Java ランタイムの監視: 第 3 回 アプリケーションのエコシステムのパフォーマンスと可用性を監視する

ホスト、データベース、メッセージングの監視、そしてパフォーマンス・データの管理と視覚化

Nicholas Whitehead
Senior Technology Architect
ADP

2008年 8月 12日

Java™ アプリケーションのランタイムの監視に関する 3 回連載の最終回では、アプリケーションがサポートしているサービスやアプリケーションの従属サービスのパフォーマンスと可用性を監視するためのストラテジーと手法に目を向けます。この場合の監視対象に含まれるのは、アプリケーション実行のベースとなるホスト・オペレーティング・システム、運用データベース、メッセージング・インフラストラクチャーです。最後に、パフォーマンス・データを管理する上での問題、そしてデータのレポートおよび視覚化について説明して記事を締めくくります。

このシリーズの他の記事を見る

この 3 回連載の第 1 回と第 2 回で紹介した Java アプリケーションの監視パターンと手法は、いずれも JVM およびアプリケーション・クラスを焦点としたものです。この最終回の記事では焦点を広げ、アプリケーション実行のベースとなるオペレーティング・システムや、ネットワーク、アプリケーションを支える運用データベースといったアプリケーションの依存関係からパフォーマンスおよび可用性に関するデータを収集する手法を紹介します。最後に、収集したデータの管理パターン、そしてデータのレポートおよび視覚化方法を取り上げて連載を締めくくります。

Spring ベースのコレクターについて

第 2 回では監視サービスを管理する基本的な Spring ベースのコンポーネント・モデルを実装しました。このモデルを使用する根拠とその利点は以下のとおりです。

- XML ベースで構成すると、複雑なパフォーマンス・データ・コレクターを構成する際に必要となる大量のパラメーター・セットが管理しやすくなります。
- 関心事の分離 (Separation of Concerns) という構成では、それぞれのコンポーネントを単純化し、Spring の依存性注入によってコンポーネントを相互作用させることができます。

- Spring は単純な収集 Bean に、初期化、開始、停止の 3 つのオペレーションからなるライフサイクルを提供します。さらにオプションで、これらの Bean への JMX (Java Management Extensions) 管理インターフェースを公開し、実行時に Bean の制御、監視、トラブルシューティングを行うこともできます。

Spring ベースのコレクターについては、この記事でも各セクションの該当する箇所で詳しく説明します。

ホストおよびオペレーティング・システムの監視

Java アプリケーションの実行は、常にそのベースとなるハードウェアと JVM をサポートするオペレーティング・システム上で行われます。そのため、包括的な監視インフラストラクチャーに不可欠となるコンポーネントは、ハードウェアと OS からパフォーマンス、正常性、可用性のメトリックを (通常は OS を介して) 収集する機能です。このセクションでは、これらのデータを取得し、第 1 回で紹介した `ITracer` クラスによってアプリケーション・パフォーマンス管理 (APM) システムにトレースする方法について説明します。

一般的な OS のパフォーマンス・メトリック

以下に、さまざまな OS に共通して関連する一般的なメトリックを要約して記します。データ収集の詳細は OS によってかなり異なるため、データを解釈する際にはその特定の OS のコンテキストを考慮に入れなければなりませんが、これらのメトリックは標準的なホストのほとんどで同じです。

- **CPU 使用率**: 特定のホストで CPU がどれだけビジーであるかを表します。通常使われる単位はパーセントで表される使用率で、特定の経過クロック時間内で CPU がビジー状態になっていたパーセンテージを示します。ホストに複数の CPU があり、CPU に複数のコアが含まれている場合もありますが、ほとんどの OS では一般に、複数のコアはそれぞれのコアを 1 つの CPU として抽象化されます。例えばデュアル・コアが搭載された 2 つの CPU は、4 つの CPU として表されるなどです。メトリックは通常、CPU ごと、あるいはすべてのプロセッサーについての使用率を集約した総リソース使用率として収集されます。個々の CPU を監視するか、あるいは集約したものを監視するかどうかは、ソフトウェアとその内部アーキテクチャーの性質によって決まります。標準的なマルチスレッド Java アプリケーションでは、使用可能なすべての CPU で負荷が分散されるようにデフォルト設定されるので、集約したものを監視すれば十分です。ただし、場合によっては個々の OS プロセスが特定の CPU に「釘付け」されるため、集約したメトリックでは十分な細分度を確保できないこともあります。CPU 使用率は一般に以下の 4 つのカテゴリーに分けられます。
  - **システム**: システム・レベルまたは OS カーネル・レベルのアクティビティーの実行に費やされたプロセッサー時間
  - **ユーザー**: ユーザーのアクティビティーの実行に費やされたプロセッサー時間
  - **I/O 待機**: I/O 要求の完了を待機してアイドル状態になったプロセッサー時間
  - **アイドル**: プロセッサーのアクティビティーが何も行われていない状態を示唆します。

  この他、関連するメトリックとしては実行キュー長とコンテキスト・スイッチの 2 つがあります。実行キュー長は基本的には CPU 時間の割り当てを待っている未処理リクエストの数で、コンテキスト・スイッチは、プロセッサー時間の割り当てをあるプロセスから別のプロセスへ切り替える回数です。

- **メモリー**: 最も単純なメモリー・メトリックは、使用可能または使用中の物理メモリーのパーセンテージです。その他の検討材料には、仮想メモリー、メモリーの割り当ておよび割り当て解除の比率、さらにメモリーのどの特定の領域が使用されているかに関する詳細なメトリックが関わってきます。
- **ディスクおよび I/O**: ディスク・メトリックは、論理または物理ディスク・デバイスごとに使用可能または使用中のディスク・スペース、そしてこれらのデバイスに対する読み取りおよび書き込みの速度についての単純な (けれども極めて重要な) レポートです。
- **ネットワーク**: ネットワーク・インターフェースでのデータ転送速度とエラーの数です。通常は、TCP や IP などの上位レベルのネットワーク・プロトコルのカテゴリーに分けられます。
- **プロセス数およびプロセス・グループ数**: 上記のすべてのメトリックは、特定のホストに対する全体的なアクティビティーとして表すことができます。また、同じメトリックでも、個々のプロセスや関連プロセスのグループでの使用率またはアクティビティーを表すメトリックに分類することも可能です。プロセスごとにリソースの使用率を監視すると、ホストでそれぞれのアプリケーションやサービスが使用しているリソースの割合を理解する上で役立ちます。一部のアプリケーションは 1 つのプロセスしかインスタンス化しませんが、その一方、Apache 2 Web Server などのサービスは、1 つの論理サービスを表すプロセスのプールをインスタンス化する場合もあります。

エージェントとエージェントレスの違い

OS の種類によって、パフォーマンス・データへのアクセスを可能にするためのメカニズムは異なります。これからデータを収集するさまざまな方法を説明しますが、監視の分野で一般的に行われている区別は、エージェントによる監視か、エージェントレスでの監視かの違いです。ここに含まれている意味は、データを収集するには、必ずしもターゲット・ホストに特定のソフトウェアを追加でインストールする必要はないということです。しかし、データを読み取るためのインターフェースがなければ監視することはできないため、ある種のエージェントは必ず必要になります。本当の意味での区別は、特定の OS に常に存在するエージェント (例えば、Linux® サーバーの SSH など) を使用するか、あるいは監視のみを目的としたソフトウェアを追加でインストールして、収集したデータを外部コレクターが使用できるようにするかの違いです。いずれの方法にもトレードオフが伴います。

- エージェントを利用する場合、追加ソフトウェアをインストールしなければなりません。また、定期的なメンテナンス・パッチを適用する必要もあるかもしれません。このようなソフトウェアの管理作業が伴うことから、ホストの数が多い環境ではエージェントはほとんど使用されません。
- エージェントが物理的にアプリケーションと同じプロセスの一部となっている場合、あるいはエージェントが別のプロセスであっても、エージェントのプロセスに障害があると、監視による真実が見えなくなってしまいます。ホスト自体は引き続き実行中で正常な状態であっても、エージェントにアクセスできなければ、APM はホストがダウンしていると見なすためです。
- ホストにインストールされたローカル・エージェントのほうが、エージェントレスのリモート監視よりも遥かにデータ収集およびイベント・リスニング機能に優れる場合があります。さらに、集約メトリックをレポートするには、その基となるメトリックの生データの収集が必要になることから、リモートで実行するには効率的ではありません。ローカル・エージェントでは効率的にデータを収集して集約し、集約したデータをリモートでの監視から使えるようにすることができます。

結局のところ、最適なソリューションはエージェントレスの監視とエージェントによる監視の両方を実装し、ローカル・エージェントで大量のメトリックを収集し、リモートからの監視でサーバーの起動状態とローカル・エージェントのステータスなどの基本事項をチェックするというわけです。

使用できるオプションも、エージェントによって異なります。自律型エージェントは独自のスケジュールでデータを収集しますが、応答型エージェントは要求に応じてデータを収集して配信します。また、リクエスターに対してデータを提供するだけのエージェントもあれば、データを直接または間接的に APM システムにトレースするエージェントもあります。

次のセクションでは、Linux および UNIX® OS を使用したホストの監視手法を紹介します。

Linux および UNIX ホストの監視

Linux および UNIX OS からパフォーマンス・データを収集することに特化したネイティブ・ライブラリーを実装する監視エージェントを使用することもできますが、Linux、そしてほとんどの UNIX バージョンには、/proc という仮想ファイルシステムによるデータ・アクセスを可能にする、組み込みデータ収集ツールが豊富に備わっています。これらのファイルは通常のファイルシステム・ディレクトリーでは一般的なテキスト・ファイルのように見えますが、実際にはメモリー内データ構造であり、テキスト・ファイルのファサードによって抽象化されています。このデータは、さまざまな標準コマンドライン・ユーティリティーやカスタム・ツールで簡単に読み取って構文解析できるため、ファイルは使用しやすく、それぞれの出力でごく一般的なものにも、極めて特有なものにもなり得ます。また、本質的にはメモリーから直接データを抽出することから、パフォーマンスにも非常に優れています。

/proc からパフォーマンス・データを抽出するためによく使用されるツールは、`ps`、`sar`、`iostat`、`vmstat` です (これらのツールのリファレンス・マニュアルについては、「参考文献」を参照してください)。つまり、Linux ホストと UNIX ホストを監視する効率的な方法は、単にシェル・コマンドを実行して応答を解析するだけということになります。同様の監視は、さまざまな種類の Linux および UNIX 実装で使用することができます。ただし、これらの実装はそれぞれに多少異なるので、収集プロシージャーを完全に再利用できるようにデータ・フォーマットを設定するのが一般的です。一方、特殊化したネイティブ・ライブラリーの場合には Linux および UNIX ディストリビューションごとに記録してリビルドする必要があるかもしれません (いずれにしても、読み取る /proc データは同じです)。さらに、特定のケースに特化した監視を行えるカスタム・シェル・コマンドや、返されたデータのフォーマットを標準化するカスタム・シェル・コマンドを作成するのも簡単で、オーバーヘッドも大きくありません。

ここで、シェル・コマンドを呼び出して、返されたデータをトレースする具体的な方法をいくつか紹介します。

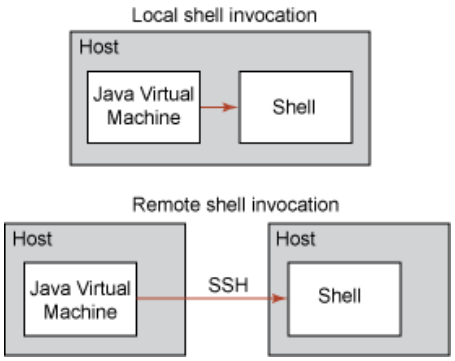
シェル・コマンドの実行

Linux ホスト上でデータを収集して監視を実行するには、シェルを呼び出す必要があります。シェルは `bash`、`csh`、`ksh` の他、ターゲットのスクリプトまたはコマンドを呼び出して出力を取得できるシェルであれば何でも構いません。シェルを呼び出す際によく使用される選択肢には、以下のものがあります。

- **ローカル・シェル**: ターゲット・ホストで JVM を実行している場合、スレッドは `java.lang.Process` を呼び出すことによってシェルにアクセスできます。
- **リモート Telnet または `rsh`**: 両方ともシェルおよびシェル・コマンドを呼び出せるサービスですが、比較的安全性に欠けるため、使用されなくなってきています。現在のほとんどのディストリビューションでは、この 2 つのサービスはデフォルトで使用不可に設定されています。
- **セキュア・シェル (SSH)**: SSH は最も一般的に使われるリモート・シェルです。Linux シェルへのフル・アクセスを可能にするこのシェルは、一般にセキュアと見なされています。この記事のシェル・ベースの例では、SSH を主要なメカニズムとして使用します。SSH サービスは、UNIX の実質上すべてのフレーバー、そして Microsoft® Windows®、OS/400、z/OS をはじめとする広範な OS で使用することができます。

図 1 に、ローカル・シェルとリモート・シェルの概念上の違いを示します。

図 1. ローカル・シェルとリモート・シェル



サーバーとの自動 SSH セッションは、多少のセットアップ作業を行うだけで開始できます。必要な作業は、秘密鍵と公開鍵からなる SSH 鍵ペアを作成することです。公開鍵の中身はターゲット・サーバーに置き、秘密鍵はデータ・コレクターがアクセス可能なリモート監視サーバーに配置します。この作業が済めば、データ・コレクターは秘密鍵と秘密鍵のパスフレーズを提供することによって、ターゲット・サーバーのセキュア・リモート・シェルにアクセスできるようになります。ターゲット・アカウントのパスワードは必要ありません。鍵ペアを使用する場合、このパスワードは余分だからです。セットアップは以下の手順に従います。

1. ローカル側に置かれた既知のホスト・ファイルに、ターゲット・ホストのエントリーがあることを確認します。このファイルは、既知の IP アドレスまたは名前、そしてそれぞれに対して認識される関連 SSH 公開鍵を記載するファイルです。ユーザー・レベルでは、このファイルは通常、ユーザーのホーム・ディレクトリーに置かれた ~/.ssh/known_hosts ファイルとなります。
2. 監視アカウント (monitoruser など) を使用してターゲット・サーバーに接続します。
3. ホーム・ディレクトリーに .ssh という名前のサブディレクトリーを作成します。
4. ディレクトリーを .ssh ディレクトリーに変更し、`ssh-keygen -t dsa` コマンドを実行します。このコマンドによって、鍵の名前とパスフレーズの入力を求めるプロンプトが出されます。続いて monitoruser_dsa (秘密鍵)、monitoruser._dsa.pub (公開鍵) という 2 つのファイルが生成されます。
5. データ・コレクターが実行される場所からアクセスできるセキュアな場所に秘密鍵をコピーします。

6. 公開鍵の中身を .ssh ディレクトリーにある authorized_keys というファイルに追加するため、`cat monitoruser_dsa.pub >> authorized_keys` コマンドを実行します。

リスト 1 に、上記で概説したプロセスを示します。

リスト 1. SSH 鍵ペアの作成

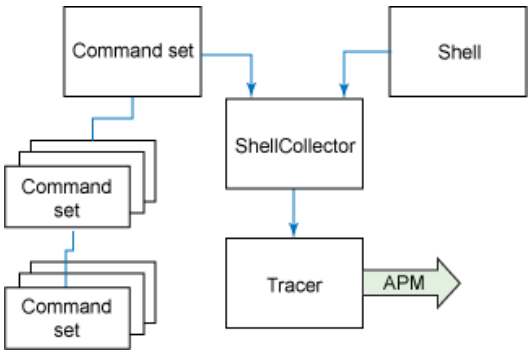
```
whitehen@whitehen-desktop:~$ mkdir .ssh
whitehen@whitehen-desktop:~$ cd .ssh
whitehen@whitehen-desktop:~/.ssh$ ssh-keygen -t dsa
Generating public/private dsa key pair.
Enter file in which to save the key (/home/whitehen/.ssh/id_dsa): whitehen_dsa
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in whitehen_dsa.
Your public key has been saved in whitehen_dsa.pub.
The key fingerprint is:
46:cd:d4:e4:b1:28:d0:41:f3:ea:3b:8a:74:cb:57:e5 whitehen@whitehen-desktop
whitehen@whitehen-desktop:~/.ssh$ cat whitehen_dsa.pub >> authorized_keys
whitehen@whitehen-desktop:~/.ssh$
```

このプロセスによって、データ・コレクターは、Ubuntu Linux を実行する whitehen-desktop というターゲット Linux ホストに対して SSH 接続を行えるようになります。

この例でのデータ収集に使用するのは、汎用コレクター・クラス、`org.runtimemonitoring.spring.collectors.shell.ShellCollector` です。このクラスのインスタンスが、`UbuntuDesktopRemoteShellCollector` という名前で Spring コンテキストにデプロイされます。しかしプロセス全体を完了するためには、以下の依存関係がまだ残っています。

- コレクターを 1 分間に 1 回呼び出すためのスケジューラー。それには、`java.util.concurrent.ScheduledThreadPoolExeutor` のインスタンスによって、スケジュールされたコールバック・メカニズムとスレッド・プールの両方を提供します。このインスタンスは `CollectionScheduler` という名前で Spring にデプロイされることになります。
- サーバーに対してコマンドを実行し、結果を返す SSH シェル実装。これは、`org.runtimemonitoring.spring.collectors.shell.ssh.JSchRemoteShell` のインスタンスによって提供します。`org.runtimemonitoring.spring.collectors.shell.IRemoteShell` というシェル・インターフェースの実装であるこのクラスは、`UbuntuDesktopRemoteShell` という名前で Spring にデプロイされることになります。
- コマンドとそれに関連付けられた構文解析ルーチンのセットをハード・コーディングする代わりに、コレクターは `org.runtimemonitoring.spring.collectors.shell.commands.CommandSet` のインスタンスを使用します。このインスタンスは、`UbuntuDesktopCommandSet` という名前で Spring にデプロイされることになります。コマンド・セットは、以下について記述する XML 文書からロードされます。
  - シェルの実行対象となるターゲット・プラットフォーム
  - 実行されるコマンド
  - 返されるデータの構文解析方法と、APM トレース名前空間へのマッピング方法
- 以上の定義についてはこの後追って詳細を説明するので、まずは図 2 に示した、コレクター、シェル、コマンド・セットの基本的な関係の概要を見てください。

図 2. コレクター、シェル、コマンド・セット



それではここから、パフォーマンス・データを生成する具体的なコマンドとその構成方法についての簡単な例を探っていくことにします。典型的なコマンドの例は `sar` です。Linux の man ページ (「参考文献」を参照) によると、`sar` の定義はシステム・アクティビティーの情報を収集、レポート、または保存することです。このコマンドはかなり柔軟で、20 を超える引数を組み合わせて使用することができます。単純なオプションとして例えば `sar -u 1 3` を呼び出すと、1 秒間隔で 3 回測定された CPU 使用率がレポートされます。リスト 2 はその出力です。

リスト 2. `sar` コマンドの出力例

```
whitehen@whitehen-desktop:~$ sar -u 1 3
Linux 2.6.22-14-generic (whitehen-desktop)      06/02/2008

06:53:24 PM     CPU     %user     %nice   %system   %iowait    %steal     %idle
06:53:25 PM     all      0.00      0.00      0.00      0.00      0.00    100.00
06:53:26 PM     all      0.00     35.71      0.00      0.00      0.00     64.29
06:53:27 PM     all      0.00     20.79      0.99      0.00      0.00     78.22
Average:        all      0.00     18.73      0.33      0.00      0.00     80.94
```

上記の出力は、プリアンブル、ヘッダー、1 秒間隔での 3 回のデータ測定値、そして測定値の平均に分けられます。ここでの目標は、このシェル・コマンドを実行し、出力を取り込んで構文解析し、APM システムにトレースすることです。フォーマットはごく単純ですが、出力フォーマットはバージョンによって (多少、あるいは大幅に) 異なり、他の `sar` オプションではまったく異なるデータが返されます (他のコマンドが異なるフォーマットを返すことは言うまでもありません)。一例として、リスト 3 に `sar` の実行によって表示されたアクティブなソケット・アクティビティーを示します。

リスト 3. `sar` によるソケット・アクティビティーの表示

```
whitehen@whitehen-desktop:~$ sar -n SOCK 1
Linux 2.6.22-14-generic (whitehen-desktop)      06/02/2008

06:55:10 PM    totsck    tcpsck    udpsck    rawsck   ip-frag
06:55:11 PM       453         7         9         0         0
Average:          453         7         9         0         0
```

したがって何が必要かと言えば、コレクターのコードを変更することなく、異なるフォーマットを素早く構成できるようにするソリューションです。また、収集されたトレースが APM システムに到達する前に、`totsck` などといった不可解な言葉を Total Used Sockets のような理解しやすいフレーズに変換できることも役立ちます。

場合によっては、このデータを XML フォーマットで取得するというオプションも使用できます。例えば、SysStat パッケージ (「参考文献」を参照) に含まれる `sadf` コマンドは、一般的に収集される Linux 監視データのほとんどを XML フォーマットで生成します。XML フォーマットはデータの予測可能性と構造に大きく貢献し、構文解析、データとトレース名前空間とのマッピング、そして不明瞭な言葉の解読というタスクを事実上不要にします。ただしこれらのツールは、シェルでアクセス可能な監視対象のシステムに対して常に使用できるというわけではないので、柔軟にテキストを構文解析してマッピングできるソリューションがかけがえのないものとなります。

以上の 2 つの `sar` の使用例に続き、今度はこのデータを監視する重要な Spring Bean 定義のセットアップ例を紹介します。ここに記載するすべての例は、この記事付属のサンプル・コード (「ダウンロード」を参照) に含まれています。

