EJB例外処理のベスト・プラクティス

EJBベースのシステムにおける、より迅速な問題解決のためのコーディングを学ぶ

Srikanth Shenoy (srikanth@srikanth.org)
J2EE Consultant
Objectseek Inc.

2002年 5月 01日

J2EEが最適なエンタープライズ開発プラットフォームになるにつれて、ますます多くのJ2EEベースのアプリケーションが生まれています。J2EEプラットフォームの重要なコンポーネントの1つは、Enterprise JavaBeans (EJB) APIです。J2EEとEJBテクノロジーは共に多くの利点を提供しますが、これらの利点によって新たな問題も発生します。特にエンタープライズ・システムでの問題は、どれも迅速に解決しなければなりません。この記事で、エンタープライズJavaプログラミングのベテランであるSrikanth Shenoy氏は、より迅速な問題解決のためのEJB例外処理のベスト・プラクティスを明らかにしています。

「hello world」を出力するような初歩的なシナリオでは、例外処理も非常にシンプルなものです。メソッドで例外が発生した場合は必ず、例外をキャッチし、スタック・トレースをプリントするか、または例外をスローするメソッドを宣言します。しかし残念ながらこのアプローチは、実際に発生する様々な例外を処理するには十分ではありません。本番システムでは、例外がスローされると、エンド・ユーザーは依頼を処理できなくなるでしょう。そのような例外が発生した場合、エンド・ユーザーは通常、次のようなことを期待しています。

- エラーが発生したことを示す明確なメッセージ
- すぐに利用できるカスタマー・サポート・システムにアクセスする際に使用する一意のエラー番号
- 問題の迅速な解決と、要求が処理されたこと、または一定時間内に処理されることを保証するもの

理想的な場合では、エンタープライズ・レベルのシステムでは、カスタマーにこれらの基本的なサービスを提供するだけでなく、いくつかの必須のバックエンド・メカニズムを適切に用意していることでしょう。たとえば、カスタマー・サービス・チームはエラー通知を即座に受け取ることができるべきです。そうすれば、カスタマーから連絡を受ける前にサービス担当者が問題を認識できます。さらにサービス担当者は、問題の迅速な特定のために、ユーザーの一意のエラー番号と本番ログを、可能であれば正確な行番号やメソッドまで相互参照できることが望まれます。エンド・ユーザーとサポート・チームの双方に必要なツールやサービスを提供するために、シス

テムを構築する際に、それを導入したときに起こりうるあらゆる問題をきちんと把握しておくことが必要です。

この記事では、EJBベースのシステムにおける例外処理について取り上げます。まず、ロギング・ユーティリティーの使用などの例外処理の基本事項について簡単に復習し、そのすぐ後で、EJBテクノロジーが様々なタイプの例外をどのように定義付け、管理しているかについて詳しく説明します。さらに、コード例を用いて、一般的な例外処理のソリューションの長所および短所を取り上げ、EJB例外処理を最大限に利用する私自身のベスト・プラクティスを公開したいと思います。

この記事は、J2EEおよびEJBテクノロジーを理解していることを前提としています。エンティティーBeanとセッションBeanの相違についても理解しておいてください。また、Bean管理による永続性 (BMP) およびコンテナー管理による永続性 (CMP) がエンティティーBeanで何を意味するのかについての知識もあると役に立つでしょう。J2EEおよびEJBテクノロジーの詳細については、参考文献を参照してください。

例外処理の基本事項

システム・エラーを解決する最初のステップは、テスト・システムを本番システムと同じビルドで構成し、スローされた例外の原因であるすべてのコードと、コードの様々な分岐をすべてトレースすることです。分散アプリケーションでは、おそらくデバッガーが機能しないので、例外の追跡には`System.out.println()`メソッドを使用することが多いでしょう。`System.out.println`は便利ですが多くの資源を必要とします。このメソッドは、ディスクI/Oを行っている間の処理を同期化しますが、それによってスループットが大幅に低下します。デフォルトでは、スタック・トレースはコンソールにログされます。しかし本番システムでは、例外トレースをコンソールで参照することはできません。またシステム管理者は、`System.out`や`System.err`をNTの`' '`やUNIXの`dev/null`にマップすることができるので、`System.out.println`が本番システムで表示される保証はありません。さらに、J2EEアプリケーション・サーバーをNTサービスとして実行している場合は、コンソールすらありません。コンソール・ログを出力ファイルにリダイレクトしても、本番J2EEアプリケーション・サーバーが再開したときに、おそらくファイルが上書きされるでしょう。

例外処理の原則

以下は、一般的に受け入れられている例外処理の原則です。

1. 例外を処理できない場合は、それをキャッチしない。
2. 例外をキャッチした場合でも、それを無条件に受け入れない。
3. できる限り原因に近いところで例外をキャッチする。
4. 例外を再スローしない場合は、それをキャッチした場所でログする。
5. 例外処理の細かさに従って、メソッドを構造化する。
6. できるだけ多く、型付きの例外を使用する (特にアプリケーション例外の場合は)。

