Oracle9*i* Application Server for Sun SPARC Solaris

Oracle HTTP Server powered by Apache パフォーマンス・ガイド

リリース 1.0.2

2000 年 11 月

部品番号 : J02448-01

ORACLE®

Oracle9*i* Application Server for Sun SPARC Solaris Oracle HTTP Server powered by Apache パフォーマンス・ガイド , リリース 1.0.2

部品番号 : J02448-01

原本名：Oracle9*i* Application Server Oracle HTTP Server powered by Apache Performance Guide, Release 1.0.2

原本部品番号：A86059-01

原本著者：Julia Pond

原本協力者：Alice Chan, Gary Hallmark, Bruce Irvin, Alexander Hoefling, Sharon Malek, Carol Orange, Mukul Paithane, Leela Rao, Joan Silverman, Sanjay Singh, Eddy So

目次

1 パフォーマンスの概要

2 Web サーバーのモニター

3 サイズ設定および構成

4 HTTP サーバーのパフォーマンスの最適化

5 Apache JServ の最適化

索引

はじめに

対象読者

このマニュアルは、Oracle HTTP Server powered by Apache の設定およびチューニングを担当する Oracle9*i* Application Server の開発者およびシステム管理者を対象としています。

前提

Web サーバー、特に Apache の設定およびチューニングについては、多くの参考情報が存在します。このマニュアルでは、有用な場合にはそれらの参考情報について言及し、実際的であれば、それらの情報で説明されている設定を行った結果、どれ位パフォーマンスが向上するかについても触れます。オラクル社の社内テストで検証されていない推奨事項については、参考情報をそのまま使用し、その旨が明記されています。

すべての社内テストは、再現性のあるテスト結果を得られるよう、専用の 100Mbps のネットワークで実行されました。ご使用の環境での結果は、ネットワーク設定や競合特性によって異なります。Adrian Cockroft、Richard Petit 著『Sun Performance and Tuning: Java and the Internet』には、様々なネットワーク設定によるパフォーマンスへの影響について、優れた記述があります。

マニュアルの表記規則

このマニュアルでは、次の表記規則を使用します。

表記規則	例	説明
太字	**tnsnames.ora** **runInstaller** **www.oracle.com**	ファイル名、 ユーティリティ、 プロセス、 および URL を表します。

表記規則	例	説明
斜体	*file1*	テキスト内の可変部分を表します。このプレースホルダを特定の値や文字列に置き換えます。
山カッコ	`<filename>`	コード内の可変部分を表します。このプレースホルダを特定の値や文字列に置き換えます。
固定幅フォント	`./httpd -d .`	表示どおりに入力するテキストまたはコマンドを表します。関数にも使用します。
大カッコ	`[-c string]`	オプション項目を表します。
	`[on|off]`	オプション項目の選択肢がそれぞれ垂直バー（|）で区切って示され、その中のいずれか 1 つを選択できます。
中カッコ	`{yes|no}`	必須項目の選択肢が垂直バー（|）で区切って示されます。
省略記号	n,...	その前の項目を何回でも繰り返すことができることを表します。

1

パフォーマンスの概要

この章では、パフォーマンスとチューニングの概念について説明し、Oracle9*i* Application Server のアーキテクチャについて簡単に説明します。

内容

パフォーマンスに関する用語

次に、このマニュアルで使用されているパフォーマンスに関する用語を示します。

同時実行性
複数のリクエストを同時に処理する能力。同時実行性メカニズムの例には、スレッドおよびプロセスがあります。

レイテンシ
全体のタスクを完了するために、あるシステム・コンポーネントが別のコンポーネントを待機している時間。レイテンシは、無駄な時間として定義できます。ネットワークの場合、レイテンシはパケットのソースから宛先への移動時間として定義されます。

応答時間
リクエストの送信から応答の完了までの時間。

スケーラビリティ
使用可能なハードウェア・リソースに比例して、そして使用可能なハードウェア・リソースによってのみ制限された状態でシステムがスループットを提供できる能力。

スケーラブルなシステムとは、応答時間およびスループットに悪影響を与えずに、増加したリクエストの処理が可能なシステムです。

サービス時間
リクエストへの応答の開始から完了までの時間。

思考時間
ユーザーが実際にプロセッサを使用していない時間。

スループット
単位時間あたりに処理されるリクエスト数。

待機時間
リクエストの送信から応答の開始までの時間。

パフォーマンスのチューニングとは

パフォーマンスは、事前に設定しておく必要があります。アプリケーションの分析および設計中にパフォーマンス要件を予測し、最適なパフォーマンスのコストと利益を考慮する必要があります（1-7 ページの「パフォーマンス目標の設定」を参照してください）。この項では、次のような基本概念について説明します。

- 応答時間
- システム・スループット
- 待機時間
- 重要なリソース
- 過度の需要による影響
- 問題解決のための調整

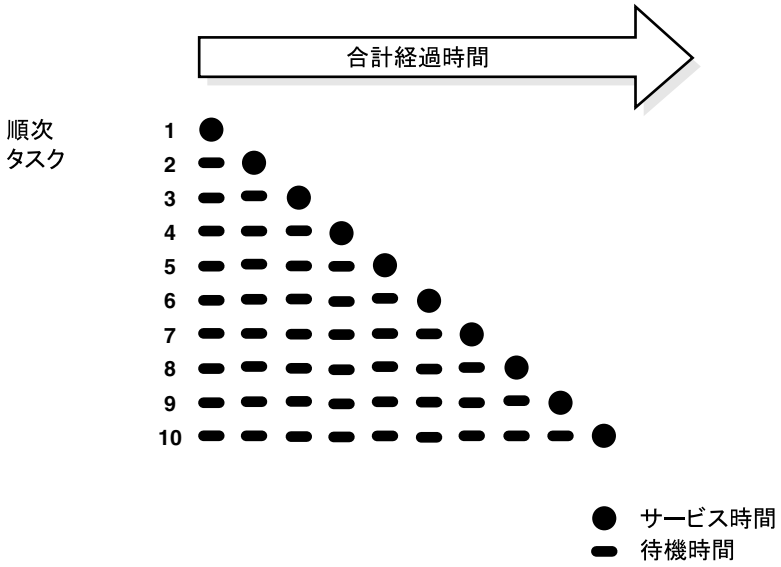
応答時間

応答時間はサービス時間と待機時間の合計であるため、次の方法でパフォーマンスを向上できます。

- 待機時間を削減する
- サービス時間を削減する

図 1-1 に、1 つのリソースに対して 10 個のタスクが競合している状態を示します。

図 1-1 個別のタスクの順次処理



この例では、待機時間なしで実行されるのはタスク 1 のみです。タスク 2 はタスク 1 が完了するまで待機し、タスク 3 はタスク 1 と 2 が完了するまで待機する必要があります。他のタスクについても同様です。（この図では各タスクの大きさは同じですが、実際のタスクのサイズはそれぞれ異なります。）

複数のリソースを使用したパラレル処理の場合、より多くのリソースをタスクに割り当てることができます。各タスクは専用のリソースを使用してすぐに実行されるため、待機時間が発生しません。

システム・スループット

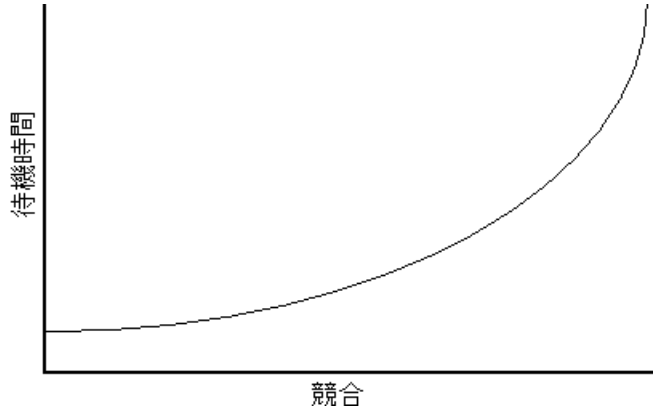
システム・スループットは、一定時間内に完了する処理量です。スループットは、次の方法で増加できます。

- サービス時間を削減する
- 不足しているリソースを増加することにより、全体の応答時間を削減する。たとえば、システムのボトルネックが CPU である場合には、CPU を追加できます。

待機時間

1 つのタスクのサービス時間が同じ場合でも、競合が増加すると待機時間は長くなります。1 秒を要するサービスを多数のユーザーが待っている場合、10 番目のユーザーは 9 秒間待機する必要があります。図 1-2 に、待機時間とリソースに対する競合の関係を示します。

図 1-2 リソースに対する競合の増加による待機時間の増加



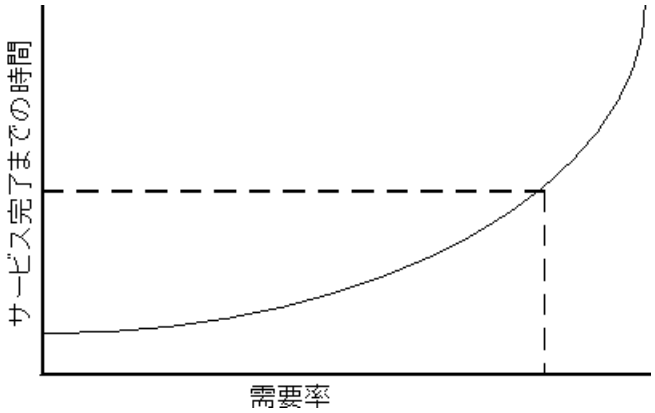
重要なリソース

CPU、メモリー、I/O 容量およびネットワーク帯域幅などのリソースは、サービス時間短縮の鍵となります。リソースを増加すると、スループットが増加し、応答時間も短縮できます。パフォーマンスは次の要因に依存します。

- 使用可能なリソースの量
- リソースを必要とするクライアントの数
- リソースに対するクライアントの待機時間
- クライアントがリソースを保持する時間

図 1-3 に、リクエスト単位数が増加すると、サービスの完了までに要する時間も増加することを示します。

図 1-3 サービス完了までの時間と需用率



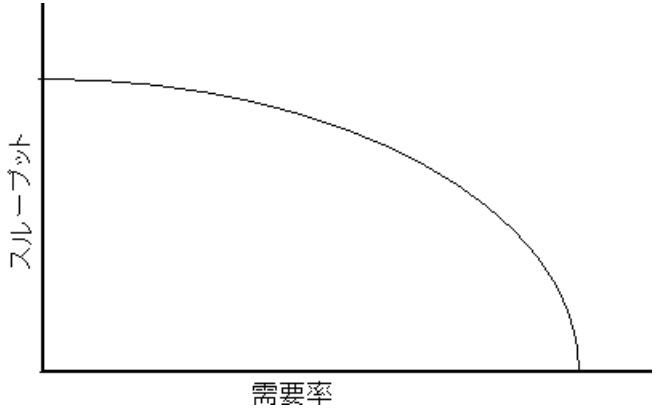
この状況を解消するには、2 つの方法があります。

- 許容範囲の応答時間を維持するために需用率を制限する
- リソースを追加する

過度の需要による影響

過度の需要により、応答時間が増加し、スループットが減少します。この様子を図 1-4 に示します。需用率がスループットの限度を超過する可能性がある場合、需要の制限手段（たとえば、Oracle HTTP Server の MaxClients および JServ の security.maxConnections など）が不可欠です。システムにどの程度の需要が生じるかを考慮し、これらの制限を考慮してアプリケーションの設計やシステム設定を行ってください。

図 1-4 需要の増加とスループットの減少



問題解決のための調整

パフォーマンスに関する問題は、次のような調整により解決できます。

単位消費	リクエストあたりのリソース（CPU、メモリー）の消費の削減により、パフォーマンスを改善できます。これは、プーリングおよびキャッシングにより実現できます。
機能面での需要	問題によっては、処理のスケジュールを変更したり、処理を分散しなおすことによって解決できます。
容量	リソース（CPU など）の増加や再割当てによって問題を解決できる場合があります。