まず、`SpringCollector` 実装のメイン・エントリー・ポイントは `org.runtimemonitoring.spring.collectors.SpringCollector` で、このクラスが取る引数は Spring Bean 構成ファイルが置かれたディレクトリーの名前だけです。`SpringCollector` は .xml 拡張子の付いたすべてのファイルをロードして Bean 記述子として扱います。これらのファイルが置かれるディレクトリーは、プロジェクトのルートにある ./spring-collectors ディレクトリーです (この記事では後で、このディレクトリーに含まれるすべてのファイルについて概説します。多くのファイルはオプションなので、すべての定義を 1 つにまとめることもできますが、概念的な機能ごとにファイルを分けておくと多少理解しやすくなります)。この例で使用する 3 つの Bean 定義は、それぞれシェル・コレクター、シェル、コマンド・セットを表します。これらの記述子はリスト 4 のとおりです。

リスト 4. シェル・コレクター、シェル、コマンド・セットの Bean 記述子

```
<!-- The Collector -->
<bean id="UbuntuDesktopRemoteShellCollector"
  class="org.runtimemonitoring.spring.collectors.shell.ShellCollector"
  init-method="springStart">
  <property name="shell" ref="UbuntuDesktopRemoteShell"/>
  <property name="commandSet" ref="UbuntuDesktopCommandSet"/>
  <property name="scheduler" ref="CollectionScheduler"/>
  <property name="tracingNameSpace" value="Hosts,Linux,whitehen-desktop"/>
  <property name="frequency" value="5000"/>
</bean>

<!-- The Shell -->
<bean id="UbuntuDesktopRemoteShell"
   class="org.runtimemonitoring.spring.collectors.shell.ssh.JSchRemoteShell"
   init-method="init"
   destroy-method="close">
   <property name="userName" value="whitehen"/>
   <property name="hostName" value="whitehen-desktop"/>
   <property name="port" value="22"/>
   <property name="knownHostsFile"
      value="C:/Documents and Settings/whitehen/.ssh/known_hosts"/>
   <property name="privateKey"
      value="C:/keys/whitehen/ubuntu-desktop/whitehen_dsa"/>
   <property name="passphrase" value="Hello World"/>
</bean>

<!-- The CommandSet -->
<bean id="UbuntuDesktopCommandSet"
   class="org.runtimemonitoring.spring.collectors.shell.commands.CommandSet">
   <constructor-arg type="java.net.URL"
      value="file:///C:/projects//RuntimeMonitoring/commands/xml/UbuntuDesktop.xml"/>
```

```
</bean>
```

リスト 4 の `CommandSet` Bean には、`id` (`UbuntuDesktopCommandSet`) と別の XML ファイルへの URL しかありません。このようにしている理由は、コマンド・セットはかなり大きいため、Spring ファイルをこれらのコマンド・セットで占領しないようにするためです。`CommandSet` については、この後すぐに説明します。

リスト 4 で最初に記載されている Bean は `UbuntuDesktopRemoteShellCollector` です。この Bean の `id` は純粋に任意の説明的な値にすることができますが、この Bean が別の Bean から参照されるときには値が一貫していなければなりません。この例でのクラスは `org.runtimemonitoring.spring.collectors.shell.ShellCollector` で、これはシェルのようなインターフェースでデータを収集するための汎用クラスです。その他の重要なプロパティーには以下のものがあります。

- **`shell`**: シェル・クラスのインスタンスです。コレクターはこのインスタンスを使用して、シェル・コマンドを呼び出し、データを取得します。Spring はこのシェルのインスタンスに Bean の `id`、`UbuntuDesktopCommandSet` を注入します。
- **`commandSet`**: コマンドのセットとそれに関連付けられた構文解析ディレクティブおよびトレース名前空間とのマッピング・ディレクティブを表す `CommandSet` インスタンスです。Spring はこのコマンド・セットのインスタンスに Bean の `id`、`UbuntuDesktopRemoteShell` を注入します。
- **`scheduler`**: データの収集スケジュールを管理し、スレッドをジョブに割り当てるスケジューリング・スレッド・プールへの参照です。
- **`tracingNameSpace`**: トレース名前空間の接頭辞です。この接頭辞が、APM ツリーのどこにメトリックをトレースするかを制御します。
- **`frequency`**: データの収集頻度です (ミリ秒単位)。

リスト 4 の 2 番目の Bean はシェルで、`org.runtimemonitoring.spring.collectors.shell.ssh.JSchRemoteShell` という SSH シェルを実装するクラスです。このクラスは JCraft.com の JSch (「参考文献」を参照) を使用して実装しており、このクラスの重要なプロパティーには以下のものがあります。

- **`userName`**: Linux サーバーに接続するために使用するユーザー名です。
- **`hostName`**: 接続先の Linux サーバーの名前 (または IP アドレス) です。
- **`port`**: `sshd` がリッスンしている Linux サーバー・ポートです。
- **`knownHostFile`**: SSH クライアントが実行中のローカル・ホストに「既知」である SSH サーバーのホスト名および SSH 証明書が含まれるファイルです (SSH でのこのセキュリティー・メカニズムの興味深い点は、従来のセキュリティー階層とは逆になっていることです。従来のセキュリティー階層では、クライアントがホストを信頼することはできないため、ホストが「既知」であり、一致する証明書を提示しない限り、クライアントはホストに接続しません)。
- **`privateKey`**: SSH サーバーに対する認証で使用する SSH 秘密鍵ファイルです。
- **`passPhrase`**: 秘密鍵のアンロックに使用するパスフレーズです。パスワードと同じように見えますが、サーバーには送信されず、ローカル側で秘密鍵を復号するためだけに使用されます。

リスト 5 に、`CommandSet` の構成要素を示します。

リスト 5. `CommandSet` の構成要素

```
<CommandSet name="UbuntuDesktop">
   <Commands>
      <Command>
         <shellCommand>sar -u 1</shellCommand>
         <paragraphSplitter>\n\n</paragraphSplitter>
    <Extractors>
       <CommandResultExtract>
          <paragraph id="1" name="CPU Utilization"/>
          <columns entryName="1" values="2-7" offset="1">
             <remove>PM</remove>
          </columns>
          <tracers default="SINT"/>
               <filterLine>Average:.*</filterLine>
          <lineSplit>\n</lineSplit>
       </CommandResultExtract>
    </Extractors>
      </Command>
      <Command>
         <shellCommand>sar -n SOCK 1</shellCommand>
    <paragraphSplitter>\n\n</paragraphSplitter>
    <Extractors>
       <CommandResultExtract>
          <paragraph id="1" name="Socket Activity"/>
          <columns values="1-5" offset="1">
             <remove>PM</remove>
             <namemapping from="ip-frag" to="IP Fragments"/>
             <namemapping from="rawsck" to="Raw Sockets"/>
             <namemapping from="tcpsck" to="TCP Sockets"/>
             <namemapping from="totsck" to="Total Sockets"/>
             <namemapping from="udpsck" to="UDP Sockets"/>
          </columns>
          <tracers default="SINT"/>
               <filterLine>Average:.*</filterLine>
          <lineSplit>\n</lineSplit>
       </CommandResultExtract>
    </Extractors>
      </Command>
   </Commands>
</CommandSet>
```

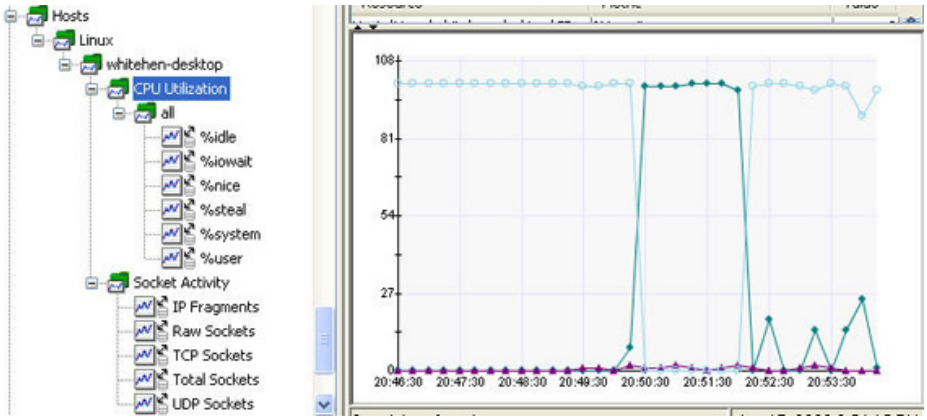
`CommandSet` の役割は、シェル・コマンドを管理し、ディレクティブを構文解析することです。それぞれの Linux または UNIX システムによって出力は多少異なるため、同じコマンドであったとしても、通常は監視対象の特定ホスト・タイプごとに 1 つの `CommandSet` があります。`CommandSet` の XML のオプションは、新しい状況に合わせて常に進化し、調整が行われているため、オプションの 1 つひとつについては詳しく説明できませんが、以下にいくつかのタグに関する概要をまとめます。

- **`<shellCommand>`**: シェルに渡される実際にコマンドを定義します。
- **`<paragraphSplitter>`**: 一部のコマンド、あるいは連鎖する複数のコマンドからなるコマンドは、テキストの複数のセクションを結果として返す場合があります。これらのテキストは、パラグラフと呼ばれます。どのようにパラグラフの境界を区分するかを指定するのが、このタグに定義される正規表現 (regex) です。コマンド・オブジェクトは結果を複数のパラグラフに分割し、要求されたパラグラフをベースとなるエクストラクター (抽出プログラム) に渡します。
- **`<Extractors>` とそこに含まれる `<CommandResultExtract>` タグ**: これらの構成体は、構文解析とマッピングを定義します。

- **<paragraph>**: エクストラクターは、id 属性内のゼロ・ベースのインデックスを使用して、結果からどのパラグラフが必要かを定義します。このパラグラフからトレースされるすべてのメトリックは、パラグラフ名に定義されたトレース名前空間に分類されます。
- **<columns>**: entryName が定義されている場合、各行のインデックス付き列がトレース名前空間に追加されます。これが該当するのは、左側の列にメトリック境界区分が含まれる場合です。例えば、sar のあるオプションが個別の CPU それぞれに関する CPU 使用率をレポートする場合には、2 番目の列に CPU 番号が記載されます。リスト 5 では、entryName が、レポートが全 CPU の集約結果であることを示す all 修飾子を抽出します。values 属性が各行のどの列をトレースするかを指定し、offset がデータ行の列の番号とそれに対応するヘッダーとの違いを調整します。
- **<tracers>**: デフォルトのトレース・タイプを定義します。このタグを使用して、指定のヘッダーまたは entryName に関連付けられた値に対してさまざまなトレーサー・タイプを定義することができます。
- **<filterLine>**: このタグが定義されている場合、regex はテキスト全体が一致しないデータ行を無視します。
- **<lineSplit>**: 各パラグラフに含まれる行を構文解析するための分割 regex を定義します。

図 3 に、この例の場合の APM ツリーを示します。

図 3. Ubuntu Desktop 監視の APM ツリー



このツリーの表示が気に入らなければ、他にもオプションがあります。サーバーに送信されるコマンドは、一連の grep、awk、sed コマンドをパイプでつなぐように簡単に変更することができます。このようにすれば、必要な構文解析の量が大幅に少ないフォーマットにデータを再フォーマット設定することができます。リスト 6 は、その一例です。

リスト 6. コマンド内でのコマンド出力のフォーマット設定

```
whitehen@whitehen-desktop:~$ sar -u 1 | grep Average | \
   awk '{print "SINT/User:"$3"/System:"$5"/IOWait:"$6}'
SINT/User:34.00/System:66.00/IOWait:0.00
```

構成、柔軟性、そしてパフォーマンスを最適に組み合わせるもう 1 つのオプションは、動的スクリプトを使用することです。追加のフォーマット設定ツールが使用できなかったり、出力フォーマットが極めて扱いにくかったりする場合には、動的スクリプトが特に役立ちます。次の例で私が構成したのは、Cisco CSS ロード・バランサーからロードバランシングの状況に関するデータを

収集する Telnet シェルです。標準化されたあらゆる類の構文解析では出力フォーマットとコンテンツが特に問題となり、シェルは限られたコマンドしかサポートしません。リスト 7 に、このコマンドの出力を記載します。

リスト 7. CSS Telnet コマンドの出力

```
Service Name                    State     Conn  Weight  Avg   State
                                                      Load  Transitions

ecommerce1_ssl                  Alive        0      1   255            0
ecommerce2_ssl                  Down         0      1   255            0
admin1_ssl                      Alive        0      1     2         2982
admin2_ssl                      Down         0      1   255            0
clientweb_ssl                   Alive        0      1   255            0
```

XML 要素と属性の違い

アプリケーションによって、明らかに XML 要素を属性に優先して使用する場合、あるいはその逆の場合があります。私はどちらかを絶対的に優先させることはしませんが、2 つの経験則に従って一貫性を保つようにしています。

- 短く定義された名前付きの値のほうが読みやすいこと。
- 値のテキストが長くなるような場合、あるいは避けなければならない不正な文字 (& や " など) が含まれることになる場合には、要素を使用すること。これは、XML パーサーに対して、該当するコンテンツを構文解析しないように指示する CDATA ブロックにデータをラップすることができるためです。

リスト 8 に、このコマンドを実行して構文解析を行うために使用したコマンド・セットを記載します。`<preFormatter beanName="FormatCSSServiceResult"/>` タグに注目してください。これは、Groovy スクリプトが何行か含まれる Spring Bean への参照です。Telnet シェル・コマンドの出力はそのままこの Groovy スクリプトに渡された後、遥かに理解しやすいフォーマットになって戻り値がコマンド・データ・エクストラクターに渡されます。また、Status のラベルが付いた列に含まれる値の場合、トレーサーの型が `STRING` 型に変更されることにも注目してください。注意深く見てみると、そのような名前の列はないことに気付くでしょうが、Groovy スクリプトはそのジョブの一環として、State という名前の列が 2 つになりそうなことから (その理由は見てのとおりです)、最初の列の名前を Status に変更します。

リスト 8. CSS の `CommandSet`

```
<CommandSet name="CiscoCSS">
   <Commands>
      <Command>
         <shellCommand>show service summary</shellCommand>
       <paragraphSplitter>\n\n\n\n</paragraphSplitter>
       <preFormatter beanName="FormatCSSServiceResult"/>
       <Extractors>
          <CommandResultExtract>
             <paragraph id="0" name="Service Summary" header="true"/>
        <columns entryName="0" values="1-5" offset="0"/>
        <tracers default="SINT">
           <tracer type="STRING">Status</tracer>
        </tracers>
        <lineSplit>\n</lineSplit>
          </CommandResultExtract>
       </Extractors>
         </Command>
   </Commands>
</CommandSet>
```

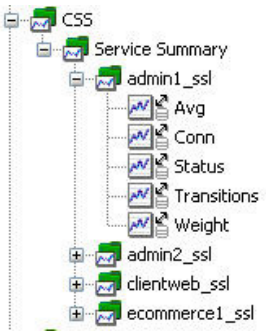
Groovy Bean には数多くの利点があります。まず、このスクリプトは動的に構成できるため、実行時に変更することが可能です。Bean はソースが変更されたことを検出すると、次の呼び出し時に Groovy コンパイラーを呼び出すことから、パフォーマンスの点でも十分です。さらに、この言語には構文解析の機能が充実しているだけでなく、作成するのも簡単です。リスト 9 に、ソース・コードのテキストをインラインに含めた Groovy Bean を示します。

リスト 9. Groovy フォーマット設定 Bean

```
<bean id="FormatCSSServiceResult"
   class="org.runtimemonitoring.spring.groovy.GroovyScriptManager"
   init-method="init" lazy-init="false">
   <property name="sourceCode"><value><![CDATA[
      String[] lines = formatTarget.toString().split("\r\r\n");
      StringBuffer buff = new StringBuffer();
      lines.each() {
         if(!(
            it.contains("Load  Transitions") ||
            it.contains("show service summary") ||
            it.trim().length() < 1)) {
               buff.append(it).append('\n');
          }
      }
      return buff.toString()
      .replaceFirst("State", "Status")
      .replaceFirst("Service Name", "ServiceName")
      .replace("State", "Transitions");
      ]]></value>
   </property>
</bean>
```

図 4 は、CSS を監視するためのメトリック・ツリーです。

図 4. CSS 監視のための APM ツリー



SSH 接続

Linux/UNIX シェルでの収集に関する最後の検討事項は、SSH 接続の問題です。シェル・クラスすべての基本インターフェースは `org.runtimemonitoring.spring.collectors.shell.IShell` で、`issueOSCommand()` というメソッドの 2 つのバリエーションを定義しています。このメソッドでは、コマンドがパラメーターとして渡されて結果が返されます。先ほどのリモート SSH クラス `org.runtimemonitoring.spring.collectors.shell.ssh.JSchRemoteShell` を使用した例で、基礎となるシェルの呼び出しのベースとなっているのは Apache Ant の `SSHEXEC` タスク実装です (「参考文献」を参照)。この手法は単純であるという利点がありますが、決定的な欠点もあります。それは、コマンドが実行されるたびに新しい接続が行われることです。当然、これでは効率的でありません。リモート・シェルが数分間隔でしかポーリングされないとしても、ポーリング・サイ

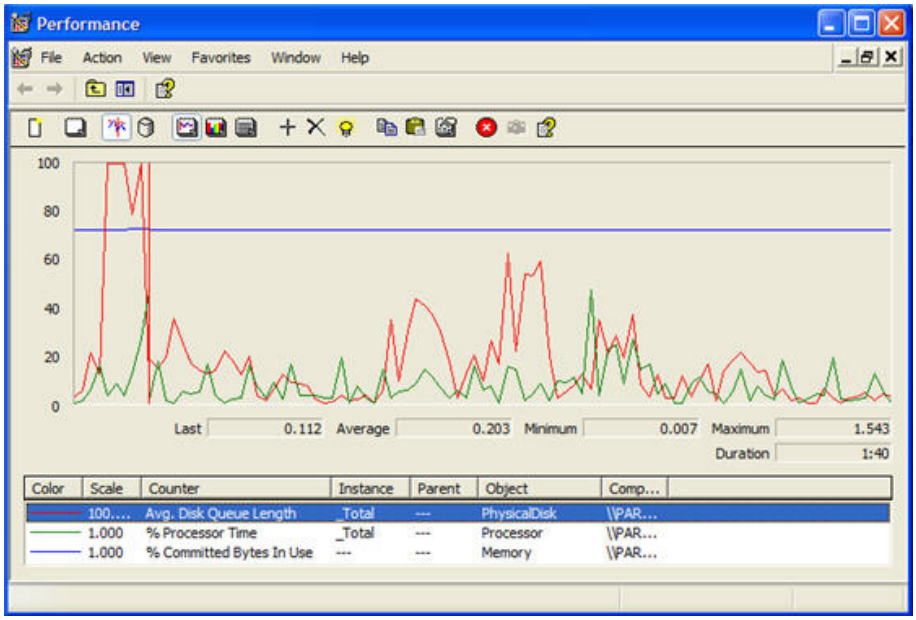
クルのそれぞれで、適切な範囲の監視データを取得するために複数のコマンドが実行されます。問題は、監視期間を通して (複数のポーリング・サイクル全体で) オープン・セッションを維持するのは難しいということです。そのためには戻りデータをさらに詳細に検査し、異なるシェル・タイプと一貫したシェル・プロンプトの表示を考慮した構文解析が必要になってきます。しかしもちろん、このような情報は期待する戻り値には含まれていません。

私はこれまで、長時間存続するセッション・シェル実装に取り組んできましたが、代わりとなる手段は妥協することです。ポーリング・サイクル・パターンごとに 1 つの接続を維持する一方、1 つのコマンドですべてのデータを取り込むようにします。その方法としては、複数のコマンドを追加するか、あるいは場合によっては 1 つのコマンドに対して複数のオプションを使用することもできます。例えば、私が使用している SuSE Linux サーバーの `sar` のバージョンには、サポートされるすべての `sar` のメトリックのサンプリングを返す `-A` オプションが用意されています。このコマンドは、`sar` -bBcdqrRuvwWy -I SUM -n FULL -P ALL に相当します。返されるデータには複数のパラグラフが含まれることになりますが、コマンド・セットで構文解析するには何の問題もないはずです。その一例としては、この記事のサンプル・コード (「ダウンロード」を参照) に含まれるコマンド・セット定義、Suse9LinuxEnterpriseServer.xml を参照してください。

Windows の監視

Microsoft Windows と Linux/UNIX との間にはかなりの違いがあります。パフォーマンス・データの収集にしても例外ではありません。事実上、Windows には Linux/UNIX に匹敵するほどの豊富なパフォーマンス・レポート・データを提供するネイティブ・コマンドライン・ツールはありません。また、/proc ファイルシステムのような比較的単純な (パフォーマンス・データへの) アクセス手段もありません。Windows ホストからパフォーマンス測定値を取得するには、WPM (Windows Performance Manager) が標準インターフェースとなります (これは、SysMon、System Monitor、Performance Monitor と呼ばれることもあります)。WPM は極めて強力で、有益なメトリックが豊富に用意されています。さらに、Windows ベースの多くのソフトウェア・パッケージはそれぞれ独自のメトリックを WPM を介してパブリッシュします。Windows では WPM によってグラフ化機能、レポート機能、アラート機能も提供しています。図 5 は、WPM インスタンスのスクリーンショットです。

