Java ランタイムの監視: 第 2 回 コンパイル後のインスツルメンテーションとパフォーマンスの監視

インターセプト、クラスのラップ、バイトコードによるインスツルメンテーション

Nicholas Whitehead
Senior Technology Architect
ADP

2008年 8月 05日

Java™ アプリケーションのランタイムの監視に関する 3 回連載の第 1 回目では、JVM の正常性、そしてパフォーマンス・メトリックを収集するためにソース・コードにインスツルメンテーションを追加する方法を取り上げました。第 2 回目となるこの記事では、オリジナルのソース・コードを変更せずに Java クラスと構成体にインスツルメンテーションを追加する手法を紹介します。

このシリーズの他の記事を見る

はじめに

この 3 回連載の第 1 回目で説明したように、Java アプリケーションに関する問題を確実に検出し、迅速に診断とトリアージ (問題に対処する優先度の判定) を行うには、実稼働環境で Java アプリケーションと、そのアプリケーションと依存関係を持つものに関して可用性とパフォーマンスを監視することが重要です。監視対象のクラスのインスツルメンテーションをソース・コード・レベルで行えば、第 1 回目で説明したメリットが得られますが、この方法が許容されない、あるいは実際的でないことは珍しくありません。例えば、監視の対象となるポイントの多くがサード・パーティーのコンポーネント内にあり、そのソース・コードが手元にない場合などです。そこでこの第 2 回では、オリジナルのソース・コードを変更せずに Java クラスとリソースの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。

ソース・コード外部にインスツルメンテーションを組み込む方法としては、以下のものがあります。

- インターセプト
- クラスのラップ
- バイトコードのインスツルメンテーション

この記事では、第 1 回で紹介したパフォーマンス・データのトレースを実装する ITracer インターフェースを使った例で、上記の手法を概説します。

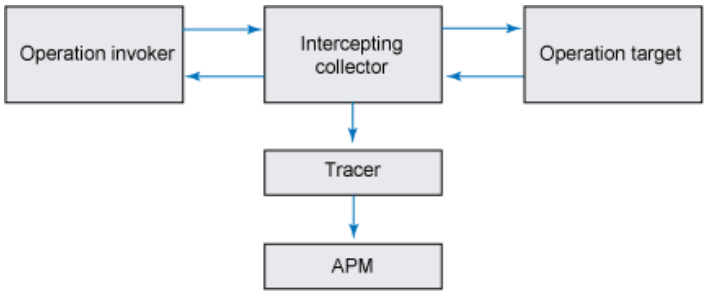
インターセプトによる Java インスツルメンテーション

インターセプトの大前提は、ターゲットを呼び出す際とそれに対する応答の際に受け渡される、構成体と情報の集合をインターセプトすることによって、特定パターンの呼び出しをアプリケーション・パフォーマンス管理 (APM) システムの方向にも転送するというものです。基本的なインターセプター実装では、以下のことを行います。

1. ターゲットが呼び出される際に、呼び出しリクエストの現在時刻を取得します。
2. ターゲットからの応答の際に、現在時刻を再度取得します。
3. この 2 つの測定値の差分として、経過時間を計算します。
4. 呼び出しの経過時間を APM システムに渡します。

このフローを図 1 に示します。

図 1. インターセプターによるパフォーマンス・データ収集の基本フロー



明確な一線

変更管理の熱烈な支持者は、ソース・コードによる変更と、構成による変更を区別することに異議を唱えるかもしれません。正直なところ、「コード」、「XML」、そして「スクリプト」を区別する線は多少曖昧になってきています。しかし、以下の 2 つの変更の間には、明確な一線が引かれます。

- ソース・コードを変更した後、コンパイル、パッケージ化、そして場合によっては終わりがないかのような一連のデプロイメント前の手順を必要とする変更
- (不変の) バイナリー・コード外部のリソースに対する変更

2 つの間の違いは、ロールフォーワードとロールバックの容易さなどではありません。この違いが理念上の分類に当てはまらない場合や、一部の環境では、その環境の複雑さや変更プロセスに要求される厳密さが、この 2 つの変更の間で大きく変わる場合もあります。

Java EE (Java Platform, Enterprise Edition) をはじめとする多くの Java フレームワークには、インターセプト・スタックのコア・サポートが組み込まれています。インターセプト・スタックでは、サービスの呼び出しが一連のプリプロセッシング・コンポーネント、ポストプロセッシング・コンポーネントに通されます。このようなスタックはインスツルメンテーションを実行過程に注入する絶好の場所となるとともに、2 つのメリットをもたらします。その 1 つは、呼び出しのターゲットとなるクラスのソース・コードを変更しなくても済むことです。そしてもう 1 つは、インターセプター・クラスを JVM のクラスパスに追加してコンポーネントのデプロイメント記述子を変更するだけで、インスツルメンテーション・インターセプターを実行フローに挿入できるという点です。

インターセプトのコア・メトリック

経過時間は、インターセプターで通常収集されるメトリックのひとつですが、それ以外の標準メトリックもインターセプト・パターンに適合します。ここで少し回り道をして、経過時間以外のメトリックをサポートする `ITracer` インターフェースの 2 つの新しい側面を紹介します。

インターセプターを使用するときに収集対象として検討することになる標準メトリックには、以下があります。

- **経過時間**: 実行を完了するまでの平均クロック時間
- **インターバル単位の呼び出しの数**: ターゲットが呼び出された回数
- **インターバル単位の応答の数**: ターゲットが呼び出しに応答した回数
- **インターバル単位の例外の数**: ターゲットの呼び出しが例外として終わった回数
- **並行性**: ターゲットを同時に実行するスレッドの数

以下の 2 つの `ThreadMXBean` メトリックも選択できますが、有用性は限られています。また、収集のコストも多少高くなります。

- **経過 CPU 時間**: 実行期間中にスレッドが使用した CPU 時間 (ナノ秒単位) です。CPU 使用率は一見すると有意義な情報のように思えますが、傾向パターンとして使用する以外はそれほど内容のある情報ではありません。あるいは、収集コストは遥かに高くなりますが、CPU リソース全体での、スレッドが実行に使用した CPU リソースのパーセンテージを概算することも可能です。
- **ブロック/待機の回数および時間**: 待機とは、特定のスレッド・スケジューリングによる同期あるいは待機状態を示します。ブロックが最もよく発生するのは、JDBC (Java Database Connectivity) 呼び出しに対するリモート・データベースからの応答を待機するなど、実行がリソース待機中の状態になった場合です (この記事の「JDBC のインスツルメンテーション」のセクションで、これらのメトリックが該当する図を参照してください)。

`ThreadMXBean` メトリックの収集方法を説明するため、リスト 1 でソース・ベースのインスツルメンテーションにいったん戻ります。このサンプルでは、`heavilyInstrumentedMethod` メソッドに対して大量のインスツルメンテーション・セットを実装しています。

リスト 1. 大量にインスツルメンテーションを追加したメソッドの例

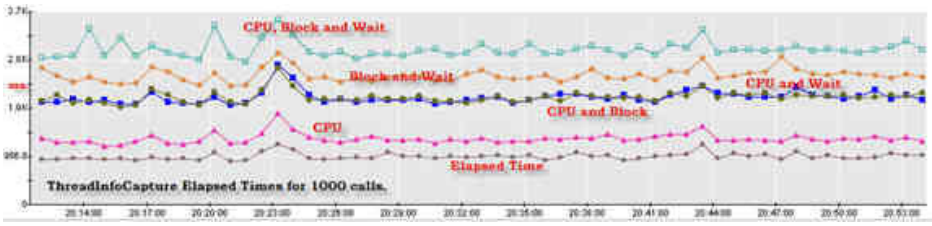
```
protected static AtomicInteger concurrency = new AtomicInteger();
.
.
for(int x = 0; x < loops; x++) {
   tracer.startThreadInfoCapture(CPU+BLOCK+WAIT);
   int c = concurrency.incrementAndGet();
   tracer.trace(c, "Source Instrumentation", "heavilyInstrumentedMethod",
      "Concurrent Invocations");
   try {
      // ==================================
      //   Here is the method
      // ==================================
      heavilyInstrumentedMethod(factor);
      // ==================================
      tracer.traceIncident("Source Instrumentation",
         "heavilyInstrumentedMethod", "Responses");
   } catch (Exception e) {
      tracer.traceIncident("Source Instrumentation",
         "heavilyInstrumentedMethod", "Exceptions");
```

```
   } finally {
      tracer.endThreadInfoCapture("Source Instrumentation",
         "heavilyInstrumentedMethod");
      c = concurrency.decrementAndGet();
      tracer.trace(c, "Source Instrumentation",
         "heavilyInstrumentedMethod", "Concurrent Invocations");
      tracer.traceIncident("Source Instrumentation",
         "heavilyInstrumentedMethod", "Invocations");
   }
   try { Thread.sleep(200); } catch (InterruptedException e) { }
}
```

リスト 1 では、2 つの新しい構成体を導入しています。

- **`ThreadInfoCapture` メソッド**: `ThreadInfoCapture` メソッドは、経過時間だけでなく、ターゲット呼び出し前と呼び出し後の `ThreadMXBean` メトリックの差分を取得するコンビニエンス・ヘルパーです。`startThreadInfoCapture` が現行スレッドの基準を取得し、`endThreadInfoCapture` が差分を計算してトレースします。これらのメトリックは絶えず増加するため、あらかじめ基準を取得し、後で差分を計算する必要がありますが、このようなシナリオはトレーサーのデルタ機能には適していません。それは、絶対値がスレッドごとに異なる上、実行中の JVM ではスレッドが常に同じというわけではないからです。さらに注目する点として、トレーサーはスタックを使用して基準を維持するため、呼び出しを (慎重に) ネストすることができます。このデータを収集するにはある程度のコストがかかります。図 2 に、異なる組み合わせで `ThreadMXBean` メトリックを収集した場合の経過時間を比較します。

  図 2. `ThreadMXBean` メトリック収集の相対コスト

  呼び出しを慎重に行うようにすれば膨大なオーバーヘッドになることはありませんが、例えば短いループ内では実行しないなど、ロギングの場合と同様の注意事項に従うのが有用です。
- **並行性**: 任意の時点でこのコードを実行しているスレッドの数を追跡するには、ターゲットとなるクラス (上記のサンプルでは、静的 `AtomicInteger`) のすべてのインスタンスに対してスレッド・セーフかつアクセス可能なカウンターを作成する必要があります。ただし、これは周知のとおり厄介なケースで、複数のクラス・ローダーがこのクラスをロードしてカウンターを非排他的にレンダリングし、測定値を完全に混乱させてしまう可能性があります。これに対する 1 つのソリューションは、JVM 内に 1 つしかないことが保証されている場所、例えばプラットフォーム・エージェント内の MBean などで並行性カウンターを管理することです。

並行性は、インスツルメンテーション・ターゲットがマルチスレッド化またはプールされる場合にしか適用できませんが、該当する場合には極めて貴重なメトリックとなります。これについてはこの後すぐに、EJB (Enterprise JavaBean) インターセプターとの関連で詳しく説明します。EJB インターセプターは、次に取り上げるインターセプト・ベースのインスツルメンテーションで最初に紹介するサンプルで、リスト 1 で検討したトレース・メソッドと同じメソッドを利用します。

EJB 3 インターセプター

EJB 3 のリリースを機に、インターセプターは Java EE アーキテクチャーの標準機能となっています (一部の Java アプリケーション・サーバーでは、これまでも EJB インターセプターをサポートしていました)。大抵の Java EE アプリケーション・サーバーには少なくとも主要なコンポーネント (EJB など) に関するパフォーマンス・メトリックのレポート機能は備わっていますが、以下の理由から、独自のレポート機能を実装する必要も考えられます。

- コンテキストや「範囲」/「しきい値」を利用したトレースが必要な場合
- アプリケーション・サーバーのメトリックで問題ないが、アプリケーション・サーバーのサイロ型のメトリックではなく、独自の APM システムでのメトリックが必要な場合
- アプリケーション・サーバーのメトリックが要件を満たしていない場合