パフォーマンス目標の設定

システムを設計する場合もメンテナンスを行う場合も、最適化の手段および対象を判断できるよう、具体的なパフォーマンス目標を設定する必要があります。特定の目標を持たずにパラメータを変更すると、目立った効果もなくシステムのチューニングに余分な時間を費やすことになります。

具体的なパフォーマンス目標の例として、注文入力の応答時間を 3 秒以内にする、などがあります。アプリケーションがその目標を達成できない場合、原因（たとえば I/O 競合など）を識別して対処します。開発中にアプリケーションをテストして、設計時に設定されたパフォーマンス目標を達成できるかどうかを調べます。

通常、チューニングには他の面とのトレード・オフが発生します。ボトルネックを判断できたら、目標を達成するために、他の部分のパフォーマンスを変更する必要がある場合もあります。たとえば、I/O が問題である場合、メモリーまたはディスクの購入が必要な場合があります。購入できない場合は、目標のパフォーマンスを得るためにシステムの同時実行性を制限する必要があります。ただし、パフォーマンスの目標が明確に定まっていれば、何が最も重要かがわかっているため、パフォーマンス向上のために何を犠牲にするかの判断が容易になります。

ユーザーの期待値の設定

アプリケーション開発者、データベース管理者およびシステム管理者は、ユーザーが期待しているパフォーマンスを、注意しながら適切に設定する必要があります。システムが特に複雑な処理を行っている場合は、単純な処理を行っている場合よりも応答時間が長くなる可能性があります。どの処理に時間がかかるかを明確にユーザーに知らせる必要があります。

パフォーマンスの評価

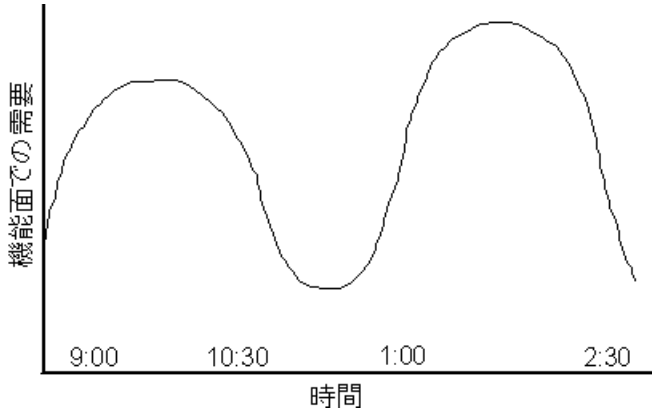
パフォーマンス目標を明確に定めると、パフォーマンスのチューニングが成功したかどうか、容易に判断できます。チューニングの成功を左右するのは、ユーザー・コミュニティに対して設定した機能面での目標、基準が満たされたかどうかを判断する能力、そして例外事項を解決するための対策を講じる能力です。

常にパフォーマンスをモニターすることにより、十分にチューニングされたシステムを維持できます。アプリケーションのパフォーマンスの履歴を記録することにより、有効な比較が可能となります。様々な大きさの負荷に関する実際のリソース消費データを使用して、客観的なスケーラビリティの調査を行うことにより、予期される負荷のボリュームに合わせたリソース要件を予測できます。

パフォーマンス管理の方法

システムの最適効率を実現するためには、計画、モニターおよび定期的な調整が必要です。パフォーマンス・チューニングの最初のステップは、目標を決定し、使用可能なテクノロジを効率的にアプリケーションに使用するよう設計することです。システムのインプリメント後には、システムを定期的にモニターし、調整する必要があります。たとえば、90% のユーザーの応答時間を 5 秒以下にし、すべてのユーザーの最長応答時間を 20 秒にするように保証するとします。通常、これは簡単なことではありません。アプリケーションには、それぞれ特徴および許容できる応答時間が異なる様々な処理が含まれます。それぞれのアプリケーションに対し、適切な目標を設定する必要があります。

図 1-5 容量と機能面での需要の調整



また、負荷の変動も判別する必要があります。たとえば、ユーザーはシステムに午前 9 時から 10 時に集中的にアクセスし、再び午後 1 時から 2 時に集中的にアクセスする可能性があります。たとえば、毎日あるいは毎週など、定期的に負荷のピークが発生する場合、一般的には負荷のピーク時の要件に合わせてシステムを設定し、チューニングします。ピーク時以外にアプリケーションにアクセスするユーザーは、ピーク時のユーザーよりも短い応答時間を得られます。負荷のピークが頻繁に発生しない場合は、少ないハードウェア構成でコストを抑えるために、負荷のピーク時には応答時間が長くても我慢することも考えられます。

パフォーマンス改善の要因

パフォーマンスは、様々な領域にまたがっています。

- アプリケーション設計 : ハードウェア・リソースを効率的に利用し、ユーザーの増加に効率的に対処するアプリケーションの設計。
- サイズ設定と構成 : パフォーマンス目標をサポートするために必要なハードウェアのタイプの判断。第 3 章「サイズ設定および構成」を参照してください。
- パラメータのチューニング : アプリケーションの最高のパフォーマンスを得るための、設定可能なパラメータの設定。第 5 章「Apache JServ の最適化」および第 4 章「HTTP サーバーのパフォーマンスの最適化」を参照してください。
- パフォーマンスのモニター : アプリケーションが使用しているハードウェア・リソースおよびユーザーが費やしている応答時間の判断。第 2 章「Web サーバーのモニター」を参照してください。
- トラブルシューティング : アプリケーションが過度にハードウェア・リソースを使用していたり、応答時間が目標よりも長い場合の理由の診断。

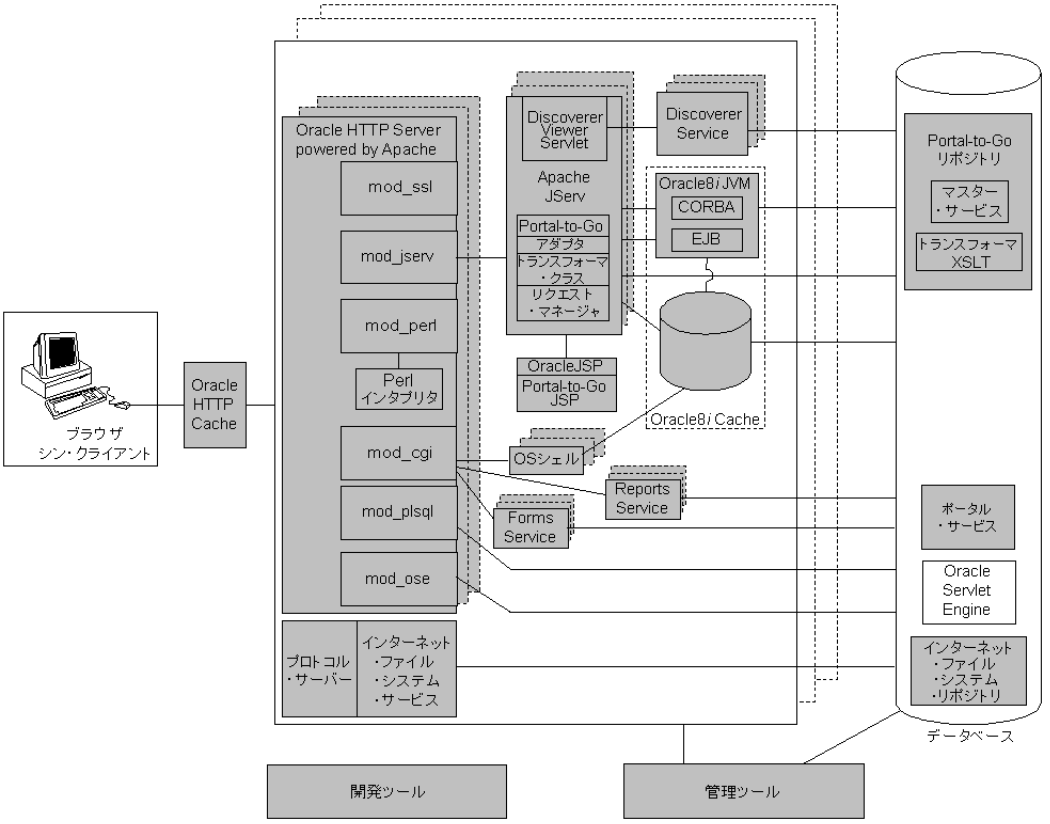
アーキテクチャ

図 1-6 に、Oracle9*i* Application Server のアーキテクチャを示します。

このガイドでは、次のコンポーネントのパフォーマンスと設定について説明します。

- Oracle HTTP Server powered by Apache
- Apache JServ
- OracleJSP

図 1-6 Oracle9*i* Application Server のアーキテクチャ



2

Web サーバーのモニター

この章では、システムに関する情報収集に使用可能なユーティリティおよびプロセスについて説明します。この情報は、リソースを最大限に活用するために役立ちます。

内容

プロセッサ使用率のモニター

プロセスの使用率を判断するためには、CPU 統計を収集する必要があります。また、ユーザーを追加し、システムのワークロードを増やして、システムのスケーラビリティをモニターする必要があります。プロセスの使用率をモニターするには、**sar**（System Activity Reporter）および **mpstat** などのユーティリティを使用します。

sar ユーティリティの使用

sar を使用すると、指定した間隔でオペレーティング・システムの累積アクティビティ・カウンタのサンプルを取得できます。

CPU 使用率のレポート

プロセスの使用率を判断するには、次の **sar** コマンドを使用します。

```
$ sar -u 5 5
```

このコマンドは、次に示すように、5 秒間隔で 5 回、CPU 使用率のサンプルを取得します。

```
$ sar -u 5 5
SunOS dummy-sun 5.5.1 Generic_103640-03 sun4u    03/02/99

15:30:25    %usr     %sys     %wio     %idle
15:30:30      49       36        0        14
15:30:35      52       41        0         7
15:30:40      46       45        0         8
15:30:45      46       44        0        10
15:30:50      50       41        0         9

Average       46       41        0         9
```

上の統計は、指定した期間中に、CPU が 9% だけアイドル状態であったことを示します。パフォーマンス基準で CPU 使用率を特定の割合よりも低くするよう指定している場合、**sar** を使用して指定した間隔で負荷のピーク時の使用率のサンプルを取得することが可能です。

sar コマンド（-u オプション）により次の統計が表示されます。

表 2-1 sar ユーティリティによってレポートされる CPU 統計

CPU 統計	説明
%usr	ユーザー・モードでプロセッサが実行される時間の割合
%sys	システム時間で実行されるプロセッサの割合
%wio	プロセッサが I/O リクエストの待機に費やす時間の割合
%idle	プロセッサがアイドルになっている割合

mpstat ユーティリティの使用

最初の引数がポーリング間隔（秒単位）という点で、**mpstat** ユーティリティと **sar** は似ています。**mpstat** の 2 番目の引数は繰返しの回数です。

次の **mpstat** コマンドは、1 秒間隔で 3 件のプロセッサ統計をレポートします。

```
$ mpstat 1 3
```

たとえば、次のようになります。

```
$ mpstat 1 3
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0    1   0    0   268   64  148   11    0    0    0    33    3   5   0  92
  0    5   0    0   250   49  157   13    0    1    0   357    2   0   0  98
  0    0   0    0   247   47  134    8    0    0    0   326    0   0   0 100
```

mpstat ユーティリティは、表 2-2 のようにプロセッサごとの統計をレポートします。

表 2-2 mpstat ユーティリティによってレポートされる CPU 統計

統計	説明
CPU	プロセッサ ID
minf	軽度の障害の数
mjf	重大な障害の数
xcal	プロセッサ間のクロス・コールの数
Intr	割込み数
ithr	スレッドとしての割込み数
csw	コンテキスト・スイッチの数
icsw	認識されないコンテキスト・スイッチの数
migr	別のプロセッサへのスレッドの移行数
smtx	mutex ロックのスピン数（つまり、最初の試行でロックが取得されなかった回数）
srw	reader-writer ロックのスピン数（つまり、最初の試行でロックが取得されなかった回数）
syscl	システム・コール数
usr	ユーザー・モードでプロセッサが消費した時間の割合
sys	システム時間でプロセッサが消費した割合

