独自のプロファイリング・ツールを構築する

Java 5 のエージェント・インターフェースと AOP を使って理想的なプロファイラーを作る

Andrew Wilcox
Senior Architect
MentorGen LLC

2006年 3月 14日

プロファイリングというのは、ソフトウェア・プログラムが (CPU 時間やメモリーを含めた) リソースをどこで消費しているかを測定するための手法です。この記事では、ソフトウェア・アーキテクトの Andrew Wilcox が、プロファイリングの利点と、現状で利用可能なプロファイリングの選択肢について、またそれらの欠点について説明します。その後で、新しい Java 5 エージェント・インターフェースと単純なアスペクト指向プログラミング手法を使って、独自のプロファイラーを構築する方法を説明します。

System.out.println() を使っている場合であれ、hprof や OptimizeIt のようなプロファイリング・ツールを使っている場合であれ、コードのプロファイリングは、ソフトウェア開発作業の必須コンポーネントであるべきものです。この記事では、最も一般的なコード・プロファイリング手法を説明し、またそれらの欠点について説明します。次に、理想的なプロファイラーに求められる機能のリストを示し、こうした機能の幾つかを実現する上で、アスペクト指向の手法が適していることを説明します。また、JDK 5.0 のエージェント・インターフェースを紹介し、これを使って独自のアスペクト指向プロファイラーを構築する手順を、順を追って説明します。

この記事で例として使ったプロファイラーと完全なソースコードは、JIP (Java Interactive Profiler) に基づいていることに注意してください。JIP は、アスペクト指向の手法と Java 5 のエージェント・インターフェースを使って構築されたオープンソースのプロファイラーです。JIP に関して、また、この記事で取り上げるツールに関しては、参考文献を見てください。

プロファイリング・ツールと手法

アプリケーションのパフォーマンスを測定しようとする場合、大部分の Java 開発者は、まず System.currentTimeMillis() や System.out.println() を使います。System.currentTimeMillis() の使い方は簡単です。単純にメソッドの最初と最後で時間を測定し、その差を出力するだけです。しかし、これには 2 つの大きな欠点があります。

- 手動プロセスであるため、測定対象コードを決定してコードを実装する必要があり、再コンパイルし、再デプロイし、実行し、結果を分析し、終わったら実装コードを終了しなければ

なりません。そして次回問題が起きた時には、まったく同じステップを完全に繰り返さなければなりません。
- アプリケーションの全部分の動作を包括的に見ることができません。

こうした問題を避けるために、一部の開発者は、hprof や JProbe、OptimizeIt などのコード・プロファイラーを使います。プロファイラーを使うためにプログラムを変更する必要はないため、プロファイラーを使うと、一時的な測定を行うことに伴う問題を避けることができます。またプロファイラーは、ある特定部分のコード・セクションのタイミング情報だけではなく、全てのメソッド・コールに対するタイミング情報を収集します。そのためプロファイラーを使うことによって、プログラムのパフォーマンスを、より包括的に見ることができます。しかし、プロファイラーにも欠点があるのです。

プロファイラーの限界

プロファイラーは、System.currentTimeMillis() のような手動ソリューションに比べて便利ですが、とても理想的なものではありません。例えば、hprof を使ってプログラムを実行すると、プログラムのスピードは 20 分の 1 ほどに低下します。これはつまり、通常ならば 1 時間で終わる ETL (extract, transform, and load) 操作が、プロファイラーを使うと丸一日かかるかも知れないということです。それを待っている時間が無駄なだけではなく、アプリケーションのタイムスケールが変わることによって、結果も歪んだものとなります。例えば、大量の I/O を行うプログラムを考えてみてください。I/O はオペレーティング・システムが行うため、プロファイラーがスピードを低下させることはありません。そのため、この I/O は実際よりも 20 倍も速く実行するように見えてしまいます。そのため、アプリケーションのパフォーマンスの正確な姿を知る上で、必ずしも hprof は信頼できないことになります。

hprof の問題のもう一つは、Java プログラムのロードや実行に関係しています。Java プログラムは C や C++のように静的にリンクされるプログラムとは異なり、コンパイル時ではなく実行時にリンクされます。JVM は、そのクラスが最初に参照されるまで、そのクラスをロードしません。また、コードが何度か実行されるまで、そのコードはバイトコードからマシン・コードにはコンパイルされません。あるメソッドのパフォーマンスを測定しようとする場合、そのクラスがまだロードされていなければ、測定時間には実行時間の他にクラスのロード時間とコンパイル時間が含まれることになります。これらはアプリケーションのライフのうち、最初の部分でしか発生しないため、ライフの長いアプリケーションのパフォーマンスを測定する場合には、そうした時間を含めたくないものです。

