効果的なログ習慣がエンタープライズ開発を楽にする

最初にログ計画を立て、その成果を後の開発プロセスの中で得る

Charles Chan
Senior Software Developer
Finetix LLC

2005年 8月 09日

エンタープライズ開発では、途中に様々な障害があることは避けられません。開発プロセスの、後の段階で効果的にバグ取りをしたいのであれば、効果的なログ戦略が必要です。しかし、エンタープライズ・アプリケーションで効果的なログを行うためには、よく練った計画と、訓練が必要です。この記事では、コンサルタントであるCharles Chanが、プロジェクトの最初から効果的なログ・コードを書くためのベスト・プラクティスを紹介します。

皆さんが開発者であれば、恐らく次のような経験があるはずです。あなたはコードとテスト・ケースを開発しました。アプリケーションは厳しいQAテストを通り、そのコードがビジネス要求を満足することに、あなたは自信を持っています。ところが、そのアプリケーションが最終的にエンド・ユーザーの手に渡ると、予期しない問題が起こってしまうのです。適切なログ・メッセージがない場合には、こうした問題を診断するために何日も要するかも知れません。残念なことに、大部分のプロジェクトでは、ログに関する明確な戦略を持っていません。そうした戦略がないと、システムが生成するログ・メッセージが、問題解決の役に立たないかも知れないのです。この記事では、エンタープライズ・アプリケーションのためのログに関して、様々な面から議論します。Java™プラットフォームでのロギングに利用できるAPIについての概要を取り上げ、またロギング用のコードを書くためのベスト・プラクティス、実稼働環境で詳細なログが必要になった場合の対処方法などについて学んで行きます。

ロギングAPIを選ぶ

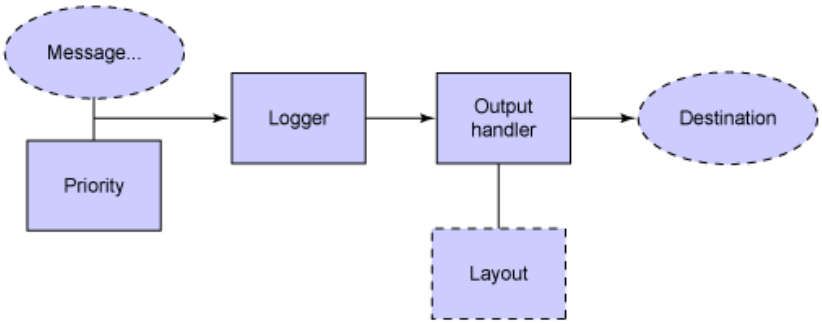
Javaプラットフォームを利用して開発を行う場合、ロギング用APIとして、大きく2つの選択肢があります。Apache Log4Jと、Javaプラットフォームのバージョン1.4以上に付属するJava Logging APIです。Log4Jは、Java Logging APIよりも成熟しており、機能も豊富です。どちらのロギング実装も、設計パターンは似ています（図1を見てください）。皆さんの会社がサードパーティーのライブラリーを使うことを制限している場合を除いて、私としてはLog4Jを使うように強くお勧めします。どちらのAPIを使うべきか決めかねる人は、基礎となるロギング実装がどんなものかによらず、そのラッパーとして動作する、Apache Commons Logging APIを使うという選択肢もあります。これを使えば、理論的には、コードを変更せずにロギング実装を切り換えることができます。ただし実際には、ロギング実装を変更する必要はめったにありません。つまりApache Commons Logging

APIを使っても、必要な機能が追加されるわけではなく、複雑さが増すだけなのです。ですから、私としてはお勧めしません。

ロギングの概要

Log4JもJava Logging APIも、設計や使い方のパターンは似ています（図1とリスト1を見てください）。最初にメッセージが作成され、特定な優先度をつけられて、ロガー・オブジェクトに渡されます。次に、こうしたメッセージの行く先とフォーマットが、出力ハンドラーとそのレイアウトによって決定されます。

図1. ロギング実装の主なコンポーネント



リスト1. ロガー・オブジェクトをインスタンス化して使用する

```
import org.apache.log4j.Logger;

public class MyClass {
  /*
   * Obtain a logger for a message category. In this case, the message
   * category is  the fully qualified class name of MyClass.
   */
  private static final Logger logger
   = Logger.getLogger(MyClass.class.getName());
  ...
  public void myMethod() {
    ...
    if (logger.isDebugEnabled()) {
      logger.debug("Executing with parameters:
       " + param1 + ":" + param2);
    }
  }
}
```

