Java ランタイムの監視: 第 1 回 Java システムのランタイム・パフォーマンスと可用性を監視する

その手法とパターン

Nicholas Whitehead
Senior Technology Architect
ADP

2008年 7月 29日

優れたパフォーマンスのシステムを実現し、その状態を維持するにはランタイム・パフォーマンスの監視が欠かせません。3 回連載の第 1 回目となるこの記事では、Nicholas Whitehead が Java™ のパフォーマンスをきめ細かく効率的に監視する方法を説明します。パフォーマンスの監視によって生成されるデータからは、システムの動作状態に関する貴重な洞察が可能となり、ランタイム環境の安定性とパフォーマンスに影響を及ぼす、制約事項とその他の要素が明らかになります。

このシリーズの他の記事を見る

はじめに

最近の Java アプリケーションの多くは、分散された依存関係と可動部の複雑な組み合わせに依存しているため、多数の外部要因がアプリケーションのパフォーマンスと可用性に影響する可能性があります。こうした外部要因による影響を完全に排除したり、あるいは実稼働前の環境で外部要因による影響を明らかにし、正確なエミュレーションを行ったりすることは事実上不可能です。そのため、実際のアプリケーションではいろいろなことが起こり得ます。それでも、アプリケーションのエコシステム全体を監視する包括的システムを作成し、維持することによって、アプリケーションに起こり得るさまざまな出来事による影響の重大性をその程度や期間も含めて大幅に軽減することが可能です。

この 3 回連載の記事では、このような監視システムを実装する上でのパターンと手法を紹介します。これらのパターン、そしてこの記事で使用する一部の用語は意図的に一般的なものを選んでいるので、サンプル・コードとイラストと併せ、アプリケーション・パフォーマンスを監視するという概念を理解する上で役立つはずです。この概念を理解することによってソリューションの必要性が浮き彫りになり、それによって、商用ソリューションとオープンソース・ソリューションのどちらを選ぶか、あるいはそのいずれかを拡張してカスタマイズするかを判断する助けとなります。さらにやる気のある読者にとっては、この連載で理解した内容が、自分でソリューションを構築する際の土台になります。

第 1 回の内容は以下のとおりです。

- アプリケーション・パフォーマンス管理 (APM: Application Performance Management) システムの特質を探ります。
- システムの監視でよく見られるアンチパターンを説明します。
- JVM のパフォーマンスを監視する方法を説明します。
- アプリケーションのソース・コードに効率的にインスツルメンテーション (システム監視のための計測機能) を追加する手法を提案します。

第 2 回では、元のソース・コードを変更せずに Java クラスとリソースにインスツルメンテーションを追加する方法に焦点を絞ります。第 3 回で取り上げるのは、ホストやオペレーティング・システム、さらにデータベースやメッセージング・システムなどのリモート・サービスをはじめとする JVM 外部のリソースを監視する方法です。この第 3 回の記事では連載の締めくくりとして、データ管理、データ仮想化、レポートの作成、アラートの起動など、付加的な APM に関する問題についても説明します。

APM システム: パターンとアンチパターン

この記事を正しく理解して読み進められるよう、最初に強調しておきますが、ここに記載する Java 固有のコンテンツの大部分はアプリケーションおよびコードのプロファイリング・プロセスと似ているかもしれませんが、この記事で私が言及しているのはプロファイリングのプロセスではありません。プロファイリングは極めて有益な実稼働前のプロセスで、このプロセスによって、Java コードがスケーラブルで効率性に優れ、高速であること、そして概して素晴らしいものであることを確認できたり、やってはいけないことを示したりすることができます。しかし、さまざまなことが起こり得るという当然の事実に基づけば、開発フェーズのコード・プロファイラーからお墨付きの承認をもらったとしても、実稼働中に説明し難い問題が起きたときにその承認は役に立ちません。

私が言及しているのは、プロファイリングのいくつかの側面を実稼働環境に実装し、実行中のアプリケーションと、そのアプリケーションと外部依存関係を持つすべてのものからリアルタイムで同じ種類のデータを収集するということです。このデータを構成するのは、継続して行われている一連の定量的測定によって得られた値で、広範囲に分散するさまざまなターゲットで収集されたこれらの測定値が、システム全体の正常性を細部まで詳細に表すことになります。さらに、測定値の履歴ストアを保持することによって正確な基準が得られていれば、この基準を利用して環境の正常性が保たれていることを確認したり、あるいは特定の部分で正常性が欠如していることの根本原因と規模を突き止めたりすることができます。

監視のアンチパターン

監視リソースがまったくないアプリケーションはめったにないでしょうが、運用中の環境ではよく、以下のアンチパターンが見受けられます。

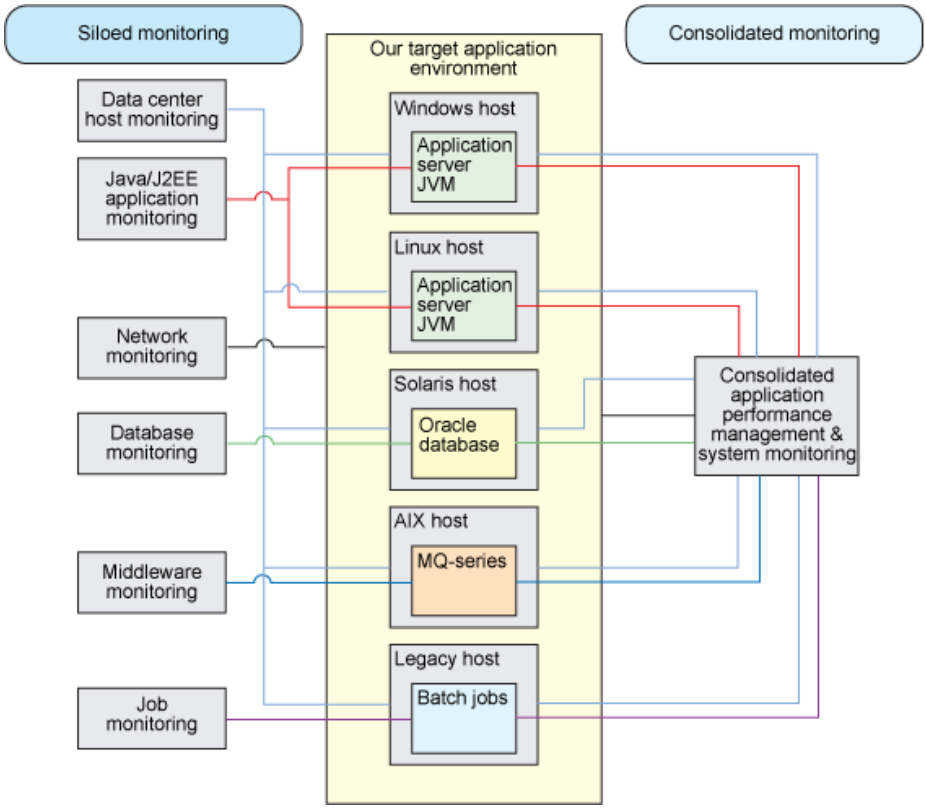
サイロ型の監視の落とし穴

サイロ型の監視は、システムの全体像を確認できるビューが 1 つもない場合に発生します。非常に複雑で診断が難しい問題には必ず、複数の関連コンポーネントと従属コンポーネントが関与するものです。単純な例として、Java アプリケーション・サーバーでホストされているアプリケーションが、所有者の知らないところで不完全な JDBC コネクション・プーリング・クラスを実装し、そのためにコネクションのリークが発生している場合を考えてみてください。

アプリケーションの所有者が自分の管理インターフェースで確認する限り、サーバーが保持しているデータベースへ張られているコネクションの数は 100 となっているはずです。ところがデータベース管理者 (DBA) のデータベース管理コンソールには、この同じホストが実際には 120 のコネクションを保持していることが示され、その数は急激に増え続けることになります。一方、統合 APM システムの場合、この両方のメトリックを示す折れ線グラフを作成するのはわけありません。2 つの数値に差が出てくると同時に、このグラフを見る誰もが実際の正確な数値をすぐに把握できるだけでなく、どこに問題があるのかについてもかなり正確に見当を付けることができます。

- **カバーし損ねている監視対象**: システムの依存関係の一部が監視されていなかったり、監視データにアクセスできなかったりすることがあります。運用データベースが監視範囲全体をカバーできるとしても、サポート・ネットワークが監視範囲をカバーしていなければ、トリアージ・チーム (問題に対処する優先度を判定するチーム) がデータベースのパフォーマンスとアプリケーション・サーバーの症状を調査している間、ネットワーク内での障害が実質的に隠されてしまいます。
- **ブラック・ボックスの監視対象**: コア・アプリケーション、あるいはコア・アプリケーションと依存関係を持ついずれかのモジュールに関して、その内部を監視するための情報が得られない場合があります。JVM は実質的にブラック・ボックスです。例えば、トリアージ・チームが JVM での説明のつかない遅延を調査しているとします。そしてこの JVM には、CPU 使用率またはプロセスを処理するためのメモリー・サイズなど、サポート・オペレーティング・システムの統計が 1 つしかないとします。この場合、チームはガーベッジ・コレクションやスレッド同期化などの問題を診断できない可能性があります。
- **結合性のない分断された監視システム**: データベース、SAN (Storage-Area-Network: ストレージ・エリア・ネットワーク) ストレージ、またはメッセージングやミドルウェア・サービスなど、多数の共有リソースが依存関係を構成する大規模な共有データ・センターでアプリケーションがホストされる場合があります。組織は時として、徹底したサイロ構造で、それぞれのグループが独自の監視システムと APM システムを管理していることがありますが (囲み記事「サイロ型の監視の落とし穴」を参照)、このような共有データ・センターでアプリケーションがホストされる場合、依存関係を統合したビューがなければコンポーネントの所有者は全体像のほんのわずかな部分しか見ていないことになります。
  図 1に、サイロ型 APM システムと統合 APM システムを比較します。

図 1. サイロ型 APM システムと統合 APM システムの比較



- **事後報告と収集データの相関性**: サイロ型の監視での問題に対処しようとして、運用サポート・チームが各種ソースからデータを収集し、データを一箇所に集めてサマリー・レポートを生成するという定期的プロセスを実行することがあります。この方法は、一定の頻度で行うには非効率的だったり、実際的ではなかったりする場合があります。また、リアルタイムで集約されたデータがないために、トリアージ・チームがその場で問題を診断できなくなる可能性もあります。さらに、事後集約したときのデータの細分度が十分でなければ、データに存在する重要なパターンを見逃す結果にもなりかねません。例えば、前の日の特定サービスの呼び出しについてのレポートでは、平均経過時間が 200 ミリ秒であったことは示すことができても、午後 1時から 1時45分までの間は常に経過時間が 3500 ミリ秒を超えていたという事実は隠されてしまいます。
- **定期的な監視またはオンデマンドでの監視**: 監視ツールによってはリソースのオーバーヘッドが高いため、継続的に実行することができません (または、継続して実行すべきではない場合もあります)。そのため、これらのツールではデータをめったに収集しないか、あるいは問題が検出された場合にだけ使用することになります。そうなると、APM システムが作成する基準は最低限の基準となり、問題が許容できないほど深刻になるまでアラートを出せなくなります。つまり、APM 自体が状況を悪化させる可能性があるということです。
- **非永続的な監視**: 多くの監視ツールには、現行のパフォーマンスおよび可用性メトリックを表示する便利な機能が備わっていますが、長期間または短期間を範囲とした比較と分析に備えて測定値を保持するようには構成されていないか、そのような機能をサポートしていません。多くの場合、履歴というコンテキストが欠けていると、パフォーマンス・メトリックの価値はほとんどなくなってしまいます。メトリックの値が正常なのか悪いのか、あるいは最