コードがアプリケーション・サーバーやサーブレット・エンジンの中で実行している場合には、さらに話が複雑になります。hprof のようなプロファイラーは、アプリケーション全体、そしてサーブレット・コンテナーなど全てのプロファイリングを行ってしまいます。これは困ったことで、通常はサーブレット・エンジンのプロファイルなどは必要なく、アプリケーションのプロファイリングだけを行いたいのです。

理想的なプロファイラー

他のツールを選ぶ場合と同様、プロファイラーの選択にもトレードオフがあります。hprof のような無料ツールは使いやすいのですが、プロファイルからクラスやパッケージをフィルターで除外できないなど、制約があります。商用ツールは豊富な機能を備えていますが、高価であり、厳格

なライセンス条件があります。一部のプロファイラーでは、プロファイラーを通してアプリケーションを起動する必要があります。これは、慣れないツールに合わせて実行環境を再構築しなければならない、ということを意味します。プロファイラーの選択には妥協が必要ですが、では理想的なプロファイラーとは、どんなものなのでしょう。下記は理想的なプロファイラーに求められる機能のリストです。

- スピード: プロファイリングは、悲しくなるほど遅いものです。しかし、全クラスを自動的にプロファイリングしたりしないプロファイラーを使えば、スピードを向上させることができます。
- 対話性: プロファイラーの対話動作が豊富であれば豊富であるほど、プロファイラーから得られる情報を微調整することができます。例えば実行時にプロファイラーをオン/オフできれば、クラス・ローディングやコンパイル、インタプリタの実行 (pre-JIT) 回数などの測定を避けることができます。
- フィルタリング: クラスやパッケージでフィルタリングできれば、手元にある問題のみに集中でき、多すぎる情報に圧倒されずに済みます。
- 100%純 Java コード: 大部分のプロファイラーはネイティブ・ライブラリーを必要としますが、そうすると使用できるプラットフォームの数が制限されてしまいます。理想的なプロファイラーは、ネイティブ・ライブラリーを必要とすべきではありません。
- オープンソース: 一般的にオープンソース・ツールの方が手軽に使え、また商用ライセンスの制限を気にせずに済みます。

自分で作りましょう

System.currentTimeMillis() を使ってタイミング情報を生成する上での問題は、手動プロセスであることです。そうした問題の多くは、コード実装を自動化できれば解消されます。そして、こうした種類の問題は、アスペクト指向ソリューションの候補として理想的なのです。Java 5 で導入されたエージェント・インターフェースを利用すると、容易にクラスローダーにフックでき、またロードしながらクラスを修正することができます。そのため、アスペクト指向のプロファイラーを構築するためには理想的なインターフェースと言うことができます。

この記事のこれから先では、BYOP (build your own profiler: 独自のプロファイラーを作る) に焦点を当てます。まずエージェント・インターフェースを紹介し、単純なエージェントの作り方を説明します。そして基本的なプロファイリング・アスペクト用のコードを学び、またそれを修正して、より高度なプロファイリングを行うための手順も学びます。

エージェントを作成する

残念ながら、-javaagent という JVM オプションは、ほとんど文書化されていません。このオプションを取り上げた本はほとんど無いようです (『猿でも分かる Java エージェント』や、『21 日でマスターする Java エージェント』などという本はありません) 。しかし参考文献には、役立ちそうな資料を幾つか挙げてあります。そして、以下の概説も役に立つはずです。

エージェントの背景にある基本概念は、「JVM がクラスをロードする場合、エージェントはそのクラスのバイトコードを修正できる」という考え方です。エージェントは、次の 3 つのステップで作成することができます。

1. java.lang.instrument.ClassFileTransformer インターフェースを実装します。

```
public interface ClassFileTransformer {

public byte[] transform(ClassLoader loader, String className,
        Class<?> classBeingRedefined, ProtectionDomain protectionDomain,
        byte[] classfileBuffer) throws IllegalClassFormatException;

}
```