表 2-2 mpstat ユーティリティによってレポートされる CPU 統計（続き）

統計	説明
wt	プロセッサが待機時間で消費した割合（イベント待ち）
idl	アイドル時間でプロセッサが消費した割合

ネットワーク・トラフィックのモニター

Solaris の snoop または Windows NT の Network Monitor などのネットワーク・モニター・ツールを使用して、リクエストがネットワーク上で送信されている間のステータスを確認できます。

snoop ユーティリティの使用

次に、**snoop** ユーティリティを使用したネットワーク・パケットの調査例を示します。**snoop** を **netstat** とともに使用すると、ネットワーク・アクティビティの様子がよくわかります。

コマンド	結果
`snoop`	受信したパケットをすべて捕捉し表示します。
`snoop Athena`	ホスト Athena の着信および送信パケットをすべて捕捉し表示します。
`snoop -o Gods Athena Zeus`	ホスト Athena と Zeus 間のすべての着信および送信パケットを捕捉し、それを **Gods** という名前のファイルに保存します。

様々なコマンド・オプションを使用して、ファイルに捕捉されたパケットを表示できます。たとえば、次のコマンドにより、**Gods** ファイルの内容と最初のパケットに対するタイムスタンプが表示されます。

```
prompt>snoop -i Gods -t r | more
```

次に、**snoop** を使用した、FIN_WAIT_2 状態に関連した問題の診断例を示します。

```
prompt>snoop -i Gods | grep FIN
```

出力の 1 列目にはパケット番号が入ります。パケットの詳細情報を取得するには、次のように入力します。

```
prompt>snoop -i Gods -v -p<packet number>
```

snoop ユーティリティに関する優れた参考文献として、H. Frank Cervone 著『Solaris Performance Administration: Performance Measurement, Fine Tuning, and Capacity Planning for Releases 2.5.1 and 2.6』があります。

Web サーバーのモニター

パフォーマンスのチューニングには、モニターが不可欠です。Oracle HTTP Server では、**mod_status** モジュールを使用して、現在のサーバー統計を含め、サーバー側のステータス情報が提供されます。これらのサーバー・ステータス・レポートを取得するには、次に説明するように Web サーバーを設定する必要があります。

mod_status ユーティリティの使用

モニターを使用可能にするには、httpd.conf ファイルを編集して、*your_domain.com* を、モニターするサーバーのホスト名に置き換えます。

```
<Location /server-status>
     SetHandler server-status
     Order deny, allow
     Deny from all
     Allow from your_domain.com
</Location>
```

最大限の情報が表示されるように、`ExtendedStatus` ディレクティブが `On` に設定されていることを確認します。

your_domain.com のみでなく、すべてのドメインからのアクセスを許可すると、ドメイン外のマシンからご使用のサーバーをモニターすることが可能です。ただし、ご使用のサーバー・ステータスにあらゆるサイトからアクセス可能であるため、セキュリティ面で問題があることを認識する必要があります。システム・モニターに使用するドメインのみ指定することをお薦めします。

モニターを使用可能にすると、現在の統計を **http://*hostname*:*port*/server-status** で表示できます。これらの統計により、ご使用のシステムの混雑状況を知ることができます。

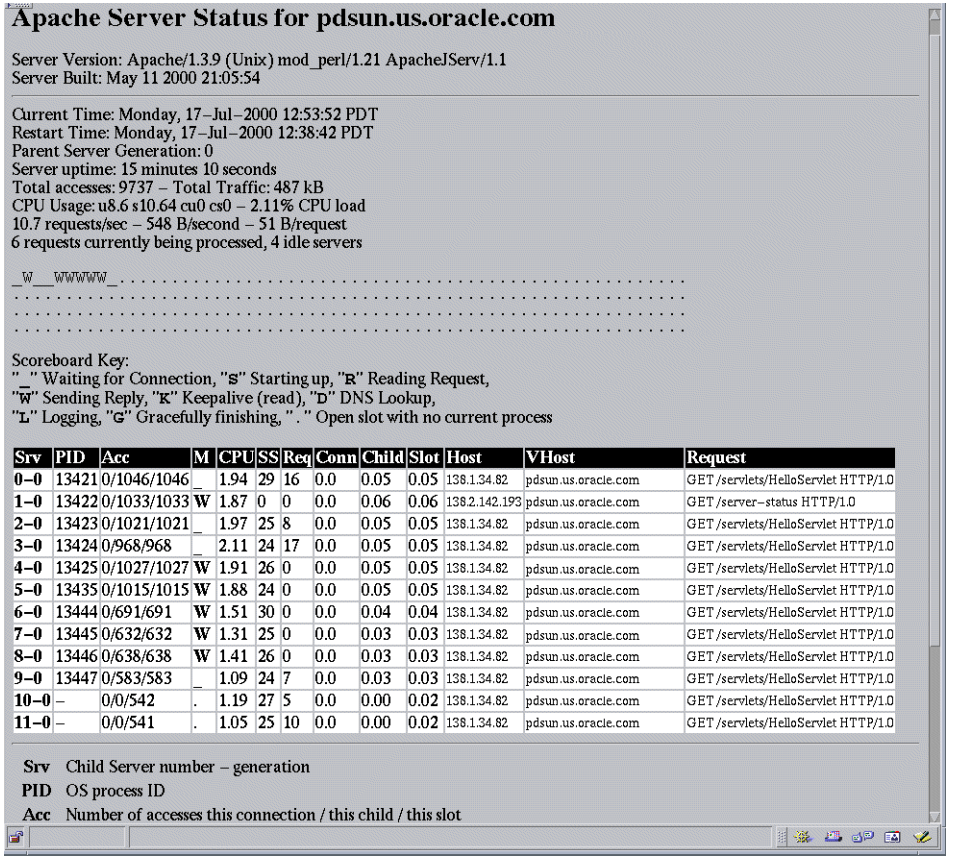
次の内容が表示されます。

- 表示中のステータスのホスト名
- サーバーのバージョン
- サーバーが構築された日付
- 現在の時間、再起動の時間、アップタイム
- 現在処理中のリクエスト数
- リクエストを処理している httpd プロセス数

- アイドル状態の httpd プロセス数
- 現在のサーバーの状態（たとえば、接続待機中、リクエストの読込み中、応答の送信中など）

図 2-1 に、ExtendedStatus がオンになっているサーバー・ステータス・ページのスナップショットを示します。

図 2-1 サーバー・ステータス・ページ



Apache Server Status for pdsun.us.oracle.com

Server Version: Apache/1.3.9 (Unix) mod_perl/1.21 ApacheJServ/1.1
Server Built: May 11 2000 21:05:54

Current Time: Monday, 17–Jul–2000 12:53:52 PDT
Restart Time: Monday, 17–Jul–2000 12:38:42 PDT
Parent Server Generation: 0
Server uptime: 15 minutes 10 seconds
Total accesses: 9737 – Total Traffic: 487 kB
CPU Usage: u8.6 s10.64 cu0 cs0 – 2.11% CPU load
10.7 requests/sec – 548 B/second – 51 B/request
6 requests currently being processed, 4 idle servers

_W__WWWWW_.....
.....
.....
.....

Scoreboard Key:
"_" Waiting for Connection, "S" Starting up, "R" Reading Request,
"W" Sending Reply, "K" Keepalive (read), "D" DNS Lookup,
"L" Logging, "G" Gracefully finishing, "." Open slot with no current process

Srv	PID	Acc	M	CPU	SS	Req	Conn	Child	Slot	Host	VHost	Request
0–0	13421	0/1046/1046	_	1.94	29	16	0.0	0.05	0.05	138.1.34.82	pdsun.us.oracle.com	GET /servlets/HelloServlet HTTP/1.0
1–0	13422	0/1033/1033	W	1.87	0	0	0.0	0.06	0.06	138.2.142.193	pdsun.us.oracle.com	GET /server–status HTTP/1.0
2–0	13423	0/1021/1021	_	1.97	25	8	0.0	0.05	0.05	138.1.34.82	pdsun.us.oracle.com	GET /servlets/HelloServlet HTTP/1.0
3–0	13424	0/968/968	_	2.11	24	17	0.0	0.05	0.05	138.1.34.82	pdsun.us.oracle.com	GET /servlets/HelloServlet HTTP/1.0
4–0	13425	0/1027/1027	W	1.91	26	0	0.0	0.05	0.05	138.1.34.82	pdsun.us.oracle.com	GET /servlets/HelloServlet HTTP/1.0
5–0	13435	0/1015/1015	W	1.88	24	0	0.0	0.05	0.05	138.1.34.82	pdsun.us.oracle.com	GET /servlets/HelloServlet HTTP/1.0
6–0	13444	0/691/691	W	1.51	30	0	0.0	0.04	0.04	138.1.34.82	pdsun.us.oracle.com	GET /servlets/HelloServlet HTTP/1.0
7–0	13445	0/632/632	W	1.31	25	0	0.0	0.03	0.03	138.1.34.82	pdsun.us.oracle.com	GET /servlets/HelloServlet HTTP/1.0
8–0	13446	0/638/638	W	1.41	26	0	0.0	0.03	0.03	138.1.34.82	pdsun.us.oracle.com	GET /servlets/HelloServlet HTTP/1.0
9–0	13447	0/583/583	_	1.09	24	7	0.0	0.03	0.03	138.1.34.82	pdsun.us.oracle.com	GET /servlets/HelloServlet HTTP/1.0
10–0	–	0/0/542	.	1.19	27	5	0.0	0.00	0.02	138.1.34.82	pdsun.us.oracle.com	GET /servlets/HelloServlet HTTP/1.0
11–0	–	0/0/541	.	1.05	25	10	0.0	0.00	0.02	138.1.34.82	pdsun.us.oracle.com	GET /servlets/HelloServlet HTTP/1.0

Srv Child Server number – generation
PID OS process ID
Acc Number of accesses this connection / this child / this slot

サーバー・ステータス情報の解析

画面（ExtendedStatus を使用可能にした状態で）には、6 つのリクエストが処理中で、4 つのサーバーがアイドル状態であることが示されています。M（モード列）の値から、サーバーが処理のどの段階にあるかを判断できます。図 2-1 では、6 つのサーバーが応答を送信中で、4 つのサーバーが接続待機中です。

ご使用のシステムの応答時間が長い場合、または httpd プロセスが応答を止めたと思われる場合、Req（リクエスト）列を見ます。この列には、最後のリクエスト処理に要した時間がミリ秒で表示されます。この数値がリクエスト処理の目標値より大きいかどうかを調べます。リクエストの完了後も、そのプロセスの M（モード）列に W と表示されている場合、プロセスは応答していないと考えられます。

これ以外にも、システムが CPU の限界に達していないかどうかをモニターすることが重要です。これは、CPU 使用率が 90% 前後の状態です。サーバー・ステータス・ページには、CPU 使用率および生成されたプロセス数が表示されます。システムが httpd プロセスの限界（**httpd.conf** 内の MaxClients ディレクティブの設定）に近づいており、パフォーマンスが悪く、プロセスが常に使用中である場合、MaxClients 設定を変更する必要があることがあります。4-5 ページの「MaxClients」を参照してください。

サーバー・ステータス表示のカスタマイズ

図 2-1 に、ある瞬間のサーバーのスナップショットを示します。サーバー・ステータス URL に refresh パラメータを含めると、任意の間隔でサーバー統計を更新して表示できます。