適切なロギング実装には、様々な出力ハンドラーが付属しています。その中で最も一般的なものは、ファイル出力ハンドラーとコンソール出力ハンドラーです。Log4Jにも、JMSトピックにメッセージを公開するためのハンドラーや、データベース・テーブルにメッセージを挿入するためのハンドラーが提供されています。カスタムのアペンダー（appender）を書くのは難しくはありませんが、コードを書き、維持管理するためのコストは軽く見るべきではありません。メッセージのフォーマットも、Layoutオブジェクトを使って構成することができます。最も一般的なレイアウト・オブジェクトは、提供されたパターンに従ってメッセージをフォーマットする、PatternLayoutです。

リスト2は、FileAppenderのコンフィギュレーションを行う、Log4Jコンフィギュレーション・ファイルの例です。このコンフィギュレーションでは、com.ambrosesoft.log.MyClassカテゴリーの下にあるエラー・メッセージはFileAppenderに送られ、それを今度はFileAppenderが、log.txtと呼ばれ

るファイルに書き込みます。これらのメッセージは、このアペンダーに関連付けられたレイアウト（この場合はPatternLayout）に従ってフォーマットされます。

リスト2. Log4J XMLコンフィギュレーション・ファイルの例

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">

  <appender name="fileAppender"
   class="org.apache.log4j.FileAppender">
    <param name="File" value="log.txt"/>
    <param name="Append" value="true"/>
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern"
        value="%d [%t] %p - %m%n"/>
    </layout>
  </appender>

  <category name="com.ambrosesoft.log.MyClass">
    <priority value="error"/>
    <appender-ref ref="fileAppender"/>
  </category>

  <root>
    <priority value="debug"/>
    <appender-ref ref="fileAppender"/>
  </root>

</log4j:configuration>
```

ロギングのベスト・プラクティス

ロギングに関する選択の中で最も重要なものは、各ログ・メッセージを、特定な『カテゴリー』に割り当てるための仕組みを決めることです。通常は、各クラスの完全修飾名を使います（これらのクラスのアクティビティーは、（リスト1の場合のように）メッセージ・カテゴリーとしてログされます）。これは、開発者が各クラスのログ設定を微調整できるためです。ただし、これがうまく行くのは、ログ・メッセージが実行トレース用に作られている場合のみです。エンタープライズ・アプリケーションでは、他にも様々な種類のログ・メッセージがあります。例えば、あるログ・メッセージはセキュリティー・アドバイザーのためのものかも知れず、あるログ・メッセージは、パフォーマンス調整のために作られたものかも知れません。もし、この両方のメッセージが同じクラスに関わり、従って同じカテゴリーに割り当てられてしまうと、ログ出力で両者を判別することが難しくなります。

この問題を避けるためには、アプリケーションが、明確なカテゴリーで分けられた専用のロガー・セットを持つべきなのです。これをリスト3に示します。これらのロガーは、それぞれ独自の優先度と出力ハンドラーを持つように構成することができます。例えばセキュリティー・ロガーは、メッセージが行き先に書き込まれる前に、メッセージを暗号化することができます。アプリケーションの設計者は、メッセージを利用する側（例えばセキュリティー・アドバイザーなど）と一緒に時々ログ出力を調べ、メッセージを適切に調整すべきでしょう。

リスト3. 特定な目的のためのロガー

```
import org.apache.log4j.Logger;

