log4jによって可能になるログの制御

Java用のオープン・ソースlog4j APIは、高速で効果的なログ・サービスを可能にします

Ceki Gulcu

2001年 1月 01日

log4jはオープン・ソース・プロジェクトです。これを使用すると、デベロッパーは、どのログ・ステートメントを出力するかを任意の細分度で制御できます。log4jでは、外部の設定ファイルを使用することにより、実行時に完全な設定ができます。とりわけ、log4jの描く学習曲線は緩やかで、取り組みやすいものです。注意:ユーザーのフィードバックから判断すると、log4jは強い中毒性も持ち合わせています。プロジェクトの創始者であるCekiGülcüが、log4j API、その固有の機能、およびその設計理論について説明します。

大部分の大規模なアプリケーションには、独自のロギング(logging)APIまたはトレースAPIが組み込まれています。経験の示すところによれば、ロギングは開発サイクル全体にわたって重要な要素の1つです。そうした要素の1つとして、ロギングにはいくつかの利点があります。第一に、アプリケーション実行の正確なコンテキストを明らかすることができます。いったんコードに挿入すれば、人手を煩わすことなくログ出力が生成されます。第二に、ログ出力を永続的なメディアに保管して、後でこれを調べることができます。最後に、十分な機能を備えたロギング・パッケージは、開発サイクルで使用するのにとどまらず、監査ツールとして利用することもできます。

この観点から、1996年初頭、EU SEMPER (Secure Electronic Marketplace forEurope) プロジェクトは独自のトレースAPIを作成することに決定しました。そのAPIは、数え切れない機能強化、何回かに渡る実装、および多くの作業を経て、Java用のポピュラーなロギング・パッケージであるlog4jに進化しました。パッケージは、OpenSource Initiativeの承認のもと、IBM Public Licenseに基づいて配布されています。

ロギングには欠点もあります。アプリケーションの実行速度を低下させる場合がありますし、ログ記録があまりにも詳細だと、スクロールしないと全体を見渡せなくなることもあります。こうした事柄を解決するため、log4jは高速性と柔軟性を目指した設計が施されています。ロギングがアプリケーションの主な焦点となることはめったにないので、log4jAPIは、簡単に理解して使えるものであることを心がけて設計されています。

この記事では、まずlog4jの主要なコンポーネントを説明します。次に、基本的な使用法と設定を説明する簡単な例を示します。最後に、パフォーマンスの問題と、Sunがまもなく発表するロギングAPIについて触れます。

カテゴリー、appender、およびレイアウト

log4jには3つの主要なコンポーネントがあります。

- カテゴリー
- appender
- レイアウト

これらの3つのコンポーネントが協働する結果、デベロッパーはメッセージのログをメッセージ・タイプや優先度に応じて記録したり、メッセージのフォーマットや報告先を実行時に制御したりすることができます。各コンポーネントを順番に調べてみましょう。

カテゴリー階層

あらゆるロギングAPIが単純な`System.out.println` に比べて勝っている最大の点は、一部のログ・ステートメントだけを無効にする一方で、残りのログ・ステートメントをそのまま出力できるという点です。この能力は、デベロッパーが選択する基準に従ってロギング・スペース(つまり、すべての可能なロギング・ステートメントのスペース) がカテゴリー化されることを前提としています。

これに合致して、`org.log4j.Category` クラスはパッケージのコア部分を成しています。カテゴリーは名前付きエンティティーです。Javaデベロッパーにとってなじみ深い命名体系に従い、次の場合、あるカテゴリーを別のカテゴリーの親であると言います。すなわち、カテゴリー名の後ろにドットを付けたものが、子カテゴリーの接頭部である場合です。たとえば、`com.foo` というカテゴリーは、`com.foo.Bar` というカテゴリーの親です。同様に、`java` は`java.util` の親であり、`java.util.Vector` の祖先です。