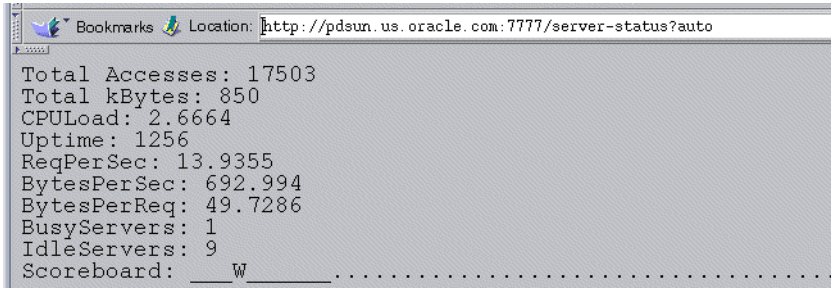
http://servername:port/server-status?refresh=x

x は、データ更新までの秒数を示す整数です。たとえば、統計を 3 秒ごとに更新するには、refresh=3 と指定します。

また、統計をデータ分析プログラムまたはスプレッドシート・プログラムで処理できるよう、統計をマシンで読込み可能な形式で表示すると便利な場合があります。これを行うには、次に示すように、URL の最後に auto を追加します。

http://servername:port/server-status?auto

図 2-2 サーバー統計表示



サーバー統計のファイルへのロギング

Apache Group により、サーバーのモニターを自動化するための Perl スクリプト **logstatus.pl** が提供されています。これは、**$ORACLE_HOME/Apache/Apache/bin/** ディレクトリに含まれています。

このスクリプトは、**cron**（またはコマンドを一定間隔で実行するその他の同様のデーモン）によって実行されるよう設計されています。このスクリプトを使用するには、次の設定値を変更する必要があります。

表 2-3 ログ・ステータス・スクリプト変数

変数	値
`$wherelog`	ログ・ファイルの場所のパス名。 例 : **/private/admin/logs/** スクリプトにより、ファイル名が作成されます。たとえば、20010945 などです。
`$port`	モニターするサーバーのポート番号。デフォルトは 80 です。
`$server`	サーバー・ホスト名。デフォルトは localhost です。
`$request`	ブラウザに入力されるとおりの、auto パラメータを使用したサーバー・ステータス・リクエスト。 例 : **http://*servername*:*port*/server-status?auto**

httpd プロセスが応答しておらず、そのプロセスを識別する必要がある場合、サーバー・ステータスを使用可能にすると大変役に立ちます。**ps**、**top** または **pmap** などのオペレーティング・システム・ユーティリティでは、応答していないプロセスは識別されません。

mod_status の詳細は、次の URL を参照してください。

http://www.oreillynet.com/pub/a/apache/2000/04/21/wrangler.html

http://www.apache.org/docs/mod/mod_status.html

JServ プロセスのモニター

Oracle9*i* Application Server の開始後、すべての JServ プロセスが正しく開始されているかどうかを確認できます。

1. **jserv.conf** ファイルの JServ ステータス・ハンドラ・セクションのコメントを削除してモニターを使用可能にし、JServ のステータスにアクセスするホストを指定します（デフォルトは localhost です）。ご使用のシステムのステータス情報にアクセス可能なホストを選択する際、必ずセキュリティ面を考慮してください。

```
<Location /jserv/>
     SetHandler jserv-status
     order deny, allow
     deny from all
     allow from oracle.com
</Location>
```

2. 次のようにブラウザに入力します。

   http://*hostname:port*/jserv/

   port には、Web サーバーがリスニングを行うポートを指定する必要があります（**httpd.conf** ファイル内に存在します）。この URL では、必ず最後にスラッシュ（/）を入力してください。最後にスラッシュを入力しないと、"not found" エラーが発生します。

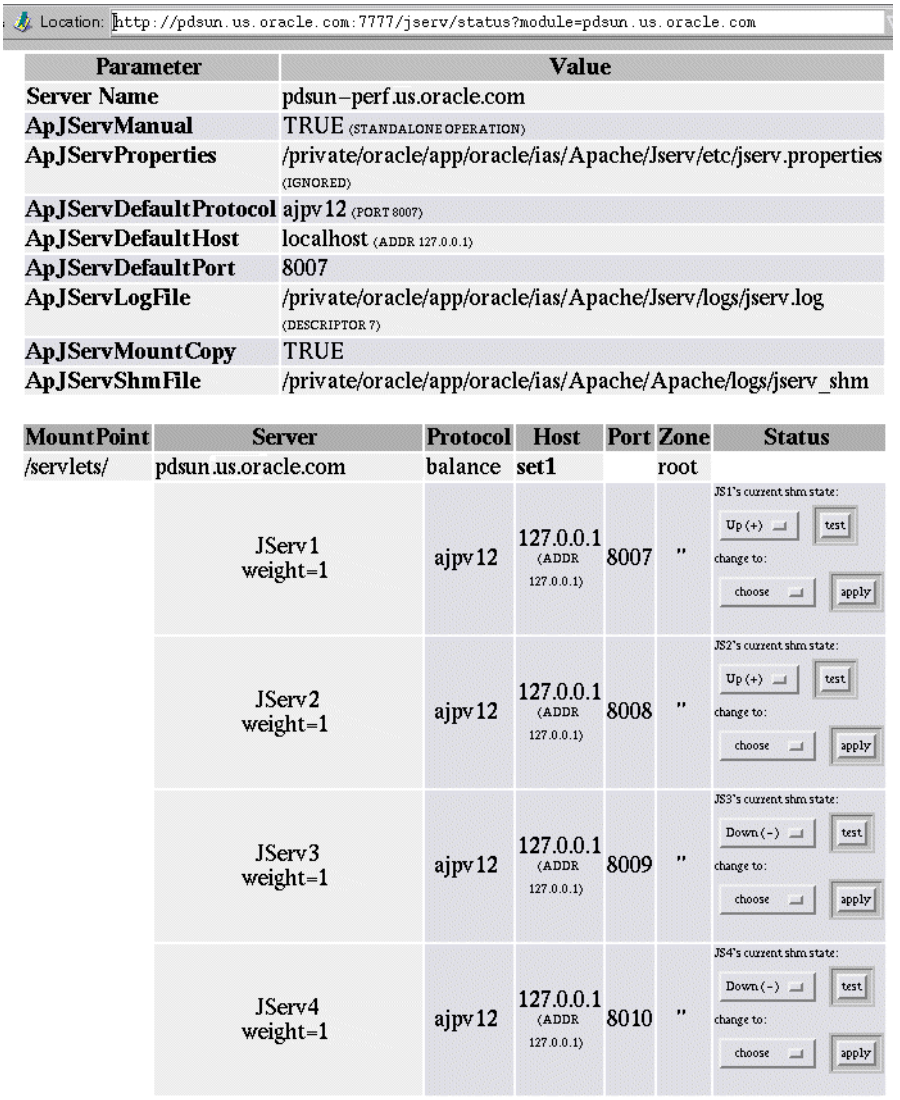
   「Configured Hosts」列にホストのリンクが表示されます。

3. モニターするホストをクリックします。

   ホストの JServ ステータス情報が、図 2-3 のように表示されます。

注意： JServ ステータス・モニターには、**jserv.conf** ファイルで設定されている JServ プロセスがすべて表示されますが、すべてが開始されているとは限りません。たとえば、図 2-3 には 4 つのプロセスが表示されていますが、ステータスが Up（そのプロセスがリクエスト処理可能であることを示す）のものは 2 つのみです。

図 2-3 JServ ステータスの表示



「Status」列には、各プロセスの現在の共有メモリー（shm）の状態が表示されます。

注意：「Status」列の値は、手動モードで開始されたプロセスについてのみ表示されます。自動モードで開始されたシングル・プロセスについては、値は表示されません。

「Up」または「Down」という単語の後のカッコ内に表示される記号には、次のような意味があります。

記号	意味
+	プロセスは実行中です。
-	プロセスは停止中です。
X	プロセスは即時停止されました。
/	プロセスは既存リクエストの実行を待って停止されました（プロセスの終了前に既存のリクエストは処理されました）。

3

サイズ設定および構成

この章では、パフォーマンス目標を達成する際に役立つ、サイズやその他の設定に関するガイドラインを示します。また、消費メモリー、I/O に関する問題、およびネットワークとソフトウェアの制約などのパフォーマンス要因についても説明します。

内容

ハードウェアおよびリソースのサイズ設定

ハードウェア・リソースは、インストールの最小推奨事項に加えて、使用するアプリケーションの要件を満たしている必要があります。ハードウェアに関連するパフォーマンスのボトルネックを避けるには、各ハードウェア・コンポーネントを容量の 80% 以下で操作してください。CPU 使用率の測定方法の詳細は、2-2 ページの「sar ユーティリティの使用」を参照してください。

特に、プロセッサとメモリー・リソースは、予期される最大のユーザー負荷に対して多めに割り当てる必要があります。

同時ユーザーおよびユーザー数について

必要なハードウェア・リソースの量は、アプリケーションによって異なります。ユーザーの思考時間やネットワークのレイテンシを考慮に入れずにリソースを推定するという誤りがよく見受けられます。アプリケーションのサイズを決定する際、予想ユーザー数と実際の同時ユーザー数との関係について把握している必要があります。これは、思考時間とアプリケーションの平均応答時間から判断します。

必要なメモリー量を判断する際、同時実行ユーザー数（ユーザー数の合計ではない）にユーザーあたりのコストを掛けた数値も考慮する必要があります。

注意： **httpd.conf** ファイルの MaxClients 設定により、同時実行ユーザー数が制限されます。MaxClients ディレクティブの詳細は、4-5 ページの「MaxClients」を参照してください。

表 3-1 に、思考時間とサービス時間が同時実行性に与える影響と、その結果生じるシステムのパフォーマンスの例を示します。

表 3-1 同時実行ユーザー

ユーザー数[1]	思考時間（秒）[2]	サービス時間（秒）[3]	同時ユーザーの範囲[4]	平均応答時間（秒）[5]	1 秒あたりのリクエスト数（スループット）[6]	CPU 使用率（%）[7]
100	0	0.3	100	5.2	19	99
100	1	0.3	65-100	4.2	19	99
100	10	0.3	0-32	0.9	9	48
100	10	0.6	0-53	2.9	8	80

1 ユーザー数 - ユーザーの合計。

2 思考時間 - ユーザーが実際にプロセッサを使用していない時間（リクエストとリクエストの間の時間）。

3 サービス時間（秒）- 1 ユーザーについて、操作を完了するまでの経過時間。

4 同時ユーザーの範囲 - サーバー上で測定されたユーザー数。**サーバー・ステータス**表示（現在処理中のリクエスト）のスナップショットから取得。**サーバー・ステータス**の詳細は、2-5 ページの「mod_status ユーティリティの使用」を参照してください。

5 平均応答時間 - 処理中のクライアントについて測定された応答時間。

6 1 秒あたりのリクエスト数（スループット）- 処理されたリクエスト数。

7 CPU 使用率 - CPU 使用率合計の平均（パーセンテージ）。

CPU 要件の決定

ほとんどのアプリケーションの場合、CPU 使用率の大部分はアプリケーションのコード処理によるものです。表 3-2 に示すように、アプリケーションの CPU 要件は、アプリケーションの複雑さと作業負荷によって異なります。

開発サイクル全体を通じて、アプリケーションの CPU 要件をモニターする必要があります。この方法は、第 2 章「Web サーバーのモニター」を参照してください。

表 3-2 336MHz SPARC プロセッサにおけるアプリケーションの CPU 要件

アプリケーション	CPU 要件（リクエストあたり）
静的ページ、20K	5ms
単純なサーブレット、JDK 1.2	20ms
単純なサーブレット、JDK 1.1.8	40ms
標準的なアプリケーション	100-200ms
複雑なアプリケーション	400-600ms

CPU 要件に対する Secure Sockets Layer の影響

Secure Sockets Layer（SSL）は、インターネット上でドキュメントを安全に送信するためのプロトコルです。SSL 接続を必要とする Web ページの URL は、**http** ではなく **https** で始まります。

SSL 接続の確立は、応答時間と CPU 使用率という点でコストがかかります。たとえば、SSL を使用しない場合に応答時間が 0.5 秒であるリクエストでは、SSL を使用した場合、応答時間が 1.7 秒でした（内部の 100Mbps ネットワーク上で測定）。SSL の使用によるパフォーマンス・コストの大部分は、接続の確立の際に発生します（336Mhz のプロセッサで 1 回の接続あたり約 125ms の CPU タイム）。