悪なのかを判断する基準がないためです。例えば、現行の CPU 使用率が 45 パーセントだとします。この測定値は、負荷の大きい期間、または小さい期間の過去の使用率がわからなければ多くの情報を提供しません。一方、標準値が x パーセントで、許容されるユーザー・パフォーマンスの上限はこれまで y パーセントだったということがわかっていれば、測定値から得られる情報は遥かに多くなります。
- **実稼働前のモデリングへの依存**: 実稼働前に行った監視の結果とシステムのモデリングだけを基に、考えられるすべての問題は実稼働環境へのデプロイメント前に環境から取り除けたとする前提を持つことで、実行時の監視内容が不十分なものになることがよくあります。この前提には予測不可能な出来事や依存関係での障害が起きることが考慮されていないため、そのような状況が発生したときにトリアージ・チームが使用できるツールやデータが何もないという事態になってしまいます。

統合 APM を実装することによって、DBA 管理ツールセットや下位レベルのネットワーク分析アプリケーション、そしてデータ・センター管理ソリューションといった非常に特化された監視ツールや診断ツールが使用できなくなったり、その価値が下がったりすることはありません。これらのツールが貴重なリソースであることには変わりませんが、それだけに頼って統合されたビューを使わないとしたら、サイロ型の監視による影響を克服することは困難です。

理想的な APM システムの特質

上記で説明したアンチパターンとは逆に、この連載で紹介する理想的な APM システムには以下の特質が備わっています。

- **広汎性**: すべてのアプリケーション・コンポーネントと依存関係を監視します。
- **細分度**: 極めて下位レベルの関数まで監視することができます。
- **統合化**: 収集した測定値はすべて、統合ビューをサポートする 1 つの論理 APM に渡されます。
- **継続性**: 1 日 24 時間、週 7 日監視します。
- **効率性**: パフォーマンス・データの収集が監視の対象に悪影響を及ぼすことはありません。
- **リアルタイム性**: 監視されるリソース・メトリックは、リアルタイムで視覚化、レポート生成、そしてアラートの起動が行われます。
- **履歴**: 監視されるリソース・メトリックはデータ・ストアに保持されるため、履歴データを視覚化、比較、そしてレポートすることができます。

このシステムの実装の詳細に入る前に、APM システムを理解する上で役立つ一般的な側面について説明します。

APM システムにおけるいくつかの概念

すべての APM システムは「パフォーマンス・データ・ソース」にアクセスできるのと同時に、データの「収集」機能と「トレース」機能を備えています。以上の用語は、一般的なカテゴリーで説明するために私が独断で選んだ一般的な用語であることに注意してください。これらの用語はいずれかの APM システムに固有の用語というわけではないので、同じ概念に対して他の用語を使うこともできますが、この記事では一貫して、この 3 つの用語を以下の定義に基づいて使用します。

パフォーマンス・データ・ソース

パフォーマンス・データ・ソース (PDS) とは、コンポーネントの相対的な正常性を示すための測定値として役立つパフォーマンス・データまたは可用性データを提供するソースのことです。その一例として、JMX (Java Management Extensions) サービスは、一般に JVM の正常性に関する豊富なデータを提供します。また、大抵のリレーショナル・データベースは SQL インターフェースを介してパフォーマンス・データを公開します。この JMX サービスとリレーショナル・データベースの 2 つは私がダイレクト (直接) ソースと呼んでいる PDS の例です。ダイレクト・ソースとはつまり、パフォーマンス・データを直接提供するソースのことです。一方、インファレンシャル (推定) ソースは意図的なアクションや、偶発的なアクションを測定するソースで、パフォーマンス・データはこの測定値から導き出されます。例えば、JMS (Java Message Service) サーバーに対してテスト・メッセージを定期的に送信し、サーバーから返されるメッセージを取得するとします。この場合、このメッセージの往復時間がこのサービスのパフォーマンスに関する推定測定値となります。

インファレンシャル・ソース (このインスタンスは、合成トランザクションと呼ばれます) は極めて重宝な場合があります。なぜなら、実際にアクティビティーと同じパスを進むことによって、複数のコンポーネントや段階的呼び出しを効率的に測定できるからです。比較的アクティブでない期間中にシステムの正常性を確認するにはダイレクト・ソースでは不十分なことがありますが、その場合、正常性を確認するために連続性を監視する上で、合成トランザクションは重要な役割を果たします。

収集およびコレクター

収集とは、PDS からパフォーマンス・データまたは可用性データを取得するプロセスのことです。ダイレクト PDS の場合は通常、コレクターがこれらのデータにアクセスするための何らかの API を実装します。この場合、コレクターは SNMP (Simple Network Management Protocol) または Telnet を使用してネットワーク・ルーターから統計を読み取ります。インファレンシャル PDS の場合には、コレクターが基礎となるアクションを実行および測定します。

トレースおよびトレーサー

トレースとは、測定値をコレクターからコア APM システムに転送するプロセスのことです。商用およびオープンソースの APM システムの多くには、トレースを目的とした API が用意されています。この記事のサンプル用には、汎用 Java トレーサー・インターフェースを実装しました。このインターフェースについては、次のセクションで詳しく検討します。

概して、APM システムはトレーサーから送られたデータを、カテゴリーで分類したある種の階層構造に体系化します。図 2 に、このデータ取り込みの一般的フローを示します。

図 2. 収集、トレース、APM システムへの取り込み

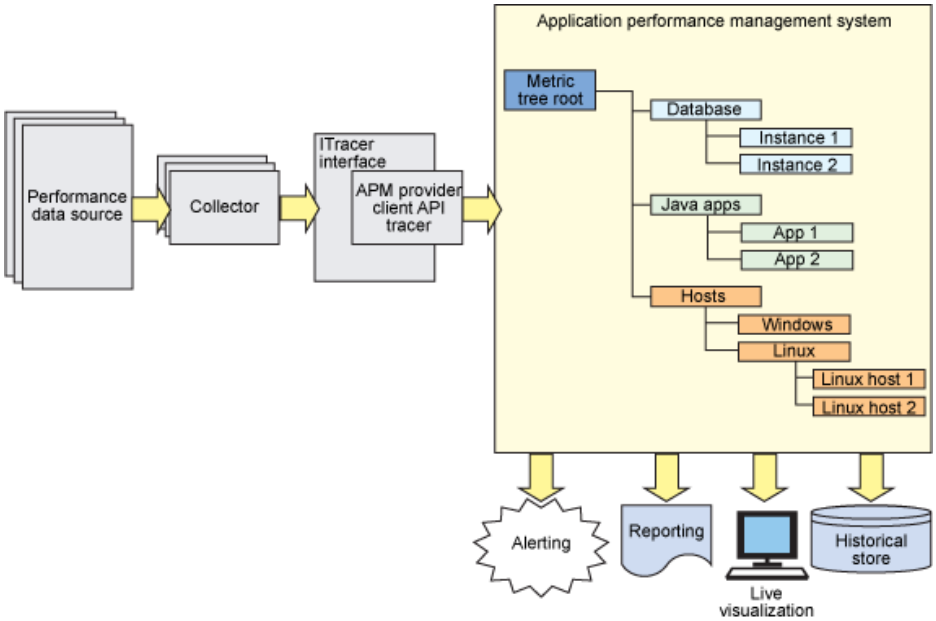


図 2 には、APM システムに共通して提供されるサービス (以下に記載します) もいくつか示されています。

- **ライブ視覚化** (Live visualization): 選択したメトリックをほぼリアルタイムで表示するグラフおよびチャート。
- **レポート** (Reporting): メトリック・アクティビティーに関して生成されるレポート。レポートに通常含まれているのは、あらかじめ決められている一連のレポート、カスタム・レポート、そしてデータを別の場所で使用するためのエクスポート機能です。
- **履歴ストア** (Historical store): 指定された時間枠でのグラフおよびチャートとレポートを表示できるように、未加工のメトリックまたは要約メトリックを保存する履歴データのストア。
- **アラート** (Alerting): 関係する個人またはグループに対し、収集されたメトリックから判断される特定の状態について通知する機能。典型的なアラート手段は、E メール、そして運用チームがイベントをイベント処理システムに伝播することを可能にするカスタム・フック・インターフェースです。

APM のターゲット環境全体で 1 つの共通したトレース API を実装して使用すると、一貫性がもたらされます。また、コレクターをカスタマイズする場合にも、開発者がトレースについて心配することなく、パフォーマンス・データの収集に専念できます。次のセクションでは、こうしたことを実現するための APM トレース・インターフェースを紹介します。

ITracer: トレーサー・インターフェース

Java 言語は、コレクターの実装言語として有効に機能します。Java 言語には以下の特徴が備わっているためです。

- 広範なプラットフォーム・サポート。ほとんどのターゲット・プラットフォームで、Java コレクター・クラスを変更することなくそのまま実行できます。そのため監視アーキテク