ポイント1は、明らかにポイント3と矛盾しています。実際のソリューションは、どれほど原因に近いところで例外をキャッチするかと、元の例外の意図または内容が完全にわからなくなる前で、どれだけ離れたところで落とすのかのバランスです。

注: これらの原則は、EJB例外処理メカニズム全体に適用されますが、EJB例外処理に特有のものではありません。

このような理由で、`System.out.println` を含めた状態で本番にコードを持ち込むことは、選択肢とは成り得ません。テストで`System.out.println` を使用し、本番の前にそれらを削除することも、お奨めできるソリューションではありません。なぜなら、この方法では、本番コードとテスト・コードが同じように機能していないからです。必要なのは、テスト・コードと本番コードが同じになるように、また、ロギングが宣言的にオフにされた場合に、本番で発生するパフォーマンス・オーバーヘッドが最小限になるような、ロギングを宣言的に制御するメカニズムです。

明らかなソリューションは、ロギング・ユーティリティーの使用です。正しいコーディング規則を採用することによって、ロギング・ユーティリティーは、システム・エラーか警告かにかかわらず、どのようなメッセージの記録もきちんと管理するようになります。さらに先へ進む前に、ロギング・ユーティリティーについて説明しましょう。

ロギングの全体図: 鳥瞰図

大規模なアプリケーションは、いづれも、開発、テスト、および本番サイクルでロギング・ユーティリティーを使用しています。今日のロギング・ユーティリティーの分野には、役に立つものは、ほんの一握りしかありませんが、その中でも特に2つが広く知られています。1つは、Apacheのオープン・ソース・プロジェクトであるJakartaのLog4Jで、もう1つは、J2SE 1.4に組み込まれ、最近登場したものです。この記事では、Log4Jを用いてベスト・プラクティスについて説明しますが、これらのベスト・プラクティスはLog4Jに限定されるものではありません。

Log4Jには、主にlayout、appender、categoryの3つのコンポーネントがあります。layoutは、ログされるメッセージの形式を表します。appenderは、メッセージがログされる物理的なロケーションのエイリアスです。また、categoryは名前付きエンティティーで、ロギングのハンドルと考えることができます。layoutとappenderは、XMLの構成ファイルで宣言されます。各categoryは、layoutとappenderの定義をもっています。categoryを取得し、それにログすると、メッセージは、そのcategoryに関連するすべてのappenderにおさまり、そのようなメッセージはすべて、XML構成ファイルで指定されたレイアウト形式で表されます。

Log4Jは、ERROR、WARN、INFO、およびDEBUGの4つの優先順位をメッセージに割り当てます。この記事では、説明のために、すべての例外はERRORの優先順位でログされています。この記事で例外をログする場所では、(`Category.getInstance(String name)` メソッドを使用して) categoryを取得するコードを見つまるでしょう。そして、`category. error()` メソッド (ERRORの優先順位を持つメッセージに対応する) を呼び出します。

ロギング・ユーティリティーによって、適切な永続的な場所にメッセージをログすることができますが、それらは問題を根本から解決することはできません。ロギング・ユーティリティーは、本番ログをもとに個々のカスタマーが求める問題レポートを作成することができるわけではありません。そのため、開発中のシステムにそのような機能を構築するかどうかは、開発者に委ねられています。

Log4JまたはJ2SEのロギング・ユーティリティーの詳細については、参考文献を参照してください。

例外カテゴリー

例外は、様々な方法で分類されます。ここでは、EJBから見て例外がどのように分類されるかについて説明します。EJB仕様は、例外を大まかに3つのカテゴリーに分類しています。

- JVM例外: このタイプの例外は、JVMによってスローされます。`OutOfMemoryError` は、JVM例外の一般的な例です。JVM例外に関しては何もできません。JVM例外は致命的な状況を示します。唯一の適切な終了は、アプリケーション・サーバーを停止し、おそらくはハードウェア・リソースを増強し、システムを再開することです。
- アプリケーション例外: アプリケーション例外は、アプリケーションまたはサード・パーティー製のライブラリーによってスローされるカスタム例外です。これらは、基本的にチェックされた例外であり、ビジネス・ロジックにおいてある状況が満たされていないことを示します。このような状況で、EJBメソッドの呼び出し側は状況を適切に処理し、代替方法を使用します。
- システム例外: 多くの場合、システム例外はJVMによって`RuntimeException` のサブクラスとしてスローされます。たとえば、`NullPointerException` や`ArrayOutOfBoundsException` は、コードのバグが原因でスローされます。システム例外のもう1つのタイプは、JNDI検索のミススペルなど、システムで不適切な構成リソースが発生した場合に、発生します。そのような場合、システムはチェックされた例外をスローします。これらのチェックされたシステム例外をキャッチし、それらをチェックされていない例外としてスローすることは、大いに意味のあることです。一般的に、例外に関してできることが何もない場合、それはシステム例外であり、チェックされていない例外としてスローすべきです。