public interface Loggers {
  Logger performance = Logger.getLogger("performance");
  Logger security = Logger.getLogger("security");
  Logger business = Logger.getLogger("business");
}
...
public class MyClass {
  ....
  if (Loggers.security.isWarnEnabled()) {
    Loggers.security.warn
     ("Access denied: Username [" + userName + "] ...");
  }
  ...
}
```

ロギング・レベルを選択する

単一のカテゴリー（例えばセキュリティー）に収まるメッセージであっても、様々な『優先度』を持つことがあります。あるメッセージはデバッグ用に生成され、あるメッセージは警告用に生成され、あるメッセージはエラー用に生成されます。メッセージの持つ、こうした優先度は、ロギングの『レベル』で表されます。最も一般的なロギング・レベルは、次のようなものです。

- **デバッグ**: このレベルのメッセージは、大量のコンテキスト情報を含みます。これらのメッセージの大部分は、問題の診断用に使われます。
- **情報**: これらのメッセージは、実稼働環境での実行トレース（キメの粗いレベルでの）を補助するための、ある種のコンテキスト情報を含んでいます。
- **警告**: 警告メッセージは、システム中に潜在的な問題があることを示します。例えば、メッセージ・カテゴリーがセキュリティーに関するものであれば、もし辞書攻撃が検出された場合には、警告メッセージを作成する必要があります。
- **エラー**: エラー・メッセージは、システム中に深刻な問題があることを示します。通常、こうした問題は回復不能であり、人の介在を必要とします。

Java Logging APIもApache Log4Jも、こうした基本レベル以上のロギング・レベルを提供しています。ロギング・レベルの主な目的は、ノイズの中から必要な情報を拾い出すことです。誤ったロギング・レベルを使ってしまい、その結果、ログ・メッセージ本来の有効さを損なうことがないよう、開発者がコーディングを開始する前に、開発者に対して明確なガイドラインを示す必要があります。

ログ・メッセージ・フォーマット

ロガーを選択し、ロギング・レベルを確立できたら、今度はログ・メッセージを構築します。この場合、できるだけ多くのコンテキスト情報、つまりユーザーが提供する情報や、アプリケーションの状態情報など、を含めることが重要です。オブジェクトをログするための1つの方法は、オブジェクトをXMLに変換してしまうことです。XStream（参考文献）などのサードパーティー・ライブラリーは、イントロスペクション（introspection）によって、Javaオブジェクトを自動的にXMLに変換することができます。これは非常に強力な機構ですが、スピードと冗長性とのトレードオフを考慮する必要があります。典型的な、アプリケーションの状態情報に加えて、次のような情報もログする必要があります。

- スレッドID: エンタープライズ・アプリケーションは通常、マルチスレッド環境の中で実行されます。スレッドID情報によって、ある一つのリクエストと別のリクエストを区別することができます。
- 呼び出し側ID: 呼び出し側のID（あるいはプリンシパル（principal））も、重要な情報です。ユーザーはそれぞれ異なる特権を持っているため、実行パスが非常に異なる可能性があります。ユーザーのIDをログ・メッセージに入れておくと、セキュリティーに対応したアプリケーションでは非常に役立ちます。
- タイムスタンプ: 一般的に言ってユーザーは、問題が起きた時の時間を、おおよそでしか特定できません。従ってタイムスタンプがないと、サポートの人が問題を特定することが困難になります。
- ソースコード情報: これにはクラス名、メソッド名、行番号などを含みます。セキュリティーが気になる場合を除いて、実稼働用にコンパイルする時にも、デバッグ・フラグ（-g）を常にオンにしておくことをお勧めします。デバッグ・フラグがないと、Javaコンパイラーは行番号情報を全て削除するため、ログ・メッセージの有効さが大幅に損なわれてしまいます。

（呼び出し側のIDを除いて）、上記の情報は、ロギング実装によって自動的に検索されます。これらの情報をメッセージの中に含めるためには、出力ハンドラーに対して適当なレイアウト・パターンを構成すればよいだけです。呼び出し側のIDを捉えるためには、Log4Jの診断コンテキスト機能（参考文献に情報があります）を利用することができます。診断コンテキストによって、コンテキスト情報を現在実行中のスレッドに関連付けることができます。そうすれば、出力用にメッセージをフォーマットする際に、この情報が全メッセージに含まれるようになります。

J2EE Webアプリケーションにおいて、ユーザーのIDを診断コンテキストの中に保存するためのロジックの置き場所としては、サーブレット・フィルターの中が最適です。リスト4は、これを実現するために必要なコードの中心部分を示しています。このコードでは、Log4J 1.3アルファにある、マップ診断コンテキスト・クラス（mapped diagnostic context class, MDC）を使っています。Log4J 1.2の、ネスト診断コンテキスト（nested diagnostic context, NDC）を使っても、同じことができます。サーブレット・フィルターに関する一般的な情報については、参考文献の項を見てください。

リスト4. サーブレット・フィルターの診断コンテキストを使う