そうは言っても、使用している APM システムとアプリケーション・サーバーの実装によっては、この作業の一部はユーザーに代わってすでに行われている場合があります。例えば、WebSphere® PMI はサーバー・メトリックを JMX (Java Management Extensions) によって公開しています (「参考文献」を参照)。APM ベンダーがこのデータを自動的に読み取る機能を提供していないとしても、この記事を読めば、自力でこの機能を提供する方法がわかるはずです。

次のサンプルでは、インターセプターをステートレス・セッション Bean、`org.aa4h.ejb.HibernateService` のコンテキストに注入します。EJB 3 インターセプターの要件と依存関係は次のように至って単純です。

- インターフェース: `javax.interceptor.InvocationContext`
- アノテーション: `javax.interceptor.AroundInvoke`
- ターゲット・メソッド: `public Object anyName(InvocationContext ic)` というシグニチャーを持つ任意の名前のメソッド

リスト 2 に、サンプル EJB のインターセプト・メソッドを記載します。

リスト 2. EJB 3 インターセプター・メソッド

```
@AroundInvoke
public Object trace(InvocationContext ctx) throws Exception {
   Object returnValue = null;
   int concur = concurrency.incrementAndGet();
   tracer.trace(concur, "EJB Interceptors", ctx.getTarget().getClass()
               .getName(), ctx.getMethod().getName(),
               "Concurrent Invocations");
   try {
      tracer.startThreadInfoCapture(CPU + BLOCK + WAIT);
      // ==================================
      //   This is the target.
      // ==================================
      returnValue = ctx.proceed();
      // ==================================
      tracer.traceIncident("EJB Interceptors", ctx.getTarget().getClass()
            .getName(), ctx.getMethod().getName(), "Responses");
      concur = concurrency.decrementAndGet();
      tracer.trace(concur, "EJB Interceptors", ctx.getTarget().getClass()
            .getName(), ctx.getMethod().getName(),
            "Concurrent Invocations");
      return returnValue;
   } catch (Exception e) {
      tracer.traceIncident("EJB Interceptors", ctx.getTarget().getClass()
            .getName(), ctx.getMethod().getName(), "Exceptions");
```

```
      throw e;
   } finally {
      tracer.endThreadInfoCapture("EJB Interceptors", ctx.getTarget()
            .getClass().getName(), ctx.getMethod().getName());
      tracer.traceIncident("EJB Interceptors", ctx.getTarget().getClass()
            .getName(), ctx.getMethod().getName(), "Invocations");
   }
}
```

リスト 1 と同じく、上記のリスト 2 には大量のインスツルメンテーション・セットが含まれています。通常は推奨されませんが、ここでは一例として示しています。リスト 2 で注目すべき点は以下のとおりです。

- `@AroundInvoke` アノテーションで EJB 呼び出しをラップすることで、メソッドをインターセプターとしてマークしています。
- メソッド呼び出しは、その呼び出しをスタックに積んでから、最終ターゲットもしくは次のインターセプターに渡します。それによって、このメソッドの前に測定値の基準が取得され、メソッドの後にトレースされます。
- トレース・メソッドに渡された `InvocationContext` が、呼び出しに関するすべてのメタデータをインターセプターに提供します。このメタデータには、以下のものがあります。
  - ターゲット・オブジェクト
  - ターゲット・メソッドの名前
  - 渡されるパラメーター

  メタデータの提供が重要な理由は、このインターセプターはさまざまな EJB に適用される可能性があることから、インターセプトされている呼び出しのタイプを想定できないためです。したがって、インターセプター内からメタデータのソースにアクセス可能であることが重要となります。メタデータがなければ、インターセプトされている呼び出しに関する情報をほとんど得ることができません。そうなると、メトリックには多くの興味深い傾向が現れている可能性があるにも関わらず、どのオペレーションに関するものなのかが極めて曖昧になってしまいます。

インスツルメンテーションの観点からすると、これらのインターセプターの最も便利な点は、デプロイメント記述子を変更すれば EJB に適用できるということです。リスト 3 に、このサンプル EJB の場合の ejb-jar.xml デプロイメント記述子を記載します。

リスト 3. EJB 3 インターセプターのデプロイメント記述子

```
<ejb-jar xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
   http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd" version="3.0">
   <interceptors>
      <interceptor>
         <interceptor-class>
    org.runtimemonitoring.interceptors.ejb.EJBTracingInterceptor
    </interceptor-class>
            <around-invoke>
               <method-name>trace</method-name>
            </around-invoke>
      </interceptor>
   </interceptors>
   <assembly-descriptor>
      <interceptor-binding>
         <ejb-name>AA4H-HibernateService</ejb-name>
         <interceptor-class>
```

```
    org.runtimemonitoring.interceptors.ejb.EJBTracingInterceptor
    </interceptor-class>
      </interceptor-binding>
   </assembly-descriptor>
</ejb-jar>
```

前述したとおり、インスツルメンテーション・インターセプターはコンテキストまたは「範囲」/「しきい値」を利用したトレースに便利です。この便利さは、`InvocationContext` に含まれる EJB 呼び出しパラメーター値を利用することでさらに強化されます。これらの値は、範囲または追加コンテキストを示すためにトレース複合名に含めることができるからです。`org.myco.regional.RemoteManagement` クラスで EJB を呼び出す場合を考えてみてください。このクラスには `issueRemoteOperation(String region, Command command)` メソッドがあります。EJB がコマンド (command) を受け入れると、リージョン (region) によって識別されたサーバーに対してリモート呼び出しを行います。このシナリオでは、リージョン内のサーバーが広い範囲の地理的エリアに分散しており、それぞれのサーバーには固有の WAN 特性があります。このパターンは、第 1 回で説明した給与計算処理の例と似ています。つまり、コマンドがどのリージョンに送られたのかを考慮に入れなければ、この EJB への呼び出し経過時間を特徴付けるのは困難です。そのリージョンが別の大陸にあるとしたら、隣接するリージョンの場合よりも経過時間が大幅に長くなると予測できます。ただし、ここでは `InvocationContext` パラメーターからリージョンを判断することが可能です。そのためリスト 4 に示すように、単にリージョン・コードをトレース複合名に追加し、リージョンごとにパフォーマンス・データを区分するだけで済みます。

リスト 4. EJB 3 インターセプターが実装するコンテキスト依存トレース

```
String[] prefix = null;
if(ctx.getTarget().getClass().getName()
   .equals("org.myco.regional.RemoteManagement") &&
    ctx.getMethod().getName().equals("issueRemoteOperation")) {
   prefix = new String[]{"RemoteManagement",
      ctx.getParameters()[0].toString(),
      "issueRemoteOperation"};
}
// Now add prefix to the tracing compound name
```

サーブレッド・フィルター・インターセプター

Java Servlet API が提供するフィルターという構成体は、注入にソース・コードが必要ないこと、そしてメタデータを使用できることを含め、EJB 3 インターセプターと非常に似ています。リスト 5 に、簡略化したインスツルメンテーションを使用したフィルターの `doFilter` メソッドを記載します。メトリックの複合名は、フィルター・クラス名とリクエストの URI (Uniform Resource Identifier) から組み立てられます。

リスト 5. サーブレット・フィルターのインターセプター・メソッド

```
public void doFilter(ServletRequest req, ServletResponse resp,
      FilterChain filterChain) throws IOException, ServletException {
   String uri = null;
   try {
      uri = ((HttpServletRequest)req).getRequestURI();
      tracer.startThreadInfoCapture(CPU + BLOCK + WAIT);
      // ==================================
      //   This is the target.
      // ==================================
      filterChain.doFilter(req, resp);
      // ==================================
   } catch (Exception e) {
   } finally {
      tracer.endThreadInfoCapture("Servlets", getClass().getName(), uri);
   }
}
```

リスト 6 に、リスト 5 に記載したフィルターの web.xml デプロイメント記述子に該当するフラグメントを抜粋します。

リスト 6. サーブレット・フィルターのデプロイメント記述子

```
<web-app >
     <filter>
      <filter-name>ITraceFilter</filter-name>
      <display-name>ITraceFilter</display-name>
      <filter-class>org.myco.http.ITraceFilter</filter-class>
   </filter>
   <filter-mapping>
   <filter-name>ITraceFilter</filter-name>
      <url-pattern>/*</url-pattern>
   </filter-mapping>
</web-app>
```

EJB クライアント・サイドのインターセプターおよびコンテキストの受け渡し

今までの例ではサーバー・サイドのコンポーネントにフォーカスしていましたが、クライアント・サイドにインターセプトなどのインスツルメンテーションを実装するためのオプションもあります。Ajax クライアントではパフォーマンス・リスナーを登録して、`XMLHttpRequest` の経過時間を測定し、次のリクエストのパラメーター・リストの終わりに、リクエストを送信した URI (複合名の場合) と経過時間を追加することができます。さらに JBoss などの一部の Java EE サーバーでは、クライアント・サイドのインターセプターが基本的に EJB 3 インターセプターと同じことを実行できるようにするだけでなく、測定結果を次に送信するリクエストに追加することも可能にします。

監視を行う際の図式では、クライアント・サイドが無視されがちです。今度アプリケーションが遅いというユーザーのクレームを耳にしても、サーバー・サイドの監視手段がサーバー・サイドに問題がないことを確約しているからといって、すぐにそのクレームを却下しないでください。クライアント・サイドのインスツルメンテーションを行うことによって、ユーザーが実際に経験している内容を確実に測定することができます。この測定内容は、サーバー・サイドのメトリックと常に一致するわけではありません。

一部の Java EE 実装がサポートするクライアント・サイドのインターセプターはインスタンス化され、EJB のクライアント・サイドにバインドされます。つまり、リモート・クライアントが RMI

(Remote Method Invocation) プロトコルによってサーバーの EJB を呼び出している場合、このリモート・クライアントからパフォーマンス·データをシームレスに収集することも可能です。リモート呼び出し関係の両側にインターセプター・クラスを実装すれば、この 2 つの間でコンテキストを受け渡して追加のパフォーマンス・データを得ることができるようになります。

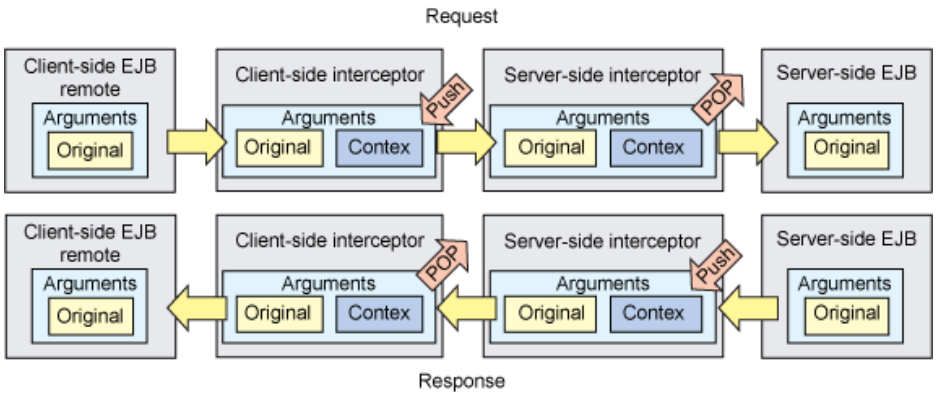
次の例では、データを共有して転送時間 (リクエストとレスポンスを配信するまでの経過時間) を算出するインターセプターのペア、そしてクライアントから見た、サーバーに対するリモート・リクエストの経過レスポンス時間を説明します。この例では、クライアント・サイドおよびサーバー・サイドに、JBoss アプリケーション・サーバー固有の EJB 3 インターセプター実装を使用します。

インターセプターのペアは、コンテキスト依存データを同じ EJB 呼び出しのなかで渡すために、同じペイロード内にコンテキスト依存データを結合させます。コンテキスト依存データは、以下の内容で構成されます。

- **クライアント・サイドでのリクエスト実行時刻**: EJB クライアント・サイドのインターセプターがリクエストを実行した時点のタイムスタンプ
- **サーバー・サイドでのリクエスト受信時刻**: EJB サーバー・サイドのインターセプターがリクエストを受信した時点のタイムスタンプ
- **サーバー・サイドでのレスポンス送信時刻**: EJB サーバー・サイドのインターセプターがクライアントにレスポンスを送り返した時点のタイムスタンプ