図 5. WPM



WPM が管理する一連のパフォーマンス・カウンターは、特定のメトリックを参照する複合名が指定されたオブジェクトです。この複合名は以下のものから構成されます。

- **パフォーマンス・オブジェクト**: パフォーマンス・メトリックの一般的なカテゴリーです (Processor、Memory など)。
- **インスタンス**: 一部のパフォーマンス・オブジェクトは、考えられるメンバーが複数ある場合にはインスタンスごとに区分されます。例えば、Processor には個々の CPU を表すインスタンスと集計インスタンスがあります。それとは対照的に、Memory は「フラット」なパフォーマンス・オブジェクトです。メモリーの場合には 1 つの形でしか表明されないためです。
- **カウンター**: インスタンス (該当する場合) およびパフォーマンス・オブジェクト内での細分化されたメトリックの名前です。例えば、Processor インスタンス 0 のカウンターは % Idle Time という名前になります。

上の名前の区分に基づき、以下の命名規則および構文によってオブジェクトが表現されます。

- **インスタンスがある場合**: \パフォーマンス・オブジェクト(インスタンス名)\カウンター名
- **インスタンスがない場合**: \パフォーマンス・オブジェクト\カウンター名

WPM の大きな欠点は、このデータにアクセスするのが難しいことです。これはリモートからアクセスする場合には顕著で、さらに Windows 以外のプラットフォームからアクセスするとなると至難の業となります。そこで、これから `ITracer` ベースのコレクターを使って WPM のデータを収集する手法をいくつか紹介します。以下に、主なオプションを簡単にまとめます。

- **ログ・ファイルの読み取り**: 収集されたメトリックをすべてログ・ファイルに記録するように WPM を構成し、このログ・ファイルを読み取って構文解析し、トレースします。
- **データベース・クエリー**: 収集されたメトリックをすべて SQL Server データベースに記録するように WPM を構成し、このデータベースからメトリックを読み取って構文解析し、トレースします。

- Win32 API: Win32 API (.NET、C++、Visual Basic など) を使って作成されたクライアントを、WPMの API を使用して直接 WPM に接続します。
- カスタム・エージェント: Windows 以外のクライアントからの WPM データに対する外部要求のプロキシーとして機能するカスタム・エージェントをターゲット Windows サーバーにインストールします。
- SNMP (Simple Network Management Protocol): SNMP はエージェントのインスタンスで、デバイスやホストなどの監視機能における仮想の普遍性に大きな重点を置いています。SNMP については、この記事で追って説明します。
- WinRM: WinRM は、Web サービスを使用したシステム管理の要点をまとめた WS-Management 仕様の Windows 実装です。Web サービスは言語とプラットフォームには依存しないため、WinRM では確実に、Windows 以外のクライアントが WPM メトリックにアクセスすることができます。エージェントの別の形として見なすこともできますが、Windows 2008 から標準装備される予定なので、エージェントレス・ソリューションの分野に分類されます。さらに興味深い点として、Java Specification Request 262 (Web Services Connector for JMX Agent) では Windows ベースの WS-Management サービスと直接相互作用することを確約しています。

これから紹介する例では、ローカル Windows シェルとエージェント実装を使用して理論的な概念実証を行います。

ローカル Windows シェル

単純な概念実証として、Windows コマンドライン実行可能ファイルを C# で作成しました。`winsar.exe` と名付けたこのファイルの目的は、Linux/UNIX の `sar` コマンドと同じようにコマンドラインを使ってパフォーマンス統計にアクセスできるようにすることです。コマンドラインを使用するための構文は、`winsar.exe` *`CategoryCounterRawInstance`* という単純なものです。

カウンターがインスタンス・カウンターではなく、すべて (`*`) を設定可能な場合を除き、インスタンス名は必須です。カウンター名も必須ですが、すべて (`*`) に設定することもできます。`Raw` は `true` または false のいずれかです。リスト 10 に、インスタンス・ベースのカウンターとインスタンス・ベースではないカウンターの使用例を記載します。

リスト 10. `winsar` での非インスタンス・ベースのカウンターおよびインスタンス・ベースのカウンターの使用例

```
C:\NetProjects\WinSar\bin\Debug>winsar Memory "% Committed Bytes In Use" false
 %-Committed-Bytes-In-Use
 79.57401

C:\NetProjects\WinSar\bin\Debug>winsar LogicalDisk "Current Disk Queue Length" false C:
         Current-Disk-Queue-Length
C:  2
```

`sar` のようなものを作成するという目的に従い、標準のシェル・コマンド・セットを使って構文解析できるように、データ出力は大まかな (フォーマット設定されていない) 表形式にします。インスタンス・ベースのカウンターでは、インスタンスはデータ行の最初の列に含め、カウンター名はヘッダー行に並べます。非インスタンス・ベースのカウンターの場合には、データ行の最初のフィールドに名前は含まれません。構文解析を明確にするため、スペースが含まれる名前には「-」文字を入れます。結果はかなり見苦しくなりますが、構文解析は容易になります。

統計 (省略して表示されます) のコレクターをセットアップするのは極めて簡単です。シェル実装は `org.runtimemonitoring.spring.collectors.shell.local.WindowsShell` で、コマンド・セットは `winsar.exe` および引数を参照します。このシェルは SSH を使用したリモート・シェルとして実装することも可能で、その場合にはターゲット Windows ホストに SSH サーバーをインストールする必要があります。しかし、このようなソリューションは極めて効率性に欠けるという難点があります。その第 1 の理由は、この実装が .NET をベースとしているためです。つまり、このような短期間の繰り返しに対して CLR (Common Language Runtime) を起動させるのは効率的ではありません。

別のソリューションは、ネイティブ C++ で `winsar` を作成し直すことですが、この方法は Windows プログラミングの専門家に任せることにします。.NET ソリューションを効率化することはできますが、このプログラムは WPM データに対するリクエストが完了するたびに終了しないように、別の何らかの手段でリクエストに対応しながらバックグラウンド・プロセスとして実行を続けなければなりません。その方法を追求するなかで、私は `winsar` に 2 つ目のオプションを実装しました。このオプションでは、`-service` という引数によってプログラムが開始され、`winsar.exe.config` という構成ファイルを読み取って JMS (Java Message Service) トピックに対するリクエストをリッスンします。この構成ファイルの内容は、ほとんどが見ればすぐにわかりますが、説明を要するものもいくつかあります。まず、`jmsAssembly` 項目が参照するのは、JMS 機能を提供している JBoss 4.2.2 クライアント・ライブラリーの .NET バージョンが含まれる .NET アセンブリーの名前です。このアセンブリーは IKVM (「参考文献」を参照) を使用して作成されています。次に、`respondTopic` は 他のリスナーもデータを受け取ることができるように、プライベート・トピックを使用する代わりに、レスポンスがパブリッシュされるパブリック・トピックの名前を参照します。`commandSet` は、汎用レシーバーがデータを構文解析してトレースするために使用するコマンド・セットへの参照です。リスト 11 に、この `winsar.exe.config` ファイルを記載します。

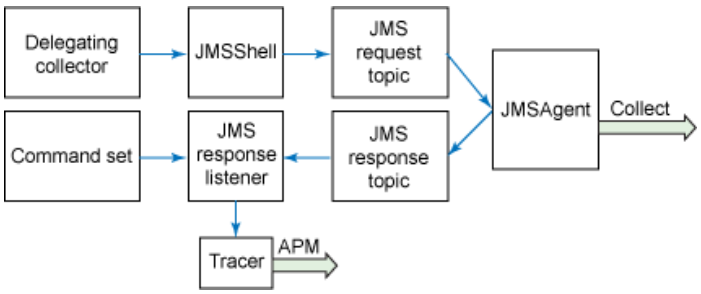
リスト 11. winsar.exe.config ファイル

```
<configuration>
   <appSettings>
      <add key="java.naming.factory.initial"
         value="org.jnp.interfaces.NamingContextFactory"/>
      <add key="java.naming.factory.url.pkgs"
         value="org.jboss.naming:org.jnp.interfaces"/>
      <add key="java.naming.provider.url" value="10.19.38.128:1099"/>
      <add key="connectionFactory" value="ConnectionFactory"/>
      <add key="listenTopic" value="topic/StatRequest"/>
      <add key="respondTopic" value="topic/StatResponse"/>
      <add key="jmsAssembly" value="JBossClient422g" />
      <add key="commandSet" value="WindowsServiceCommandSet" />
   </appSettings>
</configuration>
```

このサービスを使用するコレクターを Spring に実装する方法は、シェルのセットアップと概念的には同じです。実際、このコレクター自体は `org.runtimemonitoring.spring.collectors.shell.DelegatingShellCollector` という `org.runtimemonitoring.spring.collectors.shell.ShellCollector` の継承で、異なるところは、このシェルは通常のコレクターのような動作をしてデータに対するリクエストを出しますが、データは JMS によって受け取られ、別のコンポーネントによって構文解析およびトレースされるという点です。実装された

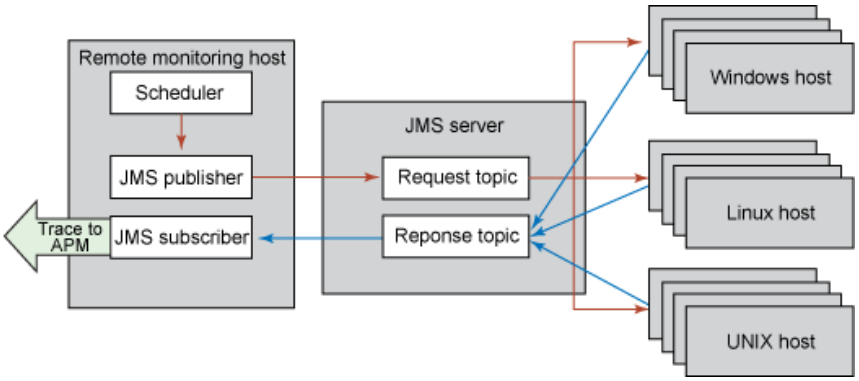
`org.runtimemonitoring.spring.collectors.shell.jms.RemoteJMSShell` というシェルは、シェルのように振る舞う一方、コマンドを JMS を介して送出します (図 6 を参照)。

図 6. 代行コレクター



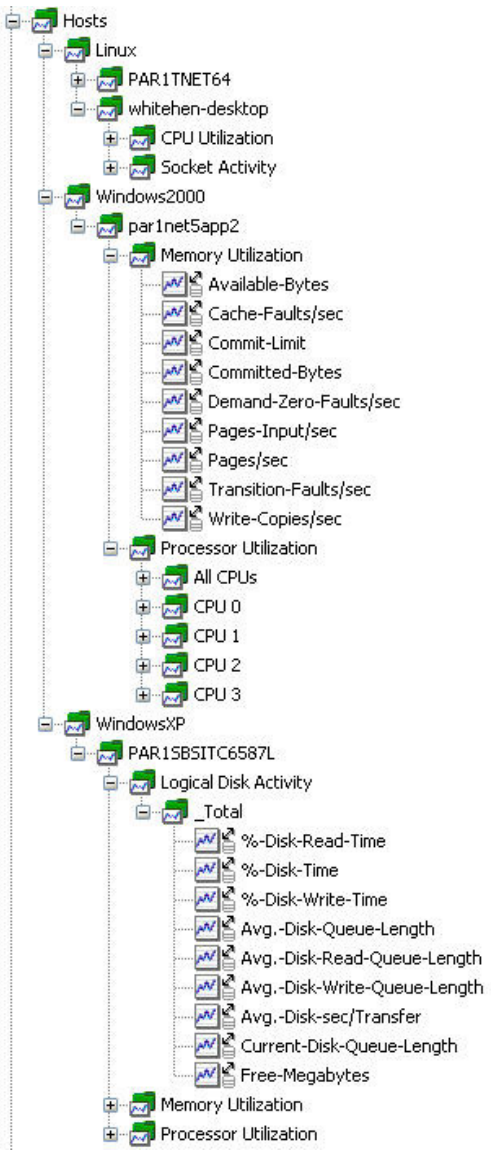
このストラテジーは全体にわたってエージェントをデプロイするのに効果を発揮しそうなので、同じ JMS ベースのエージェントを Java コードにも実装し、JVM をサポートするあらゆる OS にデプロイできるようにします。JMS のパブリッシュ/サブスクライブ型パフォーマンス・データ収集システムを図 7 に示します。

図 7. JMS パブリッシュ/サブスクライブ型の監視



JMS エージェントがどのように機能するかに関しては、さらに細かく区別することができます。上記の例で説明しているパターンは、ターゲット・ホスト上でリクエストをリッスンするエージェントを示しています。このなかで、エージェントは起動してから中央監視システムからのリクエストを受け取るまで、何のアクティビティーも実行しませんが、その一方で、これらのエージェントは独自のスケジュールでデータを収集し、同じ JMS サーバーにパブリッシュすることで、自律的な動作をすることも可能です。こうしたリスニング・エージェントには 2 つの利点があります。まず 1 つは、それぞれのターゲット・ホストにアクセスする代わりに、中央の 1 箇所で収集パラメーターを構成して維持できることです。そしてもう 1 つは (この例には実装されていませんが)、中央の要求側モニターがリクエストを送信することから、特定の既知のサーバーが応答しない場合には、このモニターがアラート条件をトリガーできるということです。図 8 に、サーバーを組み合わせた場合の APM ツリーを示します。

図 8. Windows および Linux サーバーの APM ツリー



NSClient の実装

NSClient と NC_Net は、ほぼ同一のソケット・プロトコルを使用して Windows サーバーからのデータを要求します。nsclient4j (「参考文献」を参照) はこのプロトコルをラップして、JVM を備えた事実上すべてのシステムが WPM メトリックを使用できるようにする Java API です。この記事のサンプル・コードには、/conf/spring-collectors/nsclientj.xml ディレクトリー内に nsclient4j Spring コレクターの一例があります。

`winsar` は単純な初期のプロトタイプであるため、以下のような欠点があります。

- 一部の WPM カウンター (Processor オブジェクトなど) にプログラムによってアクセスすると、空のメトリックや未処理メトリックが生成されます。そのため、CPU 使用率などのメトリックは直接読み取ることができません。必要となるのは、一定の期間中に CPU 使用率を計算するのに十分な回数だけ測定を行う手段です。`winsar` にはこのような機能は含まれていま

せんが、NSClient や NC_Net といった同様のエージェントがそのための手段となります (「参考文献」を参照)。
- 正直なところ、JMS をリモート・エージェントのトランスポートとして使用すると同時に簡潔さも併せ持つには限りがあります。NSClient と NC_Net はどちらも、低レベルながら単純なソケット・プロトコルを使ってデータを送受信します。これらのサービスでの当初目的の 1 つとされていたのは、Windows データを Nagios (Linux プラットフォームに専用同然のネットワーク監視システム) に提供することでした。したがって実際には、クライアント・サイドから見た全体像のなかに Win32 API はありませんでした。

これは前にも述べましたが、`SpringCollector` アプリケーションのブートストラップでは、単一のパラメーターとして構成 XML ファイルが含まれるディレクトリーが使用されます。このディレクトリーは、サンプル・コード・パッケージのルートにある /conf/spring-collectors です。以前の例で使用した具体的なファイルには、以下のものがあります。

- shell-collectors.jmx: シェル・コレクターすべての定義が含まれます。
- management.xml: JMX 管理 Bean と収集スケジューラーが含まれます。
- commandsets.xml: シェル・コレクターのコマンド・セットの定義が含まれます。/commands に置かれた外部 XML ファイルを参照します。
- shells.xml: すべてのシェルの定義が含まれます。
- jms.xml: JMS コネクション・ファクトリーとトピックの定義、そして JNDI (Java Naming and Directory Interface) コンテキストが含まれます。
- groovy.xml: Groovy フォーマッター Bean が含まれます。

OS の監視についての説明は以上です。次は、データベース・システムの監視について取り上げます。

JDBC を使用したデータベース・システムの監視

私はこれまで度々、データベースを監視するという作業は DBA と DBA のツールおよびアプリケーションの独占領域だと確信している人々を目にしてきました。しかし、サイロ型ではないパフォーマンスおよび可用性データの中央 APM リポジトリーを目的とするならば、統合 APM から監視して DBA の作業を補完するのが理にかなっています。そこで、メトリックの `JDBCCollector` という Spring コレクターを使用して、他の方法ではまったく監視されることがないかもしれないながらも、メトリックの一式に加えると役立つデータを収集する手法を紹介します。

一般的に検討しなければならないデータ収集のカテゴリーには、以下のものがあります。

- **単純な可用性および応答時間**: これは単純なメカニズムで、定期的にデータベースに接続し、1 つの単純なクエリーを実行して応答時間をトレースするか、接続できなかった場合にはサーバー・ダウンのメトリックをトレースするというものです。接続の失敗は必ずしもデータベースに障害が発生していることを示すわけではありません。しかし、少なくともアプリケーション側に通信の問題があることは明らかです。サイロ型のデータベースの監視ではデータベースの接続性の問題は示されないかもしれませんが、ある場所からサービスに接続できるからといって、別の場所からも接続できるとは限らないということを覚えておくと役立ちます。
- **コンテキスト依存データ**: 第 1 回で説明したコンテキスト依存トレースの概念に立ち返ると、アプリケーション・データの定期的なサンプリングからは有用な情報を引き出すことができ

ます。大抵の場合、データベースでのデータ・アクティビティーのパターンと、アプリケーション・インフラストラクチャーの振る舞いまたはパフォーマンスの間には強い相関関係があります。
- **データベース・パフォーマンス・テーブル**: 多くのデータベースは、内部パフォーマンスおよび負荷メトリックをテーブルまたはビューとして公開したり、あるいはストアード・プロシージャーによって公開したりします。したがって、このデータには JDBC を使って簡単にアクセスすることができます。この領域は明らかに従来の DBA の監視と重複します。けれども通常は、データベース・パフォーマンスはアプリケーション・パフォーマンスと密接に相関させることができるため、この 2 つのメトリックのセットを収集しておきながら統合システムによって相関付けなければ、大きな無駄になります。

`JDBCCollector` は至って単純なコレクターで、基本的な構成内容は 1 つのクエリー、そしてクエリーの結果をトレース名前空間にマッピングする方法を定義する一連のマッピング文です。リスト 12 の SQL を見てください。

リスト 12. SQL クエリーの例

```
SELECT schemaname, relname,
SUM(seq_scan) as seq_scan,
SUM(seq_tup_read) as seq_tup_read
FROM pg_stat_all_tables
where schemaname = 'public'
and seq_scan > 0
group by schemaname, relname
order by seq_tup_read desc
```

上記のクエリーは、テーブルから 4 つの列を選択します。クエリーによって返されるそれぞれの行は、各行のデータの一部となっているトレース名前空間にマッピングします。名前空間は、一連のセグメント名にメトリック名が続いた形であることを念頭に置いてください。マッピングの定義は、これらの値をリテラル、行トークン、またはその両方を使って指定することによって行います。行トークンは、番号が付けられた列に含まれる値を表します (`{2}` など)。セグメントとメトリック名が処理される際に、リテラルはそのまま残されますが、トークンについては、クエリー結果の現在の行に含まれる、対応する列の値と動的に置換されます (図 9 を参照)。

図 9. `JDBCCollector` マッピング

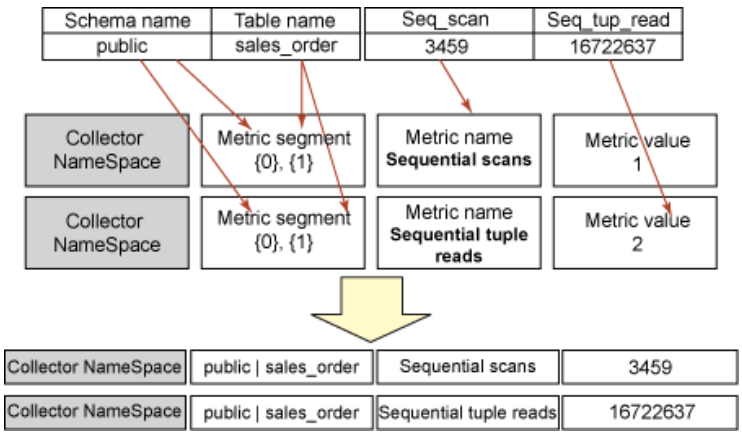


図 9 では、クエリーに対する 1 行での応答を示していますが、マッピング手順は返される各行それぞれに対して定義されるマッピングに対して毎回発生します。セグメントの値は `{1},{2}` なの

で、トレース名前空間のセグメントの部分は `{"public", "sales_order"}` となります。一方メトリック名はリテラルなので、そのまま変わりません。メトリック値は最初のマッピングでは `3459` を表す `1` として定義され、2 番目のマッピングでは `16722637` を表す `2` として定義されています。具体的な実装を見れば、さらに明らかになるはずです。

JDBC によるコンテキスト依存トレース

運用データベースに含まれているアプリケーション・データには、有益で興味深いコンテキスト依存データがあるはずです。アプリケーション・データ自体は必ずしもパフォーマンスに関連したデータではありませんが、アプリケーション・データをサンプリングして Java クラスのパフォーマンス、JVM の正常性、そしてサーバー・パフォーマンス統計を表す今までのメトリックと相関させると、特定の期間中にシステムが実際に行っていた内容を明確に理解することができます。架空の例として、今あなたは極めて活発な e-コマース Web サイトを監視しているとします。カスタマーの注文は、固有の ID と注文が行われた時点のタイムスタンプが付けられて `sales_order` というテーブルに記録されます。発注が行われた頻度は、過去 n 分間に入力されたレコード数をサンプリングすることで算出できます。

この場合もまた、`ITracer` のデルタ機能が役立ちます。特定の時間以降の行数を問い合わせるように `JDBCCollector` をセットアップして、単純にその値を差分としてトレースできるからです。この差分は数あるメトリックの中で、サイトがどれだけビジーなのかを表すメトリックとなります。さらに、貴重な履歴情報にもなります。例えば、サイクルごとの発注が 50 件に達するとデータベースの処理速度が遅くなり始めることがわかっていれば、このように確実で具体的な経験によるデータによって、容量と拡張計画のプロセスは容易になります。

