウェイ機能を利用して EJB クライアント以外から Stateless Session Bean を呼び出す構成
  クライアントとして TPBroker クライアントまたは TPBroker OTM クライアントを使用する場合の構成です。

3.13.1　JP1 を使用して運用する場合の構成

JP1 を使用して運用する場合の運用管理プログラムの配置について説明します。

JP1/IM と連携する場合，またはカスタムジョブの定義やシナリオなどアプリケーションサーバが提供する機能を利用して JP1/AJS と連携する場合には，Management Server を利用して運用していることが前提になります。「3.9.1　運用管理サーバに Management Server を配置する構成」または「3.9.2　マシン単位に Management Server を配置する構成」を参照して Management Server を利用するシステムの構成を決定した上で，JP1 のプログラムを配置してください。JP1 との連携で必要なプログラムについては，マニュアル「アプリケーションサーバ 機能解説 運用／監視／連携編」の「12.　JP1 と連携したシステムの運用」を参照してください。

なお，JP1/AJS を利用する場合に，アプリケーションサーバの機能を使用しないでジョブまたはシナリオの定義をするときには，Management Server を利用しなくても連携できます。この場合は，「3.9.3　コマンドで運用する場合の構成」に示すシステムの構成を検討した上で，JP1 のマニュアルを参照して，システム構成を決定してください。

3.13.2　TP1 インバウンド連携機能を使用して OpenTP1 の SUP から Message-driven Bean を呼び出す場合の構成

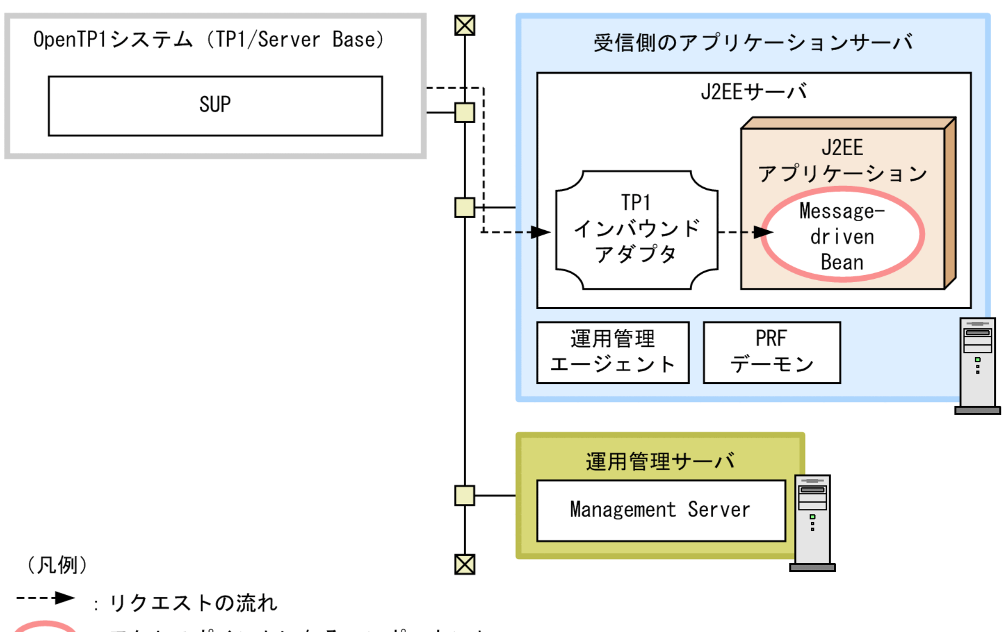
TP1 インバウンド連携機能を使用して，OpenTP1 システムの SUP から J2EE サーバ上の Message-driven Bean を呼び出す構成について説明します。

(1)　システム構成の特徴

メッセージ駆動型のシステムの一つです。

TP1 インバウンド連携機能を使用する場合のメッセージ駆動型のシステム構成の例を次の図に示します。

図 3-58　メッセージ駆動型のシステム構成の例（TP1 インバウンド連携機能を使用する場合）



これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

特徴

TP1/Server Base 上で動作する SUP から送信されたリクエストを J2EE サーバ上で動作する TP1 インバウンドアダプタで受け付けて，処理を実行します。SUP からは，SPP を呼び出すときと同様の手順※で，J2EE サーバ上の Message-driven Bean にアクセスできます。

注※

一部の手順は異なります。詳細は，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(コンテナ共通機能)」の「4.　OpenTP1 からのアプリケーションサーバの呼び出し（TP1 インバウンド連携機能)」を参照してください。

リクエストの流れ

アクセスポイントである Message-driven Bean，およびリソースアダプタである TP1 インバウンドアダプタのライブラリは，アプリケーションサーバの J2EE サーバ上で動作します。

OpenTP1 システムの TP1/Server Base 上で動作する SUP からからのリクエストをアプリケーションサーバ上の TP1 インバウンドアダプタで受け付け，Message-driven Bean を呼び出します。

(2)　それぞれのマシンに必要なソフトウェアと起動するプロセス

それぞれのマシンに必要なソフトウェアと起動するプロセスについて説明します。

(a)　アプリケーションサーバマシン

アプリケーションサーバマシン（サーバ側のアプリケーションサーバマシン）には，Application Server をインストールする必要があります。

起動するプロセスは次のとおりです。

- J2EE サーバ
- 運用管理エージェント
- PRF デーモン

(b)　OpenTP1 システム（TP1/Server Base）

OpenTP1 システムには，TP1/Server Base をインストールする必要があります。

起動する必要があるプロセスは，SUP の実行に必要なプロセスです。

(c)　運用管理サーバマシン

運用管理サーバマシンには，Application Server をインストールする必要があります。

起動するプロセスは次のとおりです。

- Management Server

3.13.3　CTM ゲートウェイ機能を利用して EJB クライアント以外から Stateless Session Bean を呼び出す構成

CTM を使用するシステムの場合，EJB クライアントのほか，次に示すクライアントからアプリケーションサーバ上で動作する J2EE アプリケーションを呼び出せます。

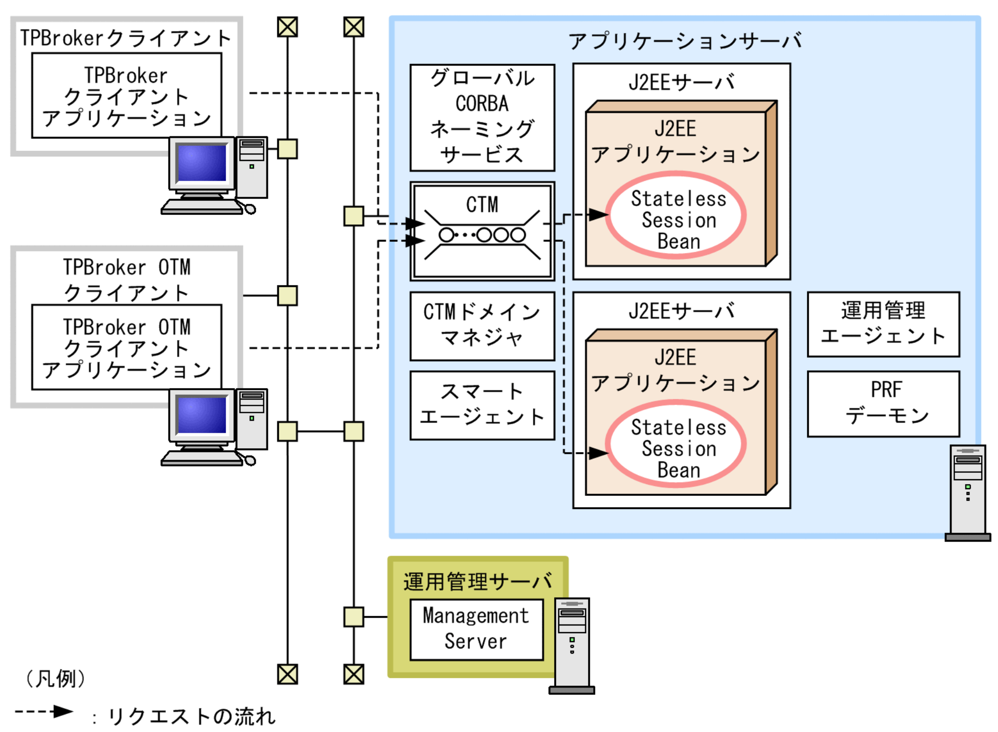
- TPBroker クライアント
  TPBroker Version 5 以降で開発されたクライアントアプリケーションです。
- TPBroker OTM クライアント
  TPBroker Object Transaction Monitor で開発されたクライアントアプリケーションです。

これらのクライアントから J2EE サーバ上のアプリケーションを呼び出すためのクライアントアプリケーションの開発方法については，CTM の CORBA／OTM ゲートウェイ機能についてのドキュメントを参照してください。

この構成では，TPBroker クライアントまたは TPBroker OTM クライアントから J2EE アプリケーションを呼び出すためのゲートウェイとしての機能を，CTM が提供します。

TPBroker クライアントまたは TPBroker OTM クライアントから CTM 経由で J2EE アプリケーションを呼び出すシステム構成の例を次の図に示します。

図 3-59　TPBroker クライアントまたは TPBroker OTM クライアントから CTM 経由で J2EE アプリケーションを呼び出す構成の例



これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

TPBroker クライアントまたは TPBroker OTM クライアントからのリクエストは，CTM が提供するプロセス群を経由して，J2EE サーバ上の J2EE アプリケーションに渡されます。

なお，CTM のプロセス群のうち，リクエストを受け付けるプロセスは，TPBroker クライアントの場合と TPBroker OTM クライアントの場合で異なります。TPBroker クライアントの場合，CTM レギュレータプロセスによってリクエストが受け付けられます。TPBroker OTM クライアントの場合，OTM ゲートウェイプロセスによってリクエストが受け付けられます。なお，CTM レギュレータプロセス，および OTM ゲートウェイプロセスは，CTM デーモンを起動するときに，同時に起動されるプロセスです。

3.14　任意のプロセスを運用管理の対象にする

この節では，ユーザが定義する任意のプロセスを，Management Server による運用管理の対象とするためのシステム構成について説明します。

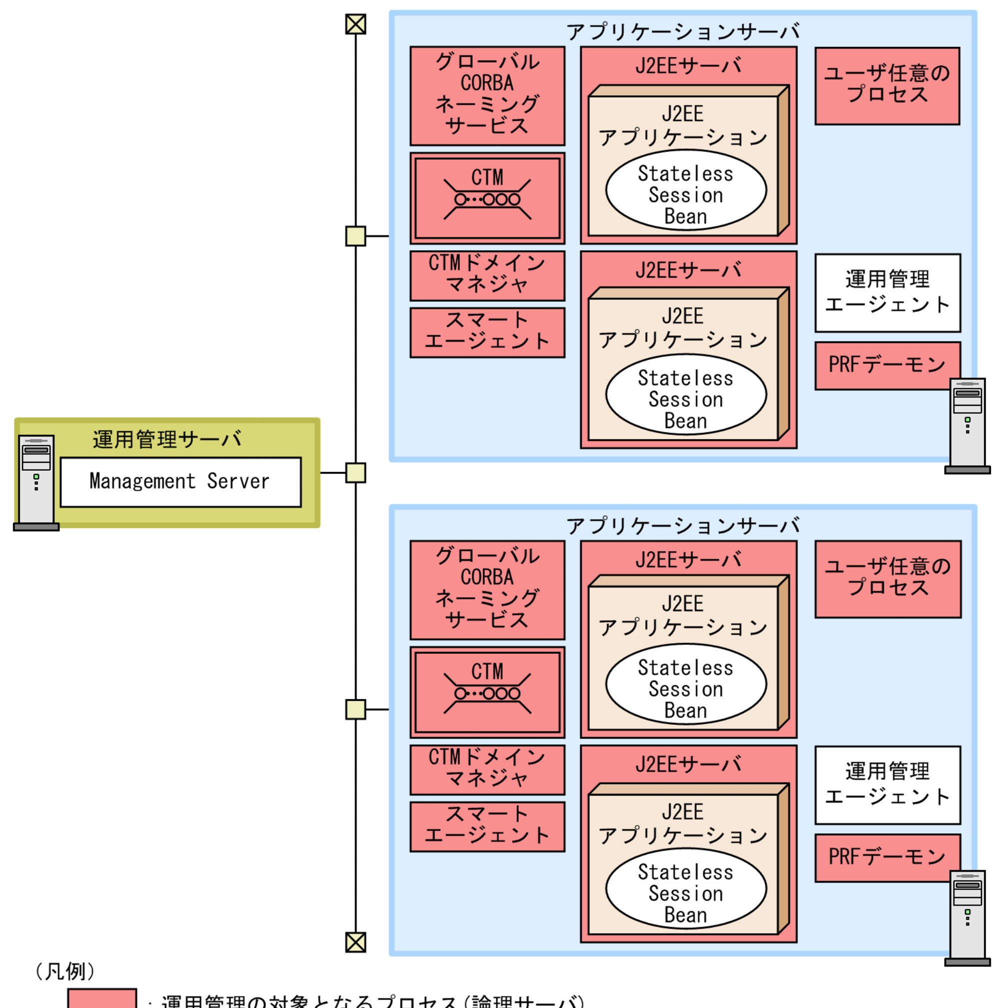
Management Server による運用管理の対象とする任意のプロセスを，論理サーバとして定義したユーザサーバとして配置します。論理サーバとして定義したユーザサーバを**論理ユーザサーバ**といいます。論理ユーザサーバとして定義しておくと，Smart Composer 機能のコマンドを使用して，任意のプロセスを開始したり，停止したり，任意のプロセスのステータスを監視したりできるようになります。

3.14.1　システム構成の特徴

このシステム構成は，Management Server で運用管理するシステムで，論理サーバとして定義したユーザサーバを配置する構成です。運用管理に使用する Management Server は，運用管理サーバに配置することも，マシン単位に配置することもできます。Management Server の配置については，「3.9　運用管理プロセスの配置を検討する」を参照してください。

ユーザサーバを配置する構成の例を次の図に示します。この構成例では，アプリケーションサーバマシンでユーザ任意のプロセスを実行します。このユーザ任意のプロセスを Management Server の運用管理の対象とするために，論理ユーザサーバとして定義して，配置しています。

図 3-60　ユーザサーバを配置する構成の例



これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

特徴

- Management Server を利用して任意のプロセスを運用管理できます。

3.14.2　それぞれのマシンで起動するプロセス

任意のプロセスをユーザサーバとして配置する構成の場合に，それぞれのマシンに必要なソフトウェアと起動するプロセスについて説明します。

(1)　アプリケーションサーバマシン

アプリケーションサーバマシンに必要なソフトウェアと起動するプロセスは，使用する機能に応じたシステム構成ごとに異なります。使用する機能に応じて必要なソフトウェアとプロセスを配置してください。

(2)　運用管理サーバマシン

運用管理サーバマシンには，Application Server をインストールする必要があります。

起動するプロセスは次のとおりです。

- Management Server

3.15 アプリケーションサーバのプロセスが使用する TCP/UDP のポート番号

ここでは，アプリケーションサーバのプロセスが使用する TCP/UDP のポート番号について説明します。

デフォルト値が「(浮動)」のポートは，ポート番号を明示的に固定しない場合にアプリケーションサーバによって自動的に番号が付けられるポートです。

アプリケーションサーバが使用する TCP/UDP のポート番号の説明を次の表に示します。なお，ご使用の OS によっては，ネットワーク単位ではなくホスト単位でファイアウォールが設定されているものがあります。これらのファイアウォールでは，localhost（127.0.0.1）以外との通信は，同一ホスト内でもファイアウォールのフィルタリングの対象になる場合があります。この場合は，ホスト内でしか通信しないポートであっても，フィルタで通信を許可する設定にしてください。

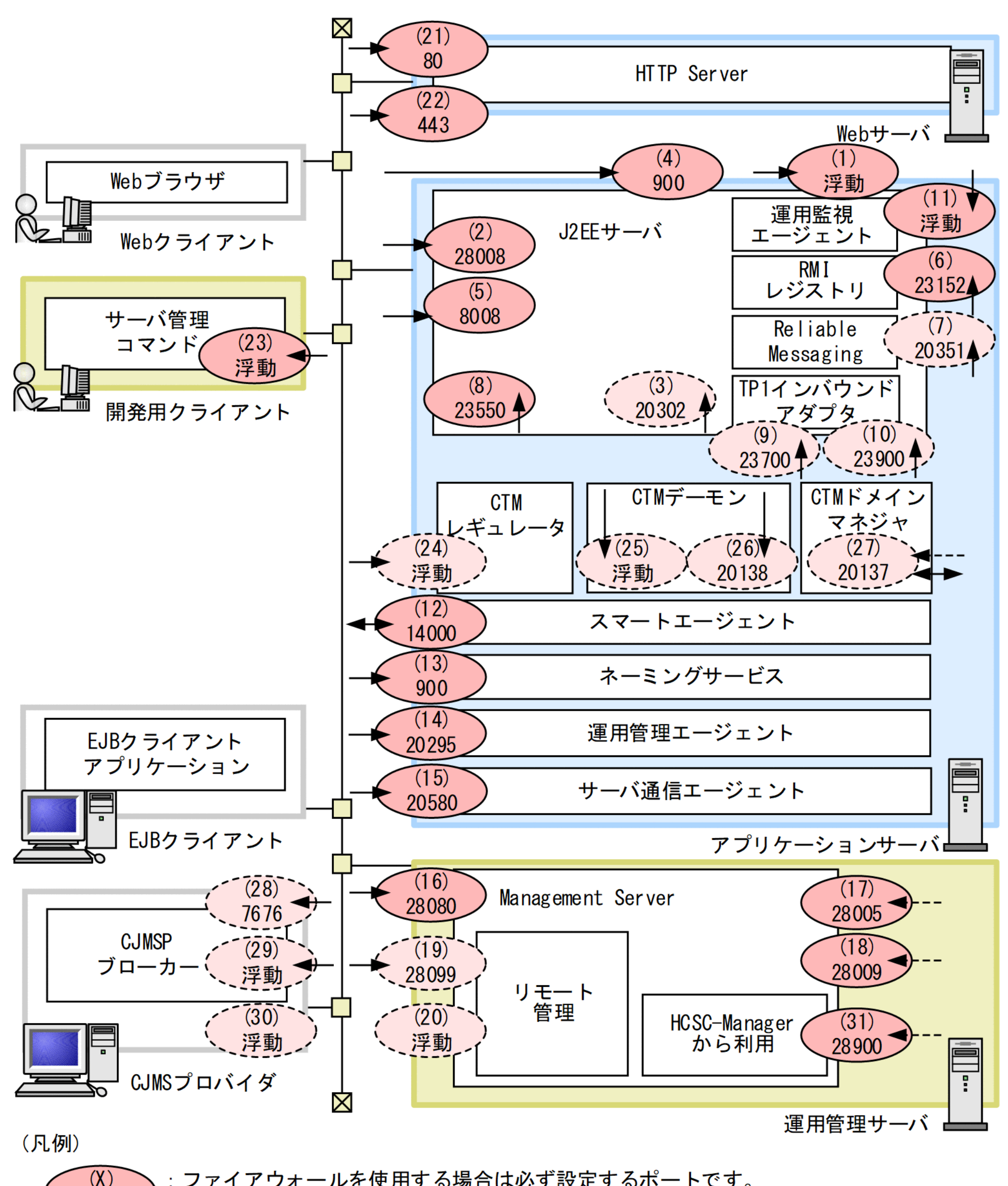
表 3–3 アプリケーションサーバが使用する TCP/UDP のポート番号

項番	プロセス	説明	デフォルト値
(1)	J2EE サーバ	EJB コンテナのリクエスト受付ポート。	(浮動)
(2)		管理用通信ポート。	28008
(3)		トランザクションサービス使用時のトランザクションリカバリ処理通信ポート。 トランザクションサービス使用時に必要です。	20302
(4)		インプロセスで起動するネーミングサービスのリクエスト受付ポート。	900
(5)		HTTP 通信/WebSocket 通信のリクエスト受付ポート。	8008
(6)		RMI レジストリのリクエスト受付ポート。	23152
(7)		共有キューを使用して複数システム間でのアプリケーション連携をする場合のイベント受信用ポート。	20351
(8)		稼働情報取得時のリクエスト受付ポート。	23550
(9)		OpenTP1 からの RPC 要求を待ち受けるポート。	23700
(10)		OpenTP1 からの同期点要求を待ち受けるポート。	23900
(11)	運用監視エージェント	運用監視エージェントの通信用ポート。	(浮動)
(12)	スマートエージェント	スマートエージェントの通信用ポート環境変数。 UDP による双方向通信に必要です。	14000
(13)	ネーミングサービス	ネーミングサービスのリクエスト受付ポート引数（TPBroker が利用）。	900
(14)	運用管理エージェント	運用管理エージェントが Management Server との通信に使用するポート。	20295
(15)	サーバ通信エージェント	サーバ通信エージェントが仮想サーバマネージャとの通信に使用するポート。	20580
(16)	Management Server	Management Server の http ポート。	28080
(17)		Management Server の終了要求ポート。	28005

項番	プロセス	説明	デフォルト値
		ホスト内通信に必要です。	
(18)		Management Server の内部通信ポート。 ホスト内通信に必要です。	28009
(19)		Manager リモート管理機能への接続ポート。	28099
(20)		Manager リモート管理機能へのクライアント接続ポート。	（浮動）
(21)	HTTP Server	HTTP Server の http ポート。	80
(22)		HTTP Server の https ポート。	443
(23)	サーバ管理コマンド	サーバ管理コマンドが J2EE サーバと通信するポート。	（浮動）
(24)	CTM レギュレータ	CTM レギュレータが EJB クライアントからのリクエストを受け付けるポートの基底値。基底値＋プロセス数だけ使用します。 CTM 使用時に必要です。	（浮動）
(25)	CTM デーモン	CTM デーモンが EJB クライアントからのリクエストを受け付けるポート。 CTM 使用時に必要です。	（浮動）
(26)		CTM デーモンがほかのデーモンや J2EE サーバなどと通信するポート。 CTM 使用時に必要です。	20138
(27)	CTM ドメインマネジャ	CTM ドメインマネジャがほかの CTM ドメインマネジャと通信するポート。 CTM 使用時に，TCP および UDP 通信（ブロードキャスト）をするために必要です。	20137
(28)	CJMSP ブローカー	CJMS プロバイダのブローカーがリソースアダプタやコマンドからのリクエストを受け付けるためのポート。	7676
(29)		CJMS プロバイダのブローカーがリソースアダプタとコネクションを確立するためのポート。	（浮動）
(30)		CJMS プロバイダのブローカーがコマンドとコネクションを確立するためのポート。	（浮動）
(31)	Management Server	HCSC-Manager から利用される Management Server の内部通信用ポート。	28900

アプリケーションサーバのプロセスが使用する TCP/UDP のポート番号について，次の図に示します。(x)は表の項番と対応しています。

図 3-61　アプリケーションサーバが使用する TCP/UDP のポート番号



これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

ポート番号の指定個所を次の表に示します。表の項番は図中の項番と対応しています。

表 3-4　アプリケーションサーバが使用する TCP/UDP のポート番号の指定個所

項番	定義ファイル	設定対象	パラメタ名※1
(1)	簡易構築定義ファイル	論理 J2EE サーバ（j2ee-server）	vbroker.se.iiop_tp.scm.iiop_tp.listener.port
(2)	簡易構築定義ファイル	論理 J2EE サーバ（j2ee-server）	ejbserver.http.port
(3)	簡易構築定義ファイル	論理 J2EE サーバ（j2ee-server）	ejbserver.distributedtx.recovery.port
(4)	簡易構築定義ファイル	論理 J2EE サーバ（j2ee-server）	webserver.connector.nio_http.port
(5)	簡易構築定義ファイル	論理 J2EE サーバ（j2ee-server）	ejbserver.rmi.naming.port
(6)	Connector 属性ファイル	Reliable Messaging	<config-property>タグに指定する RMSHPort※2
(7)	簡易構築定義ファイル	論理 J2EE サーバ（j2ee-server）	ejbserver.rmi.remote.listener.port
(8)	Connector 属性ファイル	TP1 インバウンドアダプタ	<config-property>タグに指定する scd_port※3
(9)	Connector 属性ファイル	TP1 インバウンドアダプタ	<config-property>タグに指定する trn_port※3
(10)	簡易構築定義ファイル	論理 J2EE サーバ（j2ee-server）	mngagent.connector.port
(11)	簡易構築定義ファイル	論理スマートエージェント（smart-agent）	smartagent.port
(12)	簡易構築定義ファイル	論理 J2EE サーバ（j2ee-server）	ejbserver.naming.port
(13)	adminagent.properties	運用管理エージェント	adminagent.adapter.port キー
(14)	sinaviagent.properties※4	サーバ通信エージェント	sinaviagent.port キー
(15)	mserver.properties	Management Server	webserver.connector.http.port キー
(16)	mserver.properties	Management Server	webserver.shutdown.port キー
(17)	mserver.properties	Management Server	webserver.connector.ajp13.port キー
(18)	mserver.properties	Management Server	com.cosminexus.mngsvr.management.port キー
(19)	mserver.properties	Management Server	com.cosminexus.mngsvr.management.listen.port キー
(20)	簡易構築定義ファイル	論理 Web サーバ（web-server）	Listen
(21)	簡易構築定義ファイル	論理 Web サーバ（web-server）	Listen

項番	定義ファイル	設定対象	パラメタ名※1
(22)	usrconf.properties（サーバ管理コマンド用システムプロパティファイル）	サーバ管理コマンド	vbroker.se.iiop_tp.scm.iiop_tp.listener.port キー
(23)	簡易構築定義ファイル	論理 CTM（component-transaction-monitor）	ctm.RegOption
(24)	簡易構築定義ファイル	論理 CTM（component-transaction-monitor）	ctm.EjbPort
(25)	簡易構築定義ファイル	論理 CTM（component-transaction-monitor）	ctm.port
(26)	簡易構築定義ファイル	論理 CTM ドメインマネジャ（ctm-domain-manager）	cdm.port
(27)	config.properties	CJMSP ブローカー	imq.portmapper.port キー
(28)	config.properties	CJMSP ブローカー	imq.jms.tcp.port キー
(29)	config.properties	CJMSP ブローカー	imq.admin.tcp.port キー
(30)	mserver.properties	Management Server	ejbserver.naming.port キー

（凡例）－：該当しない。

注※1　設定ファイルが簡易構築定義ファイルの場合は，<configuration>タグ内の<param-name>の指定値を指します。

注※2　RMSHPort は，リソースアダプタ Reliable Messaging のプロパティ定義で指定するコンフィグレーションプロパティです。RMSHPort については，マニュアル「Reliable Messaging」の「6.　コンフィグレーションプロパティ」を参照してください。

注※3　scd_port および trn_port は，リソースアダプタ TP1 インバウンドアダプタのプロパティ定義で指定するコンフィグレーションプロパティです。scd_port および trn_port については，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(コンテナ共通機能)」の「4.12.2　リソースアダプタの設定」を参照してください。

注※4　サーバ通信エージェントの詳細については，サーバ通信エージェントのドキュメントを参照してください。

参考

運用管理ポータルまたはファイル編集によってアプリケーションサーバを構築している場合の TCP/UDP のポートの設定個所を次に示します。

表 3-5　運用管理ポータルまたはファイル編集によってアプリケーションサーバを構築している場合の TCP/UDP のポートの設定個所

項番	運用管理ポータルで構築している場合の設定個所	ファイル編集で構築している場合の設定個所
(1)	• J2EE サーバの場合 ［EJB コンテナの設定］画面の「オプション」の「通信ポート番号」	usrconf.properties の vbroker.se.iiop_tp.scm.iiop_tp.listener.port キー
(2)	［J2EE サーバの基本設定］画面の「コンテナの設定」の「管理用サーバのポート番号」	usrconf.properties の ejbserver.http.port キー

項番	運用管理ポータルで構築している場合の設定個所	ファイル編集で構築している場合の設定個所
(3)	［トランザクションの設定］画面の「トランザクションに関する設定」の「JTA リカバリの固定ポート番号」	usrconf.properties の ejbserver.distributedtx.recovery.port キー
(4)	• J2EE サーバの場合 ［ネーミングの設定］画面の「インプロセス選択時の設定」の「ポート番号」	usrconf.properties の ejbserver.naming.port キー
(5)	［Web コンテナの設定］画面の「NIO HTTP」の「ポート番号」	usrconf.properties の ebserver.connector.nio_http.port キー
(6)	• J2EE サーバの場合 ［通信の設定］画面の「RMI レジストリの設定」の「ポート番号」	usrconf.properties の ejbserver.rmi.naming.port キー
(7)	Connector 属性ファイルの<config-property>タグに指定する RMSHPort※1	Connector 属性ファイルの<config-property>タグに指定する RMSHPort※1
(8)	• J2EE サーバの場合 ［通信の設定］画面の「RMI レジストリの設定」の「通信ポート番号」	usrconf.properties の ejbserver.rmi.remote.listener.port キー
(9)	TP1 インバウンドアダプタの Connector 属性ファイルのリソースアダプタの<config-property>タグに指定する scd_port	TP1 インバウンドアダプタの Connector 属性ファイルのリソースアダプタの<config-property>タグに指定する scd_port
(10)	TP1 インバウンドアダプタの Connector 属性ファイルのリソースアダプタの<config-property>タグに指定する trn_port	TP1 インバウンドアダプタの Connector 属性ファイルのリソースアダプタの<config-property>タグに指定する trn_port
(11)	• J2EE サーバの場合 ［J2EE コンテナの設定］画面の「運用監視エージェントの設定」の「ポート番号」	mngagent.<実サーバ名>.properties の mngagent.connector.port キー
(12)	［スマートエージェントの設定］画面の「スマートエージェントに関する設定」の「監視ポート番号」	環境変数 OSAGENT_PORT
(13)	• CORBA ネーミングサービスをインプロセスで起動する場合 ［J2EE サーバの基本設定］画面の「利用するネーミングサービスの設定」の「インプロセス用のポート番号」 • CORBA ネーミングサービスをアウトプロセスで起動する場合 ［ホスト内のサーバの設定］画面の「ネーミングサービスの設定」の「ネーミングサービスのポート番号」	• CORBA ネーミングサービスをインプロセスまたはアウトプロセスで自動起動する場合 usrconf.properties の ejbserver.naming.port キー • CORBA ネーミングサービスを手動起動する場合 nameserv コマンドのコマンド引数に「-Dvbroker.se.iiop_tp.scm.iiop_tp.listener.port=<ポート番号>」を指定。
(14)	adminagent.properties の adminagent.adapter.port キー	adminagent.properties の adminagent.adapter.port キー

項番	運用管理ポータルで構築している場合の設定個所	ファイル編集で構築している場合の設定個所
(15)	sinaviagent.properties の sinaviagent.port キー※2	sinaviagent.properties の sinaviagent.port キー
(16)	［ネットワークの設定］画面の「Management Server 接続 HTTP ポート番号」	mserver.properties の webserver.connector.http.port キー
(17)	［ネットワークの設定］画面の「Management Server 終了要求受信ポート番号」	mserver.properties の webserver.shutdown.port キー
(18)	mserver.properties の com.cosminexus.mngsvr.management.port キー	mserver.properties の com.cosminexus.mngsvr.management.port キー
(19)	mserver.properties の com.cosminexus.mngsvr.management.listen.port キー	mserver.properties の com.cosminexus.mngsvr.management.listen.port キー
(20)	［ホスト内のサーバの設定］画面の「J2EE サーバの設定」の「ポート番号」の「http」	httpsd.conf の Listen ディレクティブまたは Port ディレクティブ
(21)	［ホスト内のサーバの設定］画面の「J2EE サーバの設定」の「ポート番号」の「https」	httpsd.conf の Listen ディレクティブまたは Port ディレクティブ
(22)	usrconf.properties（サーバ管理コマンド用システムプロパティファイル）の vbroker.se.iiop_tp.scm.iiop_tp.listener.port キー	usrconf.properties（サーバ管理コマンド用システムプロパティファイル）の vbroker.se.iiop_tp.scm.iiop_tp.listener.port キー
(23)	［レギュレータの設定］画面の「CTM レギュレータの設定」の「設定ファイル」	ctmregltd コマンドまたは ctmstart コマンドの引数-CTMEjbPort
(24)	［スケジューリングの設定］画面の「詳細設定」の「EJB リクエスト受信ポート番号」	ctmstart コマンドの引数-CTMEjbPort
(25)	［CTM の基本設定］画面の「基本設定」の「ポート番号」	ctmstart コマンドの引数-CTMPort
(26)	［CTM ドメインマネジャの基本設定］画面の「ポート番号」	ctmdmstart コマンドの引数-CTMPort
(27)	config.properties の imq.portmapper.port キー	config.properties の imq.portmapper.port キー
(28)	config.properties の imq.jms.tcp.port キー	config.properties の imq.jms.tcp.port キー
(29)	config.properties の imq.admin.tcp.port キー	config.properties の imq.admin.tcp.port キー
(30)	vmx.properties の vmx.vcenterserver.agent.port キー	vmx.properties の vmx.vcenterserver.agent.port キー
(31)	mserver.properties の ejbserver.naming.port キー	mserver.properties の ejbserver.naming.port キー

注※1　RMSHPort は，リソースアダプタ Reliable Messaging のプロパティ定義で指定するコンフィグレーションプロパティです。RMSHPort については，マニュアル「Reliable Messaging」の「6.　コンフィグレーションプロパティ」を参照してください。

注※2　サーバ通信エージェントの詳細については，サーバ通信エージェントのドキュメントを参照してください。

注意事項

サーバの待ち受けポートの注意事項（UNIX の場合）

UNIX の場合，次の条件がすべて重なるときは，待ち受けをしていない TCP ポートに対して，接続に成功してしまうことがあります。

- 待ち受けをしていないポートに対して接続試行を実行する
- 接続対象が自ホストであり，かつ一時ポート番号の範囲（OS が動的に割り当てるポートの範囲）である

この現象が発生した場合，想定したプロセスとの通信ができないで，タイムアウトなどが発生します。この現象を回避するためには，サーバの待ち受けポートに一時ポート番号の範囲以外の値を指定してください。一時ポート番号の範囲は，次のファイルで確認できます。

AIX の場合

最小値（32768）：no -o tcp_ephemeral_low

最大値（65535）：no -o tcp_ephemeral_high

Linux の場合

/proc/sys/net/ipv4/ip_local_port_range

なお，サーバの待ち受けポートの設定方法は，各 OS のマニュアルを参照してください。

4 システム構成の検討（バッチアプリケーション実行基盤）

この章では，バッチアプリケーション実行基盤を構築する場合のシステム構成の検討について説明します。システムを設計する流れに沿って，標準的なシステム構成のパターンを示します。また，意識する必要があるコンポーネント，プロセスおよび処理の流れについて説明します。

J2EE アプリケーション実行基盤のシステム構成を検討する場合は，「3.　システム構成の検討（J2EE アプリケーション実行基盤）」を参照してください。

4.1　システム構成を検討するときに考慮すること

この節では，バッチアプリケーション実行基盤としてアプリケーションサーバを使用する場合に，システム構成の検討で考慮することについて説明します。

4.1.1　システムの目的と構成

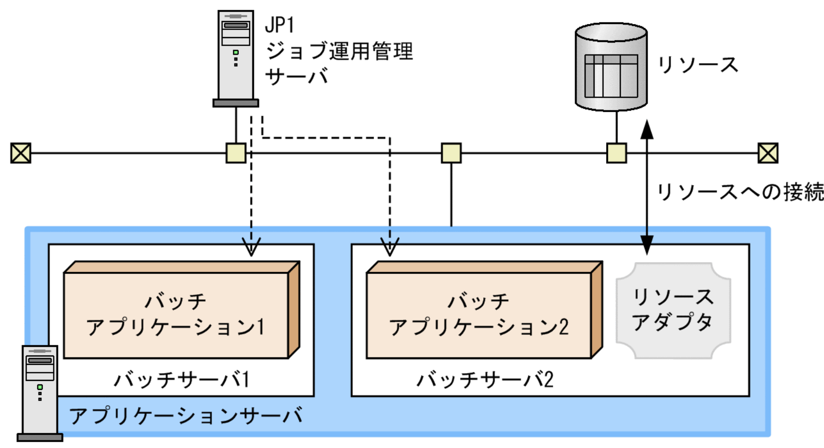
バッチアプリケーション実行基盤は，Java アプリケーションとして実装されたバッチアプリケーションを実行するための環境です。バッチアプリケーションは，JSP，Servlet および Enterprise Bean などの J2EE アプリケーションを利用しない Java アプリケーションとして実装します。なお，バッチアプリケーションから，J2EE サーバ上の Enterprise Bean を呼び出すことはできます。

バッチアプリケーションは，バッチサーバ上で動作します。バッチアプリケーションがリソースと接続する場合に使用するリソースアダプタも，バッチサーバ上で動作します。

バッチアプリケーションは，バッチ実行コマンドによって実行されます。バッチ実行コマンドは，ユーザが直接実行するほか，JP1/AJS のジョブとして自動実行できます。

バッチアプリケーションを実行するシステムの構成例を次の図に示します。この図では，アプリケーションサーバ上で，バッチアプリケーションごとに二つのバッチサーバが動作しています。また，バッチアプリケーションは，JP1/AJS を使用して実行しています。

図 4-1　バッチアプリケーションを実行するシステムの構成例



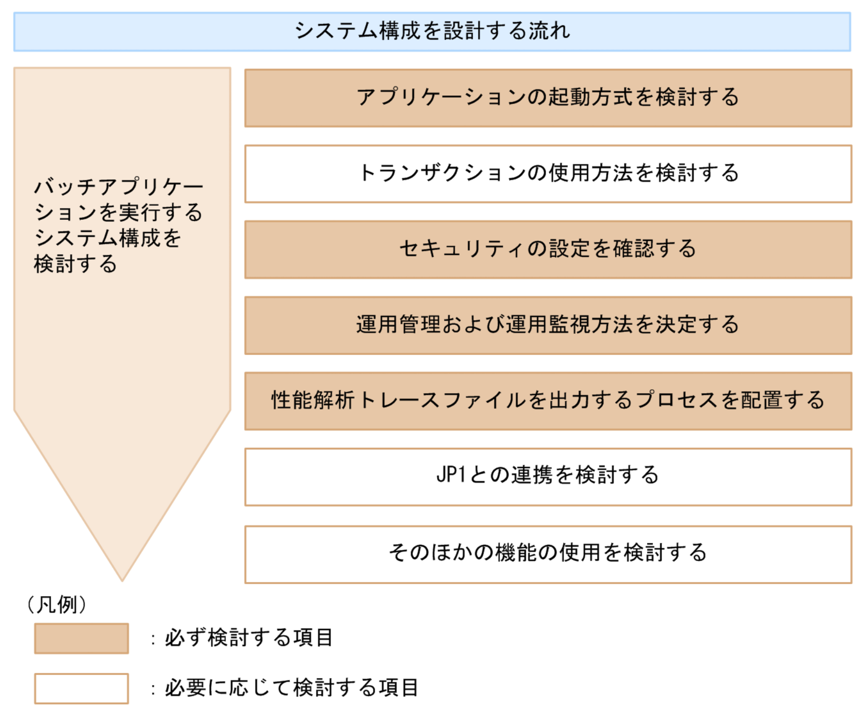
（凡例）
---> : バッチジョブの実行

このほか，バッチアプリケーションから J2EE サーバ上の Enterprise Bean に接続する場合は，オンライン処理を実行するバックシステムに接続することもできます。バックシステムについては，「3.1.1　システムの目的と構成」を参照してください。

4.1.2　システム構成の設計手順

システム構成は，次の流れで設計します。

図 4-2　システム構成を設計する流れ（バッチアプリケーション実行基盤の場合）



(1)　アプリケーションの起動方式を検討する

バッチアプリケーションの起動方式を検討します。次の 2 種類から選択します。

- **バッチサーバ上で起動する**
  バッチサーバ上で Java アプリケーションを起動する方式です。常駐型の JavaVM プロセスであるバッチサーバを使用することで，JavaVM の起動に掛かるオーバーヘッドを削減できます。また，CTM を利用すると，バッチサーバ上で動作するバッチアプリケーションの実行をスケジューリングできます。
  また，バッチサーバ上で起動する場合，リソースアダプタとして DB Connector が使用できます。
- **cjclstartap コマンドを使用して個別に起動する**
  java コマンドと同じように Java アプリケーションを起動する方式です。この方式の場合，バッチアプリケーションを実行するたびに，JavaVM の起動が必要です。

起動方式としてバッチサーバ上で起動することを選択した場合は，次の検討項目に進みます。なお，cjclstartap コマンドを使用することを選択した場合は，以降の検討項目は該当しません。

(2)　トランザクションの使用方法を検討する

リソースと接続する場合は，トランザクションの使用方法を検討します。次の 2 種類から選択します。

- DB Connector を使用する

  アプリケーションサーバが提供するリソースアダプタである DB Connector を使用する方法です。DB Connector の機能として，コネクションプーリングやステートメントプーリングが使用できます。また，フルガーベージコレクションの発生を制御する機能も使用できます。なお，バッチサーバで管理できるトランザクションは，ローカルトランザクションです。

- JDBC ドライバを直接使用する

  JDBC ドライバが提供する API を使用して，トランザクション管理に必要な処理を実装する方法です。

DB Connector を使用する場合のリソースアダプタとリソースの構成については，「3.3.2　リソースの種類とリソースアダプタ」を参照してください。ただし，バッチサーバの場合，接続できるリソースはデータベースだけです。

(3)　セキュリティの設定を確認する

バッチサーバは，SecurityManager によるセキュリティ保護を無効にして起動します。

(4)　性能解析トレースファイルを出力するプロセスを配置する

性能解析トレースファイルを出力するためのプロセスである PRF デーモン（パフォーマンストレーサ）の配置を確認します。PRF デーモンは，バッチサーバごとに配置します。

(5)　運用管理および運用監視方法を決定する

運用管理および運用監視を実行するためのプロセスである Management Server の配置を確認します。バッチアプリケーション実行基盤では，バッチサーバと同じマシンに配置します（ホスト単位管理モデルを使用します）。

(6)　JP1 との連携を検討する

バッチアプリケーションの実行を JP1/AJS のジョブとして自動投入する場合は，JP1 との連携方法を検討します。

(7)　そのほかの機能の使用を検討する

使用する機能に応じて，次の構成を検討してください。

- サーバ間連携をする構成

  バッチアプリケーションから J2EE サーバ上の Enterprise Bean を呼び出す場合に検討します。「3.5 サーバ間での連携を検討する」を参照してください。

- クラスタソフトウェアを使用した障害時の系切り替えをする構成

  バッチサーバを系切り替えの対象にする場合に検討します。

  次の説明を参照してください。

- 「3.11.1　アプリケーションサーバの実行系と待機系を 1:1 にする構成（トランザクションサービスを使用しない場合）」
- 「3.11.4　アプリケーションサーバの実行系と待機系を相互スタンバイにする構成」

参照先の記述のうち，「J2EE サーバ」は「バッチサーバ」に読み替えてください。

4.1.3　バッチアプリケーションを実行するシステムで使用する TCP/UDP のポートについての注意事項

バッチアプリケーションを実行するシステムで使用する TCP/UDP のポートについて，プロセスごとに説明します。バッチアプリケーションを実行するシステムで使用する TCP/UDP のポートを次の表に示します。

表 4-1　バッチアプリケーションを実行するシステムのプロセスが使用する TCP/UDP のポート

項番※1	プロセス	説明
(1)	バッチサーバ	EJB コンテナのリクエスト受付ポート。
(2)		管理用通信ポート。
(3)		Web サーバ（リダイレクタ）からのリクエスト受付ポート。
(5)		インプロセスで起動するネーミングサービスのリクエスト受付ポート。
(7)		RMI レジストリのリクエスト受付ポート。
(9)		稼働情報取得時のリクエスト受付ポート。
(13)	スマートエージェント※2	スマートエージェントの通信用ポート環境変数。 UDP による双方向通信に必要です。
(25)	CTM レギュレータ※2	CTM レギュレータが EJB クライアントからのリクエストを受け付けるポートの基底値。基底値＋プロセス数だけ使用します。 CTM 使用時に必要です。
(26)	CTM デーモン※2	CTM デーモンが EJB クライアントからのリクエストを受け付けるポート。 CTM 使用時に必要です。
(27)		CTM デーモンがほかのデーモンや J2EE サーバなどと通信するポート。 CTM 使用時に必要です。
(28)	CTM ドメインマネジャ※2	CTM ドメインマネジャがほかの CTM ドメインマネジャと通信するポート。 CTM 使用時に，TCP および UDP 通信（ブロードキャスト）をするために必要です。

注　スマートエージェントを使用する指定をした場合，スマートエージェントと通信するために，この表で示した以外の複数のポートが使用されます。スマートエージェントの詳細については，マニュアル「Borland(R) Enterprise Server VisiBroker(R) プログラマーズリファレンス」を参照してください。

注※1　「3.15　アプリケーションサーバのプロセスが使用する TCP/UDP のポート番号」の「表 3-3　アプリケーションサーバが使用する TCP/UDP のポート番号」の項番と対応しています。

注※2　バッチアプリケーションのスケジューリング機能を使用する場合に，必要なプロセスです。

次の場合は，使用するポートが重複しないように設定してください。

- 一つのマシンで J2EE サーバとバッチサーバを同時に使用する場合
- 一つのマシンで複数のバッチサーバを同時に使用する場合

各プロセスが使用する TCP/UDP のポートの詳細については，「3.15　アプリケーションサーバのプロセスが使用する TCP/UDP のポート番号」を参照してください。

注意事項

サーバの待ち受けポートの注意事項（UNIX の場合）

UNIX の場合，次の条件がすべて重なるときは，待ち受けをしていない TCP ポートに対して，接続に成功してしまうことがあります。

- 待ち受けをしていないポートに対して接続試行を実行する
- 接続対象が自ホストであり，かつ一時ポート番号の範囲（OS が動的に割り当てるポートの範囲）である

この現象が発生した場合，想定したプロセスとの通信ができないで，タイムアウトなどが発生します。この現象を回避するためには，サーバの待ち受けポートに一時ポート番号の範囲以外の値を指定してください。一時ポート番号の範囲は，次のファイルで確認できます。

AIX の場合

最小値（32768）：no -o tcp_ephemeral_low

最大値（65535）：no -o tcp_ephemeral_high

Linux の場合

/proc/sys/net/ipv4/ip_local_port_range

なお，サーバの待ち受けポートの設定方法は，各 OS のマニュアルを参照してください。

4.2 バッチサーバを使用する場合のシステム構成

この節では，バッチサーバを使用する場合のシステム構成について説明します。バッチサーバを使用するシステムは，バッチアプリケーションのスケジューリング機能を使用するかどうかによって，構成が異なります。

4.2.1 バッチアプリケーションのスケジューリング機能を使用しないシステムのシステム構成

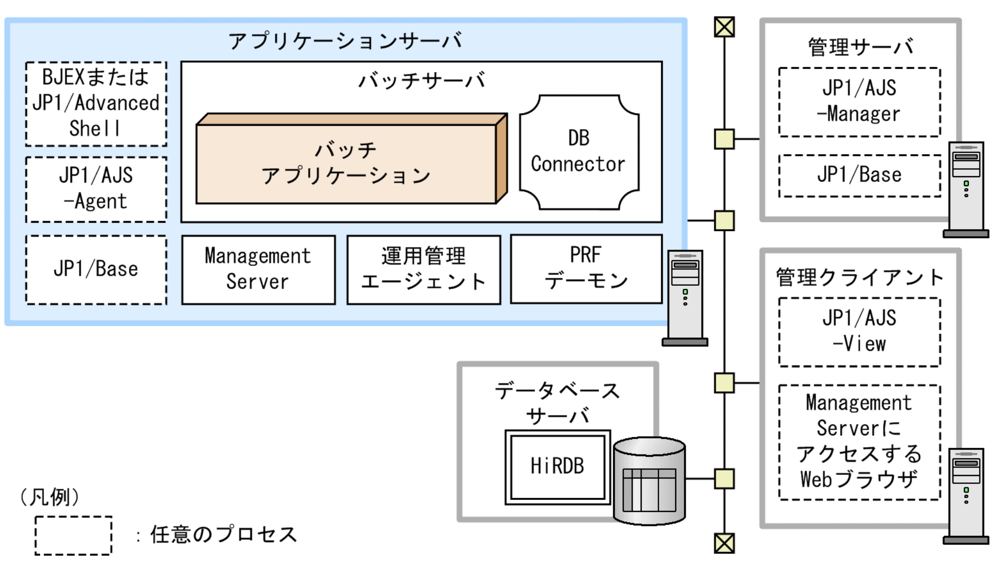
バッチサーバを使用する場合に，バッチアプリケーションのスケジューリング機能を使用しないときのシステムの構成について説明します。

(1) システム構成の特徴

バッチサーバを使用する構成の一つで，CTM を使用しないシステム構成です。この場合，アプリケーションサーバにはバッチサーバを配置します。バッチサーバは，Smart Composer 機能の Web システム（j2ee-tier）として構築，運用します。バッチアプリケーションの実行には，バッチ実行コマンドを使用します。バッチ実行コマンドは，JP1/AJS にジョブとして定義して実行できます。

バッチサーバを配置するシステム構成の例を次の図に示します。この例では，バッチアプリケーションは JP1/AJS から実行します。また，バッチアプリケーションからリソースアダプタを使用して HiRDB にアクセスします。

図 4-3　バッチサーバを配置するシステム構成の例



これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

特徴

リソースとの接続に DB Connector が使用できます。また，Batch Job Execution Server または JP1/Advanced Shell との連携もできます。

処理の流れ

バッチアプリケーションの実行は，JP1/AJS のジョブとして定義しておきます。JP1/AJS のジョブとして実行されたバッチアプリケーションは，バッチサーバ上の DB Connector を経由して HiRDB にアクセスします。

(2) それぞれのマシンに必要なプロセスとソフトウェア

それぞれのマシンに必要なソフトウェアとプロセスについて説明します。

(a) アプリケーションサーバマシン

アプリケーションサーバマシンには，Application Server をインストールする必要があります。なお，開発環境の場合は，Developer をインストールする必要があります。

さらに，JP1/AJS によってバッチアプリケーションを実行する場合や，Batch Job Execution Server または JP1/Advanced Shell と連携したジョブの制御をする場合は，次の製品もインストールする必要があります。

JP1/AJS と連携する場合に必要な製品

- JP1/Base
- JP1/AJS - Agent
- JP1/AJS - Manager
- JP1/AJS - View

Batch Job Execution Server と連携する場合に必要な製品

- Batch Job Execution Server
- JP1/AJS と連携する場合に必要な製品

JP1/Advanced Shell と連携する場合に必要な製品

- JP1/Advanced Shell
- JP1/AJS と連携する場合に必要な製品（JP1/AJS 経由で JP1/Advanced Shell を実行する場合）

起動するプロセスは次のとおりです。

- バッチサーバ
- PRF デーモン
- Management Server
- 運用管理エージェント

- JP1/Base と JP1/AJS のプロセス（JP1/AJS と連携する場合）
- Batch Job Execution Server のプロセス（Batch Job Execution Server と連携する場合）
- JP1/Advanced Shell のプロセス（JP1/Advanced Shell と連携する場合）

また，データベースと接続する場合は，使用するデータベースと接続するためのソフトウェアが必要です。データベースに接続するために必要な製品については，「3.6.1　ローカルトランザクションを使用する場合の構成」を参照してください。ただし，バッチサーバで使用できるリソースは，次に示すデータベースだけです。

- HiRDB
- Oracle
- SQL Server
- XDM/RD E2

(b)　管理サーバマシンおよび管理クライアントマシン

管理サーバマシンおよび管理クライアントマシンは，JP1/AJS と連携する場合に必要です。インストールする製品および起動するプロセスについては，マニュアル「JP1/Automatic Job Management System 設計ガイド」を参照してください。

(c)　データベースサーバマシン

データベースサーバマシンに必要な製品については，「3.6.1　ローカルトランザクションを使用する場合の構成」のリソースマネジャが動作するマシンの説明を参照してください。ただし，バッチサーバで使用できるリソースは，次に示すデータベースだけです。

- HiRDB
- Oracle
- SQL Server
- XDM/RD E2

4.2.2　バッチアプリケーションのスケジューリング機能を使用するシステムのシステム構成

バッチサーバを使用する場合に，バッチアプリケーションのスケジューリング機能を使用するときのシステムの構成について説明します。

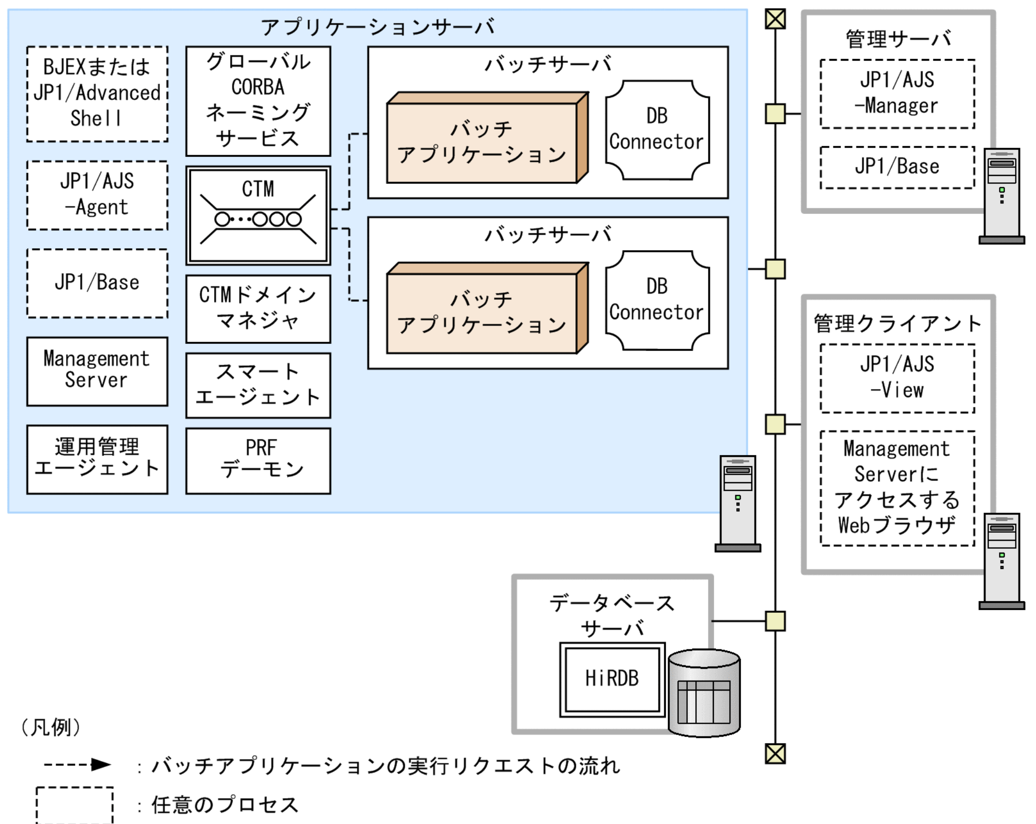
(1)　システム構成の特徴

バッチサーバを使用する構成の一つで，CTM を使用するシステム構成です。この場合，アプリケーションサーバにはバッチサーバと CTM を配置します。バッチサーバと CTM は，Smart Composer 機能の

Webシステム（ctm-tier）として構築，運用します。バッチアプリケーションの実行には，バッチ実行コマンドを使用します。JP1/AJSのジョブ，または直接コマンドから実行されたバッチアプリケーションの実行は，CTMによってスケジューリングされ，バッチサーバに振り分けられます。

CTMを使用するシステム構成の例を次の図に示します。この例では，バッチサーバを2台配置し，CTMを使用してバッチアプリケーションの実行リクエストをスケジューリングします。バッチ実行コマンドはJP1/AJSのジョブから実行します。

図 4-4　CTMを使用するシステム構成の例



これ以外の凡例については，「3.2　システム構成の説明について」を参照してください。

特徴

CTMによってバッチアプリケーションの実行リクエストをスケジューリングすることで，複数のバッチ実行コマンドを同時に実行できます。また，バッチ実行コマンドでバッチサーバを指定する必要がありません。このため，バッチ実行コマンドをJP1/AJSのジョブで定義している場合でも，ジョブの定義を変更することなく，バッチサーバの同時実行数を変更できます。

なお，このシステム構成でも，Batch Job Execution ServerまたはJP1/Advanced Shellと連携できます。

処理の流れ

JP1/AJSのジョブ，または直接バッチ実行コマンドから実行されたバッチアプリケーションは，CTMのスケジュールキューにバッチアプリケーションの実行リクエストとして登録されます。スケジュールキュー内のバッチアプリケーションの実行リクエストは，CTMによって適切なバッチサーバに振り分

けられます。なお，振り分け先のバッチサーバがない場合，バッチアプリケーションの実行リクエストはスケジュールキュー内に滞留（待機）します。

(2) それぞれのマシンに必要なプロセスとソフトウェア

それぞれのマシンに必要なソフトウェアとプロセスについて説明します。

(a) アプリケーションサーバマシン

アプリケーションサーバマシンには，Application Server をインストールする必要があります。なお，開発環境の場合は，Developer をインストールする必要があります。

さらに，JP1/AJS によってバッチアプリケーションを実行する場合や，Batch Job Execution Server または JP1/Advanced Shell と連携したジョブの制御をする場合は，次の製品もインストールする必要があります。

JP1/AJS と連携する場合に必要な製品

- JP1/Base
- JP1/AJS - Agent
- JP1/AJS - Manager
- JP1/AJS - View

Batch Job Execution Server と連携する場合に必要な製品

- Batch Job Execution Server
- JP1/AJS と連携する場合に必要な製品

JP1/Advanced Shell と連携する場合に必要な製品

- JP1/Advanced Shell
- JP1/AJS と連携する場合に必要な製品（JP1/AJS 経由で JP1/Advanced Shell を実行する場合）

起動するプロセスは次のとおりです。

- バッチサーバ
- PRF デーモン
- グローバル CORBA ネーミングサービス
- CTM のプロセス群（CTM デーモンおよび CTM レギュレータ）
- CTM ドメインマネジャ
- スマートエージェント
- Management Server
- 運用管理エージェント
- JP1/Base と JP1/AJS のプロセス（JP1/AJS と連携する場合）

- Batch Job Execution Server のプロセス（Batch Job Execution Server と連携する場合）
- JP1/Advanced Shell のプロセス（JP1/Advanced Shell と連携する場合）

また，データベースと接続する場合は，使用するデータベースと接続するためのソフトウェアが必要です。データベースに接続するために必要な製品については，「3.6.1　ローカルトランザクションを使用する場合の構成」を参照してください。ただし，バッチサーバで使用できるリソースは，次に示すデータベースだけです。

- HiRDB
- Oracle
- SQL Server
- XDM/RD E2

(b)　管理サーバマシンおよび管理クライアントマシン

管理サーバマシンおよび管理クライアントマシンは，JP1/AJS と連携する場合に必要です。インストールする製品および起動するプロセスについては，マニュアル「JP1/Automatic Job Management System 設計ガイド」を参照してください。

(c)　データベースサーバマシン

データベースサーバマシンに必要な製品については，「3.6.1　ローカルトランザクションを使用する場合の構成」のリソースマネジャが動作するマシンの説明を参照してください。ただし，バッチサーバで使用できるリソースは，次に示すデータベースだけです。

- HiRDB
- Oracle
- SQL Server
- XDM/RD E2

5 使用するリソースの見積もり（J2EE アプリケーション実行基盤）

この章では，J2EE アプリケーションを実行するシステムで使用するリソース，および仮想メモリ使用量の見積もり方法について説明します。システムを動作させるために必要なディスクおよびメモリの容量を算出するときの参考にしてください。

ファイルディスクリプタ数を設定する場合は，同一のホスト上で動作するプロセスの最大数を指定する必要があります。

バッチアプリケーション実行基盤の使用リソースおよび仮想メモリ所要量の見積もりについては，「6. 使用するリソースの見積もり（バッチアプリケーション実行基盤）」を参照してください。

5.1　システム構成ごとに使用するリソース

システムが動作するためには，OS やデータベースなどのホスティング環境の設定が必要な場合があります。システムが必要とするリソースは，システムの構成ごとに異なるため，ここではシステム構成ごとに使用するリソースと，リソースの所要量の見積もりについて説明します。システム構成ごとに使用するリソースと，リソースの見積もりの参照先を次の表に示します。

表 5-1　システム構成ごとに使用するリソースと見積もりの参照先

システム構成ごとに使用するリソース	参照先
Web サーバと J2EE サーバを同じマシンに配置する場合の使用リソース	5.1.1
Web サーバと J2EE サーバを別のマシンに配置する場合の使用リソース	5.1.2
データベースの使用リソース	5.1.3
運用管理サーバの使用リソース	5.1.4
CTM を使用する場合の使用リソース	5.1.5

また，プロセスごとに使用するリソースの見積もりについては，「5.2　プロセスごとに使用するリソース」を参照してください。

プロセスごとに使用するメモリの見積もりについては，「5.3　プロセスごとに使用するメモリの見積もり」を参照してください。ディスク占有量については，アプリケーションサーバのリリースノートを参照してください。

ポイント

Windows システムの場合は，この節で説明する内容のうち，「5.1.3　データベースの使用リソース」だけを参照してください。

システムで利用できるプロセス数，共用メモリ，ファイルディスクリプタ数，および Windows システムやプロセスで利用できるスレッド数に，特に制限はありません。

5.1.1　Web サーバと J2EE サーバを同じマシンに配置する場合の使用リソース

Web サーバと J2EE サーバを同じマシンに配置する場合の使用リソースの見積もりについて，OS ごとに説明します。

なお，使用リソースの見積もりの各表にある，「オプション設定ファイル例」については，使用している OS のバージョン，およびカーネルのバージョンごとに異なります。使用している OS のマニュアルを参照して，表中の見積もり式を基に見積もった値を設定してください。使用している OS で該当するカーネルパラメタが設定できない場合には，設定は不要です。

(1)　AIX の場合

AIX の場合の，使用リソースの見積もりについて，次の表に示します。

表 5-2　使用リソースの見積もり（AIX の場合）

システムリソース	パラメタ	所要量	オプション設定ファイル例
共用メモリ	－	PrfTraceBufferSize※1×1,024 ＋ 18,496 ＋リクエスト最大同時処理数※2×14×1,024	－
プロセス数	－	リクエスト最大同時処理数※2×2 ＋ 5	－
スレッド数	－	リクエスト最大同時処理数※2×2 ＋ 41 ＋ J2EE サーバのスレッド数※3	－
ファイルディスクリプタ数	nofiles	J2EE サーバのファイルディスクリプタ数※3 ＋ 76 ＋リクエスト最大同時処理数※2×4	/etc/security/limits

（凡例）－：該当しません。

注※1

パフォーマンストレーサのバッファメモリサイズを 512 キロバイト～102,400 キロバイトの範囲で指定します。PrfTraceBufferSize については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「4.12　論理パフォーマンストレーサで指定できるパラメタ」を参照してください。

注※2

Web サーバで同時に処理できるリクエストの最大数を指します。

注※3

J2EE サーバのスレッド数とファイルディスクリプタ数については，「5.2.1　J2EE サーバが使用するリソースの見積もり」を参照して算出してください。

(2)　Linux の場合

Linux の場合の，使用リソースの見積もりについて，次の表に示します。

表 5-3　使用リソースの見積もり（Linux の場合）

システムリソース	パラメタ	所要量	オプション設定ファイル例
共用メモリ	SHMMAX	PrfTraceBufferSize※1×1,024 ＋ 18,496 ＋リクエスト最大同時処理数※2×14×1,024	/proc/sys/kernel/shmmax
プロセス数	threads-max※3	リクエスト最大同時処理数※2×2 ＋ 5	/proc/sys/kernel/threads-max
スレッド数	threads-max※3	リクエスト最大同時処理数※2×2 ＋ 41 ＋ J2EE サーバのスレッド数※4	－
ファイルディスクリプタ数	fs.file-max	J2EE サーバのファイルディスクリプタ数※4 ＋ 76 ＋リクエスト最大同時処理数※2×4	/proc/sys/fs/file-max

（凡例）－：該当しません。

注※1

パフォーマンストレーサのバッファメモリサイズを 512 キロバイト～102,400 キロバイトの範囲で指定します。PrfTraceBufferSize については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「4.12　論理パフォーマンストレーサで指定できるパラメタ」を参照してください。

注※2

Web サーバで同時に処理できるリクエストの最大数を指します。

注※3

threads-max パラメタには，プロセス数とスレッド数の合計を指定してください。

注※4

J2EE サーバのスレッド数とファイルディスクリプタ数については，「5.2.1　J2EE サーバが使用するリソースの見積もり」を参照して算出してください。

5.1.2　Web サーバと J2EE サーバを別のマシンに配置する場合の使用リソース

Web サーバと J2EE サーバを別のマシンに配置する場合の使用リソースの見積もりについて，OS ごとに説明します。Web サーバと J2EE サーバを別マシンに配置する場合は，Web サーバマシンとアプリケーションサーバマシンのそれぞれで使用するリソースを見積もります。

なお，使用リソースの見積もりの各表にある，「オプション設定ファイル例」については，使用している OS のバージョン，およびカーネルのバージョンごとに異なります。使用している OS のマニュアルを参照して，表中の見積もり式を基に見積もった値を設定してください。使用している OS で該当するカーネルパラメタが設定できない場合には，設定は不要です。

(1)　AIX の場合

AIX の場合の，Web サーバマシンおよびアプリケーションサーバマシンの使用リソースの見積もりについて説明します。

(a)　Web サーバマシンの使用リソースの見積もり

Web サーバマシンの使用リソースの見積もりについて，次の表に示します。

表 5-4　Web サーバマシンの使用リソースの見積もり（AIX の場合）

システムリソース	パラメタ	所要量	オプション設定ファイル例
共用メモリ	－	PrfTraceBufferSize※1×1,024 + 18,496 +リクエスト最大同時処理数※2×14×1,024	－
プロセス数	－	リクエスト最大同時処理数※2×2 + 4	－
スレッド数	－	リクエスト最大同時処理数※2×2 + 35	－

システムリソース	パラメタ	所要量	オプション設定ファイル例
ファイルディスクリプタ数	nofiles	リクエスト最大同時処理数※2×4 + 75	/etc/security/limits

（凡例）－：該当しません。

注※1

パフォーマンストレーサのバッファメモリサイズを 512 キロバイト～102,400 キロバイトの範囲で指定します。
PrfTraceBufferSize については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「4.12　論理パフォーマンストレーサで指定できるパラメタ」を参照してください。

注※2

Web サーバで同時に処理できるリクエストの最大数を指します。

(b)　アプリケーションサーバマシンの使用リソースの見積もり

アプリケーションサーバマシンの使用リソースの見積もりについて，次の表に示します。

表 5-5　アプリケーションサーバマシンの使用リソースの見積もり（AIX の場合）

システムリソース	パラメタ	所要量	オプション設定ファイル例
共用メモリ	－	PrfTraceBufferSize※1×1,024 + 18,496	－
プロセス数	－	4	－
スレッド数	－	J2EE サーバのスレッド数※2 + 34	－
ファイルディスクリプタ数	nofiles	J2EE サーバのファイルディスクリプタ数※2 + 43	/etc/security/limits

（凡例）－：該当しません。

注※1

パフォーマンストレーサのバッファメモリサイズを 512 キロバイト～102,400 キロバイトの範囲で指定します。
PrfTraceBufferSize については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「4.12　論理パフォーマンストレーサで指定できるパラメタ」を参照してください。

注※2

J2EE サーバのスレッド数とファイルディスクリプタ数については，「5.2.1　J2EE サーバが使用するリソースの見積もり」を参照して算出してください。

(2)　Linux の場合

Linux の場合の，Web サーバマシンおよびアプリケーションサーバマシンの使用リソースの見積もりについて説明します。

(a)　Web サーバマシンの使用リソースの見積もり

Web サーバマシンの使用リソースの見積もりについて，次の表に示します。

表 5-6　Web サーバマシンの使用リソースの見積もり（Linux の場合）

システムリソース	パラメタ	所要量	オプション設定ファイル例
共用メモリ	SHMMAX	PrfTraceBufferSize※1×1,024 + 18,496 +リクエスト最大同時処理数※2×14×1,024	/proc/sys/kernel/shmmax
プロセス数	threads-max※3	リクエスト最大同時処理数※2×2 + 4	/proc/sys/kernel/threads-max
スレッド数	threads-max※3	リクエスト最大同時処理数※2×2 + 35	－
ファイルディスクリプタ数	fs.file-max	リクエスト最大同時処理数※2×4 + 75	/proc/sys/fs/file-max

（凡例）－：該当しません。

注※1

パフォーマンストレーサのバッファメモリサイズを 512 キロバイト～102,400 キロバイトの範囲で指定します。
PrfTraceBufferSize については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「4.12　論理パフォーマンストレーサで指定できるパラメタ」を参照してください。

注※2

Web サーバで同時に処理できるリクエストの最大数を指します。

注※3

threads-max パラメタには，プロセス数とスレッド数の合計を指定してください。

(b)　アプリケーションサーバマシンの使用リソースの見積もり

アプリケーションサーバマシンの使用リソースの見積もりについて，次の表に示します。

表 5-7　アプリケーションサーバマシンの使用リソースの見積もり（Linux の場合）

システムリソース	パラメタ	所要量	オプション設定ファイル例
共用メモリ	SHMMAX	PrfTraceBufferSize※1×1,024 + 18,496	/proc/sys/kernel/shmmax
プロセス数	threads-max※2	4	/proc/sys/kernel/threads-max
スレッド数	threads-max※2	J2EE サーバのスレッド数※3 + 34	－
ファイルディスクリプタ数	fs.file-max	J2EE サーバのファイルディスクリプタ数※3 + 43	/proc/sys/fs/file-max

（凡例）－：該当しません。

注※1

パフォーマンストレーサのバッファメモリサイズを 512 キロバイト～102,400 キロバイトの範囲で指定します。
PrfTraceBufferSize については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「4.12　論理パフォーマンストレーサで指定できるパラメタ」を参照してください。

注※2

threads-max パラメタには，プロセス数とスレッド数の合計を指定してください。

注※3

J2EE サーバのスレッド数とファイルディスクリプタ数については，「5.2.1　J2EE サーバが使用するリソースの見積もり」を参照して算出してください。

5.1.3　データベースの使用リソース

DBMS の使用リソースの見積もりについて説明します。

プロセスごとに使用するメモリの見積もりについては，「5.3　プロセスごとに使用するメモリの見積もり」を参照してください。また，ディスク占有量については，Application Server または Developer のリリースノートを参照してください。

DBMS の使用リソースの見積もりについて，次の表に示します。

表 5-8　DBMS の使用リソースの見積もり

DBMS	使用リソース	所要量
HiRDB	最大同時接続数 (pd_max_users)	$\sum_{i=1}^{n^{※1}}$（リソースアダプタ i のコネクションプールの最大値※2 ×2※3＋1※4）＋ α※5
Oracle	最大同時接続数 (SESSIONS)	$\sum_{i=1}^{n^{※1}}$（リソースアダプタ i のコネクションプールの最大値※2 ＋1※4）＋ α※5

注※1

n は，システム内の J2EE サーバにデプロイするリソースアダプタの総和です。

注※2

Connector 属性ファイルの MaxPoolSize パラメタの値を指定します。

注※3

次に示す条件に当てはまる場合に，×2 を行ってください。

1. トランザクションサポートレベルに XATransaction を使用する。
2. アプリケーションサーバが管理するトランザクション内でコネクション※を使ってデータベースにアクセスする。
3. 2.のトランザクションが決着する前に，トランザクション外でコネクション※を使ってデータベースにアクセスする。

注※　このコネクションは 1.の DB Connector から取得したコネクションで，かつ同一コネクションです。

注※4

トランザクションサポートレベルに XATransaction を指定しているリソースアダプタの場合に，＋ 1 を行ってください。

注※5

＋ α の最大値は，使用する DB Connector のコネクションプールの最大値の合計値になります。

この＋ α の値は，一時的にコネクションプールの最大値をオーバーするおそれのあるコネクションを指します。詳細を次に示します。

- コネクションの障害検知機能を使用する場合

  コネクションの障害検知機能を使用する場合，コネクションプールから取り除いた未使用のコネクションは，コネクションプール内のコネクション数としてカウントされません。そのため，コネクションプール内のコネクションとコネクションプールから取り除いた未使用のコネクションの総和が，コネクションプールのコネクション数の最大値を一時的に超えることがあります。

- cjclearpool コマンドを使用する場合

  通常モードの場合，コネクションプールから取り除いた使用中のコネクションは，コネクション数としてカウントされません。そのため，コネクションプール内のコネクションとコネクションプールから取り除いた使用中のコネクションの総和がコネクションプールの最大値を超えることがあります。

5.1.4　運用管理サーバの使用リソース

運用管理サーバの使用リソースの見積もりについて，OS ごとに説明します。

なお，使用リソースの見積もりの各表にある，「オプション設定ファイル例」については，使用している OS のバージョン，およびカーネルのバージョンごとに異なります。使用している OS のマニュアルを参照して，表中の見積もり式を基に見積もった値を設定してください。使用している OS で該当するカーネルパラメタが設定できない場合には，設定は不要です。

(1)　AIX の場合

運用管理サーバの使用リソースの見積もりについて，次の表に示します。

表 5-9　運用管理サーバの使用リソースの見積もり（AIX の場合）

システムリソース	パラメタ	所要量	オプション設定ファイル例
プロセス数	−	5 ＋ A※	−
スレッド数	−	101 ＋論理サーバ数×4 ＋ A※	−
ファイルディスクリプタ数	nofiles	218 ＋論理サーバ数×3	/etc/security/limits

（凡例）−：該当しません。

注※

A は同時に実行する Management アクション数です。Management イベント発生時，Management アクションを実行する場合だけ，次の値を加算します。

- 各 J2EE サーバの Management イベント発行用プロパティファイルの manager.mevent.send.max キーの指定値の合計

(2) Linux の場合

運用管理サーバの使用リソースの見積もりについて，次の表に示します。

表 5-10 運用管理サーバの使用リソースの見積もり（Linux の場合）

システムリソース	パラメタ	所要量	オプション設定ファイル例
プロセス数	threads-max※1	5 + A※2	/proc/sys/kernel/threads-max
スレッド数	threads-max※1	101 ＋論理サーバ数×4 + A※2	−
ファイルディスクリプタ数	fs.files-max	218 ＋論理サーバ数×3	/proc/sys/fs/file-max

（凡例）−：該当しません。

注※1

threads-max パラメタには，プロセス数とスレッド数の合計を指定してください。

注※2

A は同時に実行する Management アクション数です。Management イベント発生時，Management アクションを実行する場合だけ，次の値を加算します。

- 各 J2EE サーバの Management イベント発行用プロパティファイルの manager.mevent.send.max キーの指定値の合計

5.1.5 CTM を使用する場合の使用リソース

CTM を使用する場合の，使用リソースの見積もりについて，OS ごとに説明します。

なお，使用リソースの見積もりの各表にある，「オプション設定ファイル例」については，使用している OS のバージョン，およびカーネルのバージョンごとに異なります。使用している OS のマニュアルを参照して，表中の見積もり式を基に見積もった値を設定してください。使用している OS で該当するカーネルパラメタが設定できない場合には，設定は不要です。

(1) AIX の場合

CTM を使用する場合の，使用リソースの見積もりについて次の表に示します。

表 5-11　CTM 使用時の使用リソースの見積もり（AIX の場合）

システムリソース	パラメタ	所要量	オプション設定ファイル例
共用メモリ	－	PrfTraceBufferSize※1×1,024 + 18,496 + CTM ドメインマネジャの共用メモリ※2 + CTM デーモンの共用メモリ※2	－
プロセス数	－	7 + J2EE サーバ数※3	－
スレッド数	－	72 +(J2EE サーバのスレッド数※4 + 7)×J2EE サーバ数※3 + CTM デーモンで必要とするスレッド数※5	－
ファイルディスクリプタ数	nofiles	88 +(J2EE サーバのファイルディスクリプタ数※4 + 6)×J2EE サーバ数※3 + CTM デーモンで必要とするファイルディスクリプタ数※5	/etc/security/limits

（凡例）－：該当しません。

注※1

パフォーマンストレーサのバッファメモリサイズを 512 キロバイト～102,400 キロバイトの範囲で指定します。PrfTraceBufferSize については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「4.12　論理パフォーマンストレーサで指定できるパラメタ」を参照してください。

注※2

値については，「5.1.5(1)(a)　共用メモリ用ファイルサイズの計算式」を参照して算出してください。

注※3

簡易構築定義ファイルの<j2ee-server-count>タグの指定値を指します。

注※4

J2EE サーバのスレッド数とファイルディスクリプタ数については，「5.2.1　J2EE サーバが使用するリソースの見積もり」を参照して算出してください。

注※5

CTM デーモンで必要とするスレッド数とファイルディスクリプタ数については，「5.1.5(1)(b)　CTM デーモンで必要とするスレッド数とファイルディスクリプタ数の計算式」を参照して算出してください。

(a)　共用メモリ用ファイルサイズの計算式

共用メモリ用ファイルサイズを算出するには，CTM ドメインマネジャの共用メモリおよび CTM デーモンの共用メモリを算出する必要があります。それぞれの計算式について次に示します。

なお，計算式中の可変値には，次の値を使用してください。「ctm.」で始まるパラメタについては，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「4.14　論理 CTM で指定できるパラメタ」を参照してください。

計算式に使用する値

-CTMMaxCTM：64

-CTMQueueCount：ctm.QueueCount

-CTMClientConnectCount：256

-CTMServerConnectCount：ctm.ServerConnectCount

-CTMEntryCount：-CTMClientConnectCount ＋-CTMServerConnectCount

-CTMServerCacheSize：ctm.ServerCacheSize

-CTMQueueRegistCount：ctm.QueueRegistCount

-CTMDispatchParallelCount：ctm.DispatchParallelCount

- **CTM ドメインマネジャの共用メモリ用ファイルサイズの計算式**

  CTM ドメインマネジャの共用メモリ用ファイルサイズの計算式を次に示します。

  共用メモリ用ファイルサイズ（単位：バイト）＝

  1,018,320 ＋（2,362×-CTMMaxCTM 指定値）

- **CTM デーモンの共用メモリ用ファイルサイズの計算式**

  CTM デーモンの場合は，CTM デーモン単位で固定長の共用メモリ用ファイルと可変長の共用メモリ用ファイルを確保する必要があります。それぞれの計算式を次に示します。

  固定長の共用メモリ用ファイルサイズ（単位：バイト）＝

  551,840 ＋（1,208×-CTMQueueCount 指定値）

  可変長の共用メモリ用ファイルサイズ（単位：バイト）＝

  1,027,008

  ＋（928×-CTMClientConnectCount 指定値）

  ＋（256×-CTMServerConnectCount 指定値）

  ＋（512×-CTMEntryCount 指定値）

  ＋（1,024×-CTMServerCacheSize 指定値）

  ＋（512×-CTMQueueCount 指定値）

  ＋（544×-CTMQueueCount 指定値×-CTMQueueRegistCount 指定値）

  ＋（512×-CTMDispatchParallelCount 指定値）

(b) CTM デーモンで必要とするスレッド数とファイルディスクリプタ数の計算式

スレッド数およびファイルディスクリプタ数を算出するには，CTM デーモンで必要とするスレッド数とファイルディスクリプタ数を算出する必要があります。それぞれの計算式について次に示します。

- **CTM デーモンで必要とするスレッド数の計算式**

  最大値 ＝

  （A×4 ＋ B×3 ＋ C×2 ＋ D×E ＋ F ＋ G ＋ 32）／0.8

  （凡例）

  A：-CTMMaxCTM 値（ctmd が属する ctmdmd で指定された値）

  B：-CTMClientConnectCount 値

  C：-CTMServerConnectCount 値

  D：-CTMQueueCount 値

E：-CTMQueueRegistCount 値
F：-CTMDispatchParallelCount 値
G：Create を発行する EJB クライアントの総数

- CTM デーモンで必要とするファイルディスクリプタ数の計算式

最大値 ＝
（A×2 ＋ B×4 ＋ C×2 ＋ D×E ＋ F×EJB のインタフェース数＋ G ＋ 100）／0.8

（凡例）
A：-CTMMaxCTM 値（ctmd が属する ctmdmd で指定された値）
B：-CTMClientConnectCount 値
C：-CTMServerConnectCount 値
D：-CTMQueueCount 値
E：-CTMQueueRegistCount 値
F：-CTMDispatchParallelCount 値
G：Create を発行する EJB クライアントの総数

(2) Linux の場合

CTM を使用する場合の，使用リソースの見積もりについて，次の表に示します。

表 5-12 CTM 使用時の使用リソースの見積もり（Linux の場合）

システムリソース	パラメタ	所要量	オプション設定ファイル例
共用メモリ	SHMMAX	PrfTraceBufferSize※1×1,024 ＋ 18,496 ＋ CTM ドメインマネジャの共用メモリ※2 ＋ CTM デーモンの共用メモリ※2	/proc/sys/kernel/shmmax
プロセス数	threads-max※6 pid_max※6	7 ＋ J2EE サーバ数※3	/proc/sys/kernel/threads-max /proc/sys/kernel/pid_max
スレッド数	threads-max※6 pid_max※6	72 ＋(J2EE サーバのスレッド数※4 ＋ 7)×J2EE サーバ数※3 ＋ CTM デーモンで必要とするスレッド数※5 ＋ CTM レギュレータで必要とするスレッド数※7	/proc/sys/kernel/threads-max /proc/sys/kernel/pid_max
ファイルディスクリプタ数	fs.file-max	88 ＋(J2EE サーバのファイルディスクリプタ数※4 ＋ 6)×J2EE サーバ数※3 ＋ CTM デーモンで必要とするファイルディスクリプタ数※5	/proc/sys/fs/file-max

（凡例）－：該当しません。

注※1

パフォーマンストレーサのバッファメモリサイズを 512 キロバイト～102,400 キロバイトの範囲で指定します。

PrfTraceBufferSize については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「4.12　論理パフォーマンストレーサで指定できるパラメタ」を参照してください。

注※2

値については，「5.1.5(1)(a)　共用メモリ用ファイルサイズの計算式」を参照して算出してください。

注※3

簡易構築定義ファイルの<j2ee-server-count>タグの指定値を指します。

注※4

J2EE サーバのスレッド数とファイルディスクリプタ数については，「5.2.1　J2EE サーバが使用するリソースの見積もり」を参照して算出してください。

注※5

CTM デーモンで必要とするスレッド数とファイルディスクリプタ数については，「5.1.5(1)(b)　CTM デーモンで必要とするスレッド数とファイルディスクリプタ数の計算式」を参照して算出してください。

注※6

threads-max パラメタおよび pid_max パラメタには，プロセス数とスレッド数の合計以上を指定してください。

注※7

CTM レギュレータで必要とするスレッド数については，「12+(EJB クライアントの総数×2)/0.8)」を使用して算出してください。

EJB クライアントの総数の最大値は ctmregltd の-CTMClientConnectCount 値の合計になります。

5.2　プロセスごとに使用するリソース

この節では，アプリケーションサーバの各プロセスで使用するリソースの所要量の見積もりについて説明します。

運用管理サーバの使用するリソースについては，「5.1.4　運用管理サーバの使用リソース」を参照してください。

5.2.1　J2EE サーバが使用するリソースの見積もり

ここでは，J2EE サーバプロセスのスレッド数とファイルディスクリプタ数の見積もりについて説明します。

(1)　スレッド数

スレッド数の計算式を次に示します。(a)と(b)の合計が，J2EE サーバが使用するスレッド数です。

(a)　基本のスレッド数

次の計算式で算出してください。

- J2EE サーバの場合
  最大スレッド数＝76＋A＋B＋C＋D＋E＋F＋G＋H＋I＋J＋K＋L＋M＋N＋O＋P＋Q＋R＋S
- バッチサーバの場合
  最大スレッド数＝73＋D＋E＋F＋G＋H＋I＋O＋P

（凡例）

- A：デプロイ済みの Entity Bean 数の合計
- B：Message-driven Bean を使用する場合，Message-driven Bean の最大インスタンス数（複数の Message-Driven Bean がある場合は合計）※
  注※　Message-Driven Bean 属性ファイルの<pooled-instance><maximum>の値
- C：リモート呼び出しを行う最大 EJB クライアント数×2 ＋各 EJB クライアントの最大同時リクエスト数の合計＋ 1
- D：CORBA ネーミングサービスのスレッド数（=クライアントと CORBA ネーミングサービス間のコネクション数×2 ＋同時受け付けリクエスト数＋初期化時に生成されるスレッド数（vbroker.agent.enableLocator の値が true の場合は 6，false の場合は 4）＋ 1）
  ただし，CORBA ネーミングサービスをインプロセスで起動（usrconf.properties の ejbserver.naming.startupMode キーに inprocess を指定）した場合だけ加算する。
- E：同時に使用する最大のデータベースコネクション数

コネクションプーリング機能を使用している場合は，最大コネクションプール数（Connector 属性ファイルで指定する MaxPoolSize の値。複数のリソースアダプタがある場合は合計）となる。

コネクションプーリング機能を使用していない場合は，最大同時リクエスト数や 1 リクエストで使用するコネクション数から求める（1 リクエストで 1 コネクションを使用する場合，最大同時リクエスト数となる）。

- F：JTA トランザクションを使用する場合，最大同時実行トランザクション数（トランザクションタイムアウトが発生したトランザクション一つにつき，1 スレッドを使用する。1 リクエストが 1 トランザクションの場合，最大同時リクエスト数となる）
- G：最大コネクションプール数※（複数のリソースアダプタがある場合は合計）×2
  注※　Connector 属性ファイルで指定する MaxPoolSize の値
- H：コネクションプーリング機能を使用しているリソースアダプタ数
- I：グローバルトランザクションの決着処理，リカバリ処理で使用するスレッド数（グローバルトランザクションを使用している場合，16 を加算する）
- J：デプロイ済みの Web アプリケーションの数
- K：J2EE アプリケーションの自動リロード機能で使用するスレッド数
  次のどれかの値を指定する。
  - ejbserver.deploy.context.reload_scope=app，かつ，ejbserver.deploy.context.check_interval=1 以上の場合
    展開ディレクトリ形式で開始状態の J2EE アプリケーション数＋展開ディレクトリ形式で開始状態の J2EE アプリケーションに含まれる WAR の数×2
  - ejbserver.deploy.context.reload_scope=web，かつ，ejbserver.deploy.context.check_interval=1 以上の場合
    展開ディレクトリ形式で開始状態の J2EE アプリケーションに含まれる WAR の数×2
  - ejbserver.deploy.context.reload_scope=jsp，かつ，ejbserver.deploy.context.check_interval=1 以上の場合
    展開ディレクトリ形式で開始状態の J2EE アプリケーションに含まれる WAR の数
- L：TP1 インバウンドアダプタのスレッド数（4 + TP1 インバウンドアダプタのプロパティ rpc_max_thread_count に指定したスレッド数＋ TP1 インバウンドアダプタのプロパティ trn_max_thread_count に指定したスレッド数＋ TP1 インバウンドアダプタと連携するデプロイ済みの Message-driven Bean（サービス）の数の合計＋ TP1 インバウンドアダプタのプロパティ MaxTPoolSize に指定した数）
  TP1 インバウンド連携機能を利用する場合だけ加算する。このスレッド数は，Connector 属性ファイルによって制御できる。先頭で加算している 4 は，TP1 インバウンド連携機能の内部で使用するスレッド数。
- M：非同期 Session Bean 呼び出し用のスレッドプールの最大数（cosminexus.xml の<cosminexus-app><ejb-async-props><max-thread-pool-size>の値）の合計
- N：非同期 Session Bean を含む J2EE アプリケーション数の合計×2

- O：リプライ受信専用スレッドを管理するスレッドを起動する設定（vbroker.ce.iiop.ccm.htc.threadStarter=true）の場合，5 を加算する。
- P：タイムアウト発生時のコネクションのクローズを抑止する設定（vbroker.ce.iiop.ccm.htc.readerPerConnection=true）の場合，次の値を加算する。

  （リモート呼び出し先の EJB がある J2EE サーバの数＋ 1）×2

  CTM を使用している場合は，さらに次の値を加算する。

  - **J2EE アプリケーション単位のスケジューリングをしている場合**

    開始している J2EE アプリケーション数＋ 1
  - **Stateless Session Bean 単位のスケジューリングをしている場合**

    スケジューリング対象の Stateless Session Bean の数＋ 1
- Q：NIO HTTP サーバの最大スレッド数（webserver.connector.nio_http.max_threads の値）
- R：Java Batch で使用する最大スレッド数

  Java Batch を使用するアプリケーションがある場合だけ加算する。

  ejbserver.javaee.batch.executorService.<JNDI 名>.maxThreads の値を指定する。複数の JNDI 名を定義している場合は，それらの合計値を指定する。
- S：Concurrency Utilities for Java EE で使用する最大スレッド数