高い接続コストは、クライアントの SSL セッションの最初の接続の際に発生します。HTTP Server は、その後の接続のオーバーヘッドを削減するために SSL のセッション情報をキャッシュに入れることができるためです。詳細は、4-6 ページの「SSL セッション・キャッシュ」を参照してください。

メモリー要件の決定

この項では、次のコンポーネントのメモリー要件について説明します。

- HTTP サーバー以外のソフトウェアおよびオペレーティング・システムのメモリー
- HTTP サーバーのメモリー要件
- JServ のメモリー要件
- Java のヒープ・サイズの決定
- サーブレットおよび OracleJSP のメモリー要件
- JServ プロセスの数

HTTP サーバー以外のソフトウェアおよびオペレーティング・システムのメモリー

使用できるメモリー・リソースが解放されているアイドル状態のシステムでは、オペレーティング・システムの統計情報を見ると、常駐メモリーの使用量が仮想サイズに比較的近いことを示す場合があります。ユーザーがシステムにかける負荷が増加するにつれ、オペレーティング・システムはこれらのプロセスの不要なメモリーを利用するため、プロセスが消費する常駐メモリーが減少します。ご使用のシステムをモニターする際には、様々な使用レベルでプロセスのスナップショットを取得することをお薦めします。

オペレーティング・システムのメモリー使用量の測定およびチューニングに関する詳細は、ご使用のオペレーティング・システムのハードウェアおよびソフトウェアのドキュメントを参照してください。メモリー使用量やプロセッサの統計は、オペレーティング・システムの標準のツールでモニターできます。詳細は、第 2 章「Web サーバーのモニター」を参照してください。

サン・マイクロシステムズ社では、システムの実メモリーの 15% を、カーネルとその他のシステム・オーバーヘッド用に確保するよう薦めています。

Solaris のメモリー使用量に関する説明は、「The Solaris Memory System: Sizing, Tools and Architecture」という技術文書を参照してください。次の URL で参照可能です。

http://www.sun.com/sun-on-net/performance/vmsizing.pdf

HTTP サーバーのメモリー要件

リスナーのメモリー使用量の一連のテストで、各 HTTP リスナーは（起動時に）約 400K の常駐メモリーを使用しました。このサイズは、リスナーがアクティブである間、1 プロセスあたり 500 ～ 600K ずつ増加しました。オペレーティング・システムが休止中のときには、リスナーのメモリー使用量が起動時のサイズに戻りました。

オペレーティング・システムの標準のツールを使用して、常駐メモリー・サイズを調べることが可能です。リスナー・プロセスについて調べた場合、表示されるサイズには共有メモリーが含まれるため、前述の値より大きくなります。

JServ のメモリー要件

JDK 1.2 を使用する JServ プロセスは、起動時に 12 ～ 15MB 必要です。JDK 1.1.8 の場合、10MB 必要です。

Java のヒープ・サイズの決定

JDK 1.1.8 の場合、デフォルトの最大ヒープ・サイズは 16MB です。JDK 1.2 の場合、24MB です。

パフォーマンスを最大にするには、アプリケーション要件を満たす最大ヒープ・サイズを設定します。必要な Java ヒープを判断するには、ご使用のプログラムに `java.lang` パッケージの `Runtime.getRuntime().totalMemory()` および `Runtime.getRuntime().freeMemory` メソッドのコールを挿入します。合計メモリーからメモリーの空き容量を引きます。その差がアプリケーションが消費したヒープの量です。

128MB のヒープが必要だと判断したとします。ヒープ・サイズを変更するには、自動モードの場合、**jserv.properties** ファイルの最大 Java ヒープ・サイズを設定します。

```
wrapper.bin.parameters=-mx128m
```

手動モードの場合、複数の JServ プロセスが実行されているときは、ヒープ・サイズは各 JServ プロセスごとに、コマンド・ラインで設定する必要があります。

JServ プロセスが最大ヒープ・サイズを超えると、プロセスは終了します。自動モードの場合、新しいプロセスが開始されますが、パフォーマンスは著しく低下します。手動モードの場合、終了されたプロセスは再開されないため、ヒープ・サイズが十分な大きさであることを確認する必要があります。

注意： **top** または **ps** などのユーティリティのレポートに表示されるプロセス・サイズは、最大ヒープ・サイズより大きくなります。これは、プライベート・メモリーが最大ヒープ・サイズに足されるためです。

サーブレットおよび OracleJSP のメモリー要件

OracleJSP（サン・マイクロシステムズ社の JavaServer Pages の Oracle によるインプリメンテーション）およびサーブレットは、使用する JDK のバージョンにより、必要なメモリー量が異なります。次の表に、10 ～ 30 のアクティブ・スレッドを使用した処理における単純なサーブレットと OracleJSP のメモリー要件を比較します。サーブレットではセッションを使用しませんでした。OracleJSP ではセッションがオンでした（デフォルト）。

表 3-3 サーブレットと OracleJSP のメモリー

コンポーネント	JDK 1.1.8	JDK 1.2
サーブレット	10MB	24MB
OracleJSP	10MB	32MB

必要なメモリー量は、セッションが使用されているかどうかによって異なります。1 つのセッションあたり約 0.5MB 消費します。最大のパフォーマンスを得るためには、セッションを使用しない場合、次のようにして OracleJSP アプリケーションでセッションをオフにします。

```
<%@ page session="false" %>
<html><body>
HelloWorld
</body></html>
```

まず、アクティブなユーザーはそれぞれ Java アプリケーションについて 150K から 200K、加えてサーバー・プロセスのサイズを消費すると考えます。Java アプリケーションの場合、基本プロセスは約 12 ～ 15MB です。

アプリケーションのメモリーは、アプリケーションのサイズ、キャッシュされるデータの量、およびその他の要因によっても異なります。

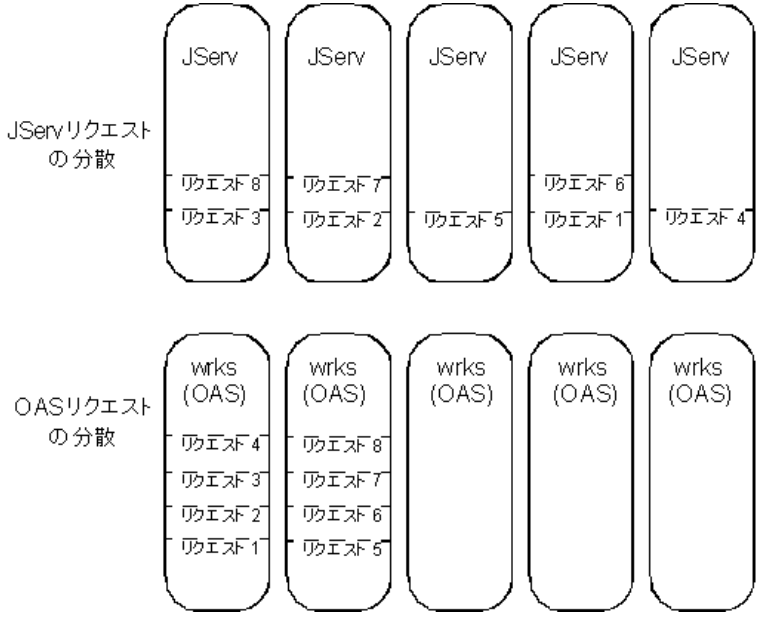
OracleJSP の詳細は、『Oracle8*i* JavaServer Pages 開発者ガイドおよびリファレンス』を参照してください。

JServ プロセスの数

オラクル社では、基本的に、1 つの CPU あたり 2 つの JServ プロセスをお薦めします。また、JServ 設定ファイルのデフォルトのスレッド設定（`security.maxConnections=50`）から検討することもできます。（設定ファイルでパラメータを変更する方法については、5-4 ページの「ロード・バランシング」を参照してください。）

ご使用のアプリケーション・コードで同期化を頻繁に行ったり、新しい Java オブジェクトを多数作成する場合、JServ プロセスの数を増やし、一方で 1 プロセスあたりのスレッド数を 10 から 20 に制限することを考慮する必要があります。これにより、JVM のオブジェクトの同期化に必要なキューイングや処理の増加を回避できます。httpd プロセス（mod_jserv）が着信リクエストを分散して JServ プロセスに送信するためです。使用可能な JServ エンジン間でリクエストを分散する方法については、5-4 ページの「ロード・バランシング」を参照してください。（Oracle Application Server を熟知している方は、あるサーブレット・エンジンのスレッドの制限に達するまでリクエストがそのサーブレット・エンジンに送信され、その後のリクエストは次のサーブレット・エンジンに送信されることをご存知でしょう。）

図 3-1 リクエストの分散



4

HTTP サーバーのパフォーマンスの最適化

この章では、TCP パラメータのチューニング、`MaxClients` パラメータの変更による効果、SSL のキャッシュおよびロギングなど、Oracle HTTP Server のパフォーマンス改善について説明します。

内容

TCP のチューニング

TCP パラメータを正しくチューニングすると、パフォーマンスが著しく改善されます。この項では、TCP チューニングの推奨事項と、各パラメータの簡単な説明を示します。TCP のチューニング全般の説明は、『Sun Performance and Tuning: Java and the Internet』（Adrian Cockcroft、Richard Pettit 著、1998 年、Sun Microsystems Press）を参照してください。

次の表に、推奨する TCP パラメータ設定を示します。

表 4-1 推奨する TCP パラメータ設定

パラメータ	設定	コメント
tcp_conn_hash_size	32768	4-3 ページの「TCP 接続テーブルのアクセス速度の増加」を参照。
tcp_close_wait_interval	60000	Solaris 2.6 のパラメータ名。4-3 ページの「接続テーブルのエントリの保存期間の指定」を参照。
tcp_time_wait_interval	60000	Solaris 2.7 のパラメータ名。4-3 ページの「接続テーブルのエントリの保存期間の指定」を参照。
tcp_conn_req_max_q	1024	4-4 ページの「ハンドシェークのキューの長さの増加」を参照。
tcp_conn_req_max_q0	1024	4-4 ページの「ハンドシェークのキューの長さの増加」を参照。
tcp_slow_start_initial	2	4-4 ページの「データ転送率の変更」を参照。
tcp_xmit_hiwat	32768	4-5 ページの「データ転送ウィンドウ・サイズの変更」を参照。
tcp_recv_hiwat	32768	4-5 ページの「データ転送ウィンドウ・サイズの変更」を参照。

TCP パラメータの設定

接続テーブルのハッシュ・パラメータを設定するには、次の行を /etc/system ファイルに追加し、その後システムを再起動します。

```
set tcp:tcp_conn_hash_size=32768
```

TCP パラメータをここで推奨する値に変更するサンプル・スクリプト **tcpset.sh** が、**$ORACLE_HOME/Apache/Apache/bin/** ディレクトリに入っています。

スクリプトの実行後にシステムが再起動された場合、デフォルトの設定が復元され、もう一度スクリプトを実行する必要があります。設定を永続的にするには、ご使用のシステムの起動ファイルに設定を入力します。

TCP 接続テーブルのアクセス速度の増加

ユーザー数が多い場合、TCP 接続テーブルのハッシュ・サイズを増加する必要があります。ハッシュ・サイズは、接続データの格納に使用されるハッシュ・バケツの数です。バケツがいっぱいになると、接続が見つかるまでに時間がかかります。ハッシュ・サイズを増加すると接続を探す時間が減少しますが、消費メモリーが増加します。

ご使用のシステムで 1 秒あたり 100 の接続を実行するとします。tcp_close_wait_interval を 60000 に設定すると、常に TCP 接続テーブルに約 6000 のエントリが存在します。ハッシュ・サイズを 2048 または 4096 に増やすと、パフォーマンスが著しく改善されます。

