UNICODE バグについて

2005 年 11 月 15 日

目　次

1　UNICODE バグとは

1.1 UNICODE バグとは(本文書の目的)

本文書は、2004 年 10 月 30 日の「まっちゃ 139 勉強会」(※)のはせがわようすけ氏の「Unicode とセキュリティ」について、より詳細に研究した内容を記述する。

また、本文書の目的は、不正行為の助長ではなく、このようなセキュリティ上の問題が存在することを衆知徹底させることが目的であり、その結果コンピュータソフトウェアの更なる発展を期待するものである。

※まっちゃ 139 勉強会 ：http://matcha139.hiemalis.org/~isamik/benkyokai.html#02

1.2 UNICODE バグの本質

「1.1 UNICODE バグとは」で説明した参考資料の通りであるが、少しだけ本質論を記述する。

UNICODE **バグの本質は、セキュリティ対策としてのサニタイジングが**

1. **サニタイジング処理**
2. **処理実行**

の二段階で行われる処理の流れの中で、1 のサニタイジング処理を空間の広い UNICODE **上で行った結果、2 の実際の処理作業時には** ANSI(※)**に変換され、1 のサニタイジングされていない(サニタイジング処理を迂回した)データによってセキュリティ侵害を起こす、という事である。**

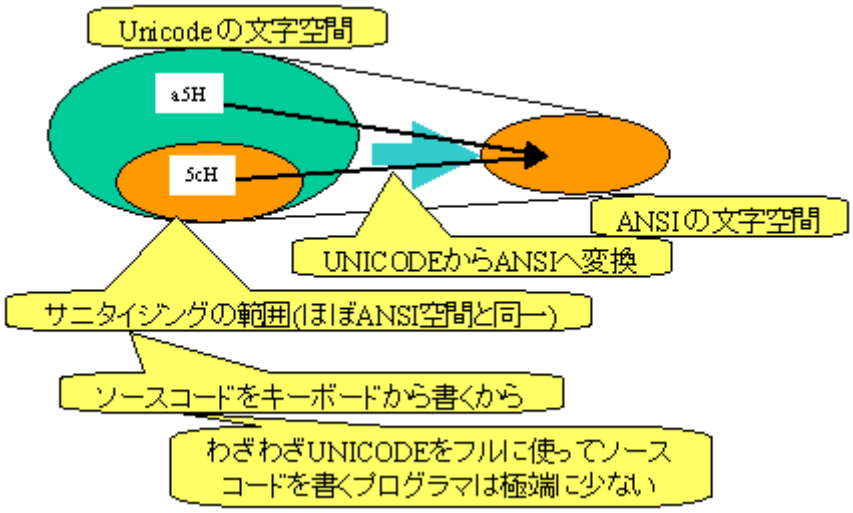

図1.2-1 : UNICODE **バグの概要** 1

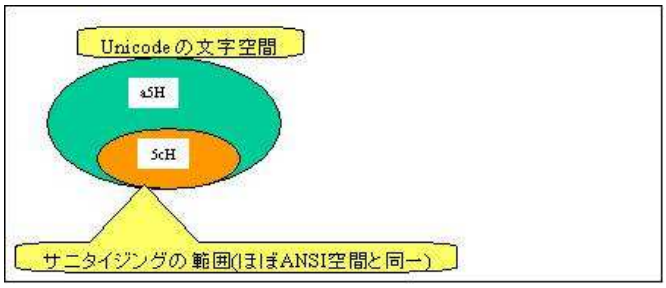

図1.2-2 : UNICODE **バグの概要** 2

(プログラムがサニタイジングするがサニタイジング範囲は限定的)

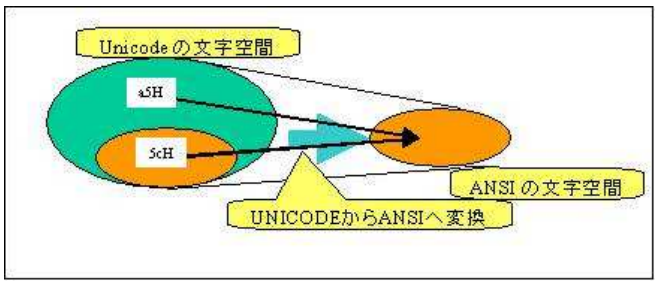

図1.2-3 : UNICODE **バグの概要** 3

(プログラムが呼び出した関数内部で ANSI **へ変換しているような場合、図** 1.2-2**のサニタイジングを迂回される)**

(※)サニタイジング処理 : 本文書では "サニタイジング処理" と "エスケープ処理" はほぼ同義である。

(※)ANSI：ここでは、UNCODE 以前の文字コードをさしている。つまり、日本国内の場合、ASCII+SJIS または ASCII+EUC-JP などの文字コードのことである。

1.3 UNICODE バグへの対策

■ **サニタイジング処理で対策する**

サニタイジング処理で対策するには、サニタイジング処理前にデータを ANSI **文字コードに対しての正規化を行ってしまえばよい。**

つまり、対象データを一度 ANSI **に変換してから、もう一度** UNICODE **へ変換しなおすという処理を行うことである。**

当然だが、ANSI **化されることで、国際化プログラムではなくなるが、元々**UNICODE **ではないモジュールを使っているため、国際化プログラムではなくなるということはデメリットではないだろう。**

- VisualBASIC6 **では、**strconv() **関数によって、**UNICODE **と** ANSI **との変換が可能である。**
  (srtconv(**データ**,vbFromUnicode)→strconv(**データ**,vbUnicode))
- VBScript **環境では** strConv() **関数はないが、**VB **の** ActiveX-DLL **を作ってもらい、それを使用するといいだろう。**
  (**一応、** http://www.cc.rim.or.jp/~sanaki/text/free/free50.htm **の** Moji_Chk.dll **の** reg_unicode()**を用意した**)
- C++**の場合、**Win32API **の** WideCharToMultiByte()/MultiByteToWideChar()**を使って、**ANSI **文字コードに対して正規化した** UNICODE **文字列に変換すればよい。**

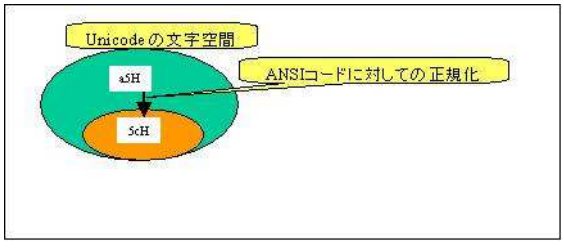

図1.3-1：UNICODE **バグの対策** 1

(**サニタイジング前に** ANSI **に対しての正規化をしてしまう**)

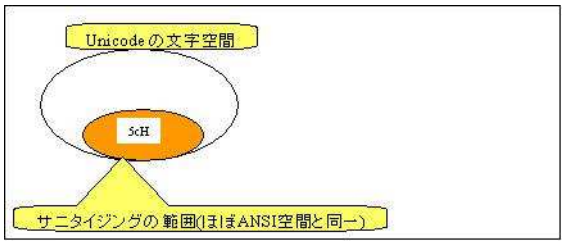

図1.3-2 : UNICODE **バグの対策** 2

(普通にサニタイジング処理を実施)

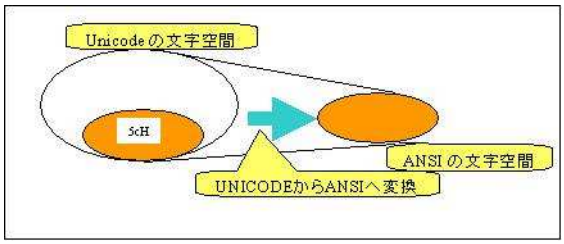

図1.3-3 : UNICODE **バグの対策** 3

(普通に内部的に ANSI **変換するメソッドを呼び出す)**

注意点は、上記の変換処理(UNICODE→ANSI→UNICODE)**がコンパイラの最適化によって省かれないように注意する必要がある**(※)。

■ **処理側**

処理側では、常に UNICODE **で処理を行うように記述する。**

(より正確には与えられた文字コードのままで処理を完結すること)

具体的には char **型ではなく、**wchar **型を使うようにする。**

- Windows **の場合、**tchar.h **と** tchar **型を使い、プリプロセッサの宣言を「**_MBCS**」から「**_UNICODE**」に書き直す。**
  (Windows9x **系での動作に支障がでる可能性に注意)**

図1.3-4 : VC6++SP6 **のプリプロセッサの設定画面**(Win32ConspleApplication & MFC)

デフォルトでは「_MBCS**」がついてビルドされる**

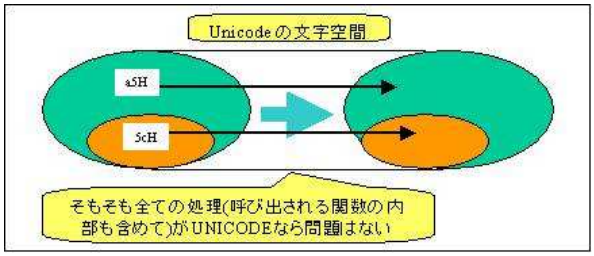

図1.3-5 : 呼び出されるメソッド内部も含めて、そもそも全ての処理が UNICODE **で行われていれば問題はない**

(※) **参考 : 良いニュースと悪いニュース**

http://www.microsoft.com/japan/msdn/columns/secure/secure10102002.asp

(ここではコンパイラの最適化によって、メモリクリアのコードが抜け落ちる可能性を指摘している)

2　UNICODE バグの影響

2.1 UNICODE バグの影響

全てのプログラムが UNCODE 化される近未来においては、このセキュテリィ問題は収束するだろうと予想できる。
しかし、一部のソフトウェアのみが UNICODE に対応している現状においては、UNICODE バグは広範囲で潜在的に存在しているものと推定される。

例えば、WindowsNT 系列の OS は内部的に UNICODE でありながら ANSI なプログラムを受け入れるような下位互換性を保持している。
また、Java 言語は UNICODE で記述されている。また Perl 言語、Python 言語、Ruby 言語なども内部的に UNICODE 化している。
これらの言語で開発したプログラムやスクリプトが、ANSI なモジュールを利用する時、UNICODE バグについて注意する必要がある。

2.2 WindowsNT 系の場合

WindowsNT では、内部的に UNICODE を採用している。採用している UNICODE のコード体系は、16bit 固定であり UCS-2 をそのまま使っている(一部に変則あり)。

WindowsNT の API には、WideCharToMultiByte()/MultiByteToWideChar()という関数があり、UNICODE(WideChar)と ANSI(MultiByte)の相互変換を行うことができる。