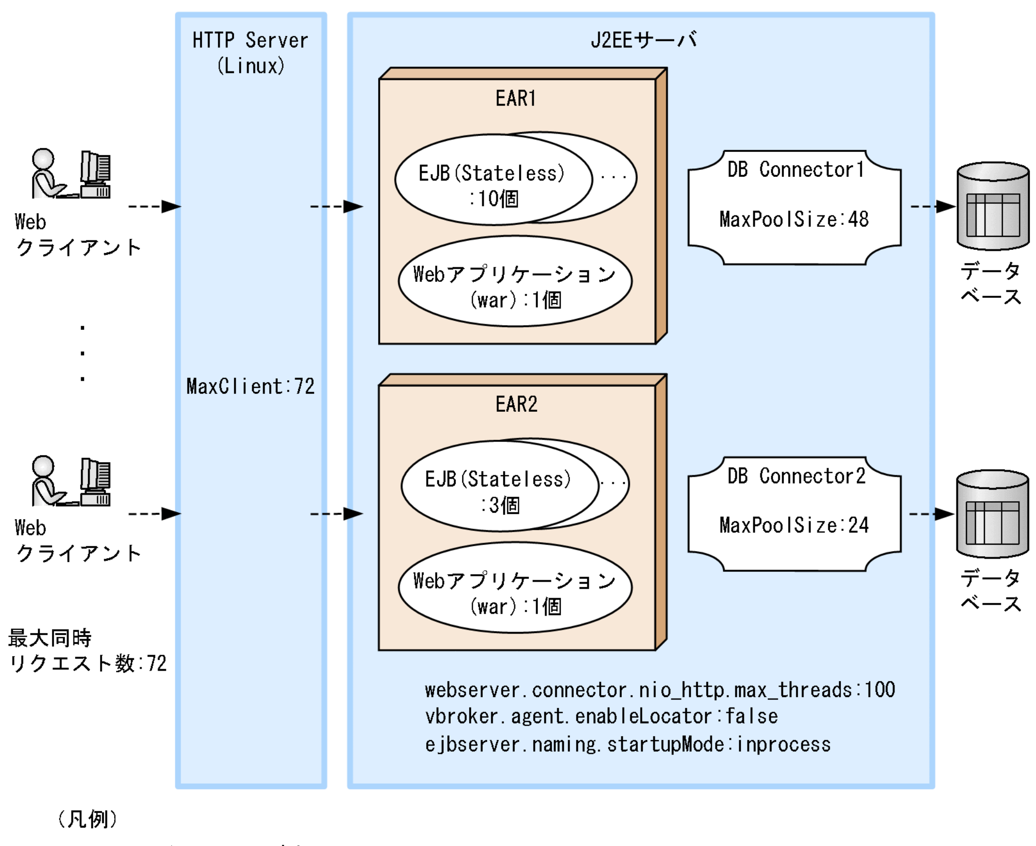
  Concurrency Utilities for Java EE を使用するアプリケーションがある場合だけ加算する。

  次の値の合計値になる。複数の JNDI 名を定義している場合は，それらの合計値を指定する。

  - ManagedExecutorService の最大スレッド数（ejbserver.javaee.concurrent.managedExecutorService.<JNDI 名>.maxPoolSize の値）
  - ManagedScheduledExecutorService で同時に実行される最大タスク数
  - ManagedThreadFactory で同時に生成される最大スレッド数

CORBA ネーミングサービスをインプロセスで起動し，HTTP Server を使用する場合の見積もり例を次の図に示します。

図 5-1　HTTP Server を使用する場合のスレッド数の見積もり例



図に示した HTTP Server を使用する場合のスレッド数の見積もり例を次に示します。

CORBA ネーミングサービスをインプロセスで起動し HTTP Server を使用する場合の計算式

```
最大スレッド数 = 76+A+B+C+D+E+F+G+H+I+J+K+L+M+N+O+P+Q+R+S
```

上記の計算式より算出した結果と設定内容について，次に示します。

最大スレッド数= 76 + 0 + 0 + 1 + 5 + 72 + 72 + 144 + 2 + 0 + 2 + 0 + 0 + 0 + 0 + 0 + 0 + 100 + 0 + 0 = 474

表 5-13　HTTP Server を使用する場合のスレッド数の見積もり内容（例）

設定項目	設定する値	説明
A	0	Entity Bean を使用していないため，0 を設定します。
B	0	Message-Driven Bean を使用していないため，0 を設定します。
C	1	リモート呼び出しは行われていません。また，EJB が Web アプリケーションからのローカル呼び出しの場合，EJB 実行でのスレッドは生成されません。
D	4+1	C と同様に EJB クライアントに関する値は 0 になります。

設定項目	設定する値	説明
E	48+24	−
F	72	1 リクエストが 1 トランザクションとした場合，最大同時リクエスト数である MaxClient の値を設定します。
G	(48+24)×2	−
H	2	−
I	0	グローバルトランザクションは使用していないため，0 を設定します。
J	2	−
K	0	展開ディレクトリ形式ではないため，0 を設定します。
L	0	TP1 インバウンドアダプタを使用していないため，0 を設定します。
M	0	非同期 Session Bean を使用していないため，0 を設定します。
N	0	非同期 Session Bean を使用していないため，0 を設定します。
O	0	専用スレッドによる応答電文受信を設定していないため，0 を設定します。
P	0	コネクションのクローズの抑止を設定していないため，0 を設定します。
Q	100	−
R	0	Java Batch を使用していないため，0 を設定します。
S	0	Concurrency Utilities for Java EE を使用していないため，0 を設定します。

(b)　JavaVM のオプション指定に応じて使用するスレッド数

JavaVM のオプション指定に応じて，次の計算式で算出してください。A は，-XX:+UseG1GC オプションを指定している場合だけ加算します。B は，-XX:+HitachiUseExplicitMemory オプションを指定した場合だけ加算します。C は，-XX:+UseZGC オプションを指定している場合だけ加算します。

最大スレッド数 = A + B + C

（凡例）

- A：G1GC で使用するスレッド数（a + b + 2）※
- B：明示管理ヒープ機能で使用するスレッド数（論理 CPU 数。ただし，論理プロセッサ数が 8 以上の場合は 8。なお，J2EE サーバ起動時の論理 CPU 数によって決定されるため，起動後に論理 CPU の数を変更してもスレッド数は変化しない）
- C：ZGC で使用するスレッド数（a + b + 5）※

注※

凡例は次のとおりです。

a：-XX:ParalleGCThreads オプションに指定した値。このオプションの指定を省略した場合は，論理 CPU 数を基にした-XX:ParalleGCThreads オプションのデフォルト値。なお，J2EE サーバ起動時の論理 CPU 数によって決定されるため，起動後に論理 CPU の数を変更してもスレッド数は変化しない。

b：-XX:ConcGCThreads オプションに指定した値。このオプションの指定を省略した場合は，論理 CPU 数を基にした-XX:ConcGCThreads オプションのデフォルト値。なお，J2EE サーバ起動時の論理 CPU 数によって決定されるため，起動後に論理 CPU の数を変更してもスレッド数は変化しない。

JavaVM のオプションについては，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の次の個所を参照してください。

- 14.5　Application Server で指定できる Java HotSpot VM のオプション
- -XX:[+|-]HitachiUseExplicitMemory（明示管理ヒープ機能オプション）

(2)　ファイルディスクリプタ数

ファイルディスクリプタ数の計算式を次に示します。

- **J2EE サーバの場合**

  最大ファイルディスクリプタ数 ＝ （153 + A + B×3 + C + D + E + F×2 + G + H + I + J + K） / 0.8

（凡例）

- A：データベースコネクションの数
- B：EJB クライアントのプロセス数
- C：J2EE サーバに接続するコネクション数の総和
- D：次の式で算出した TP1 インバウンドアダプタが使用するファイルディスクリプタ数（TP1 インバウンド連携機能を使用する場合だけ加算する。なお，先頭で加算している固定値は，TP1 インバウンド連携機能の内部のファイルディスクリプタ数）

  8 ＋ TP1 インバウンドアダプタのプロパティ max_connections に指定した値

  ＋ TP1 インバウンドアダプタのプロパティ trn_max_connections に指定した値

  ＋各 MDB（サービス）の Message-driven Bean 属性ファイルの<pooled-instance><maximum>に指定した値の総和×2

  ＋ TP1 インバウンドアダプタのプロパティ rpc_max_thread_count に指定したスレッド数×2

  ＋ TP1 インバウンドアダプタのプロパティ trn_max_thread_count に指定したスレッド数×2

- E：J2EE アプリケーションに含む JAR ファイルの数
- F：リソースアダプタ数

- G：usrconf.cfg の add.class.path キーに指定した JAR ファイルの数
- H：Web アプリケーションの WEB-INF/lib に含む JAR ファイルの数
- I：NIO HTTP サーバの最大スレッド数
- J：Java Batch の最大スレッド数
- K：Concurrency Utilities for Java EE の最大スレッド数

(3) CORBA ネーミングサービス（インプロセス起動時）のスレッド数の見積もり

CORBA ネーミングサービスを J2EE サーバ起動時にインプロセスで起動させる場合に，J2EE サーバ上で生成される CORBA ネーミングサービスのスレッド数の見積もりについて説明します。

インプロセスで起動する場合の CORBA ネーミングサービスのスレッド数は，次のように見積もります。

合計スレッド数 = 初期化時に生成されるスレッド数＋ワーカスレッド数

(a) 初期化時に生成されるスレッド数

初期化時に生成されるスレッド数は，usrconf.properties の vbroker.agent.enableLocator キーの値が true の場合は 6，false の場合は 4 です。なお，vbroker.agent.enableLocator キーは，CTM 連携機能を有効（ejbserver.ctm.enabled キーに true を指定）にした場合，自動的に true が設定されます。

(b) ワーカスレッド数

ワーカスレッド数は，「同時受け付けリクエスト数＋ 1」と「クライアントと CORBA ネーミングサービス間のコネクション数×2」の合計になります。

ワーカスレッド数に関連するキーを次に示します。

- vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.threadMax
- vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.threadMin
- vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.threadMaxIdle

これらのキーは，usrconf.properties の ejbserver.naming.exec.args キーの値として指定します。これらのキーの詳細については，マニュアル「Borland(R) Enterprise Server VisiBroker(R) デベロッパーズガイド」，およびマニュアル「Borland(R) Enterprise Server VisiBroker(R) プログラマーズリファレンス」を参照してください。

vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.threadMax キーで最大値を指定している場合のワーカスレッド数は，「このキーで指定した最大値」と「クライアントと CORBA ネーミングサービス間のコネクション数」の合計になります。

ただし，vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.threadMin キーでワーカスレッド数の最小値を指定している場合，ワーカスレッド合計数が最小値に満たないときは，最小値がワーカスレッド数となります。

最大値を指定していない場合は，多重度の増加に伴いワーカスレッド数も増加していきます。ただし，ワーカスレッドは，アイドルになってから vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.threadMaxIdle キー（デフォルト値は 300 秒）で指定した時間が経過したあとに消滅（30 秒の誤差があります）しますので，負荷が下がるとスレッド数も減少します。

ワーカスレッド数の最大値を指定している場合に，スレッド数とワーカスレッドの最大値が同じになったときは，これ以降のリクエスト受け付けはエラー扱いにはしないで，次のように処理を継続します。

- 受信済みのリクエストについては処理を継続します。
- 新規のリクエストはソケットから read()されないで，TCP の受信バッファ，クライアント側の送信バッファで滞留します。TCP のバッファがいっぱいの場合は，クライアント側で送信待ちとなります。

処理中だったワーカスレッドが空きになった（応答を返した）時点で，次のリクエストの受信処理が行われます。

5.2.2　運用管理エージェントが使用するリソースの見積もり

運用管理エージェントが使用するリソースの見積もりについて OS ごとに説明します。

(1)　Windows の場合

Windows を使用する場合のスレッド数の計算式を次に示します。

- スレッド数の計算式

  使用スレッド数 = 30 + 7×論理サーバの数※

  注※　論理 CTM は論理サーバの数を 2 で計算してください。

  （凡例）

  - 30：運用管理エージェント本体が使用するスレッド数
  - 7：論理サーバ一つ当たりの運用管理エージェントが使用するスレッド数

論理サーバのステータスが稼働になったあとのスレッド数の計算式を次に示します。

- 平時のスレッド数の計算式

  使用スレッド数 = 30 + 5×論理サーバの数※

  注※　論理 CTM は論理サーバの数を 2 で計算してください。

  （凡例）

  - 30：運用管理エージェント本体が使用するスレッド数
  - 5：論理サーバ一つ当たりの運用管理エージェントが使用するスレッド数

(2) UNIX の場合

UNIX を使用する場合のスレッド数とファイルディスクリプタ数の計算式について説明します。

(a) スレッド数

スレッド数の計算式を次に示します。

- **スレッド数の計算式**
  使用スレッド数 = 30 + 5×論理サーバの数※
  注※　論理 CTM は論理サーバの数を 2 で計算してください。
  （凡例）
  - 30：運用管理エージェント本体が使用するスレッド数
  - 5：論理サーバ一つ当たりの運用管理エージェントが使用するスレッド数

論理サーバのステータスが稼働になったあとのスレッド数の計算式を次に示します。

- **平時のスレッド数の計算式**
  使用スレッド数 = 30 + 5×論理サーバの数※
  注※　論理 CTM は論理サーバの数を 2 で計算してください。
  （凡例）
  - 30：運用管理エージェント本体が使用するスレッド数
  - 5：論理サーバ一つ当たりの運用管理エージェントが使用するスレッド数

(b) ファイルディスクリプタ数

ファイルディスクリプタ数の計算式を次に示します。

- **ファイルディスクリプタ数の計算式**
  使用ファイルディスクリプタ数 = 20 ＋論理サーバを構成するプロセスの数×6
  （凡例）
  - 20：運用管理エージェント本体が使用するファイルディスクリプタ数
  - 6：論理サーバを構成する 1 プロセス当たりで運用管理エージェントが使用するファイルディスクリプタ数

論理サーバのステータスが稼働になったあとのファイルディスクリプタ数の計算式を次に示します。

- **平時のファイルディスクリプタ数の計算式**
  使用ファイルディスクリプタ数 = 20 ＋論理サーバを構成するプロセスの数×3
  （凡例）
  - 20：運用管理エージェント本体が使用するファイルディスクリプタ数

- 3：論理サーバを構成する 1 プロセス当たりで運用管理エージェントが使用するファイルディスクリプタ数

5.2.3　パフォーマンストレーサが使用するリソースの見積もり

パフォーマンストレーサが使用するリソースの見積もりについて，OS ごとに説明します。

(1)　Windows の場合

Windows を使用する場合の，パフォーマンストレーサが使用するリソースの見積もりについて説明します。

(a)　共用メモリ用ファイルサイズ

パフォーマンストレーサが使用する共用メモリ用ファイルサイズ（単位：バイト）は，PRF デーモンごとに算出します。計算式を次に示します。

- PRF デーモンごとの共用メモリの計算式
  共用メモリ用ファイルサイズ = -PrfTraceBufferSize 指定値×1,024 + 18,496

(b)　%PRFSPOOL%のディスク占有量

%PRFSPOOL%のディスク占有量の計算式を次に示します。

- %PRFSPOOL%のディスク占有量の計算式
  ディスク占有量 = 2.0MB
  ＋ ｛(-PrfTraceBufferSize 指定値＋ 20KB）×5
  ＋-PrfTraceFileSize 指定値×-PrfTraceCount 指定値×r｝ ×n
  + 224KB×(256 + m)
  + 224KB×（64 + p)
  （凡例）
  - n：PRF デーモンの数
  - m：起動中のパフォーマンストレース出力プロセス数と正常終了しなかったパフォーマンストレース出力プロセス数
    パフォーマンストレーサでは，パフォーマンストレース出力プロセスごとに，保守情報として内部トレースをファイルに出力します。このファイルはプロセス起動時に作成されますが，プロセスが正常終了しない場合はファイルが残ります。ファイルの削除処理は PRF デーモン起動時，および PRF デーモン起動後 24 時間ごとに実施しますが，256 ファイルは削除されないで残ります。このため，最大ファイル数は「256 + 24 時間のうちに実行するパフォーマンストレース出力プロセス数」になります。
  - p：起動中の性能解析トレースで使用するコマンド，デーモンのプロセス数

パフォーマンストレーサでは，性能解析トレースで使用するコマンド，およびデーモンごとに，保守情報として内部トレースをファイルに出力します。このファイルはプロセス起動時に作成されます。PRF デーモン起動時，および PRF デーモン起動後 24 時間ごとにファイル削除処理を実施しますが，64 ファイルは削除されないで残ります。このため，最大ファイル数は「64 ＋ 24 時間のうちに実行する性能解析トレースで使用するコマンド，およびデーモンのプロセス数」になります。

- r：バックアップ係数
  PRF トレースのバックアップ分を算出する値です。cprfstart コマンドの起動オプションに-PrfNoBackUp 0 を指定する場合 2, それ以外の場合 1 になります。

上記のディスク容量は目安です。このため，十分な余裕を持って，%PRFSPOOL%を作成してください。

(2)　AIX の場合

AIX を使用する場合，パフォーマンストレーサを使用するために，次の値を考慮してカーネルパラメタを設定する必要があります。正しく設定できていない場合には，パフォーマンストレーサのプロセスが起動できなかったり，動作中にリソース不足で異常終了になったりするおそれがあります。なお，カーネルパラメタの設定については，使用している OS のマニュアルを参照してください。

(a)　共用メモリ用ファイルサイズ

パフォーマンストレーサが使用する共用メモリ用ファイルサイズ（単位：バイト）は，PRF デーモンごとに算出します。計算式を次に示します。

- PRF デーモンごとの共用メモリの計算式
  共用メモリ用ファイルサイズ = -PrfTraceBufferSize 指定値×1,024 + 18,496

共用メモリ用ファイルサイズは，環境変数 EXTSHM で設定します。計算式で算出した値以上に共用メモリが割り当てられるように設定してください。

(b)　ファイルディスクリプタ数

ファイルディスクリプタ数は，/etc/security/limits ファイルの「nofiles」で設定します。PRF デーモン起動時に使用するファイルディスクリプタの数は，32 以上に設定してください。

(c)　$PRFSPOOL のディスク占有量

$PRFSPOOL のディスク占有量の計算式を次に示します。

- $PRFSPOOL のディスク占有量の計算式
  ディスク占有量 = 2.0MB
  ＋｛(-PrfTraceBufferSize 指定値＋ 20KB）×5
  ＋-PrfTraceFileSize 指定値×-PrfTraceCount 指定値×r｝×n
  + 224KB×(256 + m)
  + 224KB×（64 + p）

（凡例）

- n：PRF デーモンの数
- m：起動中のパフォーマンストレース出力プロセス数と正常終了しなかったパフォーマンストレース出力プロセス数

  パフォーマンストレーサでは，パフォーマンストレース出力プロセスごとに，保守情報として内部トレースをファイルに出力します。このファイルはプロセス起動時に作成されますが，プロセスが正常終了しない場合はファイルが残ります。ファイルの削除処理は PRF デーモン起動時，および PRF デーモン起動後 24 時間ごとに実施しますが，256 ファイルは削除されないで残ります。このため，最大ファイル数は「256 + 24 時間のうちに実行するパフォーマンストレース出力プロセス数」になります。
- p：起動中の性能解析トレースで使用するコマンド，デーモンのプロセス数

  パフォーマンストレーサでは，性能解析トレースで使用するコマンド，およびデーモンごとに，保守情報として内部トレースをファイルに出力します。このファイルはプロセス起動時に作成されます。PRF デーモン起動時，および PRF デーモン起動後 24 時間ごとにファイル削除処理を実施しますが，64 ファイルは削除されないで残ります。このため，最大ファイル数は「64 + 24 時間のうちに実行する性能解析トレースで使用するコマンド，およびデーモンのプロセス数」になります。
- r：バックアップ係数

  PRF トレースのバックアップ分を算出する値です。cprfstart コマンドの起動オプションに-PrfNoBackUp 0 を指定する場合 2, それ以外の場合 1 になります。

上記のディスク容量は目安です。このため，十分な余裕を持って，$PRFSPOOL を作成してください。

(3)　Linux の場合

Linux を使用する場合，パフォーマンストレーサを使用するために，次の値を考慮してカーネルパラメタを設定する必要があります。正しく設定できていない場合には，パフォーマンストレーサのプロセスが起動できなかったり，動作中にリソース不足で異常終了になったりするおそれがあります。なお，カーネルパラメタの設定については，使用している OS のマニュアルを参照してください。

(a)　共用メモリ用ファイルサイズ

パフォーマンストレーサが使用する共用メモリ用ファイルサイズ（単位：バイト）は，PRF デーモンごとに算出します。計算式を次に示します。

- PRF デーモンごとの共用メモリの計算式

  共用メモリ用ファイルサイズ = -PrfTraceBufferSize 指定値×1,024 + 18,496

共用メモリ用ファイルサイズは，/etc/sysctl.conf ファイルの「kernel.shmmax」で設定します。計算式で算出した値以上に共用メモリが割り当てられるように設定してください。

(b)　ファイルディスクリプタ数

ファイルディスクリプタ数は，/etc/security/limits.conf ファイルの「nofiles」で設定します。PRF デーモン起動時に使用するファイルディスクリプタの数は，32 以上に設定してください。

(c)　$PRFSPOOL のディスク占有量

$PRFSPOOL のディスク占有量の計算式については，AIX の場合と同じです。AIX の計算式を参照してください。

5.2.4　CTM が使用するリソースの見積もり

CTM が使用するリソースの見積もりについて，OS ごとに説明します。なお，バッチアプリケーションの実行環境では CTM は使用できません。

(1)　Windows の場合

Windows を使用する場合に CTM が使用するリソースの見積もりについて説明します。

(a)　共用メモリ用ファイルサイズ

CTM で使用する共用メモリ用ファイルサイズ（単位：バイト）の計算式について説明します。CTM デーモンの場合は，複数の共用メモリを確保する必要があります。CTM デーモン単位で固定長の共用メモリファイルと，可変長の共用メモリファイルがあります。それぞれのファイルサイズの計算式を次に示します。

- CTM ドメインマネジャの共用メモリ用ファイルサイズの計算式
  共用メモリ用ファイルサイズ＝1,018,320＋（2,362×-CTMMaxCTM 指定値）
- CTM デーモンの共用メモリ用ファイルサイズの計算式
  CTM デーモンの場合は，CTM デーモン単位で固定長の共用メモリ用ファイルと可変長の共用メモリ用ファイルを確保する必要があります。
  - 固定長の共用メモリ用ファイルサイズ＝
    551,840＋（1,208×-CTMQueueCount 指定値）
  - 可変長の共用メモリ用ファイルサイズ＝
    1,027,008
    ＋（928×-CTMClientConnectCount 指定値）
    ＋（256×-CTMServerConnectCount 指定値）
    ＋（512×-CTMEntryCount 指定値）
    ＋（1,024×-CTMServerCacheSize 指定値）
    ＋（512×-CTMQueueCount 指定値）
    ＋（544×-CTMQueueCount 指定値×-CTMQueueRegistCount 指定値）
    ＋（512×-CTMDispatchParallelCount 指定値）

(b)　稼働統計情報ファイルサイズ

稼働統計情報ファイルサイズ（単位：バイト）の計算式を次に示します。

- オンライン開始から終了までに必要な稼働統計情報ファイルサイズの計算式
  ファイルサイズ＝A＋B
  （凡例）
  - A：（オンライン開始から終了までに実行されるリクエストの数）×（1 リクエストで出力される情報量）
  - B：（オンライン開始から終了までの時間（分）／ ctmstsstart -CTMInterval に指定した取得間隔）×（1 回に出力される CTM ノード単位，キュー単位統計情報量）

1 リクエストで出力される情報量と，1 回に出力される CTM ノード単位，キュー単位統計情報量の計算式を次に示します。

- 1 リクエストで出力される情報量の計算式
  情報量＝（↑（80＋A＋B＋C＋D＋63）／64↑×64）×3
  （凡例）
  - A：リクエストを実行するアプリケーションを管理している CTM ドメイン名のドメイン長
  - B：リクエストを実行するアプリケーションを管理している CTM デーモンの CTM 識別子の長さ
  - C：キュー名称の長さ
  - D：オペレーション名称の長さ
- 1 回に出力される CTM ノード単位，キュー単位統計情報量の計算式
  統計情報量＝↑（2,144＋344×キュー数＋63）／64↑×64

(c)　%CTMSPOOL%のディスク占有量

%CTMSPOOL%のディスク占有量の計算式を次に示します。

- %CTMSPOOL%のディスク占有量の計算式
  ディスク占有量＝7.0MB
  ＋（18.5MB＋1.0MB×-CTMLogFileSize 指定値×-CTMLogFileCount 指定値）×n
  ＋（1KB＋0.5KB×k）×（m＋l）
  ＋1KB×m×j
  ＋224KB×p
  ＋1,120KB×（64＋q）
  ＋CTM ドメインマネジャの共有メモリ用ファイルサイズ×5
  ＋CTM デーモンの共有メモリ用ファイルサイズ×5×n
  ＋CTM ドメインマネジャの core サイズ
  ＋CTM デーモンの core サイズ×n

+ CTMレギュレータのcoreサイズ×3×m
+ OTMゲートウェイのcoreサイズ×3×l
+ (-CTMStatsFileSize指定値×-CTMStatsFileCount指定値)×n

(凡例)

- j:EJBの総数
- k:CTMレギュレータおよびOTMゲートウェイに接続できるクライアント数(-CTMClientConnectCountオプション指定値)
- l:OTMゲートウェイの総数
- m:CTMレギュレータの総数
- n:CTMデーモンの数
- p:起動中のユーザアプリケーションのプロセス数と正常終了しなかったユーザアプリケーションのプロセス数

  CTMでは，ユーザアプリケーションごとに，保守情報として内部トレースをファイルに出力します。このファイルはプロセス起動時に作成されますが，プロセスが正常終了しない場合はファイルが残ります。ファイルの削除処理はCTMドメインマネジャ起動時，およびCTMドメインマネジャ起動後24時間ごとに実施しますが，256ファイルは削除されないで残ります。このため，最大ファイル数は「256＋起動中プロセス数」になります。
- q:起動中のシステム系プロセス数

  CTMでは，プロセスごとに，保守情報として内部トレースをファイルに出力します。このファイルはプロセス起動時に作成されます。CTMドメインマネジャ起動時，およびCTMドメインマネジャ起動後24時間ごとにファイル削除処理を実施しますが，64ファイルは削除されないで残ります。このため，最大ファイル数は「64＋起動中プロセス数」になります。

上記のディスク容量は目安です。このため，十分な余裕を持って，%CTMSPOOL%を作成してください。

(2) AIXの場合

AIXを使用する場合，CTMを使用するために，次の値を考慮してカーネルパラメタを設定する必要があります。正しく設定できていない場合には，CTMのプロセスが起動できなかったり，動作中にリソース不足で異常終了になったりするおそれがあります。なお，カーネルパラメタの設定については，使用しているOSのマニュアルを参照してください。

(a) 共用メモリ用ファイルサイズ

共用メモリ用ファイルサイズは，環境変数EXTSHMで設定します。計算式で算出した値以上に共用メモリが割り当てられるように設定してください。なお，計算式については，Windowsの場合と同じです。Windowsの計算式を参照してください。

(b)　稼働統計情報ファイルサイズ

稼働統計情報ファイルサイズの計算式については，Windows の場合と同じです。Windows の計算式を参照してください。

(c)　ファイルディスクリプタ数

CTM デーモンでは，起動時のオプションを基に，次の計算式で示すように，プロセスで使用できるファイルディスクリプタの数を増加させます。OS の設定値が，計算式で設定した CTM デーモンで必要とするファイルディスクリプタの最大値に満たない場合には，プロセス起動でエラー終了します。計算式を基に/etc/security/limits.conf ファイルの「nofiles」を変更してください。

- **CTM デーモンで必要とするファイルディスクリプタ数の最大値の計算式**
  最大値 = （A×2 + B×4 + C×2 + D×E + F×EJB 数+ G + 100） / 0.8
  （凡例）
  - A：-CTMMaxCTM 値（ctmd が属する ctmdmd で指定された値）
  - B：-CTMClientConnectCount 値
  - C：-CTMServerConnectCount 値
  - D：-CTMQueueCount 値
  - E：-CTMQueueRegistCount 値
  - F：-CTMDispatchParallelCount 値
  - G：Create を発行する EJB クライアントの総数

(d)　$CTMSPOOL のディスク占有量

$CTMSPOOL のディスク占有量の計算式を次に示します。

- **$CTMSPOOL のディスク占有量の計算式**
  ディスク占有量 = 7.0MB
  + （18.5MB + 1.0MB×-CTMLogFileSize 指定値×-CTMLogFileCount 指定値）×n
  + （1KB + 0.5KB×k）×（m + l）
  + 1KB×m×j
  + 224KB×p
  + 1,120KB×（64 + q）
  + CTM ドメインマネジャの共有メモリ用ファイルサイズ×5
  + CTM デーモンの共有メモリ用ファイルサイズ×5×n
  + CTM ドメインマネジャの core サイズ
  + CTM デーモンの core サイズ×n
  + CTM レギュレータの core サイズ×3×m
  + OTM ゲートウェイの core サイズ×3×l

＋（-CTMStatsFileSize 指定値×-CTMStatsFileCount 指定値）×n

（凡例）

- j：EJB の総数
- k：CTM レギュレータおよび OTM ゲートウェイに接続できるクライアント数（-CTMClientConnectCount オプション指定値）
- l：OTM ゲートウェイの総数
- m：CTM レギュレータの総数
- n：CTM デーモンの数
- p：起動中のユーザアプリケーションのプロセス数と正常終了しなかったユーザアプリケーションのプロセス数

  CTM では，ユーザアプリケーションごとに，保守情報として内部トレースをファイルに出力します。このファイルはプロセス起動時に作成されますが，プロセスが正常終了しない場合はファイルが残ります。ファイルの削除処理は CTM ドメインマネジャ起動時，および CTM ドメインマネジャ起動後 24 時間ごとに実施しますが，256 ファイルは削除されないで残ります。このため，最大ファイル数は「256 ＋起動中プロセス数」になります。
- q：起動中のシステム系プロセス数

  CTM では，プロセスごとに，保守情報として内部トレースをファイルに出力します。このファイルはプロセス起動時に作成されます。CTM ドメインマネジャ起動時，および CTM ドメインマネジャ起動後 24 時間ごとにファイル削除処理を実施しますが，64 ファイルは削除されないで残ります。このため，最大ファイル数は「64 ＋起動中プロセス数」になります。

上記のディスク容量は目安です。このため，十分な余裕を持って，$CTMSPOOL を作成してください。

(3)　Linux の場合

Linux を使用する場合，CTM を使用するために，次の値を考慮してカーネルパラメタを設定する必要があります。正しく設定できていない場合には，CTM のプロセスが起動できなかったり，動作中にリソース不足で異常終了になったりするおそれがあります。なお，カーネルパラメタの設定については，使用している OS のマニュアルを参照してください。

(a)　共用メモリ用ファイルサイズ

共用メモリ用ファイルサイズは，/etc/sysctl.conf ファイルの「kernel.shmmax」で設定します。計算式で算出した値以上に共用メモリが割り当てられるように設定してください。なお，計算式については，Windows の場合と同じです。Windows の計算式を参照してください。

(b)　稼働統計情報ファイルサイズ

稼働統計情報ファイルサイズの計算式については，Windows の場合と同じです。Windows の計算式を参照してください。

(c)　ファイルディスクリプタ数

ファイルディスクリプタ数は，/etc/security/limits.conf ファイルの「nofiles」で設定します。計算式を基に「nofiles」を変更してください。なお，計算式については，AIX の場合と同じです。AIX の計算式を参照してください。

(d)　$CTMSPOOL のディスク占有量

$CTMSPOOL のディスク占有量の計算式については，AIX の場合と同じです。AIX の計算式を参照してください。

5.2.5　cjclstartap プロセスの見積もり

cjclstartap プロセスのスレッド数の見積もりを次に示します。

- 明示管理ヒープ機能を使用する場合
  Java アプリケーションで生成するスレッド数＋ 30
- 明示管理ヒープ機能を使用しない場合
  Java アプリケーションで生成するスレッド数＋ 22
- G1GC を使用する場合
  Java アプリケーションで生成するスレッド数＋ 22 ＋ G1GC で使用するスレッド数
  G1GC で使用するスレッド数については，「5.2.1　J2EE サーバが使用するリソースの見積もり」を参照してください。
- ZGC を使用する場合
  Java アプリケーションで生成するスレッド数＋ 22 ＋ ZGC で使用するスレッド数
  ZGC で使用するスレッド数については，「5.2.1　J2EE サーバが使用するリソースの見積もり」を参照してください。

5.3　プロセスごとに使用するメモリの見積もり

この節では，アプリケーションサーバの各プロセスで使用するメモリの使用量の見積もりについて説明します。

5.3.1　J2EE サーバが使用する仮想メモリの使用量の見積もり

ここでは，仮想メモリの使用量の見積もり方法について説明します。なお，仮想メモリの使用量の計算式に使用している JavaVM 起動時のオプションの詳細については，「7.2.6　SerialGC 使用時の JavaVM で使用するメモリ空間の構成と JavaVM オプション」を参照してください。

また，ここで説明する内容のほか，明示管理ヒープ機能で使用するメモリサイズの見積もりが必要です。明示管理ヒープ機能で使用するメモリサイズの見積もりについては，「7.11　Explicit ヒープのチューニング」を参照してください。

(1)　仮想メモリの使用量の計算式

仮想メモリの使用量（単位：メガバイト）の計算式を次に示します。

J2EE サーバの仮想メモリの使用量＝A＋B＋C＋（D＋22）×E＋F＋G

（凡例）

- A：Java ヒープサイズ
  初期値は，JavaVM 起動時のオプション-Xms に指定した値です。この値は，J2EE サーバ起動中に，最大で JavaVM 起動時のオプション-Xmx に指定した値まで拡張されます。なお，ZGC を使用する場合は，仮想メモリを多く必要とします。ZGC を使用する場合は，この値は-Xmx に指定した値の 48 倍としてください。これは物理メモリの実際の使用サイズには影響しません。詳細については「7.20 ZGC 使用時の注意事項（JDK17 以降の場合）」を参照してください。
- B：Metaspace 領域サイズ
  初期値は，JavaVM が割り当てた値です。この値は，J2EE サーバ起動中に，最大で JavaVM 起動時のオプション-XX:MaxMetaspaceSize に指定した値まで拡張されます。

  監査ログを使用する場合

  監査ログを使用する場合は，値に 1 メガバイトを加えてください。
- C：ネイティブプログラム使用領域サイズ
  OS ごとに使用する値が異なります。ネイティブプログラム使用領域サイズの OS ごとの値について，次の表に示します。

表 5-14　ネイティブプログラム使用領域サイズの値一覧

OS 種別	使用領域の値 （単位：メガバイト）
Windows	700
Linux	600

監査ログを使用する場合

監査ログを使用する場合は，OS ごとのネイティブプログラム使用領域サイズの値に，監査ログで使用する値を加える必要があります。監査ログで使用する値を算出するには，auditlog.raslog.message.filesize キー，および auditlog.raslog.exception.filesize キーを使用します。これらのキーは，auditlog.properties（監査ログ定義ファイル）で指定するキーです。これらのキーの詳細については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「11.2.1　監査ログ定義ファイル」を参照してください。

監査ログで使用する値（単位：メガバイト）の計算式を次に示します。

監査ログで使用する値 ＝

8 ＋（auditlog.raslog.message.filesize の指定値 / $1{,}024^2$）※＋
（auditlog.raslog.exception.filesize の指定値 / $1{,}024^2$）※

注※　指定値の単位はバイトです。仮想メモリの使用量の値の単位はメガバイトであるため，$1{,}024^2$ で割る必要があります。

- D：J2EE サーバのスレッド数

  J2EE サーバが使用するスレッド数です。J2EE サーバが使用するスレッド数については，「5.2.1　J2EE サーバが使用するリソースの見積もり」を参照してください。

- E：スタック領域サイズ

  JavaVM 起動時のオプション-Xss に指定した値です。

- F：Explicit ヒープサイズ

  明示管理ヒープ機能で使用される Explicit ヒープのサイズです。この値は，J2EE サーバ起動中に最大で-XX:HitachiExplicitHeapMaxSize に指定した値まで拡張されます。

- G：コードキャッシュ領域の最大サイズ

  JavaVM 起動時のオプション-XX:ReservedCodeCacheSize に指定した値です。

(2)　仮想メモリの使用量を計算する場合の注意事項

- ネイティブプログラム使用領域サイズの値は変動します。値が変動するのは次のような場合です。
  - ORB のトレースサイズを変更した場合
  - JDBC ドライバの使用領域サイズの値を変更した場合
  - 使用する製品のネイティブライブラリ使用領域サイズの値を変更した場合

このような場合，製品ごとのメモリ使用量をネイティブプログラム使用領域サイズの値に加えてください。製品ごとのメモリ使用量は，使用する製品のドキュメントに従って算出してください。

- J2EE サーバの標準構成で使用する仮想メモリの使用量は，使用するソフトウェア製品の影響で，見積もり値よりも大きくなる場合があります。その場合の増加分については，仮想メモリの使用量の値に加えてください。増加分のメモリ使用量は，使用するソフトウェア製品のドキュメントに従い，製品ごとのメモリ使用量を算出してください。
- Linux の場合，メモリのスワップファイルが不足すると，システムがスローダウンします。仮想メモリの上限値を，実メモリサイズ分指定することをお勧めします。

5.3.2 CTM のデーモンプロセスが使用するメモリの使用量の見積もり

CTM のデーモンプロセスの見積もり式を次に示します。

```
最大メモリ所要量 ＝A＋{B＋E＋(2×D)}×C＋F×G＋(D-1)×16メガバイト
```

（凡例）

A：基本メモリ量（128 メガバイト）

B：CTM デーモンのキューに滞留する最大リクエスト数※1

C：最大リクエスト電文サイズ※2

D：-CTMClientConnectCount 値※3

E：-CTMDispatchParallelCount 値※4

F：CTM デーモンで必要とするスレッド数の最大値。見積もり式を示します。

F ＝(a×4 + D×3 + b×2 + c×d + E + e + 32) / 0.8

（凡例）

a：-CTMMaxCTM 値（ctmd が属する ctmdmd で指定された値）※6

b：-CTMServerConnectCount 値※7

c：-CTMQueueCount 値※8

d：-CTMQueueRegistCount 値※9

e：Create を発行する EJB クライアントの総数

G：スレッドスタックサイズ※5

注※1

最大リクエスト数はキュー長（キューが複数ある場合は，各キュー長の合計値）になります。

注※2

メモリサイズに余裕を持たせるため，キューに溜まるリクエストがすべて最大のサイズの場合を想定しますが，現実的な値としては，リクエスト電文サイズの平均値を指定することをお勧めします。

注※3

メモリサイズに余裕を持たせるため，CTM デーモンに接続できる最大数の場合を想定しますが，現実的な値としては，CTM デーモンに接続する CTM レギュレータと OTM ゲートウェイの数を指定することをお勧めします。

注※4

メモリサイズに余裕を持たせるため，CTM デーモンに登録できる最大の同時実行数の場合を想定しますが，現実的な値としては，属性ファイルの<parallel-count>タグの値（キューが複数ある場合は，各キューの同時実行数の合計値）になります。

注※5

OS ごとに使用する値が異なります。スレッドスタックサイズの OS ごとのデフォルト値について，次に示します。

なお，スレッドスタックサイズについては，使用している OS のマニュアルを参照してください。

Windows：1 メガバイト

AIX：0.1 メガバイト（96 キロバイト）

Linux：10 メガバイト

注※6

メモリサイズに余裕を持たせるため，CTM デーモンが最大数の場合を想定しますが，現実的な値としては，CTM ドメイン内で管理する CTM デーモンの総数-1 を指定することをお勧めします。

注※7

メモリサイズに余裕を持たせるため，CTM デーモンに接続できる最大数の場合を想定しますが，現実的な値としては，CTM デーモンに接続する J2EE サーバの数を指定することをお勧めします。

注※8

メモリサイズに余裕を持たせるため，CTM デーモンに登録できる最大数の場合を想定しますが，現実的な値としては，CTM デーモンに登録するスケジュールキュー数を指定することをお勧めします。

注※9

メモリサイズに余裕を持たせるため，同じスケジュールキューを共有できる最大数の場合を想定しますが，現実的な値としては，同じスケジュールキューを共有する J2EE アプリケーション数を指定することをお勧めします。

ポイント

CTM デーモンがリクエストを送受信する際，一時的に最大で業務電文サイズの 3～5 倍程度のメモリを使用することがあります。ここで示した見積もり式では，この増分は基本サイズの 128 メガバイトに余裕値として含まれています。

しかし，業務電文サイズが 10 メガバイトや 20 メガバイトといった，非常に長大電文で，かつ同時実行数（パラレルカウントの合計値）が大きい場合，処理が重なりこの余裕値を超えてしまうことがあります。このような場合に，正確に最大メモリ使用量を見積もろうとすると，現実的な値で

はなくなるため，概算値はここで示した見積もり式で算出した上で，実際に業務を動作させてメモリ使用量の推移を計測することをお勧めします。

6 使用するリソースの見積もり（バッチアプリケーション実行基盤）

この章では，バッチアプリケーションを実行するシステムで使用するリソース，および仮想メモリ使用量の見積もり方法について説明します。システムを動作させるために必要なディスクおよびメモリの容量を算出するときの参考にしてください。

J2EE アプリケーション実行基盤の使用リソースおよび仮想メモリ所要量の見積もりについては，「5.　使用するリソースの見積もり（J2EE アプリケーション実行基盤）」を参照してください。

6.1　システム構成ごとに使用するリソース

システムが動作するためには，OS やデータベースなどのホスティング環境の設定が必要な場合があります。システムが必要とするリソースは，システムの構成ごとに異なるため，ここではシステム構成ごとに使用するリソースと，リソースの所要量の見積もりについて説明します。システム構成ごとに使用するリソースと，リソースの見積もりの参照先を次の表に示します。

表 6-1　システム構成ごとに使用するリソースと見積もりの参照先

システム構成ごとに使用するリソース	参照先
バッチサーバを配置する場合の使用リソース	6.1.1
データベースの使用リソース	6.1.2
CTM を使用する場合の使用リソース	6.1.3

6.1.1　バッチサーバを配置する場合の使用リソース

バッチサーバを配置する場合の使用リソースの見積もりについて，OS ごとに説明します。バッチサーバを配置する場合，バッチサーバを配置するマシンで使用するリソースを見積もります。

なお，使用リソースの見積もりの各表にある，「オプション設定ファイル例」については，使用している OS のバージョン，およびカーネルのバージョンごとに異なります。使用している OS のマニュアルを参照して，表中の見積もり式を基に見積もった値を設定してください。使用している OS で該当するカーネルパラメタが設定できない場合には，設定は不要です。

また，プロセスごとに使用するリソースの見積もりについては，「6.2　プロセスごとに使用するリソース」を参照してください。

なお，仮想メモリ所要量については，「6.3　仮想メモリの使用量の見積もり」を参照してください。また，ディスク占有量については，アプリケーションサーバのリリースノートを参照してください。

(1)　AIX の場合

AIX の場合の，アプリケーションサーバマシンの使用リソースの見積もりについて次の表に示します。

表 6-2　アプリケーションサーバマシンの使用リソースの見積もり（AIX の場合）

システムリソース		パラメタ	所要量	オプション設定ファイル例
サービスユニット※1	共用メモリ	−	PrfTraceBufferSize※2×1,024 + 18,496	−
	プロセス数	−	4	−
	スレッド数	−	バッチサーバのスレッド数※3 + 34	−

システムリソース		パラメタ	所要量	オプション設定ファイル例
	ファイルディスクリプタ数	nofiles	バッチサーバのファイルディスクリプタ数※3 + 43	/etc/security/limits
Management Server	プロセス数	−	5	−
	スレッド数	−	56	−
	ファイルディスクリプタ数	nofiles	43 +バッチサーバ数	/etc/security/limits

（凡例）−：該当しません。

注※1

サービスユニットとは，次のまとまりを指します。

バッチサーバ + パフォーマンストレーサ

注※2

パフォーマンストレーサのバッファメモリサイズを 512 キロバイト～102,400 キロバイトの範囲で指定します。PrfTraceBufferSize については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「4.12　論理パフォーマンストレーサで指定できるパラメタ」を参照してください。

注※3

バッチサーバのスレッド数とファイルディスクリプタ数については，「6.2.1　バッチサーバが使用するリソースの見積もり」を参照して算出してください。

(2)　Linux の場合

Linux の場合の，アプリケーションサーバマシンの使用リソースの見積もりについて次の表に示します。

表 6-3　アプリケーションサーバマシンの使用リソースの見積もり（Linux の場合）

システムリソース		パラメタ	所要量	オプション設定ファイル例
サービスユニット※1	共用メモリ	SHMMAX	PrfTraceBufferSize※2×1,024 + 18,496	/proc/sys/kernel/shmmax
	プロセス数	threads-max※3	4	/proc/sys/kernel/threads-max
	スレッド数	threads-max※3	バッチサーバのスレッド数※4 + 34	−
	ファイルディスクリプタ数	fs.file-max	バッチサーバのファイルディスクリプタ数※4 + 43	/proc/sys/fs/file-max
Management Server	プロセス数	threads-max※3	5	/proc/sys/kernel/threads-max
	スレッド数	threads-max※3	56	−
	ファイルディスクリプタ数	fs.files-max	43 +バッチサーバ数	/proc/sys/fs/file-max

（凡例）−：該当しません。

注※1

サービスユニットとは，次のまとまりを指します。

バッチサーバ＋パフォーマンストレーサ

注※2

パフォーマンストレーサのバッファメモリサイズを512キロバイト～102,400キロバイトの範囲で指定します。PrfTraceBufferSizeについては，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「4.12　論理パフォーマンストレーサで指定できるパラメタ」を参照してください。

注※3

threads-maxパラメタには，プロセス数とスレッド数の合計を指定してください。

注※4

バッチサーバのスレッド数とファイルディスクリプタ数については，「6.2.1　バッチサーバが使用するリソースの見積もり」を参照して算出してください。

6.1.2　データベースの使用リソース

DBMSの使用リソースの見積もりについて説明します。

仮想メモリ所要量については，「6.3　仮想メモリの使用量の見積もり」を参照してください。また，ディスク占有量については，Application ServerまたはDeveloperのリリースノートを参照してください。

DBMSの使用リソースの見積もりについて，次の表に示します。

表6-4　DBMSの使用リソースの見積もり

DBMS	使用リソース	所要量
HiRDB	最大同時接続数 (pd_max_users)	$\sum_{i=1}^{n^{※1}}$（リソースアダプタiのコネクションプールの最大値※2×2※3＋1※4）＋α※5
Oracle	最大同時接続数 (SESSIONS)	$\sum_{i=1}^{n^{※1}}$（リソースアダプタiのコネクションプールの最大値※2＋1※4）＋α※5

注※1

nは，システム内のバッチサーバにデプロイするリソースアダプタの総和です。

注※2

Connector属性ファイルのMaxPoolSizeパラメタの値を指定します。

注※3

次に示す条件にすべて当てはまる場合に，×2を行ってください。

1. トランザクションサポートレベルにXATransactionを使用する。
2. アプリケーションサーバが管理するトランザクション内でコネクション※を使ってデータベースにアクセスする。

3. 2.のトランザクションが決着する前に，トランザクション外でコネクション※を使ってデータベースにアクセスする。

注※　このコネクションは 1.の DB Connector から取得したコネクションで，かつ同一コネクションです。

注※4

トランザクションサポートレベルに XATransaction を指定しているリソースアダプタの場合に，＋ 1 を行ってください。

注※5

＋ α の最大値は，使用する DB Connector のコネクションプールの最大値の合計値になります。

この＋ α の値は，一時的にコネクションプールの最大値をオーバーするおそれのあるコネクションを指します。詳細を次に示します。

- コネクションの障害検知機能を使用する場合

  コネクションの障害検知機能を使用する場合，コネクションプールから取り除いた未使用のコネクションは，コネクションプール内のコネクション数としてカウントされません。そのため，コネクションプール内のコネクションとコネクションプールから取り除いた未使用のコネクションの総和が，コネクションプールのコネクション数の最大値を一時的に超えることがあります。

- cjclearpool コマンドを使用する場合

  通常モードの場合，コネクションプールから取り除いた使用中のコネクションは，コネクション数としてカウントされません。そのため，コネクションプール内のコネクションとコネクションプールから取り除いた使用中のコネクションの総和がコネクションプールの最大値を超えることがあります。

6.1.3　CTM を使用する場合の使用リソース

CTM（バッチアプリケーションのスケジューリング機能）を使用する場合の，使用リソースの見積もりについて，OS ごとに説明します。

なお，使用リソースの見積もりの各表にある，「オプション設定ファイル例」については，使用している OS のバージョン，およびカーネルのバージョンごとに異なります。使用している OS のマニュアルを参照して，表中の見積もり式を基に見積もった値を設定してください。使用している OS で該当するカーネルパラメタが設定できない場合には，設定は不要です。

(1)　AIX の場合

CTM を使用する場合の，使用リソースの見積もりについて次の表に示します。

表 6-5　CTM 使用時の使用リソースの見積もり（AIX の場合）

システムリソース	パラメタ	所要量	オプション設定ファイル例
共用メモリ	−	PrfTraceBufferSize※1×1,024 + 18,496 + CTM ドメインマネジャの共用メモリ※2 + CTM デーモンの共用メモリ※2	−
プロセス数	−	7 +バッチサーバ数※3	−
スレッド数	−	72 +(バッチサーバのスレッド数※4 + 7)×バッチサーバ数※3 + CTM デーモンで必要とするスレッド数※5	−
ファイルディスクリプタ数	nofiles	88 +(バッチサーバのファイルディスクリプタ数※4 + 6)×バッチサーバ数※3 + CTM デーモンで必要とするファイルディスクリプタ数※5	/etc/security/limits

（凡例）−：該当しません。

注※1

パフォーマンストレーサのバッファメモリサイズを 512 キロバイト～102,400 キロバイトの範囲で指定します。PrfTraceBufferSize については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「4.12　論理パフォーマンストレーサで指定できるパラメタ」を参照してください。

注※2

値については，「6.1.3(1)(a)　共用メモリ用ファイルサイズの計算式」を参照して算出してください。

注※3

簡易構築定義ファイルの<j2ee-server-count>タグの指定値を指します。

注※4

バッチサーバのスレッド数とファイルディスクリプタ数については，「6.2.1　バッチサーバが使用するリソースの見積もり」を参照して算出してください。

注※5

CTM デーモンで必要とするスレッド数とファイルディスクリプタ数については，「6.1.3(1)(b)　CTM デーモンで必要とするスレッド数とファイルディスクリプタ数の計算式」を参照して算出してください。

(a)　共用メモリ用ファイルサイズの計算式

共用メモリ用ファイルサイズを算出するには，CTM ドメインマネジャの共用メモリおよび CTM デーモンの共用メモリを算出する必要があります。それぞれの計算式について次に示します。

なお，計算式中の可変値には，次の値を使用してください。「ctm.」で始まるパラメタについては，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「4.3　簡易構築定義ファイル」を参照してください。

計算式に使用する値

-CTMMaxCTM：64

-CTMQueueCount：ctm.QueueCount

-CTMClientConnectCount：256

-CTMServerConnectCount：ctm.ServerConnectCount

-CTMEntryCount：-CTMClientConnectCount ＋-CTMServerConnectCount

-CTMServerCacheSize：ctm.ServerCacheSize

-CTMQueueRegistCount：ctm.QueueRegistCount

-CTMDispatchParallelCount：ctm.DispatchParallelCount

- **CTM ドメインマネジャの共用メモリ用ファイルサイズの計算式**

  CTM ドメインマネジャの共用メモリ用ファイルサイズの計算式を次に示します。

  共用メモリ用ファイルサイズ（単位：バイト） ＝

  　1,018,320 ＋（2,362 × -CTMMaxCTM 指定値）

- **CTM デーモンの共用メモリ用ファイルサイズの計算式**

  CTM デーモンの場合は，CTM デーモン単位で固定長の共用メモリ用ファイルと可変長の共用メモリ用ファイルを確保する必要があります。それぞれの計算式を次に示します。

  固定長の共用メモリ用ファイルサイズ（単位：バイト） ＝

  　551,840 ＋ （1,208 × -CTMQueueCount 指定値）

  可変長の共用メモリ用ファイルサイズ（単位：バイト） ＝

  　1,027,008

  　＋（928 × -CTMClientConnectCount 指定値）

  　＋（256 × -CTMServerConnectCount 指定値）

  　＋（512 × -CTMEntryCount 指定値）

  　＋（1,024 ×-CTMServerCacheSize 指定値）

  　＋（512 × -CTMQueueCount 指定値）

  　＋（544 × -CTMQueueCount 指定値 × -CTMQueueRegistCount 指定値）

  　＋（512 × -CTMDispatchParallelCount 指定値）

(b)　CTM デーモンで必要とするスレッド数とファイルディスクリプタ数の計算式

スレッド数およびファイルディスクリプタ数を算出するには，CTM デーモンで必要とするスレッド数とファイルディスクリプタ数を算出する必要があります。それぞれの計算式について次に示します。

- **CTM デーモンで必要とするスレッド数の計算式**

  最大値 ＝

  　（A×4 ＋ B×3 ＋ C×2 ＋ D×E ＋ F ＋ G ＋ 32） / 0.8

  （凡例）

  　A：-CTMMaxCTM 値（ctmd が属する ctmdmd で指定された値）

  　B：-CTMClientConnectCount 値

  　C：-CTMServerConnectCount 値

  　D：-CTMQueueCount 値

E：-CTMQueueRegistCount 値
F：-CTMDispatchParallelCount 値
G：Create を発行する EJB クライアントの総数

- **CTM デーモンで必要とするファイルディスクリプタ数の計算式**

最大値 ＝
（A×2 ＋ B×4 ＋ C×2 ＋ D×E ＋ F×EJB のインタフェース数＋ G ＋ 100） / 0.8

（凡例）
A：-CTMMaxCTM 値（ctmd が属する ctmdmd で指定された値）
B：-CTMClientConnectCount 値
C：-CTMServerConnectCount 値
D：-CTMQueueCount 値
E：-CTMQueueRegistCount 値
F：-CTMDispatchParallelCount 値
G：Create を発行する EJB クライアントの総数

(2) Linux の場合

CTM を使用する場合の，使用リソースの見積もりについて，次の表に示します。

表 6-6 CTM 使用時の使用リソースの見積もり（Linux の場合）

システムリソース	パラメタ	所要量	オプション設定ファイル例
共用メモリ	SHMMAX	PrfTraceBufferSize※1×1,024 + 18,496 + CTM ドメインマネジャの共用メモリ※2 + CTM デーモンの共用メモリ※2	/proc/sys/kernel/shmmax
プロセス数	threads-max※6 pid_max※6	7 ＋バッチサーバ数※3	/proc/sys/kernel/threads-max /proc/sys/kernel/pid_max
スレッド数	threads-max※6 pid_max※6	72 ＋(バッチサーバのスレッド数※4 + 7)×バッチサーバ数※3 + CTM デーモンで必要とするスレッド数※5 + CTM レギュレータで必要とするスレッド数※7	/proc/sys/kernel/threads-max /proc/sys/kernel/pid_max
ファイルディスクリプタ数	fs.file-max	88 ＋(バッチサーバのファイルディスクリプタ数※4 + 6)×バッチサーバ数※3 + CTM デーモンで必要とするファイルディスクリプタ数※5	/proc/sys/fs/file-max

（凡例）－：該当しません。

注※1

パフォーマンストレーサのバッファメモリサイズを 512 キロバイト～102,400 キロバイトの範囲で指定します。
PrfTraceBufferSize については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「4.12　論理パフォーマンストレーサで指定できるパラメタ」を参照してください。

注※2

値については，「6.1.3(1)(a)　共用メモリ用ファイルサイズの計算式」を参照して算出してください。

注※3

簡易構築定義ファイルの<j2ee-server-count>タグの指定値を指します。

注※4

バッチサーバのスレッド数とファイルディスクリプタ数については，「6.2.1　バッチサーバが使用するリソースの見積もり」を参照して算出してください。

注※5

CTM デーモンで必要とするスレッド数とファイルディスクリプタ数については，「6.1.3(1)(b)　CTM デーモンで必要とするスレッド数とファイルディスクリプタ数の計算式」を参照して算出してください。

注※6

threads-max パラメタおよび pid_max パラメタには，プロセス数とスレッド数の合計以上を指定してください。

注※7

CTM レギュレータで必要とするスレッド数については，「12+(EJB クライアントの総数×2)/0.8)」を使用して算出してください。

EJB クライアントの総数の最大値は ctmregltd の-CTMClientConnectCount 値の合計になります。

6.2　プロセスごとに使用するリソース

この節では，アプリケーションサーバの各プロセスで使用するリソースの所要量の見積もりについて説明します。

6.2.1　バッチサーバが使用するリソースの見積もり

ここでは，バッチサーバのスレッド数とファイルディスクリプタ数の見積もり方法について説明します。アプリケーションサーバを動作させるために必要なディスクおよびメモリの容量を算出するときの参考にしてください。

(1)　スレッド数

スレッド数の計算式を次に示します。(a)と(b)の合計が，バッチサーバが使用するスレッド数です。

(a)　基本のスレッド数

最大スレッド数 = 68 + A + B + C + D + E + F + G + H + I

（凡例）

- A：CORBA ネーミングサービスのスレッド数（=クライアントと CORBA ネーミングサービス間のコネクション数×2 +同時受け付けリクエスト数+初期化時に生成されるスレッド数（vbroker.agent.enableLocator の値が true の場合は 6，false の場合は 4）+ 1）

  ただし，CORBA ネーミングサービスをインプロセスで起動（usrconf.properties の ejbserver.naming.startupMode キーに inprocess を指定）した場合だけ加算する。

  CORBA ネーミングサービスのスレッド数の見積もりについては，「5.2.1(3)　CORBA ネーミングサービス（インプロセス起動時）のスレッド数の見積もり」を参照してください。
- B：同時に使用する最大のデータベースコネクション数

  コネクションプーリング機能を使用している場合は，最大コネクションプール数（Connector 属性ファイルで指定する MaxPoolSize の値。複数のリソースアダプタがある場合は合計）となる。

  コネクションプーリング機能を使用していない場合は，最大同時リクエスト数や 1 リクエストで使用するコネクション数から求める（1 リクエストで 1 コネクションを使用する場合，最大同時リクエスト数となる）。
- C：JTA トランザクションを使用する場合，最大同時実行トランザクション数（トランザクションタイムアウトが発生したトランザクション一つにつき，1 スレッドを使用する。1 リクエストが 1 トランザクションの場合，最大同時リクエスト数になる）
- D：最大コネクションプール数（複数のリソースアダプタがある場合は合計※）×2

  注※　Connector 属性ファイルで指定する MaxPoolSize の値
- E：コネクションプーリング機能を使用しているリソースアダプタ数

- F：グローバルトランザクションの決着処理，リカバリ処理で使用するスレッド数（グローバルトランザクションを使用している場合，16 を加算する）
- G：管理用サーバへの同時接続クライアント数（ただし，管理用サーバへの同時接続クライアント数が 5 以下の場合は 5，100 以上の場合は 100 を指定する）
- H：リプライ受信専用スレッドを管理するスレッドを起動する設定（vbroker.ce.iiop.ccm.htc.threadStarter=true）の場合，5 を加算する。
- I：タイムアウト発生時のコネクションのクローズを抑止する設定（vbroker.ce.iiop.ccm.htc.readerPerConnection=true）の場合，次の値を加算する。
  （リモート呼び出し先の EJB がある J2EE サーバの数＋ 1）×2
  CTM を使用している場合は，さらに次の値を加算する。
  - J2EE アプリケーション単位のスケジューリングをしている場合
    開始している J2EE アプリケーション数＋ 1
  - Stateless Session Bean 単位のスケジューリングをしている場合
    スケジューリング対象の Stateless Session Bean の数＋ 1

(b) JavaVM のオプション指定に応じて使用するスレッド数

JavaVM のオプション指定に応じて，次の計算式で算出してください。A は，-XX:+UseG1GC オプションを指定している場合だけ加算します。B は，-XX:+HitachiUseExplicitMemory オプションを指定した場合だけ加算します。C は，-XX:+UseZGC オプションを指定している場合だけ加算します。

最大スレッド数 ＝ A ＋ B ＋ C

（凡例）

- A：G1GC で使用するスレッド数（a ＋ b ＋ 2）※
- B：明示管理ヒープ機能で使用するスレッド数（論理 CPU 数。ただし，論理プロセッサ数が 8 以上の場合は 8。なお，J2EE サーバ起動時の論理 CPU 数によって決定されるため，起動後に論理 CPU の数を変更してもスレッド数は変化しない）
- C：ZGC で使用するスレッド数（a ＋ b ＋ 5）※

注※

凡例は次のとおりです。

a：-XX:ParalleGCThreads オプションに指定した値。このオプションの指定を省略した場合は，論理 CPU 数を基にした-XX:ParalleGCThreads オプションのデフォルト値。なお，J2EE サーバ起動時の論理 CPU 数によって決定されるため，起動後に論理 CPU の数を変更してもスレッド数は変化しない。

b：-XX:ConcGCThreads オプションに指定した値。このオプションの指定を省略した場合は，論理 CPU 数を基にした-XX:ConcGCThreads オプションのデフォルト値。なお，J2EE サーバ起動時の論理 CPU 数によって決定されるため，起動後に論理 CPU の数を変更してもスレッド数は変化しない。

JavaVM のオプションについては，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の次の個所を参照してください。

- 14.5　Application Server で指定できる Java HotSpot VM のオプション
- -XX:[+|-]HitachiUseExplicitMemory（明示管理ヒープ機能オプション）

(2)　ファイルディスクリプタ数

ファイルディスクリプタ数の計算式を次に示します。

最大ファイルディスクリプタ数 ＝ （149 ＋ A ＋ B×2 ＋ C） / 0.8

（凡例）

- A：データベースコネクションの数
- B：リソースアダプタ数
- C：usrconf.cfg の add.class.path キーに指定した JAR ファイルの数

6.2.2　運用管理エージェントが使用するリソースの見積もり

運用管理エージェントが使用するリソースの見積もりについては，J2EE アプリケーション実行基盤の場合と同じため，「5.2.2　運用管理エージェントが使用するリソースの見積もり」を参照してください。

6.2.3　パフォーマンストレーサが使用するリソースの見積もり

パフォーマンストレーサが使用するリソースの見積もりについては，J2EE アプリケーション実行基盤の場合と同じため，「5.2.3　パフォーマンストレーサが使用するリソースの見積もり」を参照してください。

6.2.4　CTM が使用するリソースの見積もり

CTM が使用するリソースの見積もりについては，J2EE アプリケーション実行基盤の場合と同じため，「5.2.4　CTM が使用するリソースの見積もり」を参照してください。

6.3　仮想メモリの使用量の見積もり

ここでは，仮想メモリの使用量の見積もり方法について説明します。なお，仮想メモリの使用量の計算式に使用している JavaVM 起動時のオプションの詳細については，「7.2.6　SerialGC 使用時の JavaVM で使用するメモリ空間の構成と JavaVM オプション」を参照してください。

また，明示管理ヒープ機能を使用する場合は，ここで説明する内容のほか，明示管理ヒープ機能で使用するメモリサイズの見積もりが必要です。明示管理ヒープ機能で使用するメモリサイズの見積もりについては，「7.11　Explicit ヒープのチューニング」を参照してください。

6.3.1　仮想メモリの使用量の計算式

仮想メモリの使用量（単位：メガバイト）の計算式を次に示します。

バッチサーバの仮想メモリの使用量＝A＋B＋C＋（D＋22）×E＋F＋G

（凡例）

- A：Java ヒープサイズ
  初期値は，JavaVM 起動時のオプション-Xms に指定した値です。この値は，J2EE サーバ起動中に，最大で JavaVM 起動時のオプション-Xmx に指定した値まで拡張されます。なお，ZGC を使用する場合は，仮想メモリを多く必要とします。ZGC を使用する場合は，この値は-Xmx に指定した値の 48 倍としてください。これは物理メモリの実際の使用サイズには影響しません。詳細については「7.20 ZGC 使用時の注意事項（JDK17 以降の場合）」を参照してください。
- B：Metaspace 領域サイズ
  初期値は，JavaVM が割り当てた値です。この値は，J2EE サーバ起動中に，最大で JavaVM 起動時のオプション-XX:MaxMetaspaceSize に指定した値まで拡張されます。

  監査ログを使用する場合

  監査ログを使用する場合は，値に 1 メガバイトを加算してください。
- C：ネイティブプログラム使用領域サイズ
  OS ごとに使用する値が異なります。ネイティブプログラム使用領域サイズの OS ごとの値について，次の表に示します。

表 6-7　ネイティブプログラム使用領域サイズの値一覧

OS 種別	使用領域の値 （単位：メガバイト）
Windows	300
AIX	400
Linux	600

監査ログを使用する場合

監査ログを使用する場合は，OS ごとのネイティブプログラム使用領域サイズの値に，監査ログで使用する値を加算する必要があります。監査ログで使用する値を算出するには，auditlog.raslog.message.filesize キー，および auditlog.raslog.exception.filesize キーを使用します。これらのキーは，auditlog.properties（監査ログ定義ファイル）で指定するキーです。これらのキーの詳細については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「11.2.1　監査ログ定義ファイル」を参照してください。

監査ログで使用する値（単位：メガバイト）の計算式を次に示します。

監査ログで使用する値 ＝

8 ＋（auditlog.raslog.message.filesize の指定値 / $1,024^2$）※＋（auditlog.raslog.exception.filesize の指定値 / $1,024^2$）※

注※　指定値の単位はバイトです。仮想メモリの使用量の値の単位はメガバイトであるため，$1,024^2$ で割る必要があります。

- D：バッチサーバのスレッド数

  バッチサーバが使用するスレッド数です。バッチサーバが使用するスレッド数については，「6.2.1 バッチサーバが使用するリソースの見積もり」を参照してください。

- E：スタック領域サイズ

  JavaVM 起動時のオプション-Xss に指定した値です。

- F：Explicit ヒープサイズ

  明示管理ヒープ機能で使用される Explicit ヒープのサイズです。この値は，J2EE サーバ起動中に最大で-XX:HitachiExplicitHeapMaxSize に指定した値まで拡張されます。なお，デフォルトでは Explicit ヒープは無効になります。

- G：コードキャッシュ領域の最大サイズ

  JavaVM 起動時のオプション-XX:ReservedCodeCacheSize に指定した値です。

6.3.2　仮想メモリの使用量を計算する場合の注意事項

- ネイティブプログラム使用領域サイズの値は変動します。値が変動するのは次のような場合です。
  - ORB のトレースサイズを変更した場合
  - JDBC ドライバの使用領域サイズの値を変更した場合
  - 使用する製品のネイティブライブラリ使用領域サイズの値を変更した場合

  このような場合，製品ごとのメモリ使用量をネイティブプログラム使用領域サイズの値に加算してください。製品ごとのメモリ使用量は，使用する製品のドキュメントに従って算出してください。

- バッチサーバの標準構成で使用する仮想メモリの使用量は，使用するソフトウェア製品の影響で，見積もり値よりも大きくなる場合があります。その場合の増加分については，仮想メモリの使用量の値に加算してください。増加分のメモリ使用量は，使用するソフトウェア製品のドキュメントに従い，製品ごとのメモリ使用量を算出してください。

- Linux の場合，メモリのスワップファイルが不足すると，システムがスローダウンします。仮想メモリの上限値を，実メモリサイズ分指定することをお勧めします。

7 JavaVM のメモリチューニング

システムの処理性能を高めるには，基盤となる JavaVM 自体のチューニングを適切に実施する必要があります。製品の JavaVM（以降，JavaVM と呼びます）では，2 種類のメモリ空間を管理しています。

この章では，GC と JavaVM でのメモリ管理，および Java ヒープと Explicit ヒープのチューニングについて説明します。

7.1 GC と JavaVM のメモリ管理の概要

JavaVM のチューニングの目的は，システムの処理性能の向上です。特に，GC の仕組みを踏まえ，適切なメモリ管理ができるようにチューニングすることで，システムの処理性能が向上します。GC の挙動はメモリ管理方式によって変わるため，システムの要件に合わせて適切なメモリ管理方式を選択してください。アプリケーションサーバでは次のメモリ管理方式を選択できます。

表 7–1 各メモリ管理方式の特徴

項番	メモリ管理方式	特徴
1	SerialGC	• スループットを重視するシステムに適している。 • スループットが高い。 • GC には長い時間かかる GC(FullGC)と短い時間で終わる GC(CopyGC)がある。GC の時間を制御できない。 • メモリサイズのチューニングをすることで，FullGC の発生を抑止できる。
2	SerialGC と明示管理ヒープ機能の組み合わせ※1	• セッションを使用した一般的な Web フロントシステムに適している。 • スループットが高い。 • GC には長い時間かかる GC(FullGC)と短い時間で終わる GC(CopyGC)がある。GC の時間を制御できない。 • セッションを利用したシステムでは，メモリサイズのチューニングに加えて，セッションを Explicit ヒープで管理することで FullGC を抑止できる。
3	G1GC	• 大規模なメモリを使用するシステムやレスポンスを重視するシステムに適している。 • 項番 1 や項番 2 の方式と比較してスループットが低い。 • GC には長い時間かかる GC(FullGC)と短い時間で終わる GC(YoungGC，MixedGC)がある。YoungGC と MixedGC の GC の時間を制御できる。 • メモリサイズのチューニングに加え，GC を行うスレッド数を増やすことで，FullGC の発生を抑止できる。
4	ZGC※2	• 低レイテンシを重視するシステムに適している。一部を除き，GC 処理をアプリケーションの実行と並行で行うため，アプリケーションの停止時間が非常に短い。これは数テラバイトの非常に大きなヒープを使用するようなシステムでも同様である。 • 項番 1，3 の方式と比較してスループットが低い。 • 項番 1，3 の方式のように複数の種類の GC で構成されていないため，ZGC サイクルと呼ばれる一連の処理が毎回繰り返される。 • メモリサイズの単純なチューニングだけで低レイテンシを実現できる。

注※1

JDK11 以前の場合にだけ選択できます。

注※2

JDK17 以降の場合にだけ選択できます。

SerialGC については 7.2〜7.10 を，明示管理ヒープ機能については 7.11〜7.14 を，G1GC については 7.15〜7.16 を，ZGC については 7.17〜7.20 を参照してください。

7.2　SerialGC の仕組み

ここでは，SerialGC の仕組みについて説明します。

7.2.1　SerialGC の概要

GC は，プログラムが使用し終わったメモリ領域を自動的に回収して，ほかのプログラムが利用できるようにするための技術です。

GC の実行中は，プログラムの処理が停止します。このため，GC を適切に実行できるかどうかが，システムの処理性能に大きく影響します。

プログラムの中で new によって作成された Java オブジェクトは，JavaVM が管理するメモリ領域に格納されます。Java オブジェクトが作成されてから不要になるまでの期間を，**Java オブジェクトの寿命**といいます。

Java オブジェクトには，寿命の短いオブジェクトと寿命の長いオブジェクトがあります。サーバサイドで動作する Java アプリケーションの場合，リクエストやレスポンス，トランザクション管理などで，多くの Java オブジェクトが作成されます。これらの Java オブジェクトは，その処理が終わると不要になる，寿命が短いオブジェクトです。一方，アプリケーションの動作中使用され続ける Java オブジェクトは，寿命が長いオブジェクトです。

効果的な GC を実行するためには，寿命の短いオブジェクトに対して GC を実行して，効率良くメモリ領域を回収することが必要です。また，繰り返し使用される寿命の長いオブジェクトに対する不要な GC を抑止することが，システムの処理性能の低下防止につながります。これを実現するのが，**世代別 GC** です。

世代別 GC では，Java オブジェクトを，寿命が短いオブジェクトが格納される New 領域と，寿命が長いオブジェクトが格納される Tenured 領域に分けて管理します。New 領域はさらに，new によって作成されたばかりのオブジェクトが格納される Eden 領域と，1 回以上の GC の対象になり，回収されなかったオブジェクトが格納される Survivor 領域に分けられます。New 領域内で一定回数以上の GC の対象になった Java オブジェクトは，長期間必要な Java オブジェクトと判断され，Tenured 領域に移動します。

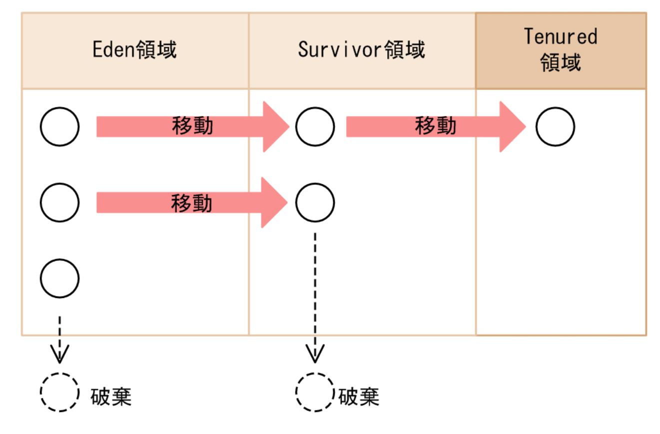
世代別 GC で管理するメモリ空間と Java オブジェクトの概要を次の図に示します。

図 7-1　世代別 GC で管理するメモリ空間と Java オブジェクトの概要

n回数以上のGCが実行される間
使用されていたオブジェクト

1回以上n回未満のGCが実行される間
使用されていたオブジェクト

1回目のGC実行時に使用されていな
かったオブジェクト



（凡例）
○ ：Javaオブジェクト
GC：ガーベージコレクション

SerialGC の世代別 GC で実行される GC には，次の 2 種類があります。

- CopyGC
  Eden 領域と Survivor 領域を対象にした GC です。Java オブジェクトの作成によって，Eden 領域を使い切ると発生します。

> **注意事項**
>
> アプリケーションサーバでは，SerialGC と G1GC（UseG1GC）が選択できます。ほかの ParallelGC，ConcurrentGC，IncrementalGC は使用できません。

- FullGC
  Tenured 領域も含む，JavaVM 固有領域全体を対象にした GC です。Tenured 領域を一定サイズまで使うと発生します。

一般的に，CopyGC の方が，FullGC よりも短い時間で処理できます。

次に，GC の処理について，ある Java オブジェクト（オブジェクト A）を例にして説明します。

Eden 領域で実行される処理

オブジェクト A の作成後，1 回目の CopyGC が実行された時点で使用されていない場合，オブジェクト A は破棄されます。

1 回目の CopyGC が実行された時点で使用されていた場合，オブジェクト A は Eden 領域から Survivor 領域に移動します。

Survivor 領域で実行される処理

Survivor 領域に移動したオブジェクト A は，その後何回か CopyGC が実行されると，Survivor 領域から Tenured 領域に移動します。移動する回数のしきい値は，JavaVM オプションや Java ヒープの利用状況によって異なります。図 7-1 では，しきい値を n 回としています。

Survivor 領域への移動後，n 回目未満の CopyGC が実行された時点でオブジェクト A が使用されていなかった場合，オブジェクト A はその CopyGC で破棄されます。

Tenured 領域で実行される処理

オブジェクト A が Tenured 領域に移動した場合，その後の CopyGC でオブジェクト A が破棄されることはありません。CopyGC は，Eden 領域と Survivor 領域だけを対象としているためです。

このようにしてオブジェクトが移動することで，Tenured 領域の使用サイズは増加します。Tenured 領域を一定サイズまで使うと，FullGC が発生します。

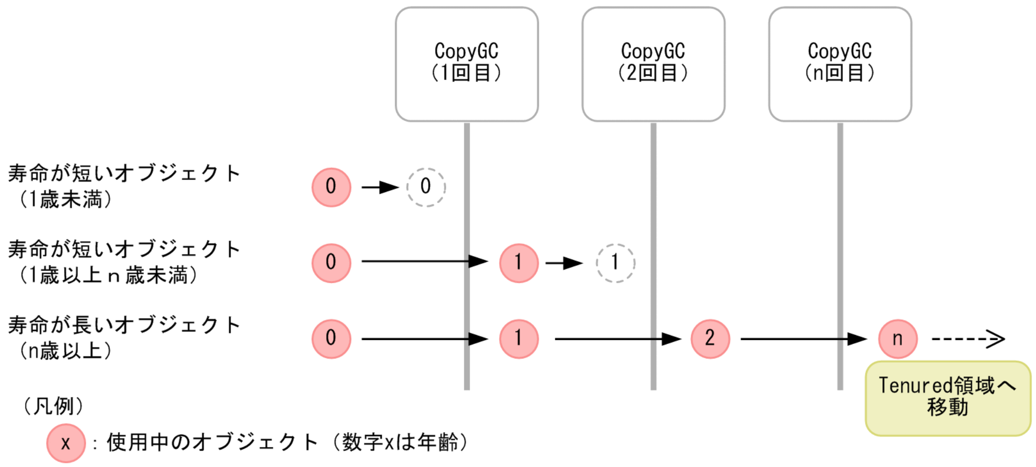
JavaVM のチューニングでは，JavaVM オプションでそれぞれのメモリ空間のサイズや割合を適切に設定することで，不要なオブジェクトが Tenured 領域に移動することを抑止します。これによって，FullGC が頻発することを防ぎます。

7.2.2　オブジェクトの寿命と年齢の関係

オブジェクトが CopyGC の対象になった回数を**オブジェクトの年齢**といいます。

オブジェクトの寿命と年齢の関係を次の図に示します。

図 7-2　オブジェクトの寿命と年齢の関係



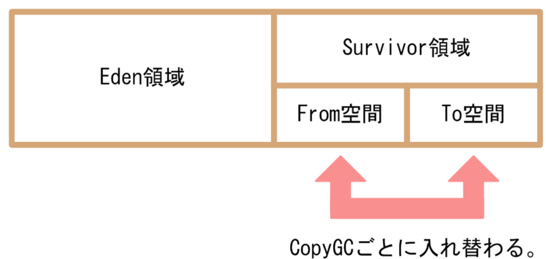
アプリケーションが開始して初期化処理が完了したあとで，何度かの CopyGC が実行されると，長期間必要になる寿命の長いオブジェクトは Tenured 領域に移動します。このため，アプリケーションの開始後しばらくすると，Java ヒープの状態は安定し，作成される Java オブジェクトとしては，寿命が短いオブジェクトが多くなります。特に，New 領域のチューニングが適切にできている場合，Java ヒープが安定したあとの大半のオブジェクトは，1 回目の CopyGC で回収される，寿命が短いオブジェクトになります。

7.2.3　CopyGC の仕組み

JavaVM では，CopyGC の対象になる New 領域のメモリ空間を，Eden 領域，Survivor 領域に分けて管理します。さらに，Survivor 領域は，From 空間と To 空間に分けられます。From 空間と To 空間は，同じメモリサイズです。

New 領域の構成を次の図に示します。

図 7-3　New 領域の構成



Eden 領域は，new によって作成されたオブジェクトが最初に格納される領域です。プログラムで new が実行されると，Eden 領域のメモリが確保されます。

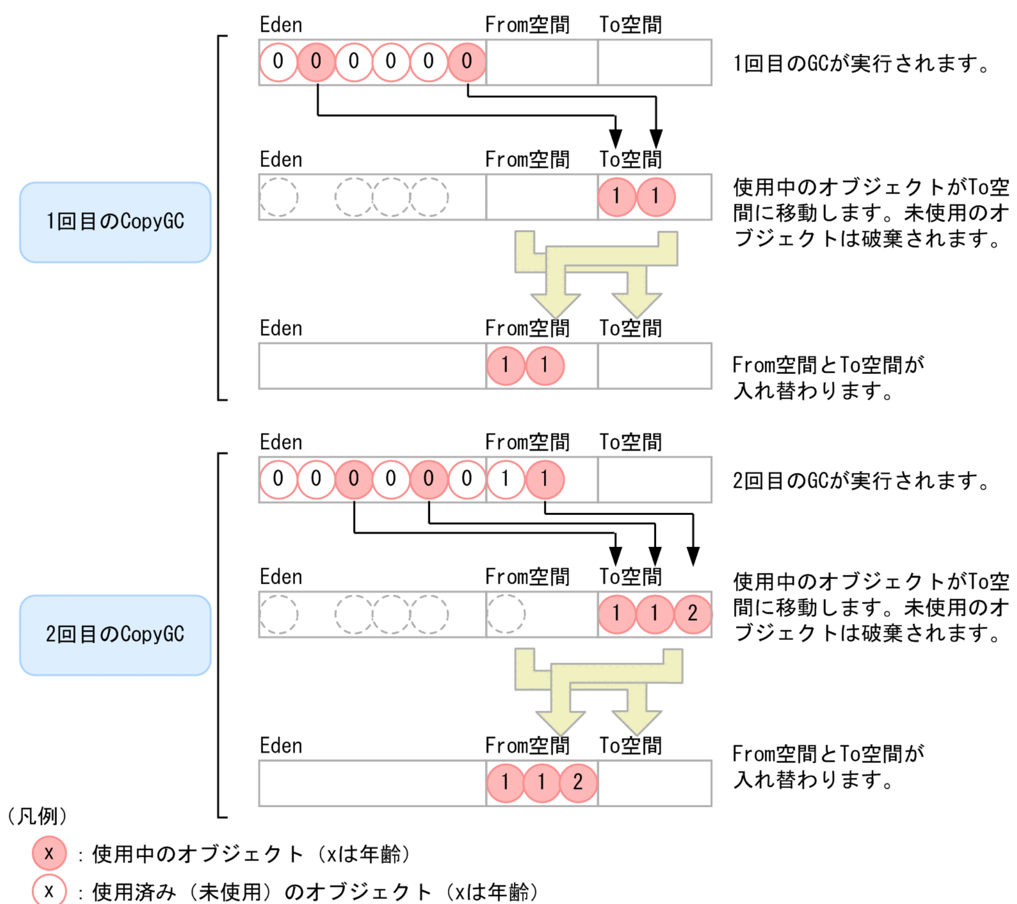
Eden 領域がいっぱいになると，CopyGC が実行されます。CopyGC では，次の処理が実行されます。

1. Eden 領域および Survivor 領域の From 空間にある Java オブジェクトのうち，使用中の Java オブジェクトが，Survivor 領域の To 空間にコピーされます。使用されていない Java オブジェクトは破棄されます。
2. Survivor 領域の To 空間と From 空間が入れ替わります。

この結果，Eden 領域と To 空間は空になり，使用中のオブジェクトは From 空間に存在することになります。

CopyGC 実行時に発生するオブジェクトの移動を次の図に示します。

図 7-4　CopyGC 実行時に発生するオブジェクトの移動

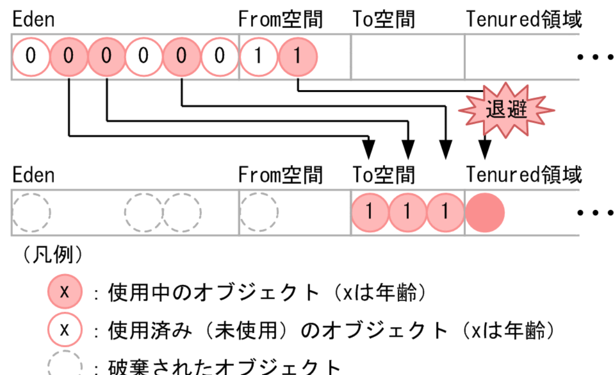


このようにして，使用中のオブジェクトは，CopyGC が発生するたびに，From 空間と To 空間を行ったり来たりします。ただし，寿命の長いオブジェクトを行き来させ続けると，コピー処理の負荷などが問題になります。これを防ぐために，From 空間と To 空間で Java オブジェクトを入れ替える回数にしきい値を設定して，年齢がしきい値に達した Java オブジェクトは Tenured 領域に移動させるようにします。

7.2.4　オブジェクトの退避

年齢がしきい値に達していない Java オブジェクトが Tenured 領域に移動することを，退避といいます。退避は，CopyGC 実行時に Eden 領域および From 空間で使用中のオブジェクトが多くなり，移動先である To 空間のメモリサイズが不足する場合に発生します。この場合，To 空間に移動できなかったオブジェクトが，Tenured 領域に移動します。

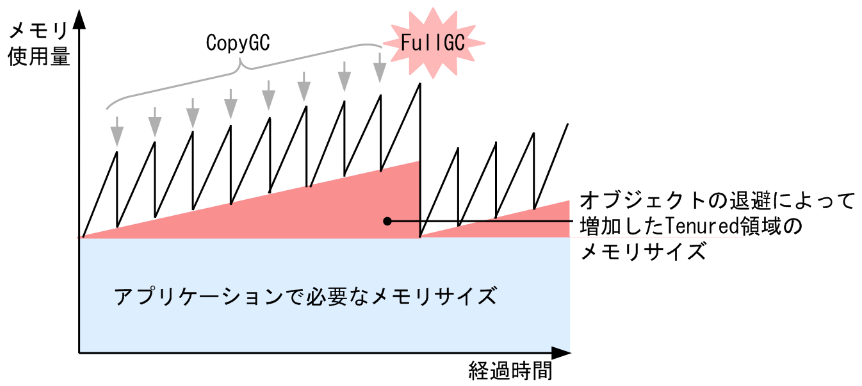
図 7-5　オブジェクトの退避



オブジェクトの退避が発生した場合，Tenured 領域に本来格納されないはずの寿命の短いオブジェクトが格納されます。これが繰り返されると，CopyGC で回収されるはずのオブジェクトがメモリ空間に残っていくため，Java ヒープのメモリ使用量が増加していき，最終的には FullGC が発生します。

オブジェクトの退避が発生した場合のメモリ使用量の変化について，次の図に示します。

図 7-6　オブジェクトの退避が発生した場合のメモリ使用量の変化



FullGC では，システムが数秒から数十秒停止することがあります。

したがって，メモリ空間の構成とメモリサイズを検討するときには，オブジェクトの退避が発生しないように，Eden 領域と Survivor 領域のバランスを検討する必要があります。

7.2.5　GC 対象外の領域（明示管理ヒープ機能を使用した Explicit ヒープ領域の利用）

JavaVM では，Eden 領域，Survivor 領域，および Tenured 領域以外の領域として Explicit ヒープという領域を利用できます。Explicit ヒープ領域は GC 対象外の領域です。

Explicit ヒープ領域に格納するオブジェクトは，自動配置設定ファイル，および明示管理ヒープ機能 API を使って指定します。指定されたオブジェクトは，Survivor 領域から Tenured 領域へ移動する際に Explicit ヒープ領域へ移動します。CopyGC で回収の対象にならない長寿命オブジェクトを指定することで，

Tenured 領域のメモリ使用量を少なくし，FullGC の発生を抑止します。また，明示管理ヒープ機能の自動配置設定ファイルや，明示管理ヒープ機能 API を使って，指定したオブジェクトを Explicit ヒープ領域に生成することもできます。

明示管理ヒープ機能の詳細については，マニュアル「アプリケーションサーバ 機能解説 拡張編」の「7. 明示管理ヒープ機能を使用した FullGC の抑止」を参照してください。

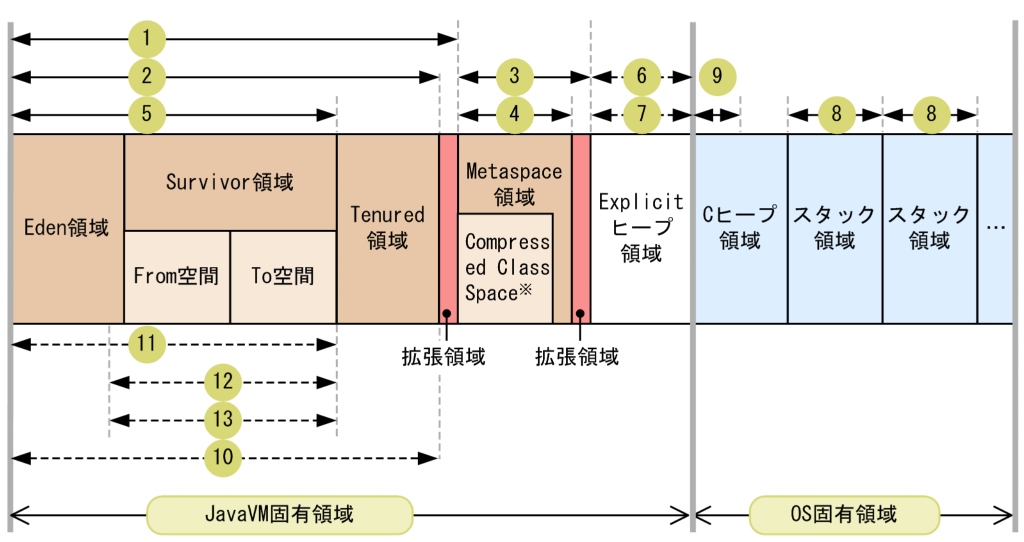
7.2.6 SerialGC 使用時の JavaVM で使用するメモリ空間の構成と JavaVM オプション