2. 「premain」メソッドを作成します。このメソッドは、アプリケーションの main() メソッドの前に呼ばれます。これは次のようなものです。

```
package sample.verboseclass;

public class Main {
    public static void premain(String args, Instrumentation inst) {
        ...
    }
}
```

3. エージェント JAR ファイルの中に、premain() メソッドを含むクラスを識別する manifest エントリーを含めます。

```
Manifest-Version: 1.0
Premain-Class: sample.verboseclass.Main
```

単純なエージェント

プロファイラーを作るための最初のステップは、クラスをロードする際に各クラスの名前を出力する単純なエージェントを作成することです。これは、-verbose:class という JVM オプションの振る舞いと似ています。リスト 1 を見ると分かるように、これには数行のコードしか必要ありません。

リスト 1.単純なエージェント

```
package sample.verboseclass;

public class Main {

    public static void premain(String args, Instrumentation inst) {
        inst.addTransformer(new Transformer());
    }
}

class Transformer implements ClassFileTransformer {

    public byte[] transform(ClassLoader l, String className, Class<?> c,
            ProtectionDomain pd, byte[] b) throws IllegalClassFormatException {
        System.out.print("Loading class: ");
        System.out.println(className);
        return b;
    }
}
```

もしエージェントが vc.jar という JAR ファイルにパッケージされている場合には、-javaagent オプションを使って JVM を起動します (下記) 。

```
java -javaagent:vc.jar MyApplicationClass
```

プロファイリング・アスペクト

エージェントの基本要素は用意できたので、次のステップは、 (アプリケーション・クラスのロードに際して) アプリケーション・クラスに単純なプロファイリング・アスペクトを追加すること

です。幸い、バイトコードを修正するための JVM 命令セットの詳細をマスターする必要はありません。そうした命令セットの代わりに、ASM ライブラリーのようなツールキット (ObjectWeb コンソーシアムのツールです。参考文献を見てください) を使って、クラスファイル・フォーマットの詳細を処理することができます。ASM は Java バイトコード操作用のフレームワークであり、Visitor パターンを使ってクラスファイルを変換することができます。この変換方法は、SAX イベントを使って XML 文書をトラバースし、変換する場合と、ほとんど同じです。

リスト 2 は、JVM がメソッドに出入りする度にクラス名やメソッド名、タイムスタンプなどを出力するプロファイリング・アスペクトです。(より高度なプロファイラーとするためには、Java 5 の System. nanoTime() のような高分解能タイマーを使うべきでしょう。)

リスト 2.単純なプロファイリング・アスペクト

```
package sample.profiler;

public class Profile {

    public static void start(String className, String methodName) {
        System.out.println(new StringBuilder(className)
            .append('\t')
            .append(methodName)
            .append("\tstart\t")
            .append(System.currentTimeMillis()));
 }

    public static void end(String className, String methodName) {
        System.out.println(new StringBuilder(className)
            .append('\t')
            .append(methodName)
            .append("\end\t")
            .append(System.currentTimeMillis()));
    }
}
```

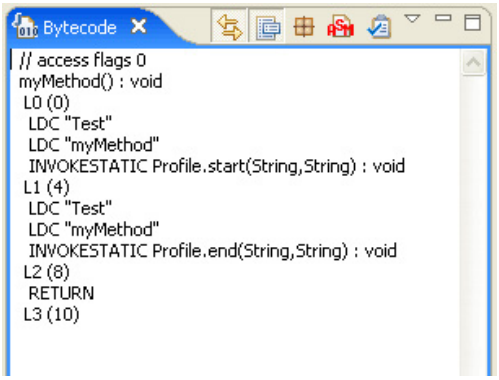
もし手動でプロファイリングする場合には、次のステップは、すべてのメソッドを下記のように修正することです。

```
void myMethod() {
    Profile.start("MyClass", "myMethod");
    ...
    Profile.end("MyClass", "myMethod");
}
```

ASM プラグインを使う

今度は、Profile.start() コールや Profile.end() コール用のバイトコードがどんなものかを考えなければなりませんここに ASM ライブラリーが登場するのです。ASM は Eclipse 用のバイトコード・アウトライン・プラグイン (参考文献) を持っており、これを使うと任意のクラスやメソッドのバイトコードを見ることができます。図 1 は、上記のメソッドに対するバイトコードを示しています。 (JDK の一部である javap のような逆アセンブラーを使うこともできます。)

