4D Advanced Debugging Techniques – Part 1

By Josh Fletcher, Technical Support Engineer, 4D Inc.

TN 06-31

Abstract

本テクニカルノートは次のような二部構成になっています：

- 4th Dimension (4D) 2004 に組み込まれているデバッグツールに関する記事。(第一部)
- テキスト形式のログファイルを出力し、4D 2004 に組み込まれているデバッグツールでは対応が困難な分野をデバッグする方法に関する記事。(第二部)

第一部では、デバッグ作業の基本コンセプト、および 4D に組み込まれているデバッグツールについて解説した後、少し高度なデバッグテクニックを幾つか紹介しています。

第二部では、ログファイルの出力を中心としたデバッグの基本コンセプトについて解説した後、汎用的なログ出力コンポーネント、そのインストール方法および使用方法を紹介しています。

Introduction

このセクションでは、デバッグの基本コンセプトおよび 4D との関係を取り上げています。

Debugging Concepts

Wikipedia(http://www.wikipedia.org/)では次のようにデバッグが定義されています。

> コンピュータプログラムまたは電子機器が期待どおりに動作するように
> バグまたは欠陥を見つけ出し、数を減らすための系統だった手法のこと。
> http://en.wikipedia.org/wiki/Debugging からの抄訳。

上記の定義でおそらく一番大切なのが「系統だった」という表現です。デバッグは、系統だった視点から取り組むことが肝要です。組織的な方法で取り組まないなら、問題を切り分けることが困難になるばかりか、デバッグの過程で新たな問題を作り出すことになりかねません。

デバッグの基本的な手順は、次のようなステップに整理することができます：

1. バグの存在を認める。
2. バグの発生を再現するパターンを特定する。
3. バグの出所を特定する。
4. バグの原因を特定する。
5. バグの修正(フィックス)を開発する。
6. バグフィックスを適用する。
7. バグフィックスをテストする。

上記リストのステップ 1 は、しばしば見過ごされる点ですが、バグの存在をきちんと認めることはとても大切です。コードにバグがあることを認めることに抵抗を感じるのは無理もないことですが、頭の中でバグの可能性を除外してしまうならば、問題のある動作を引き起こしている原因を突き止めることはそれだけ困難になるかもしれません。

一方、具体的に何が望ましくない動作なのかを特定し、それを再現する方法が分かれば、ステップ 2 に進むことができます。この場合、単に「データベースがクラッシュする」というだけでは不充分です。プログラムがクラッシュすることはもちろん大問題ですが、問題を特定するためにも、いつ、どのようにプラグラムがクラッシュするのかをきちんと突き止めなければ、次のステップ 3、ましてやステップ 4(なぜクラッシュするのかを知ること)に進むことはできません。たとえば、ソフトウェアの使用者はプログラムが「不意に」クラッシュすると報告するかもしれません。実際には、プログラムが理由もなくクラッシュするということはなく、コンピュータを流れる電気信号に乱れがある場合を除けば(それさえ理由もなく起こるわけではないのですが)、プログラムがクラッシュする理由はすべて論理的に説明することができるはずです。コンピュータはロジックに基づいて設計されており、「ランダムにクラッシュする」というのは、実際には原因が特定できていない状況を意味しています。ソフトウェアには、無数の可変要素と処理階層が存在するため、バグを確実に再現できるパターンを突き止めることは非常に骨の折れる作業となります。バグの再現パターンを突き止めることは、デバッグの過程で何よりも大変かつ時間のかかる部分であり、実際のバグを分析することよりも難しいかもしれません。

再現可能なパターンが見つかれば、ステップ 3 に進むことができます。データベースの構造、およびコンパイルされているか否かにより、この過程の所要時間はさまざまです。いずれにしても、この段階では問題を引き起こすイベントを見つけることに努めます。それは特定のメソッド、ボタンクリック、あるいは壊れたレコードに対するアクセスかもしれません。問題を引き起こすイベントが特定できれば、それは調査の範囲を絞る上で役立ちますが、実際には各所に存在する複数の要素がバグの原因となっている場合が少なくありません。

バグの出所が特定できれば、いよいよバグの原因を突き止める段階(ステップ 4)に進むことができます。壊れたレコードに対するアクセスが原因だったのでしょうか。それとも想定外の入力値または引数が問題だったのでしょうか。最近のソフトウェアは、多数のモジュールで構成されている場合がほとんどであり、デベロッパが直接アクセスできない部分も存在します。バグの出所が一行のコードにまで絞り込むことができ、そのコードが第三者の手によるライブラリをコールしているのであれば、そのバグを完全に解明することはできないかもしれません。