注:チェックされた例外は、`java.lang.Exception` をサブクラス化するJavaクラスです。`java.lang.Exception` をサブクラス化することによって、コンパイル時に例外をキャッチしなければなりません。反対に、チェックされていない例外は、`java.lang.RuntimeException` をサブクラス化するJavaクラスです。`java.lang.RuntimeException` をサブクラス化することによって、例外をキャッチするようコンパイラーに強制されなくなります。

EJBコンテナーはどのように例外を処理するか

EJBコンテナーは、EJBコンポーネントに対する各メソッドの呼び出しをインターセプトします。その結果、メソッドの呼び出しで起こる各例外もEJBコンテナーによってインターセプトされます。EJB仕様は、アプリケーション例外とシステム例外の2種類の例外の処理のみを扱っています。

アプリケーション例外は、EJB仕様によって、リモート・インターフェースのメソッド・シグニチャーで宣言された例外として定義されています (但し`RemoteException`以外)。アプリケーション例外は、ビジネス・ワークフローの特別なシナリオに相当します。このタイプの例外がスローされる場合のために、クライアントにはリカバリー・オプションが提供されます。　これは通常、異なる方法での要求の処理を伴うものです。しかし、だからといって、リモート・インターフェース・メソッドの`throws` 文節で宣言されたチェックされていない例外がアプリケーション例外として扱われるわけではありません。仕様は、アプリケーション例外が`RuntimeException` またはそのサブクラスを拡張しないと明確に述べています。

アプリケーション例外が発生した場合、EJBコンテナーは、関連する`EJBContext` オブジェクトの`setRollbackOnly()` メソッドの呼び出しによって明示的にトランザクションをロールバックす

るように要求されなければ、トランザクションのロールバックを行いません。実際、アプリケーション例外はそのままクライアントに渡されます。EJBコンテナーは例外のラップやマッサージをまったく行いません。

システム例外は、チェックされた例外またはチェックされていない例外として定義されており、EJBメソッドはリカバリーすることができません。EJBコンテナーは、チェックされていない例外をインターセプトすると、トランザクションをロールバックし、必要なクリーンアップを行います。そして、チェックされていない例外を`RemoteException` でラップし、それをクライアントにスローします。したがってEJBコンテナーは、すべてのチェックされていないシステム例外を`RemoteException` (または`TransactionRolledbackException` などのサブクラス) としてクライアントに示します。

チェックされた例外の場合、コンテナーは上述のようなハウスキーピング (切り盛り) を自動的には実行しません。EJBコンテナーの内部ハウスキーピングを使用するには、チェックされた例外をチェックされていない例外としてスローしなければなりません。チェックされたシステム例外 (`NamingException` など) が発生した場合は常に、元の例外をラップして、`javax.ejb.EJBException` またはそのサブクラスをスローします。`EJBException` 自体はチェックされていない例外なので、メソッドの`throws` 節でそれを宣言する必要はありません。EJBコンテナーは、`EJBException` またはそのサブクラスをキャッチし、`RemoteException` でそれをラップし、`RemoteException` をクライアントにスローします。

システム例外はアプリケーション・サーバーによってログされますが (EJB仕様で指示されている)、ロギング形式はアプリケーション・サーバーごとに異なります。企業は多くの場合、必要な統計にアクセスするために、生成されたログでシェル・スクリプト / Perlスクリプトを実行しなければならないでしょう。ロギング形式を統一するには、自分たちのコードで例外をログするとよいでしょう。

注: EJB 1.0仕様では、チェックされたシステム例外を`RemoteException` としてスローしなければなりませんでした。EJB 1.1の仕様から、EJB実装クラスが`RemoteException` をまったくスローしないように指示しています。

一般的な例外処理方針

例外処理の方針がないと、プロジェクト・チームの様々な開発者は例外処理のコードを異なる方法で記述するでしょう。少なくとも、単一の例外がシステムの異なる場所で異なる方法で記述され、処理されるようになり、本番サポート・チームに混乱が生じるでしょう。また方針がないと、システムの複数の場所でロギングが行われてしまいます。ロギングは集中化するか、または複数の管理しやすい単位に分割すべきです。例外ロギングは、内容をあやうくすることなく、できる限り少ない場所で発生するのが理想的です。以降のセクションでは、エンタープライズ・システム全体にわたって統一した方法で実装できるコーディング方針について説明します。この記事で開発されたユーティリティー・クラスは、参考文献からダウンロードすることができます。

リスト1は、セッションEJBコンポーネントのメソッドを示しています。このメソッドは、特定日の前にカスタマーによって行われたすべての注文を削除します。まず、`OrderEJB` のHomeインターフェースを取得します。次に、特定のカスタマーの注文をすべて取り出します。特定日の前の注文が見つかった場合、注文項目を削除し、注文自体も削除します。3種類の例外がスローさ

れ、3種類の一般的な例外処理の実践方法が示されている点に着目してください。(わかりやすくするために、コンパイラーの最適化は使用しないものとします。)