図 1. ASM プラグインを使ってバイトコードを見る



この ASM プラグインは、なんと対応バイトコードを生成する ASM コードの生成まで行うことができます。これを図 2 に示します。

図 2. ASM プラグインがコードを生成する

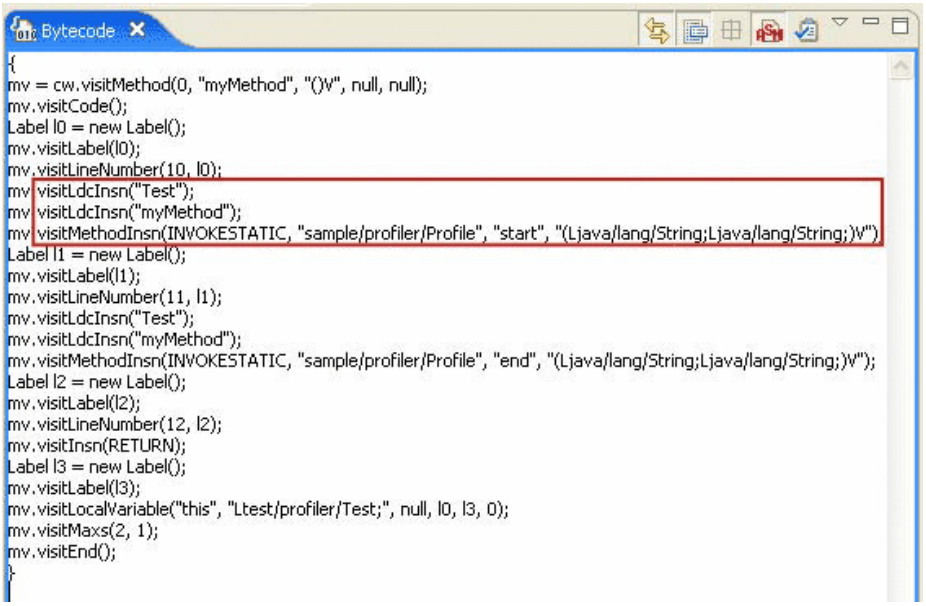


図 2 でハイライトされたコードをカット・アンド・ペーストでエージェントに貼り付け、一般化された Profile. start() メソッドを呼ぶことができます。これをリスト 3 に示します。

リスト 3.プロファイラーにコールを注入する ASM コード

```
visitLdcInsn(className);
visitLdcInsn(methodName);
visitMethodInsn(INVOKESTATIC,
 "sample/profiler/Profile",
 "start",
 "(Ljava/lang/String;Ljava/lang/String;)V");
```

起動と停止のコールを注入するためには、ASM の MethodAdapter をサブクラス化します。これをリスト 4 に示します。

リスト 4.プロファイラーにコールを注入する ASM コード

```
package sample.profiler;
```

```
import org.objectweb.asm.MethodAdapter;
import org.objectweb.asm.MethodVisitor;
import org.objectweb.asm.Opcodes;

import static org.objectweb.asm.Opcodes.INVOKESTATIC;


public class PerfMethodAdapter extends MethodAdapter {
    private String className, methodName;

    public PerfMethodAdapter(MethodVisitor visitor, String className,
            String methodName) {
        super(visitor);
        className = className;
        methodName = methodName;
    }

    public void visitCode() {
        this.visitLdcInsn(className);
        this.visitLdcInsn(methodName);
        this.visitMethodInsn(INVOKESTATIC,
            "sample/profiler/Profile",
            "start",
            "(Ljava/lang/String;Ljava/lang/String;)V");
        super.visitCode();
    }

    public void visitInsn(int inst) {
        switch (inst) {
        case Opcodes.ARETURN:
        case Opcodes.DRETURN:
        case Opcodes.FRETURN:
        case Opcodes.IRETURN:
        case Opcodes.LRETURN:
        case Opcodes.RETURN:
        case Opcodes.ATHROW:
            this.visitLdcInsn(className);
            this.visitLdcInsn(methodName);
            this.visitMethodInsn(INVOKESTATIC,
                "sample/profiler/Profile",
                "end",
                "(Ljava/lang/String;Ljava/lang/String;)V");
            break;
        default:
            break;
        }

        super.visitInsn(inst);
    }
}
```