ルート・カテゴリーはカテゴリー階層の最上位に位置し、次の2つの点で例外的な存在です。

1. 常に存在する。
2. 名前で検索できない。

`Category` クラスで静的な`getRoot()` メソッドを起動すると、ルート・カテゴリーが検索されます。静的な `getInstance()` メソッドは、他のすべてのカテゴリーのインスタンスを生成します。`getInstance()` は、必要なカテゴリーの名前をパラメーターとします。`Category` クラスの基本的なメソッドをいくつか以下にリストします。

```
package org.log4j;
public Category class {
   // Creation & retrieval methods:
   public static Category getRoot();
   public static Category getInstance(String name);
   // printing methods:
   public void debug(String message);
   public void info(String message);
   public void warn(String message);
   public void error(String message);
   // generic printing method:
   public void log(Priority p, String message);
}
```

カテゴリーには優先度を割り当てることが可能 です。優先度のセットは`org.log4j.Priority` クラスで定義されます。優先度のセットはUNIX Syslogシステムで定義されているセットと一致するものの、log4jでは、ERROR、WARN、INFO、およびDEBUGの4つの優先度だけ使用することを奨励しています(順番に優先度が低くなります)。一見するとこのセットは制限されているように思えますが、その背後にある根本な考え方は、静的な優先度セット(それがたとえ大きいセットであるとしても) を使うよりも柔軟性に富んだカテゴリー階層を利用していくことにあります。とはいえ、`Priority` クラスをサブクラス化して、独自の優先度を定義することも可能です。あるカテゴリーが、割り当てられた優先度を持たない場合、割り当てられた優先度を持つ、最も近い祖先から優先度を継承します。したがって、すべてのカテゴリーが最終的に優先度を継承できるよう、ルート・カテゴリーには、割り当てられた優先度が必ずあります。

ロギング要求を行うには、カテゴリー・インスタンスの出力メソッドの1つを起動します。以下に挙げる出力メソッドです。

- `error()`
- `warn()`
- `info()`
- `debug()`
- `log()`

定義から明らかなとおり、出力メソッドはロギング要求の優先度を決定します。たとえば、`c` がカテゴリー・インスタンスである場合、ステートメント `c.info("..")` は、優先度INFOのロギング要求です。

ロギング要求の優先度がそのカテゴリーの優先度以上である場合、そのロギング要求を有効であると言います。それ以外の場合、その要求を無効 であると言います。割り当てられた優先度のないカテゴリーは、階層から優先度を継承します。

以下に、この規則の例を示します。

```
// get a category instance named "com.foo"
Category cat = Category.getInstance("com.foo");
// Now set its priority.
cat.setPriority(Priority.INFO);
Category barcat = Category.getInstance("com.foo.Bar");
// This request is enabled, because WARN >= INFO.
cat.warn("Low fuel level.");
// This request is disabled, because DEBUG < INFO.
cat.debug("Starting search for nearest gas station.");
// The category instance barcat, named "com.foo.Bar",
// will inherit its priority from the category named
// "com.foo" Thus, the following request is enabled
// because INFO >= INFO.
barcat.info("Located nearest gas station.");
// This request is disabled, because DEBUG < INFO.
barcat.debug("Exiting gas station search");
```

同じ名前を使って`getInstance()` メソッドを呼び出すと、常にまったく同じカテゴリー・オブジェクトへの参照が戻されます。したがって、カテゴリーを設定した後、参照を渡すことなく、コード内の別の場所で同じインスタンスを検索することができます。カテゴリーの作成と設定は、どのような順序で行ってもかまいません。特に、親カテゴリーは、初期化が子より後でも、

子を見つけて、これにリンクできます。log4j環境の設定は、アプリケーションの初期化の時点で行われるのが一般的です。望ましいのは、設定ファイルを読み込んで設定する方法です。この方法については、後で簡単に説明します。

log4jでは、ソフトウェア・コンポーネント に従ってカテゴリー名を付けるのが容易です。クラスごとに静的にカテゴリーのインスタンスを生成し、クラスの完全限定名(fullyqualified name)と等しいカテゴリー名を付けるという方法でこれを行えます。これは、カテゴリーを定義するための便利で簡単な方法です。ログ出力に生成元のカテゴリーの名前が示されるので、こうした命名戦略により、ログ・メッセージの発信元を識別するのが容易になります。しかし、この命名戦略は(一般的なものであるとはいえ)、採用できるカテゴリー命名戦略の1つに過ぎません。log4jには、可能なカテゴリーに制限がありません。実際、デベロッパーは望むとおりにカテゴリーに名前を付けることができるのです。

appenderとレイアウト

カテゴリーに基づいてロギング要求を選択的に有効または無効にできることは、全体像のほんの一部に過ぎません。log4jでは、ロギング要求を複数の宛先(これをlog4jではappender という) に出力することもできます。現在のところ、コンソール、ファイル、GUIコンポーネント、リモート・ソケット・サーバー、NTイベントロガー、およびリモートUNIX Syslogデーモン用のappenderが存在しています。