呼び出しの引数はスタック構造のように扱われるため、コンテキスト依存データは引数にプッシュされ (渡され)、また引数からポップされ (取り出され) ます。クライアント・サイドのインターセプターが呼び出しにコンテキスト依存データをプッシュすると、サーバー・サイドのインターセプターがこれをポップして EJB サーバー・スタブに渡します。データが返されるときは、この順序が逆になります。図 3 に、このフローを示します。

図 3. クライアントおよびサーバー EJB インターセプターのデータ・フロー



この例でインターセプターを作成するには、クライアントとサーバーの `org.jboss.aop.advice.Interceptor` インターフェースを実装する必要があります。このインターフェースには、1 つの重要なメソッドがあります。

```
public abstract java.lang.Object invoke(
   org.jboss.aop.joinpoint.Invocation invocation) throws java.lang.Throwable
```

上記のメソッドは呼び出しをカプセル化するという考えを導入し、以下を表す別個のオブジェクトにメソッドの実行をカプセル化します。

- ターゲット・クラス
- 呼び出すメソッドの名前
- ターゲット・メソッドに引数として渡されたパラメーターからなるペイロード

このオブジェクトを受け渡しできるのは、このオブジェクトが呼び出し側に渡され、この呼び出し側によって呼び出しオブジェクトがアンマーシャルされ、エンドポイント・ターゲット・オブジェクトに対して動的に実行されるまでの間です。

呼び出しのコンテキストには、クライアント・サイドのインターセプターによって現在のリクエスト時刻が追加され、サーバー・サイドのインターセプターによってリクエスト受信タイムスタンプとレスポンス送信タイムスタンプが追加されます。オプションで、サーバーがクライアントにリクエストを送信させ、クライアントがリクエストの合計経過時間、そしてサーバーへの転送時間とサーバーからクライアントへの転送時間を計算することも可能です。それぞれの計算は以下のように行われます。

- **クライアント・サイドからの転送時間は**、`ServerSideReceivedTime` から `ClientSideRequestTime` を引いた時間です。
- **クライアント・サイドへの転送時間**は、`ClientSideReceivedTime` から `ServerSideRespondTime` を引いた時間です。
- **サーバー・サイドへの転送時間**は、`ServerSideReceivedTime` から `ClientSideRequestTime` を引いた時間です。

リスト 7 に、クライアント･サイド・インターセプターの `invoke` メソッドを記載します。

リスト 7. クライアント･サイド・インターセプターの `invoke` メソッド

```
/**
 * The interception invocation point.
 * @param invocation The encapsulated invocation.
 * @return The return value of the invocation.
 * @throws Throwable
 * @see org.jboss.aop.advice.Interceptor#invoke(org.jboss.aop.joinpoint.Invocation)
 */
public Object invoke(Invocation invocation) throws Throwable {
   if(invocation instanceof MethodInvocation) {
      getInvocationContext().put(CLIENT_REQUEST_TIME, System.currentTimeMillis());
      Object returnValue = clientInvoke((MethodInvocation)invocation);
      long clientResponseTime = System.currentTimeMillis();
      Map<String, Serializable> context = getInvocationContext();
      long clientRequestTime = (Long)context.get(CLIENT_REQUEST_TIME);
      long serverReceiveTime = (Long)context.get(SERVER_RECEIVED_TIME);
      long serverResponseTime = (Long)context.get(SERVER_RESPOND_TIME);
      long transportUp = serverReceiveTime-clientRequestTime;
      long transportDown = serverResponseTime-clientResponseTime;
      long totalElapsed = clientResponseTime-clientRequestTime;
      String methodName = ((MethodInvocation)invocation).getActualMethod().getName();
      String className = ((MethodInvocation)invocation).getActualMethod()
         .getDeclaringClass().getSimpleName();
      ITracer tracer = TracerFactory.getInstance();
      tracer.trace(transportUp, "EJB Client", className, methodName,
         "Transport Up", transportUp);
      tracer.trace(transportDown, "EJB Client", className, methodName,
         "Transport Down", transportDown);
```

```
      tracer.trace(totalElapsed, "EJB Client", className, methodName,
         "Total Elapsed", totalElapsed);
      return returnValue;
   } else {
      return invocation.invokeNext();
   }
}
```

JBoss の EJB 3 インターセプター

任意のペイロードを渡す機能は、JBoss の EJB 2 インターセプター・アーキテクチャーに組み込みました。この機能は EJB 3 で提供するつもりでしたが、それでは上手くいかないため、コンテキスト依存ペイロードをリクエストの追加呼び出しパラメーターとして渡すインターセプターを実装したわけです。また、レスポンス・オブジェクトは `Object[2]` 配列にマーシャルしています。この配列では最初の項目が「実際」の結果で、2 番目の項目がコンテキストです。いずれにしても、マーシャルされたオブジェクトは、もう一方のインターセプターによってアンマーシャルされるので、リクエスターとサービス・エンドポイントの両方が、それぞれに求めているタイプの情報を受け取ることになります。

サーバー・サイドのインターセプターも概念は同様ですが、異なる点は、この例では複雑さが増すことのないようにローカル・スレッドを使用して再入可能性を検出しているところです。再入可能性とは、同じリモート呼び出しで、同じリクエスト処理スレッドが同じ EJB (したがって、インターセプターも) を複数回呼び出す場合を指します。この場合、インターセプターは最初のリクエストを除くすべてのリクエストのトレースおよびコンテキスト処理を無視します。リスト 8 に、サーバー·サイドのインターセプターの `invoke` メソッドを記載します。

リスト 8. サーバー·サイド・インターセプターの `invoke` メソッド

```
/**
 * The interception invocation point.
 * @param invocation The encapsulated invocation.
 * @return The return value of the invocation.
 * @throws Throwable
 * @see org.jboss.aop.advice.Interceptor#invoke(org.jboss.aop.joinpoint.Invocation)
 */
public Object invoke(Invocation invocation) throws Throwable {
   Boolean reentrant = reentrancy.get();
   if((reentrant==null || reentrant==false)
      && invocation instanceof MethodInvocation) {
      try {
         long currentTime = System.currentTimeMillis();
         MethodInvocation mi = (MethodInvocation)invocation;
         reentrancy.set(true);
         Map<String, Serializable> context = getInvocationContext(mi);
         context.put(SERVER_RECEIVED_TIME, currentTime);
         Object returnValue = serverInvoke((MethodInvocation)mi);
         context.put(SERVER_RESPOND_TIME, System.currentTimeMillis());
         return addContextReturnValue(returnValue);
      } finally {
         reentrancy.set(false);
      }
   } else {
      return invocation.invokeNext();
   }
}
```

JBoss はアスペクト指向プログラミング (AOP) の手法 (「参考文献」を参照) に従い、ejb3-interceptors-aop.xml というディレクティブ·ファイルを読み取り、そこに定義された命令に基づいてインターセプターを適用します。JBoss は実行時にコア Java EE ルールを EJB 3 クラスに適用するのにも、この AOP 手法を使用します。したがってこのディレクティブ・ファイルには、パフォー

マンス監視用インターセプターだけでなく、トランザクション管理、セキュリティー、パーシスタンスなどの側面に関するディレクティブも含まれます。クライアント・サイドのディレクティブは至って単純で、一連のインターセプター・クラス名が含まれる `stack` という XML 要素の `name` 属性にスタック名を指定することで定義されます。ここに定義される各クラス名は、`PER_VM` または `PER_INSTANCE` インターセプターとしての修飾子が付けられ、EJB インスタンスに 1 つのインターセプター・インスタンスを共有させるか、あるいはインターセプター・インスタンスを共有させることなく、それぞれの EJB インスタンスに独自のインスタンスを持たせるのかを指定します。パフォーマンス監視用インターセプターの場合、この構成はインターセプター・コードがスレッド・セーフであるかどうかによって決まります。コードが安全に複数のスレッドを同時に処理できるのであれば、`PER_VM` ストラテジーを使用したほうが効率的です。一方、効率性には劣るけれどもスレッド･セーフなストラテジーとしては `PER_INSTANCE` を使用します。

サーバー・サイドのインターセプターの構成はクライアント・サイドよりも多少複雑になります。これは、XML に定義された一連の構文パターンとフィルターに従ってインターセプターが適用されるためです。対象とする特定の EJB メソッドが、定義されたパターンと一致する場合、そのパターンに対して定義されたインターセプターが適用されます。この定義をさらに絞り込めば、デプロイされた EJB の特定のサブセットをインターセプターのターゲットにすることも可能です。クライアント・インターセプターの場合、新しいカスタム・スタックを実装するにはターゲットとする Bean に固有の新しいスタックを `stack` 要素の `name` 属性に名前を指定することで作成しますが、サーバー・サイドでは、カスタム・スタックを新しいドメインとして `domain` 要素に定義することができます。個々の EJB に関連付けるクライアントの `stack` 要素の `name` 属性とサーバー・スタックの `domain` は、EJB のアノテーションに指定することができます。あるいはソース・コードを変更できない場合、または変更したくない場合には、EJB のデプロイメント記述子に、これに相当する内容を指定するか、上書きします。リスト 9 に、この例で使用する ejb3-interceptors-aop.xml ファイルを簡略化して記載します。

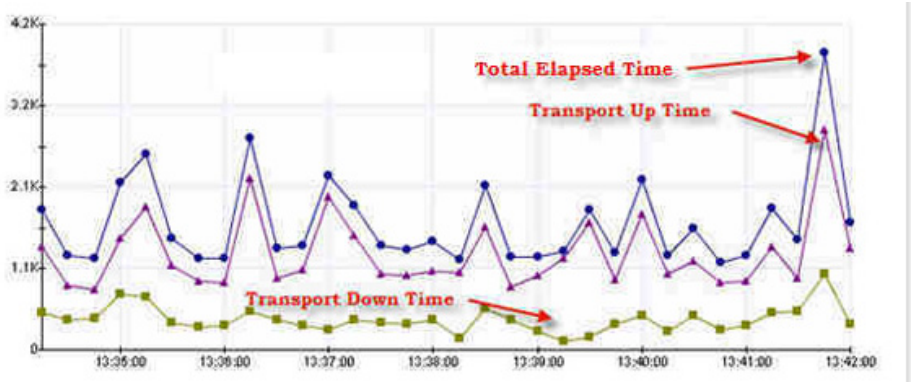
リスト 9. 簡略化した EJB 3 AOP 構成

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE aop PUBLIC
   "-//JBoss//DTD JBOSS AOP 1.0//EN"
   "http://labs.jboss.com/portal/jbossaop/dtd/jboss-aop_1_0.dtd">

<aop>
   .
   .
   <interceptor
      class="org.runtimemonitoring.ejb.interceptors.ClientContextualInterceptor"
      scope="PER_VM"/>
   .
   .
   <stack name="StatelessSessionClientInterceptors">
      <interceptor-ref
         name="org.runtimemonitoring.ejb.interceptors.ClientContextualInterceptor"/>
      .
      .
   </stack>
   .
   .
  <interceptor
     class="org.runtimemonitoring.ejb.interceptors.ServerContextualInterceptor"
     scope="PER_VM"/>
    .
    .
   <domain name="Stateless Bean">
```