これから実装する例では、`JDBCCollector` は前の例と同じスケジューラー Bean を使用します。また、このコレクターに定義された JDBC DataSource は第 2 回で説明したものとまったく同じです。この例のデータベース・コレクターは、/conf/spring-collectors/jdbc-collectors.xml ファイルに定義されています。リスト 13 に、最初のコレクターを記載します。

リスト 13. 発注頻度のコレクター

```
<bean name="OrderFulfilmentRateLast5s"
  class="org.runtimemonitoring.spring.collectors.jdbc.JDBCCollector"
   init-method="springStart">
   <property name="dataSource" ref="RuntimeDataSource" />
   <property name="scheduler" ref="CollectionScheduler" />
   <property name="query">
      <value><![CDATA[
   select count(*) from sales_order
   where order_date > ? and order_date < ?
   ]]></value>
   </property>
   <property name="logErrors" value="true" />
   <property name="tracingNameSpace" value="Database Context" />
   <property name="objectName"
      value="org.runtime.db:name=OrderFulfilmentRateLast5s,type=JDBCCollector" />
   <property name="frequency" value="5000" />
   <property name="binds">
      <map>
         <entry><key><value>1</value></key><ref bean="RelativeTime"/></entry>
         <entry><key><value>2</value></key><ref bean="CurrentTime"/></entry>
      </map>
   </property>
   <property name="queryMaps">
      <set>
```

```
         <bean class="org.runtimemonitoring.spring.collectors.jdbc.QueryMap">
            <property name="valueColumn" value="0"/>
            <property name="segments" value="Sales Order Activity"/>
            <property name="metricName" value="Order Rate"/>
            <property name="metricType" value="SINT"/>
         </bean>
      </set>
   </property>
</bean>


<bean name="RelativeTime"
   class="org.runtimemonitoring.spring.collectors.jdbc.RelativeTimeStampProvider">
   <property name="period" value="-5000" />
</bean>

<bean name="CurrentTime"
   class="org.runtimemonitoring.spring.collectors.jdbc.RelativeTimeStampProvider">
   <property name="period" value="10" />
</bean>
```

この例でのコレクター Bean の名前は `OrderFulfilmentRateLast5s` で、クラスは `org.runtimemonitoring.spring.collectors.jdbc.JDBCCollector` です。標準の `scheduler` コレクターが JDBC DataSource への参照であることから、この `scheduler` コレクターが `RuntimeDataSource` に注入されます。`query` が定義するのは、実行される SQL です。SQL クエリーはリテラルをパラメーターとして使用することも、この例でのようにバインド変数を使用することもできます。`order_date` の範囲を指定する 2 つの値は SQL 構文で簡単に表現できるはずなので、バインド変数を使用したこの例は不自然に見えますが、外部の値を提供しなければならない場合にはバインド変数を使用するのが一般的です。

外部の値をバインドできるようにするには、`org.runtimemonitoring.spring.collectors.jdbc.IBindVariableProvider` インターフェースを実装してから、そのクラスを Spring 管理対象 Bean として実装する必要があります。この例で使用しているのは `org.runtimemonitoring.spring.collectors.jdbc.RelativeTimeStampProvider` の 2 つのインスタンスで、この Bean は現行タイムスタンプのオフセットを、渡された `period` プロパティーによって指定します。このプロパティーを持つ 2 つの Bean のうち、`RelativeTime` は現在の時刻から 5 秒を引いた値を返し、`CurrentTime` は「現在」に 10 ミリ秒を加えた値を返します。これらの Bean への参照をコレクター Bean に注入するための手段は、`binds` プロパティー、つまりマップです。このマップの各エントリーの値は、対象とする SQL 文のバインド変数と一致していなければなりません。そうでないと、エラーや予期しない結果となってしまします。

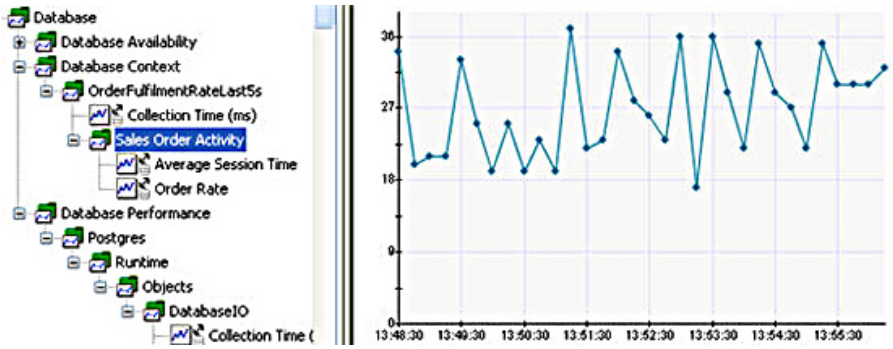
これらのバインド変数は、実質的には過去の約 5 秒間にシステムに入力された販売注文の数を取り込むために使用しています。これは実稼働テーブルに対してかなりのクエリーを行うことになるため、データベースに過剰な負荷をかけないように収集頻度、そして時間枠 (つまり、`Relative` の period プロパティー) を調整しなければなりません。これらの設定を正しく調整できるように、コレクターは収集時間を APM システムにトレースし、経過時間をクエリーのオーバーヘッドの基準として使用できるようにしています。さらに高度なコレクター実装を行うとすれば、監視用クエリーの経過時間が増えるにつれて収集頻度を少なくするという方法が考えらます。

上記に記載したマッピングは、`org.runtimemonitoring.spring.collectors.jdbc.QueryMap` タイプの内部 Bean を使用する `queryMaps` プロパティーによって定義されます。この Bean には、以下の単純な 4 つのプロパティーがあります。

- **valueColumn**: 各行でトレース値としてバインドしなければならない列のゼロを基準としたインデックス。この例では、count(*) の値をバインドしています。
- **segments**: トレース名前空間のセグメント。単一リテラルとして定義されます。
- **metricName**: メトリック名のトレース名前空間。同じくリテラルとして定義されます。
- **metricType**: ITracer メトリック・タイプ。スティッキー int として定義されます。

このコレクターでは、1 回のクエリーの実行によって複数の値をトレースするという前提で、コレクターごとに複数の queryMap を定義することができます。次に紹介する例では、rowToken を使って戻りデータの値をトレース名前空間に注入します。現行の例ではリテラルを使用していますが、同じクエリーを使った例として、クエリーを select count(*), 'Sales Order Activity', 'Order Rate' from sales_order where order_date > ? and order_date < ? に変更することも可能です。このようにすれば、目的のセグメントとメトリック名がクエリーで返されるようになります。segments を {1} に metricName を {2} にすれば、これらの値をマッピングすることができます。さらに拡張した例として、metricType がデータベースから提供されることも考えられますが、その値も rowToken で表すことができます。図 10 に、このように収集されたメトリックの APM ツリーを示します。

図 10. 販売注文頻度の監視



データベース・パフォーマンスの監視

同じプロセスを使えば、JDBCCollector はデータベース・パフォーマンス・ビューからパフォーマンス・データを取得してトレースすることもできます。PostgreSQL を使用するこの例の場合、統計ビューと呼ばれるこれらのテーブルの名前には pg_stat という接頭辞が付きます。他の多くのデータベースにも同じようなビューがあり、JDBC を使ってアクセスすることができます。この例では、前と同じ活発な e-コマース・サイトを使用し、JDBCCollector がテーブル、および最も頻繁にアクセスされている上位 5 つのテーブルでの索引を使ったアクティビティーを監視するようにセットアップします。そのための SQL はリスト 14 のとおりです。

リスト 14. テーブルおよび索引を使ったアクティビティーの監視

```
<bean name="DatabaseIO"
   class="org.runtimemonitoring.spring.collectors.jdbc.JDBCCollector"
   init-method="springStart">
   <property name="dataSource" ref="RuntimeDataSource" />
   <property name="scheduler" ref="CollectionScheduler" />
   <property name="availabilityNameSpace"
      value="Database,Database Availability,Postgres,Runtime" />
   <property name="query">
      <value><![CDATA[
   SELECT schemaname, relname, SUM(seq_scan) as seq_scan,
```

```
   SUM(seq_tup_read) as seq_tup_read,
   SUM(idx_scan) as idx_scan, SUM(idx_tup_fetch) as idx_tup_fetch,
   COALESCE(idx_tup_fetch,0) + seq_tup_read
      + seq_scan + COALESCE(idx_scan, 0) as total
   FROM pg_stat_all_tables
   where schemaname = 'public'
   and (COALESCE(idx_tup_fetch,0) + seq_tup_read
      + seq_scan + COALESCE(idx_scan, 0)) > 0
   group by schemaname, relname, idx_tup_fetch,
      seq_tup_read, seq_scan, idx_scan
   order by total desc
   LIMIT 5 ]]>
      </value>
   </property>
   <property name="tracingNameSpace"
     value="Database,Database Performance,Postgres,Runtime,Objects,{beanName}"
   />
   <property name="frequency" value="20000" />
   <property name="queryMaps">
    <set>
    <bean class="org.runtimemonitoring.spring.collectors.jdbc.QueryMap">
      <property name="valueColumn" value="2"/>
      <property name="segments" value="{0},{1}"/>
      <property name="metricName" value="SequentialScans"/>
      <property name="metricType" value="SDLONG"/>
    </bean>
    <bean class="org.runtimemonitoring.spring.collectors.jdbc.QueryMap">
      <property name="valueColumn" value="3"/>
      <property name="segments" value="{0},{1}"/>
      <property name="metricName" value="SequentialTupleReads"/>
      <property name="metricType" value="SDLONG"/>
    </bean>
    <bean class="org.runtimemonitoring.spring.collectors.jdbc.QueryMap">
      <property name="valueColumn" value="4"/>
      <property name="segments" value="{0},{1}"/>
      <property name="metricName" value="IndexScans"/>
      <property name="metricType" value="SDLONG"/>
    </bean>
    <bean class="org.runtimemonitoring.spring.collectors.jdbc.QueryMap">
      <property name="valueColumn" value="5"/>
      <property name="segments" value="{0},{1}"/>
      <property name="metricName" value="IndexTupleReads"/>
      <property name="metricType" value="SDLONG"/>
    </bean>
    </set>
</property>
</bean>
```

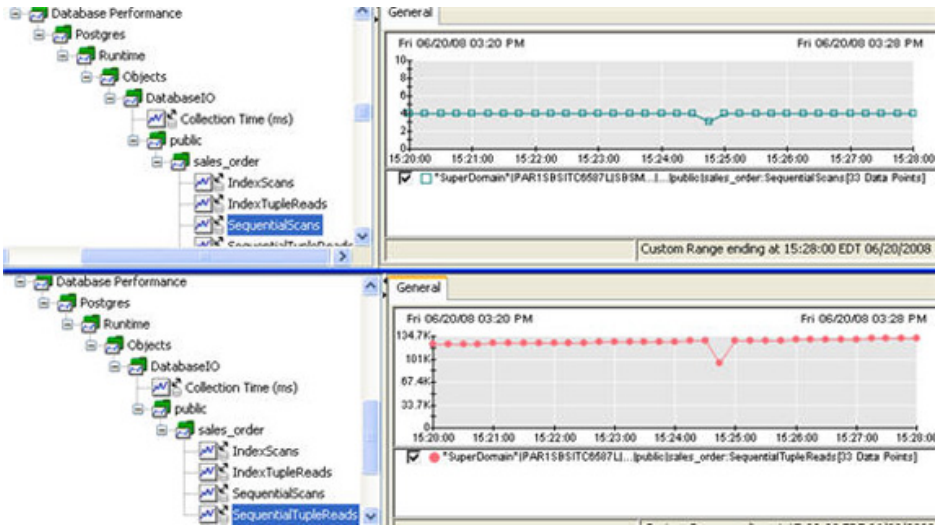
クエリーでは最も頻繁にアクセスされている 5 つのテーブルについて、以下の値を 20 秒間隔で取得します。

- データベース・スキーマの名前
- テーブルの名前
- 連続スキャンの合計回数
- 連続スキャンによって取得されたタプルの合計数
- 索引スキャンの合計回数
- 索引スキャンによって取得されたタプルの合計数

最後の 4 つの列ではいずれも常に値が増加されることから、スティッキー・デルタの `long` 型である `SDLONG` メトリック型を使用しています。リスト 14 では、この 4 つの列をトレース名前空間にマッピングするために 4 つの QueryMap を構成していることに注意してください。

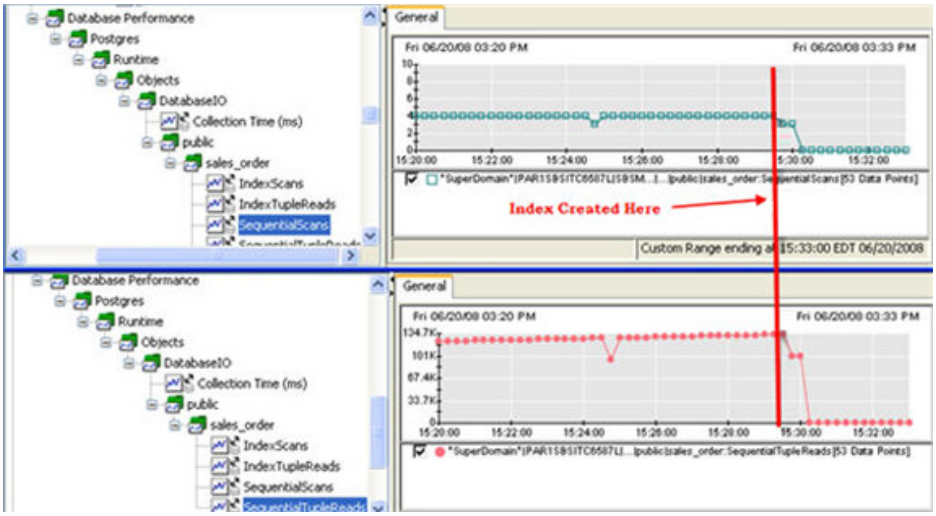
このシナリオでは、`sales_order` テーブルで索引を作成することのない例を考案しました。その結果、監視では連続スキャン (データベース用語ではテーブル・スキャンとも呼ばれます) の回数が多くなります。このようなメカニズムではテーブルのすべての行を読み込むため、データを取得するには非効率的です。これは、タプルの連続読み込み回数にも当てはまります。タプルの連続読み込み回数は、基本的に連続スキャンを使って読み込まれた行数になるからです。行とタプルには大きな違いがありますが、この場合、その違いは関係ありません。詳しい説明については、PostgreSQL のマニュアル・サイト (「参考文献」を参照) を参照してください。APM 表示でこれらの統計を見てみると、私のデータベースには索引がないことが明らかです。この様子を図 11 に示します。

図 11. 連続読み込み



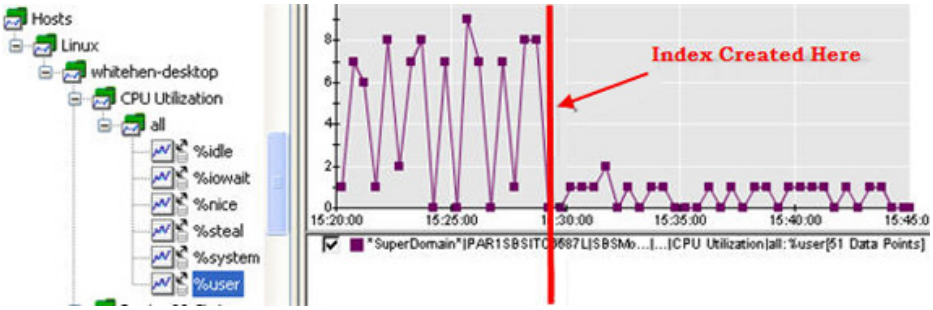
索引がないことに気付くと同時に、テーブルに索引を付けるための SQL 文をいくつか実行します。するとまもなく、双方ともに連続読み込みの回数がゼロに減り、ゼロ回だった索引操作がアクティブになります。この様子を図 12 に示します。

図 12. 索引を作成した後



索引の作成は、システム全体にわたる波及効果があります。この効果によって目に見えて安定化するもう 1 つのメトリックは、データベース・ホストのユーザー CPU 使用率です。図 13 を見てください。

図 13. 索引作成後の CPU 使用率



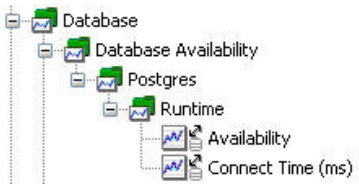
データベースの可用性

最後に取り上げる JDBC の特徴は、可用性が比較的わかりやすいことです。最も単純な形でのデータベース可用性は、標準 `JDBCCollector` にオプションとして含まれています。`availabilityNameSpace` 値を設定してコレクターを構成すると、コレクターは構成された名前空間に以下の 2 つのメトリックをトレースします。

- **可用性**: データベースに接続可能な場合は `1`、接続できない場合は `0`
- **接続時間**: 接続するまでの経過時間

データ・ソースまたは接続プールを使用して接続を行っている場合、接続時間は極めて短いはずです。ただし、大抵の JDBC 接続プール・システムは構成可能な SQL 文を実行してから接続を分配することがあるため、テストするのは当然のことと言えます。大きな負荷がかかっていると接続を一瞬にして取得できない場合があるので、別の手段として `JDBCCollector` に個別のデータ・ソースをセットアップし、それを可用性のテスト専用にすることもできます。その場合、この個別データ・ソースは、すべてのポーリング・サイクルが新しい接続を開始するように、接続を一切プールしないように構成することができます。図 14 に示す可用性チェック APM ツリーは、私の PostgreSQL ランタイム・データベースを対象としたものです。`availabilityNameSpace` プロパティーの使用例については、リスト 14 を参照してください。

図 14. ランタイム・データベースの可用性チェック



特定のステータスであることを判断するために、複数の連鎖したクエリーが必要になる場合を目にしたことがあります。例えば、最終ステータスであることを判断するにはデータベース A に対してクエリーを行わなければならないのに、それに必要なパラメーターは、データベース B に対するクエリーでしか決定できないというような場合です。このような事態に対処するには、以下の点に特に注意して 2 つの `JDBCCollector` を使用してください。

- 先に行う (データベース B に対する) クエリーは、スケジュールを持たないという点で不活性クエリーとして構成されます (収集頻度がゼロということは、スケジュールがないことを意味します)。`JDBCCollector` のインスタンスは `IBindVariableProvider` も実装するため、バインド変数を指定して別のコレクターとバインドすることができます。
- 2 番目のコレクターは最初のコレクターをバインドとして定義し、最初のクエリーの結果にバインドします。

データベースの監視についての説明は以上です。付け加えておかなければならない点として、このセクションではもっぱら JDBC インターフェースを使ったデータベースの監視に重点を置きましたが、典型的なデータベースを完全に監視するには、そのデータベースが常駐する OS の監視、データベースの個々のプロセスまたはプロセスの集合の監視、そしてデータベース・サービスにアクセスするために必要なネットワーク・リソースの監視も必要になります。

JMS およびメッセージング・システムの監視

このセクションでは、メッセージング・サービスの正常性とパフォーマンスを監視する手法を説明します。JMS を実行するメッセージング・サービス (MOM (Message-Oriented Mdleware) とも呼ばれます) は、多くのアプリケーションで重要な役割を果たします。そのため、他のあらゆるアプリケーション依存関係と同じく監視しなければなりません。メッセージング・サービスには多くの場合、非同期の、つまり「firend-for」の呼び出しポイントがありますが、これらのポイントを監視するのは多少厄介な作業になり得ます。非常に短時間でサービスへの呼び出しが頻繁にディスパッチされるため、さまざまな観点から、サービスが正常に実行しているように見えるからです。ですが、上流にボトルネックが隠されていて、メッセージが次の宛先に転送されるまでに時間がかかったり、あるいはまったく転送されなかったりする可能性があります。

大抵のメッセージング・サービスは JVM 内で行われるか、またはホスト (あるいはホストのグループ) での 1 つ以上のネイティブ・プロセスで行われるため、監視ポイントには対象とするサービスと共通するポイントがいくつか含まれます。そのようなポイントとして考えられるのは、標準 JVM JMX 属性、サポート・ホスト上の監視リソース、ネットワークの応答性、そしてサービスのプロセス特性 (メモリー・サイズや CPU 使用率) です。

この後、4 つのカテゴリーに分けてメッセージング・サービスの監視について概説します。そのうちの 3 つは JMS に特有のカテゴリーで、残りの 1 つは独自仕様の API に関連します。

- サービスのスループット・パフォーマンスを測定するために、コレクターはサービスに対し定期的に一連の合成テスト・メッセージを送信し、メッセージが返されるのを待機します。送信と受信の経過時間、そして合計往復時間が測定され、その測定値が障害やタイムアウトなどの結果と併せてトレースされます。
- 多くの Java ベースの JMS 製品は、JMX によってメトリックおよび監視ポイントを公開するため、Spring コレクターを使用した JMX の監視の実装について再び簡単に取り上げます。
- 一部のメッセージング・サービスでは、メッセージング・ブローカー管理用に独自仕様の API を提供しています。これらの API には通常、実行中サービスのパフォーマンス・メトリックを抽出するための機能が組み込まれています。
- 以上のオプションがいずれも使用できない場合は、`javax.jms.QueueBrowser` などの標準 JMS 構成体を使って有益なメトリックを取得することができます。