この関数を用いて、どのように重複するか確認した。(「5.1WindowsNT 系列での UNICODE 変換リストのプログラムについて」)

その結果は以下である。

0x21[!] (0x00a1)	0x2d[-] (0x00ad)
0x31[1] (0x00b9)	0x32[2] (0x00b2)
0x33[3] (0x00b3)	0x41[A] (0x00c0 ～ 0x00c6)
0x43[C] (0x00c7)	0x44[D] (0x00d0)
0x45[E] (0x00c8 ～ 0x00cb)	0x49[I] (0x00cc ～ 0x00cf)
0x4e[N] (0x00d1)	0x4f[O] (0x00d2 ～ 0x00d8)
0x52[R] (0x00ae)	0x54[T] (0x00de)
0x55[U] (0x00d9 ～ 0x00dc)	0x59[Y] (0x00dd)
0x5c[\] (0x00a5)	0x61[a] (0x00aa, 0x00e0 ～ 0x00e6)
0x64[d] (0x00f0)	0x65[e] (0x00e8 ～ 0x00eb)
0x69[i] (0x00ec ～ 0x00ef)	0x6e[n] (0x00f1)
0x6f[o] (0x00ba, 0x00f2 ～ 0x00f8)	0x73[s] (0x00df)
0x74[t] (0x00fe)	0x75[u] (0x00f9 ～ 0x00fc)
0x79[y] (0x00fd, 0x00ff)	0x7c[¦] (0x00a6)

0x3f[?]**は、変換できない場合に変換されるデフォルトの文字であるため、除外した。**

3　UNICODE バグの具体例

3.1 ANSI-C で書かれた Windows プログラム

WindowsNT 系の OS は内部的に UNICODE を採用している。
しかし、WindowsNT 系の OS で動作するプログラム(アプリケーション)までもが UNICODE 化されているとは限らない。

WindowsNT 系の OS で動作するプログラムを ANSI で記述した場合、以降で実験している各種の UNICODE バグでの問題が潜在的にあるということになる。

つまり、

- WindowsNT 系の OS で動作する ANSI なプログラムでは、ファイルパスを含む文字列データは ANSI コード(0x00 を終端とする char 型配列)に縮退しているため、このようなプログラムから NTFS 系ファイルシステムへアクセスした場合、UNICODE バグが発生する危険性がある。
- WindowsNT 系の OS で動作する ANSI なプログラムでは、外部コマンド呼び出し用のコマンド文字列を含む文字列データは ANSI コード(0x00 を終端とする char 型配列)に縮退しているため、このようなプログラムから(CMD.exe 経由で)外部コマンドを呼び出した場合、UNICODE バグが発生する危険性がある。
- WindowsNT 系の OS で動作する ANSI なプログラムでは、データベースへ問い合わせを行う SQL 文字列を含む文字列データは ANSI コード(0x00 を終端とする char 型配列)に縮退しているため、このようなプログラムから(PostgreSQL や mySQL などの「¥」でエスケープできる)データベースに対して SQL 実行を依頼した場合、UNICODE バグが発生する危険性がある。
- WindowsNT 系の OS で動作する ANSI なプログラムでは、LDAP 検索文字列を含む文字列データは ANSI コード(0x00 を終端とする char 型配列)に縮退しているため、このようなプログラムからディレクトリサーバに問い合わせを実行した場合、UNICODE バグが発生する危険性がある。
- その他、にもモジュールや関数への入力データのフォーマットなどによって、UNICODE バグが発生する危険性がある。

当然、ANSI な Windows プログラム上でサニタイジング処理を実施していれば、(文字列データは ANSI コードに縮退しているため)UNICODE バグは発生しない。

しかし、ANSI プログラムの呼び出し元が、UNICODE なプログラムであり、かつその UNICODE な呼び出し元でサニタイジング処理を実施している場合、UNICODE バグが発生する危険性がある。

一般的に、Windows 上での UNICODE プログラムと ANSI プログラムでは、ソースコード・レベルで互換性がない。
つまり、UNICODE 系では文字列データとして wchar 配列を用い、ANSI 系では char 配列を用いる。さらに、呼び出す各種関数について、異なっている。(一般的に UNICODE 系は「W」の付いている関数を呼び出し、ANSI 系では「A」のついている関数を呼び出す)。

また、Windows9x 系の OS での UNICODE 系の関数にバグが多かったという過去の事例もあり、多くのプログラマは、Windows9x 系の OS での動作を非動作にしてまで UNICODE プログラムを作成するようなこともないと思われる。

一方で、Windows プログラムを作成する際、ANSI な Windows9x 系と、UNICODE な WindowsNT 系でのソースコードの互換性を取るために、tchar.h を使うことがしばしばある。
このように tchar.h を使ってソースコードを作成した場合、Windows9x 系では ANSI プログラムとして動作し、WindowsNT 系では UNICODE プログラムとして動作させることができる。
しかし、UNICODE 系になるのか ANSI 系になるのかは、プログラム実行時に決定されることではなく、コンパイル時に決定される。
つまり、プログラムのバイナリ形式は異なるのである。
(よく、Win9x 系と WinNT 系でインストーラが異なるのはこのためである。また Setup.exe は同一でありながらそこから呼び出される msi ファイルが UNICODE 版と ANSI 版に分けられている場合もある)

図3.1-1 : tchar.h **を使ったプログラムの** Windows95 SP1 **での実行例**

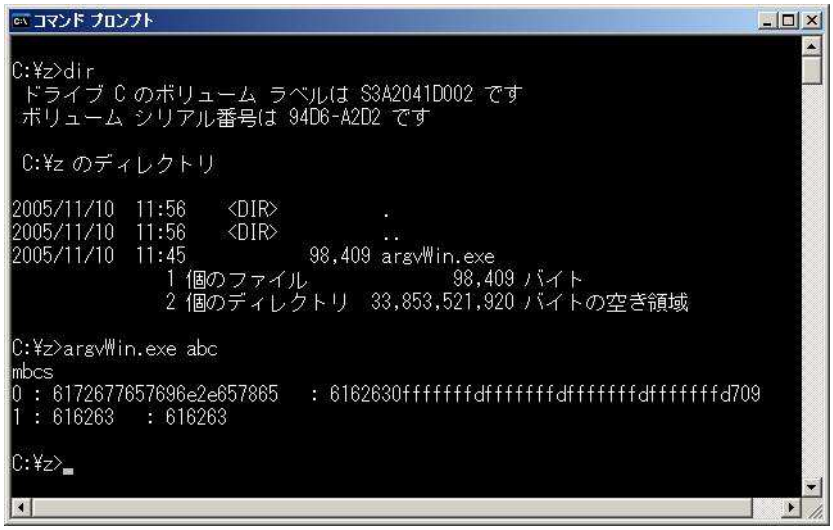

図3.1-2 : tchar.h **を使ったプログラムの** WindowsXP SP2 **での実行例(図** 3.1-1**と同一バイナリ)**

バイナリが同一だと、WinNT **系でも** ANSI **なプログラムとして動作する**

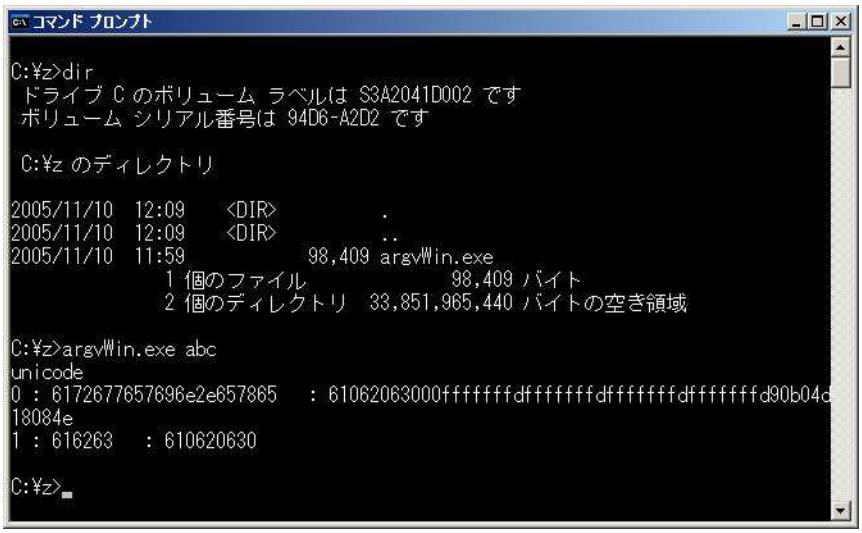

図3.1-3 : tchar.h **を使ったプログラムの** WindowsXP SP2 **での実行例**(_UNICODE **オプション付ビルド**)

また、この tchar.h **のデフォルトであるが、**MS-VisualC++6.0SP6 **では、「**_MBCS**」である。つまり、**ANSI **である。**
全てではないが、MS-VisualC++6.0SP6 **のウィザードによって生成されるプロジェクトでの「ビルド構成」の多くは、コンパイル時に「**_MBCS**」を指定している。**
この設定の変更は、プリプロセッサの定義を変更することである。(「メニュー」→「プロジェクト」→「設定」→「C/C++**」→「プリプロセッサの定義」)**

以上のことから推測すると、Windows9x **系でも** WindowsNT **系でも動作する同一バイナリの** Windows **プログラムは、「**_MBCS**」オプションでコンパイルされているのではないか思われる。つまり、このような** Windows **プログラムを** WindowsNT **系で動作させた場合、**UNICODE **バグを誘発する恐れがうる。**

また、COM **についても、内部的に** UNICODE **処理していない場合、この** COM **を呼び出した場合(呼び出し元は** UNICODE **なスクリプトなどが多いと思われる)、**UNICODE **バグが誘発する危険性がある。**

以下は、「5.3Variant **型の引数を** ANSI **で受け取るメソッドと** Unicode(Binary)**で受け取るメソッドのある** COM(MS-VC++6.0SP6)**」を「**5.4**「**5.3**」の** COM **を呼び出すテストスクリプト」から呼び出し結果である。**