バグに対するフィックスを開発する段階(ステップ 5)にはひとつ大切な基本原則があります。ひとつのフィックスで効果があがらなかった場合、そのフィックスを取り除いてから次のフィックスを試すという基本原則です。そうでないと別のバグが生まれるかもしれません。もちろん、段階的に適用しなければならないフィックスもあり、これは厳密なルールではありませんが、通常はこの基本原則に則ってデバッグを進めたほうが効率的です。

ステップ 6(フィックスの適用)は、それが 4D コードの問題である場合、インタプリタ版のコードを入手して修正することを意味します。壊れたレコードまたは入力値が問題である場合、データを修正することで当面の問題は回避できますが、同じことが再び起きないようにするためにも、やはり 4D コードを修正しておくべきでしょう。

ステップ 7 では、ステップ 2 で特定できたパターンを繰り返し、以前のような問題が発生しなくなっていれば OK です。

デバッグの手順は、必ずしもここで解説したステップの順番どおりにはいかないかもしれませんが、通常はこの順番を守ったほうが効率的にデバッグができます。バグの原因が大体予想できたとしても、ステップ 2(パターンの特定)からステップ 5(フィックス)にジャンプするようなことはなるべく避けてください。仮にその予想が外れていれば、そのフィックスを適用する前の状態に戻ってパターンの解析からやり直さなければならないからです。

Where does 4D fit in?

このテクニカルノートでは、前述のリストで挙げたステップの 2 から 4 に焦点をあてています。

2. バグの発生を再現するパターンを特定する。
3. バグの出所を特定する。
4. バグの原因を特定する。

4D には、次のようなデバッグツールが用意されています：

- デバッグ/トレースウインドウ
- ランタイムエクスプローラ
- デバッグログファイル
- エラーメッセージ

デバッグ/トレースウインドウは、インタプリタ版データベース用の極めて強力なデバッグツールであり、問題の出所および原因を探る上で非常に有用です。残念ながらコンパイル版のデータベースでは利用できません。

ランタイムエクスプローラには、デバッグ/トレースウインドウに匹敵する機能性があり、表示される情報の一部は、コンパイルされたデータベースでも利用することができます。

デバッグログファイルは、4D が出力するログファイルで、データベースのコールチェーンを基本的にすべて記録したものです。デベロッパが独自で作成するログファイルに比べ、メソッドコールだけでなくローレベルのエンジンコールも記録される点、およびログファイルに対するアクセスを 4D がすべて管理する点が優れています。

エラーメッセージは、デバッグをする上で非常に役立つ情報を提供している場合が少なくありません。エラーメッセージには、エラーの原因だけでなく、問題を起こしている特定のコードが表示されていることもあるからです。しかしデバッガやランタイムエクスプローラと同様、一部のエラーメッセージはコンパイルされたデータベースでは表示されません。

What if 4D cannot help?

上に挙げたデバッグツールは、デバッグログファイルを除き、バグを再現するパターンを突き止める上では、あまりあてにすることができません。デバッガ、ランタイムエクスプローラ、エラーメッセージは、いずれも再現ができるバグの出所と原因を特定するためのツールです。

最初に挙げたリストを思い出してください。まずバグを再現するパターンを特定し、次にバグの出所と原因を特定することになっていました。4D に組み込まれているツールは、バグを再現するパターンを特定する上では出来ることが限られており、されにコンパイルされたデータベースでは最初から利用できないものも少なくありません。

テキスト形式のカスタムログファイルを出力すれば、バグを再現するパターンの特定、およびコンパイルされたデータベースのデバッグという両面において 4D のデバッグツールに足りない点を補うことができます。この点については、テクニカルノートの第二部で取り上げます。(テキスト形式のカスタムログファイル出力を実装したコンポーネントも紹介します。)

Advanced Debugging Part 1 – 4D Techniques

このセクションでは、4D に組み込まれているデバッグツールを利用した少し高度なデバッグテクニックを幾つか紹介しています。内容はそれぞれのツールごとに分かれています：

