汎用ログ出力機能

■ 概要

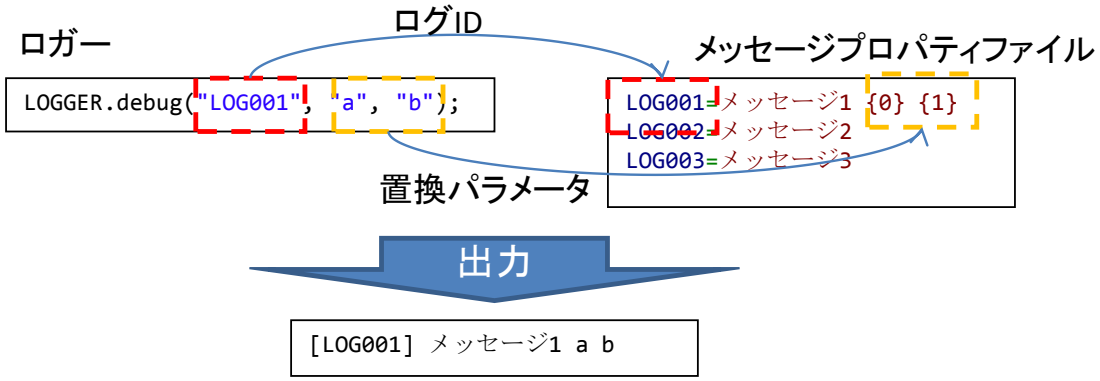
◆ 機能概要

- ログメッセージ出力機能
  1. 出力ログメッセージをプロパティファイルで管理できます。
  2. ログ出力メソッドをログ ID と置換パラメータ(可変長配列)で呼び出せます。
  3. 出力ログメッセージの先頭にログ ID を付加できます。
  4. モジュール毎にログメッセージを管理
- 例外メッセージ出力機能
  1. 例外メッセージをプロパティファイルで管理できます。
  2. 例外コンストラクタをメッセージ ID と置換パラメータ(可変長配列)で呼び出せます。
  3. 例外メッセージの先頭にメッセージ ID を付加できます。

◆ 概念図

以下にログメッセージ出力機能の概要図を示します。汎用例外メッセージ出力機能に関しても同様な仕組みです(例外メッセージ出力機能の概要図は割愛します)。

- ログメッセージ出力機能



エラー! プロパティ名に誤りがあります。

◆ 解説

- ログメッセージ出力機能
  1. アプリケーションはロガーのログ出力メソッドにログ ID と置換パラメータを渡します。置換パラメータは可変長配列であり、空でも構いません。
  2. ロガーは、設定ファイルに記述されたメッセージプロパティファイルからログ ID に該当するメッセージを取得し、置換パラメータを埋め込んで出力メッセージを作成します。次に、メッセージの先頭にログ ID を付加し、メッセージを出力します。

■ 使用方法

◆ コーディングポイント

- ログメッセージ出力機能
  - ➢ ロガー取得
    ロガーの取得は Commons-Logging や Log4J 等他のロギングライブラリとほぼ同じです。

```
private static final TLogger LOGGER = TLogger.getLogger(XXX.class);
```

または

```
private static final TLogger LOGGER = TLogger.getLogger("XXX");
```

で取得できます。前者はクラス名の FQCN が、後者は引数の文字列がカテゴリ名になります。

  - ➢ ログ出力
    次のようなログメッセージプロパティファイルがある場合、

```
DEB001=debug message
ERR001=error message
```

このメッセージを出力するには

```
LOGGER.debug("DEB001");
LOGGER.error("ERR001");
```

のようにログレベルに応じたメソッドにログ ID を渡して実行します。 出力メッセージは

```
[DEB001] debug message
[ERR001] error message
```

となります（実際に出力されるログメッセージ全文はログフォーマットにより異なります)。ログレベルは

- FATAL
- ERROR
- WARN
- INFO
- DEBUG

■ TRACE
があります。log(String logId)メソッドを使用すると、ログ ID の一文字目を見てログレベルを判断します。

```
LOGGER.log("DEB001"); // DEBUGメッセージ
LOGGER.log("ERR001"); // ERROR メッセージ
```

➢ パラメータ置換
出力するログメッセージを作成する際に、デフォルトで java.text.MessageFormat を使用しています。置換パラメータを可変長配列で渡すことができます。変更方法は「拡張ポイント」参照。

```
DEB002={0} is {1}.
```

という定義がある場合、

```
LOGGER.debug("DEB002", "foo", "bar");
```

を実行すると出力メッセージは

```
[DEB002] foo is bar.
```

となります。

内部でメッセージ文字列を作成する際にログレベルのチェックを行っているので、

```
if (LOGGER.isDebugEnabled()) {
    LOGGER.debug("DEB002", "foo", "bar");
}
```