```
      <bind pointcut="execution(public * *->*(..))">
         <interceptor-ref name="org.aa4h.ejb.interceptors.ServerContextualInterceptor"/>
         .
         .
        </bind>
   </domain>
</aop>
```

このようなパフォーマンス・データ収集には一石二鳥の利点があります。まず、クライアントの観点から EJB のパフォーマンスについての現状がわかるという利点、そしてパフォーマンスが劣化している場合には、転送時間を手掛かりに、その原因がクライアントとサーバー間のネットワーク・リンクの遅い処理速度であるかどうかを判断できるという利点です。図 4 に、クライアントから測定した合計経過時間と双方向での転送パフォーマンスのメトリックを表示します。転送時間を際立たせるために、クライアントとサーバー間ではわざと遅いネットワーク・リンクを使用しています。

図 4. クライアント・インターセプターのコンテキスト依存パフォーマンス・メトリック



クライアント・サイドのインターセプターを使用するときには、クライアント・インターセプター・クラス自体がクライアント・アプリケーションのクラスパスに含まれていなければなりません。含まれていない場合には、サーバーからのリモート・クラス・ロードを有効にして、起動時にクライアント・インターセプターとその依存関係がクライアントにダウンロードされるようにしてください。クライアントのシステム・クロックがサーバーのシステム・クロックとほぼ正確に同期されていないと、この 2 つのクロックとの差に正比例したエラー結果が出ることになります。

Spring でのインターセプター

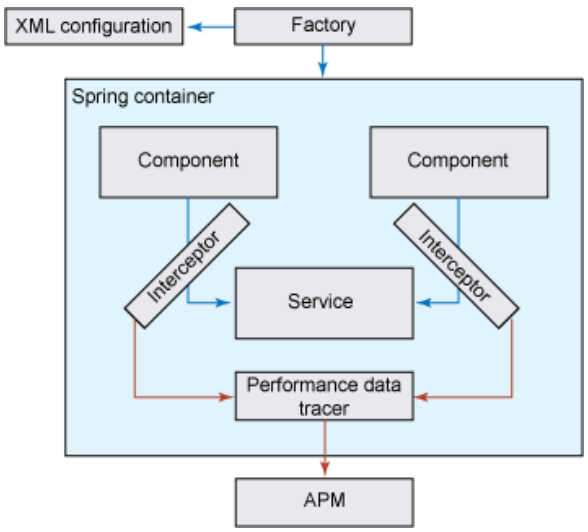
Java EE には、シームレスに他のシステムへのインターセプトを実現する方法が豊富にありますが、Java EE 以外のよく使われているコンテナーの多くでも、暗黙的および明示的インターセプトをサポートしています。ここで使用している「コンテナー」という用語は、疎結合を使用、あるいは促進する類のフレームワークを意味します。密結合がないということは、すなわちインターセプトを実装できるということです。一般に、このようなタイプのフレームワークは依存性の注入、あるいは IoC (Inversion of Control: 制御の反転) アーキテクチャーと呼ばれます。このようなフレームワークでは、コンポーネントをハード・コーディングしてコンポーネント同士を直接対話させるのではなく、個々のコンポーネントを 1 つに「つなぎ合わせる」方法を外部で定義することができます。インターセプトについては、IoC フレームワークとして定評のある Spring

Framework (「参考文献」を参照) でトレーサー・インターセプターを使用してパフォーマンス・データを収集する方法を説明して締めくくることにします。

Spring Framework では、POJO (Plain Old Java Object) を使ってアプリケーションを構築することができます。POJO にはビジネス・ロジックしか含まれないため、エンタープライズ・アプリケーションを構築するために必要なものは、このフレームワークが追加します。Spring の階層化アーキテクチャーは、Java アプリケーションを最初に作成する段階でインスツルメンテーションを考慮していなかった場合に役立ちます。アプリケーション・アーキテクチャーを Spring に適応させるのは必ずしも簡単なことでありませんが、Spring の POJO 管理特性、そして一連の Java EE と AOP の統合が、普通にハード・コーディングされた Java クラスでも Spring のコンテナー管理に委託することを十分可能にしています。そしてこの過程で、インターセプトによってパフォーマンス・インスツルメンテーションを追加することができます。ターゲット・クラスのソース・コードを変更する必要はまったくありません。

Spring は IoC (Inversion of Control: 制御の反転) コンテナーと称されることがよくあります。これは、Spring が Java アプリケーションの従来の制御形態とは逆の形態をとるためです。従来の制御形態では、1 つの特定の制御プログラムまたは制御スレッドが、それに必要なすべてのコンポーネントと依存関係をプロシージャーによってロードします。一方 IoC では、コンテナーはいくつかのコンポーネントをロードするだけで、後は外部構成に従ってコンポーネント間の依存関係を管理します。このような依存関係の管理は、コンテナーが依存関係 (JDBC `DataSource` など) をコンポーネントに注入することから、依存性の注入と呼ばれます。この場合、コンポーネント自体がそれぞれの依存関係を探し出す必要はありません。インスツルメンテーションを目的としてコンテナーの構成を変更し、コンポーネント間の「結合組織」にインターセプターを挿入するのは簡単です。図 5 に、この概念を示します。

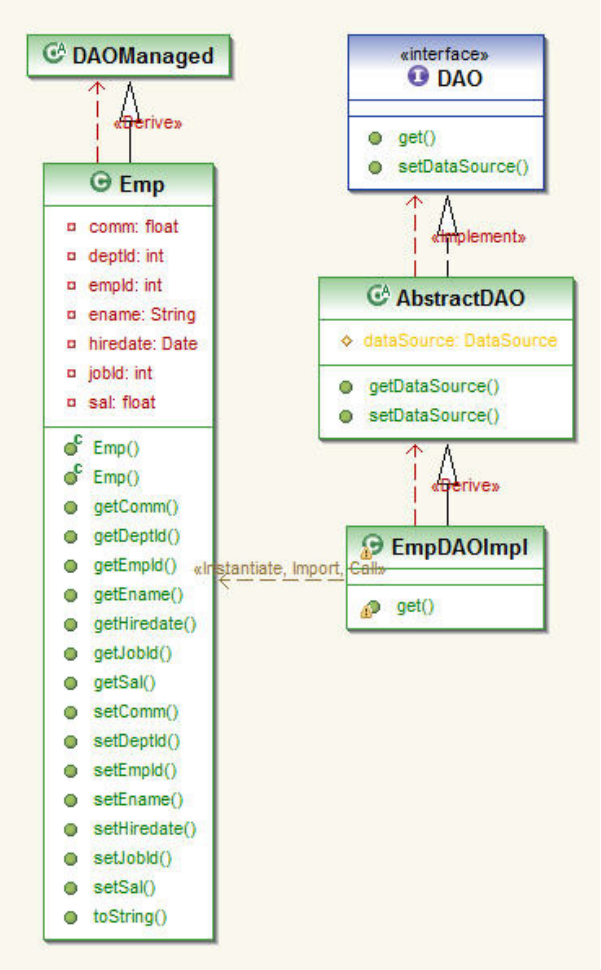
図 5. Spring とインターセプトの概要



ここで、Spring での単純なインターセプトの例を紹介します。この例に必要なのは、基本的なデータ・アクセス・オブジェクト (DAO) パターン・クラスの 1 つ、`EmpDAOImpl` クラスです。このクラスは、`public Map<Integer, ? extends DAOManaged> get(Integer...pks)` というメソッドを定義する DAO インターフェースを実装します。このインターフェースに、目的のオブジェクトすべてに対応する主キーの配列を渡すと、DAO 実装がオブジェクトの `Map` を返すことになりま

す。このコードにはあまりにも足りないものが多いので、ここで詳しく説明することはできませんが、あえて言うなら、このコードにはインスツルメンテーションに対応する部分はなく、オブジェクト・リレーショナル・マッピング (ORM) フレームワークの類も使っていないということです。図 6 は、クラス構造の概略です。ここに記載する成果物の完全なソース・コードとテキスト・ファイルを入手するには、「ダウンロード」セクションを参照してください。

図 6. EmpDAO クラス



`EmpDAOImpl` は、spring.xml ファイルでの構成に従って Spring コンテナーにデプロイされます。リスト 10 に、この部分を簡略化して記載します。

リスト 10. Spring の例での基本コンテナー構成

```
<beans>
   <bean id="tracingInterceptor"
      class="org.runtimemonitoring.spring.interceptors.SpringTracingInterceptor">
      <property name="interceptorName" value="Intercepted DAO"/>
   </bean>

   <bean id="tracingOptimizedInterceptor"
      class="org.runtimemonitoring.spring.interceptors.SpringTracingInterceptor">
      <property name="interceptorName" value="Optimized Intercepted DAO"/>
   </bean>

   <bean id="DataSource"
```

```
      class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close"
      p:url="jdbc:postgresql://DBSERVER:5432/runtime"
      p:driverClassName="org.postgresql.Driver"
      p:username="scott"
      p:password="tiger"
      p:initial-size="2"
      p:max-active="5"
      p:pool-prepared-statements="true"
      p:validation-query="SELECT CURRENT_TIMESTAMP"
      p:test-on-borrow="false"
      p:test-while-idle="false"/>

   <bean id="EmployeeDAO" class="org.runtimemonitoring.spring.EmpDAOImpl"
      p:dataSource-ref="DataSource"/>

   <bean id="empDao" class="org.springframework.aop.framework.ProxyFactoryBean">
      <property name="proxyInterfaces" value="org.runtimemonitoring.spring.DAO"/>
      <property name="target" ref="EmployeeDAO"/>
      <property name="interceptorNames">
         <list>
            <idref local="tracingInterceptor"/>
         </list>
      </property>

   </bean>
   <bean id="empDaoOptimized"
      class="org.springframework.aop.framework.ProxyFactoryBean">
      <property name="target" ref="EmployeeDAO"/>
      <property name="optimize">
         <value>true</value>
      </property>
      <property name="proxyTargetClass">
         <value>true</value>
      </property>
      <property name="interceptorNames">
         <list>
            <idref local="tracingOptimizedInterceptor"/>
         </list>
      </property>
   </bean>

</beans>
```

他にも少数のオブジェクトがデプロイされます。これらのコンポーネントは、それぞれの Spring `Bean id` を参照することによって記述されます。リスト 10 でこれらの Bean 要素に該当するものは以下のとおりです。

- `tracingInterceptor` および `tracingOptimizedInterceptor`: `SpringTracingInterceptor` クラスの 2 つのインターセプター。このクラスには、収集されたデータを APM システムにトレースするための ITracer 呼び出しが含まれています。
- `DataSource`: JDBC 接続を `runtime` というサンプル・データベースにプールする JDBC の `DataSource`。 これは `EmpDAOImpl` に注入されます。
- `EmployeeDAO`: サンプルの一部として呼び出す対象の `EmpDAOImpl`。
- `empDao` および `empDaoOptimized`: spring.xml ファイルに定義された最後の 2 つの Bean は、Spring の `ProxyFactoryBean` です。この 2 つは基本的には `EmpDAOImpl` のプロキシーで、それぞれがインターセプターを参照します。`EmpDAOImpl` には直接アクセスすることも可能ですが、プロキシーを使用することでインターセプターが呼び出され、パフォーマンス・メトリックが生成されます。リスト 10 に記載されている 2 つのプロキシーとそれぞれのインター

セプターを見ると、いくつかの相違点と構成の考慮事項がわかるはずです。囲み記事「最適化インターセプター」を参照してください。

最適化インターセプター

リスト 10 の標準インターセプターと最適化インターセプターとの違いは、最適化プロキシーに追加されている `<property name="optimize"><value>true</value></property>` プロパティーにあります。このプロパティーがない場合、プロキシーは `java.lang.reflect.Proxy` によるリフレクションを使用してインターセプターを呼び出しますが、このオプションが設定されていると、Spring は CGLIB というバイトコード・インスツルメンテーション・ライブラリーを使って、直接 (リフレクションを使用せずに) 呼び出した側に対して動的にバイトコードを作成します。このような場合、バイトコード最適化ソリューションのほうが動的プロキシーよりパフォーマンスに優れるのが通常ですが、その効果はさまざまです。その理由の一部として、最近の JVM では Java リフレクションのパフォーマンスが大幅に改善されていることが挙げられます。

Spring コンテナーは `SpringRunner` クラスからブートストラップされます。また、以下の 4 つのターゲットに対して `DAO.get` を呼び出すテスト・ループも起動します。

- `EmployeeDAO` Spring Bean。Spring によってインスツルメンテーションが追加されていない管理対象 DAO を表します。
- `empDao` Spring Bean。Spring が標準インターセプターを使用してインスツルメンテーションを追加する管理対象 DAO を表します。
- `empDaoOptimized` Spring Bean。Spring が最適化インターセプターを使用してインスツルメンテーションを追加する管理対象 DAO を表します。
- Spring 非管理対象 `EmpDAOImpl`。Spring 管理対象 Bean との比較用です。

Spring はこれらのタイプのインターセプターを `org.aopalliance.intercept.MethodInterceptor` というインターフェースによって実装します。実装するメソッドは唯一、`public Object invoke(MethodInvocation invocation) throws Throwable` だけです。`MethodInvocation` オブジェクトは 2 つの重要な項目を提供します。1 つはコンテキスト (つまり、インターセプトされているメソッドの名前) を設定したトレーサー、そしてもう 1 つは呼び出しを目的のターゲットに誘導する `proceed` メソッドです。

リスト 11 に、`SpringTracingInterceptor` クラスの `invoke` メソッドを記載します。この場合、`interceptorName` プロパティーは必要ありませんが、ここでこのプロパティーを追加している理由は、このサンプルのコンテキストを追加するためです。完全な多目的インターセプター実装では通常、トレーサーがトレース・コンテキストにクラス名を追加するので、インターセプトされるクラスに含まれるすべてのメソッドが個別の APM 名前空間にトレースされることになります。

リスト 11. `SpringTracingInterceptor` クラスの `invoke` メソッド