ここでは，SerialGC 使用時の JavaVM で使用するメモリ空間の構成と，JavaVM オプションについて説明します。

JavaVM では，**JavaVM 固有領域**と **OS 固有領域**という，2 種類のメモリ空間を使用します。

JavaVM で使用するメモリ空間の構成を次の図に示します。なお，図中の番号は，表 7-2 の項番と対応しています。

図 7-7 JavaVM で使用するメモリ空間の構成



（凡例）

←→ ：サイズを指定するオプションの対象になる範囲

←- -→ ：割合または回数を指定するオプションの対象になる範囲

注※

圧縮オブジェクトポインタ機能が有効の場合だけ割り当てられる領域

それぞれの領域について説明します。なお，Eden 領域，Survivor 領域，および Tenured 領域を合わせた領域を，**Java ヒープ**といいます。

Eden 領域

new によって作成された Java オブジェクトが最初に格納される領域です。

Survivor 領域

New 領域に格納されていた Java オブジェクトのうち，CopyGC 実行時に破棄されなかった Java オブジェクトが格納される領域です。Survivor 領域は，From 空間と To 空間で構成されます。From 空間と To 空間のサイズは同じです。

Tenured 領域

長期間必要であると判断された Java オブジェクトが格納される領域です。Survivor 領域で指定回数を超えて CopyGC の実行対象となり，破棄されなかった Java オブジェクトが，この領域に移動されます。

Metaspace 領域

ロードされた class などの情報が格納される領域です。

ただし，圧縮オブジェクトポインタ機能が有効な場合，Metaspace 領域内に Compressed Class Space という領域が作成されます。この領域に，Java ヒープ内のオブジェクトから参照されるクラス情報が配置されます。そして，それ以外のメソッド情報などが Compressed Class Space 以外の Metaspace 領域に配置されます。圧縮オブジェクトポインタ機能については，マニュアル「アプリケーションサーバ 機能解説 保守／移行編」の「9.18　圧縮オブジェクトポインタ機能」を参照してください。

Explicit ヒープ領域

FullGC の対象外になる Java オブジェクトが格納される領域です。Explicit ヒープ領域は JavaVM 独自のメモリ空間で，明示管理ヒープ機能を利用する場合だけ確保されます。

C ヒープ領域

JavaVM 自身が使用する領域です。JNI で呼び出されたネイティブライブラリでも使用されます。

> **参考**
>
> **C ヒープ領域の最大サイズについて（AIX の場合）**
>
> AIX の場合，JNI を使用して JavaVM を起動するプログラムを実行するときは，次のどれかの方法で C ヒープ領域の最大サイズを設定してください。どれも設定していない場合は，シェルの datasize リソースのデフォルト値が C ヒープ領域の最大サイズとなります。
>
> - シェルで datasize リソースの値に C ヒープ領域のサイズを指定してから，プログラムを実行してください。
>
>   csh（C シェル）の場合：limit datasize <C ヒープ領域のサイズ>
>
>   sh（標準シェル）および ksh の場合：ulimit -d <C ヒープ領域のサイズ>
>
> - プログラム生成時のリンケージオプションに，「-bmaxdata:<C ヒープ領域のサイズ>」を指定してください。
>
> - 環境変数 LDR_CNTRL で MAXDATA 値を指定してから，プログラムを実行してください。
>
>   csh（C シェル）の場合：setenv LDR_CNTRL MAXDATA=<C ヒープ領域のサイズ>
>
>   sh（標準シェル）および ksh の場合：export LDR_CNTRL=MAXDATA=<C ヒープ領域のサイズ>

- setrlimit()システムコールを使用して，C ヒープ領域のサイズを設定する処理をプログラム内で実装してください。

例えば，次のような領域があります。

コードキャッシュ領域

JIT コンパイルによって生成された JIT コンパイルコードが格納される領域です。

JavaVM は，呼び出し回数やループ回数が多い Java メソッドを JIT コンパイルして実行することで，処理の高速化を行います。

参考

コードキャッシュ領域の最大サイズについて

コードキャッシュ領域の最大サイズは，ReservedCodeCacheSize オプションに指定します。

ReservedCodeCacheSize オプションには，デフォルト値以上の値を指定してください。デフォルト値については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「14.4　Application Server で指定できる Java HotSpot VM のオプションのデフォルト値」を参照してください。

また，コードキャッシュ領域が枯渇していた場合，または枯渇するおそれがある場合は，コードキャッシュ領域の拡張を検討してください。

コードキャッシュ領域を拡張する場合は，次の点に注意してください。

- JIT コンパイルコードのサイズは計算で見積もることができません。そのため，Java アプリケーション実行環境でコードキャッシュ領域の使用量を実測し，システムが使用するコードキャッシュ領域（最大 2 メガバイト）の使用量を考慮した上で，コードキャッシュ領域の最大サイズを見積もってください。
- 仮想メモリの使用量の見積もりについては，「5.3　プロセスごとに使用するメモリの見積もり」または「6.3　仮想メモリの使用量の見積もり」を参照してください。

スタック領域

Java スレッドのスタック領域です。

参考

メインスレッドのスタック領域サイズについて

メインスレッドのスタック領域サイズを変更する場合は，-Xss に指定した値以上の値を設定してください。-Xss オプションでは，メインスレッドのスタック領域サイズは変更できません。J2EE サーバ起動時の J2EE アプリケーションの自動開始処理はメインスレッドで行うため，自動開始に失敗する場合があります。この場合，J2EE アプリケーションの自動開始を行わないように，-nostartapp オプションを指定して J2EE サーバを起動後，手動で J2EE アプリケーショ

ンを起動するか，Management アクションを使用して J2EE アプリケーションを起動してください。

それぞれの領域のサイズや割合などを指定する JavaVM オプションを次の表に示します。なお，表の項番は，図 7-7 と対応しています。

表 7-2　JavaVM メモリ空間のサイズや割合などを指定する JavaVM オプション

項番	オプション名	オプションの意味
1	-Xmx<size>	Java ヒープの最大サイズを設定します。
2	-Xms<size>	Java ヒープの初期サイズを設定します。
3	-XX:MaxMetaspaceSize=<size>	Metaspace 領域の最大サイズを設定します。
4	-XX:MetaspaceSize=<size>	Metaspace 領域の初期サイズを設定します。
5	-Xmn<size>	New 領域の初期値および最大値を設定します。
6	-XX:[+|-]HitachiAutoExplicitMemory	自動配置機能を使用する場合に，JavaVM を起動したタイミングで Explicit メモリブロックで使用するメモリを確保します。※
7	-XX:HitachiExplicitHeapMaxSize=<size>	Explicit ヒープ領域サイズの最大サイズを設定します。※
8	-Xss<size>	1 スタック領域の最大サイズを設定します。
9	-XX:ReservedCodeCacheSize=<size>	JavaVM が使用する領域のうち，コードキャッシュ領域の最大サイズを設定します。
10	-XX:NewRatio=<value>	New 領域に対する Tenured 領域の割合を設定します。 <value>が 2 の場合は，New 領域と Tenured 領域の割合が，1:2 になります。
11	-XX:SurvivorRatio=<value>	Survivor 領域の From 空間と To 空間に対する Eden 領域の割合を設定します。 <value>に 8 を設定した場合は，Eden 領域，From 空間，To 空間の割合が，8:1:1 になります。
12	-XX:TargetSurvivorRatio=<value>	CopyGC 実行後の Survivor 領域内で Java オブジェクトが占める割合の目標値を設定します。
13	-XX:MaxTenuringThreshold=<value>	CopyGC 実行時に，From 空間と To 空間で Java オブジェクトを入れ替える回数の最大値を設定します。 設定した回数を超えて入れ替え対象になった Java オブジェクトは，Tenured 領域に移動されます。

注　<size>の単位はバイトです。

注※　明示管理ヒープ機能を使用するための前提オプションが有効になっている必要があります。詳細は「7.11　Explicit ヒープのチューニング」を参照してください。

参考

- JavaVMオプションの設定の仕方

  JavaVMオプションは，次の個所に設定します。

表7-3　JavaVMオプションを設定する個所

対象	設定方法	設定個所
J2EEサーバ	Smart Composer機能	**定義ファイル** 簡易構築定義ファイル **設定対象** 論理J2EEサーバ（j2ee-server） **パラメタ名** add.jvm.arg
バッチサーバ	Smart Composer機能	**定義ファイル** 簡易構築定義ファイル **設定対象** 論理J2EEサーバ（j2ee-server） **パラメタ名** add.jvm.arg
EJBクライアントアプリケーション	ファイル編集	**定義ファイル** usrconf.cfg※ **パラメタ名** add.jvm.argキー

注※　cjclstartapコマンドを使用して開始する場合に有効になるファイルです。

ポイント

それぞれのオプションのデフォルト値は，OSによって異なります。オプションのデフォルト値については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「14.4 Application Serverで指定できるJava HotSpot VMのオプションのデフォルト値」を参照してください。

注意事項

Javaヒープ領域，Metaspace領域，Compressed Class Space，Cヒープ領域のどれかが不足するとOutOfMemoryが発生し，メモリ不足が解消されないかぎり正常に稼働できない状態が長く続くことになります。

これに対して，OutOfMemory発生時のシステムへの影響を小さくするために，次のオプションを使用できます。

- -XX:+HitachiOutOfMemoryAbort（OutOfMemory 発生時強制終了機能）
- -XX:+HitachiOutOfMemoryHandling（OutOfMemory ハンドリング機能）

OutOfMemory 発生時強制終了機能は，Java ヒープ不足や Metaspace 領域不足，Compressed Class Space 不足などが原因で OutOfMemory が発生した場合に，J2EE サーバを強制終了するための機能です。Java ヒープ領域，Metaspace 領域，Compressed Class Space，C ヒープ領域のメモリ不足によって OutOfMemory が発生したときに，J2EE サーバを強制終了して，J2EE サーバを自動再起動します。これによって，J2EE サーバを正常に稼働できる状態に早期に回復できます。

OutOfMemory ハンドリング機能は，OutOfMemory 発生時強制終了機能を前提とする機能です。OutOfMemory 発生時強制終了機能を使用している場合でも，特定の条件に合致するときに限って J2EE サーバの実行を継続したい場合に使用します。

リクエスト処理でオブジェクトを大量に確保しようとした場合，巨大なオブジェクトを確保しようとした場合などに Java ヒープ不足が原因の OutOfMemory が発生したときに，J2EE サーバの実行を継続するようにしたいときには，OutOfMemory ハンドリング機能を使用してください。

オプションの詳細については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「-XX:[+|-]HitachiOutOfMemoryAbort（強制終了オプション）」および「-XX:[+|-]HitachiOutOfMemoryHandling（OutOfMemory ハンドリングオプション）」を参照してください。

- Metaspace 領域の特殊なチューニングの考え方

  Metaspace 領域の特性を活用したチューニングについて記載します。JDK7 までの Permanent 領域はオプションで設定した値のメモリを必ず使用していましたが，Metaspace 領域はオプションに設定された値のメモリを必ず使用するわけではありません。実際には，実行に必要なサイズのメモリしか使用しません。

  (a) Metaspace 領域の OutOfMemoryError の発生リスクを下げる

  見積もり時に想定していなかった Metaspace 領域の使用量の増加によって，OutOfMemoryError が発生するリスクを下げるためには，見積もり値にバッファを持たせた値を-XX:MaxMetaspaceSize と-XX:CompressedClassSpaceSize に設定してください。Metaspace 領域の使用量がこの値を超えるまでは Metaspace 領域の OutOfMemoryError は発生しません。

  (b) Metaspace 領域に起因する FullGC の発生リスクを下げる

  見積もり時に想定していなかった Metaspace 領域の使用量の増加によって，Metaspace 領域に起因する FullGC が発生するリスクを下げるためには，見積もり値にバッファを持たせた値を-XX:MetaspaceSize，-XX:MaxMetaspaceSize，-XX:CompressedClassSpaceSize に設定してください。Metaspace 領域の使用量がこの値を超えるまでは Metaspace 領域に起因する FullGC は発生しません。

7.2.7 GC の発生とメモリ空間の関係

GC は，メモリ空間の使用状況に応じて発生します。ここでは，GC が発生するタイミングについて説明します。

> **注意事項**
>
> RMI を使ってリモートオブジェクトの登録や参照をすると，定期的に GC が発生することがあります。GC が発生するタイミングは，次のどちらかのプロパティにミリ秒単位で指定できます。デフォルトは 3600000 ミリ秒（1 時間）です。
>
> - sun.rmi.dgc.client.gcInterval プロパティ
> - sun.rmi.dgc.server.gcInterval プロパティ
>
> GC が発生するタイミングを変更する場合は，どちらかのプロパティにミリ秒単位で任意の値を指定してください。なお，指定できる値の範囲は，1～Long.MAX_VALUE-1 です。詳細は，該当ページ（http://download.oracle.com/javase/6/docs/technotes/guides/rmi/sunrmiproperties.html）を参照してください。

(1) CopyGC が発生するタイミング

CopyGC は，次のタイミングで発生します。

1. Eden 領域へのアロケーションで空き領域が不足した場合
2. jheapprof コマンドに-copygc オプションを指定して実行した場合

(2) FullGC が発生するタイミング

FullGC は，次のタイミングで発生します。

1. New 領域（Eden 領域と Survivor 領域の合計）で使用しているメモリサイズが Tenured 領域の最大値に対する未使用メモリサイズを上回っている状態の時に，Eden 領域へのアロケーションで空き領域が不足した場合

   > **注意事項**
   >
   > 上記のタイミングで必ず FullGC が発生するわけではありません。この状況になった場合，JavaVM は，次の値を基に FullGC を発生させるかを決定しています。
   >
   > - 過去に発生した CopyGC で New 領域から Tenured 領域に移動したオブジェクト量に対して，その CopyGC が発生した時点によって重みを付けて算出した平均値

2. CopyGC の実施の結果，New 領域（Eden 領域と Survivor 領域の合計）から Tenured 領域へのオブジェクトの移動に失敗した場合

3. New 領域と Tenured 領域のそれぞれの未使用メモリサイズを上回るメモリサイズ（Java オブジェクトのサイズ）のアロケーション要求があった場合
4. CopyGC の実施の結果，次のどちらかの状態になった場合
   - 確保済み Tenured 領域の未使用メモリサイズが 10,000 バイトを下回った場合
   - CopyGC 実施時の Tenured 領域へのオブジェクトの移動によって，確保済み Tenured 領域の拡張が発生した場合
5. java.lang.System.gc()メソッドが実行された場合
6. Metaspace 領域にアロケーションしたいメモリサイズが確保済み Metaspace 領域の未使用メモリサイズを上回る場合
7. javagc コマンドを実行した場合
8. jheapprof コマンドを実行した場合

JavaVM のチューニングでは，主に 1.と 3.の発生を抑えることを検討します。

参考

FullGC が発生した場合の要因は，拡張 verbosegc 情報を使用して確認できます。FullGC 発生時に要因を確認する方法については，「7.10　拡張 verbosegc 情報を使用した FullGC の要因の分析方法」を参照してください。

7.3　FullGC 発生を抑止するためのチューニングの概要

この節では，SerialGC 使用時に FullGC の発生を抑止するための Java ヒープ，および Explicit ヒープのチューニングの考え方，およびチューニング手順について説明します。

Java ヒープ，および Explicit ヒープのチューニング方法の詳細は 7.4～7.14 で説明します。

> **参考**
>
> CopyGC の違いによって，チューニング方法が変わることはありません。

7.3.1　チューニングの考え方

一般的に，CopyGC の方が，FullGC よりも短い時間で処理できます。

New 領域への CopyGC の実施によって適切にメモリを回収して，Java ヒープ全体を対象とする FullGC の発生をできるだけ抑止することが，システムの停止回数の削減や，処理性能向上につながります。これを実現するためには，JavaVM オプションでそれぞれのメモリ空間のサイズや割合を適切に設定することが必要です。

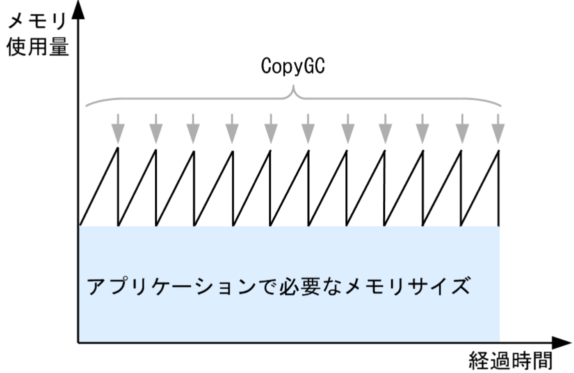
また，Tenured 領域に配置されるオブジェクトの一部を，明示管理ヒープ機能で管理する Explicit ヒープに配置することも，FullGC の発生抑止に効果的です。Explicit ヒープは，自動配置設定ファイルや明示管理ヒープ API を使ってアプリケーションから利用でき，J2EE サーバからも利用されます。明示管理ヒープ機能のデフォルトの設定では，自動解放機能が有効になっています。自動解放機能が有効の場合，JavaVM は，Explicit ヒープ領域のメモリを自動的に回収します。Explicit ヒープの利用のしかたによっては，自動解放処理が長時間終了しないことがあります。このため，長時間終了しない自動解放処理が発生しないように，明示管理ヒープ機能の適切な利用や，Explicit ヒープのメモリサイズの適切なチューニングを実施する必要があります。なお，自動解放処理については，マニュアル「アプリケーションサーバ 機能解説 拡張編」の「7.7　自動解放機能が有効な場合の Explicit メモリブロックの解放」を参照してください。

これらを踏まえ，JavaVM のチューニングは，次の 2 点を目的として実施します。

- FullGC をできるだけ発生させないこと
- FullGC の頻発を抑止した上で，不要な CopyGC を発生させないこと

理想的なメモリ使用量と経過時間の関係を次の図に示します。

図 7-8　理想的なメモリ使用量と経過時間の関係



この図の場合は，寿命の短いオブジェクトはすべて CopyGC によって回収できていて，オブジェクトの昇格や退避が発生しません。このため，CopyGC 実行後のメモリサイズが一定です。これによって，FullGC が発生しない，安定した状態での運用を実現できます。

JavaVM のチューニングでは，この状態を理想として，JavaVM の使用するメモリ空間の各領域で使用するメモリサイズを見積もってチューニングします。

ポイント

チューニングの目安

図 7-8 では，FullGC を一度も発生させない理想的な例を示しましたが，現実的には FullGC が 1 回発生する間に CopyGC が 10～20 回程度発生することを目安にして，チューニングを実施してください。

7.3.2　チューニング手順

FullGC の発生を抑止するための Java ヒープ，および Explicit ヒープのチューニング手順について説明します。

なお，手順(1)は必ず実施します。手順(2)～(4)は明示管理ヒープ機能を利用して Explicit ヒープを使う場合に，手順(1)のあとに続けて実施します。手順(5)以降については，それぞれの手順の説明を確認の上，必要に応じて実施してください。

Java ヒープ，および Explicit ヒープのチューニング手順を次の図に示します。

図 7-9　Java ヒープおよび Explicit ヒープのチューニング手順

チューニングの流れ

(1)　Javaヒープのチューニング

(2)　Explicitヒープのチューニング

(3)　稼働情報による見積もり結果の確認

(4)　Tenured領域のメモリサイズが増加するかどうかの確認

(5)　Tenured領域のメモリサイズ増加の要因となるオブジェクトの
Explicitヒープへの配置

(6)　Explicitヒープ全体のメモリサイズの見直し

(凡例)　：チューニングの流れ

(1)　Java ヒープのチューニング

Java ヒープのチューニングによって FullGC を抑止する方法を検討します。Java ヒープのチューニングについては，「7.4　Java ヒープのチューニング」を参照してください。

なお，Explicit ヒープ領域を利用しない場合は，Java ヒープのチューニングをした段階で J2EE サーバのテストを実施してください。Java ヒープのメモリを適切に見積もっても FullGC が頻発する場合は，Survivor 領域があふれているなど，チューニングに問題がないか確認してください。再度 Java ヒープのチューニングを見直しても問題が発生する場合は，明示管理ヒープ機能を使用した Explicit ヒープの利用を検討してください。Explicit ヒープを利用する場合は，手順(2)に進んでください。

(2)　Explicit ヒープのチューニング

明示管理ヒープ機能を使用して Explicit ヒープ領域を利用する場合に，Explicit ヒープ領域のメモリを見積もります。Explicit ヒープのチューニングについては，「7.11　Explicit ヒープのチューニング」を参照してください。

JDK11 以前で動作する J2EE サーバの場合，明示管理ヒープ機能はデフォルトで使用する設定になっています。また，HTTP セッションに関するオブジェクトといった Tenured 領域のメモリサイズ増加の要因となるオブジェクトが，Explicit ヒープに配置されるように設定されています。このため，J2EE サーバが配置するオブジェクトに必要な Explicit ヒープのメモリサイズを必ず見積もってください。明示管理ヒープ機能は，Explicit ヒープのメモリサイズを適切に見積もった上で使用しないと，効果が出ません。

(3)　稼働情報による見積もり結果の確認

明示管理ヒープ機能を使用する場合に，手順(1)および手順(2)で JavaVM のメモリを適切に見積もったあと，J2EE サーバのテストを実施します。テストで得た稼働情報を収集して Explicit ヒープの使用状況を確認します。稼働情報を基にした Explicit ヒープの見積もりについては「7.11.4　稼働情報による見積もり」を参照してください。

> **参考**
>
> J2EE サーバのテストを実施し，FullGC の発生回数が削減できない場合は，次の内容を確認してください。
>
> - Survivor 領域があふれているなど，Java ヒープのチューニングに問題がないか
>   手順(1)で見積もった値が適切かを見直します。
> - Explicit ヒープがあふれていないか
>   手順(2)で見積もった値が適切かを見直します。
> - Web アプリケーションの構成が適切か
>   Web アプリケーションの構成（アプリケーション内の API の使用方法など）によっては，HTTP セッションに関するオブジェクトに対して，明示管理ヒープ機能の効果が出ない場合があります。詳細は，マニュアル「アプリケーションサーバ 機能解説 拡張編」の「7.14　明示管理ヒープ機能使用時の注意事項」を参照してください。
>
> Java ヒープおよび Explicit ヒープのメモリサイズを見積もりし直しても，明示管理ヒープ機能の効果が出ないような場合に，自動解放処理が長時間化しているようなときは，「付録 A　HTTP セッションで利用する Explicit ヒープの効率的な利用」を参考にして，アプリケーションの設計を見直してください。Explicit ヒープの効率的な利用として，HTTP セッションに関するオブジェクトに対して明示管理ヒープ機能を効率良く適用するためには，アプリケーションの設計でどのような点を考慮すればよいかや，そのためのログの確認方法について説明しています。
>
> これらを確認しても問題が解決しない場合に，手順(4)に進んでください。

(4)　Tenured 領域のメモリサイズが増加するかどうかの確認

アプリケーションを開始して，Tenured 領域のメモリサイズを調査します。調査には，手順(3)で取得した稼働情報，または拡張 verbosegc 情報で取得した情報を使用します。拡張 verbosegc 情報の取得については，マニュアル「アプリケーションサーバ 機能解説 保守／移行編」の「5.7.2　拡張 verbosegc 情報の取得」を参照してください。

(5)　Tenured 領域のメモリサイズ増加の要因となるオブジェクトの Explicit ヒープへの配置

手順(3)と(4)で，FullGC の頻発と Tenured 領域のメモリサイズ増加が確認され，FullGC の抑止が必要となった場合は，Tenured 領域のメモリサイズ増加の要因となっているオブジェクトを Explicit ヒープに配置することを検討してください。ここで検討対象とするオブジェクトは，「J2EE サーバがデフォルトで Explicit ヒープに配置するオブジェクト」ではなく，それ以外の「Java アプリケーション上で作成しているオブジェクト」です。このオブジェクトを Java ヒープではなく Explicit ヒープに配置することで，FullGC の発生回数の削減が期待できます。Tenured 領域のメモリサイズ増加の要因となるオブジェクトを特定する方法については，「7.13.2　Tenured 領域利用済みサイズの増加原因が不明な場合」を参照してください。

Tenured 領域のメモリサイズ増加の要因となるオブジェクトを Explicit ヒープに配置する方法には，次の 2 種類があります。

- 明示管理ヒープ機能 API の利用
- 明示管理ヒープ機能の自動配置機能の利用

明示管理ヒープ機能 API の利用方法については，マニュアル「アプリケーションサーバ 機能解説 拡張編」の「7.12　明示管理ヒープ機能 API を使った Java プログラムの実装」を，明示管理ヒープ機能の自動配置機能の利用方法については，マニュアル「アプリケーションサーバ 機能解説 拡張編」の「7.13.2　自動配置設定ファイルを使った明示管理ヒープ機能の使用」を参照してください。

これらの機能を利用して Explicit ヒープに新たなオブジェクトを配置すると，Explicit ヒープのメモリサイズが増加します。このため，再度，Explicit ヒープのメモリサイズの見直しが必要になります。手順(6)に進んでください。

(6)　Explicit ヒープ全体のメモリサイズの見直し

手順(5)で修正したアプリケーションを動作させて，J2EE サーバとアプリケーションが使用する Explicit ヒープ全体のメモリサイズを見直します。見直し方法については，「7.12　アプリケーションで明示管理ヒープ機能を使用する場合のメモリサイズの見積もり」を参照してください。

ポイント

明示管理ヒープ機能によって FullGC の発生抑止の効果を得るためには，Explicit ヒープからオブジェクトがあふれないようにする必要があります。次の点を確認してください。

- Web アプリケーション内でセッションの破棄（invalidate メソッド呼び出し）および適切なセッションタイムアウトを設定していること。
- 適切なメモリサイズの Explicit ヒープ領域を Java ヒープ領域とは別に確保できること。

Explicit ヒープあふれが発生した場合の確認と対処については，「7.14.3　Explicit ヒープあふれが発生した場合の確認と対処」を参照してください。

以降では，Java ヒープのチューニング，および Explicit ヒープのチューニングについて説明します。なお，これ以外の注意事項については，マニュアル「アプリケーションサーバ 機能解説 拡張編」の「7.　明示管理ヒープ機能を使用した FullGC の抑止」を参照してください。

7.4 Java ヒープのチューニング

ここでは，SerialGC 使用時の Java ヒープのチューニングについて説明します。

7.4.1 Java ヒープのメモリサイズの見積もり

JavaVM のチューニングでは，JavaVM 固有領域の各領域のメモリサイズを適切に見積もる必要があります。

見積もりのポイントになるメモリサイズは次のとおりです。

- Java ヒープ全体のメモリサイズ
- Tenured 領域のメモリサイズ
- Survivor 領域のメモリサイズ
- Eden 領域のメモリサイズ

このほか，Metaspace 領域も，必要に応じて見積もります。

CopyGC 後の生存オブジェクトのサイズが Survivor 領域のサイズよりも大きい場合，Survivor 領域があふれ，1 度の CopyGC の実行で Tenured 領域に昇格するオブジェクトが発生します。また，Survivor 領域のサイズが小さい場合に，Survivor 領域の使用率が上がってくると，本来短寿命（CopyGC 間隔以下，または CopyGC 間隔の 1～2 倍程度の寿命）の Java オブジェクトが，数回の CopyGC の実行で Tenured 領域に昇格してしまいます。

ポイント

次に示す問題が確認できた場合は，Survivor 領域の不足が原因で FullGC が発生しています。なお，拡張 verbosegc 情報とは，J2EE サーバ起動時にオプションを設定しておくと，GC 発生時に JavaVM ログファイルに出力される情報です。

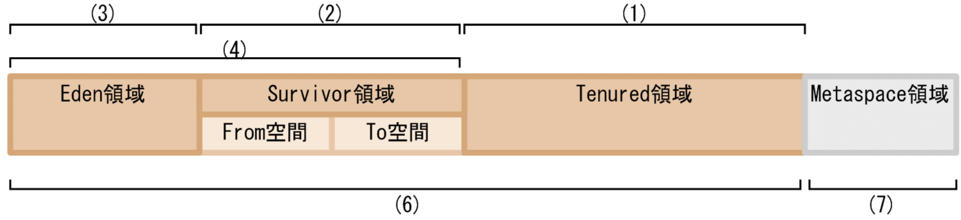
- Tenured 領域の増加要因が短寿命オブジェクトであると特定された場合
- 拡張 verbosegc 情報で，CopyGC 時の Survivor 領域あふれが確認できた場合
- -XX:+HitachiVerboseGCPrintTenuringDistribution 指定時に出力される拡張 verbosegc 情報で，オブジェクトの昇格年齢が常に 1 であることが観測できた場合

これらの問題を回避するためには，-XX:SurvivorRatio オプションの設定値を小さくして，Eden 領域と Survivor 領域の割合を最適化する必要があります。

このことを考慮し，見積もりでは，まず，Tenured 領域のメモリサイズと New 領域のメモリサイズを算出して，それらを基に Java ヒープ全体のメモリサイズを算出します。

メモリサイズを見積もる順序を次の図に示します。図中の番号の順番で見積もりを実施します。

図 7-10　メモリサイズを見積もる順序



見積もり手順を示します。なお，番号は図中の番号と対応しています。

1. Tenured 領域で使用するメモリサイズを見積もります。
   見積もり方法については，「7.5　Java ヒープ内の Tenured 領域のメモリサイズの見積もり」を参照してください。
2. Survivor 領域で使用するメモリサイズを見積もります。
   見積もり方法については，「7.6.1　Java ヒープ内の Survivor 領域のメモリサイズの見積もり」を参照してください。
3. Eden 領域で使用するメモリサイズを見積もります。
   見積もり方法については，「7.6.2　Java ヒープ内の Eden 領域のメモリサイズの見積もり」を参照してください。
4. 2.と 3.の合計として，New 領域全体のメモリサイズを算出します。
5. 一定期間存在するオブジェクトの扱いを検討して，必要なメモリサイズを Tenured 領域または New 領域のメモリサイズに追加します。
   検討方法については，「7.7　Java ヒープ内に一定期間存在するオブジェクトの扱いの検討」を参照してください。
6. 1.，4.および 5.の合計として，Java ヒープ全体のメモリサイズを算出します。
7. 必要に応じて Metaspace 領域のメモリサイズを見積もります。
   見積もり方法については，「7.9　Java ヒープ内の Metaspace 領域のメモリサイズの見積もり」を参照してください。

7.4.2　Java ヒープのメモリサイズの設定方法

見積もったメモリサイズは，「7.2.6　SerialGC 使用時の JavaVM で使用するメモリ空間の構成と JavaVM オプション」で説明したオプションで指定します。それぞれの領域のメモリサイズの設定方法を次に示します。

Java ヒープ全体のメモリサイズ

-Xmx<size>オプションで最大サイズを指定して，-Xms<size>オプションで初期サイズを指定します。

Tenured 領域のメモリサイズ

-XX:NewRatio=<value>オプションで，Java ヒープ全体に対する，Tenured 領域と New 領域の分割比を指定します。例えば，「-XX:NewRatio=5」とした場合には，-Xmx<size>オプションで指定したメモリサイズが，次のように分割されます。

```
New領域のメモリサイズ：Tenured領域のメモリサイズ = 1：5
```

Survivor 領域のメモリサイズと Eden 領域のメモリサイズ

-XX:SurvivorRatio=<value>オプションで，New 領域全体に対する，Survivor 領域と Eden 領域の分割比を指定します。なお，分割比は，Survivor 領域の From 空間および To 空間に対して Eden 領域を何倍確保するかの数値で指定します。例えば，「-XX:SurvivorRatio=2」とした場合には，-XX:NewRatio=<value>オプションで決まった New 領域のメモリサイズが，次のように分割されます。

```
Eden領域のメモリサイズ：From空間のメモリサイズ：To空間のメモリサイズ = 2：1：1
```

Metaspace 領域のメモリサイズ

-XX:MaxMetaspaceSize=<size>オプションで最大サイズを指定して，-XX:MetaspaceSize=<size>オプションで初期サイズを指定します。

7.4.3 Java ヒープのメモリサイズの使用状況の確認方法

それぞれのメモリサイズは，アプリケーションを実際に動作させて，メモリ使用量を測定しながらチューニングしていきます。アプリケーションサーバでは，usrconf.cfg ファイルに**-XX:+HitachiVerboseGC オプション**を指定して J2EE サーバを起動することで，GC 実行時の各領域の詳細なメモリサイズを**拡張 verbosegc 情報**として出力できます。この出力内容を基にチューニングを実施してください。

拡張 verbosegc 情報として出力できる主な内容を次に示します。

- GC の発生日時
- GC 種別
- GC 情報※1
- GC 経過時間
- Eden 情報※1
- Survivor 情報※1
- Tenured 情報※1
- Metaspace 領域情報※1
- GC 要因内容※2

注※1

GC 前後の使用領域長および領域サイズが出力されます。

注※2

-XX:+HitachiVerboseGCPrintCause オプションが指定されている場合に出力されます。

拡張 verbosegc 情報の出力例と FullGC の要因分析方法については，「7.10　拡張 verbosegc 情報を使用した FullGC の要因の分析方法」を参照してください。また，オプションについては，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「-XX:[+|-]HitachiVerboseGC（拡張 verbosegc 情報出力オプション)」を参照してください。

なお，アプリケーションサーバでは，javagc コマンドを使用して，任意のタイミングで FullGC を発生させることもできます。この場合，-v オプションを指定することで，拡張 verbosegc 情報と同じ内容を出力できます。javagc コマンドの詳細については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「javagc（GC の強制発生)」を参照してください。

7.5 Java ヒープ内の Tenured 領域のメモリサイズの見積もり

この節では，SerialGC 使用時の Tenured 領域のメモリサイズの見積もりについて説明します。

Tenured 領域のメモリサイズは，次のように見積もります。

```
Tenured領域のメモリサイズ
=アプリケーションで必要なメモリサイズ+New領域のメモリサイズ
```

ここでは，アプリケーションで必要なメモリサイズの算出方法について説明します。

また，見積もったメモリサイズに New 領域のメモリサイズを追加する理由についても説明します。

7.5.1 アプリケーションで必要なメモリサイズの算出

Tenured 領域のメモリサイズは，アプリケーションが最低限必要とするメモリサイズから見積もります。必要なメモリサイズが確保できない場合，OutOfMemoryError が発生して JavaVM が停止します。

アプリケーションが必要とするメモリサイズは，FullGC 実行時の拡張 verbosegc 情報で，FullGC 実行後に使用しているメモリサイズを確認することで判断できます。これは，FullGC 実行後に Java ヒープ全体から不要なオブジェクトをすべて削除した状態のメモリサイズが，アプリケーションが必要とするメモリサイズに近いと考えられるためです。

FullGC 実行時の拡張 verbosegc 情報の出力例を次に示します。

```
…
[VGC]<Wed May 11 23:12:05 2005>[Full GC 31780K->30780K(32704K), 0.2070500secs][DefNew::Eden:
 3440K->1602K(3456K)][DefNew::Survivor:58K->0K(64K)][Tenured: 28282K->29178K(29184K)][Metasp
ace:3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K
)][cause:ObjAllocFail][User: 0.0156250 secs][Sys: 0.0312500 secs]
…
```

「Full GC」に続いて出力されている情報のうち，GC の実行後の情報「->30780K」を確認します。ここでは，FullGC 実行後に，30,780 キロバイトのメモリサイズを必要としていることがわかります。

何回分かの FullGC の拡張 verbosegc 情報を集め，GC 実行後の領域長がいちばん大きい情報を，アプリケーションが必要とするメモリサイズであると判断してください。

7.5.2 Java ヒープ内の New 領域のメモリサイズを追加する理由

Tenured 領域のメモリサイズには，アプリケーションが最低限必要とするメモリサイズに，New 領域分のメモリサイズを追加することをお勧めします。これは，Tenured 領域の未使用メモリサイズが New 領域の使用メモリサイズを下回ることによって，FullGC が頻発するのを防ぐためです。

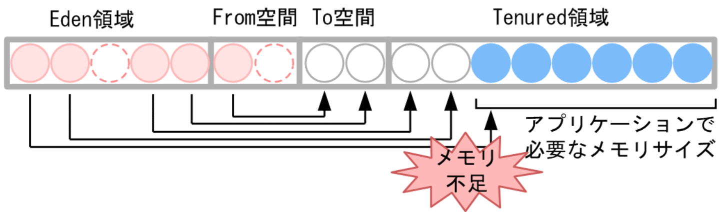
通常，Eden 領域がいっぱいになると，CopyGC が発生します。このとき，Eden 領域と Survivor 領域の From 空間に存在する使用中の Java オブジェクトが，Survivor 領域の To 空間に移動しようとします。このとき，Tenured 領域の未使用領域が Eden 領域と Survivor 領域で使用中のメモリサイズよりも小さいと，New 領域のすべての Java オブジェクトが昇格した場合に，Java オブジェクトを Tenured 領域に移動できなくなります。そこで JavaVM は，FullGC を発生させ，Tenured 領域の未使用メモリサイズを増やそうとします。

これを防ぐために，Tenured 領域には，アプリケーションが必要とするメモリサイズに加えて，New 領域分のメモリサイズを追加してください。

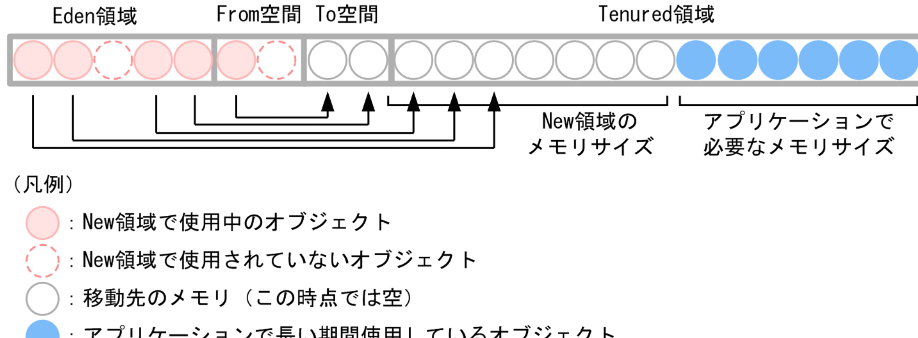
考え方を次の図に示します。

図 7-11　New 領域のメモリサイズを追加する理由の考え方

●オブジェクトが昇格できないおそれがあるためFullGCが発生する例



●オブジェクトが確実に昇格できる例



- **オブジェクトが昇格できないおそれがあるため FullGC が発生する例**

  Tenured 領域のメモリの空き領域（アプリケーションで必要なメモリ領域以外の領域）が New 領域のメモリサイズよりも小さいため，Eden 領域および From 空間からの移動オブジェクトが多い場合，オブジェクトの昇格に対応できないおそれがあります。この場合，FullGC が発生します。

- **オブジェクトが確実に昇格できる例**

  Tenured 領域のメモリの空き領域（アプリケーションで必要なメモリ領域以外の領域）が New 領域と同じサイズ分確保してあるため，Eden 領域および From 空間からの移動オブジェクトが多い場合も，オブジェクトの昇格に対応できます。

なお，New 領域のメモリサイズの見積もりについては，「7.6　Java ヒープ内の New 領域のメモリサイズの見積もり」で説明します。

ポイント

拡張 verbosegc 情報などで確認したときに，CopyGC が発生しないで FullGC が頻発している場合，New 領域からの退避オブジェクトに対して Tenured 領域のメモリサイズが小さいことが考えられます。New 領域のサイズを増やした場合などにこの状態になることがあります。必要に応じて，Tenured 領域のメモリサイズを見直してください。また，New 領域内の Eden 領域と Survivor 領域の関係もあわせて見直してください。

7.6　Java ヒープ内の New 領域のメモリサイズの見積もり

この節では，SerialGC 使用時の New 領域のメモリサイズの見積もりについて説明します。

New 領域のメモリサイズは，次のように見積もります。

```
New領域のメモリサイズ
=Survivor領域のメモリサイズ+Eden領域のメモリサイズ
```

ここでは，Survivor 領域および Eden 領域のメモリサイズを見積もる方法について説明します。

7.6.1　Java ヒープ内の Survivor 領域のメモリサイズの見積もり

Survivor 領域のメモリサイズは，実際にアプリケーションを動作させて，Survivor 領域の使用状況を確認しながらチューニングしていきます。

チューニングの流れを次に示します。

1. **アプリケーションでのリクエスト／レスポンス処理に使用するメモリサイズを見積もり，それを Survivor 領域のメモリサイズに指定して，アプリケーションを実行します。**

   このとき，チューニングで使用する情報を出力するために，**-XX:+HitachiVerboseGCPrintTenuringDistribution オプション**を指定して J2EE サーバを起動します。

2. **Survivor 領域に割り当てられているメモリサイズと，アプリケーション実行時に実際に使用されているメモリ使用量から，メモリ使用率を確認します。**

   メモリ使用率が 100%に近い場合，CopyGC 実行時に New 領域および Survivor 領域の From 空間の使用中のオブジェクトが To 空間に入り切らなくなり，オブジェクトの退避が発生します。この場合は，Survivor 領域を増やすことを検討してください。

3. **Survivor 領域のオブジェクトの年齢分布を確認します。**

   Survivor 領域のメモリサイズを増やしたり，昇格するためのしきい値を上げたりすることで，オブジェクトが昇格するのが遅くなります。寿命の長いオブジェクトを Survivor 領域に格納し続けるのは，性能を低下させる要因になります。

   逆に，Survivor 領域のメモリサイズを減らしたり，昇格するためのしきい値を下げたりすることで，オブジェクトが昇格するのが早くなります。ただし，寿命の短いオブジェクトが昇格するのは，FullGC の発生頻度を増やす要因になります。

   Survivor 領域のメモリサイズと昇格のしきい値は，この二つのバランスを取るように検討してください。

それぞれのチューニング作業について説明します。

(1)　リクエスト／レスポンス処理に使用するメモリサイズの見積もり

Survivor 領域は，寿命の短いオブジェクトを格納する領域です。サーバサイドで動作するアプリケーションの場合，リクエストやレスポンスを処理するために使われている，寿命の短いオブジェクトを格納する領域と考えることができます。このため，Survivor 領域のメモリサイズの見積もりでは，ある時点で存在する寿命が短いオブジェクトの最大サイズ，つまり，ある時点でのリクエストやレスポンスの処理に使用するメモリの最大サイズを考えます。例えば，ステートレスなサーブレットで構成されたアプリケーションの場合，Survivor 領域のメモリサイズを，「一つのリクエスト処理で使用する最大メモリサイズ×リクエストの同時実行数」と考えることができます。

(2)　メモリ使用率の確認

「7.6.1(1)　リクエスト／レスポンス処理に使用するメモリサイズの見積もり」で見積もった値を Survivor 領域のメモリサイズとして設定して，アプリケーションを実行します。実行時に使用されるメモリ使用量から，Survivor 領域に割り当てたメモリサイズに対するメモリ使用率を確認します。

参考

Survivor 領域のメモリサイズは，直接は指定できません。

Survivor 領域のメモリサイズを指定する場合は，まず，-Xmx オプションで Java ヒープの最大サイズを指定して，-XX:NewRatio=<value>によって Java ヒープのメモリサイズを New 領域と Tenured 領域で分ける割合を指定した上で，-XX:SurvivorRatio=<value>オプションによって，New 領域に対する Survivor 領域の割合を指定する必要があります。

メモリ使用率は，拡張 verbosegc 情報で確認できます。

CopyGC 実行時の拡張 verbosegc 情報の出力例を次に示します。

```
…
[VGC]<Wed May 11 23:12:05 2005>[GC 27340K->27340K(32704K), 0.0432900 secs][DefNew::Eden: 344
0K->0K(3456K)][DefNew::Survivor: 58K->64K(64K)][Tenured: 23841K->27282K(29184K)][Metaspace:
3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)][c
ause:ObjAllocFail][User: 0.0156250 secs][Sys: 0.0312500 secs]
…
```

「DefNew::Survivor: 58K->64K(64K)」は，「GC 実行前のメモリサイズ->GC 実行後のメモリサイズ（割り当てられているメモリサイズ)」を意味します。この場合，64 キロバイトの Survivor 領域中 64 キロバイトがすでに使用されていて，使用率は 100%になります。これは，CopyGC で，To 空間のメモリサイズが不足し，Java オブジェクトの退避が行われたことを示しています。退避では，Tenured 領域に本来格納されないはずの寿命の短いオブジェクトが格納され，FullGC の発生頻度を増やす要因になります。このような場合は，Survivor 領域のメモリサイズを増やすことを検討してください。新しい Survivor 領域のメモリサイズは次のように見積もります。

```
新しいSurvivor領域のメモリサイズ
=現在のSurvivor領域のメモリサイズ+退避されたJavaオブジェクトの合計サイズ
```

「退避された Java オブジェクトの合計サイズ」は，GC 実行後の Tenured 領域メモリの増加サイズで近似できます。この例では，「Tenured: 23841K->27282K(29184K)」が，Tenured 領域メモリの増加サイズを示していて，27,282 キロバイト-23,841 キロバイト=3,441 キロバイトとなります。

また，Survivor 領域のメモリサイズを増加させると，Java オブジェクトの昇格のしきい値が上がり，昇格しにくくなります。詳細については，「7.6.1(3)　オブジェクトの年齢分布の確認と見積もり」を参照してください。その結果，CopyGC で回収されない Java オブジェクトが増えることがあるため，-XX:TargetSurvivorRatio=<value>を次のように見積もり，設定してください。

```
新しい-XX:TargetSurvivorRatio=<value>
=現在の-XX:TargetSurvivorRatio=<value>×(現在のSurvivor領域のメモリサイズ/新しいSurvivor領域のメモリサイズ)
```

(3)　オブジェクトの年齢分布の確認と見積もり

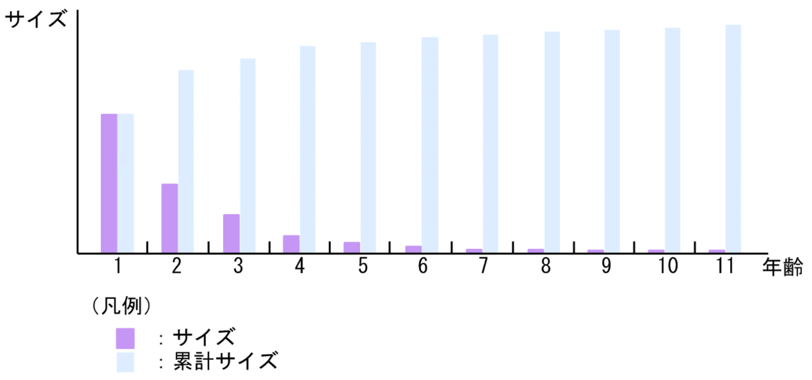
Survivor 領域のオブジェクトの年齢分布を確認し，寿命の長いオブジェクトが存在し続けていないか，または寿命の短いオブジェクトが昇格していないかを確認します。オブジェクトの年齢分布は，-XX:+HitachiVerboseGCPrintTenuringDistribution オプションの出力結果で確認できます。

J2EE サーバ起動時に usrconf.cfg に-XX:+HitachiVerboseGCPrintTenuringDistribution オプションを指定すると，Survivor 領域の使用状況が CopyGC 発生のタイミングで JavaVM ログファイルに出力されます。出力例を次に示します。

```
[PTD]<Wed May 28 11:45:23 2008>[Desired survivor:5467547 bytes][New  threshold:3][MaxTenurin
gThreshold:31][age1:1357527][age2:1539661]
```

「New threshold:」に続けて出力されているのが，次の CopyGC で昇格する Java オブジェクトの最低の年齢です。例の場合は，次回の CopyGC で，年齢が 3 歳以上の Java オブジェクトが昇格します。「age<数値>:」に続けて出力されているのが，Survivor 領域で 1 歳からその年齢までの Java オブジェクトが使用しているメモリサイズの累計です。例の場合は，1 歳の Java オブジェクトのメモリサイズが 1,357,527 バイト，1 歳と 2 歳の Java オブジェクトのメモリサイズの累計が 1,539,661 バイトであることを示しています。また，累計から逆算することで，2 歳の Java オブジェクトのメモリサイズが 182,134(1,539,661-1,357,527)バイトであることがわかります。一般的に，Survivor 領域のオブジェクトの年齢分布は次に示すグラフのようになります。

図 7-12　Survivor 領域のオブジェクトの年齢分布



グラフの「サイズ」は，ある「年齢」の Java オブジェクトの合計のサイズです。また，「累計サイズ」は，ある「年齢」までの Java オブジェクトの合計のサイズです。

このグラフでは，Survivor 領域の Java オブジェクトが占めるサイズが，年齢が上がるに連れて減少しています。また，１歳年齢が上がったときに減少するサイズは年齢が若いほど大きくなります。このことから，次のことがわかります。

- 若い年齢の Java オブジェクトほど，Survivor 領域中に占めるサイズが大きい
- 若い年齢の Java オブジェクトほど，GC で回収されやすい

この例では，6 歳以上の Java オブジェクトはほとんど CopyGC で回収されていません。そのため，Java オブジェクトの昇格のしきい値が 7 歳以上の場合，回収される可能性が低い Java オブジェクトに対して CopyGC を行うことになり，性能を低下させる要因になります。逆に，Java オブジェクトの昇格のしきい値が 2 歳以下の場合，CopyGC で回収される可能性の高いオブジェクトが昇格することになり，FullGC の発生が増す要因になります。この例では，昇格のしきい値を 5〜6 歳程度とするのが，バランスの取れたしきい値となります。

グラフの傾きはシステムによって異なるため，Survivor 領域のオブジェクトの年齢分布を確認し，システムごとの最適な昇格年齢を決定することが重要です。

参考

Java オブジェクトの昇格のしきい値は，CopyGC ごとに動的に変更され，-XX:MaxTenuringThreshold=<value>オプションと，Survivor 領域のメモリサイズおよび-XX:TargetSurvivorRatio=<value>オプションに設定した値を基に決まります。-XX:MaxTenuringThreshold=<value>は，昇格のしきい値の最大の年齢です。Java オブジェクトの年齢がこの値を超えると，必ず昇格します。-XX:TargetSurvivorRatio=<value>は，Survivor 領域のメモリ使用率の目標値です。JavaVM は，Survivor 領域のメモリ使用率が，できるだけこの値に近くなるように，昇格のしきい値を決定します。具体的には，CopyGC が終了した時の 1 歳から n 歳までの Java オブジェクトの累計サイズが，目標の使用率となる n を探し，次の CopyGC の昇格のしきい値を n にします。-XX:TargetSurvivorRatio=<value>のデフォルト

値は 50%です。Survivor 領域のメモリ使用率が大きいほど Survivor 領域を有効に利用できますが，Survivor 領域の空きに余裕がなくなるため，オブジェクトの退避が発生しやすくなります。

7.6.2 Java ヒープ内の Eden 領域のメモリサイズの見積もり

Eden 領域のメモリサイズは，CopyGC を発生させる間隔に影響します。Eden 領域のメモリサイズを大きく取ると，CopyGC の発生間隔が長くなります。なお，CopyGC に掛かる時間は，使用中のオブジェクトの個数に影響され，Eden 領域のメモリサイズにはあまり影響されません。このため，CopyGC が頻発しないように，Eden 領域のメモリサイズを十分に確保することが，性能向上には効果的です。

7.7　Java ヒープ内に一定期間存在するオブジェクトの扱いの検討

これまでの説明は，オブジェクトの寿命に応じて，それぞれの領域に次のように格納することを前提としてきました。

- アプリケーションの動作に必要な寿命の長いオブジェクトは，Tenured 領域に格納する。
- リクエスト処理やレスポンス処理などの寿命の短いオブジェクトは，New 領域に格納する。

しかし，寿命が中間的な一定期間使用されるオブジェクトがあります。このようなオブジェクトは，寿命は長くありませんが，何回かの CopyGC の対象になります。

メモリサイズの見積もりでは，これらのオブジェクトを，New 領域，Tenured 領域のどちらかに格納することを前提として，見積もりをする必要があります。

ここでは，それぞれの特徴を示します。アプリケーションの種類や目的に応じて，どちらかのメモリサイズを増加させるように見積もりをしてください。

7.7.1　Java ヒープ内の New 領域に格納する方法

一定期間存在するオブジェクトを New 領域で管理する方法です。New 領域のメモリサイズに，これらの一定期間存在するオブジェクト分のメモリサイズを追加して見積もります。

New 領域サイズを大きくしてオブジェクトの Tenured 領域への移動を抑止することによって，FullGC の発生も抑止できます。ただし，CopyGC 実行時に New 領域内にある使用中のオブジェクトの数が増えるため，New 領域内でのコピー処理に時間が掛かり，1 回当たりの CopyGC 実行時間は長くなります。CopyGC の実行時間が FullGC 実行時間よりも長くなるような場合は，メモリサイズの再見積もりが必要です。また，メモリ空間のサイズ設定によっては，本来 CopyGC で回収されるはずの寿命の短いオブジェクトが使用する領域が不足して，Tenured 領域への退避が発生するおそれがあります。この場合は，最終的には FullGC が発生してしまいます。

なお，一定期間存在するオブジェクトを New 領域で管理できているかどうかは，拡張 verbosegc 情報で確認できます。実際にアプリケーションを動作させて，出力された拡張 verbosegc 情報で，CopyGC 発生後の Tenured 領域のメモリサイズが大幅に増えていないことを確認してください。

New 領域での管理に失敗している場合，システムの処理性能が大幅に低下していることがあります。また，New 領域で管理できるオブジェクトの最大の年齢には限界があります（限界はプラットフォームやバージョンによって異なります。詳細は，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「14.4　Application Server で指定できる Java HotSpot VM のオプションのデフォルト値」の-XX:MaxTenuringThreshold=<value>を参照してください)。New 領域で管理できていないことがわかった場合は，一定期間存在するオブジェクトは，Tenured 領域に格納して管理することを検討してください。Tenured 領域で管理する方法については，「7.7.2　Java ヒープ内の Tenured 領域に格納する方法」を参照してください。

7.7.2 Java ヒープ内の Tenured 領域に格納する方法

New 領域で管理するオブジェクトの最大の年齢は，-XX:MaxTenuringThreshold=<value>オプションで指定できます。例えば，「-XX:MaxTenuringThreshold=2」を指定しておけば，3 回目の CopyGC の対象になったオブジェクトは，すべて Tenured 領域に移動します。

この方法を使用すると，CopyGC の対象になるオブジェクトが少なくなり，実行時間が短縮できます。ただし，多くのオブジェクトが Tenured 領域に移動するため，Tenured 領域がいっぱいになった段階で FullGC が定期的に発生します。システムを安定して動作させるためには，システムに掛かる負荷が低いときなどに，FullGC を強制的に発生させてください。FullGC を強制的に発生させるには，次の方法があります。

- プログラム内で System.gc()メソッドを呼び出す
- javagc コマンドを実行する
  javagc コマンドの詳細については，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「javagc（GC の強制発生）」を参照してください。

7.7.3 Explicit ヒープに格納する方法

チューニングではありませんが，Java プログラムを変更することで，一定期間存在するオブジェクトを Explicit ヒープに格納して管理する方法があります。Explicit ヒープは GC の対象とならない領域です。Explicit ヒープを利用することで，オブジェクトが Tenured 領域に移動することを防ぎ，FullGC の発生を抑えることができます。また，自動配置設定ファイルを利用することで，Explicit ヒープに直接オブジェクトを配置することもできます。詳細は，「7.11 Explicit ヒープのチューニング」を参照してください。アプリケーションサーバでは，HTTP セッションに関連するオブジェクトを Explicit ヒープで管理しています。

7.8 Java ヒープの最大サイズ／初期サイズの決定

Tenured 領域および New 領域の見積もりができたら，それらを基に Java ヒープの最大サイズと初期サイズを決定します。

Java ヒープの最大サイズは，次のように決定します。

```
Javaヒープの最大サイズ
=Tenured領域のメモリサイズ+New領域のメモリサイズ
```

Java ヒープのメモリサイズを設定する場合，まず，-Xmx オプションに，拡張領域のサイズも含む，Java ヒープ領域の最大サイズを指定します。次に，-Xms オプションに，Java ヒープ領域の初期サイズを指定します。なお，-Xms オプションに指定するサイズは，-Xmx オプションに指定したサイズよりも小さくする必要があります。

JavaVM は，起動時に，-Xms オプションで指定された分のメモリ領域を Java ヒープ領域として確保します。その後，アプリケーション実行中に-Xms オプションで指定した以上のメモリ領域が必要になった場合に，-Xmx オプションで指定したサイズになるまで，ヒープ領域を追加して割り当てていきます。逆に，アプリケーションの中で不要なメモリ領域が発生した場合は，-Xms オプションで指定したサイズまで，Java ヒープ領域として確保している領域を減らしていきます。

なお，システムの安定稼働のためには，-Xmx オプションと-Xms オプションには同じ値を指定することをお勧めします。

7.9　Java ヒープ内の Metaspace 領域のメモリサイズの見積もり

この節では，SerialGC 使用時の Metaspace 領域のメモリサイズの見積もりについて説明します。Metaspace 領域は，ロードされた class などが格納される領域です。

Metaspace 領域のメモリサイズは，「7.2.6　SerialGC 使用時の JavaVM で使用するメモリ空間の構成と JavaVM オプション」で示したとおり，-Xmx オプションで指定したメモリサイズ（Java ヒープ）とは別に確保されます。

デフォルト値については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「14.4　Application Server で指定できる Java HotSpot VM のオプションのデフォルト値」を参照してください。

次に，Metaspace 領域の使用量の見積もり方法を示します。

- **Metaspace 領域の使用量の見積もり方法**

  Metaspace 領域でのメモリ使用量は，大体，J2EE サーバにロードされるクラスファイルの合計サイズになります。アプリケーションサーバの場合，次に示すクラスファイルのサイズの総和から見積もることができます。

  1. WEB-INF/classes 以下のすべてのクラスファイル
  2. WEB-INF/lib 以下の JAR ファイルに含まれる，すべてのクラスファイル
  3. JSP コンパイル結果として生成された，すべてのクラスファイル
  4. EJB-JAR に含まれるすべてのクラスファイル
  5. コンテナ拡張ライブラリ，ライブラリ JAR，参照ライブラリを利用している場合に追加する JAR ファイルに含まれる，すべてのクラスファイル
  6. コンテナが作成するクラスファイル

     アプリケーション開始後の Metaspace 領域－アプリケーション登録前の Metaspace 領域。実際に J2EE サーバを起動し，Metaspace 領域を確認して算出してください。
  7. アプリケーションサーバ提供のクラスファイル（システムクラスファイル）

     システムクラスファイルの総和は約 160 メガバイトになります。
  8. JDK 提供のクラスファイル

     JDK 提供のクラスファイルの総和は約 110 メガバイトになります。
  9. リフレクション処理が生成する動的クラス

     リフレクション処理では処理の高速化のため，呼び出し用のクラスを動的に生成します。

     このクラスが占めるサイズについてはサーバ起動から十分な時間経過するまでの間に増加した Metaspace 利用サイズを目安として実測で求めてください。

  Metaspace 領域のメモリサイズは，-XX:MaxMetaspaceSize=<size>オプション，および-XX:MetaspaceSize=<size>オプションで指定します。これらのオプションのデフォルト値について

は，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「14.4　Application Server で指定できる Java HotSpot VM のオプションのデフォルト値」を参照してください。

なお，アプリケーションのインポート時に一時的に Metaspace 領域の使用量が増加する場合があります。

参考

開発環境でのソフト参照による Metaspace 領域の圧迫の解消方法

ソフト参照の影響によって，アプリケーションをアンデプロイしたときに Metaspace 領域の解放が遅延することがあります。このため，開発環境などでアプリケーションのデプロイおよびアンデプロイを繰り返した場合，Metaspace 領域の解放遅延が原因で，Metaspace 領域が圧迫されることがあります。ソフト参照による Metaspace 領域の圧迫は，次のオプションを指定すると解消できます。

- -XX:SoftRefLRUPolicyMSPerMB=0

-XX:SoftRefLRUPolicyMSPerMB オプションに 0 を指定すると，すべてのソフト参照が無効になります。ソフト参照は，性能向上のためのキャッシュとして使用されることが多いため，このオプションの指定によってアプリケーションの性能が劣化するおそれがあります。このため，このオプションは，開発環境で Metaspace 領域が圧迫された場合にだけ指定してください。

7.10　拡張 verbosegc 情報を使用した FullGC の要因の分析方法

この節では，SerialGC 使用時の拡張 verbosegc 情報を使用した FullGC の要因の分析方法について説明します。拡張 verbosegc 情報は，JavaVM オプションである-XX:+HitachiVerboseGC オプションを指定することによって出力できる JavaVM のログ情報です。チューニングに役立つ情報のほか，障害要因の分析にも役立つ情報が出力されます。

チューニング実行時に拡張 verbosegc 情報を参照することで，次の情報を確認できます。

- GC 実行前と実行後の各領域の使用メモリサイズ
- GC が発生した要因

また，-XX:+HitachiVerboseGC とほかの JavaVM オプションを組み合わせることによって，さらに詳細な情報が出力できます。-XX:+HitachiVerboseGC オプション，およびそのほかの JavaVM 拡張オプションの詳細については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「14.2　JavaVM 拡張オプションの詳細」を参照してください。

7.10.1　拡張 verbosegc 情報の出力形式の概要

拡張 verbosegc 情報は，CopyGC が発生した場合と，FullGC が発生した場合に出力されます。

CopyGC が発生すると，GC の種類として「GC」と出力されます。また，FullGC が発生すると，GC の種類として「Full GC」と出力されます。種類に続いて，それぞれの領域の「＜ GC 前の領域長＞->＜ GC 後の領域長＞（＜確保済み領域サイズ＞)」が出力されます。

以降に，FullGC が発生した場合の拡張 verbosegc 情報の出力例を示します。拡張 verbosegc 情報には，ほかにも GC の発生要因や GC スレッドの CPU 時間も出力されます。

拡張 verbosegc 情報の出力形式の詳細，およびそれぞれのオプションの詳細については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「14.　JavaVM 起動オプション」を参照してください。

7.10.2　FullGC 発生時の拡張 verbosegc 情報の出力例

ここでは，FullGC の発生時の拡張 verbosegc 情報の出力例を示します。

(1) New 領域（Eden 領域と Survivor 領域の合計）で使用しているメモリサイズが Tenured 領域の最大値に対する未使用メモリサイズを上回った場合

拡張 verbosegc 情報の出力例を次に示します。背景色付きの**太字**の部分が FullGC 発生の要因を示す個所です。

```
…
[VGC]<Wed May 11 23:12:05 2005>[GC 27340K->27340K(32704K), 0.0432900 secs][DefNew::Eden: 3440K->0K(3456K)][DefNew::Survivor: 58K->58K(64K)][Tenured: 23841K->27282K(29184K)][Metaspace: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)][cause:ObjAllocFail][User: 0.0156250 secs][Sys: 0.0312500 secs]
[VGC]<Wed May 11 23:12:05 2005>[Full GC 30780K->30780K(32704K), 0.2070500 secs][DefNew::Eden: 3440K->1602K(3456K)][DefNew::Survivor: 58K->0K(64K)][Tenured: 27282K->29178K(29184K)][Metaspace: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)][cause:ObjAllocFail][User: 0.0156250 secs][Sys: 0.0312500 secs]
…
```

この出力例からは，次のことがわかります。

- New 領域で使用しているメモリサイズ（3440K+58K=3498K）が，Tenured 領域の最大値に対する未使用メモリサイズ（29184K-27282K=1902K）を上回りました。
- FullGC の要因は，オブジェクトの割り付け失敗です。

(2) CopyGC の実施の結果，New 領域（Eden 領域と Survivor 領域の合計）から Tenured 領域へのオブジェクトの移動に失敗した場合

拡張 verbosegc 情報の出力例を次に示します。背景色付きの**太字**の部分が FullGC 発生の要因を示す個所です。

```
…
[VGC]<Thu Oct 20 11:04:42 2011>[GC 26418K->26418K (29696K), 0.0000000 secs][DefNew::Eden:8188K->8188K(8192K)][DefNew::Survivor: 1021K->1021K(1024K)][Tenured:17208K->17208K (20480K)][Metaspace: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)][cause:ObjAllocFail][User: 0.0000000 secs][Sys: 0.0000000 secs][IM: 877K, 1104K, 0K][TC: 9][DOE: 0K, 0]
[VGC]<Thu Oct 20 11:04:42 2011>[Full GC 26418K->6450K(29696K), 0.0156250 secs][DefNew::Eden: 8188K->0K(8192K)][DefNew::Survivor:1021K->0K(1024K)][Tenured:17208K->6450K(20480K)][Metaspace: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)][cause:PromotionFail][User: 0.0156250 secs][Sys: 0.0000000 secs][IM: 925K, 1104K, 0K][TC: 9][DOE: 0K, 0]
```

この出力例からは，次のことがわかります。

- FullGC の要因は，CopyGC による，New 領域から Tenured 領域へのオブジェクトの移動の失敗です。

(3) アロケーションしたいメモリサイズ（new で作成する Java オブジェクトのサイズ）が Tenured 領域の未使用メモリサイズを上回る場合

拡張 verbosegc 情報の出力例を次に示します。背景色付きの**太字**の部分が FullGC 発生の要因を示す個所です。

```
…
[VGC]<Wed May 11 23:53:18 2005>[GC 28499K->28490K(32704K), 0.0540590 secs][DefNew::Eden: 808K->0K(3456K)][DefNew::Survivor: 64K->62K(64K)][Tenured: 27626K->28428K(29184K)][Metaspace: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)][cause:ObjAllocFail][User: 0.0156250 secs][Sys: 0.0312500 secs]
[VGC]<Wed May 11 23:53:18 2005>[Full GC 28490K->8959K(32704K), 0.1510380 secs][DefNew::Eden: 0K->0K(3456K)][DefNew::Survivor: 62K->0K(64K)][Tenured: 28428K->8959K(29184K)][Metaspace: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)][cause:ObjAllocFail][User: 0.0156250 secs][Sys: 0.0312500 secs]
…
```

この出力例からは，次のことがわかります。

- Tenured 領域の未使用メモリサイズ（29184K-28428K=756K）を上回るメモリサイズの Java オブジェクトを，new で作成しようとしました。
- FullGC の要因は，オブジェクトの割り付け失敗です。

(4) CopyGC 実施の結果，確保済み Tenured 領域の未使用メモリサイズが 10,000 バイトを下回った場合

拡張 verbosegc 情報の出力例を次に示します。背景色付きの**太字**の部分が FullGC 発生の要因を示す個所です。

```
…
[VGC]<Fri May 25 15:21:33 2007>[GC 15436K->15416K(19840K), 0.0111825 secs][DefNew::Eden: 4413K->0K(4416K)][DefNew::Survivor: 512K->509K(512K)][Tenured: 10511K->14906K(14912K)][Metaspace: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)][cause:ObjAllocFail][User: 0.0000000 secs][Sys: 0.0000000 secs]
[VGC]<Fri May 25 15:21:33 2007>[Full GC 15416K->8622K(19840K), 0.0284614 secs][DefNew::Eden: 0K->0K(4416K)][DefNew::Survivor: 509K->0K(512K)][Tenured: 14906K->8622K(14912K)][Metaspace: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)][cause:ObjAllocFail][User: 0.0312500 secs][Sys: 0.0000000 secs]
…
```

この出力例からは，次のことがわかります。

- 1 行目の CopyGC で New 領域から Tenured 領域にオブジェクトが移動したことによって，Tenured 領域の使用済みメモリサイズが 10,511 キロバイトから 14,906 キロバイトに増加しました。これによって，確保済み Tenured 領域の未使用メモリサイズが 14,912 キロバイト-14,906 キロバイト=6 キロバイトとなり，10,000 バイト（約 10 キロバイト）を下回りました。

- 1 行目の CopyGC の原因は，オブジェクトの割り付け失敗です。1 行目の CopyGC と 2 行目の FullGC は，Java プログラムに制御が戻る前に連続して発生します。

(5) CopyGC 実施時の Tenured 領域へのオブジェクトの移動によって，確保済み Tenured 領域の拡張が発生した場合

拡張 verbosegc 情報の出力例を次に示します。背景色付きの**太字**の部分が FullGC 発生の要因を示す個所です。

```
…
[VGC]<Fri May 25 15:42:00 2007>[GC 12745K->10151K(15872K), 0.0048346 secs][DefNew::Eden: 4416K->0K(4416K)][DefNew::Survivor: 137K->512K(512K)][Tenured: 8192K->9639K(10944K)][Metaspace: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)][cause:ObjAllocFail][User: 0.0156250 secs][Sys: 0.0000000 secs]
[VGC]<Fri May 25 15:42:00 2007>[GC 14563K->14536K(19072K), 0.0104957 secs][DefNew::Eden: 4412K->0K(4416K)][DefNew::Survivor: 512K->510K(512K)][Tenured: 9639K->14026K(14144K)][Metaspace: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)][cause:ObjAllocFail][User: 0.0156250 secs][Sys: 0.0000000 secs]
[VGC]<Fri May 25 15:42:00 2007>[Full GC 14536K->8610K(19072K), 0.0287254 secs][DefNew::Eden: 0K->0K(4416K)][DefNew::Survivor: 510K->0K(512K)][Tenured: 14026K->8610K(14144K)][Metaspace: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)][cause:ObjAllocFail][User: 0.0312500 secs][Sys: 0.0000000 secs]
…
```

この出力例からは，次のことがわかります。

- 2 行目の CopyGC で New 領域から Tenured 領域へのオブジェクトが移動したことによって，Tenured 領域が最低でも 14,026 キロバイト以上必要になりました。このため，確保済み Tenured 領域サイズが 10,944 キロバイトから 14,144 キロバイトに拡張されました。
- 2 行目の CopyGC の原因は，オブジェクトの割り付け失敗です。2 行目の CopyGC と 3 行目の FullGC は，Java プログラムに制御が戻る前に連続して発生します。

(6) アプリケーション内で java.lang.System.gc()メソッドが実行された場合

拡張 verbosegc 情報の出力例を次に示します。背景色付きの**太字**の部分が FullGC 発生の要因を示す個所です。

```
…
[VGC]<Mon Apr 18 20:36:29 2005>[Full GC 330K->150K(3520K), 0.0387690 secs][DefNew::Eden: 330K->0K(2048K)][DefNew::Survivor: 0K->0K(64K)][Tenured: 0K->150K(1408K)][Metaspace: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)][cause:System.gc][User: 0.0156250 secs][Sys: 0.0312500 secs]
…
```

この出力例からは，次のことがわかります。

- FullGC の要因は，J2EE アプリケーション内またはバッチアプリケーション内での java.lang.System.gc()メソッド呼び出しです。

(7) Metaspace 領域にアロケーションしたいメモリサイズが確保済み Metaspace 領域の未使用メモリサイズを上回る場合

拡張 verbosegc 情報の出力例を次に示します。背景色付きの**太字**の部分が FullGC 発生の要因を示す個所です。

```
…
[VGC]<Wed Jan 07 01:56:13 2015>[Full GC 11273K->6037K(15872K), 0. 0060004 secs][DefNew::Eden
: 442K->0K(4416K)][DefNew::Survivor: 512K->0K(512K)][Tenured: 10319K->6037K(10944K)][Metaspa
ce: 22811K(24520K, 24576K)->22811K(24520K, 24576K)][class space: 10758K(10988K, 11008K)->107
58K(10988K, 11008K)][cause:MetaspaceAllocFail][User: 0.0312002 secs][Sys: 0.0000000 secs]
…
```

この出力例からは，次のことがわかります。

- Metaspace 領域にアロケーションしようとしたメモリサイズが，確保済み Metaspace 領域の未使用メモリサイズ（24576K-22811K=1765K）を上回りました。
- FullGC の要因は，Metaspace ヒープの割り付け失敗です。

(8) javagc コマンドを実行した場合の出力例

拡張 verbosegc 情報の出力例を次に示します。背景色付きの**太字**の部分が FullGC 発生の要因を示す個所です。

```
…
[VGC]<Mon Apr 18 21:46:50 2005>[Full GC 369K->189K(3520K), 0.0403010 secs][DefNew::Eden: 369
K->0K(2048K)][DefNew::Survivor: 0K->0K(64K)][Tenured: 0K->189K(1408K)][Metaspace: 3634K(4492
K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)][cause:JavaG
C Command][User: 0.0156250 secs][Sys: 0.0312500 secs]
…
```

この出力例からは，次のことがわかります。

- FullGC の要因は，javagc コマンド実行です。

(9) jheapprof コマンドを実行した場合の出力例

拡張 verbosegc 情報の出力例を次に示します。背景色付きの**太字**の部分が FullGC 発生の要因を示す個所です。

```
…
[VGC]<Mon Apr 18 21:46:50 2005>[Full GC 369K->189K(3520K), 0.0403010 secs][DefNew::Eden: 369
K->0K(2048K)][DefNew::Survivor: 0K->0K(64K)][Tenured: 0K->189K(1408K)][Metaspace: 3634K(4492
K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)][cause:JHeap
Prof Command][User: 0.0156250 secs][Sys: 0.0312500 secs]
…
```

この出力例からは，次のことがわかります。

- FullGC の要因は，jheapprof コマンド実行です。

7.11 Explicit ヒープのチューニング

ここでは，Explicit ヒープのチューニングについて説明します。

7.11.1 Explicit ヒープのメモリサイズの見積もり（J2EE サーバが使用するメモリサイズの見積もり）

Explicit ヒープをチューニングする前提として，明示管理ヒープ機能を使用するための設定が必要です。明示管理ヒープ機能は，JavaVM の起動オプションとして-XX:+HitachiUseExplicitMemory が指定されている場合に有効になります。JDK11 以前で動作する J2EE サーバの場合，明示管理ヒープ機能はデフォルトで使用される設定になっています。また，Tenured 領域のメモリサイズ増加の要因となるオブジェクトが Explicit ヒープに配置されるように設定されています。このため，J2EE サーバが配置するオブジェクトに必要な Explicit ヒープのメモリサイズを必ず見積もってください。

明示管理ヒープ機能は，Explicit ヒープのメモリサイズを適切に見積もった上で使用しないと，効果が出ません。

J2EE サーバでは，Tenured 領域のメモリサイズ増加の要因になる，HTTP セッションに関するオブジェクトを Explicit ヒープに配置します。

Explicit ヒープのメモリサイズを見積もった結果，HTTP セッションに関するオブジェクトが利用する Explicit ヒープのメモリサイズが極端に大きい場合は，「付録 A　HTTP セッションで利用する Explicit ヒープの効率的な利用」を参考にして，アプリケーションの設計を見直してください。

7.11.2 HTTP セッションに関するオブジェクトで利用するメモリサイズ

ここでは，HTTP セッションに関するオブジェクトで利用するメモリサイズの見積もりについて説明します。

注意事項

なお，HTTP セッションで利用する Explicit ヒープの省メモリ化機能を使用する場合，この手順での見積もりはできません。HTTP セッションで利用する Explicit ヒープの省メモリ化機能を使用する場合は，「7.11.4　稼働情報による見積もり」で示す手順でメモリサイズを見積もってください。

HTTP セッションに関するオブジェクトで利用する Explicit ヒープのメモリサイズは，次の式で見積もります。

```
HTTPセッションに関するオブジェクトで利用するExplicitヒープのメモリサイズ
= HTTPセッションで利用するExplicitヒープのメモリサイズ
 + Webコンテナ内部で利用するExplicitヒープ領域のメモリサイズ
```

HTTP セッションで利用する Explicit ヒープのメモリサイズは，次の式で見積もります。

```
HTTPセッションで利用するExplicitヒープのメモリサイズ
= 1セッションで使用するメモリサイズ※1×システムで必要なセッション数
```

HTTP セッションのために，Web コンテナ内部で利用する Explicit ヒープ領域のメモリサイズは，次の式で見積もります。

```
Webコンテナ内部で利用するExplicitヒープ領域のメモリサイズ
= HTTPセッション管理用オブジェクトのサイズ※2×（Webアプリケーションの数※3 + 2)
```

注※1

1 セッションで使用するメモリサイズは，Explicit メモリブロック 1 個のサイズに相当します。Explicit メモリブロック 1 個のサイズは，実際にアプリケーションを動作させて，Explicit ヒープの使用状況を確認しながら見積もります。なお，Explicit メモリブロックの最小サイズは，明示管理ヒープ機能の自動配置機能の使用の有無によって異なります。明示管理ヒープ機能の自動配置機能の使用の有無による Explicit メモリブロックの最小サイズを次に示します。

表 7-4 明示管理ヒープ機能の自動配置機能の使用の有無による Explicit メモリブロックの最小サイズ

項番	明示管理ヒープ機能の自動配置機能使用の有無	Explicit メモリブロックの最小サイズ
1	○	16 キロバイト
2	×	64 キロバイト

（凡例）

○：明示管理ヒープ機能の自動配置機能を使用する。

×：明示管理ヒープ機能の自動配置機能を使用しない。

なお，Explicit メモリブロックは 64 キロバイト単位で拡張されます。そのため，実際は「Explicit メモリブロック 1 個のサイズ≧1 セッションで使用するメモリサイズ」となります。また，明示管理ヒープ機能の自動配置機能を使用する場合は，さらに 16 キロバイトを加算して見積もってください。

注※2

HTTP セッション管理用オブジェクトのサイズは表 7-4 に示す Explicit メモリブロックの最小サイズです。

注※3

「Web アプリケーションの数」は，開始している Web アプリケーションの個数を表します。

(1) 見積もり手順

見積もり手順を示します。

1. HTTP セッションを作成するアプリケーションを開始して，セッション破棄（ログアウトなど）の直前まで処理を実行します。

2. javagc コマンドを実行して，FullGC を発生させます。

3. 明示管理ヒープ機能のイベントログに出力された FullGC 実行後の情報を確認します。

   明示管理ヒープ機能のイベントログは，JavaVM の起動オプションである-XX:HitachiExplicitMemoryJavaLog オプションで指定したファイルまたはディレクトリに出力されます。デフォルトの出力先は，次のとおりです。

   Windows の場合

   <Application Server のインストールディレクトリ>¥CC¥server¥public¥ejb¥<J2EE サーバ名>¥logs¥ehjavalog[n].log

   UNIX の場合

   /opt/Cosminexus/CC/server/public/ejb/<J2EE サーバ名>/logs/ehjavalog[n].log

   ここでは，一連の業務で必要な Explicit ヒープの利用サイズと Explicit メモリブロックの個数を確認します。

   メモリサイズ算出時には，イベントログの出力項目のうち，次の項目を確認します。

   - Explicit ヒープの確保済みサイズ
   - Explicit メモリブロックの個数

   明示管理ヒープ機能のイベントログの内容については，マニュアル「アプリケーションサーバ 機能解説 保守／移行編」の「5.11 明示管理ヒープ機能のイベントログ」を参照してください。これらの項目は，GC 発生時に出力される，Explicit ヒープ利用状況に含まれます。

4. 手順 3.で確認した内容を基に，Explicit メモリブロック 1 個のサイズを算出します。

   次の式で算出できます。

   ```
   Explicitメモリブロック１個のサイズ
   =Explicitヒープの確保済みサイズ／Explicitメモリブロックの個数
   ```

   Explicit メモリブロック 1 個のサイズは，1 セッションで使用するメモリサイズに相当します。

5. 手順 4.で算出した値に，業務で必要になるセッション数を掛け，HTTP セッションで使用する Explicit ヒープの合計メモリサイズを算出します。

(2) 見積もり例

ここでは，明示管理ヒープ機能のイベントログの出力例を基に，見積もり例を示します。明示管理ヒープ機能のイベントログの出力例を次に示します。

明示管理ヒープ機能のイベントログの出力例

```
[ENS]<Thu Oct 21 14:55:50 2007>[EH: 12672K->12800K(12800K/65536K)][E/F/D: 200/0/0][cause:GC]
[CF: 0]
```

この例では，Explicitヒープの確保済みサイズは12,800キロバイト，Explicitメモリブロックの個数は200個と出力されています。この値を「7.11.2(1)　見積もり手順」の手順4.で示した式に当てはめると，次のようになります。

Explicitメモリブロックサイズの見積もり例

```
Explicitメモリブロック1個のサイズ
=Explicitヒープの確保済みサイズ（12,800キロバイト）/Explicitメモリブロックの個数（200）
=64キロバイト
```

これに業務で想定するセッション数を掛けた値が，HTTPセッションで使用するExplicitヒープの合計メモリサイズになります。

7.11.3　明示管理ヒープ機能利用によるメモリサイズの見積もりへの影響

アプリケーションサーバの機能の中には，その機能を使用するかどうかでExplicitヒープ領域のメモリサイズに影響が出るものがあります。機能利用の有無による見積もりへの影響について表に示します。

表7-5　機能利用の有無による見積もりへの影響

項番	機能	機能利用の有無によるExplicitヒープ領域の違い	見積もりへの影響
1	明示管理ヒープの自動解放機能（-XX:+HitachiExplicitMemoryAutoReclaimオプション）	有効な場合 Explicitヒープ領域の中で「Javaヒープの Survivor領域のサイズ×2」の領域をJavaVMが使用します。 無効な場合 JavaVMはExplicitヒープ領域を使用しません。	機能が有効な場合の最終的なExplicitヒープ領域の見積もりサイズは，稼働情報から算出した見積もりサイズに「JavaヒープのSurvivor領域のサイズ×2」を加算した値となります。
2	明示管理ヒープ機能の自動配置機能（-XX:+HitachiAutoExplicitMemoryオプション）	有効な場合 Explicitメモリブロックの最小サイズが16キロバイトになります。 無効な場合 Explicitメモリブロックの最小サイズが64キロバイトになります。 また，見積もりには影響しませんが，機能の有効・無効によってExplicitヒープ領域の確保方法が変わります。	機能の有効・無効によって，稼働情報に出力されるExplicitヒープサイズに関する情報が異なります。

項番	機能	機能利用の有無による Explicit ヒープ領域の違い	見積もりへの影響
		有効な場合 プロセス起動時に-XX:HitachiExplicitHeapMaxSize オプションで指定したサイズのメモリを確保します。 無効な場合 Explicit メモリブロック取得時に，必要なだけのメモリサイズを確保します。	

7.11.4　稼働情報による見積もり

J2EE サーバのテストを実施する場合，または運用開始後の J2EE サーバによる実際の Explicit ヒープ使用状況は，稼働情報で確認できます。ここでは，稼働情報による確認手順について説明します。

稼働情報の出力内容，出力するための設定，および稼働情報ファイルの出力先については，マニュアル「アプリケーションサーバ 機能解説 運用／監視／連携編」の「3.3　稼働情報ファイルの出力機能」を参照してください。

(1)　稼働情報を使用した見積もりの考え方

稼働情報を使用した見積もりでは，システムに必要な Explicit ヒープ領域のメモリサイズは次のようになります。

1. HTTP セッションで使用する Explicit ヒープ領域のメモリサイズ
2. 1.の領域を除いた，内部（コンテナ）で使用する Explicit ヒープ領域のメモリサイズ
3. アプリケーションおよび JavaVM で使用する Explicit ヒープ領域のメモリサイズ
4. JavaVM が Explicit メモリブロックを管理するために使用する Explicit ヒープ領域のメモリサイズ（Java ヒープの Survivor 領域のサイズ×2）

1.～3.のメモリサイズは，稼働情報で確認できます。4.は，明示管理ヒープの自動解放機能が有効な場合に，「Java ヒープの Survivor 領域のサイズ×2」のメモリサイズを使用します。

1.～3.で示す各 Explicit ヒープ領域を使用するものの例を表で示します。なお，1.～3.は表中の項番 1～3 に対応しています。

表 7-6　Explicit ヒープ領域を使用するものの具体例

項番	Explicit ヒープ領域	Explicit ヒープ領域を使用するものの具体例
1	HTTP セッションで使用する Explicit ヒープ領域	HTTP セッション

項番	Explicit ヒープ領域	Explicit ヒープ領域を使用するものの具体例
2	コンテナで使用する Explicit ヒープ領域	HTTP セッション管理用オブジェクト
3	アプリケーションおよび JavaVM で使用する Explicit ヒープ領域	• アプリケーション • JavaVM

(2) 見積もりに使用する稼働情報取得時の注意

見積もりに使用する稼働情報は，本番環境，または本番環境と同等の環境で取得してください。

次に示す項目が本番環境と異なる場合は，稼働情報を使って適切なメモリサイズを見積もることはできません。

- 各定義ファイルに設定するプロパティ，およびオプションに指定する値
- サーバに登録されている Web アプリケーションの数
- 業務アプリケーションが処理するデータのサイズ
- 一定時間内に処理するデータの数

また，見積もりのために稼働情報を取得する場合，Explicit ヒープ領域を使い切った状態にならないよう Explicit ヒープ領域サイズの最大値を設定するオプションを指定してください。

Explicit ヒープ領域の最大サイズが不十分な状態で稼働情報を取得した場合，Explicit ヒープ領域を使い切った状態になるおそれがあります。Explicit ヒープ領域を使い切った状態で取得した稼働情報では，適切な見積もりはできません。Explicit ヒープ領域を使い切った状態かどうかは，稼働情報の EHeapSize.HighWaterMark の値が，Explicit ヒープ領域の最大サイズの値と同じ値になっているかどうかで確認できます。稼働情報の EHeapSize.HighWaterMark の値と Explicit ヒープ領域の最大サイズの値が同じだった場合，Explicit ヒープ領域を使い切っている状態となります。

(3) 見積もり方法

稼働情報を基にした見積もり方法を次に示します。

(a) HTTP セッションで利用する Explicit ヒープ領域のメモリサイズ

HTTP セッションで利用するメモリサイズは，「7.11.2 HTTP セッションに関するオブジェクトで利用するメモリサイズ」で示した HTTP セッションで利用する Explicit ヒープのメモリサイズの式で求めます。このとき，式に含まれる「1 セッションで使用するメモリサイズ」を稼働情報で確認できます。

1 セッションで使用するメモリサイズは，稼働情報に出力された「Explicit メモリブロックの最大サイズ」に該当します。「Explicit メモリブロックの最大サイズ」には，稼働情報収集間隔の間に解放された Explicit メモリブロックのうち，最大のものの利用済みサイズが出力されます。そのため，Explicit ヒープを見積もる際は，64 キロバイト単位で切り上げて見積もってください。さらに，明示管理ヒープ機能の自動配置機能を使用する場合は，16 キロバイトを加算して見積もってください。

また，見積もりをする際は，次に示す値を参考にしてください。なお，システムで必要なセッション数は，「Explicit メモリブロックの個数」に該当します。

- HTTP セッションで取得した Explicit メモリブロックの最大サイズ（HTTPSessionEMemoryBlockMaxSize.HighWaterMark の値）
- HTTP セッションで取得した Explicit メモリブロックの個数（HTTPSessionEMemoryBlockCount.HighWaterMark の値）

(b)　コンテナで利用する Explicit ヒープ領域のメモリサイズ

コンテナで使用する Explicit ヒープ領域のメモリサイズは，稼働情報の「コンテナで利用する Explicit ヒープサイズ」が該当します。見積もりに使用する稼働情報は取得した値の中で最大値（ContainerEHeapSize.HighWaterMark の値）を使用してください。

(c)　アプリケーションおよび JavaVM で利用する Explicit ヒープ領域のメモリサイズ

アプリケーションおよび JavaVM で利用する Explicit ヒープ領域のメモリサイズは，稼働情報の「アプリケーションで利用する Explicit ヒープサイズ」の値が該当します。見積もりに使用する稼働情報は取得した値の中で最大値（ApplicationEHeapSize.HighWaterMark）を使用してください。

(4)　稼働情報の確認手順

稼働情報の確認手順について説明します。ここでは，(3)による稼働情報の見積もり式を例に説明します。なお，稼働情報ファイルの出力内容については，マニュアル「アプリケーションサーバ 機能解説 運用／監視／連携編」の「3.3　稼働情報ファイルの出力機能」を参照してください。

見積もり式

```
必要なExplicitヒープ領域のメモリサイズ
＝(HTTPSessionEMemoryBlockMaxSize.HighWaterMarkを64キロバイト単位に切り上げた値
×HTTPSessionEMemoryBlockCount.HighWaterMark)
＋ContainerEHeapSize.HighWaterMark
＋ApplicationEHeapSize.HighWaterMark
＋JavaヒープのSurvivor領域のサイズ
×2(明示管理ヒープ自動解放機能が有効な場合だけ)
```

それぞれの値の確認方法について説明します。

(a)　HTTP セッションで利用する Explicit ヒープ領域のメモリサイズ

HTTP セッションで利用する Explicit ヒープ領域のメモリサイズは，JavaVM の稼働情報ファイルに出力される HTTPSessionEMemoryBlockMaxSize.HighWaterMark，および HTTPSessionEMemoryBlockCount.HighWaterMark の値を使って確認します。