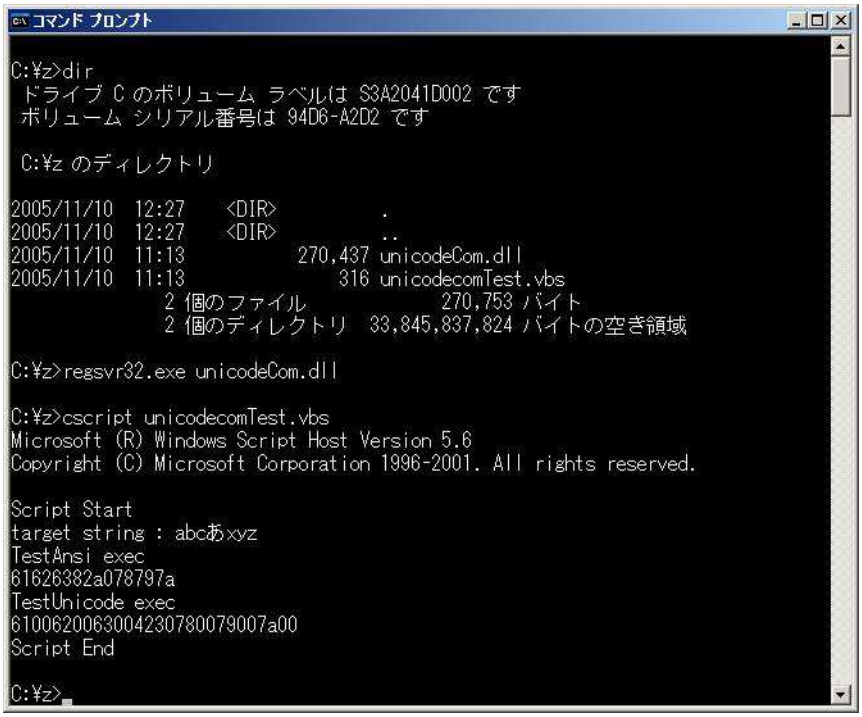

図3.1-4：「5.3」「5.4」で紹介したプログラムの実行結果

このように、例え COM のメソッドであったとしても、コーディングのしやすさ(wchar 配列のコーディング例よりも char 配列についてのコーディング指南書や教科書が多い)から、モジュールのメソッドの内部で、ANSI への置換処理を行っている可能性もある。
実際に筆者が公開しているソフトウェア(http://www.cc.rim.or.jp/~sanaki/text/free/sanaki-8.stm 以下)の MS-VC++6 .0 で記述された COM プログラムは、文字列処理として ANSI を採用している。

結論として、Windows 上でプログラムをする場合、極力 wchar 型でコーディングすることで UNICODE バグを抑えることが可能である。また、char 型でのコーディング時には使用者(モジュールを使用しているスクリプターなど)にその旨通知し、スクリプターは必要に応じて、ANSI への正規化処理などの UNICODE バグ対策を実装させるようにして使わせることでも UNICODE バグを抑えることが可能である。

3.2 Windows ファイルシステム

Windows ファイルシステムでは、「¥」がパスのデリミタとして機能している。
つまり、UNICODE上で0x5cの「¥」をチェックしていたとしても0xa5の「¥」をチェックしていないということになり、ANSIモジュールによって、0xa5のUNICODE文字が0x5cのANSI文字へと置き換えられてしまう。

3．2.1 Scripting.FileSystemObject オブジェクト(WSH5.6)

WSH/ASP/VB などでファイルアクセスする時、使われるオブジェクトである。
ソースコードは、「**エラー! 参照元が見つかりません。 エラー! 参照元が見つかりません。**」を参照。
今回は、(WSH や ASP などと比較して)バイナリ処理が簡単な VisualBASIC6.0SP6 を選択した。

このプログラムでは、「UNICODE Bug」チェックをつけると、「¥」を円記号に変換する。

「a(円記号)test.txt」というファイルを作る流れが図 3．2.1-1～図 3．2.1-3である。
図より、「円記号」がディレクトリデリミタの「¥」に置換されていないため、Scripting.FileSystemObject を使う限りにおいて、UNICODE バグのセキュリティ上の問題はない。

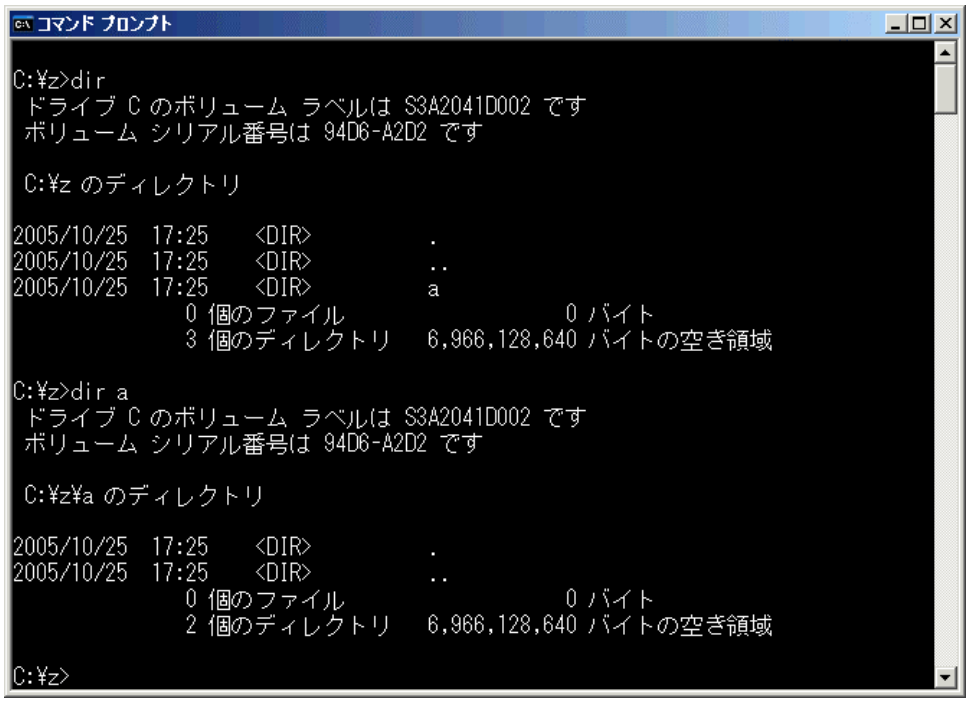

図3．2.1-1：テストプログラム実行前

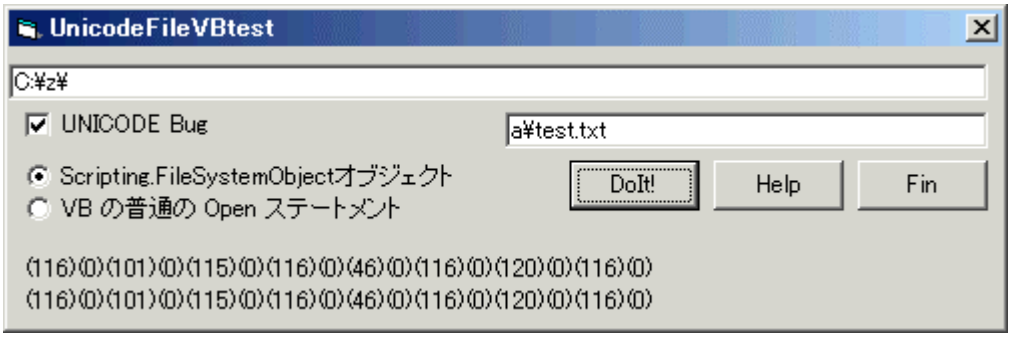

図3．2.1-2：ScriptingFileSystem **オブジェクトで「**a(**円記号**)test.txt**」というファイルを作る**

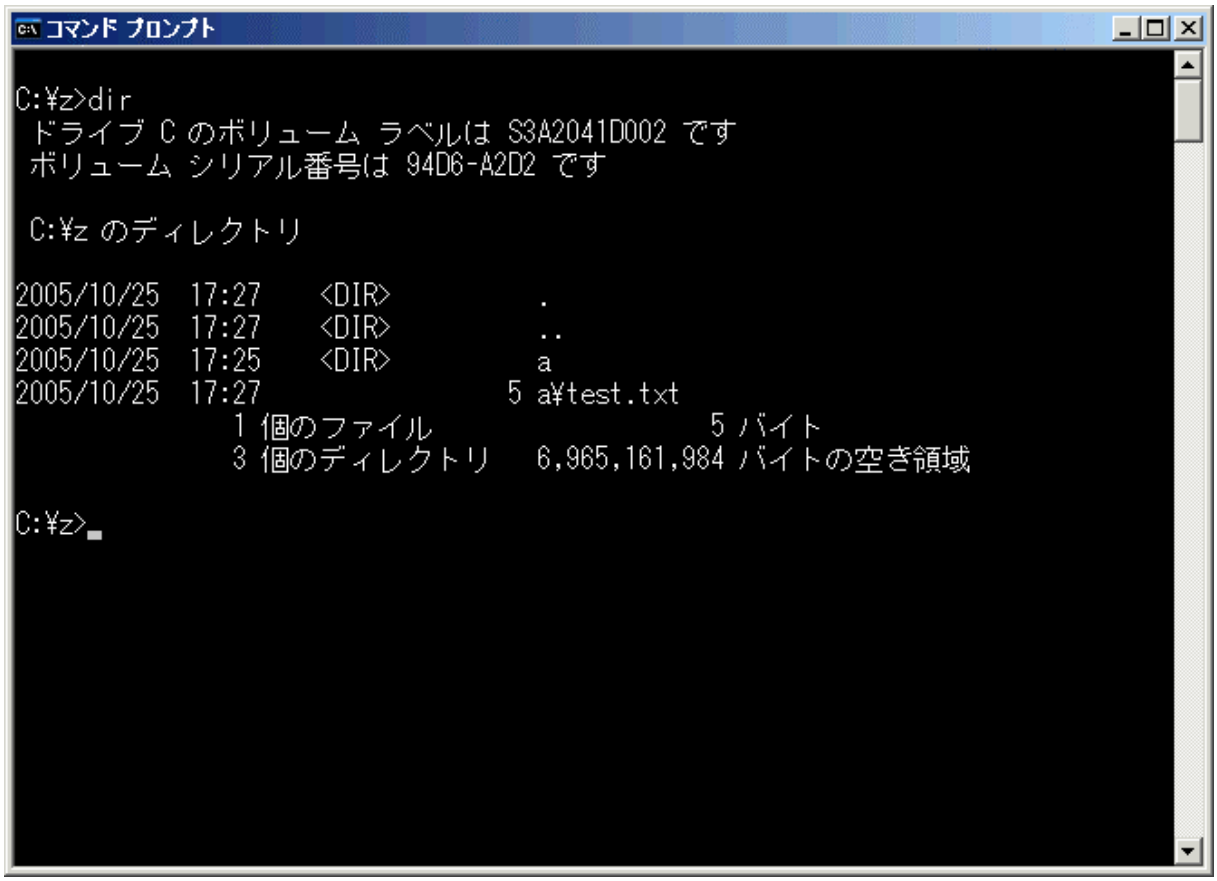

図3．2.1-3：図 3．2.1-2の結果(円記号がファイル名となり、問題はない)