という if 文を書く必要がありません。(ただし、パラメータを作成する際にメソッドを呼び出している場合は if 文を書いて明示的にログレベルチェックを行ってください。)

➢ 例外のスタックトレース出力
ログメッセージの後に例外のスタックトレースを出力したい場合、以下のように起因例外を**置換パラメータの前**に設定してください。置換パラメータの後に設定すると置換パラメータの一部になってしまうため注意してください。

```
try {
    // 処理
} catch (Exception e) {
    LOGGER.warn("WAR001", e, "xxx", "yyy"); // WARNログを出力して処理
続行
}
```

- メッセージプロパティファイルにないメッセージの出力

  メッセージプロパティファイル中のログ ID を渡す他に、直接メッセージを渡す方法があります。第 1 引数に false を設定し、第 2 引数にメッセージ本文を直接記述できます。第 3 引数以降は置換パラメータです。この場合、ログ ID は出力されません。

```
LOGGER.warn(false, "warn!!");
LOGGER.info(false, "Hello {0}!", "World");
```

  を実行した場合、
  出力メッセージは

```
warn!!
Hello World!
```

  となります。簡単なテストやデバッグ時に便利ですが、基本的にはログメッセージはプロパティファイルで管理することを推奨します。

- 設定ファイル
  TLogger クラスを使用する場合、**クラスパス直下の META-INF ディレクトリに terasoluna-logger.properties を作成してください(必須)**。

  設定項目一覧は以下です。

項番	キー名	設定内容	デフォルト値
1	message.basename	読み込むメッセージプロパティファイルのベースネーム	なし
2	message.id.format	出力するメッセージ ID のフォーマット	[%s]
3	throw.if.resource.not.found	メッセージ ID が見つからなかった場合に例外をスローするか否か	false
4	message.formatter.fqcn	メッセージのフォーマット処理を行うクラスの FQCN	jp.terasoluna.fw.message.DefaultMessageFormatter

✧ メッセージプロパティファイルのベースネーム設定
terasoluna-logger.properties の message.basename キーにメッセージプロパティファイルのベースネームをクラスパス相対(FQCN)で設定してください。

```
message.basename=foo
```

と書くとクラスパス直下の foo.properties が読み込まれます。

```
message.basename=foo1,foo2,foo3
```

のように半角カンマ区切りで設定すると foo1.properties、foo2.properties、foo3.properties を全て読み込みます。

META-INF/terasoluna-logger.properies の message.basename はモジュール毎に設定できます。
ロガーは全てのモジュール(jar)が梱包している META-INF/terasoluna-logger.properie に設定されている message.basename の値をマージしてメッセージを取得します。
これにより、**モジュール毎にログメッセージを管理することができます**。

✧ 出力ログ ID フォーマット設定

ログ出力時に自動で付加されるログ ID のフォーマットを設定できます。
message.id.format キーに java.lang.String.format()のフォーマット形式で設定してください。ログ ID が文字列として渡されます。
設定しない場合は「[%s]」がデフォルト値として使用されます。

```
message.id.format=[%-8s]
```

のように設定すると、モジュール間で異なる長さのログ ID を左寄せで揃えて出力できます。
この設定値はモジュール毎に管理することはできません。
クラスローダの読み込み優先度が一番高い META-INF/terasoluna-logger.properties の内容が反映されます。
(通常、アプリ側の設定となります。)

- 例外メッセージ出力機能

ロガーとほぼ同様です。

➢ 例外クラス作成
メッセージファイルに

エラー! プロパティ名に誤りがあります。

```
ERR01= {0} was occured!
```

と設定されている場合、

```
TException e = new TException("ERR01", "something");
```

と例外を作成すると、この例外オブジェクトの例外メッセージは

```
[ERR01] somethins was occured!
```

になります。
TException はチェック例外、TRuntimeException は非チェック例外(実行時例外)です。

- 設定ファイル
  TException
  クラスパス直下の META-INF ディレクトリに terasoluna-exception.properties を作成してください(必須)
  設定内容はログ出力機能の場合と同じです。ファイル名が異なる点に注意してください。

◆ 拡張ポイント

メッセージを作成する際に、デフォルトで java.text.MessageFormat を使用しますが、設定ファイルの message.formatter.fqcn に対する値で変更可能です。出力処理は jp.terasoluna.fw.message.MessageFormatter インタフェースを実装することで行えます。