HTTP セッションで利用する Explicit ヒープ領域のメモリサイズの稼働情報の出力例を次の図に示します。

図 7-13　HTTP セッションで利用する Explicit ヒープ領域のメモリサイズの稼働情報の出力例

Date(+0900)	HTTPSessionEMemoryBlockMaxSize.StartTime(+0900)	HTTPSessionEMemoryBlockMaxSize.HighWaterMark	HTTPSessionEMemoryBlockMaxSize.LowWaterMark	HTTPSessionEMemoryBlockMaxSize.Current	HTTPSessionEMemoryBlockCount.StartTime(+0900)	HTTPSessionEMemoryBlockCount.HighWaterMark	HTTPSessionEMemoryBlockCount.LowWaterMark	HTTPSessionEMemoryBlockCount.Current
2009/11/18 10:44:31	43:30.8	0	0	0	43:30.8	0	0	0
2009/11/18 10:45:31	43:30.8	0	0	0	43:30.8	0	0	0
2009/11/18 10:46:31	43:30.8	0	0	0	43:30.8	0	0	0
2009/11/18 10:47:31	43:30.8	0	0	0	43:30.8	0	0	0
2009/11/18 10:48:31	43:30.8	16	0	0	43:30.8	10	0	10
2009/11/18 10:49:31	43:30.8	290664	0	0	43:30.8	29	10	24
2009/11/18 10:50:31	43:30.8	403304	0	403304	43:30.8	33	23	25
2009/11/18 10:51:31	43:30.8	408424	0	0	43:30.8	35	24	31
2009/11/18 10:52:31	43:30.8	408424	0	0	43:30.8	38	24	30
2009/11/18 10:53:31	43:30.8	402280	0	402280	43:30.8	35	22	24
2009/11/18 10:54:31	43:30.8	405352	0	0	43:30.8	43	24	40
2009/11/18 10:55:31	43:30.8	404328	0	0	43:30.8	47	29	38
2009/11/18 10:56:31	43:30.8	407400	0	0	43:30.8	49	32	41
2009/11/18 10:57:31	43:30.8	404328	0	0	43:30.8	51	34	35
2009/11/18 10:58:31	43:30.8	402280	0	0	43:30.8	48	33	43
2009/11/18 10:59:31	43:30.8	396136	0	0	43:30.8	48	31	39
2009/11/18 11:00:31	43:30.8	**410472** (1)	0	0	43:30.8	46	29	34
2009/11/18 11:01:31	43:30.8	407400	0	0	43:30.8	43	27	33
2009/11/18 11:02:31	43:30.8	408424	0	0	43:30.8	52	31	39
2009/11/18 11:03:31	43:30.8	408424	0	0	43:30.8	55	39	51
2009/11/18 11:04:31	43:30.8	406376	0	0	43:30.8	**57** (2)	39	41
2009/11/18 11:05:31	43:30.8	407400	0	0	43:30.8	56	36	47
2009/11/18 11:06:31	43:30.8	409448	0	0	43:30.8	52	34	47
2009/11/18 11:07:31	43:30.8	395112	0	0	43:30.8	51	31	43
2009/11/18 11:08:31	43:30.8	393064	0	0	43:30.8	43	9	9
2009/11/18 11:09:31	43:30.8	0	0	0	43:30.8	13	9	13
2009/11/18 11:10:31	43:30.8	0	0	0	43:30.8	13	13	13

HTTPSessionEMemoryBlockMaxSize.HighWaterMark の最大値は，図中 1.で示している 11:00:31 に取得した 410472 バイト（400.85 キロバイト）です。

この値を 64 キロバイト単位に切り上げると，448 キロバイトとなります。
HTTPSessionEMemoryBlockCount.HighWaterMark の最大値は図中 2.で示している 11:04:31 に取得した 57 です。

これら二つの値を掛け合わせた値が，HTTP セッションで利用する Explicit ヒープ領域のメモリサイズとなります。

(b)　コンテナで利用する Explicit ヒープ領域のメモリサイズ

コンテナで利用する Explicit ヒープ領域のメモリサイズは，JavaVM の稼働情報ファイルに出力される ContainerEHeapSize.HighWaterMark の値を使って確認します。

コンテナで利用する Explicit ヒープ領域のメモリサイズの稼働情報の出力例を次の図に示します。

図 7-14　コンテナで利用する Explicit ヒープ領域のメモリサイズの稼働情報の出力例

Date(+0900)	ContainerEHeapSize.StartTime(+0900)	ContainerEHeapSize.HighWaterMark	ContainerEHeapSize.LowWaterMark	ContainerEHeapSize.Current
2009/11/18 10:44:31	43:30.8	262144	0	262144
2009/11/18 10:45:31	43:30.8	262144	262144	262144
2009/11/18 10:46:31	43:30.8	262144	262144	262144
2009/11/18 10:47:31	43:30.8	262144	262144	262144
2009/11/18 10:48:31	43:30.8	1572864	262144	1572864
2009/11/18 10:49:31	43:30.8	5505024	1572864	5505024
2009/11/18 10:50:31	43:30.8	**6815744**	5505024	6815744
2009/11/18 10:51:31	43:30.8	**6815744**	6815744	6815744
2009/11/18 10:52:31	43:30.8	**6815744**	6815744	6815744
2009/11/18 10:53:31	43:30.8	**6815744**	6815744	6815744
2009/11/18 10:54:31	43:30.8	**6815744**	6815744	6815744
2009/11/18 10:55:31	43:30.8	**6815744**	6815744	6815744
2009/11/18 10:56:31	43:30.8	**6815744**	6815744	6815744
2009/11/18 10:57:31	43:30.8	**6815744**	6815744	6815744
2009/11/18 10:58:31	43:30.8	**6815744**	6815744	6815744
2009/11/18 10:59:31	43:30.8	**6815744**	6815744	6815744
2009/11/18 11:00:31	43:30.8	**6815744**	6815744	6815744
2009/11/18 11:01:31	43:30.8	**6815744**	6815744	6815744
2009/11/18 11:02:31	43:30.8	**6815744**	6815744	6815744
2009/11/18 11:03:31	43:30.8	**6815744**	6815744	6815744
2009/11/18 11:04:31	43:30.8	**6815744**	6815744	6815744
2009/11/18 11:05:31	43:30.8	**6815744**	6815744	6815744
2009/11/18 11:06:31	43:30.8	**6815744**	6815744	6815744
2009/11/18 11:07:31	43:30.8	**6815744**	6815744	6815744
2009/11/18 11:08:31	43:30.8	**6815744**	6815744	6815744
2009/11/18 11:09:31	43:30.8	**6815744**	6815744	6815744
2009/11/18 11:10:31	43:30.8	**6815744**	6815744	6815744

1

ContainerEHeapSize.HighWaterMark の最大値は図中 1.で示している 10:50:31 以降に取得している 6815744 バイト（6656 キロバイト）です。

これがコンテナで利用する Explicit ヒープ領域のメモリサイズとなります。

(c)　アプリケーションおよび JavaVM で利用する Explicit ヒープ領域のメモリサイズ

アプリケーションおよび JavaVM で利用する Explicit ヒープ領域のメモリサイズは JavaVM の稼働情報ファイルに出力される ApplicationEHeapSize.HighWaterMark の値を使って確認します。

アプリケーションおよび JavaVM で利用する Explicit ヒープ領域のメモリサイズの稼働情報の出力例を次の図に示します。

図 7-15　アプリケーションおよび JavaVM で利用する Explicit ヒープ領域のメモリサイズの稼働情報の出力例

Date(+0900)	ApplicationEHeapSize.StartTime(+0900)	ApplicationEHeapSize.HighWaterMark	ApplicationEHeapSize.LowWaterMark	ApplicationEHeapSize.Current
2009/11/18 10:44:31	43:30.8	0	0	0
2009/11/18 10:45:31	43:30.8	0	0	0
2009/11/18 10:46:31	43:30.8	0	0	0
2009/11/18 10:47:31	43:30.8	0	0	0
2009/11/18 10:48:31	43:30.8	655360	0	655360
2009/11/18 10:49:31	43:30.8	1703936	655360	1507328
2009/11/18 10:50:31	43:30.8	2293760	1507328	1572864
2009/11/18 10:51:31	43:30.8	2293760	1441792	2031616
2009/11/18 10:52:31	43:30.8	2293760	1638400	2293760
2009/11/18 10:53:31	43:30.8	**2424832**	1507328	1572864
2009/11/18 10:54:31	43:30.8	2162688	1245184	1769472
2009/11/18 10:55:31	43:30.8	1769472	786432	1179648
2009/11/18 10:56:31	43:30.8	1376256	851968	1114112
2009/11/18 10:57:31	43:30.8	1441792	917504	983040
2009/11/18 10:58:31	43:30.8	1441792	917504	1376256
2009/11/18 10:59:31	43:30.8	1507328	983040	1245184
2009/11/18 11:00:31	43:30.8	2031616	1048576	1310720
2009/11/18 11:01:31	43:30.8	1769472	983040	983040
2009/11/18 11:02:31	43:30.8	1441792	655360	786432
2009/11/18 11:03:31	43:30.8	917504	655360	851968
2009/11/18 11:04:31	43:30.8	983040	589824	720896
2009/11/18 11:05:31	43:30.8	917504	393216	458752
2009/11/18 11:06:31	43:30.8	589824	327680	458752
2009/11/18 11:07:31	43:30.8	524288	196608	196608
2009/11/18 11:08:31	43:30.8	196608	0	0
2009/11/18 11:09:31	43:30.8	0	0	0
2009/11/18 11:10:31	43:30.8	0	0	0

1

ApplicationEHeapSize.HighWaterMark の最大値は図中 1.で示している 10:53:31 に取得した 2424832 バイト（2368 キロバイト）です。

(d)　必要な Explicit ヒープ領域のメモリサイズ

(a)～(c)で示した稼働情報から求められる，必要な Explicit ヒープ領域のメモリサイズは次のようになります。

```
448(キロバイト)× 57 ＋ 6656(キロバイト) ＋ 2368(キロバイト)
＝ 34560(キロバイト)≒ 34メガバイト
```

明示管理ヒープの自動解放機能が有効な場合，「Java ヒープの Survivor 領域のサイズ×2」を加算した値が，最終的な Explicit ヒープ領域の見積もりサイズとなります。

7.12　アプリケーションで明示管理ヒープ機能を使用する場合のメモリサイズの見積もり

ユーザが作成したアプリケーションに Tenured 領域のメモリサイズ増加の要因になっているオブジェクトがある場合，該当するオブジェクトを Explicit ヒープに配置できます。ここでは，アプリケーションで明示管理ヒープ機能を使用する場合のメモリサイズの見積もりについて説明します。

ポイント

この節の説明は，J2EE サーバを含めた，JavaVM 上で動作するすべての Java アプリケーションに該当します。ただし，J2EE サーバで使用する Explicit ヒープだけを使用する場合，必ず読む必要はありません。必要に応じて参照してください。

7.12.1　アプリケーションで明示管理ヒープ機能を使用するかどうかの検討

Java ヒープのチューニングおよび「7.11.1　Explicit ヒープのメモリサイズの見積もり（J2EE サーバが使用するメモリサイズの見積もり）」の手順を実施しても FullGC が頻発する場合は，アプリケーションでの明示管理ヒープ機能の使用を検討します。

まず，FullGC の発生要因となっているオブジェクトを調査します。特定のオブジェクトを Explicit ヒープに配置することで FullGC 発生を抑止できる場合は，明示管理ヒープ機能を適用します。

ただし，Explicit ヒープに配置するオブジェクトは，ライフサイクルが既知であることが必要です。オブジェクトの生成のタイミングおよびオブジェクトが不要になるタイミングが Java プログラム上で明確な場合に，適用を検討してください。

明示管理ヒープ機能を適用するためには，自動配置設定ファイル，または明示管理ヒープ機能 API を使用します。明示管理ヒープ機能を使用した FullGC の抑止については，マニュアル「アプリケーションサーバ 機能解説 拡張編」の「7.　明示管理ヒープ機能を使用した FullGC の抑止」を参照してください。

7.12.2　見積もりの考え方

アプリケーションが利用する Explicit ヒープのメモリサイズは，運用開始前に見積もります。実際にアプリケーションを動作させた上で，明示管理ヒープ機能のイベントログを確認して見積もります。見積もり方法については，「7.12.3　アプリケーションが使用するメモリサイズ」を参照してください。

アプリケーション開発中および運用開始後に Explicit ヒープの使用状況などを調査する場合は，JavaVM ログファイルや Java API を使用します。開発中や運用開始後の調査の手順については，「7.14　明示管理ヒープ機能適用時に発生しやすい問題とその解決方法」を参照してください。

参考

メモリサイズの見積もりに利用できる情報は，稼働情報にも出力されます。稼働情報を利用したメモリサイズのチューニングについては「7.11.4　稼働情報による見積もり」を参照してください。

なお，ここでは稼働情報に出力される項目と，スレッドダンプに出力される項目の対応についても説明します。

7.12.3　アプリケーションが使用するメモリサイズ

運用を開始する前に，アプリケーションが使用する Explicit ヒープのメモリサイズを見積もり，-XX:HitachiExplicitHeapMaxSize オプションに設定します。

メモリサイズは，実際に明示管理ヒープ機能を実装したアプリケーションを動作させてテストを実行し，出力されたログを確認して見積もります。見積もった値を，本番で使用する実行環境の-XX:HitachiExplicitHeapMaxSize オプションに設定します。

ここでは，テストを実行する環境に応じた 2 種類の見積もり方法について説明します。

どちらの方法でテストを実行する場合も，次のことが前提になります。

見積もりの前提

- Explicit ヒープの最大サイズを十分に大きく設定してテストを実行してください。
- -XX:HitachiExplicitMemoryLogLevel オプションには「none」以外を設定してください。

見積もりで使用するイベントログの出力項目については，マニュアル「アプリケーションサーバ 機能解説 保守／移行編」の「5.11　明示管理ヒープ機能のイベントログ」を参照してください。

(1)　本番環境と同等の環境でテストを実行できる場合

本番環境と同等の環境でテストを実行できる場合，イベントログに出力された Explicit ヒープの確保済みメモリサイズの最大値を Explicit ヒープのメモリサイズとします。

確認手順を示します。

1. **テスト用の環境でアプリケーションを一とおり実行します。**
   アプリケーション実行中に GC が発生した時に，明示管理ヒープ機能のイベントログが出力されます。
2. **出力されたイベントログのすべてのレコード（行）のうち，Explicit ヒープの確保済みメモリサイズ（<EH_TOTAL>）が最大の値を確認します。**

この値を，Explicit ヒープのメモリサイズとしてください。ただし，明示管理ヒープの自動解放機能が有効な場合は，この値に「Survivor 領域サイズ × 2」を加算した値を，Explicit ヒープのメモリサイズとしてください。

(2) 本番環境よりも小さなスケールの環境でテストを実行する場合

本番環境よりも小さなスケールの環境でテストを実行する場合，本番環境で必要なメモリサイズは次の式で見積もります。

```
本番環境でのExplicitヒープサイズ
= (Explicitヒープの最大サイズ)/(Explicitメモリブロックの数) × 本番環境でのExplicitメモリブロック数
+ Survivor領域サイズ × 2※
```

注※

明示管理ヒープの自動解放機能が有効な場合に「Survivor 領域サイズ × 2」を加算します。

1. **テスト用の環境でアプリケーションを一とおり実行します。**
   アプリケーション実行中に GC が発生した時に，明示管理ヒープ機能のイベントログが出力されます。
2. **出力されたイベントログのすべてのレコード（行）のうち，Explicit ヒープの確保済みメモリサイズ（<EH_TOTAL>）が最大の値を確認します。また，同じレコードに出力されている有効な Explicit メモリブロックの数（<AC_NUM>と<DA_NUM>の合計）を確認します。**
3. **2.で確認した<EH_TOTAL>の値を（<AC_NUM>+<DA_NUM>）で割ります。**
   Explicit メモリブロック一つ当たりのおおよそのサイズを算出できます。
4. **3.で算出した値に，本番環境で予測される最大の Explicit メモリブロック数を掛けます。**

注

この方法は，開発環境と本番環境での Explicit メモリブロックの一つ当たりのサイズが同じであり，数だけが異なる場合に適用できます。

また，Explicit ヒープを複数の用途に利用している場合，用途ごと（見積もり対象の用途以外の Explicit メモリブロックを利用しない状態）で確認する必要があります。

ポイント

スレッドダンプに出力される項目，および稼働情報に出力される項目の対応を次の表に示します。

表 7-7 スレッドダンプおよび稼働情報の出力項目の対応

スレッドダンプの出力項目	稼働情報の出力項目	出力内容
<EH_TOTAL>	EHeapSize	Explicit ヒープの確保済みメモリサイズ

スレッドダンプの出力項目	稼働情報の出力項目	出力内容
<AC_NUM> + <DA_NUM>	EMemoryBlockCount	同じレコードに出力されている有効なExplicit メモリブロックの数

7.13 明示管理ヒープ機能の自動配置機能を使用した Explicit ヒープの利用の検討

ここでは，明示管理ヒープ機能の自動配置機能を使用した Explicit ヒープの利用の検討について説明します。

明示管理ヒープ機能を使用して Explicit ヒープ領域を利用する際，自動配置機能を使用することで，明示管理ヒープ機能を容易に使用できます。また，次に示すような場合は，自動配置機能を使用することをお勧めします。

7.13.1 アプリケーション内に Tenured 領域の増加原因のオブジェクトがある場合

アプリケーション内に Tenured 領域の増加原因のオブジェクトがある場合，自動配置設定ファイルを使用してオブジェクトを Explicit ヒープに配置することをお勧めします。オブジェクトの配置を検討した方がよい Java プログラムの例を次に示します。

```
01:package abcd.efg;
02:import java.util.HashMap;
03:// KVStorageのインスタンスは，長期間生存し続ける
04:class KVStorage {
05:  HashMap _map = new HashMap();
06:
07:  public void store(MyKey k,MyData d) {
08:    // ...前処理...
09:    _map.put(k,d);
10:    // ...後処理...
11:  }
12:
13:  public MyData load(MyKey k) {
14:    // ...前処理...
15:    MyData d = map.get(k);
16:    // ...後処理...
17:    return d;
18:  }
19:}
```

この設定例の場合，KVStorage がインスタンスフィールドに保持している HashMap クラスが Tenured 領域のメモリサイズ増加の要因となる長寿命オブジェクトとなります。このオブジェクトの生成先を Explicit ヒープへ変更する場合，次の例のように自動配置設定ファイルを指定します。

```
# 生成個所（メソッド名やクラス名），生成するクラス名 の対で記載。
abcd.efg.KVStorage.<init>, java.util.HashMap
```

この例のように自動配置設定ファイルを指定することで，Java プログラムの例の 5 行目の HashMap インスタンス（_map）の生成先が Java ヒープから Explicit ヒープに変更されます。また，store メソッド

で_map に格納した MyKey のインスタンス，および MyData のインスタンスも，順次 Explicit ヒープに移動されます。これらのインスタンスは，不要となった時点で JavaVM によって自動的に解放されます。

7.13.2 Tenured 領域利用済みサイズの増加原因が不明な場合

Survivor 領域のチューニングを実施しても，Tenured 領域利用済みサイズが増加し，それによる FullGC の発生間隔がシステムの要件を満たせない場合，Tenured 領域利用済みサイズの増加原因となるオブジェクトを Explicit ヒープへ生成します。Explicit ヒープへオブジェクトを生成するには，明示管理ヒープの自動配置設定ファイルを設定します。

Tenured 領域利用済みサイズの増加原因となるオブジェクトを調査する方法，および自動配置設定ファイルの設定方法について説明します。

(1) Tenured 領域利用済みサイズ増加の調査

実行中のアプリケーションに対して，jheapprof コマンドに-garbage オプションを指定し，Tenured 領域内不要オブジェクト統計機能を実行します。

Tenured 領域内不要オブジェクト統計機能の出力例を次に示します。これによって，Tenured 領域利用済みサイズ増加の原因となっているオブジェクト（Tenured 増加要因の基点オブジェクト）のクラス名のリストがスレッドダンプログファイルに出力されます。

```
Garbage Profile Root Object Information
----------------------------------------
*, java.util.HashMap # 35234568
*, java.util.WeakHashMap # 4321000
```

この出力例では，Tenured 領域利用済みサイズの増加原因として，java.util.HashMap のオブジェクトが 35,234,568 バイト，また java.util.WeakHashMap のオブジェクトが 4,321,000 バイトであることがわかります。Tenured 領域内不要オブジェクト統計機能の詳細については，マニュアル「アプリケーションサーバ 機能解説 保守／移行編」の「9.8　Tenured 領域内不要オブジェクト統計機能」を参照してください。

(2) 自動配置設定ファイルへの記載

Tenured 領域内不要オブジェクト統計機能の出力例のリスト部分（後半 2 行）を，自動配置設定ファイルへ入力します。自動配置設定ファイルの設定例を示します。

```
*, java.util.HashMap # 35234568
*, java.util.WeakHashMap # 4321000
```

この場合，プログラム中のすべての java.util.HashMap オブジェクト，および java.util.WeakHashMap オブジェクトは Explicit ヒープに生成されます。

また，これらのオブジェクトに格納したオブジェクトも順次 Explicit ヒープに移動します。しかし，Explicit ヒープへのオブジェクトの生成は，Java ヒープへのオブジェクトの生成よりも実行時にオーバーヘッドが掛かります。このため，オブジェクトの生成個所を絞り込むことで，実行時のオーバーヘッドを削減できます。

自動配置設定ファイルでは，「*」は「JavaVM 上で動作するすべてのクラス」を意味します。この設定例の場合，すべてのクラスでの java.util.HashMap および java.util.WeakHashMap のオブジェクトの生成先が Explicit ヒープになります。これによって，実際には Tenured 利用済みサイズ増加の原因ではないオブジェクトの生成先も Explicit ヒープとなり，オーバーヘッドが増加するおそれがあります。

「*」を指定したことによって，アプリケーションのスループットが要件を満たせなくなった場合は，Tenured 領域利用済みサイズ増加の原因となっているオブジェクトの生成個所を絞り込むことを検討してください。

システム運用者とアプリケーション開発者が異なる場合は，アプリケーション開発者への調査の依頼が必要です。アプリケーションの詳細な調査が困難な場合でも，自動配置設定ファイルでは生成個所を「すべてのクラス」,「特定のパッケージ」,「特定のクラス」，および「特定のメソッド」の４段階の粒度でオブジェクトの生成個所を指定できます。そのため，調査可能な範囲で絞り込みを実施して，自動配置設定ファイルを指定することによって，スループットが向上する場合もあります。

例えば，生成個所が「com.abc.defg」パッケージ下の場合，自動配置設定ファイルの設定例を次のように変更することによって，「すべてのパッケージのすべてのクラス」から，「com.abc.defg パッケージおよびサブパッケージのすべてのクラス」まで絞り込みができます。

```
com.abc.defg.*, java.util.HashMap # 35234568
com.abc.defg.*, java.util.WeakHashMap # 4321000
```

自動配置設定ファイルの指定方法の詳細については，マニュアル「アプリケーションサーバ 機能解説 拡張編」の「7.13.2　自動配置設定ファイルを使った明示管理ヒープ機能の使用」を参照してください。

(3)　Tenured 領域内不要オブジェクト統計機能によるアプリケーションの調査

自動配置設定ファイルの内容を基に，アプリケーションを調査する場合に，Tenured 領域内不要オブジェクト統計機能を利用します。

jheapprof コマンドに-garbage オプションを指定し，Tenured 領域内不要オブジェクト統計機能を実行することで，Tenured 増加要因の基点オブジェクトリスト，および Tenured 領域内不要オブジェクトの統計情報を拡張スレッドダンプに出力します。拡張スレッドダンプの出力例を次に示します。

```
Garbage Profile
---------------
________________Size__Instances__Class________________
            35234568      10648 java.util.HashMap
             5678900      10668 [Ljava.util.HashMap$Entry;
             4456788       7436 java.util.HashMap$Entry
             4321000        200 java.util.WeakHashMap
             1234568        190 [Ljava.util.WeakHashMap$Entry
1456788       9524 java.lang.String
```

```
1256788        6424 com.abc.defg.MyData;
 :
```

Tenured 領域内不要オブジェクト統計機能では，Tenured 増加要因の基点オブジェクトリストには出力されない，java.util.HashMap や java.util.WeakHashMap に格納されているオブジェクトも出力されます。また，各オブジェクトのインスタンス数も出力されます。

さらに，このログを複数回取得して，クラス別統計情報解析機能（jheapprofanalyzer コマンド）の入力ファイルとすることで，各オブジェクトサイズ，およびインスタンス数の時間の変化を調査できます。

これらの情報を基に，アプリケーションを調査してオブジェクト生成個所の絞り込みをします。Tenured 領域内不要オブジェクト統計機能の詳細については，マニュアル「アプリケーションサーバ 機能解説 保守／移行編」の「9.8　Tenured 領域内不要オブジェクト統計機能」を参照してください。クラス別統計情報解析機能の詳細については，マニュアル「アプリケーションサーバ 機能解説 保守／移行編」の「9.10 クラス別統計情報解析機能」を参照してください。

7.14　明示管理ヒープ機能適用時に発生しやすい問題とその解決方法

Explicit ヒープのメモリサイズや，明示管理ヒープ機能の利用のしかた，Java アプリケーションの設計方法，実行環境の設定などによっては，次の表に示す現象が発生することがあります。

表 7-8　明示管理ヒープ機能適用時に発生する現象と対処方法の参照先

項番	発生する現象	対処方法の参照先
1	Explicit ヒープに空きがなく，Explicit メモリブロックへのオブジェクトの生成に失敗する。	7.14.3
2	Explicit メモリブロックの生成に失敗する。	7.14.4
3	明示管理ヒープ機能を利用しても FullGC が頻発する。	7.14.5
4	Explicit メモリブロックの自動解放処理に長時間掛かっている。	7.14.6

これらの現象が発生しているかどうかは，明示管理ヒープ機能のイベントログから調査できます。調査方法については，「7.14.1　Explicit ヒープのある時点での利用状況（スナップショット）の調査」，および「7.14.2　利用状況の推移の調査」を参照してください。なお，明示管理ヒープ機能のイベントログの内容については，マニュアル「アプリケーションサーバ 機能解説 保守／移行編」の「5.11　明示管理ヒープ機能のイベントログ」を参照してください。なお，ここで説明する出力例では，見やすさのために改行した個所に¥を入れています。実際の出力内容に改行は入りません。

7.14.1　Explicit ヒープのある時点での利用状況（スナップショット）の調査

Explicit ヒープのある時点での利用状況（スナップショット）を調査する方法には，スレッドダンプを確認する方法と，Java API で情報を取得する方法があります。

- **スレッドダンプを確認する方法**

  cjdumpsv コマンドを実行すると，任意のタイミングでスレッドダンプを出力できます。スレッドダンプには，Explicit ヒープおよび各 Explicit メモリブロックの利用状況が出力されます。

  出力例を次に示します。

```
Explicit Heap Status
--------------------
 max 65536K, total 21376K, used 20480K, garbage 1234K (31.2% used/max, 95.8% used/total,
6.0% garbage/used), 1 spaces exist
 Explicit Memories(0x12345678)
  "EJBMgrData" eid=1(0x02f25610)/R, total 21376K, used 20480K, garbage 1234K (95.8% used/
total, 6.0% garbage/used, 0 blocks) Enable
```

  背景色付きの**太字**の部分が，Explicit ヒープおよび個別の Explicit メモリブロックの利用状況です。

  この例の場合，Explicit ヒープの最大サイズは 65,536 キロバイト，確保済み Explicit ヒープサイズは 21,376 キロバイトです。また，「EJBMgrData」という名称の Explicit メモリブロックのメモリ確保済みサイズは 21,376 キロバイト，利用済みサイズは 20,480 キロバイトであることがわかります。

- Java API で情報を取得する方法

  JavaVM の Java API を使用して，Explicit ヒープおよび Explicit メモリブロックの利用状況を取得できます。次に示す API を使用してアプリケーションを実装することで，任意の処理のタイミングで情報を取得できます。

  Explicit ヒープの利用状況を取得するメソッド

  JP.co.Hitachi.soft.jvm.MemoryArea.ExplicitMemory.getMemoryUsage()

  Explicit メモリブロック利用状況を取得するメソッド

  JP.co.Hitachi.soft.jvm.MemoryArea.ExplicitMemory.freeMemory()

  JP.co.Hitachi.soft.jvm.MemoryArea.ExplicitMemory.totalMemory()

  JP.co.Hitachi.soft.jvm.MemoryArea.ExplicitMemory.usedMemory()

7.14.2　利用状況の推移の調査

Explicit ヒープの利用状況の推移を調査する方法には，明示管理ヒープ機能のイベントログを確認する方法と，Java API で情報を取得する方法があります。Java API を使用する方法については，「7.14.1 Explicit ヒープのある時点での利用状況（スナップショット）の調査」で示した API を使用します。

ここでは，明示管理ヒープ機能のイベントログを使用した調査方法について説明します。

(1)　Explicit ヒープの利用状況の推移

JavaVM の-XX:HitachiExplicitMemoryLogLevel オプションに「normal」を指定します。これによって，次のタイミングで Explicit ヒープの利用状況が出力されます。

- GC 発生時（定期的）
- Explicit メモリブロックの明示解放処理時
- Explicit メモリブロックの自動解放処理時

また，JavaVM の-XX:HitachiExplicitMemoryLogLevel オプションに「verbose」を指定すると，「normal」で出力されるタイミングに加えて，次のタイミングで Explicit ヒープの利用状況が出力されるようになります。

- ExplicitMemory.newInstance メソッドなどによる Explicit メモリブロックへのオブジェクト生成時

出力例を次に示します。

```
[ENS]<Thu Oct 21 14:55:50 2007>[EH: 12672K->12800K(12800K/65536K)][E/F/D: 200/0/0][cause:GC]
[CF: 0]
[ENS]<Thu Oct 21 14:55:50 2007>[EH: 12800K->12800K(12800K/65536K), 0.1124626 secs][E/F/D: 20
0/0/0]¥
[DefNew::Eden: 0K->0K(243600K)][DefNew::Survivor: 0K->0K(17400K)][Tenured: 103400K->103400K(
556800K)]¥
[target:584K/0K/584K][cause:Migrate]
```

背景色付きの**太字**で示した「EH:」で始まる部分が，Explicit ヒープの利用状況を示します。ログに出力された内容のうち，Explicit ヒープの利用状況を示す行には，必ず背景色付きの**太字**に該当する情報が含まれます。この値をグラフなどに記してプロットすることによって，利用状況の推移を確認できます。

この例の場合，「cause:」で始まる部分に示すとおり，一つ目の[ENS]で始まる行は CopyGC（GC）発生時に出力されたログであり，二つ目の[ENS]で始まる行は Explicit メモリブロックの自動解放処理（Migrate）時に出力されたログであることがわかります。1 行目のログでは，CopyGC 発生前の Explicit ヒープの利用済みサイズは 12,672 キロバイト，発生後の Explicit ヒープの利用済みサイズは 12,800 キロバイトであることがわかります。2 行目のログでは，Explicit ヒープの利用済みサイズは 12,800 キロバイト，Explicit メモリブロックの自動解放処理に 0.1124626 秒掛かったことがわかります。

なお，利用状況を示すログの先頭は，[ENS]または[EVS]で開始されます。この文字列でイベントログをフィルタリングすると，利用状況を確認しやすくなります。

(2)　Explicit メモリブロックごとの利用状況の推移

JavaVM の-XX:HitachiExplicitMemoryLogLevel オプションに「verbose」を指定します。これによって，次のタイミングでサイズ変化のあった Explicit メモリブロックの利用状況が出力されます。

- Explicit メモリブロックへのオブジェクト移動時
- Explicit メモリブロックへのオブジェクト生成時

出力例を示します。

```
[ENS]<Thu Oct 21 14:55:50 2007>[EH: 11422K->12800K(12800K/65536K)][E/F/D: 200/0/0][cause:GC]
[CF: 0]
[EVS]["REM2" eid=2/R: 0K->88K(128K)]["REM3" eid=3/R: 30K->230K(256K)]["REM6" eid=6/R: 30K->2
00K(256K)]¥
["Session1" eid=8/R: 30K->250K(256K)]["Session2" eid=10/R: 30K->250K(256K)]
[EVS]["Session3" eid=12/R: 30K->510K(512K)]
```

背景色付きの**太字**で示した部分が，「"REM2"」という名称の一つの Explicit メモリブロックについての利用状況を示します。

また，-XX:HitachiExplicitMemoryLogLevel オプションに「verbose」を指定した場合，Explicit メモリブロックの解放時に，解放した Explicit メモリブロックの情報が出力されます。出力例を示します。

```
[ENS]<Tue Jul 24 01:23:51 2007>[EH: 12800K->11776K(11776K/65536K), 0.1129602 secs][E/F/D: 52
3/0/0]¥
[DefNew::Eden: 0K->0K(243600K)][DefNew::Survivor: 12K->0K(17400K)][Tenured: 103400K->103400
K(556800K)][cause:Reclaim]
[EVS]["REM2" eid=2/R: 320K]["BEM3" eid=5/B: 320K]["BEM1" eid=7/B: 384K]
```

背景色付きの**太字**で示した部分が，解放された「"REM2"」という名称の Explicit メモリブロックの情報を示しています。「320K」は，解放されたメモリのサイズ（確保済みだった Explicit メモリブロックのサイズ）です。

これらの値をグラフなどに記してプロットすることによって，Explicit メモリブロックごとの利用状況の推移を確認できます。なお，個々の Explicit メモリブロックのサイズは，解放するまで単調増加します。

7.14.3 Explicit ヒープあふれが発生した場合の確認と対処

Explicit ヒープあふれが発生した場合の確認と対処について説明します。

Explicit ヒープあふれとは，次の状態を示します。

- Explicit ヒープを最大サイズまで使い切った状態
- Explicit メモリブロック拡張時に OS からのメモリ確保に失敗する状態

Explicit ヒープあふれが発生すると，発生後に領域を拡張しようとした Explicit メモリブロックのサブ状態が，「Enable」から「Disable」に変わります。「Disable」になった Explicit メモリブロックには，オブジェクトを配置できません。

Explicit ヒープあふれが発生しているかどうかは，明示管理ヒープ機能のイベントログまたはスレッドダンプの内容から調査できます。また，Java API で取得した情報で確認することもできます。

Explicit ヒープあふれが発生した場合は，次の対処を実施してください。

Explicit ヒープあふれが発生した場合の対処

- Explicit ヒープの最大サイズを増やす。
  -XX:HitachiExplicitHeapMaxSize オプションの指定を変更します。
- Explicit ヒープの最大サイズに達していない状態であふれた場合は，OS からのメモリ確保可能サイズを増やす。
  アプリケーションサーバが利用できるメモリサイズを増やしてください。
- Explicit ヒープを大量に消費している原因を取り除く。

ここでは，Explicit ヒープあふれが発生しているかどうかの確認方法について説明します。

(1) 明示管理ヒープ機能のイベントログの調査

明示管理ヒープ機能のイベントログで調査をするためには，あらかじめ JavaVM の-XX:HitachiExplicitMemoryLogLevel オプションに「normal」を指定しておく必要があります。これによって，GC が発生するごとに，明示管理ヒープ機能のイベントログに Explicit メモリブロックの利用状況が出力されるようになります。

出力例を示します。

```
[ENS]<Thu Oct 21 14:55:50 2007>[EH: 12672K->12800K(12800K/65536K)][E/F/D: 200/0/0][cause:GC]
[CF: 0]
```

背景色付きの**太字**で示した部分が，Explicit メモリブロックの数を示しています。「E」および「D」は，Explicit メモリブロックのサブ状態である「Enable」および「Disable」を表します。「Disable」の Explicit メモリブロックがある場合は，Explicit ヒープあふれが発生しています。この例の場合は，「Enable」の Explicit メモリブロックが 200 個あり，「Disable」の Explicit メモリブロックはないことがわかります。なお，「Disable」の Explicit メモリブロックがある場合は，Explicit ヒープ最大サイズとの関係を確認してください。Explicit ヒープ最大サイズまでに余裕があるときには，OS からのメモリ確保に失敗していることが考えられます。

また，JavaVM の-XX:HitachiExplicitMemoryLogLevel オプションに「verbose」を指定した場合，Explicit メモリブロックのサブ状態が「Disable」になった要因も出力されます。

出力例を示します。

```
[EVO]<Tue Jul 24 01:23:51 2007>[alloc failed(Disable)][EH: 32760K(0K)/32768K/65536K][E/F/D:
321/0/1][cause:GC]¥
["BasicExplicitMemory-3" eid=3/B: 128K(0K)/128K][Thread: 0x00035a60]
[EVO][Thread: 0x00035a60] at ExplicitMemory.newInstance0(Native Method)
[EVO][Thread: 0x00035a60] at BasicExplicitMemory.newInstance(Unknown Source)
[EVO][Thread: 0x00035a60] at AllocTest.test(AllocTest.java:64)
[EVO][Thread: 0x00035a60] at java.lang.Thread.run(Thread.java:2312)
```

この例は，Explicit ヒープあふれが発生した場合の例です。

背景色付きの**太字**で示した部分のうち，[alloc failed(Disable)]が，Explicit メモリブロックのサブ状態が「Disable」になった要因を示します。["BasicExplicitMemory-3" eid=3/B: 128K(0K)/128K]は，「Disable」になった Explicit メモリブロックの情報を示します。また，[EVO][Thread: 0x00035a60]で始まる行は，イベントが発生した時のスタックトレースを表しています。ただし，GC によるオブジェクトの移動で Explicit ヒープあふれが発生した場合，スタックトレースは出力されません。

(2)　スレッドダンプで出力されたログファイルからの調査

cjdumpsv コマンドなどを使用してスレッドダンプを出力することによって，各 Explicit メモリブロックのサブ状態を出力できます。

出力例を次に示します。

```
Explicit Heap Status
--------------------
 max 65536K, total 21888K, used 20992K, garbage 1288K (32.0% used/max, 95.9% used/total, 6.1
% garbage/used), 2 spaces exist

 Explicit Memories(0x12345678)

  "EJBMgrData" eid=1(0x02f25610)/R, total 21376K, used 20480K, garbage 1234K (95.8% used/tot
al, 6.0% garbage/used, 0 blocks) Enable

  "ExplicitMemory-4" eid=4(0x02f45800)/B, total 512K, used 512K, garbage 54K (100.0% used/to
tal, 10.5% garbage/used, 0 blocks) Disable
```

背景色付きの**太字**で示した部分が，それぞれの Explicit メモリブロックのサブ状態を表しています。

(3)　Java の API からの調査

次に示すメソッドを使用して，Explicit メモリブロックのサブ状態を調査できます。

- JP.co.Hitachi.soft.jvm.MemoryArea.ExplicitMemory.isActive()
- JP.co.Hitachi.soft.jvm.MemoryArea.ExplicitMemory.isReclaimed()

これらのメソッド両方の戻り値が false の場合，その Explicit メモリブロックのサブ状態は Disable と判断できます。

7.14.4　Explicit メモリブロックの初期化が失敗した場合の確認と対処

Explicit メモリブロックの初期化が失敗した場合の確認と対処について説明します。

Explicit メモリブロックの数が最大になると，それ以上 Explicit メモリブロックを初期化できなくなります。

この場合は，Explicit メモリブロックの数を減らしてください。

ここでは，Explicit メモリブロックの初期化が失敗しているかどうかの確認方法について説明します。

(1)　明示管理ヒープ機能イベントログからの調査

明示管理ヒープ機能イベントログで調査をするためには，あらかじめ JavaVM の-XX:HitachiExplicitMemoryLogLevel オプションに「normal」を指定しておく必要があります。これによって，GC が発生するごとに，Explicit メモリブロックの初期化に失敗した回数が明示管理ヒープ機能イベントログに出力されるようになります

出力例を示します。

```
[ENS]<Thu Oct 21 14:55:50 2007>[EH: 12672K->12800K(12800K/65536K)][E/F/D: 200/0/0][cause:GC]
[CF: 0]
```

背景色付きの**太字**で示した部分が，前回の出力から今回の出力までの間に Explicit メモリブロックの初期化に失敗した回数です。この例の場合は，「0」です。初期化失敗が発生していない，問題のない状態です。

また，JavaVM の-XX:HitachiExplicitMemoryLogLevel オプションに「verbose」を指定した場合，Explicit メモリブロックの初期化失敗イベントについての情報も出力されます。

出力例を示します。

```
[EVO]<Tue Jul 24 01:23:51 2007>[Creation failed][EH: 32760K(0K)/32768K/65536K][E/F/D: 65535/
0/0][Thread: 0x00035a60]
[EVO][Thread: 0x00035a60] at ExplicitMemory.registerExplicitMemory(Native Method)
[EVO][Thread: 0x00035a60] at BasicExplicitMemory.<init>(Unknown Source)
```

```
[EVO][Thread: 0x00035a60] at AllocTest.test(AllocTest.java:64)
[EVO][Thread: 0x00035a60] at java.lang.Thread.run(Thread.java:2312)
```

背景色付きの**太字**で示した部分で，Explicit メモリブロック初期化に失敗したことが確認できます。また，[EVO][Thread: 0x00035a60]で始まる行は，イベントが発生した時のスタックトレースを示しています。

さらに，JavaVM の-XX:HitachiExplicitMemoryLogLevel オプションに「debug」を指定した場合，初期化に失敗したイベント以外の Explicit メモリブロック初期化イベントの詳細情報が出力されます。Explicit メモリブロック数が一定以上になると，初期化は失敗します。このため，初期化に失敗する前の初期化イベントの情報が，調査に役立つことがあります。

出力例を次に示します。

```
[EVO]<Tue Jul 24 01:23:51 2007>[Created]["BasicExplicitMemory-2" eid=2(0x1234568)/B][Thread:
 0x00035a60]
[EDO][Thread: 0x00035a60] at ExplicitMemory.registerExplicitMemory(Native Method)
[EDO][Thread: 0x00035a60] at BasicExplicitMemory.<init>(Unknown Source)
[EDO][Thread: 0x00035a60] at AllocTest.test(AllocTest.java:64)
[EVO][Thread: 0x00035a60] at java.lang.Thread.run(Thread.java:2312)
```

背景色付きの**太字**で示した部分で，Explicit メモリブロック初期化イベントであることが確認できます。また，[EDO][Thread: 0x00035a60]で始まる行は，イベントが発生した時のスタックトレースを示しています。

(2)　スレッドダンプで出力されたログファイルからの調査

スレッドダンプで出力された情報からは，Explicit メモリブロック初期化失敗の直接要因は確認できませんが，Explicit メモリブロックの個数は調べられます。

出力例を次に示します。

```
Explicit Heap Status
--------------------
 max 65536K, total 21888K, used 20992K, garbage 1288K (32.0% used/max, 95.9% used/total, 6.1
% garbage/used), 2 spaces exist

 Explicit Memories(0x12345678)

  "EJBMgrData" eid=1(0x02f25610)/R, total 21376K, used 20480K, garbage 1234K (95.8% used/tot
al, 6.0% garbage/used, 0 blocks) Enable

  "ExplicitMemory-4" eid=4(0x02f45800)/B, total 512K, used 512K, garbage 54K (100.0% used/to
tal, 10.5% garbage/used, 0 blocks) Disable
```

背景色付きの**太字**で示した部分が Explicit メモリブロックの個数を示しています。

(3)　Java の API からの調査

次に示すメソッドを使用して，Explicit メモリブロックの個数を調査できます。

- JP.co.Hitachi.soft.jvm.ExplicitMemory.countExplicitMemories()

ただし，この API では，Explicit メモリブロック初期化失敗の直接の要因は確認できません。

7.14.5 Explicit メモリブロック明示解放処理時に Java ヒープへのオブジェクト移動が発生した場合の確認と対処

Explicit メモリブロック明示解放処理時に，解放対象の Explicit ヒープ内のオブジェクトに対する参照があると，参照されているオブジェクトおよびそのオブジェクトから直接または間接的に参照されているオブジェクトが Java ヒープに移動します。オブジェクトは，Tenured 領域に優先的に移動されます。このため，移動が多いと Tenured 領域の利用済みサイズが増加して，FullGC 発生の要因となってしまいます。

Java ヒープへの移動が発生したかどうかは，JavaVM ログファイルの拡張 verbosegc 情報または明示管理ヒープイベントログで調査できます。

(1) 拡張 verbosegc 情報を使用した確認

明示管理ヒープ機能を利用しない場合，Tenured 領域の利用済みサイズの増加は，CopyGC だけで発生します。このため，N 回目の CopyGC 終了後の Tenured 領域利用済みサイズは，N+1 回目の CopyGC 開始前の Tenured 領域利用済みサイズと一致します。

これに対して，Explicit ヒープから Java ヒープへのオブジェクトの移動が発生した場合は，Explicit ヒープ解放時に Tenured 領域の利用済みサイズが増加します。この差分から，Explicit メモリブロック明示解放処理時にオブジェクトの移動が発生したことが確認できます。

N 回目の CopyGC 終了後の Explicit メモリブロック明示解放処理時に Java ヒープに移動したオブジェクトのサイズは，次の式で算出できます。

```
Explicitヒープから Javaヒープに移動したオブジェクトのサイズ
=N+1回目のCopy GC前のTenured領域利用済みサイズ
 -N回目のCopy GC後のTenured領域利用済みサイズ
```

(2) 明示管理ヒープのイベントログを使用した確認

JavaVM の-XX:HitachiExplicitMemoryLogLevel オプションに「none」以外を指定した場合，Explicit メモリブロックの明示解放処理についてのログが出力されます。このログでは，Explicit メモリブロック明示解放処理時の Tenured 領域利用済みサイズの増加を直接確認できます。

出力例を次に示します。

```
[ENS]<Tue Jul 24 01:23:51 2007>[EH: 12800K->11776K(11776K/65536K), 0.1129602 secs][E/F/D: 52
3/0/0]¥
[DefNew::Eden: 0K->0K(243600K)][DefNew::Survivor: 0K->0K(17400K)][Tenured: 103400K->103464K(
556800K)][cause:Reclaim]
```

背景色付きの**太字**で示した部分のうち，[cause:Reclaim]は，Explicit メモリブロック明示解放処理時に出力された情報であることを示します。また，[DefNew::Eden: 0K->0K(243600K)][DefNew::Survivor: 0K->0K(17400K)][Tenured: 103400K->103464K(556800K)]の部分は，Explicit メモリブロック明示解放処理時の Java ヒープの変化を示しています。この例の場合は，Tenured 領域のメモリサイズが 103,400 キロバイトから 103,464 キロバイトに，64 キロバイト分増えています。このことから，Explicit メモリブロックの明示解放処理で，64 キロバイトのオブジェクトが Java ヒープに移動していることがわかります。

また，JavaVM の-XX:HitachiExplicitMemoryLogLevel オプションに「verbose」を指定した場合，解放された Explicit メモリブロックについての情報も出力されます。これによって，どの Explicit メモリブロックの解放で Tenured 領域利用済みサイズが増加したかを確認できます。

出力例を次に示します。

```
[ENS]<Tue Jul 24 01:23:51 2007>[EH: 12800K->11776K(11776K/65536K), 0.1129602 secs][E/F/D: 52
3/0/0]¥
[DefNew::Eden: 0K->0K(243600K)][DefNew::Survivor: 0K->0K(17400K)][Tenured: 103400K->103464K(
556800K)][cause:Reclaim]
[EVS]["REM2" eid=2/R: 320K]["BEM3" eid=5/B: 320K]["BEM1" eid=7/B: 384K]
```

背景色付きの**太字**で示した部分が，解放された Explicit メモリブロックを示しています。出力内容から，Java ヒープに移動した 64 キロバイトのオブジェクトが，「REM2」「BEM3」「BEM1」のどれかの Explicit メモリブロックから移動したことがわかります。

さらに，JavaVM の-XX:HitachiExplicitMemoryLogLevel オプションに「debug」を指定した場合，明示解放処理をしたときに解放対象 Explicit ヒープ内のオブジェクトを参照していたオブジェクトが確認できます。

出力例を次に示します。

```
[EDO][eid=3: Reference to ClassZ(0x1234680), total 64K]
[EDO]   ClassU(0x1233468)(Tenured)
```

[eid=3: Reference to ClassZ(0x1234680), total 64K]の部分から，次のことがわかります。

- Java ヒープへ移動したオブジェクトは"ClassZ"のインスタンスである。
- "ClassZ"のインスタンスから参照されていることによって Java ヒープに移動したオブジェクトの合計サイズは 64 キロバイトである。

また，「ClassU(0x1233468)(Tenured)」の部分から，"ClassZ"のインスタンスを参照しているオブジェクトが"ClassU"のインスタンスであることがわかります。

これらの情報を基に，Explicit メモリブロック明示解放処理時にその Explicit メモリブロック内のオブジェクトへの参照をなくすように，Java プログラムを修正してください。

7.14.6　Explicit メモリブロックの自動解放処理が長時間化した場合の確認と対処

Explicit メモリブロックの自動解放処理が長時間化した場合の確認と対処について説明します。

Explicit メモリブロックの自動解放処理は，GC と同じタイミングで発生し，自動解放処理中は J2EE サーバの処理が停止します。このため，自動解放処理が長時間化すると，システム上問題となることがあります。

巨大なサイズを持つ Explicit メモリブロック（以降，巨大ブロックといいます）が生成されると，自動解放処理に長時間掛かるようになります。巨大ブロックは，アプリケーションの停止まで使用されるオブジェクトなど，FullGC でも回収されないオブジェクトが Explicit ヒープに配置されると生成されることがあります。このため，Explicit ヒープに配置しないほうがよいオブジェクトは Java ヒープに配置するようにして，巨大ブロックの生成を防止する必要があります。巨大ブロックが生成されて自動解放処理が長時間化してしまう現象については，マニュアル「アプリケーションサーバ 機能解説 拡張編」の「7.10.2　自動解放処理に掛かる時間を短縮する仕組み」，および「付録 B.1　Explicit メモリブロックの自動解放処理への影響」を参照してください。

巨大ブロックが生成されているかどうかは，スレッドダンプの内容から調査できます。

巨大ブロックが生成されている場合は，次に示す対処手順に従って，巨大ブロックの要因となるオブジェクトを Java ヒープに配置するように設定してください。

対処手順

1. Explicit メモリブロックへのオブジェクト移動制御機能の適用
2. 明示管理ヒープ機能適用除外クラス指定機能の適用
3. 巨大ブロックの要因となるオブジェクトの特定とそのオブジェクトの Explicit ヒープへの移動防止
4. Java ヒープ領域および Explicit ヒープ領域の再チューニング

ここでは，巨大ブロックが生成されているかどうかの確認方法と，対処方法について説明します。

(1)　巨大ブロックが生成されているがどうかの確認

eheapprof コマンドを実行して，スレッドダンプに出力された Explicit ヒープ情報を確認します。

出力例を次に示します。

```
  "NULL" eid=1(0x1000000000123456)/B, total 112K, used 55K, garbage 0K (49.2% used/total, 0.0% garbage/used, 0 blocks) Enable
  "NULL" eid=2(0x1000000000223456)/A, total 153744K, used 144766K, garbage 0K (94.2% used/total, 0.0% garbage/used, 0 blocks) Enable
  "ReferenceExplicitMemory-2" eid=3(0x1000000000323456)/R, total 112K, used 55K, garbage 0K (49.3% used/total, 0.0% garbage/used, 0 blocks) Enable
```

注

Explicit メモリブロックの名称の"NULL"は，一度自動解放処理を実施した Explicit メモリブロックであることを示します。

背景色付きの**太字**で示した「total」の部分が Explicit ヒープの確保済みメモリサイズを示します。

この例の場合，各 Explicit メモリブロックの「total」に示すメモリサイズは，eid=2 の Explicit メモリブロックは 153,744 キロバイトであり，eid=1 や eid=3 の Explicit メモリブロックの 112 キロバイトに比べて，極端に大きなサイズになっています。このことから，巨大ブロックが生成され，それが eid=2 の Explicit メモリブロックであることがわかります。

(2)　Explicit メモリブロックへのオブジェクト移動制御機能の適用

明示管理ヒープ機能のイベントログで Explicit ヒープの利用状況の推移を調査します。Explicit ヒープの利用状況の推移の調査方法については，「7.14.2　利用状況の推移の調査」を参照してください。

出力例を次に示します。

```
[ENS]<Thu Oct 21 14:55:50 2007>[EH: 12672K->172032K(172032K/196608K)][E/F/D: 200/0/0][cause:
Full GC][CF: 0]
[ENS]<Thu Oct 21 14:55:50 2007>[EH: 172032K->172032K(172032K/196608K), 0.1124626 secs][E/F/D
: 200/0/0]¥
[DefNew::Eden: 0K->0K(243600K)][DefNew::Survivor: 0K->0K(17400K)][Tenured: 103400K->103400K(
556800K)]¥
[target:584K/0K/584K][cause:Migrate]
```

背景色付きの**太字**で示した「EH:」で始まる部分が，Explicit ヒープの利用状況を示します。

この例の場合，「cause:」で始まる部分に示すとおり，1 行目は FullGC 発生時に出力されたログで，その直後にある 2 行目は Explicit メモリブロックの自動解放処理（Migrate）時に出力されたログです。このことから，FullGC の直後に自動解放処理が発生したことがわかります。1 行目のログの「EH:」で始まる部分から，FullGC 発生前の Explicit ヒープの利用済みサイズは 12,672 キロバイト，発生後の Explicit ヒープの利用済みサイズは 172,032 キロバイトであり，Explicit ヒープの使用サイズが大幅に増加していることがわかります。なお，このような現象（Explicit ヒープの使用サイズの大幅な増加）が発生していない場合は，手順(3)に進んでください。

この例のように，FullGC 発生時に Explicit ヒープの使用サイズが大幅に増加している場合は，-XX:ExplicitMemoryFullGCPolicy オプションに 1 を指定して，Explicit メモリブロックへのオブジェクト移動制御機能を適用してください。この機能を適用すると，FullGC 発生時に参照関係に基づくオブジェクトを Explicit メモリブロックへ移動しなくなります。Explicit メモリブロックへのオブジェクト移動制御機能については，マニュアル「アプリケーションサーバ 機能解説 拡張編」の「7.10　Explicit メモリブロックの自動解放処理に掛かる時間の短縮」を参照してください。

この機能を適用して，巨大ブロックの生成を防止できるようになった場合は手順(5)に，巨大ブロックがまだ生成されている場合は手順(3)に進んでください。

(3)　明示管理ヒープ機能適用除外クラス指定機能の適用

次の現象が発生している場合は，-XX:+ExplicitMemoryUseExcludeClass オプションを指定して，明示管理ヒープ機能適用除外クラス指定機能を適用してください。

- 自動解放処理直前に実施された FullGC の Explicit ヒープの利用状況で，Explicit ヒープの使用サイズに大幅な増加が見られない
- Explicit メモリブロックへのオブジェクト移動制御機能を適用（手順(2)を実施）しても，巨大ブロックが生成されている

明示管理ヒープ機能適用除外クラス指定機能を適用すると，特定のクラスのオブジェクトを Explicit ヒープへ移動させなくなります。特定のクラスのオブジェクトとは，システムで提供している明示管理ヒープ機能適用除外設定ファイル（sysexmemexcludeclass.cfg）に記述されているクラスのオブジェクトのことです。明示管理ヒープ機能適用除外クラス指定機能については，マニュアル「アプリケーションサーバ機能解説 拡張編」の「7.10　Explicit メモリブロックの自動解放処理に掛かる時間の短縮」を参照してください。

この機能を適用して，巨大ブロックの生成を防止できるようになった場合は手順(5)に，巨大ブロックがまだ生成されている場合は手順(4)に進んでください。

(4)　巨大ブロックの要因となるオブジェクトの特定とそのオブジェクトの Explicit ヒープへの移動防止

手順(3)で明示管理ヒープ機能適用除外クラス指定機能を適用しても，巨大ブロックがまだ生成されている場合は，次に示すオブジェクトが Explicit ヒープに移動されて，巨大ブロックが生成されていることがあります。これらのオブジェクトのうち生存し続けるオブジェクトは，Explicit メモリブロックの自動解放処理で回収されません。

- Java アプリケーション内のユーザ作成クラスのオブジェクト
- 使用しているフレームワークによって自動的に作成されるオブジェクト

このため，これらの一定期間で回収されないオブジェクトを Explicit ヒープに配置しても明示管理ヒープ機能の効果がないため，Java ヒープ（Tenured 領域）に配置する方が適切です。これらのオブジェクトのうち，巨大ブロックの要因となっているオブジェクトを特定し，Explicit ヒープへ移動しないようにします。

巨大ブロックの要因となるオブジェクトを特定する場合，オブジェクト解放率情報を調査します。オブジェクト解放率情報は，Explicit メモリブロックの自動解放処理で解放されたオブジェクトの割合です。eheapprof コマンドに-freeratio オプションを指定して実行すると，スレッドダンプに出力された Explicit ヒープ情報にオブジェクト解放率情報が出力されます。

出力例を次に示します。

```
  "NULL" eid=1(0x1000000000123456)/B, total 112K, used 55K, garbage 0K (49.2% used/total, 0.
0% garbage/used, 0 blocks) Enable
```

```
    deployed objects
    ________________Size__Instances__FreeRatio__Class________________
                   49256         10          0 [B
                    3680          4         20 package1.session.StandardManager
                   52936         14 total


  "NULL" eid=2(0x1000000000223456)/A, total 153744K, used 144766K, garbage 0K (94.2% used/to
tal, 0.0% garbage/used, 0 blocks) Enable
    deployed objects
    ________________Size__Instances__FreeRatio__Class________________
                77862918     433523         10 [C
                52622946     441714         10 java.lang.String
                12838192      39462         35 [B
                    3680          4         20 package1.session.StandardManager
                     104          4          0 framework.ut.impl.performList
               143327840     914707 total

  "ReferenceExplicitMemory-2" eid=3(0x1000000000323456)/R, total 112K, used 55K, garbage 0K
(49.3% used/total, 0.0% garbage/used, 0 blocks) Enable
    deployed objects
    ________________Size__Instances__FreeRatio__Class________________
                   49256          4          - [B
                    3416         10          - package3.ajp.RequestHandler
                      64          2          - java.lang.StringBuffer
                      64          1          - java.net.SocketInputStream
                      48          1          - [I
                      24          1          - [C
                   52872         19 total
```

注 1

Explicit メモリブロックの名称の"NULL"は，一度自動解放処理を実施した Explicit メモリブロックであることを示します。

注 2

クラス名の[B は Byte クラスの配列型，[C は Char クラスの配列型，[I は Integer クラスの配列型を示します。

背景色付きの**太字**で示した「total」の部分が Explicit ヒープの確保済みメモリサイズを，「FreeRatio」の部分がオブジェクト解放率情報を示します。

この例の場合，Explicit メモリブロックは，eid=1，eid=2，eid=3 の三つがあります。各 Explicit メモリブロックの「FreeRatio」に示すオブジェクト解放率情報を見てみると，eid=3 の Explicit メモリブロックのオブジェクト解放率情報は「-」で，自動解放処理が実行されなかったことがわかります。

まず，各 Explicit メモリブロックの「total」に示すメモリサイズを見てみます。eid=2 の Explicit メモリブロックは 153,744 キロバイトであり，eid=1 や eid=3 の Explicit メモリブロックの 112 キロバイトに比べて，極端に大きなサイズになっています。このことから，eid=2 の Explicit メモリブロックが巨大ブロックであることがわかります。

次に，巨大ブロック（eid=2）内のオブジェクトのクラスごとに，オブジェクト解放率と Explicit メモリブロック内のサイズを見てみます。この例では，五つのオブジェクトがあり，そのオブジェクトのクラス

には Java SE が提供しているクラス（「[B」,「[C」,「java.lang.String」）も含まれています。ほとんどの場合，Java SE が提供しているクラスは，巨大ブロックの要因となるオブジェクトから参照されて Explicit メモリブロックへ移動します。このため，巨大ブロックの要因となるオブジェクトを Explicit メモリブロックへ移動しないようにすれば，Java SE が提供しているクラスのオブジェクトの移動も防止できます。また，Java SE が提供しているクラスを明示管理ヒープ機能適用除外クラス指定機能の対象にすると，影響範囲が広く，理想的な Explicit メモリブロック上に配置されたオブジェクトにも制約が生じます。そのため，Java SE が提供しているクラスは，明示管理ヒープ機能適用除外クラス指定機能の対象にしないでください（ただし，Java SE が提供しているクラスのオブジェクトすべてを Java ヒープに配置しても問題ない場合は除きます）。

Java SE が提供しているクラス以外のクラスには,「package1.session.StandardManager」と「framework.ut.impl.performList」があります。「package1.session.StandardManager」のオブジェクトは eid=1 の Explicit メモリブロックでも使用されていますが，eid=1 は巨大ブロックになっていません。このことから，このクラスのオブジェクトは巨大ブロックの要因とならないことがわかります。これによって,「framework.ut.impl.performList」のオブジェクトが，巨大ブロックの要因となるオブジェクトであると特定できます。

巨大ブロックの要因となるオブジェクトを特定したあと，明示管理ヒープ機能適用除外クラス指定機能で使用する設定ファイルに，そのオブジェクトのクラスを記述してください。設定ファイルへの記述方法については，マニュアル「アプリケーションサーバ 機能解説 拡張編」の「7.13.3　設定ファイルを使った明示管理ヒープ機能の適用対象の制御」を参照してください。

(5)　Java ヒープ領域および Explicit ヒープ領域の再チューニング

オブジェクト移動制御機能や，明示管理ヒープ機能適用除外クラス指定機能を適用すると，オブジェクトの配置される領域が変わり，Java ヒープおよび Explicit ヒープのメモリサイズにも増減が発生します。このため，次の個所で説明している方法を参照して，再度，各領域のメモリサイズをチューニングしてください。

- 7.4　Java ヒープのチューニング
- 7.5　Java ヒープ内の Tenured 領域のメモリサイズの見積もり
- 7.6　Java ヒープ内の New 領域のメモリサイズの見積もり
- 7.7　Java ヒープ内に一定期間存在するオブジェクトの扱いの検討
- 7.8　Java ヒープの最大サイズ／初期サイズの決定
- 7.9　Java ヒープ内の Metaspace 領域のメモリサイズの見積もり
- 7.10　拡張 verbosegc 情報を使用した FullGC の要因の分析方法
- 7.11　Explicit ヒープのチューニング
- 7.12　アプリケーションで明示管理ヒープ機能を使用する場合のメモリサイズの見積もり
- 7.13　明示管理ヒープ機能の自動配置機能を使用した Explicit ヒープの利用の検討

7.15　G1GC の仕組み

ここでは，G1GC の仕組みを説明します。

7.15.1　G1GC の概要

GC は，プログラムが使用し終わったメモリ領域を自動的に回収して，ほかのプログラムが利用できるようにするための技術です。

GC の実行中は，プログラムの処理が停止します。このため，GC を適切に実行できるかどうかが，システムの処理性能に大きく影響します。

プログラムの中で new によって作成された Java オブジェクトは，JavaVM が管理するメモリ領域に格納されます。Java オブジェクトが作成されてから不要になるまでの期間を，Java オブジェクトの寿命といいます。

Java オブジェクトには，寿命の短いオブジェクトと寿命の長いオブジェクトがあります。サーバサイドで動作する Java アプリケーションの場合，リクエストやレスポンス，トランザクション管理などで，多くの Java オブジェクトが作成されます。これらの Java オブジェクトは，その処理が終わると不要になる，寿命が短いオブジェクトです。一方，アプリケーションの動作中使用され続ける Java オブジェクトは，寿命が長いオブジェクトです。

効果的な GC を実行するためには，寿命の短いオブジェクトに対して GC を実行して，効率良くメモリ領域を回収することが必要です。また，繰り返し使用される寿命の長いオブジェクトに対する不要な GC を抑止することが，システムの処理性能の低下防止につながります。これを実現するのが，世代別 GC です。

世代別 GC では，Java オブジェクトを，寿命が短いオブジェクトが格納される New 領域と，寿命が長いオブジェクトが格納される Tenured 領域に分けて管理します。New 領域はさらに，new によって作成されたばかりのオブジェクトが格納される Eden 領域と，1 回以上の GC の対象になり，回収されなかったオブジェクトが格納される Survivor 領域に分けられます。New 領域内で一定回数以上の GC の対象になった Java オブジェクトは，長期間必要な Java オブジェクトと判断され，Tenured 領域に移動します。

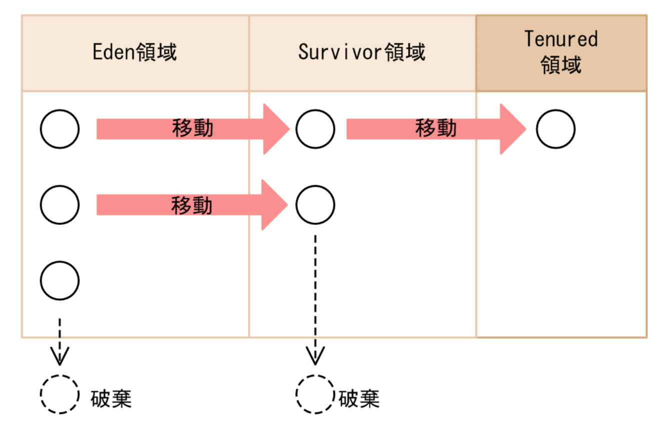
世代別 GC で管理するメモリ空間と Java オブジェクトの概要を次の図に示します。

図 7-16　世代別 GC で管理するメモリ空間と Java オブジェクトの概要

n回数以上のGCが実行される間使用されていたオブジェクト

1回以上n回未満のGCが実行される間使用されていたオブジェクト

1回目のGC実行時に使用されていなかったオブジェクト



（凡例）

○ ：Javaオブジェクト

GC：ガーベージコレクション

G1GC の世代別 GC で実行される GC には，次の 3 種類があります。

- YoungGC

  Eden 領域と Survivor 領域を対象にした GC です。Java オブジェクトの作成によって，Eden 領域を使い切ると発生します。

  注意

  アプリケーションサーバでは，SerialGC と G1GC（UseG1GC）が選択できます。ほかの ParallelGC，ConcurrentGC，IncrementalGC は使用できません。

- MixedGC

  Eden 領域と Survivor 領域と Tenured 領域の一部を対象にした GC です。Concurrent Marking と呼ばれるオブジェクトの解析処理に基づき発生します。Java オブジェクトの作成によって，Eden 領域を使い切ると発生します。

- FullGC

  Tenured 領域も含む，JavaVM 固有領域全体を対象にした GC です。Tenured 領域や Metaspace 領域，Humongous 領域を新たに確保できないと発生します。

一般的に，YoungGC と MixedGC の方が，FullGC よりも短い時間で処理できます。

次に，GC の処理について，ある Java オブジェクト（オブジェクト A）を例にして説明します。

Eden 領域で実行される処理

　オブジェクト A の作成後，1 回目の YoungGC または MixedGC が実行された時点で使用されていない場合，オブジェクト A は破棄されます。

　1 回目の YoungGC または MixedGC が実行された時点で使用されていた場合，オブジェクト A は Eden 領域から Survivor 領域に移動します。

Survivor 領域で実行される処理

Survivor 領域に移動したオブジェクト A は，その後何回か YoungGC または MixedGC が実行されると，Survivor 領域から Tenured 領域に移動します。移動する回数のしきい値は，JavaVM オプションや Java ヒープの利用状況によって異なります。図 7-16 では，しきい値を n 回としています。

Survivor 領域への移動後，n 回目未満の YoungGC または MixedGC が実行された時点でオブジェクト A が使用されていなかった場合，オブジェクト A はその YoungGC または MixedGC で破棄されます。

Tenured 領域で実行される処理

オブジェクト A が Tenured 領域に移動した場合，その後の YoungGC でオブジェクト A が破棄されることはありません。YoungGC は，Eden 領域と Survivor 領域だけを対象としているためです。MixedGC が発生した場合は，Tenured 領域内の一部のオブジェクトが破棄されます。

MixedGC で破棄されるオブジェクトの数よりも，Tenured 領域に移動するオブジェクトの数が多い場合，Tenured 領域の使用サイズは増加します。新たに Tenured 領域を確保できなくなると，FullGC が発生します。

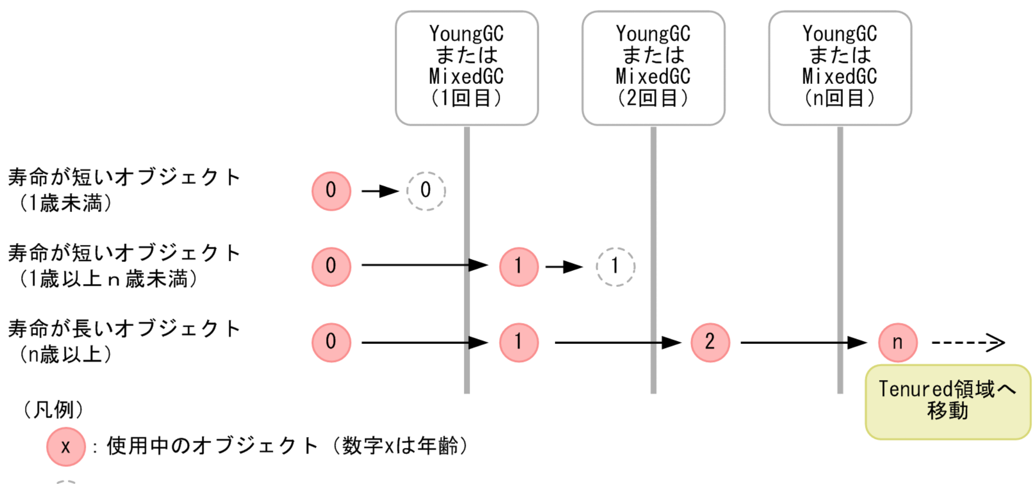
JavaVM のチューニングでは，JavaVM オプションでそれぞれのメモリ空間のサイズや割合を適切に設定することで，不要なオブジェクトが Tenured 領域に移動することを抑止します。これによって，FullGC が頻発することを防ぎます。

7.15.2　オブジェクトの寿命と年齢の関係

オブジェクトが YoungGC または MixedGC の対象になった回数をオブジェクトの年齢といいます。

オブジェクトの寿命と年齢の関係を次の図に示します。

図 7-17　オブジェクトの寿命と年齢の関係



アプリケーションが開始して初期化処理が完了したあとで，何度かの YoungGC または MixedGC が実行されると，長期間必要になる寿命の長いオブジェクトは Tenured 領域に移動します。このため，アプリ

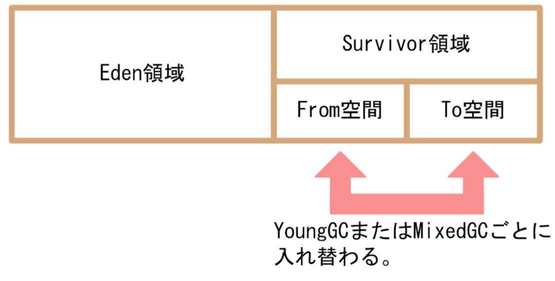
ケーションの開始後しばらくすると，Java ヒープの状態は安定し，作成される Java オブジェクトとしては，寿命が短いオブジェクトが多くなります。特に，New 領域のチューニングが適切にできている場合，Java ヒープが安定したあとの大半のオブジェクトは，1 回目の YoungGC または MixedGC で回収される，寿命が短いオブジェクトになります。

7.15.3 New 領域を対象とした GC の仕組み

JavaVM では，YoungGC または MixedGC の対象になる New 領域のメモリ空間を，Eden 領域，Survivor 領域に分けて管理します。さらに，Survivor 領域は，From 空間と To 空間に分けられます。From 空間と To 空間は，同じメモリサイズです。

New 領域の構成を次の図に示します。

図 7-18 New 領域の構成



Eden 領域は，new によって作成されたオブジェクトが最初に格納される領域です。プログラムで new が実行されると，Eden 領域のメモリが確保されます。

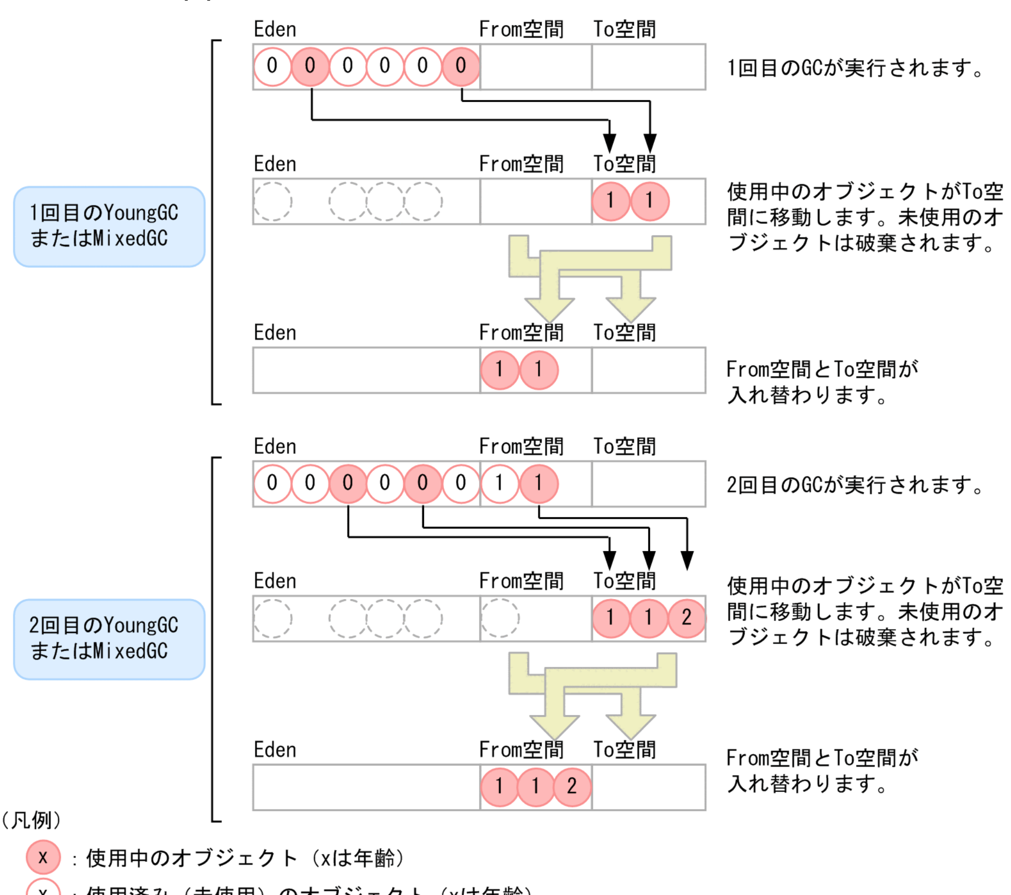
Eden 領域がいっぱいになると，YoungGC または MixedGC が実行され，次の処理が実行されます。

1. Eden 領域および Survivor 領域の From 空間にある Java オブジェクトのうち，使用中の Java オブジェクトが，Survivor 領域の To 空間にコピーされます。使用されていない Java オブジェクトは破棄されます。
2. Survivor 領域の To 空間と From 空間が入れ替わります。

この結果，Eden 領域と To 空間は空になり，使用中のオブジェクトは From 空間に存在することになります。

YoungGC または MixedGC 実行時に発生するオブジェクトの移動を次の図に示します。

図 7-19　CopyGC 実行時に発生するオブジェクトの移動

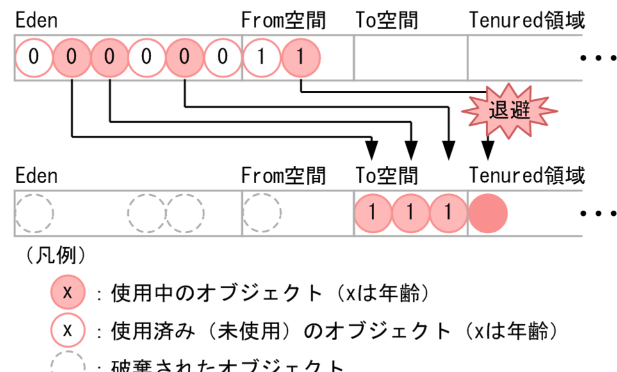


このようにして，使用中のオブジェクトは，YoungGC または MixedGC が発生するたびに，From 空間と To 空間を行ったり来たりします。ただし，寿命の長いオブジェクトを行き来させ続けると，移動処理の負荷などが問題になります。これを防ぐために，From 空間と To 空間で Java オブジェクトを入れ替える回数にしきい値を設定して，年齢がしきい値に達した Java オブジェクトは Tenured 領域に移動させるようにします。

7.15.4　オブジェクトの退避

年齢がしきい値に達していない Java オブジェクトが Tenured 領域に移動することを，退避といいます。退避は，YoungGC または MixedGC 実行時に Eden 領域および From 空間で使用中のオブジェクトが多くなり，移動先である To 空間のメモリサイズが不足する場合に発生します。この場合，To 空間に移動できなかったオブジェクトが，Tenured 領域に移動します。

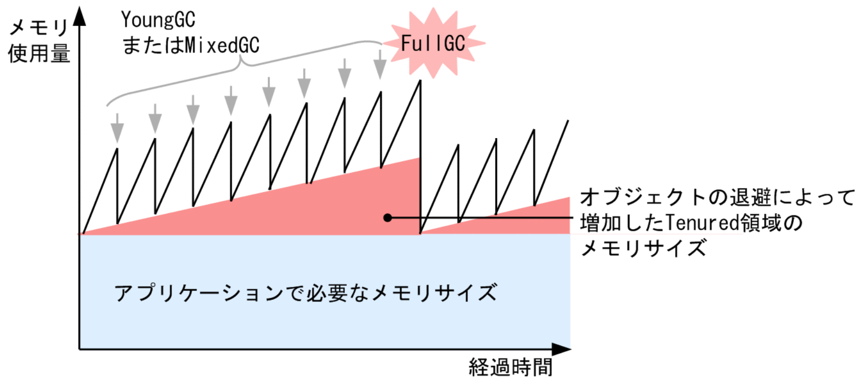
図 7-20　オブジェクトの退避



オブジェクトの退避が発生した場合，Tenured 領域に本来格納されないはずの寿命の短いオブジェクトが格納されます。これが繰り返されると，YoungGC または MixedGC で回収されるはずのオブジェクトがメモリ空間に残っていくため，Java ヒープのメモリ使用量が増加していき，最終的には FullGC が発生します。

オブジェクトの退避が発生した場合のメモリ使用量の変化について，次の図に示します。

図 7-21　オブジェクトの退避が発生した場合のメモリ使用量の変化



FullGC では，システムが数秒から数十秒停止することがあります。

したがって，メモリ空間の構成とメモリサイズを検討するときには，オブジェクトの退避が発生しないように，Eden 領域と Survivor 領域のバランスを検討する必要があります。

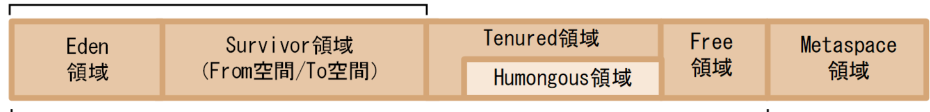
7.15.5　メモリ空間とリージョンの関係

G1GC で管理する Java ヒープ領域は，オブジェクトの寿命が短いオブジェクトが格納される New 領域と，寿命が長いオブジェクトが格納される Tenured 領域に分けて管理します。このうち New 領域はさらに 2 つの領域に分けて管理されており，作成されたばかりのオブジェクトが格納される Eden 領域と，1 回以上の GC の対象になり，回収されなかったオブジェクトが格納される Survivor 領域に分けられます。New 領域内で一定回数以上の GC の対象になったオブジェクトは，寿命が長いオブジェクトと判断され，Tenured 領域に移動します。Survivor 領域では使用中のオブジェクトと使用済みのオブジェクトが混在

する From 空間と，使用中のオブジェクトだけが存在する To 空間が必要なタイミングで動的に確保されます。タイミングの違いから，From 空間と To 空間はどちらか一方だけが存在します。

アプリケーション実行中の G1GC で管理するメモリ空間の構成を次の図に示します。なお，Eden 領域，Survivor 領域，Tenured 領域，Free 領域を合わせた領域を Java ヒープ領域といいます。

図 7-22　アプリケーション実行中の G1GC で管理するメモリ空間の構成



G1GC では，Tenured 領域があらかじめ割り当てられていません。Tenured 領域にオブジェクトを移動させる場合，Free 領域のリージョンを Tenured 領域に割り当ててオブジェクトを移動します。また，G1GC では GC 後に New 領域に対して拡張や縮小をします（以降，リサイズ）。リサイズで New 領域を拡張する場合は，Free 領域のリージョンに割り当て，縮小する場合は New 領域のリージョンを回収し，Free 領域に割り当てます。サイズの大きなオブジェクトを作成する場合は，Humongous 領域にオブジェクトが格納されます。

各領域の用途を次に示します。

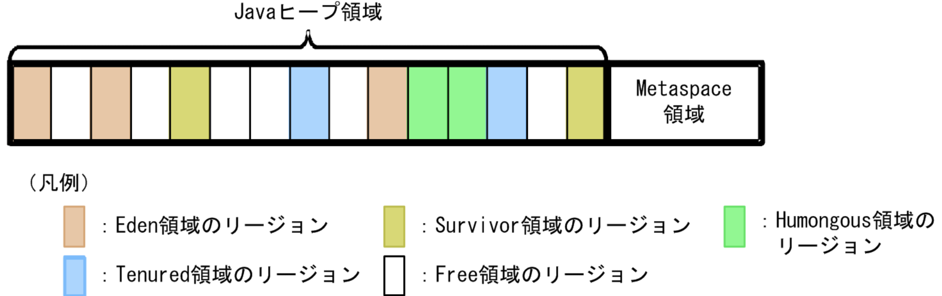
- New 領域
  寿命が短いオブジェクトが格納される領域です。Eden 領域と Survivor 領域から構成されます。GC 後に次の GC に向けてリサイズされます。詳細については，「7.15.8　YoungGC」を参照してください。
- Eden 領域
  作成されたオブジェクトが最初に格納される領域です。
- Survivor 領域
  New 領域に格納されていたオブジェクトのうち，GC 実行時に回収されなかったオブジェクトが格納される領域です。Survivor 領域は From 空間と To 空間で構成されます。From 空間と To 空間は同じサイズです。
- Tenured 領域
  寿命が長いオブジェクトが格納される領域です。Survivor 領域で指定回数を超えて GC 実行対象になり，回収されなかったオブジェクトが移動します。
- Humongous 領域
  Tenured 領域の一部でサイズの大きなオブジェクトを格納する領域です。Humongous 領域のオブジェクトは，FullGC または Remark フェーズで回収されます。
- Free 領域
  New 領域や Tenured 領域に割り当てられていない領域です。Tenured 領域のリージョンが足りなくなった場合や GC 後の New 領域をリサイズする場合に，New 領域や Tenured 領域に割り当てられます。

- Metaspace 領域

  ロードされた class などの情報を格納する領域です。なお，Metaspace 領域はリージョン単位で管理されていません。Metaspace 領域のオブジェクトは FullGC でだけ回収されます。

図 7-22 は Java ヒープ領域の各領域を連続領域として表現していますが，実際には各領域は連続領域として確保されていません。G1GC では Java ヒープ領域をリージョンと呼ばれるブロック単位で管理しており，次の図のように領域が分散して確保されています。

図 7-23　実際のメモリ空間の構造



ここではそれぞれの領域のリージョンを領域名＋リージョンと表記します。例えば，Eden 領域のリージョンを Eden リージョン，New 領域のリージョンは New リージョンと表記します。

7.15.6　リージョンの使われ方

オブジェクトが作成されると，最初に Eden リージョンに格納されます。1 つのオブジェクトのサイズが，リージョンサイズの半分を越えている大きいオブジェクト※の場合は，1 つのオブジェクトに対し，連続した複数のリージョンが割り当てられ格納されます。連続した複数のリージョンは Free 領域から確保され，Humongous 領域に割り当てられます。Humongous 領域は FullGC でだけ回収の対象となるため，サイズの大きなオブジェクトが多数作成されるような場合には，G1GC は不向きとなります。詳細については，「7.16.3(1)　FullGC の発生を抑止する考え方」を参照してください。

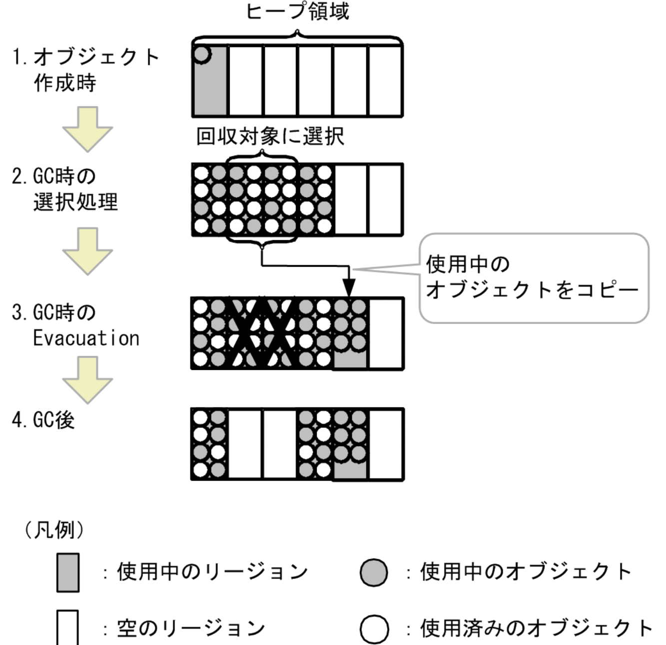
また，1 つのリージョンのサイズは，起動時に Java ヒープ領域の初期サイズの比率から求められ，起動中は固定です。1 つのリージョンの最小サイズは 1 メガバイトであり，最大サイズは 32 メガバイトです。

注※

ここでいうサイズの大きな 1 つのオブジェクトとは，多数のインスタンスフィールドを持つオブジェクトや長大な配列を持つオブジェクトのことを指します。リージョンのサイズが 1 メガバイトの場合，512 キロバイトを超えるようなオブジェクトを指し，例えば約 13 万個の int 型のインスタンスフィールドを持つようなオブジェクトや約 50 万個の要素数のバイト配列を持つオブジェクトなどが当てはまります。

次にリージョンの使用から回収までの流れを示します。

図 7-24　リージョンの使われ方



上図はリージョンの使われ方を示した図です。図の 1.～4.の処理の詳細を次に示します。

1. オブジェクト作成時

   オブジェクトが作成される場合，Eden リージョンにオブジェクトを格納します。サイズの大きなオブジェクトを作成する場合は，Free リージョンを Humongous リージョンに割り当てオブジェクトを格納します。Eden リージョンに空き領域がなくなると，新しい Eden リージョンにオブジェクトを格納し，Humongous リージョンに空き領域がなくなると，新たに Free リージョンから Humongous リージョンを割り当て，オブジェクトを格納します。

2. GC 時の選択処理

   GC の要件を満たすと，実行する GC 方式の基準に従って，対象とするリージョンを選択します。

3. GC 時の Evacuation

   回収の対象となったリージョン内で使用中のオブジェクトは，ほかのリージョンにコピーされます。この処理を Evacuation といいます。Evacuation 後のリージョンは，使用済みのオブジェクトとコピー済みである不要となったオブジェクトだけ格納されているため，リージョン単位で回収されます。

4. GC 後

   回収されたリージョンは Free リージョンとなり，再利用されます。

7.15.7　G1GC で実行される GC

G1GC の世代別 GC で実行される GC には，次の 3 種類があります。

1. YoungGC
   New 領域を対象とする GC です。YoungGC には YoungGC(normal)と YoungGC 中にマーキングをする YoungGC(initial-mark)があります。詳細については，「7.15.8　YoungGC」を参照してください。なお，単に YoungGC と表記した場合，YoungGC(normal)と YoungGC(initial-mark)の両方に当てはまる事項となります。また，YoungGC(normal)と YoungGC(initial-mark)を区別して表す場合は，“(normal)”と“(initial-mark)”を明記します。ただし，ログファイルの GC 種別には YoungGC(normal)は YoungGC という種別で出力します。YoungGC は Java オブジェクトの作成によって，Eden 領域を使い切ると発生します。
2. MixedGC
   New 領域と Tenured 領域を対象とする GC です。Tenured リージョンは目標停止時間に併せて部分的な範囲を対象とします。詳細については，「7.15.10　MixedGC」を参照してください。MixedGC は YoungGC 同様，Java オブジェクトの作成によって，Eden 領域を使い切ると発生しますが，Concurrent Marking と呼ばれるオブジェクトが使用中かどうかの解析処理の結果に基づいて発生します。詳細については，「7.15.9　Concurrent Marking（CM)」を参照してください。そのため，解析が十分にされていない場合や解析の結果 MixedGC の効果が低いと予測される場合は，YoungGC(normal)が発生します。
3. FullGC
   Tenured 領域や Metaspace 領域，Humondous 領域を含む，JavaVM 固有領域全体を対象にした GC です。詳細については，「7.15.11　FullGC」を参照してください。Tenured 領域や Metaspace 領域，Humongous 領域を確保できなかった場合に発生します。