リスト1. 3種類の一般的な例外処理の実践方法

```
100  try {
101    OrderHome homeObj = EJBHomeFactory.getInstance().getOrderHome();
102    Collection orderCollection = homeObj.findByCustomerId(id);
103    iterator orderItter = orderCollection.iterator();
104    while (orderIter.hasNext()) {
105      Order orderRemote = (OrderRemote) orderIter.getNext();
106      OrderValue orderVal = orderRemote.getValue();
107      if (orderVal.getDate() < "mm/dd/yyyy") {
108        OrderItemHome itemHome = EJBHomeFactory.getInstance().getItemHome();
109        Collection itemCol = itemHome.findByOrderId(orderId)
110        Iterator itemIter = itemCol.iterator();
111        while (itemIter.hasNext()) {
112          OrderItem item = (OrderItem) itemIter.getNext();
113          item.remove();
114        }
115        orderRemote.remove();
116      }
117    }
118  } catch (NamingException ne) {
119    throw new EJBException("Naming Exception occurred");
120  } catch (FinderException fe) {
121    fe.printStackTrace();
122    throw new EJBException("Finder Exception occurred");
123  } catch (RemoteException re) {
124    re.printStackTrace();
125    //Some code to log the message
126    throw new EJBException(re);
127  }
```

次に、上記のコード例を用いて、3種類の例外処理の実践方法の欠点について見てみましょう。

エラー・メッセージによる例外のスロー / リスロー

`NamingException` は、ライン101または108で発生します。`NamingException` が発生した場合、このメソッドの呼び出し側が`RemoteException` を受け、ライン119に至ります。呼び出し側は、実際の`NamingException` がライン101で発生したのか、ライン108で発生したのかはわかりません。例外の内容は、ログされるまで保存されないので、問題の原因を追跡することができません。このようなシナリオでは、例外の内容が「無条件で受け入れられた(swallowed)」と言われます。この例が示しているように、メッセージによる例外のスロー / 再スローは、優れた例外処理のソリューションとは言えません。

コンソールへのログと例外のスロー

`FinderException` は、ライン102または109で発生します。しかし、例外はコンソールにログされるので、呼び出し側は、コンソールが利用できる場合しか、ライン102または109に追いつめることができません。これは明らかに仮定できないので、例外はライン122で捉えるしかありません。その理由は、上述の理由と同様です。

元の例外をラップし、内容を保存する

`RemoteException` は、ライン102、106、109、113または115で発生する可能性があり、ライン123の`catch` ブロックでキャッチされます。`RemoteException` は、`EJBException` でラップされるので、呼び出し側がログする場合は常に無傷です。このアプローチは前述の2つの方法よりは優れて

いますが、ロギング方針がありません。`deleteOldOrders()` メソッドの呼び出し側が例外をログした場合は、重複ロギングが発生します。そしてロギングが行われているにもかかわらず、本番ログまたはコンソールは、カスタマーが問題を報告する場合に相互参照できません。

EJB例外処理の発見的手法

EJBコンポーネントはどの例外をスローすべきでしょうか、また皆さんはシステムのどこでそれをログすべきでしょうか。この2つの問題は複雑につながっており、同時に対処する必要があります。答えは、以下のような要因によって異なります。

- **EJBシステム設計**: 優れたEJB設計では、クライアントはエンティティーEJBコンポーネントのメソッドを呼び出しません。ほとんどのエンティティーEJBメソッドの呼び出しは、セッションEJBコンポーネントで発生します。そのような設計の場合は、例外をセッションEJBコンポーネントでログします。クライアントがエンティティーEJBメソッドを直接呼び出す場合は、むしろエンティティーEJBコンポーネントでメッセージをログします。ただしキャッチがあります。すなわち、同じエンティティーEJBメソッドがセッションEJBコンポーネントによって呼び出されることもあります。そのようなシナリオで、どのように重複ロギングを防ぐのでしょうか。同様に、セッションEJBコンポーネントが他のセッションEJBコンポーネントを呼び出した場合に、どのように重複ロギングを防ぐのでしょうか。これらのケースを処理する一般的なソリューションについて説明しましょう。(EJB1.1では、アーキテクチャー上、クライアントがエンティティーEJBコンポーネントのメソッドを呼び出すことを制御しない点に注意してください。EJB2.0では、エンティティーEJBコンポーネントのローカル・インターフェースを定義することによって、この制限を指示することができます。)
- **計画されたコード再利用の程度**: ここでの問題は、複数の場所でロギング・コードを追加するか、またはロギング・コードを減らすためにコードの再設計 / リファクタリングを行うかということです。
- **使用するクライアントのタイプ**: J2EE Web層、スタンドアロンのJavaアプリケーション、PDA、または他のクライアントのいずれを使用するかについて検討することが重要です。Web層設計には、あらゆる形態とサイズがあります。毎回異なるコマンドをパスすることによってWeb層がEJB層で同じメソッドを呼び出すCommandパターンを使用する場合は、コマンド実行が発生するEJBコンポーネントで例外をログすることが役立ちます。他のほとんどのWeb層設計では、例外ロギング・コードを追加する場所がより少ないので、Web層自体で例外をログするほうがより簡単で優れています。Web層とEJB層を同じ場所に配置し、他のタイプのクライアントをサポートする必要がない場合は、後者について検討するとよいでしょう。
- **処理する例外のタイプ (アプリケーションまたはシステム)**:アプリケーション例外の処理とシステム例外の処理とは大きく異なります。システム例外は、EJB開発者が意図しないところで発生します。システム例外の意図は不明確なので、その内容が例外のコンテキストを示しているはずです。これまで見てきたように、これを処理する最良の方法は、元の例外をラップすることです。一方、アプリケーション例外は、多くの場合メッセージをラップすることにより、EJB開発者によって明示的にスローされます。アプリケーション例外の意図は明確なので、そのコンテキストを保存する理由はありません。このタイプの例外では、EJB層またはクライアント層でのログは必要ありません。むしろ解決に至る代替方法と共に分かりやすい方法でエンド・ユーザーに提示すべきです。システム例外メッセージは、エンド・ユーザーに対してさほど分かりやすくする必要はありません。