```
public Object invoke(MethodInvocation invocation) throws Throwable {
   String methodName = invocation.getMethod().getName();
   tracer.startThreadInfoCapture(WAIT+BLOCK);
   Object returnValue = invocation.proceed();
   tracer.endThreadInfoCapture("Spring", "DAO",
      interceptorName, methodName);
   tracer.traceIncident("Spring", "DAO", interceptorName,
      methodName, "Responses Per Interval");
   return returnValue;
}
```

このサンプルでは、`SpringRunner` クラスがメインのエントリー・ポイントです。このクラスは Spring Bean ファクトリーを初期化してから長いループを開始し、Bean のそれぞれに負荷をかけます。このループのコードはリスト 12 のとおりです。`daoNoInterceptor` と `daoDirect` は Spring インターセプターによるインスツルメンテーションの追加が行われないため、この `SpringRunner` ループには手動でインスツルメンテーションを追加しました。

リスト 12. 簡略化した `SpringRunner` ループ

```
Map<Integer, ? extends DAOManaged> emps = null;
DAO daoIntercepted = (DAO) bf.getBean("empDao");
DAO daoOptimizedIntercepted = (DAO) bf.getBean("empDaoOptimized");
DAO daoNoInterceptor = (DAO) bf.getBean("EmployeeDAO");
DataSource dataSource = (DataSource) bf.getBean("DataSource");
DAO daoDirect = new EmpDAOImpl();
// Not Spring Managed, so dependency is set manually
daoDirect.setDataSource(dataSource);
for(int i = 0; i < 100000; i++) {
    emps = daoIntercepted.get(empIds);
    log("(Interceptor) Acquired ", emps.size(), " Employees");
    emps = daoOptimizedIntercepted.get(empIds);
    log("(Optimized Interceptor) Acquired ", emps.size(), "
       Employees");
    tracer.startThreadInfoCapture(WAIT+BLOCK);
    emps = daoNoInterceptor.get(empIds);
    log("(Non Intercepted) Acquired ", emps.size(), " Employees");
    tracer.endThreadInfoCapture("Spring", "DAO",
       "No Interceptor DAO", "get");
    tracer.traceIncident("Spring", "DAO",
       "No Interceptor DAO", "get", "Responses Per Interval");
    tracer.startThreadInfoCapture(WAIT+BLOCK);
    emps = daoDirect.get(empIds);
    log("(Direct) Acquired ", emps.size(), " Employees");
    tracer.endThreadInfoCapture("Spring", "DAO",
       "Direct", "get");
    tracer.traceIncident("Spring", "DAO", "Direct",
       "get", "Responses Per Interval");
}
```

APM システムからレポートされる結果には、いくつかの相対的な項目が示されます。表 1 に、テスト実行結果から抜粋した各 Spring Bean からの呼び出し平均経過時間を記載します。

表 1. Spring インターセプターのテスト実行結果

Spring bean	平均経過時間 (ミリ秒)	最小経過時間 (ミリ秒)	最大経過時間 (ミリ秒)	カウント
直接	145	124	906	5110
最適化インターセプター	145	125	906	5110
インターセプター未使用	145	124	891	5110
インターセプター使用	155	125	952	5110

図 7 に、このテスト・ケースで APM に作成されたメトリック・ツリーを表示します。

図 7. Spring インターセプター・テスト実行の APM メトリック・ツリー

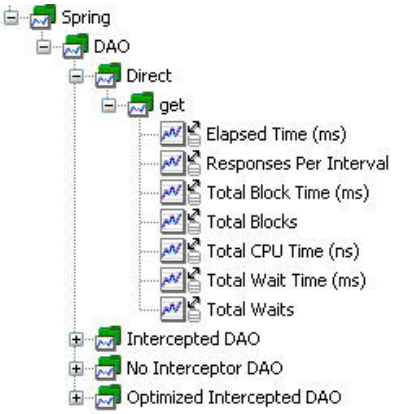
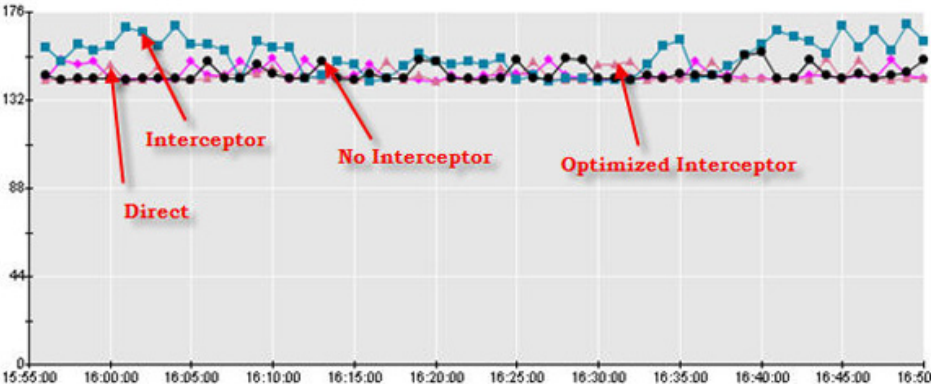


図 8 は、このデータのグラフ表示です。

図 8. Spring インターセプターのテスト実行結果



明らかに、それぞれの結果はかなり寄り集まったものになっていますが、ここには何らかのパターンが現れています。さらに、最適化インターセプターは非最適化インターセプターよりわずかに勝っています。ただし、このテストでは 1 つのスレッドしか実行していないので、比較分析にはそれほど役立ちません。次のセクションでは、このテスト・ケースを拡張して複数のスレッドを実装します。

クラスのラップによる JDBC のインスツルメンテーション

典型的なエンタープライズ Java アプリケーションで長期的にパフォーマンス問題が続くことがありますが、私が気付いたのは、その多くの根本原因はデータベース・インターフェースにあるということです。これはまったくの予想外というわけではありません。ローカルの JVM 内にはないデータ・セットまたはリソースを取得するために JVM が外部サービスを呼び出すときに最も一般的に使われるのは、JDBC によるデータベース呼び出しだからです。論理的に言えば、このシナリオで問題を起こす犯人には、データベース・クライアント、データベース自体、またはその両方が考えられます。一方、多くのデータベース・クライアント指向のアプリケーションでは、以下の例をはじめとする多数のパフォーマンス・アンチパターンが障害になります。

- 論理的には正しい一方、パフォーマンスに劣る SQL
- リクエストが十分に具体化されていないために、目的の機能の実現に必要なデータより遥かに大量のデータが取得される

- 同じデータを何度も繰り返し取得する
- 同じデータ・セットを数少ないリクエストで効率的に取得できるにもかかわらず、リクエストに不備があるため、1 つの論理構成体のデータを取得するために多数のデータベース・リクエストが行われる (私自身のデータベース・アクセスについての公理は、複数のクエリーで狭い範囲のデータ・セットを取得するよりも、1 つのクエリーで多数の行と列を返すようにすることです)。多くの場合、クラス構造がネストされていたり、開発者が共通の統一データ・リクエスターに委任する代わりに、各オブジェクトが固有のデータ取得を管理することを規定する正統なカプセル化の概念を適用しようとしたりすると、このパターンになります。

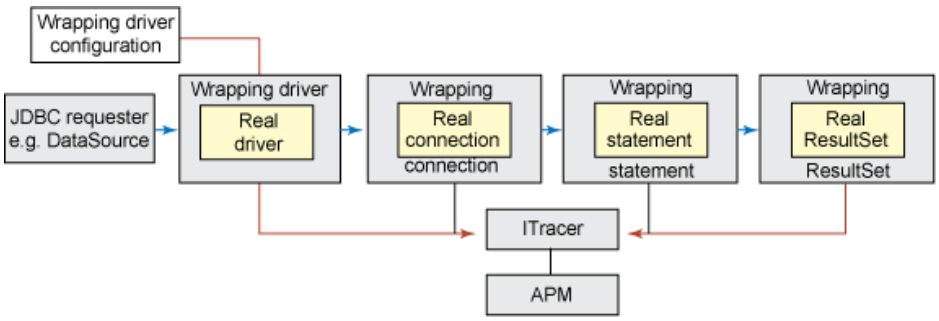
もちろん、いずれの場合にしても、データベース・クライアント指向のアプリケーションを設計することやコーディングすることに、反対しているわけではありません。この連載でも第 3 回では、パフォーマンス統計を目的としたデータベースの監視方法も紹介することにしています。しかし全般的に見た場合、傾向としてはクライアントで解決するのが最も効果的です。したがって、Java アプリケーションでデータベース・インターフェースのパフォーマンスを監視するのに最適なターゲットは JDBC となります。

ここからは、クラスのラップという概念を用いて JDBC クライアントにインスツルメンテーションを追加する方法を説明します。クラスのラップの背後にある考えは、ターゲット・クラスをインスツルメンテーション・コードの層にラップし、このコードによって、ラップされたクラスと同じ振る舞いを外部に対してするというものです。このようなシナリオでは、従属する構造に干渉することなく、ラップしたクラスをシームレスに導入する方法が課題となります。

この例では、根本的に JDBC は完全にインターフェース定義の API であるという事実を利用します。つまり、仕様にはほとんど具象クラスが含まれていないため、JDBC のアーキテクチャーでは、データベース・ベンダーが独自に提供するクラスと直接密結合する必要性がまったくなくなります。JDBC の具体的な実装は暗黙的にロードされ、ソース・コードがこれらの具象クラスを直接参照することはほとんどありません。そのため、まったく新しい JDBC ドライバーを定義し、この新しい JDBC ドライバーに対する呼び出しを、そのベースとなる「実際」のドライバーに委任して、後はプロセスのパフォーマンス・データを収集するだけという動作を実装することができます。

私は `WrappingJDBCDriver` という実装を作成しました。この実装は、パフォーマンス・データの収集方法を説明し、前の Spring の例での `EmployeeDAO` テスト・ケースをサポートするには十分な機能を持ちます。図 9 に、`WrappingJDBCDriver` の動作の流れを大まかに示します。

図 9. `WrappingJDBCDriver` の概要



JDBC ドライバーをロードする際の標準的な処理には、2 つの項目が必要です。1 つはドライバーのクラス名、そしてもう 1 つは接続の対象とするデータベースの JDBC URL です。ドライバー・ローダーはドライバー・クラスを (おそらく `Class.forName(jdbcDriverClassName)` の呼び出しにより) ロードします。大抵の JDBC ドライバーはクラスをロードする時点で JDBC の `java.sql.DriverManager` に自己登録します。すると、ドライバー・ローダーは JDBC の URL を JDBC ドライバーのインスタンスに渡し、ドライバーがこの URL を受け入れるかどうかをテストします。URL が受け入れられれば、ローダーはこのドライバーで `connect` を呼び出し、`java.sql.Connection` を取得します。

ラップされたドライバーのクラス名は `org.runtimemonitoring.jdbc.WrappingJDBCDriver` となります。ドライバーはインスタンス化されると、クラスパスから wrapped-driver.xml という名前の構成ファイルをロードします。このファイルに含まれるのは、ターゲット・ドライバーに関連付けられた表意名ごとにインデックスが付けられた以下のインスツルメンテーション構成項目です。

- <Figurative Name>.driver.prefix: JDBC ドライバーの実際の JDBC URL 接頭辞 (`jdbc.postgresql:` など)。
- <Figurative Name>.driver.class: JDBC ドライバーのクラス名 (`org.postgresql.Driver` など)。
- <Figurative Name>.driver.class.path: カンマで区切られた、JDBC ドライバーの場所へのクラスパスの一連のエントリー。この項目はオプションです。この項目が含まれていない場合、`WrappingJDBCDriver` は独自のクラス・ローダーを使ってドライバー・クラスを見つけます。
- <Figurative Name>.tracer.pattern.<Zero Based Index>: 特定のターゲット・データベースのトレース・カテゴリーを抽出するために使用される一連の正規表現パターン。インデックスは 0 から始まり、インデックスのシーケンスがトレース・カテゴリーの階層を定義します。

`WrappingJDBCDriver` で大前提となるのは、他の JDBC ドライバー (インスツルメンテーションのターゲットとなるドライバーも含まれます) が認識できないほどに「大幅に変更された」JDBC URL を使うように JDBC クライアント・アプリケーションを構成し、`WrappingJDBCDriver` 以外にはこの URL が受け入れられないようにすることです。`WrappingJDBCDriver` は変更後の URL を認識し、内部でターゲット・ドライバーをロードして、この変更された URL に関連付けます。すると変更された URL が「元の状態に復元」され、ターゲット・データベースと実際に接続するために内部ドライバーに委託されます。実際の接続は `WrappingJDBCConnection` 内にラップされて、要求側のアプリケーションに返されます。この変更アルゴリズムは極めて基本的なものにすることができますが、ただし JDBC URL を「実際」のターゲット JDBC ドライバーが認識できない状態にするアルゴリズムであることが条件です。そうでないと、`WrappingJDBCDriver` が実際のドライバーによってバイパスされてしまう可能性があります。この例では、`jdbc:postgresql://DBSERVER:5432/runtime` という実際の JDBC URL を `jdbc:!itracer!wrapped:postgresql://DBSERVER:5432/runtime` に変更しています。

「実際」のドライバーのクラス名とオプションのクラスパス構成項目は、`WrappingJDBCDriver` がドライバー・クラスを見つけてクラスをロードし、クラスをラップして委任できるようにする役割を果たします。トレーサー・パターンの構成項目は一連の正規表現で、`WrappingJDBCDriver` に対し、このターゲット・データベースのトレース名前空間を決定する方法を指定します。これらの表現は「実際」の JDBC URL に適用されることから、トレーサーがターゲット・データベースごとに区分したパフォーマンス・メトリックを APM システムに提供するために必要となりま

す。WrappingJDBCDriver を複数の (おそらく異なる) データベースに使用する場合には、この区分を設定して、収集されたメトリックをターゲット・データベースごとにグループ化できるようにしなければなりません。例えば、jdbc:postgresql://DBSERVER:5432/runtime という JDBC URL は、postgresql, runtime の組み合わせに対する名前空間を生成することになります。

リスト 13 は、PostgreSQL 8.3 JDBC Driver にマッピングされた postgres の表意名を使ってラップした driver.xml ファイルの例です。

リスト 13. ラップされた driver.xm ファイルの例