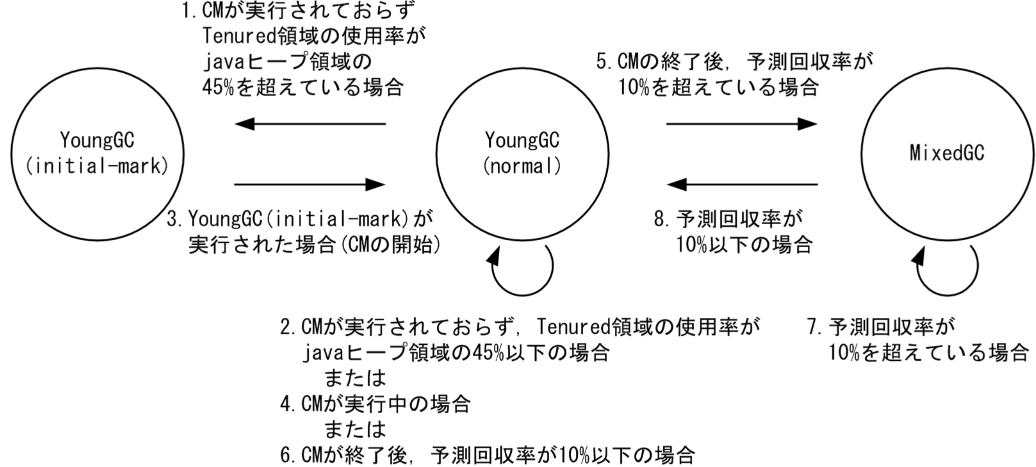
それぞれの GC が対象とする領域をまとめると次の図となります。

図 7-25　各 GC の対象範囲

New領域
Eden領域
Survivor領域
From空間
To空間
Tenured領域
Humongous領域
Metaspace領域
Free領域
YoungGC
MixedGC
FullGC
目標停止時間に合わせて変動

また，各 GC の状態遷移図を次の図に示します。

図 7-26　各 GC の状態遷移図



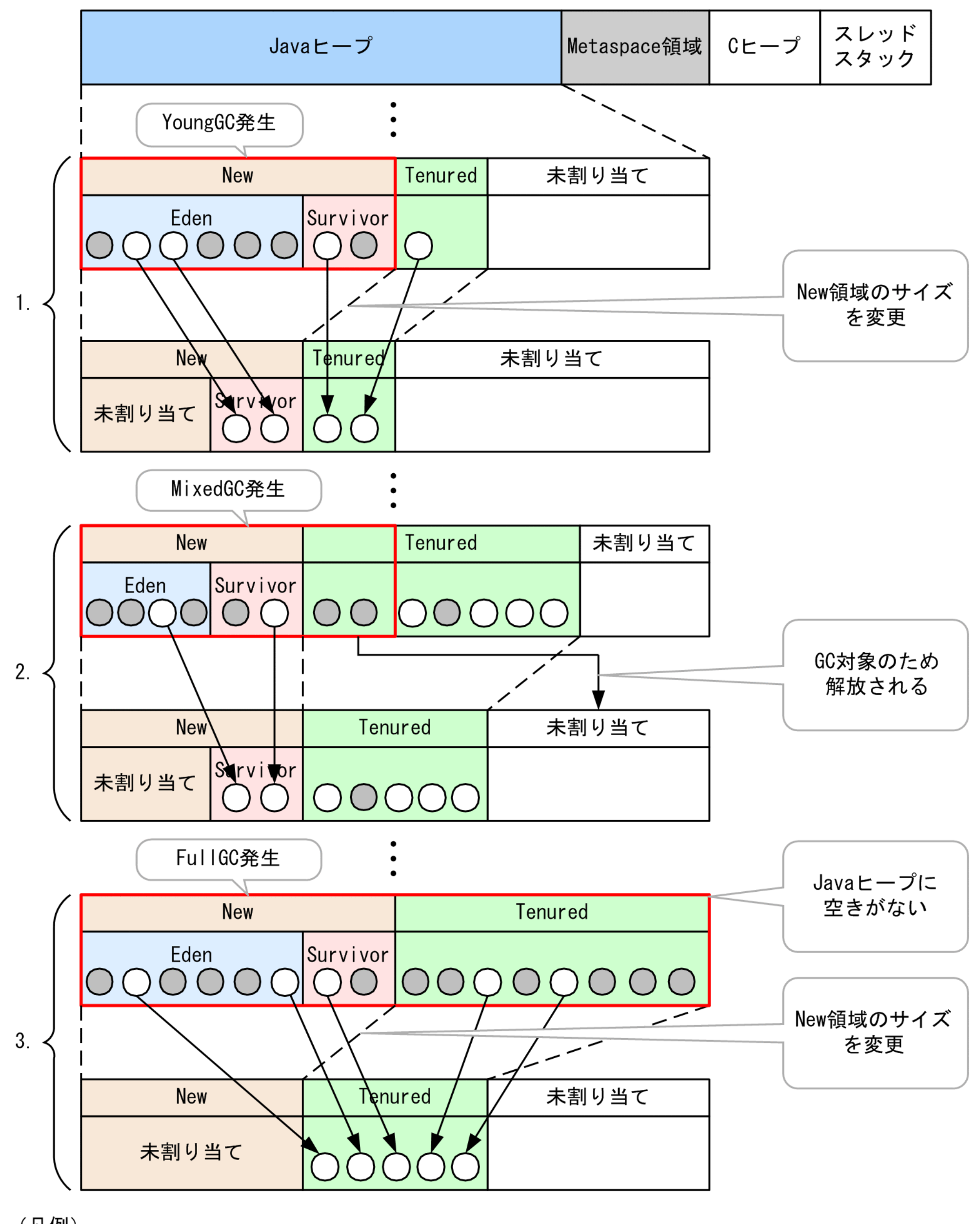
1. YoungGC(normal)実行後，CM が実行されておらず，Tenured 領域の使用率が Java ヒープ領域の 45%を超えた場合は，次の GC では YoungGC(initial-mark)を実行する状態に遷移します。
2. YoungGC(normal)実行後，CM が実行されておらず，Tenured 領域の使用率が Java ヒープ領域の 45%以下の場合は，次の GC も YoungGC(normal)を実行する状態のままです。
3. YoungGC(initial-mark)実行後，次の GC では YoungGC(normal)と CM を並行して実行する状態に遷移します。
4. CM が実行中の場合は，YoungGC(normal)を実行する状態のままです。
5. CM 終了直後の YoungGC(normal)で予測回収率が 10%を超えている場合，次の GC では MixedGC を実行する状態に遷移します。
6. 予測回収率が 10%以下の場合，次の GC も YoungGC(normal)を実行する状態のままです。
7. MixedGC 実行後，予測回収率が 10%を超えている場合，次の GC では MixedGC を実行する状態のままです。
8. MixedGC 実行後，予測回収率が 10%以下の場合，次の GC では YoungGC(normal)を実行する状態に遷移します。

ただし，次の場合 GC の状態によらず条件を満たすと対応する GC を実行します。

- YoungGC または，MixedGC を実行しても空き領域を確保できなかった場合，FullGC を実行します。FullGC 実行後は YoungGC(normal)を実行する状態に遷移します。
- サイズの大きなオブジェクトを確保時，CM が実行されておらず，Tenured 領域の使用サイズと確保するオブジェクトのサイズの合計が Java ヒープ領域の 45%を超えた場合，YoungGC(initial-mark)を実行します。

次に，G1GC の処理の流れについて，Java オブジェクトを例にして説明します。

図 7-27　G1GC の流れ



1. YoungGC

   上図のように New 領域に割り当てたリージョンに空きがなくなると YoungGC が発生します。YoungGC では使用中のオブジェクトは Survivor 領域に割り当てたリージョンに移動し，使用済みのオブジェクトはリージョンごと解放します。また，SerialGC の CopyGC と同様，YoungGC 発生時に使用中のオブジェクトは Survivor 領域に割り当てたリージョン間を移動し続け，ある一定の回数を移動すると Tenured 領域に割り当てたリージョンに移動します。YoungGC 後上図のように GC にかかった時間から，次の GC にかかる時間を予測し，New 領域のサイズを変更します。上図の場合は，予測した時間より GC にかかる時間が長いため，New 領域を縮小した場合の例です。

2. MixedGC

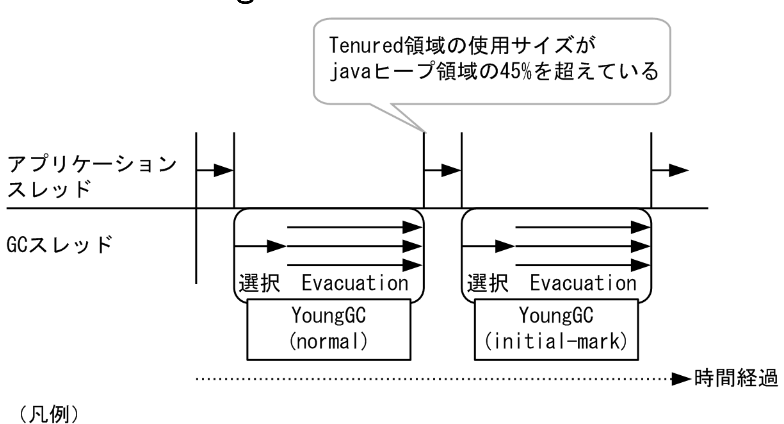
Tenured 領域の使用率が増加すると，MixedGC が発生します。MixedGC では New 領域に割り当てたリージョンに加えて，目標停止時間内に収まる範囲で，一部の Tenured 領域に割り当てたリージョンを GC の対象とします。この一部の Tenured 領域に割り当てたリージョンは，アプリケーションと並行して実行しているオブジェクトが使用中かどうかの解析情報に基づき，解放されるサイズが大きいと予測されるリージョンから優先して GC の対象となります。そのため，オブジェクトの情報解析が十分にされていない場合や解析の結果 MixedGC の効果が低い場合は，MixedGC は発生しません。

3. FullGC

Java ヒープ内のリージョンに空きがなくなり，MixedGC が発生しない場合，Java ヒープ全体を対象として FullGC が発生します。

7.15.8 YoungGC

図 7-28　YoungGC の流れ



(1)　実行契機

1. YoungGC(normal)

Eden 領域にオブジェクトを確保できなかった場合に発生します。また，サイズの大きなオブジェクトを Humongous 領域に確保できなかった場合にも発生します。

2. YoungGC(initial-mark)

直前に実行された YoungGC(normal)終了時に，Tenured 領域の使用サイズが Java ヒープ領域の 45%を超えていた場合，1.の実行契機を満たすと YoungGC(initial-mark)が実行されます。また，サイズの大きなオブジェクトを確保時，Tenured 領域の使用サイズとオブジェクトの確保サイズの合計サイズが Java ヒープ領域の 45%を超えた場合，YoungGC(initial-mark)が実行されます。

(2)　対象範囲

New 領域

(3)　処理内容

- YoungGC が発生するとシングルスレッドでリージョンの選択処理をし，その後マルチスレッドで Evacuation をします。
- YoungGC の Evacuation では Eden 領域と From 空間内の使用中のオブジェクトを To 空間または Tenured 領域に移動し，Eden 領域と From 空間を回収します。移動や回収の仕組みは CopyGC の仕組みと同じです。CopyGC の詳細については，「7.2.3　CopyGC の仕組み」を参照してください。
- YoungGC では，これまでに発生した GC の GC 停止時間の統計から予測をし，次回の GC の予測停止時間が目標停止時間内に収まるように New 領域のサイズを変更します。
- New 領域は最小サイズと最大サイズが存在し，その範囲内でリサイズをします。New 領域は全 GC で対象となるため，GC 停止時間は New 領域が最小サイズの場合にかかる GC 停止時間より短くすることはできません。
- CM が終了後の YoungGC では，予測回収サイズが Java ヒープ領域の 10%を超えていた場合，次回の GC を MixedGC にするか判定します。次回の GC に MixedGC が選択された場合，Tenured リージョンを多く対象とするように予測して New 領域のサイズを変更します。
- YoungGC(initial-mark)の Evacuation では Evacuation 中にローカル変数や使用中のオブジェクトから直接参照されているオブジェクトにマーク付けをします。このマーキングの結果は，Concurrent Marking に利用されます。Concurrent Marking の処理の詳細については，「7.15.9　Concurrent Marking（CM）」を参照してください。

(4)　処理結果

Eden 領域：オブジェクトが回収され，空になります。GC 後リサイズされます。

Survivor 領域：From 空間のオブジェクトが回収され，空になります。GC 後リサイズされます。

Tenured 領域：長期間必要と判断されたオブジェクトが Tenured 領域に移動します。

Humongous 領域：変化はありません。

Metaspace 領域：変化はありません。

Free 領域：GC 後のリサイズによって，増減します。

(5)　アプリケーションの停止の有無

停止します。

(6)　ほかの GC との関係

CM：YoungGC 中に実行されません。

MixedGC：YoungGC 中に実行されません。

FullGC：YoungGC 中に実行要件を満たすと，YoungGC を中止して実行されます。

(7) 補足

- 関連オプション

  Evacuation をするスレッド数は-XX:ParallelGCThreads オプションで変更することができます。スレッド数を増やすと YoungGC にかかる時間が小さくなります。また，オプションを指定しない場合，スレッド数は-XX:ParallelGCThreads オプションのデフォルト値が用いられます。-XX:ParallelGCThreads オプションについては，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「14.5　Application Server で指定できる Java HotSpot VM のオプション」の-XX:ParallelGCThreads を参照してください。

- 確認方法

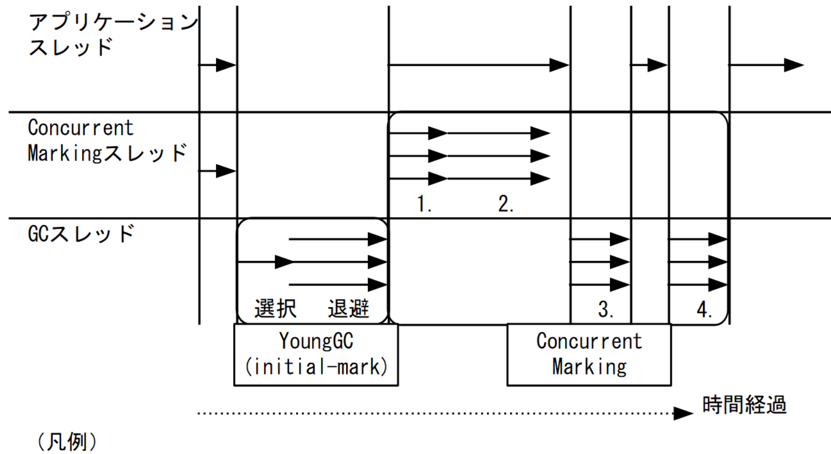
  YoungGC の確認はログ上の GC の種別が“YoungGC”または“YoungGC(initial-mark)”であることから確認できます。また，New 領域のリサイズは Eden 領域のサイズ変化と Survivor 領域のサイズ変化から確認できます。

  ```
  [VG1]<Wed Jun 12 11:21:10 2013>[Young GC 899070K/899072K(1048576K)->501755K/501760K(10485
  76K), 0.0931560 secs][Status:-][G1GC::Eden: 389120K(389120K)->0K(397312K)][G1GC::Survivor
  : 41984K->41984K][G1GC::Tenured: 459776K->459776K][G1GC::Humongous: 2048K->2048K][G1GC::F
  ree: 609536K->607232K][Metaspace: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space:
  356K(388K, 388K)->356K(388K, 388K)][cause:G1EvacuationPause][RegionSize: 1024K][Target: 0
  .2000000 secs][Predicted: 0.2495800 secs][TargetTenured: 0K][Reclaimable: 0K(0.00%)][User
  : 0.0156250 secs][Sys: 0.0312500 secs][IM: 729K, 928K, 0K][TC: 509][DOE: 16K, 171][CCI: 2
  301K, 49152K, 2304K]
  ```

上記のログの場合 GC 前の New 領域のサイズは 389120K + 41984K = 431104K であり，GC 後の New 領域のサイズは 397312K + 41984K = 439296K であるため，New 領域が拡張されたことが分かります。ログの記述内容や詳細に関しては，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「-XX:[+|-]HitachiVerboseGC（拡張 verbosegc 情報出力オプション）」を参照してください。

7.15.9　Concurrent Marking（CM）

図 7-29　Concurrent Marking の流れ



(1)　実行契機

CM は大きく次の 4 つの処理から構成されます。それぞれの実行契機について示します。なお，1.～4.の表記は上図の 1.～4.に対応しています。

1. Concurrent Root Region Scan：
   YoungGC(initial-mark)終了後，実行されます。
2. Concurrent Mark：
   1.の終了後，1.に続けて実行されます。
3. Remark：
   2.の終了後，アプリケーションを停止できるタイミングで実行されます。
4. Cleanup：
   3.の終了後，アプリケーションを停止できるタイミングで実行されます。

(2)　対象範囲

New 領域，Tenured 領域および Humongous 領域

(3)　処理内容

CM は大きく次の 4 つの処理から構成されます。それぞれの処理の詳細を示します。

1. Concurrent Root Region Scan：
   ローカル変数や使用中のオブジェクトから直接参照されている Survivor 領域内のオブジェクトにマークを付ける処理です。マルチスレッドで実行されます。
2. Concurrent Mark：

YoungGC(initial-mark)のマーキングと 1.のマーキングでマークを付けたオブジェクトが参照しているオブジェクトにマークを付ける処理です。マルチスレッドで実行されます。

3. Remark：

   2.のマーキング中に参照関係が変化したオブジェクトのマークを付け直す処理です。マルチスレッドで実行されます。

4. Cleanup：

   リージョンごとに使用中のオブジェクトの合計サイズを求める処理です。また，次の CM に備え，マークの初期化もします。マルチスレッドで実行されます。

(4)　処理結果

Eden 領域：

　使用中のオブジェクトに対応した領域にマークが付きます。

　3.の処理で Eden リージョンが回収された場合，領域サイズが減少します。

Survivor 領域：

　使用中のオブジェクトに対応した領域にマークが付きます。

　3.の処理で Survivor リージョンが回収された場合，領域サイズが減少します。

Tenured 領域：

　使用中のオブジェクトに対応した領域にマークが付きます。

　3.の処理で Tenured リージョンが回収された場合，領域サイズが減少します。

Humongous 領域：

　使用中のオブジェクトに対応した領域にマークが付きます。

　3.の処理で Humongous リージョンが回収された場合，領域サイズが減少します。

Metaspace 領域：

　変化はありません。

Free 領域：

　使用中のオブジェクトが 1 つも存在しないリージョンを回収した場合,領域サイズが増加します。

(5)　アプリケーションの停止の有無

1. Concurrent Root Region Scan：

   CM スレッドで，アプリケーションを停止しないで実行されます。

2. Concurrent Mark：

   CM スレッドで，アプリケーションを停止しないで実行されます。

3. Remark：

   VM スレッドで，アプリケーションを停止して実行されます。

4. Cleanup：

VM スレッドで，アプリケーションを停止して実行されます。

(6) ほかの GC との関係

注　CM の処理中は YoungGC(initial-mark)は発生しません。

1. Concurrent Root Region Scan：
   - YoungGC(normal)：実行されません。
   - MixedGC：実行されません。
   - FullGC：実行されません。
2. Concurrent Mark：
   - YoungGC(normal)：2.の処理中に実行された場合，2.の処理を中断します。YoungGC(normal)が終了すると，再開されます。
   - MixedGC：実行されません。
   - FullGC：2.の処理中に実行された場合，2.の処理を中止します。途中結果は破棄されます。
3. Remark：
   - YoungGC(normal)：実行されません。
   - MixedGC：実行されません。
   - FullGC：実行されません。
4. Cleanup：
   - YoungGC(normal)：実行されません。
   - MixedGC：実行されません。
   - FullGC：実行されません。

(7) 補足

- 関連オプション

  CM をするスレッド数は-XX:ConcGCThreads オプションで指定できます。CM をするスレッド数を増やすとスループットが低下します。-XX:ConcGCThreads オプションを指定しなかった場合，スレッド数は-XX:ConcGCThreads オプションのデフォルト値となります。詳細については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「14.5　Application Server で指定できる Java HotSpot VM のオプション」の-XX:ConcGCThreads を参照してください。

- 確認方法

  Concurrent Root Region Scan，Concurrent Mark の確認はログの先頭に［VCM］の識別子が付いていることから確認できます。Concurrent Mark のログのイメージを次に示します。

```
[VCM]<Wed Jul 31 11:45:23 2013>[Concurrent Mark Start][User: 0.0000000 secs][Sys: 0.00000
00 secs]
```

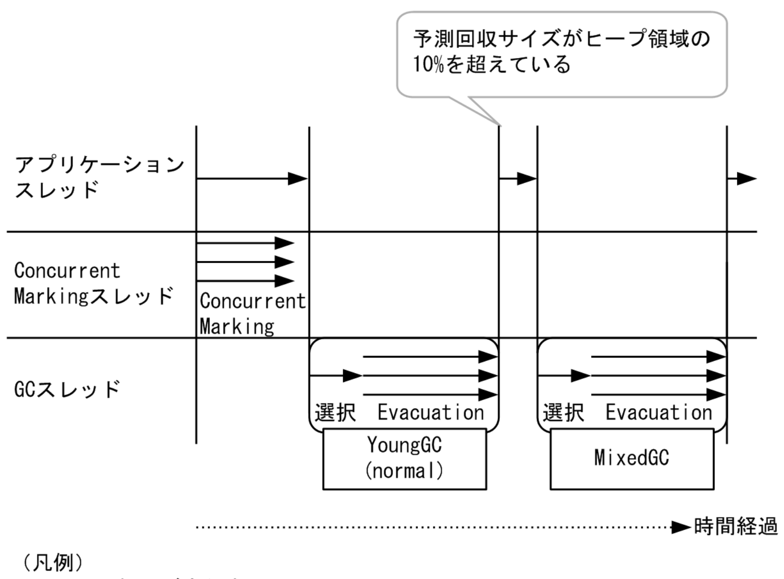
```
[VCM]<Wed Jul 31 11:45:31 2013>[Concurrent Mark End][User: 0.0321549 secs][Sys: 0.0129454
 secs]
```

VCM のログが複数行にわたるため，CM ログは開始時に Start，終了時に End が出力されます。複数行に渡る場合，CPU 使用時間は End のログに Start から End までの処理時間が出力され，Start のログは 0 が出力されます。ログの記述内容や詳細に関しては，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「-XX:[+|-]HitachiVerboseGC（拡張 verbosegc 情報出力オプション）」を参照してください。

また Remark，Cleanup の処理は処理中にアプリケーションを停止するため，VG1 ログに出力されます。Remark，Cleanup の処理は GC ログの GC 種別が“CM Remark”，“CM Cleanup”という出力から判断できます。

7.15.10 MixedGC

図 7-30 MixedGC の処理の流れ



(1) 実行契機

予測回収サイズが Java ヒープ領域の 10%を超えている場合，次の GC に MixedGC が予約されます。その判定は CM 終了直後の YoungGC(normal)終了時，または MixedGC の終了時にされます。

(2) 対象範囲

New 領域と Tenured 領域の一部

(3)　処理内容

- MixedGC が発生すると，シングルスレッドでリージョンの選択処理を，マルチスレッドで Evacuation を実行します。
- MixedGC では New 領域を GC 対象とし，予測停止時間が目標停止時間に収まる範囲で Tenured 領域を部分的に GC 対象に追加します。
- MixedGC の Evacuation では，New 領域に対しては YoungGC の Evacuation と同じ処理をします。詳細については，「7.15.8　YoungGC」を参照してください。
- GC 対象に追加した Tenured 領域に対しては，Tenured リージョン内の使用中のオブジェクトを別の Tenured リージョンに詰め直します。
- MixedGC 後も予測回収サイズが Java ヒープ領域の 10%を超えている場合，継続して MixedGC が選択され，要件を満たしていない場合，通常の YoungGC が実行されます。

(4)　処理結果

Eden 領域：

　オブジェクトが回収され，空になります。GC 後リサイズされます。

Survivor 領域：

　From 空間のオブジェクトが回収され，空になります。GC 後リサイズされます。

Tenured 領域：

　長期間必要と判断されたオブジェクトが Tenured 領域に移動します。

　GC 対象に追加された領域のオブジェクトが回収されます。

Humongous 領域：

　変化はありません。

Metaspace 領域：

　変化はありません。

Free 領域：

　GC 後のリサイズによって，増減します。

(5)　アプリケーションの停止の有無

停止します。

(6)　ほかの GC との関係

CM：MixedGC 中に実行されません。

YoungGC：MixedGC 中に実行されません。

FullGC：MixedGC 中に実行要件を満たすと，MixedGC を中止して実行されます。

(7) 補足

- 関連オプション

  Evacuation をするスレッド数は-XX:ParallelGCThreads オプションで変更することができます。スレッド数を増やすと MixedGC にかかる時間が小さくなります。また，オプションを指定しない場合，スレッド数は OS が認識している CPU 数が用いられます。-XX:ParallelGCThreads オプションについては，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「14.5 Application Server で指定できる Java HotSpot VM のオプション」の-XX:ParallelGCThreads を参照してください。

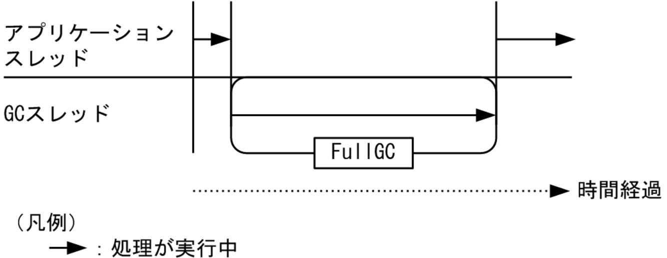
- 確認方法

  MixedGC の確認はログの GC の種別の“Mixed GC”から確認できます。また，TargetTenured の項目から MixedGC で選択された Tenured 領域のサイズを確認できます。オブジェクトの予測回収サイズに関しては Reclaimable の項目から確認できます。ログの記述内容や詳細に関しては，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「-XX:[+|-]HitachiVerboseGC (拡張 verbosegc 情報出力オプション)」を参照してください。

```
[VG1]<Wed Jun 12 11:21:10 2013>[Mixed GC 899070K/899072K(1048576K)->501742K/501760K(10485
76K), 0.0931560 secs][Status:-][G1GC::Eden: 389120K(389120K)->0K(397312K)][G1GC::Survivor
: 41984K->41984K][G1GC::Tenured: 459776K->459776K][G1GC::Humongous: 2048K->2048K][G1GC::F
ree: 609536K->607232K][Metaspace: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space:
356K(388K, 388K)->356K(388K, 388K)][cause:G1EvacuationPause][RegionSize: 1024K][Target: 0
.2000000 secs][Predicted: 0.2495800 secs][TargetTenured: 2048K][Reclaimable: 17703K(1.69%
)][User: 0.0156250 secs][Sys: 0.0312500 secs][IM: 729K, 928K, 0K][TC: 509][DOE: 16K, 171]
[CCI: 2301K, 49152K, 2304K]
```

7.15.11 FullGC

図 7-31 FullGC の流れ



(1) 実行契機

Free 領域がなくなり，Evacuation のコピー先が確保できなかった場合や Humongous 領域が確保できなかった場合に実行されます。また，Metaspace 領域の空き領域がなくなった場合や System.gc()が呼ばれた場合も実行されます。

(2)　対象範囲

New 領域，Tenured 領域，Humongous 領域および Metaspace 領域

(3)　処理内容

- FullGC はシングルスレッドでアプリケーションスレッドを停止して実行されます。
- FullGC が発生すると目標停止時間に関係なく，GC が終了するまでアプリケーションを停止します。
- FullGC の処理は SerialGC の FullGC と同じ処理です。FullGC を実行しても領域が確保できなかった場合は，OutOfMemory が発生します。詳細については，「7.2　SerialGC の仕組み」を参照してください。
- 一般的に，YoungGC や MixedGC の方が FullGC に比べて短い時間で処理できます。

(4)　処理結果

Eden 領域：オブジェクトが回収され，空になります。

Survivor 領域：使用中のオブジェクトが Tenured 領域に移動し，空になります。

Tenured 領域：使用中のオブジェクトが詰め直されます。

Humongous 領域：使用中のオブジェクトが詰め直されます。

Metaspace 領域：使用中のオブジェクトが詰め直されます。

Free 領域：回収されたリージョンによって増加します。

(5)　アプリケーションの停止の有無

停止します。

(6)　ほかの GC との関係

CM：FullGC 中は実行されません。

YoungGC：FullGC 中は実行されません。

MixedGC：FullGC 中は実行されません。

(7)　補足

- 確認方法
  FullGC の確認はログ上の GC の種別が“FullGC”であることから確認できます。また，FullGC ではこれから実行する GC に対して予測をしないため，予測停止時間や予測回収サイズは 0 で表示されます。

```
[VG1]<Wed Jan 15 12:51:32 2014>[Full GC 130443K/131072K(131072K)->55462K/56320K(131072K),
 2.3265610 secs][Status:-][G1GC::Eden: 0K(43008K)->0K(43008K)][G1GC::Survivor: 0K->0K][G1
GC::Tenured: 131072K->56320K][G1GC::Humongous: 0K->0K][G1GC::Free: 0K->74752K][Metaspace:
 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K
)][cause:ObjAllocFail][RegionSize: 1024K][Target: 0.2000000 secs][Predicted: 0.0000000 se
cs][TargetTenured: 0K][Reclaimable: 0K(0.00%)][User: 2.1700000 secs][Sys: 0.0000000 secs]
[IM: 277185K, 261856K, 44544K][TC: 1168][DOE: 0K, 0][CCI: 5808K, 49152K, 5952K]
```

7.16 G1GC のチューニング

ここでは G1GC のチューニング方法について記載します。チューニングはシステム構築時にし，システム要件を満たしていることを確認してから本番環境に適用してください。

Java ヒープ領域のサイズに関係するオプションの一覧を次の表に示します。なお，表の項番 1～8 は図の番号と対応しています。

表 7-9　Java ヒープ領域内の各領域と Metaspace 領域のサイズを指定するオプション

項番	オプション名	オプションの意味
1	-Xms<size>	Java ヒープの初期サイズを設定します。※1
2	-Xmx<size>	Java ヒープの最大サイズを設定します。※1
3	-XX:NewSize=<value>	New 領域の最小サイズを設定します。※2
4	-XX:MaxNewSize=<value>	New 領域の最大サイズを設定します。※2
5	-XX:NewRatio=<value>	New 領域に対する Tenured 領域の割合を設定します。※2 New 領域のサイズは次の式で求められます。 (New 領域のサイズ)=(Java ヒープ領域のサイズ)/(NewRatio + 1)
6	-XX:SurvivorRatio=<value>	Survivor 領域の最大で取ることができる領域サイズに対する New 領域の割合を設定します。 Survivor 領域の最大で取ることができる領域サイズは次の式で求められます。 (Survivor 領域のサイズ)※3=(New 領域のサイズ)/SurvivorRatio
7	-XX:MetaspaceSize=<size>	Metaspace 領域の初期サイズを設定します。※4
8	-XX:MaxMetaspaceSize=<size>	Metaspace 領域の最大サイズを設定します。※4

注※1

-Xms と-Xmx の値は同じ値を指定することを推奨します。

注※2

3～5 のオプションを指定した場合，New 領域のリサイズが制限され，目標停止時間内に GC 停止時間を抑えようとする G1GC のメリットを損なうため，G1GC では指定しないことを推奨します。また，3～5 のオプションを指定しなかった場合，New 領域の初期サイズは Java ヒープの初期サイズ×0.05 のサイズが確保されます。また，Survivor 領域のサイズは初期時点では 0 であるため，Eden 領域の初期サイズは New 領域の初期サイズと等しくなります。

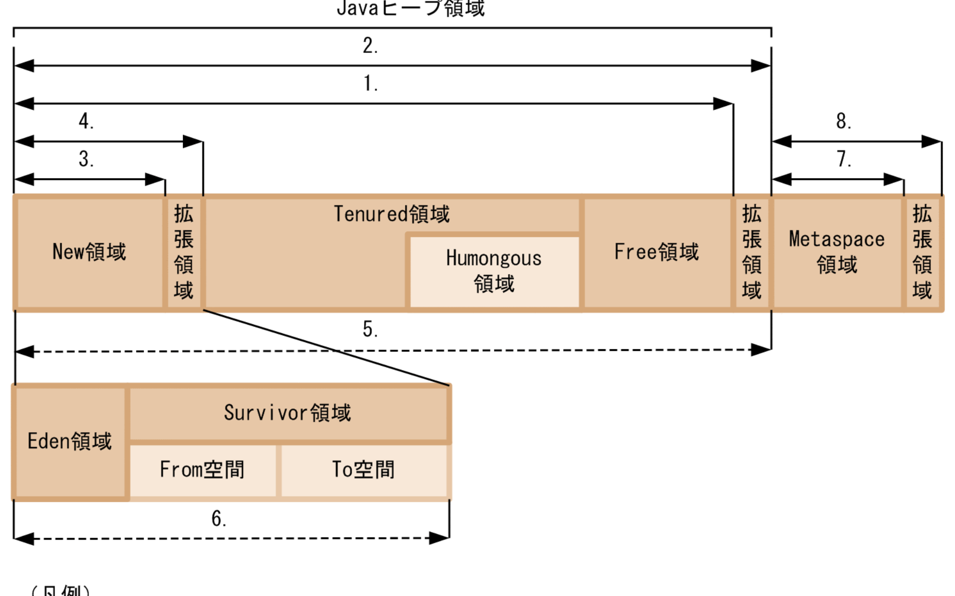
注※3

G1GC では From 空間と To 空間が必要なタイミングで動的に確保されるため，両空間が同時に存在することはありません。したがって，Survivor 領域のサイズは From 空間または To 空間と同じサイズになります。

注※4

-XX:MetaspaceSize と-XX:MaxMetaspaceSize の値は同じ値を指定することを推奨します。

図 7-32　Java ヒープ領域内の各領域と Metaspace 領域のサイズとオプションの対象範囲



次に G1GC のチューニングに利用するオプションを次の表に示します。各オプションの詳細については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「14.5　Application Server で指定できる Java HotSpot VM のオプション」の各オプションの説明を参照してください。

表 7-10　G1GC のチューニングオプション

項番	オプション名	デフォルト値	オプションの意味
1	-XX:MaxGCPauseMillis=<value>	200	目標停止時間[ms]
2	-XX:ParallelGCThreads=<value>	CPU 数※ (OS が認識している CPU 数)	Evacuation の処理をするスレッド数
3	-XX:ConcGCThreads=<value>	(ParallelGCThreads + 2) / 4	CM 処理をするスレッド数

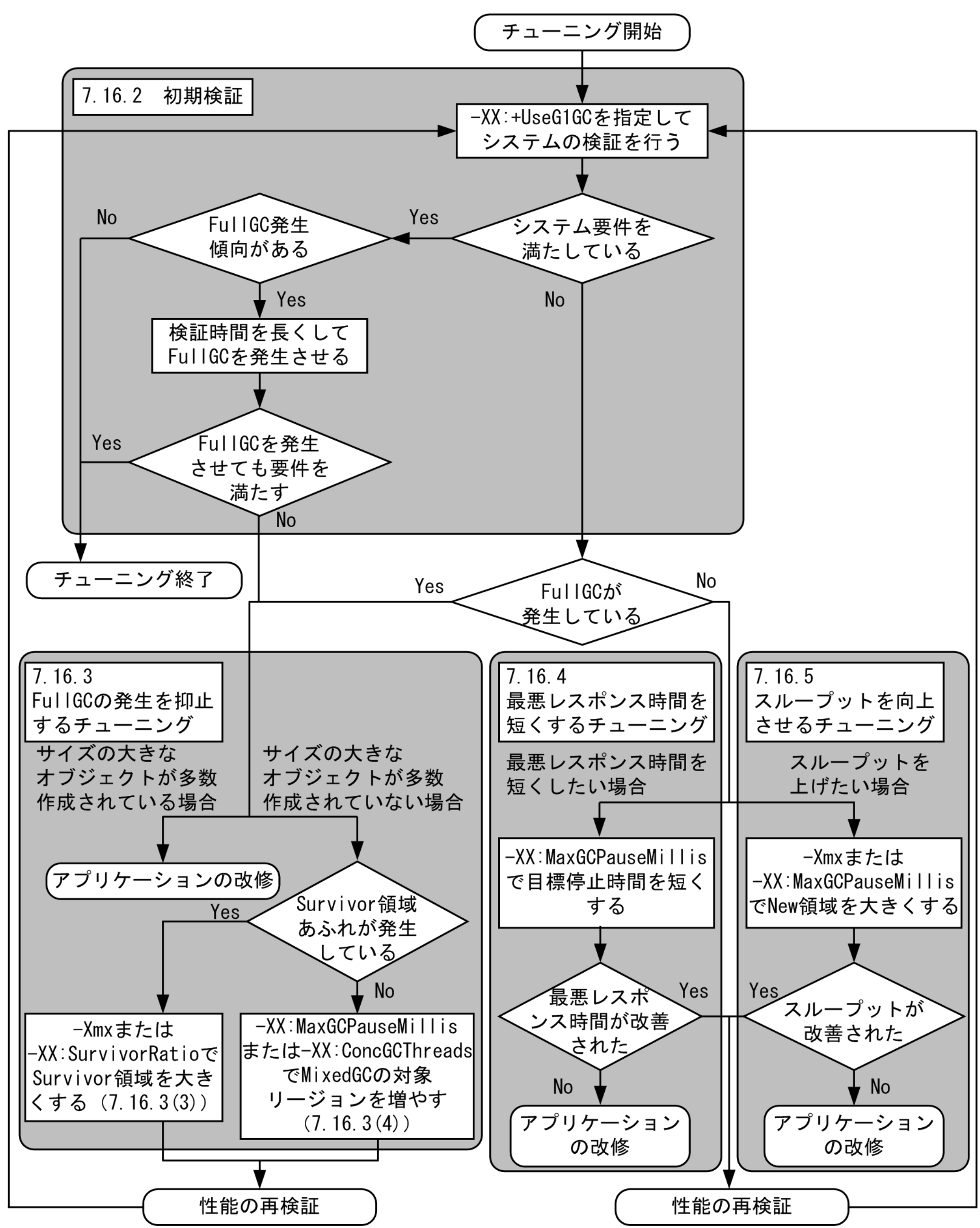
注※

実行環境の CPU 数によって計算式が異なります。

7.16.1　チューニングの流れ

次にチューニングの全体の流れを示します。

図 7-33　チューニングの全体の流れ



次から，それぞれのチューニングの詳細を記載します。

7.16.2　初期検証

図 7-34　初期検証の流れ

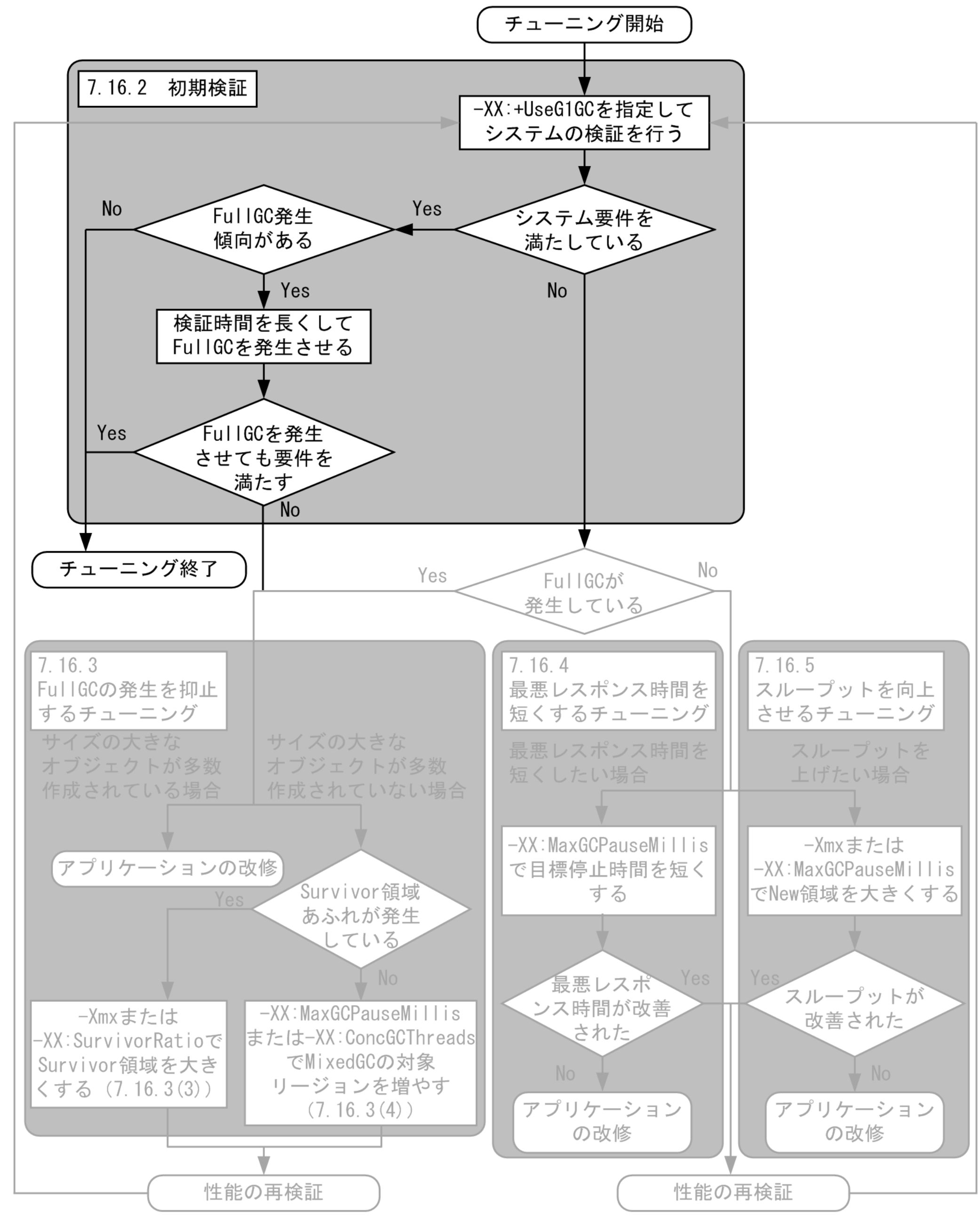


図 7-34 は初期検証の流れを表した図です。

1. G1GC を有効にしてシステムの検証をします。検証の結果，システム要件を満たしている場合でも，FullGC の発生傾向がないか確認をします。システム要件を満たしていない場合は，「7.16.3　FullGC の発生を抑止するチューニング」へ進んでください。
2. 検証中に FullGC が発生していない場合でも，システムを長期間動かすことで，将来 FullGC が発生する場合があります。そのため，FullGC の発生傾向がないか確認をします。

図 7-35　FullGC の発生傾向のイメージ

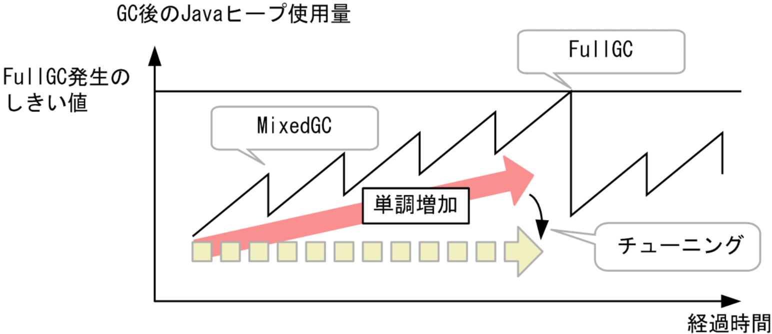


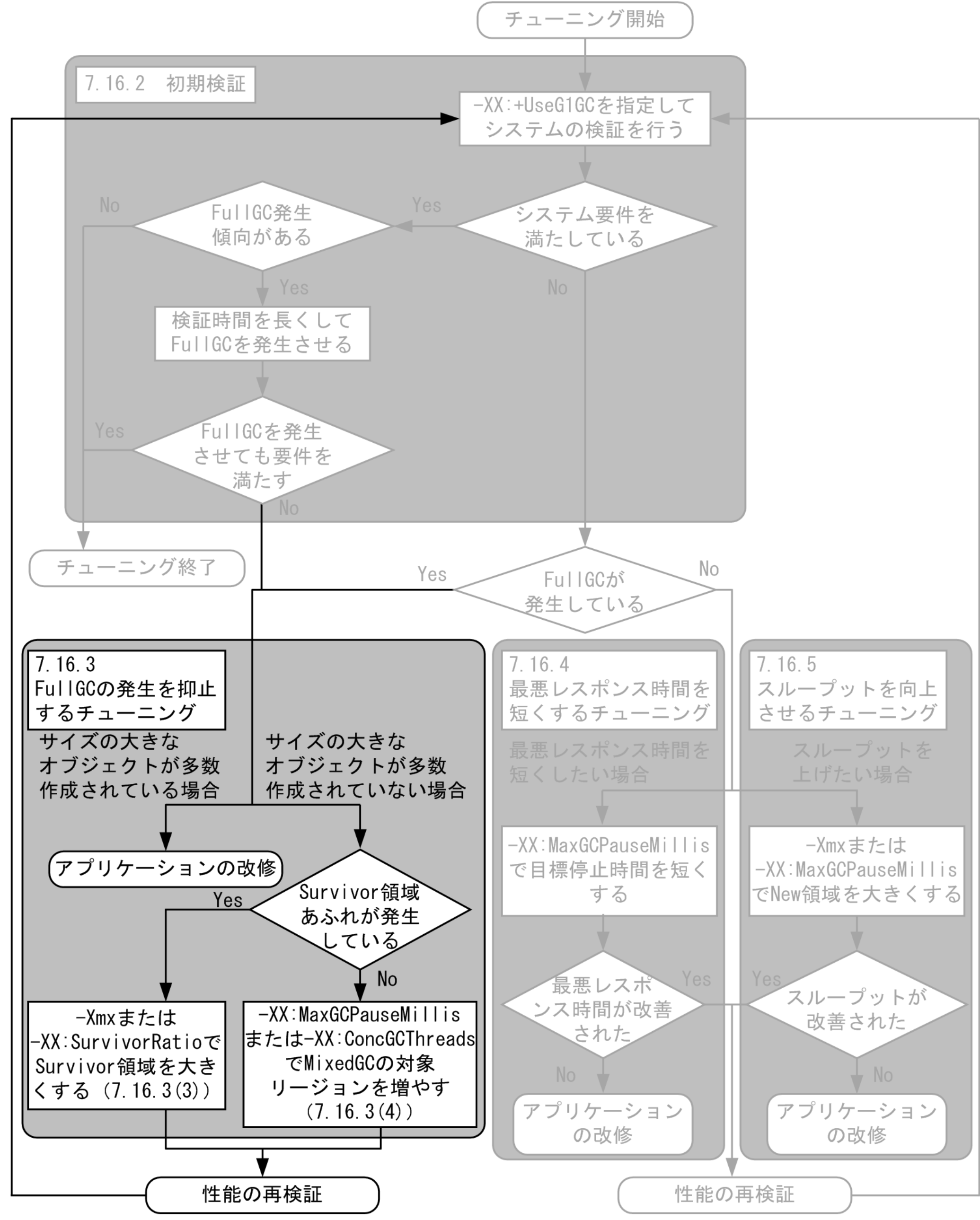
図 7-35 は検証中の GC 後の使用量をグラフにしたイメージです。図 7-35 のように MixedGC が発生していても，Java ヒープ使用量が長期的には単調増加している場合，FullGC の発生傾向があります。

3. システム要件を満たしている場合でも，FullGC の発生傾向があった場合，検証時間を延ばし FullGC を発生させ，FullGC が発生してもシステム要件が満たしているか確認をします。FullGC を発生させた結果，システム要件を満たさなくなった場合は，FullGC の発生を抑止するチューニングをします。また，検証時間を延ばすことができない場合は，FullGC が発生していなくても，FullGC の発生を抑止するチューニングをし，FullGC の発生傾向をなくしてください。FullGC の発生を抑止するチューニングについては，「7.16.3　FullGC の発生を抑止するチューニング」を参照してください。

7.16.3 FullGC の発生を抑止するチューニング

(1) FullGC の発生を抑止する考え方

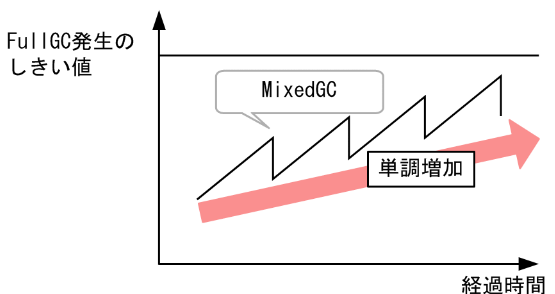
図 7–36 FullGC の発生を抑止するチューニングの流れ



FullGC は Tenured 領域の使用サイズが増加し続け，Free リージョンが確保できなくなった場合実行されます。

次に FullGC の発生を抑止するチューニングのイメージを示します。

図 7-37　FullGC の発生を抑止するチューニングのイメージ



FullGC の発生を抑止するには，Tenured 領域の使用サイズの増加を防ぐ必要があります。そのため，FullGC の発生を抑止するには次の 2 つの方法が考えられます。

1. Tenured 領域へ移動するオブジェクト数を減らす
2. Tenured 領域内で不要なオブジェクトを回収する

Survivor 領域あふれが発生している場合は，1 の方法で，発生していない場合は 2 の方法でチューニングしてください。1 の方法の詳細については，「7.16.3(3)　Survivor 領域のサイズチューニング」を，2 の方法の詳細については，「7.16.3(4)　MixedGC で選択されるリージョンを増やすチューニング」を参照してください。チューニングをした場合，再度システムの検証をし，要件を満たしているか確認してください。

(2)　サイズの大きなオブジェクトが多数作成されている場合

1 つのオブジェクトのサイズが大きいオブジェクトが多数作成される場合，連続した領域が確保できないで，FullGC が実行される場合があります。G1GC ではメモリをリージョンで管理しているため，オブジェクトのサイズが大きいオブジェクトが多数作成されるシステムには不向きです。オブジェクトのサイズが大きいオブジェクトが多数作成されることによって，FullGC が発生している場合は，アプリケーションの改修をしてください。

サイズの大きなオブジェクトの確認はログの Humongous の項目から確認できます。

```
[VG1]<Wed Jan 15 12:51:32 2014>[Full GC 130443K/131072K(131072K)->55462K/56320K(131072K), 2.
3265610 secs][Status:-][G1GC::Eden: 0K(43008K)->0K(43008K)][G1GC::Survivor: 0K->0K][G1GC::Te
nured: 131072K->56320K][G1GC::Humongous: 1024K->0K][G1GC::Free: 0K->74752K][Metaspace: 3634
K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 356K(388K, 388K)->356K(388K, 388K)][cause
:ObjAllocFail][RegionSize: 1024K][Target: 0.2000000 secs][Predicted: 0.0000000 secs][TargetT
enured: 0K][Reclaimable: 0K(0.00%)][User: 2.1700000 secs][Sys: 0.0000000 secs][IM: 277185K,
261856K, 44544K][TC: 1168][DOE: 0K, 0][CCI: 5808K, 49152K, 5952K]
```

(3)　Survivor 領域のサイズチューニング

Tenured 領域へ移動するオブジェクト数を減らす方法は，Survivor 領域にオブジェクトが存在する間に，オブジェクトを回収する方法です。Survivor 領域の空き領域がなくなると，本来寿命の短いオブジェクト

が Survivor 領域あふれによる退避で Tenured 領域に昇格します。そのため，Survivor 領域を大きくすることで，寿命の短いオブジェクトを Survivor 領域で回収し，Tenured 領域への移動量を減らします。

ログが次のように(to exhausted)が出力されている場合，Survivor 領域あふれが発生しています。この場合 Survivor 領域を大きくするために，-Xmx オプションまたは-XX:SurvivorRatio オプションでチューニングをしてください。

```
[VG1]<Wed Jun 12 11:21:10 2013>[Young GC 899070K/899072K(1048576K)->501755K/501760K(1048576K
), 0.0931560 secs][Status:to exhausted][G1GC::Eden: 389120K(389120K)->0K(397312K)][G1GC::Sur
vivor: 41984K->41984K][G1GC::Tenured: 459776K->459776K][G1GC::Humongous: 2048K->2048K][G1GC:
:Free: 609536K->607232K][Metaspace: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space: 3
56K(388K, 388K)->356K(388K, 388K)][cause:G1EvacuationPause][RegionSize: 1024K][Target: 0.200
0000 secs][Predicted: 0.2495800 secs][TargetTenured: 0K][Reclaimable: 0K(0.00%)][User: 0.015
6250 secs][Sys: 0.0312500 secs][IM: 729K, 928K, 0K][TC: 509][DOE: 16K, 171][CCI: 2301K, 4915
2K, 2304K]
```

チューニングに用いるオプション名とチューニング方法を次に示します。

- -Xmx オプション

  指定する値を大きくすることで，Java ヒープ領域のサイズが大きくなります。Java ヒープ領域全体のサイズが大きくなることで，Survivor 領域のサイズも大きくなります。ただし，Java ヒープ領域の最大サイズを大きくすると，New 領域がリサイズされる範囲の最小値と最大値が大きくなります。これによって最小停止時間が大きくなるため，チューニングをした後に再度システム要件を満たしているか検証してください。

- -XX:SurvivorRatio オプション

  指定する値を小さくすることで，New 領域に対する Survivor 領域の最大で取ることができる領域サイズの割合を大きくします。ただし，この方法は Eden 領域がチューニング前に比べて小さくなるおそれがあるため，YoungGC や MixedGC が発生しやすくなる副作用があります。そのため，チューニングをした後に再度システム要件を満たしているか検証してください。

注意

Survivor 領域を大きくする方法として-XX:NewRatio や-XX:NewSize，-XX:MaxNewSize オプションを使って New 領域のサイズを大きくする方法もあります。しかし，この場合 GC 後の New 領域のリサイズが制限されるため，New 領域のサイズを変更するオプションの指定は推奨しません。

(4) MixedGC で選択されるリージョンを増やすチューニング

MixedGC で対象とするリージョンを増やすには，1 度の MixedGC の対象リージョンを増やす方法と，MixedGC の発生頻度を上げて対象リージョン延べ数を増やす方法があります。1 度の MixedGC の対象リージョン数を増やす方法はレスポンスが低下し，対象リージョンの延べ数を増やす方法はスループットが低下します。システム要件にあわせてチューニング方法を選択してください。

- -XX:MaxGCPauseMillis オプション

  指定する値を大きくすることで，目標停止時間が大きくなります。MixedGC では New 領域と Tenured 領域の一部を対象として GC をします。Tenured 領域は New 領域を選択しても予測停止時間に対し

て目標停止時間に余裕がある場合に選択されます。そこで，目標停止時間を大きくすることで，1 度のMixedGC で選択される Tenured リージョンの数が増えます。ただし，この方法は GC 停止時間が長くなるため，レスポンスが低下します。

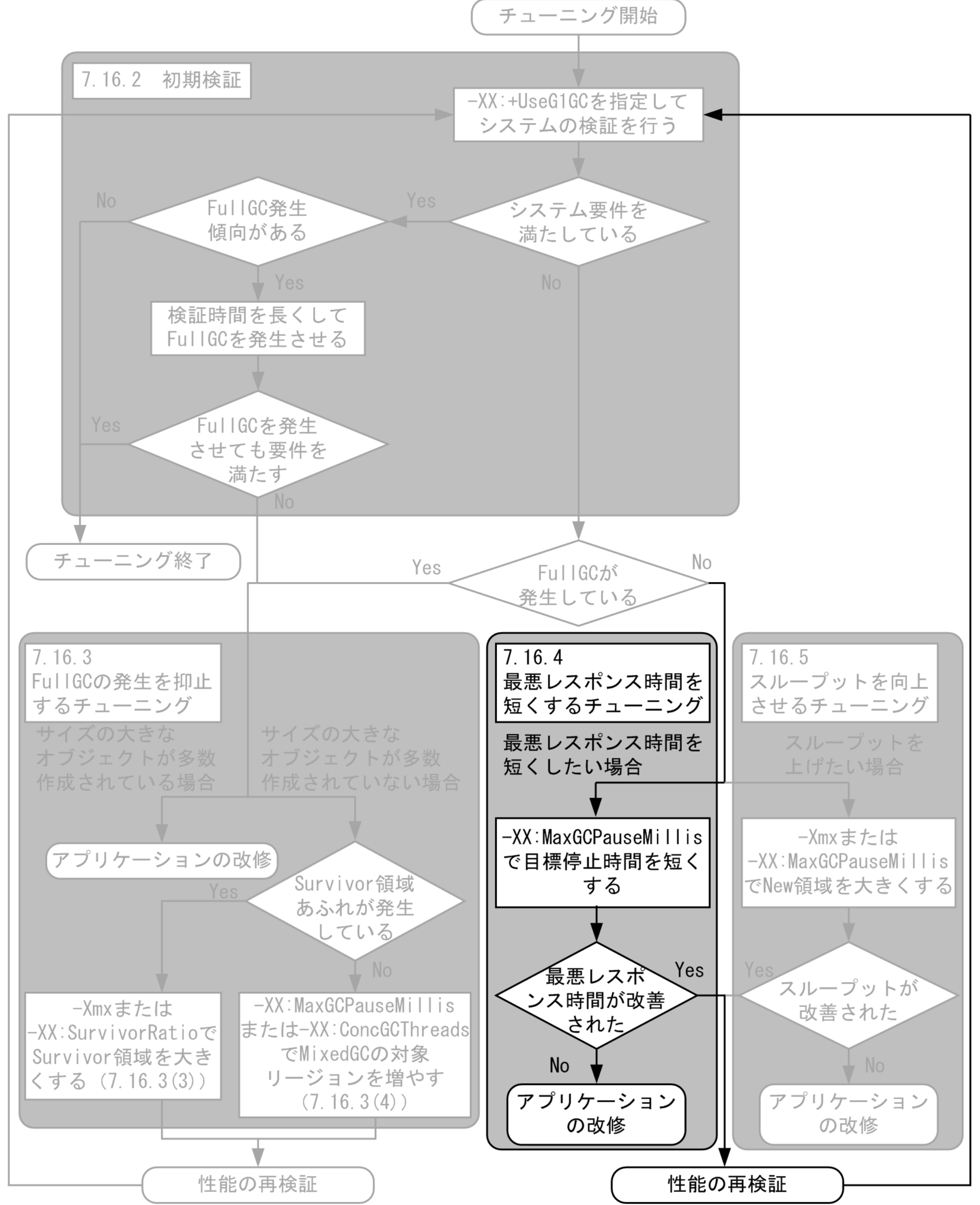
- -XX:ConcGCThreads オプション

  指定する値を大きくすることで，CM をするスレッド数が増えます。MixedGC が実行されるためには，CM が終了している必要があります。そのため，CM をするスレッド数を増やし，CM の終了間隔を短くします。これによって MixedGC の発生回数が増え，MixedGC で選択されるリージョンの延べ数が増えます。ただし，この方法では，アプリケーションスレッドと並行して処理をするスレッド数が増えるため，スループットが低下します。

  また，CM スレッド数は Evacuation をするスレッド数より多い値を指定できません。そのため，-XX:ConcGCThreads オプションと併せて，-XX:ParallelGCThreads オプションの値も大きくする必要があります。

7.16.4 最悪レスポンス時間を短くするチューニング

図 7-38 最悪レスポンス時間を短くするチューニングの流れ



このチューニングは，GC によるアプリケーションの停止時間を短くすることで，最悪レスポンス時間を短くする方法です。そのため，レスポンス時間のうち，GC によるアプリケーションの停止時間が占める割合が大きい場合に有用です。GC によるアプリケーションの停止時間は，VG1 ログの gc_time の項目から取得できます。VG1 ログの詳細については，マニュアル「アプリケーションサーバ アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「-XX:[+|-]HitachiVerboseGC（拡張 verbosegc 情報出力オプション)」を参照してください。

最悪レスポンス時間を短くするには，GC によるアプリケーションの停止時間を短くする必要があります。G1GC では停止時間を指定することができるため，指定する値を小さくすることで最悪レスポンス時間を短くします。

- -XX:MaxGCPauseMillis オプション

  指定する値を小さくすることで，目標停止時間が短くなり，最悪レスポンス時間が短くなります。ただし，この方法では GC 回数が多くなりスループットが低下します。そのため，最悪レスポンス時間を短くするチューニングをした場合，再度システム要件を満たしているか検証してください。また，G1GC ではどの GC でも New 領域を GC の対象とするため，New 領域の回収にかかる時間以下には GC 時間を抑えられません。それ以上 GC 停止時間を短くできない GC 停止時間を最小停止時間と呼びます。目標停止時間を短くしても最悪レスポンス時間が短くならない場合は，ログの GC 停止時間を確認してください。

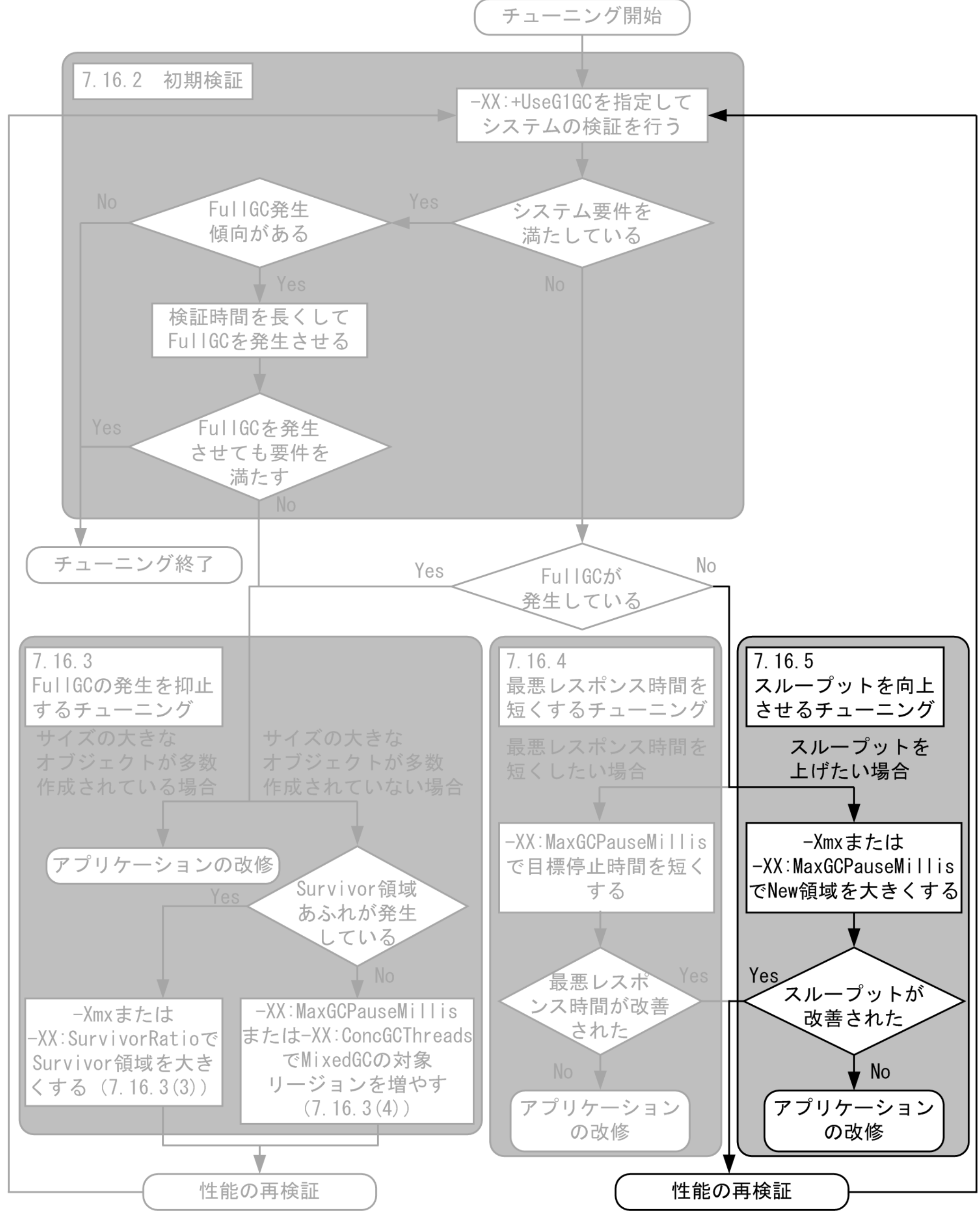
  GC 停止時間は次のログの背景色付き**太字**の部分から確認できます。

```
[VG1]<Wed Jun 12 11:21:10 2013>[Young GC 899070K/899072K(1048576K)->501755K/501760K(10485
76K), 0.0931560 secs][Status:-][G1GC::Eden: 389120K(389120K)->0K(397312K)][G1GC::Survivor
: 41984K->41984K][G1GC::Tenured: 459776K->459776K][G1GC::Humongous: 2048K->2048K][G1GC::F
ree: 609536K->607232K][Metaspace: 3634K(4492K, 4492K)->3634K(4492K, 4492K)][class space:
356K(388K, 388K)->356K(388K, 388K)][cause:G1EvacuationPause][RegionSize: 1024K][Target: 0
.2000000 secs][Predicted: 0.2495800 secs][TargetTenured: 0K][Reclaimable: 0K(0.00%)][User
: 0.0156250 secs][Sys: 0.0312500 secs][IM: 729K, 928K, 0K][TC: 509][DOE: 16K, 171][CCI: 2
301K, 49152K, 2304K]
```

目標時間を短くしても GC の停止時間が短くなっていない場合は，最小停止時間に達しています。また，目標時間を最小停止時間に近づけすぎると，目標停止時間を超えたときの超え幅が大きくなり，最悪レスポンス時間が長くなる場合があります。目標停止時間を短くしてもシステム要件を満たせない場合は，チューニングではシステム要件を満たせません。そのため，アプリケーションの改修を行ってください。

7.16.5　スループットを向上させるチューニング

図 7–39　スループットを向上させるチューニングの流れ



スループットを向上させるためには，GC の発生回数を減らす必要があります。YoungGC や MixedGC は割り当てられた Eden 領域を使い切ると発生します。そこで，New 領域を大きくすることで Eden 領域を大きくし，GC の発生回数を減らします。New 領域を大きくするためには，-Xmx オプションまたは-XX:GCMaxPauseMillis オプションを用いてチューニングをします。

- -Xmx オプション

  指定する値を大きくすることで，Java ヒープ領域の最大サイズを大きくします。Java ヒープ領域の最大サイズが大きくなることで，New 領域も大きくなります。

- -XX:MaxGCPauseMillis オプション

指定する値を大きくすることで，目標停止時間が長くなります。YoungGC では目標停止時間内に収まる範囲で，できるだけ大きく New 領域を取るため，目標停止時間を長くすることで，New 領域も大きくなります。

スループットを向上させるチューニングをしてもスループットが向上しない場合は，New 領域のサイズがリサイズされる範囲の最大サイズに達していることが考えられます。ログの Eden 領域＋ Survivor 領域の値の変化からリサイズがされているか確認してください。ログの確認方法の詳細については，「7.15.8 YoungGC」を参照してください。なお，スループットを向上させるチューニングをするとレスポンスが低下する場合があるため，チューニング後は性能の再検証をする必要があります。

7.17　ZGC の仕組み（JDK17 以降の場合）

ここでは，ZGC の仕組みについて説明します。

7.17.1　ZGC の概要

ZGC は，チューニングが簡単で，かつ停止時間が短いスケーラブルな GC です。そのため，低レイテンシが要求されるシステム，および大規模なメモリ環境のシステムに適しています。

7.17.2　ZGC の用語

「7.17　ZGC の仕組み（JDK17 以降の場合）」～「7.18　ZGC のチューニング（JDK17 以降の場合）」で使用する用語を次の表に示します。

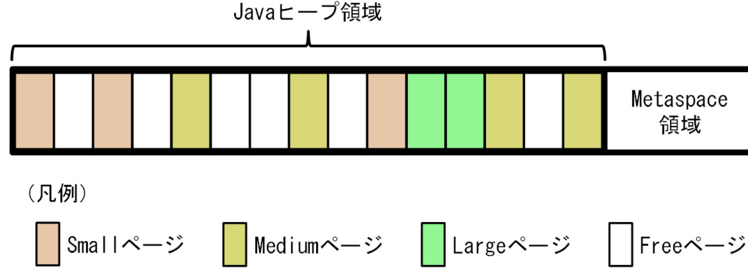
表 7-11　ZGC の用語

用語	説明
ページ	Java ヒープ領域のメモリ空間を分割するブロックです。ZGC では，Java ヒープ領域のメモリ空間を 3 種類のページに分割して管理しています。
ページの回収	オブジェクトを含んだページをページごと削除し，メモリ空間を再利用できるようにすることです。
フォワーディングテーブル	使用中のオブジェクトを再配置するときに，使用中のオブジェクトの古いアドレスから新しいアドレスへのマッピング情報を書き込むテーブルです。

7.17.3　JavaVM で使用するメモリ空間の構成

アプリケーション実行中の ZGC で管理するメモリ空間の構成を次の図に示します。

図 7-40　アプリケーション実行中の ZGC で管理するメモリ空間の構成



各ページのサイズについて説明します。

- Small ページ

サイズは 2 メガバイトです。

- Medium ページ
  サイズは，Java ヒープの最大サイズの 3.125%です。ただし，値は 2 の累乗の数値になるように切り捨てられます。また，値の範囲は 2 メガバイト～32 メガバイトです。なお，Java ヒープの最大サイズは-Xmx オプションで設定できます。-Xmx オプションの詳細は，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「14.5　Application Server で指定できる Java HotSpot VM のオプション」を参照してください。
- Large ページ
  サイズは，Medium ページのサイズの 12.5%より大きな値です。なお，サイズはページごとに異なります。

次に，各領域について説明します。

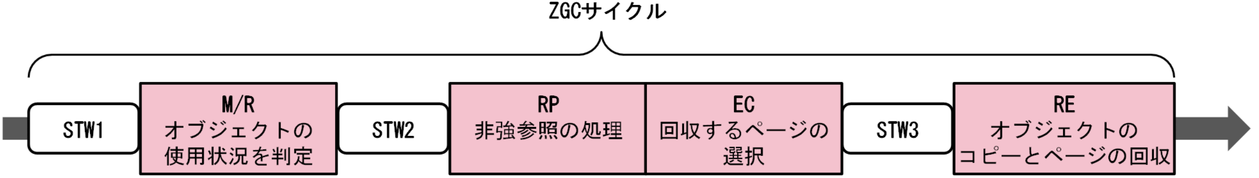
- Free 領域
  ページが割り当てられていない領域を指します。この領域から，ページの新規作成に必要な領域を割り当てます。また，回収されたページの領域は Free 領域に追加されます。
- Metaspace 領域
  ロードされたクラスなどの情報が格納される領域を指します。

7.17.4　ZGC サイクル

ZGC では，ZGC サイクルと呼ばれる一連の処理を実行することで GC 処理をします。

ZGC サイクルの概要を次の図に示します。

図 7-41　ZGC サイクルの概要



ZGC サイクルは，アプリケーションと GC 処理を並行して実行するフェーズ（M/R，RP，EC，RE）と，アプリケーションを一時停止して GC 処理を実行するフェーズ（STW1，STW2，STW3）で構成されます。

各フェーズで実行する処理の内容は，次のとおりです。

(1)　STW1

M/R フェーズでオブジェクトの参照をたどるときの起点となるオブジェクトをマークします。

(2)　M/R（mark/remap）

通常の参照を処理します。Java ヒープ領域内のすべてのオブジェクトに対して，使用中か使用済みかを判別し，マーキングします。マーキングの結果を基に，EC フェーズで回収するページを選択します。また，前回の ZGC サイクルで回収したページに含まれていた使用中のオブジェクトを，フォワーディングテーブルを経由せずに直接参照できるようマッピング処理をします。

(3)　STW2

アプリケーションの実行を一時停止し，M/R フェーズですべてのオブジェクトがマーキングされたかどうかを確認します。マーキングされていないオブジェクトがある場合は，再び M/R フェーズを開始します。

(4)　RP（reference processing）

M/R フェーズでのマーキング結果を基に，弱参照，ソフト参照，およびファントム参照をたどって，オブジェクトが使用済みかどうかを判別します。

(5)　EC（evacuation candidates）

M/R および RP フェーズでマーキングしたオブジェクトの情報を基に，回収するページを選択します。また，回収するページにある使用中のオブジェクトについて，現在のアドレスから新しいアドレスへのマッピング情報をフォワーディングテーブルに書き込みます。

(6)　STW3

現在のサイクルが終了してから次のサイクルが開始するまでの間で，RE フェーズで移動する使用中のオブジェクトを参照する場合に，フォワーディングテーブルを介して参照できるようマッピング処理をします。

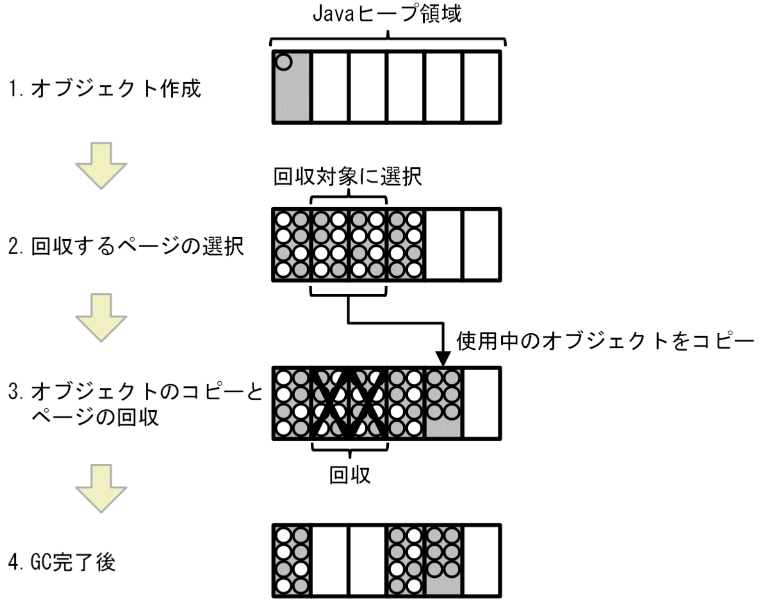
(7)　RE（relocation）

使用中のオブジェクトを，Free 領域から割り当てたページにコピーします。そのあと，使用済みのオブジェクトをページごと回収します。これによって，使用済みのオブジェクトの削除が完了します。

7.17.5　Java ヒープ領域のメモリ管理方法

オブジェクト作成から GC 完了までの流れを次の図に示します。

図 7-42　オブジェクト作成から GC 完了までの流れ



(1)　オブジェクト作成時

オブジェクトのサイズに応じて，Small ページ，Medium ページまたは Large ページにオブジェクトが配置されます。

各ページに配置されるオブジェクトは次のとおりです。

- Small ページおよび Medium ページ
  各ページのサイズの 12.5%以下のオブジェクトが配置されます。
- Large ページ
  Medium ページのサイズの 12.5%より大きなオブジェクトが配置されます。ただし，Large ページに配置できるオブジェクトは，1 ページごとに 1 つだけです。

なお，すでにオブジェクトがあるページに新しいオブジェクトを配置できる場合は，そのページを優先してオブジェクトを配置します。

(2)　回収するページの選択

使用済みのオブジェクトの割合に応じて，回収するページを選択します。この割合は，-XX:ZFragmentationLimit オプションで設定できます。詳細は，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「14.5　Application Server で指定できる Java HotSpot VM のオプション」を参照してください。

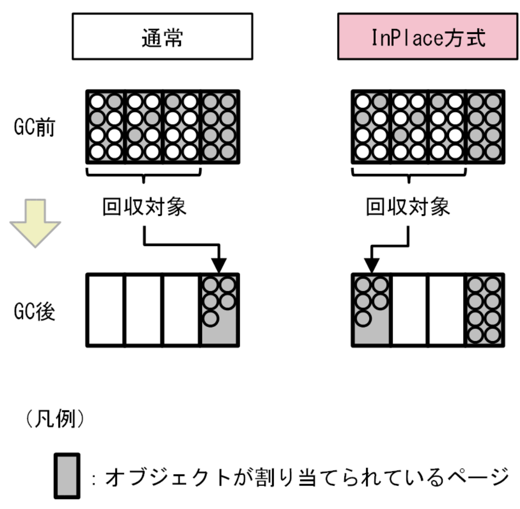
(3)　使用中のオブジェクトのコピーとページの回収

選択したページに含まれる使用中のオブジェクトのサイズに応じて，Free 領域からページを割り振り，使用中のオブジェクトをコピーします。そのあと，ページを回収して，削除します。Free 領域が不足してコピーできない場合は，InPlace を使用します。

InPlace とは，回収するページ内に使用中のオブジェクトをコピーする方法です。これによって，Free 領域が不足している場合でも，OutOfMemory にならないで GC 処理を実行できます。

InPlace の概要を次の図に示します。

図 7-43　InPlace の概要



(4)　GC 完了

回収されたページの領域は，Free 領域に追加されて再利用されます。

7.17.6　ZGC 独自の GC 発生要因

GC 発生要因のうち，ZGC を使用している場合にだけ適用される要因について説明します。

表 7-12　ZGC 独自の GC 発生要因

GC 発生要因	説明
ZTimer	前回の GC から，-XX:ZCollectionInterval=<seconds>オプションで指定した時間が経過したため，GC が発生しました。
ZWarmup	ZWarmup 以外の要因の GC が発生していないときに，ヒープ領域の使用率が 10%，20%または 30%を超えたため，GC が発生しました。
ZAllocationRate	最大割り当て率と空きメモリ量から，OutOfMemory が発生するおそれがあると判断されたため，GC が発生しました。
ZAllocationStall	アプリケーションの実行に必要な Java ヒープ領域の空き容量が不足しているため，GC が発生しました。
ZProactive	GC でスループットが低下してもシステムの動作に大きな影響がない場合に，前もって GC が発生しました。
ZHighUsage	Java ヒープ領域の 95%以上が使用されたため，GC が発生しました。

7.18　ZGC のチューニング（JDK17 以降の場合）

ここでは，ZGC を使用するためのチューニング方法について説明します。

7.18.1　ZGC での GC の考え方

ZGC を使用している場合，GC は「先行して，前もって，予防的に」実行する必要があります。

ZGC では，多くの GC 処理をアプリケーションと同時に実行しています。そのため，アプリケーションがオブジェクトを作成する速度よりも，GC 処理でオブジェクトを削除する速度の方が遅い場合，Java ヒープを有効に使用できません。

7.18.2　チューニングの流れ

ZGC を使用している場合のチューニングでは，OutOfMemory を抑制するために Java ヒープ領域の最大サイズを設定します。Java ヒープ領域の最大サイズの設定以外の方法で OutOfMemory を抑制する場合は，「7.18.3　GC 停止時間およびスループットの改善方法」を参照してください。

(1)　Java ヒープ領域の最大サイズの設定

Java ヒープ領域の最大サイズを設定する手順は次のとおりです。

1. SerialGC を有効にして，最大の負荷を掛けた状態でアプリケーションを実行します。
2. 拡張 verbosegc 情報から，FullGC 発生後の Java ヒープ領域のサイズを確認します。
   FullGC 発生後の拡張 verbosegc 情報の出力例を次に示します。下線部分が FullGC 発生後の Java ヒープ領域のサイズです。
   ```
   …
   [VGC]<Wed Dec 28 14:12:05 2022>[Full GC 31780K->30780K(32704K), 0.2070500secs][DefNew::Ed
   en: 3440K->1602K(3456K)][DefNew::Survivor:58K->0K(64K)][Tenured: 28282K->29178K(29184K)][
   Metaspace:1269K->1269K(4096K)][cause:ObjAllocFail][User: 0.0156250 secs][Sys: 0.0312500 s
   ecs]
   …
   ```
3. FullGC 発生後の拡張 verbosegc 情報を何度か収集します。
4. 手順 3.で収集した FullGC 発生後の Java ヒープ領域のサイズのうち，最も大きな値を，アプリケーションの動作に必要な Java ヒープ領域の最小サイズとします。
5. Java ヒープ領域の最大サイズを設定します。
   -Xmx オプションを設定します。GC 実行中のオブジェクトを余裕を持って作成できるようにするため，Java ヒープ領域の最大サイズは，手順 4.で確認した値の 1.5 倍～2 倍に設定することを推奨します。
   なお，Java ヒープ領域の初期サイズ（-Xms オプション）は，最大ヒープサイズと同じ値に設定することを推奨します。

(2)　OutOfMemory の発生調査

ZGC を有効にして，システムの検証を行います。OutOfMemory が発生しているかどうかを確認し，結果に応じて次の作業を実施してください。

- OutOfMemory が発生している場合
  -Xms オプションの値を大きくしたあと，再度「(1)　Java ヒープ領域の最大サイズの設定」の手順を実施します。
- OutOfMemory が発生していない場合
  チューニングは完了です。停止時間をさらに短くしたい場合，またはスループットを改善したい場合は，「7.18.3　GC 停止時間およびスループットの改善方法」を参照してください。

7.18.3　GC 停止時間およびスループットの改善方法

Java ヒープ領域の最大サイズの調整以外に GC 停止時間およびスループットを改善する方法について説明します。ただし，この項で説明する方法は，GC 停止時間が短くなる代わりにメモリの消費量が増えるなどの悪影響があるため，お使いの環境に合わせて実施してください。

- GC 停止時間をより短くしたい場合
  - -XX:ParallelGCThreads オプションの値を大きくします
    GC 停止時間が短くなる代わりに，メモリの消費量が増えます。
- スループットをより向上させたい場合
  - -XX:ConcGCThreads オプションの値を小さくします
    スループットが向上する代わりに，OutOfMemory が発生しやすくなります。
  - -XX:ParallelGCThreads を大きくします
    スループットが向上する代わりに，メモリの消費量が増えます。

7.18.4　OS のパラメタ設定（Linux の場合）

ZGC では多くのメモリマップ領域を使用するため，OS で定義されているプロセスごとのメモリマップ領域の上限を増やしてください。

手順は次のとおりです。

1. 次の計算式に従って，ZGC 使用時の Java プロセスのメモリマップ領域の数を算出してください。

```
ZGC使用時のJavaプロセスのメモリマップ領域の数 = Javaヒープ領域の最大サイズ / 2メガバイト
× 3 × 1.2
```

2. 算出した値を，/proc/sys/vm/max_map_count に設定してください。

7.19　ZGC 使用時の他機能への影響（JDK17 以降の場合）

ここでは，ZGC 使用時の他機能への影響について説明します。

7.19.1　使用が制限される機能

ZGC を使用している場合，次の機能を使用できません。

(1)　クラス別統計機能

この機能の詳細は，マニュアル「アプリケーションサーバ 機能解説 保守／移行編」の「9.3　クラス別統計機能」を参照してください。

また，次に示すコマンドおよびオプションが無効になります。

- jheapprof コマンド
  ZGC を使用している場合にこのコマンドを指定すると，次のエラーメッセージを出力して，スレッドダンプを出力しません。
  ```
  jheapprof: can't use jheapprof and zgc at the same time.
  ```
  このコマンドの詳細は，マニュアル「アプリケーションサーバ リファレンス コマンド編」の「jheapprof（クラス別統計情報付き拡張スレッドダンプの出力）」を参照してください。
- -XX:+HitachiOutOfMemoryAbortThreadDumpWithJHeapProf オプション
  このオプションの詳細は，マニュアル「アプリケーションサーバ リファレンス 定義編（サーバ定義）」の「-XX:[+|-]HitachiOutOfMemoryAbortThreadDumpWithJHeapProf（クラス別統計情報出力オプション）」を参照してください。

(2)　クラス別統計情報解析機能

この機能の詳細は，マニュアル「アプリケーションサーバ 機能解説 保守／移行編」の「9.10　クラス別統計情報解析機能」を参照してください。

(3)　Survivor 領域の年齢分布情報出力機能

この機能の詳細は，マニュアル「アプリケーションサーバ 機能解説 保守／移行編」の「9.11　Survivor 領域の年齢分布情報出力機能」を参照してください。

(4)　ローカル変数情報出力機能

この機能の詳細は，マニュアル「アプリケーションサーバ リファレンス 定義編（サーバ定義）」の「-XX:[+|-]HitachiLocalsInThrowable（例外発生時のローカル変数情報収集オプション）」を参照してください。

また，次のオプションが無効になります。

- -XX:+HitachiLocalsInStackTrace オプション
- -XX:+HitachiLocalsSimpleFormat オプション

これらのオプションの詳細は，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「14.2　JavaVM 拡張オプションの詳細」を参照してください。

(5)　メモリ情報取得機能

この機能の詳細は，マニュアル「アプリケーションサーバ リファレンス API 編」の「10.5　MemoryInfo クラス」を参照してください。

7.19.2　出力内容が変更される機能

ZGC を使用している場合，次の機能で出力される内容が変更されます。

(1)　日立拡張 VerboseGC 機能

ZGC を使用している場合，ZGC 向けの日立拡張 Java ログが出力されます。

ZGC 使用している場合の日立拡張 Java ログの詳細は，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「-XX:[+|-]HitachiVerboseGC（拡張 verbosegc 情報出力オプション）」を参照してください。

(2)　日立拡張スレッドダンプ機能

ZGC を使用している場合，Java ヒープ領域の構成が SerialGC および G1GC の場合と異なります。そのため，日立拡張スレッドダンプに出力される Java ヒープ情報の出力フォーマットが異なります。日立拡張スレッドダンプの出力フォーマットの詳細については，アニュアル「アプリケーションサーバ 機能解説 保守／移行編」の「4.7　JavaVM のスレッドダンプ」を参照してください。

(3)　エラーリポートファイル

ZGC を使用している場合，Java ヒープ領域の構成が SerialGC および G1GC の場合と異なります。そのため，エラーリポートに出力される Java ヒープ情報の出力フォーマットが異なります。エラーリポートファイルの出力フォーマットの詳細については，マニュアル「アプリケーションサーバ 機能解説 保守／移行編」の「5.8　JavaVM が出力するメッセージログ（標準出力およびエラーリポートファイル）」を参照してください。

7.20　ZGC 使用時の注意事項（JDK17 以降の場合）

ここでは，ZGC を使用する場合の注意事項について説明します。

- 外部から観測できるメモリ使用量

  ZGC では，複数の異なる範囲の仮想アドレス空間を 1 つの物理アドレス空間に割り当てているため，OS が提供するツールでメモリ使用量を確認したとき，実際の使用量よりも値が大きくなることがあります。

  例えば，Windows で取得できるワーキングセット，Linux で使用できる top コマンド，および ps コマンドで取得できる RES，RSS では，複数の仮想アドレス空間がカウントされるため，実際の約 3 倍のメモリ使用量が表示されることがあります。

- ユーザダンプおよびコアダンプのサイズ

  ZGC では，複数の異なる範囲の仮想アドレス空間を 1 つの物理アドレス空間に割り当てているため，ほかの GC を使用する場合よりも，ユーザダンプおよびコアダンプのサイズが大きくなります。

  JavaVM がプロセスダウンした場合に生成されるユーザダンプおよびコアダンプのサイズは，使用する仮想アドレス空間の使用量（物理アドレス空間が割り当てられている量）と同じのため，ユーザダンプおよびコアダンプのサイズは，Java ヒープ領域の最大サイズの約 3 倍になります。

- 必要な仮想アドレス空間のサイズ

  ZGC を使用している場合，ほかの GC を使用している場合よりも多くの仮想アドレス空間が必要です。これは，複数の仮想アドレス空間を同一の物理メモリにマッピングするという ZGC の仕組みによるものです。物理メモリの実際の使用サイズには影響しません。ZGC を使用している場合に必要な仮想アドレス空間のサイズは次のとおりです。

  - Java ヒープ領域の最大サイズが 4 テラバイト未満の場合

    必要な仮想アドレス空間のサイズは，次の計算式で算出した値です。ただし，最大値は 12 テラバイトです。

```
ZGC使用時に必要な仮想アドレス空間 = Javaヒープ領域の最大サイズ × 48
```

