快速デバッグ技法

脱!休日出勤

.NETアプリケーション障害解析の極意

第4回 予期せぬ例外への対応 ―プログラミング工程―

株式会社NTTデータ
飯山 教史
IIYAMA, Takashi

ソースコードがない場合のデバッグ

前回はサードパーティ製のモジュール（DLL）が悪さをしているときにどのようにデバッグするかについて説明した。モジュールのソースコードが手元にあるという前提でデバッグ方法を紹介したが、現場でそんなラッキーなケースは少ないだろう。

各開発者のコーディングが終了してから他者の開発したモジュールと結合し、「実行中例外発生」→「どのコードで起こっているのか？」と問題の切り分け作業をする。このときに、モジュールのソースコードがあることは少ない。モジュールのバイナリを渡され、それを実行すると例外が発生するが、ソースコードはない。このような状態でどのようにデバッグを進めたらよいのだろうか？

ソースコードがないので、当然だが根本的な原因はつかめない。もちろん、自分の担当分のコードに原因があるのなら話は別だが、結合している他者のモジュールのソースコードがなければ、できることは限られてくる。この場合、故障原因の追求はあきらめ、

故障部位（例外が発生したコードの場所）の特定

を目指そう。ソースコードがないのにどうして故障部位が特定できるのか、疑問に思う方も多いだろう。こんなときに役に立つのがWinDbg（Debugging Tools for Windows）とSOS.DLLである。これらは、例外発生時に例外を起こしたアプリケーションの情報を記録したuser.dmpを解析するためのツールおよびモジュールである。

今回はこれらの利用手順と、コードがない環境で故障部位を特定する方法を説明する。user.dmpのような「ダンプファイル」の解析は敷居が高いと思うかもしれないが、故障部位の特定であればそれほど困難ではないのでぜひマスターしてもらいたい。

レベル >>> Level

1 2 3 4 5

サンプル >>> Sample

この記事で取り上げたソースコードおよびサンプルプログラムは、**http://www.shoeisha.com/mag/windev/** からダウンロード可能です。

言語 >>> Language

- Visual Basic

ツール >>> Tool

- Visual Studio 2005 Team Edition for Software Testers
- WinDbg（Debugging Tools for Windows）
- SOS.DLL

例外が起こると？

まずデフォルトの状態でアプリケーションが例外を起こすとどうなるのか、0除算する簡単なサンプルコード（リス

リスト1：0除算する簡単なサンプルコード（サンプルMakeExcpt）

```
Private Sub btnException_Click( _
 ByVal sender As System.Object, _
 ByVal e As System.EventArgs _
 ) Handles btnException.Click
   Dim i As Integer = 1
   Dim j As Integer = 0
   Dim k As Integer = 0

   k = Division(i, j)
End Sub

Private Function Division( _
 ByVal numerator As Integer, _
 ByVal denominator As Integer _
 ) As Integer
   Return numerator / denominator
End Function
```

ト1）を例にみていこう。

実システムでの動きを確認するためにはRelease版（リリースビルド）の挙動を調査すべきである[注1]。そのため、ここではRelease版を実行する（図1）[注2]。

図1：リリースビルドの設定（ソリューションのプロパティの「構成」で“Release”を選択）

サンプルコードのプロジェクトをリリースビルドして実行すると、図2のように例外が発生したことを伝えるポップアップが表示される。このポップアップで注目してもらいたい箇所が2つある。

ひとつ目は「例外テキスト」以下に表示されているメッセージである。今回は次のメッセージが表示されている。

System.OverflowException: 算術演算の結果オーバーフローが発生しました。

0除算なので冒頭の「算術演算の結果」はまだわかるが「オーバーフローが発生しました」は直感的にわかりにくいだろう。このように、ここで表示されるメッセージはあまり役に立たないことが多い。

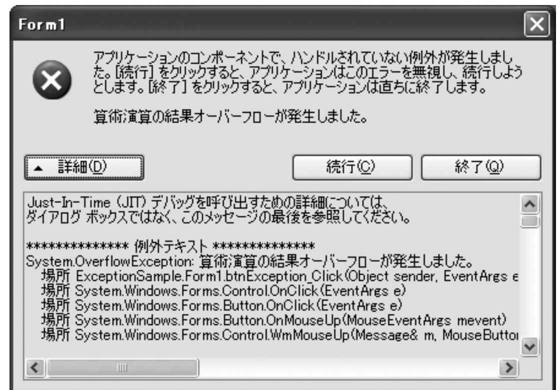

図2：リスト1実行時に表示される例外のポップアップ

次に注目してもらいたいのは上記のメッセージの下に書かれている「場所」のリストである。特に、最上位に書かれている「btnException_Click」に注目してほしい。このイベントハンドラはリスト1でも示したように、0除算をしている関数そのものである。

この「場所」は、例外が起こるまでの関数呼び出しの順番を示しており、表記されている下の関数から上の関数へと順に呼び出されている。そのため、今回であれば例外を起こした（0除算した）イベントハンドラ「btnException_Click」が最上位にきているのである。したがって、今回は「btnException_Click」から調査を始めることになる。

ではこのポップアップの情報で故障部位がわかるかというと、これだけでは不十分なことが多い。たとえばリスト2に示すコードをみてもらいたい。EXEからDLLのメソッドを呼び出して例外が発生していることはEXEのコード、

```
em.MakeException()
```

ですぐに確認できる。しかしポップアップ（図3）の「場所」の最上位が「MakeException」ではなく「CallException_Click」であることがわかる。これでは本当に0除算が行なわれた場所とは異なるので故障部位を特定しているとは言えない。ではどうすればよいのか？　こ

注1）Release版とDebug版の違いは以下のURLを参照。
http://msdn2.microsoft.com/ja-jp/library/dykf6bx9.aspx

注2）Release版の作成方法の詳細は以下のURLを参照。
http://msdn2.microsoft.com/ja-JP/library/wx0123s5.aspx