```
<properties>
   <entry key="postgres.driver.prefix">jdbc:postgresql:</entry>
   <entry key="postgres.driver.class">org.postgresql.Driver</entry>
   <entry key="postgres.driver.class.path">
      C:\Postgres\psqlJDBC\postgresql-8.3-603.jdbc3.jar
   </entry>
   <entry key="postgres.tracer.pattern.0">:([a-zA-Z0-9]+):</entry>
   <entry key="postgres.tracer.pattern.1">.*\/\/.*\/([\S]+)</entry>
</properties>
```

この部分的な実装は、P6Spy というオープンソースの製品から発想を得ています (「参考文献」を参照)。

WrappingJDBCDriver をどのように使用するかを具体的に説明するため、EmpDAO Spring テスト・ケースの拡張バージョンを新たに作成しました。新しい Spring 構成ファイルには spring-jdbc-tracing.xml、新しいエントリー・ポイント・クラスはSpringRunnerJDBC です。このテスト・ケースには、追加の比較テスト・ポイントがいくつか含まれるため、簡潔さを期して一部の命名規則は更新されます。さらに、この拡張テスト・ケースはマルチスレッド化されているので、収集されるメトリックには、さまざまに異なる興味深い振る舞いが出てくることになります。また、ばらつきを与えるため、DAO の引数はランダムにすることができます。

この新しいテスト・ケースでは、以下のようにトレース機能を拡張しました。

- 2 つのデータ・ソースが定義されています。一方はダイレクト JDBC ドライバーを使用し、もう一方はインスツルメンテーションを追加した JDBC ドライバーを使用します。
- オプションで、Spring プロキシーを介してデータ・ソースにアクセスできるようになっています。Spring プロキシーには、接続を取得するまでの経過時間を監視するようにインスツルメンテーションが追加されています。
- DAO インターセプターを拡張し、インターセプターを通過する同時スレッドの数を監視するようにしています。
- 追加のバックグラウンド・スレッドを発生させて、データ・ソースの使用率統計をポーリングします。
- すべての WrappingJDBC クラスは、基底クラスである WrappingJDBCCore を使用してトレーサー呼び出しのほとんどを行います。この基底クラスはその ITracer に対して単純に転送するだけでなく、データベース・インスタンスごとにまとめた形でのトレースも実行します。こうして行われているのは、下位レベルの特定のメトリックを上位レベルの名前空間に複数回トレースすることによってサマリー・レベルのメトリックを提供するという、APM システムに共通の機能です。例えば、あらゆるオブジェクトでの JDBC 呼び出しはすべてデータ

ベース・レベルまでまとめられ、そのデータベースに対するすべての呼び出しの平均経過時間とリクエストの数を要約します。

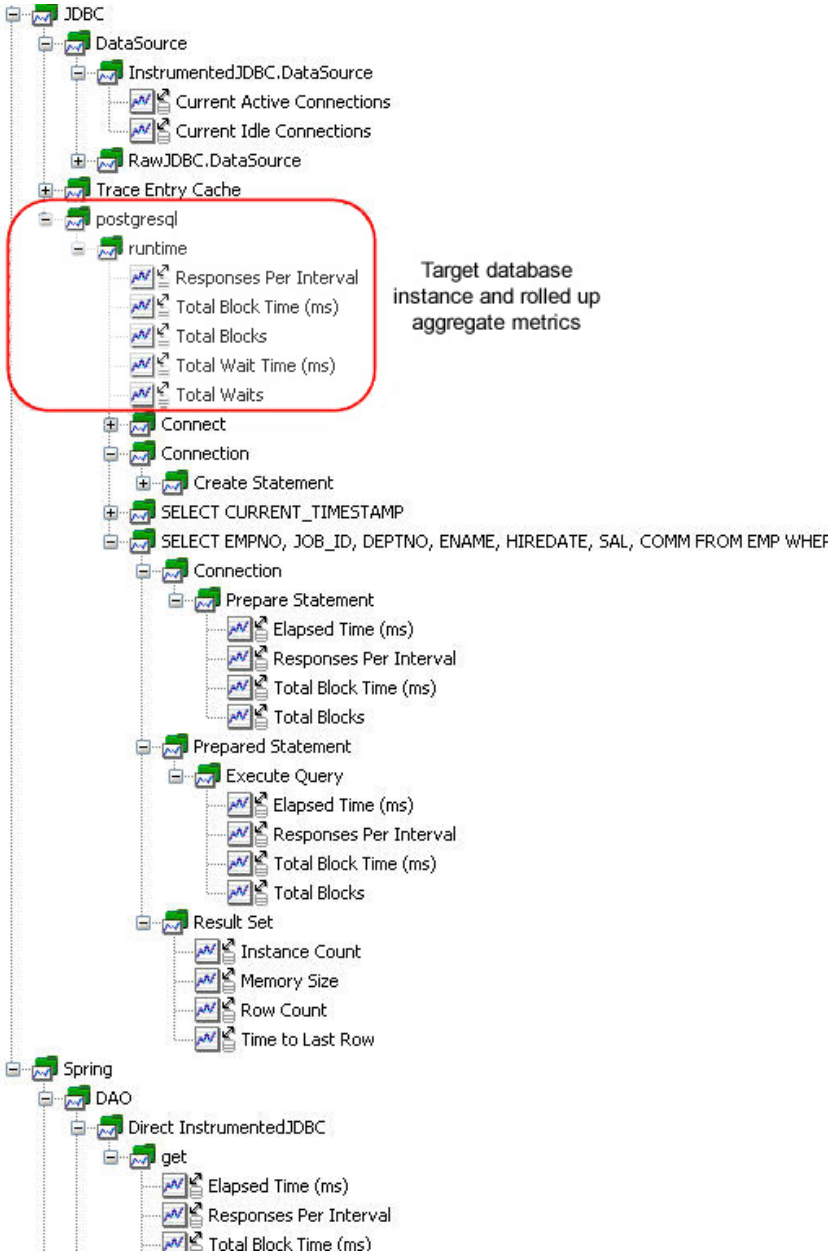
リスト 14 に、spring-jdbc-tracing.xml ファイル内の新しい Bean 定義のインスタンスを示します。`InstrumentedJDBC.DataSource Bean` に定義された JDBC URL は大幅に変更された規約を使用することに注意してください。

リスト 14. spring-jdbc-tracing.xml からの抜粋

```
<!-- A DataSource Interceptor -->
<bean id="InstrumentedJDBCDataSourceInterceptor"
   class="org.runtimemonitoring.spring.interceptors.SpringDataSourceInterceptor">
   <property name="interceptorName" value="InstrumentedJDBC.DataSource"/>
</bean>

<!-- A DataSource for Instrumented JDBC -->
<bean id="InstrumentedJDBC.DataSource"
   class="org.apache.commons.dbcp.BasicDataSource"
   destroy-method="close"
   p:url="jdbc:!itracer!wrapped:postgresql://DBSERVER:5432/runtime"
   p:driverClassName="org.runtimemonitoring.jdbc.WrappingJDBCDriver"
   p:username="scott"
   p:password="tiger"
   p:initial-size="2"
   p:max-active="10"
   p:pool-prepared-statements="true"
   p:validation-query="SELECT CURRENT_TIMESTAMP"
   p:test-on-borrow="false"
   p:test-while-idle="false"/>


<!-- The Spring proxy for the DataSource -->
<bean id="InstrumentedJDBC.DataSource.Proxy"
   class="org.springframework.aop.framework.ProxyFactoryBean">
   <property name="target" ref="InstrumentedJDBC.DataSource"/>
   <property name="optimize"><value>true</value></property>
   <property name="proxyTargetClass"><value>true</value></property>
   <property name="interceptorNames">
      <list>
         <idref local="InstrumentedJDBCDataSourceInterceptor"/>
      </list>
   </property>
</bean>