3．2.2　open ステートメント(VisualBASIC6.0SP6)

VisualBASIC では、旧来からファイル・アクセス時に用いている open ステートメントがある。

この open ステートメントを使った場合に、UNICODE バグの脆弱性があるかどうか確かめてみた。図 3．2.2-1～図 3．2.2-2がそれである。
この図より「a(円記号)test1.txt」というファイルを作成しようとしたのにも関わらず、open ステートメントでは、「円記号」をディレクトリ・デリミタの「¥」に変換してしまっていることが分かる。
つまり、「¥」のサニタイジング処理を回避される可能性がある。
open ステートメントを呼び出す前にディレクトリ・トラバーサル問題対策のために「..¥」などサニタイジング処理を行っていたとしても、「..(円記号)」を与えることで、任意のファイルへの

アクセスが可能となる。

図3．2.2-1：今度は、open ステートメントにチェックする

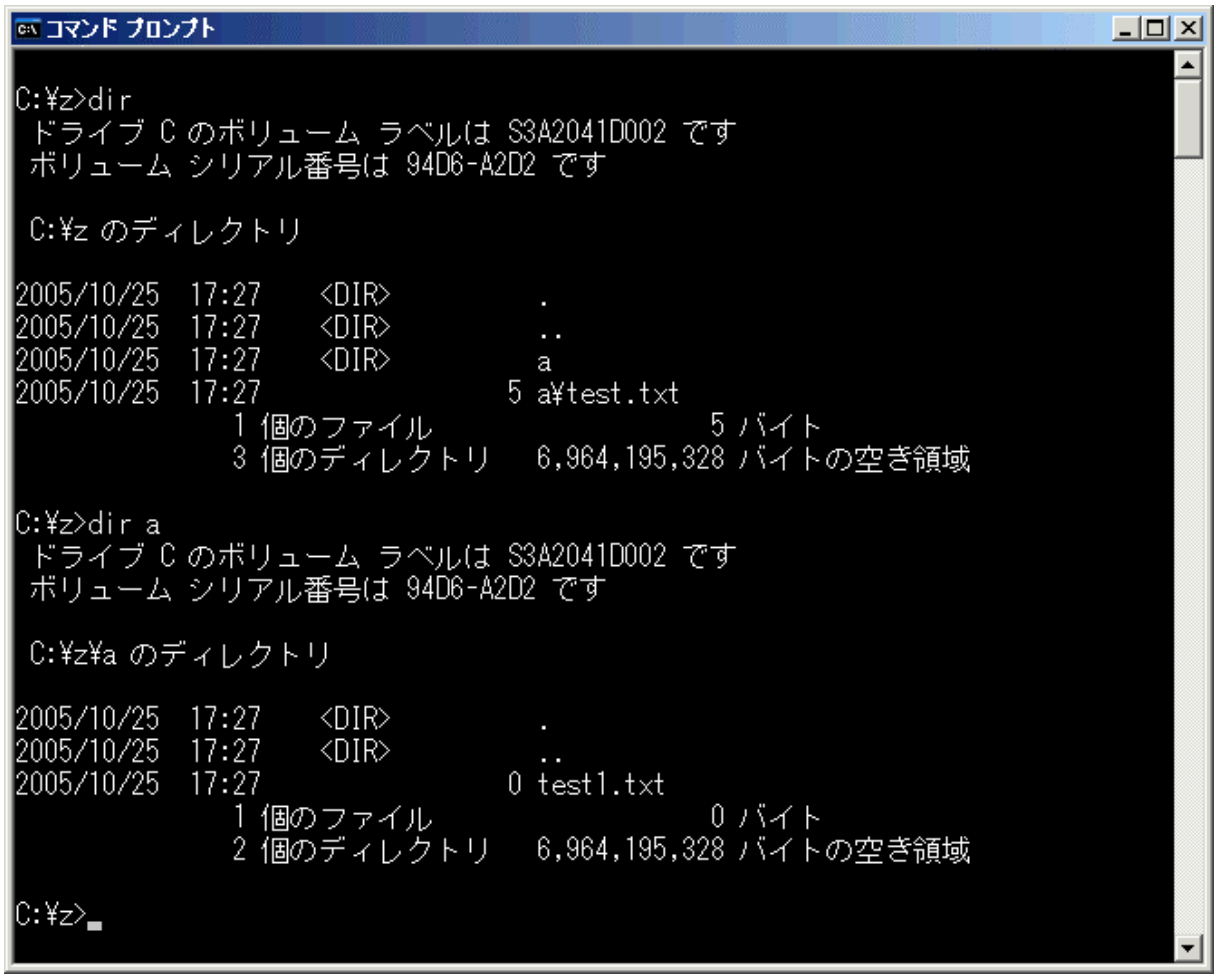

図3．2.2-2：図 3．2.2-1の結果

3．2.3　Windows 上での Java

Java 言語は内部処理は UNICODE で行っている。
Java 言語のファイルシステムへのアクセスでは File クラスを使うのが基本的な方法である。
実際にテストをしてみた結果、Windows 上での Java プログラムは、ファイル名を UNICDE で扱っており、UNICODE バグは発生しない。

図 3．2.3-1のように、c:¥java¥a(円)test.txt というファイルが、「c:¥java¥a¥test.txt」としてファイルが生成されていないことから、Windows 上での Java のファイルアクセスにおいては、UNICODE バグは発現しないものと思われる。

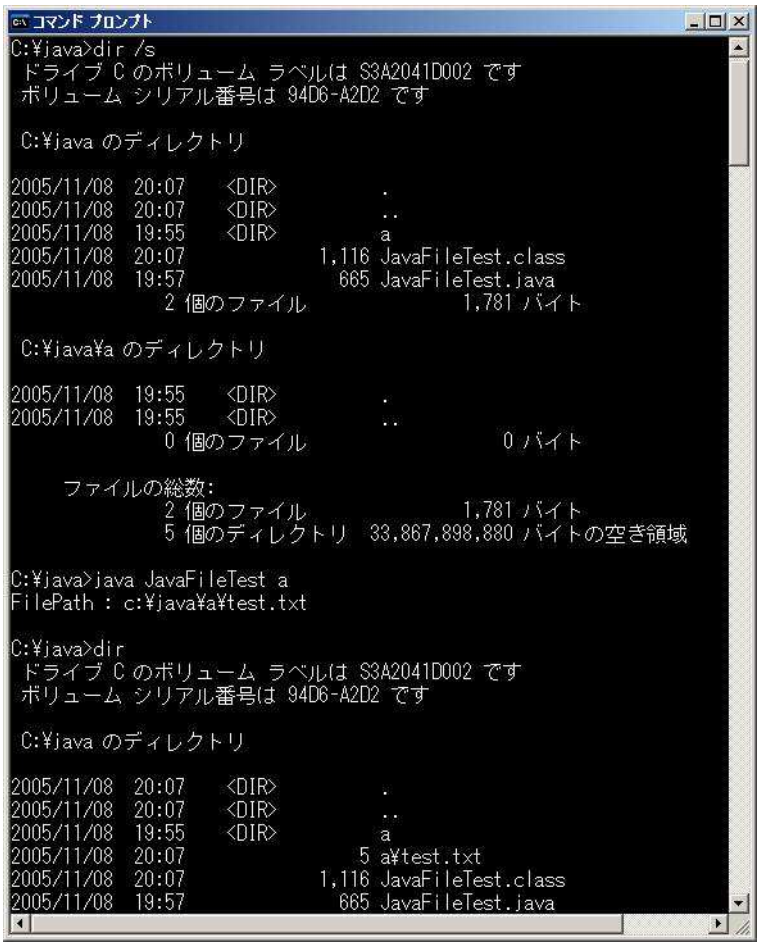

図3．2.3-1 : Java **プログラムと** UNICODE **ファイルパスと** Windows

3．2.4　Windows 上での Python2.4.2

現在、スクリプト言語全盛の時代であるが、Perl,Python,Ruby **は** Windows **上でも動作する。これらのファイルアクセスについても調査した。**
Python **の最新版は、**UNICODE **化されており、**Windows **上で動作する** Python **を用いてテストした。(実行環境は** Python 2.4.2 (#67, Sep 28 2005, 12:41:11) [MSC v.1310 32 bit (Intel)] on win32)
ソースコードは、「5.6 Python **で書かれたファイル読み出しプログラム」である。**

結果は、図 3．2.4-2**である。**
このことから、ファイルパス中の「円」記号が、ディレクトリデリミタの「¥」に置換されることはないということになる。つまり、Python **で書かれたスクリプトでのファイルパスによる** UNICODE **バグの影響はない。ということになる。**

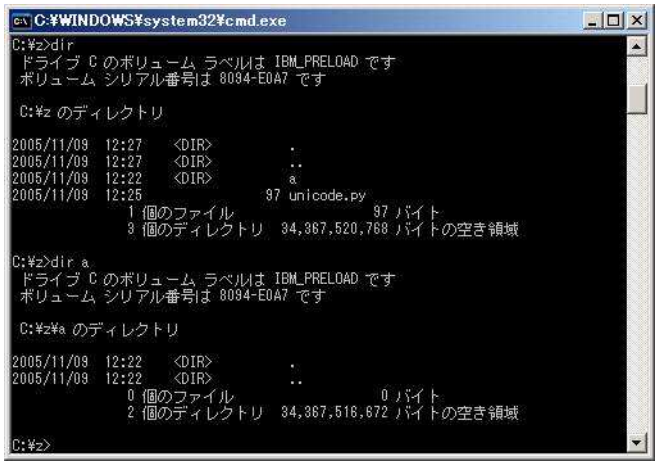

図3．2.4-1：テスト前のディレクトリ状態「c:¥z **以下」「**c:¥z¥a **以下」には** unicode.py **以外はない**

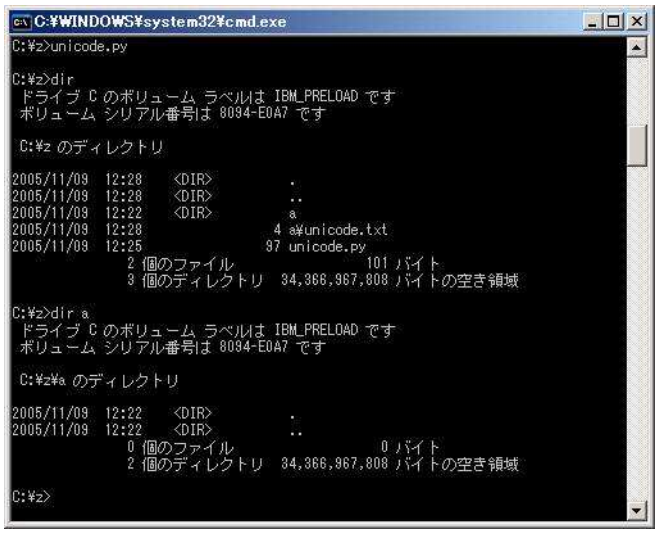

図3．2.4-2：テスト実行後。「C:¥Z」直下にファイルができていることから、UNICODE バグの問題はない

3.3 Windows の OS コマンド呼び出しについて

Windows では、「|」が UNICODE で同一化される文字になっている。
そして「|」はパイプである。しかし、Windows シェル(CMD.EXE)は基本的に UNICODE で処理されるため、問題ないはずである。また、Windows 上のモジュールはシェルを経由することなく外部コマンドを実行する場合が多いため、そもそも「OS コマンドインジェクション」の危険性は少ない。しかし、ANSI な system() 関数を用いている場合、「|」のサニタイジングに注意する必要がある。