```
import javax.servlet.Filter;
...
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;
import org.apache.log4j.MDC;

public class LoggerFilter implements Filter {

    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {

        // Retrieves the session object from the current request.
        HttpSession session = ((HttpServletRequest)request).getSession();

        // Put the username into the diagnostic context.
        // Use %X{username} in the layout pattern to
        // include this information.
        MDC.put("username",
         session.getAttribute("username"));

        // Continue processing the rest of the filter chain.
```

```
        chain.doFilter(request, response);

        // Remove the username from the diagnostic context.
        MDC.remove("username");
    }
    ...
}
```

AspectJで実行トレースを行う

問題の診断を行う際には、プログラムの実行をトレースできると便利な場合が多いものです。メソッドへの入り口やメソッド出口など、プログラム実行の中の様々なポイントにおいて、一貫したログ・ステートメントを得るにはどうすべきなのでしょう。これは昔からの問題であり、AspectJが登場するまでは、良い答えがありませんでした。ところがAspectJを利用すると、アプリケーションの様々なポイントにおいて、コード断片を実行できるのです。AspectJでは、こうしたポイントをポイント・カット（point cut）と呼び、ポイント・カットにおいて実行するコードはアドバイス（advice）と呼びます。また、ポイント・カットとアドバイスの組み合わせは、アスペクト（aspect）と呼びます。

AspectJが素晴らしいのは、アスペクトをアプリケーション全体に適用することが、大した手間をかけずにできるという点です。AspectJに関して、さらに詳しくは参考文献を見てください。リスト5は、メソッドへの入り口と出口でロギングをイネーブルにするための、AspectJソース・ファイルを示しています。この例では、com.ambrosesoftパッケージのパブリック・メソッドへの出入りがある毎に、トレース・ロガーがメッセージをログします。

リスト5. AspectJを使って、メソッドへの入出力をログする

```
import org.apache.log4j.Logger;
import java.lang.reflect.Field;

public aspect AutoTrace {

    private static final Logger logger
     = Logger.getLogger(AutoTrace.class);

    pointcut publicMethods() :
     execution(public * com.ambrosesoft..*(..));

    pointcut loggableCalls() : publicMethods();

    /**
     * Inspect the class and find its logger object. If none is found,
     * use the one defined here.
     */
    private Logger getLogger(org.aspectj.lang.JoinPoint joinPoint) {
        try {
            /*
             * Try to discover the logger object.
             * The logger object must be a static field called logger.
             */
            Class declaringType
            = joinPoint.getSignature().getDeclaringType();
            Field loggerField
            = declaringType.getField("logger");
            loggerField.setAccessible(true);
            return (Logger)loggerField.get(null);
        } catch(NoSuchFieldException e) {
            /*
             * Cannot find a logger object, use the internal one.
             */
```

```
            return logger;
        } catch(Exception e) {
            throw new RuntimeException(e);
        }
    }

    /**
     * An aspect to log method entry.
     */
    before() : loggableCalls(){
        getLogger(thisJoinPoint).debug("Entering.."
        + thisJoinPoint.getSignature().toString());
    }

    /**
     * An aspect to log method exit.
     */
    after() : loggableCalls(){
        getLogger(thisJoinPoint).debug("Exiting.."
        + thisJoinPoint.getSignature().toString());
    }
}
```

実稼働環境でのロギング

アプリケーションがいったん実稼働に入ると、通常はランタイムのパフォーマンスを最適化するために、デバッグ・ログや情報ログのメッセージをオフします。しかし、何か悪いことが発生し、開発環境で問題を再現できない場合には、実稼働状態でデバッグ・メッセージをオンする必要があるかも知れません。こうした場合、サーバーを停止させずにログ設定を変更できることが重要です。実稼働状態での問題診断には、何日とまでは言わないまでも、何時間かの注意深い検査が必要です。ログ設定を変更する度に実稼働アプリケーションを再起動していたのでは、すぐに身動きの取れない状態になってしまいます。

幸いLog4Jには、この問題に対処するための簡単な機構が用意されています。Log4J 1.2では、DOMConfiguratorの中のメソッド、configureAndWatch() がLog4Jのコンフィギュレーションを行い、ログ・コンフィギュレーション・ファイルの中の変化を自動的に監視するのです。リスト6は、これを説明しています。（ただし、現在アルファであるLog4J 1.3では、DOMConfiguratorが強く非難を浴びています。今後は、より柔軟な実装、JoranConfiguratorで置き換えられる予定なので注意してください。）

Log4Jが初期化される前にconfigureAndWatch() が確実に呼ばれるようにするには、スタートアップ・クラスの中でconfigureAndWatch()を呼び出します。スタートアップ・コードを実行するための機構は、アプリケーション・サーバーによって異なります（参考文献）。詳細に関しては、皆さんのアプリケーション・サーバーの実装を調べてください。アプリケーション・サーバーによっては、Log4Jライブラリーをサーバーのクラスパスに置くように要求するかも知れません。ログ・コンフィギュレーション・ファイルは、サポートの人がアクセスできる場所に保存する必要があります。

リスト6. DOMConfiguratorでLog4Jをコンフィギュレーションする