これをエージェントにフックするためのコードは非常に単純であり、この記事のソース・ダウンロードの一部に含まれています。

ASM クラスをロードする

このエージェントは ASM を使っているので、全てが動作するためには確実に ASM クラスがロードされている必要があります。Java アプリケーションには、多くのクラスパスがあります (アプリケーション・クラスパスやエクステンション・クラスパス、ブートストラップ・クラスパスなど) 。驚いたことに、ASM JAR は、これらのどこにも入りません。代わりにマニフェストを使って、どの JAR ファイルがエージェントに必要かを JVM に伝えるのです。これをリスト 5 に示し

ます。この場合 JAR ファイルは、エージェントの JAR と同じディレクトリーにある必要があります。

リスト 5.プロファイラー用のマニフェスト・ファイル

```
Manifest-Version: 1.0
Premain-Class: sample.profiler.Main
Boot-Class-Path: asm-2.0.jar asm-attrs-2.0.jar asm-commons-2.0.jar
```

プロファイラーを実行する

全てのコンパイルが終わり、パッケージができたら、任意の Java アプリケーションに対してプロファイラーを実行することができます。リスト 6 は、エージェントをコンパイルする build.xml を実行している Ant プロファイルからの出力の一部です。

リスト 6.プロファイラーからの出力の例

```
org/apache/tools/ant/Main                       runBuild          start   1138565072002
org/apache/tools/ant/Project                    <init>            start   1138565072029
org/apache/tools/ant/Project$AntRefTable        <init>            start   1138565072031
org/apache/tools/ant/Project$AntRefTable        <init>            end     1138565072033
org/apache/tools/ant/types/FilterSet            <init>            start   1138565072054
org/apache/tools/ant/types/DataType             <init>            start   1138565072055
org/apache/tools/ant/ProjectComponent           <init>            start   1138565072055
org/apache/tools/ant/ProjectComponent           <init>            end     1138565072055
org/apache/tools/ant/types/DataType             <init>            end     1138565072055
org/apache/tools/ant/types/FilterSet            <init>            end     1138565072055
org/apache/tools/ant/ProjectComponent           setProject        start   1138565072055
org/apache/tools/ant/ProjectComponent           setProject        end     1138565072055
org/apache/tools/ant/types/FilterSetCollection  <init>            start   1138565072057
org/apache/tools/ant/types/FilterSetCollection  addFilterSet      start   1138565072057
org/apache/tools/ant/types/FilterSetCollection  addFilterSet      end     1138565072057
org/apache/tools/ant/types/FilterSetCollection  <init>            end     1138565072057
org/apache/tools/ant/util/FileUtils             <clinit>          start   1138565072075
org/apache/tools/ant/util/FileUtils             <clinit>          end     1138565072076
org/apache/tools/ant/util/FileUtils             newFileUtils      start   1138565072076
org/apache/tools/ant/util/FileUtils             <init>            start   1138565072076
org/apache/tools/ant/taskdefs/condition/Os      <clinit>          start   1138565072080
org/apache/tools/ant/taskdefs/condition/Os      <clinit>          end     1138565072081
org/apache/tools/ant/taskdefs/condition/Os      isFamily          start   1138565072082
org/apache/tools/ant/taskdefs/condition/Os      isOs              start   1138565072082
org/apache/tools/ant/taskdefs/condition/Os      isOs              end     1138565072082
org/apache/tools/ant/taskdefs/condition/Os      isFamily          end     1138565072082
org/apache/tools/ant/util/FileUtils             <init>            end     1138565072082
org/apache/tools/ant/util/FileUtils             newFileUtils      end     1138565072082
org/apache/tools/ant/input/DefaultInputHandler  <init>            start   1138565072084
org/apache/tools/ant/input/DefaultInputHandler  <init>            end     1138565072085
org/apache/tools/ant/Project                    <init>            end     1138565072085
org/apache/tools/ant/Project                    setCoreLoader     start   1138565072085
org/apache/tools/ant/Project                    setCoreLoader     end     1138565072085
org/apache/tools/ant/Main                       addBuildListener  start   1138565072085
org/apache/tools/ant/Main                       createLogger      start   1138565072085
org/apache/tools/ant/DefaultLogger              <clinit>          start   1138565072092
org/apache/tools/ant/util/StringUtils           <clinit>          start   1138565072096
org/apache/tools/ant/util/StringUtils           <clinit>          end     1138565072096
```