  - Java ヒープ領域の最大サイズが 4 テラバイト以上，8 テラバイト未満の場合

    必要な仮想アドレス空間のサイズは，24 テラバイトです。

  - Java ヒープ領域の最大サイズが 8 テラバイト以上，16 テラバイト以下の場合

    必要な仮想アドレス空間のサイズは，48 テラバイトです。

8 パフォーマンスチューニング（J2EE アプリケーション実行基盤）

この章では，J2EE アプリケーションを実行するシステムのパフォーマンスをチューニングする方法について説明します。

パフォーマンスチューニングによって動作環境を最適化することで，システムの性能を最大限に生かせるようになります。

バッチアプリケーション実行基盤のパフォーマンスチューニングについて検討する場合は，「9. パフォーマンスチューニング（バッチアプリケーション実行基盤）」を参照してください。

8.1　パフォーマンスチューニングで考慮すること

この節では，J2EE アプリケーション実行基盤のパフォーマンスチューニングで考慮することについて説明します。

8.1.1　パフォーマンスチューニングの観点

J2EE アプリケーション実行基盤のパフォーマンスチューニングは，次の観点で実施します。

- 同時実行数の最適化
- Enterprise Bean の呼び出し方法の最適化
- データベースアクセス方法の最適化
- タイムアウトの設定
- Web アプリケーションの動作の最適化
- CTM の動作の最適化
- そのほかの項目のチューニング

それぞれのポイントについて説明します。

(1)　同時実行数の最適化

同時実行数の最適化は，処理を多重化して CPU の処理能力を最大限に引き出して，システムのスループットを向上させることを目的とします。しかし，次のような場合，多重化しただけではスループットが向上しません。場合によっては，スループットが低下するおそれがあります。

- 入出力処理，排他処理などのボトルネックがある場合
- 最大スループットに到達している場合
- CPU の利用率が飽和した状態で多重度以上の負荷を掛けた場合
- 実行待ちキューのサイズが不適切な場合
- 階層的な最大実行数の設定が不適切な場合

パフォーマンスチューニングでは，これらを考慮しながら適切なチューニングを実施して，同時実行数の最適化を図ります。

(2)　Enterprise Bean の呼び出し方法の最適化

Enterprise Bean の呼び出し方法の最適化は，同じ J2EE アプリケーションや同じ J2EE サーバ内のコンポーネントを呼び出すときに，ローカルインタフェースやリモートインタフェースのローカル呼び出し機能を利用することで，不要なネットワークアクセスを削減することを目的とします。

次の機能を利用することで，RMI-IIOP 通信によって発生する不要なネットワークアクセスを削減できます。

- ローカルインタフェースの利用
- リモートインタフェースのローカル呼び出し機能の利用

また，引数や戻り値の渡し方を参照渡しにすることで，さらに処理性能を向上できる場合があります。パフォーマンスチューニングでは，アプリケーションやシステムの特徴によってこれらの機能を有効に活用して，処理性能の向上を図ります。

(3) データベースアクセス方法の最適化

データベースアクセス方法の最適化は，処理に時間が掛かるコネクションやステートメントを事前に生成しておくことで，データベースアクセス時のオーバーヘッドを削減することを目的とします。

パフォーマンスチューニングでは，次に示す機能を有効に活用することで，データベースアクセス処理を最適化し，スループットを向上させます。

- コネクションプーリング
- ステートメントプーリング（PreparedStatement および CallableStatement のプーリング）

(4) タイムアウトの設定

タイムアウトの設定は，システムのトラブル発生を検知して，リクエストの応答が返らなくなることを防ぎ，適宜リソースを解放することを目的とします。

設定できるタイムアウトには，次の種類があります。

- Web フロントシステムのタイムアウト
- バックシステムのタイムアウト
- トランザクションのタイムアウト
- データベースのタイムアウト

(5) Web アプリケーションの動作の最適化

Web アプリケーションの動作の最適化は，コンテンツの配置方法の検討やキャッシュの利用によって不要なネットワークアクセスを削減して処理速度を速めたり，負荷分散によってシステムのスループットの向上を図ったりすることを目的とします。

(6) CTM の動作の最適化

CTM の動作の最適化は，CTM で使用するプロセス間の通信間隔を最適化して通信負荷を軽減したり，トラブル発生時に迅速に検知して対処したりすることで，システムとしての性能を向上させることを目的とします。また，CTM によってリクエストの処理に優先順位を付けることで，重要なリクエストをすばやく処理するようにもチューニングできます。

(7) そのほかの項目のチューニング

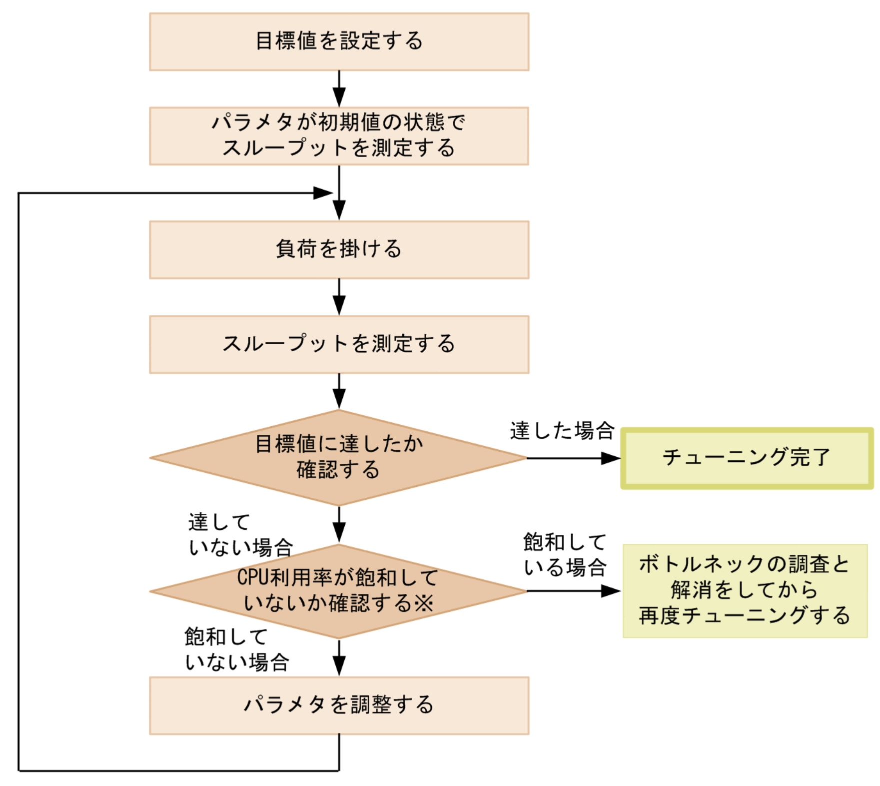
アプリケーションサーバでは，(1)～(6)で説明した項目以外にも，チューニングできる項目があります。必要に応じてチューニングを実施してください。

8.1.2 チューニング手順

パフォーマンスチューニングは，システムのパフォーマンスを生かす最適な設定を見つける作業です。構築した環境で，実際に処理を実行したり，模擬的な負荷を掛けたりしながら，パラメタの調整やボトルネックの調査，解消によってパフォーマンスを向上させていきます。

パフォーマンスチューニングの手順の例として，ここでは，同時実行数のチューニング手順を示します。

図 8-1 パフォーマンスチューニングの手順（同時実行数をチューニングする場合）



注※ パラメタを変更してもスループットが向上しない場合，飽和しています。

チューニング作業では，まず，目標値を決定します。ここでは，CPU 利用率などが該当します。

次に，各パラメタに初期値を設定した状態でのスループットを測定し，そのあとで模擬的な負荷を掛けながら各パラメタを調整して，目標値に近い最適な値を見つけていきます。模擬的な負荷は，専用のツールを使用して発生させます。

チューニングの際，CPU の利用率の測定には，OS に付属している監視ツールなどが利用できます。スループットの測定は，負荷発生ツールなどによって測定できます。また，稼働スレッド数など，アプリケーションサーバの稼働情報については，稼働情報収集機能などで確認できます。確認方法については，マニュ

アル「アプリケーションサーバ 機能解説 運用／監視／連携編」の「3. 稼働情報の監視（稼働情報収集機能）」を参照してください。

スループットが目標値に達したところで，パフォーマンスチューニングは完了です。なお，CPU 利用率が100%からかなり低い状態で飽和した場合は，システム上に入出力処理や排他処理などのボトルネックがあるおそれがあります。ボトルネックを調査し，対策してから，再度パフォーマンスチューニングを実行してください。アプリケーションサーバのシステムのボトルネックの調査には，性能解析トレースを利用できます。性能解析トレースの機能詳細，および性能解析トレースを利用して取得したトレースファイルの利用方法については，マニュアル「アプリケーションサーバ 機能解説 保守／移行編」の「7. 性能解析トレースを使用した性能解析」を参照してください。

8.1.3 アプリケーションの種類ごとにチューニングできる項目

チューニング項目は，アプリケーションの種類によって異なります。アプリケーションに含まれるコンポーネントごとのチューニング項目について，次に示します。

表 8-1 サーブレットと JSP で構成されるアプリケーション（Web アプリケーション）のチューニング項目

チューニング項目	利用できる機能	参照先
Web コンテナのリクエスト処理スレッド数の最適化	Web コンテナのリクエスト処理スレッド数の制御	8.3.3
同時実行数の最適化	Web アプリケーションでの同時実行スレッド数制御（Web コンテナ単位，Web アプリケーション単位，または URL グループ単位）	8.3.4
データベースアクセス方法の最適化	コネクションプーリング	8.5.1
	ステートメントプーリング	8.5.2
タイムアウトの設定	Web フロントシステムでのタイムアウトの設定	8.6.2
	J2EE アプリケーションのメソッド実行時間に対するタイムアウトの設定	8.6.7
Web アプリケーションの動作の最適化	静的コンテンツと Web アプリケーションの配置の切り分け	8.7.1
	静的コンテンツのキャッシュ	8.7.2

表 8-2 Enterprise Bean で構成されるアプリケーションのチューニング項目

チューニング項目	利用できる機能	参照先
同時実行数の最適化	Stateless Session Bean のインスタンスプーリング	8.3.5
	Stateful Session Bean のセッション制御	
	Message-driven Bean のインスタンスプーリング	

チューニング項目	利用できる機能	参照先
	CTM による同時実行数制御※（CTM を使用している場合）	8.3.6
Enterprise Bean の呼び出し方法の最適化	ローカルインタフェースの使用	8.4.1
	リモートインタフェースのローカル呼び出し最適化	8.4.2
	リモートインタフェースの参照渡し	8.4.3
データベースアクセス方法の最適化	コネクションプーリング	8.5.1
	ステートメントプーリング	8.5.2
タイムアウトの設定	バックシステムでのタイムアウトの設定	8.6.3
	トランザクションタイムアウトの設定	8.6.4
	データベースでのタイムアウトの設定	8.6.6
	J2EE アプリケーションのメソッド実行時間に対するタイムアウトの設定	8.6.7

注※　Stateless Session Bean だけが対象です。

また，CTM を使用したシステムの場合に設定できる，CTM の動作のチューニング項目を次の表に示します。CTM は，アプリケーションが Stateless Session Bean で構成されている場合に使用できます。

表 8-3　CTM の動作についてのチューニング項目

チューニング項目	利用できる機能	参照先
CTM の動作の最適化	CTM ドメインマネジャおよび CTM デーモンの稼働状態を監視する間隔のチューニング	8.8.1
	負荷状況監視間隔のチューニング	8.8.2
	CTM デーモンのタイムアウト閉塞の設定	8.8.3
	CTM で振り分けるリクエストの優先順位の設定	8.8.4

8.2　チューニングの方法

この節では，チューニングの方法について説明します。チューニングの方法は，設定対象の種類によって異なります。

8.2.1　J2EE サーバおよび Web サーバのチューニング

J2EE サーバおよび Web サーバのチューニングには，Smart Composer 機能の簡易構築定義ファイルを使用します。簡易構築定義ファイルでは，<configuration>タグ下の<logical-server-type>に設定対象とする論理サーバの種類（J2EE サーバまたは Web サーバ）を指定して，<param>タグ下でパラメタ名とその値を設定します。簡易構築定義ファイルの詳細については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「4.3　簡易構築定義ファイル」を参照してください。

Smart Composer 機能を使用できない場合，Web サーバのチューニングはファイルを編集して定義します。詳細については，マニュアル「HTTP Server」を参照してください。

8.2.2　アプリケーションまたはリソースのチューニング

アプリケーションおよびリソースのチューニングをする場合は，サーバ管理コマンドを使用します。

サーバ管理コマンドを使用する場合は，属性ファイルを編集します。属性ファイルの詳細については，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」を参照してください。

8.2.3　CTM の動作のチューニング

CTM のチューニングには，Smart Composer 機能の簡易構築定義ファイルを使用します。簡易構築定義ファイルでは，<configuration>タグ下の<logical-server-type>に設定対象とする論理サーバの種類（CTM ドメインマネジャまたは CTM）を指定して，<param>タグ下でパラメタ名とその値を設定します。

簡易構築定義ファイルの詳細については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「4.3　簡易構築定義ファイル」を参照してください。

8.2.4　それ以外の項目のチューニング

このほか，チューニングで使用するパラメタには，API で設定する項目や，データベースで設定する項目などがあります。これらの設定については，それぞれの節のチューニングパラメタの説明を参照してください。

8.3　同時実行数を最適化する

この節では，アプリケーションのリクエストの同時実行数を最適化するための考え方とチューニング方法について説明します。

8.3.1　同時実行数制御および実行待ちキュー制御の考え方

アプリケーションサーバのシステムでアプリケーションのスループットを向上させるためには，複数のリクエストを多重処理することが有効です。複数のスレッドで多重にリクエストを処理した方が，単一スレッドで一度に 1 リクエストずつ処理するのに比べて，多くの場合，スループットを向上させられます。

ただし，入出力処理，排他処理などにボトルネックがあったり，すでに最大スループットに到達していたりする場合は，多重化によるスループットの向上はできません。このため，同時実行数のチューニングは，次の点を確認しながら進めます。

入出力処理，排他処理などのボトルネックの排除

スレッドを多重化しても CPU 利用率が少ないままでスループットが向上しない場合は，アプリケーションのデータベースアクセスなどの入出力処理や排他処理などにボトルネックがあるおそれがあります。この場合は，ボトルネックになっている処理を特定して，排除してからチューニングする必要があります。例えば，データベースアクセス方法をチューニングしたり，排他処理方法を変更したりして，入出力処理，排他処理などのボトルネックを取り除きます。

最大スループットの確認

リクエストの多重度を大きくしてスレッド数を増やしていくと，スレッド数の増加に伴って CPU の空き時間が減少し，さらに増やしていくと，ほとんど CPU の空き時間がない状態になります。この状態になると，スレッドを増やしてもスループットは上がりません。

これは，CPU がボトルネックになっている状態であり，マシン単体の性能として，限界に達している状態です。つまり，この時点のスループットが，マシン単体でのアプリケーションの最大スループットになります。

これ以上のスループットを確保したい場合は，CPU やマシン数を増加させるなど，ハードウェアの増強が必要になります。

同時実行数制御によるスループットの維持

CPU の利用率が飽和した状態で，さらに多重度を大きくしてスレッドを増やした場合，実行可能な状態で CPU が割り当てられないスレッドが増えます。この状態では，スレッド間のロックが競合したり，スレッドのコンテキストスイッチが多発したりするので，スループットが低下するおそれがあります。

また，スレッド数を増やすと，アプリケーションサーバ内ではスレッド数に応じてメモリ使用量が増えます。つまり，スループット低下とメモリ使用量の増加を防ぐためには，スレッド数の増加を実行可能なスレッド数までに制限する必要があります。

同時実行数制御機能を適切に使用することで，最大同時実行数をチューニングして，リクエストの多重度を大きくした場合でも，最大同時実行数分以上のリクエストの実行を待たせることができます。この

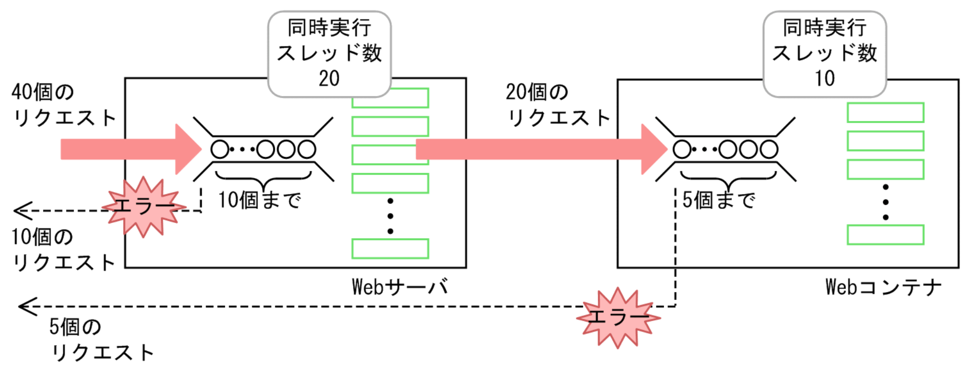
結果，一時的に過負荷状態が発生したり，負荷のピークの状態になったりしても，高いスループットを維持できます。

実行待ちキューサイズの調整

アプリケーションサーバに到着したリクエストが同時実行数の上限を超えたとき，リクエストをキューに登録することで，実行中のほかのリクエストの処理が終わるまでそのリクエストを待たせることができます。しかし，ほかのリクエストの処理が終わるまでリクエストを待たせておくためのキューのサイズ（**実行待ちキューサイズ**）に上限を設定している場合，実行待ちキューが上限に達したあとで到着したリクエストについては，実行待ちキューには登録されないでクライアントにエラーとして返却されます。このため，実行待ちキューに上限を設定する場合は，待たせるキューに必要な数を確保しておく必要があります。

実行待ちキューへのリクエストの登録とエラーの返却の考え方を，次の図に示します。

図 8-2　実行待ちキューへのリクエストの登録とエラーの返却



・Webサーバでは，40個のリクエストに対して，20個のスレッドを同時実行して，10個のリクエストを実行待ちキューに登録できます。
　実行待ちキューの上限以上の10個のリクエストについては，エラーが返却されます。

・Webコンテナでは，20個のリクエストに対して，10個のスレッドを同時実行して，5個のリクエストを実行待ちキューに登録できます。
　実行待ちキューの上限以上の5個のリクエストについては，エラーが返却されます。

ポイント

リクエストを実行待ちキューで待たせるのは，一時的な過負荷状態の場合や負荷のピークの場合にエラーが発生するのを防ぐためです。エラーが返却されるのを避けるために実行待ちキューサイズをむやみに増やすのは，本質的な解決にはなりません。同時実行数を増やしたり，必要に応じて CPU やマシン数を増設したりするなどの対処をしてください。

また，クライアント側でタイムアウトを設定している場合は，リクエストが実行待ちキューに登録されてから実際に実行されるまでの時間が掛かり過ぎると，リクエストの実行前にタイムアウトが発生してエラーになるおそれがあります。

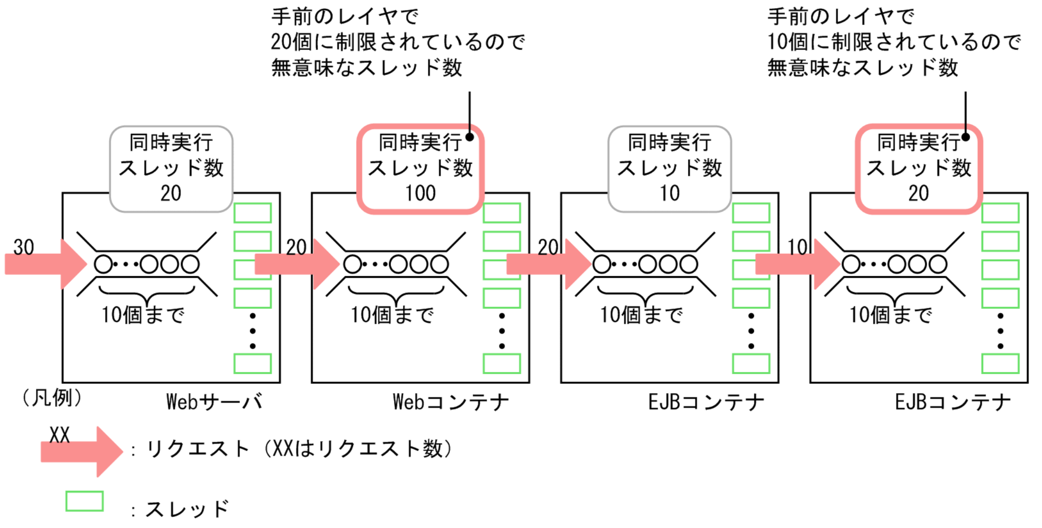
このため，実行待ちキューサイズは，最適なサイズに設定してください。

階層的なアプリケーションでの最大同時実行数のバランス

あるレイヤだけをチューニングして同時実行数を増やしても，システム全体の性能は向上しません。これは，システム全体としての性能は，性能が低いレイヤに制約されてしまうためです。

特定のレイヤだけを最適化したために無意味な設定を含んでしまった例を，次の図に示します。

図 8-3　特定のレイヤだけを最適化したために無意味な設定を含んでしまった例



また，同時実行数を増やすと，実際にはリクエストを処理していないアイドル状態の場合にも，メモリなどのリソースをむだに消費するおそれがあります。

このため，同時実行数を制御する場合には，システム全体を見渡して，適切な同時実行数をそれぞれのレイヤに設定するようにしてください。

8.3.2　最大同時実行数と実行待ちキューを求める手順

最大同時実行数および実行待ちキューサイズは，次の手順に従ってチューニングできます。

1. **負荷発生ツールなどを利用して，リクエストの多重度を増やします。**

   このとき，サーバ側の CPU 利用率が 80%～90%に達した場合は，手順 2.に進みます。80%～90%に達しない，低い状態でスループットが向上しなくなった場合は，入出力処理や排他処理などにボトルネックがあることが考えられます。この場合は，ボトルネックになっている処理を特定して，性能を改善します。

2. **サーバ側の CPU 利用率が 80%～90%に達した多重度を，最大同時実行数としてチューニングパラメタに設定します。**

   この状態でのスループットが，単体マシンの最大スループットになります。これ以上のスループットを求めたい場合は，ハードウェアの増強が必要です。

3. **負荷発生ツールなどでさらに高い負荷を掛けて，最大スループットが維持できるかどうかを確認します。**

   維持できない場合は，最大スループット以上の負荷が掛からないように，チューニングパラメタを調整します。

4. **実際のシステムでの一時的な過負荷状態，および負荷のピークの状態でのリクエスト数を見積もり，実行待ちキューサイズを決定します。**

5. 階層的な構造を持つアプリケーションでは，各レイヤでの同時実行数および実行待ちキューサイズのバランスが取れるよう，調整します。

8.3.3　リクエスト処理スレッド数を制御する

Web フロントシステムの場合，Web ブラウザなどのクライアントからのリクエストは，アプリケーションサーバが作成するリクエスト処理スレッドによって処理されます。リクエスト処理スレッド数を適切に制御することで，処理性能の向上が図れます。

ここでは，アプリケーションサーバでのリクエスト処理スレッド数を制御する目的と，チューニングの指針について説明します。

なお，ここでは，J2EE サーバ上の Web コンテナが作成するリクエスト処理スレッド数のチューニングの方法について説明します。

> **参考**
>
> J2EE サーバの前段に置くリバースプロキシとして HTTP Server を使用している場合は，HTTP Server の設定で同様のチューニングができます。詳細は，マニュアル「HTTP Server」を参照してください。

(1)　リクエスト処理スレッド数を制御する目的

リクエスト処理スレッド数を J2EE サーバが動作しているホストの性能やクライアントからのアクセス状況に合わせてチューニングすることで，性能向上を図れます。

リクエスト処理スレッドの生成は，負荷が高い処理です。リクエスト処理スレッドをあらかじめ生成してプールしておくことで，Web ブラウザなどのクライアントからのリクエスト処理要求時の負荷を軽くして，処理性能を高めることができます。

Web コンテナでは，J2EE サーバ起動時にリクエスト処理スレッドをまとめて生成してプールしておき，Web ブラウザなどのクライアントからリクエスト処理要求があった場合にそれを利用するようにできます。これによって，リクエスト処理要求時の処理性能の向上を図れます。なお，プールしているスレッドの数を監視しておくことで，プールしているスレッド数が少なくなった場合はさらに追加生成して，プールに確保しておくこともできます。

ただし，使用しないスレッドを大量にプールしておくと，むだなリソースを消費します。このため，システムの処理内容に応じて，プールするリクエスト処理スレッド数を適切に制御し，場合によっては不要なスレッドを削除することが必要です。

リクエスト処理スレッドの制御では，これらを考慮して，パラメタに適切な値を設定してください。

(2) 設定の指針

リクエスト処理スレッド数の制御では，次のパラメタを使用してチューニングできます。

- J2EE サーバ起動時に生成するリクエスト処理スレッドの数
- Web クライアントとの接続数（リクエスト処理スレッド数）の上限数
- 接続数の上限を超えた場合の TCP/IP の Listen キュー（バックログ）の最大値
- リクエスト処理スレッド数の最大数

これらのパラメタを設定するときには，次の点に留意してください。

- 提供するサービスの内容によっては，J2EE サーバ起動直後から大量のリクエストを処理する必要があります。この場合は，J2EE サーバ起動時に生成するリクエスト処理スレッドの数に，大きな値を指定してください。
- リクエスト処理スレッドの最大数を大きくしておくと，クライアントからのアクセスが急に増加した場合にも，処理性能を下げることなく迅速に対応できます。ただし，多くのスレッドをプールしておくと，多くのリソースが消費されます。このため，急な増加が予想されるアクセス数を見積もって，適切な数のスレッドがプールされるように，注意して設定してください。
- 一度作成したスレッドを削除しないでプールし続けたい場合は，リクエスト処理スレッドの最大数を，リクエスト処理スレッドの最小数と同じ値にしてください。

このほか，Web アプリケーションの同時実行スレッド数との関係についても留意してください。Web アプリケーションの同時実行スレッド数については，「8.3.4 Web アプリケーションの同時実行数を制御する」を参照してください。

8.3.4 Web アプリケーションの同時実行数を制御する

Web アプリケーションの同時実行数制御では，Web フロントシステムの場合に，Web サーバが Web ブラウザなどのクライアントから受け付けた HTTP リクエストの処理を，同時に幾つのスレッドで実行するかを制御します。

(1) 同時実行スレッド数制御の違い

Web コンテナ単位，Web アプリケーション単位，および URL グループ単位の同時実行スレッド数制御の違いは次のとおりです。

Web コンテナ単位

Web コンテナ全体で同時に HTTP リクエストを処理するスレッド数を設定できます。

Web アプリケーション単位

Web コンテナ上で動作する Web アプリケーションごとに，同時に HTTP リクエストを処理するスレッド数を設定できます。

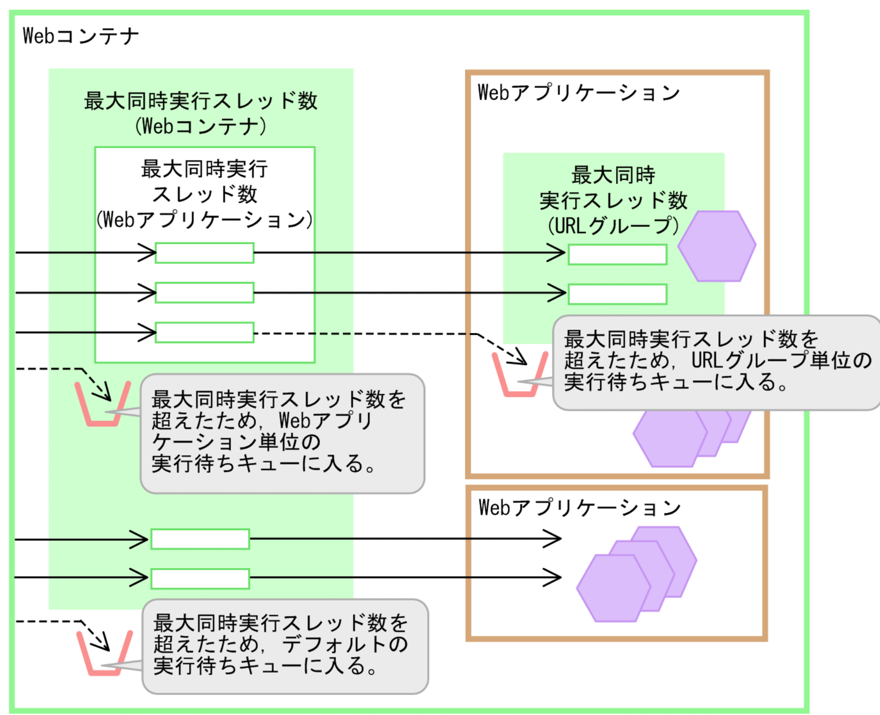
URL グループ単位

HTTP リクエストを Web アプリケーション内の特定の業務処理（業務ロジック）に対応する URL に振り分ける場合，振り分け先 URL の処理ごとに，同時に HTTP リクエストを処理するスレッド数を設定できます。

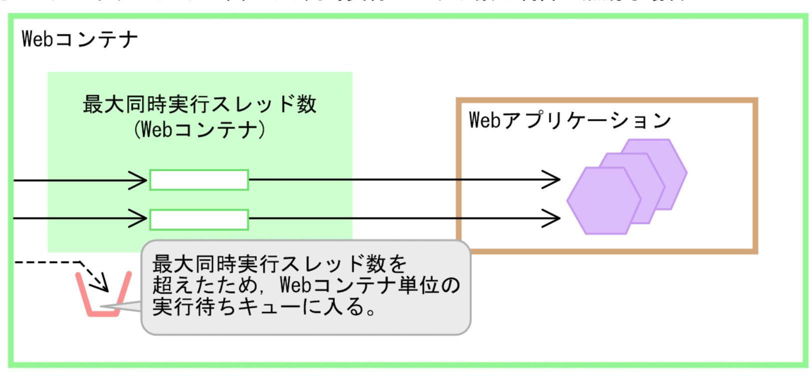
Web コンテナ単位，Web アプリケーション単位，および URL グループ単位の同時実行スレッド数の関係を次の図に示します。

図 8-4　Web コンテナ単位，Web アプリケーション単位，および URL グループ単位の同時実行スレッド数の関係

●Webアプリケーション単位での同時実行スレッド数の制御が有効な場合



●Webアプリケーション単位での同時実行スレッド数の制御が無効な場合



（凡例）
→：実行されるリクエスト
--->：実行待ちキューに入るリクエスト
：実行待ちキュー　：スレッド　：業務ロジック

Web アプリケーションに対するリクエストの実行は，Web コンテナ単位，Web アプリケーション単位，および URL グループ単位に設定した同時実行スレッド数に制限されます。Web コンテナ単位，Web アプリケーション単位および URL グループ単位に設定した同時実行スレッド数を超えるリクエストは，それぞれの実行待ちキューに入ります。

(2) 選択の指針

同時実行スレッド数制御の単位を選択するときの指針について説明します。

なお，同時実行スレッド数を制御する機能の詳細については，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(Web コンテナ)」の「2.13　同時実行スレッド数の制御の概要」を参照してください。

- Web アプリケーション単位の選択の指針

  Web アプリケーション単位の同時実行数を制御することで，J2EE サーバが TCP 接続要求だけではなく，Web アプリケーションの実行待ちキューを管理できるようになります。このため，J2EE サーバ上で実行する Web アプリケーションが一つだけの場合でも，Web アプリケーション単位の同時実行スレッド数を設定することをお勧めします。

  Web アプリケーション単位での同時実行スレッド数の設定は，Web コンテナ単位で設定する場合に比べて，次のような利点があります。

  - Web アプリケーションごとの同時実行スレッド数に上限を設けることで，特定の業務に対応する Web アプリケーションへのリクエストが増大した場合に，その Web アプリケーションが Web コンテナ全体の処理能力を占有しないようにできます。これによって，ほかの業務も滞りなく実行できます。
  - CPU や I／O 処理に掛かる負荷が異なる複数の Web アプリケーションが Web コンテナ上にある場合，それぞれの条件に適した同時実行スレッド数が設定できます。
  - Web アプリケーションごとにリクエストの実行待ちキューサイズが設定できるので，Web アプリケーションの特徴に応じた実行待ちキュー管理ができます。また，Web アプリケーション単位の実行待ちキュー以上のリクエストが送信された場合には，クライアントに HTTP レスポンスコードで通知できます。

  なお，Web アプリケーション単位の同時実行スレッド数は，稼働中の J2EE サーバでも動的に変更できます。稼働中の J2EE サーバで実行する Web アプリケーションの同時実行スレッド数の動的変更の手順については，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(Web コンテナ)」の「2.17.2　同時実行スレッド数の動的変更の流れ」を参照してください。

- URL グループ単位の選択の指針

  Web アプリケーション単位で同時実行スレッド数を制御している場合に，さらに業務ロジック単位で同時実行スレッド数の制御をしたいときには，URL グループ単位で同時実行スレッド数を制御します。

  Web アプリケーションが次のような業務ロジックを含む場合，URL グループ単位の設定を検討してください。

  - ほかの処理に影響を受けないで優先して実行したい業務ロジック
  - ほかの処理に比べて処理時間が掛かる，または CPU や I／O の負荷が大きい業務ロジック

URL グループ単位での同時実行スレッド数の設定は，Web アプリケーション単位だけの設定に比べて，次のような利点があります。

- 重要度が高い業務ロジック（URL グループ）には，確実に実行するためのスレッド数を割り当てられます。これによって，ほかの業務ロジックに対するリクエスト数が増大した場合も，Web アプリケーション全体の同時実行スレッド数をその業務ロジックに占有されないで，重要度が高い業務ロジックを実行できます。
- 処理時間が掛かる業務ロジック（URL グループ）の同時実行数に上限を設けることで，特定の業務ロジックによって Web アプリケーション全体の同時実行数が占有されないように制御できます。
- Web アプリケーション内に CPU や I／O の負荷が異なる複数の業務ロジック（URL グループ）がある場合は，業務ロジックに応じた同時実行数が設定できます。
- Web アプリケーション内の業務ロジック（URL グループ）ごとにリクエストの待ち行列長（実行待ちキューのキューサイズ）が設定できるので，業務ロジックの特徴に応じた実行待ちキューを管理できます。また，この URL グループ単位の実行待ちキューがあふれた場合，クライアントに HTTP レスポンスコード 503（Service Temporarily Unavailable）を通知できます。

8.3.5 Enterprise Bean の同時実行数を制御する

ここでは，Enterprise Bean の同時実行数を制御する方法について，Enterprise Bean の種類ごとに説明します。なお，インスタンスプーリングおよびセッション制御による Enterprise Bean の同時実行数制御は，EJB コンテナの機能を利用して実現します。EJB コンテナの機能の詳細は，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(EJB コンテナ)」の「2. EJB コンテナ」を参照してください。

(1) Enterprise Bean の同時実行数制御で使用できる機能の種類

Enterprise Bean の同時実行数を制御する場合，EJB コンテナの機能である，次の 2 種類の機能を利用できます。

インスタンスプーリング

Enterprise Bean のインスタンスを事前に作成しておくことで，クライアントからリクエストが送信されたときにすぐ処理を実行できるようにする機能です。プールするインスタンス数の上限値を設定することで，上限値以上のリクエストの実行を待たせることができます。これによって，同時実行数が制御できます。

> **参考**
>
> 運用時に CTM の同時実行数を動的に変更する場合は，上限値は無制限に設定してください。

セッション制御

セッション内で，同時に生成できるセッション（インスタンス）数を制限する機能です。

なお，Enterprise Bean の種類ごとに使用できる機能が異なります。

Enterprise Bean の種類ごとに使用できる同時実行数制御の機能を次の表に示します。

表 8-4　Enterprise Bean の種類ごとに使用できる同時実行数制御の機能

Enterprise Bean の種類	使用できる制御機能
Stateless Session Bean※	インスタンスプーリング
Stateful Session Bean	セッション制御
Entity Bean	インスタンスプーリングとセッション制御
Message-driven Bean	インスタンスプーリング

注※　Stateless Session Bean の同時実行数を制御する場合は，CTM を利用することをお勧めします。CTM を利用して同時実行数を制御する方法については，「8.3.6　CTM を使用して同時実行数を制御する」で説明します。

なお，Enterprise Bean の同時実行数制御のうち，実行待ちリクエストをキューの概念で管理できるのは，Message-driven Bean だけです。Message-driven Bean では，JMS のキューを使用して実行待ちリクエストを管理します。これ以外の Enterprise Bean では，同時実行数以上のリクエストが送信された場合，設定に応じて次のどれかの処理が実行されます。

- インスタンスに空きが出るまで待ち続ける
- すぐに例外としてクライアントに返却する
- インスタンス取得用に設定したタイムアウト時間が経過したら，例外としてクライアントに返却する（Stateless Session Bean の method-ready プールまたは Entity Bean の pool プールの場合）

(2)　Stateless Session Bean の同時実行数制御

Stateless Session Bean では，インスタンスプーリングを利用できます。通常の同時アクセス数をインスタンスプーリングの最小値として設定して，想定している最大同時アクセス数以上の値を，インスタンスプーリングの上限値に指定します。これによって，通常のアクセス時にはインスタンスを生成する時間を省略できるので，処理性能が向上します。さらに，アクセス数が増大した場合でも，想定している最大同時アクセス数までは処理でき，それ以上のリクエストの実行は待たせるように制御できます。

なお，デフォルトの設定の場合，上限値以上のリクエストがエラーで返却されることはありません。プールされているインスタンスに空きができるまで待ち続けます。エラーで返却したい場合は，必要に応じてインスタンス取得待ちのタイムアウトを設定してください。

(3)　Stateful Session Bean の同時実行数制御

Stateful Session Bean では，クライアントごとにセッションごとの状態があるため，厳密な同時実行数制御はできません。ただし，セッション単位での流量制御（セッション制御）ができます。

想定している最大同時セッション数以上の値を，セッション制御の上限値として指定します。アクセス数が増大して，想定している最大同時セッション数以上のアクセスがあった場合には，セッションを確立しないで，クライアントに例外（java.rmi.RemoteException）を通知できます。

ポイント

リクエスト単位で同時実行制御が必要な場合は，サーブレット，JSP または Stateless Session Bean を経由して Stateful Session Bean を呼び出すことで，Web アプリケーションの同時実行数制御またはインスタンスプーリングを利用した同時実行数制御ができるようになります。

(4) Entity Bean の同時実行数制御

Entity Bean では，クライアントごとに管理する状態を持つセッションの上限値の設定とインスタンスプーリングによって，同時実行数の制御ができます。なお，セッション数が上限に達した場合，インスタンスプーリングに空きがあっても，新しいセッションでのリクエストは実行できません。

セッション数の上限を超えた場合は，セッションの作成に失敗した時点ですぐにクライアントに例外（java.rmi.RemoteException）が通知されます。また，デフォルトの設定の場合，インスタンスプール数の上限値以上のリクエストは，エラーで返却されることはなく，プールされているインスタンスに空きができるまで待ち続けます。エラーで返却したい場合は，必要に応じてインスタンス取得待ちのタイムアウトを設定してください。

なお，Entity Bean に対するリクエストの実行では，データベースへのアクセスが発生します。このため，同時実行できるリクエスト数は，データベースにアクセスするためのコネクション数にも制限されます。

(5) Message-driven Bean の同時実行数制御

Message-driven Bean では，インスタンスプーリングを利用できます。想定している最大メッセージ数以上の値を，インスタンスプーリングの上限値として指定します。これによって，メッセージ到着時にインスタンスを生成する時間を省略できるので，処理性能が向上します。さらに，メッセージ数が増大した場合でも，インスタンスプーリングの上限値以上のメッセージは，実行を待たせるように制御できます。

なお，Message-driven Bean では，JMS のキューによって，到着メッセージを実行待ちキューで管理できます。

参考

TP1 インバウンド連携機能を使用して OpenTP1 の SUP から Message-driven Bean を呼び出す場合や，CJMS プロバイダを使用して Message-driven Bean を呼び出す場合，ここで説明した内容以外に，使用するコンポーネントを考慮した同時実行数を検討する必要があります。詳細は，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(コンテナ共通機能)」の「4. OpenTP1 からのアプリケーションサーバの呼び出し（TP1 インバウンド連携機能)」，または「7. CJMS プロバイダ」を参照してください。

8.3.6　CTM を使用して同時実行数を制御する

CTM を使用している場合，Stateless Session Bean の同時実行数を制御できます。

CTM は，J2EE サーバとは独立したプロセス群です。EJB クライアントと J2EE サーバ間の Stateless Session Bean の呼び出しを中継し，Stateless Session Bean を呼び出す際に，同時実行数制御をします。なお，CTM による同時実行数制御の単位は J2EE アプリケーションです。

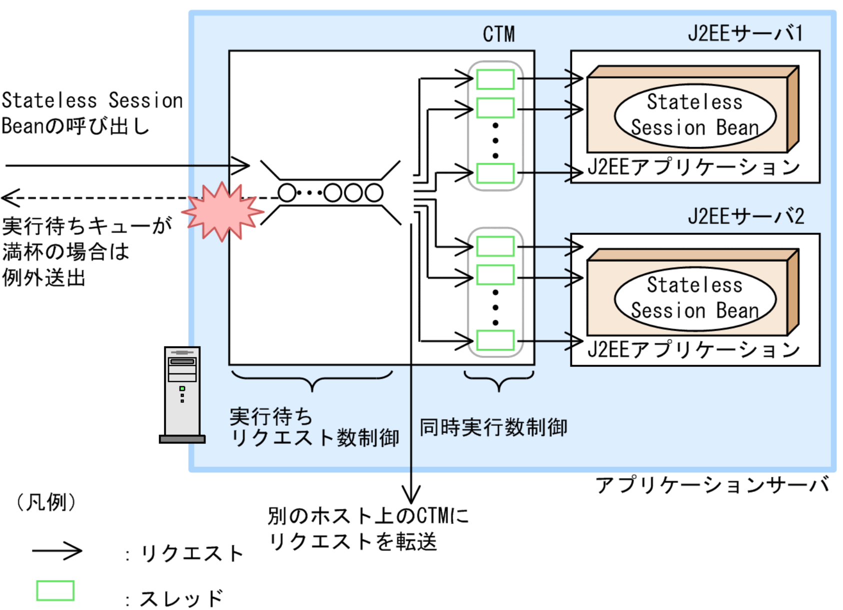
(1)　CTM を使用した Stateless Session Bean の同時実行数制御

CTM による同時実行数制御（流量制御）によって，次に示すチューニングができます。

- CPU や I／O 処理に掛かる負荷が異なる，複数の J2EE アプリケーションが J2EE サーバ上にある場合に，それぞれの条件に適した同時実行数が設定できます。
- CTM で実行待ちキュー（スケジュールキュー）を管理するので，実行待ちリクエスト数を一定数以下に保ち，それ以上のリクエストが送信された場合にはクライアントに例外を通知できます。
- 特定の J2EE サーバの負荷が高い場合に，ほかの J2EE サーバにリクエストを振り分けられます。

CTM による Stateless Session Bean の同時実行数制御の例を次の図に示します。

図 8-5　CTM による Stateless Session Bean の同時実行数制御の例



ポイント

CTM は同一ホスト上の J2EE サーバでの Stateless Session Bean の呼び出しを制御して，そのホスト上での同時実行スレッド数を制御できます。アプリケーションサーバマシンのマシンスペックにも左右されますが，1 台のマシン当たり CTM デーモンを 1 プロセス起動して，J2EE サーバを 2～4 プロセス起動する構成を推奨します。

なお，CTM による同時実行スレッド数は，稼働中の CTM デーモンでも動的に変更できます。

CTM の同時実行数を制御する機能の詳細，および稼働中の CTM デーモンで実行する CTM の同時実行スレッド数動的変更の手順については，マニュアル「アプリケーションサーバ 機能解説 拡張編」の「3.4 リクエストの流量制御」を参照してください。

(2) EJB コンテナのインスタンスプーリングとの使い分けの指針

CTM を使用して同時実行数を制御することをお勧めします。なお，CTM による同時実行数制御は，EJB コンテナでのインスタンスプーリングを利用した同時実行数制御と併用できます。

EJB コンテナの機能を使用した同時実行数制御に加えて，CTM によって同時実行数を制御するメリットは，次のとおりです。

- ある EJB コンテナで同時実行数が上限に達した場合に，ほかの J2EE サーバにリクエストを振り分けられます。
- 同時実行数が上限に達していなくても，特定の J2EE サーバの負荷が高い場合，ほかの J2EE サーバにリクエストを振り分けられます。
- CTM で実行待ちキュー（スケジュールキュー）の管理をして，実行待ちリクエスト数を一定に保ち，それ以上のリクエストを受け付けた場合はクライアントにエラーを通知できます。

ポイント

CTM による同時実行数制御と EJB コンテナでのインスタンスプーリングを併用する場合，Stateless Session Bean のインスタンスプーリング数は CTM の同時実行数以上に設定する必要があります。

また，運用時に CTM の同時実行数を動的に変更する場合は，Stateless Session Bean のインスタンスプーリング数の上限は無制限にする必要があります。なお，デフォルトでは無制限に設定されています。デフォルトから変更しないでください。

8.3.7 同時実行数を最適化するためのチューニングパラメタ

ここでは，同時実行数の最適化で使用するチューニングパラメタの設定方法についてまとめて示します。

(1) リクエスト処理スレッド数

Web コンテナのリクエスト処理スレッド数のチューニングパラメタの設定方法について説明します。

次の表に示す項目を Smart Composer 機能で設定します。パラメタは，簡易構築定義ファイルに定義します。

表 8-5 Web コンテナのリクエスト処理スレッド数のチューニングパラメタ

設定項目	設定対象	設定個所（パラメタ名）
J2EE サーバ起動時に生成するリクエスト処理スレッド数	論理 J2EE サーバ（j2ee-server）	webserver.connector.nio_http.min_threads
Web クライアントまたはリバースプロキシとの接続数の上限※1	論理 J2EE サーバ（j2ee-server）	webserver.connector.nio_http.max_connections
Web クライアントまたはリバースプロキシとの接続数の上限を超えた場合に使用される TCP/IP の Listen キュー（バックログ）の最大値※2	論理 J2EE サーバ（j2ee-server）	webserver.connector.nio_http.backlog
リクエスト処理スレッド数の最大数	論理 J2EE サーバ（j2ee-server）	webserver.connector.nio_http.max_threads

注※1

常設コネクションの場合，リバースプロキシから接続されるコネクション数以上の値を定義してください。

注※2

常設コネクションの場合，キューとして使用できません。

なお，J2EE サーバの前段に置くリバースプロキシとして HTTP Server を使用している場合のチューニングパラメタについては，マニュアル「HTTP Server 」を参照してください。

(2) Web アプリケーションの同時実行数

URL グループ単位，Web アプリケーション単位，または Web コンテナ単位に設定します。

(a) URL グループ単位の同時実行数

URL グループ単位の同時実行数のチューニングパラメタの設定方法について説明します。設定項目ごとに設定方法と設定個所が異なります。

次の表に示す項目は，Smart Composer 機能で設定します。パラメタは，簡易構築定義ファイルに定義します。

表 8-6 URL グループ単位の同時実行数のチューニングパラメタ（Smart Composer 機能で設定する項目）

設定項目	設定対象	設定個所（パラメタ名）
Web コンテナ単位での最大同時実行スレッド数	論理 J2EE サーバ（j2ee-server）	webserver.connector.nio_http.max_servlet_execute_threads
デフォルトの実行待ちキューサイズ	論理 J2EE サーバ（j2ee-server）	webserver.container.thread_control.queue_size

次の表に示す項目は，サーバ管理コマンド（cjsetappprop）で設定します。パラメタは，WAR 属性ファイルに定義します。

表 8-7　URL グループ単位の同時実行数のチューニングパラメタ（サーバ管理コマンド（cjsetappprop）で設定する項目）

設定項目	設定個所（パラメタ名）
Web アプリケーション単位での最大同時実行スレッド数	<thread-control>タグ下の<thread-control-max-threads>
Web アプリケーションの占有スレッド数	<thread-control>タグ下の<thread-control-exclusive-threads>
Web アプリケーション単位の実行待ちキューサイズ	<thread-control>タグ下の<thread-control-queue-size>
URL グループ単位の同時実行スレッド数制御の定義名	<thread-control><urlgroup-thread-control>タグ下の<urlgroup-thread-control-name>
URL グループ単位での最大同時実行スレッド数	<thread-control><urlgroup-thread-control>タグ下の<urlgroup-thread-control-max-threads>
URL グループ単位の占有スレッド数	<thread-control><urlgroup-thread-control>タグ下の<urlgroup-thread-control-exclusive-threads>
URL グループ単位の実行待ちキューサイズ	<thread-control><urlgroup-thread-control>タグ下の<urlgroup-thread-control-queue-size>
URL グループ単位の制御対象となる URL パターン	<thread-control><urlgroup-thread-control>タグ下の<urlgroup-thread-control-mapping>

(b)　Web アプリケーション単位の同時実行数

Web アプリケーション単位の同時実行数をチューニングするパラメタの設定方法について説明します。設定項目ごとに設定方法と設定個所が異なります。

次の表に示す項目は，Smart Composer 機能で設定します。パラメタは，簡易構築定義ファイルに定義します。

表 8-8　Web アプリケーション単位の同時実行数のチューニングパラメタ（Smart Composer 機能で設定する項目）

設定項目	設定対象	設定個所（パラメタ名）
Web コンテナ単位での最大同時実行スレッド数	論理 J2EE サーバ（j2ee-server）	webserver.connector.nio_http.max_servlet_execute_threads
デフォルトの実行待ちキューサイズ	論理 J2EE サーバ（j2ee-server）	webserver.container.thread_control.queue_size

次の表に示す項目は，サーバ管理コマンド（cjsetappprop）で設定します。パラメタは，WAR 属性ファイルに定義します。

表 8-9　Web アプリケーション単位の同時実行数のチューニングパラメタ（サーバ管理コマンド（cjsetappprop）で設定する項目）

設定項目	設定個所（パラメタ名）
Web アプリケーション単位での最大同時実行スレッド数	<thread-control-max-threads>
Web アプリケーションの占有スレッド数	<thread-control-exclusive-threads>
Web アプリケーション単位の実行待ちキューサイズ	<thread-control-queue-size>

(c)　Web コンテナ単位の同時実行数

Web コンテナ単位の同時実行数をチューニングするパラメタの設定方法について説明します。

次の表に示す項目を Smart Composer 機能で設定します。パラメタは，簡易構築定義ファイルに定義します。

表 8-10　Web コンテナ単位の同時実行数のチューニングパラメタ

設定項目	設定対象	設定個所（パラメタ名）
Web コンテナ単位での最大同時実行スレッド数	論理 J2EE サーバ（j2ee-server）	webserver.connector.nio_http.max_servlet_execute_threads

(3)　Enterprise Bean の同時実行数

Enterprise Bean の同時実行数は，Enterprise Bean 単位に設定します。Enterprise Bean の種類ごとに説明します。

(a)　Stateless Session Bean の同時実行数

Stateless Session Bean の同時実行数をチューニングするパラメタの設定方法について説明します。

次の表に示す項目を，サーバ管理コマンド（cjsetappprop）で設定します。パラメタは，Session Bean 属性ファイルに定義します。

表 8-11　Stateless Session Bean の同時実行数のチューニングパラメタ

設定項目	設定個所（パラメタ名）
プールで管理するインスタンスの最大値	<stateless><pooled-instance>タグ下の<maximum>※
プールで管理するインスタンスの最小値	<stateless><pooled-instance>タグ下の<minimum>

注※　運用時に CTM の同時実行数を動的に変更する場合は，最大値は無制限（「0」）に設定する必要があります。

(b)　Stateful Session Bean の同時実行数

Stateful Session Bean の同時実行数をチューニングするパラメタの設定方法について説明します。

次の表に示す項目を，サーバ管理コマンド（cjsetappprop）で設定します。パラメタは，Session Bean 属性ファイルに定義します。

表 8-12　Stateful Session Bean の同時実行数のチューニングパラメタ

設定項目	設定個所（パラメタ名）
クライアントから作成可能なセッションの上限	<stateful>タグ下の<maximum-active-sessions>
使われていないインスタンスが削除されるまでの時間（分）	<stateful>タグ下の<removal-timeout>

(c)　Entity Bean の同時実行数

Entity Bean の同時実行数をチューニングするパラメタの設定方法について説明します。

次の表に示す項目を，サーバ管理コマンド（cjsetappprop）で設定します。パラメタは，Entity Bean 属性ファイルに定義します。

表 8-13　Entity Bean の同時実行数のチューニングパラメタ

設定項目	設定個所（パラメタ名）
クライアントから作成可能な Entity Bean の最大値	<maximum-instances>
使われていない Entity Bean の EJBObject が削除されるまでの時間（分）	<entity-timeout>

(d)　Message-driven Bean の同時実行数

Message-driven Bean の同時実行数をチューニングするパラメタの設定方法について説明します。

次の表に示す項目を，サーバ管理コマンド（cjsetappprop）で設定します。パラメタは，Message-driven Bean 属性ファイルに定義します。

表 8-14　Message-driven Bean の同時実行数のチューニングパラメタ

設定項目	設定個所（パラメタ名）
プールで管理するインスタンスの数の最大値	<pooled-instance><maximum>
プールで管理するインスタンスの数の最小値	<pooled-instance><minimum>

(4)　CTM で制御する同時実行数

CTM で制御する同時実行数をチューニングするパラメタの設定方法について説明します。CTM デーモン，アプリケーション，および Stateless Session Bean に設定する項目があります。

- CTM デーモンに設定する項目
  次の表に示す項目は，Smart Composer 機能で設定します。パラメタは，簡易構築定義ファイルに定義します。

表 8-15　CTM で制御する同時実行数のチューニングパラメタ（Smart Composer 機能で設定する項目）

設定項目	設定対象	設定個所（パラメタ名）
CTM が制御するスレッドの最大値およびキューごとのリクエストの登録数	論理 CTM（component-transaction-monitor）	ctm.DispatchParallelCount

- アプリケーションまたは Stateless Session Bean に設定する項目

次の表に示す項目は，サーバ管理コマンドで設定します。パラメタは，アプリケーション属性ファイルまたは Session Bean 属性ファイルに定義します。

表 8-16　CTM で制御する同時実行数のチューニングパラメタ（サーバ管理コマンドで設定する項目）

設定項目	定義ファイル	設定対象	設定個所（パラメタ名）
アプリケーションを CTM による同時実行数制御の対象にするかどうか	アプリケーション属性ファイル	アプリケーション	<managed-by-ctm>
アプリケーションの同時実行スレッド数	アプリケーション属性ファイル	アプリケーション	<scheduling>タグ下の<parallel-count>
Stateless Session Bean を CTM による同時実行数制御の対象にするかどうか	Session Bean 属性ファイル	Stateless Session Bean	<enable-scheduling>
Bean 単位のキューについての設定※	Session Bean 属性ファイル	Stateless Session Bean	<scheduling>タグ下の<parallel-count>

注※　スケジュールキューを Bean 単位に配置する場合に必要です。スケジュールキューの配置方法は，アプリケーション属性ファイルの<scheduling-unit>タブに設定します。

8.4　Enterprise Bean の呼び出し方法を最適化する

この節では，Enterprise Bean の呼び出し方を最適化する方法について説明します。

通常，Enterprise Bean は，リモートインタフェースを使用して RMI-IIOP 経由で呼び出されます。しかし，この方法の場合，同じ J2EE アプリケーションや同じ J2EE サーバ内で動作している Enterprise Bean から呼び出すときにも，リモート接続と同じオーバーヘッドが掛かってしまいます。

これに対して，Enterprise Bean の呼び出し方を最適化してスループットの向上を図るために，次の呼び出し方法が使用できます。

- ローカルインタフェースを使用した呼び出し
- リモートインタフェースのローカル呼び出し
- リモートインタフェースの参照渡し

これらの呼び出し方法には，それぞれ，次の表に示す特徴があります。この表では，標準仕様に準拠しているか（標準仕様），性能は向上するか（性能），位置透過性はあるか（位置透過），保守性に優れているか（保守性）の四つの特徴で比較しています。アプリケーションやシステムの特徴に応じて使い分けてください。

表 8-17　Enterprise Bean の呼び出し方法の特徴

呼び出し方法の種類	標準仕様	性能	位置透過	保守性
ローカルインタフェース	○	○	×	○
リモートインタフェースのローカル呼び出し	○	△	○	○
リモートインタフェースの参照渡し	×	○	△	×

（凡例）

○：対応している。／優れている。

△：対応しているが，一部制限がある。／やや悪い。

×：対応していない。／悪い。

なお，ローカルインタフェースの詳細については，EJB の仕様を参照してください。

リモートインタフェースのローカル呼び出しおよびリモートインタフェースの参照渡しの詳細については，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(EJB コンテナ)」の「2.13　EJB のリモートインタフェースの呼び出し」を参照してください。

8.4.1　ローカルインタフェースを使用する

アプリケーション開発時に，J2EE 標準仕様に準拠したローカルインタフェースを使用する方法です。

ローカルインタフェースは，Enterprise Bean の呼び出しを通常の同一スレッドのメソッド呼び出しとして実行する機能です。J2EE の標準機能なので，ポータビリティがあります。ただし，クライアント側，サーバ側の両方でローカル呼び出し専用のローカルインタフェースを使用したプログラムを作成する必要があるため，位置透過性はなくなります。

なお，ローカルインタフェースは，参照渡しによる呼び出しになります。また，同一 J2EE サーバ内であっても，異なる J2EE アプリケーション間での呼び出しには適用できません。

8.4.2　リモートインタフェースのローカル呼び出し最適化機能を使用する

同一アプリケーション内または同一 J2EE サーバ内での Enterprise Bean のリモートインタフェースによる呼び出しを，同一スレッドの延長によって実行する機能（**ローカル呼び出し最適化機能**）を使用する方法です。クライアント側，サーバ側の両方でリモートインタフェースを使用したプログラムを作成するので，位置透過性が保てます。

なお，デフォルトの状態では，同一アプリケーション内のローカル呼び出し最適化機能を使用する設定になっています。

8.4.3　リモートインタフェースの参照渡し機能を使用する

ローカル呼び出し最適化機能を使用する場合に，引数や戻り値で使用するオブジェクト型などのサイズが大きいデータの受け渡しを参照渡しにすることで，処理の高速化を図る方法です。

次のような条件の場合に，高速化が期待できます。

表 8-18　リモートインタフェースの参照渡しで高速化が期待できる条件

項目	処理内容
ビジネスメソッドの処理	仮引数経由で取得したデータに対して，直接変更をしないで参照だけする処理
ビジネスメソッドの引数と戻り値	配列や Collection などのクラスで大量のデータを受け渡す処理
呼び出す Enterprise Bean の位置	同一の J2EE アプリケーション内，または同一の J2EE サーバ内

ただし，リモートインタフェースでのデータの受け渡しは本来値渡しです。このため，値渡しを前提に作成されたプログラムは正しく動作しなくなるおそれがあるので注意してください。正しく動作しなくなるプログラムの例を次に示します。

図 8-6　正しく動作しなくなるプログラムの例

```
int[] int_array = {0, 1, 2, 3, 4, 5};
int result = adder.add(int_array);

if (int_array[0] == result) {
  // pass by reference
} else {
  // pass by value
}
```

呼び出す側のプログラム

```
public int add(int[] array) {
  for (int I = 1; I < array.length; i++)
  {
    array[0] += array[i];
    // 仮引数に対する変更
  }
  return array[0];
}
```

呼び出される側のプログラム

また，次の場合，この機能を使用しても効果がありません。

- プリミティブ型（単純型）の場合（常に値渡しになるため）
- 受け渡すデータのサイズが小さい場合

この機能は，J2EE サーバ単位または Enterprise Bean 単位で設定できます。

8.4.4　Enterprise Bean の呼び出し方法を最適化するためのチューニングパラメタ

ここでは，Enterprise Bean の呼び出し方法の最適化で使用するチューニングパラメタの設定方法についてまとめて示します。

(1)　ローカルインタフェースの使用

アプリケーション作成時に，J2EE で定義されているローカルインタフェースを使用してください。

(2)　リモートインタフェースのローカル呼び出し機能の使用

リモートインタフェースのローカル呼び出し機能のチューニングパラメタの設定方法について説明します。

次の表に示す項目を Smart Composer 機能で設定します。パラメタは簡易構築定義ファイルに定義します。

表 8-19　リモートインタフェースのローカル呼び出し機能のチューニングパラメタ

設定項目	設定対象	設定個所（パラメタ名）
ローカル呼び出し最適化機能の適用範囲	論理 J2EE サーバ（j2ee-server）	ejbserver.rmi.localinvocation.scope

(3)　リモートインタフェースの参照渡し機能の使用

リモートインタフェースの参照渡し機能のチューニングパラメタの設定方法について説明します。

次の表に示す項目は，Smart Composer 機能で設定します。パラメタは簡易構築定義ファイルに定義します。

表 8-20　リモートインタフェースの参照渡し機能のチューニングパラメタ（Smart Composer 機能で設定する項目）

設定項目	設定対象	パラメタ名
リモートインタフェースの参照渡し機能の使用（J2EE サーバ単位）	論理 J2EE サーバ（j2ee-server）	ejbserver.rmi.passbyreference

次の表に示す項目は，サーバ管理コマンド（cjsetappprop）で設定します。パラメタは Session Bean 属性ファイルまたは Entity Bean 属性ファイルに定義します。

表 8-21　リモートインタフェースの参照渡し機能のチューニングパラメタ（サーバ管理コマンド（cjsetappprop）で設定する項目）

設定項目	パラメタ名
リモートインタフェースの参照渡し機能の使用（Enterprise Bean 単位）	<pass-by-reference>

8.5　データベースへのアクセス方法を最適化する

データベースへのアクセス方法の最適化について説明します。

J2EE サーバでは，データベースアクセス方法のチューニングに次の 2 種類の方法を使用できます。

- コネクションプーリング
- ステートメントプーリング

8.5.1　コネクションプーリングを使用する

ここでは，J2EE サーバでコネクションプーリングを使用する利点と，コネクションプーリングに関連して使用できる機能について説明します。

また，JPA プロバイダでは，DB Connector のコネクションを取得します。

(1)　コネクションプーリングを使用する利点

データベースなどの EIS とのコネクションの確立は，負荷が高くなる処理です。コネクションプーリングを使用することで，負荷を軽くできます。**コネクションプーリング**は，J2EE サーバによって一度取得，生成したコネクションをプールしておき，それを再利用することで処理性能の向上を図る機能です。データベースなどにアクセスするたびにコネクションを再取得する場合に比べて，性能劣化を防ぐ効果があります。

なお，EIS との接続方法によって，使用できる機能が異なります。詳細は，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(コンテナ共通機能)」の「3.14.1　コネクションプーリング」を参照してください。

なお，デフォルトの設定では，コネクションプーリングは有効になっています。

(2)　使用できる機能

コネクションプーリングで使用できる機能と設定時の指針について示します。

コネクションプーリングでは，次の機能を使用できます。

- プールするコネクションの最大値と最小値を指定する
- プール内のコネクションを検査して無効なコネクションを破棄する※1
- コネクション取得失敗時にリトライする※2
- スイーパによって使わないコネクションをプールから削除する※2
- コネクションウォーミングアップによってあらかじめコネクションをプールしておく
- コネクション枯渇時のコネクション取得要求を取得待ちキューに入れる

- コネクションプール内の不要なコネクションを段階的に減少させる

注※1　この機能を使用する場合，コネクションの検査にタイムアウトを設定することもできます。

注※2　これらの機能は，デフォルトの設定では無効になっています。必要に応じて使用してください。

(a)　プールするコネクションの最小値と最大値を指定する

コネクションプーリングを設定する場合は，プールするコネクションの最小値と最大値を設定します。無制限にした場合はコネクションが無制限に確立されます。

最小値および最大値は，定常状態で発生するデータベースなどの EIS への同時アクセス数，トランザクション数，業務の同時実行数などを参考にして決定してください。

プールするコネクションの最大値は，同時実行スレッド数，Session Bean インスタンスプールとの間で，次の関係式を満たすように設定することを推奨します。

プールするコネクションの最大値≧Session Bean インスタンスプール≧同時実行スレッド数

(b)　プール内のコネクションを検査して無効なコネクションを破棄する

プール内のコネクションに障害が発生していないかどうかをコネクション取得時または定期的にチェックして，障害が発生したコネクションをプールから削除します（**コネクション障害検知**）。コネクション取得時に障害が発生しているコネクションを取得してしまうことを防ぎ，接続に失敗する可能性を低くできます。この機能は，DB Connector を使用してデータベースにアクセスするためのコネクションに対して有効です。なお，SQL Server を使用する場合，DB Connector の selectMethod プロパティに direct を指定したときには，この機能は使用できません。

コネクション障害検知を実行するタイミングの使い分けの指針は次のとおりです。業務の種類に応じて使い分けてください。

表 8-22　コネクション障害検知を実行するタイミングの使い分けの指針

業務の種類	タイミング
• EIS との接続の失敗が許容されない業務 • コネクション取得頻度が低い業務	コネクション取得時に障害検知を実行する設定をお勧めします。 コネクション取得時に掛かる処理時間は，コネクション障害検知を実行しない場合に比べて多く掛かりますが，障害が発生しているコネクションを取得する可能性が低くなります。
• コネクション取得頻度が高い業務 • 障害が発生したコネクションを取得してもある程度許容されるような業務	定期的に障害検知を実行する設定をお勧めします。チェック間隔をある程度長くすることで，コネクション障害検知処理のために性能が劣化することを防げます。ただし，障害が発生しているコネクションを取得してしまうおそれがあります。

また，コネクション障害検知を実行する場合，コネクション障害検知の実行にタイムアウトを設定できます。サーバ障害やネットワーク障害が発生してリソースからの応答が返らない場合，コネクション障害検知の実行に対しても応答が返らなくなることがあります。タイムアウトを設定しておくことで，リソース

からの応答が返らない場合も，コネクション障害検知処理を終了して，処理を継続できます。なお，この場合は，コネクションに障害が発生していると判断されます。

ポイント

- コネクション障害検知にタイムアウトを設定した場合，システム内で，コネクションプールのコネクション数に応じたコネクション管理スレッドが生成されます。このため，コネクション障害検知にタイムアウトを設定すると，設定しない場合に比べて多くのメモリを消費するので，注意が必要です。

  コネクション管理スレッドは，コネクションプールのコネクション数の最大値の 2 倍の数が作成されます。必要なメモリ使用量を適切に見積もってください。
- コネクション管理スレッドは，コネクション数調節機能のタイムアウトと共通で使用されます。このため，コネクション障害検知にタイムアウトを設定した場合，コネクション数調節機能のタイムアウトも有効になります。
- コネクション障害検知のタイムアウトを有効にしてコネクション障害検知を実施する場合，コネクションプールから取り除いた未使用コネクションは，コネクションプール内のコネクション数としてカウントされません。そのため，コネクションプール内のコネクションとコネクションプールから取り除いた未使用コネクションの総数が，コネクションプールのコネクション数の最大値を一時的に超える場合があります。

(c) コネクション取得失敗時にリトライする

コネクションの取得に失敗したときに，ユーザプログラムでリトライする必要がなくなります。業務処理のレスポンスが下がっても，障害発生による業務停止を防ぎたい場合に設定してください。

リトライ回数とリトライ間隔を設定する場合の指針について次に示します。

- リトライ回数を増やしたり，リトライ間隔を大きくしたりすると，コネクション取得処理が発生した場合に，待ち時間が発生するおそれがあります。
- リトライ間隔×リトライ回数の時間が長過ぎると，RMI-IIOP 通信のタイムアウトなどが発生しますが，タイムアウト後もリトライを続けます。このため，リトライ間隔×リトライ回数の時間は，タイムアウト値以下の値になるように設定してください。
- 同一リソースのコネクションを使用する業務の所要時間にも左右されますが，10 秒以上を指定することを推奨します。

(d) スイーパによって使わないコネクションをプールから自動削除する

一定時間使用しなかったコネクションをスイーパによってプールから自動削除します。コネクションを未使用のままプーリングすると問題が発生する場合に設定します。リソースアダプタ単位に設定できます。

(e) コネクションウォーミングアップによってあらかじめコネクションをプールしておく

J2EE サーバの起動時またはリソースアダプタのスタート時にコネクションを最小値まであらかじめ取得してプールしておく機能です。コネクションプールの使用を開始した直後の，アプリケーションからのコネクション要求のレスポンスを向上できます。ただし，J2EE サーバ起動またはリソースアダプタのスタート時の処理時間が掛かります。

(f) コネクション枯渇時のコネクション取得要求を待ち状態にする

プールしているコネクションをすべて使い切っている状態でコネクション取得要求があった場合に，コネクション取得要求を待ち状態にする機能です。

使用中のコネクションが解放されるか，コネクションが破棄されてコネクション最大数よりも少なくなった場合に，すぐにコネクション取得要求を再開できます。また，待ち時間も設定できるので，一定時間を経過してコネクションを取得できなかった場合はエラーにできます。

(g) コネクションプール内の不要なコネクションを段階的に減少させる

コネクションプール内に不要なコネクションがある場合に，コネクション数を段階的に減らす機能です（**コネクション数調節機能**）。

一定間隔でコネクションプールの数をチェックします。チェック直前までの間の最大同時実行コネクション数を基準として，プール内にそれ以上の数のコネクションがある場合は，多い分のコネクションを削除します。これによって，プール内のコネクションは実際の稼働実績に適した数になるので，コネクション生成コストの削減や，リソース資源の節約ができます。

また，コネクションの削除処理にタイムアウトを設定できます。タイムアウトには，コネクション管理スレッドを使用します。なお，コネクション管理スレッドは，コネクション障害検知のタイムアウトでも使用されます。

ポイント

コネクション管理スレッドは，コネクション障害検知のタイムアウトと共通で使用されます。このため，コネクション数調節機能にタイムアウトを設定した場合，コネクション障害検知のタイムアウトも有効になります。

また，コネクション数調節機能を使用すると，コネクションプールのコネクション数を調整するために取り除いた未使用コネクションは，コネクションプール内のコネクション数としてカウントされません。そのため，コネクションプール内のコネクションとコネクションプールから取り除いた未使用コネクションの総数が，コネクションプールのコネクション数の最大値を一時的に超える場合があります。

8.5.2　ステートメントプーリングを使用する

ここでは，ステートメントプーリングを使用する利点と，設定の指針について説明します。

この機能は，DB Connector を利用している場合に使用できる機能です。また，使用できるかどうかは，EIS との接続方法によって異なります。XDM/RD E2 11-01 以前のバージョンの場合，ステートメントプーリングは使用できません。詳細は，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(コンテナ共通機能)」の「3.14.4　ステートメントプーリング」を参照してください。

(1)　ステートメントプーリングを使用する利点

データベースにアクセスするときに必要な SQL 文やストアドプロシジャなどのステートメントの生成は，負荷が高くなる処理です。ステートメントプーリングを使用することで，負荷を軽くできます。**ステートメントプーリング**は，一度生成したステートメントである PreparedStatement と CallableStatement をプーリングしておき，それを再利用することで処理性能の向上を図る機能です。データベースなどにアクセスするたびにステートメントを再生成する場合に比べて，処理性能が向上します。

なお，PreparedStatement および CallableStatement とは，それぞれ，JDBC の API であるjava.sql.PreparedStatement と java.sql.CallableStatement のインスタンスです。

(2)　設定の指針

ステートメントプーリングを利用するためには，アプリケーション開発時に次の点に留意する必要があります。

- 同じシグネチャを持つ java.sql.Connection#prepareStatement メソッドを使用します。
- java.sql.Connection#prepareStatement メソッドでは同じ引数値を指定します。

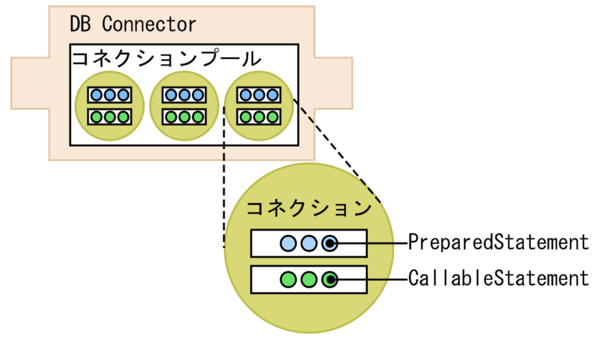
これらを満たさないアプリケーションでは，ステートメントプーリングは有効に動作しません。

また，ステートメントプーリングを使用するときには，コネクションプーリングとの関連を理解した上で使用してください。アプリケーション構成および環境設定時に留意する点を次に示します。

- 物理コネクションのインスタンスごとに PreparedStatement および CallableStatement がプールされます。このため，コネクション取得時にプールされているコネクションが複数ある場合には，ステートメントがまだプールされていないコネクションが割り当てられることがあります。
- それぞれのステートメントのインスタンスをプールするときには，使用している JDBC ドライバのコネクションごとの上限値を考慮してプールサイズを設定する必要があります。

コネクションプールとステートメントプールの関係を次の図に示します。

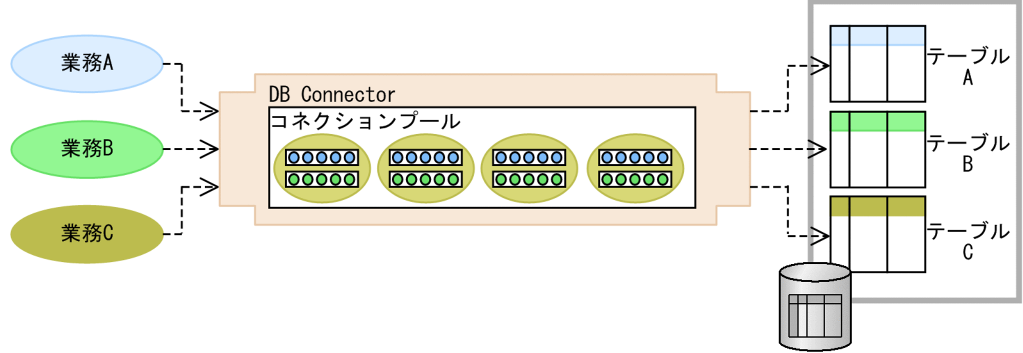
図 8-7　コネクションプールとステートメントプールの関係



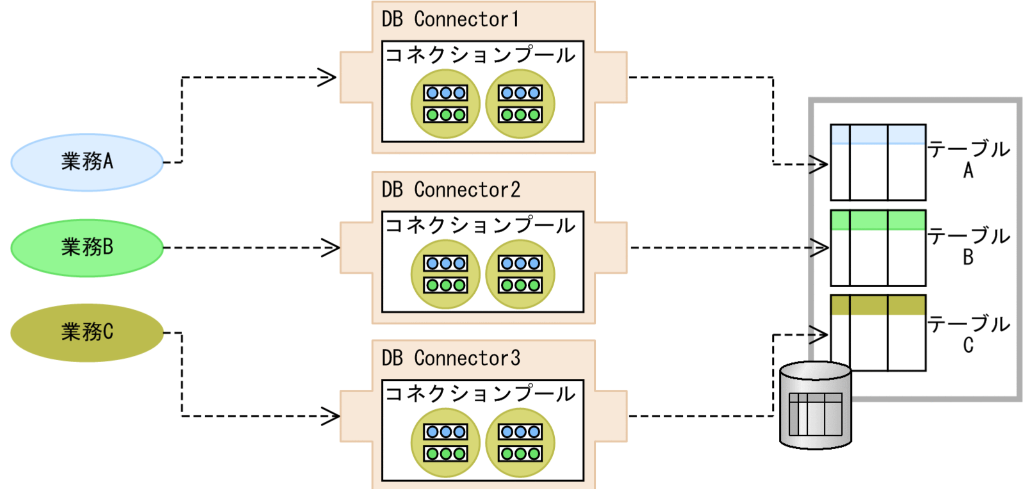
ステートメントプーリングのプールサイズを決めるときには，業務処理の内容に応じて，コネクションプーリングとの関係を考慮した上で決めてください。例えば，異なる業務処理で同じデータベースを使用する場合であっても，異なるテーブルにアクセスする場合や異なる SQL 文を使用する場合は，一つの DB Connector でコネクションプールとステートメントプールのサイズを大きくするよりも，業務処理ごとに別々の DB Connector を用意して，その業務で必要な分だけの小さなコネクションプールやステートメントプールを用意する方が，効率が良い場合があります。

図 8-8　業務処理に応じたコネクションプールとステートメントプールの利用

●複数の業務で一つの大きなプールを使用する方法



●業務ごとに一つの小さなプールを占有して使用する方法



ただし，DB Connector を複数用意してプールサイズを小さくすると，業務ごとにプールされているコネクション数の余裕がなくなり，アクセスのピーク時にコネクション不足が発生するおそれがあります。こ

のため，業務の同時実行数などを詳細に見積もって，コネクション数不足ができるだけ発生しないようにチューニングしてください。また，プールのサイズは，業務処理ごとに，どのくらいのステートメントを利用するかによって調整してください。

一つの DB Connector を利用する場合のステートメントプールサイズは，次の値を集計して決定します。なお，DB Connector が内部でステートメントプールを一つ利用することを考慮してプールサイズを見積もってください。

PreparedStatement のプールサイズを設定する指針とサイズの算出方法

コネクションごとの JDBC ドライバのリソース制限を上限として，同一の DB Connector を利用する PreparedStatement の利用数を基に算出してください。なお，JDBC ドライバのリソース制限については，使用している JDBC ドライバの制限値に従ってください。

利用数は，次の数を集計して求めます。

- サーブレットおよび JSP から，異なる引数を指定して java.sql.Connection#prepareStatement メソッドを呼び出している数
- Session Bean および Entity Bean（BMP）から，異なる引数を指定して java.sql.Connection#prepareStatement メソッドを呼び出している数
- Entity Bean（CMP）に対して，J2EE サーバが内部的に PreparedStatement で使用する SQL の数
- Message-driven Bean から，異なる引数を指定して java.sql.Connection#prepareStatement メソッドを呼び出している数

CallableStatement のプールサイズを設定する指針とサイズの算出方法

コネクションごとの JDBC ドライバのリソース制限を上限として，同一の DB Connector を利用する CallableStatement の利用数を基に算出してください。なお，JDBC ドライバのリソース制限については，使用している JDBC ドライバの制限値に従ってください。

利用数は，次の数を集計して求めます。

- サーブレットおよび JSP から，異なる引数を指定して java.sql.Connection#prepareCall メソッドを呼び出している数
- Session Bean および Entity Bean（BMP）から，異なる引数を指定して java.sql.Connection#prepareCall メソッドを呼び出している数
- Message-driven Bean から，異なる引数を指定して java.sql.Connection#prepareCall メソッドを呼び出している数

8.5.3 データベースへのアクセス方法を最適化するためのチューニングパラメタ

ここでは，データベースへのアクセス方法の最適化で使用するチューニングパラメタの設定方法についてまとめて示します。

(1)　コネクションプーリング

コネクションプーリングのチューニングパラメタの設定方法について説明します。

次の表に示す項目は，リソースアダプタ単位に，サーバ管理コマンド（cjsetresprop/cjsetrarprop）で設定します。パラメタは，Connector 属性ファイルに定義します。

表 8-23　コネクションプーリングのチューニングパラメタ

設定項目	設定個所（パラメタ名）※1
コネクションプールにプールするコネクションの最小値	<property>タグに指定する MinPoolSize
コネクションプールにプールするコネクションの最大値	<property>タグに指定する MaxPoolSize
プール内のコネクションに障害が発生しているかどうかをチェックする方法の選択	<property>タグに指定する ValidationType
プール内のコネクションに障害が発生しているかどうかのチェックを定期的に実行する場合の間隔	<property>タグに指定する ValidationInterval
コネクション枯渇時にコネクション取得要求をキューで管理するかどうかの選択	<property>タグに指定する RequestQueueEnable
コネクション枯渇時のコネクション取得要求をキューで管理する場合の待ち時間	<property>タグに指定する RequestQueueTimeout
コネクションの取得に失敗した場合のリトライ回数※2	<property>タグに指定する RetryCount
コネクションの取得に失敗した場合のリトライ間隔※2	<property>タグに指定する RetryInterval
コネクションの最終利用時刻からコネクションを自動破棄するかを判定するまでの時間	<property>タグに指定する ConnectionTimeout
コネクションの自動破棄（コネクションスイーパ）が動作する間隔	<property>タグに指定する SweeperInterval
コネクションのウォーミングアップを使用するかどうかの選択	<property>タグに指定する Warmup
コネクション障害検知にタイムアウト時間を設定するかどうかの選択※3	<property>タグに指定する NetworkFailureTimeout
コネクション数調節機能での削除処理にタイムアウト時間を設定するかどうかの選択※3	
コネクション数調節機能が動作する間隔	<property>タグに指定する ConnectionPoolAdjustmentInterval

注※1　クラスタコネクションプール機能（互換機能）を使用している場合は，メンバリソースアダプタで設定します。

注※2　クラスタコネクションプール機能（互換機能）を使用している場合は，設定できません。

注※3　コネクション障害検知のタイムアウトとコネクション数調節機能のタイムアウトでは，共通のコネクション管理メソッドを使用します。このため，どちらかのタイムアウトを設定すると，コネクション障害検知とコネクション数調節機能の両方のタイムアウトが有効になります。コネクション障害検知，およびコネクション数調節機能のタイムアウト時間を変更したい場合は，簡易構築定義ファイルで設定する J2EE サーバ用のパラメタ（ejbserver.connectionpool.validation.timeout）の設定を変更してください。詳細は，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「4.11.2　J2EE サーバ用ユーザプロパティを設定するパラメタ」を参照してください。

ポイント

HiRDB または XDM/RD E2 でコネクションプーリング機能を使用する場合，次の設定を推奨します。

- HiRDB でコネクションプーリング機能を使用する場合
  HiRDB のサーバからコネクションを切断されないように HiRDB クライアント環境変数を設定してください。詳細は，マニュアル「HiRDB UAP 開発ガイド」を参照してください。
- XDM/RD E2 でコネクションプーリング機能を使用する場合
  DB コネクションサーバのコントロール空間起動制御文／サーバ空間起動制御文の SVINTERVAL パラメタに「0」を指定します。このパラメタについては，マニュアル「VOS3　Database Connection Server」を参照してください。

上記を設定しなかった場合，プールしているコネクションが，タイムアウトによってデータベースから切断されることがあります。

- 障害検知機能を使用している場合
  障害検知機能によって，タイムアウトでデータベースから切断されたコネクションを破棄します。コネクションは正常に取得できます。
- 障害検知機能を使用していない場合
  タイムアウトでデータベースから切断されたコネクションを取得します。

(2)　ステートメントプーリング

ステートメントプーリングのチューニングパラメタの設定方法について説明します。

次の表に示す項目は，サーバ管理コマンド（cjsetresprop/cjsetrarprop）で設定します。パラメタは，Connector 属性ファイルに定義します。

表 8-24　ステートメントプーリングのチューニングパラメタ

設定項目	パラメタ名※
物理コネクションごとにプールする PreparedStatement の数	<config-property>タグに指定する PreparedStatementPoolSize
物理コネクションごとにプールする CallableStatement の数	<config-property>タグに指定する CallableStatementPoolSize

注※　クラスタコネクションプール機能（互換機能）を使用している場合は，メンバリソースアダプタで設定します。

8.6　タイムアウトを設定する

アプリケーションサーバのシステムでは，トラブル発生時にリクエストの応答が戻ってこない状態になることを防ぐために，幾つかのポイントにタイムアウトを設定できます。

この節では，システム全体でタイムアウトが設定できるポイントと，設定する場合の指針について説明します。

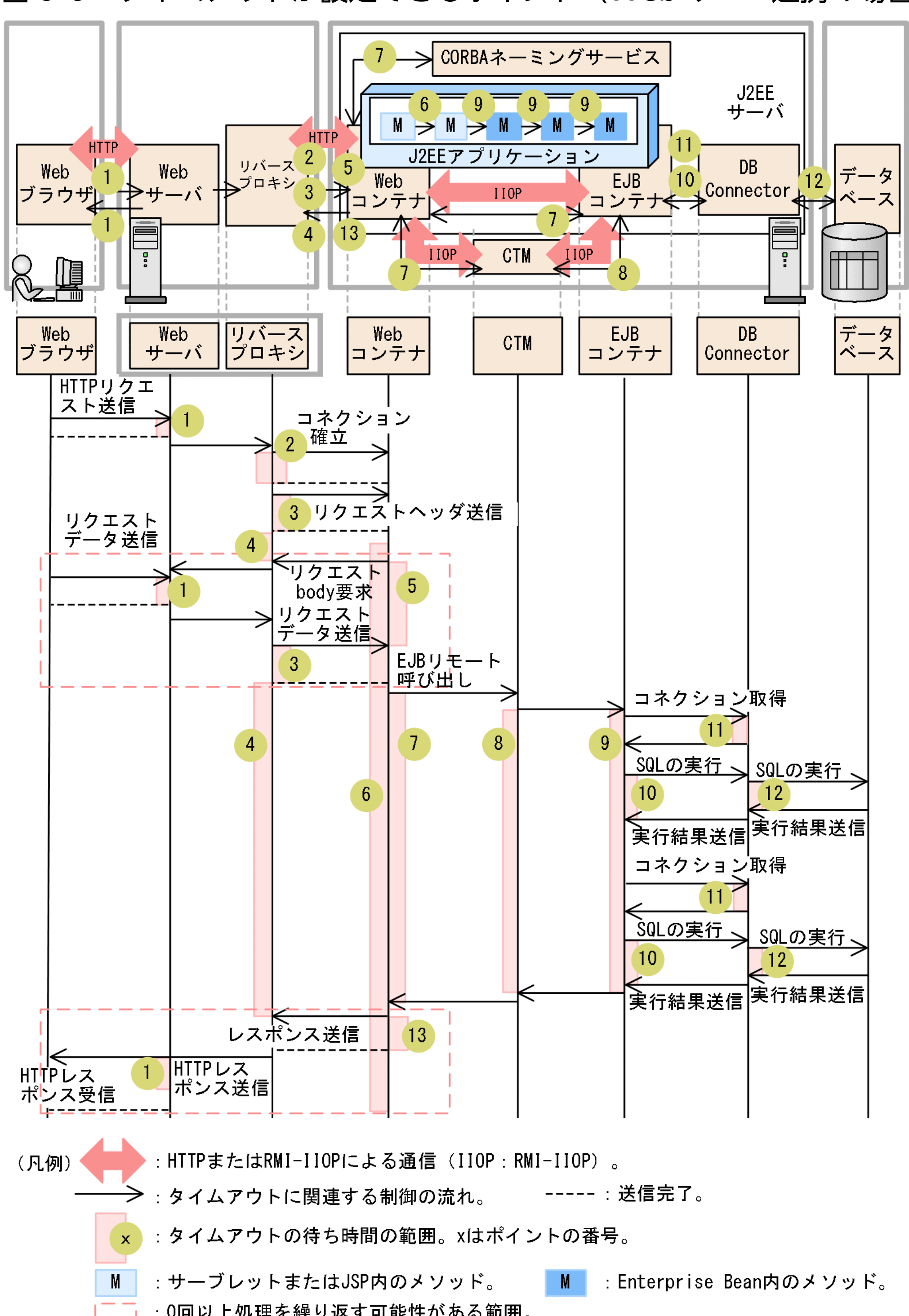
参考

TP1 インバウンド連携機能を使用して OpenTP1 からアプリケーションサーバを呼び出す場合，この節で説明する内容のほか，OpenTP1 側の設定を考慮したタイムアウトの設定が必要です。詳細は，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(コンテナ共通機能)」の「4. OpenTP1 からのアプリケーションサーバの呼び出し（TP1 インバウンド連携機能)」を参照してください。

8.6.1　タイムアウトが設定できるポイント

J2EE アプリケーションを実行するシステムでは，次の図に示すポイントにタイムアウトが設定できます。なお，次の図は，クライアントが Web ブラウザの場合です。また，Web サーバと連携する場合と，Web サーバを経由しないで J2EE サーバに直接リクエストを送受信する場合で，ポイントが異なります。