```
/*
 * Configure Log4J library and periodically
 * monitor log4j.xml for any update.
 */
DOMConfigurator.configureAndWatch
 ("/apps/config/log4j.xml");
```

ログ・コンフィギュレーション・ファイルに対して簡単にアクセスできない場合（例えば実稼働環境が別の組織によって維持管理されている場合など）には、別の方法を使う必要があります。標準的な方法としては、アプリケーションの設定管理のための標準APIを提供している、JMXを使います。最近のJMX準拠サーバーでは、アプリケーション・サーバーの管理コンソールの機能を、MBean（managed bean）で拡張することができます。（JMXの使い方や、WebSphere Application Server 6.0でのJMXの使い方に関しては、参考文献の項を見てください。）ただし、JMXを使う方法は非常に複雑なため、これを使うのは、本当に必要な状況だけにとどめるべきでしょう。

機密性を持つデータのロギング

実稼働環境でロギングを行うには、技術的な問題の他に、ビジネス上でも克服すべき問題があります。例えば、機密性を持つデータのロギングには、セキュリティーのリスクが伴います。ユーザー名やパスワードは、単純テキストでもログできてしまいます。他にも、eメール・アドレスや電話番号、アカウント情報など、保護すべき機密情報があります。こうした情報をログする際に、必ずマスキングするようにすることは、セキュリティー・アドバイザーと設計者の責任です。機密性を持つメッセージのログにはセキュリティー専用のロガーを使うことも、リスクを減らす上で有効です。このロガーを、特別なアペンダーを使って構成し、暗号化フォーマットでメッセージを保存したり、セキュアな場所に保存したりすることもできます。しかし一番良いのは、プロジェクトのスタート前に適切なコーディング・ガイドラインを設定し、コード・レビューの際にも、そのガイドラインを強制してセキュリティー・リスクを回避するようにすることです。

例外を判断する

予期せぬ例外が起きた場合、例えばデータベース・コネクションが突然切れる、あるいはシステム・リソースが不足しかけているといった場合には、適切な処理を行わないと、問題診断に役立つ重要な情報を失ってしまうかも知れません。まず何よりも、例外と、そのスタック・トレースをログする必要があります。第2に、エンド・ユーザーにもサポート・チームにも便利な、ユーザーに分かりやすいエラー・ページを表示すべきです。

サポートを求める問い合わせを受けたサポート・チームが直面する困難の1つは、ユーザーがレポートする問題と、具体的にログされた例外とを関連付けることが難しい、という点です。こうした場合、非常に簡単かつ有効なのは、例外それぞれに対して固有のIDをログすることです。このIDを、ユーザーに対して提示するか、あるいは、ユーザーが書き込むレポート・フォームに含めるようにします。そうすることで、サポート・チームは類推する必要が無くなり、問題状況に対して迅速に対応できます。また、IDが分かりにくくならないように、時々IDをリサイクルすることも考慮すべきでしょう。

ログ・ファイルの管理

忙しいアプリケーションであれば、ログ・ファイルはすぐに巨大になります。巨大なログ・ファイルでは、必要なシグナルを取り出すために大量のノイズをフィルターにかける必要があるため、扱いが面倒です。この問題を避けるために一般的に使われるのが、『ログ・ローテーション』と呼ばれる方法です。ログ・ローテーションでは、古いログ・メッセージを定期的にアーカ

イブし、新しいメッセージは、常に比較的小さなファイルに書き込むようにするのです。ログ・メッセージが有効である期間は、ごく短い間です。1週間以上古いログ・メッセージを参照しなければななら い場合は、極めて稀です。Log4J 1.2では、DailyRollingFileAppenderアペンダーが、与えられた日付パターンに基づいてログ・ファイルをローテーションします。（Log4J 1.3では、このローリング・ファイル・アペンダーが再設計されており、ローリングの実行方法をポリシーで制御できるようになっています。例えばTimeBasedRollingPolicyは、日付と時間に基づくローテーションを定義します。）リスト7は、毎日夜12時に、Log4Jにログ・ファイルをローテーションさせるための、コンフィギュレーションの断片です。

リスト7. DailyRollingFileAppenderを使ってログ・ファイルをローテーションさせる

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">

  <appender name="fileAppender" class
   ="org.apache.log4j.DailyRollingFileAppender">
    <param name="File" value="log.txt"/>
    <param name="Append" value="true"/>
    <param name="DatePattern"
     value="'.'yyyy-MM-dd"/>
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern"
       value="%d [%t] %p - %m%n"/>
    </layout>
  </appender>
...
</log4j:configuration>
```