3．3.1 WindowsScriptingHost 5.6 の WScript.Shell オブジェクトの Run メソッド

WSH で外部コマンドを実行する場合、WScript.Shell オブジェクトの Run メソッドを使うことが多い。
Run メソッド自体は、cmd.exe を呼び出していないため「OS コマンドインジェクション」とは無縁の存在ではある(図 3．3.1-1と図 3．3.1-2)。

しかし、Run メソッドから cmd.exe を呼び出して外部コマンドを使えば、「|」(パイプ)などのシェルの機能を利用することができる。

そこで、Run メソッドは内部的にどのような文字コードを使っているか調査したのが、図 3．3.1-3と図 3．3.1-4である。
図 3．3.1-3と図 3．3.1-4で結果が異なっていることから、0xA6 の記号は、シェルのパイプを意味するコードに変換されないことが分かる。
よって、WScript.Shell の Run メソッドは UNICODE バグに対して安全である。

ちなみに外部コマンドとして呼び出しているプログラムは、a.exe,b.exe 共に「5.9 外部コマンド呼び出しによって呼び出されるプログラム」を VisualC++6.0SP6 でコンパイルしたものをファイル名を「a.exe」および「b.exe」として保存したものである。

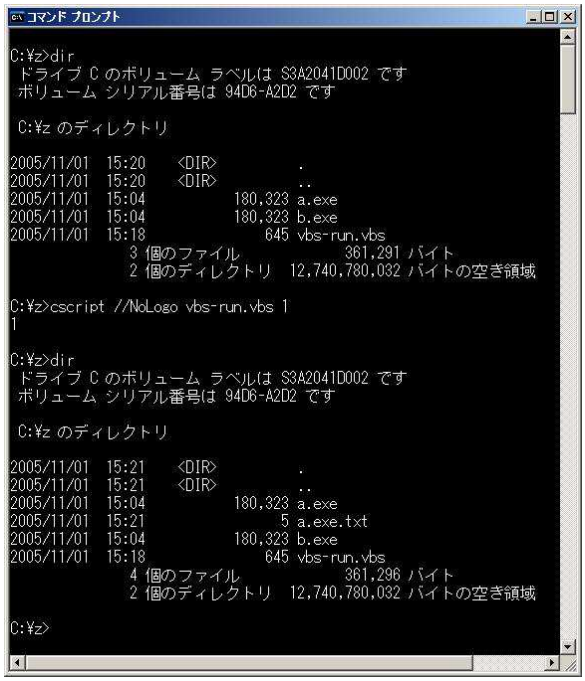

図3．3.1-1：「a.exe|b.exe**」の結果**

b.exe.txt **がないことから「|」以下はコマンドとして実行されていない**

(cmd.exe **を明示的に呼び出す必要がある**)

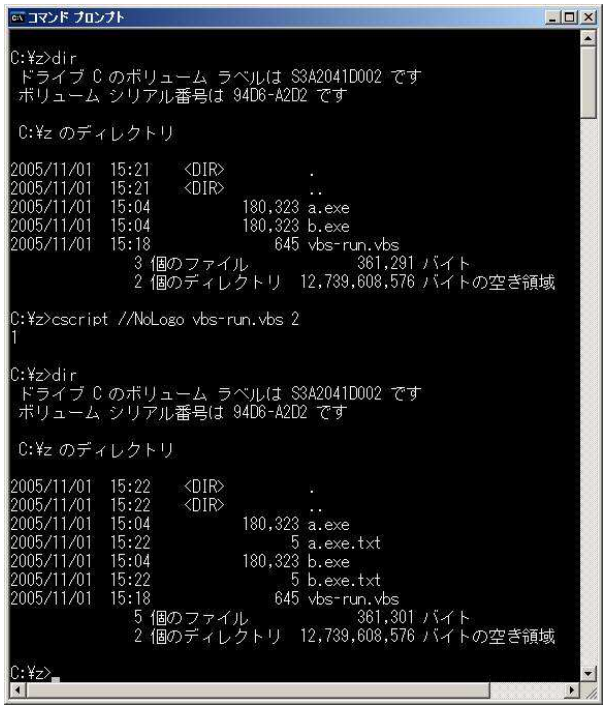

図3．3.1-2：「cmd.exe /c a.exe|b.exe**」の結果**

(cmd.exe **を明示的に呼び出すことで「パイプ」機能を利用できる**)

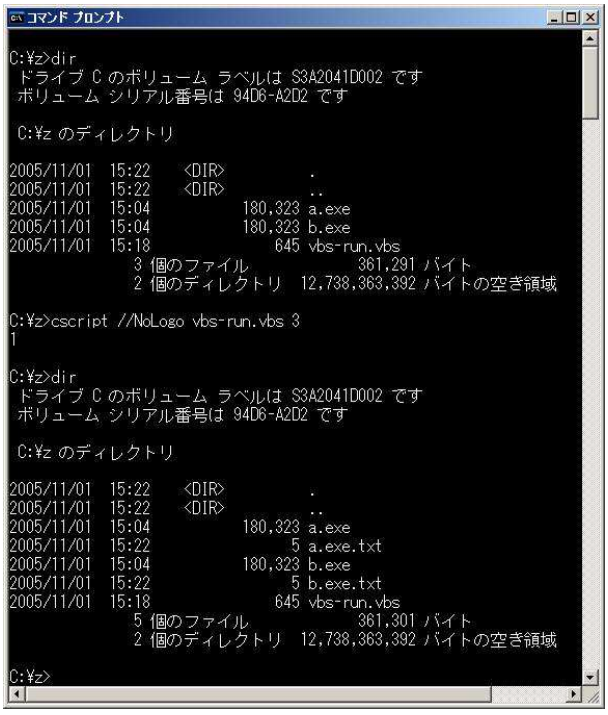

図3．3.1-3：「”cmd.exe /c a.exe” & chrW(124) & ”b.exe**」の結果**

図 3．3.1-2**と同じなので、同じ結果となっている**

(chrW()**関数の確認**)

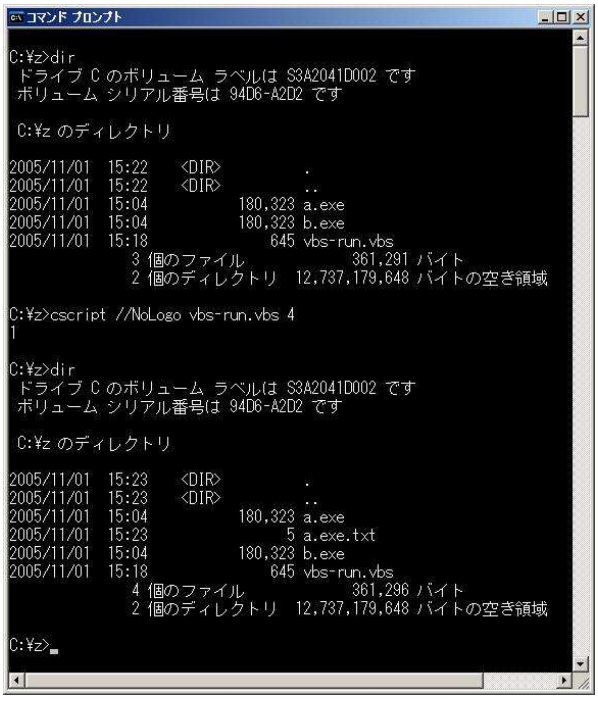

図3．3.1-4：「”cmd.exe /c a.exe” & chrW(166) & ”b.exe**」の結果**

図 3．3.1-3**とは異なり「|」以下がコマンドとして実行されていない**

3．3.2　WindowsScriptingHost5.6 の WScript.Shell オブジェクトの Exec メソッド

WSH で外部コマンドを実行する場合、WScript.Shell オブジェクトの Exec メソッドを使うことが多い。Run メソッドは標準出力が取得できないが、Exec メソッドで取得することができる。Exec メソッド自体は、cmd.exe を呼び出していないため「OS コマンドインジェクション」とは無縁の存在ではある(図 3．3.2-1と図 3．3.2-2)。

しかし、Exec メソッドから cmd.exe を呼び出して外部コマンドを使えば、「|」(パイプ)などのシェルの機能を利用することができる。

そこで、Exec メソッドは内部的にどのような文字コードを使っているか調査したのが、とである。とで結果が異なっていることから、0xA6 の記号は、シェルのパイプを意味するコードに変換されないことが分かる。
よって、WScript.Shell の Exec メソッドは UNICODE バグに対して安全である。