コール・スタックを追跡する

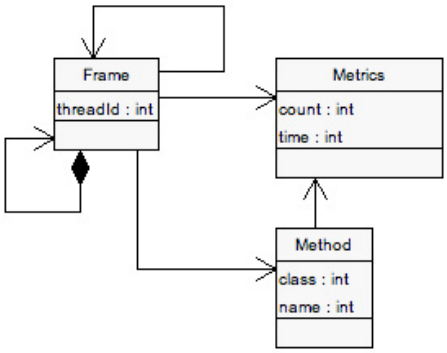
ここまでは、単純なアスペクト指向プロファイラーを数行のコードのみで構築する方法を見てきました。このプロファイラーは手始めとしては適切ですが、スレッドやコール・スタック・デー

タは収集しません。コール・スタック情報は、全体的な実行時間や実質の実行時間を判断するために必要です。また、各コール・スタックはスレッドに関連付けられているため、コール・スタック・データを追跡する場合にはスレッド情報も必要です。大部分のプロファイラーは、こうした分析を 2 パス方式で行います。つまり最初にデータを収集し、その後で分析を行うのです。ここでは単に収集されるデータを出力するよりも、この方法の実際を示すことにしましょう。

Profile クラスを修正する

コール・スタックやスレッド情報をキャプチャーできるように Profile クラスを機能強化するのは簡単です。まず、各メソッドの開始と終了で時間を出力する代わりに、その情報を、図 3 に示すデータ構造を使って保存します。

図 3. コール・スタックやスレッド情報を追跡するためのデータ構造



コール・スタックに関する情報を収集するためには幾つかの方法があります。その 1 つは Exception をインスタンス化する方法ですが、これを各メソッドの開始と終了で行うのでは遅すぎます。もっと単純な方法としては、プロファイラーが自分の内部コール・スタックを管理するようにします。start() はメソッド毎に呼ばれるので、この方法は簡単です。唯一、面倒な点は、例外が投げられた時に内部コール・スタックを「巻き戻す」部分です。しかし Profile.end() が呼ばれた時に想定されていたクラスとメソッド名をチェックすれば、いつ例外が投げられたかを判断することができます。

出力の設定も、同じくらい容易です。Runtime.addShutdownHook() を使って『shutdown』フックを作り、Thread を登録します (この Thread はシャットダウン時に実行され、プロファイリング・レポートをコンソールに出力します) 。

まとめ

この記事では、現在最も一般的に使われているプロファイリング用のツールと技術を紹介し、またそれらの限界を説明しました。そして、理想的なプロファイラーに望まれる機能をリストアップしました。最後に、アスペクト指向プログラミングと Java 5 のエージェント・インターフェースの使い方を学び、理想的な機能の幾つかを含んだ独自のプロファイラーを構築しました。

この記事で使用したコードは、ここで解説した方法で構築されたオープンソースのプロファイラーである JIP (Java Interactive Profiler) に基づいています。JIP は、この記事の例で取り上げた基本機能の他に、次のような機能も持っています。

- 対話型プロファイリング
- クラスやパッケージを排除する機能
- ある特定なクラスローダーがロードしたクラスのみを含める機能
- オブジェクト・アロケーションを追跡する機構
- (コード・プロファイリングに加えて) パフォーマンス測定機能

JIP は BSD スタイルのライセンスで配布されています。ダウンロード方法については参考文献を見てください。

ダウンロード

内容	ファイル名	サイズ
Source code for the simple profiler developed here	j-jipsimple-profiler-src.zip	51KB
Source code for the verbose-class example	j-jipverbose-class-src.zip	2KB

著者について

Andrew Wilcox

ndrew Wilcoxはオハイオ州コロンバスにあるMentorGen LLCのソフトウェア・アーキテクトです。15年以上の業界経験があり、そのうちの9年間はJavaプラットフォームを使ってきました。専門は、フレームワークやパフォーマンス調整、メタプログラミングなどです。彼は、開発者の生産性やソフトウェアの信頼性を高めるためのツールや手法に焦点を当てた活動を行っています。彼はJava Interactive Profilerの作成者でもあります。