1 秒あたり 300 の接続を処理するシステムの場合、ハッシュ・サイズをデフォルトの 256 から接続テーブルのエントリ数に近い数値に変更すると、往復に要する平均時間が 3 から 4 秒減少します。ハッシュ・サイズの最大値は 262144 です。必要に応じてメモリーを必ず増加してください。

tcp_conn_hash_size を設定するには、次の行を **/etc/system** ファイルに追加します。パラメータは、システムの再起動後に有効になります。

```
set tcp:tcp_conn_hash_size=32768
```

接続テーブルのエントリの保存期間の指定

TCP 接続テーブルには、接続に関連付けられたデータが保持されます。サーバーは接続のクローズ後も、その後にクライアントから着信したパケットを識別し正しく破棄できるよう、一定期間 TCP 接続テーブルのエントリを維持します。

このテーブルへのアクセス速度はパフォーマンスに影響を与えます。アクセス速度は、テーブル内のエントリ数とそのハッシュ・サイズによって異なります。テーブル内のエントリ数は、着信リクエストの割合と、各接続の存続期間によって異なります。

TCP 接続テーブルのエントリが保管される期間は、tcp_close_wait_interval パラメータ（Solaris 2.7 では tcp_time_wait_interval に名前が変更された）によって制御できます。このパラメータは、通常 60,000ms に設定されています。このパラメータは、次のコマンドを使用して設定できます（Solaris 2.6 と 2.7 ではパラメータ名が異なるため注意してください）。

Solaris 2.6 の場合

```
prompt>/usr/sbin/ndd -set /dev/tcp tcp_close_wait_interval 60000
```

Solaris 2.7 の場合

```
prompt>/usr/sbin/ndd -set /dev/tcp tcp_time_wait_interval 60000
```

注意： ユーザー数に大きなばらつきがある場合（インターネットのトポロジにより）、このパラメータをこれより大きな値にします。TCP 接続テーブルへのアクセス時間は、tcp_conn_hash_size パラメータによって改善できます。

ハンドシェークのキューの長さの増加

TCP 接続のハンドシェーク中、サーバーは、クライアントからリクエスト（SYN）を受信した後、応答を送信し、クライアントから応答が返ってくるのを待ちます。クライアントがサーバーのメッセージに応答すると、ハンドシェークは完了します。クライアントから最初のリクエストを受信すると、サーバーはリスニング・キューにエントリを作成します。クライアントがサーバーのメッセージに応答すると、完了したハンドシェークのメッセージのキューに移動されます。2 番目のキューにより、サーバーはハンドシェークが完了したリクエストの処理を継続できます。

完了していないハンドシェークのキューの最大長は、`tcp_conn_req_max_q0` によって決定されます。デフォルトは 1024 です。完了したハンドシェークのリクエストのキューの最大長は、`tcp_conn_req_max_q` で定義されます（デフォルトは 128 です）。

ほとんどの Web サーバーではデフォルトで十分ですが、同時ユーザーが 1025 以上の場合、これらの設定では低すぎる可能性があります。その場合、キューがいっぱいであるため、接続はハンドシェークの段階で排除されます。この問題がご使用のシステムによるものであるかどうかを判断するには、`netstat -s` を使用して、`tcpListenDrop`、`tcpListenDropQ0` および `tcpHalfOpenDrop` の値を調べます。最初の 2 つの値がゼロではない場合、最大値を増加する必要があります。

おそらくデフォルト値で十分ですが、オラクル社では tcp_conn_req_max_q の値を 1024 に増やすことをお薦めします。これらのパラメータを設定するには、次のコマンドを使用します。

```
prompt>/usr/sbin/ndd -set /dev/tcp tcp_conn_req_max_q 1024
prompt>/usr/sbin/ndd -set /dev/tcp tcp_conn_req_max_q0 1024
```

データ転送率の変更

通常、データ転送時のパケットはすべて一度に送信されます。TCP では、インターネットの混雑したセグメントに負荷がかかり過ぎないよう、低速でデータ転送を開始します。低速で開始する場合、1 つのパケットが送信され、確認が受信された後に 2 つのパケットが送信されます。サーバーに送信される数は、確認を受信するたびに倍になり、TCP 転送ウィンドウの制限に達するまで増加します。

Microsoft Windows の一部のバージョン（NT 4.0 および 95 を含む）では、接続の開始時に 1 つのパケットを受け取っても確認を送信しません。しかし、2 つのパケットを受信すると、すぐに確認が送信されます。Solaris では接続の開始時に 1 つのパケットのみ送信するため（TCP の標準に準拠して）、接続開始時間が長くなる可能性があります。これは、レイテンシが短いはずの高速のローカル・ネットワークの場合に特に顕著になります。

データ転送の開始時に、Solaris が 2 つのパケットを送信するよう設定することが可能です。

```
prompt>/usr/sbin/ndd -set /dev/tcp tcp_slow_start_initial 2
```

データ転送ウィンドウ・サイズの変更

データの送受信に使用する TCP 転送ウィンドウのサイズにより、確認を待たずに送信できるデータ量が決まります。デフォルトのウィンドウ・サイズは 8192 バイトです。ご使用のシステムのメモリーに制約がある場合以外は、これらのウィンドウの値を最大値の 32768 に増やします。これにより、大きなデータの転送速度が著しく向上します。ウィンドウを大きくするには、次のコマンドを使用します。

```
prompt>/usr/sbin/ndd -set /dev/tcp tcp_xmit_hiwat 32768

prompt>/usr/sbin/ndd -set /dev/tcp tcp_recv_hiwat 32768
```

通常クライアントは大量のデータを受信するため、エンド・ユーザーのシステムの TCP 受信ウィンドウを大きくすると効果的です。

MaxClients

`MaxClients` ディレクティブは、Web サーバーに同時に接続できるクライアント数、ひいては httpd プロセスの数を制限します。Oracle9*i* Application Server リリース 1.0.2 では、**httpd.conf** ファイルのこのパラメータを最大 1024 に設定できます（前バージョンでは最大値は 256 でした）。デフォルトは 150 で、ほとんどの用途ではこれが適切な値です。`MaxClients` 設定が低すぎ、限界に達すると、クライアントは接続できません。

100Mbps のネットワーク上で 2 プロセッサの 168MHz Sun UltraSparc を使用して静的ページのリクエスト（平均サイズ 20K）でテストしたところ、次の結果が得られました。

- `MaxClients` のデフォルト設定 150 は、ネットワークの飽和に十分だった。
- 300 ユーザーのサポートには、約 60 の httpd プロセスが必要だった（思考時間なし）。

前述のシステム、さらに 4 および 6 プロセッサの 336MHz システムでは、`MaxClients` 設定を 150 から 256 に増加しても、顕著なパフォーマンスの向上はみられませんでした。これは、最高 1000 ユーザーで静的ページとサーブレットを使用したテストの結果です。

システム・リソースが飽和状態のときに `MaxClients` を増加しても、パフォーマンスは改善されません。使用可能な httpd プロセスがない場合、接続リクエストはプロセスが使用可能になるまで TCP/IP システムのキューに入れられ、しばらくするとクライアントにより接続が終了されます。

注意： 永続的な接続を使用する場合、さらに多くの同時 httpd サーバー・プロセスが必要な場合があります。永続的な接続とサーバー・プロセス数の関係については、4-9 ページの「httpd プロセスの可用性」を参照してください。

動的リクエストの場合、システムの負荷が大きければ、リクエストをネットワークでキューに入れる方がよいことがあります（これにより、システムの負荷が適度な状態になります）。システム管理者は、応答時間の長さよりタイムアウト・エラーと再試行の方がよいかどうかを検討する必要があります。この場合、MaxClients 設定を小さくし、サーバー上の同時リクエスト数のスロットルの役割を担うようにすることが可能です。

SSL セッション・キャッシュ

Oracle HTTP Server は、デフォルトで、クライアントの SSL セッション情報をキャッシュに入れます。セッション・キャッシュを使用すると、レイテンシが長いのはサーバーへの最初の接続時のみになります。たとえば、SSL 対応のサーバーで接続と切断の簡単なテストを行ったところ、SSL セッション・キャッシュを使用しない場合、5 回の接続に要した時間は 11.4 秒でした。SSL セッション・キャッシュを使用可能にした場合、5 回の往復に要した時間は 1.9 秒でした。

httpd.conf の SSLSessionCacheTimeout ディレクティブにより、サーバーがセッションを維持する期間が決まります（デフォルトは 300 秒です）。セッション情報はファイルに保管されます。SSLSessionCache ディレクティブを使用してセッション情報の保管場所を指定できます。デフォルトは、**$ORACLE_HOME/Apache/Apache/logs/** ディレクトリです。このファイルは、複数の Oracle HTTP Server プロセスで使用可能です。

SSL セッションの存続期間は、HTTP の永続的な接続の使用とは関係ありません。

ロギングの影響

この項では、ロギングのタイプ、ログ・レベル、さらにこれらの使用によるパフォーマンスの影響について説明します。

アクセス・ロギング

静的ページのリクエストの場合、デフォルト・フィールドのアクセス・ロギングにより、パフォーマンス・コストが 2 ～ 3% 増加します。

HostNameLookups

デフォルトでは、HostNameLookups ディレクティブはオフに設定されています。サーバーにより、着信リクエストの IP アドレスがログ・ファイルに書き込まれます。HostNameLookups がオンに設定されると、サーバーは各リクエストの IP アドレスに関連付けられたホスト名をインターネット上の DNS システムに問い合せ、ホスト名をログに書き込みます。

オラクル社内テストで HostNameLookups をオンに設定したところ、パフォーマンスは約 3%（最良の場合）低下しました。サーバーの負荷とご使用の DNS サーバーへのネットワーク接続性により、DNS 検索のパフォーマンス・コストが高くなる可能性があります。リアルタイムでログにホスト名を含める必要がある場合以外は、IP アドレスをログに記録することをお薦めします。**logresolve** ユーティリティ（**$ORACLE_HOME/Apache/Apache/bin/** ディレクトリ内に存在）を使用すると、IP アドレスをオフラインでホスト名に解決できます。

詳細は、次の URL の Dale Gaudet 著『Apache Performance Notes』を参照してください。

http://www.apache.org/docs/misc/perf-tuning.html

エラーのロギング

サーバーにより、エラー・ログに異常なアクティビティが記録されます。`ErrorLog` および `LogLevel` ディレクティブにより、ログ・ファイルと、記録されるメッセージの詳細レベルを指定します。デフォルトのレベルは `warn` です。負荷のかかったシステムにおいて、静的ページのパフォーマンスは、`warn`、`info` および `debug` レベルで違いはありませんでした。

LogLevel ディレクティブの詳細は、次の URL を参照してください。

http://www.apache.org/docs/mod/core.html#loglevel

HTTP/1.1

Oracle HTTP Server では、HTTP/1.1 が使用可能です。Netscape Navigator 4.0 では HTTP/1.0 を使用しており、永続的な接続などの一部の 1.1 の機能を使用しています。Internet Explorer では HTTP/1.1 を使用しています。永続的な接続を使用すると、接続の確立と終了を繰り返すために発生する（1 つのリクエストにつき 1 回の接続）オーバーヘッドの削減により、パフォーマンス上での利点が得られます。永続的な接続では、1 ユーザーからの複数のリクエストを受け付けます。

小さな静的ページ・リクエストの場合、接続のレイテンシは応答のレイテンシ（接続確立後にリクエストを完了するまでの時間）と同じかまたはそれより大きくなります。そのため、永続的な接続を使用すると、パフォーマンスが大きく改善されます。

パフォーマンスと HTTP/1.1 プロトコルに関する詳細は、次の URL を参照してください。

http://www.w3.org/Protocols/HTTP/Performance/Pipeline.html

永続的な接続

ユーザーのブラウザが永続的な接続をサポートしている場合（HTTP/1.1 のデフォルトの動作）、Apache の `KeepAlive` ディレクティブを使用して、サーバー上で永続的な接続をサポート可能です。（HTTP/1.1 の全機能をサポートしていないブラウザでも、Netscape の最近のバージョンなど、一部のブラウザでは永続的な接続をサポートしています。）