<!--
The Spring proxy for the DataSource which is injected into
the DAO bean instead of the DataSource bean itself.
-->
<bean id="InstrumentedJDBC.DataSource.Proxy"
   class="org.springframework.aop.framework.ProxyFactoryBean">
   <property name="target" ref="InstrumentedJDBC.DataSource"/>
   <property name="optimize"><value>true</value></property>
   <property name="proxyTargetClass"><value>true</value></property>
   <property name="interceptorNames">
      <list>
         <idref local="InstrumentedJDBCDataSourceInterceptor"/>
      </list>
   </property>
</bean>
```

図 10 に、このテスト・ケースの APM メトリック・ツリーを表示します。

図 10. インスツルメンテーションが追加された JDBC メトリック・ツリー



この例ではデータの量を増やすことで、スレッドの `BLOCK` と `WAIT` の原因を具体的な例で説明することができます。`SpringRunnerJDBC` は各ループの最後にある単純なステートメント、`Thread.currentThread().join(100)` の近くに `ThreadInfoCapture(WAIT+BLOCK)` トレースを追加します。APM システムによると、このトレースを追加することによって、スレッドの平均待機時間は 103 ミリ秒を示すことになっています。したがって、スレッドが何らかの理由で呼び出されるのを待って待機状態になると、そのスレッドによってある程度の待機期間が発生します。それとは対照的に、スレッドが `DataSource` から接続を取得しようとするときには厳密に同期化されたリソースにアクセスするため、接続を取得しようと競合するスレッド数が増え、`DAO.get` メソッドが明らかにスレッド・ブロック回数の増加を示すことになります。

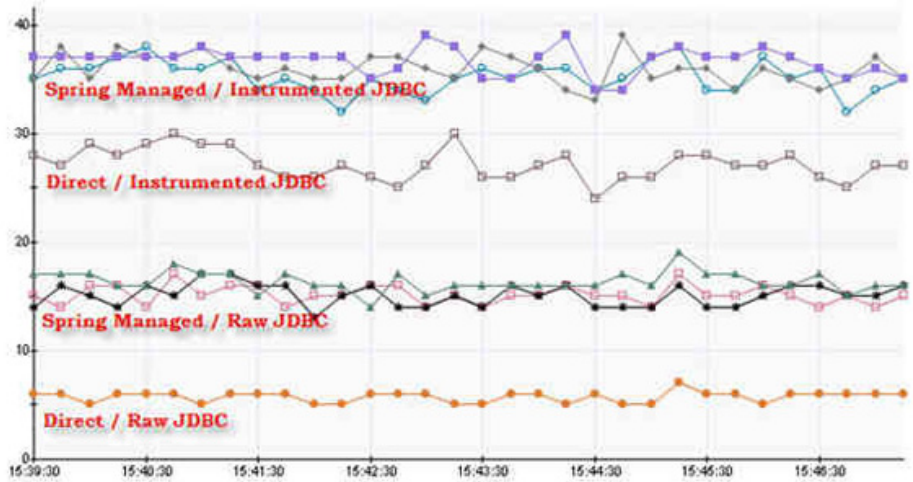
このテスト・ケースで示される `DAO.get` Bean インスタンスの数は少し多くなります。これは、インスツルメンテーションを追加されたデータ・ソースとされていないデータ・ソースが追加されているためです。表 2 に、インスツルメンテーション追加後の各シナリオとそのテスト結果の数値を比較したものを記載します。

表 2. インスツルメンテーションを追加した JDBC のテスト結果

テスト・ケース	平均経過時間 (ミリ秒)	最小経過時間 (ミリ秒)	最大経過時間 (ミリ秒)	カウント
ダイレクト・アクセス、そのままの JDBC	5	0	78	12187
ダイレクト・アクセス、インスツルメンテーションを追加した JDBC	27	0	281	8509
インターセプター Spring Bean 未使用、そのままの JDBC	15	0	125	12187
インターセプター Spring Bean 未使用、インスツルメンテーションを追加した JDBC	35	0	157	8511
インスツルメンテーションを追加した Spring Bean、そのままの JDBC	16	0	125	12189
インスツルメンテーションを追加した Spring Bean、インスツルメンテーションを追加した JDBC	36	0	250	8511
最適化されたインスツルメンテーションを追加した Spring Bean、そのままの JDBC	15	0	203	12188
最適化されたインスツルメンテーションを追加した Spring Bean、インスツルメンテーションを追加した JDBC	35	0	187	8511

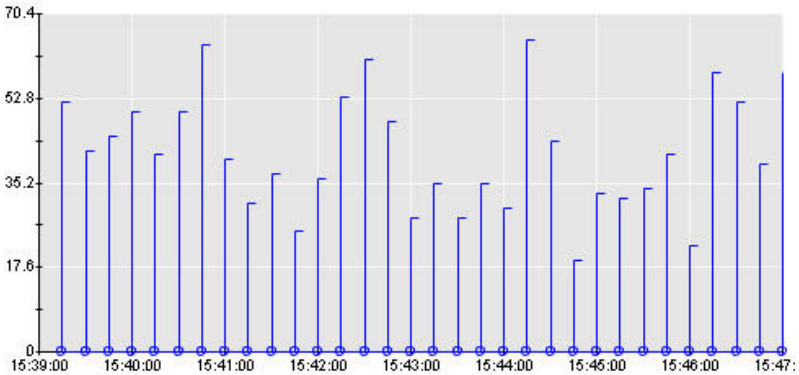
結果には興味深いパターンがいくつか現れていますが、なかでも 1 つの側面が際立っています。それは、インスツルメンテーションを追加した JDBC の速度は、そのままの JDBC の速度に比べて明らかに劣っているという点です。これは、インスツルメンテーションには可能な限りの効率化と調整が必要であることを物語る訓話だと考えてください。この基本的な JDBC のインスツルメンテーションの例でパフォーマンスの違いをもたらしている原因は、挿入したトレース、長いコード・パス、そして一連のクエリーを実行するために追加で必要となるオブジェクト作成量の組み合わせです。パフォーマンスが集中する環境で敢えてこの手法を使用するとしたら、コード・ベースでの作業がさらに必要となってきます。インスツルメンテーションを追加した `DAO.get` Bean を使用すれば、他にも明らかな影響がでてきますが、その程度は極めて小さいもので、これもやはり、分析情報の取得も含んだ呼び出しでの追加オーバーヘッド、長いコード・パス、トレース・アクティビティーによるものです。トレーサー・アダプターでどうにか調整できそうにも思えますが、どのようなインスツルメンテーションでもある程度のオーバーヘッドが伴うというのが厳しい現実です。図 11 に、このテストでの経過時間の結果を示します。

図 11. インスツルメンテーションを追加した JDBC の結果



このセクションで最後に説明するのは、データベース・レベルでまとめられるスレッド・ブロックの回数についてです。このようなデータベース・レベルの統計は、該当するデータベースに対して行われたすべての呼び出しについて、インターバル単位で収集したすべてのメトリックの集約値を表します。経過時間については平均値が出されますが、カウント (インターバル単位でのレスポンス回数、ブロック回数、待機回数) についてはインターバルごとに合計されます。このテスト・ケースでは、インターバル単位で集約された平均ブロック回数はゼロでしたが、図 12 を見れば APM 視覚化ツールの機能の 1 つがわかります。平均ブロック回数はゼロだったものの、各インターバルには最大 (および最小) 経過時間の測定値もあります。このグラフには、私が使用している APM がゼロのフラットなラインで平均ブロック回数を示しているだけでなく、経過時間の最大値も示してあります。

図 12. JDBC の集約ブロック回数



この記事の最後のセクションでは、ソースを変更しないで Java クラスにインスツルメンテーションを追加する最後の手法として、バイトコードのインスツルメンテーションを紹介します。

バイトコードのインスツルメンテーション

これまで説明してきたソースをベースとしないインスツルメンテーション手法では、オブジェクトを追加し、大抵の場合はトレース・コード自体の実行だけでなく、コードの実行過程を拡張しなければなりませんでした。バイトコード・インスツルメンテーション (BCI) は、Java クラスに

直接バイトコードを注入することによって、そのクラスが元々サポートしていなかった目的を達成する手法です。このプロセスは、開発者がソースに手を付けることなくクラスを変更したい場合、あるいはクラス定義を実行時に動的に変更させたい場合に、さまざまな使い方ができます。このセクションでは、BCI を使用してパフォーマンス監視用のインスツルメンテーションをクラスに注入する方法を説明します。

さまざな BCI フレームワークが、さまざまな方法によってパフォーマンス監視用のインスツルメンテーションをクラスに注入しています。メソッド・レベルでの単純なインスツルメンテーション手法は、ターゲット・メソッドの名前を変更し、トレース・ディレクティブが含まれ、元の (名前変更された) メソッドを呼び出す元のシグニチャーを使って新しいメソッドを挿入するというものです。オープンソースの BCI ツールである JRat は メソッド実行の経過時間収集に限った手法を説明しているため、汎用の BCI AOP ツールに比べて簡潔になっています (「参考文献」を参照)。リスト 15 に、JRat プロジェクトのサンプルを凝縮しました。

リスト 15. BCI を使用してインスツルメンテーションを追加したメソッドの例

```
//////////////////////////////////////////////////////////////
//  The Original Method
//////////////////////////////////////////////////////////////
public class MyClass {
 public Object doSomething() {
        // do something
    }
}
//////////////////////////////////////////////////////////////
//  The New and Old Method
//////////////////////////////////////////////////////////////
public class MyClass {
    private static final MethodHandler handler = HandlerFactory.getHandler(...);
    // The instrumented method
    public Object doSomething() {
        handler.onMethodStart(this);
        long startTime = Clock.getTime();
        try {
           Object result = real_renamed_doSomething(); // call your method
           handler.onMethodFinish(this, Clock.getTime() - startTime, null);
        } catch(Throwable e) {
           handler.onMethodFinish(this, Clock.getTime() - startTime, e);
           throw e;
        }
    }
    // The renamed original method
    public Object real_renamed_doSomething() {
        // do something
    }
}
```

BCI を実装するには、以下の 2 つの一般的な方法があります。

- **静的**: Java クラスまたはクラス・ライブラリーにインスツルメンテーションを追加し、そのクラスを元のクラスまたはライブラリーのコピーに保存します。このコピーをアプリケーションにデプロイすると、インスツルメンテーション化したクラスが他のすべてのクラスと同じように扱われることになります。
- **動的**: 実行時に、クラス・ロード・プロセスのなかで Java クラスにインスツルメンテーションを追加します。インスツルメンテーションを追加したクラスはメモリー内にだけ存在し、JVM が終了するとこれらのクラスは消滅します。

動的な BCI を実装することによるメリットの 1 つは、これによってもたらされる柔軟性です。動的な BCI の実装は一般に、すでに (通常はファイル内に) 構成されている一連のディレクティブに従って実行されます。そのため、インスツルメンテーションを変更するには、単にそのファイルを更新して JVM を再生すればよいだけです (ホット・スワップも広くサポートされています)。ここではまず、静的インスツルメンテーションの手順から説明することにします。

静的な BCI の実装

このサンプルでは、静的 BCI によって `EmpDAOImpl` クラスにインスツルメンテーションを追加します。ここで使用するのはオープンソースの BCI フレームワーク、JBoss AOP です (「参考文献」を参照)。

最初のステップとして、メソッド呼び出しのパフォーマンス・データを収集するために使用するインターセプターを定義します。このクラスは、`EmpDAOImpl` クラスのバイトコードに静的に組み込まれるからです。この場合の JBoss インターフェースは、私が Spring 用に定義したインターセプターと同じで、唯一、インポートされるクラス名が異なるだけです。このサンプル用のインターセプターは `org.runtimemonitoring.aop.ITracerInterceptor` です。次のステップでは、jboss-aop.xml ファイルを定義します。それには、EJB 3 インターセプターの場合に使った構文と同じ構文を使用します。このファイルは、リスト 16 のとおりです。

リスト 16. 静的 BCI 実装の jboss-aop.xml ファイル

```
<aop>
   <interceptor class="org.runtimemonitoring.aop.ITracerInterceptor" scope="PER_VM"/>
   <bind
    pointcut="execution(public * $instanceof{org.runtimemonitoring.spring.DAO}->get(..))">
      <interceptor-ref name="org.runtimemonitoring.aop.ITracerInterceptor"/>
   </bind>
</aop>
```

続いて、JBoss が提供する Aop Compiler (aopc) というツールを使って静的にインスツルメンテーションを追加するプロセスを実行します。このプロセスは Ant スクリプト内で実行すると最も簡単です。リスト 17 に、該当する Ant タスク、そしてコンパイラーによる出力のスニペットを記載します。このスニペットには、私が定義したポイントカットがターゲット・クラスにマッチしたことが示されています。

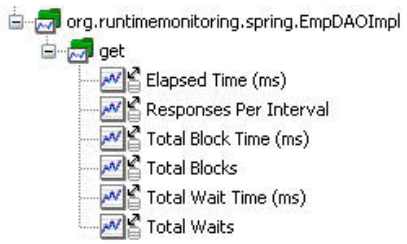
リスト 17. **aopc** Ant タスクおよび出力