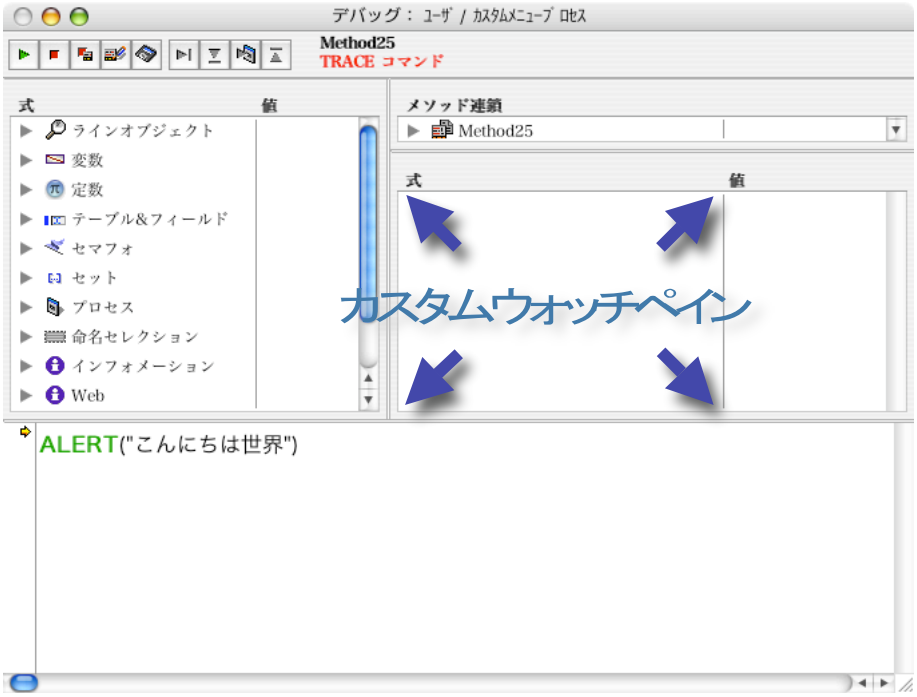
- デバッグ/トレースウインドウ
- ランタイムエクスプローラ
- デバッグログファイル
- エラーメッセージ

The Debug/Trace Window

デバッグ/トレースウインドウの正式な説明は 4D ドキュメントに含まれているので、ここで詳細に取り上げることはしません。対応するバージョンのランゲージリファレンス、デザインリファレンスを参照してください。

ここでは、しばしば過小評価されているふたつの機能であるカスタムウオッチエリア(カスタムウオッチペインともいいます)とプログラムカウンタの用法を簡単に紹介したと思います。

カスタムウオッチペインは、非常に強力なデバッグツールです。スクリーンショットに示されているデバッガの右下部分がカスタムウオッチペインです。



変数や式の値がここで確認できることはご存知だと思います。ポイントは、4D コマンド、プロジェクトメソッド、プラグインメソッドなど、あらゆる 4D 表記が入力できるという点です。

簡単な例を紹介しましょう：

倍長整数の引数を加算する単純なプロジェクトメソッドがあったとします：

` 値を加算...

```
C_LONGINT($1;$2;$0)
$0:=$1+$2
```

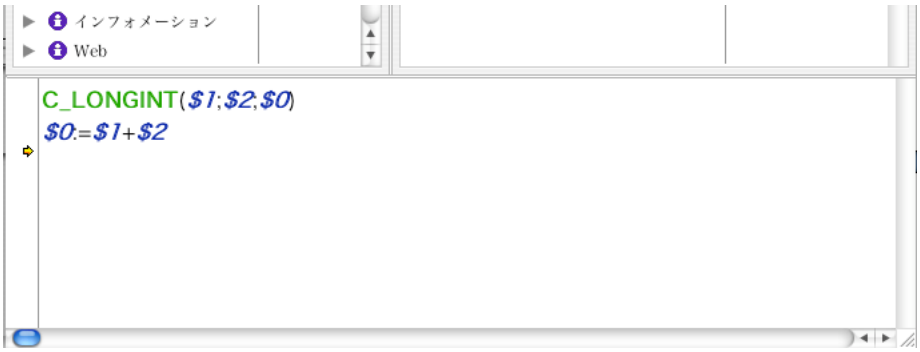
このメソッドをデバッグしている際､別の値を$2 に代入して再試行する必要が生じたとします。メソッドをアボートし、引数を変えて再びコールする代わりに、カスタムウオッチペインを利用して直接、新しい値で結果を確認することができます。そのためには、まず$2 を式のリストに追加しなくてはなりません。式を追加するにはいくつかの方法があります：