応答時間の短縮

永続的な接続を使用すると、接続の確立による遅延が 1 回しか発生しないため、複数の HTTP リクエストが含まれる Web 対話処理における合計応答時間を改善できます。

永続的な接続を使用しない場合に、3 つのイメージを持つ Web ページをクライアントがサーバーから取得するために要する合計時間を見てみます。

アクティビティ	秒
接続の確立	1
ページのテキスト部分の生成と送信	5
接続の確立	1
最初のイメージ・ファイルの転送	2
接続の確立	1
2 番目のイメージ・ファイルの転送	2
接続の確立	1
3 番目のイメージ・ファイルの転送	2
合計	15

永続的な接続を使用すると、同じリクエストの応答時間が次のように短縮されます。

アクティビティ	秒
接続の確立	1
ページのテキスト部分の生成と送信	5
最初のイメージ・ファイルの転送	2
2 番目のイメージ・ファイルの転送	2
3 番目のイメージ・ファイルの転送	2
合計	12

これは、サービス時間の 20% の減少です。システムに負荷がかかっている場合、永続的な接続によって接続時間が削減されると、これに対応して TCP キューが削減され、さらに利益が大きくなります。

サーバーの作業負荷の減少

永続的な接続がもたらすもう 1 つの利点は、サーバーにかかる作業負荷の減少です。サーバーはクライアントとの接続を確立する作業を繰り返す必要がないため、かわりに他の作業を行うことができます。非常にコストが低いサーブレット（Hello World）で、同じクライアントが 1 回の接続あたり 4 つのリクエストを行った場合、1 リクエストあたりの CPU 時間（ms）は約 10% 減少しました。（これより多くの作業を行う実際のサーブレット・アプリケーションでは、効果はこれよりもかなり小さくなります。）

httpd プロセスの可用性

Apache で永続的な接続を使用する場合、いくつか重大な欠点があります。特に、httpd プロセスはシングル・スレッドであるため、1 つのクライアントによりプロセスが一定の期間拘束されることがあります（期間の長さは、KeepAlive 設定によって異なります）。ユーザー数が多く、KeepAlive の制限を高くしすぎると、httpd デーモンの不足により、クライアントが拒否される可能性があります。

KeepAlive ディレクティブのデフォルトの設定は、次のとおりです。

```
KeepAlive on
MaxKeepAliveRequests 100
KeepAliveTimeOut 15
```

これらの設定により、永続的な接続の欠点を最小限に抑えながら、利点を活用できるように、接続あたりのリクエストおよびリクエスト間の時間が設定されています。ご使用のシステムでこれらの値を設定する際は、ご使用のシステムのユーザー数と作業内容を考慮する必要があります。たとえば、ユーザー数が多く、ユーザーが送信するリクエストが小規模で頻度が低い場合は、前述の設定の値を小さくするか、または KeepAlive をオフに設定します。ユーザー数が少なく、サイトに頻繁にアクセスする場合は、設定値を大きくします。

FIN_WAIT_2

一部のブラウザでは、サーバーの TCP 接続を FIN_WAIT_2 の状態にするという問題が判明しています。この状態の接続が多くなると、システムは TCP 接続の保存に割り当てているメモリーが不足し、停止します。

この問題は、接続がアイドル状態になり、キープ・アライブ制限時間を超えたためにサーバーが接続を切断したときに、クライアント・ホストが接続の切断の完了に必要なステップを実行しないことがある、というものです。ホストは、切断のリクエストを送信した後、接続が FIN_WAIT_2 の状態のままになり、適切なパケットがクライアントから返ってくるか、もしくは内部フラッシュが発生するまでメモリーを使用し続けます。接続が FIN_WAIT_2 の状態のままになった場合、その接続が関連付けられている httpd プロセスは指示どおりに他のリクエスト処理用に解放されるため、この問題では Web サーバー・プロセスは拘束されません。

Solaris では、tcp_fin_wait_2_flush_interval パラメータで、これらの接続がクリーン・アップされる頻度を指定します。通常はデフォルトの設定で十分です。システムが正しく動作しない場合以外は変更しないでください。FIN_WAIT_2 の詳細は、次の URL を参照してください。

http://apache.put.poznan.pl/misc/fin_wait_2.html

注意： FIN_WAIT_2 の状態は、KeepAlive の使用とは無関係のシステムのバグによって発生することもあります。このバグは、Solaris のクラスタ・パッチ 105181-20 で修正されています。

Apache のバージョン

Apache のバージョン 1.3.9 と 1.3.12 の違いは、主にバグの修正です。静的ページとサーブレットのパフォーマンス測定では、これらのバージョン間でパフォーマンスの違いは測定されませんでした。

5

Apache JServ の最適化

この章では、JServ のアーキテクチャと、パフォーマンスの改善方法について説明します。また、OracleJSP（Oracle によるサン・マイクロシステムズ社の JavaServer Pages 1.1 のインプリメンテーション）に関するパフォーマンス情報についても説明します。

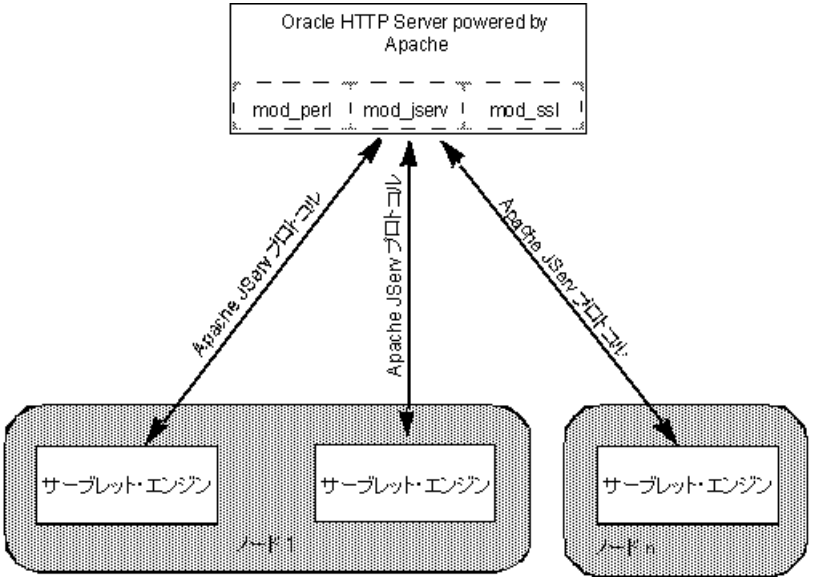
内容

JServ の概要

Apache JServ は、httpd プロセスで実行される **mod_jserv** という Apache モジュールと、Java プロセスで実行されるサーブレット・エンジンで構成されています。**mod_jserv** は C でインプリメントされており、ディスパッチャとして機能し、各サーブレット・リクエストの実行を JServ プロセスにルーティングします。

サーブレット・エンジンは専用の JVM で実行され、リクエストの解析とレスポンスの生成のみ行います。図 5-1 に示すように、複数の JServ でリクエストを処理可能です。HTTP サーバー・プロセスと JServ プロセスは、Apache JServ Protocol 1.2 を使用して通信します。

図 5-1 Apache JServ のコンポーネント



サーブレットのパフォーマンスの最適化

この項では、JServ のパフォーマンスの最適化の方法として、JVM の開始時のサーブレットのロードと、ロード・バランシングについて説明します。

この説明では、「リポジトリ」と「ゾーン」という用語を使用します。サーブレット、リポジトリおよびゾーンは、それぞれファイル、ディレクトリおよび仮想ホストに似ています。サーブレットは 1 つの単位で、リポジトリはサーブレットの集合、ゾーンはリポジトリの集合です。

サーブレット・クラスのロード

Apache JServ では、JVM の起動時にサーブレット・クラスをロードすることが可能です。これを行うには、ロードするサーブレットを、サーブレット・ゾーンのプロパティ・ファイル内の servlets.startup ディレクティブに含めます。サーブレットのロード時に、そのサーブレットの init() メソッドがコールされます。他のすべてのサーブレット（servlets.startup のリストに含まれていないもの）は最初のリクエスト時にロードおよび初期化されます。

この機能を使用すると、JServ プロセスの起動時間は長くなりますが、サーブレットの最初のリクエストのレイテンシが改善されます。

JSP を使用する場合の事前ロード

サーブレットに JSP を使用している（コードが HttpServlet を拡張しない）場合、この事前ロード・オプションは使用できませんが、oracle.jsp.jspServlet を servlets.startup に含めることにより、JSP ランナーを事前にロードすることが可能です。

初期化ルーチンの最初のリクエストのレイテンシがパフォーマンス上の大きな問題である場合、ダミーのサーブレットを作成し、1 回だけ実行される初期化ルーチンを init() メソッドでコールすると、若干前述のような効果が得られます。この場合、ダミーのサーブレットの名前を servlets.startup に追加する必要があります。

クラスの自動リロード

あるゾーンで autoreload.classes が True に設定されている場合（デフォルト）、そのゾーンのサーブレットがリクエストされるたび、そのゾーン内のリポジトリからロードされたクラスはすべて変更されているかどうかチェックされます。クラスのいずれかが変更されている場合、そのゾーンのリポジトリから事前にロードされたクラスはすべてアンロードされます。そして、必要に応じて、クラスはクラス・ファイルから再びロードされます。

この機能を使用すると、サーバーを再起動せずに、新しいバージョンをインストールしたり新しいクラス・ファイルを追加できるため、開発時には便利です。ただし、本番環境で最適なパフォーマンスを得るためには、自動クラス・リロード・パラメータを両方とも False に設定します。サーブレットの実行のたびにリポジトリをチェックすると、パフォーマンス・コストがかかるためです。ゾーン・プロパティ・ファイルの次のパラメータを変更します。

```
autoreload.classes=false
autoreload.file=false
```

ロード・バランシング

サーブレット・アプリケーションの負荷を複数の JServ プロセスに分散すると効果的な場合がよくあります。特に、アプリケーションがマルチプロセッサで実行されている場合、またはサーブレットと HTTP サーバーが別のノードで実行されている場合です。複数の Apache JServ プロセスを実行すると、単一のプロセッサ・ホスト上であっても、通常はより高いスループットとより短い応答時間が実現されます。（具体的な推奨事項については、第 3 章「サイズ設定および構成」を参照してください。）

この項では、HTTP サーバーと同じホストで実行されている 2 つの JServ プロセス間で着信リクエストを均衡させる方法について説明します。手順とともに **jserv.properties** ファイルの例が示されています。必要に応じて、ご使用のポート番号およびディレクトリの場所に置き換えてください。

JServ は、複数の JServ プロセスを自動的に起動および停止できないため、ロード・バランシングを使用する場合は、プロセスを手動で起動および停止する必要があります。（JServ プロセスおよび Oracle HTTP Server を起動および停止するサンプル・スクリプトは、**$ORACLE_HOME/Apache/Apache/bin/ ディレクトリ**に含まれています。）このため、何らかの理由でプロセスが終了すると、JServ はそのプロセスを再起動しません。メモリー不足によるプロセスの終了を防ぐために、JServ プロセスで最大ヒープ・サイズが十分な大きさに設定されていることを確認してください。3-6 ページの「Java のヒープ・サイズの決定」を参照してください。

JServ プロセスの設定

ご使用のロード・バランシング方式で、各 JServ プロセスは、それぞれ専用のポートをリスニングし、専用のファイルにログを記録するよう設定されている必要があります。アプリケーションの実行に必要なパラメータが含まれている **jserv.properties** ファイルが存在する場合、それを複製して各 JServ プロセス用にプロパティ・ファイルを作成できます。

1. 各 JServ プロセス用にプロパティ・ファイルを作成します。

```
prompt>cp jserv.properties jserv1.properties
prompt>cp jserv.properties jserv2.properties
```