アプリケーション例外の処理

以降のセクションでは、アプリケーション例外およびシステム例外のEJB例外処理と、Web層設計について詳しく見ていきましょう。このトピックの一部として、セッションEJBコンポーネントおよびエンティティーEJBコンポーネントからスローされる例外の様々な処理方法について説明します。

エンティティーEJBコンポーネントのアプリケーション例外

リスト2は、エンティティーEJBの`ejbCreate()` メソッドを示しています。このメソッドの呼び出し側は、`OrderItemValue` を渡し、`OrderItem` エンティティーの作成を要求します。`OrderItemValue` に名前がないと、`CreateException` がスローされます。

リスト2. エンティティーEJBコンポーネントのejbCreate() メソッドの例

```
public Integer ejbCreate(OrderItemValue value) throws CreateException {
    if (value.getItemName() == null) {
      throw new CreateException("Cannot create Order without a name");
    }
    ..
    ..
    return null;
}
```

リスト2は、`CreateException`　の典型的な使用法です。同様に、メソッドの入力引数が正確な値を持っていない場合は、ファインダー・メソッドが`FinderException`　をスローします。

しかしコンテナー管理による永続性 (CMP) を使用している場合、開発者はファインダー・メソッドに手出しできないので、`FinderException` はCMP実装によってスローされません。それでも、Homeインターフェースで、ファインダー・メソッドの`throws` 節で`FinderException` を宣言するほうがよいでしょう。`RemoveException` は、エンティティーが削除される場合にスローされるもう1つのアプリケーション例外です。

エンティティーEJBコンポーネントからスローされるアプリケーション例外は、これらの3つのタイプ (`CreateException`、`FinderException`、および`RemoveException`) とそのサブクラスにかなり限定されます。アプリケーション例外のほとんどは、セッションEJBコンポーネントで起こります。というのも、それが、インテリジェントな意思決定が行われる場所だからです。一般的にエンティティーEJBコンポーネントは、データの作成と取り出しを行うだけのダム・クラスです。

セッションEJBコンポーネントのアプリケーション例外

リスト3は、セッションEJBコンポーネントのメソッドを示しています。このメソッドの呼び出し側は、特定のタイプの項目をn 個注文しようとしています。`SessionEJB()` メソッドは、在庫が十分にないことを把握し、`NotEnoughStockException` をスローします。`NotEnoughStockException` は、ビジネス固有のシナリオに当てはまります。この例外がスローされると、代替ルートが呼び出し側に提案され、呼び出し側は項目の発注量を少なくして注文することができます。

リスト3. セッションEJBコンポーネントのコンテナー・コールバック・メソッドの例

```
public ItemValueObject[] placeOrder(int n, ItemType itemType) throws
NotEnoughStockException {
    //Check Inventory.
    Collection orders = ItemHome.findByItemType(itemType);
    if (orders.size() < n) {
      throw NotEnoughStockException("Insufficient stock for " + itemType);
    }
}
```

システム例外処理

システム例外処理は、アプリケーション例外処理よりも複雑なトピックです。セッションEJBコンポーネントとエンティティーEJBコンポーネントは同じようにシステム例外を処理するので、このセクションでは、例としてエンティティーEJBコンポーネントに焦点を置きたいと思います。ただし、ほとんどの例はセッションEJBコンポーネントによる作業にも適用することができます。

エンティティーEJBコンポーネントが他のEJBリモート・インターフェースを参照する場合には`RemoteException`に、他のEJBコンポーネントを検索しているときには`NamingException`に、そしてBean管理による永続性 (BMP) を使用している場合に`SQLException` に遭遇することになります。このようなチェックされたシステム例外は、`EJBException` またはそのサブクラスのいずれかとしてキャッチ / スローされ、元の例外はラップされるはずです。リスト4は、システム例外に対するEJBコンテナーの振る舞いに適合する、システム例外の処理方法を示しています。元の例外をラップし、それをエンティティーEJBコンポーネントに再スローすることによって、例外をログしたい場合にそれにアクセスすることができます。