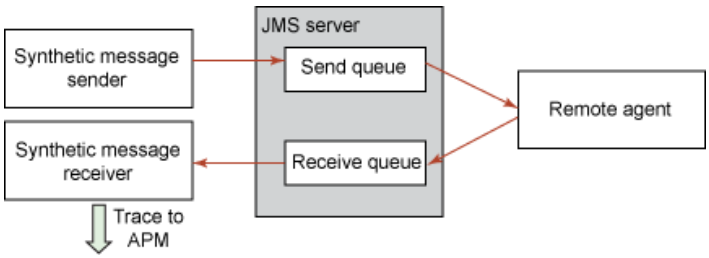
合成テスト・メッセージによるメッセージング・サービスの監視

合成メッセージで前提となるのは、対象となるメッセージング・サービスに対するテスト・メッセージの送受信をスケジュールし、メッセージの送信時間、受信時間、そして合計往復時間を測定することです。メッセージの戻りを計画し、さらに可能な場合にはリモート側からメッセージが配信されるまでの応答時間を測定するには、以下のタスクだけを専門に行うリモート・エージェントをデプロイするのが最適なソリューションとなります。

1. 中央モニターのテスト・メッセージをリッスンする
2. テスト・メッセージを受信する
3. 受信した各メッセージにタイムスタンプを追加する
4. テスト・メッセージをメッセージング・サービスに再送信して中央モニターに返す

中央モニターがメッセージを受け取ると、返されたメッセージを分析してプロセス内の各ホップに費やされた時間を導き出し、その結果を APM システムにトレースします。この流れを図 15 に示します。

図 15. 合成メッセージ



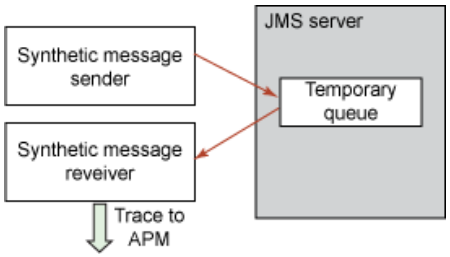
この方法は監視領域の大部分をカバーするものの、以下の欠点もあります。

- リモート・エージェントをデプロイして管理しなければなりません。
- テスト・メッセージを送信するために追加のキューをメッセージング・サービスに作成する必要があります。
- 一部のメッセージング・サービスではファースト・クラスのキューをオンザフライで動的に作成できますが、ほとんどのメッセージング・キューでは管理インターフェースや管理 API を使って手動でキューを作成しなければならなくなります。

JMS に固有の代わりの手段 (ただし、他のメッセージング・システムでも同様の手段があるかもしれません) としてここで説明するのは、一時キューまたはトピックを使用するという方法です。一時キューは標準 JMS API によってオンザフライで作成できるため、管理の介入は必要ありません。これらの一時構成体には、元の作成者を除き、他のすべての JMS 参加者には見えないという利点もあります。

このシナリオでは、起動時に一時キューを作成する `JMSCollector` を使用します。スケジューラーがプロンプトを出すと、このコレクターはターゲット JMS サービスの一時キューに多数のテスト・メッセージを送信し、それから戻ってきたメッセージを受信します。これは実質上、JMS サーバーのスループットをテストすることになるため、具体的なキューを作成したり、リモート・エージェントをデプロイしたりする必要はありません。このシナリオを図 16 に示します。

図 16. 一時キューを使用した合成メッセージ



このシナリオでの Spring コレクター・クラスは `org.runtimemonitoring.spring.collectors.jms.JMSCollector` です。この構成の依存関係は至って単純で、ほとんどの依存関係のセットアップはすでにこれまでの例で完了しています。JMS 接続性に必要な JMS `javax.jms.ConnectionFactory` を取得するために使用する Spring Bean は、Windows の WPM を収集する例で JMS 接続ファクトリーを取得するために定義したものと同じです。復習としてもう一度説明すると、JMS 接続ファクトリーの取得に必要なのは、ターゲット JMS サービスと JNDI 接続を行う `org.springframework.jndi.JndiTemplate` 型の Spring Bean のインスタンスが 1 つ、そしてその JNDI 接続を使用して JMS 接続ファクトリーをルックアップする `org.springframework.jndi.JndiObjectFactoryBean` 型の Spring Bean のインスタンスが 1 つです。

合成メッセージのペイロードを柔軟に構成できるよう、`JMSCollector` は `org.runtimemonitoring.spring.collectors.jms.ISyntheticMessageFactory` というインターフェースの一連の実装を使用して構成されています。このインターフェースを実装するオブジェクトが、テスト・メッセージの配列を提供します。コレクターは構成された各ファクトリーを呼び出し、提供されたメッセージを使用して往復テストを実行します。このようにして、メッセージ・サイズとメッセージ・カウントがさまざまに異なるペイロードを使って JMS サービスのスループットをテストできるようにしています。

それぞれの `ISyntheticMessageFactory` には、`JMSCollector` がトレース名前空間に追加する際に使用する構成可能な任意の名前があります。リスト 15 に、完全な構成を記載します。

リスト 15. 合成メッセージの `JMSCollector`

```
<!-- The JNDI Provider -->
<bean id="jbossJndiTemplate" class="org.springframework.jndi.JndiTemplate">
   <property name="environment"><props>
      <prop key="java.naming.factory.initial">
         org.jnp.interfaces.NamingContextFactory
      </prop>
      <prop key="java.naming.provider.url">
         localhost:1099
      </prop>
      <prop key="java.naming.factory.url.pkgs">
         org.jboss.naming:org.jnp.interfaces
      </prop>
   </props></property>
</bean>

<!-- The JMS Connection Factory Provider -->
<bean id="RealJMSConnectionFactory"
   class="org.springframework.jndi.JndiObjectFactoryBean">
   <property name="jndiTemplate" ref="jbossJndiTemplate" />
   <property name="jndiName" value="ConnectionFactory" />
```

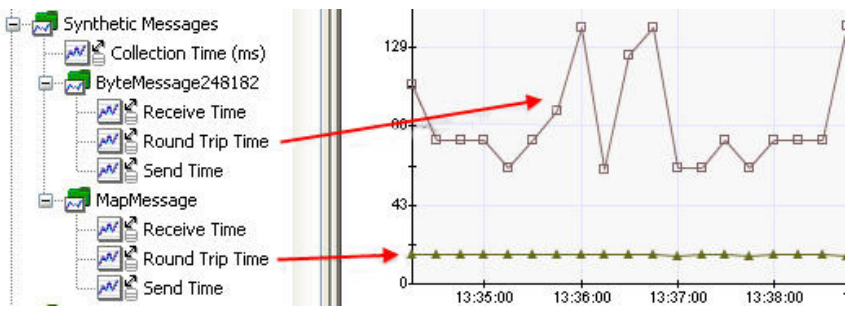
```
</bean>

<!-- A Set of Synthetic Message Factories -->
<bean id="MessageFactories" class="java.util.HashSet">
   <constructor-arg><set>
     <bean
      class="org.runtimemonitoring.spring.collectors.jms.SimpleSyntheticMessageFactory">
         <property name="name" value="MapMessage"/>
         <property name="messageCount" value="10"/>
      </bean>
     <bean
     class="org.runtimemonitoring.spring.collectors.jms.ByteArraySyntheticMessageFactory">
         <constructor-arg type="java.net.URL"
       value="file:///C:/projects3.3/RuntimeMonitoring/lib/jta26.jar"/>
         <property name="name" value="ByteMessage"/>
         <property name="messageCount" value="1"/>
      </bean></set>
   </constructor-arg>
</bean>

<!-- The JMS Collector -->
<bean id="LocalJMSSyntheticMessageCollector"
   class="org.runtimemonitoring.spring.collectors.jms.JMSCollector"
   init-method="springStart">
   <property name="scheduler" ref="CollectionScheduler" />
   <property name="logErrors" value="true" />
   <property name="tracingNameSpace" value="JMS,Local,Synthetic Messages" />
   <property name="frequency" value="5000" />
   <property name="messageTimeOut" value="10000" />
   <property name="initialDelay" value="3000" />
   <property name="messageFactories" ref="MessageFactories"/>
   <property name="queueConnectionFactory" ref="RealJMSConnectionFactory"/>
</bean>
```

リスト 15 には、以下の 2 つのメッセージ・ファクトリーが実装されています。

- `javax.jms.MapMessage`。現行 JVM のシステム・プロパティーを各メッセージのペイロードにロードするファクトリーです。サイクルごとに 10 のメッセージを送信するように構成されています。
- `javax.jms.ByteMessage`。JAR ファイルのバイトを各メッセージのペイロードにロードするファクトリーです。サイクルごとに 10 のメッセージを送信するように構成されています。

図 17 に、合成メッセージ監視用の APM ツリーを示します。`javax.jms.ByteMessage` メッセージ・ファクトリー名の末尾にはバイト・ペイロードのサイズが追加されていることに注意してください。

図 17. 一時キューを使用した合成メッセージの APM ツリー



JMX によるメッセージング・サービスの監視

JBossMQ や ActiveMQ などのメッセージング・サービスは、JMX によって管理インターフェースを公開します。JMX ベースの監視については第 1 回で説明しましたが、もう一度簡単に `org.runtimemonitoring.spring.collectors.jmx.JMXCollector` クラスをベースとした Spring コレクターの概要と、このコレクターを使用して JBossMQ インスタンスを監視する方法を説明します。JMX は一定した標準なので、JMX が公開するあらゆるメトリックの監視には同じプロセスを幅広く適用することができます。

`JMXCollector` が持つ依存関係は、以下の 2 つです。

- ローカル JBossMQ を対象とした `javax.management.MBeanServerConnection` は、`LocalRMIAdaptor` という名前の Bean によって提供されます。この場合、JBoss `org.jboss.jmx.adaptor.rmi.RMIAdaptor` の JNDI ルックアップを実行することによって接続が取得されます。他のプロバイダーを取得するのは簡単で (適用可能な認証クレデンシャルを提供できるという前提)、Spring `org.springframework.jmx.support` パッケージが `MBeanServerConnection` のさまざまな実装を取得する数々のファクトリー Bean を提供します (「参考文献」を参照)。
- 収集 Bean にパッケージ化された JMX 収集属性のプロファイルには、`org.runtimemonitoring.spring.collectors.jmx.JMXCollection` のインスタンスが含まれます。これらのインスタンスは、どの属性を収集するかを `JMXCollector` に指示するディレクティブです。

`JMXCollection` クラスには、JMX モニターに共通の属性があります。基本的な構成プロパティーは以下のとおりです。

- **`targetObjectName`**: 収集対象とする `MBean` の完全な JMX `ObjectName` 名ですが、ワイルドカードにすることもできます。コレクターは JMX エージェントに対して、ワイルドカードのパターンと一致するすべての `MBean` を得られるようにクエリーを実行してから、それぞれのデータを収集します。
- **`segments`**: 収集されたメトリックがトレースされる APM トレース名前空間のセグメント。
- **`metricNames`**: 各 `MBean` 属性をマッピングする必要があるメトリック名の配列。または、`MBean` によって提供される属性名を使用するようにコレクターに指示する単一の `*` 文字。
- **`attributeNames`**: 対象とする各 `MBean` から収集する必要のある `MBean` 属性名の配列。
- **`metricTypes`** または **`defaultMetricType`**: 各属性に使用するメトリック・タイプの配列。または、すべての属性に共通して適用する 1 つのメトリック・タイプ。

MBean `ObjectName` をワイルドカードに設定すると、実質上、個々のターゲットごとにモニターの構成をしなくても監視ターゲットを発見できるようになるため、このオプションは強力です。JMS キューの場合、JBossMQ は各キューに個別の MBean を作成します。つまり、各キューに入れられたメッセージ数 (キューの深さと呼ばれます) を監視するには、単に `jboss.mq.destination:service=Queue,*` のような汎用ワイルドカードを指定するだけで、JMS キュー MBean のすべてのインスタンスが収集されることになります。ただし、キューの名前を動的に判断するという問題が新たに持ち上がってきます。これらのオブジェクトはオンザフライで検出されるからです。この場合、検出された MBean の `ObjectName name` プロパティーの値でキューの名前を判断します。例えば、検出される MBean のオブジェクト名は `jboss.mq.destination:service=Queue,name=MyQueue` のようになります。したがって、検出さ

れたオブジェクトからトレース名前空間へのマッピングを行って、トレースされたメトリックを各ソースから区別できるようにする方法が必要となりますが、それには `JDBCCollector` での `rowToken` のようにトークンを使用することで対処します。`JMXCollector` でサポートされるトークンは以下のとおりです。

- **`{tar-property:name}`**: このトークンは、ターゲット MBean の `ObjectName` にある名前付きプロパティーと置換されます。例: `{tar-property:name}`.
- **`{this-property:name}`**: このトークンは、コレクターの `ObjectName` にある名前付きプロパティーと置換されます。例: `{this-property:type}`.
- **`{tar-domain:index}`**: このトークンは、ターゲット MBean の `ObjectName` ドメインにあるインデックス付きセグメントと置換されます。例: `{target-domain:2}`.
- **`{this-domain:index}`**: このトークンは、コレクターの `ObjectName` ドメインにあるインデックス付きセグメントと置換されます。例: `{target-domain:0}`.

リスト 16 に、JBossMQ `JMXCollector` の XML 構成を簡略化して記載します。

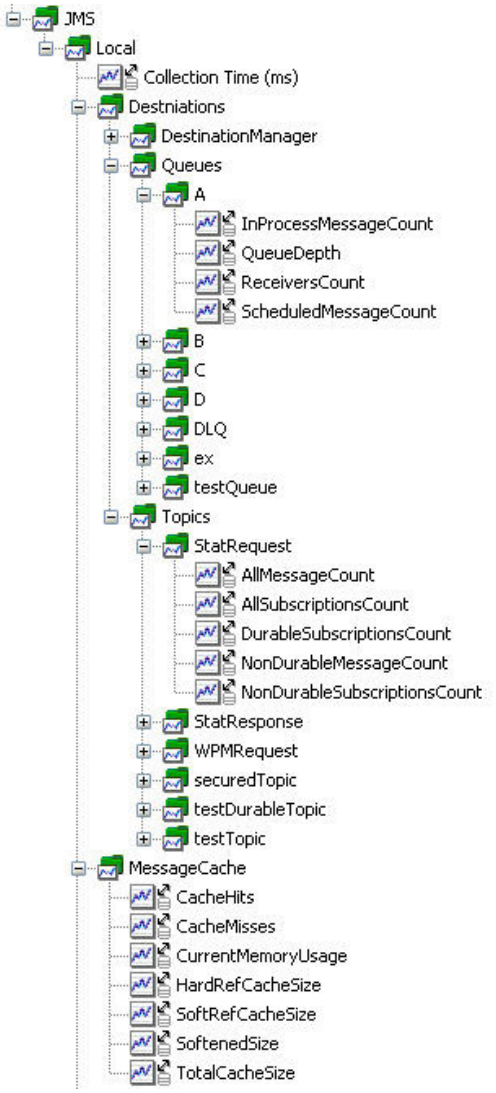
リスト 16. ローカル JBossMQ の `JMXCollector`

```
<!-- The JBoss RMI MBeanServerConnection Provider -->
<bean id="LocalRMIAdaptor"
   class="org.springframework.jndi.JndiObjectFactoryBean">
   <property name="jndiTemplate" ref="jbossJmxJndiTemplate" />
   <property name="jndiName" value="jmx/invoker/RMIAdaptor" />
</bean>

<!-- The JBossMQ JMXCollection Profile -->
<bean id="StandardJBossJMSProfile"
   class="org.runtimemonitoring.spring.collectors.collections.InitableHashSet"
   init-method="init" >
   <constructor-arg><set>
      <bean class="org.runtimemonitoring.spring.collectors.jmx.JMXCollection">
         <property name="targetObjectName" value="*:service=Queue,*"/>
         <property name="segments" value="Destinations,Queues,{target-property:name}"/>
         <property name="metricNames" value="*"/>
         <property name="attributeNames"
       value="QueueDepth,ScheduledMessageCount,InProcessMessageCount,ReceiversCount"/>
         <property name="defaultMetricType" value="SINT"/>
       </bean>
       <bean class="org.runtimemonitoring.spring.collectors.jmx.JMXCollection">
         <property name="targetObjectName" value="jboss.mq:service=DestinationManager"/>
         <property name="segments" value="Destniations,{target-property:service}"/>
         <property name="metricNames" value="*"/>
         <property name="attributeNames" value="ClientCount"/>
         <property name="defaultMetricType" value="SINT"/>
       </bean>
       <!-- MBeans Also Included: Topics, ThreadPool, MessageCache -->
       </set>
    </constructor>
</bean>

<!-- The JMXCollector for local JBoss MQ Server -->
<bean id="LocalJBossCollector"
   class="org.runtimemonitoring.spring.collectors.jmx.JMXCollector"
      init-method="springStart">
   <property name="server" ref="LocalRMIAdaptor" />
   <property name="scheduler" ref="CollectionScheduler" />
   <property name="logErrors" value="true" />
   <property name="tracingNameSpace" value="JMS,Local" />
   <property name="objectName"
      value="org.runtime.jms:name=JMSQueueMonitor,type=JMXCollector" />
```

```
   <property name="frequency" value="10000" />
   <property name="initialDelay" value="10" />
   <property name="jmxCollections" ref="StandardJBossJMSProfile"/>
</bean>
```

図 18 に、`JMXCollector` を使用して JBossMQ サーバーの JMS キューを監視した場合の APM ツリーを示します。

図 18. JBossMQ キューの JMX 監視用の APM ツリー



キュー・ブラウザーを使用した JMS キューの監視

JMS キューを監視する適切な管理 API がない場合には、`javax.jms.QueueBrowser` を使用することができます。キュー・ブラウザーの振る舞いは `javax.jms.QueueReceiver` とほぼ同じですが、異なる点として、このブラウザーはメッセージを取得してもキューから削除せずに、そのまま配信します。キューの深さは通常の場合、重要なメトリックです。多くのメッセージング・システムでは一般にメッセージ送信側の数がメッセージ受信側の数を上回っているものですが、どれだけ釣り合いが取れていないかは、ブローカーのキューに入れられたメッセージの数によって判断

するからです。そのため、他にキューの深さにアクセスする方法がなければ、キュー・ブラウザーを使用するのが最後の手段となりますが、この手法にはいくつもの欠点があります。まず、キュー内のメッセージ数をカウントするには、キュー・ブラウザーがキュー内のすべてのメッセージを取得しなければなりません (その後、破棄する必要もあります)。これは極めて非効率的で、管理 API を使用するよりも収集に遥かに長い時間がかかり、さらに JMS サーバーのリソースにかなりの負担となることも考えられます。キューをブラウズする上でのもう 1 つの側面として、システムがビジーの場合、ブラウズが完了する頃には誤ったカウントになってしまう可能性が高いという点も上げられます。そうは言っても、概算を監視するのであれば、その目的は十分果たせるはずです。また、負荷の高いシステムでは、ある時点で極めて正確に測定されたキューの深さでも、どのみち次の瞬間には使いものにならなくなってしまいます。

キューのブラウズには 1 つの利点があります。それは、キューのメッセージをブラウズする過程のなかで、最も古いメッセージの存続時間を判断できることです。これは最も優れた JMS 管理 API でさえも手に入れるのが難しいメトリックですが、場合によっては重要な監視ポイントになります。例えば重要なメッセージの送信で使用される JMS キューを考えてみてください。メッセージ送信側とメッセージ受信側には典型的な違いがあるため、キューの深さが標準的なポーリングでは 1 つか 2 つのメッセージが取得されるといったようなトラフィックのパターンになります。通常、その原因はわずかな遅延ですが、1 分間隔のポーリングでは、ポーリングごとにキュー内に存在するメッセージは変わってくるはずです。でも、本当にそうでしょうか。キュー内のメッセージが同じでなければ、それは正常な状態ですが、メッセージ送信側とメッセージ受信側の両方が同時に失敗することもあり得ます。その場合、ポーリングをする度にキュー内に同じメッセージがいくつか確認されることになります。このようなシナリオでは、キューの深さと同時に最も古いメッセージの存続時間を監視することで状態が明らかになります。通常のメッセージ存続時間は数秒に満たないものですが、メッセージ送信側とメッセージ受信側が同時に失敗した場合には、2 回のポーリング・サイクルの時間間隔を見るだけで、APM に顕著なデータが現れてきます。

この機能を表しているのが、Spring コレクターの `org.runtimemonitoring.spring.collectors.jmx.JMSBrowserCollector` です。このコレクターには 2 つの構成プロパティーが追加されています。1 つは `JMSCollector` と同じような `javax.jms.ConnectionFactory`、そしてもう 1 つはブラウズするキューの集合です。このコレクターの構成をリスト 17 に記載します。