1つのカテゴリーが複数のappenderを参照する場合があります。あるカテゴリーに関する有効なロギング要求は、そのカテゴリーに属するすべてのappender、およびより高い階層に属するappenderに送信されます。言い換えれば、appenderはカテゴリー階層から加算的に継承されていきます。たとえば、コンソールappenderをルート・カテゴリーに追加すると、有効なロギング要求はすべて、少なくともコンソールには出力されることになります。さらに、ファイルappenderをC というカテゴリーに追加すると、C とC の子に関する有効なロギング要求は、ファイルとコンソールに出力されることになります。デフォルトの動作を指定変更して、appenderが加算的に累積されないようにすることもできることに注意してください。

ユーザーはたいてい出力の宛先だけでなく、出力フォーマットもカスタマイズすることを望むものです。これは、appenderにレイアウトを関連付けることによって行えます。レイアウトはユーザーの希望に従ってロギング要求をフォーマット設定します。一方、appenderはフォーマット済みの出力を宛先に送信する責任を担います。標準的なlog4j配布版の一部である`PatternLayout`を使用すると、ユーザーは出力フォーマットを指定できます。C言語の`printf`関数に似た変換パターンに従って指定できます。

たとえば、`PatternLayout`に`%r [%t]%-5p %c - %m%n`という変換パターンを指定すると、次のような出力が得られます。

```
176 [main] INFOorg.foo.Bar - Located nearest gas station.
```

上記の出力の各部について説明します。

- 最初のフィールドは、プログラム開始以来の経過時間です (ミリ秒単位)。
- 2番目のフィールドは、ログ要求を出したスレッドを示します。
- 3番目のフィールドは、ログ・ステートメントの優先度を表します。

- 4番目のフィールドは、ログ要求に関連したカテゴリー名です。

`-` の後のテキストは、ステートメントのメッセージです。

設定

アプリケーション・コードにログ要求を挿入するには、相当の計画と努力が必要です。観察によると、ロギング用のコードはアプリケーション全体の約4%を占めます。したがって、中規模のアプリケーションでさえ、数千のログ・ステートメントがコードの中に組み込まれていることになります。この数字を考えると、これらのログ・ステートメントを管理することが絶対必要です。手作業でこれらのログ・ステートメントを修正する事態は避けなければなりません。

log4j環境はプログラムを介して完全に設定が可能です。しかし、設定ファイルを使用した方がはるかに柔軟に設定できます。現在のところ、設定ファイルはXMLフォーマット、またはJavaのプロパティー(キー=値) フォーマットで作成できます。

log4jを使用する架空のアプリケーション (`MyApp`) を例にとって、これをどのように行うか調べてみましょう。

```
import com.foo.Bar;
// Import log4j classes.
import org.log4j.Category;
import org.log4j.BasicConfigurator;
public class MyApp {
  // Define a static category variable so that it references the
  // Category instance named "MyApp".
  static Category cat = Category.getInstance(MyApp.class.getName());
  public static void main(String[] args) {
    // Set up a simple configuration that logs on the console.
    BasicConfigurator.configure();
    cat.info("Entering application.");
    Bar bar = new Bar();
    bar.doIt();
    cat.info("Exiting application.");
  }
}
```

上記のコードをご覧になればお分かりのとおり、`MyApp` はまずlog4j関連のクラスをインポートします。そして、`MyApp` という静的カテゴリー変数を定義します。これは、図らずもクラスの完全限定名となっています。

`MyApp` は`Bar` クラスを使用します。このクラスはパッケージ`com.foo` に定義されています。

```
package com.foo;
import org.log4j.Category;
public class Bar {
  static Category cat = Category.getInstance(Bar.class.getName());
  public void doIt() {
    cat.debug("Did it again!");
  }
}
```