- 式のリストを右クリックし、コンテキストメニューから「新規式」を選択します。フォーミュラエディタが表示されるので、$2 と入力します。
- 式のリストをダブルクリックし、空の新規式を追加します。$2 と入力し、RETURN キーを押した時点で式は評価され、コマンドは実行されます。
- メソッド内に登場する$2 をハイライトし、Ctrl(Windows)/command(Macintosh)を押しながらクリックします。このようにすれば、たとえ無効な式であっても、任意のテキストを式のリストに追加することができます。

どの方法を使用しても結果は同じです。

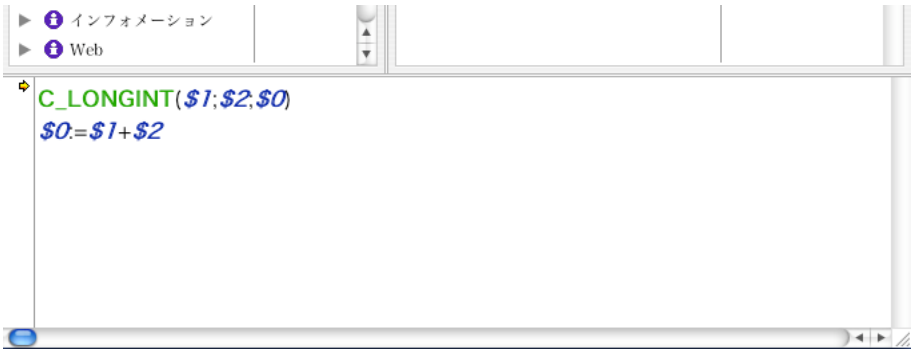
$2 の値を 5 に変更したければ、値リストの 20 をクリックして 5 と入力します。新しい値は ENTER/RETURN キーを押した時点で効になります。コードの以降の箇所では、すべて$2 の新しい値が使用されます。

$2 の値が変更された今、コードをアボートして再実行する必要はありません。ここでデバッガのもうひとつの目玉であるプログラムカウンタを使用してみましょう。プログラムカウンタとは、ソースエリア(ペインとも呼ばれます)の左側に表示される小さな黄色の矢印のことです。



この矢印は、デバッガが次に実行しようとしているコードの位置、つまり「Step Over F4: 同一メソッド内のみ」がクリックされた場合に実行されるコードを示しています。この矢印の位置は自由に移動することができます。つまりコードが実行される順番を勝手に変えることができるのです。何と強力なツールなのでしょう！

この例では、プログラムカウンタを一行上に戻すことにより、再び引数の加算を実行することができます：



「Step Over F4: 同一メソッド内のみ」をクリックすれば、もう一度加算が実行されます。

カスタムウオッチペインの利用価値を示す別の例を紹介しましょう：

$2 の値を変更して加算を再実行する代わりに、式をカスタムウオッチペインにそのまま追加してしまうという例です。

- 式のリストをダブルクリックして新規式を追加します。
- $1+20 と入力し、デバッガが式を評価するように Enter/return キーを押します。

この場合、実際に$2 の値を変更することなく、コードを別の値で実行したときの結果を知ることができました。繰り返しになってしまいますが、カスタムウオッチペインには、あらゆる 4D 表記を追加することができるのです。

カスタムウオッチペインにはひとつ気をつけなくてはならいない点があります。それは、デバッガウインドウがフォーカスを得たとき、あるいはプログラムカウンタをひとつ進めたときには毎回、式のリストが再評価されるということです。このことを正しく理解していないと非常に厄介なことになるかもしれません。

たとえば、式のリストに ALERT("Hello World!")と入力した場合、アラートを閉じるたびにデバッガがフォーカスを得るため、そこで式が評価される結果、再度アラートが表示されるという無限ループになります。(そのような状況になった場合、アラートを閉じるためにスペースバーを押したまま、デバッガに式が表示される近辺を何度もクリックしてみてください。タイミングが合えば式が編集状態になります。スペースバーでアラートをキャンセルしながら delete または Backspace キーで式を削除してゆけば、やがて無限ループを抜け出すことができます。)

カスタムウオッチペインとプログラムカウンタは、デバッガの主要かつ不可欠な部分です。これらの機能をうまく活用すれば、プラグラムの実行順序は自由に変更することができ、デバッグ中はいつでも、そしてどんな式であっても実行して結果を調べることができます。データベースの構造やサイズによっては再現が非常に面倒または難しい条件であっても、デバッガを使用すれば、すぐに状況を再現して問題となる現象を確認することができます。