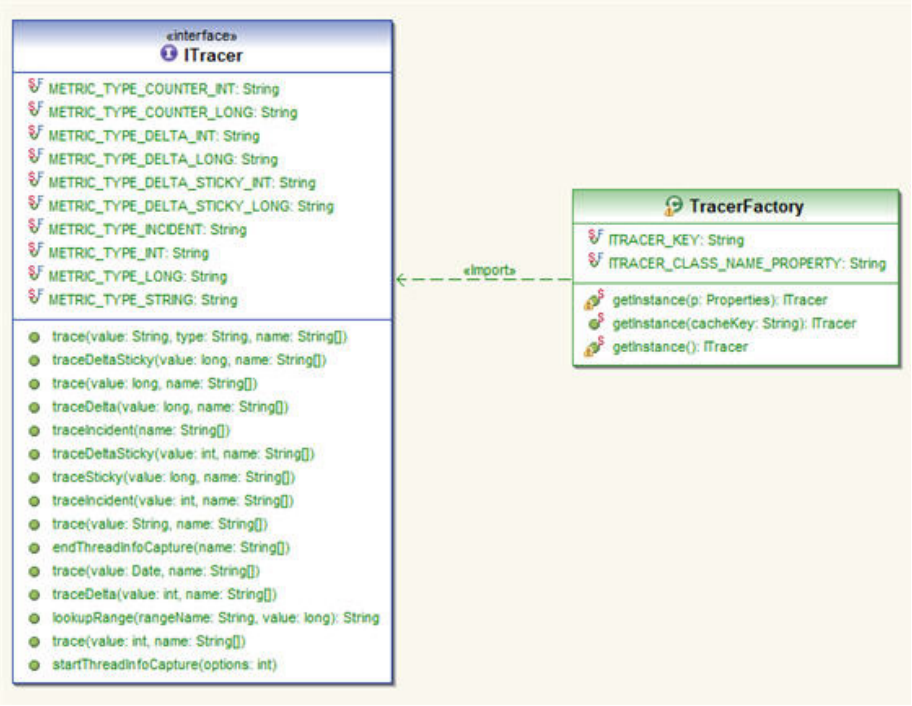
チャーには柔軟性がもたらされ、PDS でのコレクター・プロセスをローカルにまとめて配置し、リモート収集を不要にすることができます。
- 一般的に優れたパフォーマンス (ただし、使用可能なリソースによって異なります)。
- 強力な同時並行性と非同期実行のサポート。
- 多種多様な通信プロトコルのサポート。
- サード・パーティーの API による広範なサポート。例えば JDBC 実装、SNMP、独自仕様の Java インターフェースなどがサポートされることから、さまざまなコレクターがサポートされます。
- アクティブなオープンソース・コミュニティーによるサポート。コミュニティーによって、膨大な数のソースのデータにアクセスしたり、そこからデータを抽出するための Java 言語対応のツールおよびインターフェースが追加されています。

ただし、欠点もあります。それは、Java コレクターは、ターゲット APM システムによって提供されるトレース API と統合可能でなければならないことです。APM のトレース・メカニズムが Java インターフェースを提供しないとしても、パターンのいくつかは適用できます。けれどもターゲット PDS が完全に Java ベース (JMX など) で、アプリケーション・プラットフォームがそうでない場合は、ブリッジ・インターフェースが必要になります。その一例としては、Java を .NET で使用するためのコンパイラーを含む IKVM が挙げられます(「参考文献」を参照)。

トレース API には正式な標準がないため、提供元の APM 製品ごとにそれぞれ異なります。私はこの問題を取り除くために、汎用トレース Java インターフェース、`org.runtimemonitoring.tracing.ITracer` を実装しました。この ITracer インターフェースは、独自仕様のトレース API の汎用ラッパーです。この手法は、ソース・ベースがバージョンや API プロバイダーによって変わることを防ぐだけでなく、ラップされた API にはない機能を追加で実装する機会を与えます。この後に記載するサンプルの大半では、`ITracer` インターフェースと、このインターフェースがサポートする一般的な基本概念を実装します。

図 3 は、`org.runtimemonitoring.tracing.ITracer` インターフェースの UML クラス図です。

図 3. `ITracer` インターフェースとファクトリー・クラス



トレースのカテゴリーと名前

`ITracer` の大前提は、測定とそれに関連付けた名前を中央の APM システムに送信することです。このアクティビティーを実装する `trace` メソッドは、送信される測定データの性質によって異なります。各 trace メソッドは `String[] name` パラメーターを受け入れます。このパラメーターに含まれるのはコンテキストに依存した複合名のコンポーネントで、この複合名の構造は APM システムに固有のものです。複合名は APM システムに、送信される測定データが所属する名前空間と実際のメトリック名の両方を示します。したがって、複合名には通常、少なくともルート・カテゴリーと測定の記述が含まれることになります。基礎となる `ITracer` 実装は、渡された `String[]` からどのように複合名を作成するかを認識していなければなりません。表 1 に、複合名の命名規則の例を 2 つ記載します。

表 1. 複合名の例

名前の構造	複合名
単純なスラッシュ区切り構造	Hosts/SalesDatabaseServer/CPU Utilization/CPU3
JMX MBean `ObjectName`	com.myco.datacenter.apm:type=Hosts,service=SalesDatabaseServer,group=CPU Utilization,instance=CPU3

リスト 1 は、この API によるトレース呼び出しを簡略化した例です。

リスト 1. トレース API 呼び出しの例

```
ITracer simpleTracer = TracerFactory.getInstance(sprops);
ITracer jmxTracer = TracerFactory.getInstance(jprops);
.
.
simpleTracer.trace(37, "Hosts", "SalesDatabaseServer",
   "CPU Utilization", "CPU3", "Current Utilization %");
jmxTracer.trace(37,
   "com.myco.datacenter.apm",
   "type=Hosts",
   "service=SalesDatabaseServer",
   "group=CPU Utilization",
   "instance=CPU3", "Current Utilization %");
);
```

トレーサー測定値のデータ型

このインターフェースでは、測定値の型を以下のいずれかにすることができます。

- `int`
- `long`
- `java.util.Date`
- `String`

収集された測定値に対して、APM システムのプロバイダーが上記以外のデータ型をサポートすることも考えられます。

トレーサーのタイプ

測定値のデータ型を特定の 1 つ (`long` など) に指定したとすると、指定された値は APM システムでサポートされる型によってさまざまに解釈される可能性があります。また、基本的には同じ型でも、APM 実装ごとに異なる用語を使用する場合もありますが、`ITracer` が使用するのは一般的な命名方法であることに注意してください。

インターバル

トレーサーのタイプを話題に取り上げるには、インターバルという概念について説明しないわけにはいきません。あるオペレーションの経過時間を収集し、そのデータを APM システムにトレースするというプロセスの一般的な概念について考えてみてください。毎分、数百あるいは数千の呼び出しがあるはずですが、測定ごとにその詳細を送信して保存するのは効率的でないばかりか、1 つひとつの測定をパフォーマンスのレポートや視覚化に反映させるとなると、極めてまれな不規則な呼び出しでさえ情報の全体像を歪めてしまう可能性があります。それと同時に、長期間にわたってあまりにも全般的な平均値を収集するのでは細分度が失われてしまいます。その期間中に正当な理由があって起こった急激な値の上昇が重要な意味を持つこともあり得るからです。

この問題に対処するパターンは、集約対象の測定インターバルを細分度が最も小さいものにすることです。1 時間では長すぎる一方、200 ミリ秒では短すぎるため、30 秒のインターバルを設定したとします。架空のオペレーションを呼び出すごとにトレースを呼び出すことには変わりありませんが、各インターバルの終了時には以下のデータが保持されることになります。

- インターバルでの平均経過時間
- インターバルでの最大経過時間
- インターバルでの最小経過時間
- インターバル中の呼び出し回数
- 開始されたインターバルの終了時刻を示すタイム・スタンプ

これは、測定値を過度に集約するわけでもなければ、測定ごとに個々のデータを保存するわけでもない、効率的かつ有効な妥協策となります。

`ITracer` で表現されるトレーサー・タイプには以下の種類があります。

- **インターバル・アベレージ**:`trace(long value, String[] name)` メソッドと `trace(int value, String[] name)` メソッドは、あるインターバル内で平均化した値のトレースを実行します (囲み記事「インターバル」を参照)。これはつまり、送信される各測定データが現行インターバルの集約値の要素とされるということです。新しいインターバルが始まると、集約値カウンターはゼロにリセットされます。
- **スティッキー**:`raceSticky(value long, String[] name)` メソッドと traceSticky(value int, String[] name) メソッドは、スティッキー値のトレースを実行します。これは、インターバル・アベレージ・メトリックとは逆に、集約値がすべてのインターバルで保持されるということです。例えば今日、5 という値をトレースし、明日のある時点までは再びトレースを行わないとすると、新しい値が指定されるまで、このメトリックには 5 という値が保持されます。
- **デルタ**: デルタ・トレースは数値を渡しますが、APM システムに提供される (または APM システムによって解釈される) 実際の値は、この測定値と前の測定値との差分です。これは、レート・タイプと呼ばれることもあり、デルタ・タイプが適している実行内容を表した呼び方になっています。トランザクション・マネージャーの合計コミット数の測定を例に取ると、この値は常に増加するため、測定結果の絶対数は役に立ちません。この値で有益な点は値が上昇する比率 (レート) です。そのため、定期的に絶対数を収集し、その読み取り値の差分をトレースすれば、トランザクション・コミットのレートが反映されます。インターバル・アベレージを使用するケースは少ないですが、デルタ・トレースを使用するケースはインターバル・アベレージとスティッキーの間に位置します。デルタ・トレースでは、増加するのみの測定値を、増加も減少もする測定値から区別できなければなりません。送信された測定値が前の値よりも小さい場合は無視するか、あるいはそのデルタ・トレースをリセットする必要があります。
- **インシデント**: このタイプは、集約されない単純なメトリックで、特定のイベントがインターバル中に何回発生したかを示す増分カウントです。コレクターにしても、トレーサーにしても、指定された時点での合計発生回数を把握する必要はないため、基本の `traceIncident(String[] name)` 呼び出しには値がなく、このメソッドが呼び出されてから当該事象が発生するごとに暗黙的に値が増分されます。増分値を 1 より大きくし、このメソッドをループで複数回呼び出さないようにするには、`traceIncident(int value, String[] name)` メソッドを使用します。すると、合計が `value` 単位で増分されます。
- **スマート**: スマート・トレーサーは、パラメーターによって他のタイプのいずれかのトレーサーに対応付けられるタイプです。測定を行うための値とトレース・タイプは `String` として渡され、使用可能なタイプは定数としてインターフェースに定義されます。このタイプのトレーサーが重宝するのは、コレクターには収集されているデータの型またはトレーサー・タイプについての知識がまるでないけれども、収集された値と構成されたタイプ名を単にトレーサーに渡すようにコレクターに指示することは可能だという場合です。

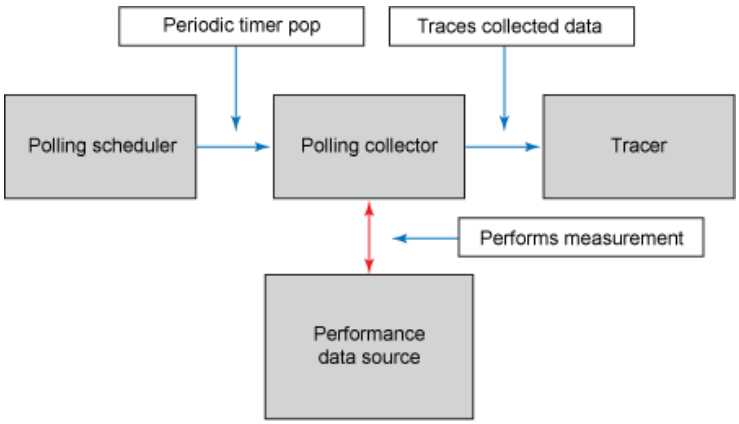
`TracerFactory` は、構成プロパティーに基づいて新しい `ITracer` インスタンスを作成したり、あるいはキャッシュから作成された `ITracer` を参照したりするために一般化されたファクトリー・クラスです。