`MyApp` で`BasicConfigurator.configure()` メソッドを起動すると、どちらかと言えば単純なlog4jのセットアップが作成されます。このメソッドは、ルート・カテゴリーにコンソールへの`FileAppender` 出力を追加するようハードコーディングされています。出力は、`PatternLayout`

を使ってフォーマット設定されます。`PatternLayout` は、パターン`%-4r [%t] %-5p %c %x - %m%n` に設定されています。

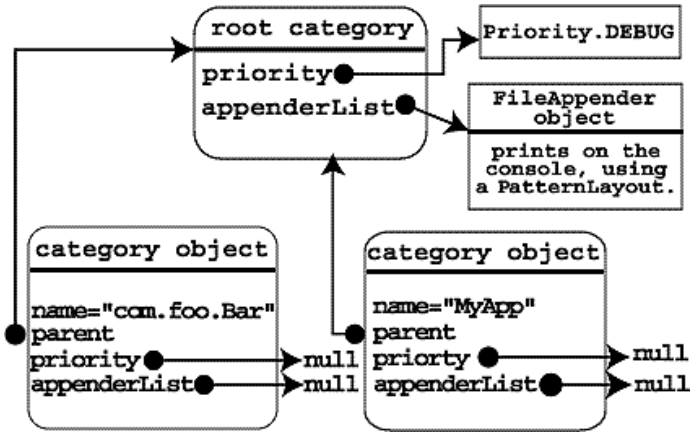
デフォルトでは、ルート・カテゴリーに`Priority.DEBUG` が割り当てられることに注意してください。

MyAppの出力は次のとおりです。

```
0 [main] INFO MyApp - Entering application.
36 [main] DEBUG com.foo.Bar - Did it again!
51 [main] INFO MyApp - Exiting application.
```

図1は、`BasicConfigurator.configure()` メソッドを呼び出した直後の`MyApp` のオブジェクト・ダイアグラムを描いたものです。

図1. BasicConfigurator.configure() メソッドを呼び出した後のMyAppのオブジェクト・ダイアグラム



`MyApp` クラスは、`BasicConfigurator.configure()` メソッドを起動することによって、log4jを設定します。他のクラスがしなければならないことは、`org.log4j.Category` クラスをインポートし、自分の使用するカテゴリーを検索し、絶えずログを記録することだけです。

先の例では、常に同じログ情報が出力されます。幸い、`MyApp` を修正して、実行時に出力を制御できるようにすることは容易です。以下に、わずかに変更を加えたバージョンを示します。

```
import com.foo.Bar;
 import org.log4j.Category;
 import org.log4j.PropertyConfigurator;
 public class MyApp {
   static Category cat = Category.getInstance(MyApp.class.getName());
   public static void main(String[] args) {
     // BasicConfigurator replaced with PropertyConfigurator.
     PropertyConfigurator.configure(args[0]);
     cat.info("Entering application.");
     Bar bar = new Bar();
     bar.doIt();
     cat.info("Exiting application.");
   }
```

このバージョンのMyAppはPropertyConfiguratorに対し、設定ファイルを構文解析して、それに合わせてロギングをセットアップするよう命じます。

ここで、設定ファイルのサンプルを見てみましょう。このサンプルでは、BasicConfiguratorを基にした先の例とまったく同じ出力結果が得られます。

```
# Set root category priority to DEBUG and its only appender to A1.
log4j.rootCategory=DEBUG, A1
# A1 is set to be a FileAppender which outputs to System.out.
log4j.appender.A1=org.log4j.FileAppender
log4j.appender.A1.File=System.out
# A1 uses PatternLayout.
log4j.appender.A1.layout=org.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-4r [%t] %-5p %c %x - %m%n
```

ここで、com.fooパッケージに属するコンポーネントの出力を見る必要がなくなったとします。以下の設定ファイルは、これを実行するための1つの方法を示しています。

```
log4j.rootCategory=DEBUG, A1
log4j.appender.A1=org.log4j.FileAppender
log4j.appender.A1.File=System.out
log4j.appender.A1.layout=org.log4j.PatternLayout
# Print the date in ISO 8601 format
log4j.appender.A1.layout.ConversionPattern=%d [%t] %-5p %c - %m%n
# Print only messages of priority WARN or above in the package com.foo.
log4j.category.com.foo=WARN
```