2. 次のように、**jserv1.properties** を編集します。

```
port=8001
log.file=/usr/local/jserv/logs/jserv1.log
```

3. 次のように、**jserv2.properties** を編集します。

```
port=8002
log.file=/usr/local/jserv/logs/jserv2.log
```

注意： ご使用の HTTP サーバーが JServ プロセスとは別のホストで稼動している場合、HTTP サーバーを実行しているホストの IP アドレスも、各 **jserv.properties** ファイル内の `security.allowedAddresses` パラメータに追加する必要があります。

JServ がご使用の CLASSPATH に含まれている場合、次のコマンドを使用して JServ プロセスを起動できます。

```
java JServ jserv1.properties
java JServ jserv2.properties
```

プロセスおよび Web サーバーの起動と終了には、スクリプトを使用すると便利です。**$ORACLE_HOME/Apache/Apache/bin/** ディレクトリにサンプルが含まれています（**startJServ.sh** と **stopJServ.sh**）。

負荷分散のための jserv.conf の変更

1. プロセスを手動で起動するようにフラグを設定します。

   ```
   ApJServManual on
   ```

2. サーブレット・リクエストの送信先を指定します。

   a. ApJServMount ディレクティブに場所を指定します。

      ```
      ApJServMount /servlets /root
      ```

      ユーザーが **http://*your.server.com*/servlets/testServlet** をリクエストすると、前述の ApJServMount ディレクティブは、/root というゾーン内の testServlet を実行します。

   b. ゾーン識別子を **/root** から **balance://set/root** に変更し、その後負荷を共有するプロセスの指定に必要なディレクティブを追加します。

      ```
      ApJServMount /servlets balance://JServ_set/root
      ApJServBalance JServ_set JServ1
      ApJServBalance JServ_set JServ2 2
      ApJServHost JServ1 ajpv12://127.0.0.1:8001
      ApJServHost JServ2 ajpv12://127.0.0.1:8002
      ApJServRoute JS1 JServ1
      ApJServRoute JS2 JServ2
      ApJServShmFile /usr/local/apache/logs/jserv_shm
      ```

      - これにより、`ApJServMount` ディレクティブは `/servlets balance://set/root` を使用して、**/servlets** 内のサーブレットのリクエストを JServ1 と JServ2 に分散します。

- ApJServBalance ディレクティブで、JServ1 と JServ2 を、負荷を共有するプロセスとして指定します。JServ2 の後の '2' は比率の値です。ここでは、指定していない場合の 2 倍のリクエストが JServ2 に送信されることを指定します。つまり、JServ2 は着信リクエストの約 2/3 を受信します。詳細は、次の「JServ リクエストの分散」を参照してください。
- ApJServHost ディレクティブで、プロセスがリスニングしているホストとポートを指定します。
- ApJServRoute ディレクティブで、JServ プロセスをセッションに関連付けます。JServ は、この情報を使用して、セッションのすべてのリクエストを 1 つのプロセスに統合します。JServ セッション・メカニズムにより、プロセスのルーティング情報がユーザーに送り返されます（通常は Cookie を使用します）。ご使用のアプリケーションでセッションが使用されている場合のみ変更が必要です。
- ApJServShmFile ディレクティブで、httpd プロセスが JServ プロセスの状態の追跡に使用する共有メモリー・ファイルを指定します。

JServ リクエストの分散

mod_jserv は、次の手順でリクエスト処理用の JServ エンジンを選択します。

1. httpd プロセスが起動されます。
2. **mod_jserv** により、使用可能な JServ のリストが作成されます。このとき、比率の値が 1 より大きい JServ には余分なエントリが作成されます（たとえば、前述の例における、ApJServBalance set JServ2 2 で指定された JServ2 など）。
3. httpd デーモンがサーブレット・リクエストを受信し、それを **mod_jserv** に渡します。
4. **mod_jserv** は、リクエストを処理する JServ エンジンを選択します。
   a. **mod_jserv** は、リクエストが現行セッションの一部であるかどうかを調べます。現行セッションの一部である場合、ApJServRoute ディレクティブを使用して、そのセッションの他のリクエストを処理した JServ を探します。
   b. リクエストがセッションの一部ではない場合、**mod_jserv** は、httpd プロセスのプロセス ID と、使用可能な JServ のリストのエントリ数に基づき、エンジンを選択します。次のようになります。

```
JServ_id to handle the request = httpd_pid % number of JServs in
the list
```

この方法で、使用可能な JServ エンジンにリクエストがほぼ均等に分散されます。

シングル・スレッド・モデルのサーブレットの使用

オラクル社では、作成するサーブレットに SingleThreadModel（STM）インタフェースをインプリメントすることをお薦めします。STM インタフェースをインプリメントするように変更されたアプリケーションでは、応答時間が 25% 改善されました。これは同期化のボトルネックの削減によるものと考えられます。

また、STM サーブレットを使用すると、データベース接続の管理が大変容易になります。データベース接続をサーブレットの `init()` メソッドで開始し、`destroy()` メソッドでクローズすることが可能です。サーブレットの `doGet()` または `service()` メソッドの実行時には、データベース接続の取得を考慮する必要はありません。かわりに、JDBC 接続キャッシュを使用できます。

zone.properties ファイルには、STM サーブレットのパフォーマンスに特に影響を与えるパラメータが 3 つ存在します。これらにより、次のことが決定されます。

- サーブレット・クラスのロード後に生成され使用可能になるサーブレット・オブジェクト・インスタンスの最小数
- 生成可能な最大数
- 使用可能なインスタンスが不足している場合に生成される数

システムの実行中にインスタンスを生成すると非常にコストがかかるため、オラクル社では、最小値と最大値を同じ値に設定することをお薦めします。最適な値は、ご使用のデータベース・サーバーで処理可能な接続数によって多少異なります。これは、次のようにして、複数の JServ プロセスに分割する必要があります。

DB の合計接続数 / JServ プロセス数 = 1 プロセスあたりの STM サーブレット・インスタンス数

ご使用のアプリケーションに適した JServ プロセス数の決定方法については第 3 章「サイズ設定および構成」、その設定方法については 5-4 ページの「ロード・バランシング」を参照してください。1 プロセスあたり 10 のサーブレット・インスタンスを使用すると決めたとします。その場合、ゾーンのプロパティ・ファイルで、次の設定を行います。

```
singleThreadModelServlet.initialCapacity = 10
singleThreadModelServlet.incrementCapacity = 0
singleThreadModelServlet.maximumCapacity = 10
```

警告： ゾーン・プロパティ・ファイルの `singleThreadModelServlet.maximumCapacity` の値は、少なくとも jserv.properties ファイルの security.maxConnections の値と同じである必要があります。そのように設定されておらず、JServ プロセスに送信されたリクエスト数が最大許容量を超えた場合、リクエストは失敗します。

OracleJSP とは

OracleJSP 1.1.0.0 は、サン・マイクロシステムズ社の JavaServer Pages 1.1 仕様の Oracle によるインプリメンテーションです。追加機能には、Oracle データベースにアクセスするためのカスタム JavaBeans、SQL サポートおよび拡張データ型などがあります。これらの機能の詳細は、Oracle9*i* Application Server のドキュメント『Oracle9*i* Application Server 概要』を参照してください。

OracleJSP のパフォーマンス・チューニング

この項では、Oracle JSP のパフォーマンスの改善方法について説明します。

セッション管理の影響

一般的に、セッションを使用するとパフォーマンス・オーバーヘッドが追加され、常駐メモリーが約 0.5MB 消費されます。各リクエストごとに新しいセッションを作成しない場合は、セッションをオフにする必要があります。OracleJSP では、デフォルトでセッションは使用可能になっています。このため、セッションを使用しない場合は、ページの先頭に次の行を追加して、オフに設定します。

```
<%@ page session="false" %>
```

セッションを使用する場合は、明示的にセッションをクローズする必要があります。そうしないと、タイムアウトになるまで、セッションは存続し続けます（セッション・タイムアウトのデフォルト値は 30 分です）。セッションを手動でクローズするには、`session.invalidate()` メソッドを使用します。

OracleJSP の設定方法に関する詳細は、『Oracle8*i* JavaServer Pages 開発者ガイドおよびリファレンス』を参照してください。

開発者モード

パフォーマンスに大きな影響を与えるもう 1 つのパラメータは、開発モードです。開発中のデバッグには便利な機能ですが、パフォーマンスを低下させます。デフォルト値は True なので、次のように **jserv.properties** ファイルで False に設定する必要があります。

```
servlet.oracle.jsp.JspServlet.initArgs=developer_mode=false
```

開発モードが True に設定されていると、OracleJSP およびサーブレット・エンジンは、すべてのリクエストを調べ、そのページまたはアプリケーションをリロードまたは再変換するかどうかを決定します。開発モードがオフになっていると、最初のリクエストのみ調べられます。

JDK 1.2 を使用して 50 ユーザー、128MB ヒープおよびデフォルトの TCP 設定でテストしたところ、開発モードをオフにした場合、スループットで 14%、平均応答時間で 28%、パフォーマンスが向上しました。

バッファリング

OracleJSP で、バッファのリセットが必要な機能（たとえばエラー・ページ、contextType 設定、転送など）が使用されていない場合、JSP ページ・バッファを使用不可にすると、パフォーマンスが改善されます。バッファの作成にメモリーが使用されず、出力が直接ブラウザに送られるためです。バッファを使用不可にするには、次のページ・ディレクティブを使用します。

```
<%@ page buffer="none" %>
```

OracleJSP のバッファのデフォルト・サイズは 8KB です。

OracleJSP のパフォーマンスの向上

『Oracle8*i* JavaServer Pages 開発者ガイドおよびリファレンス』では、次に示すような、OracleJSP、インプリメンテーションのガイドライン、設定に関する問題、およびパフォーマンス上のヒントなどについて詳細に説明されています。

データベース接続のキャッシュ

データベース接続の作成には非常にコストがかかるため、接続のキャッシュを使用したほうがパフォーマンスが改善されます。これにより、OracleJSP アプリケーションは、データベース接続のプールから接続を取得し、完了後にはプールに戻すことが可能です。

更新文のバッチ

JDBC ドライバは、実行リクエストをいくつか蓄積し（バッチ値）、それらの処理をまとめてデータベースに渡します。バッチ値を設定することにより、この処理の実行頻度を制御できます。

JDBC 文のキャッシュ

繰り返し使用される実行文はキャッシュに入れると、再解析、文オブジェクトの再作成およびパラメータ・サイズ定義の再計算が回避されます。

行の事前フェッチ

問合せ時に、クライアントに複数の行を事前にフェッチすると、データベースとサーバー間の往復回数が削減されます。

データベースの行セットのキャッシュ

頻繁にアクセスされ、あまり変更されないデータの小さなセットをキャッシュします。大きなデータ・セットの場合、消費メモリーが大きいため、この方法はあまり効果がありません。

静的インクルードの使用

静的インクルードを起動するには、次のページ・ディレクティブを使用します。

```
<%@ include file="/jsp/filename.jsp" %>
```

静的インクルードにより、JSP のファイルのコピーが作成されるため、ページ・サイズに影響します。これは、リクエスト・ディスパッチャとの間の往復回数を減らすために役立ちます（動的インクルードの場合は毎回リクエスト・ディスパッチャを通す必要があるため、効果はありません）。ただし、生成されたページのインプリメンテーション・クラスのサービス・メソッドの制限である 64K を超えないよう、ファイル・サイズを小さくする必要があります。

動的インクルード

動的インクルードを起動するには、次のページ・ディレクティブを使用します。

```
<jsp:include page="/jsp/filename.jsp" flush="true" />
```

このディレクティブはファンクション・コールに似ており、JSP のページ・サイズを増やしません。ただし、動的インクルードの場合、リクエスト・ディスパッチャを通す必要があるため、処理のオーバーヘッドが増大します。動的インクルードは、ページ・サイズを増やさずに他のページを挿入する場合に便利です。

索引

Z

あ

お

か

き

さ

し

す

せ

そ

た