コレクターのパターン

コレクターは一般に以下の 3 つのパターンのいずれかを使用します。コレクターのパターンによって、どのトレーサー・タイプを使用するかが決まってきます。

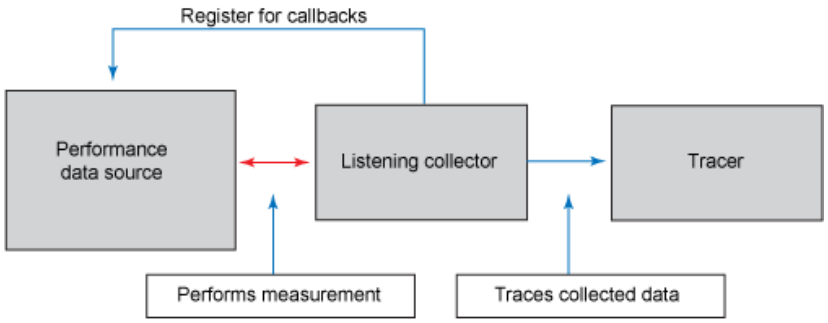
- **ポーリング**: このコレクターは一定の頻度で呼び出され、メトリックまたはメトリック・セットの現行の値を PDS から取得してトレースします。例えば、ホストの CPU 使用率を読み取ったり、トランザクション・マネージャーが JMX インターフェースからコミットしたトランザクションの合計数を読み取ったりするために、コレクターを毎分呼び出す場合が考えられます。ポーリング・パターンで前提となるのは、ターゲット・メトリックの定期的サンプリングです。したがって、ポーリング・イベントではメトリックの値が APM システムに提供されますが、ポーリングの合間では、その値は変わらないという前提となります。そのため、ポーリング・コレクターでは通常、スティッキー・トレーサーを使用します。APM システムはこの値を、すべてのポーリング・イベント間で不変の値としてレポートします。図 4 に、このパターンを示します。

  図 4. ポーリング収集パターン



- **リスニング**: この汎用データ・パターンは、オブザーバー・パターンの形をとります。このコレクターはターゲット PDS にイベントのリスナーとして自己登録するため、対象のイベントが発生したときには常にコールバックを受け取ることになります。コールバックの結果として送出される可能性のあるトレース値はコールバックのペイロード自体の内容に依存しますが、コレクターは少なくともコールバックのたびにインシデントをトレースすることができます。図 5 に、このパターンを示します。

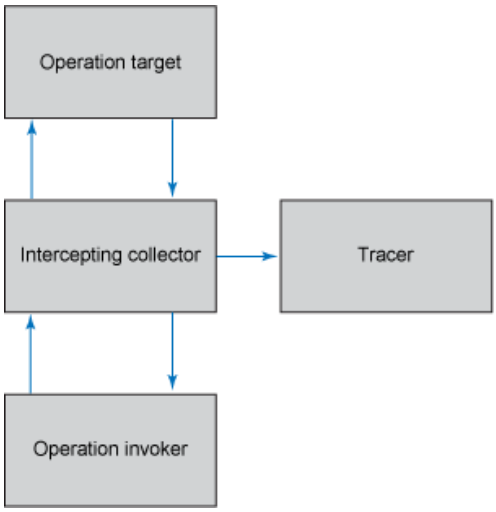
  図 5. リスニング収集パターン



- **インターセプト**: このパターンでは、コレクターがインターセプターとして、ターゲットとその呼び出し側の間に介入します。インターセプターを通過するアクティビティーが発生するたびに、このインターセプターが測定を行って測定値をトレースします。インターセプト・パターンがリクエスト/レスポンスである場合は、コレクターがリクエスト数、レスポンス時間、そして場合によってはリクエストまたはレスポンスのペイロードを測定することができます。例えば、コレクターとしても機能する HTTP プロキシー・サーバーは、以下の内容を測定することができます。
  - リクエスト数。オプションで、HTTP リクエスト・タイプ (`GET`、`POST` など) または URI (Uniform Resource Identifier) ごとのリクエスト数をカウントします。
  - リクエストに対する応答時間
  - リクエストおよびレスポンスのサイズ

  インターセプト・コレクターはすべてのイベントを「確認している」とみなすことができるため、実装するトレーサー・タイプは通常、インターバル・アベレージとなります。したがって、アクティビティーが何もないままインターバルが終了すると、そのインターバルの集約値は、前のインターバルでのアクティビティーとは関係なくゼロとなります。図 6 に、このパターンを示します。

図 6. インターセプト収集パターン



パフォーマンス・データ・トレース API とその基礎となるデータ型、そしてデータ収集のパターンについての概要は以上です。ここからは、この API が実際に活躍する特定の使用事例とサンプルを紹介します。

JVM の監視

パフォーマンス監視の実装に着手するには、JVM そのものが妥当な場所となります。まずは、すべての JVM に共通のパフォーマンス・メトリックから始め、それからエンタープライズ・アプリケーションで一般的に見られる JVM 常駐のコンポーネントに移ることにします。Java アプリケーションのインスタンスはほとんど例外なく、基礎となるオペレーティング・システムのサポートによって処理されるため、JVM 監視のいくつかの側面は JVM をホストする OS の観点から検討するのが最も適切です。これについては、第 3 回で取り上げます。

Java SE (Java Platform, Standard Edition 5) がリリースされるまでは、実行時に効率的かつ確実に収集できる内部の標準 JVM 診断はかなり限られていました。現在は、Java SE 5 (およびそれ以降) 準拠のすべての JVM バージョンで標準となっている java.lang.management インターフェースにより、複数の便利な監視ポイントを使用できるようになっています。これらの JVM の一部の実装では独自のメトリックを追加で指定していますが、アクセス・パターンに関してはほぼ同じです。ここでは JVM の以下の MXBean、つまり VM 内にデプロイされた管理および監視インターフェースを公開する JMX MBean を使ってアクセス可能な標準メトリックに焦点を絞ります。 (「参考文献」を参照)。

- `ClassLoadingMXBean`: クラス・ロード・システムを監視します。
- `CompilationMXBean`: コンパイル・システムを監視します。
- `GarbageCollectionMXBean`: JVM のガーベッジ・コレクションを監視します。
- `MemoryMXBean`: JVM のヒープおよびヒープ以外のメモリー空間を監視します。
- `MemoryPoolMXBean`: JVM によって割り当てられたメモリー・プールを監視します。
- `RuntimeMXBean`: ランタイム・システムを監視します。この MXBean は有用な監視メトリックをほとんど提供しませんが、JVM の入力引数、開始時刻、およびアップ·タイムを提供します。いずれにしても、他のメトリックを導き出す要素として役立ちます。
- `ThreadMXBean`: スレッド・システムを監視します。

JMX コレクターの前提は、`MBeanServerConnection` を取得することです。このオブジェクトは、JVM にデプロイされた MBean から属性を読み取ってターゲット属性の値を読み取り、両方の属性を `ITracer` API によってトレースします。このタイプの収集で重要な決定事項となるのは、コレクターをどこにデプロイするかです。選択肢には、ローカル・デプロイメントとリモート・デプロイメントがあります。

アクセス権の問題

ローカル・デプロイメントとリモート・デプロイメントはいずれも、コレクターが JVM データにアクセスできないようにする、各種の構成可能なアクセス権によって制限される場合があります。ほとんどのアクセス権には回避方法がありますが、その方法は実にさまざまなので、この問題についてはこの記事では説明しません。

ローカル・デプロイメントでは、コレクターとその呼び出しスケジューラーが、ターゲット JVM 自体のなかにデプロイされます。この場合、JMX コレクターのコンポーネントは、JVM 内で静的にアクセス可能な `MBeanServerConnection` である `PlatformMBeanServer` を使用して MXBean にアクセスします。一方、リモート・デプロイメントではコレクターが独立したプロセスで動作し、JMX Remoting 形式でターゲット JVM に接続します。この方法はローカル・デプロイメントに比べて効率性に劣るかもしれませんが、追加コンポーネントをターゲット・システムにデプロイする必要はなくなります。JMX Remoting についてはこの記事では説明しませんが、`RMIConnectorServer` をデプロイするか、あるいは単に JVM で外部接続を有効にすることによって簡単に実現することができます (「参考文献」を参照)。

サンプル JMX コレクター

この記事のサンプル JMX コレクター (記事の完全なソース・コードは「ダウンロード」を参照) には、`MBeanServerConnection` を取得するために 3 つの個別メソッドが含まれています。この 3 つのメソッドを使用して、コレクターは以下のことを実行できます。

- 静的 `java.lang.management.ManagementFactory.getPlatformMBeanServer()` メソッドを呼び出して、ローカル JVM のプラットフォーム `MBeanServer` に対する `MBeanServerConnection` を取得します。
- 静的 `javax.management.MBeanServerFactory.findMBeanServer(String agentId)` メソッドを呼び出して、JVM のプラットフォームにローカルにデプロイされたセカンダリー MBeanServer に対する `MBeanServerConnection` を取得します。1 つの JVM には複数の `MBeanServer` を常駐させられることに注意してください。そのため、Java EE (Java Platform, Enterprise Edition) サーバーなどの複雑なシステムには必ずと言っていいほど、プラットフォーム `MBeanServer` とは独立したアプリケーション・サーバー固有の `MBeanServer` があります (補足記事「MBean の多重登録」を参照)。
- `javax.management.remote.JMXServiceURL` メソッドを使用して、標準 RMI リモーティングの手段でリモート `MBeanServerConnection` を取得します。

リスト 2 は、`JMXCollector collect()` メソッドの一部を省略したスニペットです。ここに、`ThreadMXBean` によるスレッドのアクティビティーの収集とトレースが示されています。

リスト 2. サンプル JMX コレクターの `collect()` メソッドで ThreadMXBean を使用している部分

```
.
.
objectNameCache.put(THREAD_MXBEAN_NAME, new ObjectName(THREAD_MXBEAN_NAME));
.
.
public void collect() {
   CompositeData compositeData = null;
   String type = null;
   try {
      log("Starting JMX Collection");
      long start = System.currentTimeMillis();
      ObjectName on = null;
.
.
      // Thread Monitoring
      on = objectNameCache.get(THREAD_MXBEAN_NAME);
      tracer.traceDeltaSticky((Long)jmxServer.getAttribute(on,"TotalStartedThreadCount"),
        hostName, "JMX", on.getKeyProperty("type"), "StartedThreadRate");
      tracer.traceSticky((Integer)jmxServer.getAttribute(on, "ThreadCount"), hostName,
        "JMX", on.getKeyProperty("type"), "CurrentThreadCount");
.
.
      // Done
      long elapsed = System.currentTimeMillis()-start;
      tracer.trace(elapsed, hostName, "JMX", "JMX Collector",
         "Collection", "Last Elapsed Time");
      tracer.trace(new Date(), hostName, "JMX", "JMX Collector",
         "Collection", "Last Collection");
      log("Completed JMX Collection in ", elapsed, " ms.");
   } catch (Exception e) {
      log("Failed:" + e);
      tracer.traceIncident(hostName, "JMX", "JMX Collector",
         "Collection", "Collection Errors");
   }
}
```