クラスター環境でのロギング

クラスター環境、あるいは分散環境に展開されるエンタープライズ・アプリケーションが、次第に多くなっています。しかし、クラスター環境では、様々なソース（通常は別のマシン）からメッセージが生成されるため、ロギングには一層綿密な計画が必要です。別々のマシンでロギングが行われる場合には、各マシンのタイムスタンプを同期しないと、ログ・メッセージの順序が狂ってしまいます。マシン間のクロックを簡単に同期させるためには、タイム・サーバーを使います。こうしたタイム・サーバーを設定するには、2つの方法があります。1つは、内部用のマシンの1台をタイム・サーバーにする方法です。他のマシン群は、このタイム・サーバーのタイムスタンプと、自分たちのタイムスタンプを、NPT（network time protocol）を使って同期させるのです。もう1つの方法は、インターネット上で利用できるタイム・サーバーを使用する方法です（参考文献）。AIXでは、xntpdデーモンを使ってマシン間のシステム時間を同期させています。いったん全マシンが同じ時間を持つようになれば、それらのマシンのログ・メッセージを一緒に解析できるようになります。

クラスター環境でログ・メッセージを集める場合には、困難も伴います。この環境で簡単にログ・メッセージを保存するには、ホスト専用のログ・ファイルの中に保存することです。クラスターが、セッション親和性（session affinity）を持って構成されている場合、つまり、ある特定なユーザー・セッションが常に同じサーバーで行われ、EJBがローカルにデプロイされている場合には、これがうまく行きます。こうした構成では、クラスター中のマシンが生成したログ・ファ

イルは、独立に解析されます。それ以外の場合では、つまり、あるリクエストが、潜在的に複数マシンで処理される可能性のある場合には、ログ・メッセージは別々のログ・ファイルにあるかも知れず、解析は一層難しくなります。こうした場合には、IBMのTivoli®ソフトウェア（参考文献にリンクがあります）などのシステム管理ソフトウェアを使ってログ・メッセージを管理した方が賢明です。こうしたソフトウェアであれば、全ログ・メッセージ（システム管理ソフトウェアの用語では『イベント』）に対する統合ビューを提供してくれるため、管理が容易になるのです。またシステム管理ソフトウェアは、受信するイベントのタイプに基づいて、アクション（e メール・メッセージやポケベル・メッセージを送るなど）をトリガーすることもできます。

まとめ

この記事では、ロギング戦略を立てる際に何を考慮すべきかを解説しました。プログラミングでは常にそうですが、よく練った計画を最初に立てておく方が、作業を進めながら計画して行くよりも、長期的には大幅な作業削減につながるものです。的確なロギング戦略を立てることによって、問題の診断が非常に楽になります。それが、より良いアプリケーションと、サポート・チームからの迅速な対応につながり、究極的にエンド・ユーザーの利益につながるのです。

著者について

Charles Chan

Charles Chanは、カナダのトロントで活躍する独立のコンサルタントです。彼の関心領域は、分散システムやハイ・パフォーマンス・コンピューティング、国際化、ソフトウェア設計パターンなど多岐に渡っています。時間のある時には、オープンソース・コミュニティーに貢献しています。