このファイルを使って設定した場合のMyAppの出力は、次のとおりです。

```
2000-09-07 14:07:41,508 [main] INFO MyApp - Entering application.
2000-09-07 14:07:41,529 [main] INFO MyApp - Exiting application.
```

カテゴリーcom.foo.Barには、割り当てられた優先度がないため、com.fooから優先度を継承します。com.fooについては、設定ファイルで優先度がWARNに設定されています。Bar.doIt()メソッドのログ・ステートメントの優先度はDEBUGであり、カテゴリーの優先度WARNより低くなっています。したがって、doIt()のログ要求は抑止されます。

次に、複数のappenderを使用する、別の設定ファイルを示します。

```
log4j.rootCategory=debug, stdout, R
log4j.appender.stdout=org.log4j.FileAppender
log4j.appender.stdout.File=System.out
log4j.appender.stdout.layout=org.log4j.PatternLayout
# Pattern to output the caller's file name and line number.
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] (%F:%L) - %m%n
log4j.appender.R=org.log4j.RollingFileAppender
log4j.appender.R.File=example.log
log4j.appender.R.MaxFileSize=100KB
# Keep one backup file
log4j.appender.R.MaxBackupIndex=1
log4j.appender.R.layout=org.log4j.PatternLayout
log4j.appender.R.layout.ConversionPattern=%p %t %c - %m%n
```

この設定ファイルで拡張を施したMyAppを呼び出すと、次の内容がコンソールに出力されます。

```
 INFO [main] (MyApp2.java:12) - Entering application.
DEBUG [main] (Bar.java:8) - Doing it again!
 INFO [main] (MyApp2.java:15) - Exiting application.
```

さらに、ルート・カテゴリーに2番目のappenderが割り振られているため、出力は`example.log`ファイルにも書き出されます。このファイルは、100 KBに達すると切り替えられます。その場合、`example.log` の古いバージョンは自動的に`example.log.1` に移動します。

ロギングの動作をこのように変更するために、コードを再コンパイルする必要はなかったことに注意してください。まったく同様に、UNIXSyslogデーモンにログを送ったり、`com.foo` の出力をすべてNTイベント ロガーにリダイレクトすることも容易です。同様に、ロギング・イベントをリモートlog4jサーバーに送信することも可能です。そのサーバーでは、ローカル・サーバー・ポリシーに従ってログを記録できます。たとえば、ローカル・ファイルにログを記録したり、ログ・イベントを第2のlog4jサーバーに転送したりすることができます。

Nested Diagnostic Context

現実のシステムの大部分は、複数のクライアントを同時に処理しなければなりません。マルチスレッドを利用した、そのようなシステムの典型的なインプリメンテーションの場合、さまざまなスレッドがさまざまなクライアントを処理することになります。その点を考えると、ロギングは、特に複雑な分散アプリケーションのトレースとデバッグに適しています。クライアントごとにログ出力を区別する一般的な方法は、クライアントごとに新しい別個のカテゴリーのインスタンスを生成することです。しかし、この方法を採用するなら、カテゴリーは急増し、ロギングの管理オーバーヘッドが増大します。

より手軽な技法として、同じクライアント対話によって開始されたログ要求ごとに固有のスタンプを付けることができます。これは、NeilHarrisonが説明している方法です (「参考文献」を参照)。各要求に固有のスタンプを付けるには、ユーザーはコンテキスト情報をNestedDiagnostic Context (NDC) にpushします。`NDC` クラスを以下に示します。

```
public class NDC {
    // Used when printing the diagnostic
    public static String get();
    // Remove the top of the context from the NDC.
    public static String pop();
    // Add diagnostic context for the current thread.
    public static void push(String message);
    // Remove the diagnostic context for this thread.
    public static void remove();
  }
```

NDCは、コンテキスト情報のスタック として、スレッドごとに管理されます。`org.log4j.NDC` のメソッドがすべてstaticであることに注意してください。NDCの出力をオンにすると、ログ要求が出されるたびに、log4jの適切なコンポーネントが、現在のスレッドに関するNDCスタック全体をログ出力に組み込むことになります。これは、ユーザーの介入なしに実行されます。ユーザーが行わなければならないのは、コード中のいくつかの明確な場所で`push()` メソッドと`pop()` メソッドを使用して、NDCに正しい情報を入れることだけです。対照的に、クライアントごとにカテゴリーを設ける方法を採用するなら、コードに大規模な変更を加えなければならなくなります。