完全なリストは次のようになります。

JMX collector's`collect()`メソッド

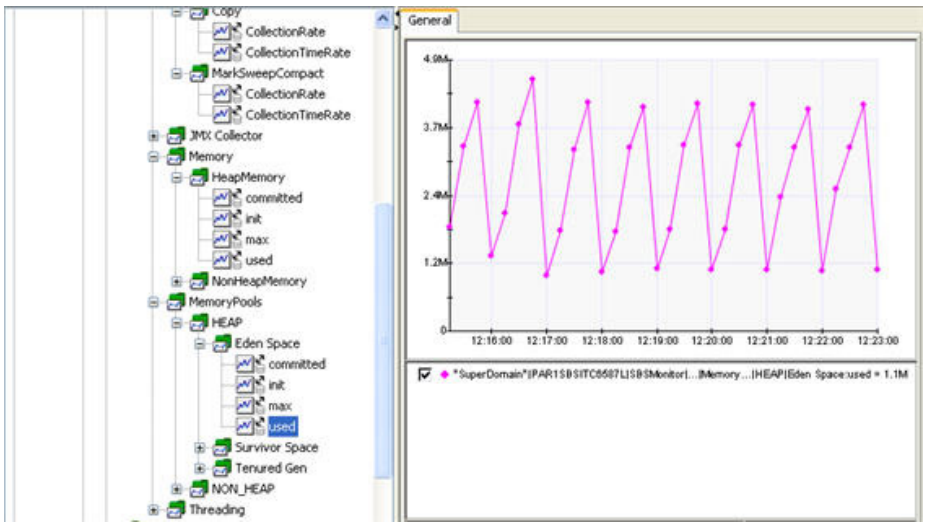
```
public void collect() {
   CompositeData compositeData = null;
   String type = null;
   try {
      log("Starting JMX Collection");
      long start = System.currentTimeMillis();
      ObjectName on = null;
      // Class Loading Monitoring
      on = objectNameCache.get(CLASS_LOADING_MXBEAN_NAME);
      tracer.traceDeltaSticky((Long)jmxServer.getAttribute(on, "TotalLoadedClassCount"),
        hostName, "JMX", on.getKeyProperty("type"), "TotalLoadedClassCount");
      tracer.traceDeltaSticky((Long)jmxServer.getAttribute(on, "UnloadedClassCount"),
        hostName, "JMX", on.getKeyProperty("type"), "UnloadedClassCount");
      tracer.traceSticky((Integer)jmxServer.getAttribute(on, "LoadedClassCount"),
        hostName, "JMX", on.getKeyProperty("type"), "LoadedClassCount");
      // Compilation Monitoring
      on = objectNameCache.get(COMPILATION_MXBEAN_NAME);
      tracer.traceDeltaSticky((Long)jmxServer.getAttribute(on, "TotalCompilationTime"),
        hostName, "JMX", on.getKeyProperty("type"), "CompilationTime");
      // Thread Monitoring
      on = objectNameCache.get(THREAD_MXBEAN_NAME);
      tracer.traceDeltaSticky((Long)jmxServer.getAttribute(on,"TotalStartedThreadCount"),
        hostName, "JMX", on.getKeyProperty("type"), "StartedThreadRate");
      tracer.traceSticky((Integer)jmxServer.getAttribute(on, "ThreadCount"), hostName,
        "JMX", on.getKeyProperty("type"), "CurrentThreadCount");
      // Memory Usage
      on = objectNameCache.get(MEMORY_MXBEAN_NAME);
      compositeData = (CompositeData)jmxServer.getAttribute(on, "HeapMemoryUsage");
      for(Object key: compositeData.getCompositeType().keySet()) {
         tracer.traceSticky((Long)compositeData.get(key.toString()), hostName, "JMX",
          "Memory", "HeapMemory", key.toString());
      }
      compositeData = (CompositeData)jmxServer.getAttribute(on, "NonHeapMemoryUsage");
      for(Object key: compositeData.getCompositeType().keySet()) {
         tracer.traceSticky((Long)compositeData.get(key.toString()), hostName, "JMX",
          "Memory", "NonHeapMemory", key.toString());
      }
      // Memory Pool Monitoring
      if(!objectNameQueryCache.containsKey("MP_MBEAN_QUERY")) {
         objectNameQueryCache.put("MP_MBEAN_QUERY",(Set<ObjectName>)
             jmxServer.queryNames(objectNameCache.get("MP_MBEAN_QUERY"), null));
      }
      for(ObjectName mpOn: objectNameQueryCache.get("MP_MBEAN_QUERY")) {
         compositeData = (CompositeData)jmxServer.getAttribute(mpOn, "Usage");
         type = (String)jmxServer.getAttribute(mpOn, "Type");
         for(Object key: compositeData.getCompositeType().keySet()) {
           tracer.traceSticky((Long)compositeData.get(key.toString()), hostName,
           "JMX","MemoryPools", type,
           mpOn.getKeyProperty("name"), key.toString());
         }
      }
      // Garbage Collection Monitoring.
      if(!objectNameQueryCache.containsKey("GC_MBEAN_QUERY")) {
         objectNameQueryCache.put("GC_MBEAN_QUERY",
            (Set<ObjectName>)jmxServer.queryNames(
            objectNameCache.get("GC_MBEAN_QUERY"), null));
      }
      for(ObjectName gcOn: objectNameQueryCache.get("GC_MBEAN_QUERY")) {
         tracer.traceDeltaSticky((Long)jmxServer.getAttribute(gcOn, "CollectionCount"),
            hostName, "JMX", gcOn.getKeyProperty("type"),
            gcOn.getKeyProperty("name"), "CollectionRate");
         tracer.traceDeltaSticky((Long)jmxServer.getAttribute(gcOn, "CollectionTime"),
            hostName, "JMX", gcOn.getKeyProperty("type"),
            gcOn.getKeyProperty("name"), "CollectionTimeRate");
      }
```

```
      // Done
      long elapsed = System.currentTimeMillis()-start;
      tracer.trace(elapsed, hostName, "JMX", "JMX Collector",
         "Collection", "Last Elapsed Time");
      tracer.trace(new Date(), hostName, "JMX", "JMX Collector",
         "Collection", "Last Collection");
      log("Completed JMX Collection in ", elapsed, " ms.");
   } catch (Exception e) {
      log("Failed:" + e);
      tracer.traceIncident(hostName, "JMX", "JMX Collector",
         "Collection", "Collection Errors");
   }
}
```

リスト 2 のコードは、`TotalThreadsStarted` と `CurrentThreadCount` の値をトレースします。これはポーリング・コレクターなので、どちらのトレースでもスティッキー・オプションを使用します。ただし、`TotalThreadsStarted` の値は常に増加するため、最も興味深い側面は絶対数ではなく、スレッドが作成されていくレートです。そのため、トレーサーは `DeltaSticky` オプションを使用しています。

図 7 に、このコレクターによって作成される APM メトリック・ツリーを示します。

図 7. JMX コレクターの APM メトリック・ツリー



JMX コレクターには、リスト 2 には示されていない側面がいくつかあります (ただし、完全なソース・コードには記載されています)。その 1 つは、10 秒ごとに `collect()` メソッドに対して定期的なコールバックを作成するスケジューリング登録です。

リスト 2 では以下のように、データ・ソースに応じて異なるトレーサー・タイプとデータ型を実装しています。

- `TotalLoadedClasses` と `UnloadedClassCount` はスティッキー・デルタとしてトレースされます。それは、この 2 つが常に増加する値で、クラス・ロードのアクティビティーを測定する手段としては絶対数より差分のほうがおそらく有益なためです。
- `ThreadCount` は可変の値で、増加することもあれば、減少することもあります。そのため、スティッキーとしてトレースされます。

- `Collection Errors` は、インターバル・インシデントとしてトレースされ、収集中に例外が発生すると増分されます。

ターゲット MXBean の JMX `ObjectName` はターゲット JVM の動作期間中に変更されることはないので、コレクターは効率性を追求し、`ManagementFactory` 定数名を使用して名前をキャッシュに入れます。

2 つの MXBean のタイプ、`GarbageCollector` と `MemoryPool` では正確な`ObjectName` は前もってわからない可能性がありますが、一般的なパターンを指定することはできます。この場合、最初に収集を行うときに、`MBeanServerConnection` に対してクエリーを実行し、指定したパターンと一致するすべての MBean のリストを要求してください。ターゲット JVM の動作期間中にさらにクエリーを行わなくても済むように、一致結果として返される MBean `ObjectName` はキャッシュに入れられます。

場合によっては、コレクターのターゲット MBean 属性が単純な数値型ではないことがあります。これに該当するのは、`MemoryMXBean` と `MemoryPoolMXBean` です。この 2 つの場合、属性タイプは `CompositeData` オブジェクトで、このオブジェクトに対してはキーと値が問い合わせられます。`java.lang.management` JVM 管理インターフェースでは MXBean 標準が JMX Open Type のモデルを採用します。つまり、すべての属性は `java.lang.Boolean` や `java.lang.Integer` など、言語に関係のない型になります。あるいは `javax.management.openmbean.CompositeType` のような複合型であれば、同じ単純な型のキーと値のペアに分解することができます。単純な型の完全なリストは、静的 `javax.management.openmbean.OpenType.ALLOWED_CLASSNAMES` フィールドに列挙されます。このモデルは、型の非依存性をサポートするため、JMX クライアントは非標準クラスに対して依存関係を持たないと同時に、基礎となる型が比較的単純であることから非 Java クライアントをサポートすることもできます。JMX Open Type についての詳細は、「参考文献」を参照してください。

ターゲット MBean 属性が標準以外の複合型である場合は、その型を定義するクラスがコレクターのクラスパスに確実に含まれていなければなりません。さらに、取得した複合オブジェクトから有用なデータをレンダリングするカスタム・コードを実装する必要もあります。

単一の接続が取得され、その接続がすべての収集に対して維持される場合には、接続に障害が起きたときに新しい接続を作成するためのエラー検出および復旧の処理が必要です。一部の収集用 API には、コレクターに対して接続を閉じてクリーンアップし、新しい接続を作成するように促す、接続リスナーが用意されています。保守のために停止状態になっているか、あるいはその他の理由でアクセスできない PDS に接続しようとした場合に備え、コレクターは再接続のためのポーリングを適切な頻度で行わなければなりません。接続の経過時間を追跡して、接続の低速化が検出された場合に収集の頻度を減らすのも有効です。こうすることにより、一時的にかなりの負荷がかかる可能性のあるターゲット JVM のオーバーヘッドを軽減することができます。

サンプルには実装されていませんが、JMX コレクターの効率性を向上させ、ターゲット JVM に対してコレクターを実行する上でのオーバーヘッドを軽減させるには、2 つの手法があります。そのうちの 1 つは、単一の MBean から複数の属性を問い合わせる場合に使える手法です。`getAttribute(ObjectName name, String attribute)` の代わりに `getAttributes(ObjectName name, String[] attributes)` を使用すれば、1 度に 1 つの属性を要求するのではなく、1 回の呼び出しで複数の属性に対するリクエストを実行することができます。この手法を使うことによる