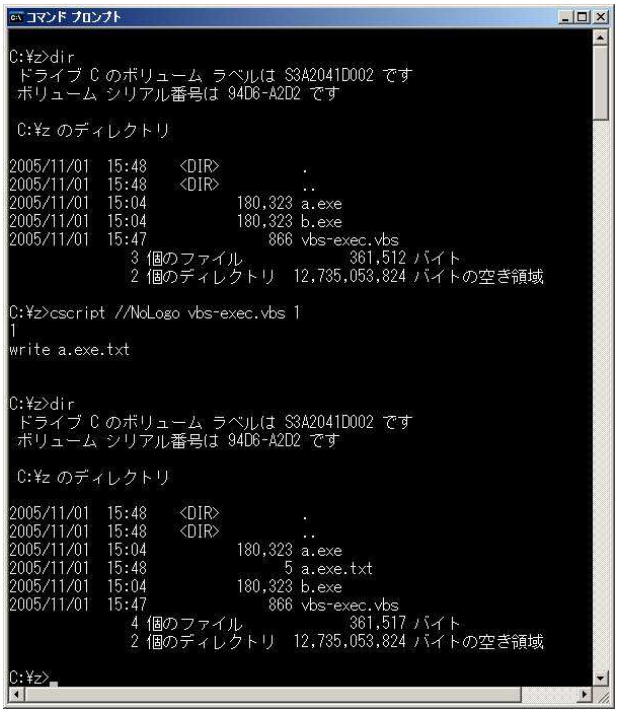

図3．3.2-1：「a.exe|b.exe**」の結果**

b.exe.txt **がないことから「|」以下はコマンドとして実行されていない**

(cmd.exe **を明示的に呼び出す必要がある**)

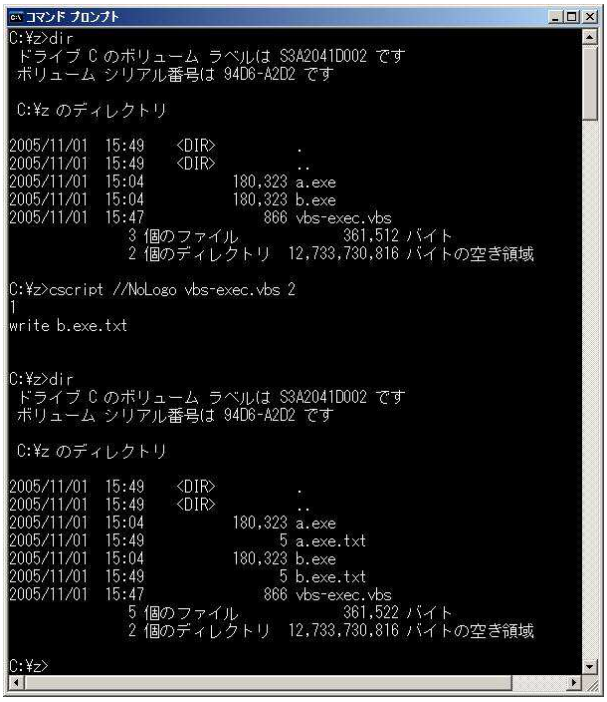

図3．3.2-2：「cmd.exe /c a.exe|b.exe**」の結果**

(cmd.exe **を明示的に呼び出すことで「パイプ」機能を利用できる**)

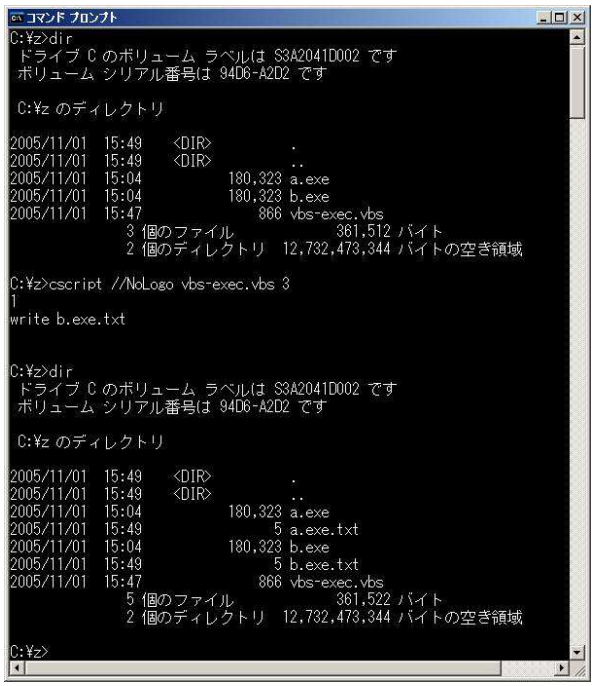

図3．3.2-3：「”cmd.exe /c a.exe” & chrW(124) & ”b.exe**」の結果**

図 3．3.2-2**と同じなので、同じ結果となっている**

(chrW()**関数の確認**)

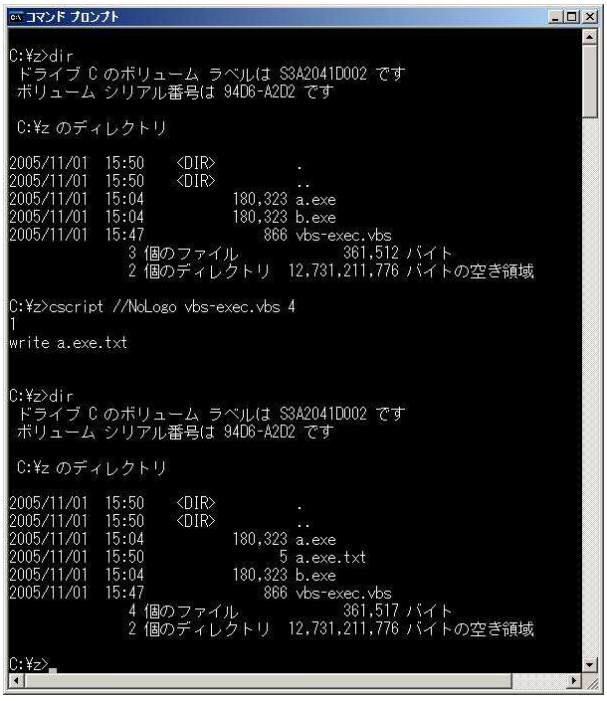

図3．3.2-4：「”cmd.exe /c a.exe” & chrW(166) & ”b.exe**」の結果**

図 3．3.2-3**とは異なり「|」以下がコマンドとして実行されていない**

3．3.3　VisualBASIC6.0SP6 の Shell()メソッド

VisualBASIC6.0 で外部コマンドを実行する場合、上記の WSH のオブジェクトを利用する場合もあるが、VisualBASIC6.0 には、外部コマンド呼び出しに Shell()メソッドが用意されている。Shell()メソッド自体は、cmd.exe を呼び出していないため「OS コマンドインジェクション」とは無縁の存在ではある(図 3．3.3-1～図 3．3.3-4)。

しかし、Shell()メソッドから cmd.exe を呼び出して外部コマンドを使えば、「|」(パイプ)などのシェルの機能を利用することができる。

そこで、Shell()メソッドは内部的にどのような文字コードを使っているか調査したのが、図 3．3.3-5～図 3．3.3-6である。
図 3．3.3-3～図 3．3.3-4と、図 3．3.3-5～図 3．3.3-6では結果が異なっていることから、0xA6 の記号は、シェルのパイプを意味するコードに変換されないことが分かる。
よって、VisualBASIC6.0SP6 の Shell()メソッドは UNICODE バグに対して安全である。

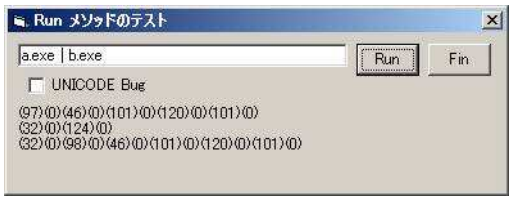

図3．3.3-1：「a.exe | b.exe」を shell に与えた

図3．3.3-2：図 3．3.3-1**の結果**

b.exe.txt **がないことから「|」以下はコマンドとして実行されていない**

(cmd.exe **を明示的に呼び出す必要がある**)

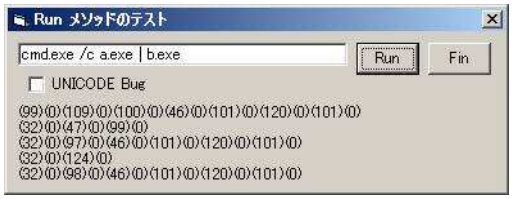

図3．3.3-3：「cmd.exe /c a.exe | b.exe**」を** shell **に与えた**

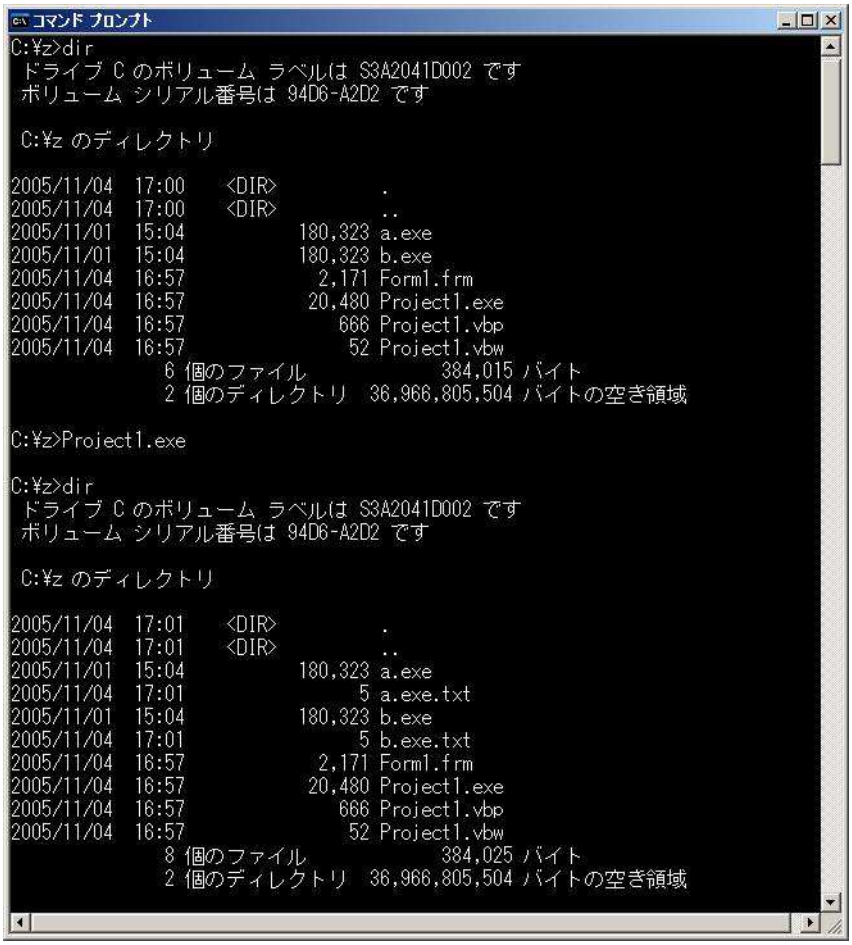

図3．3.3-4：図 3．3.3-3**の結果**

cmd.exe **を呼び出せば「|」を使うことができる。**

「a.exe.txt**」「**b.exe.txt**」共に作成されたので、「|」以下がコマンドとして実行された。**

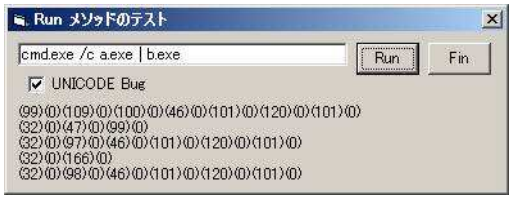

図3．3.3-5：「cmd.exe /c a.exe (0xA6) b.exe**」を** shell **に与えた**

図3．3.3-6：図 3．3.3-5**の結果**

「|」は「0xA6**」に変換した場合は「**a.exe.txt**」のみ作成されたので、**UNICODE **バグは発生しない。**

3.4 PostgreSQL/mySQL の SQL 利用時

PostgreSQL/mySQL では、「'」を「¥'」とエスケープすることができる。(「¥」を「¥¥」にエスケープする必要もある)
このような機能があるため「円記号」と「バックスラッシュ」による挙動を確認してみた。