この点を例証するために、無数のクライアントにコンテンツを送信するサーブレットの例を考えてみましょう。サーブレットは、他のコードを実行する前、要求の開始の時点でNDCを作成できます。コンテキスト情報は、クライアントのホスト名や、要求に固有の他の情報とすることができます。一般的なのはCookieに含まれている情報です。したがって、サーブレットが同時に複数のクライアントにサービスを提供するとしても、同じコードによって開始される(つまり同じカテゴリーに属する) ログを区別することは依然として可能です。各クライアント要求には、それぞれ異なるNDCスタックがあるからです。新たにインスタンス生成されたカテゴリーを、クライアントの要求の処理中に実行されるすべてのコードに渡すという複雑な処理と、この技法とを比較してみてください。

とはいえ、仮想ホスティングWebサーバーなどの複雑なアプリケーションの中には、仮想ホストのコンテキストや要求を発行するソフトウェア・コンポーネントに応じて、異なる方法でログを記録する必要のあるものもあります。最新リリースのlog4jでは、複数の階層木がサポートされています。この拡張機能により、各仮想ホストはカテゴリー階層のコピーを個別に所有することができます。

パフォーマンス

しばしば引き合いに出されるロギングへの反対論の1つは、その計算コストに由来しています。これは、道理にかなったことです。中規模のサイズのアプリケーションでさえ、数千のログ要求を生成するからです。ロギングのパフォーマンスの計測と微調整のために、多大の努力が傾けられてきました。log4jは高速かつ柔軟であると主張されています。速度が第一であり、柔軟性は第二です。

とはいえ、ユーザーは以下のパフォーマンス上の問題に注意してください。

1. ロギングをオフにした場合のロギングのパフォーマンス
   ロギングを全体に渡って、あるいは一連の優先度に対してオフにした場合、ログ要求のコストは、メソッドの起動に整数の比較を加えたものとなります。メソッドの起動には、パラメーターの組み立てという隠れたコストが含まれます。
   たとえば、`cat` というカテゴリーに関して次のように書くとします。
   ```
   cat.debug("Entry number: " + i + " is " + String.valueOf(entry[i]));
   ```
   この場合、メッセージ・パラメーターの組み立てのコストが必要です。メッセージのログが記録されるかどうかにかかわりなく、整数`i` と`entry[i]` をストリングに変換し、中間のストリングを連結することになります。
   速度が心配なら、次のように書いてください。
   ```
   if(cat.isDebugEnabled() {
            cat.debug("Entry number: " + i + " is " + String.valueOf(entry[i]));
         }
   ```
   このように書けば、デバッグを使用不可にした場合に、パラメーター組み立てのコストがかかりません。一方、カテゴリーのデバッグを使用可能にした場合、カテゴリーが使用可能か使用不可かを評価するために2倍のコストがかかることになります。すなわち、`debugEnabled()` のコストと`debug()` のコストです。このオーバーヘッドはわずかなものです。カテゴリーの評価に必要な時間は、ログの記録自体に必要な時間の約1%であるからです。

log4jでは、ロギング要求は`Category`クラスのインスタンスに対して行われます。`Category`はクラスであって、インターフェースではないため、メソッド起動のコストはおそらく低くなりますが、柔軟性は犠牲になります。
233 MHzのPentium IIマシンでは、ロギングをオフにした場合、ロギングのコストは一般に5～50ナノ秒(ns) の範囲です。