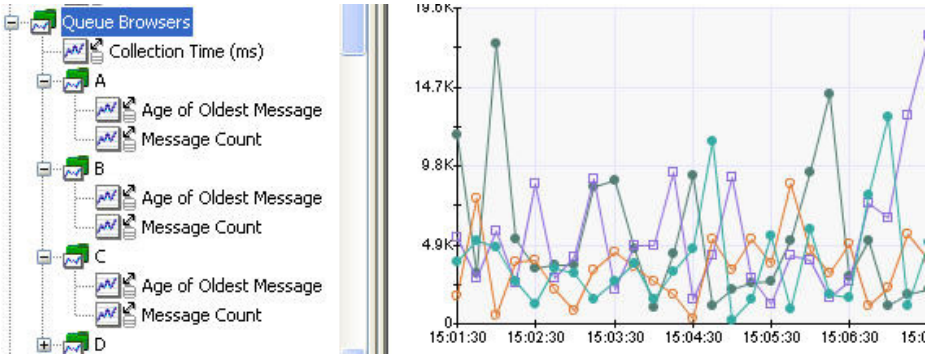
リスト 17. ローカル JBossMQ の `JMSBrowserCollector`

```
<!-- A collection of Queues to browse -->
<bean id="BrowserMonitorQueues" class="java.util.HashSet">
   <constructor-arg>
      <set>
         <bean id="QueueA"
            class="org.springframework.jndi.JndiObjectFactoryBean">
            <property name="jndiTemplate" ref="jbossJndiTemplate" />
            <property name="jndiName" value="queue/A" />
         </bean>
               <bean id="QueueB"
            class="org.springframework.jndi.JndiObjectFactoryBean">
            <property name="jndiTemplate" ref="jbossJndiTemplate" />
            <property name="jndiName" value="queue/B" />
         </bean>
      </set>
   </constructor-arg>
```

```
</bean>

<!-- the JMS Queue Browser -->
<bean id="LocalQueueBrowserCollector"
   class="org.runtimemonitoring.spring.collectors.jms.JMSBrowserCollector"
     init-method="springStart">
   <property name="scheduler" ref="CollectionScheduler" />
   <property name="logErrors" value="true" />
   <property name="tracingNameSpace" value="JMS,Local,Queue Browsers" />
   <property name="frequency" value="5000" />
   <property name="initialDelay" value="3000" />
   <property name="queueConnectionFactory" ref="RealJMSConnectionFactory"/>
   <property name="queues" ref="BrowserMonitorQueues"/>
</bean>
```

図 19 は、このコレクターの APM ツリーです。

図 19. `JMSBrowserCollector` の APM ツリー



テスト・メカニズムとしてループされたロード・スクリプトが 2、3 百のメッセージをループにある各キューに送信します。すべてのループでは、パージされるキューがランダムに選出されているため、各キューの最大メッセージ存続時間は不規則に変わっています。

独自仕様の API を使用したメッセージング・システムの監視

一部のメッセージング・システムには、監視などの管理機能を実装するための独自仕様の API があります。これらのうちのいくつかは、リクエスト/レスポンス・パターンを持つ独自のメッセージング・システムを使って管理リクエストを送信します。例えば ActiveMQ (「参考文献」を参照) には、JMS メッセージング管理 API と JMX 管理 API が用意されています。独自仕様の管理 API を実装するには、カスタム・コレクターが必要です。このセクションでは、WebSphere® MQ (旧称 MQ Series) のコレクターを紹介します。このコレクターが使用するのは、以下の 2 つの API の組み合わせです。

- MS0B: PCF 用 WebSphere MQ Java クラス: この PCF API は、WebSphere MQ の管理 API です。
- コア WebSphere MQ Java クラス: 以前 MA88 と呼ばれていた API は、コア WebSphere MQ Java クラス・ライブラリーに統合されています (「参考文献」を参照)。

2 つの API を使用するのは冗長ですが、この例では、2 つの異なる独自仕様の API を使用する方法を説明します。

この Spring コレクター実装は、`org.runtimemonitoring.spring.collectors.mq.MQCollector`. というクラスです。このクラスは WebSphere MQ サーバーですべてのキューを監視し、

各キューのキューの深さと現在オープンしている入力および出力ハンドルの数を収集します。`org.runtimemonitoring.spring.collectors.mq.MQCollector` の構成はリスト 18 のとおりです。

リスト 18. WebSphere MQ コレクター

```
<bean id="MQPCFAgentCollector"
   class="org.runtimemonitoring.spring.collectors.mq.MQCollector"
      init-method="springStart">
   <property name="scheduler" ref="CollectionScheduler" />
   <property name="logErrors" value="true" />
   <property name="tracingNameSpace" value="MQ, Queues" />
   <property name="frequency" value="5000" />
   <property name="initialDelay" value="3000" />
   <property name="channel" value="SERVER1.QM2"/>
   <property name="host" value="192.168.7.32"/>
   <property name="port" value="50002"/>
</bean>
```

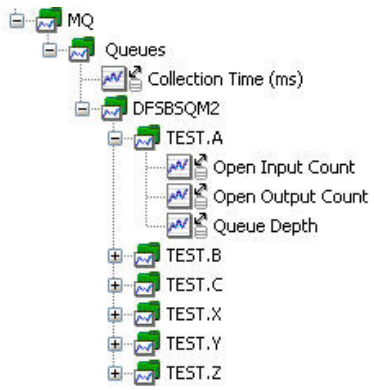
このコレクターに固有の構成プロパティーには以下のものがあります。

- **`host`**: WebSphere MQ サーバーのホスト名の IP アドレス
- **`port`**: WebSphere MQ プロセスが接続をリッスンするポート
- **`channel`**: 接続先の WebSphere MQ チャネル

この例では、認証に関しては何も考慮していないことに注意してください。

図 20 に、`org.runtimemonitoring.spring.collectors.mq.MQCollector` 用に生成した APM ツリーを示します。

図 20. **`MQCollector`** の APM ツリー



これで、メッセージング・サービスの監視についての説明は終わります。前に約束したとおり、次は SNMP による監視について説明します。

SNMP を使用した監視

SNMP は当初、ルーター、ファイアウォール、スイッチなどのネットワーク・デバイス相互での情報交換用アプリケーション層プロトコルとして作成されました。最もよく使われているのは今でもこの機能だと思いますが、SNMP はパフォーマンスおよび可用性を監視するための柔軟かつ

標準化されたプロトコルとしての役割も果たします。監視ツールとしての SNMP とその実装の全容はこの記事でとても説明しきれませんが、監視の分野では SNMP が普遍的になってきているので、この話題についてはある程度、取り上げないわけにはいきません。

SNMP の中心的な構造の 1 つはエージェントです。エージェントは、特定のデバイスを対象とした SNMP リクエストを仲介する役目を持ちます。SNMP は比較的単純で難しくないことから、SNMP エージェントはさまざまなハードウェア・デバイスとソフトウェア・サービスに簡単かつ効率的に組み込むことができます。このような SNMP はアプリケーションのエコシステムに含まれるほとんどのサービスを監視するプロトコルを 1 つに標準化することを約束します。さらに、SNMP は IP アドレスとポートが指定された範囲をスキャンすることによって問題の検出を行うために広く使われています。監視の観点からすると、SNMP を使って中央インベントリーに自動的に監視ターゲットを追加し、更新することで管理オーバーヘッドの削減を実現できます。さまざまな点で、SNMP は JMX によく似ています。明らかな違いはいくつかあるものの、この 2 つの共通点を引き出すことは可能です。このような JMX と SNMP の相互運用性は広くサポートされ、実装されています。表 1 に、共通点をいくつか抜粋します。

表 1. SNMP と JMX の比較

SNMP の構造	対応する JMX の構造
エージェント	エージェントまたは `MBeanServer`
マネージャー	クライアント、`MBeanServerConnection`、プロトコル・アダプター
MIB	`MBean`, `ObjectName`, `MBeanInfo`
OID	`ObjectName`、`ObjectName`+ `Attribute` 名
トラップ	JMX `Notification`
`GET`, `SET`	`getAttribute`, `setAttribute`
`BULKGET`	`getAttributes`

単純な監視という点から見ると、SNMP による問い合わせを実行するときに知っておくべき重要な要素には以下があります。

- **ホスト・アドレス**: ターゲット SNMP エージェントが常駐する IP アドレスまたはホスト名
- **ポート**: ターゲット SNMP エージェントがリッスンするポート。1 つのネットワーク・アドレスが複数の SNMP エージェントの前に置かれる場合があるため、各エージェントがそれぞれ異なるポートでリッスンする必要があります。
- **プロトコル・バージョン**: SNMP プロトコルは何度もリビジョンが行われているため、サポート・レベルはエージェントによって異なります。選択肢は 1、2c、および 3 です。
- **コミュニティー**: SNMP コミュニティーとは、大まかに定義された管理ドメインのことです。SNMP クライアントがエージェントに対してリクエストを送信するには、その SNMP コミュニティーを認識している必要があります。つまり、コミュニティーはある意味、認証の役割も果たします。
- **OID**: メトリックまたはメトリックのグループに固有の ID です。フォーマットはドットで区切られた一連の整数となります。例えば、Linux ホストの 1 Minute Load の場合の SNMP OID は `.1.3.6.1.4.1.2021.10.1.3.1` で、1、5、15 Minute Load からなるメトリックのサブセットの場合の OID は `.1.3.6.1.4.1.2021.10.1.3` となります。

一部のエージェントは、コミュニティーとは別に認証の層を追加で定義することができます。

SNMP API の詳細に入る前に、SNMP メトリックは 2 つの一般的なコマンドライン・ユーティリティー、`snmpget` と `snmpwalk` を使って取得できることに注意してください。前者は 1 つの OID の値を取得し、後者は OID 値のサブセットを取得するユーティリティーです。この点を覚えていれば、いつでも `ShellCollector CommandSet` を拡張して SNMP OID 値をトレースすることができます。リスト 19 に、Linux ホストでの 1、5、および 15 Minute Load を取得してそのまま出力する `snmpwalk` の例と、簡潔な出力にする `snmpwalk` の例を記載します。ここで使用しているのは、バージョン 2c のプロトコルおよび公開コミュニティーです。

リスト 19. `snmpwalk` の例

```
$> snmpwalk -v 2c -c public 127.0.0.1 .1.3.6.1.4.1.2021.10.1.3
UCD-SNMP-MIB::laLoad.1 = STRING: 0.22
UCD-SNMP-MIB::laLoad.2 = STRING: 0.27
UCD-SNMP-MIB::laLoad.3 = STRING: 0.26

$> snmpwalk -v 2c -c public 127.0.0.1 .1.3.6.1.4.1.2021.10.1.3 \
   | awk '{ split($1, name, "::"); print name[2] " " $4}'
laLoad.1 0.32
laLoad.2 0.23
laLoad.3 0.22
```

2 つ目のコマンドは、私の Linux コマンド・セットで簡単に表現することができます (リスト 20 を参照)。

リスト 20. `snmpwalk` コマンドを処理する `CommandSet`

```
<CommandSet name="LinuxSNMP">
   <Commands><Command>
      <shellCommand><![CDATA[snmpwalk -v 2c -c public 127.0.0.1
      .1.3.6.1.4.1.2021.10.1.3 | awk '{ split($1, name, "::"); print name[2] "
      " $4}']]></shellCommand>
         <Extractors><CommandResultExtract>
             <paragraph id="0" name="System Load Summary" header="false"/>
                <columns entryName="0" values="1" offset="0">
                   <namemapping from="laLoad.1" to="1 Minute Load"/>
                    <namemapping from="laLoad.2" to="5 Minute Load"/>
                    <namemapping from="laLoad.3" to="15 Minute Load"/>
                </columns>
                <tracers default="SINT"/>
                <lineSplit>\n</lineSplit>
          </CommandResultExtract></Extractors>
      </Command></Commands>
</CommandSet>
```

SNMP Java API には商用およびオープンソースの両方があります。私が実装した基本的な Spring コレクター、`org.runtimemonitoring.spring.collectors.snmp.SNMPCollector` が使用するのは、joeSNMP というオープンソースの API です (「参考文献」を参照)。このコレクターには、以下の重要な構成プロパティーがあります。

- **`hostName`**: ターゲット・ホストの IP アドレスまたはホスト名です。
- **`port`**: ターゲット SNMP エージェントがリッスンするポート番号です (デフォルトは `161`)。
- **`targets`**: `org.runtimemonitoring.spring.collectors.snmp.SNMPCollection` のインスタンスからなる SNMP OID ターゲットのセットです。この Bean の構成プロパティーには、以下のものがあります。

- **`nameSpace`**: トレース名前空間のサフィックス
- **`oid`**: ターゲット・メトリックの SNMP OID
- **`protocol`**: SNMP プロトコルです。v1 の場合は `0`、v2 の場合は `1` となります (デフォルトは v1)。
- **`community`**: SNMP コミュニティーです (デフォルトは `public`)。
- **`retries`**: 操作を再試行する回数です (デフォルトは `1`)。
- **`timeOut`**: ミリ秒で指定された SNMP 呼び出しのタイムアウトです (デフォルトは `5000`)。

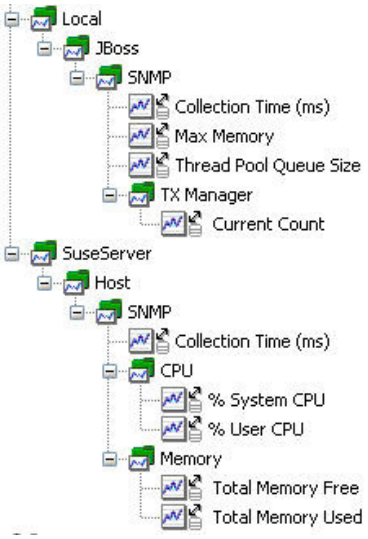
リスト 21 は、ローカル JBoss アプリケーション・サーバーを監視する `SNMPCollector` セットアップの構成例です。

リスト 21. `SNMPCollector` の構成

```
<!-- Defines the SNMP OIDs I want to collect and the mapped name -->
<bean id="JBossSNMPProfile" class="java.util.HashSet">
   <constructor-arg><set>
     <bean class="org.runtimemonitoring.spring.collectors.snmp.SNMPCollection">
             <property name="nameSpace" value="Free Memory"/>
             <property name="oid" value=".1.2.3.4.1.2"/>
     </bean>
     <bean class="org.runtimemonitoring.spring.collectors.snmp.SNMPCollection">
             <property name="nameSpace" value="Max Memory"/>
             <property name="oid" value=".1.2.3.4.1.3"/>
     </bean>
     <bean class="org.runtimemonitoring.spring.collectors.snmp.SNMPCollection">
             <property name="nameSpace" value="Thread Pool Queue Size"/>
             <property name="oid" value=".1.2.3.4.1.4"/>
     </bean>
     <bean class="org.runtimemonitoring.spring.collectors.snmp.SNMPCollection">
             <property name="nameSpace" value="TX Manager, Rollback Count"/>
             <property name="oid" value=".1.2.3.4.1.7"/>
     </bean>
     <bean class="org.runtimemonitoring.spring.collectors.snmp.SNMPCollection">
             <property name="nameSpace" value="TX Manager, Current Count"/>
             <property name="oid" value=".1.2.3.4.1.8"/>
     </bean>
  </set></constructor-arg>
</bean>

<!-- Configures an SNMP collector for my local JBoss Server -->
<bean id="LocalJBossSNMP"
   class="org.runtimemonitoring.spring.collectors.snmp.SNMPCollector"
   init-method="springStart">
   <property name="scheduler" ref="CollectionScheduler" />
   <property name="logErrors" value="true" />
   <property name="tracingNameSpace" value="Local,JBoss,SNMP" />
   <property name="frequency" value="5000" />
   <property name="initialDelay" value="3000" />
   <property name="hostName" value="localhost"/>
   <property name="port" value="1161"/>
   <property name="targets" ref="JBossSNMPProfile"/>
</bean>
```

このコレクターは、構成が冗長であるという点で完全ではありません。また、ランタイムは一括で収集するのではなく、OID ごとに 1 回の呼び出しを行うため効率性にも欠けます。サンプル・コード (「ダウンロード」を参照) の snmp-collectors.xml ファイルにも、Linux サーバーを監視する SNMP コレクターの構成例が記載されているので参照してください。図 21 に、APM システム・メトリック・ツリーを示します。

図 21. `SNMPCollector` の APM ツリー



この段階まで来れば、コレクターをどのように作成するかはおそらく理解できているはずです。環境全体を対象にするには、さまざまなコレクターのタイプが必要になります。興味がある方は、この記事のソース・コードにその他の監視ターゲットのコレクター例が記載されているので参照してください。コレクター例はすべて、`org.runtimemonitoring.spring.collectors` パッケージに含まれています。表 2 は、その要約です。

表 2. その他のコレクター例

収集ターゲット	クラス
Web サービス: セキュア Web サービスの応答時間と可用性をチェック	`webservice.MutualAuthWSClient`
Web サービスおよび URL のチェック	`webservice.NoAuthWSClient`
Apache Web Server のパフォーマンスおよび可用性	`apacheweb.ApacheModStatusCollector`
Ping	`network.PingCollector`
NTop: 詳細なネットワーク統計を収集するユーティリティー	`network.NTopHostTrafficCollector`

データ管理

大規模でビジーな可用性およびパフォーマンス・データ収集システムでとりわけ厄介な問題の 1 つは、収集したデータを共通メトリック・データベースに効率的に存続させる方法です。データベースとパーシスタンス (永続化) メカニズムについての検討事項には、以下のものがあります。

- メトリック・データベースがサポートする履歴メトリックのクエリーは、データを視覚化し、レポートを生成して分析を行う上で必要な迅速さと単純さを備えていなければなりません。
- メトリック・データベースは、データの履歴と細分度を維持してレポートの時間枠、正確さ、そして必要な精度をサポートしなければなりません。
- パーシスタンス・メカニズムは、フロントエンドでの監視の活動に影響を及ぼすことがないよう、うまく並行して機能しなければなりません。

- メトリック・データの取得と保存は、互いに悪影響を与えることなく同時に実行可能でなければなりません。
- データベースに対するデータの要求は、一定期間での集約をサポート可能でなければなりません。
- データベース内のデータを保存する際には、時系列パターンでデータを取得できたり、あるいは複数のデータ・ポイントが同じ有効期間に関連付けられている場合にはそれらのデータ・ポイントが深く関連付けられることを保証できたりする方法で保存する必要があります。

これらの点を考慮すると、以下のことが必要になります。

- 大量のディスク・スペースを備え、優れたパフォーマンスを有するスケーラブルなデータベース
- 効率的な検索アルゴリズムを使用するデータベース。基本的にメトリックは複合名で保存されることから、1 つのソリューションとなるのは複合名をストリングとして保存し、何らかの形のパターン・マッチングを使用してターゲットのクエリー・メトリックを指定することです。多くのリレーショナル・データベースは SQL 構文に統合された正規表現をサポートします。このような正規表現は、複合名によるクエリーには最適ですが、通常は索引を同時に使用することはできないため、処理速度が遅くなりがちです。ただし、多くのリレーショナル・データベースは機能的索引もサポートするので、これを使用すれば、パターン・マッチング検索を使用したクエリーを高速化できるはずです。別の手段としては、データを完全に正規化し、複合名を個々のセグメントに分割するという方法もあります (この後の「正規化とフラット・データベース構造の比較」を参照してください)。
- データベースに対する書き込みの回数と書き込まれるデータ合計量を制限するためのストラテジーは、階層化した一連の集約と融合を実装することです。データがデータベースに書き込まれる前にこのストラテジーを実行するには、メトリックのロールアップ用のバッファーを保持します。一定の期間、保存用にタグが付けられたすべてのメトリックを累積バッファーに書き込み、このバッファーでメトリックの実効開始時刻と実効終了時刻、そしてその期間中のメトリックの平均値、最小値、最大値を保持します。このようにして、メトリック値をデータベースに書き込まれる前にまとめて集約します。一例として、コレクターの収集間隔が 15 秒で集約期間が 1 分だとすると、4 つのメトリックが 1 つに合体されます。この場合、永続化されるデータの細分度がある程度損なわれるのは仕方がないことなので、細分度は低下するけれどもデータ量を少なくするか、あるいはデータ量は多くなるけれども細分度を高くするかのトレードオフとなります。同時に、永続化されたデータだけが集約されるように事前に集約されたメモリー内の循環バッファーのデータをグラフにすることで、メトリックをリアルタイムで視覚化することができます。当然のことながら、実際に永続化するメトリックを選択することは可能です。
- 保存されるデータ量を減らすための別のストラテジーは、未処理メトリック y につき x だけを保存するというサンプリング頻度を実装することです。この場合にも、永続化されるデータの細分度は損なわれますが、集約バッファーを維持するよりは使用するリソース (特にメモリー) は少なくなります。
- メトリック・データを永続化した後に、データをメトリック・データベースにロールアップしてパージすることも可能です。この場合も同じく細分度を犠牲にできる範囲内で、データベース内のデータを例えば 1 時間ごと、1 日ごと、1 週間ごと、あるいは 1 ヶ月ごとにサマ

リー・テーブルにロールアップすることも、未処理データをパージすることも、さらに妥当な数の有効なメトリック・データのセットを維持することもできます。
- データ・パーシスタンス・プロセスのアクティビティーによってデータ・コレクター自体が影響されることがないようにするには、個々のメトリック・データの集合をデータ保存プロセスにフラッシュでき、他のスレッドへ影響を及ぼすことのないバックグラウンド・スレッドを実装することが不可欠です。
- 実効時間枠が異なる複数のデータ・セットから正確な時系列的レポートを作成するのは難しい問題です。例えばグラフの x 軸が時間を表し、y 軸が特定メトリックの値と、一連の異なるソースで同じメトリックについて測定された読み取り値の複数の数列 (線またはプロット) を表すとします。各数列の読み取り値の実効タイムスタンプが顕著に異なる場合には、データを処理してグラフ表示を有効に保たなければなりません。それには、プロットされた数列すべてに共通する最も小さい時間枠にデータを集約するという方法がありますが、やはりこの場合も細分度が失われてしまいます。それよりも遥かに簡単なソリューションは、すべてのメトリック値に共通した一貫性のある実効タイムスタンプを維持することです。これは時系列データベースの 1 つの機能で、なかでもよく使用されている RRDTool は実質上、数列内のさまざまなデータ値に一貫した等間隔のタイムスタンプを施行します。リレーショナル・データベース内で実効タイムスタンプの一貫性を保つには、1 つの統一したスケジューラーに従って、すべてのメトリックのサンプリングを取るのが効果的なストラテジーとなります。例えば、単一のタイマーを 2 分間隔で作動させ、その時点で取り込まれているすべてのメトリックをまとめて「スワイプ」します。こうすれば、すべてに同じタイムスタンプでタグが付けられてから、パーシスターの空いた時間にパーシスタンス用のキューに入れられることになります。