デバッグ/トレースウインドウにはいくつか制限があります。コンパイルされたデータベースでは利用できない、というのもそのひとつです。実行を中断するというデバッガの性質上、処理のタイミングに由来する問題、データベース全体の構造に起因する問題については、あまり参考にならないかもしれません。

The Runtime Explorer

デバッガもそうですが、ランタイムエクスプローラも 4D のドキュメントで説明されています。ここではランタイムエクスプローラの包括的な説明は省略し、デバッガの一部ではないためか、しばしばデバッグの手段としては見過ごされてしまう機能である「処理中断コマンドリスト」について少し述べたいと思います。

ランタイムエクスプローラのキャッチタブは次のような構成になっています：

コマンドをキャッチすることの利点は、何といってもデベロッパがコードのどこをデバッグするべきかを知らなくてもよいという点でしょう。4D は、リストに追加されたコマンドを実行するとき、それがコード内のどの場所であってもデバッグ/トレースウインドウを開きます。こうして得た情報は、問題発生のパターンを突き止める上で有力な材料となります。デベロッパは、バグが特定コマンドの実行と結びついているように思える場合、コマンドをキャッチをすることにより、そのコマンドを使用しているすべての箇所を監視することができます。

処理中断コマンドのリストに追加されたコマンドは、毎回の実行でデバッガを起動します。データベースの至るところで使用されているようなコマンドをキャッチする場合、その数に圧倒されるかもしれません。そのことも考慮に入れるようにしてください。

デバッガ同様、キャッチもコンパイルされたデータベースでは利用することができません。キャッチを使用すれば、デバッガ単体よりは効率的に問題発生のパターンを突き止められます。しかし大きくて複雑なデータベースでは相応の作業量になることを覚悟しなくてはなりません。

最後に一言：ランタイムエクスプローラのウオッチタブには、デバッグ/トレースウインドウと同じように 4D 表記の式を自由に追加することができます。ランタイムエクスプローラは、デバッガのようにソースが表示されていない以上、ソースからドラッグ＆ドロップまたはコピーで式を追加することはできませんが、あとはデバッガと同じように式を追加することができます。

The Debug Log File

4D 2004.3 のリリースで追加された新機能に「デバッグログ機能」というものがあります。このオプションを有効にすると、4D がデバッグを目的としたログファイルを自動的に出力します。

このログファイルは、データベースのコール連鎖を基本的にすべて記録したものです。

デバッグログファイルの使用に関する詳細なテクニカルノートがすでに公開されています。テクニカルノート 06-01 “The 4DDebugLog.txt File”を参照してください。

デバッグログファイルの利用価値は、その記録内容次第であるといえます。たとえば、データベースがクラッシュするようなケースでは、このログファイルを大変重宝するかもしれません。(ログファイルに記録された最後の処理がクラッシュの原因と密接に関連しているからです。) しかし、このデバッグログファイルは一切カスタマイズできないため、デベロッパがほんとうに知りたいと思っている情報が必ずしも記録されているわけではありません。第二部でカスタムログファイルによるデバッグという話題が取り上げらているのは、そのような理由からです。

Error Messages

4D が返すエラーメッセージには、シンタックスエラー、データベースエンジンエラー、ネットワークエラーなど、いくつかのタイプが存在します。エラーメッセージは、第一にエラーが発生したことを通知するものですが、問題が起きたタイミングだけでなく、問題の原因に関する有力な情報も提供していることを忘れないでください。

デバッグの際、デベロッパがエラーメッセージの正確な意味を理解しているということは、問題の早期解決につながるかもしれない非常に重要な要素です。

エラーメッセージは、デベロッパを惑わして注意を間違った方向に向けることがあるのも事実です。この点は意識している必要があります。しかしそうはいっても、メッセージを無視するべきではなく、問題の原因を的確に知らせていないかどうかを調べるようにするべきです。

Conclusion

このテクニカルノートでは、一般的なデバッグの手順、および 4D にはじめから組み込まれている機能を活用したデバッグの方法をいくつか紹介しました。

第二部では、4D の機能だけでは足りない分野、あるいはより効率的な方法でデバッグできる分野について取り上げます。これにはテキスト形式のログファイルを出力し、解析するという手法が関係しています。テクニカルノートの第二部には、この手法を例証する 4D コンポーネントも収録されています。