2. **ロギングがオンになっている場合の、ログを記録するかどうかの決定のパフォーマンス**
このシナリオの中心的な問題となるのは、カテゴリー階層を調べて回ることのパフォーマンス・コストです。ログがオンになっていても、log4jは依然としてログ要求の優先度と要求カテゴリーの優先度を比較する必要があります。しかし、カテゴリーが、割り当てられた優先度を持たない場合もあります。この場合、カテゴリー階層から優先度を継承できます。したがって、優先度を発見するために、カテゴリーは自分の祖先を探索する必要があるかもしれません。
階層を調べて回るこの処理をできるだけ速く行うために、真剣な努力が傾けられてきました。たとえば、子カテゴリーは自分の既存の祖先にだけリンクします。上述の`BasicConfigurator`の例では、`com.foo.Bar`というカテゴリーは直接ルート・カテゴリーにリンクして、存在していない`com`または`com.foo`を回避しています。これにより、特にまばらな階層の場合、調べて回る速度が大幅に改善されます。
再び233 MHzのPentium IIマシンを例にとると、階層を調べて回るコストは一般に5～15マイクロ秒の範囲です。
3. **ロギング自体**
もう1つのパフォーマンスの問題は、ログ出力をフォーマットして、宛先にこれを送信するコストに由来します。この分野においても、レイアウト(フォーマッター) の実行速度をできるだけ上げるための真剣な努力が傾けられました。これはappenderにも当てはまります。ロギング自体のコストは、一般に100～300マイクロ秒の範囲です。

log4jには多数の機能がありますが、第一の設計目標は速度でした。log4jのコンポーネントの一部は、パフォーマンスを改善するために何回も書き直されてきました。それにもかかわらず、コントリビューターは新しい最適化の手法を頻繁に考え付きます。

Sunがまもなく発表するロギングAPI

SunはJava用のロギングAPIを定義するために、Java Community Process (JCP)でJSR47を開始しました。完成したAPIはJDKバージョン1.4を必要とする予定であり、まもなく公開レビュー用の準備ができるはずです。

JSR47 APIとlog4jは、アーキテクチャーのレベルで大変よく類似しています。JSR47APIは階層的な名前空間をサポートしています。これは、log4jの中心的な機能の1つです。一方、log4jには、JSR47にはない便利な機能が多数備わっています。log4jに勢いがあることを考えると、わたしの党派的な意見では、JSR47はその出発の時点までに時代遅れになっているように思われます。log4jを作成しているのは、閉鎖的な委員会ではなく、実際にそれを使っている人々です。

ついでながら、log4jは、JSR47 APIが使用可能になるときに備えて、このAPIとのブリッジングをすでにサポートしています。

IBM alphaWorksから入手可能な、もう1つのロギングAPI、JLogについても言及しておきます。IBMが名前変更するまで、JLogはRASToolkitと呼ばれていました。alphaWorksサイトで、JLogには

"Logging Toolkitfor Java" というラベルが付けられています。これは、log4jがhttp://log4j.orgに移動する前、alphaWorksで付けられていたのとほとんど同じラベルです。JLogは優れたロギング・パッケージですが、log4jと混同しないようにしてください。

結論

log4jはJavaで書かれたポピュラーなロギング・パッケージです。その独特な機能の1つとして、カテゴリーの継承の概念があります。カテゴリー階層を使用することにより、どのログ・ステートメントを出力するかを任意の細分度で制御できます。これにより、ログ出力のボリュームの削減とロギングのコストの最小化が可能です。

log4j APIの利点の1つは、管理が容易であることです。いったんコードにログ・ステートメントを挿入すれば、設定ファイルを使ってそれらのステートメントを制御できます。その上、それらのログ・ステートメントを選択的に有効または無効にできますし、複数のさまざまな出力ターゲットに、ユーザーが選んだフォーマットで送信できます。さらに、log4jパッケージは、出荷するコードにログ・ステートメントを残しておいても、あまりパフォーマンス・コストを必要としないように設計されています。

log4jは努力の結集の結果です。プロジェクトに貢献してこられた作成者の皆さんに特別の感謝をお伝えします。このパッケージの最良の特徴の数々は、すべて例外なく、ユーザー・コミュニティーに源を発しています。

関連トピック

- "Patterns for Logging Diagnostic Messages" (Neil Harrison)。PatternLanguages of Program Design 3 (R. Martin、D. Riehle、F. Buschmann編: Addison-Wesley,1997) 所収。
- Log4jプロジェクト・ホーム・ページ
- SunのJava Logging API Specification
- IBMのJLog API
- EUのSecure Electronic Marketplace for Europe (SEMPER) プロジェクト・ホーム・ページ
- OpenSource.org