3．4.1 Windows 上での PostgreSQL8.0.3(pgODBC7.01.0006)と VisualBASIC6.0SP6

基本的には、デフォルトインストール状態の PostgreSQL に対して、ODBC 経由で VisualBASIC からアクセスした。
VisualBASIC 上では、「'」と「¥」のサニタイジング処理を実施している。

図3．4.1-1：中央上のテキストボックス(「IE」という文字があるボックス)がサニタイジング対象

図3．4.1-2：エラー処理などは実施していないので、「aruyo」「naiyo」ダイアログよりも、エラーとなるかどうかがポイント

図3．4.1-3：「¥」を与える。「UNICODE bug」にチェックを入れると内部的に「バックスラッシュ」を「円記号」に変換する

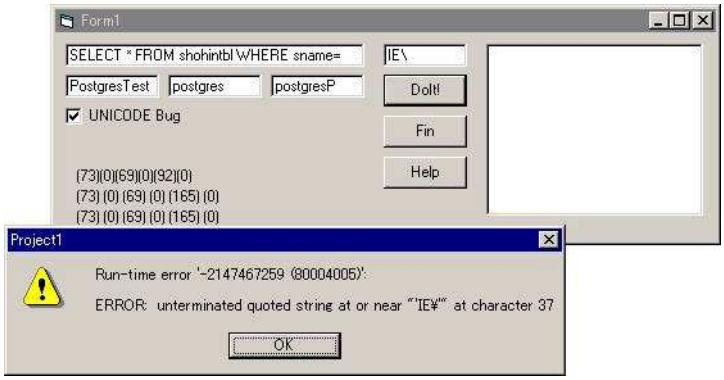

図3．4.1-4：図 3．4.1-3の結果(円記号が「¥」になったため SQL 文法エラーとなる)

図3．4.1-5：「IE¥¥」を与えた(円記号が「¥」になることを見越して「¥¥」としてみた)

図3．4.1-6：図 3．4.1-3の結果

以上より、Windows 版 PostgreSQL の SQL 解釈部は、ANSI 文字で行っているものと推測される。よって、Windows 版 PostgreSQL と VisualBASIC(VBScript も同様だと思われる)の組み合わせでは、UNICODE バグにより、SQL インジェクション問題が発現する可能性がある。

3．4.2　Windows 上での mySQL4.1.2alpha(myODBC3.51.07)と VisualBASIC6.0SP6

基本的には、デフォルトインストール状態の mySQL に対して、ODBC 経由で VisualBASIC からアクセスした。
VisualBASIC 上では、「'」と「¥」のサニタイジング処理を実施している。

図3．4.2-1：中央上のテキストボックス(「IE」という文字があるボックス)がサニタイジング対象

図3．4.2-2：エラー処理などは実施していないので、「aruyo」「naiyo」ダイアログよりも、エラーとなるかどうかがポイント

図3．4.2-3：「¥」を与える。「UNICODE bug」にチェックを入れると内部的に「バックスラッシュ」を「円記号」に変換する

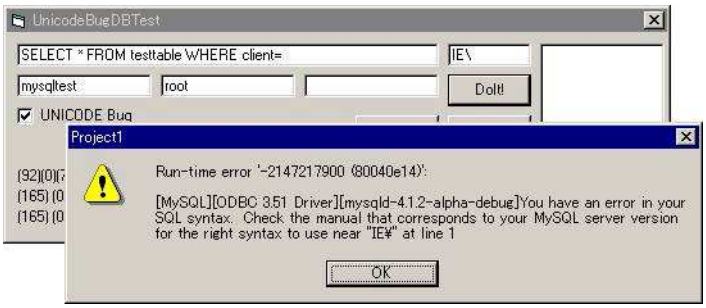

図3．4.2-4：図 3．4.2-3の結果(円記号が「¥」になったため SQL 文法エラーとなる)

図3．4.2-5：「IE¥¥」を与えた(円記号が「¥」になることを見越して「¥¥」としてみた)

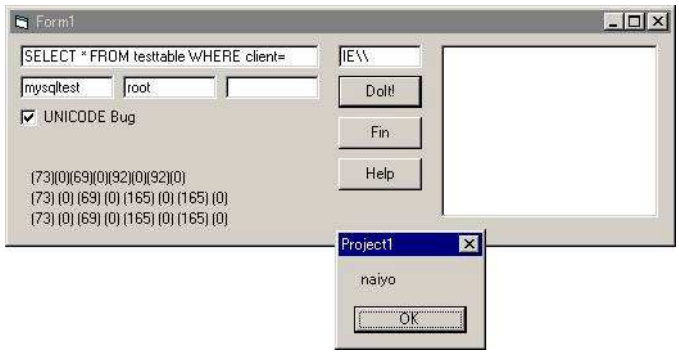

図3．4.2-6：図 3．4.2-5の結果

以上より、Windows 版 mySQL の SQL 解釈部は、ANSI 文字で行っているものと推測される。
よって、Windows 版 mySQL と VisualBASIC(VBScript も同様だと思われる)の組み合わせでは、

UNICODE **バグにより、**SQL **インジェクション問題が発現する可能性がある。**

4　執筆者と更新履歴など

4.1 執筆者

- active@window.goukaku.com
- yuya.yamaguchi@gmail.com
- forcing@gmail.com

4.2 更新履歴

最初のバージョン : 2005 年 11 月 15 日

4.3 本文書の最新バージョン

http:// racketeer.dip.jp/unicode/index.htm

5　付記

5.1 WindowsNT 系列での UNICODE 変換リストのプログラムについて

- UnicodeIsAll.bat

```
@FOR /L %%I IN (0,1,65535) DO @CALL UnicodeIsSub.bat %%I
```

- UnicodeIsSub.bat

```
@ECHO OFF
UnicodeIs.exe c %1
SET myANS=1
IF %ERRORLEVEL% == 0 SET myANS=0
IF %ERRORLEVEL% == 1 SET myANS=0
IF NOT %myANS%==0 ECHO %1
SET myANS=
```

- UnicodeIs.cpp

```
CWinApp theApp;
using namespace std;
int myMain(int hiKisu,char hiKisuC3,int inCode2,int inCode,int meate){
        unsigned char *p;
        int ansRet;
        unsigned char hako[6];
        unsigned char ansHako[6];
        int mojiSu;
        int ans;
        unsigned char c1;
        unsigned char c2;
        unsigned int ansInt;
        memset(hako,0x00,sizeof(hako));
        memset(ansHako,0x00,sizeof(ansHako));
        p = (unsigned char*)hako;
        ansRet = 0;
//      i = 0x2025;
```

```
		*(p+1) = (char)((int)(inCode/256));
		*p = (char)((int)(inCode%256));
		if(inCode2 != 0){
			*(p+3) = (char)(inCode2/256) + 220;
			*(p+2) = (char)(inCode2%256);
			*(p+1) = (char)((int)(*(p+1)) + 216);
		}
		mojiSu = WideCharToMultiByte(CP_ACP,NULL,(LPWSTR)p,-1,NULL,0,NULL,NULL);
		ans = WideCharToMultiByte(CP_ACP,NULL,(LPWSTR)p,-1,(char*)ansHako,sizeof(ansHako)-1,NULL,NULL);
		p = ansHako;
		c1 = *p;
		c2 = *(p+1);
		ansInt = 256 * (int)(c2) + (int)(c1);
		if(hiKisu == 2){
			if(2 < mojiSu){
				if(0 < (int)(c2) && (int)(c2) < 128 && (int)(c1) < 128){
					printf("%d(u%02x%02x)
- %d(0x%02x)[%c]¥n",inCode,(int)(inCode/256),(int)(inCode%256),ansInt,ansInt,*ansHako);
					ansRet = 1;
				}
			}
		}else{
			if(ansInt == meate){
				if(3 < hiKisu && hiKisuC3 == 'v'){
					if(inCode2 != 0){

		printf("%d,%d(u%02x%02x,%02x%02x)",inCode2,inCode,(int)(inCode2/256),(int)(inCode2%256),(int)(inCode/256),(i
nt)(inCode%256));
					}else{

		printf("%d(u%02x%02x)",inCode,(int)(inCode/256),(int)(inCode%256));
					}
					printf(" - %d(0x%02x)[%c]¥n",ansInt,ansInt,*ansHako);
				}
				ansRet = 1;
			}
```

```
	}
	return ansRet;
}
int _tmain(int argc, TCHAR* argv[], TCHAR* envp[]){
	int nRetCode = 0;
	// MFC の初期化および初期化失敗時のエラーの出力
	if (!AfxWinInit(::GetModuleHandle(NULL), NULL, ::GetCommandLine(), 0)){
		cerr << _T("Fatal Error: MFC initialization failed") << endl;
		nRetCode = -1;
	}else{
		unsigned int meate = 0;
		unsigned long i;
		unsigned long iMax;
		unsigned long iMin;
		unsigned long j;
		unsigned char c='a';
		int ansI = 0;
		iMax = 65535;
//		iMax = 25;
		iMin = 0;
//		iMin = 20;
			// 引数チェック
		if(1 < argc){
			if(2 < argc){
				meate = (unsigned long)atol(argv[2]);
				if(3 < argc && *argv[3] == 'v'){
					printf("meate : %d(0x%02x)¥n",meate,meate);
					c = *argv[3];
				}
			}
			i=iMin;
			for(i=iMin;i<=iMax;i++){
				ansI = ansI + myMain(argc,c,0,i,meate);
			}
			for(j=1;j<1024;j++){
				for(i=0;i<1024;i++){
```

```
                                        ansI = ansI + myMain(argc,c,j,i,meate);
                                }
                        }
                        nRetCode = ansI;
                        if(3 < argc && *argv[3] == 'v'){
                                printf("Count=%d¥n",ansI);
                        }
                }else{
                        printf("ASCII コードを指定して、重複する UNICODE 文字を探す旅 ver1.0¥n");
                        printf("usage: %s c ASCIIcode v¥n",argv[0]);
                        printf("       %s c ASCIIcode¥n",argv[0]);
                        printf("       %s a¥n",argv[0]);
                }
        }
        return nRetCode;
}
```