違いはローカル収集ではごくわずかですが、リモート収集では、ネットワーク呼び出しの数を減らすことによってリソース使用量は大幅に削減されます。もう 1 つの手法は、ポーリング・パターンではなくリスニング・コレクター・パターンを実装することによって、JXM が公開するメモリー・プールのポーリング・オーバーヘッドを軽減するというものです。`MemoryPoolMXBean` は、使用量しきい値の設定機能をサポートします。つまり、設定したしきい値を超えるとリスナーに対する通知が起動され、それによってリスナーが値をトレースできるようにするということです。メモリー使用量が増えるにつれ、使用量しきい値は増加します。この手法の欠点は、使用量しきい値の増分値を極めて小さくしないと、データの細分度が失われ、しきい値を下回るメモリー使用量のパターンが隠されてしまうということです。

実装されていない手法にはもう 1 つ、経過時間全体と、ガーベッジ・コレクションが行われた合計の時間とを測定し、ガーベッジ・コレクションがアクティブになっている時間のパーセンテージを算出する単純な計算を実装するというものがあります。ほとんどのアプリケーションではガーベッジ・コレクションが (当面は) 避けられないことから、これは有益なメトリックです。それぞれ一定期間続くガーベッジ・コレクションの回数が予測されることになるため、ガーベッジ・コレクションの実行時間のパーセンテージが、JVM のメモリー正常性のコンテキストをより明らかにします。一般的な経験則として (ただし、アプリケーションごとにかなり差があります)、ガーベッジ・コレクションの実行時間が 15 分間で 10 パーセントを超えている場合、問題が潜んでいる可能性を示唆します。

コレクターの外部構成

このセクションで概説する JMX コレクターは収集プロセスを説明するために単純化してあるものの、このコレクターは徹底して常に収集をハード・コーディングしています。理想としては、コレクターがデータ・アクセスの方法を実装し、外部に提供された構成がその内容を指定することです。このような設計により、コレクターは遥かに実用的で再利用しやすくなります。再利用レベルを最大限にするには、外部で構成したコレクターが以下の構成の点をサポートすることが条件となります。

コレクターに対して、PDS への接続に使用するインターフェースと接続時に使用する構成を指定する PDS 接続ファクトリー・ディレクティブ

- 収集の頻度
- 再接続を試行する頻度
- 収集対象の MBean、またはワイルド・カード・オブジェクト名
- 測定のトレース先とするターゲットごとのトレース・コンポーネント名またはフラグメント、およびトレースに適用するデータ型

リスト 3 に、JMX コレクターの外部構成を記載します。

リスト 3. JMX コレクターの外部構成例

```
<?xml version="1.0" encoding="UTF-8"?>
<JMXCollector>
   <attribute name="ConnectionFactoryClassName">
      collectors.jmx.RemoteRMIMBeanServerConnectionFactory
   </attribute>
   <attribute name="ConnectionFactoryProperties">
      jmx.rmi.url=service:jmx:rmi://127.0.0.1/jndi/rmi://127.0.0.1:1090/jmxconnector
   </attribute>
```

```
   <attribute name="NamePrefix">AppServer3.myco.org,JMX</attribute>
   <attribute name="PollFrequency">10000</attribute>
   <attribute name="TargetAttributes">
      <TargetAttributes>
         <TargetAttribute objectName="java.lang:type=Threading"
            attributeName="ThreadCount" Category="Threading"
            metricName="ThreadCount" type="SINT"/>
         <TargetAttribute objectName="java.lang:type=Compilation"
            attributeName="TotalCompilationTime" Category="Compilation"
            metricName="TotalCompilationTime" type="SDINT"/>
      </TargetAttributes>
   </attribute>
</JMXCollector>
```

`TargetAttribute` 要素に含まれる、`type` という属性に注目してください。この属性は、スマート・タイプのトレーサーに対してパラメーター化された引数を表します。`SINT` 型は増分する `int` を表し、`SDINT` 型はデルタ・スティッキー `int` を表します。

JMX によるアプリケーション・リソースの監視

これまで説明してきた JMX による監視は、標準 JVM リソースだけが対象でした。けれども Java EE をはじめとする多くのアプリケーション・フレームワークは、ベンダーそれぞれの重要なアプリケーション固有のメトリックを JMX によって公開することができます。その典型的な例は、`DataSource` 使用率です。`DataSource` とは、外部リソース (最も一般的にはデータベース) への接続をプールして同時接続の数を制限し、リソースの誤った振る舞いを防いだり、アプリケーションに負担がかからないようにしたりするサービスのことです。データ・ソースの監視は、監視計画全体のなかで重要な部分を占めます。このプロセスは、JMX の抽象化層のおかげで、今まで説明してきたプロセスと変わりません。

以下に、JBoss 4.2 アプリケーション・サーバーのインスタンスから引用した典型的なデータ・ソース・メトリックを挙げます。

- **使用可能な接続数**: プール内で現在使用可能な接続の数。
- **接続数**: プール内にある接続からデータベースへの実際の物理接続の数。
- **最大使用接続数**: プール内で使用中の最大接続数。
- **使用中の接続数**: 現在使用中の接続数。
- **作成済み接続数**: このプールで作成された接続の合計数。
- **破棄済み接続数**: このプールで破棄された接続の合計数。

今回、コレクターはバッチ属性検索を使用して、1 回の呼び出しですべての属性を収集します。ここで 1 つ注意しなければならないのは、返されたデータを問い合わせ、異なるデータ型とトレーサー・タイプを有効にする必要があるという点です。また、アクティビティーが行われなければ `DataSource` メトリックはほとんど変化しないため、数値の変化を見るにはある程度の負荷を生成しなければなりません。リスト 4 に、DataSource コレクターの `collect()` メソッドを記載します。

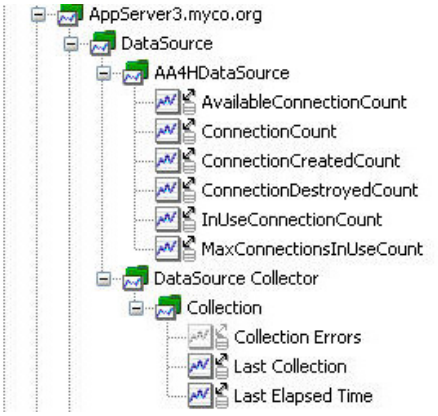
リスト 4. DataSource コレクター

```
public void collect() {
   try {
      log("Starting DataSource Collection");
      long start = System.currentTimeMillis();
```

```
      ObjectName on = objectNameCache.get("DS_OBJ_NAME");
      AttributeList attributes  = jmxServer.getAttributes(on, new String[]{
            "AvailableConnectionCount",
            "MaxConnectionsInUseCount",
            "InUseConnectionCount",
            "ConnectionCount",
            "ConnectionCreatedCount",
            "ConnectionDestroyedCount"
      });
      for(Attribute attribute: (List<Attribute>)attributes) {
         if(attribute.getName().equals("ConnectionCreatedCount")
            || attribute.getName().equals("ConnectionDestroyedCount")) {
               tracer.traceDeltaSticky((Integer)attribute.getValue(), hostName,
               "DataSource", on.getKeyProperty("name"), attribute.getName());
         } else {
            if(attribute.getValue() instanceof Long) {
               tracer.traceSticky((Long)attribute.getValue(), hostName, "DataSource",
                  on.getKeyProperty("name"), attribute.getName());
            } else {
               tracer.traceSticky((Integer)attribute.getValue(), hostName,
                  "DataSource",on.getKeyProperty("name"), attribute.getName());
            }
         }
      }
      // Done
      long elapsed = System.currentTimeMillis()-start;
      tracer.trace(elapsed, hostName, "DataSource", "DataSource Collector",
         "Collection", "Last Elapsed Time");
      tracer.trace(new Date(), hostName, "DataSource", "DataSource Collector",
         "Collection", "Last Collection");
      log("Completed DataSource Collection in ", elapsed, " ms.");
   } catch (Exception e) {
      log("Failed:" + e);
      tracer.traceIncident(hostName, "DataSource", "DataSource Collector",
         "Collection", "Collection Errors");
   }
}
```

図 8 は、この DataSource コレクターに対応するメトリック・ツリーです。

図 8. DataSource コレクターのメトリック・ツリー



JVM 内のコンポーネントの監視

MBean の多重登録

多くの場合、ターゲットとする MBean は同じ JVM 内の異なる MBeanServer に登録することができます。例えば、java.lang MXBean はプラットフォーム・エージェント (JVM

MBeanServer とも呼ばれます) に登録され、JBoss サーバー内の DataSource MBean は jboss MBeanServer に登録されます。しかしリモートからの監視の実装では、このように登録するとそれぞれの MBeanServer ごとに 2 つのリモート接続が必要になるため、余分なオーバーヘッドがもたらされ、構成が複雑になります。その上、リモート接続に対するプラットフォーム MBeanServer の公開に伴うオーバーヘッドも追加されてしまいます。通常、このような場合には簡単に MBean を多重登録して、同じ MBeanServer インターフェースによってすべてのターゲット MBean が監視されるようにすることが可能です。この記事のソース・コードで、map-platform-mxbeans.bsh という名前の Bean-shell の例を参照してください。このスクリプトを JBoss サーバーにデプロイすると、そのスクリプトがプラットフォーム MBeanServer の MXBean を JBoss の MBeanServer に多重登録します。

このセクションでは、アプリケーション・コンポーネント、サービス、クラス、メソッドの監視に適用できる手法について取り上げます。興味の対象となる主な項目は以下のとおりです。

- **呼び出しレート**: サービスまたはメソッドが呼び出されるレート
- **呼び出し応答レート**: サービスまたはメソッドが応答するレート
- **呼び出しエラー・レート**: サービスまたはメソッドがエラーを生成するレート
- **呼び出し経過時間**: インターバル単位での呼び出しの平均、最小、最大経過時間
- **呼び出し並行性**: サービスまたはメソッドを同時に呼び出す実行スレッドの数

Java SE 5 (およびそれ以降) の `ThreadMXBean` の実装に用意されているメトリックを使用すれば、以下のメトリックを収集することもできます。

- **システムおよびユーザー CPU 時間**: メソッド呼び出しの所要 CPU 時間
- **待機回数と合計待機時間**: スレッドがメソッドまたはサービス呼び出し中に待機状態になった回数と合計経過時間。待機が発生するのは、あるスレッドが `WAITING` または `TIMED_WAITING` 状態になり、別のスレッドのアクティビティーが終わるのを待っている場合です。
- **ブロック回数と合計ブロック時間**: スレッドがメソッドまたはサービス呼び出し中に `BLOCKED` 状態になった回数と合計経過時間。ブロックが発生するのは、監視ロックが同期ブロック状態になるか、またはその状態に再入するのを、スレッドが待っている場合です。