```
<target name="staticBCI" depends="compileSource">
   <taskdef name="aopc" classname="org.jboss.aop.ant.AopC"
      classpathref="aop.lib.classpath"/>
   <path id="instrument.target.path">
      <path location="${classes.dir}"/>
      </path>
   <aopc compilerclasspathref="aop.class.path" verbose="true">
      <classpath path="instrument.target.path"/>
      <src path="${classes.dir}"/>
      <aoppath path="${conf.dir}/jboss-aop/jboss-aop.xml"/>
   </aopc>
</target>

Output:

[aopc] [trying to transform] org.runtimemonitoring.spring.EmpDAOImpl
[aopc] [debug] javassist.CtMethod@955a8255[public transient get
   ([Ljava/lang/Integer;)Ljava/util/Map;] matches pointcut:
   execution(public * $instanceof{org.runtimemonitoring.spring.DAO}->get(..))
```

jboss-aop.xml ファイルに定義したポイント·カットは、リスト 16 に定義されているポイントカットと同様に、AOP 固有の構文を実装します。その目的は、ポイントカットのターゲットを具体的、あるいは大まかに定義するための、表現とワイルドカードを使用した言語を提供することです。実質的には、メソッドを識別するすべての属性は、クラス名およびパッケージ名からアノテーションおよび戻りの型にマッピングすることができます。リスト 17 では、`org.runtimemonitoring.spring.DAO` のすべてのインスタンスで、`get` という public メソッドをターゲットとするように指定しています。この基準にマッチする具象クラスは唯一、`org.runtimemonitoring.spring.EmpDAOImpl` だけなので、このクラスがインスツルメンテーションを追加された唯一のクラスということになります。

これで、インスツルメンテーションの追加は完了です。このインスツルメンテーションを有効にして `SpringRunner` テスト・ケースを実行するには、JVM の起動時に `-Djboss.aop.path=[directory]/jboss-aop.xml` のような引数を使用して、jboss-aop.xml ファイルの場所をシステム・プロパティーに定義しなければなりません。前提として、jboss-aop.xml は静的インスツルメンテーションのビルド時に使用された後、実行時にも再び使用されることから、ある程度の柔軟性はあります。最初にあらゆるクラスにインスツルメンテーションを追加することができても、実行時に有効にできるのは 1 つの特定のインスツルメンテーションだけだからです。これで、`SpringRunner` テスト・ケースに対して生成される APM システムのメトリック・ツリーには、`EmpDAOImpl`のメトリックが含まれるようになります。ツリーのこの部分を、図 13 に示します。

図 13. 静的 BCI 実装のメトリック・ツリーe



静的インスツルメンテーションにはある程度の柔軟性が考えられますが、結局のところ、クラスが静的に処理されるのでない限り (これはかなり困難な作業です)、クラスのインスツルメンテーションを有効にできないという点が制約になります。さらに、静的にインスツルメンテーションが追加されたクラスは、インスツルメンテーション追加時に定義されたインターセプターに対してしか有効にすることができません。次のサンプルでは、この同じテスト・ケースを今度は動的な BCI の実装を使用して繰り返します。

動的な BCI の実装

動的な BCI の実装を実現するには、さまざまな方法があります。そのなかでも明らかに利点があるのは Java 1.5 `javaagent` インターフェースを使用する方法です。このインターフェースについては概要を説明するだけにとどめるので、詳細については Andrew Wilcox の記事「独自のプロファイリング・ツールを構築する」(「参考文献」を参照) を読んでください。

`javaagent` は 2 つの構成体によって実行時の動的な BCI の実装を可能にします。まず、`-javaagent:a` JAR file (JAR file には `javaagent` 実装が含まれる JAR ファイルを指定) を指定して JVM を起動すると、JVM は特殊なマニフェスト・エントリーに定義されたクラスの `public static void premain(String args, Instrumentation inst)` メソッドを呼び出します。`premain` という名前からわかるように、このメソッドはメインの Java アプリケーション・エントリー・ポイントの前で呼び出されるため、呼び出されたクラスは、ロードされたクラスに必ず真っ先にアクセスして変更を開始することができます。メソッドはそのために `ClassTransformer` (2 つ目の構成体) のインスタンスを登録します`#ClassTransformer` インターフェースの役割は、クラス・ローダーからの呼び出しを効率的にインターセプトし、ロードされたクラスのバイトコードをその場で書き直すことです。この `ClassTransformer`が持つ唯一のメソッド、`transform` には、再定義されるクラスと、そのクラスのバイトコードが含まれるバイト配列が渡されます。すると、transform メソッドはあらゆる類の変更を実装し、変更後 (またはインスツルメンテーション追加後) のクラスのバイトコードが含まれる新しいバイト配列を返します。このモデルでは、素早く効率的なクラス変換が可能で、今までのメソッドとは異なり、ネイティブ・コンポーネントを操作する必要はありません。

動的な BCI を `SpringRunner` テスト・ケースに実装するには、2 つのステップが必要です。まず、`org.runtimemonitoring.spring.EmpDAOImpl` クラスを再コンパイルして、前のテスト・ケースでの静的な BCI を削除します。次に、JVM の起動オプションとして `-Djboss.aop.path=[directory]/jboss-aop.xml` オプションをそのまま使用し、`javaagent` オプションを追加する必要があります。

```
-javaagent:[directory name]/jboss-aop-jdk50.jar
```

リスト 18 に、動的 BCI を実装するメリットを説明するために少し変更した jboss-aop.xml ファイルを記載します。

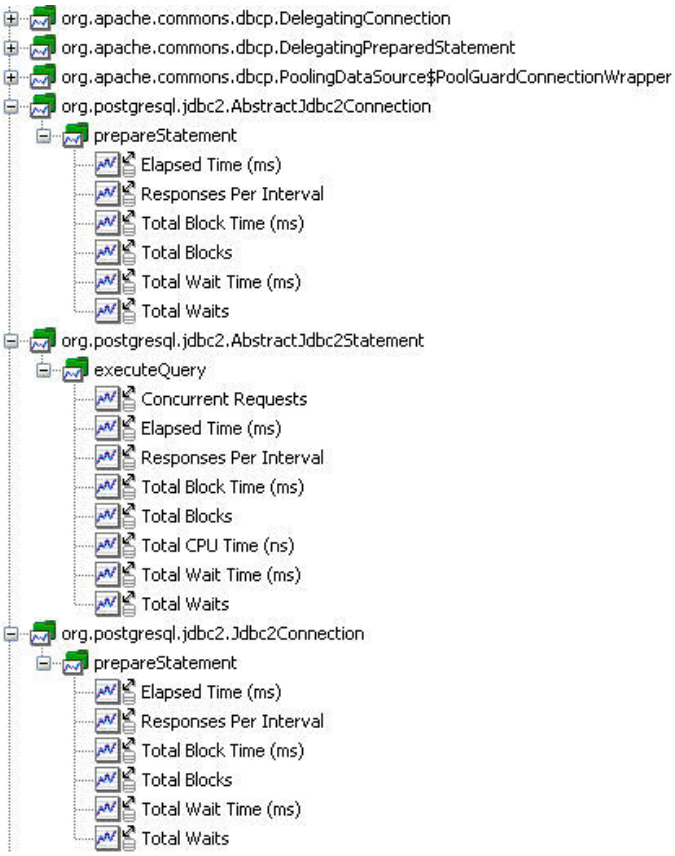
リスト 18. 簡略化した動的な BCI を実装した jboss-aop.xml ファイル

```
<interceptor class="org.runtimemonitoring.aop.ITracerInterceptor"
   scope="PER_VM"/>
<interceptor class="org.runtimemonitoring.aop.PreparedStatementInterceptor"
   scope="PER_VM"/>
<bind
 pointcut="execution(public * $instanceof{org.runtimemonitoring.spring.DAO}->get(..))">
   <interceptor-ref name="org.runtimemonitoring.aop.ITracerInterceptor"/>
</bind>
<bind
 pointcut="execution(public * $instanceof{java.sql.Connection}->prepareStatement(..))">
   <interceptor-ref name="org.runtimemonitoring.aop.ITracerInterceptor"/>
</bind>
 pointcut="execution(public * $instanceof{java.sql.PreparedStatement}->executeQuery(..))">
   <interceptor-ref name="org.runtimemonitoring.aop.ITracerInterceptor"/>
</bind>
```

メリットの 1 つとして挙げられるのは、サード・パーティーのライブラリーを含め、どのクラスにでもインスツルメンテーションを追加できることです。そのため、リスト 18 には `java.sql.Connection` のすべてのインスタンスでのインスツルメンテーションを記載しています。それにも増して強力なメリットは、任意の (ただし適用可能な) インターセプターを、定義されたあらゆるポイントカットに適用できることです。例えば、`org.runtimemonitoring.aop.PreparedStatementInterceptor` は平凡なインターセプターですが、`ITracerInterceptor` とは多少異なります。インターセプターのライブラリー (AOP 用語ではより一般的で広範な意味を持つアスペクト) は、まるごと開発することも、オープンソースのプロバイダーから入手することもできます。これらのアスペクト・ライブラリーによって、適用するインスツルメンテーションのタイプ、インスツルメンテーションを追加する必要のある API、あるいは複数の項目からなるカスタマイズしたオーバーラップのそれぞれに応じて役に立つ、広範なパースペクティブを提供することができます。

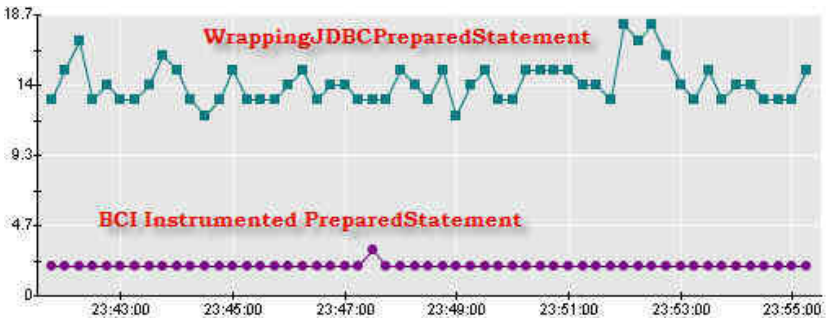
これらのメトリックが追加されたメトリック・ツリーは図 14 のとおりです。Spring で Jakarta Commons の `DataSource` プロバイダーを使用しているため、いくつかのクラスでは `java.sql` インターフェースを実装することに注意してください。

図 14. 動的な BCI 実装のメトリック・ツリー



BCI 手法を全体的に見た場合の最も大きな利点は、この記事で説明した `WrappingJDBC` インスツルメンテーションの手法と BCI によりインスツルメンテーションを追加したドライバーとでパフォーマンスの違いを比較してみれば明らかです。図 15 に、この比較を表示します。この図には、`PreparedStatement.executeQuery` の相対的な経過時間が示されています。

図 15. BCI とラップのパフォーマンス比較



第 2 回のまとめ

今回の記事では、有益なパフォーマンス監視データを APM システムにトレースするために Java アプリケーションにインスツルメンテーションを追加する数々の方法を紹介しました。ここで概説した手法では、元のソース・コードを変更する必要はありません。どの手法が最適なのかを判断するには個々のケースごとの評価が必要ですが、BCI が主流になっていることは明らかです。Java

パフォーマンス管理を目的にインスツルメンテーションを実装する自作の AMP システム、あるいはオープンソースまたは商用の APM システムは、パフォーマンスと可用性にすぐれたシステムに無くてはならない部分となります。

連載最終回の第 3 回目の記事では、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を説明します。そしてデータ管理、データ仮想化、レポートの作成、アラートの起動など、アプリケーション・パフォーマンス管理に伴う問題を取り上げて、連載を締めくくります。

今すぐ、第 3 回を読んでください。

ダウンロード

内容	ファイル名	サイズ
Sample code for this article	j-rtm2.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。