リスト4. システム例外を処理する一般的な方法

```
try {
    OrderHome orderHome = EJBHomeFactory.getInstance().getOrderHome();
    Order order = orderHome.findByPrimaryKey(Integer id);
} catch (NamingException ne) {
    throw new EJBException(ne);
} catch (SQLException se) {
    throw new EJBException(se);
} catch (RemoteException re) {
    throw new EJBException(re);
}
```

重複ロギングの回避

通常、例外ロギングはセッションEJBコンポーネントで発生します。しかし、エンティティーEJBコンポーネントがEJB層の外から直接アクセスされる場合はどうでしょうか。その場合、エンティティーEJBコンポーネントで例外をログし、それをスローしなければなりません。ここでの問題は、呼び出し側には例外が既にログされていることを知るすべがないため、それを再びログしてしまう可能性が高く、その結果として重複ロギングが起こることです。さらに重要なことは、呼び出し側には最初のロギングの間に生成された一意のIDにアクセスするすべがないことです。相互参照のメカニズムがないロギングは役に立ちません。

最悪のケースのシナリオについて考えてみましょう。エンティティーEJBコンポーネントの`foo()` メソッドが、スタンドアロンのJavaアプリケーションからアクセスされるとします。同じメソッ

ドが、`bar()` と呼ばれるセッションEJBメソッドでアクセスされます。Web層クライアントはセッションEJBコンポーネントで`bar()` メソッドを呼び出し、例外をログします。セッションEJBメソッド`bar()` がWeb層から呼び出されたときに、例外がエンティティーEJBメソッド`foo()` で発生した場合、例外は3つの場所でログされたことになります。1つはエンティティーEJBコンポーネント、2つめはセッションEJBコンポーネント、そして3つめはWeb層です。そして、スタック・トレースはどれも相互参照できません。

幸運にも、このような問題は、一般的にかなり容易に対処することができます。必要なのは、呼び出し側が以下のことを行うメカニズムだけです。

- 一意のIDへのアクセス
- 例外が既にログされているかどうかを調べること

この情報を格納するために、`EJBException` をサブクラス化することができます。リスト5は、`LoggableEJBException` サブクラスを示しています。

リスト5. LoggableEJBException -- EJBExceptionのサブクラス

```
public class LoggableEJBException extends EJBException {
    protected boolean isLogged;
    protected String uniqueID;
    public LoggableEJBException(Exception exc) {
 super(exc);
 isLogged = false;
 uniqueID = ExceptionIDGenerator.getExceptionID();
    }
 ..
 ..
}
```

`LoggableEJBException` クラスは、例外がログされたかどうかを確認するインディケーター・フラグ (`isLogged`) を持っています。`LoggableEJBException` をキャッチする場合は必ず、例外が既にログされているかどうかを確認してください (`isLogged == false`)。falseの場合は、例外をログし、フラグを`true` に設定してください。

`ExceptionIDGenerator` クラスは、マシンの現在の時間とホスト名を使用して、例外の一意のIDを生成します。お望みであれば、手のこんだアルゴリズムを使用して、一意のIDを生成することもできます。エンティティーEJBコンポーネントで例外をログした場合、その例外は他の場所ではログされません。ロギングを行わずにエンティティーEJBコンポーネントで`LoggableEJBException` をスローした場合は、セッションEJBコンポーネントでログされ、Web層ではログされません。

リスト6は、このテクニックを使用してリスト4を再記述したものを示しています。自分のニーズに合わせて`LoggableException` を拡張することもできます (エラー・コードを例外に割り当てるなど)。

リスト6. LoggableEJBExceptionによる例外処理

```
try {
    OrderHome orderHome = EJBHomeFactory.getInstance().getOrderHome();
    Order order = orderHome.findByPrimaryKey(Integer id);
} catch (NamingException ne) {
    throw new LoggableEJBException(ne);
} catch (SQLException se) {
    throw new LoggableEJBException(se);
} catch (RemoteException re) {
    Throwable t = re.detail;
     if (t != null && t instanceof Exception) {
       throw new LoggableEJBException((Exception) re.detail);
     }  else {
       throw new LoggableEJBException(re);
     }
}
```

RemoteExceptionのロギング

リスト6を見ておわかりのように、ネーミングとSQL例外は、スローされる前に`LoggableEJBException` でラップされます。しかし`RemoteException` は、わずかに異なる方法 -- 少し労働集約的な方法で処理されます。

セッションEJBコンポーネントのシステム例外

セッションEJB例外をログする場合は、リスト7 で示されているロギング・コードを使用します。あるいは、リスト6 のように例外をスローします。セッションEJBコンポーネントが、エンティティーEJBコンポーネントとは異なるやり方で例外を処理する1つの方法があります。ほとんどのEJBシステムはWeb層からしかアクセスできず、セッションEJBはEJB層のファサードとして機能することができるので、実際にはセッションEJB例外のロギングをWeb層まで遅らせることができます。

この処理方法が異なるのは、`RemoteException` では実際の例外が`detail` (`Throwable` タイプ) と呼ばれるpublic属性に格納されるためです。ほとんど常に、このpublic属性は例外を保持します。`RemoteException` で`printStackTrace` を呼び出すと、詳細のスタック・トレースに加えて、例外そのもののスタック・トレースがプリントされます。したがって`RemoteException` のスタック・トレースは必要ありません。