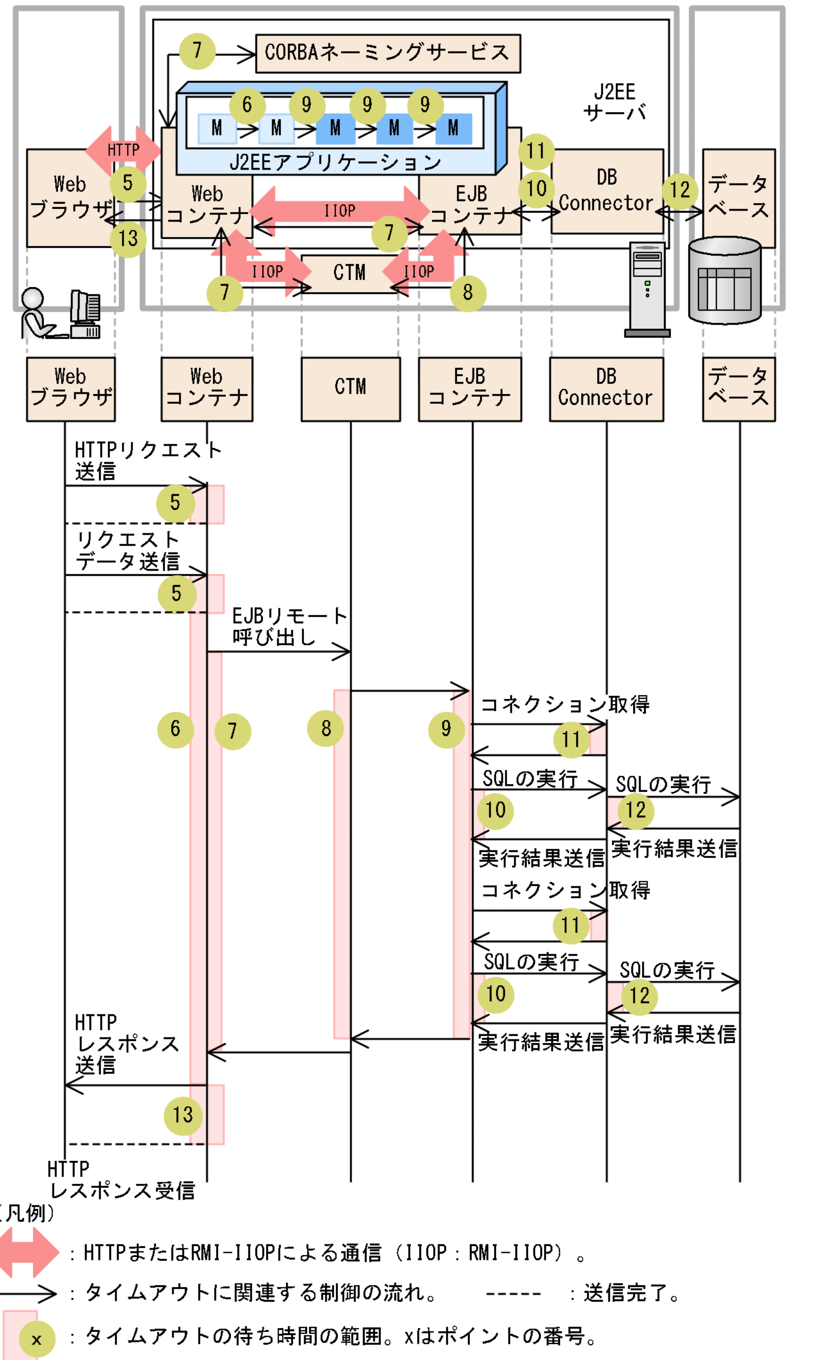
図 8-9　タイムアウトが設定できるポイント（Web サーバ連携の場合）



なお，クライアントが EJB クライアントの場合は，Web コンテナを EJB クライアントに置き換えてください。EJB クライアントからデータベースまでの範囲のタイムアウトが設定できます。

また，Web サーバを経由しないで J2EE サーバに直接リクエストを送受信する場合は，Web サーバとリバースプロキシは該当しません。このため，タイムアウトを設定するポイントとして，ポイント 1～4 は該当しません。J2EE サーバに直接リクエストを送受信する場合にタイムアウトが設定できるポイントについて，次の図に示します。

図 8-10　タイムアウトが設定できるポイント（J2EE サーバに直接リクエストを送受信する場合）



それぞれのポイントに設定するタイムアウトは，次の表に示すような用途で使い分けられます。

表 8-25　各ポイントに設定するタイムアウトの目的とデフォルトのタイムアウト設定

ポイント	タイムアウトの種類	主な用途
1	Web サーバ側で設定するクライアントからのリクエスト受信およびクライアントへのデータ送信のタイムアウト	通信路の障害，または Web サーバの障害の検知

ポイント	タイムアウトの種類	主な用途
2	リバースプロキシ側で設定する Web コンテナへのリクエスト送信処理のうち，コネクション確立のタイムアウト	通信路の障害または Web コンテナの障害検知
3	リバースプロキシ側で設定する Web コンテナへのリクエスト送信処理のうち，リクエストヘッダおよびリクエストボディ送信のタイムアウト	通信路の障害または Web コンテナの障害検知
4	リバースプロキシ側で設定する Web コンテナからのデータ受信のタイムアウト	J2EE サーバの業務処理の障害（無限ループ，デッドロックなど），または通信路の障害の検知
5	Web コンテナ側で設定するリバースプロキシまたは Web クライアントからのデータ受信のタイムアウト	通信路の障害，Web サーバの障害検知，または不正なクライアントからのアクセスの検知
6	Web アプリケーションで設定するメソッドの実行時間のタイムアウト	J2EE サーバの業務処理の障害（無限ループ，デッドロックなど）
7	EJB クライアント側で設定する Enterprise Bean のリモート呼び出し（RMI-IIOP 通信）と JNDI ネーミングサービス呼び出しのタイムアウト	J2EE サーバの業務処理の障害（無限ループ，デッドロックなど），または通信路の障害の検知
8※	EJB クライアント側で設定する CTM からの Enterprise Bean 呼び出しのタイムアウト	J2EE サーバの業務処理の障害（無限ループ，デッドロックなど），または通信路の障害の検知
9	EJB で設定するメソッドの実行時間のタイムアウト	J2EE サーバの業務処理の障害（無限ループ，デッドロックなど）
10	EJB コンテナ側で設定するデータベースのトランザクションタイムアウト	データベースサーバの障害（サーバダウンまたはデッドロックなど）の検知，またはリソースの長時間占有防止
11	DB Connector で設定するコネクション取得時のタイムアウト	コネクション取得時の障害検知（通信路の障害またはリソース枯渇）
12	データベースのタイムアウト	データベースサーバの障害（サーバダウンまたはデッドロックなど）の検知，またはリソースの長時間占有防止
13	Web コンテナ側で設定するリバースプロキシまたは Web クライアントへのレスポンス送信のタイムアウト	通信路の障害または Web サーバの障害検知

注※　CTM を使用している場合にだけ存在するポイントです。CTM を利用しない構成の場合，ポイント 7 の範囲は Web コンテナから EJB コンテナに EJB リモート呼び出しを実行してから，EJB コンテナから Web コンテナに実行結果が送信されるまでの間になります。

これらのタイムアウトの基本的な設定指針は次のとおりです。

- タイムアウト値の設定は，呼び出し元（Web クライアントまたは EJB クライアント）に近いほど大きな値を設定するのが原則です。このため，次の関係で設定することを推奨します。
  - ポイント 1<ポイント 5
  - ポイント 4>ポイント 6>ポイント 7
  - ポイント 7=ポイント 8>ポイント 9>ポイント 10

- ポイント 10>ポイント 11
- ポイント 9>ポイント 12
- ポイント 1<ポイント 13
- 4，7，10，12 のポイントのタイムアウト値を設定する場合は，呼び出し処理に通常どの程度の時間が掛かっているかを見極めた上で，呼び出す処理（業務）ごとに算出して設定してください。

なお，1～13 のポイントは，システムでの位置づけによって，次の三つに分けられます。

- Web フロントシステムで意識する必要があるポイント（1～6，および 13）
  詳細は，「8.6.2　Web フロントシステムでのタイムアウトを設定する」を参照してください。
- バックシステムで意識する必要があるポイント（7～9）
  詳細は，「8.6.3　バックシステムでのタイムアウトを設定する」を参照してください。
- データベース接続時に意識する必要があるポイント（10～12）
  このポイントは，さらにトランザクションでのタイムアウト，DB Connector でのタイムアウト，およびデータベースでのタイムアウトに分けて意識する必要があります。
  詳細は，「8.6.4　トランザクションタイムアウトを設定する」，「8.6.6　データベースでのタイムアウトを設定する」を参照してください。

それぞれのポイントでの設定については，「8.6.8　タイムアウトを設定するチューニングパラメタ」を参照してください。

参考

それぞれのポイントのデフォルト値は次のとおりです。

ポイント	デフォルト値
1	60 秒
2	60 秒
3	60 秒
4	60 秒
5	300 秒
6	設定されていません。タイムアウトしません。
7	設定されていません。レスポンスを待ち続けます。
8	ポイント 7 と同じ値が Enterprise Bean 呼び出し時に自動的に引き継がれて設定されます。
9	設定されていません。タイムアウトしません。
10	180 秒
11	タイムアウトの設定個所ごとに異なります。

ポイント	デフォルト値
	• 物理コネクション確立時のタイムアウト：8 秒 • コネクション枯渇時のコネクション取得要求のタイムアウト：30 秒 • コネクション障害検知時のタイムアウト：5 秒
12	データベースの種類とタイムアウトの設定個所ごとに異なります。 HiRDB の場合 ロック解放待ちタイムアウト：180 秒 レスポンスタイムアウト：0 秒（HiRDB クライアントは HiRDB サーバからの応答があるまで待ち続けます） リクエスト間隔タイムアウト：600 秒 Oracle の場合（グローバルトランザクションを使用するとき） ロック解放待ちタイムアウト：60 秒 SQL Server の場合 メモリ取得待ちタイムアウト：-1（-1 を指定した場合の動作は，SQL Server のドキュメントを参照してください） ロック解放待ちタイムアウト：-1（ロックが解放されるまで待ち続けます） XDM/RD E2 の場合 ロック解放待ちタイムアウト：なし（タイムアウト時間を監視しません） SQL 実行 CPU 時間タイムアウト：10 秒 SQL 実行経過時間タイムアウト：0 秒（タイムアウト時間を監視しません） トランザクション経過時間タイムアウト：600 秒 レスポンスタイムアウト：0 秒（HiRDB クライアントは XDM/RD E2 サーバからの応答があるまで待ち続けます）
13	300 秒

8.6.2　Web フロントシステムでのタイムアウトを設定する

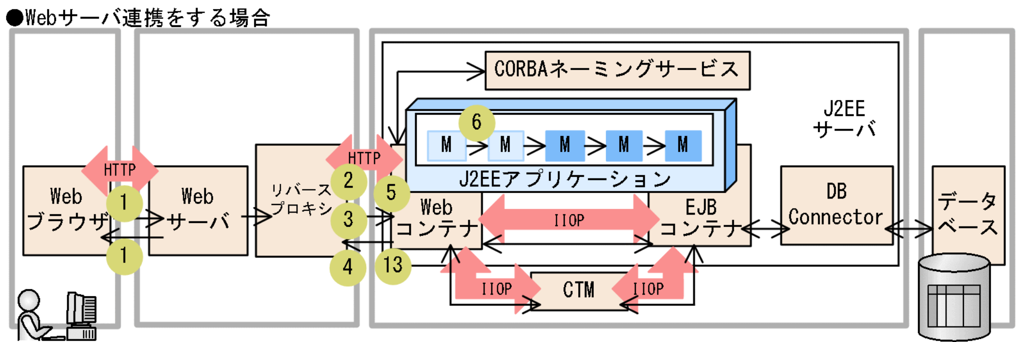
ここでは，Web フロントシステムでのタイムアウトの設定について説明します。

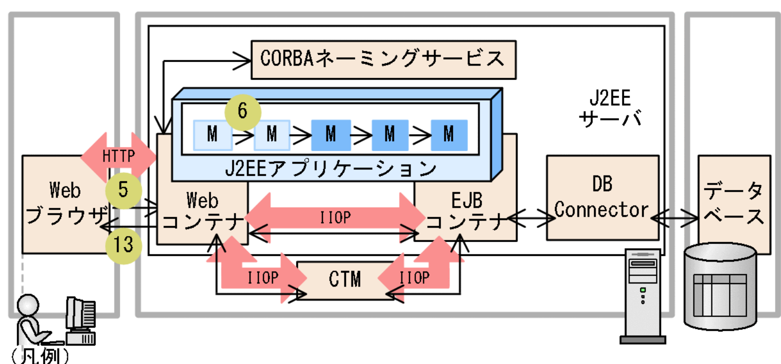
Web フロントシステムのタイムアウトを設定する場合は，システム全体のタイムアウトのうち，次の図に示す，1～6 および 13 のポイントについて意識する必要があります。この番号は，図 8-9 または図 8-10 と対応しています。

ポイント

Web サーバを経由しないで J2EE サーバに直接リクエストを送受信する場合，設定できるのは 5～13 のポイントです。1～4 のポイントは該当しません。

図 8-11　Web フロントシステムの場合に意識するタイムアウトのポイント（1〜6，および 13 のポイント）





（凡例）

：Webフロントシステムで意識するタイムアウトを設定する対象。

：HTTPまたはRMI-IIOPによる通信（IIOP：RMI-IIOP）。

：タイムアウトに関連する制御の流れ。

M ：サーブレットまたはJSP内のメソッド。

- Web サーバでのクライアントからのリクエスト受信，およびクライアントへのデータ送信待ち時間（1 のポイント）

  Web ブラウザからの要求が滞った場合に，タイムアウトによって Web サーバのリソースを解放します。また，Web ブラウザへの応答が滞った場合（Web ブラウザが受信しない場合）に，タイムアウトによって Web サーバおよび J2EE サーバ内の Web コンテナのリソースを解放します。

  これらの待ち時間には，同じ値が設定されます。Web サーバと連携する場合にだけ設定できるポイントです。

- Web サーバに登録したリバースプロキシでの Web コンテナへのリクエスト送信待ち時間（2 および 3 のポイント）

  リバースプロキシから Web コンテナへのリクエスト送信時に，Web コンテナ自体のトラブル，または Web サーバと Web コンテナ間の通信路でのトラブルによって制御が戻らなくなった場合に，タイムアウトによって Web サーバのリソースを解放します。また，同時に Web ブラウザにエラーを通知します。Web サーバと連携する場合にだけ設定できるポイントです。

  ポイント 2 は Web コンテナとのコネクション確立の待ち時間，ポイント 3 は Web コンテナへのリクエスト送信処理の待ち時間です。

- **Web サーバに登録したリバースプロキシでの Web コンテナからのデータ受信待ち時間（4 のポイント）**

  J2EE アプリケーションで何かのトラブルが発生して制御が戻らなくなった場合に，タイムアウトによって Web サーバのリソースを解放します。また，同時に Web ブラウザにエラーを通知します。Web サーバと連携する場合にだけ設定できるポイントです。

  ポイント

  設定の単位はリバースプロキシの転送先単位です。このため，業務によって処理に掛かる時間が異なる場合は，業務に対応する Web アプリケーション単位でリバースプロキシの転送先を定義してタイムアウトを設定することをお勧めします。

- **Web コンテナでのリバースプロキシまたは Web クライアントからのデータ受信待ち時間（5 のポイント）**

  Web サーバ連携の場合は，リバースプロキシから Web コンテナへのリクエスト送信時に，Web サーバ自体のトラブル，または Web サーバと Web コンテナ間の通信路でのトラブルによって要求が滞ったときに，J2EE サーバ（Web コンテナ）のリソースを解放します。

  Web サーバを経由しないで J2EE サーバに直接リクエストを送受信する場合は，Web ブラウザからの要求が滞ったときに，J2EE サーバ（Web コンテナ）のリソースを解放します。

  ポイント

  リバースプロキシを使用する場合は，タイムアウトを無効にした常設コネクションにすることをお勧めします。

- **Web コンテナ上でのリクエスト処理待ち時間（6 のポイント）**

  J2EE アプリケーションの実行時間監視機能を利用します。

  「8.6.7　J2EE アプリケーションのメソッドタイムアウトを設定する」を参照してください。

- **Web コンテナからリバースプロキシまたは Web クライアントへのレスポンス送信待ち時間（13 のポイント）**

  Web サーバ連携の場合は，Web コンテナからリバースプロキシへのレスポンス送信時に，Web サーバ自体のトラブル，または Web サーバと Web コンテナ間の通信路でのトラブルによって制御が戻らなくなった場合に，タイムアウトによって Web コンテナのリソースを解放します。

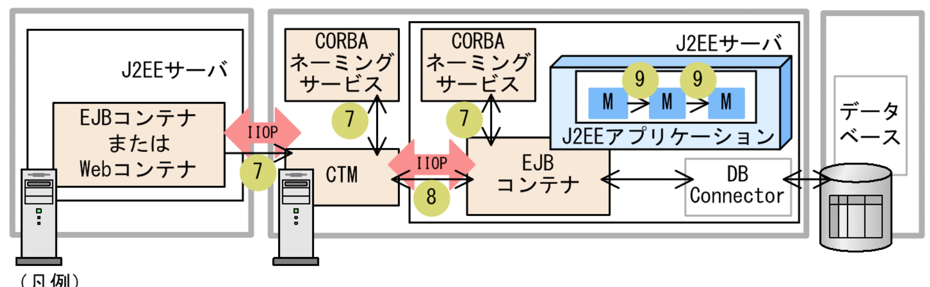
  Web サーバを経由しないで J2EE サーバに直接リクエストを送受信する場合は，Web ブラウザへの通信が滞ったときに，タイムアウトによって Web コンテナのリソースを解放します。

8.6.3　バックシステムでのタイムアウトを設定する

ここでは，バックシステムでのタイムアウトの設定について説明します。なお，バックシステムでのタイムアウトのうち，データベースなどの EIS とのトランザクションに関連するタイムアウトについては，「8.6.4　トランザクションタイムアウトを設定する」で説明します。ここでは EJB クライアントと EJB コンテナに関連するタイムアウトについて説明します。

バックシステムのタイムアウトを設定する場合は，システム全体のタイムアウトのうち，次の図に示す，7～9 のポイントについて意識する必要があります。なお，この番号は，図 8-9 または図 8-10 と対応しています。

図 8-12　バックシステムの場合に意識するタイムアウトのポイント



（凡例）

：バックシステムで意識するタイムアウトを設定する対象

：RMI-IIOPによる通信（IIOP：RMI-IIOP）。

：タイムアウトに関連する制御の流れ。

M ：Enterprise Bean内のメソッド。

- **Enterprise Bean をリモート呼び出し（RMI-IIOP 通信）する場合と CORBA ネーミングサービスを呼び出す場合のクライアント側待ち時間（7 のポイント）**
  CORBA ネーミングサービスまたは J2EE アプリケーションへのアクセスで何かのトラブルが発生して制御が戻らなくなった場合に，タイムアウトによって EJB クライアントにエラーを通知します。
- **CTM または EJB クライアントから Enterprise Bean を呼び出す場合のクライアント側待ち時間（8 のポイント）**
  J2EE アプリケーションで，無限ループやデッドロックなど何らかのトラブルが発生した場合に，CTM のリソースを解放します。また，EJB クライアントにエラーを通知します。

ポイント

EJB クライアントから Enterprise Bean を呼び出す場合，タイムアウトは usrconf.properties またはアプリケーションサーバが提供する API（com.hitachi.software.ejb.ejbclient.RequestTimeoutConfig クラスのメソッド）に指定できます。

usrconf.properties の定義は，プロセス全体に影響します。API に指定したタイムアウトは，ビジネスメソッドを呼び出すスレッドまたはオブジェクトの範囲に影響します。また，API の指定は，usrconf.properties の定義よりも優先されます。

このため，プロセス全体に設定したい標準的な値を usrconf.properties に定義して，呼び出す業務によって細かく設定したい値は適宜 API を使用して設定することをお勧めします。

Enterprise Bean の呼び出しでタイムアウトが発生した場合，EJB クライアント側には，javax.rmi.RemoteException（org.omg.CORBA.TIMEOUT など）例外が通知されます。これによって，クライアント側のリクエストはキャンセルされます。ただし，この時点でサーバ側の Enterprise

Bean の処理がすでに開始されていた場合は，そのまま処理が継続されます。このため，タイムアウトが発生した場合でも，サーバ側の処理は正常に終了することがあります。

- **EJB コンテナ上でのメソッド処理待ち時間（9 のポイント）**
  J2EE アプリケーションの実行時間監視機能を利用します。
  「8.6.7　J2EE アプリケーションのメソッドタイムアウトを設定する」を参照してください。

また，9 のポイントで，Stateless Session Bean の method-ready プールおよび Entity Bean の pool プールのインスタンスが時間内に取得できない場合も，タイムアウトが設定できます。この場合は，クライアントに java.rmi.RemoteException（リモートクライアントの場合）または javax.ejb.EJBException（ローカルクライアントの場合）が送出されます。

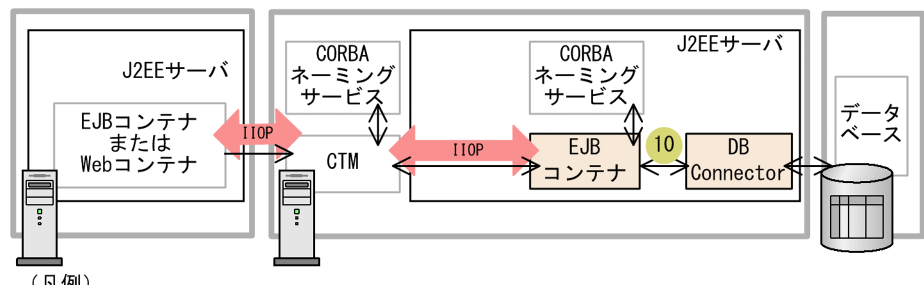
8.6.4　トランザクションタイムアウトを設定する

ここでは，トランザクションタイムアウトの設定について説明します。トランザクションタイムアウトは，データベースシステムなど EIS とのトランザクションに設定します。

DB Connector を使用してデータベースにアクセスするときのトランザクションタイムアウトについて説明します。

トランザクションタイムアウトを設定する場合は，システム全体のタイムアウトのうち，EJB コンテナとデータベースのトランザクション（図の 10 のポイント）について意識する必要があります。なお，この番号は，図 8-9 または図 8-10 と対応しています。

図 8–13　EIS とのトランザクションで意識するタイムアウトのポイント



トランザクションタイムアウトが発生すると，アプリケーションサーバによって，次の処理が実行されます。

- 実行中のトランザクションはロールバックされます。
- トランザクションに参加しているコネクションはクローズされ，コネクションプールから削除されます。

ポイント

トランザクションのタイムアウトは，CMT の場合と UserTransaction の場合で設定方法が異なります。

- **CMT の場合**
  CMT の場合，トランザクションのタイムアウトは，usrconf.properties で定義するか，または Enterprise Bean，インタフェース，もしくはメソッドの属性として設定できます。Enterprise Bean，インタフェースまたはメソッドの属性は，サーバ管理コマンドで設定します。
  usrconf.properties の定義は，プロセス全体に影響します。Enterprise Bean，インタフェースまたはメソッドの属性に設定したタイムアウトは，該当する Enterprise Bean，インタフェースまたはメソッドが使用するトランザクションの範囲だけに影響します。また，この指定は，usrconf.properties の定義よりも優先されます。
  このため，プロセス全体に設定したい標準的な値を usrconf.properties に定義して，呼び出す業務によって細かく設定したい値は Enterprise Bean，インタフェース，またはメソッドの属性として設定することをお勧めします。
- **UserTransaction の場合**
  UserTransaction の場合，トランザクションのタイムアウトは，usrconf.properties または JTA の API（javax.transaction.UserTransaction#setTransactionTimeout メソッド）に指定できます。
  usrconf.properties の定義は，プロセス全体に影響します。API に指定したタイムアウトは，API を発行したトランザクションの範囲だけに影響します。また，API の指定は，usrconf.properties の定義よりも優先されます。
  このため，プロセス全体に設定したい標準的な値を usrconf.properties に定義して，呼び出す業務によって細かく設定したい値は適宜 API を使用して設定することをお勧めします。

ポイント

トランザクションタイムアウトとコネクション取得に関連する設定値として，次のような関係が成り立つように設定してください。

トランザクションタイムアウト＞コネクション枯渇時のコネクション取得要求のタイムアウト＋障害検知タイムアウト＋接続確立の最大待ち時間×コネクション取得実行回数＋リトライ間隔×リトライ回数

これをパラメタで示すと次のようになります。

ejbserver.jta.TransactionManager.defaultTimeOut ＞ RequestQueueTimeout ＋ ejbserver.connectionpool.validation.timeout ＋ loginTimeout×（RetryCount ＋ 1）＋ RetryInterval×RetryCount

> また，トランザクションタイムアウトの設定値には，コネクション取得に掛かる時間以外にトランザクションの実行時間を加算する必要があります。

トランザクションタイムアウトが発生した場合，ユーザアプリケーションに例外は通知されません。ただし，メッセージ KDJE31002-W がログファイルと J2EE サーバのコンソールに出力されます。また，トランザクションタイムアウトが発生したあとで，ユーザアプリケーションから該当するトランザクションを使用して JTA インタフェースまたは JDBC インタフェースを使用しようとすると，例外が通知されます。

8.6.5 DB Connector でのタイムアウトを設定する

ここでは，DB Connector でのタイムアウトの設定について説明します。

DB Connector では，次の 3 種類のタイムアウトを設定できます。

- **物理コネクション確立時のタイムアウト**

  物理コネクションの確立時に発生するタイムアウトです。HiRDB，Oracle および SQL Server で設定できます。

  このタイムアウトが発生した場合に実行される動作は次のとおりです。

  - ユーザアプリケーションに例外が通知されます。

- **コネクション枯渇時のコネクション取得要求のタイムアウト**

  コネクション枯渇時のコネクション取得要求待ち時に発生するタイムアウトです。

  このタイムアウトが発生した場合に実行される動作は次のとおりです。

  - ユーザアプリケーションに例外（java.sql.SQLException）が通知されます。

- **コネクション障害検知時のタイムアウト**

  コネクション障害検知時に発生するタイムアウトです。コネクション取得要求時に障害検知された場合に発生します。

  このタイムアウトが発生した場合に実行される動作は次のとおりです。

  - タイムアウトが発生したことを示すメッセージ KDJE48602-W が出力されます。障害検知されたコネクションは破棄されて，新たに作成されたコネクションがユーザプログラムに返されます。

8.6.6 データベースでのタイムアウトを設定する

ここでは，次に示すデータベースでのタイムアウトの設定について説明します。

- HiRDB
- MySQL
- Oracle

- PostgreSQL
- SQL Server
- XDM/RD E2

なお，Oracle の場合は，グローバルトランザクションとローカルトランザクションのどちらを使用しているかによって，設定できる項目が異なります。

(1) HiRDB のタイムアウト

HiRDB では，次の 3 種類のタイムアウトを設定できます。

- ロック解放待ちタイムアウト
  デッドロックやリソースの長時間占有を防止するために設定するタイムアウトです。HiRDB サーバのシステム共通定義の pd_lck_wait_timeout パラメタに設定します。ここで設定するタイムアウト時間は，排他待ち時間を監視する最大時間です。排他待ち時間とは，排他要求が待ち状態になってから解除されるまでの時間です。
  このタイムアウトが発生した場合にアプリケーションサーバおよび HiRDB によって実行される動作は次のとおりです。
    - ユーザアプリケーションに例外（java.sql.SQLException）が通知されます。
    - タイムアウトが発生したことを示す HiRDB のメッセージ KFPA11770-I が出力されます。または，デッドロックが発生したことを示す HiRDB のメッセージ KFPA11911-E が出力されます。
    - 実行中のトランザクションはロールバックされます。
    - ユーザアプリケーションのビジネスメソッド終了後に，コネクションはクローズされ，コネクションプールから削除されます。
- レスポンスタイムアウト
  データベースシステムのサーバ側の障害を検知するためのタイムアウトです。
  HiRDB のクライアント環境変数の PDCWAITTIME に設定します。ここで設定するタイムアウト時間は，HiRDB クライアントから HiRDB サーバに要求をしてから，応答が戻ってくるまでの HiRDB クライアントの最大待ち時間です。長時間 SQL の時間を監視する場合などに指定します。
  このタイムアウトが発生した場合にアプリケーションサーバおよび HiRDB によって実行される動作は次のとおりです。
    - ユーザアプリケーションに例外（java.sql.SQLException）が通知されます。
    - タイムアウトが発生したことを示す HiRDB のメッセージ KFPA11732-E が出力されます。
    - 実行中のトランザクションはロールバックされます。
    - コネクションはクローズされて，コネクションプールから削除されます。
- リクエスト間隔タイムアウト
  データベースシステムのクライアント側の障害を検知するためのタイムアウトです。

HiRDBのクライアント環境変数のPDSWAITTIMEに設定します。ここで設定するタイムアウトは，HiRDBサーバがHiRDBクライアントからの要求に対する応答を返してから，次にHiRDBクライアントから要求が送信されるまでのHiRDBサーバの最大待ち時間です。時間監視は，トランザクションの処理中（SQL実行開始からコミットまたはロールバックまでの間）が対象になります。HiRDBクライアントからの要求がHiRDBサーバに到着した段階でリセットされます。

このタイムアウトが発生した場合にアプリケーションサーバおよびHiRDBによって実行される動作は次のとおりです。

- ユーザアプリケーションに例外（java.sql.SQLException）が通知されます。
- タイムアウトが発生したことを示すHiRDBのメッセージKFPA11723-Eが出力されます。
- 実行中のトランザクションはロールバックされます。
- コネクションはクローズされて，コネクションプールから削除されます。

- ノンブロックモードでのコネクション確立タイムアウト

  LAN障害を早く検知するためのタイムアウトです。

  HiRDBのクライアント環境変数のPDNBLOCKWAITTIMEに設定します。ここで設定するタイムアウトは，HiRDBサーバとHiRDBクライアントの間のコネクション確立を監視する時間です。

  このタイムアウトが発生した場合にアプリケーションサーバおよびHiRDBによって実行される動作は次のとおりです。

  - ユーザアプリケーションに例外（java.sql.SQLException）が通知されます。

(2)　MySQLのタイムアウト

MySQLのタイムアウトについてはMySQLのマニュアルを参照してください。

(3)　Oracleのタイムアウト（ローカルトランザクションの場合）

Oracleでローカルトランザクションを使用している場合は，次のタイムアウトを設定できます。

- クエリータイムアウト

  クエリータイムアウトは，JDBCドライバとしてOracle JDBC Thin Driverを使用する場合だけ設定できるタイムアウトです。java.sql.StatementインタフェースのsetQueryTimeoutメソッドを利用してタイムアウトを設定します。Oracle JDBC Thin Driverを使用してOracleに接続する場合の注意事項は，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(コンテナ共通機能)」の「3.6.7　Oracleと接続する場合の前提条件と注意事項」を参照してください。

なお，デッドロックが発生した場合は，OracleのメッセージORA-00060が出力されます。また，アプリケーションサーバによってユーザアプリケーションのビジネスメソッド終了後にコネクションがクローズされ，コネクションプールから削除されます。

(4)　Oracleのタイムアウト（グローバルトランザクションの場合）

Oracleでグローバルトランザクションを使用している場合は，次のタイムアウトを設定できます。

- **クエリータイムアウト**

  クエリータイムアウトについては，「8.6.6(3)　Oracle のタイムアウト（ローカルトランザクションの場合）」のクエリータイムアウトの説明を参照してください。

- **ロック解放待ちタイムアウト**

  デッドロックやリソースの長時間占有を防止するために設定するタイムアウトです。Oracle のサーバ定義の DISTRIBUTED_LOCK_TIMEOUT パラメタに設定します。このタイムアウトが発生した場合にアプリケーションサーバおよび Oracle によって実行される動作は次のとおりです。

  - ユーザアプリケーションに例外（java.sql.SQLException）が通知されます。
  - タイムアウトが発生したことを示す Oracle のメッセージ ORA-02049 が出力されます。または，デッドロックが発生したことを示す Oracle のメッセージ ORA-00060 が出力されます。
  - ユーザアプリケーションのビジネスメソッド終了後に，コネクションはクローズされ，コネクションプールから削除されます。

  なお，実行中のトランザクションはロールバックされません。

(5)　PostgreSQL のタイムアウト

PostgreSQL のタイムアウトについては PostgreSQL のマニュアルを参照してください。

(6)　SQL Server のタイムアウト

SQL Server では，次の 2 種類のタイムアウトを設定できます。

- **メモリ取得待ちタイムアウト**

  SQL 実行時のメモリ取得待ち時間を監視するために設定するタイムアウトです。SQL Server の環境設定オプションの query wait パラメタに設定します。ここで設定するタイムアウト時間は，SQL の実行に必要なメモリが得られなかったときのメモリ取得待ち時間です。

  このタイムアウトが発生した場合にアプリケーションサーバおよび SQL Server によって実行される動作は次のとおりです。

  - ユーザアプリケーションに例外（java.sql.SQLException）が通知されます。
  - タイムアウトが発生したことを示す SQL Server のメッセージ 8645 が出力されます。
  - 実行中のトランザクションはロールバックされます。
  - ユーザアプリケーションプログラムのビジネスメソッドの終了後に，コネクションはクローズされ，コネクションプールから削除されます。

- **ロック解放待ちタイムアウト**

  デッドロックやリソースの長時間占有を防止するために設定するタイムアウトです。SQL Server の SET LOCK_TIMEOUT ステートメントを実行することで設定します。ここで設定するタイムアウト時間は，ロックが解除されるまでの待ち時間です。

  このタイムアウトが発生した場合にアプリケーションサーバおよび SQL Server によって実行される動作は次のとおりです。

- ユーザアプリケーションに例外（java.sql.SQLException）が通知されます。
- タイムアウトが発生したことを示す SQL Server のメッセージ 1222 が出力されます。
- ユーザアプリケーションプログラムのビジネスメソッドの終了後に，コネクションはクローズされ，コネクションプールから削除されます。

また，SQL Server でデッドロックが発生した場合にアプリケーションサーバおよび SQL Server によって実行される動作は次のとおりです。

- ユーザアプリケーションに例外（java.sql.SQLException）が通知されます。
- デッドロックが発生したことを示す SQL Server のメッセージ 1205 が出力されます。
- 実行中のトランザクションはロールバックされます。
- ユーザアプリケーションプログラムのビジネスメソッドの終了後に，コネクションはクローズされ，コネクションプールから削除されます。

(7) XDM/RD E2 のタイムアウト

XDM/RD E2 では，次の 5 種類のタイムアウトを設定できます。

- ロック解放待ちタイムアウト

  デッドロックやリソースの長時間占有を防止するために設定するタイムアウトです。XDM/BASE のシステムオプション定義の TIMER パラメタに設定します。ここで設定するタイムアウト時間は，排他待ち時間を監視する最大時間です。排他待ち時間とは，排他要求が待ち状態になってから解除されるまでの時間です。

  このタイムアウトが発生した場合に，アプリケーションサーバおよび XDM/RD E2 によって実行される動作は次のとおりです。

  - J2EE アプリケーションに例外（java.sql.SQLException）が通知されます。
  - タイムアウトまたはデッドロックが発生したことを示す XDM/RD E2 のメッセージ JXZ1911I が出力されます。
  - 実行中のトランザクションはロールバックされます。
  - J2EE アプリケーションのビジネスメソッドの終了後に，コネクションはクローズされ，コネクションプールから削除されます。

  また，XDM/RD E2 でデッドロックが発生したときの動作は，ロック解放待ちタイムアウトが発生したときと同じになります。

- SQL 実行 CPU 時間タイムアウト

  SQL 実行時の CPU 処理時間を監視するために設定するタイムアウトです。DB コネクションサーバのコントロール空間起動制御文またはサーバ空間起動制御文の SQLCTIME パラメタに設定します。ここで設定するタイムアウト時間は，一つの SQL を実行したときの CPU 処理時間を監視する最大時間です。長時間 SQL の時間を監視する場合などに指定します。

  このタイムアウトが発生した場合にアプリケーションサーバおよび XDM/RD E2 によって実行される動作は次のとおりです。

- J2EE アプリケーションに例外（java.sql.SQLException）が通知されます。
- タイムアウトが発生したことを示すメッセージが出力されます。メッセージは，DB コネクションサーバのコントロール空間起動制御文に指定した VPARTOPTION パラメタの指定値によって変わります。指定を省略するか ERROR NORMAL を指定した場合には，HiRDB クライアントのメッセージ KFPA11723-E が出力されます。それ以外の値を指定した場合には，XDM/RD E2 のメッセージ JXZ1874I が出力されます。
- 実行中のトランザクションはロールバックされます。
- VPARTOPTION パラメタの指定を省略するか ERROR NORMAL を指定した場合には，コネクションはクローズされ，コネクションプールから削除されます。それ以外の値を指定した場合には，タイムアウト発生後の初回のデータベースアクセス時，または J2EE アプリケーションのビジネスメソッドの終了後に，コネクションはクローズされ，コネクションプールから削除されます。

- SQL 実行経過時間タイムアウト

SQL 実行時の経過時間を監視するために設定するタイムアウトです。DB コネクションサーバのコントロール空間起動制御文またはサーバ空間起動制御文の SQLETIME パラメタに設定します。ここで設定するタイムアウト時間は，一つの SQL を実行したときの経過時間を監視する最大時間です。長時間 SQL の時間を監視する場合などに指定します。

このタイムアウトが発生した場合にアプリケーションサーバおよび XDM/RD E2 によって実行される動作は次のとおりです。

- J2EE アプリケーションに例外（java.sql.SQLException）が通知されます。
- タイムアウトが発生したことを示すメッセージが出力されます。メッセージは，DB コネクションサーバのコントロール空間起動制御文に指定した VPARTOPTION パラメタの指定値によって変わります。指定を省略するか，ERROR NORMAL または ERROR SQLCTIME を指定した場合には，HiRDB クライアントのメッセージ KFPA11723-E が出力されます。それ以外の値を指定した場合には，XDM/RD E2 のメッセージ JXZ1874I が出力されます。
- 実行中のトランザクションはロールバックされます。
- VPARTOPTION パラメタの指定を省略するか，ERROR NORMAL または ERROR SQLCTIME を指定した場合には，コネクションはクローズされ，コネクションプールから削除されます。それ以外の値を指定した場合には，タイムアウト発生後の初回のデータベースアクセス時，または J2EE アプリケーションのビジネスメソッドの終了後に，コネクションはクローズされ，コネクションプールから削除されます。

- トランザクション経過時間タイムアウト

トランザクションの開始時点からの経過時間を監視するために設定するタイムアウトです。DB コネクションサーバのコントロール空間起動制御文またはサーバ空間起動制御文の SVETIME パラメタに設定します。ここで設定するタイムアウト時間は，トランザクションの経過時間を監視する最大時間です。

このタイムアウトが発生した場合にアプリケーションサーバおよび XDM/RD E2 によって実行される動作は次のとおりです。なお，タイムアウトが発生したときに SQL を実行していた場合には，その時点で実行されます。SQL を実行していなかった場合には，タイムアウト発生後の SQL 実行時に実行されます。

- J2EE アプリケーションに例外（java.sql.SQLException）が通知されます。
- タイムアウトが発生したことを示すメッセージが出力されます。メッセージは，DB コネクションサーバのコントロール空間起動制御文に指定した VPARTOPTION パラメタの指定値によって変わります。指定を省略するか，ERROR NORMAL または ERROR SQLCTIME を指定した場合には，HiRDB クライアントのメッセージ KFPA11723-E が出力されます。それ以外の値を指定した場合には，XDM/RD E2 のメッセージ JXZ1874I が出力されます。
- 実行中のトランザクションはロールバックされます。
- VPARTOPTION パラメタの指定を省略するか，ERROR NORMAL または ERROR SQLCTIME を指定した場合には，コネクションはクローズされ，コネクションプールから削除されます。それ以外の値を指定した場合には，タイムアウト発生後の初回のデータベースアクセス時，または J2EE アプリケーションのビジネスメソッドの終了後に，コネクションはクローズされ，コネクションプールから削除されます。

- **レスポンスタイムアウト**

データベースシステムのサーバ側の障害を検知するためのタイムアウトです。

HiRDB のクライアント環境変数の PDCWAITTIME に設定します。ここで設定するタイムアウト時間は，HiRDB クライアントから XDM/RD E2 サーバに要求をしてから，応答が戻ってくるまでの HiRDB クライアントの最大待ち時間です。長時間 SQL の時間を監視する場合などに指定します。

このタイムアウトが発生した場合にアプリケーションサーバおよび HiRDB によって実行される動作は次のとおりです。

- J2EE アプリケーションに例外（java.sql.SQLException）が通知されます。
- タイムアウトが発生したことを示す HiRDB クライアントのメッセージ KFPA11732-E が出力されます。
- 実行中のトランザクションはロールバックされます。
- コネクションはクローズされて，コネクションプールから削除されます。

(8) データベースへのアクセスでタイムアウトまたはデッドロックが発生した場合のユーザアプリケーションの処理

ユーザアプリケーションでデータベースのタイムアウトまたはデッドロックによる例外が発生した場合には，実行中のトランザクションをロールバックして，ビジネスメソッドの処理を中止してください。また，必要に応じてこの項で説明したタイムアウトパラメタを見直してください。

8.6.7 J2EE アプリケーションのメソッドタイムアウトを設定する

ここでは，J2EE アプリケーションのメソッドタイムアウトの設定について説明します。メソッドタイムアウトを設定することによって，ポイント 6 とポイント 9 で，Web コンテナまたは EJB コンテナ上の業務処理での無限ループが発生した場合などに，タイムアウトによって検知できるようになります。また，タイムアウトを検知したメソッドを強制的にキャンセル（**メソッドキャンセル**）することもできます。メソッ

ドキャンセル機能の詳細については，マニュアル「アプリケーションサーバ 機能解説 運用／監視／連携編」の「5.3.2　J2EE アプリケーション実行時間の監視とは」を参照してください。

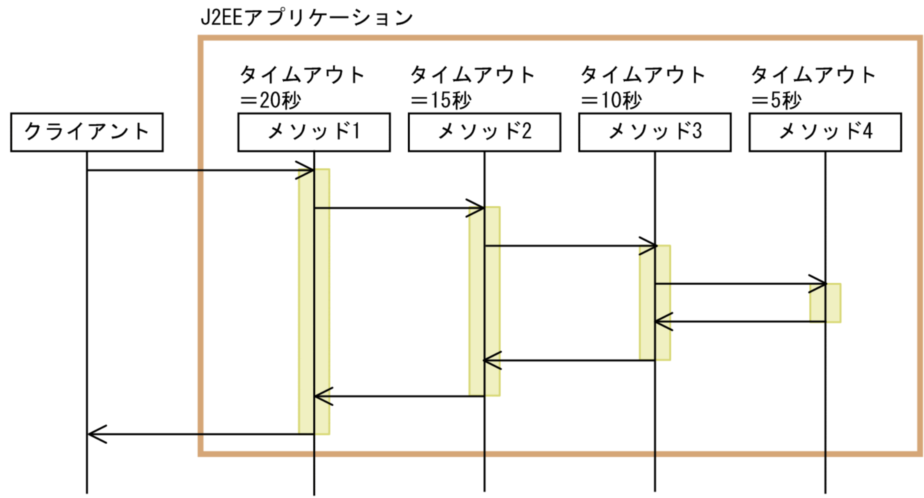
メソッドタイムアウトを設定する場合の考え方について説明します。

J2EE アプリケーション内でメソッドの呼び出しが入れ子になっている場合，タイムアウト値として，呼び出し元の方に大きな値を設定するように，メソッドの呼び出し順序を考慮して設定してください。

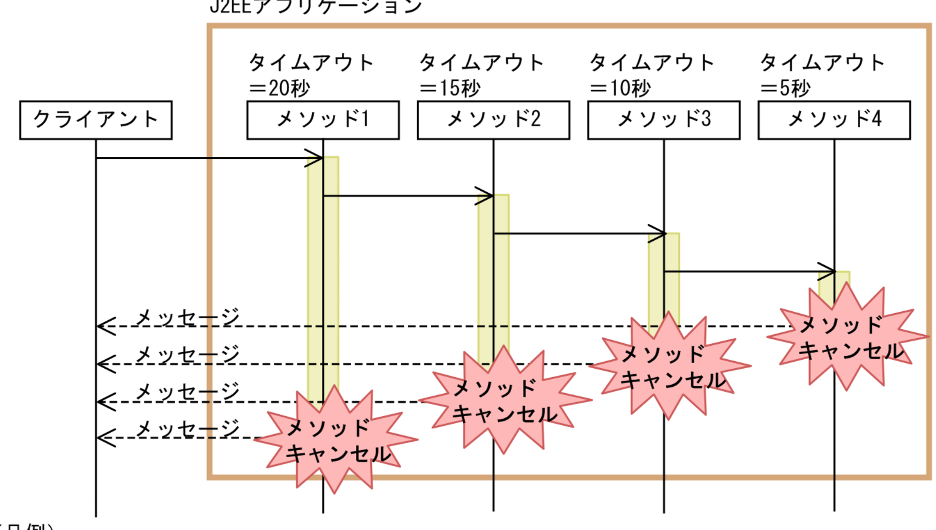
設定例を次に示します。

図 8-14　メソッドのタイムアウトの設定例

●通常の処理の流れ



●タイムアウトが発生した場合



（凡例）

→ ：タイムアウトに関連する制御の流れ。

- - -> ：メッセージの流れ。

この例では，呼び出し元に近いメソッドに大きな値を設定しています。これによって，メソッドのどこかでタイムアウトが発生した場合，クライアントから遠いメソッドから順番にタイムアウトが発生します。

タイムアウトはメッセージで通知されます。設定によっては，このタイミングでメソッドキャンセルを自動実行できます。

リモート呼び出しを実行するメソッドに対してメソッドタイムアウトおよびメソッドキャンセルを設定した場合は，呼び出し順序に注意してください。メソッドキャンセル機能では，リモート呼び出し中のメソッドは保護区で実行中と判断されます。呼び出し元に近いメソッドで先にタイムアウトが発生した場合，メソッドはリモート呼び出し中であるため，メソッドキャンセルができません。例のように呼び出し元に近い方に大きい値を設定しておけば，タイムアウトは呼び出し元から遠い順に発生するため，タイムアウトが発生したメソッドがリモート呼び出しをしていることはありません。このため，確実にメソッドキャンセルを実行できます。

ローカル呼び出しを実行するメソッドに対してメソッドタイムアウトおよびメソッドキャンセルをする設定をした場合も，呼び出し元から遠い順にキャンセルされるようにすることで，タイムアウトが発生したメソッドとキャンセルが実行されたメソッドを一致させることができます。

タイムアウト値を設定できるメソッドについては，マニュアル「アプリケーションサーバ 機能解説 運用／監視／連携編」の「5.3.4　メソッドキャンセルとは」を参照してください。

ローカル呼び出しのメソッドを入れ子で呼び出している場合の注意

簡易構築定義ファイルの<param-name>タグに指定する ejbserver.rmi.localinvocation.scope に app または all を指定している場合，一つのメソッドから入れ子で呼び出されるローカル呼び出しのメソッドは，すべて同一スレッド上で実行されます。このとき，次の点に注意してください。

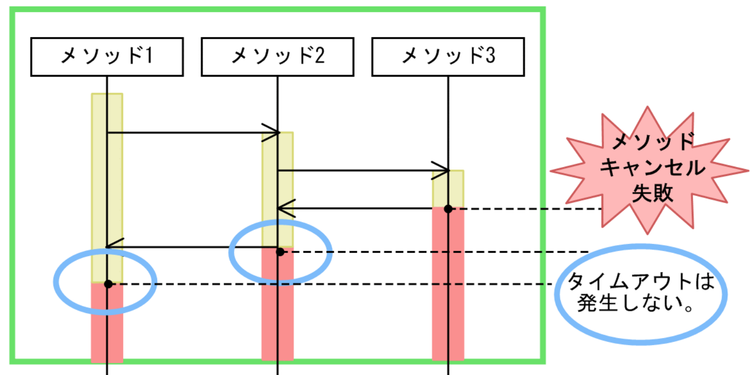
- 入れ子で呼び出されるメソッドのタイムアウトやメソッドキャンセルが失敗すると，そのメソッドが終了するまで同一スレッドのほかのメソッドではタイムアウトが発生しません。
- タイムアウトが発生したメソッドから入れ子で呼び出されているメソッドに対してメソッドキャンセルを実行した場合，コンテナによってキャンセルされるのは，メソッドキャンセルを実行した対象のメソッドだけです。タイムアウトが発生した呼び出し元のメソッドは，キャンセルの対象にはなりません。このため，キャンセルを実行したあとも，通常入れ子のメソッドの呼び出す場合と同様に，順次呼び出されたメソッドが終了していきます。なお，それらの順次呼び出されるメソッドもタイムアウト監視の対象になります。

同一スレッド上でローカル呼び出しのメソッドを入れ子で呼び出している場合の注意を次の図に示します。

図 8-15　同一スレッド上でローカル呼び出しのメソッドを入れ子で呼び出している場合の注意

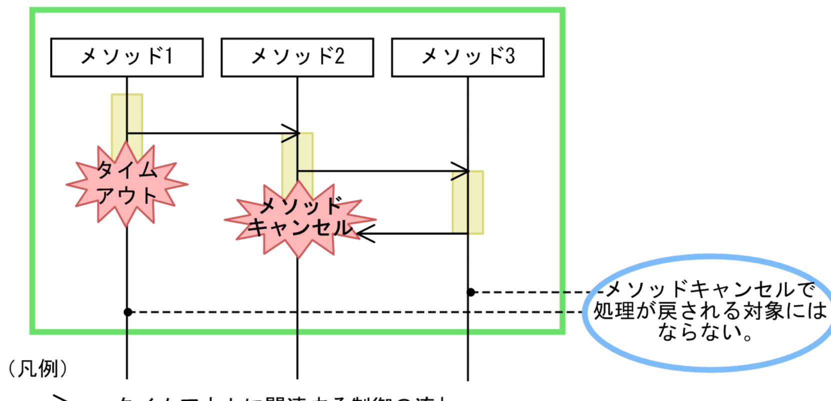
●入れ子で呼び出されるメソッドのタイムアウトやメソッドキャンセルが失敗した場合

同一スレッド上で実行されるメソッド



●タイムアウトが発生したメソッドから入れ子で呼び出されているメソッドに対して
キャンセルが実行された場合

同一スレッド上で実行されるメソッド



（凡例）

→ ：タイムアウトに関連する制御の流れ。

---> ：メッセージの流れ。

（黄色の枠）：タイムアウト監視の対象になるメソッド処理時間。

（赤色の枠）：タイムアウト時間を超えたメソッド時間。

8.6.8　タイムアウトを設定するチューニングパラメタ

ここでは，タイムアウトの設定で使用するチューニングパラメタの設定方法についてまとめて示します。

(1)　Web サーバ側で設定するクライアントからのリクエスト受信，およびクライアントへのデータ送信のタイムアウト

図 8-9 または図 8-10 のポイント 1 のタイムアウトを設定するチューニングパラメタです。Web サーバ連携の場合だけ指定できます。

表 8-26　Web サーバ側で設定するクライアントからのリクエスト受信，およびクライアントへのデータ送信のタイムアウトのチューニングパラメタ

設定項目	設定個所
クライアントからのリクエスト受信，およびクライアントへのデータ送信のタイムアウト	httpsd.conf の Timeout ディレクティブ

(2)　リバースプロキシ側で設定する Web コンテナへのデータ送信のタイムアウト

図 8-9 のポイント 2，およびポイント 3 のタイムアウトを設定するチューニングパラメタです。リバースプロキシ側で設定するタイムアウトのチューニングパラメタについて説明します。なお，これらのチューニングパラメタは，Web サーバ連携の場合だけ指定できます。

次の表に示す項目は，Smart Composer 機能で設定します。パラメタは，簡易構築定義ファイルに定義します。

表 8-27　リバースプロキシ側で設定するタイムアウトのチューニングパラメタ

ポイント	設定項目	設定対象	設定個所（パラメタ名）
2	リクエスト送信時の Web コンテナに対するコネクション確立のタイムアウト	論理 Web サーバ（web-server）	manager.web.reverseproxy.mapping の timeout キー
3	リクエスト送信のタイムアウト	論理 Web サーバ（web-server）	manager.web.reverseproxy.mapping の timeout キー

(3)　リバースプロキシ側で設定する Web コンテナからのデータ受信のタイムアウト

図 8-9 のポイント 4 のタイムアウトを設定するチューニングパラメタです。

リバースプロキシの転送先単位で設定します。リバースプロキシ側で設定するタイムアウトのチューニングパラメタについて説明します。

次の表に示す項目は，Smart Composer 機能で設定します。パラメタは，簡易構築定義ファイルに定義します。

表 8-28　リバースプロキシ側で設定するタイムアウトのチューニングパラメタ

設定項目	設定対象	設定個所（パラメタ名）
レスポンスデータ待ちの通信タイムアウト	論理 Web サーバ（web-server）	manager.web.reverseproxy.mapping の timeout キー

Web サーバ連携の場合だけ指定できます。

(4) Web コンテナ側で設定するリバースプロキシまたは Web クライアントからのデータ受信のタイムアウト

図 8-9 のポイント 5 のタイムアウトを設定するチューニングパラメタです。

J2EE サーバ単位で設定します。Web コンテナ側で設定するタイムアウトのチューニングパラメタについて説明します。

次の表に示す項目は，Smart Composer 機能で設定します。パラメタは，簡易構築定義ファイルに定義します。

表 8-29 Web コンテナ側で設定するタイムアウトのチューニングパラメタ

設定項目	設定対象	設定個所（パラメタ名）
リバースプロキシまたは Web クライアントからの応答待ちのタイムアウト	論理 J2EE サーバ（j2ee-server）	webserver.connector.nio_http.receive_timeout

(5) Web コンテナ側で設定するリバースプロキシまたは Web クライアントへのデータ受信のタイムアウト

図 8-9 のポイント 13 のタイムアウトを設定するチューニングパラメタです。

J2EE サーバ単位で設定します。Web コンテナ側で設定するタイムアウトのチューニングパラメタについて説明します。

次の表に示す項目は，Smart Composer 機能で設定します。パラメタは，簡易構築定義ファイルに定義します。

表 8-30 Web コンテナ側で設定するタイムアウトのチューニングパラメタ

設定項目	設定対象	パラメタ名
レスポンス送信処理のタイムアウト	論理 J2EE サーバ（j2ee-server）	webserver.connector.nio_http.send_timeout

(6) EJB クライアント側で設定する Enterprise Bean のリモート呼び出し（RMI-IIOP 通信）と JNDI によるネーミングサービス呼び出しのタイムアウト

図 8-9 または図 8-10 のポイント 7 のタイムアウトを設定するチューニングパラメタです。

J2EE サーバ単位，EJB クライアントアプリケーション単位または API による呼び出し単位に設定します。

EJB クライアント側で設定するタイムアウトのチューニングパラメタ（RMI-IIOP 通信によるリモート呼び出し）を次の表に示します。

表 8-31　EJB クライアント側で設定するタイムアウトのチューニングパラメタ（RMI-IIOP 通信によるリモート呼び出し）

単位	設定方法	設定項目	設定個所
J2EE サーバ単位	Smart Composer 機能	クライアントとサーバ間の通信タイムアウト	**定義ファイル** 簡易構築定義ファイル **設定対象** 論理 J2EE サーバ（j2ee-server） **パラメタ名** ejbserver.rmi.request.timeout
EJB クライアントアプリケーション単位	ファイル編集または開始時に指定するシステムプロパティの指定		**定義ファイル（ファイル編集の場合）** usrconf.properties **パラメタ名** ejbserver.rmi.request.timeout キー
API 単位	API		**オブジェクト単位で設定する場合** RequestTimeoutConfig#setRequestTimeout(java.rmi.Remote obj, int sec)メソッド※ **スレッド単位で設定する場合** RequestTimeoutConfig#setRequestTimeout(int sec)メソッド※

注※　パッケージ名は com.hitachi.software.ejb.ejbclient です。

EJB クライアント側で設定するタイムアウトのチューニングパラメタ（ネーミングサービス呼び出し）を次の表に示します。

表 8-32　EJB クライアント側で設定するタイムアウトのチューニングパラメタ（ネーミングサービス呼び出し）

単位	設定方法	設定項目	設定個所
J2EE サーバ単位	Smart Composer 機能	ネーミングサービスとの通信タイムアウト時間	**定義ファイル** 簡易構築定義ファイル **設定対象** 論理 J2EE サーバ（j2ee-server） **パラメタ名** ejbserver.jndi.request.timeout
EJB クライアントアプリケーション単位	ファイル編集または開始時に指定するシステムプロパティの指定		**定義ファイル（ファイル編集の場合）** usrconf.properties **パラメタ名** ejbserver.jndi.request.timeout キー

(7)　EJB クライアント側で設定する CTM から Enterprise Bean 呼び出しのタイムアウト

図 8-9 または図 8-10 のポイント 8 のタイムアウトを設定するチューニングパラメタです。

J2EE サーバ単位，EJB クライアントアプリケーション単位または API による呼び出し単位に設定します。

なお，このタイムアウトの設定値には，「8.6.8(6)　EJB クライアント側で設定する Enterprise Bean のリモート呼び出し（RMI-IIOP 通信）と JNDI によるネーミングサービス呼び出しのタイムアウト」で指定した設定値と同じ値が引き継がれます。

(8)　EJB コンテナ側で設定するデータベースのトランザクションタイムアウト（DB Connector を使用した場合）

図 8-9 または図 8-10 のポイント 10 のタイムアウトを設定するチューニングパラメタです。

J2EE サーバ単位，Enterprise Bean，インタフェース，メソッド単位（CMT の場合），または API による呼び出し単位（BMT の場合）に設定します。

トランザクションタイムアウトのチューニングパラメタを次の表に示します。

表 8-33　トランザクションタイムアウトのチューニングパラメタ

単位	設定方法	設定項目	設定個所
J2EE サーバ単位	Smart Composer 機能	トランザクションのトランザクションタイムアウト時間のデフォルト	**定義ファイル** 簡易構築定義ファイル **設定対象** 論理 J2EE サーバ（j2ee-server） **パラメタ名** ejbserver.jta.TransactionManager.defaultTimeOut
Enterprise Bean，インタフェース，メソッド単位（CMT の場合）	サーバ管理コマンドの cjsetappprop コマンド	トランザクションタイムアウト時間	**定義ファイル** Session Bean 属性ファイル，Entity Bean 属性ファイル，または Message-driven Bean 属性ファイル **パラメタ名** <ejb-transaction-timeout>
API 単位（BMT の場合）	API		UserTransaction#setTransactionTimeout メソッド※

注※　パッケージ名は javax.transaction です。

(9)　DB Connector のタイムアウト

図 8-9 または図 8-10 のポイント 11 のタイムアウトを設定するチューニングパラメタです。

DB Connector 単位に設定します。

DB Connector のチューニングパラメタを次の表に示します。

表 8-34　DB Connector のチューニングパラメタ

単位	設定方法	設定項目	設定個所
DB Connector 単位	サーバ管理コマンドの cjsetrarprop または cjsetresprop	物理コネクション確立時のタイムアウト	**定義ファイル** Connector 属性ファイル **設定対象** DB Connector **パラメタ名** loginTimeout
		コネクション枯渇時のコネクション取得要求のタイムアウト	**定義ファイル** Connector 属性ファイル **設定対象** DB Connector **パラメタ名** RequestQueueTimeout
J2EE サーバ単位	Smart Composer 機能	コネクション障害検知のタイムアウト	**定義ファイル** 簡易構築定義ファイル **設定対象** 論理 J2EE サーバ（j2ee-server） **パラメタ名** ejbserver.connectionpool.validation.timeout※

注※　コネクション数調節機能のタイムアウト時間と同じプロパティになります。

(10)　データベースのタイムアウト

図 8-9 または図 8-10 のポイント 12 のタイムアウトを設定するチューニングパラメタです。

データベースのタイムアウトは，使用するデータベースの種類によって異なります。なお，ここでは，DB Connector を使用して HiRDB，Oracle，SQL Server または XDM/RD E2 にアクセスする場合のタイムアウトの設定方法について説明します。

参考

Oracle を使用している場合，チューニングパラメタによってタイムアウトが設定できるのは，グローバルトランザクションを使用しているときだけです。ローカルトランザクションを使用している場合，パラメタによってタイムアウトを設定することはできません。ただし，メソッドで設定するクエリータイムアウトは，ローカルトランザクション，グローバルトランザクションのどちらでも設定できます。

(a)　HiRDB のタイムアウトの設定

HiRDB サーバのシステム共通定義または HiRDB のクライアント環境変数に設定します。詳細は，マニュアル「HiRDB システム定義」またはマニュアル「HiRDB UAP 開発ガイド」を参照してください。

HiRDB のタイムアウトを設定するチューニングパラメタを次の表に示します。

表 8-35　HiRDB のタイムアウトを設定するチューニングパラメタ

タイムアウトの種類	設定個所	設定方法（パラメタ名）	設定内容
ロック解放待ちタイムアウト	HiRDB サーバのシステム共通定義	pd_lck_wait_timeout パラメタ	設定値は任意です。
レスポンスタイムアウト	HiRDB のクライアント環境変数	PDCWAITTIME	設定値は任意です。ただし，グローバルトランザクションの場合はトランザクションタイムアウトの値よりも大きな値を指定します。
リクエスト間隔タイムアウト	HiRDB のクライアント環境変数	PDSWAITTIME	設定値は任意です。ただし，グローバルトランザクションの場合はトランザクションタイムアウトの値よりも大きな値を指定します。

参考

PDCWAITTIME や PDSWAITTIME がトランザクションタイムアウトの値よりも小さいと，トランザクションとしては制限時間内であっても，データベースの処理は制限時間オーバーになってタイムアウトが発生します。

その場合，トランザクション中であるにもかかわらずデータベースからコネクションが切断され，トランザクションマネージャがトランザクションを決着できなくなります。

また，グローバルトランザクションの場合，コネクション切断後はトランザクション決着の指示が届かないため，トランザクションのリカバリが必要になります。

(b)　Oracle のタイムアウトの設定（グローバルトランザクションを使用している場合）

Oracle のサーバ定義の DISTRIBUTED_LOCK_TIMEOUT パラメタに設定します。

なお，このほか，XAOpenString の SesTm パラメタの設定がタイムアウトに影響します。このパラメタは，チューニングできません。

(c)　SQL Server のタイムアウトの設定

SQL Server の環境設定オプションのパラメタ，またはステートメントの実行によって設定します。

SQL Server のタイムアウトを設定するチューニングパラメタを次の表に示します。

表 8-36　SQL Server のタイムアウトを設定するチューニングパラメタ

タイムアウトの種類	設定個所	設定方法（パラメタ名／ステートメント名）	設定内容
メモリ取得待ちタイムアウト	サーバー構成オプション	query wait パラメタ	設定値は任意です。
ロック解放待ちタイムアウト	−	SET LOCK_TIMEOUT ステートメント	設定値は任意です。

（凡例）

−：該当しない。

(d)　XDM/RD E2 のタイムアウトの設定

XDM/BASE のシステムオプション定義，HiRDB のクライアント環境変数，または DB コネクションサーバのコントロール空間起動制御文／サーバ空間起動制御文に設定します。

XDM/RD E2 のタイムアウトを設定するチューニングパラメタを次の表に示します。

表 8-37　XDM/RD E2 のタイムアウトを設定するチューニングパラメタ

タイムアウトの種類	設定個所	設定方法（パラメタ名）	設定内容
ロック解放待ちタイムアウト	XDM/BASE のシステムオプション定義	TIMER	設定値は任意です。※1
SQL 実行 CPU 時間タイムアウト	DB コネクションサーバのコントロール空間起動制御文／サーバ空間起動制御文	SQLCTIME	設定値は任意です。ただし，グローバルトランザクションの場合はトランザクションタイムアウトの値よりも大きな値を指定します。※2
SQL 実行経過時間タイムアウト	DB コネクションサーバのコントロール空間起動制御文／サーバ空間起動制御文	SQLETIME	設定値は任意です。ただし，グローバルトランザクションの場合はトランザクションタイムアウトの値よりも大きな値を指定します。※2
トランザクション経過時間タイムアウト	DB コネクションサーバのコントロール空間起動制御文／サーバ空間起動制御文	SVETIME	設定値は任意です。ただし，グローバルトランザクションの場合はトランザクションタイムアウトの値よりも大きな値を指定します。※2
レスポンスタイムアウト	HiRDB クライアント環境変数	PDCWAITTIME	設定値は任意です。ただし，グローバルトランザクションの場合はトランザクションタイムアウトの値よりも大きな値を指定します。※3

注※1

詳細については，マニュアル「VOS3　データマネジメントシステム XDM E2 系 システム定義（XDM/BASE・SD・TM2）」を参照してください。

注※2

詳細については，マニュアル「VOS3　Database Connection Server」を参照してください。

注※3

詳細については，マニュアル「HiRDB XDM/RD E2 接続機能」を参照してください。

(11)　J2EE アプリケーションのメソッドタイムアウト

図 8-9 または図 8-10 のポイント 6 とポイント 9 のタイムアウトを設定するチューニングパラメタです。

Web アプリケーション内または Enterprise Bean 内のメソッド単位にタイムアウトを設定する場合は，アプリケーションの属性として設定します。また，タイムアウトが発生した場合の動作についても，アプリケーションの属性として設定します。これらの項目は，サーバ管理コマンド（cjsetappprop）で設定します。

メソッドの実行時間のタイムアウトを設定するチューニングパラメタを次の表に示します。なお，ポイントごとに設定個所が異なります。

表 8-38　メソッド実行時間のタイムアウトを設定するチューニングパラメタ

対象になるポイント	タイムアウトの種類およびタイムアウト時の動作	設定個所
6	フィルタ，サーブレットまたは JSP のリクエスト処理メソッド	**定義ファイル** サーブレット属性ファイル **パラメタ名** <method-observation-timeout>
9	Enterprise Bean のリクエスト処理メソッド	**定義ファイル** Session Bean 属性ファイル，Entity Bean 属性ファイルまたは Message-driven Bean 属性ファイル **パラメタ名** <ejb-method-observation-timeout>
6 および 9	タイムアウト発生時のアプリケーション単位の動作	**定義ファイル** アプリケーション属性ファイル **パラメタ名** <method-observation-recovery-mode>

8.7　Web アプリケーションの動作を最適化する

この節では，Web アプリケーションのパフォーマンスチューニングの方法について説明します。Web フロントシステムの場合に検討してください。

ここでは，次の 3 種類のチューニング方法について説明します。

- 静的コンテンツと Web アプリケーションの配置を切り分ける
- 静的コンテンツをキャッシュする

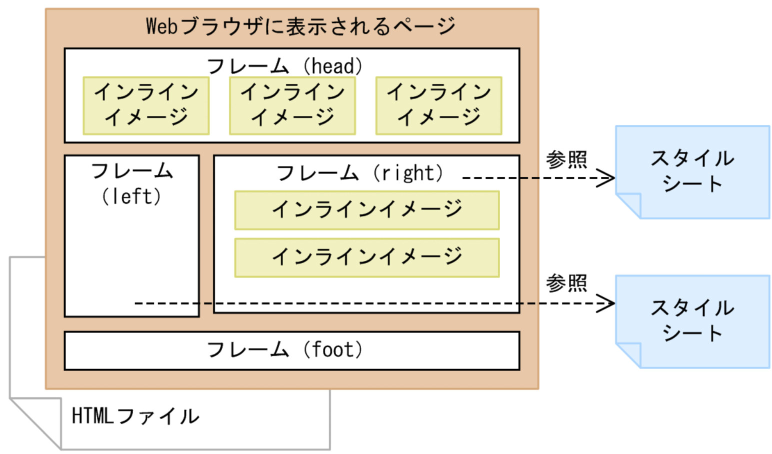
8.7.1　静的コンテンツと Web アプリケーションの配置を切り分ける

HTML ファイルや画像ファイルなど，クライアントからの要求に対する応答に使用するファイルのうち，リクエスト内容に影響されないで常に同じ内容になるコンテンツを，**静的コンテンツ**といいます。一方，サーブレットや JSP のように，クライアントからの要求に応じて動的に生成されるコンテンツを**動的コンテンツ**といいます。

ここでは，静的コンテンツを動的コンテンツである Web アプリケーションと切り分けて配置することで，パフォーマンスの向上を図る方法について説明します。

なお，それぞれの設定例では，Web ブラウザに，次の図に示すようなフレームやインラインイメージで構成される HTML ページを表示する場合の例を基に説明します。

図 8-16　静的コンテンツと動的コンテンツで構成される HTML ページの例



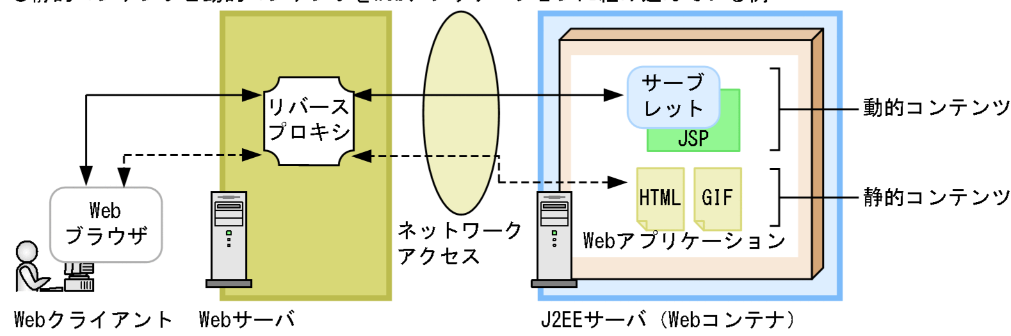
この構成の場合，次のファイルは動的に生成されない静的コンテンツです。

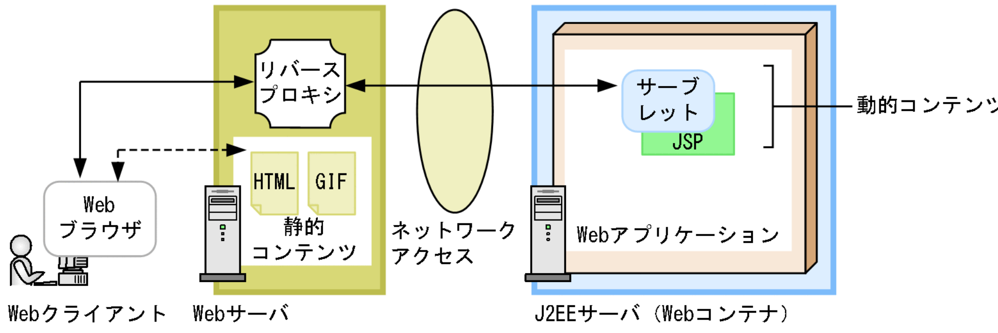
- スタイルシート（CSS ファイルなど）
- インラインイメージ（画像ファイル）
- フレームなどを定義した HTML ファイル

静的コンテンツを Web アプリケーションに組み込んで扱うと，Web コンテナで処理する必要がない静的コンテンツをやり取りする場合でも Web サーバと Web コンテナ間で常にアクセスが発生します。特に画像ファイルなどの場合は，データサイズが大きいため，処理時間が掛かります。

静的コンテンツは Web アプリケーションと分離して，Web サーバ上に配置することをお勧めします。これによって，ネットワークアクセスの回数およびやり取りするデータのサイズを減らし，パフォーマンスの向上が図れます。静的コンテンツと Web アプリケーションの処理のイメージを次の図に示します。

図 8-17　静的コンテンツと Web アプリケーションの処理のイメージ





（凡例）
◀──▶：動的コンテンツの処理の流れ
◀- - -▶：静的コンテンツの処理の流れ

Web アプリケーションと分離した静的コンテンツの配置方法について説明します。なお，ここでは，図 8-16 で示した構成の HTML ページを例として説明します。

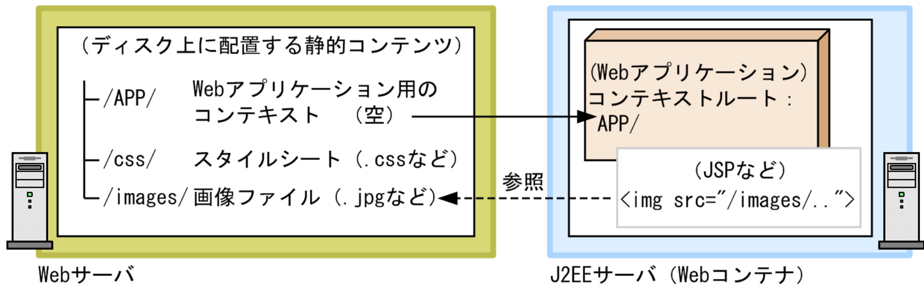
図 8-16 の場合，次のような静的コンテンツを Web サーバに配置することで，パフォーマンスの向上が図れます。

Web サーバに配置するコンテンツ

- スタイルシート（CSS ファイルなど）
- インラインイメージ（画像ファイル）
- フレームなどを定義した HTML ファイル

配置の例を次の図に示します。

図 8-18　静的コンテンツを Web サーバに配置する例（Web サーバ連携の場合）



/APP/がWebアプリケーションのコンテキストルートにマッピングされています。

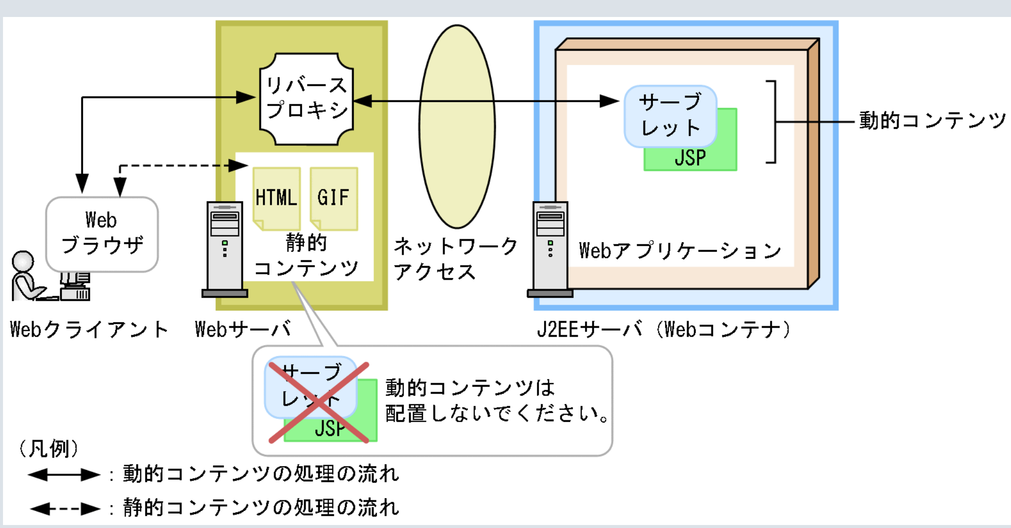
このマッピングをするためには，マッピング定義を次のように記述します。詳細は，マニュアル「HTTP Server」の「6.2.6(17)　ProxyPass」ディレクティブを参照してください。

リバースプロキシの定義

```
ProxyPass /APP/    http://myj2eeserver/APP/
```

注意事項

Web サーバ連携の場合，Web サーバには，Web アプリケーションのファイルをすべて配置するのではなく，クライアントから直接アクセスされる静的コンテンツだけを配置するようにしてください。



リバースプロキシでは，リクエストの拡張子やパス情報を正しく読み取ることができない場合に，動的コンテンツに対するリクエストを誤って静的コンテンツに対するリクエストと判断することがあります。この場合，リバースプロキシは J2EE サーバに処理を振り分けないで，Web サーバ上のコンテンツをそのままクライアントに送信してしまいます。Web サーバ上にサーブレットや JSP などの動的コンテンツを配置している場合に，動的コンテンツに対するリクエストが静的コンテンツへのリクエストとして扱われてしまうと，クラスファイルの実体や JSP などのコンテンツのソースがレスポンスとしてリクエスト元のクライアントに送信されてしまうおそれがあります。

8.7.2　静的コンテンツをキャッシュする

Web コンテナでは，静的コンテンツをメモリに保持（キャッシュ）できます。一度アクセスした静的コンテンツの内容をメモリに保持しておくことによって，2 回目以降のアクセス時のファイルシステムへのアクセス回数を減らし，応答速度の向上を図れます。

ここでは，静的コンテンツをキャッシュする場合に必要なチューニングの方法について説明します。

チューニングできるのは，次の項目です。

- 静的コンテンツをキャッシュするかどうかの選択
- 静的コンテンツのキャッシュに使用するメモリサイズの上限値
- キャッシュする静的コンテンツのファイルサイズの上限値

それぞれについて説明します。なお，これらの項目は，Web コンテナ単位および Web アプリケーション単位で設定できます。Web アプリケーション単位の設定は，Web コンテナ単位の設定よりも優先されます。このため，J2EE サーバ全体としてデフォルトの値を指定したい場合は Web コンテナ単位の値に設定して，細かな設定をしたい場合は適宜 Web アプリケーション単位に設定してください。

ポイント

メモリサイズの見積もりとの関係

静的コンテンツのキャッシュでは，メモリを使用して応答速度の向上を図ります。このため，この機能を使用する場合は，サーバで使用できるメモリ量に応じてチューニングする必要があります。

静的コンテンツのキャッシュで使用するメモリサイズは，Web アプリケーション単位に設定します。Web アプリケーション単位に設定したメモリサイズの合計値が，J2EE サーバ全体で静的コンテンツのキャッシュのために使用するメモリサイズの最大値になります。このため，この機能を使用する場合は，J2EE サーバで必要なメモリサイズを見積もる際に，Web アプリケーション単位に設定したメモリサイズの合計値を加算してください。

(1)　静的コンテンツをキャッシュするかどうかの選択

静的コンテンツのキャッシュは，デフォルトの設定では使用されない機能です。このため，使用する場合はパラメタを変更する必要があります。

なお，静的コンテンツのキャッシュの使用は，Web コンテナ単位および Web アプリケーション単位で設定でき，Web アプリケーション単位の設定は，Web コンテナ単位の設定よりも優先されます。ただし，Web コンテナ単位の設定で強制的な無効を選択している場合は，Web アプリケーション単位の設定も無効になります。強制的な無効は，次のような場合に使用できます。

- 静的コンテンツのキャッシュを有効にしたときと無効にしたときのメモリ使用量の差を検証したい場合

- Web アプリケーション単位の設定を保持した状態で静的コンテンツのキャッシュを一時的に無効にしたい場合

(2) 静的コンテンツのキャッシュに使用するメモリサイズの上限値

静的コンテンツをキャッシュするために，Web アプリケーション単位で使用するメモリサイズを設定できます。なお，それぞれの Web アプリケーションで設定した値よりもキャッシュの合計サイズが大きくなる場合は，アクセスされていない時間が最も長いキャッシュから削除されます。削除は，キャッシュの合計サイズが設定値以下になるまで繰り返されます。

メモリサイズの設定の指針を次に示します。

メモリサイズの設定の指針

- Web アプリケーション内に含まれる静的コンテンツのサイズの合計値以下の値を設定します。
- Web アプリケーションの内容によって最適な値は異なります。このため，値を設定してから静的コンテンツに対するリクエストの応答速度を測定して，高い効果が出る最適なキャッシュのサイズを見つける必要があります。

(3) キャッシュする静的コンテンツのファイルサイズの上限値

静的コンテンツのキャッシュの対象とするコンテンツのファイルサイズの上限を設定できます。上限を設定した場合，上限を超えるファイルサイズのコンテンツについてはキャッシュされないで，使用するたびに毎回ファイルシステムから取得されます。

ファイルサイズの設定の指針を次に示します。

ファイルサイズの設定の指針

- Web アプリケーション内に含まれる静的コンテンツ中の，ファイルサイズが最大であるコンテンツのファイルサイズ以下の値を設定します。
- 大きなサイズの静的コンテンツがキャッシュされることによって，アクセス頻度の高いほかの静的コンテンツのキャッシュが破棄されないように留意して，値を設定します。

(4) 静的コンテンツの稼働状況の確認

静的コンテンツの稼働状況は，Web アプリケーションを停止したときに出力されるメッセージ KDJE39234-I の出力内容で確認できます。キャッシュされている静的コンテンツの合計サイズやコンテンツの個数などが出力されるので，必要に応じて各パラメタをチューニングしてください。

8.7.3 Web アプリケーションの動作を最適化するためのチューニングパラメタ

ここでは，Web アプリケーションの動作を最適化するために使用するチューニングパラメタの設定方法についてまとめて示します。

(1) 静的コンテンツと Web アプリケーションの配置を切り分けるためのチューニングパラメタ

静的コンテンツと Web アプリケーションの配置の切り分けは，Web サーバの動作を定義するファイルのパラメタとして指定します。設定個所，ファイルおよびパラメタは，使用する Web サーバの種類によって異なります。

設定方法および設定個所を次に示します。

表 8-39 静的コンテンツと Web アプリケーションの配置を切り分けるためのチューニングパラメタ

設定方法	設定個所
Smart Composer 機能	**定義ファイル** 簡易構築定義ファイル **設定対象** 論理 Web サーバ（web-server） **パラメタ名** manager.web.reverseproxy.mapping

(2) 静的コンテンツをキャッシュするためのチューニングパラメタ

静的コンテンツをキャッシュするためのチューニングパラメタについて説明します。これらのチューニングパラメタは，Web コンテナ単位または Web アプリケーション単位に設定します。

Web コンテナ単位に設定するチューニングパラメタの設定方法について，次の表に示します。これらの項目は，Smart Composer 機能で設定します。

表 8-40 静的コンテンツをキャッシュするためのチューニングパラメタ（Web コンテナ単位で設定する項目）

設定項目	設定個所
静的コンテンツのキャッシュを使用するかどうかの選択	**定義ファイル** 簡易構築定義ファイル **設定対象** 論理 J2EE サーバ（j2ee-server）

設定項目	設定個所
	パラメタ名 webserver.static_content.cache.enabled
Web アプリケーション単位のメモリサイズの上限値の設定	**定義ファイル** 簡易構築定義ファイル **設定対象** 論理 J2EE サーバ（j2ee-server） **パラメタ名** webserver.static_content.cache.size
キャッシュする静的コンテンツのファイルサイズの上限値の設定	**定義ファイル** 簡易構築定義ファイル **設定対象** 論理 J2EE サーバ（j2ee-server） **パラメタ名** webserver.static_content.cache.filesize.threshold

Web アプリケーション単位に設定するチューニングパラメタについて示します。Web アプリケーション単位に設定する項目は，web.xml を直接編集するか，サーバ管理コマンドを使用して設定します。デプロイ前の Web アプリケーションに設定する場合は，web.xml を編集してください。デプロイ後の Web アプリケーションに設定する場合は，サーバ管理コマンド（cjsetappprop）を使用してください。

設定内容を次の表に示します。

表 8-41　静的コンテンツをキャッシュするためのチューニングパラメタ（Web アプリケーション単位で設定する項目）

設定項目	設定内容※
静的コンテンツのキャッシュを使用するかどうかの選択	<param-name>タグ com.hitachi.software.web.static_content.cache.enabled <param-value>タグ （設定値）
Web アプリケーション単位のメモリサイズの上限値の設定	<param-name>タグ com.hitachi.software.web.static_content.cache.size <param-value>タグ （設定値）
キャッシュする静的コンテンツのファイルサイズの上限値の設定	<param-name>タグ com.hitachi.software.web.static_content.cache.filesize.threshold <param-value>タグ （設定値）

注

（設定値）に設定できる値の詳細については，マニュアル「アプリケーションサーバ 機能解説 基本・開発編(Web コンテナ)」の「2.19.2　DD での定義（Web アプリケーション単位での設定)」を参照してください。

注※

web.xml を直接編集する場合，<web-app>タグ内に<context-param>タグを追加して，<context-param>タグ内に<param-name>タグおよび<param-value>タグを追加します。

サーバ管理コマンドを使用する場合，WAR 属性ファイルの<hitachi-war-property>タグ内に<context-param>タグを追加して，<context-param>タグ内に<param-name>タグおよび<param-value>タグを追加します。

8.8 CTM の動作を最適化する

CTM を使用したシステムのパフォーマンスチューニングの方法について説明します。CTM を使用したバックシステムの場合に検討してください。

ここでは，次の 4 種類のチューニング方法について説明します。

- CTM ドメインマネジャおよび CTM デーモンの稼働状態の監視間隔をチューニングする
- 負荷状況監視間隔をチューニングする
- CTM デーモンのタイムアウト閉塞を設定する
- CTM で振り分けるリクエストの優先順位を設定する

8.8.1 CTM ドメインマネジャおよび CTM デーモンの稼働状態の監視間隔をチューニングする

システムに存在する複数の CTM ドメインマネジャ間，および CTM ドメイン内の複数の CTM デーモン間では，お互いの稼働状態を監視するために，定期的に通信処理が実行されています。

ここでは，それぞれの通信処理間隔をチューニングする場合の考え方について説明します。

(1) CTM ドメインマネジャ間で稼働状態を監視する間隔のチューニング

CTM ドメインマネジャ間では，お互いのホストにある CTM デーモンの情報を定期的に交換しています。この情報を基に，自ホストで受け付けたリクエストを適宜ほかのホストの CTM デーモンに振り分けています。

情報を交換するタイミングでは，お互いが稼働状態であるかどうかも同時に確認します。相手のホストの CTM ドメインマネジャが停止している場合は，そのホストにはリクエストを振り分けないようにします。CTM ドメインマネジャでは，情報を交換する間隔に一定の係数を掛けた時間以上応答がない場合に，相手の CTM ドメインマネジャが停止していると判断します。

情報を交換する間隔の設定個所は，相手の CTM ドメインマネジャが同じネットワークセグメント内にあるか，異なるネットワークセグメントにあるかによって異なります。稼働状態を判断するときに使用する係数のデフォルトは 2 です。例えば，同じネットワークセグメント内にある CTM ドメインマネジャの稼働情報を確認する場合，Management Server を使用して構築したシステムでは，CTM ドメイン構成情報の送信間隔に 2 を掛けた時間以上応答がない場合は，停止していると判断します。CTM ドメイン構成情報の送信間隔が 60 秒の場合は，120 秒間応答がない場合に，停止していると判断します。

この係数を変更することによって，通信処理の最適化が図れます。

係数に指定する値は，通信によって発生する処理の負荷を考慮して，適切な値を検討してください。基準になる送信間隔も必要に応じて検討してください。係数を小さくすると，間隔が短くなり，相手の CTM

ドメインマネジャが停止したことがすぐに検知できます。これによって，CTM デーモンが停止しているホストにリクエストを送信してしまうことを防げます。ただし，間隔が短過ぎると，通信処理が多く発生し，通信負荷が掛かります。

(2)　CTM デーモン間で稼働状態を監視する間隔のチューニング

CTM デーモン間では，CTM ドメインマネジャ間でやり取りした情報を基に，相互にリクエストの振り分け処理をしています。

リクエストの振り分け先の CTM デーモンから一定時間応答がない場合，振り分け元の CTM デーモンでは，相手の CTM デーモンが停止していると判断して，以降のリクエストを振り分けないようにします。

デフォルトの設定では，CTM デーモンは 180 秒間応答を待ちます。180 秒以上応答がない場合，相手の CTM デーモンが停止していると判断します。この値を変更することで，不要な処理待ち時間を短縮できます。

値は，リクエストとして送信するデータの大きさなど考慮して，適切な値を設定してください。間隔を短くすることで，相手のホストのトラブルを迅速に検知できるため，トラブルの影響が少ない状態でシステムから切り離すことができるようになります。ただし，間隔が短過ぎると，大きなサイズのデータを転送している間にタイムアウトが発生してしまうおそれがあります。

8.8.2　負荷状況監視間隔をチューニングする

CTM ドメイン内の複数の CTM デーモン間では，それぞれのスケジュールキューの負荷情報を監視しています。監視結果は，CTM デーモン間のリクエスト振り分け時に利用されます。

負荷状況監視間隔は，チューニングできます。なお，デフォルトの状態では 10 秒です。

負荷状況監視間隔を短くすることで，その時点の状況に応じた振り分けができるようになります。ただし，短くし過ぎると，通信が多発して，負荷が高くなります。

なお，負荷状況監視間隔に 0 を設定した場合は，J2EE サーバ起動時の負荷状況を一度だけ収集して，その値を使用し続けます。

8.8.3　CTM デーモンのタイムアウト閉塞を設定する

CTM デーモンに対応する J2EE サーバにトラブルが発生した場合，CTM デーモンが送信したリクエストでタイムアウトが発生します。そのままの状態で運用を続けると，CTM デーモンはトラブルが発生した J2EE サーバに対してリクエストを送信し続けるため，そのつどリクエストでタイムアウトが発生してしまいます。

これに対処するために，CTM デーモンには，タイムアウト閉塞を設定できます。タイムアウト閉塞とは，一定時間内に規定回数以上のリクエストのタイムアウトが発生した場合に，CTM デーモンのスケジュー

ルキューを閉塞する機能です。これによって，トラブルが発生した CTM デーモンでそれ以上リクエストを受け付けなくなり，ほかの CTM デーモンが受け付けるようになります。リクエストは正常に稼働している J2EE サーバに振り分けられるようになります。

8.8.4 CTM で振り分けるリクエストの優先順位を設定する

CTM で制御するリクエストには，優先順位が付けられます。すぐに実行する必要があるリクエストに高い優先順位を設定しておくことで，スケジュールキューの中で滞留することなく，迅速な処理ができるようになります。

リクエストの優先順位は，CTM にリクエストを送信する，EJB クライアントアプリケーションまたは J2EE サーバに対して設定できます。優先順位を高く設定した EJB クライアントアプリケーションまたは J2EE サーバから送信されたリクエストは，スケジュールキューに格納されているほかのクライアントからのリクエストよりも優先されて処理されます。