5.2 tchar.h を使って、引数の hex 表示をする VC++6.0SP6 プログラム

テンプレート : Win32ConsoleApplication → MFC を使った Hello プログラム

```
// argvWin.cpp : コンソール アプリケーション用のエントリ ポイントの定義
//

#include "stdafx.h"
#include "argvWin.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//////////////////////////////////////////////////////////////////////////
// 唯一のアプリケーション オブジェクト

CWinApp theApp;

using namespace std;

int _tmain(int argc, TCHAR* argv[], TCHAR* envp[])
{
        int nRetCode = 0;
        int i;
        int j;
        int jLen;
        TCHAR tc;
        TCHAR *tcp;
        char *pp;
```

```
int bai;


// MFC の初期化および初期化失敗時のエラーの出力
if (!AfxWinInit(::GetModuleHandle(NULL), NULL, ::GetCommandLine(), 0)){
        // TODO: 必要に応じてエラー コードを変更してください。
        cerr << _T("Fatal Error: MFC initialization failed") << endl;
        nRetCode = 1;
}
else{
        bai = 1;
        #ifdef _UNICODE
                bai = 2;
                printf("unicode¥n");
        #endif

        #ifdef _MBCS
                bai = 1;
                printf("mbcs¥n");
        #endif

        for(i=0;i<argc;i++){
                jLen = lstrlen(argv[i]);
                _tprintf(_T("%d : "),i);
                tcp = argv[i];
                for(j=0;j<jLen;j++){
                        tc = *tcp;
                        _tprintf(_T("%x"),tc);
                        tcp++;
                }
                printf("     : ");
                pp = (char*)argv[1];
                jLen = bai * jLen;
                for(j=0;j<jLen;j++){
                        printf("%x",(char)(*pp));
                        pp++;
```

```
                                }
                                _tprintf(_T("¥n"));
                        }
                }
                return nRetCode;
        }
```

5.3 Variant 型の引数を ANSI で受け取るメソッドと Unicode(Binary) で受け取るメソッドのある COM(MS-VC++6.0SP6)

テンプレート : ATL-COM Wizard → ATL クラス(シンプルオブジェクト)

```
// UnicodeTest.cpp : CUnicodeTest のインプリメンテーション
#include "stdafx.h"
#include "UnicodeCom.h"
#include "UnicodeTest.h"

#include <comdef.h>
/////////////////////////////////////////////////////////////////////////////
// CUnicodeTest


STDMETHODIMP CUnicodeTest::testANSI(VARIANT vin){
        _variant_t myVariant;
        _bstr_t myBSTR;
        unsigned long myLen;
        unsigned char *cp;
        char *pp;
        unsigned char *p;
        unsigned char c;

                // 自分のバリアント型へコピー
        myVariant = _variant_t(&vin);
                // BSTR 型へ変換
        myVariant.ChangeType(VT_BSTR,NULL);
                // 自分の BSTR 型へコピー
        myBSTR = _bstr_t(myVariant);
                // 自分の Variant 型を解放
        myVariant.Clear();
                // BSTR 型の文字長はいくつですか?
```

```
	myLen = (unsigned long)myBSTR.length();
		// UNICODE も考えて、2 倍のメモリを確保
	myLen = 2* myLen + 2;
	cp = (unsigned char*)malloc(myLen);
	memset(cp,0x00,myLen);
	myLen -= 2;
		// 確保したメモリへコピー
	pp = (char*)myBSTR;
	strncpy((char*)cp,pp,myLen);
		// NULL まで Hex で表示
	p = cp;
	while(*p != '¥0'){
		c = *p;
		if((int)c < 10){
			printf("0");
		}
		printf("%x",c);
		p++;
	}
	printf("¥n");
		// メモリ解放
	free(cp);
	return S_OK;
}

STDMETHODIMP CUnicodeTest::testUNICODE(VARIANT vin){
	_variant_t myVariant;
	unsigned long myLen;
	unsigned char *cp;
	unsigned char *p;
	unsigned char c;
	unsigned long i;

		// 自分のバリアント型へコピー
	myVariant = _variant_t(&vin);
		// unsigned int の SAFEARRAY 配列へ変換
```

```
	myVariant.ChangeType(VT_ARRAY | VT_UI1,NULL);
		// 配列長はいくつですか?
	myLen = (unsigned long)myVariant.parray->rgsabound->cElements;
		// メモリを確保してコピーする
	myLen++;
	cp = (unsigned char*)malloc(myLen);
	memset(cp,0x00,myLen);
	myLen -= 1;
	memcpy((char*)cp,(char*)myVariant.parray->pvData,myLen);
		// 自分の Variant 型を解放
	myVariant.Clear();
		// 長さ分 hex 表示
	// NULL まで Hex で表示
	p = cp;
	for(i=0;i<myLen;i++){
		c = *p;
		if((int)c < 10){
			printf("0");
		}
		printf("%x",c);
		p++;
	}
	printf("¥n");
		// メモリ解放
	free(cp);
	return S_OK;
}
```

5.4 「5.3」の COM を呼び出すテストスクリプト

```
Set obj = WScript.CreateObject("UnicodeCom.UnicodeTest")

str = "abc あ xyz"
WScript.Echo "Script Start"
WScript.Echo "target string : " & str

WScript.Echo "TestAnsi exec"
obj.testANSI(str)

WScript.Echo "TestUnicode exec"
obj.testUNICODE(str)

WScript.Echo "Script End"
Set obj = Nothing
WScript.QUit
```

5.5 Java で書かれたファイルアクセスプログラム

あらかじめ「c:¥java¥a」というディレクトリを作成しておく。

```
import java.lang.*;
import java.io.*;

class JavaFileTest{
        public static void main(String[] args){
                Character ch = new Character('¥u00a5');
                String chStr = ch.toString();
                String str;
                String str1 = "Hello";
                File myFileClass;
```

```
                FileOutputStream fOut;
                if(args.length == 0){
                        str = "c:¥¥java¥¥a" + "¥¥" + "test.txt";
                }else{
                        str = "c:¥¥java¥¥a" + chStr + "test.txt";
                }
                System.out.println("FilePath : " + str);
                myFileClass = new File(str);
                try{
                        fOut = new FileOutputStream(myFileClass);
                        try{
                                fOut.write(str1.getBytes());
                                fOut.close();
                        }catch(IOException e){
                        }
                }catch(FileNotFoundException e){
                }
        }
}
```

5.6 Python で書かれたファイル読み出しプログラム

あらかじめ「c:¥java¥a」というディレクトリを作成しておく。

```
import os

yen = u'C:¥¥z¥¥a¥u00a5unicode.txt'
f = open(yen, 'a')
f.write('test')
f.close()
```

5.7 WSH の Run メソッドでコマンド実行するスクリプト

```
Option Explicit
Dim WshObj
Dim ArgObj
Dim str

Set WshObj = WScript.CreateObject("WScript.Shell")
Set ArgObj = WScript.Arguments

REM 0x7c = 124 [|]
rem str = chrW(124)
rem str = chrW(166)
rem WScript.Echo str

WScript.Echo ArgObj.Count

If 0 < ArgObj.Count Then
 Select Case CStr(ArgObj(0))
  Case "1"
   WshObj.Run "a.exe | b.exe"
  Case "2"
   WshObj.Run "cmd.exe /c a.exe | b.exe"
  Case "3"
   WshObj.Run "cmd.exe /c a.exe " & chrW(124) & " b.exe"
  Case "4"
   WshObj.Run "cmd.exe /c a.exe " & chrW(166) & " b.exe"
 End Select
Else
 WScript.Echo "Read Sourcecode"
End If

Set WshObj = Nothing
WScript.Quit
```

5.8 WSH の Exec メソッドでコマンド実行するスクリプト

```
Option Explicit
Dim WshObj
Dim ArgObj
Dim ExecObj
Dim str

Set WshObj = WScript.CreateObject("WScript.Shell")
Set ArgObj = WScript.Arguments

REM 0x7c = 124 [|]
rem str = chrW(124)
rem str = chrW(166)
rem WScript.Echo str

WScript.Echo ArgObj.Count

If 0 < ArgObj.Count Then
 Select Case CStr(ArgObj(0))
  Case "1"
   str = "a.exe | b.exe"
  Case "2"
   str = "cmd.exe /c a.exe | b.exe"
  Case "3"
   str = "cmd.exe /c a.exe " & chrW(124) & " b.exe"
  Case "4"
   str = "cmd.exe /c a.exe " & chrW(166) & " b.exe"
 End Select
 Set ExecObj = WshObj.Exec(str)
 While ExecObj.Status = 0
  WScript.Sleep 100
 Wend
```

```
 str = ""
 While Not ExecObj.StdOut.AtEndOfStream
  str = str & ExecObj.StdOut.Read(1)
 Wend
 Set ExecObj = Nothing
 WScript.Echo str
Else
 WScript.Echo "Read Sourcecode"
End If

Set WshObj = Nothing
WScript.Quit
```

5.9 外部コマンド呼び出しによって呼び出されるプログラム

本プログラムが実行されると、実行プログラムと同一ディレクトリに、「実行プログラム」+「.txt」というファイルが作成される。

```
/* テストプログラム */
/* プログラム名 +  */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc,char *argv[]){
    unsigned char *filename;
    unsigned char *p;
    unsigned long filenameLen;
    FILE *fp;
    filenameLen = strlen(argv[0]);
    filename = (unsigned char*)malloc(filenameLen+5);
    memset(filename,0x00,filenameLen+5);
```

```
		/* get filename */
	strcpy(filename,argv[0]);
	p = filename + filenameLen;
	strcpy(p,".txt");

		/* write file */
	fp = fopen(filename,"w");
	fprintf(fp,"Hello");
	fclose(fp);

		/* ending */
	printf("write %s¥n",filename);

	return 0;
}
```

以　上