以上の問題それぞれに最適な方法で対処するのは明らかに難問となるので、妥協案を検討しなければなりません。図 22 に、概念的なデータ・コレクターのデータ・パーシスタンスのフローを表します。

図 22. データ管理フロー

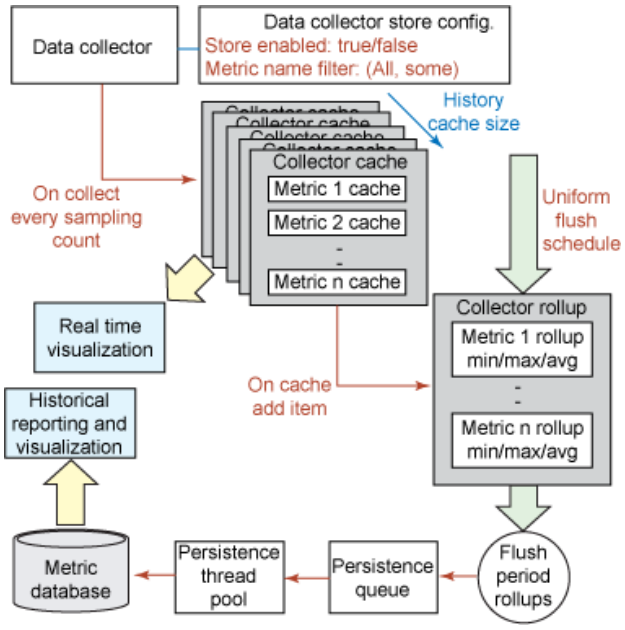


図 22 は、この記事で紹介した Spring コレクターのような概念的なデータ・コレクターからのデータ・フローを表しています。このフローの目的は、個々のメトリック・トレース測定値が永続化されるまで一連の層にプッシュすることです。以下にプロセスを説明します。

1. メトリック・パーシスタンス用のデータ・コレクターは、以下のオプション・プロパティーを使用して構成します。
   - **パーシスタンス有効化**: `true` または false
   - **メトリック名フィルター**: 正規表現。正規表現と一致するメトリック名には、履歴キャッシュ後も永続化されるようにマークが付けられます。
   - **サンプリング・カウント**: 永続化処理をスキップするまでの各メトリックの収集回数。0 はスキップしないことを意味します。
   - **履歴キャッシュ・サイズ**: キャッシュに保存するトレース済みメトリックの数。リアルタイムの視覚化を行うには、永続化のマークが付いていないメトリックを含め、ほとんどのメトリックで履歴キャッシュを有効に設定し、メトリックがリアルタイムでグラフにレンダリングされるようにする必要があります。
2. コレクター・キャッシュとは個別のメトリック測定値の集まりで、この数はコレクターによって生成される個々のメトリックの数と同じです。このキャッシュはファーストイン・ファーストアウト (FIFO) のキャッシュなので、履歴キャッシュがフル・サイズになると、最新のメトリック用の場所を空けるために最も古いメトリックが破棄されます。また、このキャッシュにキャッシュ・イベント・リスナーを登録しておくと、登録したリスナーに個々のメトリックの追加およびキャッシュからのメトリックの破棄を通知することができます。
3. コレクター・ロールアップとは一連の複数の循環キャッシュのことで、この循環キャッシュには、コレクターによって生成される各メトリックの 1 つのインスタンスが入れられます。循環バッファー内の各キャッシュ・インスタンスがアクティブになると、履歴キャッシュ用に新しいトレース・イベントを登録し、新しく入ってきたそれぞれの値を集約して、一定の期間のデータ・ストリームにします。このコレクター・ロールアップには以下のものが含まれます。
   - その期間の開始時刻と終了時刻 (終了時刻は期間が終了するまで設定されません。)
   - その期間中の最小、最大、平均測定値
   - メトリック・タイプ
4. 中央タイマーは各期間の終了時にフラッシュ・イベントを起動します。タイマーが起動すると、コレクター・ロールアップの循環バッファーのインデックスがインクリメントされ、履歴キャッシュ・イベントがバッファー内の次のキャッシュに渡されます。すると、「閉鎖された」バッファーで集約された各コレクター・ロールアップに共通のタイムスタンプが適用され、データベースへの保存を待機中のパーシスタンス・キューに書き込まれます。
5. スティッキー・メトリック・タイプの場合を除き、循環バッファーがインクリメントされると次のバッファー要素内の最小値、最大値、平均値がゼロになることに注意してください。パーシスタンス・スレッド・プールに含まれるスレッド・プールが、パーシスタンス・キューからロールアップ項目を読み取ってデータベースに書き込みます。

同じコレクターのロールアップが合体される集約期間の長さは、中央フラッシュ・タイマーの長さによって決まります。この期間が長ければ長いほど、データベースに書き込まれるデータの量は少なくなりますが、データの細分度は損なわれます。

正規化データベースとフラット・データベースの構造の比較

RRDTool: ほぼ完全な監視システムを 1 つに凝縮したツール

RRDTool (Round Robin Database Tool) は、膨大な機能を実質的に 1 つの実行ファイルで提供する強力なツールです (「参考文献」を参照)。このツールは実際のデータを自動的に収集するわけではありませんが、包括的な監視ソリューションを作成するという目的で、`snmpget` や `snmpwalk` などのコマンドライン・ツールと併せて RRDTool を使用するのが一般的となっています。RRDTool では、複数の集約スケジュールのオプションを使用して、ラウンドロビン式の時系列データベース・ファイル (ラウンドロビンによってファイルのサイズが一定になり、手に負えないほどファイルが大きくなることがありません) を作成することができます。さらに作成したファイルのデータに対してクエリーを実行し、そのデータをエクスポートし、データ視覚化とレポート用の充実したグラフをオンザフライで生成することもできます。グラフやクロス・テーブル形式のデータからなるページを常に更新するように単純な Web サーバーを拡張すれば、強力な監視ソリューションになります。

メトリック・ストレージとしては、RRDTool などの特殊化したデータベースも候補となりますが、よく使用されているのはリレーショナル・データベースです。リレーショナル・データベースにメトリック・ストレージを実装する場合には、一般に 2 つの選択肢があります。データを正規化するか、またはそれよりもフラットな構造に維持するかという選択肢です。この選択肢は主として複合メトリック名の保存方法に関わるもので、他のすべての参照データはデータ・モデリングのベスト・プラクティスに従って正規化されることになります。

データベース構造を完全に正規化した場合の利点は、メトリック複合名を個別のセグメントに分割することによって、ほぼすべてのデータベースが索引付けを利用してクエリーを高速化できることです。また、重複して保存されるデータも少なくなるため、データベースを小規模に抑えられ、データの密度も高まり、パフォーマンスが改善されることになります。欠点は、クエリーの複雑さとサイズです。比較的単純な複合メトリック名やメトリック名のパターン (例えば、ホスト 11 から 93 でのユーザー CPU 使用率など) でさえも、SQL に複数の結合と多数の述部が必要になってきます。この問題を緩和するには、データベース・ビューを使用したり、共通メトリック名デコードを固定的に保存したりする手段があります。個別のメトリックを保存するには、複合名を分解してメトリック参照データ項目を見つける (あるいは新しく作成する) 必要がありますが、このパフォーマンス・オーバーヘッドはすべての参照データをパーシスタンス・プロセスにキャッシュすることによって緩和することができます。

図 23 に、正規化した構造で複合メトリックを保存する場合のモデルを示します。

図 23. メトリック・ストレージの正規化モデル

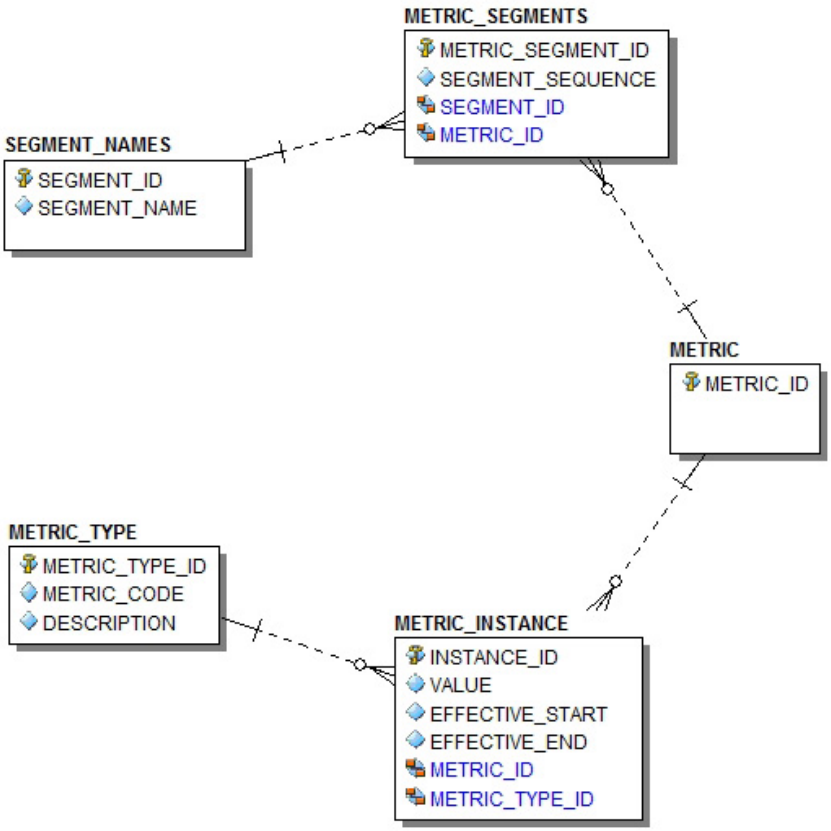
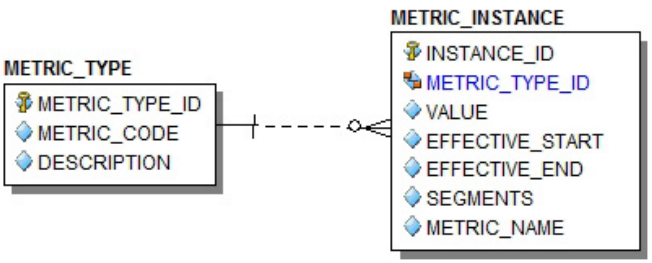


図 23 では、それぞれの複合メトリックに固有の `METRIC_ID` が割り当てられ、各セグメントには `SEGMENT_ID` が割り当てられています。それらの ID が割り当てられた上で複合名が関連エンティティーに組み込まれるため、エンティティーには `METRIC_ID`、`SEGMENT_ID`、そしてセグメントの表示シーケンスが含まれることになります。メトリック・タイプは参照テーブル `METRIC_TYPE` に保存され、メトリック値自体が `METRIC_INSTANCE` に保存されるときには値、タイムスタンプの開始および終了プロパティー、さらにメトリック・タイプと固有の `METRIC_ID` への参照が設定されます。

一方、フラット・モデルは図 24 に示すように、その単純さが利点となります。

図 24. メトリック・ストレージのフラット・モデル



上記の図では、複合名からメトリック名を切り離し、残りのセグメントは `segments` 列に元のパターンのまま維持しています。実装されたデータベース・エンジンが、幅の広いテキスト列でパターン (正規表現など) をベースとした述部によって優れたパフォーマンスのクエリーを実行できる場合、このモデルにはクエリーを実行しやすいという長所があります。この長所は過小評価しないでください。データ分析、可視化、そしてレポート・ツールの作成作業は、簡潔なクエリー構造によって効率性が大幅に向上します。さらに急を要するトリアージ・セッションでは、おそらくクエリーを作成する速度が何よりも重要なポイントになるからです。

永続データ・ストレージが必要な場合には、正しいデータベースの選出が重要なステップとなります。リレーショナル・データベースで十分なパフォーマンスが得られ、要求に応じてフォーマット設定されたデータを抽出できるとしたら、リレーショナル・データベースを使用するのが実際的です。時系列によるデータを生成するのは困難かもしれませんが、適切なグループ化と集約、そして JFreeChart (「参考文献」を参照) などのツールを併せて使用することによって、表現力に優れたレポートとグラフを作成することはできます。それよりも特殊化したデータベース (RRDTool など) を実装するほうを選ぶ場合には、データを抽出して事後レポートを作成する段階で長い道のりが待っていることを覚悟してください。データベースが ODBC や JDBC などの標準をサポートしていなければ、一般的に使用できる標準化されたレポート・ツールは使用することができません。

データ管理についての説明はこれで終わりです。この記事の最後のセクションでは、データをリアルタイムで視覚化する手法を紹介します。

視覚化とレポート作成

この時点で、インスツルメンテーションとパフォーマンス・データ・コレクターは実装されていて、データは APM システムにストリーム配信されるようになっているはずです。そこで当然必要となってくるステップは、APM システムにストリーム配信されたデータをリアルタイムで視覚表示することです。ここではリアルタイムという言葉を、ごく最近収集されたデータを視覚化によって表すという意味で使っています。

データの視覚化にはダッシュボードという言葉がよく使われます。ダッシュボードでは事実上、考えられるあらゆるデータ・パターンやアクティビティーの側面を表すことが可能で、制約を受けるのは収集されたデータの品質と量によってのみです。基本的に、ダッシュボードはエコシステムで何が行われているのかを伝えます。APM システムでダッシュボードが発揮する真の実力の 1 つは、非常にさまざまな種類のデータ (つまり、異なるソースから収集されたデータ) を 1 つの統一したディスプレイに表示できることです。例えば、データベースでの CPU 使用率の最近の傾向と現在の傾向、アプリケーション・サーバー間のネットワーク・アクティビティー、アプリケーションに現在ログインしているユーザーの数をまとめて同時に 1 つのディスプレイに示すことができます。このセクションでは、データ視覚化のさまざまなスタイルを説明し、この記事で取り上げた Spring コレクターの視覚化層を実装する例を紹介します。

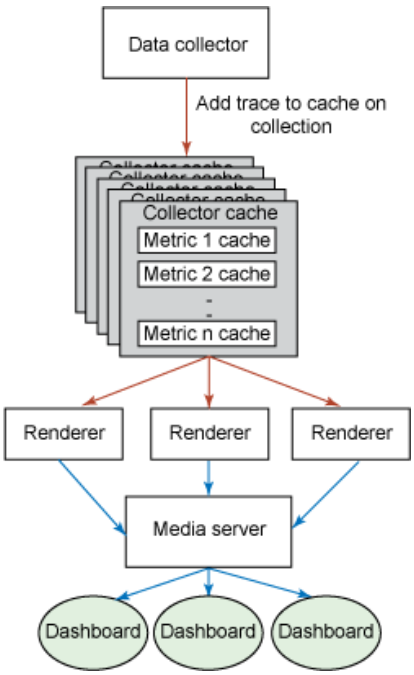
Spring コレクター視覚化層の前提は以下のとおりです。

- Spring Bean としてデプロイされるのは、キャッシュ・クラスのインスタンスです。このキャッシュ・クラスは、任意の数の履歴 `ITracer` トレースを維持するように構成することができますが、そのサイズは一定であり、FIFO になっています。例えば、履歴サイズを 50 に

してキャッシュを構成したとすると、キャッシュがフルになった場合には最後の 50 のトレースが維持されることになります。
- Spring コレクターは、`ITracer` のインスタンスをキャッシング・プロセスでラップする `cacheConfiguration` と一緒に Spring XML 構成ファイルに構成されます。この構成は、コレクターを、定義されたキャッシュ・インスタンスにも関連付けます。収集されたトレースはいつもと同じように処理されますが、コレクターに関連付けられたキャッシュ・インスタンスが追加されます。前の例で言うと、キャッシュの履歴サイズが 50 で、コレクターが 30 秒間隔でデータを収集する場合、フルになったキャッシュは最後の 25 分間にコレクターが収集したすべてのトレースを維持することになります。
- Spring コレクター・インスタンスには複数のレンダリング・クラスがデプロイされています。レンダラーは、`org.runtimemonitoring.spring.rendering.IRenderer` インターフェースを実装するクラスです。レンダラーの役目は、キャッシュからデータの配列を取得し、そのデータから何らかの形の視覚化を行いレンダリングすることです。レンダラーから視覚化メディアを定期的に取得することで、新しい最新の (またはキャッシュ・データの現状に合わせた) 表示が生成されます。
- レンダリングされたコンテンツは、ダッシュボードのコンテキスト内のクライアント (Web ブラウザーや何らかの形のリッチ・クライアントなど) に配信することができます。

図 25 に、このプロセスの概略を示します。

図 25. キャッシングとレンダリング



この例でのキャッシュ実装は `org.runtimemonitoring.spring.collectors.cache.TraceCacheStore` です。その他のオブジェクトは、レンダラーがイベントのなかでもとりわけ新しい値がキャッシュに追加されたことを示す新しいキャッシュ項目のイベントをリッスンできるように、キャッシュ・イベント・リスナーとして登録することができます。この場合、レンダラーは生成するコンテンツを実際にキャッシュに入れる一方、新しいデータが利用可能になったときにはキャッシュを無効にすることができ

ます。レンダラーが生成するコンテンツをクライアント・ダッシュボードに配信する手段として使用するのは、`org.runtimemonitoring.spring.rendering.MediaServlet` というサーブレットです。このサーブレットは受け取ったリクエストを構文解析し、レンダラーを見つけてコンテンツ (すべてのコンテンツはバイト配列としてレンダリングされて配信されます) とコンテンツの MIME タイプを要求します。バイト配列は、クライアントが配信されたストリームを解釈できるように、MIME タイプと併せてクライアントにストリーム配信されます。グラフィック・コンテンツは Web ブラウザー、リッチ・クライアントの類のもので使用できるように URL ベースのサービスから提供するのが最適です。レンダラーがメディア・サーバーからコンテンツに対するリクエストを受け取ると、キャッシュ・イベントによってキャッシュにダーティーのマークが付けられていない限り、コンテンツはキャッシュから配信されます。このようにすることで、レンダラーはリクエストごとに毎回コンテンツを再生成しなくても済むようになります。

バイト配列形式でビジュアル・メディアを生成、キャッシング、配信するのが有効な理由は、これが最小公分母だからです。また、大抵のクライアントは MIME タイプで提供されたコンテンツについては再構成することができます。この実装では生成されたコンテンツをメモリー内にキャッシュするため、私は圧縮方式を使っています。大量のコンテンツをキャッシュすることによってメモリーの合計使用量は大幅に加算されますが、圧縮アルゴリズムの符号をコンテンツと一緒に提供すれば、大抵のクライアントが解凍できるようになります。例えば最近のブラウザーのほとんどは、`gzip` の解凍をサポートします。ただし、圧縮レベルはそれほど高くないので (サイズの大きな画像では 30 から 40 %)、レンダリング実装がディスクにキャッシングできるようにするか、あるいはディスクにアクセスするとオーバーヘッドが増える場合にはコンテンツをオンザフライで再生成したほうがリソースへの負担が軽くなります。

具体的な例で説明するとわかりやすいと思うので、ここで、ビジー状態のワーカー・スレッド数を監視する 2 つの Apache Web Server コレクターをセットアップします。各コレクターにはキャッシュを割り当て、サーバーごとのビジー状態ワーカー・スレッド数のグラフを作成する少数のレンダラーをセットアップします。この場合、レンダラーが PNG ファイルを生成し、両方のサーバーの数列を示す時系列の線グラフを表示します。リスト 22 は、一方のサーバーに対するコレクターとキャッシュのセットアップです。

リスト 22. Apache Web Server コレクターおよびキャッシュ

```
<!-- The Apache Collector -->
<bean id="Apache2-AP02"
   class="org.runtimemonitoring.spring.collectors.apacheweb.ApacheModStatusCollector"
   init-method="springStart">
   <property name="scheduler" ref="CollectionScheduler" />
   <property name="logErrors" value="true" />
   <property name="tracingNameSpace" value="WebServers,Apache" />
   <property name="frequency" value="15000" />
   <property name="initialDelay" value="3000" />
   <property name="modStatusURL" value="http://WebAP02/server-status?auto" />
   <property name="name" value="Apache2-AP02" />
   <property name="cacheConfiguration">
      <bean
         class="org.runtimemonitoring.spring.collectors.cache.CacheConfiguration">
         <property name="cacheStores" ref="Apache2-AP02-Cache"/>
      </bean>
   </property>
</bean>

<!-- The Apache Collector Cache -->
<bean id="Apache2-AP02-Cache"
```

```
   class="org.runtimemonitoring.spring.collectors.cache.TraceCacheStore">
   <constructor-arg value="50"/>
</bean>
```

コレクターの `cacheConfiguration` プロパティーが `Apache2-AP02-Cache` というキャッシュ・オブジェクトをどのように参照しているかに注目してください。

`org.runtimemonitoring.spring.rendering.GroovyRenderer` のインスタンスであるレンダラーもセットアップしました。このレンダラーはすべてのレンダリングを、ファイルシステム上の基礎となる Groovy スクリプトに委任します。この方法は、生成されるグラフィックの詳細を実行時に微調整できるので理想的です。このレンダラーの一般プロパティーには以下のものがあります。