`RemoteException` などの複雑さからアプリケーション・コードを切り離すために、これらの行は、`ExceptionLogUtil` と呼ばれるクラスにリファクタリングされます。このクラスで必要なことは、`LoggableEJBException` を作成する必要がある場合は必ず`ExceptionLogUtil.createLoggableEJBException(e)` を呼び出すことだけです。リスト6では、エンティティーEJBコンポーネントは例外をログしません。しかし、エンティティーEJBコンポーネントで例外をログする場合でも、このソリューションは機能します。リスト7は、エンティティーEJBコンポーネントの例外ロギングを示しています。

リスト7. エンティティーEJBコンポーネントの例外ロギング

```
try {
    OrderHome orderHome = EJBHomeFactory.getInstance().getOrderHome();
    Order order = orderHome.findByPrimaryKey(Integer id);
} catch (RemoteException re) {
    LoggableEJBException le = ExceptionLogUtil.createLoggableEJBException(re);
    String traceStr = StackTraceUtil.getStackTrace(le);
    Category.getInstance(getClass().getName()).error(le.getUniqueID() +
":" + traceStr);
    le.setLogged(true);
    throw le;
}
```

リスト7には、絶対に安全な例外ロギング・メカニズムが示されています。チェックされたシステム例外をキャッチしたら、新しい`LoggableEJBException` を作成します。次に`StackTraceUtil` クラスを使用して、LoggableEJBExceptionのスタック・トレースをストリングとして取得します。そしてLog4Jカテゴリーを使用して、そのストリングをエラーとしてログします。

StackTraceUtilクラスはどのように機能するか

リスト7には、`StackTraceUtil` と呼ばれる新しいクラスがありました。Log4J は`String` メッセージをログすることしかできないため、このクラスが、スタック・トレースを`String` に変換する問題に対処します。リスト8は`StackTraceUtil` クラスの機能を示しています。

リスト8. StackTraceUtilクラス

```
public class StackTraceUtil {
public static String getStackTrace(Exception e)
      {
          StringWriter sw = new StringWriter();
          PrintWriter pw = new PrintWriter(sw);
          return sw.toString();
      }
      ..
      ..
}
```

`java.lang.Throwable` のデフォルトの`printStackTrace()` メソッドは、エラー・メッセージを`System.err` にログします。`Throwable` はまた、`PrintWriter` または`PrintStream` にログするためにオーバーロードされた`printStackTrace()` メソッドも持っています。`StackTraceUtil` の上記のメソッドが、`PrintWriter` 内に`StringWriter` をラップします。`PrintWriter` にスタック・トレースがある場合、それは単に、`StringWriter` で`toString()` を呼び出し、スタック・トレースの`String` 表示を取得します。

Web層のEJB例外処理

Web層設計では、多くの場合、クライアント側に例外ロギング・メカニズムを置くほうが、さらに容易で効果的になります。これをうまく行うには、Web層はEJB層の唯一のクライアントでなければなりません。またWeb層は、以下のパターンやフレームワークのいずれか1つに基づく必要があります。

- **パターン**: Business Delegate、 FrontController、 またはIntercepting Filter

- フレームワーク: Struts、または階層を含む同様のMVCフレームワーク

例外ロギングは、なぜクライアント側で行うべきなのでしょうか。まず、アプリケーション・サーバー外に制御が渡されたわけではありません。JSPページ、サーブレット、またはそのヘルパー・クラスで構成されているいわゆるクライアント層は、J2EEアプリケーション・サーバーそのものにおいて実行されます。第二に、優れた設計のWeb層のクラスには、階層 (たとえば、Business Delegateクラス、Intercepting Filterクラス、http要求処理プログラム・クラス、JSPベース・クラス、またはStruts Actionクラス) あるいはFrontControllerサーブレットの形式の単一の呼び出し点があります。これらの階層のベース・クラスまたはControllerクラスの中心点に、例外ロギング・コードを含むことができます。セッションEJBベースのロギングの場合、EJBコンポーネントの各メソッドはロギング・コードを持っていなければなりません。ビジネス・ロジックの拡大につれて、セッションEJBメソッド数もロギング・コード数も増大します。Web層システムは、より少ないロギング・コードですみます。Web層とEJB層を同じ場所に配置し、他のタイプのクライアントをサポートする必要がない場合は、この代替手段について検討してみてください。いずれにしろ、ロギング・メカニズムは変わらないので、前述したのと同じテクニックを使用することができます。

実世界の複雑さ

ここまでは、セッションEJBコンポーネントとエンティティーEJBコンポーネントの例外処理のテクニックについて簡単なシナリオで説明しました。しかしアプリケーション例外の組み合わせには、より複雑で様々に解釈できるものもあります。リスト9に例を示してあります。`OrderEJB`の`ejbCreate()` メソッドは、`CustomerEJB` で遠隔参照の取得を試み、その結果`FinderException` になります。`OrderEJB` と`CustomerEJB` は両方ともエンティティーEJBコンポーネントです。この`FinderException` を`ejbCreate()` でどのように解釈するとよいでしょうか。アプリケーション例外として処理しますか (EJB仕様では標準のアプリケーション例外として定義されている)、それともシステム例外として処理しますか。