上記をはじめとするメトリックは、代替ツール·セットとネイティブ・インターフェースを使用して決定することもできますが、そうすると通常はある程度のオーバーヘッドが伴うため、実稼働での実行時の監視には望ましくないメトリックになってしまいます。そうは言っても、メトリック自体のオーバーヘッドはその収集中でさえ、低いレベルです。上記のメトリックは傾向分析以外には役に立たない場合があります。また、他の手段では識別できない程度の、何かしらの原因による影響と相関させるのは至難の業です。

上記のすべてのメトリックは、対象のクラスとメソッドがパフォーマンス・データを収集してターゲット APM システムにトレースするようにインスツルメンテーションを追加するプロセスによって収集することができます。Java クラスに直接インスツルメンテーションを追加したり、あるいは Java クラスから間接的にパフォーマンス・メトリックを導き出したりするには、以下の手法があります。

- **ソース・コードのインスツルメンテーション**: 最も基本的な手法は、ソース・コードのレベルでインスツルメンテーションを追加することです。こうすれば、コンパイルしてデプロイしたクラスには、実行時にあらかじめインスツルメンテーションが含まれていることになります。これは場合によっては妥当な手法で、特定のプラクティスによって許容できるプロセスおよび投資になります。

- **インターセプト**: 測定とトレースを実行するインターセプターによって呼び出しを APM システムの方向へも転送することにより、ターゲットとするクラスやそのソース・コード、またはランタイム・バイトコードに手をつけることなく、正確かつ効率的な監視が可能になります。このプラクティスをかなり利用しやすくしているのは、多くの Java EE フレームワークやその他のよく使われる Java フレームワークに備わっている以下の性質です。
  - 構成による抽象化を優先します。
  - インターフェースによるクラスの注入と参照を有効にします。
  - 場合によっては、インターセプト・スタックの概念を直接サポートします。実行のフローは、構成で定義されたオブジェクトのスタックを通されます。このスタックの目的と設計は、呼び出しを受け入れ、呼び出しを使った操作を行ってから渡すというものです。
- **バイトコードのインスツルメンテーション**: これは、バイトコードをアプリケーション・クラスに注入するプロセスです。注入されたバイトコードによって追加される、パフォーマンス・データを収集するインスツルメンテーションは、基本的には新しいクラスの一部として呼び出されます。インスツルメンテーションは完全にコンパイルされたバイトコードです。このコードの実行パスの拡張は、データの収集が引き続き行われる一方で、できるだけ小規模に行われるため、このプロセスは極めて効率的なものとなります。また、元のソース・コードを変更する必要がなく、環境の構成の変更もおそらくは最小限で済むという長所もあります。さらに、バイトコード注入の一般的なパターンと手法では、多くのサード・パーティーのクラスの場合のように、ソース・コードが入手できないクラスとライブラリーにインスツルメンテーションを追加することもできます。
- **クラスのラップ**: 同じ機能を実装しながらもインスツルメンテーションが含まれる別のクラスで、ターゲットをラップまたは置換するというプロセスです。

この第 1 回では、ソース・コードをベースとしたインスツルメンテーションについてのみ取り上げます。インターセプト、バイトコードのインスツルメンテーション、そしてクラスのラップについては、第 2 回で詳しく説明します (インターセプト、バイトコードのインスツルメンテーション、クラスのラップは、オリジナルのソース・コードを変更しないという点では実質上同じですが、結果を達成するためのアクションには、各手法によって多少異なる意味あいがあります)。

非同期インスツルメンテーション

非同期インスツルメンテーションは、クラスのインスツルメンテーションにおける根本的な問題です。前のセクションではパフォーマンス・データを対象としたポーリングの概念を検討しました。ポーリングが適切に行われる場合は、コア・アプリケーションのパフォーマンスやオーバーヘッドには影響しません。その一方、アプリケーション・コード自体にインスツルメンテーションを追加すると、直接コア・コードを変更するため、その実行に影響を及ぼします。あらゆる類のインスツルメンテーションの第一の目標は何にもまして、害を及ぼさないことです。つまり、オーバーヘッドによる影響はほとんど無視できる程度に抑えなければなりません。測定を実行することによる極めて小さな影響までを排除することはできませんが、パフォーマンス・データを収集した後には、残りのトレース・プロセスを非同期にすることが重要となります。非同期トレースを実装するには、いくつかのパターンがあります。図 9 は、その方法を示す一般的な概要です。

図 9. 非同期トレース

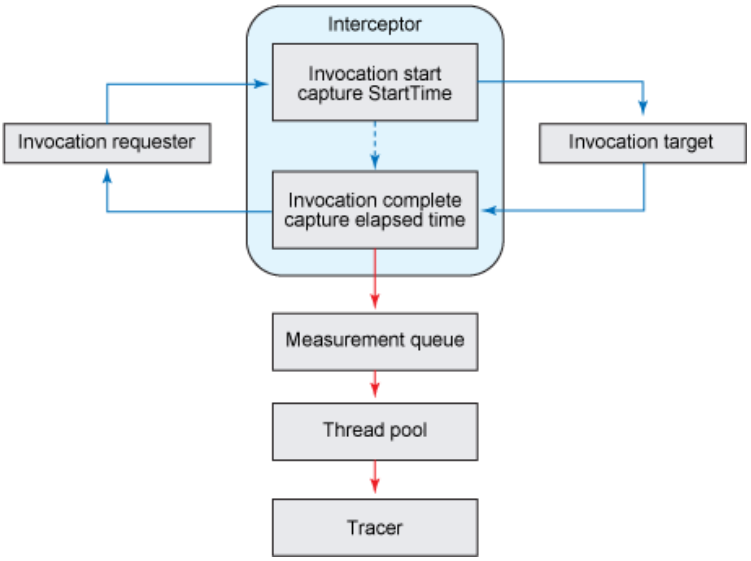


図 9 に示しているのは単純なインスツルメンテーション・インターセプターで、呼び出しの開始時刻と終了時刻を収集することによって経過時間を測定し、測定値 (経過時間とメトリック複合名) を処理キューに送ります。続いてスレッド・プールが処理キューを読み取り、測定値を取得してトレース・プロセスを完了します。

ソース・コードによる Java クラスのインスツルメンテーション

このセクションではソース・レベルのインスツルメンテーションという話題を取り上げ、ベスト・プラクティスとサンプル・ソース・コードを紹介します。また、ここでは新しいトレース構文も導入します。これらの構成についてはソース・コードにインスツルメンテーションを追加するというコンテキストで詳細を説明し、それぞれのアクションと実装パターンを明らかにするつもりです。

代替策は普及しているものの、ソース・コードにインスツルメンテーションを追加することが避けられない場合があります。場合によっては、それが唯一のソリューションになります。しかし、適切な事前の注意事項がわかっていれば、必ずしも悪いソリューションになるとは限りません。注意する点としては以下の内容があります。

- ソース・コードにインスツルメンテーションを追加するというオプションを選択できる一方、その効果をより高めるために構成を変更することが許されていない場合には、構成可能で柔軟性の高いトレース API が不可欠となります。
- 抽象化トレース API は log4j などのロギング API と似ており、以下の共通した特質があります。
  - **ランタイム冗長制御**: log4j ロガーとアペンダーの冗長レベルは開始時に構成し、実行時に変更することができます。同様に、トレース API でも階層化命名パターンに基づいて、トレースにどのメトリック名を有効にするかを制御することができます。
  - **出力エンドポイントの構成**: log4j はロギング・ステートメントをロガーによって発行し、そのロギング・ステートメントをアペンダーに送信します。アペンダーは、ログ・ストリームをファイル、ソケット、E メールといった各種の出力に送信するように構成

することができます。トレース API にはこのような出力の多様性は必要ありませんが、独自仕様あるいは APM システム固有のライブラリーを抽象化する機能によって、ソース・コードが外部構成によって変更されないようにします。
- 場合によっては、他の手段では特定の項目をトレースできないことがあります。通常、これが当てはまるのは私がコンテキスト依存トレースと呼んでいる場合です。私がこの用語を使って表しているのは、とりわけ重要ではないけれども、主要なデータにコンテキストを追加するパフォーマンス・データです。

コンテキスト依存トレース

コンテキスト依存トレースはその特定のアプリケーションによって大きく左右されますが、`processPayroll(long clientId)` メソッドによる給与計算処理クラスを単純化したようなものだと考えてください。このメソッドが呼び出されると、顧客の従業員ごとに給与を計算して保存します。このメソッドにはさまざまな手段でインスツルメンテーションの機能を持たせることができるはずですが、基本的な実行パターンからは、このメソッドが呼び出されて実行している時間の長さが従業員の数には比例していないことが明らかになります。したがって、`processPayroll` の経過時間の傾向を検討しても、その実行で何人の従業員の給与が計算されたのかがわからなければ何の判断もできません。もっと簡単に説明すると、特定期間の `processPayroll` の平均経過時間が x ミリ秒だったとします。しかしこの値が許容されるパフォーマンスを示すのか、あるいはパフォーマンス不足を示すのかはわかりません。この時間枠でたった 1 人の従業員の給与しか計算していなければ、パフォーマンス不足であると理解できますが、150 人の従業員の分が計算されたのであれば、非常に早い処理速度だと考えられるからです。リスト 5 に、この単純化した概念のコードを示します。

リスト 5. コンテキスト依存トレースの場合

```
public void processPayroll(long clientId) {
   Collection<Employee> employees = null;
   // Acquire the collection of employees
   //...
   //...
   // Process each employee
   for(Employee emp: employees) {
      processEmployee(emp.getEmployeeId(), clientId);
   }
}
```

ここで大きな課題となるのは、ほとんどのインスツルメンテーション手法では、`processPayroll()` の内側には手を付けられないという点です。そのため、`processPayroll` やさらには `processEmployee` にインスツルメンテーションの機能を持たせられるとしても、従業員の数をトレースしてメソッドのパフォーマンス・データにコンテキストを与える手段はありません。リスト 6 は、対象のコンテキスト依存データを捕捉する方法を不完全にハード・コーディングした (いくぶん非効率的な) 例です。

リスト 6. コンテキスト依存トレースの例

```
public void processPayrollContextual(long clientId) {
   Collection<Employee> employees = null;
   // Acquire the collection of employees
   employees = popEmployees();
   // Process each employee
   int empCount = 0;
   String rangeName = null;
   long start = System.currentTimeMillis();
   for(Employee emp: employees) {
      processEmployee(emp.getEmployeeId(), clientId);
      empCount++;
   }
   rangeName = tracer.lookupRange("Payroll Processing", empCount);
   long elapsed = System.currentTimeMillis()-start;
   tracer.trace(elapsed, "Payroll Processing", rangeName, "Elapsed Time (ms)");
   tracer.traceIncident("Payroll Processing", rangeName, "Payrolls Processed");
log("Processed Client with " + empCount + " employees.");
}
```

リスト 6 で肝心な部分は、`tracer.lookupRange` の呼び出しです。「範囲」は名前付きコレクションで、これらのコレクションには数値範囲の限界によってキーが設定され、数値範囲の名前を表す `String` 値があります。給与計算の単純で変化に乏しい経過時間をトレースするのではなく、リスト 6 では、経過時間をその変化がわかるよう効果的に区別し、同じような従業員数ごとにグループ化することで、従業員数をいくつかの範囲に区分けしています。図 10 に、この APM システムによって生成されたメトリック・ツリーを示します。