- 拡張シーン：監視キーワードを Log4J の MDC パラメータに設定したい場合

  メッセージプロパティに、メッセージ ID/ログ ID をキーとして運用監視キーワードとメッセージ本文が設定されている場合、MessageFormatter 内でキーワードとメッセージ本文を分割し、キーワード MDC パラメータに設定することで実現できます。キーワードの出力は Log4J 設定ファイルの ConverrsionPattern で設定できます

  MessageFormatter の実装、META-INF/terasoluna-logger.properties の設定、log4j.properties(または log4j.xml)の設定が必要です。
  以下に設定例を示します。

  - MessageFormatter 実装クラス

```java
package com.example.commom;

import java.text.MessageFormat;

import org.apache.log4j.MDC;

import jp.terasoluna.fw.message.MessageFormatter;

public class KeywordMessageFormatter implements MessageFormatter {

    public String format(String pattern, Object... args) {
        //「監視キーワード,メッセージ本文」という形式を想定
        String[] vals = pattern.split(",");
        String pat = null;
        if (vals.length > 1) {
            // 監視キーワードが設定されている場合
            String keyword = vals[0];
            pat = vals[1];
            // キーワードをMDCに設定する
            MDC.put("keyword", keyword);
        } else {
            pat = pattern;
        }
        return MessageFormat.format(pat, args);
    }
}
```

- META-INF/terasoluna-logger.properties
  キーmessage.formatter.fqcn に対する値に拡張 MessageFormatter の FQCN を設定します。

```
# 拡張メッセージフォーマッタ
message.formatter.fqcn=com.example.common.KeywordMessageFormatter
# メッセージプロパティファイルベースネーム
message.basename=application-messages,common-messages
```

- log4j.properties(一部抜粋)
  ConverrsionPattern に”%X{MDC パラメータ名}”を設定することで MessageFormatter 中に設定した MDC パラメータを任意の箇所で監視キーワードとして出力できます。

```
…
# コンソールアペンダ設定
log4j.appender.consoleLog=org.apache.log4j.ConsoleAppender
log4j.appender.consoleLog.Target = System.out
log4j.appender.consoleLog.layout = org.apache.log4j.PatternLayout
log4j.appender.consoleLog.layout.ConversionPattern=[%d{yyyy/MM/dd
HH:mm:ss}][%-5p][%X{keyword}][%c{1}] %m%n
```

- 出力例メッセージ
  - メッセージプロパティファイル(application-messages)
    “監視キーワード,メッセージ本文”の形式で記述します

```
WAR001=WAR,warning!!
```

  - ログ出力コード(Sample.java)

```
import jp.terasoluna.fw.logger.TLogger;

public class Sample {
    public static final TLogger LOGGER =
TLogger.getLogger(Sample.class);

    public static void main(String[] args) {
        try {
            // ここで例外発生
        } catch (Exception e) {
            LOGGER.warn("WAR001", e);
        }
    }
}
```

  - 出力結果

```
[2011/10/21 20:09:33][WARN ][WAR][Sample] [WAR001] warning!!
java.lang.RuntimeException: sample
    at Sample.main(Sample.java:10)
```

■ 留意事項

Log4J と組み合わせて使用し、Log4J 設定ファイルの ConverrsionPattern の設定で%C (ロガーを生成したクラス名)を使用している場合、常に「jp.terasoluna.fw.logger.TLogger」になってしまいます(ロガーのラッパーであるため)。%C は使用せず、%c (カテゴリ名)を使用してください。尚、%C はパフォーマンスの観点からも推奨しません。

- 設定例（推奨）：

```
log4j.appender.consoleLog.layout.ConversionPattern=[%d{yyyy/MM/dd  HH:mm:ss}]
[%-32c{1}] [%-5p] %m%n
```

- 設定例（非推奨）

```
log4j.appender.consoleLog.layout.ConversionPattern=[%d{yyyy/MM/dd  HH:mm:ss}]
[%-32C{1}] [%-5p] %m%n
```

■ リファレンス

◆ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.logger.TLogger	ログメッセージ出力ロガークラス
2	jp.terasoluna.fw.exception.TException	例外メッセージ出力例外クラス
3	jp.terasoluna.fw.exception.TRuntimeException	例外メッセージ出力実行時例外クラス
4	jp.terasoluna.fw.message.MessageFormatter	メッセージフォーマッタインタフェース
5	jp.terasoluna.fw.message.DefaultMessageFormatter	メッセージフォーマッタデフォルト実装クラス

◆ 依存ライブラリ

下記に挙げたライブラリが当該機能の依存ライブラリです。動作検証は下記のバージョンで実施しています。他のバージョンでも動作すると予想されますが、十分な検証の上ご利用してください。

- common-logging-1.1.1.jar

■ 関連機能

- なし。

■ 使用例

- TERASOLUNA Batch Framework for Java 3.1 以降

■ 備考

- なし。