リスト9. ejbCreate() メソッドのFinderException

```
public Object ejbCreate(OrderValue val) throws CreateException {
     try {
        if (value.getItemName() == null) {
          throw new CreateException("Cannot create Order without a name");
        }
        String custId = val.getCustomerId();
        Customer cust = customerHome.fingByPrimaryKey(custId);
        this.customer = cust;
     } catch (FinderException ne) {
     //How do you handle this Exception ?
     } catch (RemoteException re) {
   //This is clearly a System Exception
   throw ExceptionLogUtil.createLoggableEJBException(re);
     }
     return null;
}
```

`FinderException` をアプリケーション例外として処理できないこともありませんが、システム例外として処理するほうがよいでしょう。というのは、EJBクライアントはEJBコンポーネントをブラック・ボックスとして処理するからです。`createOrder()` メソッドの呼び出し側が`FinderException` 受け取っても、呼び出し側には意味がありません。`OrderEJB` がカスタマーの

遠隔参照の設定を試みているという事実は、呼び出し側には透過的です。クライアントから見れば、障害の意味するものは、単に注文を作成できないという事実だけです。

このタイプのシナリオのもう1つの例は、セッションEJBコンポーネントが別のセッションEJBを作成し、`CreateException` を受け取る場合です。同様に、エンティティーEJBメソッドがセッションEJBコンポーネントを作成し、`CreateException` を受け取るというシナリオもあります。これらの例外は両方とも、システム例外として処理されます。

もう1つの課題は、セッションEJBコンポーネントがコンテナー・コールバック・メソッドの1つで`FinderException` を受け取る場合です。このタイプのシナリオは、ケース・バイ・ケースで処理しなければなりません。`FinderException` は、アプリケーション例外またはシステム例外として処理することができます。呼び出し側がセッションEJBコンポーネントで`deleteOldOrder` メソッドを呼び出すリスト1のケースについて考えてみましょう。`FinderException` をキャッチする代わりに、スローするとどうなるでしょうか。この特定のケースでは、`FinderException` をシステム例外として処理するのが論理的であるように思えます。というのも、セッションEJBコンポーネントは、ワークフローの状況を処理し、呼び出し側に対してブラック・ボックスとして機能するので、メソッドで多くの作業を行う傾向があるからです。

一方、セッションEJBが発注を処理するシナリオについて考えてみましょう。発注するには、ユーザーはプロファイルを持っていなければなりません。しかし、この特定のユーザーはプロファイルを持っていません。ビジネス・ロジックは、プロファイルがないことをユーザーに明示的に通知するようセッションEJBに求める可能性があります。欠落したプロファイルは、セッションEJBコンポーネントで`javax.ejb.ObjectNotFoundException` (`FinderException` のサブクラス) として明らかになる場合が多いでしょう。このような場合、最もよいアプローチは、セッションEJBコンポーネントで`ObjectNotFoundException` をキャッチし、アプリケーション例外をスローし、ユーザーにプロファイルがないことを知らせることです。

優れた例外処理方針があっても、しばしばテスト中に発生する別の問題があり、またさらに重要なこととして、本番で発生する問題もあります。コンパイラーとランタイムの最適化はクラスの全体的な構造を変えてしまう可能性があり、これによって、スタック・トレース・ユーティリティーを使用して例外を追跡する機能が制限されるおそれがあります。この場合には、コードのリファクタリングが役に立ちます。大規模なメソッド呼び出しは、より小さく、より管理しやすい塊に分割します。また、可能な場合は、必要なだけ例外を型付けしてください。例外をキャッチする場合は必ず、あらゆるものをキャッチするのではなく、型付けされた例外をキャッチするようにします。

結論

この記事では多くのことを取り上げましたし、説明したすべての最新の設計が価値あるものかどうか疑問に思われるかもしれません。私の経験では、中小規模のプロジェクトでさえ、テストと本番サイクルは言うまでもなく、開発サイクルにおいてもその努力が報われます。さらに、ダウン時間がビジネスに致命的となる本番システムでは、優れた例外処理アーキテクチャーの重要性はいくら強調してもしすぎることはありません。

この記事で示したベスト・プラクティスがお役に立てば幸いです。この記事で示した情報の詳細については、参考文献にあるリストを確認してください。

著者について

Srikanth Shenoy

Srikanth Shenoyは大規模なJ2EEプロジェクトやEAIプロジェクトのアーキテクチャ、設計、開発、展開などが専門。製造、運輸、金融などの分野の顧客に対し、Javaの"write once, run anywhere" (一度作成すれば、どこでも実行可能)という夢の実現に尽力していきました。Sun Certified Enterprise Architect（ Sun認定、エンタープライズ設計者 ）であり、共著でPractical Guide to J2EE Web Projects を出版予定。メールアドレスはsrikanth@srikanth.orgです