図 10. 範囲によってグループ化した給与計算処理時間

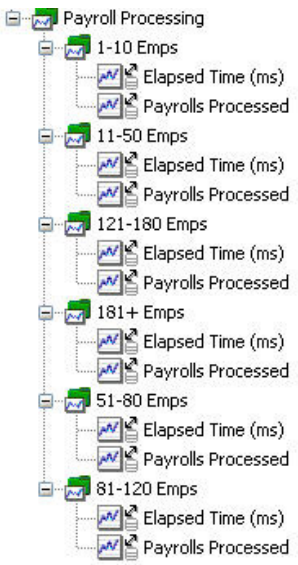
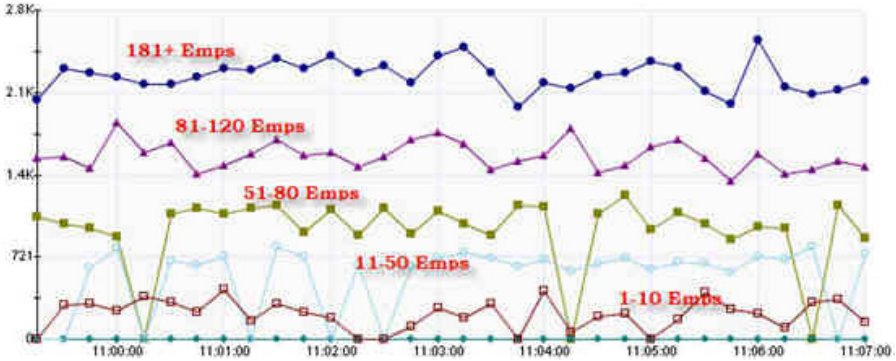


図 11 に示しているのは、従業員数で区分された給与計算処理の経過時間です。この図では、従業員数と経過時間の相対関係が明らかにわかります。

図 11. 範囲別の給与計算処理経過時間



トレーサー構成のプロパティーには、範囲としきい値を定義できるプロパティー・ファイルへの URL を含めることができます (しきい値については、この後すぐに説明します。)トレーサーの作成時に読み取られる、これらのプロパティーは、`tracer.lookupRange` 実装のバッキング・データを提供します。リスト 7 に、`Payroll Processing` 範囲の構成例を記載します。ここでは、`java.util.Properties` の XML 表現を使うことにしました。このほうが、型破りな特徴でも許容しやすいからです。

リスト 7. 範囲構成の例

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
   <comment>Payroll Process Range</comment>
   <entry key="L:Payroll Processing">181+ Emps,10:1-10 Emps,50:11-50 Emps,
      80:51-80 Emps,120:81-120 Emps,180:121-180 Emps</entry>
</properties>
```

外部で定義された範囲を注入すれば、アプリケーションが予測値の調整やビジネス上の要因によるサービス・レベル・アグリーメント (SLA) の変更に応じて絶えずソース・コード・レベルで更新する必要がなくなります。範囲としきい値の変更を適用するには、外部ファイルを更新するだけで済み、アプリケーション自体を更新する必要はありません。

しきい値と SLA の追跡

外部で構成可能なコンテキスト依存トレースの柔軟性により、より正確でより詳細なパフォーマンス「しきい値」を定義し、測定することが可能になります。「範囲」が測定をカテゴリー分けするための一連の数値枠を定義する一方、「しきい値」は範囲をさらに細かくカテゴリー分けし、測定値の定義済み範囲に応じて、取得した測定値に等級を付けます。収集したパフォーマンス・データの分析に共通して求められるのは、「成功した」実行と、指定された時間内に発生しなかったために「失敗した」とみなされる実行とを決定し、レポートすることです。このデータの集合は、システム動作の正常性および容量に関する一般的なレポート・カード (成績表) として、あるいは何らかの形での SLA 準拠評価として必要となることがあります。

給与計算処理システムで例えると、内部サービス・レベルの目標は (定義された従業員数の範囲内での) 給与計算の実行時間を `Ok`、`Warn`、`Critical` の変動幅に定義することです。しきい値カウントを生成するプロセスは概念的には単純で、必要な作業としてはまず、各変動幅に対応するグループの経過時間上限と考える値をトレーサーに指定します。そしてカテゴリー分けされた経過時間に対する `tracer.traceIncident` と、それに続いて (レポートを単純化するため) 合計を送出す

るようにトレーサーに指示するだけです。表 2 に、説明のために仮定した SLA 経過時間をまとめます。

表 2. 給与計算処理のしきい値

従業員数	Ok (ミリ秒)	Warn (ミリ秒)	Critical (ミリ秒)
1-10	280	400	>400
11-50	850	1200	>1200
51-80	900	1100	>1100
81-120	1100	1500	>1500
121-180	1400	2000	>2000
181+	2000	3000	>3000

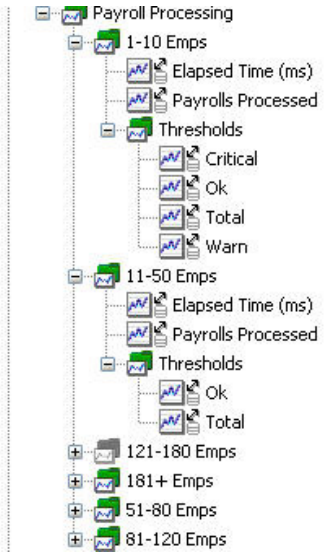
`ITracer` API は、前に説明した「範囲」と同じ XML (プロパティー) ファイルに定義した値を使用して、「しきい値」レポートの作成を実装します。「範囲」の定義と「しきい値」の定義は 2 つの点でわずかに異なります。その 1 つは、しきい値の定義ではキー値が正規表現であることです。`ITracer` は数値をトレースするときに、しきい値の正規表現がトレース対象のメトリックが持つ複合名と一致するかどうかをチェックします。一致する場合、しきい値によって測定値は `Ok`、`Warn`、または `Critical` の等級に分けられ、さらに `tracer.traceIncident` がトレースに追加されます。もう 1 つの違いは、しきい値が定義するのは 2 つの値 (`Critical` 値は `Warn` 値より大きいと定義されます) だけなので、構成は 2 つの数値のみからなるということです。リスト 8 に、前に概説した給与計算処理 SLA のしきい値構成を記載します。

リスト 8. 給与計算処理のしきい値構成

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
   <!-- Payroll Processing Thresholds -->
   <entry key="Payroll Processing.*81-120 Emps.*Elapsed Time \(ms\)">1100,1500</entry>
   <entry key="Payroll Processing.*1-10 Emps.*Elapsed Time \(ms\)">280,400</entry>
   <entry key="Payroll Processing.*11-50 Emps.*Elapsed Time \(ms\)">850,1200</entry>
   <entry key="Payroll Processing.*51-80 Emps.*Elapsed Time \(ms\)">900,1100</entry>
   <entry key="Payroll Processing.*121-180 Emps.*Elapsed Time \(ms\)">1400,2000</entry>
   <entry key="Payroll Processing.*181\+ Emps.*Elapsed Time \(ms\)">2000,3000</entry>
</properties>
```

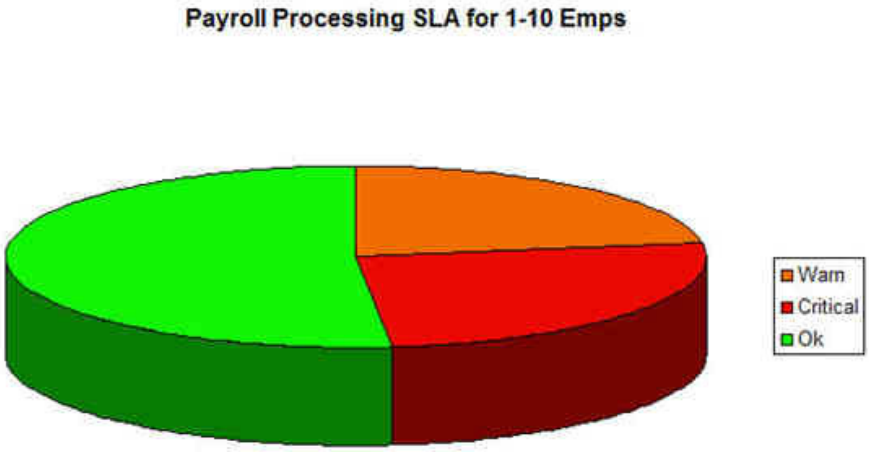
図 12 に、しきい値メトリックを追加した給与計算処理のメトリック・ツリーを示します。

図 12. しきい値を使用した給与計算処理のメトリック・ツリー



収集されたデータは、図 13 にように円グラフ形式で表示することができます。

図 13. 給与計算処理の SLA サマリー (1 人から 10 人の従業員の場合)



重要な点として、コンテキスト依存カテゴリーとしきい値カテゴリーの参照は、できるだけ効率的に短時間で実行されるようにしてください。なぜなら、このような参照は実際の作業を行うスレッドと同じスレッドで実行されるためです。`ITracer` 実装では、すべてのメトリック名がメトリック用に指定された (スレッド・セーフな) マップに保存されます。この場合、トレーサーが最初に確認した指定しきい値と一緒に保存されることもあれば、指定しきい値が保存されないこともあります。特定のメトリックに対する最初のトレース・イベントの後にしきい値 (またはしきい値を指定しないこと) の決定に費やされた時間がすなわち `Map` 参照時間で、通常は速度の点で問題ありません。ただし、しきい値のエントリー数や個々のメトリック名の数がかなり多い場合には、しきい値の決定を据え置き、非同期トレースのスレッド・プール・ワーカーで処理させるのが妥当なソリューションとなるはずです。

第 1 回のまとめ

連載第 1 回目のこの記事では、監視のアンチパターンとともに、APM システムに望ましい特質を説明しました。さらに一般的ないくつかのパフォーマンス・データ収集パターンの概要を説明し、`ITracer` インターフェースを紹介しました。このインターフェースは、この後の連載でも引き続き使用します。記事では続いて、JVM の正常性を監視する手法、そして JMX による一般的なパフォーマンス・データの取得方法についてサンプルを用いて説明し、最後に効率的でコードの変更に左右されないソース・レベルのインスツルメンテーションを実装する方法を簡単に説明しました。このインスツルメンテーションでは、そのままのパフォーマンス統計値とコンテキストに依存して得られた統計値を監視します。これらの統計値をアプリケーション SLA に関するレポート作成に利用する方法についても、この記事を読んで理解できたはずです。第 2 回では、アプリケーションのソース・コードを変更せずに、インターセプト、クラスのラップ、そして動的バイトコードのインスツルメンテーションによって Java システムにインスツルメンテーションの機能を持たせる手法を探ります。

第 2 回を読む。

ダウンロード

内容	ファイル名	サイズ
Sample code for this article	j-rtm1.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。