- **`groovyRenderer`**: Groovy スクリプトをディレクトリーからロードするように構成された `org.runtimemonitoring.spring.groovy.GroovyScriptManager` への参照です。このクラスは、Telnet セッションから私の Cisco CSS に返されたデータを処理するために使用したクラスと同じです。
- **`dataCaches`**: レンダラーはこのキャッシュのセットに対してデータを要求し、レンダリングします。また、レンダラーはこれらのキャッシュに新しい項目が追加されたときにキャッシュからイベントを受け取るように登録します。レンダラーはイベントを受け取るとキャッシュのコンテンツにダーティーのマークを付け、次の要求時に再生成されるようにします。
- **`renderingProperties`**: 生成されたグラフィックの特定の詳細 (画像のデフォルト・サイズなど) を指示するためにレンダラーに渡されるデフォルト・プロパティーです。以下のリストを見るとわかるように、これらのプロパティーはクライアントのリクエストによって上書きすることができます。
- **`metricLocatorFilters`**: コレクターのキャッシュには、コレクターが生成したすべてのメトリックのトレースがキャッシュされます。必要なメトリックに絞り込むための正規表現の配列を指定するには、このプロパティーを使用します。

リスト 23 に、このキャッシュのセットアップを記載します。

リスト 23. Apache Web Server のビジー・ワーカーを監視するグラフィック・レンダラー

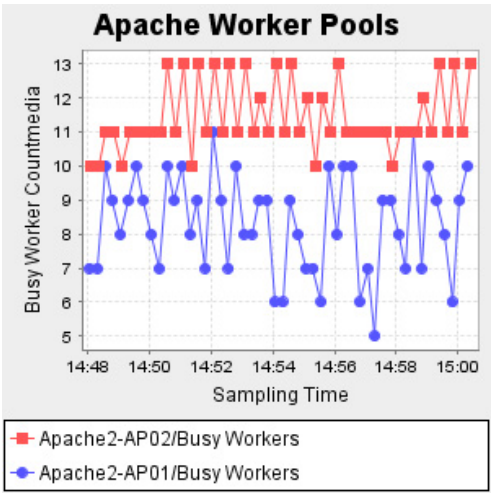
```
<bean id="Apache2-All-BusyWorkers-Line"
   class="org.runtimemonitoring.spring.rendering.GroovyRenderer"
         init-method="init"
         lazy-init="false">
   <property name="groovyRenderer">
      <bean class="org.runtimemonitoring.spring.groovy.GroovyScriptManager">
         <property name="sourceUrl" value="file:///groovy/rendering/multiLine.groovy"/>
      </bean>
   </property>
   <property name="dataCaches">
      <set>
         <ref bean="Apache2-AP01-Cache"/>
          <ref bean="Apache2-AP02-Cache"/>
      </set>
   </property>
   <property name="renderingProperties">
      <value>
        xSize=700
        ySize=300
        title=Apache Servers Busy Workers
        xAxisName=Time
```

```
        yAxisName=# of Workers Busy
      </value>
   </property>
   <property name="metricLocatorFilters" value=".*/Busy Workers"/>
</bean>
```

レンダラーを実装するのは至って簡単ですが、レンダラーには常に調整の必要があることはわかっています。そこで役立つのが、ここで概説する Groovy の手法です。Groovy の手法では、新しいグラフ・タイプや、あるいは新しいグラフィック・パッケージを素早くプロトタイプ化します。いったんこの Groovy コードをコンパイルすればパフォーマンスが向上し、適切なコンテンツを使用できるため、キャッシングは問題にならなくなるはずです。オンザフライでの更新を容易にする、動的ホット・アップデートと (とても機能性に優れた) Groovy 構文については、この後、レンダラーに必要な具体的実行内容、そしてレンダラーがサポートしなければならないすべてのオプションが明らかになってから Java コードに移植します。

メトリックの名前は、`org.runtimemonitoring.tracing.Trace` クラスによって生成されます。それぞれに 1 つの `ITracer` 測定値を表すこのクラスのインスタンスは、トレースされる値、タイムスタンプ、そして完全名前空間のカプセル化です。メトリックの名前は、メトリック名を含めた完全名前空間となります。この例で私が表示しているメトリックは `WebServers/Apache/Apache2-AP01/Busy Workers` なので、リスト 23 のレンダラーに定義されたフィルターは、この 1 つのメトリックをレンダリング範囲に設定します。生成された JPG を図 26 に示します。

図 26. レンダリングされた Apache ビジー・ワーカー

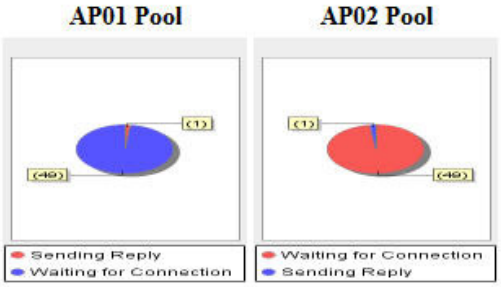


クライアントによって必要となるグラフィックのレンダリング方法は異なります。例えば、あるクライアントにはサイズを小さくした画像が必要になる場合もありますが、オンザフライでサイズ変更した画像はぼやけてしまうのが通常です。また、別のクライアントには同じくサイズを小さくして、しかもタイトルを省いた画像が必要な場合もあります (タイトルはクライアント独自の UI に表示)。このような場合、`MediaServlet` では、コンテンツの要求中に追加オプションを実装することができます。これらのオプションはコンテンツ・リクエストの URL に追加され、REST フォーマットで処理されます。オプションの基本フォーマットは、media サーブレット・パス (これは構成可能です) にキャッシュ名または /media/Apache2ll-BusyWorkers-Line を続けた形になります。レンダラーによってサポート可能なオプションはさまざまです。上記で使用したレンダラーの場合は、以下のオプションが好例となります。

- デフォルト URI: /media/Apache2ll-BusyWorkers-Line
- 300 X 300 に縮小: /media/Apache2ll-BusyWorkers-Line/300/300
- 300 X 300 に縮小し、タイトルおよび軸名を最小化: /media/Apache2ll-BusyWorkers-Line/300/300/BusyWorkers/Time/#Workers

図 27 は、URI Apache2P02-WorkerStatus-Pie/150/150/ /: を使ってタイトルを省略し、サイズを縮小した円グラフです。

図 27. Apache Server ワーカー・プールの縮小画像



レンダラーは、要求側のクライアントが表示可能な形式であれば、事実上どんな形式でもコンテンツを生成することができます。画像形式は、JPG、PNG、または GIF のいずれかにすることができます。その他の画像形式もサポートできますが、Web ブラウザー・クライアントを対象とした静的画像の場合には、PNG および GIF が最適なはずです。その他の選択肢としては、HTML などのマークアップを使用したテキスト・ベースの形式もあります。HTML はブラウザーとリッチ・クライアントの両方がそのフラグメントをレンダリングできるため、個々のデータ・フィールドと列と行からなるテーブルを表示するには最適です。また、プレーン・テキストも役立ちます。例えば Web ブラウザー・クライアントはレンダラーからシステムが生成したイベント・メッセージを表すテキストを取得し、それをテキスト・ボックスやリスト・ボックスに挿入する場合があるからです。他の種類のマークアップもまた適応性に優れています。多くのリッチ・クライアントとクライアント・サイドのブラウザー用レンダリング・パッケージは、後からクライアント・サイドで生成できるグラフィックを定義する XML 文書を読み込むため、パフォーマンスには最適です。

クライアント・サイドのレンダリングには、さらに最適化の可能性があります。クライアントが独自の視覚表示をレンダリングできる場合、マークアップ・タグを追加するためにレンダラーが必要な場合を除き、レンダラーをバイパスしてクライアントに直接キャッシュの更新をストリーム配信することができます。この場合、クライアントがキャッシュ更新イベントを購読すれば、イベントの受信時に独自の視覚表示を更新することができます。クライアントへのデータのストリーミングには、さまざまな方法があります。ブラウザー・クライアントでは、単純な Ajax スタイルのポーリング・プログラムがサーバーで定期的に更新をチェックし、更新をデータ構造に挿入するハンドラーを実装して、そのハンドラーにブラウザーでのレンダリングを処理させることができます。もう少し複雑な方法になると、Comet パターンを使用した実際のデータ・ストリーミングが必要になってきます。Comet パターンによってサーバーへの接続を常にオープンに維持し、データがサーバーに書き込まれると同時にクライアントが読み取れるようにします (「参考文献」を参照)。リッチ・クライアントの場合には、クライアントがデータ更新フィードを購読するメッセージング・システムを使用するのが最適です。ActiveMQ は Jetty Web サーバーと連動

し、Comet 機能を持つことから、この両方 (データ・ストリーミングおよびデータ更新フィードの購読) を実行できるので、ブラウザー・ベースの JavaScript JMS クライアントを作成し、キューとトピックを購読することができます。

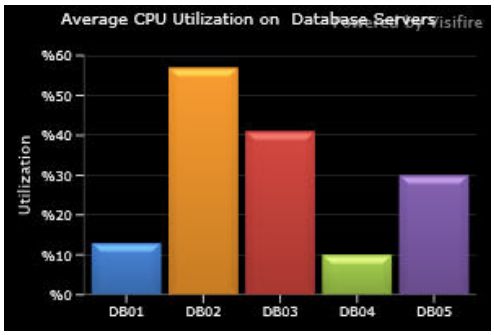
クライアント・サイドで完全なレンダリングが可能な場合、フラット画像では使用できない機能も追加されることになります。例えば、要素をクリックしてドリルダウンするなどの機能です。この機能は、ナビゲートや、グラフに示された特定の項目の詳細を表示するためにドリルダウンを使用する APM ダッシュボードでの共通要件となっています。その一例は、Silverlight と連動するオープンソースのグラフ作成ツール、Visifire です (「参考文献」を参照)。リスト 24 に、複数のデータベース・サーバー全体での CPU 使用率を示す棒グラフを生成する XML のフラグメントを記載します。

リスト 24. データベース・サーバー全体での平均 CPU 使用率のグラフィック・レンダラー

```
<vc:Chart xmlns:vc="clr-namespace:Visifire.Charts;assembly=Visifire.Charts"
    Theme="Theme3">
         <vc:Title Text="Average CPU Utilization on  Database Servers"/>
         <vc:AxisY Prefix="%" Title="Utilization"/>
         <vc:DataSeries Name="Utilization" RenderAs="Column">
                  <vc:DataPoint AxisLabel="DB01" YValue="13"/>
                  <vc:DataPoint AxisLabel="DB02" YValue="57"/>
                  <vc:DataPoint AxisLabel="DB03" YValue="41"/>
                  <vc:DataPoint AxisLabel="DB04" YValue="10"/>
                  <vc:DataPoint AxisLabel="DB05" YValue="30"/>
         </vc:DataSeries>
</vc:Chart>
```

上記の XML はかなり単純なので、この XML のレンダラーを作成するのはわけありません。その表示もなかなかのものです。クライアント・サイドのレンダラーは表示をアニメーションにすることもできます。アニメーションにすると APM システムの表示にさまざまな価値がもたらされますが、その他の場合でも役立つことがあります。図 28 に、Silverlight クライアントを有効にしたブラウザーで生成されたグラフを示します。

図 28. VisiFire Silverlight によってレンダリングされたグラフ

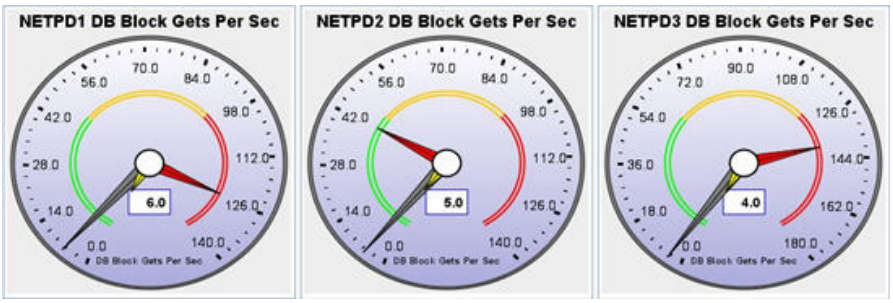


標準のグラフ・タイプはすべて、APM ダッシュボード内に配置されます。最もよく使われるタイプは、折れ線グラフ、複数の折れ線からなる折れ線グラフ、棒グラフ、円グラフです。グラフは組み合わせられることがよくあります。例えば、2 種類のデータを重ねて表示するために棒グラフと折れ線グラフを組み合わせるなどです。また、折れ線グラフの y 軸を二重にして、大きさがまったく異なる値を示すデータ数列を同じグラフ上に表示できるようにすることもあります。こ

れに該当するのは、ルーターの CPU 使用率など、スカラー値に対するパーセンテージの値を、転送されたバイト数と比較して描画する場合などです。

一部のシナリオでは、データをカスタマイズして表示したり、あるいは直観的なディスプレイにデータを表示したりできるように特殊化したウィジェットを作成することもあります。例えば、シンボルを列挙して、監視ターゲットのステータスに応じた特定のアイコンを表示するなどです。ステータスは通常、限られた数の可能な値で表されるため、グラフを作成するまでのことはありません。そこで、信号機のようなディスプレイを使用して、停止している場合は赤、注意が必要な場合は黄色、そして正常な場合は緑で示すという方法があります。同じ目的では、ダイヤル・ウィジェットもよく使用されます (よく速度計として表されるダイヤル)。ダイヤルが表すのは 1 つのベクトルだけで、履歴は示さないため、私はダイヤルは画面スペースの無駄だと思います。折れ線グラフを使えば、同じデータでもこれまでの傾向と併せて示すことができます。唯一の例外は複数の針を持つダイヤルで、この場合、高/低/現在などといった範囲を示すことができます。とは言っても、ダイヤルの大きな目的は見た目です。その一例として、過去 1 時間のデータベース・ブロックの 1 秒あたりのバッファー取得回数を高/低/現在の範囲で示す図 29 を見てください。

図 29. ダイヤル・ウィジェットの例



私の考えでは、視覚化の効果があるのは情報の密度が高い場合、つまり限られた画面スペースにできるだけ多くのデータを表示する場合です。データの密度を高くするにはいくつも方法が考えられますが、なかでも興味深い方法は、カスタム・グラフィック表示を作成して、1 つの小さな画像のなかに複数の次元から見たデータを組み合わせるというものです。その好例を図 30 に示します。これは (悲しくも) バージョンアップされることのなくなったデータベース監視製品から引用したものです。

図 30. Savant キャッシュ・ヒット率の表示

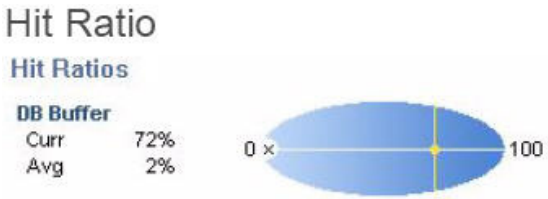


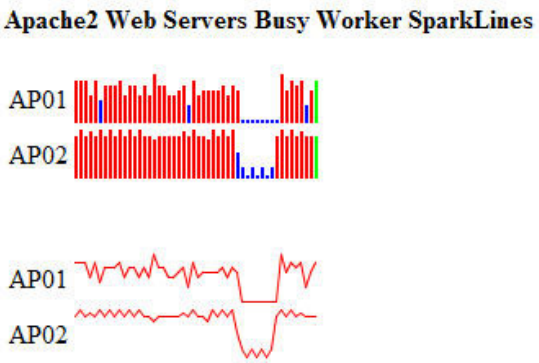
図 30 には、データベースのバッファー・ヒット率に関連する複数のベクトルでデータが示されています。

- 水平軸は、キャッシュ内で見つかったデータのパーセンテージを表します。
- 垂直軸は、現在のヒット率を表します。

- X は、ヒット率の傾向を表します。
- グレーの円 (この画像ではほとんど見えません) は、標準偏差です。グレーの円の直径が大きくなればなるほど、キャッシュ・パフォーマンスの偏差が大きいということになります。
- 黄色の丸は、過去 1 時間のキャッシュ・パフォーマンスを表します。

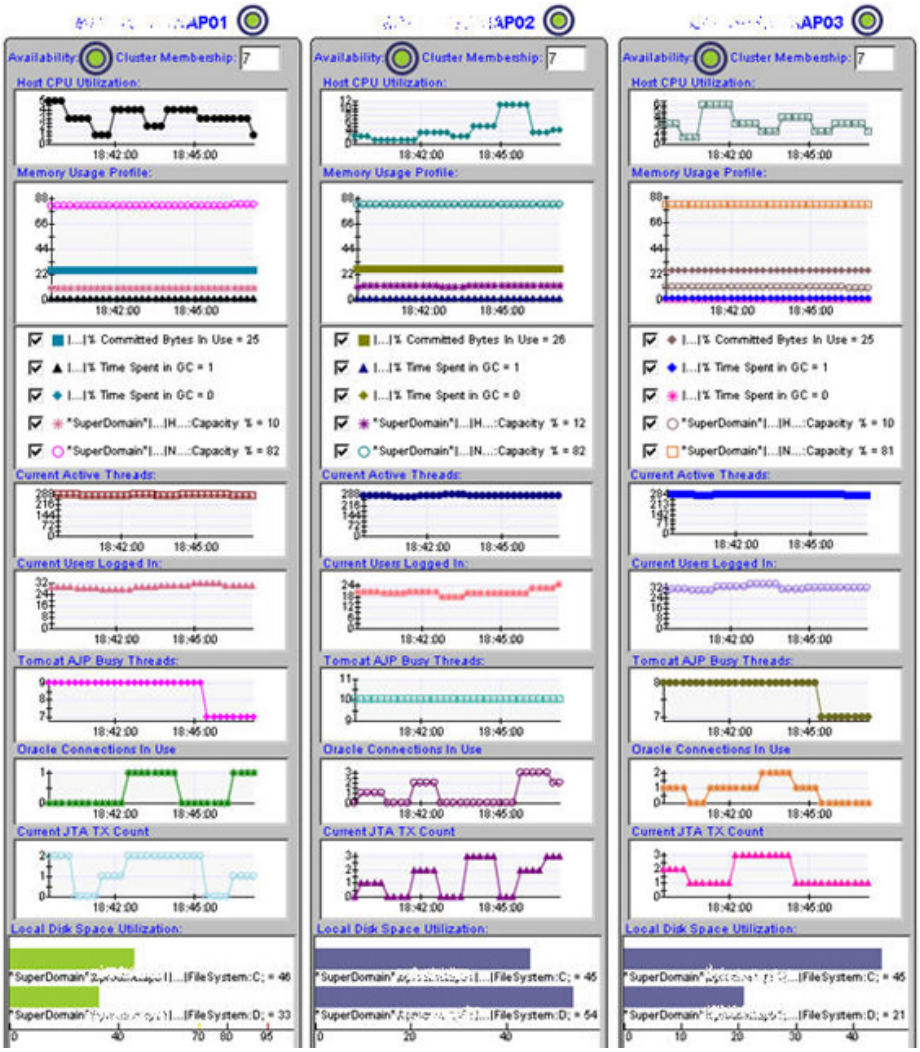
データ密度に対処するためのもう 1 つの方法は、最低必要限度に抑えるという手法を採用したスパークライン (Sparkline) です。スパークラインとはデータ視覚化のエキスパート、Edward Tufte (「参考文献」を参照) によって作られた用語で、「テキスト、数値、画像のコンテキストで組み込んだ小型で解像度の高いグラフィック」を意味します。スパークラインは一般に、多数の金融統計を表示するために使用されています。コンテキストは欠けているものの、その目的は多数のメトリックでの相対的傾向を表すことです。スパークラインのレンダラーを実装するのは `org.runtimemonitoring.spring.rendering.SparkLineRenderer` クラスで、このクラスは Java ライブラリーのオープンソース・スパークラインを実装します (「参考文献」を参照)。図 31 に、2 つの (拡大) スパークラインの例を示します。一方は棒による表示、もう一方は折れ線による表示です。

図 31. Apache 2 ビジー・ワーカーを示すスパークライン



ここで概説した例も、付属のコードに記載した例もかなり基本的なものですが、APM システムには明らかに、高度な機能を備えた詳細なダッシュボードが必要です。ほとんどのユーザーはスクラッチからダッシュボードを作成したがりませんが、APM システムには通常、何らかの形のダッシュボード生成プログラムが用意されています。ユーザーはそれを使って使用可能なメトリックのリポジトリーを表示または検索し、ダッシュボードに組み込むメトリック、そしてメトリックの表示形式を選択することができます。図 32 に、私の APM システムで作成したダッシュボードの一部を示します。

図 32. ダッシュボードの例



まとめ

これでこの連載は完結です。これまでパフォーマンス監視のガイドラインとして、一般的な監視手法、そして独自の監視システムを独自に拡張または作成するために実装できる具体的な開発パターンを紹介しました。有効なデータを収集して分析することで、アプリケーションのアップタイムとパフォーマンスは大幅に改善することができます。開発者の皆さんはぜひ、実稼働アプリケーションの監視プロセスに参加してください。自分が作成したソフトウェアが負荷状態で実際にどのように動作するかを判断し、経験するには、実稼働アプリケーションを監視する以上に有効な手段はありません。そしてこの監視によるフィードバックが、現在進行中の改善サイクルの一部としてかけがえのないものとなります。ぜひ監視を有効に活用してください。

謝辞

Web サービス・コレクターに関して支援してくれた Sandeep Malhotra 氏に深く感謝します。

ダウンロード

内容	ファイル名	サイズ
Sample code for this article	j-rtm3.zip	316KB

著者について

Nicholas Whitehead

Nicholas Whitehead は、ニュージャージー州 Florham Park にある ADP の Small Business Services 部門に所属するシニア・テクノロジー・アーキテクトです。これまで 10 年以上、投資銀行、e-コマース、ソフトウェアをはじめとするさまざまな業界で Java アプリケーションを開発しています。実稼働アプリケーション (その一部は彼自身のアプリケーション) のデプロイメントとサポートにおける彼の経験が、パフォーマンス管理システムの調査と実装に生かされています。