8.8.5 CTM の動作を最適化するチューニングパラメタ

ここでは，CTM の動作の最適化で使用するチューニングパラメタの設定方法についてまとめて示します。

(1) CTM ドメインマネジャおよび CTM デーモンの稼働状態の監視間隔を設定するチューニングパラメタ

CTM ドメインマネジャの稼働状態の監視間隔をチューニングするパラメタについて説明します。

次の表に示す項目は，Smart Composer 機能で設定します。パラメタは，簡易構築定義ファイルに定義します。

なお，監視間隔は，送信間隔×係数の値になります。

表 8-42 CTM ドメインマネジャの稼働状態の監視間隔をチューニングするパラメタ

対象	設定項目	設定対象	設定個所（パラメタ名）
同じネットワークセグメント内にある CTM ドメインマネジャ	送信間隔	論理 CTM ドメインマネジャ（ctm-domain-manager）	cdm.SendInterval
	係数	論理 CTM ドメインマネジャ（ctm-domain-manager）	cdm.AliveCheckCount
異なるネットワークセグメントにある CTM ドメインマネジャ	送信間隔	論理 CTM ドメインマネジャ（ctm-domain-manager）	cdm.SendHostInterval
	係数	論理 CTM ドメインマネジャ（ctm-domain-manager）	cdm.AliveCheckCount

CTM デーモンの稼働状態の監視間隔をチューニングするパラメタについて説明します。

次の表に示す項目は，Smart Composer 機能で設定します。パラメタは，簡易構築定義ファイルに定義します。

表 8-43　CTM デーモンの稼働状態の監視間隔をチューニングするパラメタ

設定項目	設定対象	設定個所（パラメタ名）
CTM デーモン間転送時のタイムアウト	論理 CTM	ctm.DCSendTimeOut

(2)　負荷状況監視間隔を設定するチューニングパラメタ

負荷状況監視間隔をチューニングするパラメタについて説明します。

次の表に示す項目は，Smart Composer 機能で設定します。パラメタは，簡易構築定義ファイルに定義します。

表 8-44　負荷情報監視間隔をチューニングするパラメタ

設定項目	設定対象	設定個所（パラメタ名）
CTM デーモン間転送時のタイムアウト	論理 CTM	ctm.LoadCheckInterval

(3)　CTM デーモンのタイムアウト閉塞を設定するチューニングパラメタ

タイムアウト閉塞は，タイムアウト発生回数と監視間隔を設定しておくことによって，実行されます。

CTM デーモンのタイムアウト閉塞をチューニングするパラメタについて説明します。

次の表に示す項目は，Smart Composer 機能で設定します。パラメタは，簡易構築定義ファイルに定義します。

表 8-45　CTM デーモンのタイムアウト閉塞をチューニングするパラメタ

設定項目	設定対象	設定個所（パラメタ名）
タイムアウト発生回数	論理 CTM	ctm.RequestCount
監視時間間隔	論理 CTM	ctm.RequestInterval

(4)　CTM で振り分けるリクエストの優先順位を設定するチューニングパラメタ

CTM で振り分けるリクエストの優先順位の設定は，EJB クライアントアプリケーションの場合と，J2EE サーバの場合で異なります。また，J2EE サーバの場合，システムの構築方法によって設定個所が異なります。CTM で振り分けるリクエストの優先順位を設定するチューニングパラメタを次の表に示します。

表 8-46　CTM で振り分けるリクエストの優先順位を設定するチューニングパラメタ

設定単位	設定方法	設定個所
EJB クライアントアプリケーション	ファイル編集または EJB クライアントアプリケーション開始時に指定するシステムプロパティの指定	**定義ファイル（ファイル編集の場合）** usrconf.properties **パラメタ名** ejbserver.client.ctm.RequestPriority キー
J2EE サーバ	Smart Composer 機能	**定義ファイル** 簡易構築定義ファイル **設定対象** 論理 J2EE サーバ（j2ee-server） **パラメタ名** ejbserver.client.ctm.RequestPriority

9 パフォーマンスチューニング（バッチアプリケーション実行基盤）

この章では，バッチアプリケーションを実行するシステムのパフォーマンスをチューニングする方法について説明します。

パフォーマンスチューニングによって動作環境を最適化することで，システムの性能を最大限に生かせるようになります。

J2EE アプリケーション実行基盤のパフォーマンスチューニングについて検討する場合は，「8. パフォーマンスチューニング（J2EE アプリケーション実行基盤）」を参照してください。

9.1　パフォーマンスチューニングで考慮すること

この節では，バッチアプリケーション実行基盤のパフォーマンスチューニングで考慮することについて説明します。

9.1.1　パフォーマンスチューニングの観点

バッチアプリケーション実行基盤のパフォーマンスチューニングは，次の観点で実施します。

- **データベースアクセス方法の最適化**
- **タイムアウトの設定**
- **フルガーベージコレクションを発生させるメモリ使用量のしきい値の設定**

それぞれのポイントについて説明します。

(1)　データベースアクセス方法の最適化

データベースアクセス方法の最適化は，生成に時間が掛かるコネクションやステートメントをプールしておくことで，データベースアクセス時のオーバーヘッドを削減することを目的とします。

パフォーマンスチューニングでは，次に示す機能を有効に活用することで，データベースアクセス処理を最適化し，スループットを向上させます。

- コネクションプーリング
- ステートメントプーリング（PreparedStatement および CallableStatement のプーリング）

データベースアクセス方法は，データベースとの接続に DB Connector を使用している場合に最適化できます。

(2)　タイムアウトの設定

タイムアウトの設定は，システムのトラブル発生を検知して，リクエストの応答が返らなくなることを防ぎ，適宜リソースを解放することを目的とします。

設定できるタイムアウトには，次の種類があります。

- Enterprise Bean を呼び出す時のタイムアウト
- トランザクションのタイムアウト
- データベースのタイムアウト

(3)　フルガーベージコレクションを発生させるメモリ使用量のしきい値の設定

フルガーベージコレクションの制御で使用するしきい値は，オンライン処理とバッチ処理で同じリソースを使用する場合に設定します。バッチサーバのフルガーベージコレクションによってオンライン処理を中断させないことを目的とします。

フルガーベージコレクションの実行を制御することによって，リソースを排他していないタイミングで適切にフルガーベージコレクションを実行できます。

9.1.2　チューニング手順

パフォーマンスチューニングは，システムのパフォーマンスを生かす最適な設定を見つける作業です。構築した環境で，実際に処理を実行しながら，パラメタの調整やボトルネックの調査，解消によってパフォーマンスを向上させていきます。

チューニング作業では，まず，目標値を決定します。次に，各パラメタに初期値を設定した状態でスループットを測定します。各パラメタを調整しながら，目標値に近い最適な値を見つけていきます。

チューニングの際，CPU の利用率の測定には，OS に付属している監視ツールなどが利用できます。JavaVM に関するアプリケーションサーバの稼働情報については，稼働情報収集機能などで確認できます。確認方法については，マニュアル「アプリケーションサーバ 機能解説 運用／監視／連携編」の「3. 稼働情報の監視（稼働情報収集機能）」を参照してください。

なお，CPU 利用率が 100%からかなり低い状態で飽和した場合は，システム上に入出力処理や排他処理などのボトルネックがあるおそれがあります。ボトルネックを調査し，対策してから，再度パフォーマンスチューニングを実行してください。アプリケーションサーバでは，システムのボトルネックの調査に，性能解析トレースを利用できます。性能解析トレースの機能詳細，および性能解析トレースを利用して取得したトレースファイルの利用方法については，マニュアル「アプリケーションサーバ 機能解説 保守／移行編」の「7.　性能解析トレースを使用した性能解析」を参照してください。

9.1.3　チューニング項目

バッチアプリケーション実行基盤でのチューニング項目について，次の表に示します。

表 9-1　バッチアプリケーション実行基盤でのチューニング項目

チューニング項目	利用できる機能	参照先
データベースアクセス方法の最適化	コネクションプーリング※1	8.5.1※2
	ステートメントプーリング※1	8.5.2※2
タイムアウトの設定	Enterprise Bean 呼び出しでのタイムアウトの設定	8.6.3※2，※3
	トランザクションタイムアウトの設定	9.3.2

チューニング項目	利用できる機能	参照先
	データベースでのタイムアウトの設定	8.6.6※2
フルガーベージコレクションの制御※1	しきい値の設定	9.4

注※1

DB Connector を使用している場合に利用できる機能です。

注※2

J2EE アプリケーション実行基盤の説明を参照してください。その際，説明中の「J2EE サーバ」を「バッチサーバ」に読み替えてください。また，「J2EE アプリケーション」を「バッチアプリケーション」に読み替えてください。

注※3

Enterprise Bean 呼び出しでは，J2EE アプリケーション実行基盤のバックシステムと同じ項目にタイムアウトを設定できます。

9.2 チューニングの方法

この節では，チューニングの方法について説明します。チューニングの方法は，設定対象の種類によって異なります。

9.2.1 バッチサーバのチューニング

バッチサーバのチューニングには，Smart Composer 機能の簡易構築定義ファイルを使用します。簡易構築定義ファイルでは，<configuration>タグ下の<logical-server-type>に設定対象とする論理サーバの種類（J2EE サーバ※）を指定して，<param>タグ下でパラメタ名とその値を設定します。簡易構築定義ファイルの詳細については，マニュアル「アプリケーションサーバ リファレンス 定義編(サーバ定義)」の「4.3 簡易構築定義ファイル」を参照してください。

注※

Smart Composer 機能では，バッチサーバを J2EE サーバとして扱います。

9.2.2 リソースのチューニング

リソースのチューニングをする場合は，サーバ管理コマンドを使用します。

サーバ管理コマンドを使用する場合は，Connector 属性ファイルを編集します。Connector 属性ファイルの詳細については，マニュアル「アプリケーションサーバ リファレンス 定義編(アプリケーション／リソース定義)」の「4.1 Connector 属性ファイル」を参照してください。

9.3　タイムアウトを設定する

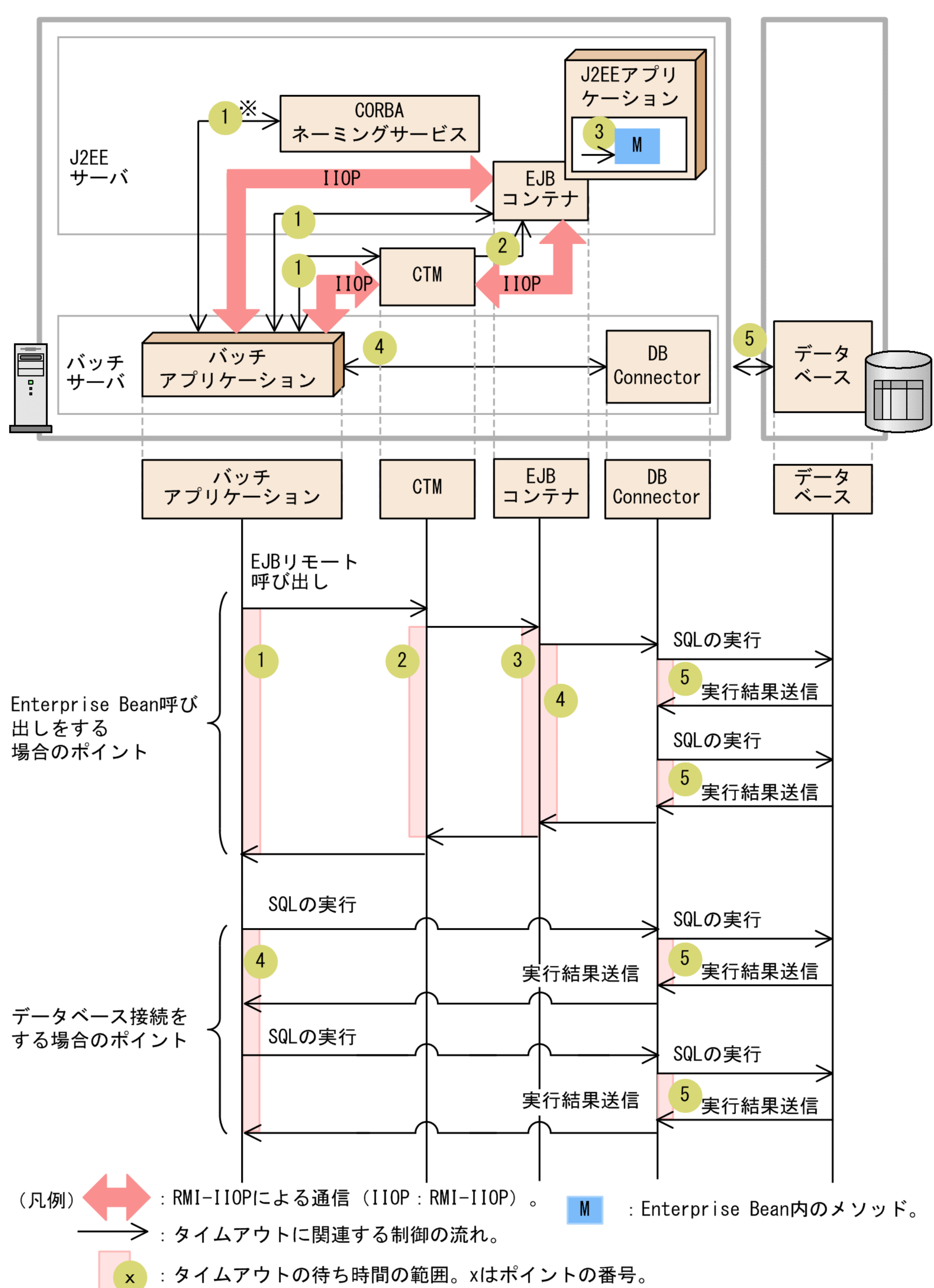
アプリケーションサーバのシステムでは，トラブル発生時にリクエストの応答が戻ってこない状態になることを防ぐために，幾つかのポイントにタイムアウトを設定できます。

この節では，システム全体でタイムアウトが設定できるポイントと，設定する場合の指針について説明します。

9.3.1　タイムアウトが設定できるポイント

バッチアプリケーションを実行するシステムでは，次の図に示すポイントにタイムアウトが設定できます。

図 9-1　タイムアウトが設定できるポイント



それぞれのポイントに設定するタイムアウトは，次の表に示すような用途で使い分けられます。

表 9-2　各ポイントに設定するタイムアウトの目的とデフォルトのタイムアウト設定

ポイント	タイムアウトの種類	主な用途
1	バッチサーバ側で設定する Enterprise Bean のリモート呼び出し（RMI-IIOP 通信）と JNDI ネーミングサービス呼び出しのタイムアウト	バッチサーバの業務処理の障害（無限ループ，デッドロックなど），または通信路の障害の検知
2※	バッチサーバ側で設定する CTM からの Enterprise Bean 呼び出しのタイムアウト	バッチサーバの業務処理の障害（無限ループ，デッドロックなど），または通信路の障害の検知
3	Enterprise Bean 呼び出しアクセスした EJB で設定するメソッドの実行時間のタイムアウト	J2EE サーバの業務処理の障害（無限ループ，デッドロックなど）
4	バッチサーバ側で設定するデータベースのトランザクションタイムアウト	データベースサーバの障害（サーバダウンまたはデッドロックなど）の検知，またはリソースの長時間占有防止
5	データベースのタイムアウト	データベースサーバの障害（サーバダウンまたはデッドロックなど）の検知，またはリソースの長時間占有防止

注※　CTM を使用している場合にだけ存在するポイントです。CTM を利用しない構成の場合，ポイント 2 の範囲はバッチサーバから EJB コンテナに EJB リモート呼び出しを実行してから，EJB コンテナからバッチサーバに実行結果が送信されるまでの間になります。

これらのタイムアウトの基本的な設定指針は次のとおりです。

- タイムアウト値の設定は，呼び出し元（バッチサーバ）に近いほど大きな値を設定するのが原則です。このため，次の関係で設定することを推奨します。
  - ポイント 1=ポイント 2>ポイント 3>ポイント 4>ポイント 5
- 1，4，5 のポイントのタイムアウト値を設定する場合は，呼び出し処理に通常どの程度の時間が掛かっているかを見極めた上で，呼び出す処理（業務）ごとに算出して設定してください。

なお，1～5 のポイントは，システムでの位置づけによって，次の二つに分けられます。

- Enterprise Bean 呼び出しで意識する必要があるポイント（1～3）
  このポイントでタイムアウトを設定する項目は，J2EE アプリケーション実行基盤のバックシステムで設定できる項目と同じです。詳細は，「8.6.3　バックシステムでのタイムアウトを設定する」を参照してください。
- データベース接続時に意識する必要があるポイント（4 と 5）
  このポイントは，さらにトランザクションでのタイムアウトとデータベースでのタイムアウトに分けて意識する必要があります。
  トランザクションでのタイムアウトの詳細は，「9.3.2　トランザクションタイムアウトを設定する」を参照してください。
  データベースでのタイムアウトを設定する項目は，J2EE アプリケーション実行基盤のバックシステムで設定できる項目と同じです。詳細は，「8.6.6　データベースでのタイムアウトを設定する」を参照してください。

それぞれのポイントでの設定については，バッチアプリケーション実行基盤の「9.3.3　タイムアウトを設定するチューニングパラメタ」，および J2EE アプリケーション実行基盤の「8.6.8　タイムアウトを設定するチューニングパラメタ」を参照してください。

参考

それぞれのポイントのデフォルト値は次のとおりです。

ポイント	デフォルト値
1	設定されていません。レスポンスを待ち続けます。
2	ポイント 1 と同じ値が Enterprise Bean 呼び出し時に自動的に引き継がれて設定されます。
3	設定されていません。タイムアウトしません。
4	180 秒
5	データベースの種類とタイムアウトの設定個所ごとに異なります。※ HiRDB の場合 ロック解放待ちタイムアウト：180 秒 レスポンスタイムアウト：0 秒（HiRDB クライアントは HiRDB サーバからの応答があるまで待ち続けます） リクエスト間隔タイムアウト：600 秒 SQL Server の場合 メモリ取得待ちタイムアウト：-1（-1 を指定した場合の動作は，SQL Server のドキュメントを参照してください） ロック解放待ちタイムアウト：-1（ロックが解放されるまで待ち続けます） XDM/RD E2 の場合 ロック解放待ちタイムアウト：なし（タイムアウト時間を監視しません） SQL 実行 CPU 時間タイムアウト：10 秒 SQL 実行経過時間タイムアウト：0 秒（タイムアウト時間を監視しません） トランザクション経過時間タイムアウト：600 秒 レスポンスタイムアウト：0 秒（HiRDB クライアントは XDM/RD E2 サーバからの応答があるまで待ち続けます）

注※

Oracle の場合，ロック解放待ちタイムアウトのデフォルトはありません。

9.3.2　トランザクションタイムアウトを設定する

ここでは，トランザクションタイムアウトの設定について説明します。トランザクションタイムアウトは，データベースシステムなど EIS とのトランザクションに設定します。DB Connector を使用してデータベースにアクセスするときのトランザクションタイムアウトについて説明します。

トランザクションタイムアウトを設定する場合は，システム全体のタイムアウトのうち，バッチサーバとデータベースのトランザクションについて意識する必要があります。

トランザクションタイムアウトが発生すると，アプリケーションサーバによって，次の処理が実行されます。

- 実行中のトランザクションはロールバックされます。
- トランザクションに参加しているコネクションはクローズされ，コネクションプールから削除されます。

ポイント

トランザクションの管理方法は BMT になります。トランザクションのタイムアウトは，usrconf.properties または JTA の API（javax.transaction.UserTransaction#setTransactionTimeout メソッド）に指定できます。

usrconf.properties の定義は，プロセス全体に影響します。API に指定したタイムアウトは，API を発行したトランザクションの範囲だけに影響します。また，API の指定は，usrconf.properties の定義よりも優先されます。

このため，プロセス全体に設定したい標準的な値を usrconf.properties に定義して，呼び出す業務によって細かく設定したい値は適宜 API を使用して設定することをお勧めします。

トランザクションタイムアウトが発生した場合，バッチアプリケーションに例外は通知されません。ただし，メッセージ KDJE31002-W がログファイルとバッチサーバのコンソールに出力されます。また，トランザクションタイムアウトが発生したあとで，バッチアプリケーションから該当するトランザクションを使用して JTA インタフェースまたは JDBC インタフェースを使用しようとすると，例外が通知されます。

9.3.3 タイムアウトを設定するチューニングパラメタ

ここでは，タイムアウトの設定で使用するチューニングパラメタの設定方法についてまとめて示します。

(1) バッチサーバ側で設定する Enterprise Bean のリモート呼び出し（RMI-IIOP 通信）と JNDI によるネーミングサービス呼び出しのタイムアウト

図 9-1 のポイント 1 のタイムアウトを設定するチューニングパラメタです。

設定方法は，J2EE アプリケーション実行基盤と同じです。「8.6.8(6)　EJB クライアント側で設定する Enterprise Bean のリモート呼び出し（RMI-IIOP 通信）と JNDI によるネーミングサービス呼び出しのタイムアウト」を参照してください。

なお，「8.6.8　タイムアウトを設定するチューニングパラメタ」の説明中の「ポイント 7」が，図 9-1 の「ポイント 1」に対応します。

(2)　EJB クライアント側で設定する CTM から Enterprise Bean 呼び出しのタイムアウト

図 9-1 のポイント 2 のタイムアウトを設定するチューニングパラメタです。

設定方法は，J2EE アプリケーション実行基盤と同じです。「8.6.8(7)　EJB クライアント側で設定する CTM から Enterprise Bean 呼び出しのタイムアウト」を参照してください。

なお，「8.6.8　タイムアウトを設定するチューニングパラメタ」の説明中の「ポイント 8」が，図 9-1 の「ポイント 2」に対応します。

(3)　Enterprise Bean 呼び出しアクセスした EJB で設定するメソッドの実行時間のタイムアウト

図 9-1 のポイント 3 のタイムアウトを設定するチューニングパラメタです。

設定方法は，J2EE アプリケーション実行基盤と同じです。「8.6.8(11)　J2EE アプリケーションのメソッドタイムアウト」を参照してください。

なお，「8.6.8　タイムアウトを設定するチューニングパラメタ」の説明中の「ポイント 9」が，図 9-1 の「ポイント 3」に対応します。

(4)　バッチサーバ側で設定するデータベースのトランザクションタイムアウト（DB Connector を使用した場合）

図 9-1 のポイント 4 のタイムアウトを設定するチューニングパラメタです。

バッチサーバ単位，Enterprise Bean，インタフェース，API による呼び出し単位（BMT の場合）に設定します。

トランザクションタイムアウトのチューニングパラメタを次の表に示します。

表 9-3　トランザクションタイムアウトのチューニングパラメタ

単位	設定方法	設定項目	設定個所
バッチサーバ単位	Smart Composer 機能	トランザクションのトランザクションタイムアウト時間のデフォルト	**定義ファイル** 簡易構築定義ファイル **設定対象** 論理 J2EE サーバ（j2ee-server） **パラメタ名** ejbserver.jta.TransactionManager.defaultTimeOut
API 単位（BMT）	API	トランザクションタイムアウト時間	UserTransaction#setTransactionTimeout メソッド※

注※　パッケージ名は javax.transaction です。

(5)　データベースのタイムアウト

図 9-1 のポイント 5 のタイムアウトを設定するチューニングパラメタです。

設定方法は，J2EE アプリケーション実行基盤と同じです。「8.6.8(10)　データベースのタイムアウト」を参照してください。

なお，「8.6.8　タイムアウトを設定するチューニングパラメタ」の説明中の「ポイント 12」が，図 9-1 の「ポイント 5」に対応します。

9.4 GC 制御で使用するしきい値を設定する

バッチサーバの FullGC を実行するタイミングは，メモリ使用量にしきい値を設定することで制御できます。しきい値は，バッチ処理とオンライン処理で同じリソースにアクセスする場合に設定することをお勧めします。適切なしきい値を設定しておくことで，オンライン処理とバッチ処理の両方のスループットを確保できます。

ポイント

FullGC の詳細については，「7.2.7 GC の発生とメモリ空間の関係」を参照してください。

9.4.1 しきい値を設定する目的

オンライン処理とバッチ処理で同じリソースにアクセスする場合，オンライン処理のスループットに影響を与えないように考慮する必要があります。

バッチアプリケーション実行時に空きメモリが少なくなると，JavaVM によってバッチサーバの FullGC が実行されます。この場合，バッチサーバ上で動作するすべてのプログラムの処理が中断されます。バッチアプリケーションがリソースを排他していた場合，そのリソースはバッチサーバの FullGC 実行中も排他された状態になります。オンライン処理の中に排他中のリソースを使用する処理があった場合は，そのオンライン処理も中断されてしまいます。

これを防ぐために，メモリ使用量のしきい値を設定して，メモリ不足が起こる前に明示的に FullGC を発生させるようにします。明示的な FullGC は，リソースを排他していないタイミングで発生するように制御できます。JavaVM によって FullGC が実行される前に空きメモリを増やしておくことで，リソース排他中に FullGC が実行されることを防ぎます。

しきい値を設定した場合，次に示す状態になると FullGC が実行されます。ただし，そのときにバッチアプリケーションがリソースを排他していた場合は，排他が解除されるまで待ってから実行されます。

SerialGC が有効な場合

- Tenured 領域消費サイズの Tenured 領域合計サイズに対する割合≧しきい値
- New 領域合計サイズの Tenured 領域最大空きサイズに対する割合≧しきい値
- Metaspace 領域消費サイズの Metaspace 領域最大サイズに対する割合≧しきい値

G1GC が有効な場合

- Java ヒープ領域消費サイズの Java ヒープ領域合計サイズに対する割合≧しきい値
- Metaspace 領域消費サイズの Metaspace 領域最大サイズに対する割合≧しきい値

9.4.2　しきい値設定の考え方

しきい値は，次の式で算出した値を目安に算出できます。

```
しきい値≦100-（100×リソース排他解除を待つ間に必要な空きメモリのサイズ）／最大メモリサイズ
```

しきい値設定の際には，次の点を考慮してください。

- FullGC の発生頻度
- リソース排他解除を待つ間に必要な空きメモリ

ここでは，FullGC の発生頻度としきい値の関係と，リソース排他解除を待つ間に必要な空きメモリのサイズの見積もり方法について説明します。

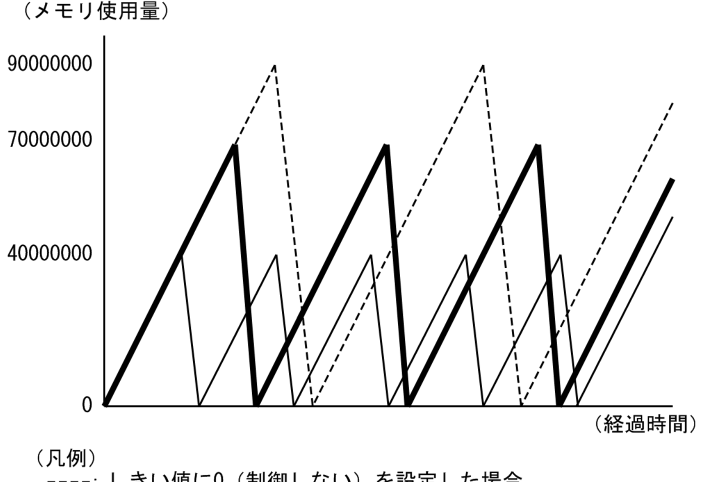
(1)　FullGC の発生頻度としきい値の関係

FullGC はプログラムの実行速度に比べて時間の掛かる処理です。このため，JavaVM のチューニングでは，FullGC をできるだけ発生させないようにします。JavaVM のチューニングの考え方については，「7.3.1　チューニングの考え方」を参照してください。

しきい値を設定して明示的に実行する FullGC も，頻度が少なくなるよう，チューニングする必要があります。

設定したしきい値ごとのメモリ使用量の例を，次の図に示します。

図 9-2　設定したしきい値ごとのメモリ使用量の例



JavaVM でのメモリ使用量は，時間の経過とともに増加していき，FullGC が発生すると減少します。

しきい値に 0 を設定した場合は，JavaVM によって自動的に実行されるまで FullGC は実行されません。しきい値に小さな値を設定した場合は，大きな値を指定した場合に比べて，FullGC が実行される頻度が

高くなります。図の場合は，しきい値として 80 を設定した場合の方が，50 を設定した場合よりも FullGC の実行頻度を低く抑えることができています。

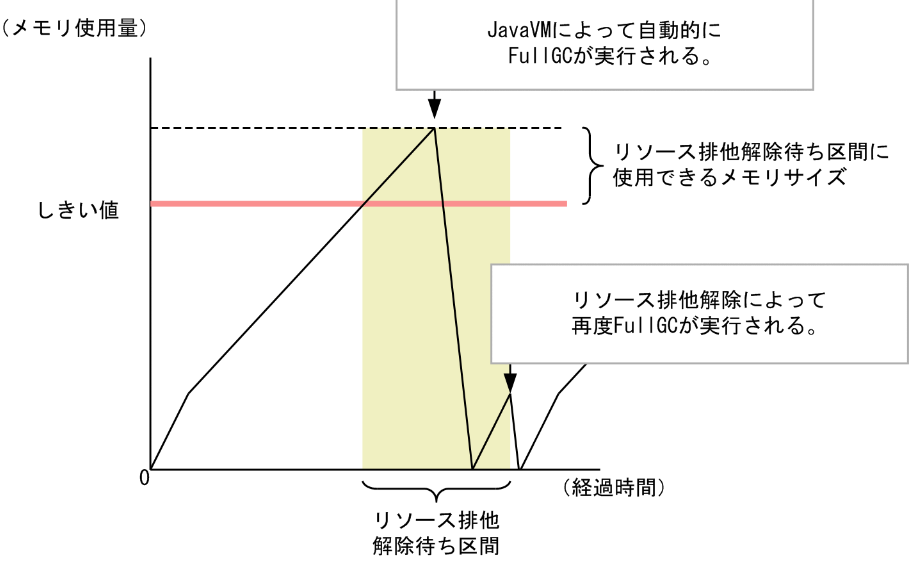
ただし，FullGC の実行頻度を少なくした場合，1 回の実行に掛かる時間は，頻繁に実行する場合に比べて長くなります。

(2) リソース排他解除を待つ間に必要な空きメモリの見積もり

メモリ使用量がしきい値以上になった場合，リソース排他が解除されるまで FullGC は実行されません。しかし，リソース排他の解除を待っている間に JavaVM が必要とするメモリが不足した場合は，リソース排他解除を待たないで JavaVM によって FullGC が実行されてしまいます。

リソース排他解除を待つ間に空きメモリが不足した場合の例を次の図に示します。

図 9–3 リソース排他解除を待つ間に空きメモリが不足した場合の例



> **ポイント**
>
> JavaVM に FullGC を実行させないために必要な空きメモリについては，「7.2.7 GC の発生とメモリ空間の関係」を参照してください。

FullGC が発生するメモリ使用量を 100 とする場合，リソース排他解除待ち区間に使用できるメモリサイズは，「100－しきい値（%）」です。

例えば，しきい値として「95」のように高い値を指定した場合は，リソース排他解除を待っている間に使用できる空きメモリが 5%と少ないため，リソース排他が解除される前に JavaVM によって自動的に FullGC が実施されてしまうおそれがあります。

このため，しきい値を見積もる場合には，リソース排他待ち区間にメモリが不足しないよう，十分に余裕を持った値を設定するようにしてください。

注意事項

リソース排他解除待ち区間で空きメモリが不足して JavaVM による FullGC が実行された場合，リソースの排他が解除されたタイミングで，FullGC 制御によってもう一度 FullGC が実行されます。

9.4.3 GC 制御で使用するしきい値を設定するためのチューニングパラメタ

ここでは，バッチサーバの FullGC を実行するしきい値を設定するために使用するチューニングパラメタの設定方法について示します。

表 9-4 バッチサーバの FullGC を実行するしきい値を設定するチューニングパラメタ

設定項目	設定個所
しきい値	**定義ファイル** 簡易構築定義ファイル **設定対象** 論理 J2EE サーバ（j2ee-server） **パラメタ名** ejbserver.batch.gc.watch.threshold

付録

付録 A　HTTP セッションで利用する Explicit ヒープの効率的な利用

明示管理ヒープ機能を利用する場合，J2EE サーバのデフォルトでは，HTTP セッションに関するオブジェクトが Explicit ヒープに配置されます。HTTP セッションを配置する Explicit メモリブロック領域の確保および解放のタイミングについては，マニュアル「アプリケーションサーバ 機能解説 拡張編」の「7.4.1 HTTP セッションに関するオブジェクト」を参照してください。なお，オブジェクトの寿命の考え方や，どのような Explicit メモリブロックが自動解放の対象となるかなどについては，「付録 B　Explicit ヒープに配置するオブジェクトの寿命による明示管理ヒープ機能への影響」を参照してください。

HTTP セッションに明示管理ヒープ機能を適用する場合は，次の観点を考慮してアプリケーションを実装すると，効率良く適用できます。

- HTTP セッションに格納するオブジェクトの寿命
- HTTP セッションに格納するオブジェクトの更新頻度
- HTTP セッションを作成するタイミング

また，これらの観点を考慮した実装となっているかどうかは，J2EE サーバのログで確認できます。

ここでは，観点ごとに，アプリケーションの実装方法，および J2EE サーバのログによる確認方法について説明します。

付録 A.1　HTTP セッションに格納するオブジェクトの寿命を考慮する

HTTP セッションに格納するオブジェクトの寿命を考慮したアプリケーションの実装方法，および J2EE サーバのログによる確認方法について説明します。

(1)　アプリケーション実装時の考慮点

HTTP セッションには，HTTP セッションが破棄されるときに解放されるオブジェクトだけを格納することをお勧めします。このようなオブジェクトの割合が多いと，自動解放処理の時間が短くなります。HTTP セッションが破棄されたあとも残存するオブジェクトがあると，新たな Explicit メモリブロックが生成され，オブジェクトはそこに移動され，以降の自動解放処理の対象となります。また，移動されたオブジェクトは，HTTP セッションと対応しないオブジェクトとなります。

明示管理ヒープ機能では，setAttribute メソッドで HTTP セッションに格納したオブジェクトから直接または間接的に参照されているオブジェクトは，HTTP セッションに格納されていると見なします。これらのオブジェクトは，昇格するタイミングで Explicit メモリブロックに移動します。HTTP セッションに格納したオブジェクトからの参照関係が複雑になると，HTTP セッションを破棄したあとも使用され続けるオブジェクトが，HTTP セッションに格納されてしまう可能性が高くなります。

このため，HTTP セッションに格納するオブジェクトは，String 型のオブジェクトや，プリミティブ型またはプリミティブ型配列を格納した場合など，できるだけシンプルにするのをお勧めします。

アプリケーションの設計上，HTTP セッションが破棄したあとも使用され続けるオブジェクトを HTTP セッションに格納する必要がある場合は，明示管理ヒープ機能適用除外クラス指定機能の利用を検討してください。明示管理ヒープ機能適用除外クラス指定機能については，マニュアル「アプリケーションサーバ 機能解説 拡張編」の「7.10　Explicit メモリブロックの自動解放処理に掛かる時間の短縮」を参照してください。

(2)　J2EE サーバログによる確認方法

HTTP セッションに格納するオブジェクトの寿命は，スレッドダンプの内容から確認します。

確認手順を次に示します。

1. J2EE サーバを起動し，HTTP セッションが破棄されるまでの一とおりの業務を実行します。

2. HTTP セッションが破棄されたあと，eheapprof コマンドを実行してスレッドダンプを取得します。

3. <EM_NAME>が"NULL"の Explicit メモリブロックを検索します。

   "NULL"の Explicit メモリブロックがない場合は，HTTP セッションが破棄されたあとも使用され続けるオブジェクトがないことを示します。

   "NULL"の Explicit メモリブロックがある場合は，「used」（Explicit メモリブロックの利用済みサイズ）の値を確認します。

   出力例を次に示します。

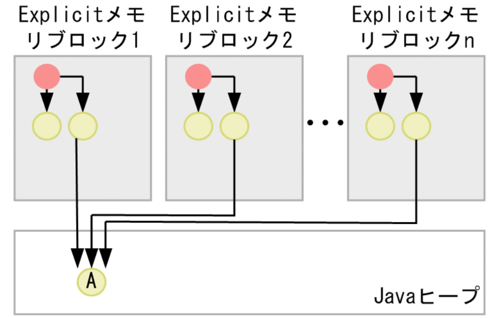
   ```
   "NULL" eid=1(0x02f25610)/A, total 542K, used 501K, garbage 0K (92.4% used/total, 0.0% gar
   bage/used, 0 blocks) Enable
   ```

   この Explicit メモリブロックには，HTTP セッションが破棄されたあとも使用され続けるオブジェクトがあることを示します。ただし，「used」の値が数キロバイト～数メガバイト程度の場合は，自動解放処理時間に大きな影響を与えることはありません。

(3)　HTTP セッションの破棄後に HTTP セッションに格納したオブジェクトが使用される仕組み

HTTP セッションが破棄したあとも HTTP セッションに格納したオブジェクトが使用されてしまう仕組みを説明します。ここでは，代表的な例として，HTTP セッションに格納したオブジェクトから共有データ（オブジェクト）を参照する例を次の図に示します。

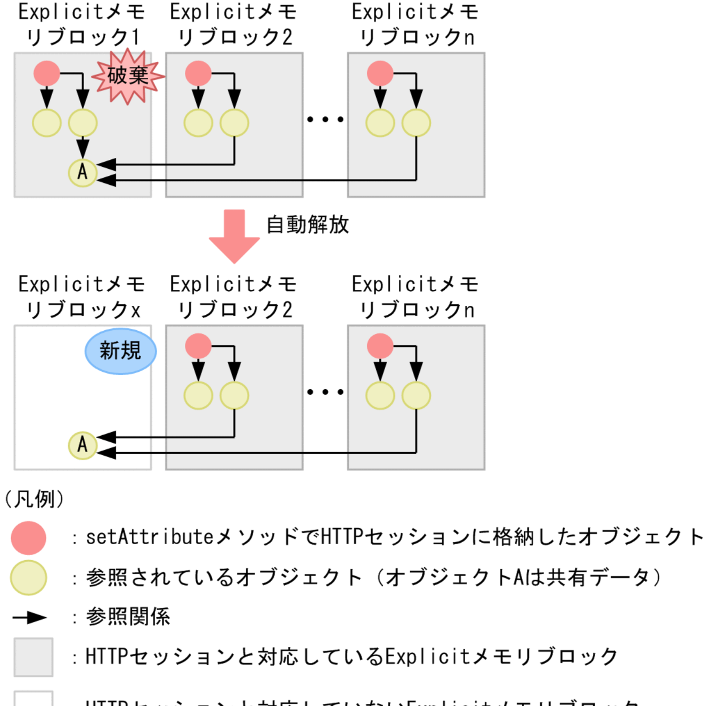
図 A-1　HTTP セッションに格納したオブジェクトから共有データ（オブジェクト）を参照する例



この例は，setAttribute メソッドで HTTP セッションに格納したオブジェクトから共有データであるオブジェクト A を参照している例です。オブジェクト A は HTTP セッションに格納されていないようにも見えますが，参照関係に基づくオブジェクトの移動によって，参照元のどれかの HTTP セッションに対応する Explicit メモリブロックに移動します。この例では，Explicit メモリブロック 1 に移動するとします。参照関係の基づくオブジェクトの移動については，マニュアル「アプリケーションサーバ 機能解説 拡張編」の「7.6.5　参照関係に基づくオブジェクトの Java ヒープから Explicit メモリブロックへの移動」を参照してください。

このあと，Explicit メモリブロック 1 に対応する HTTP セッションが破棄されて，自動解放処理が発生すると，外部（解放対象外の Explicit メモリブロック）から参照されているオブジェクト A は，新規の Explicit メモリブロック x に移動されます。この Explicit メモリブロック x は HTTP セッションと対応していない Explicit メモリブロックとなります。HTTP セッションを破棄したあとの Explicit メモリブロックの状態を次の図に示します。

図 A-2　HTTP セッションを破棄したあとの Explicit メモリブロックの状態



HTTP セッションの破棄と対応した自動解放は，対象となる Explicit メモリブロックに使用中のオブジェクトが多いほど，処理に時間が掛かります。また，HTTP セッションの破棄と対応した自動解放処理時に使用中のオブジェクトは，HTTP セッションとは対応しない Explicit メモリブロックに移動され，自動解放処理の対象となります。このオブジェクトを解放できない期間が長いほど，このオブジェクトが自動解放処理の対象となる回数が増え，スループットやレイテンシの悪化につながることがあります。

この例の場合，共有データであるオブジェクト A への参照は，n 個の HTTP セッションが一つもなくなるまで消えないと考えられます。その間は，新規の Explicit メモリブロック x の自動解放処理を繰り返すことになり，効率が悪くなります。共有データのサイズが小さければ，特に大きな影響はありませんが，数百メガバイト以上の巨大なサイズの場合は，自動解放処理時間への影響が大きく，スループットやレイテンシの悪化の原因となることがあります。

付録 A.2　HTTP セッションに格納するオブジェクトの更新頻度を考慮する

HTTP セッションに格納するオブジェクトの更新頻度を考慮したアプリケーションの実装方法，および J2EE サーバのログによる確認方法について説明します。

(1)　アプリケーション実装時の考慮点

HTTP セッションに格納したオブジェクトの更新回数が少ないほど，Explicit メモリブロックのメモリ効率が良くなります。例えば，setAttribute メソッドを第 1 引数の「name」が同じ値で複数回呼び出すと，同じ値で呼び出した回数分だけ第 2 引数の「value」のオブジェクトが更新されます。これがオブジェクトの更新回数となります。

HTTPセッションに格納したオブジェクトが更新されると，更新前のオブジェクトは使用済みになりますが，そのメモリ領域はHTTPセッションの破棄まで解放されません。このため，更新回数が少ないほどメモリ効率は良くなります。オブジェクトの更新を，短い周期（CopyGC間隔の10倍程度以下の周期）で繰り返す場合は問題ありませんが，長い周期で繰り返すと長寿命オブジェクトとなってExplicitメモリブロックに移動するため，注意してください。Explicitメモリブロックに移動したオブジェクトは，使用済みとなっても，HTTPセッションの破棄まで解放されません。明示管理ヒープ機能を適用しない場合，このようなオブジェクトはフルガーベージコレクション発生時に解放されます。特に，日中はHTTPセッションが破棄されないといった，HTTPセッションの生存期間が長いシステムの場合は，長い周期でオブジェクトの更新を繰り返す状況となりやすいことがあります。

アプリケーションの設計上，HTTPセッションに格納したオブジェクトの更新回数が多くなる場合は，次の点を考慮して，明示管理ヒープ機能の適用可否を判断してください。

- 明示管理ヒープ機能を適用しない場合のフルガーベージコレクションによるレイテンシのスパイク
- 明示管理ヒープ機能を適用した場合のメモリ効率の悪化

(2) J2EEサーバログによる確認方法

HTTPセッションに格納するオブジェクトの更新頻度は，スレッドダンプの内容から確認します。

確認手順を次に示します。

1. J2EEサーバを起動し，HTTPセッションが破棄される直前までの業務を実行します。
2. eheapprofコマンドを実行してスレッドダンプを取得します。
3. <EM_NAME>が"CCC#HttpSession"のExplicitメモリブロックを検索します。
   "CCC#HttpSession"のExplicitメモリブロックは，HTTPセッションと対応するブロックです。「used」（Explicitメモリブロックの利用済みサイズ）の値と，アプリケーション開発時に算出して見積もったHTTPセッションに格納するオブジェクトの合計サイズを比較します。オブジェクトの更新頻度が高いと，二つの値に大きな差が生じ，「used」の値が大きくなります。
   出力例を次に示します。

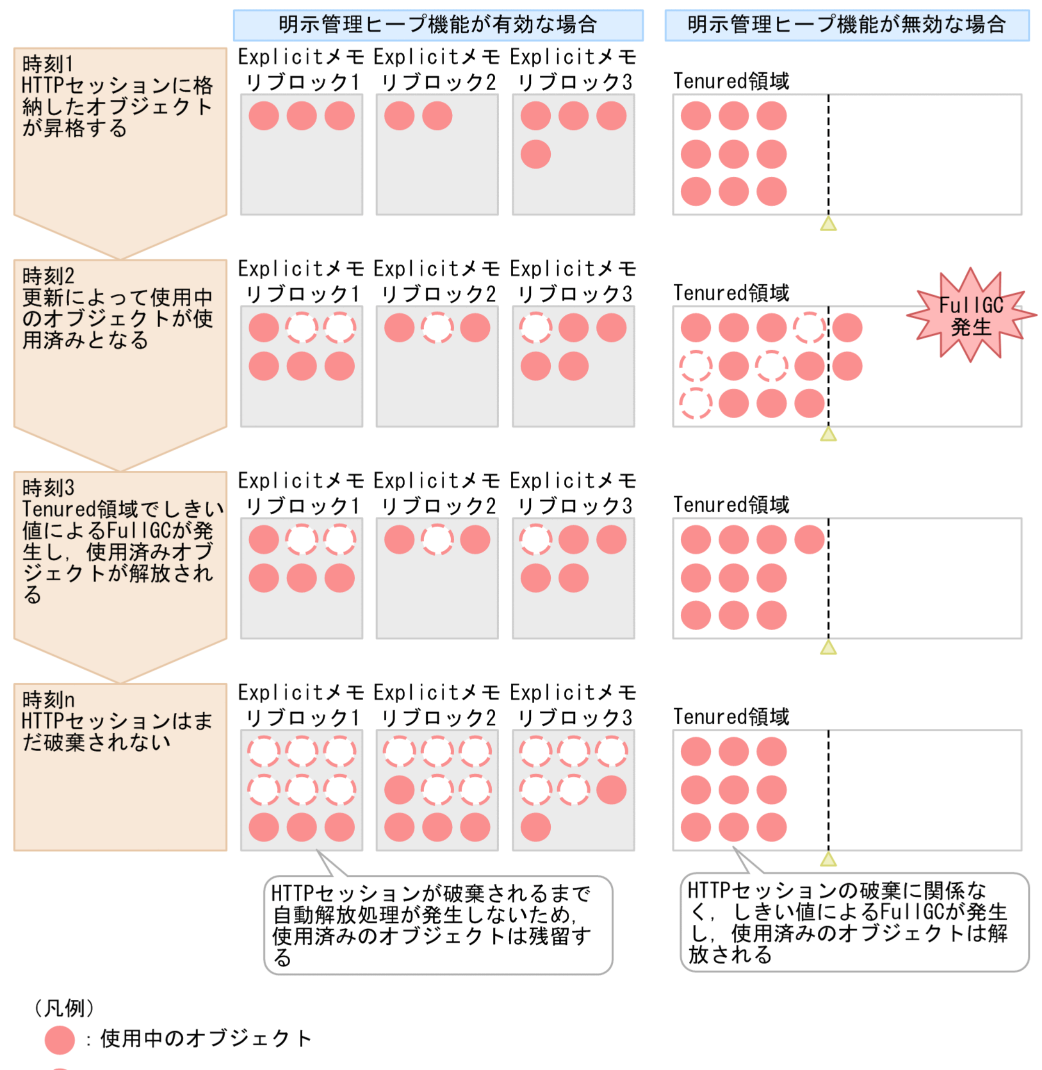
```
"CCC#HttpSession" eid=97(0x02f25610)/R, total 16K, used 8K, garbage 0K (50.1% used/total,
 0.0% garbage/used, 0 blocks) Enable
```

(3) ExplicitメモリブロックにHTTPセッションに関するオブジェクトが残存する仕組み

更新を繰り返した結果，使用済みのHTTPセッションに関するオブジェクトが，Explicitメモリブロックに残存してしまう仕組みを説明します。ここで説明するオブジェクトは，HTTPセッションの破棄よりも生存期間が短い長寿命オブジェクトです。

明示管理ヒープ機能が有効な場合と明示管理ヒープ機能が無効な場合のオブジェクトの解放処理の違いを次の図に示します。

図 A-3　明示管理ヒープ機能が有効な場合と明示管理ヒープ機能が無効な場合のオブジェクトの解放処理の違い



この図は，HTTP セッションが長時間破棄されない例です。時刻 2 では，オブジェクトの更新によって，4 個のオブジェクトが使用済みとなります。時刻 3 では，明示管理ヒープ機能が無効な場合，Tenured 領域のサイズがフルガーベージコレクションを発生するためのしきい値を超えたため，フルガーベージコレクション（図中の FullGC 発生が該当します）が発生し，使用済みのオブジェクトは解放されます。明示管理ヒープ機能が有効な場合，フルガーベージコレクションが発生しないため，使用済みのオブジェクトは解放されないで残ります。さらに時間が経過した時刻 n でも，HTTP セッションはまだ破棄されません。明示管理ヒープ機能が無効な場合，しきい値によるフルガーベージコレクションの発生で Tenured 領域の使用済みのオブジェクトは解放されます。明示管理ヒープ機能が有効な場合，HTTP セッションが破棄されるまで自動解放処理が発生しないため，使用済みで解放されていないオブジェクトが残り，Explicit ヒープがあふれる可能性が高くなります。

この例の時刻 n では，使用済みで解放されていないオブジェクトは 16 個ですが，長時間（例えば，日中の間など）HTTP セッションを破棄しないことを要件とするシステムでは，さらに多くの使用済みのオブジェクトが解放されない状況になるおそれがあります。

付録 A.3　HTTP セッションを作成するタイミングを考慮する

HTTP セッションを作成するタイミングを考慮したアプリケーションの実装方法，および J2EE サーバのログによる確認方法について説明します。

(1)　アプリケーション実装時の考慮点

アプリケーションで使用しない HTTP セッションは作成しないようにすると，Explicit メモリブロックのメモリ効率が良くなります。J2EE サーバには，暗黙的に HTTP セッションを使用する機能があるため，注意が必要です。

Explicit メモリブロックと HTTP セッションは 1 対 1 で対応します。また，一つの Explicit メモリブロックの最小サイズは，16 キロバイト（デフォルトの場合は 64 キロバイト）です。HTTP セッションを一つ作成すると，HTTP セッションに格納するオブジェクトがなくても，Explicit ヒープ領域を 16 キロバイト使用します。このため，実際には使用しない HTTP セッションは作成しないほうがメモリ効率は良くなります。例えば，使用しない HTTP セッションをサーブレットへのアクセスごとに作成し，HTTP セッションのタイムアウトまで破棄しないアプリケーションがあったとします。タイムアウトが 30 分，30 分に 1 万回のサーブレットへのアクセスがあるとすると，16 キロバイト×1 万=160 メガバイト（デフォルトの場合は 640 メガバイト）の Explicit ヒープ領域を使用します。

また，JSP では，デフォルトでアクセスごとに HTTP セッションを作成します。このため，HTTP セッションが不要な処理（例えば，J2EE サーバのヘルスチェック）で JSP を使用している場合は，不要な HTTP セッションのオブジェクトが生成されて Explicit ヒープがあふれるおそれがあります。セッションを必要としない JSP では，明示的に HttpSession オブジェクトを作成しない設定にしてください。設定には，page ディレクティブの session 属性を使用します。

アプリケーションが変更できない場合は，HTTP セッションで利用する Explicit ヒープの省メモリ化機能の利用を検討してください。HTTP セッションで利用する Explicit ヒープの省メモリ化機能については，マニュアル「アプリケーションサーバ 機能解説 拡張編」の「7.11　HTTP セッションで利用する Explicit ヒープのメモリ使用量の削減」を参照してください。この機能を利用すると，HTTP セッションの破棄前でも利用率の低い Explicit メモリブロックに格納されていたオブジェクトをほかの領域に移動して集約して，利用率の低い Explicit メモリブロックを自動解放します。アプリケーションサーバ内での HTTP セッションと Explicit メモリブロックの関係が多対 1 になります。複数の HTTP セッションで一つの Explicit メモリブロックを共有できるため，Explicit メモリブロックの利用率が向上します。これによって，HTTP セッションが確保する Explicit ヒープのメモリ使用量を削減できます。

(2)　J2EE サーバログによる確認方法

不要な HTTP セッションに対応する Explicit メモリブロックが作成されていないかどうかは，スレッドダンプの内容から確認します。

確認手順を次に示します。

1. **J2EE サーバを起動し，業務を実行します。**

   アプリケーションで暗黙的に HTTP セッションを使用していることもあるので，HTTP セッションを使用する業務に絞り込まないようにしてください。

2. **業務の実行中に，eheapprof コマンドを実行してスレッドダンプを取得します。**

   Explicit ヒープ情報に出力された，「used/total」（Explicit メモリブロック利用率）と，「spaces exist」（有効な Explicit メモリブロックの数）の値を確認します。

   出力例を次に示します。

```
max 31415926K, total 162816K, used 150528K, garbage 10004K (0.0% used/max, 91.1% used/tot
al, 6.6% garbage/used), 3 spaces exist
```

   この例では，「used/total」の値が 91.1％となっています。この値が 100％に近いほどメモリ効率がよい状態です。この値が 1 けただったり，「spaces exist」の値が想定する HTTP セッションの数を大幅に上回ったりしていないかを確認します。

付録 B　Explicit ヒープに配置するオブジェクトの寿命による明示管理ヒープ機能への影響

明示管理ヒープ機能の効率的な利用には，Explicit ヒープに配置するオブジェクトの寿命が大きく影響します。Explicit ヒープ内の Explicit メモリブロックを自動解放するタイミングで，そのブロック内のすべてのオブジェクトが使用済みであると，自動解放処理時間が短くなり，メモリ効率も良くなります。

ここでは，明示管理ヒープ機能の自動解放処理で対象となる Explicit メモリブロックの選択方法，および Explicit ヒープのメモリ使用量やオブジェクトの寿命の考え方について説明します。これらを理解した上で，「付録 A　HTTP セッションで利用する Explicit ヒープの効率的な利用」の考慮点を確認し，明示管理ヒープ機能の効率的な利用を検討してください。

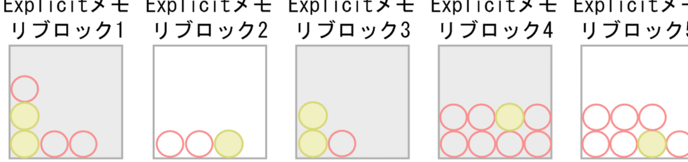
付録 B.1　Explicit メモリブロックの自動解放処理への影響

明示管理ヒープ機能の効率的な利用には，Explicit メモリブロックの自動解放処理の効率が大きく影響します。明示管理ヒープ機能では，レイテンシのスパイクが発生しないようにこまめな自動解放をするために，メモリ領域（Explicit ヒープ）を Explicit メモリブロック単位に分割して管理し，そのブロック単位に自動解放処理を実行します。Explicit メモリブロックの自動解放処理は，GC のタイミングで必要に応じて実行されます。また，自動解放処理中は，フルガーベージコレクションと同様にアプリケーションの実行が停止します。このため，自動解放処理の時間が短いほどスループットやレイテンシが向上します。

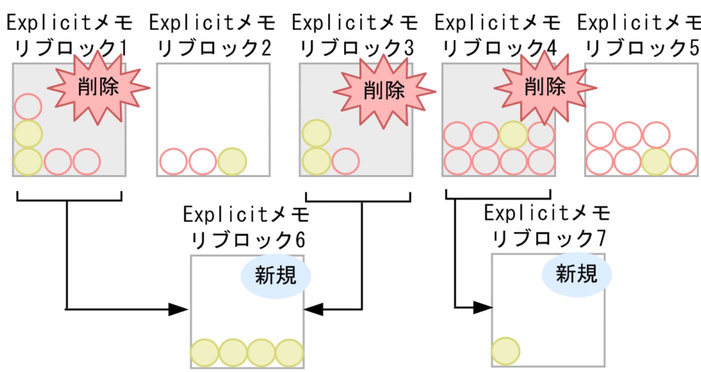
Explicit メモリブロックの自動解放処理を次の図に示します。

図 B-1　Explicit メモリブロックの自動解放処理

1. 自動解放処理対象の
Explicitメモリブロック
を選択（予約）します。



2. オブジェクトに対して
自動解放処理を実行
します。



（凡例）

：使用済みのオブジェクト

：使用中のオブジェクト

：自動解放処理対象のExplicitメモリブロック

：自動解放処理対象外のExplicitメモリブロック

自動解放処理では，自動解放処理対象の Explicit メモリブロックに使用中のオブジェクトがあると，新規に Explicit メモリブロックを作成して移動します。使用中のオブジェクトが大量にあると，その量に比例して自動解放処理の時間は増加します。また，新規に作成された Explicit メモリブロック（この例では 6 と 7）の自動解放処理も再度実行する必要があるため，全体的な自動解放処理の時間も増加します。このため，自動解放処理時に使用中のオブジェクトが Explicit メモリブロックに少ないほど，自動解放処理の効率は向上します。このことからも，明示管理ヒープ機能では，オブジェクトの寿命が効率に影響することがわかります。

なお，自動解放処理対象の Explicit メモリブロックは，次の方法で選択されます。

自動解放処理対象の Explicit メモリブロックの選択方法について

JavaVM は，次のどれかに該当する Explicit メモリブロックを自動解放処理対象として選択（解放予約）します。

1. 前回から今回までの自動解放処理の間で破棄された HTTP セッションに対応するブロック
2. 自動解放処理で新規作成されたブロック
3. 自動配置機能で作成されたブロック

ただし，2.および 3.のブロックは，Explicit ヒープの増分と解放率予測，またはしきい値によって選択されたブロックだけが，自動解放処理対象となります。

Explicit ヒープの増分と解放率予測，またはしきい値による選択方法について説明します。

Explicit ヒープの増分と解放率予測による選択

次の情報を基に複数のブロックを選択します。通常はこの方法を使用します。

- 前回の自動解放処理以降の Explicit ヒープサイズの増分
- 過去の自動解放処理の解放率（自動解放できたオブジェクトの割合）
- 各ブロック内のオブジェクトの使用率の予測値

まず，「前回の自動解放処理以降の Explicit ヒープサイズの増分」から，自動解放する予定の Explicit ヒープサイズの目標値（以降，目標自動解放サイズといいます）を設定します。Explicit ヒープが単調増加しないためには，Explicit ヒープサイズの増分以上自動解放する必要があり，増分が大きいほど目標自動解放サイズは大きくなります。目標自動解放サイズを自動解放するためには，複数のブロックの合計でどの程度のサイズを選択すればよいか（以降，選択サイズといいます）を計算します。あるブロックを自動解放処理しても，そのブロック内のすべてのオブジェクトが解放できるとは限りません。そこで，「過去の自動解放処理の解放率」を基に，自動解放する予定のオブジェクトの解放率を予測します。予測した解放率で目標自動解放サイズを達成するために必要となる選択サイズを計算します。「過去の自動解放処理の解放率」が低いほど，選択サイズは大きくなります。

次に，「各ブロック内のオブジェクトの使用率の予測値」ですべてのブロックをソートし，使用中のオブジェクトの割合が低いと予測するブロックから順に選択します。選択したブロックの合計サイズが選択サイズに達すると選択を打ち切ります。ここで選択されたブロックが自動解放処理対象になります。

しきい値による選択

「Explicit ヒープの増分と解放率予測による選択」では，使用中のオブジェクトの割合が常に高いブロックや，選択サイズを超えるサイズのブロックは選択されません。このため，Explicit メモリブロックのサイズが Explicit ヒープ全体サイズの一定の割合（しきい値）を超えた場合，強制的に自動解放処理対象として選択します。なお，「しきい値による選択」で選択したブロックは，「Explicit ヒープの増分と解放率予測による選択」では選択されません。

付録 B.2　Explicit ヒープのメモリ使用量への影響

明示管理ヒープ機能の効率的な利用には，Explicit ヒープのメモリ使用量が大きく影響します。自動解放処理の対象となるまでは，Explicit メモリブロック中に使用済みのオブジェクトが多いほど，Explicit ヒープのメモリ効率が悪くなります。例えば，図 B-1 の Explicit メモリブロック 2 は，三つのうち二つのオブジェクトが使用済み（66%が使用済み）です。しかし，Explicit メモリブロック 2 の使用済みのオブジェクトは，自動解放処理対象に選択されるまでは自動解放されません。このことから，自動解放処理の前は，Explicit メモリブロック中に使用中のオブジェクトが多いほうがメモリ効率は良くなります。

付録 B.3　Explicit ヒープに配置するオブジェクトの参照関係と寿命との関係

Explicit ヒープに配置するオブジェクトから参照されるオブジェクトは，GC が発生すると，「参照関係に基づくオブジェクトの Java ヒープから Explicit メモリブロックへの移動」処理によって，Java ヒープか

ら Explicit メモリブロックへ移動します。この処理は再帰的に適用され，参照関係の末端にあるオブジェクトまで Explicit メモリブロックへ移動します。このため，Explicit メモリブロック内のオブジェクトから，寿命の異なるオブジェクトへの参照が少ないほど，Explicit メモリブロック内のオブジェクトの寿命が同一に近づきます。寿命が同一に近づくことで，同じタイミングで自動解放され，メモリ効率は良くなります。すなわち，オブジェクトの参照関係は，できるだけ同一寿命のオブジェクト間で閉じるように，アプリケーションを実装することが必要になります。例えば，寿命の長い管理テーブルオブジェクトと，寿命の短い 1 業務で使用するデータオブジェクトの間では，参照関係がないほうが効率は良くなります。
参照関係に基づくオブジェクトの Java ヒープから Explicit メモリブロックへの移動については，マニュアル「アプリケーションサーバ 機能解説 拡張編」の「7.6.5　参照関係に基づくオブジェクトの Java ヒープから Explicit メモリブロックへの移動」を参照してください。

付録 C　推奨手順以外の方法でパフォーマンスチューニングをする場合のチューニングパラメタ

ここでは，「8.　パフォーマンスチューニング（J2EE アプリケーション実行基盤）」および「9.　パフォーマンスチューニング（バッチアプリケーション実行基盤）」で説明した項目について，推奨手順以外の方法でチューニングを実施するためのチューニングパラメタについて説明します。推奨以外の方法とは，運用管理ポータルを使用して設定する方法，およびファイル編集によって設定する方法です。

チューニングの考え方については，「8.　パフォーマンスチューニング（J2EE アプリケーション実行基盤）」および「9.　パフォーマンスチューニング（バッチアプリケーション実行基盤）」を参照してください。

付録 C.1　同時実行数を最適化するためのチューニングパラメタ（推奨手順以外の方法）

ここでは，同時実行数の最適化で使用するチューニングパラメタの設定方法と設定個所についてまとめて示します。

(1)　NIO HTTP サーバ使用時のリクエスト処理スレッド数

NIO HTTP サーバを使用している場合の，リクエスト処理スレッド数のチューニングパラメタの設定方法および設定個所について，次の表に示します。

なお，Web サーバ連携で HTTP Server を使用している場合のチューニングパラメタについては，マニュアル「HTTP Server」を参照してください。

表 C-1　NIO HTTP サーバ使用時のリクエスト処理スレッド数のチューニングパラメタ（推奨手順以外の方法）

設定項目	設定方法	設定個所
J2EE サーバ起動時に生成するリクエスト処理スレッド数	運用管理ポータル	［通信・スレッド制御に関する設定］画面の「スレッド制御設定」の「最小スレッド数」
	ファイル編集	usrconf.properties の webserver.connector.nio_http.min_threads キー
Web クライアントまたはリバースプロキシとの接続数の上限	運用管理ポータル	［通信・スレッド制御に関する設定］画面の「Web クライアントとの接続設定」の「最大接続数」
	ファイル編集	usrconf.properties の webserver.connector.nio_http.max_connections キー
Web クライアントまたはリバースプロキシとの接続数の上限を超えた場合に使用される TCP/IP の	運用管理ポータル	［通信・スレッド制御に関する設定］画面の「Web クライアントとの接続設定」の「TCP リスンキューの長さ」
	ファイル編集	usrconf.properties の webserver.connector.nio_http.backlog キー

設定項目	設定方法	設定個所
Listen キュー（バックログ）の最大値		
リクエスト処理スレッドの最大数	運用管理ポータル	［通信・スレッド制御に関する設定］画面の「スレッド制御設定」の「最大スレッド数」
	ファイル編集	usrconf.properties の webserver.connector.nio_http.max_threads キー

(2) Web アプリケーションの同時実行数

URL グループ単位，Web アプリケーション単位，または Web コンテナ単位に設定します。

(a) URL グループ単位の同時実行数

URL グループ単位の同時実行数のチューニングパラメタの設定方法および設定個所について，次の表に示します。

表 C-2 URL グループ単位の同時実行数のチューニングパラメタ（推奨手順以外の方法）

設定項目	設定方法	設定個所
Web コンテナ単位での最大同時実行スレッド数	運用管理ポータル	［通信・スレッド制御に関する設定］画面の「スレッド制御設定」の「同時実行スレッド数」
	ファイル編集	usrconf.properties の webserver.connector.nio_http.max_servlet_execute_threads キー
デフォルトの実行待ちキューサイズ	運用管理ポータル	［Web コンテナの設定］画面の「Web コンテナの設定」の「実行待ちキューサイズ」
	ファイル編集	usrconf.properties の webserver.container.thread_control.queue_size キー

次の設定項目については，推奨手順の場合と違いがありません。

- Web アプリケーション単位での最大同時実行スレッド数
- Web アプリケーションの占有スレッド数
- Web アプリケーション単位の実行待ちキューサイズ
- URL グループ単位の同時実行スレッド数制御の定義名
- URL グループ単位での最大同時実行スレッド数
- URL グループ単位の占有スレッド数
- URL グループ単位の実行待ちキューサイズ
- URL グループ単位の制御対象となる URL パターン

(b)　Web アプリケーション単位の同時実行数

Web アプリケーション単位の同時実行数をチューニングするパラメタの設定方法および設定個所について，次の表に示します。

表 C-3　Web アプリケーション単位の同時実行数のチューニングパラメタ（推奨手順以外の方法）

設定項目	設定方法	設定個所
Web コンテナ単位での最大同時実行スレッド数	運用管理ポータル	［通信・スレッド制御に関する設定］画面の「スレッド制御設定」の「同時実行スレッド数」
	ファイル編集	usrconf.properties の webserver.connector.nio_http.max_servlet_execute_threads キー
デフォルトの実行待ちキューサイズ	運用管理ポータル	［Web コンテナの設定］画面の「Web コンテナの設定」の「実行待ちキューサイズ」
	ファイル編集	usrconf.properties の webserver.container.thread_control.queue_size キー

次の設定項目については，推奨手順と違いがありません。

- Web アプリケーション単位での最大同時実行スレッド数
- Web アプリケーションの占有スレッド数
- Web アプリケーション単位の実行待ちキューサイズ

(c)　Web コンテナ単位の同時実行数

Web コンテナ単位の同時実行数をチューニングするパラメタの設定方法および設定個所について，次の表に示します。

表 C-4　Web コンテナ単位の同時実行数のチューニングパラメタ（推奨手順以外の方法）

設定項目	設定方法	設定個所
Web コンテナ単位での最大同時実行スレッド数	運用管理ポータル	［通信・スレッド制御に関する設定］画面の「スレッド制御設定」の「同時実行スレッド数」
	ファイル編集	usrconf.properties の webserver.connector.nio_http.max_servlet_execute_threads キー

(3)　Enterprise Bean の同時実行数

Enterprise Bean の同時実行数は，Enterprise Bean 単位に設定します。

Enterprise Bean の同時実行数のチューニングパラメタについては，推奨手順の場合と違いがありません。

(4)　CTM で制御する同時実行数

CTM で制御する同時実行数のチューニングパラメタの設定方法および設定個所について，次の表に示します。CTM デーモン，アプリケーション，および Stateless Session Bean に設定する項目があります。

CTM で制御する同時実行数のチューニングパラメタの設定方法および設定個所について，次の表に示します。

表 C-5　CTM で制御する同時実行数のチューニングパラメタ（推奨手順以外の方法）

設定対象	設定項目	設定方法	設定個所
CTM デーモン	CTM が制御するスレッドの最大値およびキューごとのリクエストの登録数	運用管理ポータル	［スケジューリングの設定］画面の「CTM キューの設定」の「スレッド最大値」
		ファイル編集	ctmstart コマンドの引数「-CTMDispatchParallelCount」

なお，アプリケーションまたは Stateless Session Bean に対する設定項目については，推奨手順の場合と違いがありません。

付録 C.2　Enterprise Bean の呼び出し方法を最適化するためのチューニングパラメタ（推奨手順以外の方法）

ここでは，Enterprise Bean の呼び出し方法の最適化で使用するチューニングパラメタの設定個所についてまとめて示します。

(1)　ローカルインタフェースの使用

アプリケーション作成時に，J2EE で定義されているローカルインタフェースを使用してください。

推奨手順の場合と違いはありません。

(2)　リモートインタフェースのローカル呼び出し機能の使用

リモートインタフェースのローカル呼び出し機能をチューニングするパラメタの設定方法および設定個所について，次の表に示します。

表 C-6　リモートインタフェースのローカル呼び出し機能のチューニングパラメタ（推奨手順以外の方法）

設定項目	設定方法	設定個所
ローカル呼び出し最適化機能の適用範囲	運用管理ポータル	［EJB コンテナの設定］画面の「オプション」の「J2EE アプリケーションの呼び出し方式」

設定項目	設定方法	設定個所
	ファイル編集	usrconf.properties の ejbserver.rmi.localinvocation.scope キー

(3) リモートインタフェースの参照渡し機能の使用

リモートインタフェースの参照渡し機能をチューニングするパラメタの設定方法および設定個所について，次の表に示します。

表 C-7 リモートインタフェースの参照渡し機能のチューニングパラメタ（推奨手順以外の方法）

設定項目	設定方法	設定個所
リモートインタフェースの参照渡し機能の使用（J2EE サーバ単位）	運用管理ポータル	［EJB コンテナの設定］画面の「オプション」の「リモートインタフェースの参照渡し」
	ファイル編集	usrconf.properties の ejbserver.rmi.passbyreference キー

なお，リモートインタフェースの参照渡し機能の使用（Enterprise Bean 単位）については，推奨手順の場合と違いがありません。

付録 C.3 データベースへのアクセス方法を最適化するためのチューニングパラメタ（推奨手順以外の方法）

データベースへのアクセス方法の最適化で使用するチューニングパラメタについては，推奨手順の場合と違いがありません。

付録 C.4 タイムアウトを設定するチューニングパラメタ（推奨手順以外の方法）

ここでは，タイムアウトの設定で使用するチューニングパラメタの設定個所についてまとめて示します。

(1) Web サーバ側で設定するクライアントからのリクエスト受信，およびクライアントへのデータ送信のタイムアウト

Web サーバ連携の場合は，Web サーバ単位に設定します。Web サーバ連携の場合だけ指定できます。

表 C-8　Web サーバ側で設定するクライアントからのリクエスト受信，およびクライアントへのデータ送信のタイムアウトのチューニングパラメタ（推奨手順以外の方法）

設定方法	設定個所
運用管理ポータル	［Web サーバの設定］画面の「項目ごとに設定します。」の「追加ディレクティブ」の Timeout ディレクティブ
ファイル編集※	httpsd.conf の Timeout ディレクティブ

注※　HTTP Server の定義ファイルである httpd.conf を編集して設定します。

(2)　HTTP Server のリバースプロキシで設定する Web コンテナとのデータ送受信のタイムアウト

HTTP Server のリバースプロキシ側で設定するタイムアウトのチューニングパラメタを次の表に示します。なお，これらのチューニングパラメタは，Web サーバ連携の場合だけ指定できます。

表 C-9　HTTP Server のリバースプロキシ側で設定するタイムアウトのチューニングパラメタ（推奨手順以外の方法）

設定方法	設定個所
運用管理ポータル	［リバースプロキシの設定］画面の「リクエストのマッピングに関する設定」の「タイムアウト時間」
ファイル編集	httpsd.conf の ProxyPass パラメタの timeout キー

(3)　Web コンテナ側で設定するリバースプロキシまたは Web クライアントからのデータ受信のタイムアウト

J2EE サーバ単位で設定します。Web コンテナ側で設定するタイムアウトのチューニングパラメタを次の表に示します。

表 C-10　Web コンテナ側で設定するタイムアウトのチューニングパラメタ（推奨手順以外の方法）

設定方法	設定個所
運用管理ポータル	［通信・スレッド制御に関する設定］画面の「Web クライアントとの接続設定」の「通信タイムアウト」の「リクエスト受信」
ファイル編集	usrconf.properties の webserver.connector.nio_http.receive_timeout キー

(4)　Web コンテナ側で設定するリバースプロキシまたは Web クライアントへのデータ送信のタイムアウト

J2EE サーバ単位で設定します。Web コンテナ側で設定するタイムアウトのチューニングパラメタを次の表に示します。

表 C-11　Web コンテナ側で設定するタイムアウトのチューニングパラメタ（推奨手順以外の方法）

設定方法	設定個所
運用管理ポータル	［通信・スレッド制御に関する設定］画面の「Web クライアントとの接続設定」の「通信タイムアウト」の「レスポンス送信」
ファイル編集	usrconf.properties の webserver.connector.nio_http.send_timeout キー

付録 C.5　Web アプリケーションの動作を最適化するためのチューニングパラメタ（推奨手順以外の場合）

ここでは，Web アプリケーションの動作を最適化するために使用するチューニングパラメタの設定個所についてまとめて示します。

(1)　静的コンテンツと Web アプリケーションの配置を切り分けるためのチューニングパラメタ

静的コンテンツと Web アプリケーションの配置の切り分けは，Web サーバの動作を定義するファイルのパラメタとして指定します。

表 C-12　静的コンテンツと Web アプリケーションの配置を切り分けるためのチューニングパラメタ（推奨手順以外の場合）

設定方法	設定個所
運用管理ポータル	［論理 Web サーバの定義］画面の「リバースプロキシの設定」
ファイル編集	httpsd.conf の ProxyPass ディレクティブ※

注※　httpsd.conf の詳細については，マニュアル「HTTP Server」を参照してください。

(2)　静的コンテンツをキャッシュするためのチューニングパラメタ

静的コンテンツをキャッシュするためのチューニングパラメタについて説明します。これらのチューニングパラメタは，Web コンテナ単位または Web アプリケーション単位に設定します。

Web コンテナ単位に設定するチューニングパラメタの設定方法および設定個所について，次の表に示します。

表 C-13　静的コンテンツをキャッシュするためのチューニングパラメタ（Web コンテナ単位で設定する項目）（推奨手順以外の場合）

設定項目	設定方法	設定個所
静的コンテンツのキャッシュを使用するかどうかの選択	運用管理ポータル	［Web コンテナの設定］画面の「Web コンテナの設定」の「静的コンテンツキャッシュ機能」
	ファイル編集	usrconf.properties の webserver.static_content.cache.enabled キー
Web アプリケーション単位のメモリサイズの上限値の設定	運用管理ポータル	［Web コンテナの設定］画面の「Web コンテナの設定」の「キャッシュサイズ」
	ファイル編集	usrconf.properties の webserver.static_content.cache.size キー
キャッシュする静的コンテンツのファイルサイズの上限値の設定	運用管理ポータル	［Web コンテナの設定］画面の「Web コンテナの設定」の「ファイルサイズ」
	ファイル編集	usrconf.properties の webserver.static_content.cache.filesize.threshold キー

Web アプリケーション単位に設定するチューニングパラメタについては，推奨手順の場合と違いがありません。

付録 C.6　CTM の動作を最適化するチューニングパラメタ（推奨手順以外の方法）

ここでは，CTM の動作の最適化で使用するチューニングパラメタの設定個所についてまとめて示します。

(1)　CTM ドメインマネジャおよび CTM デーモンの稼働状態の監視間隔を設定するチューニングパラメタ

CTM ドメインマネジャの稼働状態の監視間隔をチューニングするパラメタを次に示します。

監視間隔は，送信間隔×係数の値になります。

表 C-14　CTM ドメインマネジャの稼働状態の監視間隔をチューニングするパラメタ（推奨手順以外の方法）

対象	設定項目	設定方法	設定個所
同じネットワークセグメント内にある CTM ドメインマネジャ	送信間隔	運用管理ポータル	論理 CTM ドメインマネジャの［ネットワーク設定］画面の「CTM ドメイン構成情報の送信間隔」
		コマンド実行	ctmdmstart コマンドの引数「-CTMSendInterval」

対象	設定項目	設定方法	設定個所
	係数	運用管理ポータル	論理 CTM ドメインマネジャの［ネットワーク設定］画面の「ドメインマネジャ生存判定間隔係数」
		コマンド実行	ctmdmstart コマンドの引数「-CTMAliveCheckCount」
異なるネットワークセグメントにある CTM ドメインマネジャ	送信間隔	運用管理ポータル	論理 CTM ドメインマネジャの［ネットワーク設定］画面の「指定ホストへの構成情報の送信間隔」
		コマンド実行	ctmdmstart コマンドの引数「-CTMSendHostInterval」
	係数	運用管理ポータル	論理 CTM ドメインマネジャの［ネットワーク設定］画面の「ドメインマネジャ生存判定間隔係数」
		コマンド実行	ctmdmstart コマンドの引数「-CTMAliveCheckCount」

CTM デーモンの稼働状態の監視間隔をチューニングするパラメタを次に示します。

表 C-15　CTM デーモンの稼働状態の監視間隔をチューニングするパラメタ（推奨手順以外の方法）

設定項目	設定方法	設定個所
CTM デーモン間転送時のタイムアウト	運用管理ポータル	論理 CTM の［CTM 間通信の設定］画面の「リクエスト転送時のタイムアウト時間」
	コマンド実行	ctmstart コマンドの引数「-CTMDCSendTimeOut」

(2)　負荷状況監視間隔を設定するチューニングパラメタ

負荷状況監視間隔のチューニングパラメタを次に示します。

表 C-16　負荷情報監視間隔のチューニングパラメタ（推奨手順以外の方法）

設定項目	設定方法	設定個所
CTM デーモン間転送時のタイムアウト	運用管理ポータル	論理 CTM の［スケジューリングの設定］画面の「負荷情報監視間隔」
	コマンド実行	ctmstart コマンドの引数「-CTMLoadCheckInterval」

(3)　CTM デーモンのタイムアウト閉塞を設定するチューニングパラメタ

タイムアウト閉塞は，タイムアウト発生回数と監視間隔を設定しておくことによって，実行されます。

CTM デーモンのタイムアウト閉塞のチューニングパラメタを次に示します。

表 C-17　CTM デーモンのタイムアウト閉塞のチューニングパラメタ（推奨手順以外の方法）

設定項目	設定方法	設定個所
タイムアウト発生回数	運用管理ポータル	論理 CTM の［スケジューリングの設定］画面の「タイムアウト閉塞」の「自動閉塞するタイムアウト発生回数」
	コマンド実行	ctmstart コマンドの引数「-CTMWatchRequest」（一つ目のオプション引数）
監視時間間隔	運用管理ポータル	論理 CTM の［スケジューリングの設定］画面の「タイムアウト閉塞」の「監視時間間隔」
	コマンド実行	ctmstart コマンドの引数「-CTMWatchRequest」（二つ目のオプション引数）

(4)　CTM で振り分けるリクエストの優先順位を設定するチューニングパラメタ

CTM で振り分けるリクエストの優先順位の設定は，EJB クライアントアプリケーションの場合と，J2EE サーバの場合で異なります。また，J2EE サーバの場合，システムの構築方法によって設定個所が異なります。CTM で振り分けるリクエストの優先順位を設定するチューニングパラメタを次に示します。

表 C-18　CTM で振り分けるリクエストの優先順位を設定するチューニングパラメタ（推奨手順以外の方法）

設定単位	設定方法	設定個所
J2EE サーバ	運用管理ポータル	論理 J2EE サーバの［EJB コンテナの設定］画面の「CTM の設定」の「リクエストの優先順位」
	ファイル編集	usrconf.properties の ejbserver.client.ctm.RequestPriority キー

EJB クライアントアプリケーション単位のチューニングパラメタについては，推奨手順の場合と違いがありません。

付録 C.7　バッチサーバのフルガーベージコレクションを発生させるしきい値を設定するためのチューニングパラメタ（推奨手順以外の方法）

バッチサーバのフルガーベージコレクションを実行するしきい値を設定するために使用するチューニングパラメタについて説明します。

この項目は，バッチサーバで，フルガーベージコレクションの実行を制御するときにチューニングを検討してください。

表 C-19　バッチサーバのフルガーベージコレクションを実行するしきい値を設定するチューニングパラメタ

設定項目	設定方法	設定個所
しきい値	運用管理ポータル	［コンテナの設定］画面の「拡張パラメタ」
	usrconf.properties	ejbserver.batch.gc.watch.threshold キー

付録 D　用語解説

マニュアルで使用する用語について

マニュアル「アプリケーションサーバ & BPM/ESB 基盤 用語解説」を参照してください。

索引

記号

-XX:+HitachiVerboseGCPrintTenuringDistribution オプション　261

C

cjclstartap プロセスの見積もり　210
CJMSP リソースアダプタ　77
Concurrent Marking（CM）　324
CopyGC　235
CopyGC の仕組み　236
CORBA ネーミングサービス（インプロセス起動時）のスレッド数の見積もり　199
CTM　24, 34
CTM と EJB コンテナ（J2EE サーバ）の配置　62
CTM のデーモンプロセスが使用するメモリの使用量の見積もり　213
C ヒープ領域　241

D

DB Connector　72
DB Connector for Reliable Messaging　74
DD　28

E

Eden 領域　240
EJB クライアント　43, 67
EJB クライアント構成　83
EJB コンテナ　30
EJB-JAR　29
Enterprise Bean の呼び出し方法の最適化　357
Explicit ヒープ　239, 249
Explicit ヒープあふれ　298
Explicit ヒープのチューニング　277
Explicit ヒープ領域　241

F

FullGC　235, 310, 329
FullGC 発生を抑止するためのチューニングの概要 248

G

G1GC で実行される GC　317
G1GC の概要　309
G1GC の仕組み　309
G1GC のチューニング　332
GC　234, 309
GC 対象外の領域（明示管理ヒープ機能を使用した Explicit ヒープ領域の利用）　239
GC と JavaVM のメモリ管理の概要　232
GC の発生とメモリ空間の関係　246

I

InPlace　349

J

J2EE アプリケーション　26, 28
J2EE アプリケーションと J2EE コンポーネントの関係　26
J2EE アプリケーションの構造　27
J2EE コンテナ　29
J2EE コンポーネント　26
J2EE サーバ　23, 25
J2EE サーバが使用する仮想メモリの使用量の見積もり　211
J2EE サーバの構成　25
J2EE サーバの構造　26
J2EE サービス　30, 38
J2EE リソース　31, 39
JavaVM オプション　240
JavaVM 固有領域　240
JavaVM で使用するメモリ空間の構成（ZGC の場合）　345
JavaVM のメモリチューニング　231
Java ヒープ　240
Java ヒープ内に一定期間存在するオブジェクトの扱いの検討　266

Java ヒープ内の Metaspace 領域のメモリサイズの見積もり 269
Java ヒープ内の New 領域のメモリサイズの見積もり 261
Java ヒープ内の Tenured 領域のメモリサイズの見積もり 258
Java ヒープの最大サイズ／初期サイズの決定 268
Java ヒープのチューニング 254
Java ヒープ領域のメモリ管理方法（ZGC の場合） 347
JCA と EIS の配置 62
JP1 と連携したシステムの運用 50
JPA プロバイダ 32

M

Management Server 25, 35, 50, 122
Metaspace 領域 241
MixedGC 310, 327

N

N:1 リカバリシステム 142
New 領域を対象とした GC の仕組み 312
NIO HTTP サーバ 42

O

OS 固有領域 240
OS のパラメタ設定（ZGC の場合） 352
OutOfMemory 発生時強制終了機能 245
OutOfMemory ハンドリング機能 245

P

PRF デーモン 24, 34

S

SerialGC 使用時の JavaVM で使用するメモリ空間の構成 240
SerialGC の概要 234
SerialGC の仕組み 234
Survivor 領域 241

T

Tenured 領域 241
TP1/Client/J 76
TP1/Message Queue - Access 75
TP1 Connector 76
TP1 インバウンドアダプタ 76
TPBroker OTM クライアント 155
TPBroker クライアント 155

W

Web アプリケーション 29
Web クライアント構成 79
Web コンテナ 29
Web サーバ 24
Web サーバ選択の指針 42
Web サーバと Web コンテナ（J2EE サーバ）の配置 61
Web フロントシステム 53

Y

YoungGC 310, 321

Z

ZGC サイクル 346
ZGC 使用時の他機能への影響（JDK17 以降の場合） 353
ZGC 使用時の注意事項（JDK17 以降の場合） 355
ZGC での GC の考え方 351
ZGC 独自の GC 発生要因 349
ZGC の概要 345
ZGC の仕組み（JDK17 以降の場合） 345
ZGC のチューニング（JDK17 以降の場合） 351

あ

アーカイブ形式の J2EE アプリケーション 28
アクセスポイント 55, 66
アクセスポイントになるコンポーネントの種類ごとに検討が必要な項目 56
アセンブル 28
アプリケーションサーバ 16

アプリケーションサーバが使用する TCP/UDP のポート番号 160
アプリケーションサーバマシン 79
アプリケーションで明示管理ヒープ機能を使用する場合のメモリサイズの見積もり 287
アプリケーションの構成 66
アプリケーションの種類ごとにチューニングできる項目 360

い

インスタンスプーリング 370

う

運用管理エージェント 25, 34
運用管理サーバ 122
運用管理サーバマシン 65
運用管理サーバモデル 122
運用管理ドメイン 122

お

オブジェクトの寿命 236, 311
オブジェクトの退避 238, 313
オンライン業務 22

か

拡張 verbosegc 情報 256
拡張 verbosegc 情報を使用した FullGC の要因の分析方法 271
仮想メモリの使用量の見積もり 228

き

機能ごとに必要なプロセス（アプリケーションサーバ以外によって提供されるもの）と提供するソフトウェア 45
機能ごとに必要なプロセスまたはモジュール（アプリケーションサーバによって提供されるもの） 44, 47
機能ごとに必要なモジュール（アプリケーションサーバ以外によって提供されるもの）と提供するソフトウェア 44, 48
機能レイヤ 60

く

クエリータイムアウト 406
クライアント側のアプリケーションサーバ 88
クライアントマシン 79
クラスタソフトウェアと連携したシステムの運用 50

け

系切り替え 50

こ

コードキャッシュ領域 242
コネクション障害検知 385
コネクション数調節機能 387
コネクションプーリング 384
コンテナ拡張ライブラリ 32, 39

さ

サーバ側のアプリケーションサーバ 88

し

システム構成図で使用する図の凡例 65
システム構成図で使用する図の凡例（ホストの分類） 65
システム構成に共通する留意点 64
システム構成の考え方 60
システム設計の流れ 17
システム設計の目的 16
システム全体の自動運転 50
システム全体の集中監視 50
システムの分類に応じて必要なプロセス 41
実行待ちキューサイズ 364
出力内容が変更される機能（ZGC の場合） 354
使用が制限される機能（ZGC の場合） 353

す

スタック領域 242
ステートメントプーリング 388

せ

静的コンテンツ 422

静的コンテンツのキャッシュ　425
世代別 GC　234, 309
セッション制御　370, 371

そ

相互系切り替えシステム　138
相互スタンバイ　138
相互スタンバイ構成　138

ち

チューニング手順　249, 359, 437
チューニングの考え方　248
チューニングの流れ　333
チューニングの流れ（ZGC の場合）　351
チューニングの方法　362, 439

て

データベースアクセス方法の最適化　358, 436
データベースセッションフェイルオーバ機能　127
デプロイ　28
展開ディレクトリ形式の J2EE アプリケーション　28

と

統合ネーミングスケジューラサーバ　109
同時実行数の最適化　357
動的コンテンツ　422
トランザクションコンテキストのプロパゲーション 99

ね

ネーミングサービスの配置　62

は

バックシステム　53
バッチアプリケーション　37, 46
バッチ業務　22
バッチサーバ　34, 35
バッチサーバの構成　35
バッチサーバの構造　36
バッチサービス　37

パフォーマンスチューニング　356, 435
パフォーマンストレーサ　24, 34

ふ

プロセスごとに使用するメモリの見積もり　211

ほ

ホスト単位管理モデル　122

め

明示管理ヒープ機能適用時に発生しやすい問題とその解決方法　295
明示管理ヒープ機能の自動配置機能を使用した Explicit ヒープの利用の検討　291
メソッドキャンセル　410
メモリ空間とリージョンの関係　314

ゆ

ユーザサーバ　25, 35

ら

ライブラリ JAR　29

り

リージョンの使われ方　316
リカバリ専用サーバ　142
リバースプロキシ機能を有効にした Web サーバ　42

れ

レイヤ 5 スイッチ　101, 103

ろ

ローカル呼び出し最適化機能　381
論理サーバ　50
論理ユーザサーバ　157